

Ejercicio 2:

Explicación de los principios de diseño usados:

Empezando por los diseños solid, hemos utilizado:

-El principio de responsabilidad única, con el claro ejemplo de la clase tarea, la cual se encarga de gestionar el estado interno de la propia tarea sin saber nada de la lista que la contiene o si sus dependencias existen en la misma. Esto se debe a que esa parte es delegada en el gestor de tareas cuya responsabilidad es controlar la lista de las tareas, haciendo las operaciones relacionadas con esta. Y luego estan las distintas estrategias concretas donde cada una tiene como única responsabilidad encontrar la forma de resolver una lista de tareas con la que no interactúan, pues de nuevo delegan en el gestor de tareas.

.-El principio de sustitución de Liskov con la estrategia(la clase abstracta) y las estrategias concretas(las clases que heredan de la primera). Se cumple el principio ya que ambos métodos no abstractos definidos en la estrategia son comunes para todas las concreciones, permitiendo a cualquiera de estas hacerse pasar por su superclase sin ningún problema.

El ejercicio no cumple el resto de los principios solid, pues no es abierto-cerrado al no tener ninguna subclase sobreescribiendo a su superclase para poder modificarla sin saber como esta implementada esa superclase por dentro, aunque lo cumpliría si, por ejemplo, creáramos otro algoritmo que fuera especificase una de las estrategias concretas, por ejemplo haciendo que la dependencia débil en vez de, cuando hay mas de un candidato, decidir el adecuado por orden alfabético cogerlo por ejemplo en el sentido contrario. Y no cumple el de segregación de interfaces al solo necesitar una pequeña clase abstracta la cual define el comportamiento de todas las estrategias.

El resto de principios que cumple son:

-El principio de inversión de la dependencia con la estrategia, es decir, la clase abstracta y la clase de gestor_tareas pues la segunda depende de una abstracción obviando así cualquier posible interpretación que esta pueda tener. Para ello tiene como uno de sus atributos un objeto de la clase abstracta que puede ser cualquiera de las clases que heredan de ella.


-El principio de encapsula lo que varía al utilizar el patrón estrategia, pues dicho patrón encapsula la parte mas propensa a cambios, a la que llama estrategia mediante un interfaz o clase abstracta y luego dichos cambios se pueden crear fácilmente implementando la interfaz o heredando la clase abstracta, mientras por otro lado tienes las clases menos propensas a cambios como el contexto que interacciona con el interfaz.

Explicación del patrón usado:

Para resolver este ejercicio nos decidimos por usar el patrón estrategia al ser el que más se adecuaba al problema. La razón de esta adecuación es que el patrón estrategia te permite separar la parte del código inmutable de tu programa (o que rara vez se va a alterar, por ejemplo en este ejercicio el gestor de tareas será siempre el mismo, independientemente de

que método usemos para resolver las tareas) o contexto (una vez mas, en este ejercicio seria el gestor de tareas) de la parte mas mutable o con probabilidad de cambios (las estrategias concretas, en este caso, las 3 formas de obtener como hacer las distintas tareas, es decir, la dependencia fuerte, la débil y la orden jerárquica). Esto nos facilita quitar/modificar o añadir con mucha mas facilidad los distintos algoritmos que tenemos a nuestra disposición sin afectar a la parte común de todos. Para hacerlo, usamos una interfaz (o en nuestro caso una clase abstracta puesto que todos los algoritmos comparten una parte importante del código) que será la que se relacione con nuestra parte inmutable.

Diagrama de clases:

package Model[ Model]

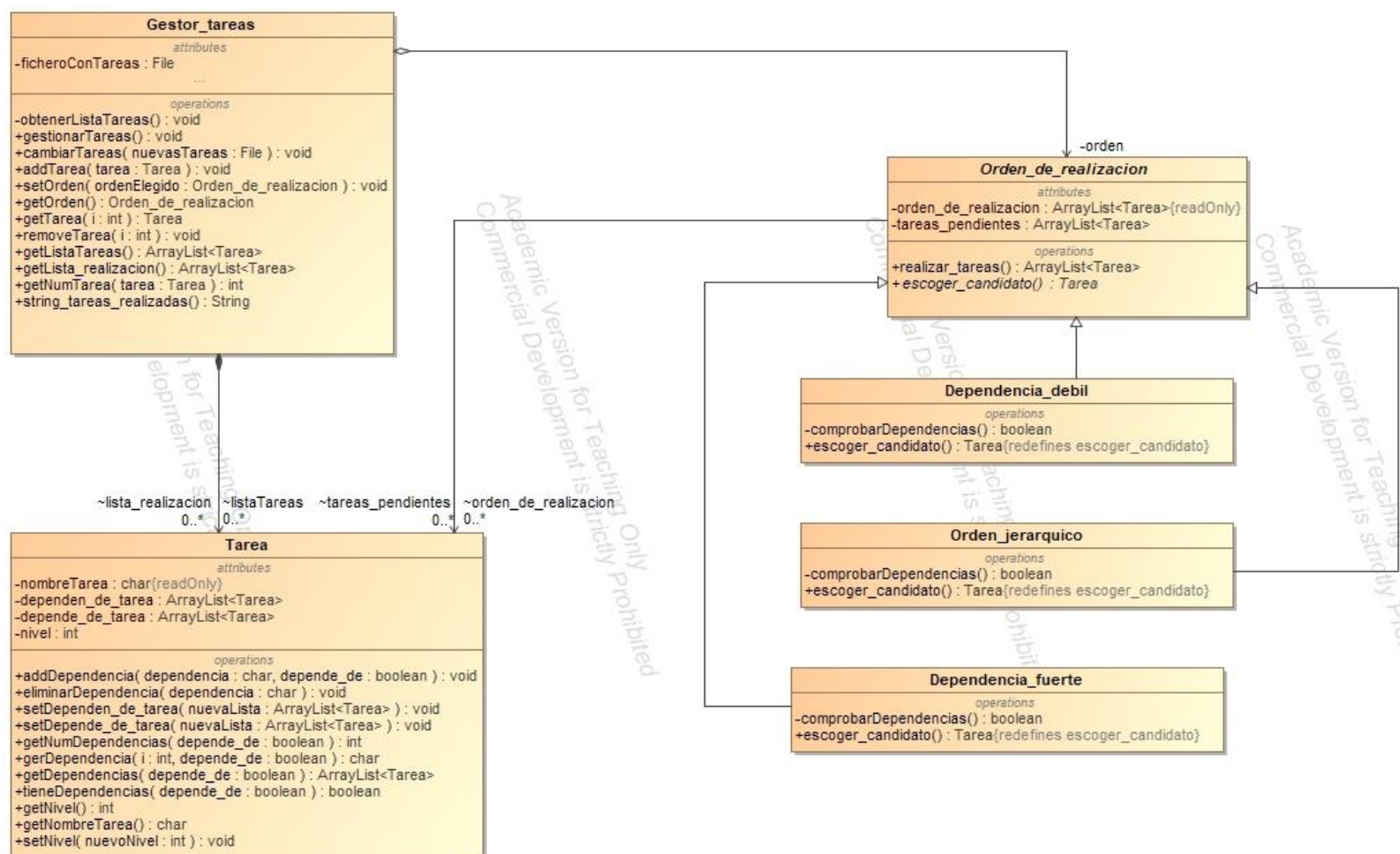


Diagrama de secuencia:

