

Práctica de Diseño

Problema 1:

Nos han 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.

Respecto a los principios de diseño SOLID:

- Principio de responsabilidad única: este principio plantea que 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.

En el caso particular de nuestro programa, esto sucede ya que cada clase se encarga de una función específica de la siguiente manera:

- “API_Tickets.java” tiene una función llamada buscar, esta se encarga de delegar el trabajo de búsqueda a los diferentes tipos de búsqueda, recibiendo como parámetros el tipo de búsqueda a realizar (tipo “And” o “Or”) y las propiedades de búsqueda (origen, destino, precio y fecha).

- “Operación_AND.java” esta clase es llamada por la función API_Tickets.java y se encarga de redefinir la función “Operar” para adaptarla al operador lógico AND del modo de búsqueda y filtrado.

- “Operación_OR.java” esta clase es llamada por la función API_Tickets.java y se encarga de redefinir la función “Operar” para adaptarla al operador lógico OR del modo de búsqueda y filtrado.

- “Operaciones.java” tiene definida una la función Operar la cual es redefinida dependiendo del tipo de operación que reciba como parámetro dicha llamada, encargándose así de filtrar dependiendo de la clase de parámetro que recibe.

- Principio abierto - cerrado: Las entidades softwares (clases, módulos, etc.) deben ser abiertas para permitir su extensión, pero cerradas frente a la modificación.

En el caso de nuestro programa se puede ver que se cumple dicho principio porque si la empresa quiere añadir algún parámetro del billete, se crea una clase que implemente la interfaz “SearchingBar.java” y en la clase de la

propiedad nueva se implementa la función “filtrado”. De esta forma, el programa es flexible a futuros cambios en el Ticket.

- Principio de segregación de interfaces: este principio plantea que es preferible tener interfaces específicos para cada cliente en comparación a un único interfaz de propósito general.

En el caso particular de nuestro programa, podemos observar claramente dicho principio ya que contamos con múltiples interfaces, por un lado “SearchingBar.java” que se encarga del filtrado (sin importar que tipo de operación está haciendo) de las propiedades pasadas en la función Operar, que está situada en la interfaz “Operaciones.java” que se encarga de dirigir el filtrado al tipo de operación que se quiera hacer.

- Principio de inversión de la dependencia: La inversión de la dependencia es la estrategia de depender de interfaces o clases y funciones abstractas, en vez de depender de clases y funciones concretas.

En el caso particular de nuestro programa, el funcionamiento de este está basado completamente en las interfaces “SearchingBar.java” y “Operaciones.java” en vez de depender de clases y/o funciones concretas.

- Principio “Tell, don’t ask”: El código procedural pide información (ask) y luego toma decisiones. El código orientado a objetos le dice (tell) a los objetos que hagan cosas.

En el caso particular de nuestro programa, se cumple debido a que en las funciones operar, se llama a la función *filtrado* que está definida en “Destino”, “Fecha”, “Precio”, “Origen” que esta, dirige la propiedad que se le pasa al objeto en cuestión y así se evita preguntar si la propiedad [i - éxima] es instancia de dichas clases.

Patrones de diseño utilizados en este problema:

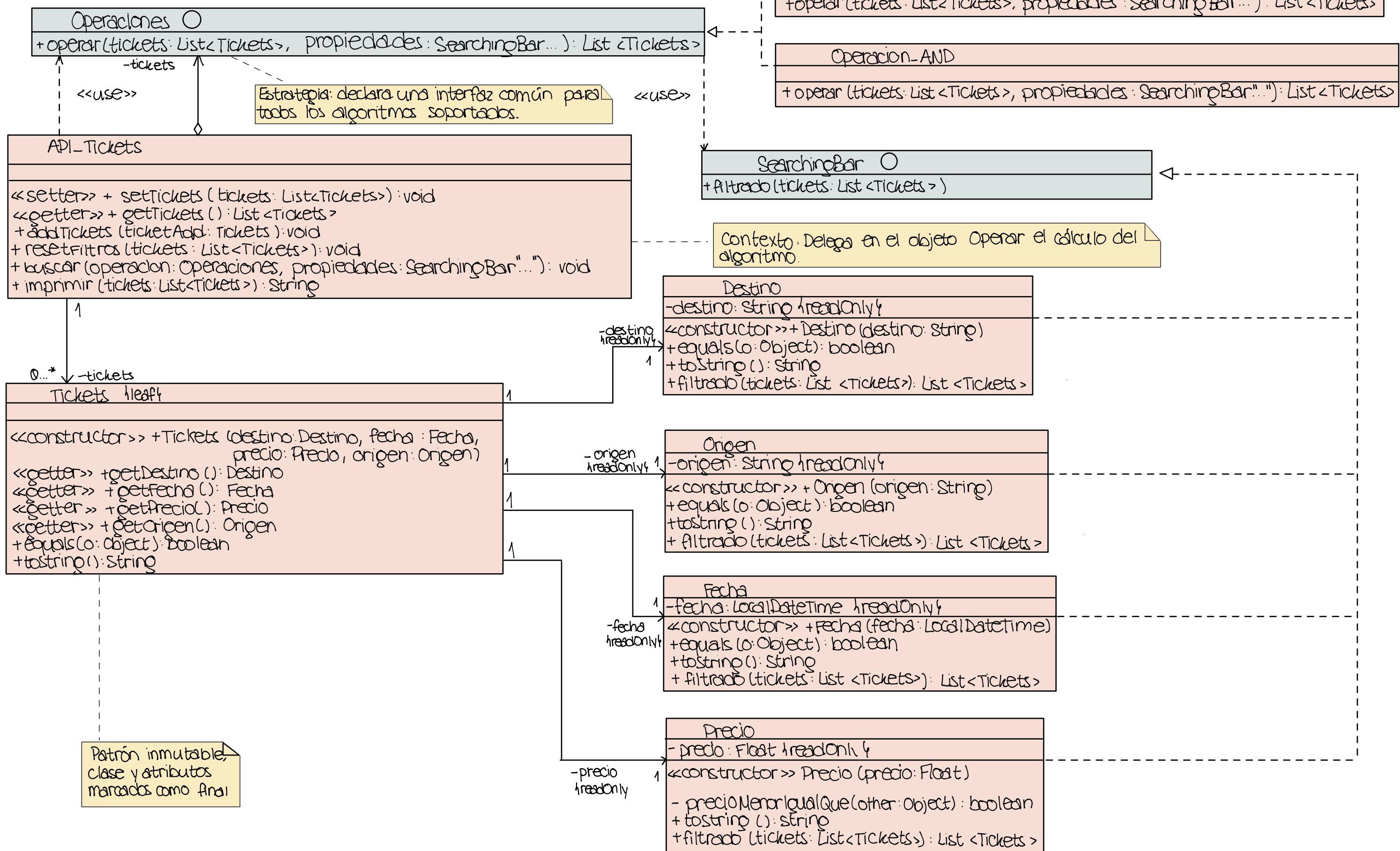
- Patrón Inmutable: Se encarga de diseñar clases en las cuales toda la información contenida en cada instancia es proporcionada en el momento de la creación y no puede modificarse durante el tiempo de vida del objeto.

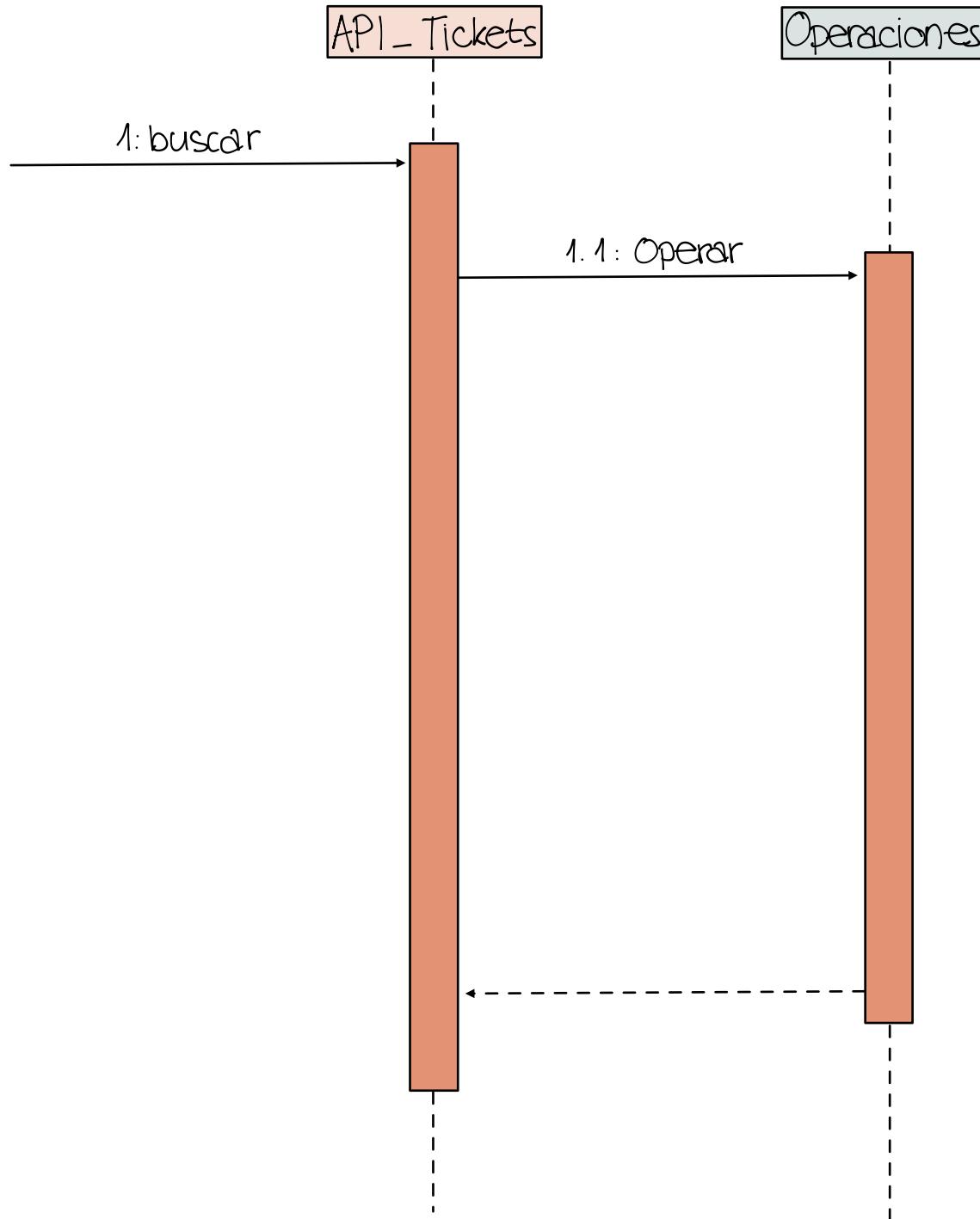
Elegimos este patrón ya que la clase principal “Tickets” es la que contiene todos los atributos de un Billete de bus propiamente dicho. Todos estos atributos son finales y privados ya que estos no pueden ser modificados una vez creado un objeto de esta clase (como es de esperar, un billete una vez emitido no se le puede cambiar sus datos como precio, valor, destino u origen). De esta manera, nos aseguramos de no incluir métodos que puedan modificar dichos objetos y así la clase se convierte en inmutable.

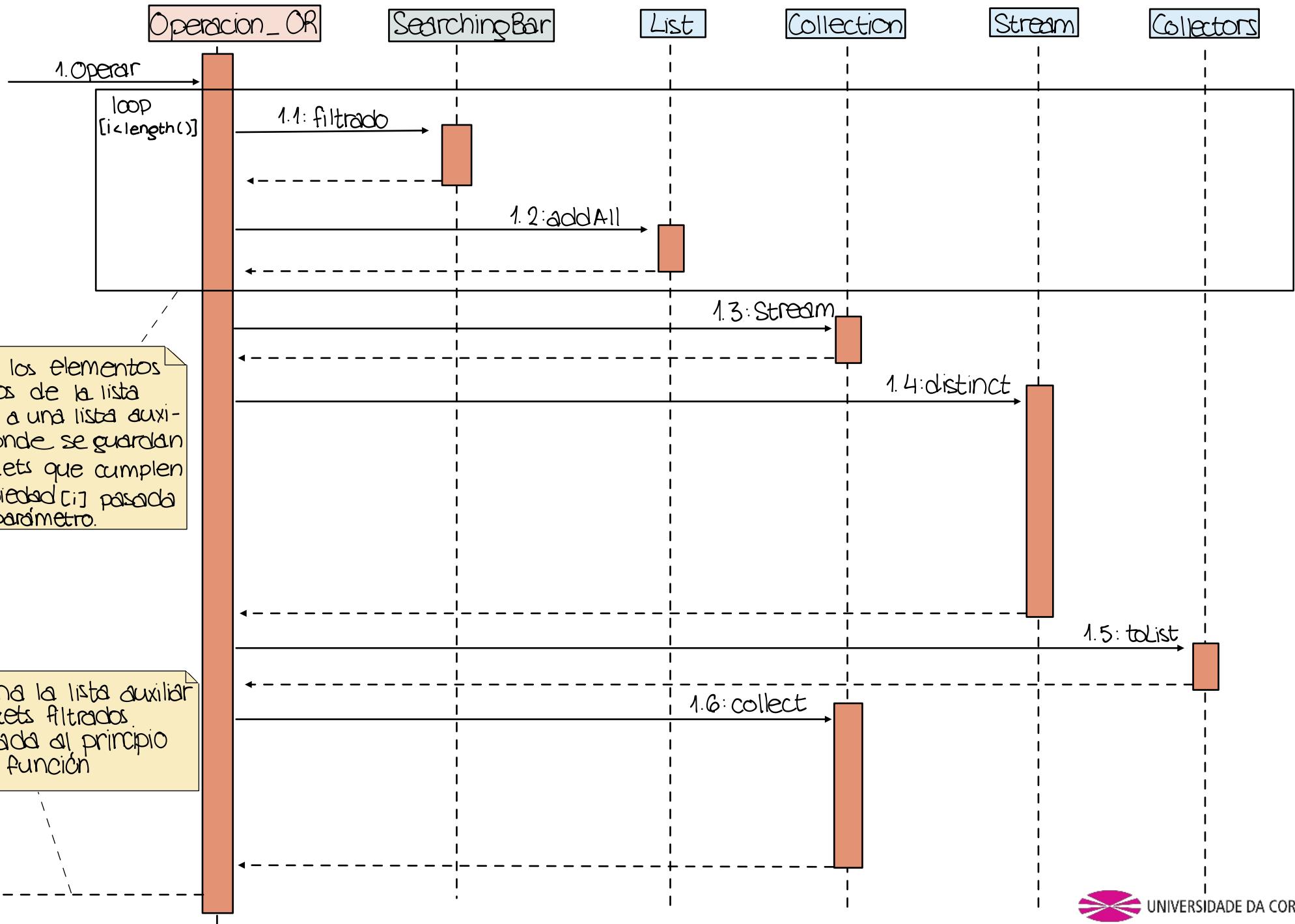
- Patrón Estrategia: Este patrón de comportamiento se utiliza para definir una familia de algoritmos, encapsularlos y hacerlos intercambiables.

Elegimos este patrón para separar la parte que varía (el método/tipo de búsqueda) del contexto (el ticket) y permitir cambiar dinámicamente de algoritmo de búsqueda.

e1 - Diagrama de clases







Operacion - AND

SearchingBar

List

Collections

Collection

stream

Collectors

1: operar

loop
[i < length()]

1.1: filtrado

1.2: addAll

1.3: size

loop
[j < size()])
opt
[length = frequency]

1.4: get

1.5: frequency

1.6: get

1.7: add

1.8: stream

1.9: distinct

1.10: toList

1.11: collect

Si el ticket está el mismo nº de veces que el nº de propiedades, es que las cumple todas.

Añade los elementos filtrados de la lista tickets a una lista auxiliar donde se guardan los tickets que cumplen la propiedad [i] pasada como parámetro.



Problema 2:

Nos han encargado un software que nos permita fácilmente almacenar el gráfico de tareas y obtener los distintos órdenes de ejecución de estas.

La empresa gestiona los proyectos dividiéndolos en sus tareas correspondientes. Las tareas tienen dependencias entre sí que se incluyen en un documento de texto con el siguiente formato:

Una entrada $X \rightarrow Y$ indica que hay que realizar la tarea X antes de pasar a realizar la tarea Y .

El orden de realización de las tareas variaría según entendamos cómo funcionan las dependencias entre las mismas y según la política de la empresa, en este caso trabajamos con tres tipos de realización: dependencia fuerte, débil y orden jerárquico.

Cada tarea tiene un nombre que la identifica, en caso de haber varias tareas disponibles para realizar al mismo tiempo se escogería por simplicidad, realizar las tareas por orden alfabético. También por simplicidad supondremos que las tareas se identifican por una única letra en mayúsculas (obviando las letras especiales como Ñ, Ç, etc.).

A la hora de introducir las tareas con sus dependencias lo hacemos a través de la lectura del fichero “*tareas.txt*” donde introducimos los datos con el formato descrito en el boletín de ejercicios.

Respecto a los principios de diseño SOLID:

- Principio de responsabilidad única: este principio plantea que 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.

En el caso particular de nuestro programa, esto sucede ya que cada clase se encarga de una función específica de la siguiente manera:

- “*API_Dependencias.java*” esta clase se encarga de configurar el tipo de dependencia a trabajar, y de leer el fichero *tareas.txt*.

- “*DependenciaDebil.java*” esta clase se encarga de obtener el orden de ejecución de las tareas teniendo en cuenta los criterios recibidos como parámetros para tratarla como una débil.

En este orden una tarea puede realizarse si no tiene antecesoras, o si bien una cualquiera de sus antecesoras ya se ha realizado. De esta forma si tenemos las dependencias $X \rightarrow Y$ y $Z \rightarrow Y$, X se realizaría primero, luego Y (al tener una de sus antecesoras realizadas) y finalmente se realizaría Z siguiendo el orden alfabético.

- “DependenciaFuerte.java” esta clase se encarga de obtener el orden de ejecución de las tareas teniendo en cuenta los criterios recibidos como parámetros para tratarla como una dependencia fuerte.

Este orden quiere decir que no se puede realizar una tarea Y mientras todas las tareas antecesoras de Y (las que tienen una flecha que lleva a Y) no se hayan realizado.

- “DependenciaJerarquico.java” esta clase se encarga de obtener el orden de ejecución de las tareas teniendo en cuenta los criterios recibidos como parámetros para tratarla como un orden jerárquico.

En este orden las tareas se van ejecutando según una jerarquía que viene determinada por lo lejos que quedan dichas tareas de las tareas iniciales, no pudiendo pasar de un nivel de la jerarquía hasta que se haya acabado el nivel anterior. Es posible que alguna tarea de un nivel superior tenga una dependencia con una tarea de un nivel inferior, pero como en el caso anterior se consideran dependencias débiles que no impiden que la tarea se lleve a cabo.

- “TareasMap.java” esta clase se encarga de la implementación del hashMap que se usa para guardar y gestionar las tareas.

- Principio abierto - cerrado: Las entidades softwares (clases, módulos, etc.) deben ser abiertas para permitir su extensión, pero cerradas frente a la modificación.

En el ejercicio dos, también se cumple porque el programa es capaz de añadir nuevas formas de ordenar tareas usando métodos de “TareasMap.java” y la interfaz “Dependencia.java” con las funciones de “ordenEjecución” y “addTareasAListas”. De esta forma el programa es flexible para implementar nuevos métodos de búsqueda.

- Principio de Sustitución de Liskov: este principio plantea que aquellos métodos que utilicen referencias a clases base deben ser capaces de usar objetos de clases derivadas sin saberlo.

En el caso particular de nuestro programa, esto sucede ya que las clases que gestionan las dependencias de las tareas (“DependenciaDebil.java”, “DependenciaFuerte.java” y “OrdenJerarquico.java”) extienden a la clase “TareasMap.java”, es por esto por lo que nuestras clases de dependencia pueden utilizar las diferentes funciones que se encuentran definidas en “TareasMap.java”.

- Principio de segregación de interfaces: este principio plantea que es preferible tener interfaces específicos para cada cliente en comparación a un único interfaz de propósito general.

En el caso particular de nuestro programa, podemos observar claramente dicho patrón ya que contamos con múltiples interfaces, por un lado “Dependencia.java” que se encarga de llamar a la función ordenEjecucion la cual a su vez se encarga de devolver la lista de tareas en el orden de ejecución

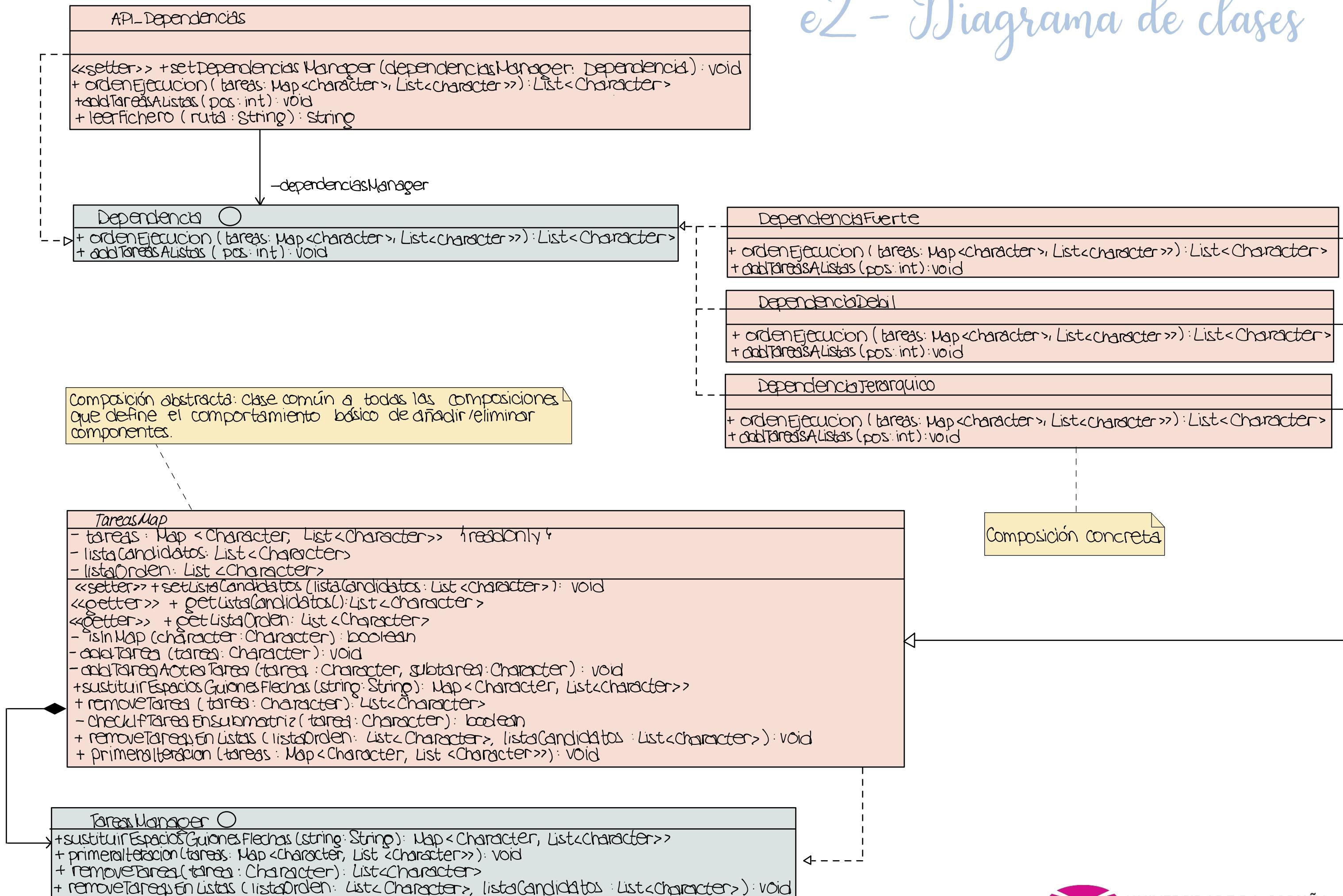
final, dependiendo este, de la dependencia utilizada y por otro lado nos encontramos con la interfaz “TareasManager.java” la cual se encarga de llamar a diferentes funciones encargadas de gestionar y tratar la lista de tareas dependiendo de la dependencia a implementar, estas gestiones pueden ser: devolver las subtareas de una tarea, borrar los elementos de las listas o mirar si es una tarea raíz, entre otras operaciones.

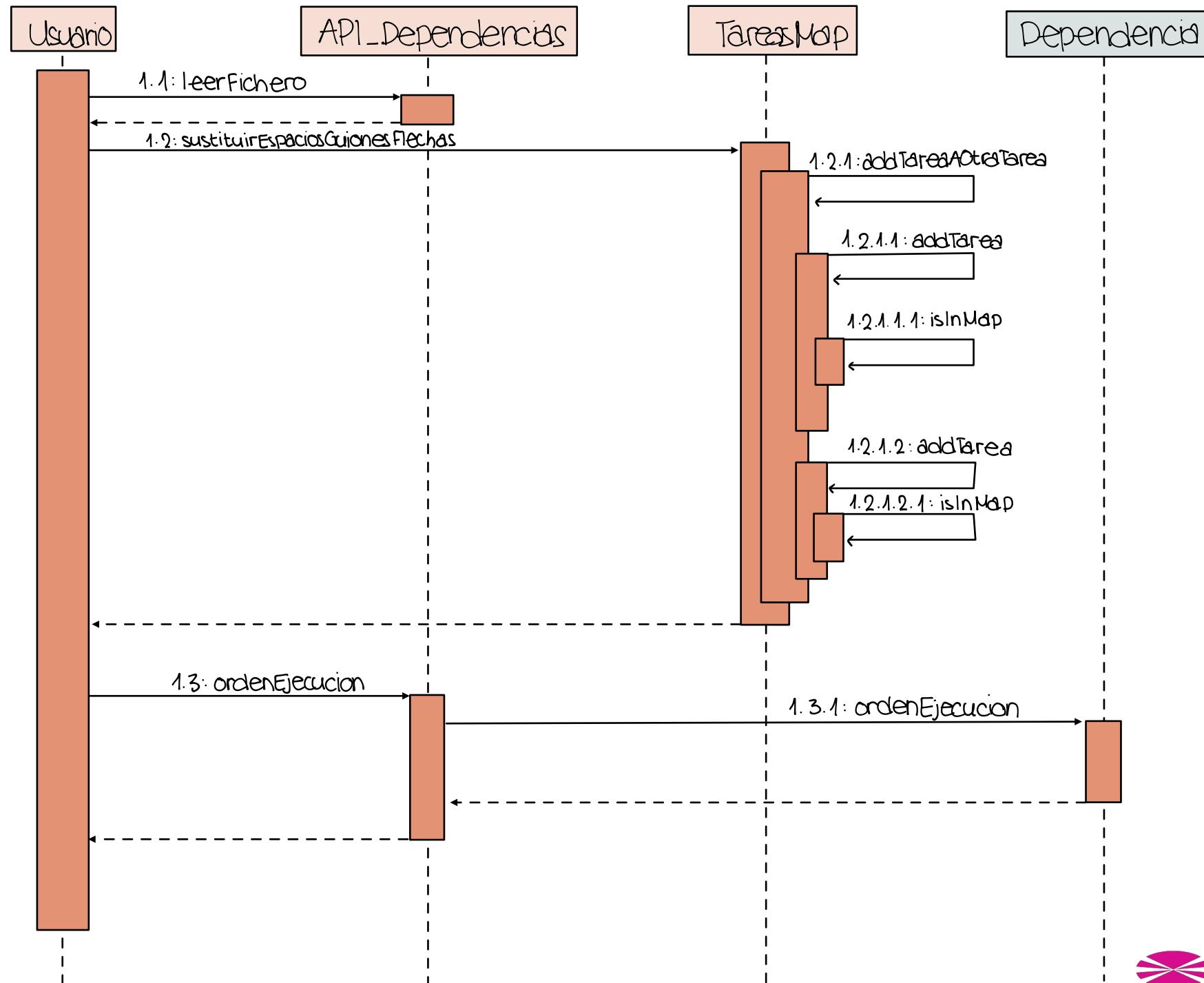
- Principio “Encapsula lo que varía”: este principio dice “Identifica los aspectos de tu aplicación que varían y sepáralos de aquellos que permanecen estables”. En el caso particular de nuestro programa, podemos observar claramente que este principio se cumple ya que está diseñado de tal manera que el cliente pueda tanto añadir como eliminar alguna de las dependencias sin afectar de ninguna manera al funcionamiento del programa ni al resto de clases.
- Principio DRY “Don’t Repeat Yourself”: ese principio plantea que cada pieza de conocimiento debe tener una representación única, inequívoca y autorizada dentro de un sistema.
En el caso particular de nuestro programa podemos observar fácilmente la utilización de dicho principio ya que evitamos utilizar el menor número de, por ejemplo, variables auxiliares, como también la definición de métodos que nos ayudan a optimizar el programa evitando así la repetición innecesaria de código.

Patrones de diseño utilizados en este problema:

- Patrón Estrategia: Este patrón de comportamiento se utiliza para definir una familia de algoritmos, encapsularlos y hacerlos intercambiables.
Utilizamos este patrón ya que así declaramos una interfaz común (Dependencias.java) para los algoritmos soportados (Débil, fuerte o Jerárquica) y de esta manera delegamos el funcionamiento desde el contexto del problema (siendo este la manera de tratar las dependencias, a la interfaz común).
En este caso concreto podemos decir que el contexto de nuestro patrón sería las llamadas “Tareas”, la estrategia sería nuestra interfaz “Dependencias” y por último nuestras estrategias concretas serían las diferentes dependencias que podemos aplicar, siendo estas: Débil, Fuerte y Jerárquica.
- Patrón Composición: es un patrón estructural que se utiliza para componer objetos en estructuras de árbol que representan jerarquías todo-parte.
En el caso concreto de nuestro ejercicio podemos observar este patrón y más concretamente “Composición Compleja” ya que podemos observar con facilidad la *Composición Abstracta* que en nuestro caso se llama “TareasMap.java” que es la que hace de clase común a todas las composiciones que define el comportamiento básico como puede ser añadir en la lista. Luego por otro lado nos conseguimos con las composiciones concretas que en nuestro caso son las clases “DependenciaDebil.java”, “DependenciaFuerte.java” y “DependenciaJerarquico.java”.

e2 - Diagrama de clases





Este esquema es aplicable a Dependencia jerárquico también.

