TEMPERATURE CONTROL REPORT

Cristian Ferreiro Montoiro – cristian.ferreiro

Xoel González Pereira – xoel.gonzalezp

Introduction

The code solution for this exercise allows to design a thermostat, its modes of functioning and the transitions between them, in order to control the heating in an apartment.

It is composed of four modes:

- Off(Heating is turned off)
- Manual(Heating is turned on)
- Timer(Heating is turned on for a specified time and after that goes back to Off mode)
- Program(Program threshold is established, if it is below the threshold heating is turned on, otherwise heating is off).

The thermostat can switch between its modes but it cannot do it directly from Timer to Program or vice versa, it needs to change to Off in the middle.

Numerous classes have been made (**Thermostat**, **Manual**, **Program**, **Off and Timer**), each one with all the methods necessary to allow the functioning of thermostat's modes and the transitions between them.

Design principles

We considered several SOLID principles to code this solution:

- Open Closed Principle (OCP)

The modules can be extended without requiring to be modified. We can change what modules do, without changing the already existing source code.

Code example

If we want to add a new state for the thermostat, we can create a new class **ExampleState** that implements the **State** interface, overriding the methods `update()`, `switchState()` and `info()` by a concrete implementation specific of that state.

It is not necessary to modify the other existing classes that represent the states.

- Dependency Inversion Principle (DIP)

Each class (**Manual**, **Off, Program** and **Timer**) depends upon the `State` interface, not concrete classes.

Code example

The classes use states to assign them to the thermostat. To see how they are implemented, we use the class **Thermostat** as an example.

The `state` is defined as following

```
private State state;
```

and then instantiated in the class constructor with

```
this.state = Off.getInstance();
```

In the first line we declare that we want a state, and in the second line we assign it. It would be easy to switch to another state:

```
this.state = new Timer();
```

Moreover, classes use dependency injection.

Code examples

The `switchState()` and `update()` methods receive a **Thermostat** object as a parameter, which is necessary for that methods to work.

```
public void switchState(Thermostat thermostat){...}
public void update(Thermostat thermostat){...}
```

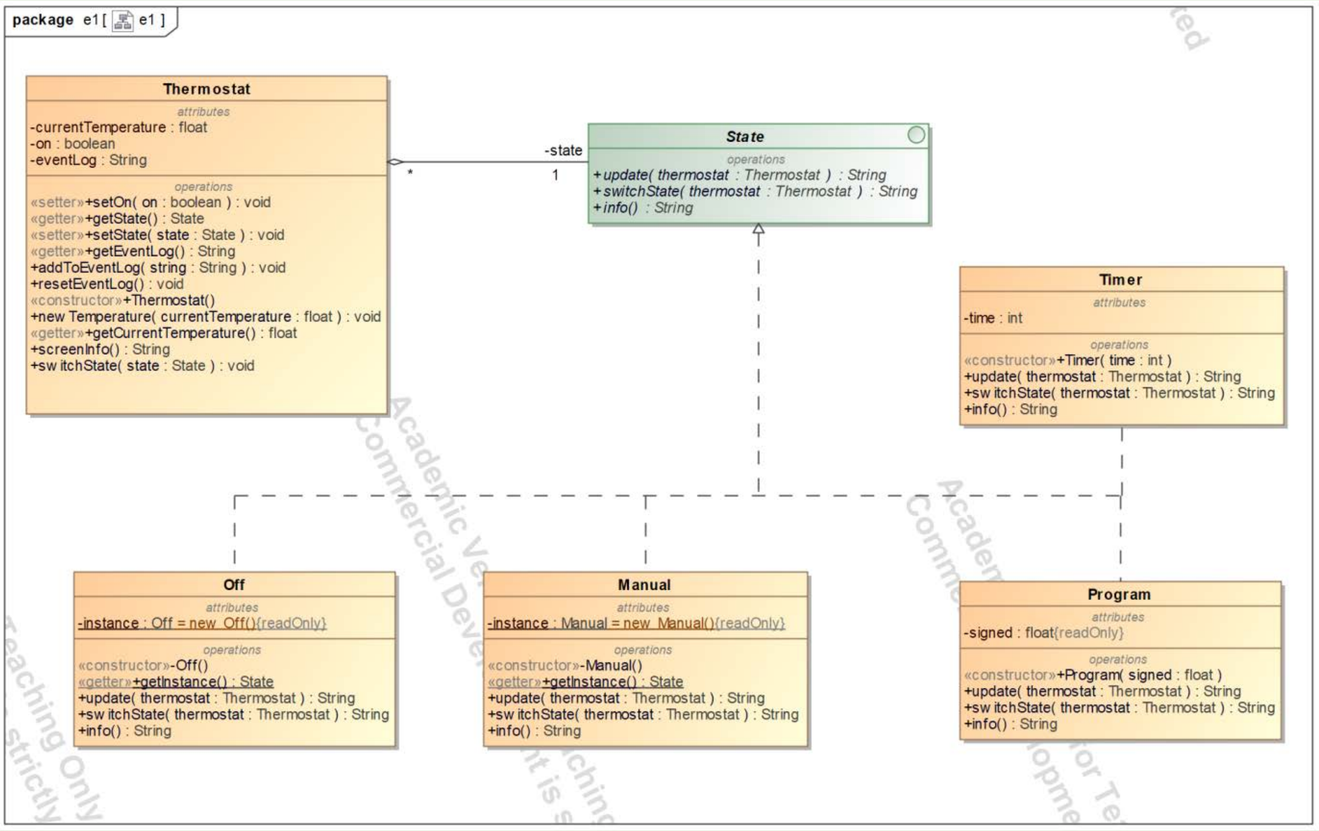
Design patterns

We used the following design patterns:

- State Pattern

This pattern fits the structure of the states of the thermostat: it can change from Off to Manual, or from Off to Timer, etc. By creating an interface (**State**) we can manipulate the states uniformly and alter the thermostat functioning easily.

Class diagram


package e1[e1]
Thermostat
attributes
-currentTemperature : float
-on : boolean
-eventLog : String
operations
«setter»+setOn(on : boolean) : void
«getter»+getState() : State
«setter»+setState(state : State) : void
«getter»+getEventLog() : String
+addToEventLog(string : String) : void
+resetEventLog() : void
«constructor»+Thermostat()
+new Temperature(currentTemperature : float) : void
«getter»+getCurrentTemperature() : float
+screenInfo() : String
+sw itchState(state : State) : void
*
-state
1
State
operations
+update(thermostat : Thermostat) : String
+switchState(thermostat : Thermostat) : String
+info() : String
Timer
attributes
-time : int
operations
«constructor»+Timer(time : int)
+update(thermostat : Thermostat) : String
+sw itchState(thermostat : Thermostat) : String
+info() : String
Off
attributes
-instance : Off = new Off(){readOnly}
operations
«constructor»-Off()
«getter»+getInstance() : State
+update(thermostat : Thermostat) : String
+sw itchState(thermostat : Thermostat) : String
+info() : String
Manual
attributes
-instance : Manual = new Manual(){readOnly}
operations
«constructor»-Manual()
«getter»+getInstance() : State
+update(thermostat : Thermostat) : String
+sw itchState(thermostat : Thermostat) : String
+info() : String
Program
attributes
-signed : float{readOnly}
operations
«constructor»+Program(signed : float)
+update(thermostat : Thermostat) : String
+sw itchState(thermostat : Thermostat) : String
+info() : String

- Singleton Pattern

This pattern ensures that a class has only one instance. If fits the **Off** and **Manual** classes, which need to be instantiated only once and can be used in several thermostats at the same time. The **Timer** and **Program** classes can have more than one instance, since we can have thermostats with different timers or threshold temperatures.

Code example

In the **Off** class the instance is created when the class is loaded for the first time:

```
private static final Off instance = new Off();
```

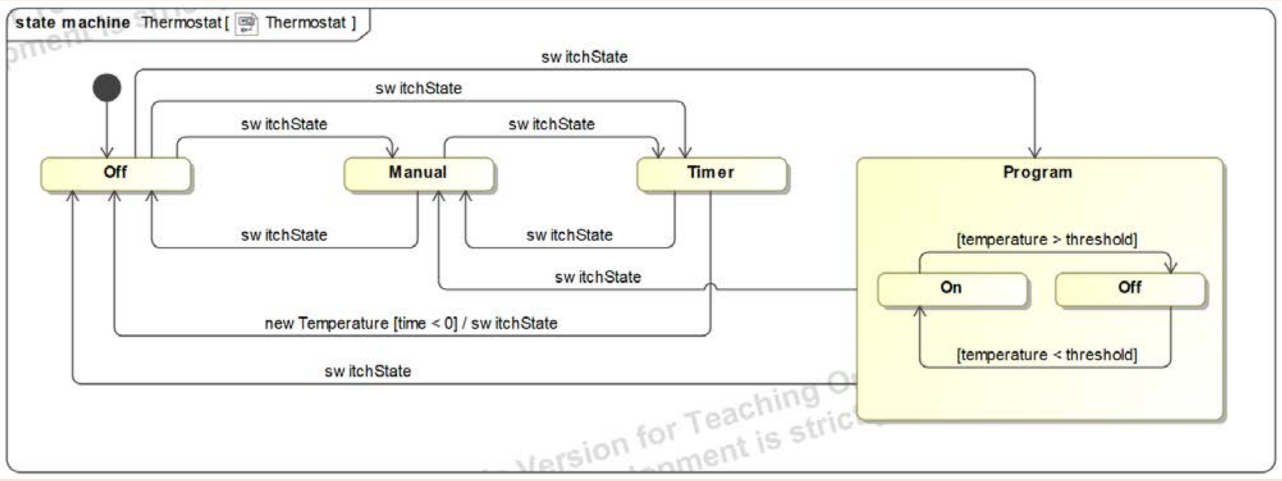
We use a static getter method that allows access to the unique instance:

```
public static State getInstance() {
        return instance;
    }
```

The constructor of **Off** is defined `private` to avoid the creation of more instances:

```
private Off(){ }
```

State machine diagram


state machine Thermostat[Thermostat]
sw itchState
sw itchState
sw itchState
sw itchState
Off
Manual
Timer
Program
sw itchState
sw itchState
[temperature > threshold]
sw itchState
On
Off
new Temperature [time < 0] / sw itchState
[temperature < threshold]
sw itchState

MANAGEMENT OF WORK GROUPS REPORT

Cristian Ferreiro Montoiro – cristian.ferreiro

Xoel González Pereira – xoel.gonzalezp

Introduction

The code solution designed for this exercise allows to create multiple projects, which include a list of associated work teams. Each team will be defined by the workers in it and the subteams associated. In addition, a worker will have associated a cost per hour and the number of hours that they worked.

The user is responsible to make sure that a worker is not added to several teams in the same project, and the teams do not have more than one level of subteams (that is, a team cannot have a subteam inside a subteam, for example).

Four classes have been made (**Project**, **Team**, **TeamElement** and **Worker**), each one with all the methods necessary to manage the different projects and work teams.

Design principles

We considered several SOLID principles to code this solution:

- Single Responsibility Principle (SRP)

Each class (**Project**, **Team** and **Worker**) encapsulates a unique responsibility and provides the services related to it. In that way, a change in one class will limit the propagation to other classes, making the code more robust.

Code example

The class **Project** provides methods to manage the list of teams associated (add, remove, get teams, etc.), but if we want to access specific information about a team or a worker, we have to use the methods from those classes.

If we want to print the cost per hour of *daveGrohl*, which belongs to the team *bandMembers* in the project *fooFighters*, we will use

```
System.out.println(fooFighters.getTeam("bandMembers").
getWorker("daveGrohl").getCostPerHour());
```

where the method `getTeam()` belongs to **Project**, `getWorker()` to **Team** and `getCostPerHour()` to **Worker**.

- Dependency Inversion Principle (DIP)

Each class (**Project**, **Team** and **Worker**) depends upon the `List` interface, not concrete classes.

Code example

The classes **Project** and **Team** use lists to store information about teams or workers. To see how they are implemented, we use the class **Team** as the example. The `elementsList` is defined as following

```
private List<TeamElement> elementsList;
```

and then instantiated in the class constructor with

```
this.elementsList = new ArrayList<>();
```

In the first line we declare that we want a list, and in the second line the type of Java list to use. It would be easy to switch to another kind of list:

```
this.elementsList = new LinkedList<>();
```

Moreover, the Project and Team classes use dependency injection.

Code examples

The `addTeam()`, `addWorker()` and `addSubteam()` methods receive a **Team** or **Worker** object as a parameter, which is necessary for that methods to work

```
public void addTeam(Team team){...}
public void addWorker(Worker worker){...}
public void addSubteam(Team team){...}
```

A lot of the methods which start by *get* also receive a String object in order to return a specific element

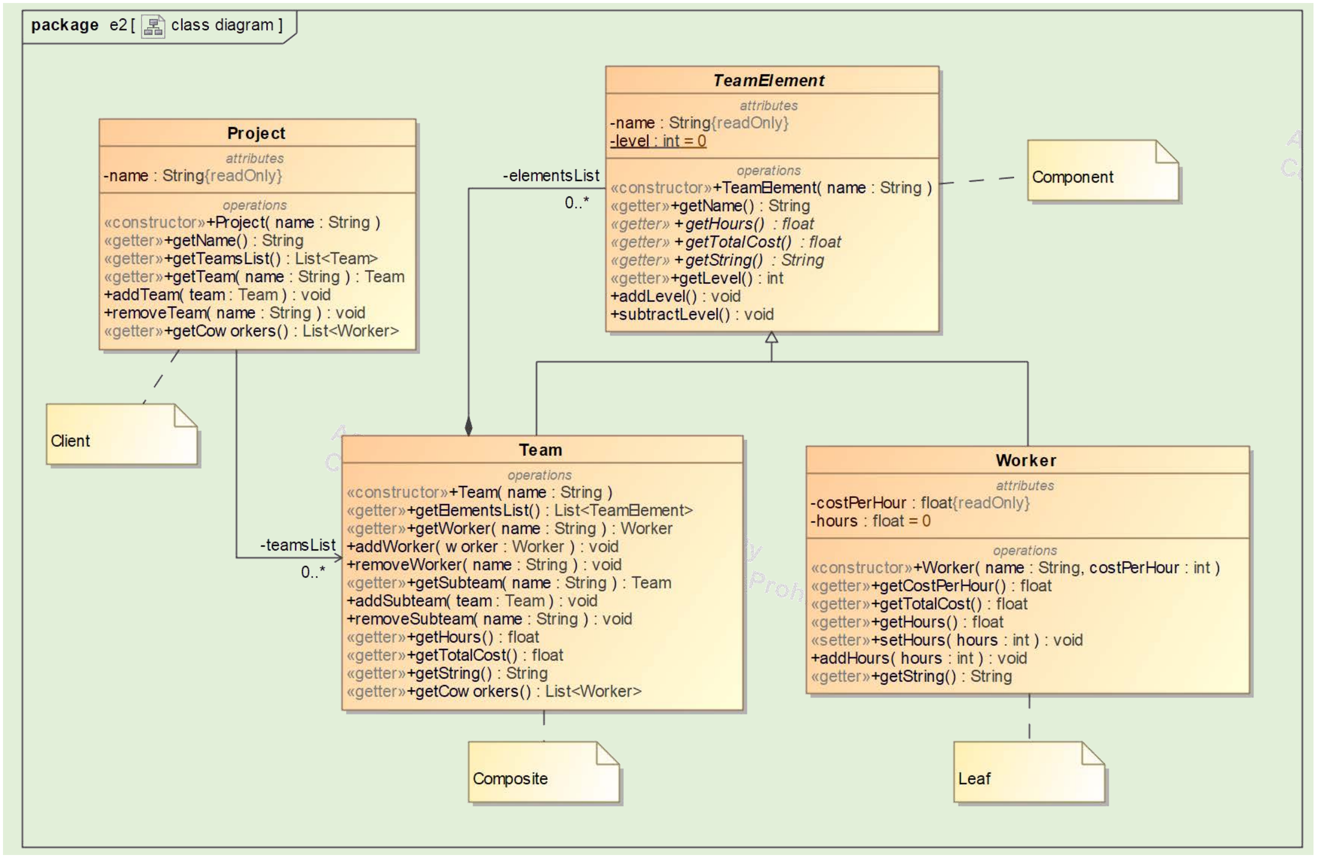
`public Team getTeam(String name){...}` (in the **Project** class)
`public Worker getWorker(String name){...}` (in the **Team** class)
`...`

We used the following design pattern:

- Composite Pattern

This pattern fits the structure of the work groups: a project is composed of teams, and each of these will contain workers and other teams. By creating an abstract class (**TeamElement**), we can treat the elements inside a team uniformly.

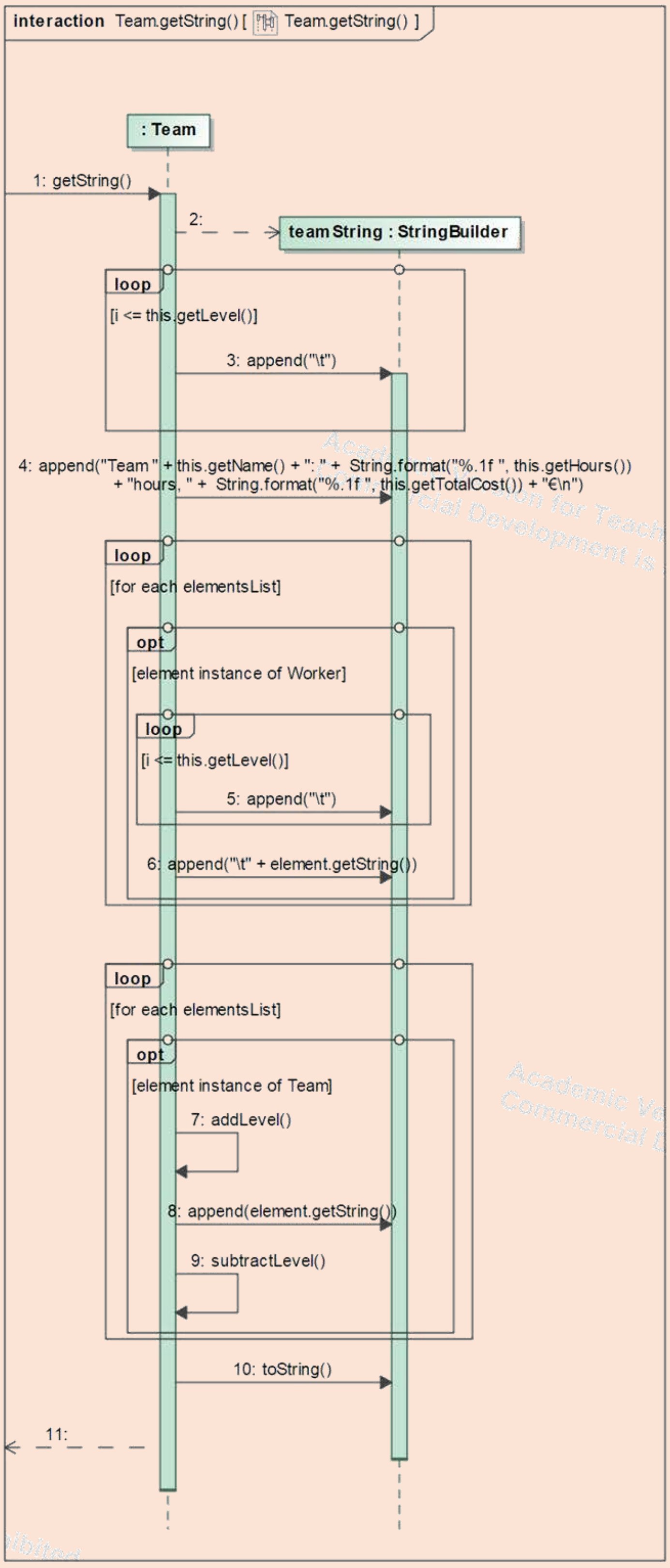
Class diagram

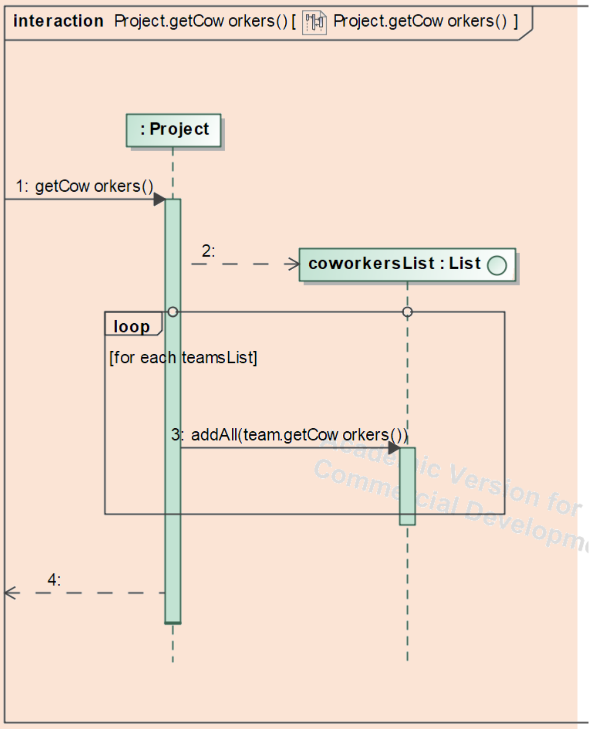

package e2 [class diagram]
Project
attributes
-name : String{readOnly}
operations
«constructor»+Project(name : String)
«getter»+getName() : String
«getter»+getTeamsList() : List<Team>
«getter»+getTeam(name : String) : Team
+addTeam(team : Team) : void
+removeTeam(name : String) : void
«getter»+getCoworkers() : List<Worker>
TeamElement
attributes
-name : String{readOnly}
-level : int = 0
operations
«constructor»+TeamElement(name : String)
«getter»+getName() : String
«getter» +getHours() : float
«getter» +getTotalCost() : float
«getter» +getString() : String
«getter»+getLevel() : int
+addLevel() : void
+subtractLevel() : void
-elementsList
0..*
Component
Client
-teamsList
0..*
Team
operations
«constructor»+Team(name : String)
«getter»+getElementsList() : List<TeamElement>
«getter»+getWorker(name : String) : Worker
+addWorker(worker : Worker) : void
+removeWorker(name : String) : void
«getter»+getSubteam(name : String) : Team
+addSubteam(team : Team) : void
+removeSubteam(name : String) : void
«getter»+getHours() : float
«getter»+getTotalCost() : float
«getter»+getString() : String
«getter»+getCoworkers() : List<Worker>
Worker
attributes
-costPerHour : float{readOnly}
-hours : float = 0
operations
«constructor»+Worker(name : String, costPerHour : int)
«getter»+getCostPerHour() : float
«getter»+getTotalCost() : float
«getter»+getHours() : float
«setter»+setHours(hours : int) : void
+addHours(hours : int) : void
«getter»+getString() : String
Composite
Leaf

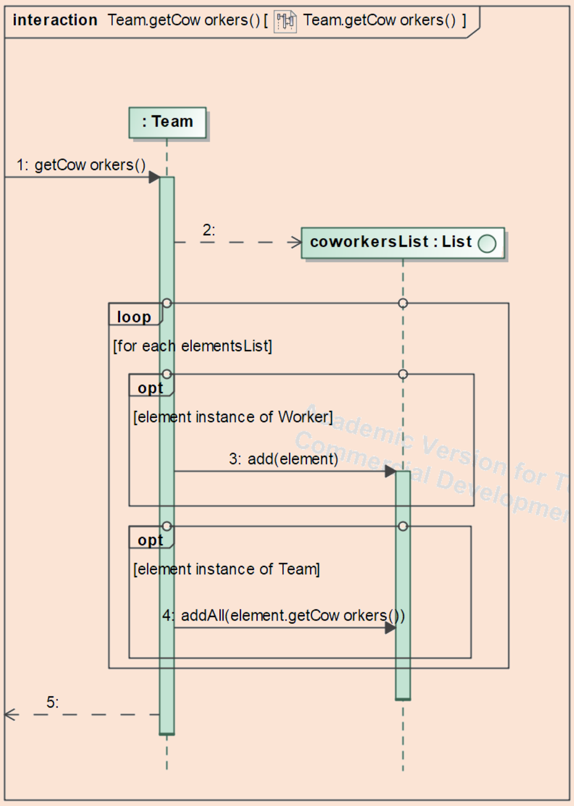
Dynamic diagrams

Three sequence diagrams are shown below. The left one represents the operation of the `getString()` method of the **Team** class, and other two represent the `getCoworkers()` methods of the **Project** and **Team** classes.

Other methods, like those used to add, get or remove elements to projects or teams are not shown here because their operations are very simple.


interaction Team.getString() [Team.getString()]
: Team
1: getString()
2:
teamString : StringBuilder
loop
[i <= this.getLevel()]
3: append("\t")
4: append("Team " + this.getName() + ": " + String.format("%.1f ", this.getHours())
+ "hours, " + String.format("%.1f ", this.getTotalCost()) + "€\n")
loop
[for each elementsList]
opt
[element instance of Worker]
loop
[i <= this.getLevel()]
5: append("\t")
6: append("\t" + element.getString())
loop
[for each elementsList]
opt
[element instance of Team]
7: addLevel()
8: append(element.getString())
9: subtractLevel()
10: toString()
11:


interaction Project.getCow orkers() [Project.getCow orkers()]
: Project
1: getCow orkers()
2:
coworkersList : List
loop
[for each teamsList]
3: addAll(team.getCow orkers())
4:


interaction Team.getCow orkers() [Team.getCow orkers()]
: Team
1: getCow orkers()
2:
coworkersList : List
loop
[for each elementsList]
opt
[element instance of Worker]
3: add(element)
opt
[element instance of Team]
4: addAll(element.getCow orkers())
5: