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

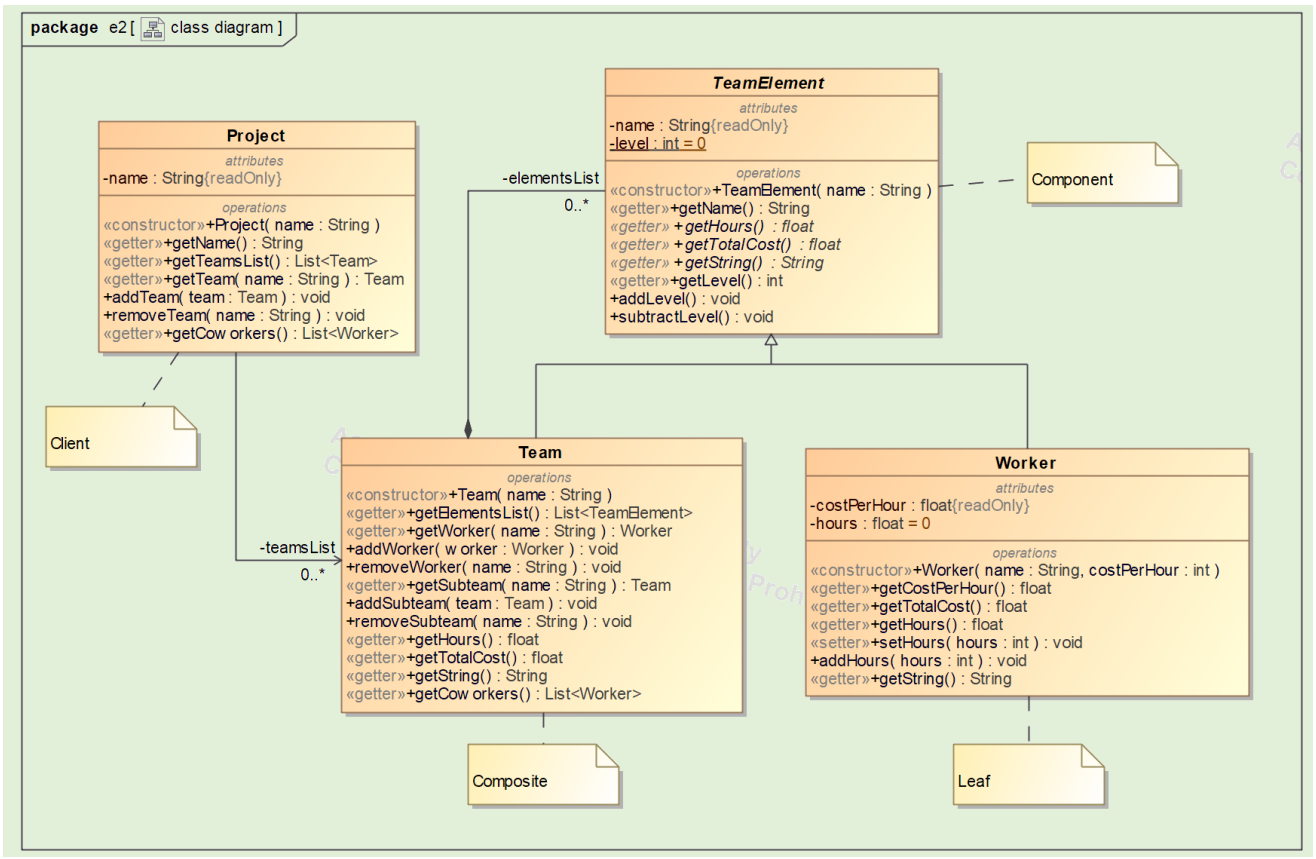
`public Team getTeam(String name){...}` (in the **Project** class)
`public Worker getWorker(String name){...}` (in the **Team** class)
`...`

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



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.