Report DS-61-02-DA

AUTHOR 1: Alejandro Esteban Martinez LOGIN 1: a.estebanm DNI 1: 49683259Q
AUTHOR 2: Samuel Ramos Varela LOGIN 2: s.ramosv DNI 2: 54151857J
GROUP: 6.1

Exercise 1: Thermostat

The design principles used in this exercise were:

- Single Responsibility principle: It is used in the class Thermostat and ThermostatInfo,
By creating the ThermostatInfo class we avoid giving too much responsibility to the class Thermostat, which is responsible of the control of the temperature and of commanding to change state.
With ThermostatInfo we create the log of the Thermostat, making it more loosly coupled.

- Open-Closed principle: It is used in states of the Thermostat, because we can add new states by implementing the ThemostatState interface and without having to modify the rest of the code
We avoid using characteristics of the implementation of the different concrete states, making it able to use any state we give it, so we could expand the number of states by implementing new ones.

- Dependency inversion principle: It is used in class Thermostat and ThermostatInfo with the state classes.
We avoid in both classes using elements of the exact implementation of the other class and of the different states, making it able to add new states and extending the different classes without having to modify the already existing code.
We “invert” the dependency by using method declared in the ThermostatState interface instead of having different methods in each state, so the dependency would be with the ThermostatState interface instead of the individual concrete states.

-Encapsulate what varies principle: It is used in the class Thermostat with the use of the states, which can change when setState is called, this way we can modify the states in the future without having to modify the code.

-Loose coupling principle: We use this principle as a product of the previous principles, because by having each class having a single responsibility and using abstract terms instead of depending in the implementation of each other class we get a loosely coupled program, which is more cohesive and clear.

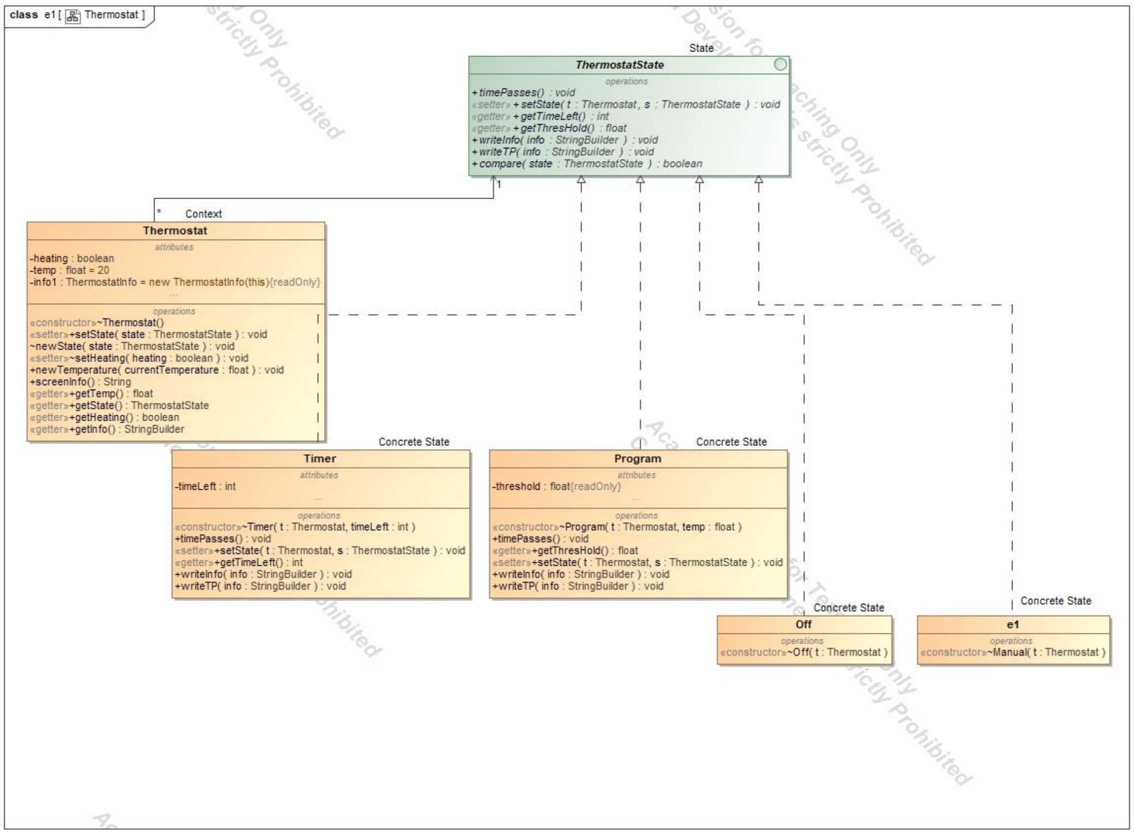
-Tell, don’t ask principle: It is used when having to change state, because depending on the internal current state of the thermostat it will change one way or another, so instead of doing the logic asking about it’s state we tell the state to change and depending of it’s internal state it does something or another thing, like when changing from timer to program, it has to change to Off/Manual, and because of the implementation of this principle, it is done automatically.

- DRY, KISS and YAGNI: we try to use this principles in all the code as a good programming habit, avoiding repeating unnecessary code, keeping it as simple as possible and not doing unnecessary methods or classes to overcomplicate the code

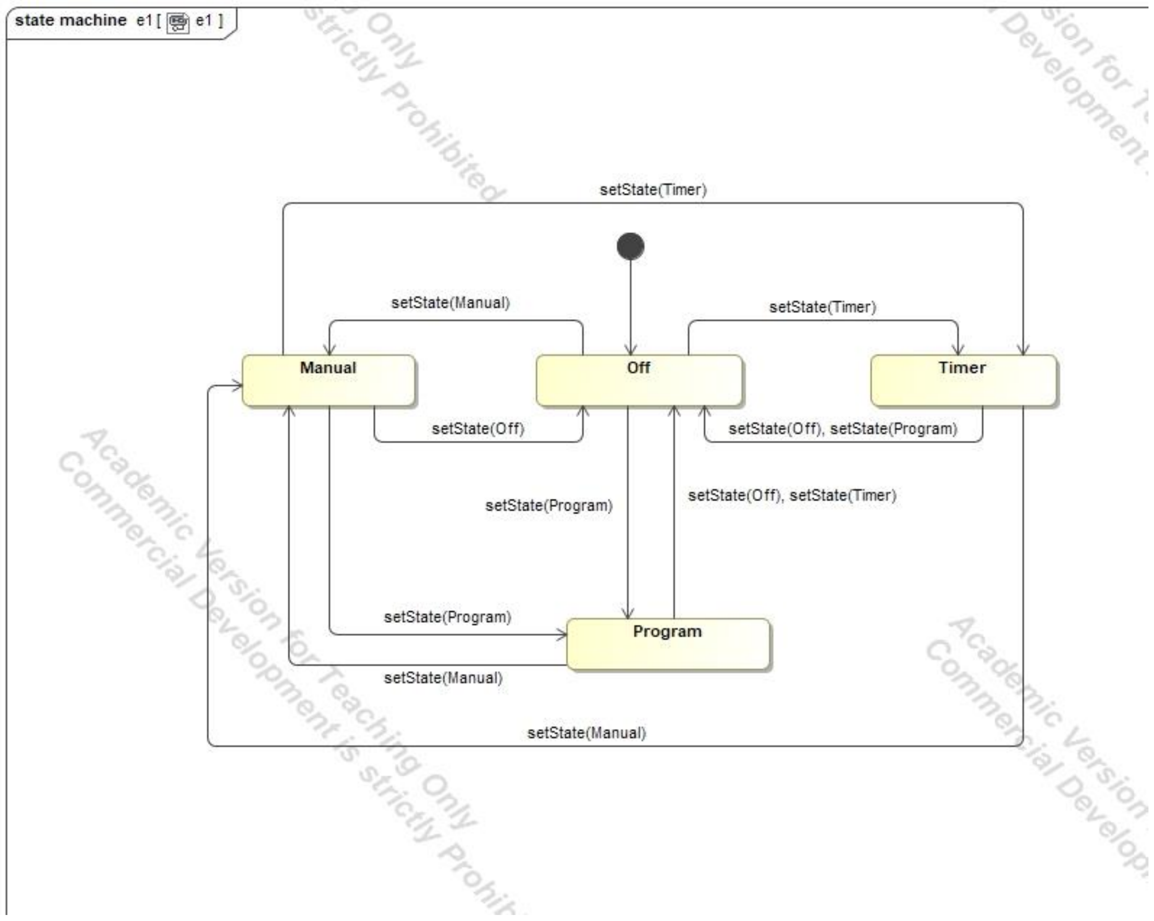
The design patterns used were:

-State Pattern: This pattern is choosen to control the different modes of the thermostat, in this case Off, Manual, Timer and Program, but there could be more because adding is possible without having to change the implementation.
It is used to change the behavior of the thermostat when it’s internal state (mode) is changed. This way mode change transitions are explicit.

Class diagram:


class e1 [Thermostat]
State
ThermostatState
operations
+timePasses() : void
«setter» +setState(t : Thermostat , s : ThermostatState) : void
«getter» +getTimeLeft() : int
«getter» +getThresHold() : float
+writeInfo(info : StringBuilder) : void
+writeTP(info : StringBuilder) : void
+compare(state : ThermostatState) : boolean
1
*
Context
Thermostat
attributes
-heating : boolean
-temp : float = 20
-info1 : ThermostatInfo = new ThermostatInfo(this){readOnly}
operations
«constructor»~Thermostat()
«setter»+setState(state : ThermostatState) : void
~newState(state : ThermostatState) : void
«setter»~setHeating(heating : boolean) : void
+newTemperature(currentTemperature : float) : void
+screenInfo() : String
«getter»+getTemp() : float
«getter»+getState() : ThermostatState
«getter»+getHeating() : boolean
«getter»+getInfo() : StringBuilder
Concrete State
Timer
attributes
-timeLeft : int
operations
«constructor»~Timer(t : Thermostat, timeLeft : int)
+timePasses() : void
«setter»+setState(t : Thermostat, s : ThermostatState) : void
«getter»+getTimeLeft() : int
+writeInfo(info : StringBuilder) : void
+writeTP(info : StringBuilder) : void
Concrete State
Program
attributes
-threshold : float{readOnly}
operations
«constructor»~Program(t : Thermostat, temp : float)
+timePasses() : void
«getter»+getThresHold() : float
«setter»+setState(t : Thermostat, s : ThermostatState) : void
+writeInfo(info : StringBuilder) : void
+writeTP(info : StringBuilder) : void
Concrete State
Off
operations
«constructor»~Off(t : Thermostat)
Concrete State
e1
operations
«constructor»~Manual(t : Thermostat)

Dynamic Diagram: The one that better represented the use of this exercise was the State Diagram



Exercise 2: Work Groups

The design principles used in this exercise were:

-Open Closed Principle: We can easily add a new type of object extending ProjectElem without having to change any implementation, due to the fact that time(), cost() and str() are abstract methods. This allow us to have different ways of computing the cost and time, depending on the class an object is. In team.time() we do a recursive computation, taking into account its elements costs, meanwhile worker.time() just returns that workers time. (NO ES CREO)

-Dependency Inversion Principle: In the iterations through the ProjectElem list in Team.time(), we are able to do enum.time() independently of the type of object enum is at the moment. If we were to add another type of ProjectElement, we wouldn't have to change any bit of the implementation. This also happens with .str(), which we make use of in Team.teamStr(), and of course on cost(), similarly to time().

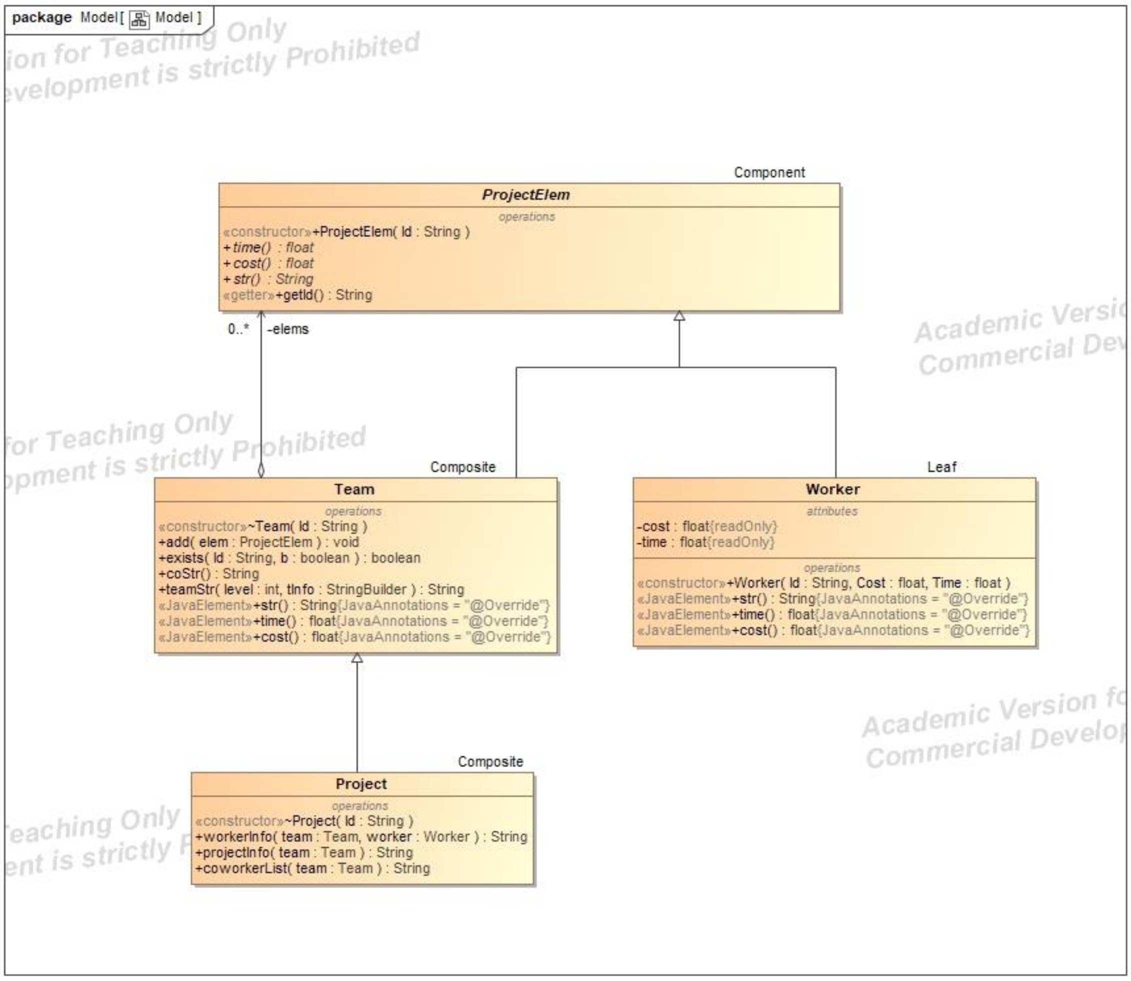
-Principle of Least Knowledge: All methods invoked on any Team and Project method are from an object that: has been passed as an argument or is an element of a collection which is an attribute of the enclosing object. For example in workerInfo a worker is passed as a parameter so worker.str() does not break this principle. Another example is the time() of the Team class, in which we can do enum.time() even though enum might be a worker, because it is part of a collection in the Team class (enums List).

-Tell Don't Ask: In Project, the functions tell the Team class to iterate and do the operations over the elems List. As the list is already on Team, rather than asking for the list and operating over it, Project limits itself to receiving the output that Team generates.

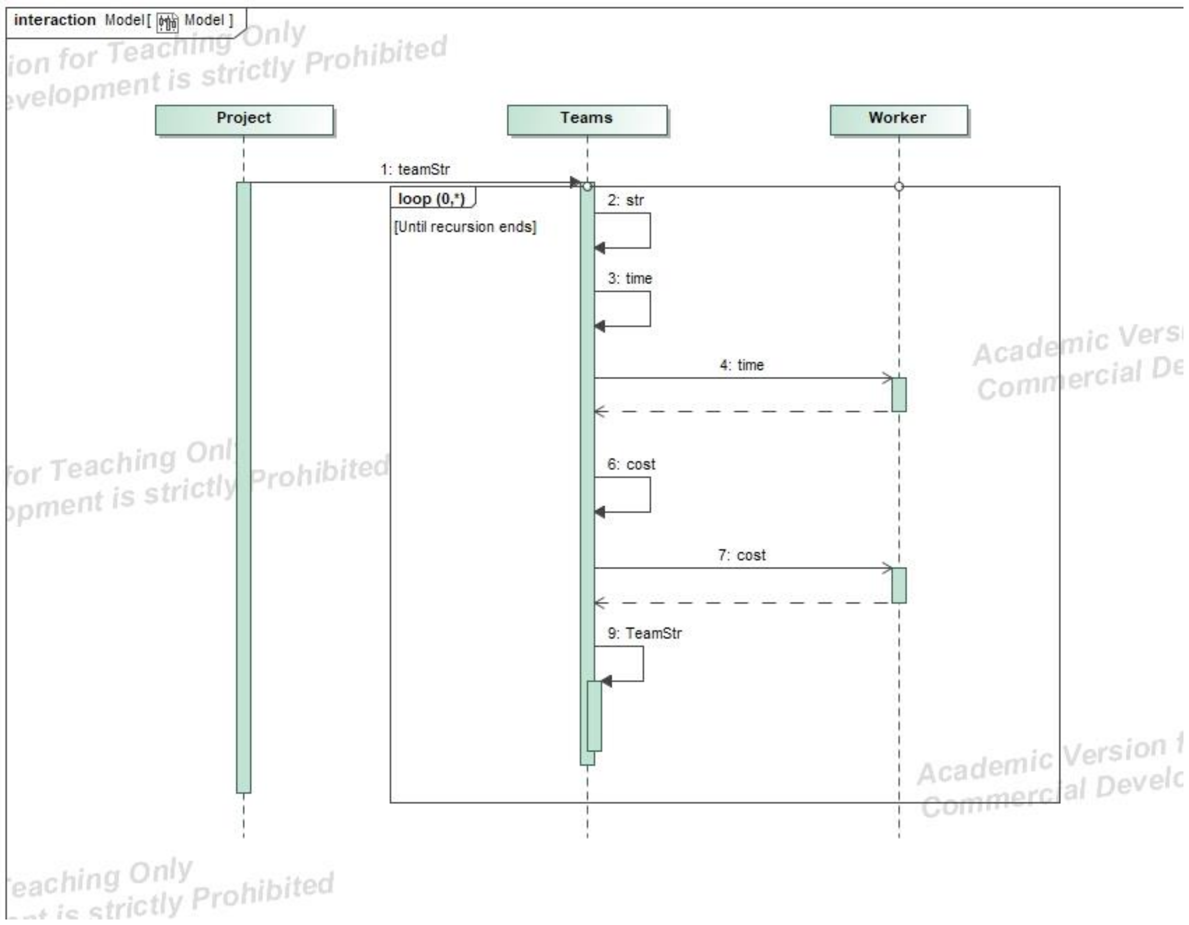
The design patterns used were:

-Composite Pattern: Composite pattern allows us to build structures with complex and simple objects. Every object here was supposed to have cost and time, and we needed that some objects could contain others (teams contain workers), but leave the behavior flexible, maintaining the cost and time in each object, so we decided that the best option would be to use a Composite Pattern: A class ProjectElem with two abstract methods, cost() and time() with two classes that extend it (Team and Worker), in which the cost and time of a team is calculated adding its childrens. Along the way we realized that the Project was indeed a Team, so we made that the Project class extends directly from Team. We also added another abstract method called str(), as it was quite convenient for most of the tasks we had to do (info methods).

Class Diagram:


package Model[Model]
Component
ProjectElem
operations
«constructor»+ProjectElem(Id : String)
+time() : float
+cost() : float
+str() : String
«getter»+getId() : String
0..*
-elems
Composite
Team
operations
«constructor»~Team(Id : String)
+add(elem : ProjectElem) : void
+exists(Id : String, b : boolean) : boolean
+coStr() : String
+teamStr(level : int, tInfo : StringBuilder) : String
«JavaElement»+str() : String{JavaAnnotations = "@Override"}
«JavaElement»+time() : float{JavaAnnotations = "@Override"}
«JavaElement»+cost() : float{JavaAnnotations = "@Override"}
Leaf
Worker
attributes
-cost : float{readOnly}
-time : float{readOnly}
operations
«constructor»+Worker(Id : String, Cost : float, Time : float)
«JavaElement»+str() : String{JavaAnnotations = "@Override"}
«JavaElement»+time() : float{JavaAnnotations = "@Override"}
«JavaElement»+cost() : float{JavaAnnotations = "@Override"}
Composite
Project
operations
«constructor»~Project(Id : String)
+workerInfo(team : Team, worker : Worker) : String
+projectInfo(team : Team) : String
+coworkerList(team : Team) : String

Dynamic diagrams: The one that better represented the use of this exercise was the State Diagram



interaction Model [Model]
Project
Teams
Worker
1: teamStr
loop (0,*)
[Until recursion ends]
2: str
3: time
4: time
6: cost
7: cost
9: TeamStr