Guillermo Fernández Sánchez
Javier Fernández Rozas

2. Task Planner. Report

Design principles:

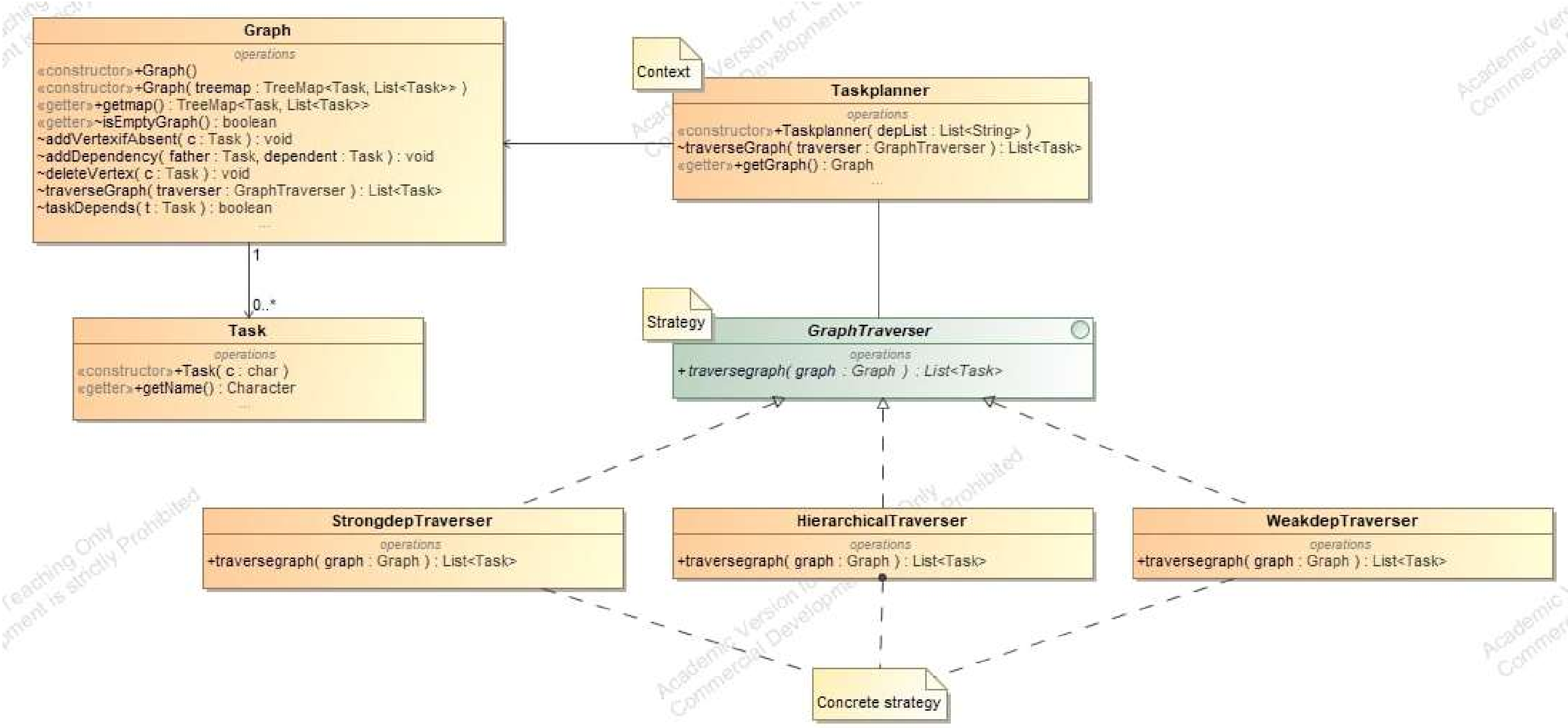
- **The Open-Closed Principle (OCP):**
  We have used the open-closed principle on the Taskplanner class and the GraphTraverser interface since we can add more graph traversals whithout knowing the source code of the base classes.
- **Favor Immutability principle:**
  We used this principle on the classes *Taskplanner*, *Graph* and *Task*. This means that all this classes are immutable since there is no reason to make them mutable (no set methods, final and private classes, all their atributes are private and there is no access to mutable components).
- **Encapsulate what varies principle:**
  The strategy pattern, which we use in this exercise is an example of the “Encapsulate what varies principle”. Polymorphism and dynamic binding is used to perform similar operations that vary.

Desing patterns:

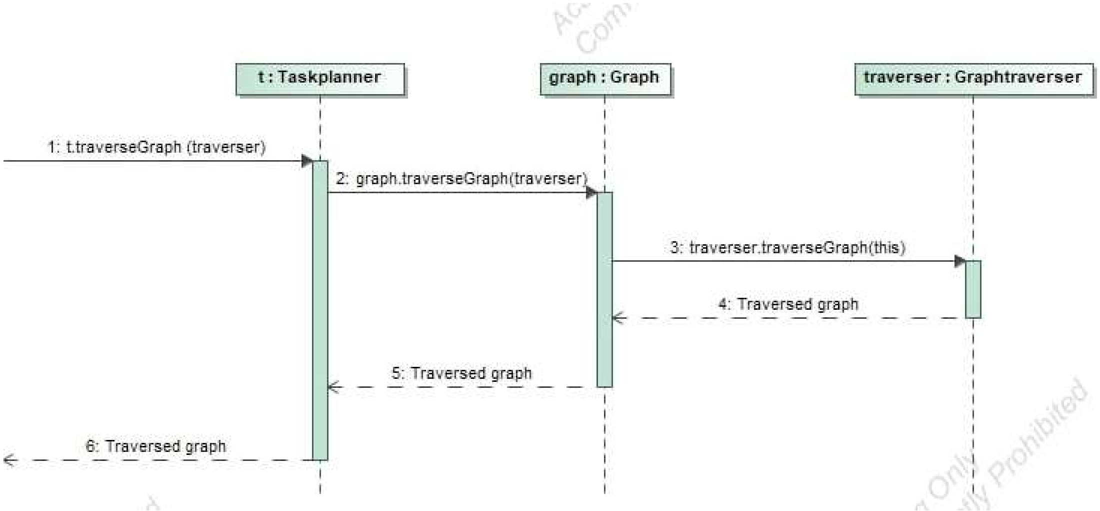
In this exercise we have used the **strategy pattern** wich defines a family of algorithms, encapsulates each one, and makes them interchangeable. Independently from its client the algorithm might change. In this case we have different algorithms or strategies to traverse a graph and we decide at each moment which one we want to use. We can add more traverse strategies at any point

The interface *GraphTraverser* plays the role of **strategy**. It is an interface common to all supported algorithms.
The class Taskplanner plays the role of **Context**.
The classes *WeakdepTraverser*, *StrongdepTraverser* and *HierarchicalTraverser* play the role of concrete **strategies**. They implement the algorithm using the GraphTraverser interface

The roles of these classes and the connections between them are shown in the following class diagram:


Graph
operations
«constructor»+Graph()
«constructor»+Graph(treemap : TreeMap<Task, List<Task>>)
«getter»+getmap() : TreeMap<Task, List<Task>>
«getter»~isEmptyGraph() : boolean
~addVertexifAbsent(c : Task) : void
~addDependency(father : Task, dependent : Task) : void
~deleteVertex(c : Task) : void
~traverseGraph(traverser : GraphTraverser) : List<Task>
~taskDepends(t : Task) : boolean
1
0..*
Task
operations
«constructor»+Task(c : char)
«getter»+getName() : Character
Context
Taskplanner
operations
«constructor»+Taskplanner(depList : List<String>)
~traverseGraph(traverser : GraphTraverser) : List<Task>
«getter»+getGraph() : Graph
Strategy
GraphTraverser
operations
+traversegraph(graph : Graph) : List<Task>
StrongdepTraverser
operations
+traversegraph(graph : Graph) : List<Task>
HierarchicalTraverser
operations
+traversegraph(graph : Graph) : List<Task>
WeakdepTraverser
operations
+traversegraph(graph : Graph) : List<Task>
Concrete strategy

The following sequence diagram shows a call to *Taskplanner's traverseGraph* function, taking a Graphtraverser object as a parameter.


t : Taskplanner
graph : Graph
traverser : Graphtraverser
1: t.traverseGraph (traverser)
2: graph.traverseGraph(traverser)
3: traverser.traverseGraph(this)
4: Traversed graph
5: Traversed graph
6: Traversed graph

Due to polimorfism and dynamic binding, depending of the concrete strategy *traverser* is an instance of, the graph is traversed on a different way.