Rapport Projet Zuul

By Gabriel Leroux

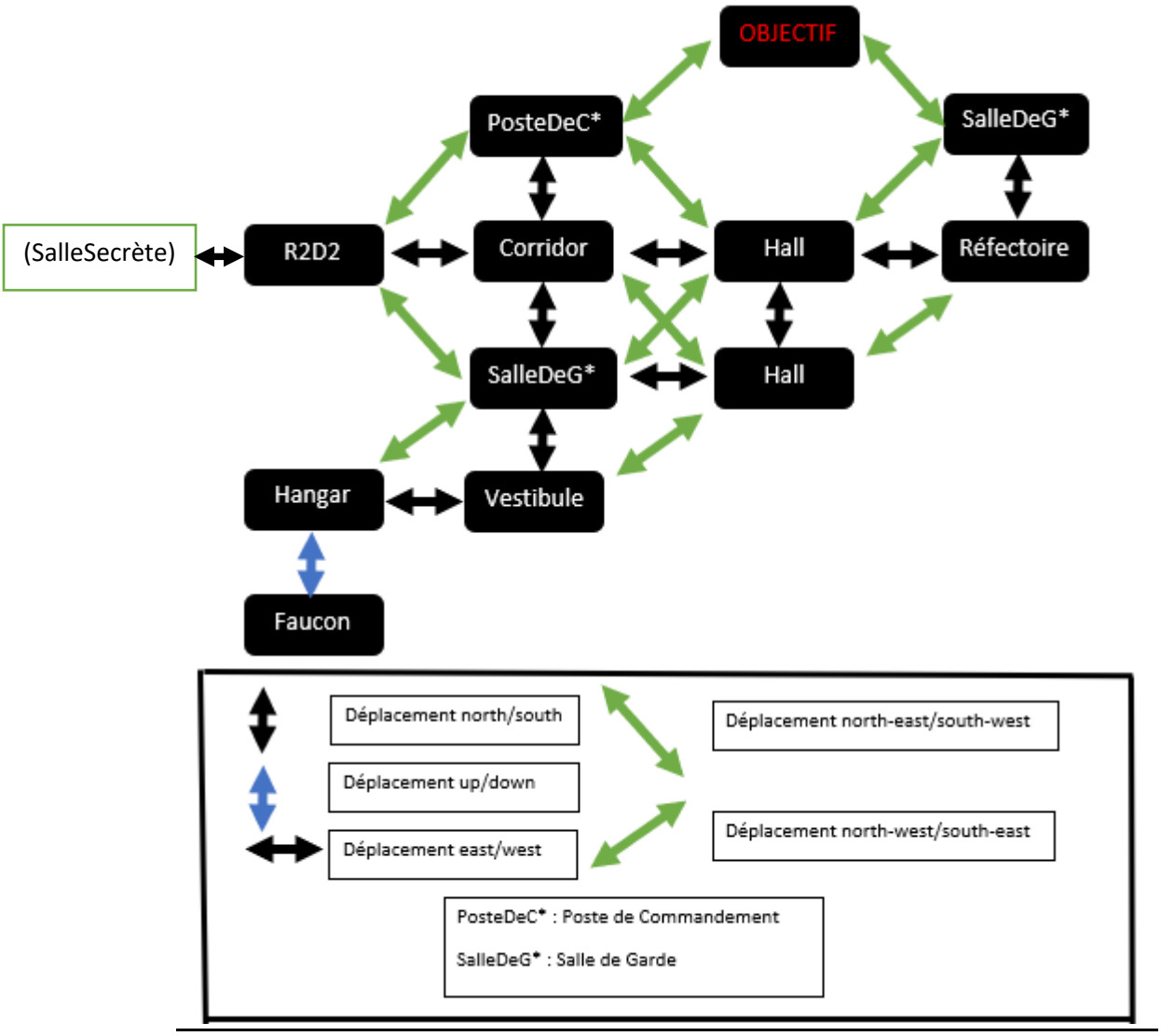
Voici le rapport de projet demandé, à partir de la liste officielle des exercices du projet et de la présentation décrite sur icampus.

I.A) Auteur : Gabriel Leroux, étudiant ESIEE Paris.

I.B) Phrase thème : Un jedi, vous, doit aller secourir la Princesse Leïa de l'Empire, en traversant l'Etoile de la Mort.

I.C) Résumé du scénario : Vous devez sortir du Faucon Millenium et vous déplacer à l'intérieur de l'Etoile de la Mort. Votre but est de vous rendre là où sont enfermés les prisonniers de marque (prison) pour aller sauver la Princesse Leïa d'une mort certaine. Suite à cela, vous devrez, bien entendu, retourner avec elle au Faucon Millenium afin de vous enfuir.

I.D) Plan complet du jeu :



I.E) Scénario détaillé du sujet :

Un jedi, prenant son courage à mains décide d'aller secourir la Princesse Leïa, leader des troupes rebelles des mains de l'Empire. Vous savez de source sûre qu'elle est retenue captive dans l'Etoile de la Mort de l'Empire. A l'aide des espions rebelles vous avez réussi à vous emparer des plans de l'Etoile de la Mort qui vous permettront de vous repérer dans cette gigantesque station spatiale. L'objectif est donc d'aller secourir la Princesse et de vous enfuir à bord du vaisseau Millenium.

I.F) Détail des lieux, items, personnages…

La salle de départ est, vous vous en douterez, le Hangar. Il vous est impossible de faire demi-tour. En effet, il vous faut impérativement la Princesse Leïa pour pouvoir remonter à bord du Faucon Millenium et ainsi, gagner la partie. Il existe différent type de sorties. Il existe les TrapDoor se refermant après être sorti de la pièce et les passages nécessitant un objet dans votre inventaire pour débloquer la sortie et ainsi, rentrer dans la salle suivante. Ces portes verrouillées et ces TrapDoor ne sont pas indiqués sur les plans qui vous sont fournis. Je les détails ci-après. Chaque item permettant de déverrouiller coute un certain nombre de rouble. L'objectif est que vous gériez votre argent pour toujours pouvoir acheter l'item qui vous permettra d'avancer dans le jeu. Le jeu ne possède qu'un seul point essentiel. Il y a un nombre limité de déplacement. Passé ce nombre, si vous n'avez pas réussi à vous enfuir accompagné de la Princesse vous perdez la partie. L'objectif est donc de mémoriser où se trouve les objets pour pouvoir, ensuite, les récupérer et débloquer les salles qui vous feront avancer vers votre objectif. Il y a aussi la possibilité de trouver des « bourses » dans les alles, qui, ne vous coûte rien et vous octroie un bonus d'un certain nombre de rouble en fonction de la taille de la Bourse. Vous l'aurez compris, plus on a d'agent mieux c'est. C'est pour cela, que dans la salle R2D2, vous trouverez un droïde qui pourra jouer avec vous à pierre feuille ciseaux. Si vous gagnez, le droïde vous donnera 10 roubles. Si vous perdez, vous perdez 20 roubles. Faites donc bien attention. Un dernier détail et pas des moindre. Il existe un item disposé dans une certaine pièce se nommant Téléporteur. Deux commandes lui sont associés. Après l'avoir pris, vous l'aurez dans votre inventaire. En utilisant la commande « charger » puis le nom de l'item concerné, en l'occurrence « Téléporteur », vous choisissez de rentrer en mémoire la salle où vous vous trouvez actuellement. En utilisant la commande « tirer Téléporteur », vous serait instantanément téléporté vers la salle précédemment mémorisée. Attention, se téléporter compte quand même pour un déplacement.

- Du hangar au vestibule, il vous faut l'item « sabre », se trouvant dans le Hangar et coutant 50 roubles pour passer.
- Du vestibule au hall1, il vous faut l'item « chaise-longue », se trouvant dans le hangar et coutant 50 roubles pour passer.
- De la salleDeGarde1 (celle la plus au sud de la carte) au Hall1 (hall le plus au sud), il vous faut l'item « lance-flamme », se trouvant dans la salle R2D2 et coutant 50 roubles pour passer.
- De la salledegarde1 au Hall2(Hall le plus au nord), il vous faut l'item « épuisette », se trouvant dans le Corridor et coutant 50 roubles pour passer.
- Du PosteDeCommandement à votre Objectif (Prison), il vous faut l'item « perceuse-visseuse (Bosch) » se trouvant dans la salle R2D2 et coutant 220 roubles pour passer.
- De votre Objectif (Prison) à la SalleDegarde2(salle de garde la plus au nord de la carte), il vous faut l'item « disqueuse-ACDC » coutant 220 roubles pour passer.

Tous les autres items, hormis le Téléporteur ne servent à rien. Bien entendu, si vous déposer l'item que vous possédez vous regagner le prix que cela vous a couté de le prendre.

Il y a une bourse de 20 roubles dans le Vestibule et une de 100 roubles dans le Refectoire pour l'instant. A l'avenir il y aura peut-être une bourse de 80 roubles servant à éponger les dettes des joueurs dans une salle secrète à l'ouest de la salle R2D2.

Vous partez du départ avec la modeste somme de 100 roubles. Enfin, il y a une Trap Door :

- Du hangar à la salleDeGarde1
- De R2D2 au posteDeCommandement
- Du posteDeCommandement à votre objectif

- Du hangar au Faucon.

I.G) Situations gagnantes et perdantes

Réussir à sauver la princesse et à revenir avec elle au Faucon Millenium est la seule et unique situation gagnante.

La seule situation perdante est de ne plus avoir de déplacement. Vous partez avec 30 (X) nombre de déplacement. Si vous ne pouvez plus vous déplacez, vous ne pouvez plus avancer dans le jeu et par conséquent perdez et tombez aux mains de l'Empire.

I.H) Eventuellement énigmes, mini-jeux, combats…

Possibilité de jouer au pierre feuille ciseaux avec R2D2.

I.I) Commentaires (ce qui manque, ce qui reste à faire…)

Peut-être la construction de la Salle secrète dont je vous ai parlé ci-haut qui se débloquerai seulement si le joueur joue 4 fois ou plus avec R2D2.

II)Réponses aux exercices :

Exercice 7.5 : Pour éviter les duplications de code qui étaient de plus en plus présentes avec les précédents tps, j'ai créé une nouvelle procédure printLocationInfo() qui a pour rôle d'afficher les différentes sorties des salles. En effet, les sorties sont affichées par goRoom(), si la Room courante n'est pas égal à null, et par printWelcome() au commencement du jeu. Le fait d'avoir créé cette procédure permet d'encapsuler : il suffit de faire appel à cette procédure dans goRoom() et printWelcome() plutôt que de recopier plusieurs fois la même chose. Il faut à tout prix éviter de dupliquer du code.

Exercice 7.6 : Pour éviter qu'à chaque modification des directions, je modifie du code dans deux classes différentes à savoir Room et Game, il faut avoir un couplage faible. Pour réduire le couplage, j'ai créé une fonction getExit() retournant une Room, prenant comme paramètre une direction. Cette direction indique au programme, la prochaine Room qui sera ensuite stocké dans la variable aCurrentRoom. Le but final étant de basculer les attributs de la classe Room (directions) qui étaient publics i.e (c'est-à-dire) à qui on pouvait faire appel librement, en attributs privés. Je peux désormais dans la classe Game simplement faire appel à la fonction getExit() pour connaître la prochaine Room après le déplacement du joueur. Je me suis rendu compte qu'après cette modification je perdais la possibilité de tester si la commande était valide ou non comme l'indique le post du 2 octobre 2018 sur le forum. Pour résoudre ce problème, la classe Room retourne la pièce courante quand la direction n'est pas connue. Suite à cela nous pouvons rajouter dans goRoom(), pour bien différencier les deux types d'erreurs (direction inconnue et le fait qu'il n'y a pas de sortie dans la direction demandée) :

```
if (vNextRoom==this.aCurrentRoom){
    System.out.println("Commande inconnue");}
else if (vNextRoom==null){
    System.out.println("Il n'y a pas de sortie dans cette diretcion !");
}
else{// sinon la room current devient celle tapée au clavier et on affic
    this.aCurrentRoom=vNextRoom;
    printLocationInfo();
}
```

Exercice 7.7 : Exercice 7.7 :

Une nouvelle fois, pour éviter de réécrire identiquement les différentes sorties disponibles, j'ai créé dans la classe Room une fonction getExitString(). Cette même fonction, nouvellement créé se trouve effectivement dans la classe Room car il est logique que ce soit cette même classe qui produise ses propres informations et que Game ne fasse qu'appeler cette fonction. getExitString() est donc une concaténation qui permet de tester chaque sortie pour connaître son existence à partir de la pièce courante.

Exercice 7.8 : Dans le but de simplifier et d'optimiser au maximum l'ajout de nouvelles directions j'ai utilisé un nouvel outil : une Hashmap. J'ai reconstruit entièrement sur la base de ce nouvel outil la classe Room. Déjà lors de la déclaration des attributs, on voit à quel point une Hashmap est pratique. Au lieu de déclarer un attribut par direction, il nous suffit de taper : « private HashMap<String, Room>exits; ». Bien entendu, il nous faut aussi modifier les fonctions setExit() et getExit() en utilisant les fonctionnalités de la Hashmap.

```
import java.util.HashMap;
import java.util.Set;
public class Room
{
    private String aDescription;
    private HashMap<String, Room>exits;

    public Room(final String pDescription){
       this.aDescription=pDescription;
       exits=new HashMap <String, Room>();
     }

    public String getDescription(){
        return this.aDescription;
     }
    /**
     * Définit les sorti'es de cette pièce. Chaque direction
     * soit conduit à une autre pièce soit est null (il n'y
     * a pas de sortie dans cette direction).
     */
    public void setExit(final String pDirection, final Room pNeighbor){

      exits.put(pDirection, pNeighbor);

    }
```

Exercice 7.8.1 : Après la création de la Hashmap, il nous est aisé de créer de nouvelles directions. Ainsi j'ai créé les directions north-east, south-east, north-west, south-west et up et down. Je les ai créés dans la classe Game comme ceci : « vFaucon.setExit("down", vHangar); ». Ainsi le code n'est pas forcément plus simple i.e moins long dans la classe Game mais surtout beaucoup plus pratique dans le sens où l'on peut créer n'importe quelle direction qui donne à n'importe quelle pièce de notre choix.

Exercice 7.9 :

keySet

```
public Set<K> keySet()
```

Returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.

Specified by:
keySet in interface Map<K,V>
Overrides:
keySet in class AbstractMap<K,V>

Returns:
a set view of the keys contained in this map

D'après la documentation java, la méthode keySet de la classe Hashmap renvoie une vue d'ensemble des clés contenues dans la carte.

Exercice 7.10 : Lorsque l'on pénètre, dans la fonction getExitString(), on initialise une variable locale de type String nommée returnString contenant la chaine de caractère «Sorties ». Suite à cela, nous allons utiliser la méthode keySet expliquée à la question précédente. Nous allons mettre dans la variable keys l'ensemble des clés de la Hashmap exits. On parcoure keys, à l'aide d'une boucle for each. Cette même boucle initialise une variable de type String exit. Pour chaque chaîne de caractère exit de la collection keys, changer la variable returnString, en concaténant la précédente variable returnString et exit. On sort de la boucle et on return returnString. D'après le code 7.7 donné dans le livre.

Exercice 7.11 : Pour encore réduire le couplage, et laisser encore plus d'indépendance (c'est-à-dire que modifier la classe Room modifie toute les classes car toutes les données sont centralisées

```
/**
 * Renvoie une description détaillée de ette pièce
 * sous la forme :
 *  Vous êtes dans ...
 *  Les sorties sont : ...
 *  @ return Une description de la pièce, avec les sorties
 */
public String getLongDescription(){
    return ""+ getDescription() +"\n" + getExitString();
}
```

dans la classe Room) à la classe Room, cette dernière va dorénavant se charger d'afficher la description de la pièce et des sorties disponibles. Je peux maintenant seulement faire appel à la nouvelle méthode getLongDescription(), dans la classe Room depuis printLocationInfo() présentes dans la classe Game.

Exercice 7.14 : Je viens d'ajouter une commande « look ». Pour ce faire, j'ai définit un tableau dans la class CommandWords avec l'ensemble de toutes mes commandes. A la suite de cela, j'ai rajouté dans la fonction processCommand() de la classe Game, que si la commande entrée est la commande look, on exécute la procédure look(). J'ai donc, créé une procédure look(), qui affiche le retour de la fonction getLongDescription(). Elle fais donc strictement la même action que la procédure printLocationInfo(). J'ajusterai, ce que la commande look effectue en fonction de la suite du projet.

```
public class CommandWords
{
    // a constant array that will hold all valid command words
    private final String[] aValidCommands={"go","help","quit","look"};

    /**
     * Constructor - initialise the command words.
     */
    public CommandWords()
    {

    } // CommandWords()
```

Exercice 7.15 : De même, j'ai rajouté une commande eat de la même manière que ce que j'avais fait pour la commande look. Cette commande affiche simplement « Vous avez mangé, vous n'avez plus faim ».

Exercice 7.16 : Pour éviter qu'à chaque rajout de commande, je modifie le code de la procédure printHelp() manuellement, l'objectif et de faire appel à la méthode showCommands() de Parser qui elle-même fais appel à la méthode showAll() de la classe CommandWords. Dans le corps de la méthode showAll(), il y a une boucle for each. Pour chaque String command de la collection aValidCommand, j'affiche la variable command en question suivi d'un espace. On a donc bien la liste des différentes commandes existantes qui s'affiche.

```
public void showAll(){
    for(String command : aValidCommands){
        System.out.print(command + "  ");
    }
    System.out.println();
}
```

Exercice 7.18 à 7.18.5 : Je commence par modifier la méthode showAll de la classe CommandWords permettant d'afficher l'ensemble des commandes existantes. Au lieu, d'afficher, à l'aide la boucle for each comme expliquer ci-avant, du texte directement dans la fenêtre, je choisis de transformer cette procédure en une fonction que l'on nommera getCommandList et qui retournera une String. Cela me permettra à l'avenir, d'utiliser dans d'autre situation qu'un affichage textuel, cette variable de type String.

Suite à cela, j'ai suivi les consignes de la liste officielle des exercices du projet. J'ai comparé mon projet à zuul-better, j'ai cherché et enregistré toutes mes images et j'ai donné un titre à mon jeu que j'affiche lors de son lancement. Enfin, j'ai créé une HashMap contenant toutes les rooms du jeu dans la classe Game à la suite de la création des rooms.

Exercice 7.18.6 : Le but de cette exercice est d'étudier le projet fourni zuul-with image dans la liste des exercices. J'ai ensuite, modifié mon code en conséquence en remplaçant ce qu'il y avait à remplacer et en ajoutant les différentes classes que je ne possédais pas (séparer la classe Game en deux classes, GameEngine qui contiendra ce que contenait précédemment la classe Game et la classe Game qui lance simplement le jeu, j'ai changé la classe Parser n'utilisant dorénavant plus de Scanner, et j'ai créé la classe UserInterface). Comme demandé, j'ai gardé ma méthode printLocationInfo() qui était dans ma classe Game que j'ai transféré dans la classe GameEngine. Je l'ai modifié de tel façon à ce qu'elle affiche l'image de la salle courante.

```
private void printLocationInfo(){
    this.aGui.println(this.aCurrentRoom.getLongDescription());
    if ( this.aCurrentRoom.getImageName() != null )
        this.aGui.showImage( this.aCurrentRoom.getImageName() );
}
```

J'ai ensuite, bien compris la classe UserInterface, capital pour afficher et positionner les images. Pour cela je me suis aidé de la JavaDoc et j'ai aussi supprimé les imports avec * pour simplement les remplacer par des imports plus précis contenant juste la classe à importer.

Exercice 7.18.8 : L'objectif de cette exercice est de créer un bouton fonctionnel. Pour ce faire, il me suffit de créer un attribut de type JButton dans la classe UserInterface que je nomme aBouton. Ensuite, dans la procédure createGUI(), j'initialise mon bouton avec ce que je veux qu'il y ai marqué (quit). Je place le bouton, à un endroit où il n'y a rien de déjà définit, en l'occurrence dans mon cas, l'est. Toujours dans cette même méthode, je fais appel à la classe ActionListener sur l'objet courant qui ne contient qu'une procédure actionPerformed prenant une ActionEvent en paramètre. Finalement, dans

actionPerformed je dis que si, la référence du paramètre (à l'aide de getSource()) est la même que celle de mon bouton, alors j'exécute la fonction interpretCommand de la classe GameEngine avec comme paramètre la string retourné par la commande aBouton.getActionCommand().

```
public void actionPerformed( final ActionEvent pE )
{
    // no need to check the type of action at the moment
    // because there is only one possible action (text input) :
    if(pE.getSource()==this.aBouton){
    this.aEngine.interpretCommand(aBouton.getActionCommand());return;}

    this.processCommand(); // never suppress this line
} // actionPerformed(.)
```

Exercice 7.19 : Création d'un dossier « Images » à la racine du projet contenant toutes les images présentes dans le jeu (pour l'instant de chaque salle), que j'affiche de la même manière que précédemment. Je change simplement dans la création de la Room en précisant le chemin d'accès. Par exemple « Images/Vestibule.gif ».

Exercice 7.20-7.21 : L'objectif de cette exercice et de rajouter des items dans les pièces de nos choix. J'ai donc rajouté un item sabre laser de poids (prix) 3. J'ai commencé par créer une classe Item, contenant deux attributs de types int et String, un constructeur et un accesseur. Dans la classe GameEngine avant la création des salles j'ai créé un Item vSabre à l'aide de la méthode setItem présente dans le code de la classe Game Engine, ayant pour but de créer un Item, à partir des paramètres demandés (une String et un int) : Item vSabre=new setItem(« sabre laser », 3). A partir de là, j'ai rajouté quand je créé mes rooms une virgule, et l'item présent dans la salle. S'il n'y a pas d'item, je mets null. Ensuite, dans la classe Room, je suis donc contraint de rajouter un attribut de type Item aItem. Je l'initialise dans le constructeur. Ainsi, je n'ai plus qu'à créer une fonction retournant une String nommée afficheItem(), qui dit que si l'item est égal à null, alors je retourne, il n'y a pas d'objet dans cette pièce sinon, je retourne ce qu'il y a dans l'accesseur de ma classe Item, c'est-à-dire le nom : le prix. Il ne me reste donc plus qu'à appeler cette fonction dans getLongDescription(). Toujours dans la classe Room. La classe EngineGame se charge donc de l'affichage de l'item grâce à la procédure printLocationInfo() qui fais appel à la fonction getLongDescription() de la même manière que pour l'affichage des sorties et la description de la pièce.

Exercice 7.22-7.22.2 : Au lieu d'être limité à un seul item par Room, je choisis de créer une nouvelle HashMap <String, Item> items dans la classe Room. J'ai donc supprimé le paramètre Item, qui était dans mon constructeur naturel, et j'ai initialisé ma HashMap items dans ce même constructeur. De la même manière que la méthode setExit, je crée, une méthode setItem qui prend en paramètre une Srting et un prix, pour ensuite créer un item et l'intégrer à la HashMap items que je viens de créer, toujours dans la classe Room. Par conséquent, j'ai dans ma classe GameEngine, le nom de la salle.setItem(« sonNom », sonPrix). Pour afficher ces items, comme précédemment on va se servir de

```
/**
 * sers à afficher les items présents dans la pièce
 */
public String afficheItem(){
    if (items==null){
        return "Il n'y a pas d'objet dans cette salle";
    }
    else{
        String vRetour = "";
        Set <String> vItemm=items.keySet();
        for(String vItem : vItemm){
           vRetour+=items.get(vItem).getItem();}
        return vRetour;}
}
```

la méthode getLongDescription() qui fais toujours appel à notre méthode affiche() qui retourne la liste des items présent dans la pièce comme ceci :

Par la suite, j'ai mis autant d'objet dans chaque pièce que je le souhaite.

Exercice 7.23 : L'objectif est de créer une commande back, nous redirigeant vers la salle précédente où le joueur se trouvait. Pour cela, en plus de rajouter une String « back » au tableau de commande dans la classe CommandWords, il faut initialiser un nouvel attribut Room piecePassee dans GameEngine. Dans la méthode interpretCommand on rajoute :

```
else if ( vCommandWord.equals( "back" ) )
    if (vCommand.hasSecondWord()){
        this.aGui.println( "Que vouliez-vous dire ? Pour retourner en arrière taper simplement 'back'. " );
    }

    else{ this.back();}
```

Je crée une méthode back qui nous dis :

```
public void back(){
    if (this.piecePassee==null){
        this.aGui.println( "Vous n'avez pas encore bougé, ou voulez vous donc retourner ?!" );
        return;}
    Room vLocal=this.aCurrentRoom;
    this.aCurrentRoom=this.piecePassee;
    this.piecePassee=vLocal;
    printLocationInfo();

}
```

Finalement, dans goRoom, lorsqu'il y a un déplacement je dis que avant de dire que this.aCurrentRoom=vNextRoom, this.piecePassee=this.aCurrentRoom.

Exercice 7.26 : Je choisis de changer maintenant la méthode back et son fonctionnement en utilisant la collection stack(). Pour ce la au lieu de créer un attribut un attribut de type Room, je choisis de créer un attribut aStackRoom avec : « private Stack<Room> aStackRoom. On initialise ensuite cet attribut dans le constructeur. Ensuite, j'ajoute, dans goRoom, la room courante avec la méthode push() de stack de cette manière : this.aStackRoom.push(CurrentRoom). Dans back, il nous suffit alors simplement d'enlever une pièce de cette pile si la pile n'est pas vide en utilisant la méthode pop() de stack : this.aCuurentRoom=this.aStackRoom.pop(). Il ne me reste plus qu'à afficher en utiliser toujours la méthode printLocationInfo().

En résumé, stack(), est une collection qui ressemble à une pile. On peut ajouter des objet à la pile à l'aide de la méthode push() et en enlever à l'aide de pop(). Empty(), est une fonction booléenne qui revoit true si la stack() et vide est false sinon. Peek() permet de renvoyer l'objet se trouvant au-dessus de la stack() sans la supprimer.

Exercice 7.28.1 : J'ai créé une commande test en l'ajoutant au tableau des mots de commande dans la classe CommandWord(). Suite à cela, j'ai créé un fichier .text dans mon répertoire courant dénommé

```
public void test(final String pTest){
    this.aGui.println("yo");
    try {Scanner sr =new Scanner (new File(pTest+".txt"));
    while (sr.hasNextLine()){
        String ligne = sr.nextLine();
        this.interpretCommand(ligne);
    }
    sr.close();
    }
    catch (java.io.FileNotFoundException sr){ this.aGui.println("Le fichier n'existe pas.");}
}
```

court avec quelques commandes courantes dedans. J'ai modifié ma méthode interpretCommand(), de manière à ce que, lorsque la commande test est exécutée, cela exécute la procédure test de la classe GameEngine(), avec comme paramètre le second mot rentré après test. Il ne me reste donc plus qu'à créer la procédure test comme ceci :

Exercice 7.29 : Création de la classe Player. La classe GameEngine contenant trop de code, je choisis de réorganiser mon code en passant par la classe Player contenant toutes les informations pratiques liés aux joueurs, principalement en déplacement toutes les instructions liées à la Room courante. Ma Classe GameEngine est maintenant simplifiée et voici ce que donne ma classe Player :

```
/**
 * Affiche la méthode look qui affiche une description de la pièce courante
 */
public void look(){
    printLocationInfo();
}

/**
 * Changer la Room
 */
public void changerLaRoom(final Room pRoom){
    this.aCurrentRoom=pRoom;
}
```

```
public class Playe
{
    private Room a
    private Stack
    private UserIn
    private String
    private int aP
    private Audio
    // ===== const
    /**
     * Constructeu
     */
    public Player(
    {
        this.aCurr
        this.aNom=pNom;
        this.aStackRoom= new Stack <Room>();
        this.aSon = new Audio();
    } // Player(.)
    /**
     * Constructeur de gui
     * @param GUI
     */
    public void setGUI( final UserInterface pUserInterface )
    {
        this.aGui = pUserInterface;
        this.printWelcome();
    }

    /**
     * Accesseur de aCurrentRoom
     */
    public Room getCurrentRoom(){
        return this.aCurrentRoom;
    }
```

```
/**
 * méthode permettant de se rediriger vers la précédente room visitée par le joueur.
 */
public void back(){
    if (this.aStackRoom.empty()==true){
        this.aGui.println( "Vous n'avez pas encore bougé, ou voulez vous donc retourner ?!" );
        return;}
    this.aCurrentRoom=this.aStackRoom.pop();
    printLocationInfo();
}
```

```
    this.aGui.println("Joueur : "+this.getNom());
    this.aGui.println("Bienvenue dans le Gaby's Game !");
    this.aGui.println("Je vous souhaite bonne fortune, et que la chance vous sourit ! ");
    this.aGui.println("  ");
    this.aGui.println("Type 'help' if you need help.");
    this.aGui.println("  ");
    printLocationInfo();
    //lance la musique
    this.aSon.run("imperial_march.wav");
}
```

```
/**
 * Procédure permettant d'afficher la description de la Room courante
 * ainsi que ses sorties en faisant appel à la foction getLon,gDescription()..
 *
 */
public void printLocationInfo(){
    this.aGui.println(this.aCurrentRoom.getLongDescription());
    if ( this.aCurrentRoom.getImageName() != null )
        this.aGui.showImage( this.aCurrentRoom.getImageName() );
```

```
public void prendre(final String pNom){
    if(this.aCurrentRoom.retourneItem(pNom).getPrix()<this.aArgent){
    this.aItem=this.aCurrentRoom.retourneItem(pNom);
    this.aCurrentRoom.suppItem(pNom);
    this.aArgent=this.aArgent-this.aItem.getPrix();
}
}
```

```
/**
* méthode permettant de déposer un objet
*/
public void deposer(){
    this.aCurrentRoom.addItem(this.aItem);
    this.aArgent=this.aArgent+this.aItem.getPrix();
}
```

Exercice 7.30 : J'ai créé en plus des commandes déjà disponible de nouvelles commandes. Les commandes prendre et déposer permettant respectivement d'emporter un objet qui était présent dans la pièce et de déposer, c'est-à-dire de laisser l'objet que j'ai sur moi dans la pièce et tout cela en respectant l'argent que le joueur possède sur lui. Pour ce faire, j'ai créé deux nouveaux attributs à ma classe Player. Un attribut de type int gérant l'argent que possède le joueur et un autre attribut de type Item, indiquant l'item que le joueur porte sur lui. Les nouvelles méthodes prendre et déposer de la

classe Player sont construites comme suit avec comme paramètre de la méthode prendre le second mot tapé par l'utilisateur :

C'est mêmes méthodes font appel aux méthodes add et supp Item de la classe Room qui supprime et ajoute l'item à la HashMap items de ma classe Room. Ce n'est pas précisé, mais bien entendu, il faut aussi codé correctement la méthode interpretCommand() de la classe GameEngine.

Exercice 7.31 : Simplement en modifiant l'attribut item de la classe Player en une HashMap <String,Item>, on peut désormais avoir plus d'un objet sur le joueur.

```
public void prendre(final String pNom){
   if(this.aCurrentRoom.retourneItem(pNom).getPrix()<this.aArgent){
    aListeItem.put(pNom,this.aCurrentRoom.retourneItem(pNom));
    this.aCurrentRoom.suppItem(pNom);
    this.aArgent=this.aArgent-aListeItem.get(pNom).getPrix();
}
}
```

```
/**
* méthode permettant de déposer un objet
*/
public void deposer(final String pNom){
    this.aCurrentRoom.addItem(aListeItem.get(pNom));
    this.aArgent=this.aArgent+aListeItem.get(pNom).getPrix();
}
```

```
 * @param String pNom
 */
private HashMap <String,Item> aListeItem;
```

```
/**
 * Constructeur d'objets de classe ItemList
 * Instancie la HashMap
 */
public ItemList()
{
    aListeItem=new HashMap<String,Item>();
}
```

```
/**
 * Rajouter un item dans la HashMap
 */
public void addItem(final String pString, final Item pItem)
{
    aListeItem.put(pString,pItem);
}
```

```
/**
 * Accesseur de la HashMap aListeItem
 */
public Item getItem(final String pString ){
    return aListeItem.get(pString);
}
```

```
/**
 * Enlever un item de la liste apres l'avoir déposer
 */
public void suppItem(final String pString){
    aListeItem.remove(pString);
}
```

```
/**
 * sers à afficher les items présents dans la pièce.
 */
public String afficheItem(){
    if (this.aListeItem==null){
        return "Il n'y a pas d'objet dans cette salle";
    }
    else{
        String vRetour = "";
        int vPoidsTotal=0;
        Set <String> vItemm=aListeItem.keySet();
        for(String vItem : vItemm){
            vRetour+=aListeItem.get(vItem).getItem()+ " ";
            vPoidsTotal+=aListeItem.get(vItem).getPrix();
        }
        return vRetour + "\n Le prix total de votre inventaire est de : "+ vPoidsTotal;}
}
```

Exercice 7.31.1-7.32 : L'objectif était de centraliser la gestion des listes des items dans une seule et même classe, la classe ItemList, que voici :

Il ne me reste plus qu'à supprimer les deux HashMap précédemment crée dans les classes room et player pour les remplacer par un attribut de type ItemList. Après, j'ai corrigé les erreurs de compilation pour arriver à un résultat fonctionnel. Cependant petit problème, j'affiche à chaque appel de afficheItem, que ce soit dans Room ou encore dans Player le prix que l'utilisateur/la Room possède dans son inventaire/dans la salle. Il faut corriger cela pour valider l'exercice 7.33 il me semble. J'ai omis de le préciser, mais bien évidemment il faut changer la méthode printlocationInfo() de la classe Player et la méthode printLongDescription() de la classe Room pour afficher ce que je souhaite afficher.

Exercice 7.33 : L'objectif de cet exercice est de calculer le prix total de l'inventaire et de montrer au joueur les objets qu'il possède dans son inventaire. En plus de corriger le problème d'affichage énoncé

à l'exercice précédent (manipulation de printLocationInfo()), j'ai rajouté une méthode affPoidsTot() dans la classe ItemList calculant le prix total de l'inventaire.

Exercice 7.34 :

```
public void ouvrir(final String pString){
    if (pString.equals("bourse")){
        this.aGui.println("Magnifique ! La bourse contient 10 roubles. Cette bourse est la chance de votre vie.");
        this.aArgent+=10;
        this.aListe.suppItem(pString);
    }
    else{
        this.aGui.println("Cet objet n'a pas la possibilité d'être ouvert.");
    }
}

}
```

Rajout d'un item permettant d'augmenter le nombre de rouble que le joueur possède. Cet objet à la capacité d'être « ouvert » (création d'une commande « ouvrir »). La procédure ouvrir() de la classe Player prend en paramètre le deuxième mot tapé par l'utilisateur (capture d'écran ci-haut pour montrer le fonctionnement de la méthode).

Exercice 7.42 : Le but de cet exercice est d'ajouter une limite de déplacement au joueur. Je fixe dans la classe Player car le nombre de déplacement est associé au joueur, un nouvel attribut de type int initialisé à 30 par exemple. Il suffit juste de créer la méthode compteDep()qui décrémente de 1 à chaque fois que je lui fais appel. Une fois que le nombre de déplacement à atteint 0 je désactive la zone de saisie de texte à la manière de la méthode endGame(). Je fais appel à compteDep() dans changerLaRoom() et dans back(). Il ne me reste plus qu'à afficher le nombre de déplacement dans PrintLocationInfo().

Exercice 7.42.2 : Je me contente de l'IHM actuelle.

Exercice 7.43-7.45 : Création d'une nouvelle classe Door contenant trois attributs : un item, et deux booléens. Le premier attribut, l'item, sera désigné comme l'item que le joueur doit avoir dans son inventaire pour déverrouiller la porte associée à l'item. Le premier booléen vaut true si la porte en question est verrouillée et false sinon. Le deuxième booléen sert à savoir si la porte est une TrapDoor, c'est-à-dire une porte que l'on peut passer dans un sens mais qui se verrouille automatiquement ensuite. Dans cette classe, je dois donc mettre un accesseur pour chacun des attributs et un modificateur setPorteVer(final boolean pPorteVer) pour le premier booléen aPorteVer qui sert à changer si la pièce vient à être déverrouillée. Si la porte est déverrouillée, elle le reste et vous pouvez

```
Door vDoor=this.aPlayer.getCurrentRoom().getDoor(vDirection);
try{if (this.aPlayer.getListe().getItem(vDoor.getCle().getNom())!=null){
        vDoor.setPorteVer(false);}
}
catch(java.lang.NullPointerException gaby){
}
if (vDoor.getPorteVer()==false){//si la porte n'est pas vérouillée on bouge
    if(vDoor.getTrap()==false){//si la porte n'est pas trap on bouge
        this.aPlayer.changerLaRoom(vNextRoom,false);
        this.aPlayer.printLocationInfo();

    }
    else{//Si la porte est trap, on bouge et on la vérouille ensuite
        this.aPlayer.changerLaRoom(vNextRoom,true);
        this.aPlayer.printLocationInfo();
        this.aGui.println("Ceci est une porte piégée, vous ne pourrez pas faire demi-tour.");

        vDoor.setPorteVer(true);
    }
}
else{//sinon on ne bouge pas
    if(vDoor.getCle().getNom()!=null){
    this.aGui.println("Cette porte est verouillée. Essayez de trouver l'item permettant de l'ouvrir.\n L'item est : "+vDoor.ge
    else{
        this.aGui.println("Cette porte est maintenant verrouillé et aucun item ne peut la dévérouiller. C'est une trap Door.")
```

déposer l'objet vous servant à la déverrouiller. Il ne me reste plus qu'à modifier les classe gameEngine, Room et Player. Dans Room on rajoute une méthode setDoor, associant une String direction à une porte de la même manière que setExit. Dans GameEngine je créé l'ensemble de toutes mes Door. Une pour deux pièces adjacentes. Je modifie goRoom de cette façon :

Avec l'affichage de l'item attendu pour déverrouiller la porte (ligne coupée sur la capture d'écran) : vDoor.getCle().getNom(). Vous remarquerez que l'appel de la méthode de la classe Player changerLaRoom a changé. En effet elle prend en paramètre un booléen supplémentaire permettant de réinitialiser la Stack si la porte est une trapDoor. Ainsi, on ne peut pas repasser la trapDoor en utilisant ma méthode back().

Exercice 7.44 : Création de la classe Beamer. Une capture d'écran vaut mieux qu'un long discours :

```
public class Beamer extends Item
{
    private Room aCharger;
    private boolean aEteCharge;

    /**
     * Constructeur d'objets de classe Beamer
     */
    public Beamer(final String pNom, final int pPrix, final String pDescription ){

        super(pNom,pPrix,pDescription);
        this.aEteCharge=false;
    }
    /**
     * Accesseur pour savoir si le beamer a été chargé
     * @return vrai s'il est chargé et false sinon
     */
    public boolean getEteCharge(){
        return this.aEteCharge;
    }
```

```
    /**
     * Accesseur de aCharger
     * @return Room qui a été chargée
     */
    public Room getCharger(){
        return this.aCharger;
    }
    /**
     * Un exemple de méthode - remplacez ce commentaire par le vôtre
     * @param pRoom qui mémorise la Room actuelle
     *
     */
    public void charger(final Room pRoom)
    {
        this.aCharger=pRoom;
        this.aEteCharge=true;

    }
```

Il faut ensuite créer le Beamer dans la classe GameEngine en faisant appel à la méthode setBeamer de la classe Room prenant, comme pour un item, en paramètre un nom, un prix, et une description. De la même manière que ma méthode setItem, setBeamer créé un Beamer à partir des paramètres et je l'ajoute à la aListe qui, je le rappelle est un attribut de type ItemList. Par la suite, je code deux nouvelles commandes. La commande « charger » et la commande « tirer ». Ce qui donne maintenant dans ma méthode interpretCommand() de GameEngine :

```
else if (vCommandWord.equals("charger")){
    try{this.aPlayer.getListe().getBeamer(vCommand.getSecondWord()).charger(this.aPlayer.getCurrentRoom());}
    catch(java.lang.NullPointerException gaby){this.aGui.println ("Que voulez-vous charger ? Rentrer un téléporteur valide.");}}
else if (vCommandWord.equals("tirer")){

    try{this.aPlayer.seTeleporter(vCommand.getSecondWord());}
    catch(java.lang.NullPointerException gaby){this.aGui.println("Avec quoi voulez-vous tirer ? Rentrer un téléporteur valide.");}
```

Dans itemList, comme le montre cette capture j'ai rajouté la méthode getBeamer qui prend en paramètre une String, le nom de l'item. Je vérifie que l'item est de Classe Beamer avec l'aide de la commande « instanceof » et je retourne l'Item, converti en Beamer : « return (Beamer) aListeItem.get(pString) ». Sinon je retourne null, NullPointerException traitée dans mon try catch de ma méthode interpretCommand(). Voilà ce que me donne ma méthode seTeleporter auquel je fais appel quand le joueur « tire ». On supprime le téléporteur après l'avoir utilisé, et on réinitialise la Stack, pour éviter que le joueur retourne à la salle précédente avec back().

```
/**
 * méthode permettant de "tirer"
 * @param pNom, le nom de l'item de type Beamer
 */
public void seTeleporter(final String pNom){

    this.changerLaRoom(aListe.getBeamer(pNom).getCharger(),true);

    this.aListe.suppItem(pNom);
    this.printLocationInfo();
}
```

III. Mode d'emploi (si nécessaire, instructions d'installation ou pour démarrer le jeu)

Après avoir téléchargé l'archive ci-jointe, ouvrez BlueJ. Dans Files, cliquez sur Open jar/zip project et séléctionnez l'archive. Sur la case Game, clique droit, new game(). Le jeu se lance alors.

N'hésitez pas à taper la commande help. Le fichier de la musique sera déjà dans le dossier du jeu ainsi que les fichiers de commandes. Relisez la partie I.F) pour bien comprendre tous les mécanismes du jeu.

Sur jNews, par soucis de stockage, nous avons remplacé les gifs animés par des images fixes et nous avons enlevé la musique pour que le projet respecte la taille de 20 Mo. Autre précision concernant le projet. Les fichiers test des différents parcours sont dans le dossier. Cependant pour le parcourscomplet, vous faites intervenir l'aléatoire du jeu de pierre feuille ciseau. Il se peut que vous vous arrêtiez au vestibule pour ça, que vous ne finissiez pas le jeu parce que vous avez misé trop de roubles. Essayer de relancer le jeu et la commande « test parcourscomplet » pour tenter de finir le jeu.

IV. Déclaration obligatoire anti-plagiat (préciser toutes les parties de code que vous n'avez pas écrites vous-même et citez la source, sauf les fichiers zuul-*.jar qui sont fournis évidemment)

Pour mon audio, j'ai pris le code sur internet sur openclassrooms : https://openclassrooms.com/forum/sujet/jouer-de-la-musique-96546

Sous réserve des droits de « La marche impériale » (titre de la musique). Et des images que j'ai enregistrée depuis le site gyphy.com.

Pour le reste, je me suis aidé de toutes les consignes présentes sur icampus.