SUMMARY OF THE GIVEN PROBLEM:

We are given a set of relation between tasks on the form "X -> Y" where X and Y are the tasks and the arrow indicates that X must be executed prior to Y
All tasks will be represented with a character

The objective is to obtain the order of tasks to be followed, given a certain input, and given a certain interpretation:
- strong dependency
- weak dependency
- hierarchical
When two task could be processed, we must follow alphabetical order

Moreover, our solution must facilitate storing the task graph and obtaining the distinct execution orders described above, while allowing, and if possible, facilitating the implementation of new execution orders in the future

OVERVIEW:

Given the fact that we are asked specifically to be able to easily sort our tasks in one of the different given ways, we use the strategy pattern, which lets the algorithm vary independently form its clients
Moreover, in order to represent the relations between the different tasks, we decided to implement a non-weighted directed graph, where the nodes are the tasks

TASKS AND THE GRAPH:

Naturally, we started creating a class Task, which stores the char that is used as an identifier.
Then, we started constructing the graph in a separated class, Graph: we decided to implement it using adjacency lists, since it seemed to be the most natural way to do it given its similarities with the given input
We decided to store the "Children" of each tasks inside the Task itself, to ease its access and to simplify the Graph class (done by a LinkedList)
We implemented the necessary methods such as getID(), constructors, a method to check if two tasks are related (check relation) adding a child (addChild)
The need to sort the tasks, as mentioned by the given problem, by their ID, led us to make an specific Task comparator, to ensure the correct behaviour of the comparator method

Given this Task class, the class Graph needed to store an ordinary list of tasks.
Most of the methods in this class have Task or Lists of tasks as input/ output even though most of the code would be easier to program if they worked with integer (easier to access to the elements given the indexes), in order for the other classes to remain as independent from the specific implementation of the Graph (the private taskToPos method helps with this)

We however, decided to make most of the methods public: since we are expected to be able to implement new methods, it is easier to let methods of the graphs accessible that to try and guess which information will be needed by future algorithms: we see the advantages of a more private code, only letting very specific information be known, but decided to sacrifice that in order to make the code more future- proof

Moreover, the class Graph is the one that plays the Context role for the Strategy pattern

SORTING ALGORITHMS:

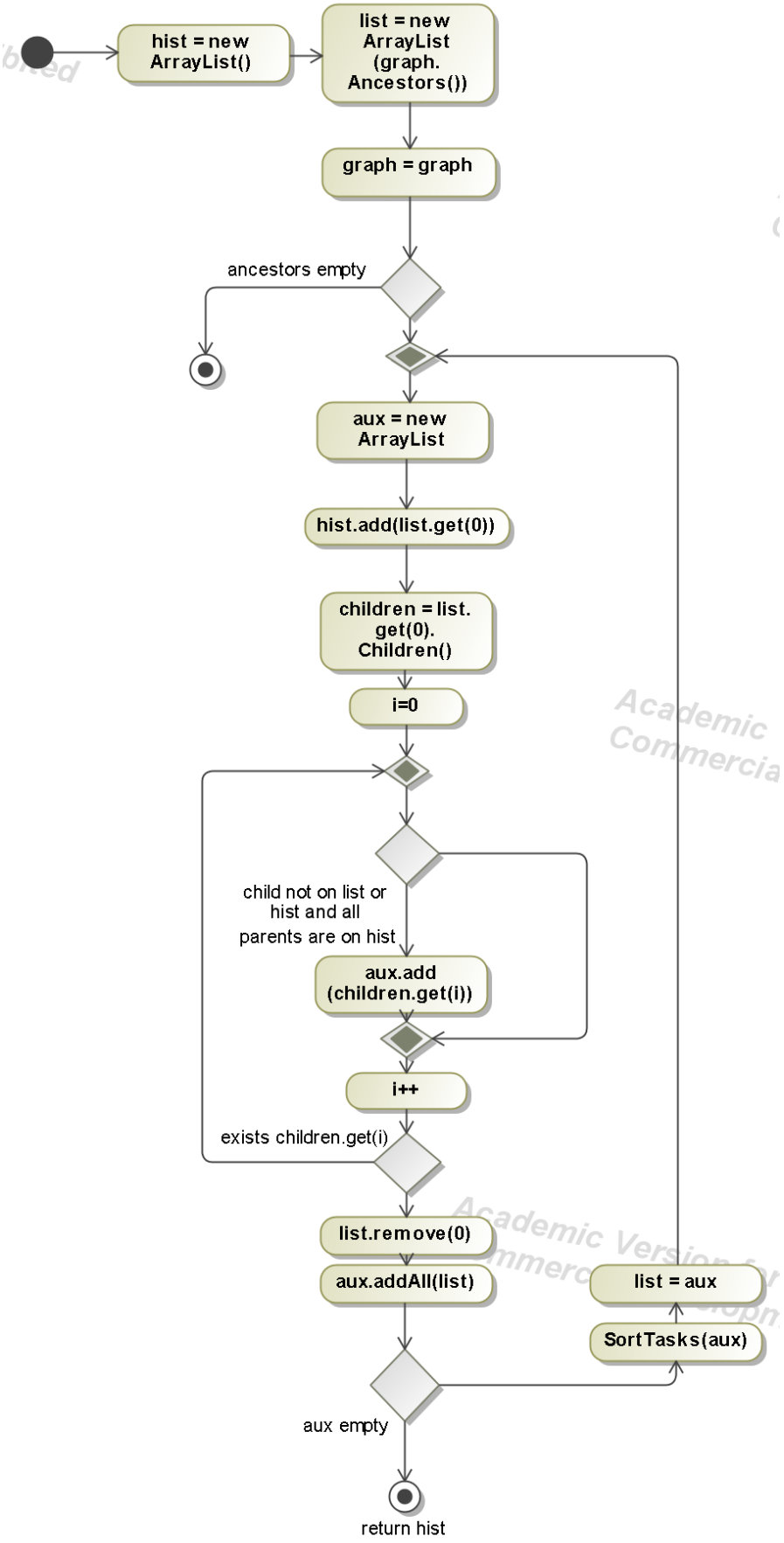
We started declaring an interface that would be common to all algorithms:
At first the method sort() had code common between the 3 given algorithms, but we ended up rejecting the idea since we are told that more algorithms could be implemented in the future, and that would require more work, hence, we decided to leave just the sort(), without any specific implementation.

This interface plays as the Strategy role

Referring to the specific algorithms, to ease the use of recursion (which seemed the most natural to us given the problem), we decided to implement a different auxiliary method, called from the sort() method, which returned the list (at first it returned void, but we wanted to keep the List since it already has stored the result)

All of the implementations use auxiliary lists, which behave closely to queues (it's their head node the one that is always processed), but we preferred a the less restricting lists, since it eased some operations
The specific diagrams to each algorithm are the following

Strong dependency sorting algorithm:




```
Weak dependency sorting algorithm:
![[e2_activity_dynamic_diagram_weak 1.png]]

Hierarchical sorting algorithm:
![[e2_activity_dynamic_diagram_hierch 1.png]]
```

All these algorithms work as the Concrete Strategy role

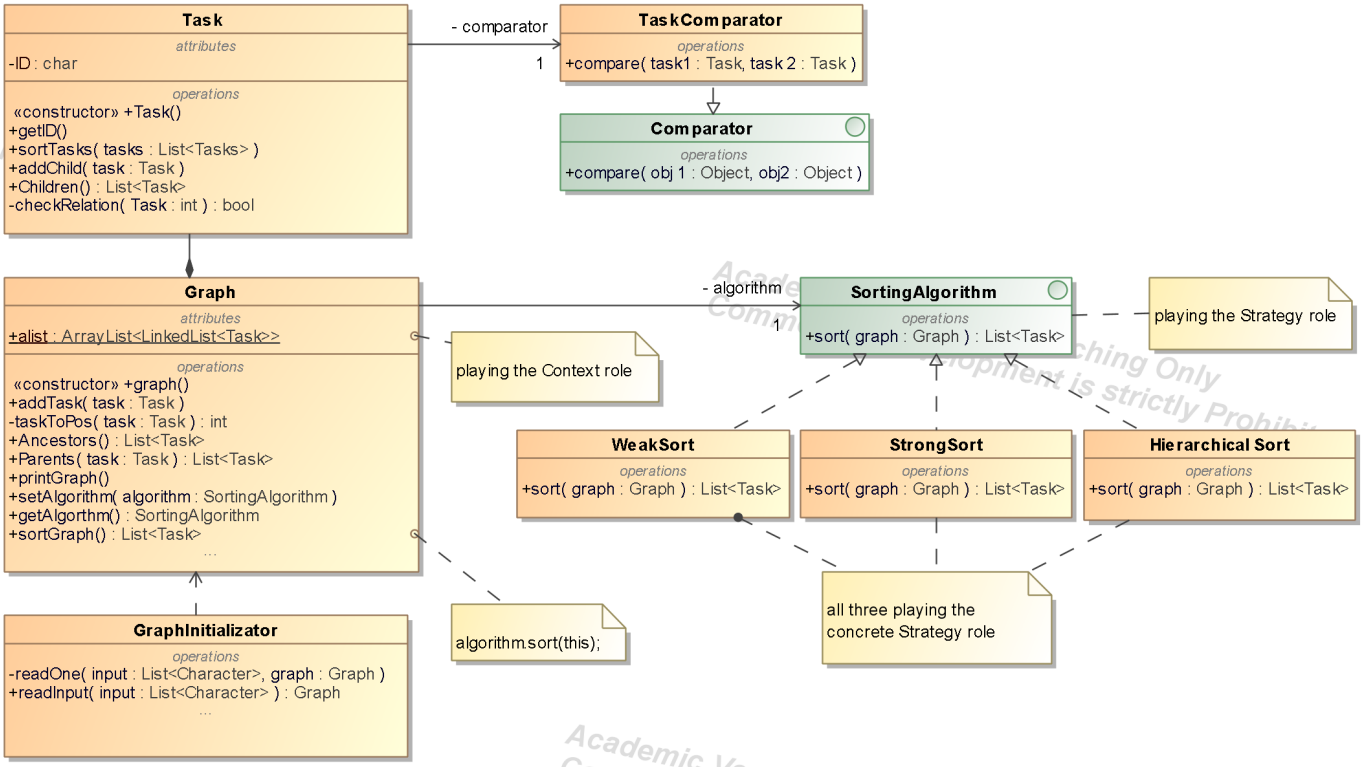
INPUT:

We need to get a graph from the given input (interpreted on the tests as a List) we decided to implement an specific class for this, to keep what each class does to a minimum

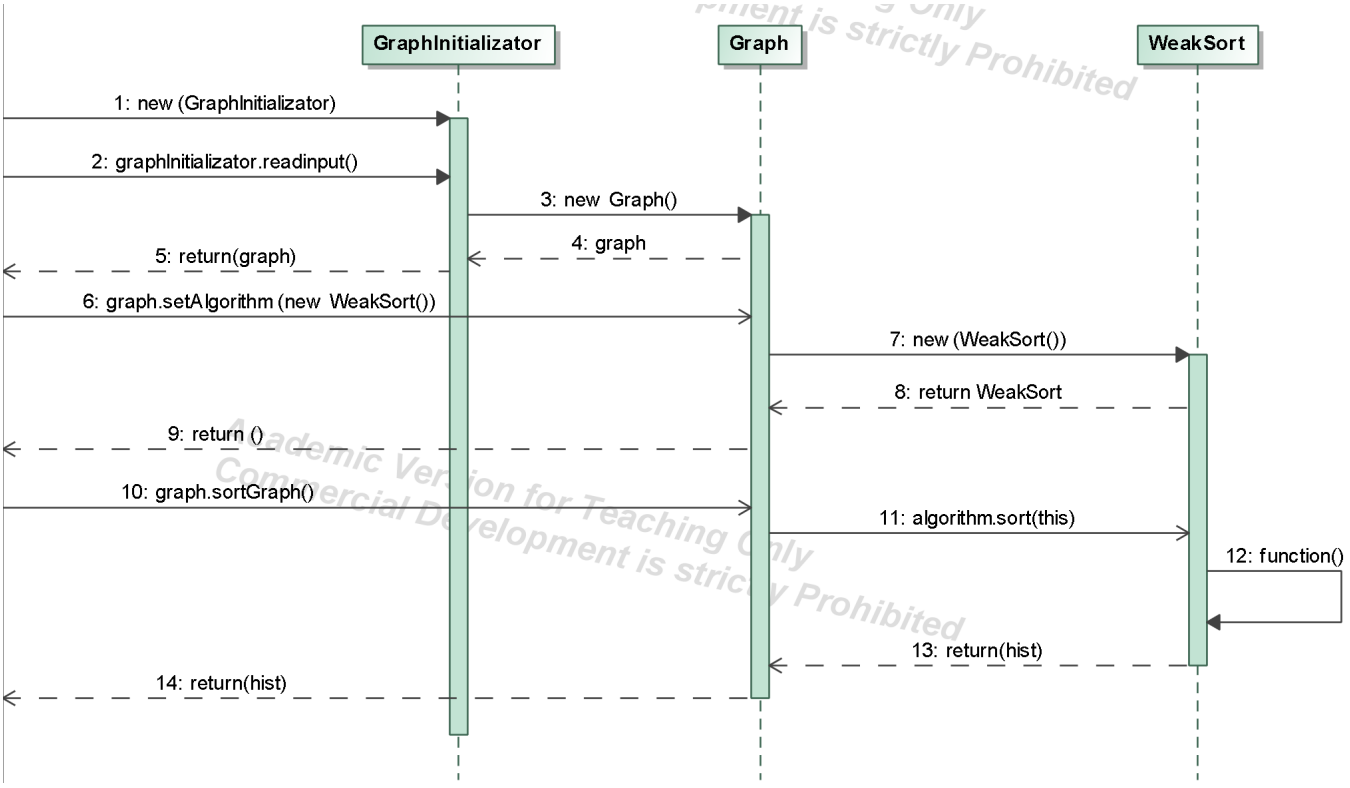
DIAGRAMS:

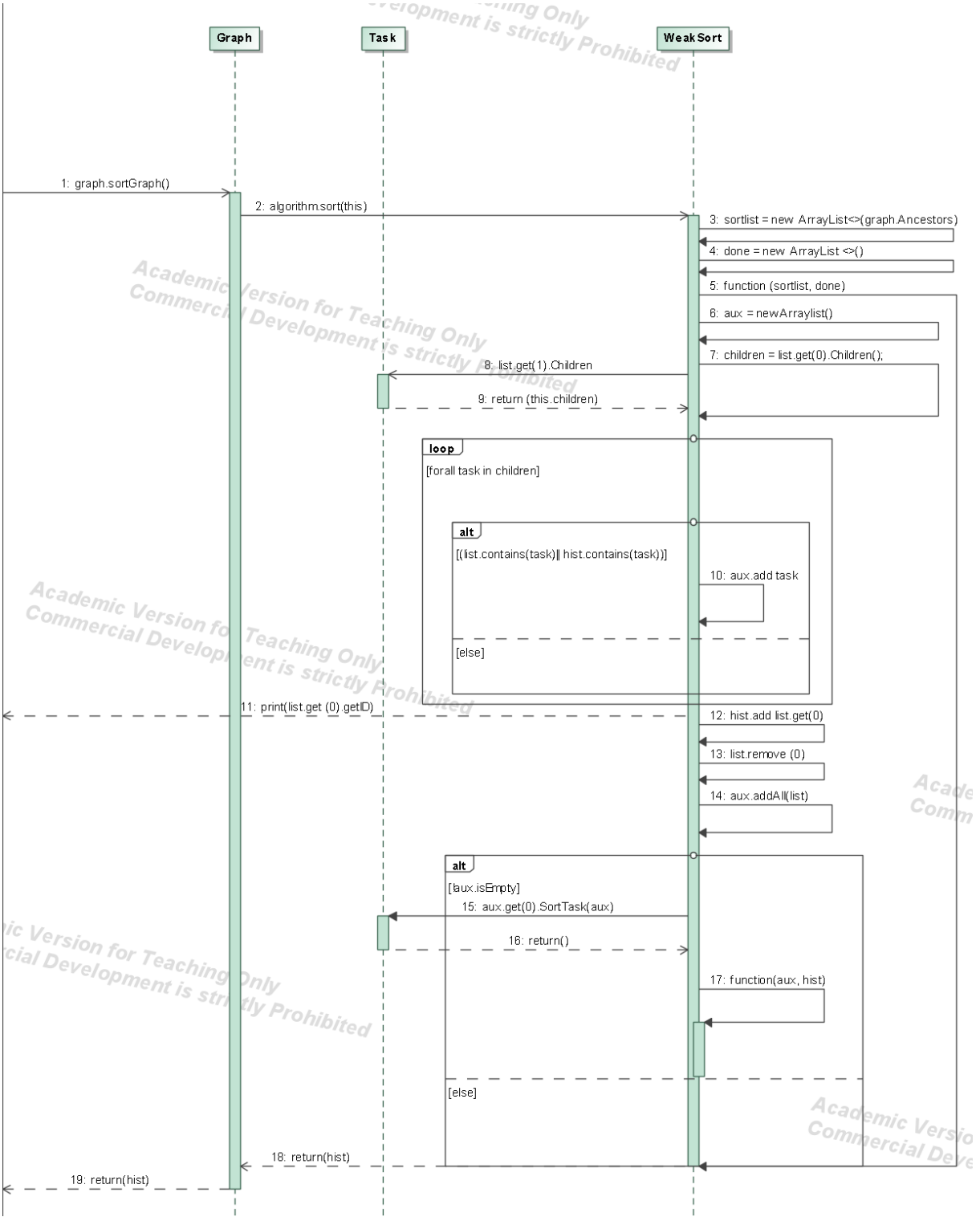
The diagrams contain the summary of this project are:

The class diagram:



Two dynamic diagrams representing the overview of the creation, uses and overall calls for a sort() without much detail, and a detailed explanation of the calls made by an specific algorithm, the WeakSort, respectively :





There are other activity diagrams, since we thought that their represent better the inner workings of the algorithms