Duc Long Hoang

Software Design – Design Assignment
Management of work groups

Design principles

In this exercise I used the SOLID principles although not all of them.

Single responsibility principle

- I think I did not use this principle because my code is very intertwined, more than I would like, to be honest.

Open-Closed principle

- This is achieved in the classes ProjectElement and ProjectSegment. After inheriting from these classes, most of the methods are implemented and the rest is abstract for the user to finish. Or if he so chooses, the user can override the methods to behave in a slightly different way.

Liskov Substitution principle

- This principle is respected because in places where we expect a parameter of type ProjectElement we can without any problems use all its subclasses like ProjectSegment, Team, Worker.
- A clear example would be the addElement() method. Where we can insert every subclass instead of a ProjectElement. But this method should be overridden in its subclasses with some conditions of adding an element. For e.g., it should not be possible to add an object of class Project under a Team.

Interface Segregation principle

- I did not use this principle because I do not have any interfaces but instead I worked with 2 abstract classes.

Dependency Inversion principle

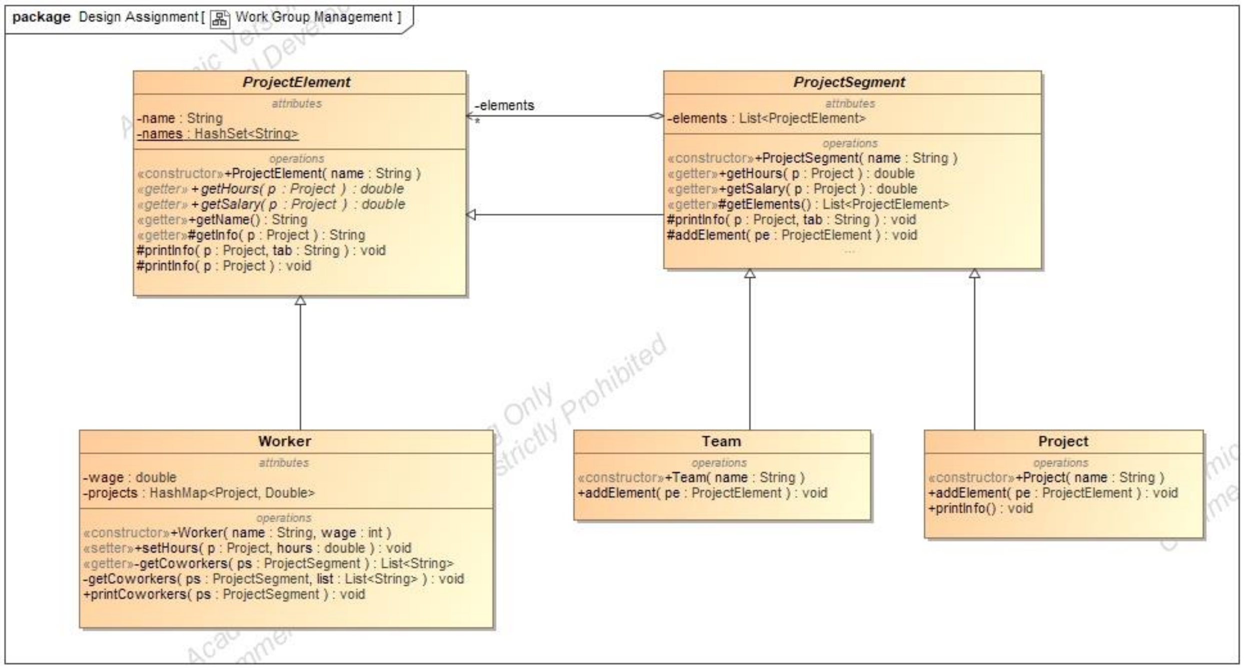
- This principle is achieved for example in class Worker. Where we set the wage per hour in the constructor while creating an instance or by the method setHours() where we can set how much time the Worker has spent on a certain Project

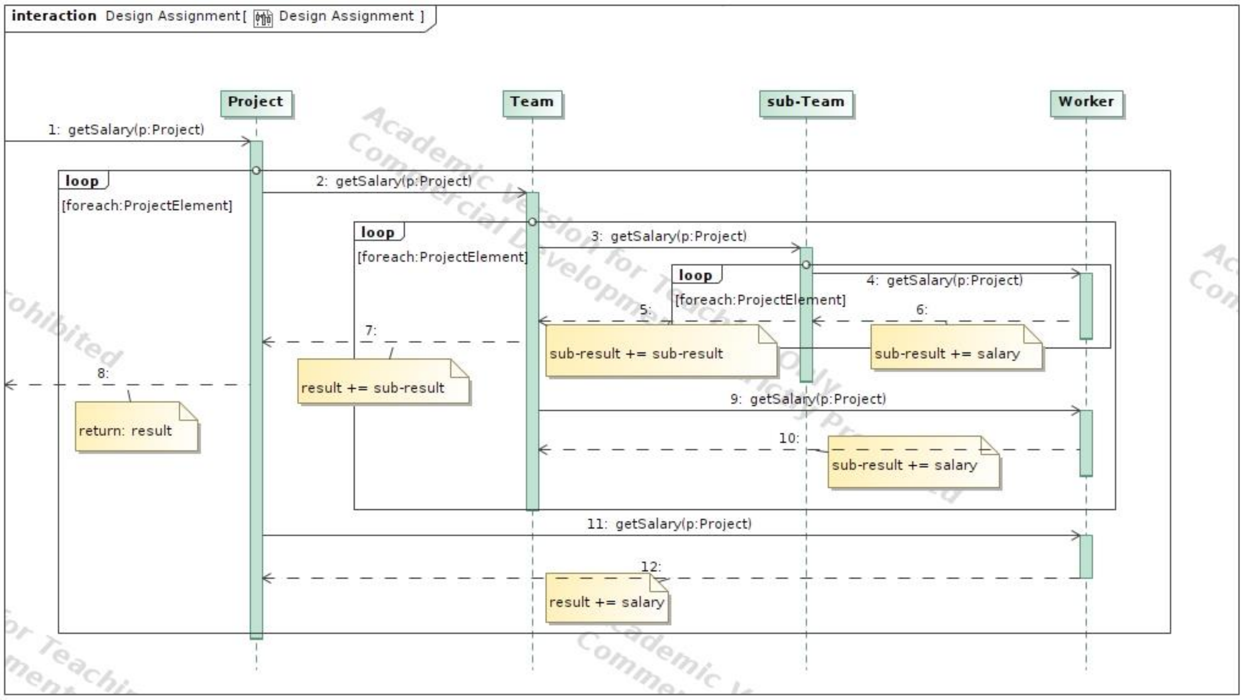
Design patterns

For this exercise I used the **Composite pattern** as it was the most obvious one to use. We have a Worker which is an element in a Project and Teams that are working on a Project and are aggregates of Workers and/or other sub-Teams. The same way in the U6 presentation we have a FileSystem (=Project), Directory (=Team) and File (=Worker).

This led me to use an abstract class called ProjectElement that defines the basic properties of every element (Worker/Team/Project). To make it more complicated I created another abstract class called ProjectSegment that extends the ProjectElement class. This class is basically a composite of ProjectElements and as such will be inherited by classes: Team and Project. While class Worker extends only the ProjectElement class. The reason I did this, is to not repeat code that would appear in the Team and Project classes for getting the sum of hours and the total money.

In this way, if we want to add later another class for example Division or Department which could be a bigger aggregate of Teams, we just need to inherit from the ProjectSegment class and override 1-2 methods. If we want to add a different type of employee for example a Trainee or Part-time worker, we extend from the ProjectElement abstract class.


package Design Assignment[Work Group Management]
ProjectElement
attributes
-name : String
-names : HashSet<String>
operations
«constructor»+ProjectElement(name : String)
«getter» +getHours(p : Project) : double
«getter» +getSalary(p : Project) : double
«getter»+getName() : String
«getter»#getInfo(p : Project) : String
#printInfo(p : Project, tab : String) : void
#printInfo(p : Project) : void
-elements
*
ProjectSegment
attributes
-elements : List<ProjectElement>
operations
«constructor»+ProjectSegment(name : String)
«getter»+getHours(p : Project) : double
«getter»+getSalary(p : Project) : double
«getter»#getElements() : List<ProjectElement>
#printInfo(p : Project, tab : String) : void
#addElement(pe : ProjectElement) : void
...
Worker
attributes
-wage : double
-projects : HashMap<Project, Double>
operations
«constructor»+Worker(name : String, wage : int)
«setter»+setHours(p : Project, hours : double) : void
«getter»-getCoworkers(ps : ProjectSegment) : List<String>
-getCoworkers(ps : ProjectSegment, list : List<String>) : void
+printCoworkers(ps : ProjectSegment) : void
Team
operations
«constructor»+Team(name : String)
+addElement(pe : ProjectElement) : void
Project
operations
«constructor»+Project(name : String)
+addElement(pe : ProjectElement) : void
+printInfo() : void


interaction Design Assignment[Design Assignment]
Project
Team
sub-Team
Worker
1: getSalary(p:Project)
loop
[foreach:ProjectElement]
2: getSalary(p:Project)
loop
[foreach:ProjectElement]
3: getSalary(p:Project)
loop
[foreach:ProjectElement]
4: getSalary(p:Project)
5:
6:
sub-result += sub-result
sub-result += salary
7:
result += sub-result
8:
return: result
9: getSalary(p:Project)
10:
sub-result += salary
11: getSalary(p:Project)
12:
result += salary