Design Assignment (2021-22)

David García Ramallal (david.ramallal@udc.es)
Alfredo Javier Freire Bouzas (javier.freire.bouzas@udc.es)

REPORT OF EXERCISE 2 (E2):

Design Principles:

SOLID design principles are the following ones:

- *Single Responsibility Principle*
- *Open-Closed Principle*
- *Liskov Substitution Principle*
- *Dependency Inversion Principle*
- *Interface Segregation Principle*

SOLID principles used in our exercise:

- **Single Responsibility Principle (SRP)**
  "Every object should have a single responsibility entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility."

  This principle is applied in all our classes, since each of them has its own responsibility and there are not "God classes".
  Task, as the name suggests, contains the information about a specific element of the graph.
  TaskManager, which is the biggest class, is focused on the action of execute the different orders that we can use.
  Dependencies Classes (StrongDependency, WeakDependency and HierarchicalOrder) are only dedicated to the implementation of the dependency criteria algorithm.

- **Open-Closed Principle (OCP)**
  "A module (class) should be open for extension but closed for modification."

  Open-Closed principle is used in the sense that, fulfilling the pattern that we will detail later, we can create as many dependencies criteria as we want without modifying the rest of the code.

- **Liskov Substitution Principle (LSP)**
  "Subclasses should be substitutable for their base classes."

  Since we do not use subclasses (hierarchy), we can not say we fulfil this principle.

- **Dependency Inversion Principle (DIP)**
  "Depend upon abstractions. Do not depend upon concretions."

  This principle is fulfilled since the implementations of the different dependencies criteria (StrongDependency, WeakDependency and HierarchicalOrder) depend on an interface (ExecutionOrder).
  Also, we do not use concrete classes when working with lists (we declare them as List<> … = new ArrayList<>() and not as ArrayList <> … = new ArrayList<>()).

- **Interface Segregation Principle (ISP)**
  "Having many client-specific interfaces is better than having one general-purpose interface."

  This principle is not applied in this exercise because we only use one interface, since we consider that the project is simple enough not to need more.

Design Pattern:

For this exercise we have focused on working with the following pattern:

- *Strategy Pattern*

"Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from its clients."

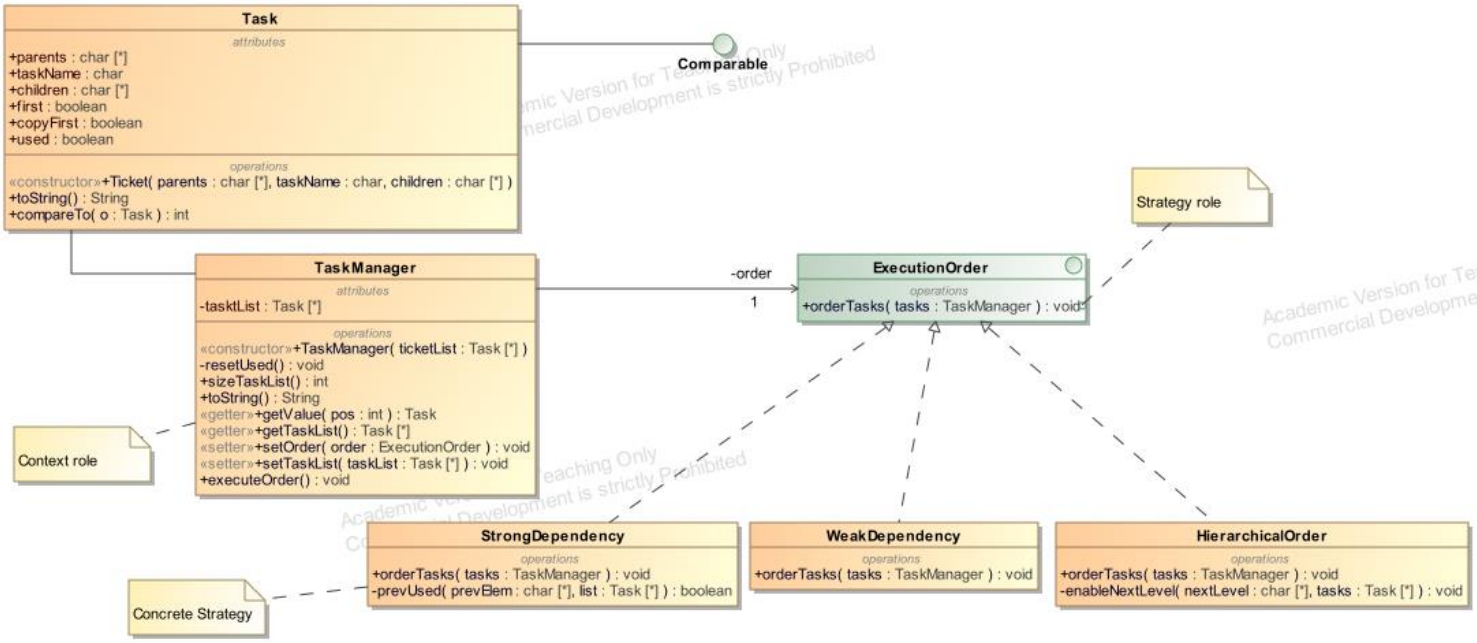
Strategy Pattern, as its name indicates, allows the same problem to be approached with different strategies. To do that, an interface is declared with one or more methods, and several classes implement them to work with the data in different ways.

There are three main elements in this pattern:

- Context: contains a concrete strategy and has a reference to the strategy object. In our code it is the class *TaskManager*
- Concrete Strategy: implements a specific algorithm. It would be one of the three classes dedicated to the application of the dependencies criterias (*StrongDependency, WeakDependency* and *HierarchicalOrder*)
- Strategy: an interface common for the different algorithms. It is *ExecutionOrder*

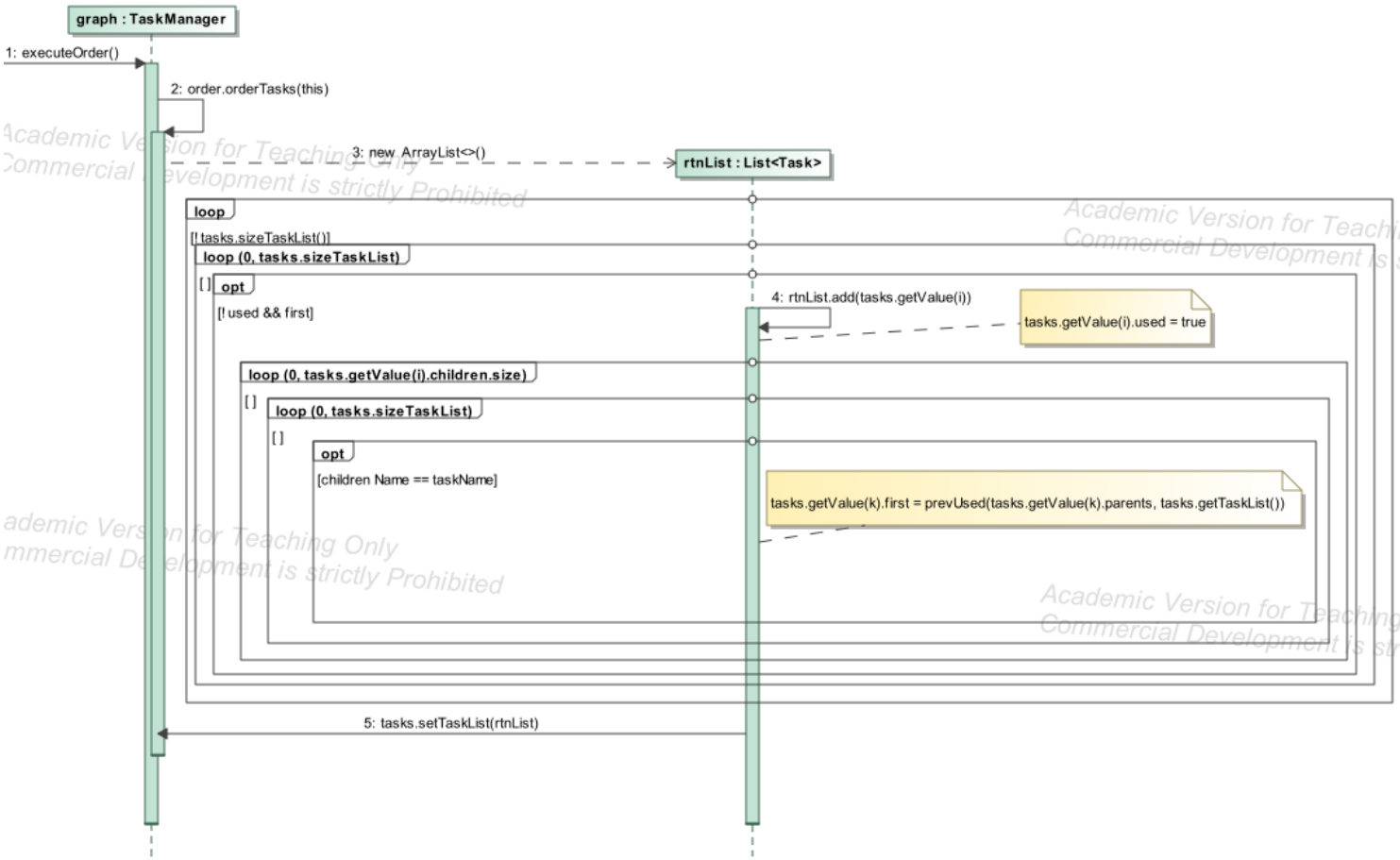
The decision of using this pattern is based on several facts. On the one hand, in the declaration of the exercise it is said that it is possible that new dependencies criteria may be added in the future, and with this pattern we can do that without touching the rest of the code. Also, Strategy Pattern is useful for the situation when we have different algorithms that we want to apply over the same data.

Class Diagram:

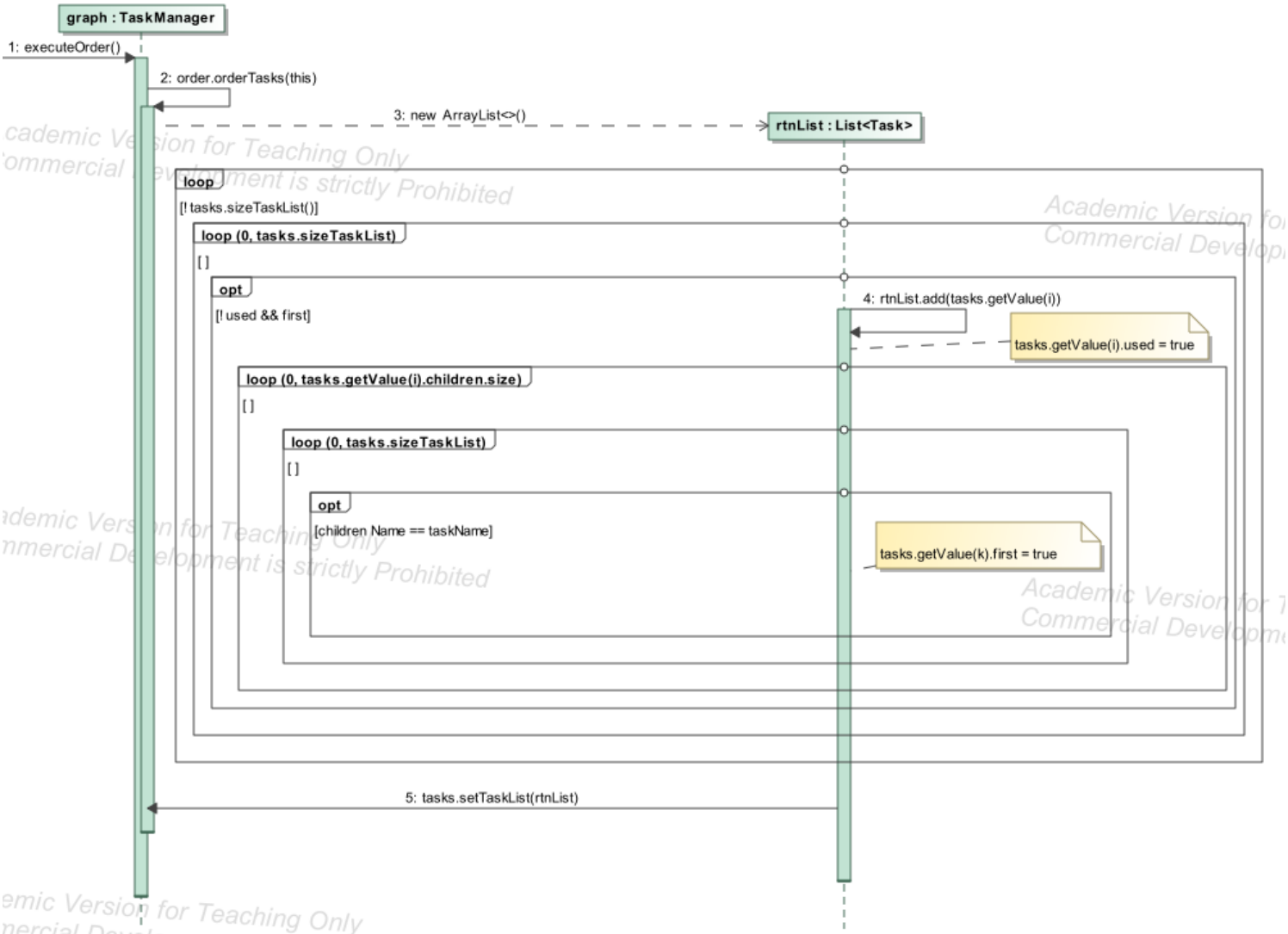


Dynamic Diagrams:

- **StrongDependency**



- **WeakDependency**



- **HierarchicalOrder**