- E1

Para la implementación del ejercicio 1 decidimos usar **el patrón composición.**

Decidimos usar este patrón ya que a la hora de construir el código pensamos que la mejor manera de hacer era hacer agregaciones de los filtros de manera recursiva, y el patrón esta en la idea de clausulas “base” y las AND y OR que incluyen otras que pueden ser la su vez, las mismas una y otra vez.
El hecho de poseer una clase con una estructura de elementos de tipo interfaz, nos permite añadir instancias de clases que implementan dicha función, por lo que podríamos crear a partir de nuevas clases nuevas implementaciones de esta interfaz, sin tener que modificar el código.

El patrón composición viola el **principio de sustitución de Liskov**, ya que en la clase Gestor tenemos los métodos ‘add’ y ‘remove’ que deberían también ser heredados por las clases hoja, pero estas clases no poseen hijos, por lo que no tendría sentido.

También viola el **principio de responsabilidad única**, ya que podríamos instanciar una clase como una hoja o un contenedor, así que no tendrá únicamente una sola responsabilidad, deberíamos comprobar esto en tiempo de ejecución si queremos evitar esto.

El principio **Abierto-cerrado** es un gran amigo de este patrón, ya que como dice este principio “una clase tiene que estar abierta a su extensión, pero cerrado a su modificación”, como hemos explicado arriba, no necesitamos modificar código, solo añadirlo.

**Principio dry**: En diseño orientado a objetos, El principio No te repitas (en inglés Don't Repeat Yourself o DRY, también conocido como Una vez y sólo una) establece que, en un entorno informático, la información no debe repetirse. Es decir, el conocimiento almacenado en un programa informático debe mantenerse en un, y sólo en un, lugar.

**Principio kiss:** La simplicidad debe ser mantenida como un objetivo clave del diseño, y cualquier complejidad innecesaria debe ser evitada.

**Principio Yangui:** Implementar los cambios cuando realmente se necesiten, no cuando solo se prevea que se necesitar

En cuanto a los diagramas, hemos realizado un diagrama de clases concorde al patrón, y un diagrama de secuencias, basado en la función ejecutada en el bucle de la instancia contenedor, que nos permite obtener de manera recursiva una lista filtrada gracias a todas esas clases hoja que implementan dicha función y se encuentran dentro de la estructura del contenedor.

**Autores:** Clara Barrios Hermida // Santiago Julio Iglesias Portela

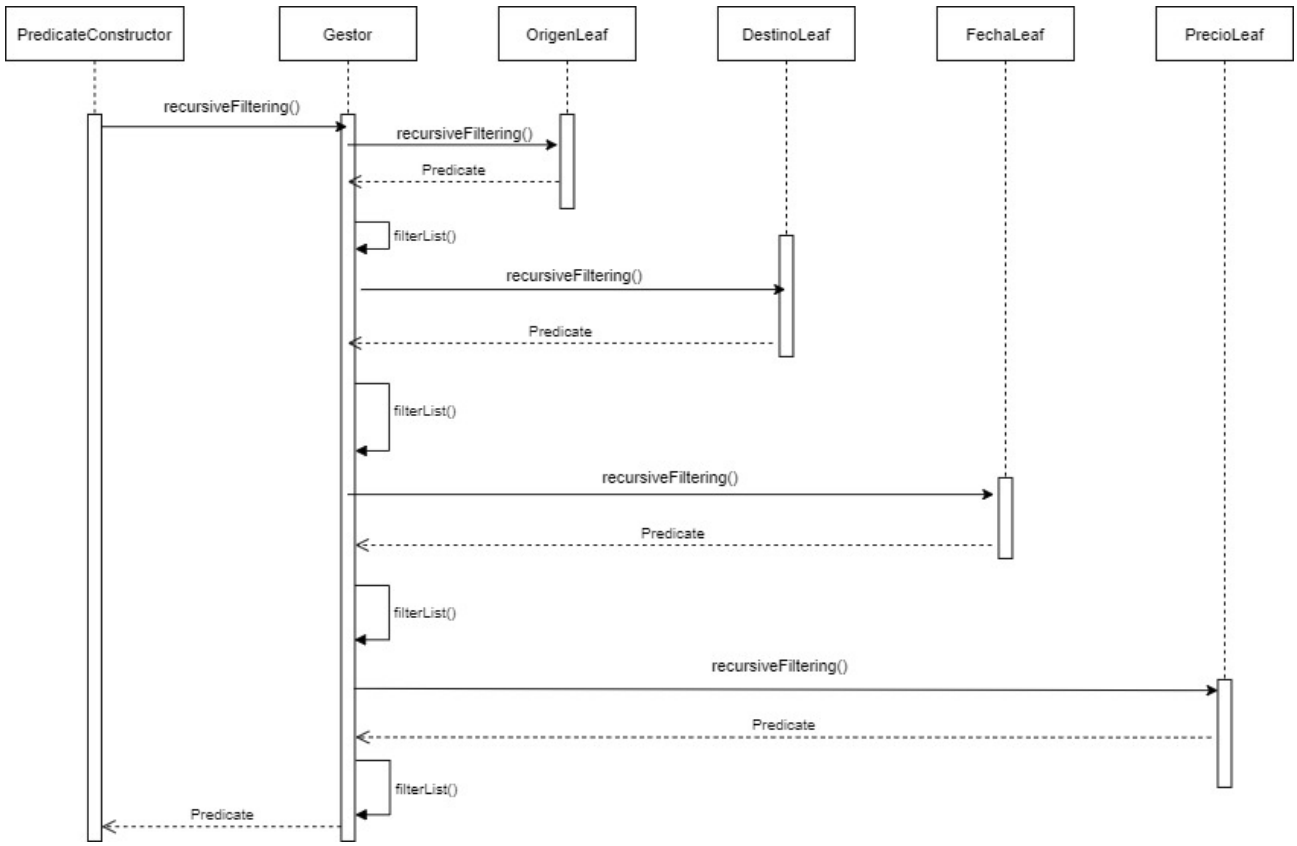
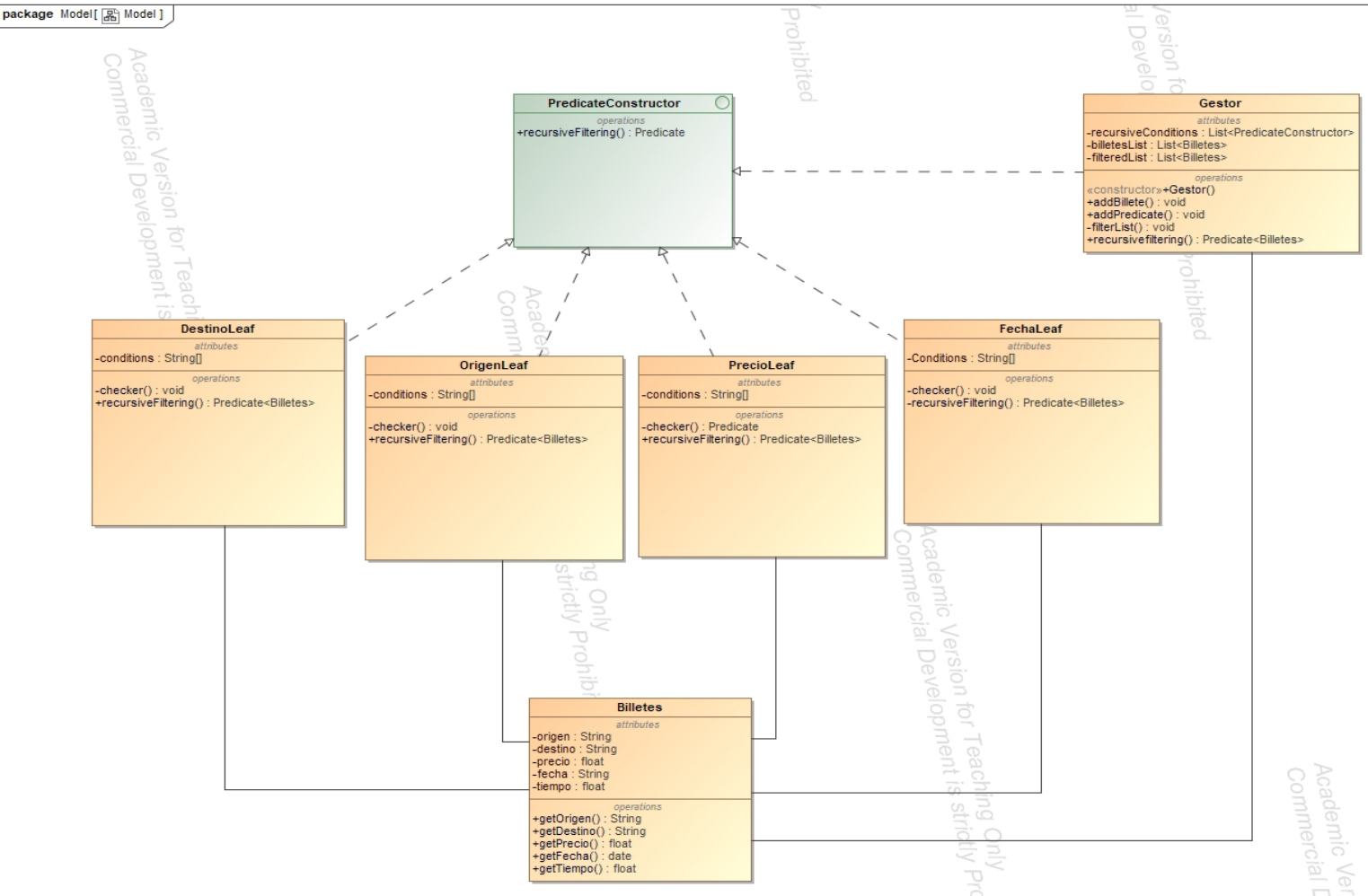



Diagrama de clases e1



- E2

Para la implementación del ejercicio 2 decidimos usar el **patrón de diseño Estrategia.**

El por qué de nuestra decisión fue porque al final del enunciado pone "*Ten en cuenta que, en un futuro, es posible que aparezcan nuevos ordenes de ejecución que deberíamos poder integrar fácilmente en nuestro diseño"* .

Por esta razón decidimos usar el patrón estrategia ya que cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato a utilizar el patrón estrategia.

Este patrón nos permite tener cualquier número de estrategias posibles para resolver el problema, y en cualquier momento una estrategia puede ser cambiada por otra, incluso en tiempo de ejecución.
Además añadir estrategias nuevas resulta muy sencillo, ya que no hay que hacer grandes cambios en el código para añadir un nuevo algoritmo, simplemente crear una nueva clase.

En cuanto a los principios de diseño, usamos los **SOLID** el primero que usamos fue el **Principio de Responsabilidad Única** ya que cada objeto tiene una responsabilidad única que esta enteramente encapsulada en la clase, es decir que cada clase debe tener una razón para cambiar, un objeto debe realizar una única cosa.

El **principio abierto-cerrado** sería otro de los principios usados. Este expresa la necesidad de que ante un cambio de los requisitos, el diseño de las entidades existentes permanezca inalterado, recurriéndose a la extensión del comportamiento de dichas entidades añadiendo nuevo código, pero nunca cambiando el código ya existente**.**

Otro usado es el **principio de sustitución de Liskov** nos dice que si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido.

Con el **principio de inversión de dependencia** los módulos de alto nivel no dependen de los módulos de bajo nivel. Ambos dependen de interfaces. Las abstracciones no dependen de detalles.

Además usamos otro tipo de principios:

**"Principio de Hollywood":** su nombre viene de la habitual frase empleada por los directores de casting en Hollywood para evitar estar recibiendo llamadas de aspirantes preguntado si han sido aceptados o no. Existe un modelo en diseño de software, llamado **inversión de control**, en el que el flujo de ejecución de un programa es completamente distinto o "invertido" a los métodos de desarrollo tradicionales. En lugar de procesar los datos que hay en el momento del proceso, el propio software se encarga de conectar con la entidad correspondiente y obtener el dato que necesita en cada momento.

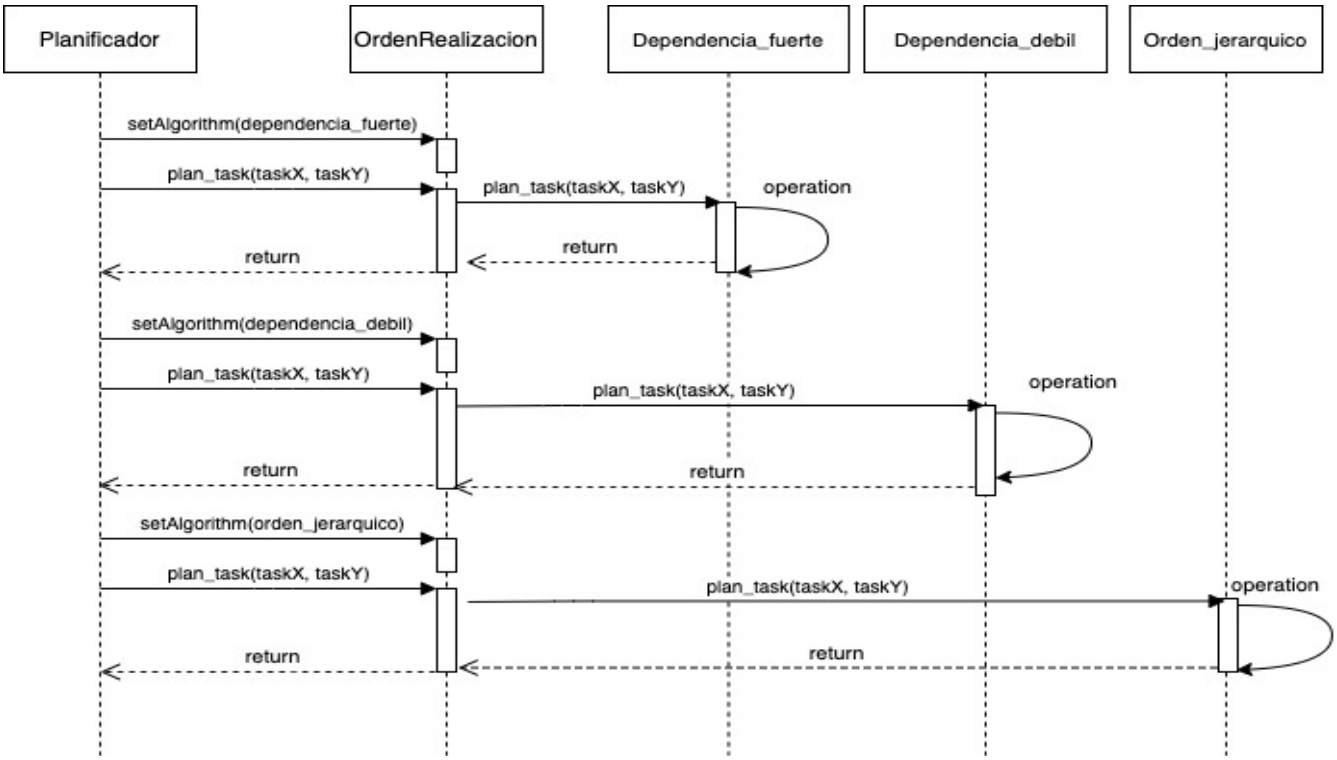
**Principio dry**: En diseño orientado a objetos, El principio No te repitas (en inglés Don't Repeat Yourself o DRY, también conocido como Una vez y sólo una) establece que,

en un entorno informático, la información no debe repetirse. Es decir, el conocimiento almacenado en un programa informático debe mantenerse en un, y sólo en un, lugar.

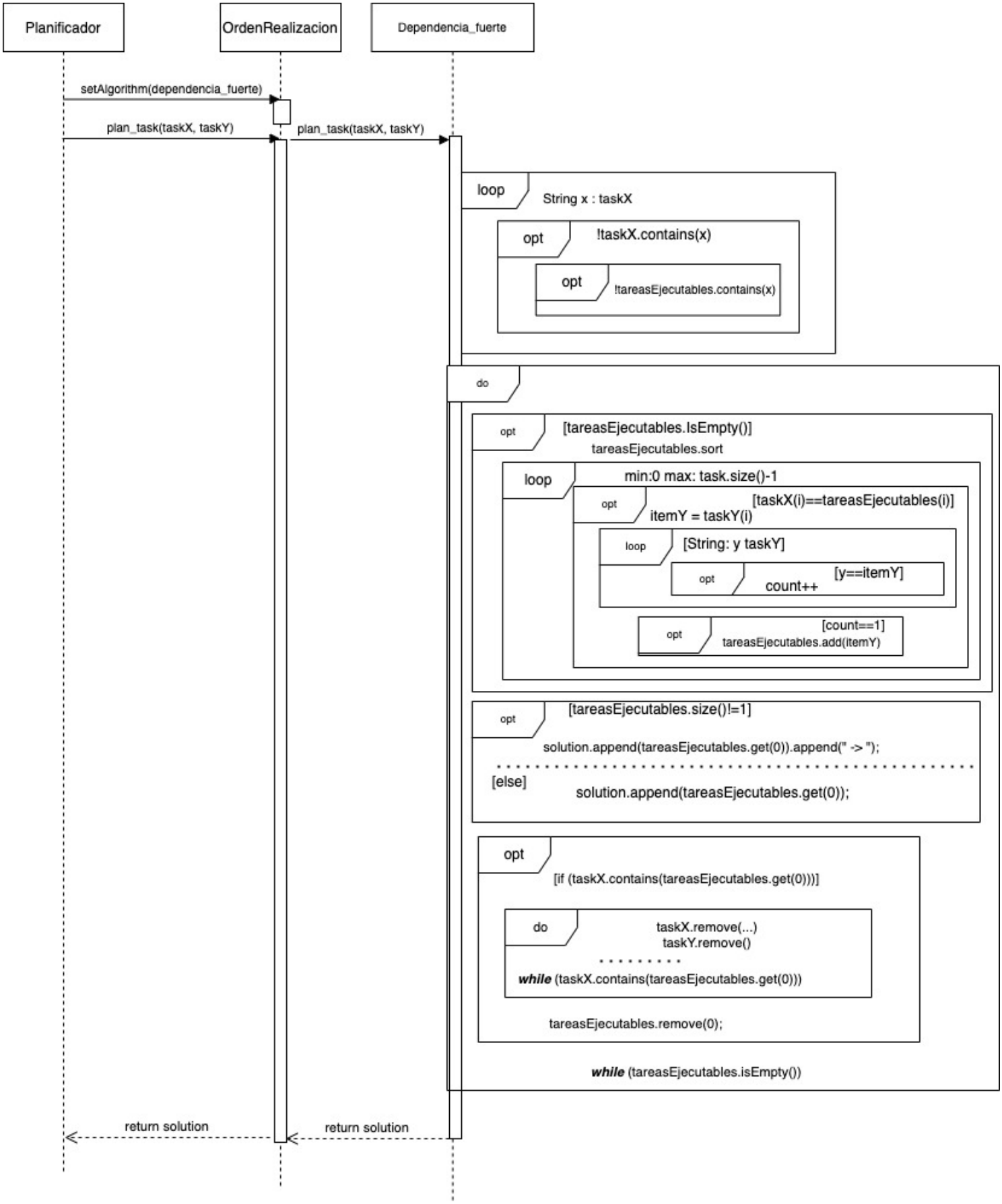
**Principio kiss:** La simplicidad debe ser mantenida como un objetivo clave del diseño, y cualquier complejidad innecesaria debe ser evitada.

**Principio Yangui:** Implementar los cambios cuando realmente se necesiten, no cuando solo se prevea que se necesitarán.

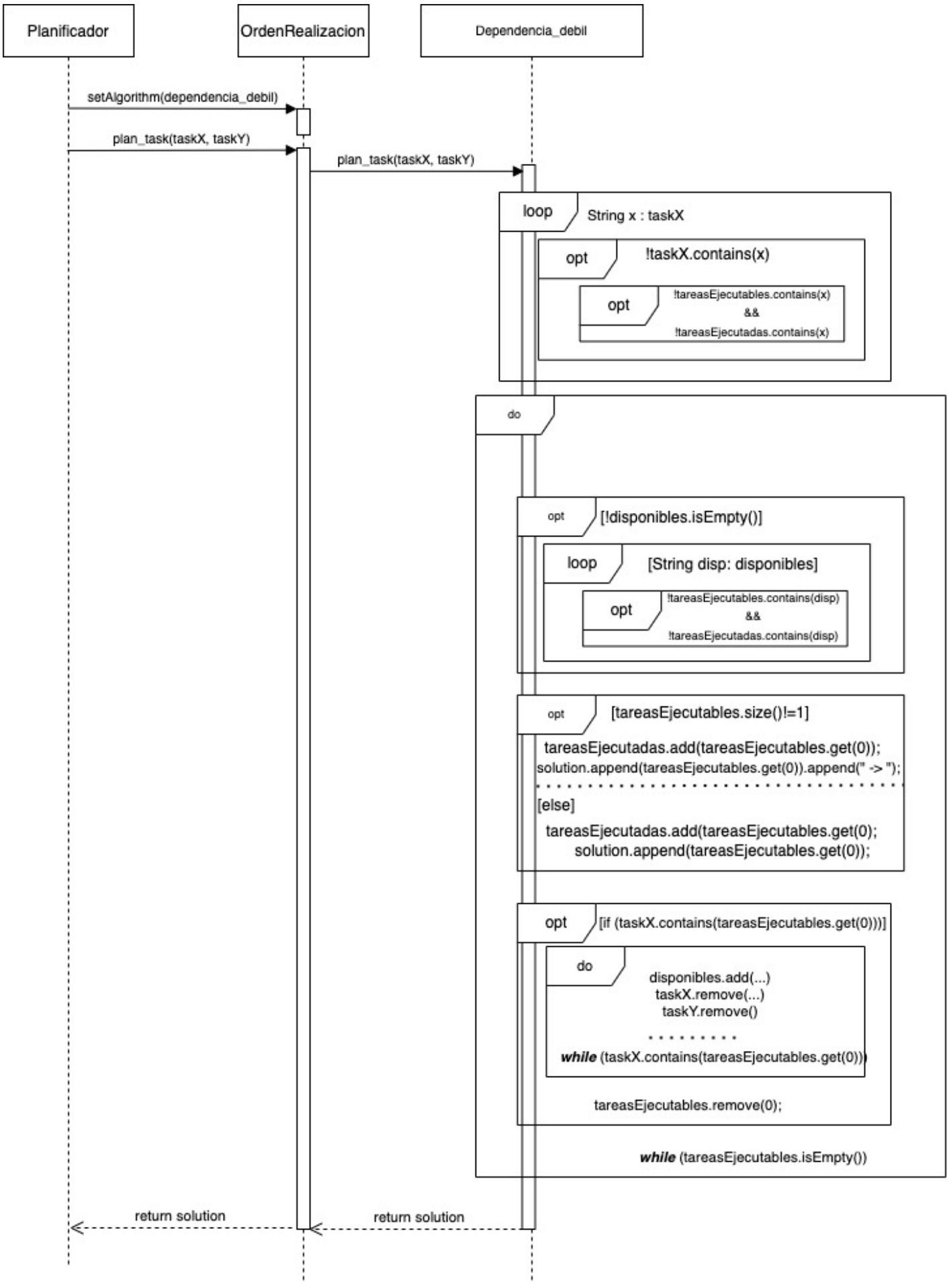
En cuanto a los diagramas dinámicos elegimos el de secuencia, para este ejercicio hicimos 3. Uno que explica como se comunican todas las clases del ejercicio y los otros dos explican el comportamiento de la función “plan_task” en la clase dependencia fuerte y débil. No hemos hecho el diagrama de secuencia de la clase orden jerárquico porque su comportamiento es similar al de dependencia débil pero cambiando algunas líneas de código de lugar.




**Autores:** Clara Barrios Hermida // Santiago Julio Iglesias Portela

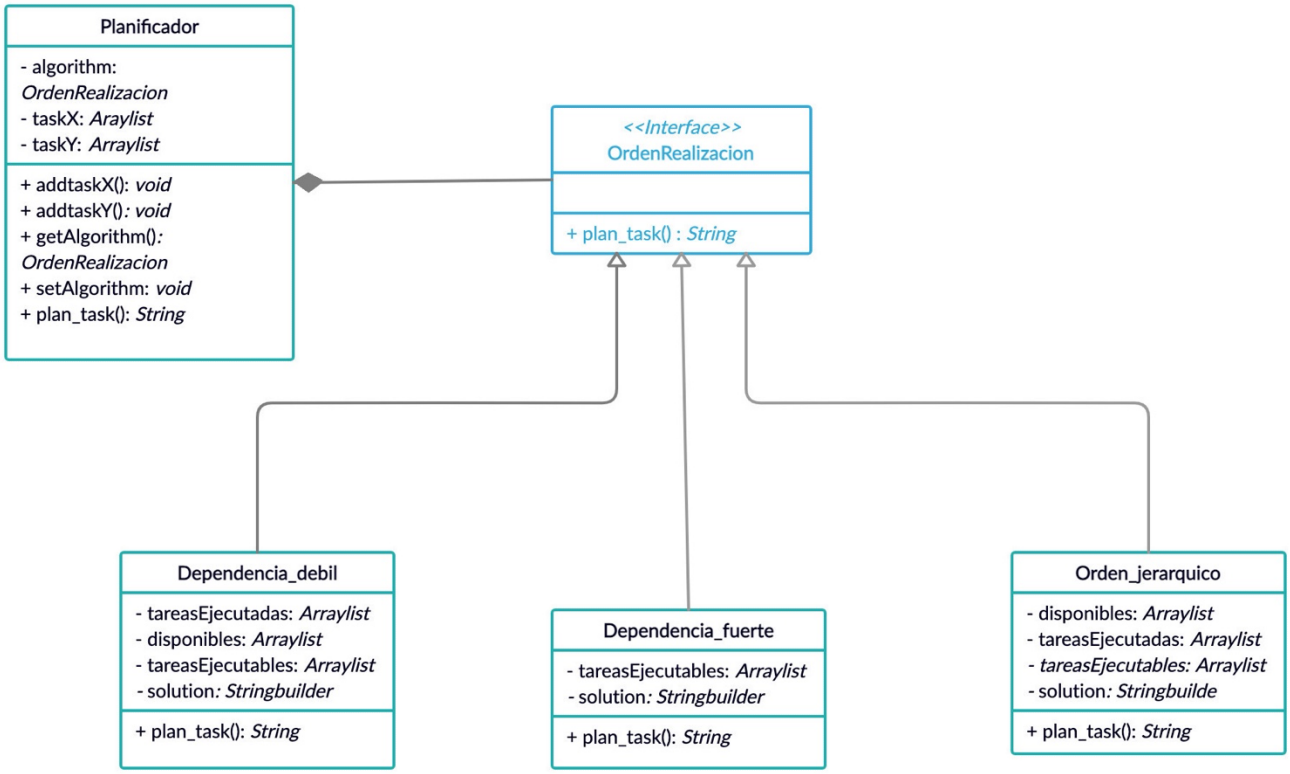



**Autores:** Clara Barrios Hermida // Santiago Julio Iglesias Portela




**Autores:** Clara Barrios Hermida // Santiago Julio Iglesias Portela

Diagrama de clases e2:
Para este diagrama la interfaz seria OrdenRealización que juega el rol de STRATEGY , la clase Planificador correspondería al rol de CONTEXT y las clases dependencia_debil, dependencia_fuerte y orden_jerarquico corresponden a los roles CONCRET STRATEGY 1 CONCRET STRATEGY2 y CONCRET STRATEGY3 respectivamente.




**Autores:** Clara Barrios Hermida // Santiago Julio Iglesias Portela