Rapport IPO

Lien du site : https://perso.esiee.fr/~peresset

1.A) Auteur

Théo Péresse-Gourbil, étudiant Esiee Paris E1 promo 2023.

1.B) Thème

Dans un **temple Jedi**, toi, un jeune Padwan, dois achever sa formation en créant son propre sabre laser.

1.C) Scenario

Au **temple Jedi** de Coruscant, tu devras, pour finir ta formation Jedi, construire ton propre sabre lasert, arme légendaire des Jedi et craint à travers toute la galaxie. Pour cela, tu devras recupérer les pièces manquantes en parcourant le temple à la recherche de ces pièces nécessaires à sa création.

1.D) Plan

1.E) Scénario détaillé

Un jeune Jedi doit passer par une phase d'apprentissage. Il devient alors un Padawan. Afin de terminer sa formation, il doit rassembler 5 objets et construire son sabre lasert dans l'armurerie du temple. Pour récupéré ces 5 objets, il devra parcourir seul le temple Jedi et se rendre dans les bonnes pièces.

1.F) Détail des lieux, items, personnages...

Détail des lieux :

- Entrée gardée : Cette salle est la salle la plus au sud du temple. Elle est gardée par 2 gardes ainsi qu'un garde d'élite.
- Salle de la fontaine : Cette salle, située à l'ouest du couloir, protège une fontaine dont l'eau est sacrée et qui fournit tout le temple en eau.
- Salle de combat : Dans cette salle située à l'est du couloir, vous pourrez trouver le Maitre Jedi Kannan Jarrus, qui vous expliquera comment construire votre sabre laser. C'est aussi la salle de départ du jeu.

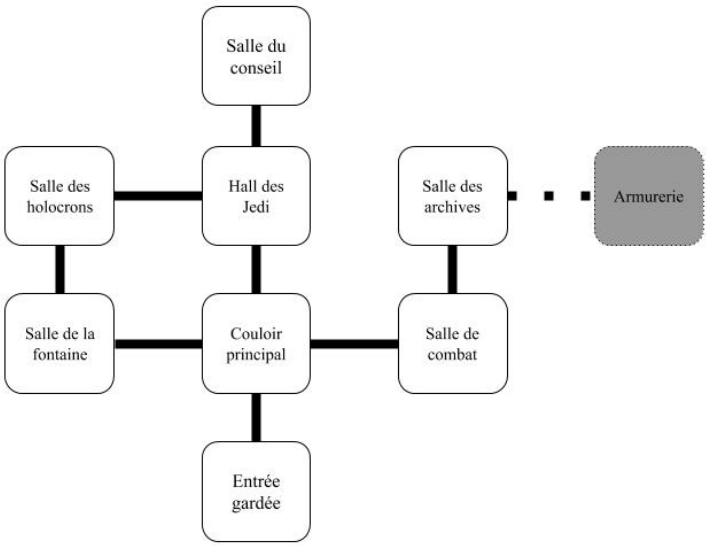

Figure 1: Plan du jeu

- Salle des holocrons : Cette salle protégée au nord de la salle de la fontaine abrite les holocrons, objets de la Force qui abritent les secrets de tous les jedi. Vous y trouverez le gardien, le maitre jedi Cin Darllign.
- Salle des archives : Au nord de la salle de combat, on peut se rendre dans la salle des archives, qui abrite de nombreux savoirs qui pourraient vous être utile dans la suite de votre aventure.
- Hall des Jedi : Le grand hall des Jedi est situé au nord du couloir et fait le lien entre la salle du conseil Jedi et le couloir.
- Salle du conseil : C'est dans cette salle que les maitres jedi les plus puissants méditent et débatent du futur du temple. Leur aide vous sera indispensable.
- Armurerie : L'Armurerie est une salle vérouillée. Une fois ouverte, vous pourrez construire votre sabre et ainsi finir le jeu.

Détail des items :

- Le cristal de Kyber : Le cristal permet de concentrer l'energie brut en un rayon, ce qui en fait un composant essentiel. C'est aussi lui qui détermine la couleur du sabre. Allez convaincre le conseil Jedi de vous donner un cristal. Vous devrez répondre a une énigme afin de savoir quelle couleur de sabre sera la plus adapté pour vous.
- La cellule d'energie : La cellule d'energie permet d'alimenter la lame du sabre. Allez parler au garde qui vous demandera un verre d'eau (à aller chercher dans la salle de la fontaine). En échange, il vous donnera une cellule d'energie récupéré sur un vaisseau spatial abandonné non loin du temple.
- L'emeteur : L'emetteur permet de concenter la puissance de la cellule en une lame droite. Vous pourrez trouver cet objet dans le hall des Jedi, dans une lampe a énergie.
- La lentille : La lentille permet de régler la taille et l'épaisseur de la lame. Allez parler au Maitre Cin Drallig dans la salle des holocorons qui vous dira qu'il en a vu une vers l'est du temple. Allez dans la salle des archives pour récuperer une lentille, qui est stockée dans un vieux contenaire
- Le verre d'eau : Un garde vous demander surement un verre d'eau. Vous pourrez remplir ce verre dans la salle de la fontaine.

Détail des personnages :

- Le conseil Jedi : Haute autorité du temple, le conseil Jedi vous a mis à l'épereuve pour finir votre formation au temple. Il vous faudra passer un test afin qu'il vous donne un cristal de Kyber.
- Garde d'élite : Ce garde d'élite, choisis sur au vollet, garde l'entrée principale du temple. Il ne peut quitter son poste, même pour prendre une pause et boire un verre.

1.G) Situations gagnantes et perdantes

Situations gagnantes :

Pour gagner et ainsi finir le jeu, tu devras récuperer le plus rapidement possible les 4 objets suivants : - Cristal de Kyber : tu devras aller voir le conseil jedi. Ils te soumetteront une enigme en rapport avec l'univers StarWars. Si tu réponds correctement, ils te donneront un cristal de Kyber. - Cellule d'energie : tu devras aller voir le garde à la sortie du temple. Il te demandera un verre d'eau. Tu te rendras alors dans la salle de la fontaine où tu pourras récuperer un verre qui sera apparu. Tu lui rapporteras et il te donneras l'objet. - Emetteur : Tu trouveras cet objet dans le hall des Jedi, qui sert d'alimentation à une lampe. - Lentille convergente : Vas parler au maitre jedi qui t'attend dans la salle des holocrons. Après lui avoir parler, tu devras te rendre dans les archives où la piece aura apparu. Une fois tous les objets réunis, une salle se débloquera. Une fois dedans, tu devras assembler ton sabre et le jeu se terminera alors.

Situations perdantes :

Pour perdre dans le jeu, il existe plusieurs solutions : - trop de mouvements effécutés qui montrerais que tu n'es pas près;

1.H) Enigme, mini-jeux, combat

Il n'y a pas d'enigmes mais des mécanismes avec les PNJ pour récupere les objets.

1.I) Commantaires

2) Réponse aux exercices

Exercice 7.0 :

Pour la création du site web, j'au utilisé un CMS disponible sur internet utilisant du html, du css et du js. J'ai alors remodelé ce CMS pour qu'il convienne à mes envies pour le site de Zuul. J'ai alors redéfinis les droits pour le repertoire public_html afin qu'il soit accessible depuis un autre pc.

Exercice 7.4 :

Pour supprimer v1, j'ai copier les classes au meme niveau que v1. Puis j'ai supprimé v1, ce qui fait que mes classes de Zuul se retrouvent à la racine de

mon projet.

Exercice 7.5 :

Dans cet exercice, on cherche a éviter la duplication de code présente dans les méthodes printWelcome et goRoom de la classe Game. En effet, ces deux méthodes affichent toutes les deux les differentes sorties valides de la piece courrante. Pour eviter cette duplication, on créer une troisième méthode, printLocationInfos, qui a pour but d'afficher les sorties disponibles.

Exercice 7.6 :

On observe actuellement uun gros couplage de code, notement pour definir les sorties des differentes pieces. Nous allons alors créer une accesseur getExit dans la classe Room qui renvoi les differentes sorties disponibles. On peut alors remplacer un gros morceau de code par simplement la ligne suivante :

```
vNextRoom = aCurrentRoom.getExit("East");
```

Ainsi, on a plus de couplage à ce niveau grace à l'utilisation de cette méthode.

Exercice 7.7 :

Nous voulons maintenant afficher sur la console de sortie les differentes sorties disponibles pour la pièce courante. Nous allons donc créer une nouvelle méthode getExitString. Nous allons maintenant appeler cette méthode à chaque changement de pièce afin d'afficher les sorties disponibles.

Exercice 7.8 :

Nous allons maintenant créer une HashMap afin de pouvoir moduler nos sorties et eviter les null à chaque sorties non prises par une piece, et ma duplication de code, ainsi que le code inutile. Il faut dans un premier temps importer la HashMap avec :

```
import java.util.HashMap;
```

Il faut ensuite l'initialier en créant la HashMap comme indiqué dans le livre, avant d'initialier les sorties dans setExits avec la commande :

```
nom.put(String,Room);
```

Exercice 7.8.1 :

Il nous est demandé maintenant de déclarer une sortie up ou down. On ajoute donc une sortie à une salle comme ceci :

```
nom.put("down",vConseil);
```

Exercice 7.9 :

On doit maintenant changer la méthode getExitString afin d'eviter la duplication de code avec tous les if. On return donc une string que l'on affichera plus tard.

Exercice 7.10 (optionnel) :

La méthode getExitString revoie une string composée de toutes les clées liées à la piéce et efféctue une concaténation grâce à la boucle for. On return alors la String final.

Exercice 7.10.1 :

On doit mettre à jour la javadoc avec les balises /** à l'aide d'une déscription et de tags comme @param, @return ... Par la suite, on génère cette Javadoc avec la ligne bash dans un terminal linux :

```
javadoc -d userdoc -author -version *.java
javadoc -d progdoc -author -version -private -linksource *.java
```

Exercice 7.10.2 :

On génere la javadoc automatiquement grace à BlueJ.

Exercice 7.11 :

On veut maintenant anticiper la suite du jeu (objets ...) et on va donc creer une méthode getLongDescription qui évitera un peu la duplication de code mais permettera une plus grands modularité du code. On va donc par la suite appeler cette methode dans la classe Game.

Exercice 7.14 :

On veut désormais ajouter au jeu de nouvelles commandes. On ajoute alors dans la liste des commandes possibles “look”. De plus, on doit aussi la rajouter

dans Game. On affichera aussi la méthode getLongDescription apres l'appel de la méthode look() que l'on créer dans la class Game

Exercice 7.15 :

On fait de même pour la commande "eat".

Exercice 7.16 :

On remarque que PrintHelp n'est pas à jour. En effet, elle affiche les commandes disponibles mais de facon stricte. On veut pouvoir ajouter automatiquemnt les commandes. On va créer une méthode ShowAll dans la class CommandWord qui affiche toutes les commandes valides dans Game. Or, ce tableau n'est pas accessible depuis cette classe. Il faut donc créer une méthode dans Parser en la faisant renvoyer une String qui sera par la suite affichée.

Exercice 7.17 :

Pour ajouter une nouvelle commande, il faut en plus de l'ajouter au tableau, modifier la class Game. En effet, il faut, au meme titre que eat et look, définir ce que fait cette commande et retourner false à la fin pour eviter la fin du jeu.

Exercice 7.18 :

On veut éviter le couplage implicite dans la méthode showAll. On modifie alors cette méthode pour qu'elle n'affiche plus mais retourne la String qui nous intéresse, que l'on récupere dans printHelp afin de l'afficher, ce qui évite le couplage implicite.

Exercice 7.18.1 :

En comparant mon projet au projet zuul-better, les deux se ressemblent beaucoup

Exercice 7.18.3 :

J'ai pris mes images sur un site de jeu vidéo StarWars, "Star Wars, the old republic".

Exercice 7.18.4 :

J'ai choisis le nom de mon jeu en foncion de l'univers : Zuul-wars.

Exercice 7.18.5 :

On creer une HashMap de Room et leur description afin de les retrouver plus tard si besoin

`private HashMap<String,Room> aRooms;` et on ajoute dans le constructeur de game : `aRooms = new HashMap<String,Room>;` Et on ajoute les rooms et les string avec : `aRooms.put("Description",Room);`

Exercice 7.18.6 :

On creer les classes qui manquent, et on renomme les variables et attributs pour plus de coherence avec la reste du projet, ainsi que les spécificitées liées à mon jeu comme les images, textes, descriptions, salles ...

Exercice 7.18.7 :

adddActionListener permet de detecter un clic de souris ou enfoncement de la touche Enter. On importe le paquet java.awt.event.*; On déclare qu'on utilisera aussi des interfaces d'écoutes avec : `implements ActionListener`. Chaque 'auditeur' envoi ses informations à differentes méthodes pour traiter les action. ActionListener envoie ses informations à la méthode actionPerformed( ). actionPerformed() permet alors de gerer le traitement des evenements de l'actionListener.

Exercice 7.18.8 :

Pour creer un boutton, on import la classe JButton avec : `import javax.swing.*;` pour avoir les bouttons, les frames, les panels ... On ajoute un attribut par boutton : `private JButton aButtonNorth;` On les initialise dans le contructeur de UserInterface : `this.aButtonNorth = new JButton("go nord");` J'ai ensuite crée un panel complet position à l'est pour tous mes bouttons de directions : `JPanel vPanelGo = new JPanel(); vPanel.add(vPanelGo, BorderLayout.EAST); vPanelGo.setLayout( new BorderLayout());` On ajoute ce bouton au panel avec : `vPanelGo.add(this.aButtonNorth, BorderLayout.NORTH);` Mainetenat qu'on a notre boutton, on veut pouvoir cliquer dessus. On ajoute donc un actionListener : `this.aButtonNorth.addActionListener(this);` Cependant, si on ne traite pas la source du clic, il ne se passera rien. On ajoute donc cette ligne dans ActionPerfomed :

```
if (vSource == aButtonNorth) {
            aEngine.interpretCommand("go nord");
        }
```

Exercice A savoir expliquer :

ActionListener : Permet de savoir si clic de souris à été fait ou la touche Enter appuyée. **addActionListener()** : ActionListener envoie des événements à une méthode nommée actionPerformed() : `public void actionPerformed( final ActionEvent pE )` **ActionEvent** : Pour identifier le composant qui a généré l'événement **actionPerformed()** : L'interface ActionListener envoie des événements à la méthode nommée actionPerformed() **getActionCommand()** : Renvoie la String correspondant au texte sur le bouton **getSource()** : Renvoie l'objet qui a généré l'évenement.

Exercice 7.19.2 :

Pour déplacer rapidement toutes les images, on saisis les commandes suivant en étant dans le dossier du projet : `mkdir Images` `mv *.jpg Images`

Exercice 7.20 :

On créer une nouvelle classe Item avec un attribut poids, une description et un nom. On ajoute ensuite les methodes classiques de constructeur, getters et setters. On en profite pour faire une hashmap pour réuperer toutes les infos dont on aura besoin plus tard. On modifie aussi le constructeur de game engine pour pouvoir inculre un objet dans chaque room. On initialisie les items dans le contructeur de game engine aussi. On ajoute à getLongDescription une fonction pour pouvoir aficher l'item disponible dans la salle.

Exercice 7.21 :

Les informations qui relatives aux Items doivent être produites par la classe Item. Les informations relatives aux items presents dans une room doivent être produites par la classe Room. Touces ces informations doivent être affichées par la classe GameEngine.

Exercice 7.22 :

Pour avoir un nombre illimité d'item, on peut fare la meme chose que pour les sorties, c'est a dire faire une hashmap entre les room et les objet, et apres les ajouter un a un. On peut ainsi attribuer plusisuers items à chaque room. Pour retourner la description de chaque item, on peut faire une méthode qui renvoie chaque description de tous les items presents dans cette salle, avec :

```
public String getItemName(){
        StringBuilder vReturnString = new StringBuilder( "" );
```

```
        for ( String vS : aItemHM.keySet() )vReturnString.append( " " + vS );
        return vReturnString.toString();
    }
```

On appelle alors cette methode dans getLongDescription, ce qui permet d'afficher à chaque changement de salle les objets disponible dans la salle.

Exercice 7.22.2 :

On creer une hashmap qui contiendra les rooms et les items dans Item, et on fait une methode publique pour ajouter des elements. On crer les items dans createRoom avec le contructeur d'item, et de la même facon qu'on a ajouté les sorties, on ajoutes les objets dans la hashmap. On ajoute une fonction qui renoie une string qui continet les items dans une room, qu'on affiche ensuite dans gameEngine.

Exercice 7.23 :

Pour faire une commande back, il faut dans un premier temps ajouter cette commande dans le tableau de commande, ainsi que dans la commande interpretCommand afin qu'elle y soit interprétée. Concernant la commande, il faut stocker la précédente piece dans une variable avant chaque changement de Romm. Cela se fait dans goRoom. Par la suite, on doit, lors de l'appel de back, on change aCurrentRoom et on refait l'affichage de la description et l'image.

Exercice 7.26 :

Le prombleme avec la commande back est qu'elle ne peut marcher qu'une fois. En effet, chaque room n'est stockée qu'une fois. On resoud cela en créant une stack de Room dans laquelle on stock toutes les salles visitées. On créer un atribut aStackRoom avec : `private Stack<Room> aStackRoom;`, qu'on initialise dans le constructeur `this.aStackRoom = new Stack<Room>();`. Ensuite, on ajoute la room courante avant le changement de room dans goRoom avec :

```
this.aStackRoom.push(aCurrentRoom);
```

Et dans back, on veut en réalité enlever une piece de cette pile, bien evidement en verifiant si la pile n'est pas vide. Dans back, on fait :

```
this.aCurrentRoom = this.aStackRoom.pop();
```

Et on refait l'affichage de la piece courrante (image et description).

A savoir expliquer :

Stack : Une stack est une collection que l'on peut rapporcher d'une pile d'assiette. On peut ajouter des objets sur le dessus de la pile, mais pas au milieu, tout comme on peut enlever l'objet qui se trouve sur le dessus. **push() :** push sur une stack permet d'ajouter l'objet passé en parametre sur le dessus de la Stack. **pop() :** pop sur une Stack permet de retirer et de renvoyer l'objet qui se trouvait sur le dessus de la Stack. **empty() :** la commande empty est une fonction boolenne qui revoir vrai si la stack en question est vide et faux si la stack contient au moins un element. **peek() :** la commande peek permet de renvoyer l'objet sur le dessus de la Stack sans la supprimer de la Stack

Exercice 7.28.1 :

On creer une methode qui prend en parametre une string correspondant au chemin du fichier a tester. Ensuite, on reprend le meme schema de methode que pour le TP des exceptions, sauf qu'au lieu d'afficher les lignes on appelle interprete command sur nextLine, ce qui nous donne :

```
private void test (final String pFichier){
        Scanner vSc;
        String vFichier = pFichier;
        if (! vFichier.endsWith(".txt")) vFichier += ".txt";
        if (! vFichier.startsWith("test/")) vFichier = "test/"+vFichier;
        try {
            InputStream vIs = getClass().getResourceAsStream(vFichier);
            vSc = new Scanner( vIs );

            while ( vSc.hasNextLine() ) {
                String vLigne = vSc.nextLine();
                this.interpretCommand(vLigne);
            } // while()

        } // try
        catch ( final Exception pE) {
            this.aGui.println("Erreur dans le fichier" + pE.getMessage());
        }// catch

    }//test
```

Pour ajouter une nouvelle commande, comme pour back, on doit ajouter test dans les mots autorisés, ainsi que dans interpret command. J'ai deplacé les fichier.txt dans un dossier test afin de faciliter leur modification, et ai prévu cela dans mon code. Mon premier fichier court s'appelle court.txt Le second fichier qui test toutes les commandes s'appelle command.txt Le fichier qui permet de visiter toutes les salles du jeu s'appelle long.txt

Exercice 7.29 :

On creer une classe Player qui comport les attributs suivants : aCurrentRoom, aGui, aNom, une Stack et un poid max. On fait un constructeur classique, pour initialiser les attributs. On ajoutes les accesseurs et modificateurs classiques. Ensuite, on supprime l'attribut aCurrentRoom de la classe GameEngine et on passe dans Player toutes les methodes qui ne compilent plus, puis on ajoute un atribut aPlayer à GameEngine. Ensuite, on corrige es erreurs, notement en remplacant this.methode par aPlayer.methode, ou encore aPlayer.get.methode(). On obtient à la fin :

```
import java.util.*;

public class Player
{
    private Room aCurrentRoom;
    private String aNom;
    private UserInterface aGui;
    private int aPoidMAX = 100;
    private Stack<Room> aStackRoom;
    /**
     * constructeur de Player
     * @param String nom du joueur
     * @param Room room de départ
     */

    public Player (final String pNom , final Room pCurrentRoom ){
        this.aCurrentRoom = pCurrentRoom;
        this.aNom = pNom;
        aStackRoom = new Stack<Room>();

    }

    /**
     * Constructeur de gui
     * @param GUI
     */
    public void setGui(final UserInterface pUserInterface)
    {
        this.aGui = pUserInterface;
    }

    /**
     * retourne la room actuelle
     * @return Room aCurrentRoom
     */
```

```java
public Room getCurrentRoom()
{
    return this.aCurrentRoom;
}

/**
 * Affiche une description du lieu courant
 */
public void look(){
    printLocationInfo();
}

/**
 * manger, pas utile dans ce jeu mais demandé
 */
public void eat(){
    aGui.println("You have eaten now and you are not hungry any more.");
}

/**
 * Print infos about wher you are and the exits available
 */
private void printLocationInfo(){
    aGui.println(aCurrentRoom.getLongDescription());
}//printLocationInfo()

/**
 * retourne le nom du jour actuel
 * @return nom
 */
public String getName(){
    return this.aNom;
}

/**
 * Permet de revenir en arriere au moyen d'une stack
 */

public void back(){
    if ( !this.aStackRoom.empty()){
        this.aCurrentRoom = this.aStackRoom.pop();
        changeRoom(aCurrentRoom);
    } else aGui.println("Vous ne pouvez pas aller en arrière");
}

/**
```

```
     * Permet de retourner la stack de room
     * @return stack room
     */
    public Stack<Room> getStackRoom(){
        return this.aStackRoom;
    }

    /**
     * Change de room
     * @param la room ou aller
     */

    public void changeRoom(final Room pRoom)
    {
        this.aCurrentRoom = pRoom;
        this.aGui.println(this.aCurrentRoom.getLongDescription());
        if(this.aCurrentRoom.getImageName() != null)
            this.aGui.showImage(this.aCurrentRoom.getImageName());
    }

    /**
     * Print the welcome tips
     */
    public void printWelcome(){
        aGui.println("Player : " + this.getName());
        aGui.println("Bienvenue dans le jeu d'aventure StarWars !");
        aGui.println("Ta formation est bientôt terminé jeune Padawan. Récpuère les objets né
        aGui.println("Tappe help si tu as besoin d'aide !");
        aGui.println("\n");
        aGui.println(aCurrentRoom.getLongDescription());
        aGui.showImage(aCurrentRoom.getImageName());
    }//printWelcome()
}
```

Exercice 7.30 :

Pour la méthode `drop`, on créer un attribut Item qui correspond à l'item que la joueur porte. On verifie que l'objet porté n'est pas nul. Si c'est le cas, alors on ajoute l'objet porté à la piece avec la methode addItem, puis on met à nul l'objet porté. On fait l'inverse pour drop.

Exercice 7.31 :

On peut faire une HashMap inventaire qui contiendra les objets pris par le joueur dans la classe Player car c'est elle qui gère take, drop et les objets protés. On passe à take une command dont on recupere le second mot, avant de récuperer l'objet associé à cette description.

Exercice 7.31.1 :

On creer une nouvelle classe avec deux attributs : une HashMap qui représente les items portés par le joueur et le poids total de l'inventaire. On ajoute des fonctions pour ajouter et enlever les items de la HashMap, ainsi que differents accesseurs pour récuperer la description de l'inventaire, le poids ... On modifie alors player pour que les methodes take et drop remplissent ou vident la hashMap. On ajoute aussi la possiblilité d'afficher l'inventaire. Voici la methode take modifiée :

```
public void take (final Command pCommand){
        String vDescription = pCommand.getSecondWord();
        Item vItem = this.aCurrentRoom.getItemHM(vDescription);
        if ( vItem == null ) {
            this.aGui.println("Cet objet n'est pas la !");
            return;
        }
        if ( this.aInventaire.getPoids()+vItem.getPoids() > this.aPoidsMax){
            this.aGui.println("Votre inventaire est plein. Il faut lacher des objets pour s'
        }
        else {
            this.aInventaire.add(vItem,vDescription);
            this.aCurrentRoom.rmItemHM(vDescription);
            this.aGui.println("Objets présents dans la pièce : "+ this.aCurrentRoom.getItemS
            this.aInventaire.addPoids(vItem.getPoids());
        }
        this.aGui.println(this.aInventaire.getItemList());
        this.aGui.println(this.aCurrentRoom.getLongDescription());
    }
```

Exercice 7.32 :

On creer un attribut aPoidsMax dans Player, que l'on initialisie directement à sa valeur. On modifie le constructeur d'item pour que chaque item ai son poids. Dans take, on ajoute un test pour verifier si le poids porté plus le poids de l'item que l'on veut prendre est inferieur au poids max.

Exercice 7.33 :

On ajoute une commande (comme avant). Dans cette commande, on affiche la String renvoyé par une methode de item list qui donne la liste des objets de l'inventaire et le poids total.

Exercice 7.34 :

On creer un nouvel item, magiccookie, que l'on place dans une room. On modifie ensuite la methode eat comme take et drop pour pouvoir prendre un second mot. On recupere ensuite l'item correspondant dans la HashMap d'item. Si il est null, c'est que soit l'item n'existe pas soit il n'est pas dans cette salle. Dans ce cas, on return. Sinon, on le compare à l'objet MagicCookie et alors, on peut doubler l'inventaire avant de detruire le makicCookie. Voici le résultat :

```
public void eat( final Command pCommand){
        String vDescription = pCommand.getSecondWord();
        Item vItem = this.aInventaire.getItem(vDescription);
        if (vItem == null) {
            this.aGui.println("Eat quoi ?");
            return;
        }
        if ( vDescription.equals("MagicCookie") && this.aInventaire.getItemListHM().contains
            this.aPoidsMax = this.aPoidsMax * 2;
            this.aGui.println("Vous avez mangé un cookie magique. Votre inventaire vient de
            this.aGui.println(this.aInventaire.showPoids() + "\n Le poids maximal est de " +
            this.aInventaire.remove(vItem, vDescription);
            this.aInventaire.rmPoids(vItem.getPoids());
        } else this.aGui.println("Vous ne pouvez pas manger cela !");
    }
```

Exercice 7.34.1 :

on remet à jour le fichier test en ajoutant la prise d'un magic cookie notament.

Exercice 7.32.2 :

On regenere la javadoc avec les deux commandes deja détaillées plus haut.

Exercice 7.42 :

On ajoute une limite sur le nombre de déplacement possible du joueur. On implémente cela avec un attribut de type entier qui vaut 30 par exemple. A

chaque fois qu'un "ChangeRoom" se passe bien, on enleve un si il n'est pas nul, et si il est nul, alors on arrete le jeu. On ne met cette fonctionalité dans Player évidamnt car chaque player a un nombre de déplacement personel. De plus, pour que cela fonctionne avec Back, il faut le mettre dans changeRoom.

Exercice 7.42.2 :

J'ai décidé de ne pas toucher à l'interface graphique donc cet exercice n'est pas traité.

Exercice 7.43 :

Pour cet exercice, on creer une nouvelle classe Door, car on traitera en meme temps les TrapDoors et les LockedDoor. Cette classe comporte 3 attributs qui sont des boolean qui correspondent à isLocked, isStrap, et isGoodDirection (dans le cas d'une trapdoor uniquement). On creer alors des doors aux endrois voulu dans le game engine comme pour les objets par exemple. On fait aussi une hashMap afin d'avoir accès facilement aux doors depuis la room.

```
public class Door
{
    // ## Attributs ##
    private boolean isLocked;
    private boolean isTrap;
    private boolean isGoodGirection;


    // ## Constructor ##

    /**
     * constructeur naturel
     * @param pLocked boolean qui dit si la door est locked
     * @param pTrap boolean qui dit si la door est trap
     * @param pGoodDirection (si trop only) boolean qui dit si bonne direction de passage
     */
    public Door (final boolean pLocked , final boolean pTrap , final boolean pGoodDirection)
        this.isLocked = pLocked;
        this.isTrap = pTrap;
        this.isGoodGirection = pGoodDirection;
    }


    // ## Accesseur ##

    /**
```

```
     * Accesseur de lock
     * @return boolean si la door est locked
     */
    public boolean isLocked(){
        return this.isLocked;
    }

    /**
     * Accesseur de trap
     * @return boolean si la door est trap
     */
    public boolean isTrap(){
        return this.isTrap;
    }

    /**
     * Accesseur de cango
     * @return boolean si la door trap peut etre franchie dans ce sens
     */
    public boolean canGo(){
        return this.isGoodGirection;
    }


    // ## Modificateurs ##

    /**
     * setteur de locked. Permet de deverouiller une piece
     * @param pLocked permet de debloquer une door
     */
    public void setLocked (final boolean pLocked){
        this.isLocked = pLocked;
    }

    /**
     * setteur de cango
     * @param pGoodDirection boolean qui permet de set gooddirection
     */
    public void setCanGo(final boolean pGoodDirection){
        this.isGoodGirection = pGoodDirection;
    }
}
```

On modifie aussi le goRoom pour interagir avec des doors. Une trap door ne peut etre franchie que dans un sens, et la command Back ne doit pas pouvoir marcher. Pour se faire, on verifie à chaque fois dans quel cas on est, puis on accepte ou

non e changeRoom. Pour la LockedDoor, il faut verifier si la condition pour la dverouiller est remplie. Dans mon cas, il s'agit d'avoir tous les items sur sois. Voici la méthode goROom modifiée :

```
private void goRoom ( final Command pCommand){

        if (!  pCommand.hasSecondWord()){
            aGui.println("Go where ?");
            return;
        } //mot pas bon apres le go
        else {
            Room vNextRoom = null;
            String vDirection = pCommand.getSecondWord();
            vNextRoom = aPlayer.getCurrentRoom().getExit(vDirection);

            if (vNextRoom == null){
                aGui.println("There is no door !");
                return;
            }
            this.aPlayer.getStackRoom().push(this.aPlayer.getCurrentRoom());
            Door vDoor = this.aPlayer.getCurrentRoom().getDoor(vDirection);
            if ( vDoor != null){ // si c'est une porte spéciale
                if (vDoor.isTrap()){ // si c'est une trap
                    if (! vDoor.canGo()){
                        this.aGui.println("It' a trap ! Vous ne pouvez pas passer par une po
                        return;
                    }
                    else {
                        this.aPlayer.clearStack();
                    }
                }
                else if (vDoor.isLocked()){
                    HashMap vHMobj = this.aPlayer.getInventaire().getItemListHM();//hash map
                    if (vHMobj.containsKey("Crystal") && vHMobj.containsKey("Cellule") && vH
                        vDoor.setLocked(false);
                        this.aGui.println("Vous avez trouvé l'armurerie secrète");
                    }
                    else {
                        this.aGui.println("Vous ne pouvez pas ouvrir cette porte ...");
                        return;
                    }
                }
            }


            this.aPlayer.changeRoom(vNextRoom);
```

```
        }
        if (this.aAudio.isFinished()){
            this.aAudio.playSound("son/imperial.wav");
        }

    }//goRoom(.)
```

Exercice 7.44 :

Je n'ai pas réussi à implémenter le beamer. J'ai creer la classe correspondante mais pas le mécanisme.

Exercice 7.45 :

Voir exercice 7.43.

Exercice 7.45.1 :

On regenere les fichiers test, notemant avec la possiblilté de gagner qui a été implémenter.

Exercice 7.45.2 :

On regener la javadoc avec les deux commandes comme précedement.

Exercice 7.46 à 7.48 n'ont pas été traité à cause du temps

Exercice 7.48 :

Pour cet exercice, on va se calcer sur l'implémentation des Items dans le jeu. On va donc creer une class PNJ avec trois attributs : un item qui correspond à l'item que va drop le PNJ une fois qu'on lui a parlé, un nom qui sera le nom affiché à l'ecran et qui servira à la trouver dans la future HashMap, et une Room qui correspond à la room dans lequelle il se trouve. On ajoutes les 3 accesseurs. Dans la classe Room, on creer un attribut HashMap de PNJ. ON ajoute les methodes pour modifier la HashMap. Dans GameEngine, on creer les PNJ, qu'on attribut ensuite à des rooms grace à la hashmap. On ajoute ensuite dans Player la methode pour parler avec les PNJ :

```
public void parler (final Command pCommand){
        String vS = pCommand.getSecondWord();
        PNJ vPNG = this.aCurrentRoom.getPNJ(vS);
        if (vPNG == null){
            this.aGui.println("Cette personne n'est pas ici");
            return;
        }

        if (vPNG == this.aCurrentRoom.getPNJ("Jedi")){
            if (!aParleJedi) {
                this.aGui.println("Bonjour jeune Padawan. Pour contruire ton sabre, tu aura
                this.aCurrentRoom.addItem("Crystal",vPNG.getItem());
                this.aGui.println(this.aCurrentRoom.getLongDescription());
                this.aParleJedi = true;
            } else {
                this.aGui.println("Tu es prometteur, jeune Padawan, mais tu as la memoire bi
            }
            return;
        }

        if (vPNG == this.aCurrentRoom.getPNJ("Garde")){
            if (! this.aInventaire.getItemListHM().containsKey("Verre")){
                this.aGui.println("Va me chercher une verre d'eau je meurs de soif");
                return;
            }
            if (!aParleGarde){
                this.aGui.println("Merci garcon. Tiens prend cette cellule d'energie que j'a
                this.aCurrentRoom.addItem("Cellule",vPNG.getItem());
                this.aInventaire.rmPoids(this.aInventaire.getItem("Verre").getPoids());
                this.aInventaire.remove(this.aInventaire.getItem("Verre"), "Verre");
                this.aParleGarde = true;
            } else {
                this.aGui.println("Hey ! que fais tu la ! tu n'a rien à faire ici !");
            }
        }
    }
```

Fonctionnalitée supplémentaire :

J'ai ajouté de la musique à mon jeu, avec la classe audio qui me permet de gérer le lancement ou l'arrêt de la lecture d'un fichier audio en .wav. J'ai ajouté une fonctionnalité permettant de demander le nom du joueur au début de la partie.

3) Mode d'emploi

Apres 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(). Un carré rouge apparait alors en bas à gauche de l'écran. Clique droit et void play() pour lancer le jeu. ## 4) Déclaration anti-plagiat

Code :

Tout le code non fourni par le livre Zuul-Bad et donné dans les PDF à été écrit par moi-même. La classe qui gere le son à été réalisée en collaboration avec Quentin Martins ainsi qu'internet Le site unternet est un template HTML/CSS/JavaScript remanié par moi meme afin de correspondre à mes besoins

Son :

La bande son du jeu qui n'est pas encore implémenté : www.youtube.com/watch?v=D0ZQPqeJkk, de Coltsrock56 le 21 juil. 2012

Images

Les images du jeu qui ne sont pas encore visibles : - www.game-guide.fr/98292-swtor-pvf-academie-des-jedi-disten - www.anakinworld.com/encyclopedie/temple-jedi-de-tython - https://vignette.wikia.nocookie.net/fr.starwars/images/f/f6/Grand_Holocron.png/revision/lates

Références StarWars :

- www.starwars-holonet.com/encyclopedie/arme-sabrelasert.html