# ODL - Rapport Exercice Dirigé

ABDOUN Abderrahmen & ELGHAZOUANI Nada

20 décembre 2019

## 1 Explication du programme

Pour réaliser ce programme, nous avons repris le diagramme de classes présent dans le protocole donné. En effet, l'organisation est très semblable mais nous avons néanmoins notre propre implémentation du logiciel que ce soit au niveau du personnel ou au niveau de la gestion des projets.

Tout d'abord, nous créons un projet qui est une instanciation de la classe *Project*. Ce projet est un évènement quelconque comme un concert ou une pièce de théâtre par exemple. Dans le constructeur de Project, on incrémente le numéro de projet (projectID). Nous avons ensuite créer des fonctions *set()* et *get()* afin de pouvoir modifier les membres de cette classe.

Afin de mettre ce projet sur place (prenons l'exemple du concert), nou-s avons besoin de réaliser diverses tâches par l'intermédiaire de plusieurs instanciations de la classe *Task*. Dans notre exemple, nous avons créé 3 tâches : Montage de la scène, installation du matériel audio et installation du jeu de lumières. Ces 3 tâches sont contenues dans une List. Nous avons d'ailleurs créé un constructeur alternatif pour Project dans lequel on passe cette List ainsi que la deadline du projet. Afin de créer une tâche, nous faisons appel à son constructeur qui initialise le nom de la tâche (montage de la scène, installation du matériel audio, installation du jeu de lumières), initialise une éventuelle description (pas description pour les trois tâches dans notre exemple). Il initialise la liste de skills nécessaire à la réalisation de la tâche (*skills* est une classe qui contient un string *skillName*). Liste de skills est : ouvrier, machiniste de plateau, cameraman, régisseur du spectacle, éclairagiste, technicien du son, metteur en scène et électricien. La classe contient aussi les méthodes *set()* et *get()* afin de pouvoir modifier les membres de celle-ci.

Chaque tâche est réalisée dans une tranche horaire définie via la classe Agenda. Cette classe contient une date de départ et de fin. Le type de ces deux variables est 'Date' importée via *import java.util.Date* qui est elle-même une classe contenant les integers suivants : year, month, day, hour. La classe contient aussi les méthodes *set()* et *get()* afin de pouvoir modifier les membres de celle-ci.

La tâche a aussi besoin de ressources matérielles et humaines. Ces deux ressources sont représentées respectivement via les classes *Ressource* et *Worker*. La première dispose des membres suivants : nom de la ressource (ex : boite à outils), quantité de la ressource et une List de ressources(même type) tels que marteau, tournevis, niveau, etc. La classe contient aussi les méthodes *set()* et *get()* ainsi que des méthodes pour ajouter et supprimer une ressource de List. La classe Worker hérite de la classe abstraite *User*. Cette dernière contient le nom, prénom et un id s'auto-incrémentant dans

le constructeur. Les méthodes *get()* sont définies dans la classe et récrites dans le Worker mais les *set()* seulement dans le Worker. Le Worker contient en plus les membres disponibilité (booléen) et une liste de skills propres.

Les workers sont réunis dans des équipes via une classe *Team* affectée à une tâche. Les membres de cette classe sont : id du chef, nom de l'équipe et la liste de Worker. La classe contient aussi les méthodes *set()* et *get()* afin de pouvoir modifier les membres de celle-ci.

Worker hérite de User tout comme la classe Manager. Ce dernier est celui qui crée les tâches, peut modifier les membres des autres classes et qui peut avoir accès à tous les agendas (via une map disposant des Agendas avec la clé qui est le Worker).

# 2 Implémentation des Design Patterns

## 2.1 Composite

Nous avons implémenté le design pattern *composite* pour gérer les ressources. Pour ce faire, nous avons créé une interface *RessourceInterface* que nous avons changé en classe abstraite par soucis d'héritage pour la classe *Observable* (cf. point 2.3 Observer). Cette classe abstraite ne contient que des méthodes (des *get() set()*) réécrites dans la classe Ressource (Ressource hérite de RessourceInterface).

## 2.2 Strategy

Nous implémentons l'algorithme de création de tâches selon le design pattern *Strategy*. De ce fait, nous avons deux manières de les créer. Nous avons donc créé deux classes *CreateTask1* et *CreateTask2* et chacune dispose d'une seule fonction se chargeant de créer la tâche (composer l'équipe, affecter un agenda et affecter une liste des ressources nécessaires. Le CreateTask1 crée une liste de travailleurs dans laquelle le chef d'équipe est choisi par le manager et puis une équipe est instanciée en donnant la liste des travailleurs, l'id du chef et le nom de l'équipe. On implémente ensuite deux boucles 'while' afin de former cette équipe en fonction de leur disponibilité et des skills requis pour cette tâche. Ensuite, nous affectons l'agenda et la liste des ressources passés en paramètres de la fonction. Le createTask2 est identique quant à l'affectation de l'agenda et des ressources mais pas pour la création de l'équipe. En effet, ici nous avons choisi d'affecter les quatre premiers workers de la liste reprenant tous les travailleurs et le premier ajouté devient automatiquement le chef.

## 2.3 Observer

Afin d'implémenter ce design pattern, nous avons implémenté une interface *Observer* qui contient une fonction *update()* que nous allons réécrire dans la classe Manager. Ensuite étant donné que nous devons observer les classes Worker, Task et Ressource, nous devons dire que ces dernières héritent de la classe *Observable* (tout en important *java.util.Observable*). Dans ces classes, à chaque fois qu'un membre est modifié via les fonctions *set()*, *add()* ou encore *delete()*, nous envoyons une notification

via la fonction *notifyObservers()*. Ainsi le manager est averti d'une modification par la fonction *update()* réécrite par nous-même.