

RAPPORT D'IPO

MOROCCAN MARVEL



Sofyan Guillermet-Laouad

2022

Promo A – Groupe 6

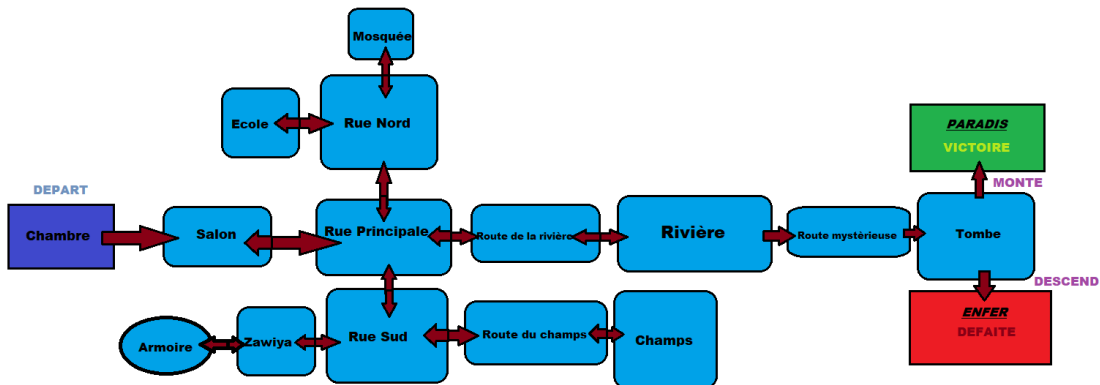
Thème

On suit Tamashe, un marocain du 12^e siècle, à travers sa vie et ses choix.

Scénario

Tamashe est d'abord bébé puis enfant, adolescent, adulte et enfin âgé. A chaque moments de sa vie il devra faire des choix qui détermineront son destin posthume : Aller en Enfer ou au Paradis. Cela dépendra de si il a fait plus de bons choix ou de mauvais choix durant sa vie.

Plan:



Scénario détaillé

Tamashe commence bébé dans sa chambre, il ressent une envie soudaine de casser le jouet en bois que son grand-père lui a fabriqué : Si il le casse sans raison, c'est 1 mauvaise action ; Si il le laisse tranquille, c'est 1 bonne action.

Il rentre ensuite dans le salon et c'est alors un enfant, sa mère lui a demandé de ranger la pile de linge sale qui y traîne :

Il peut soit ranger les vêtements(bonne action) soit ne pas écouter sa mère(mauvaise action).

Quand il revient de l'école et repasse par la Rue Principale c'est maintenant un adolescent, un troisième choix s'offre à lui : abreuver un chat avec une jarre d'eau trouvée dans la rue(bonne action) ou le laisser mourir de soif(mauvaise action).

En descendant vers la route Sud, le voilà adulte. Comme tout adulte il doit cependant travailler, dans les champs de blé en l'occurrence. Il y va donc pour travailler et sur le chemin du retour il veut aller à la Zawiya(lieux de cultes mystiques typiques du Maghreb médiéval) et rencontre un mendiant qui lui demande quelques pièces. Deux nouveaux choix : Donner une partie de la petite somme qu'on vient de gagner en travaillant au champs / L'ignorer. La première solution est celle qu'il faut choisir. Après son tour à la Zawiya, il retourne à la Rue Principale.

Quand il y arrive, c'est maintenant un homme âgé. Et Tamashe souhaite maintenant profiter un peu de son temps en contemplant la rivière donc il s'y dirige. Sur le chemin il croise des enfants excités lui demandant de leur lire un vieux livre qu'ils ont trouvé : si le joueur lit le livre aux enfants il a prit la bonne décision. Tamashe arrive donc à la rivière après ce dernier choix et la contemple, cependant un chemin qui n'était pas là avant est apparu, et pour une raison qu'il ignore Tamashe se sent obligé de l'emprunter, comme si le chemin était inévitable et que c'était son destin.

Arrivé au bout du chemin, Tamashe ne peut voir qu'un trou destiné à enterrer quelqu'un, et il ne peut plus faire demi tour car le chemin a disparu, il ne peut pas fuir la mort. En rentrant dans sa demeure finale il se pose la question : « Ai-je fais les bons choix ? » c'est la réponse à cette question (que le joueur est sensé connaître) qui va déterminer le destin de Tamashe : Paradis ou Enfer ? Victoire ou Défaite ?

Détails des lieux

Le jeu se déroule dans un village typique du Maroc médiéval.

Le lieu principal est probablement la Rue Principale, car c'est là que le personnage principal passe le plus et que les ellipses temporelles prennent place.

Il y a la chambre, qui est la première pièce dans laquelle le jeu commence et à laquelle on ne peut plus accéder une fois sorti.

Il y a ensuite le salon, dans lequel Tamashe est un enfant et depuis lequel il peut enfin quitter la maison pour découvrir le monde extérieur.

Puis, la fameuse Rue Principale, principal axe et repère de la ville de Tamashe.

Au Nord de la Rue, l'école dans laquelle Tamashe va puis la mosquée dans laquelle il peut aller durant son adolescence.

Au Sud de la Rue, la Zawiya et le chemin pour aller au champs, deux lieux que Tamashe fréquentera une fois adulte.

Finalement, à l'Est de la Rue Principale, le chemin de la rivière dans laquelle Tamashe peut s'amuser à l'adolescence et qu'il contempera dans sa vieillesse avant de suivre son destin.

La salle de la tombe n'a que deux issues : en haut le Paradis, la victoire et en bas l'Enfer et donc la défaite.

Détails des objets:

Pour le moment quelques items ont été créés mais je ne suis pas sûr de savoir si je vais en rajouter :

Un jouet en bois fait par notre grand-père dans la chambre du début du jeu.

Une pile de vêtements sales et un plat de couscous dans le salon.

Une jarre d'eau dans la rue principale.

Un sac de pièces dans les champs où l'on travaille.

Un msemmen (crêpe marocaine traditionnelle) magique dans la Zawiya (le magic-cookie)

Un vieux livre sur le chemin de la rivière.

Un encens magique (qui sert de beamer)

Détails des personnages:

Le personnage incarné est donc Tamashe comme dit précédemment. Il rencontre le mendiant lui demandant de l'argent, un chat, et des enfants voulant lire un livre.

Situation gagnante

Avoir pris au moins 3/5 bon choix et aller au Paradis.

Situation perdante:

Avoir pris au moins 3/5 mauvais choix et aller en Enfer.

Éventuellement énigmes, mini-jeux, combats, etc.:

Hormis les différents choix possibles dans le jeu et obligatoires, le jeu ne possède pas(ou pas encore) de contenu supplémentaire pour le joueur

Commentaires:

J'ai eu recours quelques fois à l'aide d'un ami et d'internet pour avoir des conseils sur le code mais hormis cela le jeu provient à 100 % de mon travail.

Mode d'emploi:

Tout d'abord pour lancer le jeu il faut ouvrir le projet en jar puis compiler le projet et faire clique droit sur la classe Game et cliquer sur newGame(). Cela lance automatiquement le jeu.

Le jeu est simple il y a des commandes qui peuvent être retrouvés en tapant "aide" qui sont:

- goRoom: permet de se déplacer entre les pièces du jeu
- printHelp: montre toutes les commandes possibles
- quit: quitte le jeu

Déclaration anti-plagiat:

J'ai eu recours quelques fois à l'aide d'un ami et d'internet pour avoir des conseils sur le code mais hormis cela le jeu provient à 100 % de mon travail.

Réponses aux exercices:

7.5.4. J'ai créé la procédure `printLocationInfo()` qui affiche la description de la room courante ainsi que toutes ses sorties. On nous demande de créer et d'utiliser cette procédure dans d'autres méthodes tel que `goRoom()` car quand on a plusieurs room avec plus de 4 sorties différentes, il est plus facile de les utiliser. Et si il est nécessaire de changer quelque chose dans le code, on aura qu'à le faire une fois au lieu de le faire room par room, cela évite la duplication de code.

7.6. La création de l'accessor `getExit()` permet d'éviter la redondance de `setExits` qui effectue toute une série de vérification pour déterminer si les sorties sont null ou pas. Le getter `getExit` permet de retourner la room qui correspond au paramètre `pDirection` qu'on transmet sans avoir à vérifier les autres sorties si elles existent.

7.7. J'ai créé une procédure `getExitString()` dans la classe `Room`, cette procédure retourne les différentes sorties existantes d'une room dans un `String`. On appelle ensuite cette procédure dans `printLocationInfo()`, au lieu de créer un `String` pour chaque sortie de chaque room, on a maintenant toutes les sorties d'une room en un seul `String`.

On met `getExitString()` dans la classe `Room` car on a besoin de ses attributs privés et parce qu'elle est plus adaptée que la classe `Game` qui est le moteur du jeu.

Dans la classe `Game` on affiche simplement ce que retourne la procédure dans `printLocationInfo()` qui a été modifié.

7.8. J'ai remplacé les 4 attributs par une `HashMap` dans le but de faciliter l'ajout d'autres directions. Ensuite, j'ai dû modifier le constructeur naturel, `getExit()` ainsi que `setExits` qui étaient devenu obsolètes à cause de la suppression des précédents arguments.

Suite à cela j'ai dû également modifier la création des sorties des room et pour régler un problème qui était que le programme n'affichait plus de message pour une direction inconnue mais seulement qu'il n'y avait pas de porte, j'ai créé un nouvel attribut et une nouvelle méthode dans la classe `CommandWords` et je n'ai eu qu'à l'intégrer dans `goRoom()` afin d'avoir les messages attendus affichés.

7.8.1. J'ai ajouté des mouvements verticaux "haut" et "bas" et j'ai trouvé que cela était mieux pour la fin du jeu plutôt que d'aller vers le Nord pour aller au Paradis et vers le Sud pour aller en Enfer.

7.9/7.10. On commence par une petite chaîne de caractères "Exits :" qui va servir à présenter les sorties de la pièce courante et on initialise la variable vKeys qui va contenir toutes les sorties de la pièce courante grâce à la méthode keySet(). Ensuite avec une boucle *for each* on parcourt toutes les sorties de la room courante dans vKeys et le programme ajoute au fur et à mesure dans exit déclarée précédemment si elles existent.

On finit par retourner un String composé de « Exits : » suivi de toutes les sorties de la room courante.

7.10.2. Malgré le fait que la classe Game soit le moteur du jeu et qu'on pourrait s'attendre à ce que sa javadoc soit la plus garnie, la classe Room a une javadoc plus conséquente. Cela est dû au fait que malgré sa taille, la majorité des méthodes de Game sont en *private* tandis que les méthodes de la classe Room sont en *public*

7.11. On observe de la duplication de code dans le programme car il y a plusieurs méthodes affichant notre position et les sorties de la room.

On corrige donc ce problème à l'aide de cette la méthode getLongDescription() :

```
/**
 *Retourne la description de la salle courante et ses sorties
 */
public String getLongDescription(){
    return "You are " + Description + ".\n" + "Exits:" + "\n" + getExitString();
}
```

Et on remplace toutes les lignes de codes inutiles par cette méthode.

~~7.12.~~

~~7.13.~~

7.14. On commence par implémenter les nouvelles commandes dans l'argument validCommands de la classe CommandWords:

```
private static final String[] validCommands = {"go","quit","help","look","eat"}
```

Puis dans Game on ajoute:

```
else if(commandWord.equals("look")){
    this.look();
}
```

Ainsi que la méthode look().

7.15. (A partir de cette question il n'y aura plus de captures d'écran du code car je m'y suis pris trop tard.)

Comme pour look ajoute la commande « eat » l'attribut aValidCommands et le constructeur de la classe CommandWord. On ajoute également la méthode eat() dans la classe Game ainsi

que ceci :

7.16. Dans cet exercice je crée une méthode `showAll()` qui affichera une liste de toutes les commandes stockées dans `aValidCommands` de `CommandWords`.

Après avoir fait cela, on crée une méthode `showCommands()` dans `Parser` qui appelle `showAll()`, ce qui va nous permettre d'ensuite de l'utiliser dans `Game` en appelant `showCommands()`, notamment dans `printHelp()`, sans lier `Game` et `CommandWords`.

7.17. Si nous voulons ajouter une commande nous n'avons pas à modifier `Game` :

En l'ajoutant dans `aValidCommands` dans `CommandWords` la commande sera automatiquement dans `showAll()` de la classe `Parser` et donc dans `showCommands()` de la classe `Game` sans avoir à la modifier.

7.18. Dans la classe `CommandWords` on remplace `showAll()` par un getter `getCommandList()` qui va retourner une chaîne de caractères que l'on pourra ensuite afficher en remplaçant `showAll()` dans `showCommands()` de `Parser` et `printHelp()` de `Game` par notre nouvelle méthode.

7.18.1 Je n'ai pas effectué de réelles modifications de mon programme par rapport à `zuul-better`.

~~7.18.2~~

7.18.3 Pour le moment 5 images ont été ajoutées dont une quasiment obsolète car on y voit la fenêtre de l'outil capture d'écran. Elles proviennent toutes du jeu `Genshin Impact`, et les images que je compte ajouter en proviendront également. (Ces images sont maintenant visibles dans le dossier `images`)

7.18.4 Le titre de mon jeu est `Moroccan Marvel`, car le jeu se situe au Maroc et qu'il raconte la merveilleuse histoire(ou non?) de notre personnage.

7.18.5 Pour que les objets `Room` soit accessibles autre part que dans la méthode `createRooms()` de la classe `Game`, on crée une `HashMap` stockant tous ces objets.

On importe donc `java.util.HashMap` dans `Game` et on déclare le nouvel attribut `private HashMap<String, Room> = aRooms` que l'on initialise ensuite dans le constructeur.

Ensuite nous pouvons remplir cette `HashMap` avec toutes les `rooms` à l'aide de `put` et créer l'accessor `getRoom()` qui va permettre aux autres classes d'accéder aux `room` à l'aide de la clé de la `HashMap` si cela devient nécessaire à un quelconque moment.

7.18.6 Les ajouts, modifications et suppressions demandés demandés dans mon programme par rapport à zuul-with-images ont été effectué sans soucis.

Ajouts : `UserInterface`, `GameEngine` (classes) `setGUI` et `endGame()` dans `GameEngine` (méthodes), import de classes java dans `UserInterface` pour le GUI.

Suppressions : Majorité du code de `Game` transféré vers `GameEngine`, donc suppression de la majorité du code de `Game`. La méthode `quit()`, `Scanner` dans `Parser`

Modifications : `processCommand()` → `interpretCommand()` , `System.out.println()` → `this.aGUI.println()`, `getCommand()` dans `Parser`.

~~7.18.7~~

7.18.8 Dans la classe `UserInterface` on importe la classe `javax.swing.JButton` ce qui va nous permettre de créer des boutons dans notre interface et de les utiliser.

On déclare donc un attribut pour chaque bouton que nous souhaitons utiliser(ici les commandes « go up », « go down », « go west », « go north », « go east », « go south », « help », « quit »)

On importe ensuite `java.awt.GridLayout` qui va nous permettre de placer ces boutons sur l'interface en les stockant sur un `Panel` que l'on déclare en tant qu'attribut (`aPanelEast`) avant de l'initialiser dans `createGUI()` et `java.awt.Color` qui va nous permettre d'en colorer certains si on le souhaite.(ici j'ai colorié « help » en vert et « quit » en rouge pour contraster avec les boutons directionnels)

On initialise ensuite chaque bouton dans `createGUI()` avec le mot que l'on veut voir écrit dessus avant de lui ajouter un `ActionListener` pour que le bouton réagisse quand on clique dessus. Enfin, on ajoute chacun de ces boutons au `panel` en partant de celui placé 1ere ligne 1ere colonne et en finissant par celui à la 3eme ligne 2eme colonne(dans mon cas) avant d'enfin placer le `panel` à la droite de l'interface, c'est donc là que seront nos boutons.

On crée ensuite la méthode `ActionPerformed()` dans laquelle on va assigner une commande pour chaque réaction d'un bouton.(par exemple quand l'`ActionListener` du bouton `QUIT` détecte un clic sur le bouton, `ActionPerformed` va effectuer la commande `quit()` en réponse.) Quand à la couleur des boutons `HELP` et `QUIT`, on utilise simplement `this.aButton*.setBackground(Color.*)` en assignant une couleur reconnue par java au bouton.

~~7.19~~

~~7.19.1~~

1.19.2 Comme mentionné dans une des questions précédentes, les images du jeu ont été déplacées dans un dossier `images`, on modifie donc la méthode `createRoom()` en indiquant

le chemin vers le fichier.png à la place du nom de l'image.

7.20. On crée la classe Item avec ses getters et ses setters et bien sûr son constructeur, avant d'ajouter un attribut `item` dans la classe Room et d'ajouter l'item dans la méthode `createRoom()` de GameEngine.

7.21 On modifie la méthode `getLongDescription()` de la classe Room pour qu'elle affiche les items présent dans la room courante ou un message si il n'y en a pas.

7.22 Pour l'instant nous étions limités à un objet par room, pour changer ça on crée une HashMap qui stocke les item et les associe à leur nom String. On crée alors un attribut `private HashMap <String, Item> items` ; dans la classe Room avant de l'initialiser dans son constructeur.

On modifie alors les méthodes concernant les items tel que `getItem()` et `addItem()` pour spécifier quel item on souhaite récupérer/ajouter en le liant à son String que l'on rajoute dans les paramètres des méthodes. On modifie `getItemString()` pour qu'elle puisse retourner le String de tous les items de la room.

7.22.1 J'ai choisi une HashMap car cela rend l'accès et l'ajout d'éléments plus facile.

7.22.2 Tous les items ont été initialisés et placés dans leurs pièces respectives.

7.23. On crée une nouvelle méthode `back()` dans la classe GameEngine permettant au personnage de retourner dans la pièce précédente immédiatement. On doit donc d'abord rajouter le mot « back » comme un mot de commande valide dans la classe `CommandWords`. On crée ensuite un attribut dans la classe GameEngine et on l'initialise dans `goRoom()`. Enfin, on crée la méthode `back()` dans cette même classe.

7.26. L'un des principaux problèmes de la méthode `back` que nous avons créé est qu'elle effectue un aller-retour entre la dernière room et la room courante au moment du retour en arrière du personnage. Pour contourner ce problème, nous allons créer une pile qui est une sorte de liste à utiliser dans ce genre de cas, car elle récupère et supprime le dernier élément du stack avec une seule commande.

On importe donc la classe `java.util.Stack` avant de déclarer une nouvelle pile avec un attribut `aPreviousRooms` qui remplace `aPreviousRoom` et qu'on initialise dans le constructeur.

Dans la méthode `goRoom`, on ajoute alors une ligne pour ajouter la pile à la room précédente.

Push permet d'ajouter un élément au dessus de la pile et nous devons ensuite modifier la méthode `back` pour que la méthode `pop` récupère et supprime l'élément du haut de la pile.

7.26.1 On utilise les deux commandes fournies et on génère donc la javadoc.

7.28.1 On commence une fois de plus par ajouter « test » comme mot de commande valide dans `CommandWords` et on importe `java.io.FileNotFoundException` , `java.io.File` et `java.util.Scanner` dans la classe `GameEngine` car elles vont nous servir.

On crée donc la méthode `test()` qui va parcourir le fichier du jeu et chercher le fichier.txt ayant le nom que l'on demande de tester dans le terminal.

7.28.2 J'ai créé 3 fichiers test : `court.txt` qui teste quelques commandes de déplacement, `long.txt` qui effectue quelques commandes de déplacement et d'autres comme `look`, et `tourist.txt` qui teste toutes les mécaniques du jeu.

7.29 La classe `GameEngine` est surchargée et il devient dur de se repérer parmi toutes ces méthodes. On crée donc la classe `Player` dans laquelle toutes les méthodes de `GameEngine` concernant le joueur sont transférées, ces méthodes seront donc construites dans `Player` tandis que dans `GameEngine` on vérifiera plutôt les conditions permettant de les effectuer.

Dans `Player` on déclare donc le nom du joueur, la pièce courante et le stack des pièces précédentes qu'on initialise ensuite dans le constructeur naturel avant de créer les getters nous permettant d'accéder à ces attributs dans les autres classes .

Dans `GameEngine` on crée l'attribut `aPlayer` qu'on initialise ensuite en précisant le nom du personnage et sa room de départ. Comme dit précédemment les méthodes de `GameEngine` concernant le joueur servent maintenant à vérifier les conditions, si les bonnes conditions sont vérifiées, on appelle la méthode de `Player`.

7.30 Rien de très surprenant, on commence par ajouter « take » et « drop » dans la liste des mots de commandes valides de `CommandWords`.

On crée ensuite les méthodes `addItem()` et `removeItem()` dans la classe `Room` qui nous permettent respectivement d'ajouter ou de supprimer des items des room.

On ajoute ensuite l'attribut `HashMap aInventory` dans `aPlayer` qui stockera les items récupérés par le joueur qu'on initialise ensuite dans le constructeur, ainsi que les méthodes `getItem()` qui retourne un item en indiquant son nom et `getInventoryString()` qui retourne une chaîne de commande contenant tous les `String` des items stockés dans l'inventaire du joueur.

Enfin, on ajoute les deux méthodes `take()` qui supprime un item de la room et le stock dans l'inventaire et `drop()` qui supprime un item de l'inventaire et le place dans la room courante. On les ajoute également dans `GameEngine` où nous allons vérifier les conditions valides avant d'appeler ou non `this.aPlayer.take()` et `this.aPlayer.drop()`, on n'oublie pas de les ajouter dans la méthode `interpretCommand()` également.

7.31. Comme mentionné dans la question précédente, afin de porter plusieurs item une

HashMap représentant l'inventaire du joueur pouvant stocker les items a été créé, encore une fois pour une question de facilité d'accès et de modifications.

7.31.1 Afin d'éviter la duplication de code et la surcharge des autres classes, on crée la classe ItemList contenant comme attributs la HashMap aItemList qui stock les items et aLocation qui indique l'emplacement de la liste d'items (si elle est dans une room ou dans l'inventaire) et on importe java.util.HashMap et java.util.Set .

On initialise alors ces attributs dans le constructeur naturel ItemList() et on y ajoute les 3 getters getItem(), getItemString() et getItems() qui étaient présents dans d'autres classes avant d'ajouter les procédures addItem() et removeItem() qui étaient également dans les autres classes.

On retire tous les getters concernant les items de Player et on retire l'accesseur HashMap stockant les items pour le remplacer par un attribut ItemList aInventory qu'on initialise également avec « the inventory » en String pour préciser que c'est la liste des items de l'inventaire.

On ajoute dans GameEngine un getter nous permettant d'accéder à cet attribut.

On effectue la même chose dans Room mais cette fois avec un String « the room » pour préciser qu'il s'agit d'une liste d'items dans une room.

Finalement, on enlève addItem() de GameEngine et on modifie createRoom() pour faire en sorte que pour placer les items on accède à la classe ItemList vu que addItem() n'est plus dans GameEngine.

7.32. Une fois de plus, on ajoute « items » dans la liste de mots de commande valides dans CommandWords et on crée la méthode printInventory() dans GameEngine nous permettant d'afficher le contenu de l'inventaire du joueur, le poids des items qui y sont présents, et le poids max que le joueur peut porter.

On n'oublie toujours pas d'ajouter items dans interpretCommand() pour qu'elle soit fonctionnelle.

7.34. On crée un item magic-msemmen dans la room Zawiya et on modifie la méthode eat() de sorte à ce que lorsque le joueur mange cet item, le poids que son inventaire peut supporter(donc sa capacité) double.

7.34.1 Fichiers de test mis à jour pour ajouter les nouvelles fonctionnalités.

7.34.2 Javadoc générée une seconde fois.

7.42. Exercice non fait car incompatible avec le scénario, bien que j'aurais pu le faire en déclarant une limite de déplacements quasiment inatteignable.

7.42.1

7.42.2. Je me contenterais de l'IHM actuelle.

7.43. Une « trap room » est déjà incorporée dans le jeu pour des raisons scénaristiques :

Il s'agit de la room « Tombe », car quand on y rentre on ne peut plus retourner dans la room « Route mystérieuse » qui est la room précédente, et on peut seulement aller soit dans la room « Paradis » en cas de victoire ou dans la room « Enfer » en cas de défaite.

7.44. On commence par créer la classe Beamer qui est une sorte d'objet. On la dote de plusieurs commandes utiles qui vont nous faciliter l'utilisation du Beamer dans les autres classes :

On ajoute maintenant l'implémentation des commandes dans Player, GameEngine et CommandWords :




```
1  public CommandWords()
2  {
3      this.aValidCommands = new String[9];
4      this.aValidCommands[0] = "go";
5      this.aValidCommands[1] = "help";
6      this.aValidCommands[2] = "quit";
7      this.aValidCommands[3] = "look";
8      this.aValidCommands[4] = "eat";
9      this.aValidCommands[5] = "back";
10     this.aValidCommands[6] = "take";
11     this.aValidCommands[7] = "drop";
12     this.aValidCommands[8] = "inventory";
13     this.aValidCommands[9] = "charge";
14     this.aValidCommands[10] = "fire";
15 } // CommandWords()
```

On ajoute ci-dessus les deux commandes afin qu'elles soient maintenant valide pendant l'exécution du programme. Ci-dessous on créer les deux méthodes dans GameEngine qui elles-mêmes appelleront leurs méthodes respectives dans Player.

```

1  /**
2   * Charges the Beamer
3   * @param pCommand The command to execute
4   */
5  public void charge(final Command pCommand) {
6      if (!pCommand.hasSecondWord()) {
7          this.aGUI.println("Charge what?");
8          return;
9      }
10
11     String vItemName = pCommand.getSecondWord();
12
13     if (!this.aPlayer.getInventory().hasItem(vItemName)) {
14         this.aGUI.println("You don't have " + vItemName + ".");
15         return;
16     }
17
18     if (!(this.aPlayer.getInventory().getItem(vItemName) instanceof Beamer)) {
19         this.aGUI.println("You can't charge " + vItemName + " !");
20         return;
21     }
22
23     Item vItem = this.aPlayer.getInventory().getItem(vItemName);
24     Beamer vBeamer = (Beamer)vItem;
25
26     if (vBeamer.isCharged()) {
27         this.aGUI.println("You can't charge " + vItemName + " ! \n It is already charged.");
28         return;
29     }
30
31     this.aPlayer.charge(this.aPlayer.getCurrentRoom(), vBeamer);
32     this.aGUI.println("You charged " + vItemName + ".");
33 }
34
35 /**
36 * Fires the Beamer
37 * @param pCommand
38 */
39 public void fire(final Command pCommand) {
40     if (!pCommand.hasSecondWord()) {
41         this.aGUI.println("Fire what?");
42         return;
43     }
44
45     String vItemName = pCommand.getSecondWord();
46
47     if (!this.aPlayer.getInventory().hasItem(vItemName)) {
48         this.aGUI.println("You don't have " + vItemName + ".");
49         return;
50     }
51
52     if (!(this.aPlayer.getInventory().getItem(vItemName) instanceof Beamer)) {
53         this.aGUI.println("You can't fire " + vItemName + " !");
54         return;
55     }
56
57     Item vItem = this.aPlayer.getInventory().getItem(vItemName);
58     Beamer vBeamer = (Beamer)vItem;
59
60     if (!vBeamer.isCharged()) {
61         this.aGUI.println("You can't fire " + vItemName + " ! \n It is not charged.");
62         return;
63     }
64
65     this.aPlayer.fire(vBeamer);
66     this.aGUI.println("You fired " + vItemName + ".");
67     this.printLocationInfo();
68 }
69
70

```




```

1  /**
2   * Charges the beamer that is passed as
3   * @param pRoom the room to charge
4   * @param pBeamer the beamer to charge
5   */
6  public void charge(final Room pRoom, final Beamer pBeamer) {
7      pBeamer.charge(pRoom);
8  }
9
10 /**
11  * Fires the beamer that is passed as parameter
12  * @param pBeamer the beamer to fire
13  */
14 public void fire(final Beamer pBeamer) {
15     pBeamer.fire(this);
16 }

```

On ajoute maintenant un Beamer dans la liste des Items :

On ajoute maintenant cet Item dans la salle vAdultZawiya

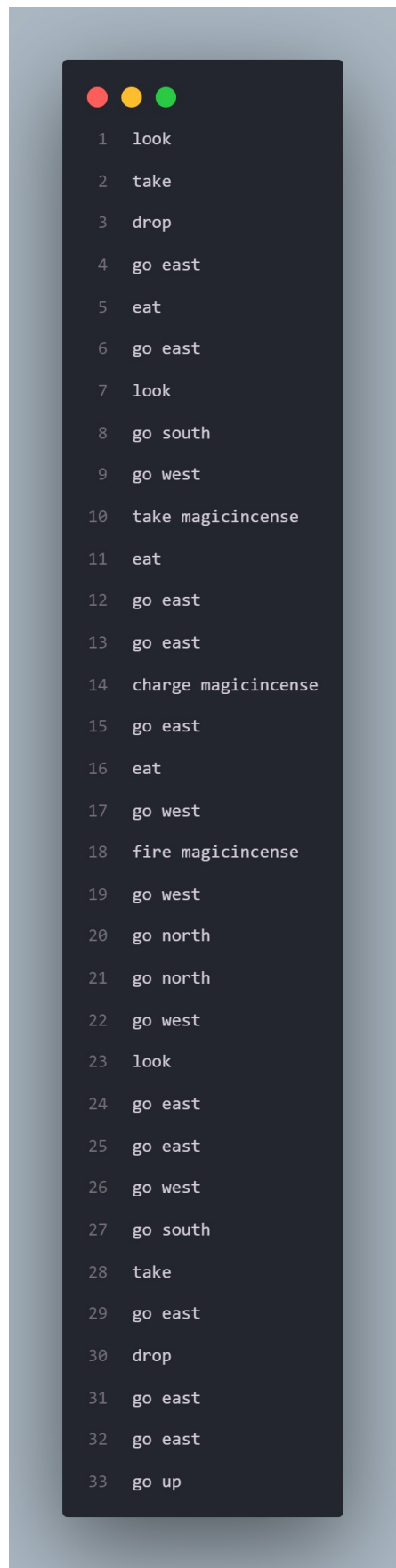


```

1  vAdultZawiya.getRoomItems().addItem("magicincense", vMagicIncense);

```

7.45.1. Les fichiers de tests ont été mis à jour pour refléter l'utilisation du Beamer.



7.46. Dans cet exercice on crée deux nouvelles classes `TransporterRoom` et `RoomRandomizer` qui serviront respectivement à téléporter l'utilisateur dans une pièce

aléatoire à la sortie de cette salle, et l'autre à retourner une salle aléatoire parmi la liste totale des salles disponibles.

J'ai choisi de coder d'abord la classe RoomRandomizer par soucis de simplicité :

```

1  import java.util.ArrayList;
2  import java.util.List;
3  import java.util.Random;
4
5  /**
6   * This class is used to generate a random room for the player to enter.
7   *
8   * @author Sofyan Guillermet-Laouad and Aloïs Fournier 19.05.23
9   */
10
11 public class RoomRandomizer {
12     private List<Room> aRooms;
13     private Random aRandom;
14
15     /**
16      * Constructor of the class RoomRandomizer
17      */
18     public RoomRandomizer() {
19         this.aRooms = new ArrayList<>();
20         this.aRandom = new Random();
21     }
22
23     /**
24      * Add a room to the list of rooms
25      *
26      * @param pRoom
27      */
28     public void addRoom(final Room pRoom) {
29         this.aRooms.add(pRoom);
30     }
31
32     /**
33      * Returns a random room from the list of rooms
34      *
35      * @return a random room
36      */
37     public Room getRandomRoom() {
38         if (this.aRooms.isEmpty()) {
39             return null;
40         }
41         return this.aRooms.get(this.aRandom.nextInt(this.aRooms.size()));
42     }
43 }

```

On code ensuite la classe TransporterRoom

```

1  /**
2   * TransporterRoom is a type of room that teleports the player to a random room
3   */
4  public class TransporterRoom extends Room {
5
6      private RoomRandomizer aRandomizer;
7
8      /**
9       * Constructor of the class TransporterRoom
10      * @param pDescription
11      * @param
12      */
13      public TransporterRoom(final String pName, final String pImage) {
14          super(pName, pImage);
15          this.aRandomizer = new RoomRandomizer();
16      }
17
18      /**
19       * Add a room to the list of rooms
20       * @param pRoom the room to add
21       */
22      public void addRoom(final Room pRoom) {
23          this.aRandomizer.addRoom(pRoom);
24      }
25
26      /**
27       * Returns a random room from the list of rooms
28       * @return a random room
29       */
30      public Room getRandomExit() {
31          return this.aRandomizer.getRandomRoom();
32      }
33  }

```

Dans `createRooms()` on ajoute maintenant toutes les Rooms présente pour faire fonctionner la `TransporterRoom`.

On modifie enfin la méthode `goRoom()` pour rendre la `TransporterRoom` utilisable:

```

1  if (this.aPlayer.getCurrentRoom() instanceof TransporterRoom) {
2      TransporterRoom vTransporter = (TransporterRoom) this.aPlayer.getCurrentRoom();
3      if (this.aIsInTestMode && !this.aFakeRandomRoom.isEmpty()) {
4          ArrayList<Room> vRooms = new ArrayList<Room>(vTransporter.getExits());
5          for (Room pRoom : vRooms) {
6              String vRoomName = pRoom.getDescription().toLowerCase();
7              if (this.aFakeRandomRoom.contains(vRoomName.toLowerCase())) {
8                  this.aPlayer.setCurrentRoom(pRoom);
9                  this.printLocationInfo();
10                 return;
11             }
12         }
13         this.printLocationInfo();
14         return;
15     }
16     this.aPlayer.setCurrentRoom(vTransporter.getRandomExit());
17     this.aPlayer.getPreviousRooms().clear();
18     return;
19 }

```

7.46.1. On créer une commande alea dans GameEngine et dans CommandWords (pour la validité de la commande). On ajoute aussi deux attribut pour de un vérifier que nous sommes dans le mode de test et de deux mémoriser la salle qu'on veut prédire

On ajoute dans la commande test() les instructions nécessaires :

```

1 private void test(final Command pCommand) {
2     this.aIsInTestMode = true;
3     if (pCommand.hasSecondWord() == false) { // We want to know if there is a second word
4         this.aGUI.println("Test what?");
5         return;
6     }
7     // We use the second word
8     String vTest = pCommand.getSecondWord();
9     try {
10         Scanner vScanner = new Scanner(new File("" + vTest + ".txt"));
11         this.aGUI.println("" + vTest + "is being tested...");
12         while (vScanner.hasNextLine()) {
13             interpretCommand(vScanner.nextLine());
14         }
15     } catch (FileNotFoundException pE) { // If the file is not existing
16         this.aGUI.println("File not found.");
17     }
18     this.aIsInTestMode = false;
19 }
20 }
21

```

On code maintenant la méthode alea :

```

1 private void alea(Command vCommand) {
2     if (!vCommand.hasSecondWord()) {
3         this.aGUI.println("Alea what?");
4         return;
5     }
6
7     if (!this.aIsInTestMode) {
8         this.aGUI.println("You can't use this command if you are not in test mode.");
9         return;
10    }
11    String vRoomName = vCommand.getSecondWord();
12
13    if (this.aFakeRandomRoom.isEmpty()) {
14        this.aFakeRandomRoom = vRoomName;
15    } else {
16        this.aFakeRandomRoom = "";
17    }
18 }

```

Ajouts supplémentaires relatifs au scénario:

Les classes Event, Flag et FlagList ont été ajoutées pour les besoin du scénario.

```
1  /**
2   * An even is fired up when you enter certain Rooms, impeaching you to exit while the condition is not met.
3   * @author Sofyan Guillermet-Lauouad and Aloïs Fournier 19.05.23
4   */
5  public class Event {
6      private String aName;
7      private UserInterface aGUI;
8      private boolean aEventDone = false;
9      private String choice1;
10     private String choice2;
11     private int correctChoice;
12
13     public Event(final UserInterface pGUI, final String pName, final String pChoice1, final String pChoice2) {
14         this(pGUI, pName, pChoice1, pChoice2, 1);
15     }
16     public Event(final UserInterface pGUI, final String pName, final String pChoice1, final String pChoice2, final int pCorrectChoice) {
17         this.aGUI = pGUI;
18         this.aName = pName;
19         this.choice1 = pChoice1;
20         this.choice2 = pChoice2;
21         this.correctChoice = pCorrectChoice;
22     }
23
24     public void displayChoice() {
25         this.aGUI.println("You have to choose between two options :");
26         this.aGUI.println("1. " + this.choice1);
27         this.aGUI.println("2. " + this.choice2);
28     }
29
30     public boolean testChoice(final int pChoice) {
31         if (pChoice == this.correctChoice) {
32             return true;
33         } else {
34             return false;
35         }
36     }
37
38     public void run() {
39         this.aGUI.println("You entered a Room with an event in it, you can not leave until you have completed the event.");
40         this.aGUI.println("You have to find a way to exit the Room.");
41         this.displayChoice();
42     }
43
44     public void setEventDone() {
45         this.aEventDone = true;
46         this.aGUI.println("The event has been completed !");
47         this.aGUI.println("Check your bad/good action counter to see if you made the right choice !");
48     }
49
50     public void setGUI(final UserInterface pGUI) {
51         this.aGUI = pGUI;
52     }
53     public boolean isEventDone() {
54         return this.aEventDone;
55     }
56 }
57
58
59
```

Cette classe sert à créer des événements offrant deux choix à l'utilisateur, elle n'est pas encore complète car l'utilisation des Flags n'est pas encore implémentée. Pour chaque salle comportant un événement, l'utilisateur sera empêché de bouger jusqu'à la complétion de celui-ci, résultant en une mauvaise ou bonne action suivant le choix. Ces bonnes/mauvaises actions seront comptabilisées dans deux attributs de Player et définiront la condition de victoire ou défaite.

Les événements sont créés comme suit :

```

1 // Event Creation
2 Event vBirthEvent = new Event(this.aGUI, "You feel a strange urge to break your toy.", "Break your toy", "Leave your toy be");
3 Event vChildEvent = new Event(this.aGUI, "Your mother told you earlier to clean the pile of dirty clothes.", "Clean up your clothes", "Don't listen to your mother");
4 Event vTeenagerEvent = new Event(this.aGUI, "You see a cat that looks thirsty.", "Feed the cat", "Leave the cat thirstier than ever");
5 Event vAdultEvent = new Event(this.aGUI, "You see a beggar in the street.", "Give him some coins", "Ignore him");
6 Event vSeniorEvent = new Event(this.aGUI, "You see some kids playing in the street.", "Read them a story", "Tell them to go bother someone else");
7
8 // Add Events to Room
9 vBirth.addEvent(vBirthEvent);
10 vChildR.addEvent(vChildEvent);
11 vLaStreet.addEvent(vTeenagerEvent);
12 vLaStreet.addEvent(vAdultEvent);
13 vRiverWay.addEvent(vSeniorEvent);

```

La classe Flag a également été rajoutée car nous avons scénaristiquement besoin de suivre l'avancement du joueur dans son parcours. Il y a 4 flags pour les 4 stades de l'humain : Baby, Child, Teenager et Senior. Chacun de ces flags peuvent être obtenus à la fin de chaque event quel que soit le choix effectué. Ces flags seront régis par une FlagList, très similaire à l'ItemList qu'on connaît. Si l'utilisateur possède un flag, il est impérativement dans la FlagList (c'est donc un attribut de la classe Player).


```
1  /**
2   * Flag class used to track the advancement of the Player
3   */
4  public class Flag {
5      private String aFlagName;
6
7      /**
8       * Constructor of the class Flag
9       * @param pFlagName
10      */
11     public Flag(final String pFlagName) {
12         this.aFlagName = pFlagName;
13     }
14
15     /**
16      * Returns the name of the flag
17      * @return the name of the flag
18      */
19     public String getFlagName() {
20         return this.aFlagName;
21     }
22 }
23
```

La classe Flag est ci-dessus. Elle sert à définir un « drapeau » permettant de suivre l'utilisateur dans sa progression comme expliqué au dessus.
la classe FlagList ci-dessous permet de mutualiser les Flags et donc de gérer les Flags plus facilement.

```

1  import java.util.HashMap;
2
3  /**
4   * The FlagList that mutualize the flags of the game
5   *
6   * @author Sofyan Guillermet-Laouad (with the help of Aloïs Fournier)
7   */
8  public class FlagList {
9      private HashMap<String, Flag> aFlags;
10
11      /**
12       * Constructor of the class FlagList
13       */
14      public FlagList() {
15          this.aFlags = new HashMap<>();
16      }
17      /**
18       * Add a flag to the list of flags
19       *
20       * @param pFlag
21       */
22      public void addFlag(final Flag pFlag) {
23          this.aFlags.put(pFlag.getFlagName(), pFlag);
24      }
25
26      /**
27       * Returns the flag with the given name
28       *
29       * @param pFlagName
30       * @return the flag with the given name
31       */
32      public Flag getFlag(final String pFlagName) {
33          return this.aFlags.get(pFlagName);
34      }
35
36      /**
37       * Return true if the flag exists, false otherwise
38       *
39       * @param pFlagName
40       */
41      public boolean hasFlag(final String pFlagName) {
42          return this.aFlags.containsKey(pFlagName);
43      }
44
45      /**
46       * Remove the flag with the given name
47       *
48       * @param pFlagName
49       */
50      public void removeFlag(final String pFlagName) {
51          this.aFlags.remove(pFlagName);
52      }
53  }
54

```

On obtient donc un jeu complètement fonctionnel avec comme condition de victoire avoir un nombre supérieur ou 3 de bonnes actions et comme condition de défaite un nombre

supérieur ou égal à 3 de mauvaises actions.

