SECOND EXERCISE

SUMMARY OF THE GIVEN PROBLEM:

We are given a set of relations 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 tasks 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 from 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:

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 todo it given its similarities with the given input.

We decided to store the "Children" of each task inside the Task itself, to ease its access and to simplify the Graph class (done by a LinkedList).

The need to sort the tasks, as mentioned by the given problem, by their ID, led us to make a specific Task comparator, to ensure the correct behavior of the comparator method.

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 integers (it is 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 graph accessible than 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.

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

SORTING ALGORITHMS:

We started by 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.

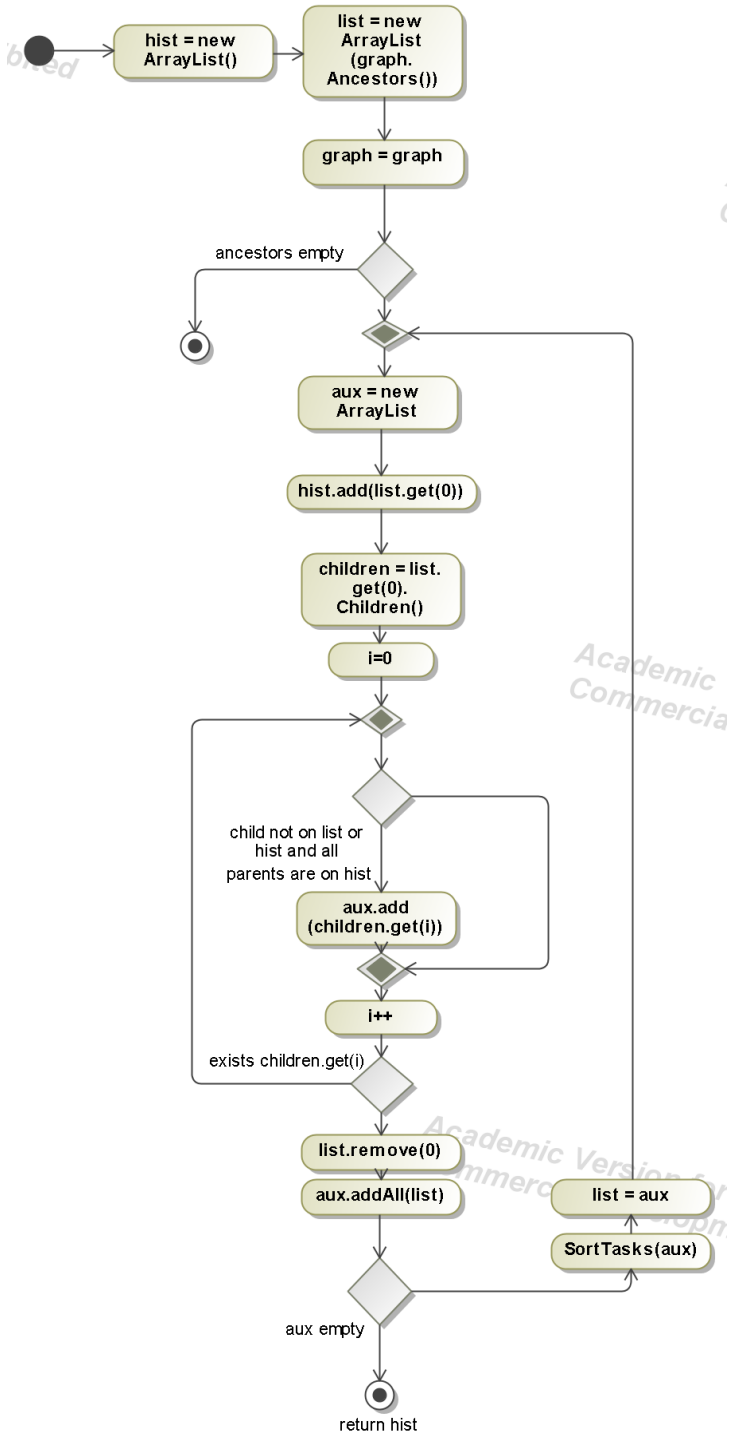
This interface plays the Strategy role.

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).

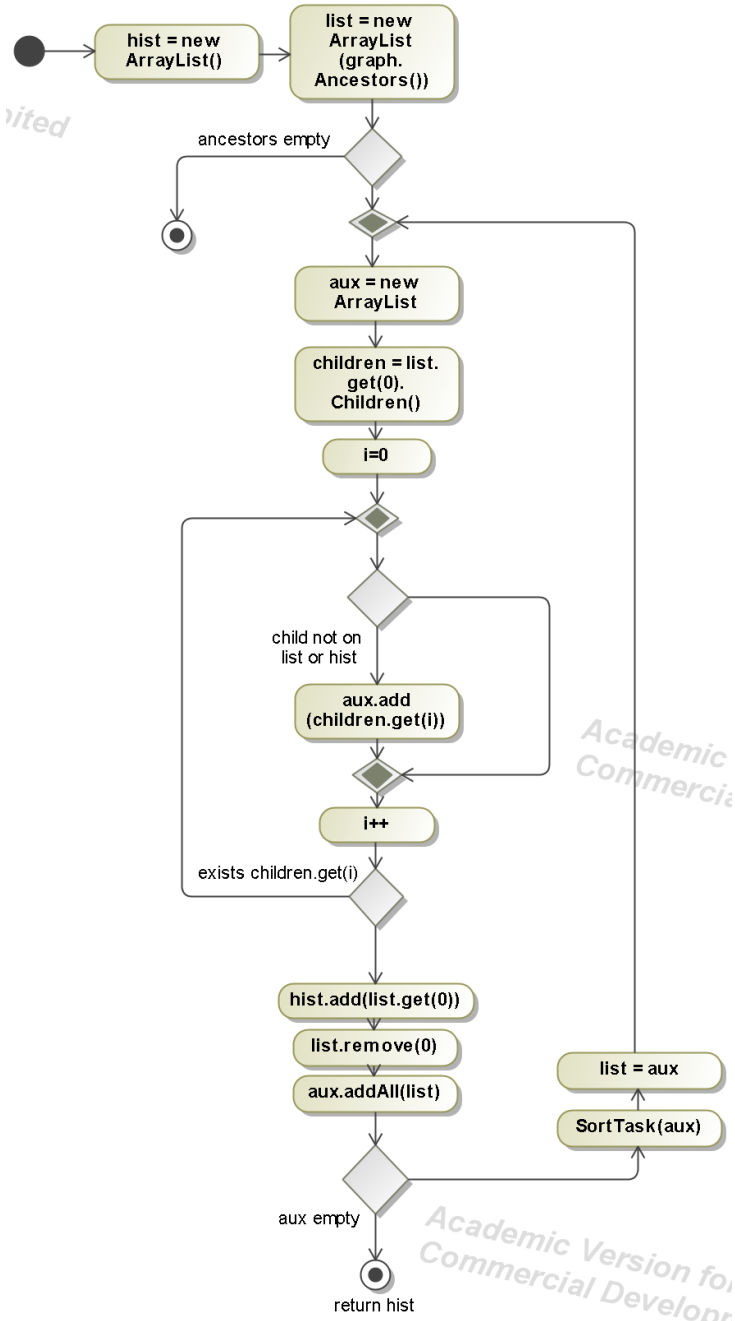
All of the implementations use auxiliary lists, which behave closely to queues (it's their head node the one that is always processed), but we preferred the less restricting lists, since it eased some operations.

The specific diagrams to each algorithm are the following:

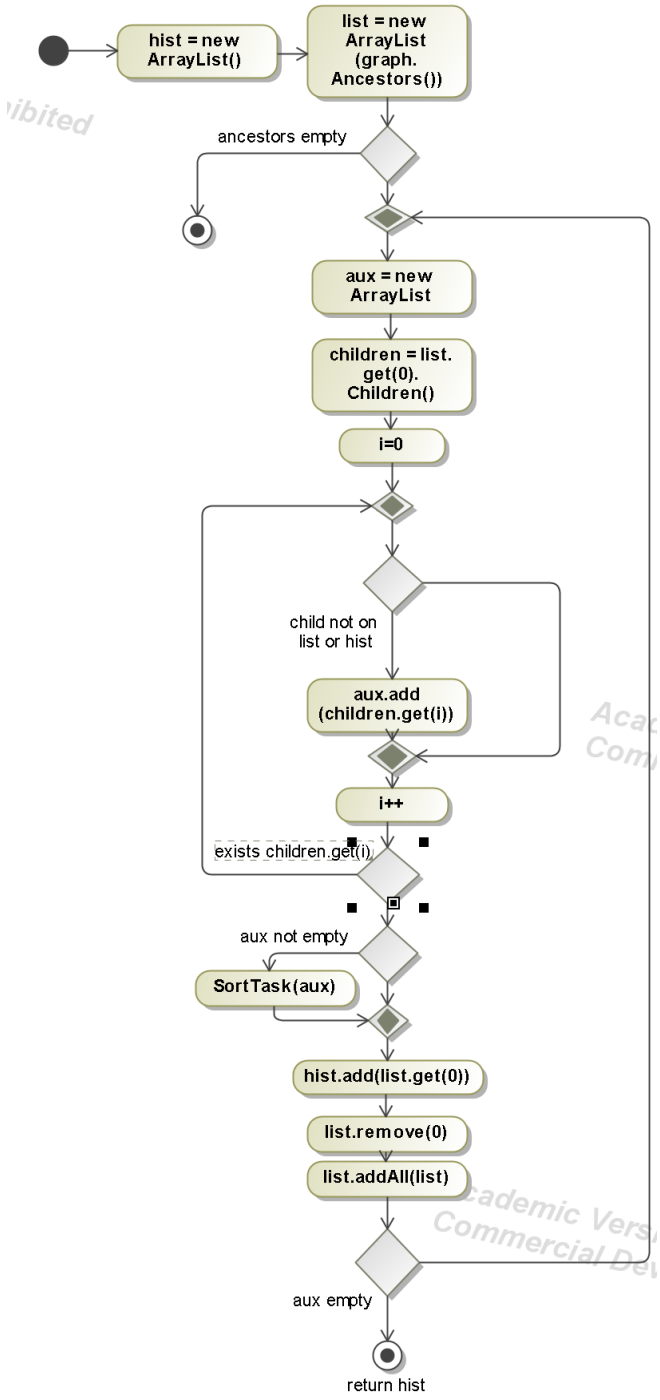
Strong dependency sorting algorithm:



Weak dependency sorting algorithm:



Hierarchy sorting algorithm:

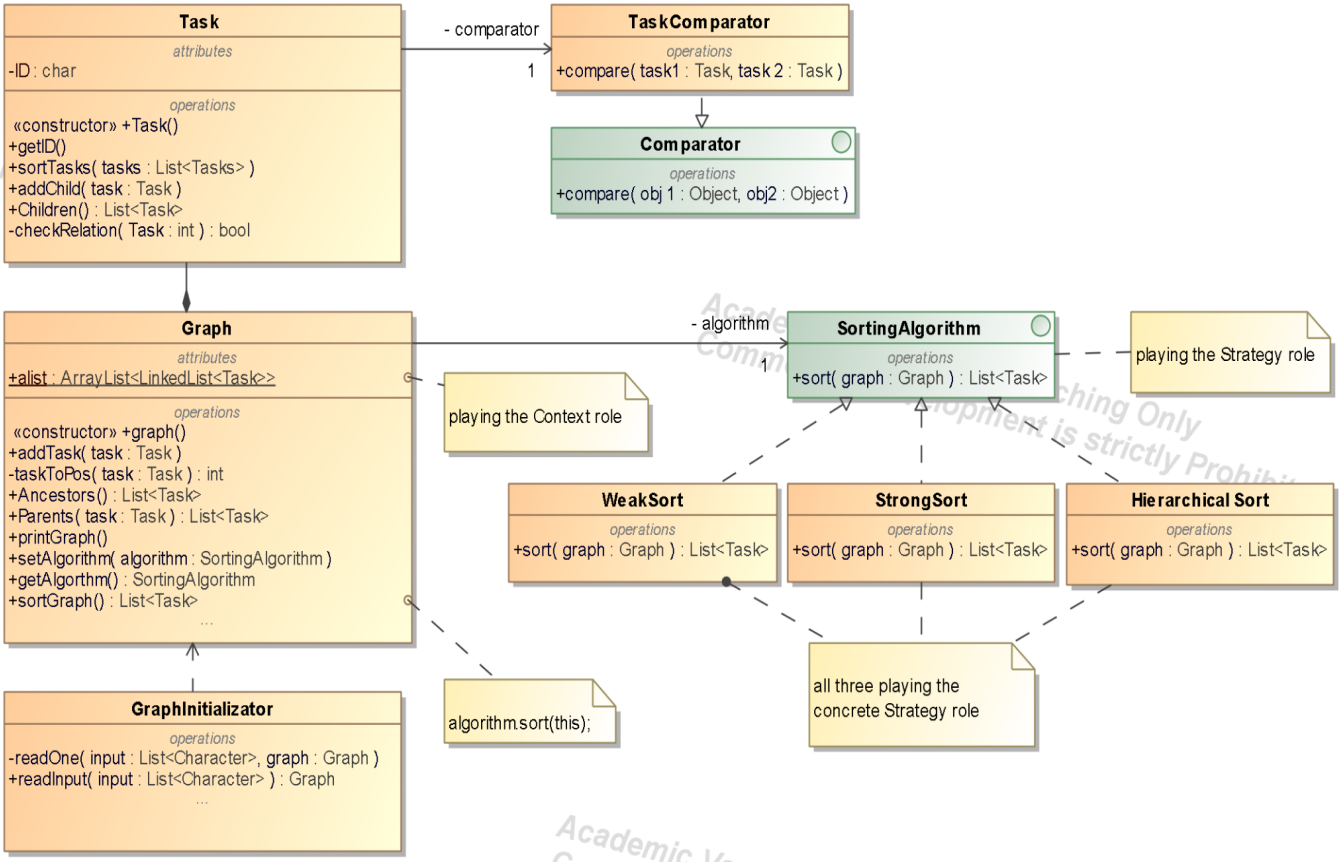


All these algorithms work as the Concrete Strategy role.

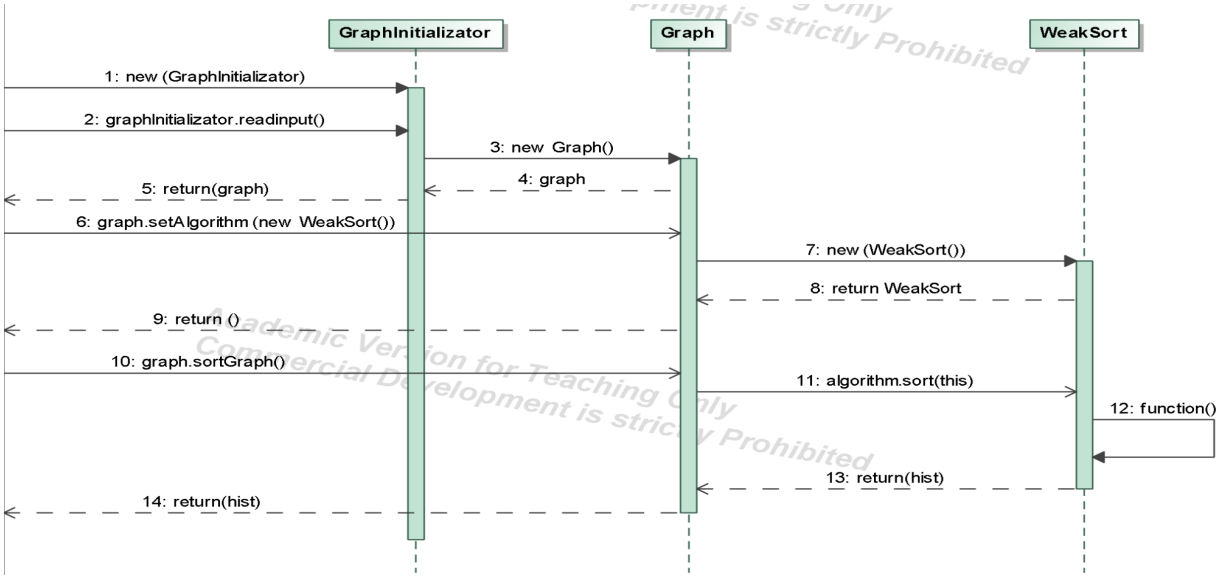
DIAGRAMS:

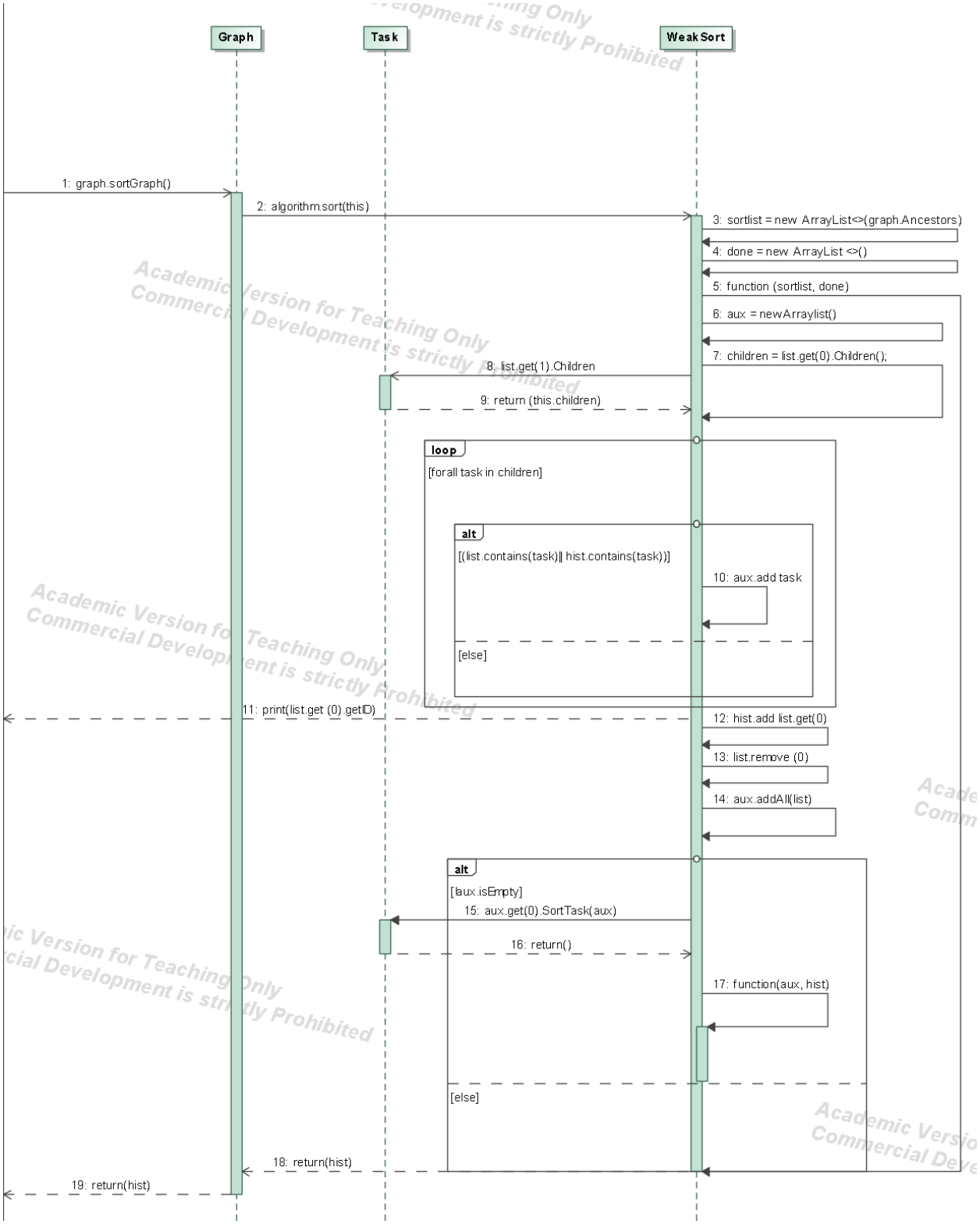
The diagrams contain the summary of this project are:

The class diagram:



Two different dynamic diagrams representing the overview of creation and sorting of the algorithm, and the detailed implementation of the WeakSort respectively:




Graph
Task
WeakSort
1: graph.sortGraph()
2: algorithm.sort(this)
3: sortlist = new ArrayList<>(graph.Ancestors)
4: done = new ArrayList <>()
5: function (sortlist, done)
6: aux = newArraylist()
7: children = list.get(0).Children();
8: list.get(1).Children
9: return (this.children)
loop
[forall task in children]
alt
[(list.contains(task)|| hist.contains(task))]
10: aux.add task
[else]
11: print(list.get (0).getID)
12: hist.add list.get(0)
13: list.remove (0)
14: aux.addAll(list)
alt
[!aux.isEmpty]
15: aux.get(0).SortTask(aux)
16: return()
17: function(aux, hist)
[else]
18: return(hist)
19: return(hist)

FIRST EXERCISE

SUMMARY OF THE GIVEN PROBLEM:

We are asked to implement a ticket management piece of software that allows users to search for a ticket by different criteria and, given the possibilities, we decide on the ticket that best fists:

- 4 main criteria: origin, destination, price and date
- they can be combined with AND clauses and OR clauses, usually in pairs
- a search may return an empty result, either because there is no ticket with those conditions or because the query was formulated incorrectly
- once obtained the list, if not empty, we can select the ticket

It is compulsory to have a flexible design since it is going to be expanded in the future.

OVERVIEW:

We started by creating the Ticket class, which stores 3 strings: origin, destination and date and an int, price.

Also, a Ticket Manager class which manages and sorts the list with all Tickets.
And implemented the conditions using the Composite pattern.

TICKET AND TICKET MANAGER:

Ticket remained fairly simple, easily modifiable for the future
TicketManager implements most of the logic behind the search() method that returns those elements in the ticket list that meet the wanted criteria.

Most of the inner workings of this class remain private in order not to be used by other classes, with the different addCriteria() (methods to add different logic to the criteria used for searching) and search() itself.

OBJECT LIST

ObjectList is an abstract class that plays the Composer role on the Composite pattern, it represents either an operator (AND / OR) or some criterium (date / origin / destination / price)

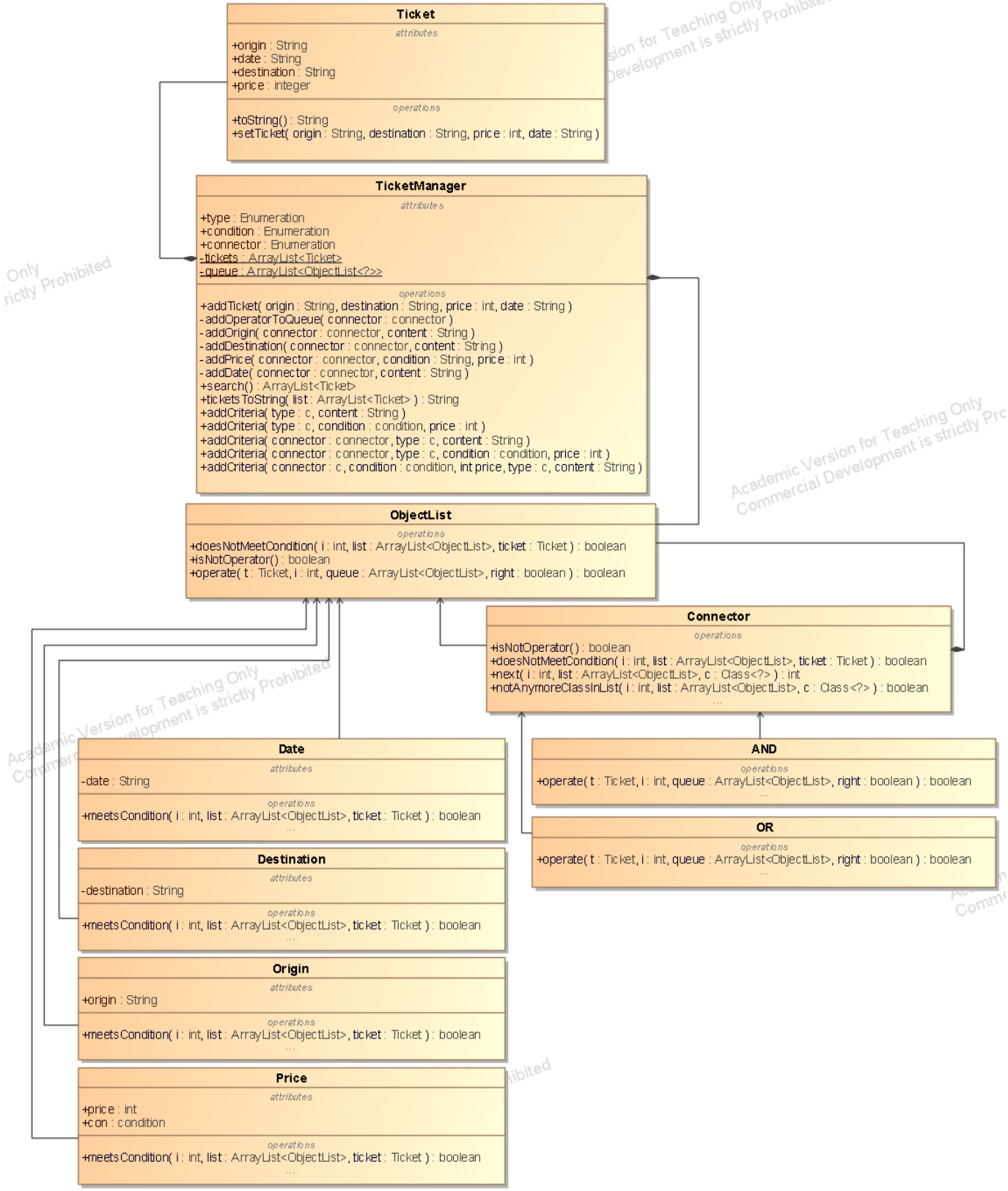
CRITERIUM

Criterium is an interface that plays the Leaf role on the Composite pattern. There are three classes that implement it : date, origin, destination and price, each of them with different attributes and a different implementation of meetsCondition() and filter()

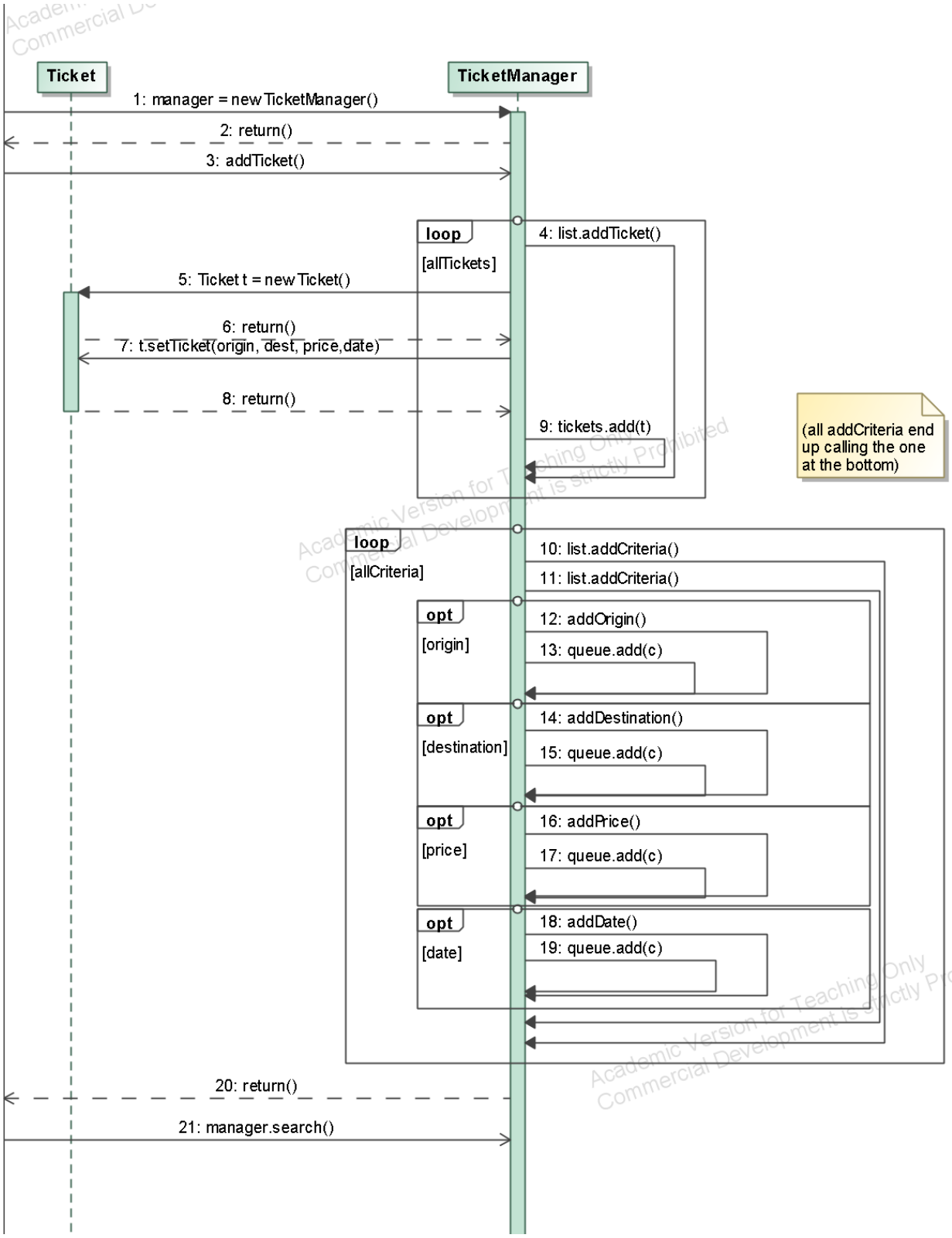
OPERATOR

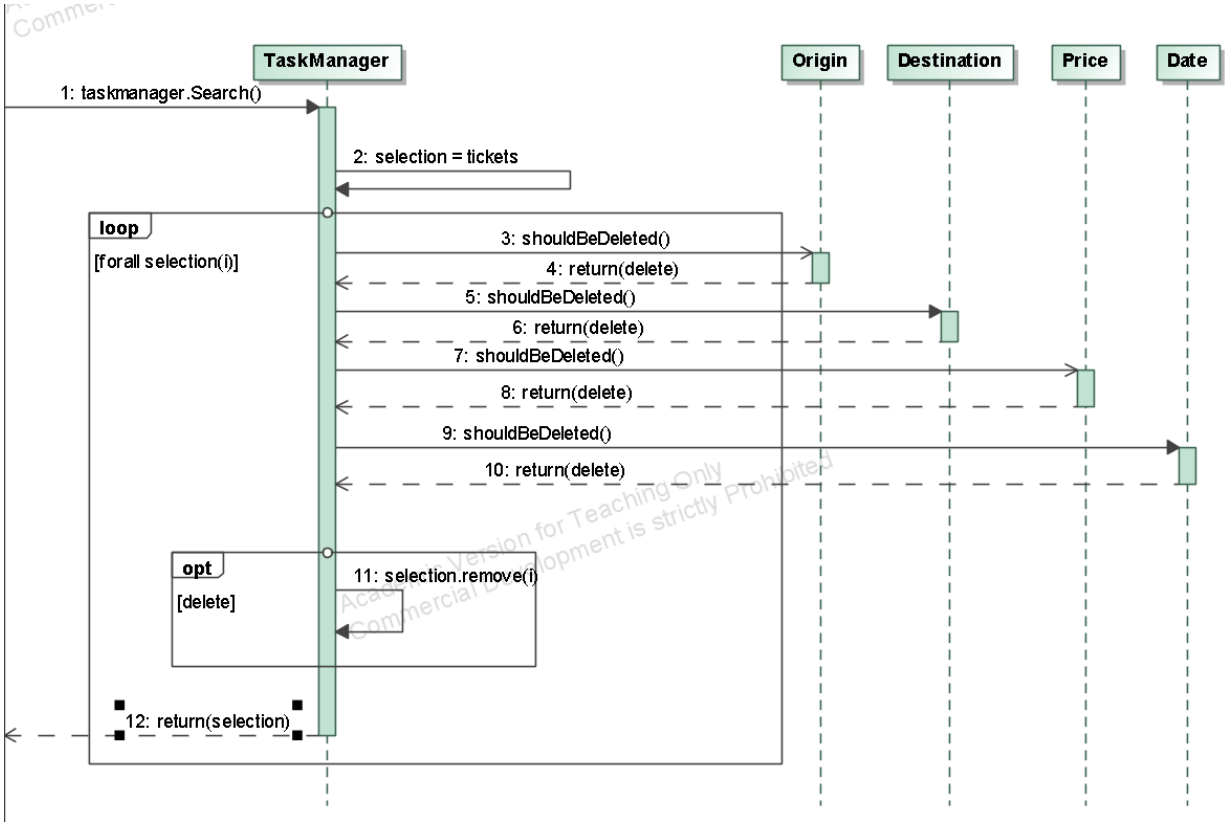
Operator is a class that inherits from ObjectList and plays the Composite role on the pattern with the same name,

We have implemented a class diagram:



As well as a dynamic diagram explaining the overall workings of setting a TaskManager , filling the List and Sorting it:




TaskManager
Origin
Destination
Price
Date
1: taskmanager.Search()
2: selection = tickets
loop
[forall selection(i)]
3: shouldBeDeleted()
4: return(delete)
5: shouldBeDeleted()
6: return(delete)
7: shouldBeDeleted()
8: return(delete)
9: shouldBeDeleted()
10: return(delete)
opt
[delete]
11: selection.remove(i)
12: return(selection)