



# The Ring and the Lost Tower

Rapport projet Zuul



## Table des matières

I).....	4
A – Auteur.....	4
B – Thème.....	4
C – Résumé du scénario .....	4
D – Plan.....	4
E – Scénario détaillé .....	6
F – Détail des items, personnages.....	6
G – Situations gagnantes et perdantes .....	7
H – Eventuellement énigmes, mini-jeux, combats, etc.....	7
I – Commentaires .....	7
II) Réponses aux exercices.....	7
Exercice 7.5 :.....	8
Exercice 7.6 :.....	8
Exercice 7.7 :.....	8
Exercice 7.8 :.....	8
Exercice 7.8.1 :.....	9
Exercice 7.9.....	9
Exercice 7.10 :.....	9
A Savoir Expliquer :.....	9
Exercice 7.10.1 :.....	10
Exercice 7.10.2 :.....	10
Exercice 7.11 :.....	10
Exercice 7.14 :.....	10
Exercice 7.15 :.....	11
Exercice 7.16 :.....	12
A Savoir Expliquer :.....	12
Exercice 7.18 :.....	12
Exercice 7.18.1 :.....	13
Exercice 7.18.3 :.....	13
Exercice 7.18.4 :.....	13
Exercice 7.18.5 :.....	13

Exercice 7.18.6 :.....	14
Exercice 7.18.7 :.....	15
Exercice 7.18.8 :.....	15
A Savoir Expliquer :.....	16
Exercice 7.19.2 :.....	16
Exercice 7.20 :.....	16
Exercice 7.21 :.....	17
Exercice 7.21.1 :.....	18
Exercice 7.22 :.....	18
Exercice 7.22.2 :.....	19
Exercice 7.23 :.....	20
Exercice 7.24 :.....	20
Exercice 7.25 :.....	20
Exercice 7.26 :.....	21
A Savoir Expliquer :.....	21
Exercice 7.26.1 :.....	22
Exercice 7.28.1 :.....	22
A Savoir Expliquer :.....	23
Exercice 7.28.2 :.....	24
Exercice 7.29 :.....	24
Exercice 7.30 :.....	26
Exercice 7.31 :.....	28
Exercice 7.31.1.....	29
Exercice 7.32 :.....	29
Exercice 7.33 :.....	30
Exercice 7.34 :.....	31
Exercice 7.34.1 :.....	32
Exercice 7.34.2 :.....	32
Exercice 7.42 :.....	32
Exercice 7.42.1 :.....	33
Exercice 7.42.2 :.....	34
Exercice 7.43 :.....	36
Exercice 7.44 :.....	37

Exercice 7.45 : .....	39
Exercice 7.45.1 : .....	40
Exercice 7.45.2 : .....	40
Exercice 7.46 : .....	40
A savoir expliquer : .....	42
Exercice 7.46.1 : .....	43
Exercice 7.46.3 : .....	44
Exercice 7.46.4 : .....	44
Exercice 7.47 : .....	44
A Savoir Expliquer : .....	45
Exercice 7.47.1 : .....	45
A Savoir Expliquer : .....	45
Exercice 7.47.2 : .....	45
Exercice 7.53 : .....	48
III) Mode d'emploi .....	48
IV) Ajouts .....	48
V) Déclaration anti-plagiat .....	49
VI) Sources .....	49

I)

A – Auteur

Joël Trésor Mbiapa Ketchekmen

B – Thème

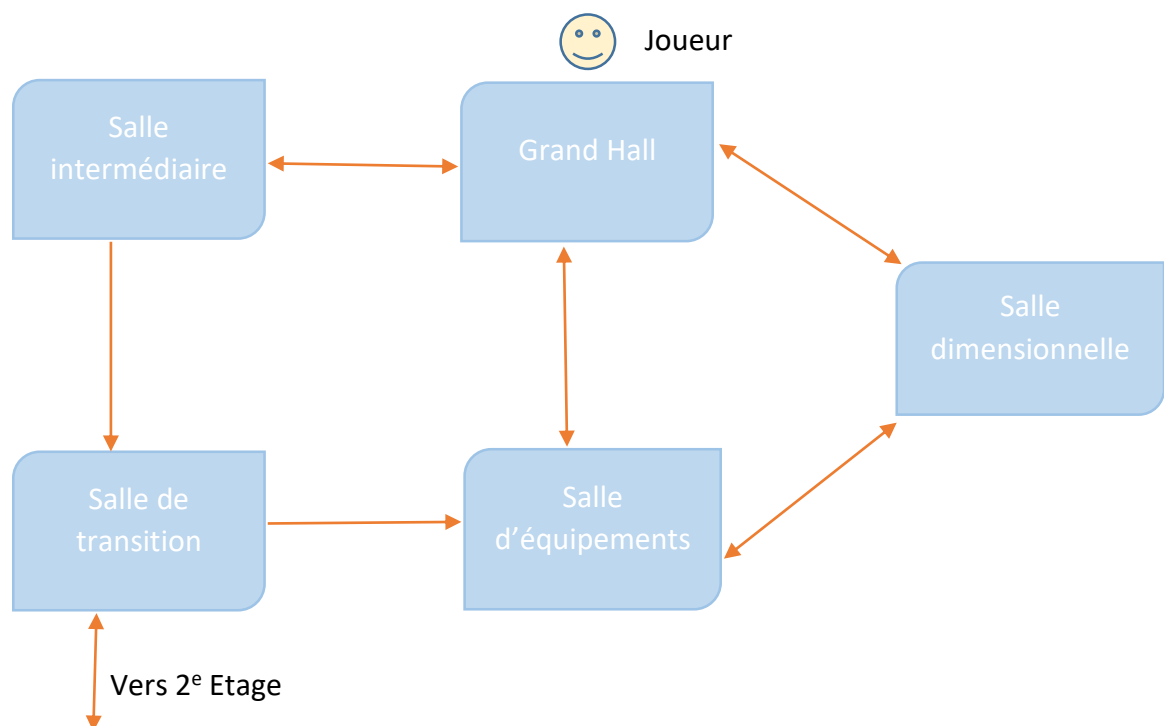
Sur l'île de Jeju, dans la Tour Perdu, récupérer la Bague de l'Illusion.

C – Résumé du scénario

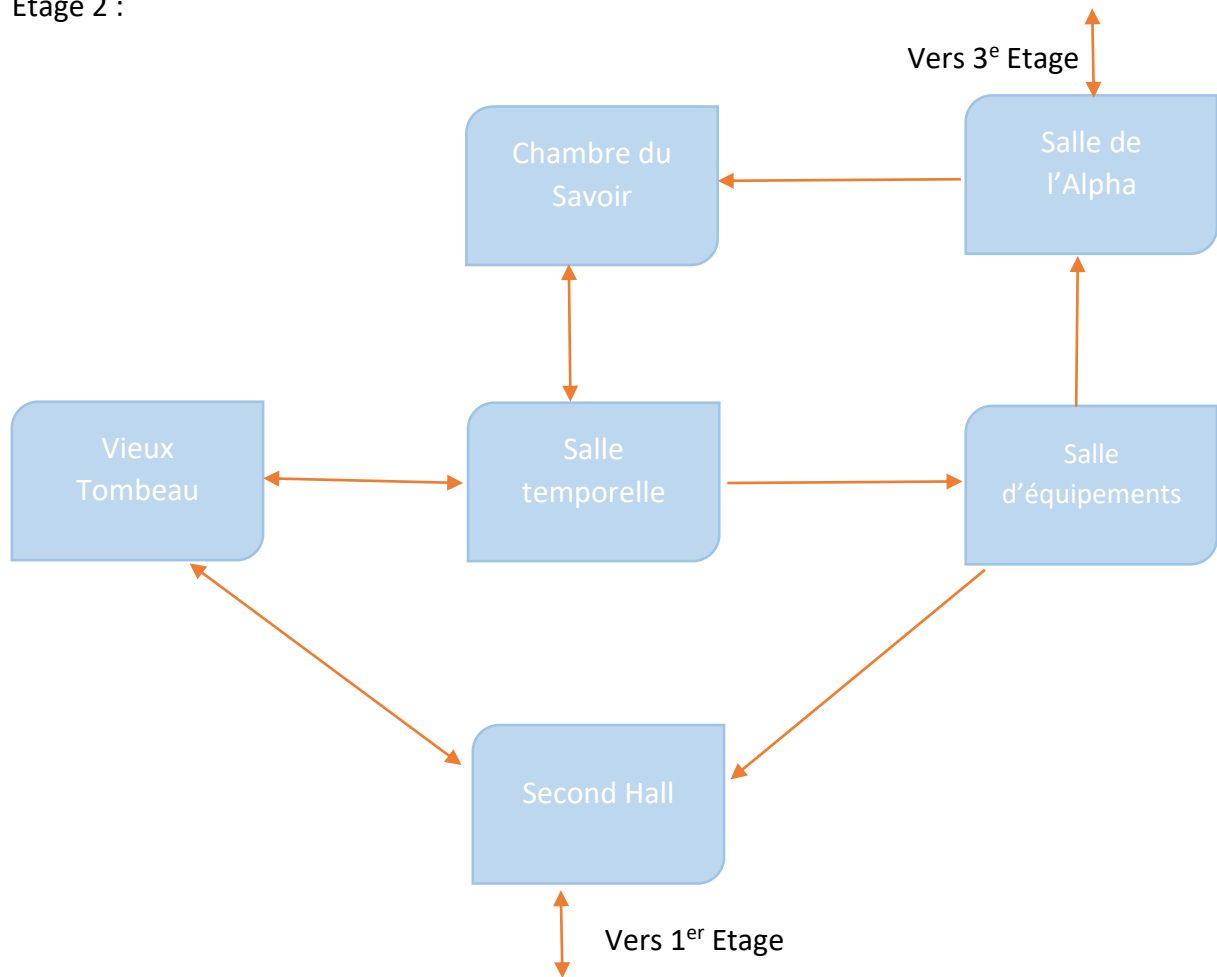
Convoqué quelques jours auparavant par les dirigeants américain, le joueur se retrouve aujourd'hui face à la Tour Perdu Coréenne après avoir trouvé son emplacement. Mercenaire indépendant, Trey Thomas est missionné pour récupérer la relique sacrée d'Asie : la Bague de l'Illusion. Il doit récupérer la relique Sacrée avant que les forces armées nippono-coréenne ne le capture dans la Tour.

D – Plan

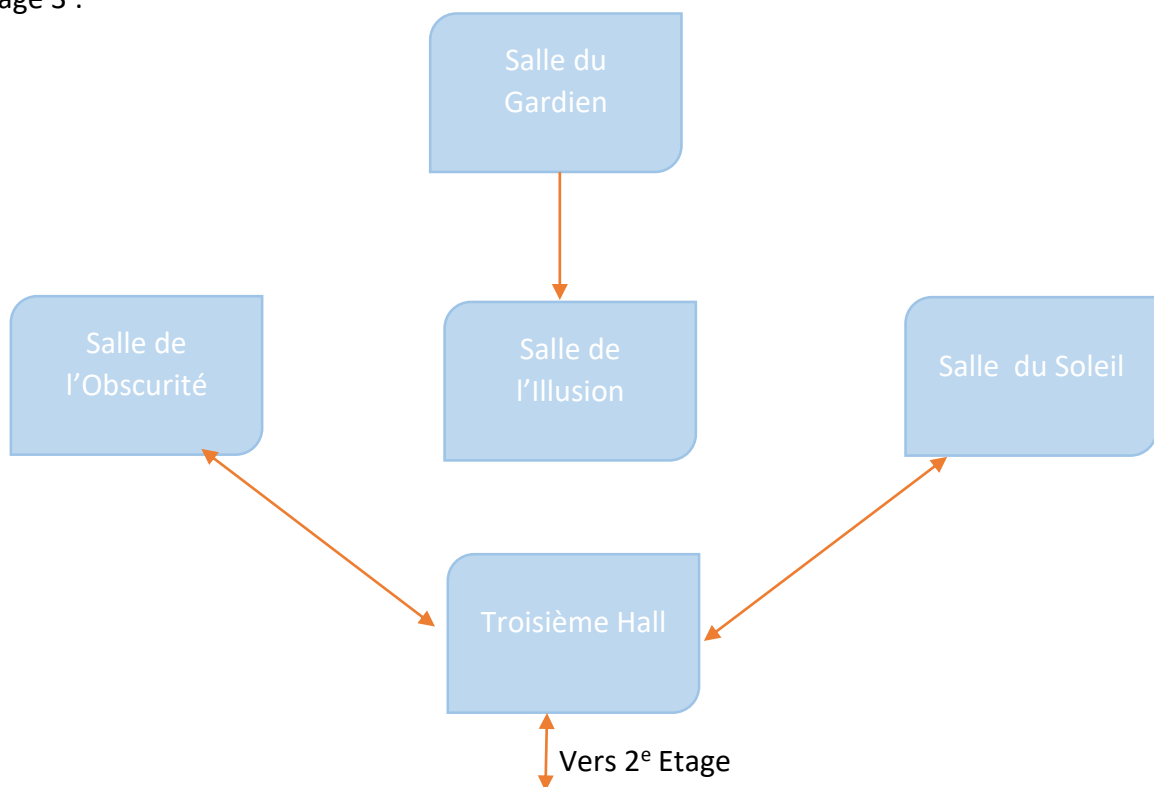
Etage 1 :



Etage 2 :



Etage 3 :



## E – Scénario détaillé

De nos jours, chaque continent possède une Tour Perdu gardant une relique Sacrée et ses mystères. Il existe 5 de ses Tours dans le monde : en Amérique, en Europe, en Afrique, en Asie et en Océanie. Chacune protégeant une relique différente, et dont seules les personnes au sommet des gouvernements connaissent l'emplacement exactes. Ces reliques sont des objets magiques d'une grande importance qui permettent de garder l'équilibre entre les grandes nations : toute les réunir peut avoir des risques...

Mercenaire franco-britannique, Trey Thomas incarné par le joueur offre ses services aux plus offrant. Après avoir été convoqué à la Maison Blanche ainsi qu'au Palais des Elysées, il accepte l'offre venant des américains, qui le charge de récupérer la relique Sacrée d'Asie.

## F – Détail des items, personnages

### Items :

*Bracelet de stockage* : bracelet dimensionnelle qui stock les objets que le joueur décide de prendre dans une autre dimension. Il possède une limite de stockage. Il est possible d'augmenter sa capacité avec un autre item

*Parchemin d'augmentation* : cet item lié au précédent permet d'augmenter les capacités de stockage d'un bracelet

*Rune de téléportation* : rune qui lorsqu'elle est brisé peut ramener le joueur dans un endroit déjà visité par celui-ci

*Amulette de lumière* : amulette qui produit de la lumière lorsque l'on se trouve dans l'obscurité

*Plastron de téléportation incomplet* : il s'agit d'un semi talisman qui nécessite d'être assembler avec les pièces manquantes pour donner une forme complète de l'artéfact d'origine

*Fiole d'Énergie Lunaire* : c'est une fiole contenant de l'énergie magique appelé énergie lunaire concocté par un Maître Sorcier Supreme. Il s'agit d'un mélange de résidus lunaire et d'huile d'une roche : la roche de l'Éternité. C'est d'un des composants nécessaire pour assembler le Plastron

*Vanadinite* : la vanadinite est une pierre de niveau légendaire utilisé par les Sorciers Suprêmes en alchimie. Elle permet d'avoir des artéfacts de qualité qui sortent de l'ordinaire. Il s'agit du second composant nécessaire pour assembler le Plastron

*Talisman de téléportation* : le Talisman de téléportation est la forme complète (assembler) du Plastron. Il s'agit d'un téléporteur qui une fois utiliser téléporte le joueur vers le lieu préféré par le Sorcier Suprême à l'origine de sa création

Oracle : l'Oracle est une clé permettant d'ouvrir la porte blindé du premier étage

Obsidienne : l'Obsidienne et la second clé permettant d'ouvrir la second porte blindé se trouvant au deuxième étage de la Tour.

Sandana : épée Maudite de niveau normal faite en zinc, et marqué par sa grande durabilité. On raconte qu'elle aurait appartenu au Créateur des 5 Tours, ainsi que des reliques. Elle aurait donc plus de 2000 ans.

Bague de l'Illusion : relique de la Tours asiatique, il s'agit d'une bague qui permet à son utilisateur de plongé toute personne désirer dans un état qui le fait vivre dans un monde purement fictif

### **Personnages :**

L'Alpha :

Le Gardien :

Le protecteur :

Maitre Martial :

L'ancienne :

Adam :

### G – Situations gagnantes et perdantes

Si le joueur récupère la relique avant que les forces armées ne le rattrapent (fin du temps), il gagne la partie.

Au contraire, les forces armée vont progresser dans la Tour à la poursuite du joueur, il sera donc compliqué de revenir sur ces pas à certains moment (chaque objet trouvé peut être important ^^). Si l'armée rattrape le joueur (se trouve dans la même pièce que lui ?) le joueur perd la partie.

### H – Eventuellement énigmes, mini-jeux, combats, etc.

### I – Commentaires

Il peut être important de lire les changements au début du II) ET d'aller au Mode d'emploi ( III) ) pour comprendre certaines choses.

## II) Réponses aux exercices



### Changement :

Après consultation de Monsieur Bureau, j'ai eu l'autorisation de modifier l'inventaire : pour pouvoir prendre un item, le joueur devra absolument posséder l'item *Bracelet* qui se trouve dans la première pièce du jeu. Si le joueur décide de ne pas le prendre, il ne pourra rien ajouter à son inventaire qui est le Bracelet de stockage (voir détails des items). Par ailleurs, une fois ramassé, le joueur n'aura pas la possibilité de déposer cet item qui constitue son inventaire.

Un autre changement concernant la façon d'accéder à la salle final est fait ( voir I) D) ainsi que le III) ).

### Exercice 7.5 :

Dans cet exercice, on décide de créer une procédure *printLocationInfo()* qui a pour but d'afficher la description de la salle où se trouve le joueur, ainsi que les directions des sorties disponible. On nous demande de créer cette procédure dans le but d'éviter la duplication de code. En effet, la situation était inacceptable puisque dans le cas de modifications de cette partie du code, le développeur devra modifier tous les endroits du code qui refont la même action (ici afficher la description et les sorties). Ce qui pourrait créer des incohérences en cas d'erreur et augmenterait la charge de travail.

### Exercice 7.6 :

Dans cet exercice, on a créé un accesseur *getExit()* aux directions pour diminuer le couplage de la classe Game avec la classe Room. Puis on modifie la méthode *goRoom()* de la classe Game pour réduire la duplication de code avec l'instruction :

```
Room vNextRoom = this.aCurrentRoom.getExit(vDirection);
```

### Exercice 7.7 :

Pour l'exercice 7.7, on a créé une méthode *getExitString* dans la classe Room qui retourne une chaîne de caractère contenant les sorties disponibles en étant dans la salle courante. Cela a ainsi permis de faire passer les attributs de la classe Room en *private* pour éviter que la classe Game n'accède à ces derniers. Il est donc logique de demander à Room de produire les informations et à Game de les afficher pour éviter un couplage élevé entre ces classes et donc éviter d'avoir à faire des modifications conséquentes sur le code lorsqu'il y a de la duplication de code.

### Exercice 7.8 :

Dans cet exercice, on rajoute un objet de type HashMap dans le code de la classe Room. De plus, on modifie aussi l'accesseur *getExit()* ainsi que le modificateur avec les indications du chapitre 7.

Après modification, on remarque que la méthode *setExits()* devient inutile dans la classe Room puisqu'elle a été remplacé par le modificateur *setExit()* qui lui, définit les sorties des pièces une par une. Ce qui est plus pratique dans le cas de l'ajout de nouveaux déplacements que de faire un modificateur qui modifie toutes les directions d'un coup par bloc (avec plusieurs paramètres).

#### Exercice 7.8.1 :

Pour cet exercice, je n'ai pas ajouté dans le scénario de lieu à une hauteur différente de l'étage de départ puisque mon scénario possède déjà différents lieux à différentes hauteurs (une Tour). Néanmoins, j'ai rajouté dans *createRoom()* de la classe Game les étages supérieurs dans mon code.

#### Exercice 7.9 :

Dans cet exercice, nous avons dû modifier la méthode *getExitString* dans la classe Room puisqu'on avait modifié dans l'exercice 7.8 l'attribution des sorties dans le modificateur *setExit()*. Le *keySet* retourne toutes les clés de la *HashMap*.

#### Exercice 7.10 :

La méthode *getStringExit()* permet d'afficher sous forme d'un chaîne de caractères les sorties disponibles à partir de la pièce où l'on se trouve. Pour cela, elle va prendre les noms des sorties qui se trouvent dans les clés de la *HashMap* et mettre l'ensemble de ces clés dans une variable intermédiaire *vKeys* de type *String*. Puis à travers un *for each*, elle va mettre chaque élément de *vKeys* dans une variable de type *String* (déclarer dans la boucle) pour pouvoir les concaténer à la variable qui retourne les sorties lorsque cette dernière est disponible. Après cela, elle le retourne.

#### A Savoir Expliquer :

- ✚ **HashMap** : La *HashMap* est une table de hachage qui associe des données par paire : une Clé et une valeur. Il s'agit d'un tableau associatif auquel on peut rajouter/modifier/supprimer des valeurs. Une valeur peut être attribuée à différentes clés et donc apparaître plusieurs fois, tandis qu'une clé ne peut apparaître qu'une seule fois. Il s'agit d'un tableau, mais à la différence du tableau classique, elle n'a pas de taille lors de sa déclaration. De plus, on peut associer des clés avec des valeurs de n'importe quel type.
- ✚ **Set** : *Set* désigne un ensemble, une collection qui n'a pas de doublons.
- ✚ **keySet()** : Le *keySet* liste et retourne l'ensemble de toutes les clés contenues dans la table de hachage.

#### Exercice 7.10.1 :

Amélioration de la documentation java.

#### Exercice 7.10.2 :

La documentation de la classe Game contient beaucoup moins de méthodes que celle de la classe Room puisque seul les méthodes *public* apparaissent sur cette dernière. Or la classe Game contient moins de méthodes *public* que celle de Room, d'où la différence.

#### Exercice 7.11 :

Dans cet exercice, on se rend compte que l'affiche de la description dans *printLocationInfo()* n'était pas optimale pour de futures implémentations. On a donc ajouté dans la classe Room une méthode *getLongDescription()* qui affiche la description de la salle ainsi que les sorties disponibles (par un appel de *getExitString()* pour éviter la duplication de code).

#### Exercice 7.14 :

Dans cet exercice, nous avons ajouté une nouvelle commande : look. Pour ce faire, nous avons dû rajouter dans la classe *CommandWords* une nouvelle case dans le tableau *aValidCommands* pour pouvoir ajouter la commande « look ».

```
*/  
public CommandWords()  
{  
    this.aValidCommands = new String[4];  
    this.aValidCommands[0] = "go";  
    this.aValidCommands[1] = "help";  
    this.aValidCommands[2] = "quit";  
    this.aValidCommands[3] = "look";  
} // CommandWords()
```

Puis, nous avons dû créer une procédure privée *look* dans la classe Game pour afficher ce que le joueur regarde. Après cela nous avons modifié la méthode *processCommand()* pour indiquer dans quelle méthode rentrer lorsque l'on tape la commande look. De plus, ayant décidé de faire la partie optionnel de l'exercice, on a dû ajouter dans le cas de la commande look la vérification de la présence du second mot comme ceci :

```

else if ( pCommand.getCommandWord().equals("look") ){
    if ( pCommand.hasSecondWord() ){
        System.out.println("Je ne sais pas comment regarder " +
            "quelque chose en particulier pour le moment");
        return false;
    }
    this.look();
    return false;
}

```

Par ailleurs, après cela, l'ajout dans *printHelp()* de 'look' dans le message d'aide des commandes était alors nécessaire pour rester cohérent.

```

System.out.println("\n\nVos mots de commandes sont:\n go quit help look");

```

#### Exercice 7.15 :

Dans cet exercice comme dans le précédent, nous avons ajouté une nouvelle commande : eat. Pour ce faire, nous avons donc là aussi modifié la classe *commandWord* pour ajouter le mot de commande à la liste avec :

```

this.aValidCommands [4] = « eat » ;

```

Puis nous avons ajouté une procédure privée qui affiche le message après le repas puisque la méthode *processCommand* a pour but de faire l'appel de la méthode correspondant à la commande rentré en paramètre.

```

/**
 * Procédure privée qui affiche que l'on a manger
 */
private void eat()
{
    System.out.println("Vous avez mangé maintenant et " +
        "vous n'avez plus faim.");
} // eat()

```

Enfin, nous avons ajouté dans *processCommand* le test de la commande et son appel :

```

else if (pCommand.getCommandWord().equals("eat")){
    this.eat();
    return false;
}

```

De plus une mise à jour de *printHelp()* a été faite :

```

System.out.println("\n\nVos mots de commandes sont:\n" +
    " go quit help look eat");

```

### Exercice 7.16 :

L'exercice nous fait se rendre compte que rentrer les mots de commandes manuellement dans *printHelp()* n'est pas la meilleure des techniques. On a donc dans cet exercice créé une méthode *showAll()* dans la classe *CommandWords* pour afficher chaque mot de commandes du tableau de commande. Par la suite, on nous fait ajouter dans la classe *Parser* une méthode (*showCommands()*) pour afficher les commandes valides. Enfin, nous avons ajouté dans *printHelp* de *Game* l'appel de la méthode *showCommands()* sur l'attribut *Parser* de *Game*.

```
System.out.println("\n\nVos mots de commandes sont:\n");
this.aParser.showCommands();
```

### A Savoir Expliquer :

for each ( typeElement vElement : tableau) / for each ( TypeElement vElement : ensemble) :  
Il s'agit d'une boucle for each, c'est-à-dire qui prend chaque élément de l'ensemble ou du tableau et le mets dans une variable intermédiaire vElement. Il n'y a pas de différence entre ces deux boucles, mis à part le fait que dans le premier cas on parcourt un tableau, tandis que dans le deuxième, il s'agit d'un ensemble.

### Exercice 7.18 :

Dans cet exercice, on modifie la procédure *showAll()* pour qu'elle n'affiche plus les mots de commandes, mais pour qu'elle les retourne pour laisser à la classe *CommandWord* gérer la donnée (ici des mots commandes) elle-même. Pour cela, on change le type de retour de la méthode et on rajoute une variable intermédiaire de type *String*, *vCommandString* dans lequel on va rajouter les mots de commandes avec le for each. Puis on le retourne : la méthode qu'on renomme *getCommandList()* :

```
public String getCommandList()
{
    String vCommandString = "";
    for (String vCommand : this.aValidCommands){
        vCommandString += vCommand + " ";
    }
    return vCommandString;
} // getCommandList()
```

Pour continuer, nous avons dû modifier dans la classe *Parser* la méthode *showCommands()* pour qu'elle retourne les commandes en appelant *showAll()* nouvellement appeler *getCommandList()*. Nous avons donc dû modifier la aussi son type

de retour en String puisqu'une procédure ne retourne rien, mais aussi son nom puisqu'elle n'affiche plus rien :

```
public String getCommands()
{
    return this.aValidCommands.getCommandList();
} // getCommands()
```

Enfin, nous avons rajouté un *System.out.println* dans *printHelp* pour permettre à la classe *Game* d'afficher les commandes :

```
System.out.println(this.aParser.getCommands() );
```

#### Exercice 7.18.1 :

Dans cet exercice, on se contente seulement de comparer si ce que l'on fait dans notre *zuul-v4* correspond avec ce qui est fait dans le fichier *zuul-better.jar*, et d'y apporter de petite modification lorsque cela est nécessaire.

#### Exercice 7.18.3 :

Recherche de tilesets pour la création des images des pièces du jeu avec Unity3D.

#### Exercice 7.18.4 :

Choix du titre du jeu : *The Ring and the Lost Tower*

#### Exercice 7.18.5 :

Pour cet exercice, on nous demande d'ajouter un tableau, une liste, ou une HashM pour pouvoir accéder aux différentes pièces même en dehors de la méthode *createRooms()*. Pour ce faire, on décide alors de rajouter une HashMap dans la classe *Game*. On crée alors un attribut de type HashMap après avoir importé cette classe avant la classe *Game* :

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

Puis l'initialisation dans le constructeur :

```

public Game()
{
    this.createRooms();
    this.aParser = new Parser();
    this.aRooms = new HashMap<String, Room>();
} //Game()

```

On a ensuite rajouter dans *createRooms()* les différentes salles dans la table de Hashage avec des instructions comme celle-ci :

```
this.aRooms.put("Hall", vHall);
```

#### Exercice 7.18.6 :

Dans cet exercice, nous avons dû modifier la majorité de notre travail pour ajouter l'interface graphique. Pour cela, nous avons fait les modifications demandées avec le zuul-with-images.

On a donc d'abord dû ajouter un attribut à la classe Room (private String aImageName) pour le nom de l'image, puis un accesseur pour pouvoir accéder à ce nom. De plus, il a fallu rajouter un paramètre au constructeur naturel pour pouvoir initialiser le nouvel attribut :

```

public Room( final String pDescription, final String pImageName )
{
    this.aDescription = pDescription;
    this.aExits = new HashMap<String, Room>();
    this.aImageName = pImageName;
} //Room (.)

```

Il a ensuite fallu modifier la classe Parser pour faire une version sans Scanner. Nous avons donc dû supprimer tous ce qui était en rapport avec Parser, et le remplacer par des appels aux méthodes de la classe StringTokenizer. La prochaine étape était de transférer les méthodes de la classe Game (sauf le constructeur) dans une nouvelle classe : GameEngine. On a donc modifié le constructeur par défaut de Game ainsi que ses attributs comme dans le zuul-with-images.

Pour continuer, on modifie certaines méthodes de la classe GameEngine, on en rajoute une (le modificateur *setGUI()*), on remplace *quit()* par *endGame()*, on supprime *play()*, on renomme *processCommand()* par *ineterpretCommand()* et on modifie les affichages : non plus System.out.println, mais this.aGui.println .

Enfin, on rajoute la classe UserInterface ainsi que ses méthodes. Dans cette dernière, il a fallu vérifier que les imports était nécessaire, et ainsi remplacer l'import du package par celui des classes qui en présentaient le besoin. On se retrouve donc avec une liste d'import comme celle-ci pour le moment :

```

import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JTextArea;
import javax.swing.JLabel;
import javax.swing.ImageIcon;
import javax.swing.JScrollPane;
import javax.swing.JPanel;

import java.awt.Dimension;
import java.awt.BorderLayout;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
|
import java.net.URL;

```

#### Exercice 7.18.7 :

`addActionListener()` : Il s'agit d'une méthode de l'interface `ActionListener` qui permet d'ajouter un objet comme écouteur d'action d'un autre objet. Permettant ainsi la réaction lorsque l'action est effectuée.

`actionPerformed()` : cette méthode de l'interface `ActionListener` redéfini dans `UserInterface` vérifie qu'un évènement eu lieu, qu'une action s'est produite. Si c'est le cas, elle appelle `processCommand()` pour pourvoir interpréter la commande.

#### Exercice 7.18.8 :

Pour cet exercice, il a fallu rajouter un bouton. On se sert alors de la classe `JButton` du JDK pour réaliser cet exercice. La première étape est l'import de la classe au début de la classe `UserInterface` avec la ligne suivante :

```
import javax.swing.JButton;
```

Ensuite, il a fallu créer un attribut de type `JButton` appelé `aButton`, puis l'initialiser dans `createGUI()` avec en appelant le constructeur permettant de nommer le bouton. On décide alors de créer un bouton d'aide :

```
// Création d'un bouton d'aide
```

```
    this.aButton = new JButton("aide");
```

Il a ensuite été nécessaire d'ajouter le bouton au Panel en utilisant une des places restantes :



```
vPanel.add( this.aButton, BorderLayout.WEST );
```

Enfin nous avons dû ajouter un écouteur d'action pour le bouton `aButton` avec :

```
this.aButton.addActionListener( this );
```

Et modifier `actionPerformed()` pour que s'il l'on appuie sur le bouton, il effectue la commande associé au bouton au nom de ce dernier à l'aide d' `interpretCommand()` de `GameEngine`, et dans le cas contraire, s'il ne s'agit pas d'un clic mais d'un ENTREE sur le clavier, il effectue la commande nécessaire à l'aide de `processCommand()`.

```
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()).getClass() == this.aButton.getClass() ){
        this.aEngine.interpretCommand( pE.getActionCommand() );
    } else this.processCommand(); // never suppress this line
} // actionPerformed(.)
```

A Savoir Expliquer :

- + ActionListener : cette interface permet de recevoir les actions qui ont eu lieu
- + addActionListener() : cette méthode permet d'ajouter un « écouteur » sur un composant pour pouvoir l'écouter lorsqu'il réagira a un événement, par exemple le clic sur un bouton.
- + actionPerformed() : méthode qui permet lorsqu'un événement se produit de vérifier la source de l'évènement et d'agir en conséquence
- + ActionEvent : cette classe indique si un évènement généré par un composant a eu lieu
- + getActionCommand() : méthode qui permet de récupérer sous forme de chaîne de caractère la commande de l'action. Dans le cas de boutons, elle récupère le nom qui est inscrit sur le bouton.
- + getSource() : méthode qui retourne une référence de l'objet sur lequel il est appelé

Exercice 7.19.2 :

Création d'un dossier Images dans le projet zuul-v4 dans lequel on met les images des différentes pièces du jeu.

Exercice 7.20 :

Le but de cet exercice est d'ajouter un item dans une salle. Pour ce faire, on crée une nouvelle classe `Item` qui possède trois attributs : un pour la description de l'item (*aItemDescription*), un autre pour son poids (*aItemWeight*) et un dernier pour son prix

(*ItemPrice*). On ajoute de plus un constructeur naturel et trois accesseurs aux attributs. On passe ensuite dans la classe *Room* où on ajoute un attribut de type *Item* *aItem*. On crée ensuite un modificateur pour pouvoir mettre un item dans une salle :

```
public void setItem(final Item pItem )
{
    this.aItem = pItem;
} // setItem(.)
```

Pour continuer, on modifie la méthode *getLongDescription()* pour qu'elle nous affiche en plus de la description de la pièce le nom de l'item disponible, sinon qu'il n'y a pas d'item :

```
public String getLongDescription()
{
    return "Vous êtes " + this.getDescription() + "\n"
        + this.getExitString() + "\n" + this.getItemString();
} // getLongDescription()
```

Pour ce faire, on rajoute un accesseur dans *Room* *getItemString()* qui retourne l'item disponible dans la pièce sous forme de chaîne de caractère, et que l'on va appeler dans *getLongDescription()* pour respecter la cohésion des méthodes.

Enfin, on crée dans *createRooms()* de *GameEngine* un nouveau item puis on l'ajoute à une salle :

```
// Création d'un item du premier étage
Item vBracelet = new Item("Bracelet de stockage", 10);
vEquipment.setItem(vBracelet);
```

#### Exercice 7.21 :

Toutes les informations à propos d'un *Item* doivent être produites dans cette même classe qui est responsable de ses données : il s'agit de la conception dirigée par les responsabilités. La classe produisant la *String* décrivant l'Item est donc produite dans la classe *Item*. La classe *GameEngine* est celle qui se chargera de son affichage car il s'agit du moteur du jeu, et donc se charge de l'utilisation de toutes les données qui sont utiles au jeu. On rajoute par ailleurs une méthode *getLongItemDescription()* dans *Item* qui affiche une description longue de l'item.

```
public String getLongItemDescription()
{
    return "Item disponible : " + this.getItemDescription() +
        "\nPoids " + this.getItemWeight() + "\nPrix : " +
        this.getItemPrice();
}
```

### Exercice 7.21.1 :

Pour cet exercice, on veut améliorer la commande *look* pour pouvoir regarder un item. Pour cela on rajoute d'abord un attribut *aItemName* dans la classe *Item* ainsi qu'un accesseur à cet attribut. De plus on modifie *getLongItemDescription()* comme ceci :

```
public String getLongItemDescription()
{
    return "Item disponible : " + this.getItemName() +
        "\n" + this.getItemDescription() + "\nPoids " +
        this.getItemWeight() + " grammes | Prix : " +
        this.getItemPrice() + " Coins";
} // getLongItemDescription()
```

On ajoute donc par la suite un accesseur *getItem()* dans *Room* qui retourne le nom de l'item présent dans la pièce. On passe ensuite dans *interpretCommand()* de *GameEngine* pour vérifier s'il y a un second mot, et si ce second mot est le nom d'un item pour regarder l'item.

```
} else if ( vCommand.getCommandWord().equals("regarder") ){
    if ( vCommand.hasSecondWord() ){
        if(vCommand.getSecondWord().equals(this.aCurrentRoom.getItem()) ){
            this.aGui.println(this.aCurrentRoom.getItemString());
            return;
        }
        else
        {
            this.aGui.println("Je sais seulement regarder un item");
            return;
        }
    }
    this.look();
}
```

### Exercice 7.22 :

Pour cet exercice, on veut rajouter la possibilité de mettre plusieurs items dans une pièce. Pour cela, on ajoute un attribut *HashMap* contenant les items de la salle associé à leurs noms dans la classe *Room* :

```
private HashMap<String,Item> aItems;
```

Après l'avoir initialiser dans le constructeur de *Room*, on modifie la méthode *setItem()* qui devient *addItem()*, et qui ajoute un *Item* à la table de hachage de la pièce lorsque que l'on en ajoute un dans une salle.

```

/**
 * Procédure qui attribue un item a une salle
 */
public void addItem( final Item pItem ) //anciennement setItem()
{
    this.aItems.put(pItem.getItemName(), pItem);
} // addItem(.)

```

Enfin, on modifie la méthode *getItemString()* pour qu'elle parcoure la HashMap, prend l'item associé au nom, puis retourne la description longue de l'item lorsqu'il existe d'un ou des item(s) sous forme de chaîne de caractère.

```

public String getItemString()
{
    String vItems = "Item(s) disponible(s) :\n";

    if (this.aItems.size() > 0) {
        Set<String> vKeys = this.aItems.keySet();

        for(String vItemName : vKeys){
            Item vReturn = this.aItems.get(vItemName);
            vItems += vReturn.getLongItemDescription();
        }

        return vItems;
    }

    return "Il n'y a pas d'item ici";
} // getItemString()

```

Par ailleurs, il a aussi fallu modifier dans la classe *GameEngine* le 'look item' pour l'adapter avec le fait de pouvoir avoir plusieurs items dans une pièce. On modifie donc *getItem()* de *Room* pour qu'il retourne l'item de la HashMap portant le nom entré en paramètre. Puis on rentre modifier *interpretCommand()*.

```

} else if ( vCommand.getCommandWord().equals("regarder") ){

    if ( vCommand.hasSecondWord() ){

        if ( this.aCurrentRoom.getItem(vCommand.getSecondWord()) == null ){
            this.aGui.println("Je sais seulement regarder un item existant !");
            return;
        } else{
            this.aGui.println( (this.aCurrentRoom.getItem(vCommand.getSecondWord()) ).getLongItemDescription() );
            return;
        }
    }

    this.look();
}

```

#### Exercice 7.22.2 :

Ajout de certains items dont plusieurs dans une salle.

### Exercice 7.23 :

Dans cet exercice, on veut rajouter une commande « back ». Pour cela, on rajoute dans la classe `CommandWord` une case à notre attribut *aValidCommand*, puis on ajoute la case contenant le mot de commande « retour ». On passe ensuite dans la classe `GameEngine` où on ajoute un attribut *aPreviousRoom* de type `Room` pour mémoriser la salle précédente. Pour cela, on mémorise cette salle lorsque l'on change de salle dans *goRoom()* :

```
this.aPreviousRoom = this.aCurrentRoom;  
this.aCurrentRoom = vNextRoom;  
this.printLocationInfo();
```

On ajoute ensuite une procédure privée *back()* qui change la salle actuelle et affiche sa localisation :

```
private void back()  
{  
    this.aCurrentRoom = this.aPreviousRoom;  
    this.aPreviousRoom = null;  
    this.printLocationInfo();  
} // back()
```

Enfin, on ajoute un test dans *interpretCommand()* pour vérifier si le mot de commande est « retour ».

### Exercice 7.24 :

Après le test de la commande, si le joueur rentre un deuxième mot de commande, la commande n'est pas effectuée, et rien n'est affiché. Il devrait au moins y avoir un message qui prévient le joueur qu'il n'est pas possible de retourner dans une pièce en particulier. On peut voir de plus un autre cas de mauvais test : le fait d'utiliser la commande « retour » dans la première pièce du jeu lors de son lancement par exemple, qui n'est pas empêché.

### Exercice 7.25 :

S'il on se déplace trois fois et que l'on rentre la commande « retour » une première fois, on retourne dans la pièce précédente et non dans la première pièce de départ. Dans le cas où on tape cette commande une deuxième fois, puisque le premier appel renvoie dans la salle précédente, l'attribut *aPreviousRoom* devient alors *null*. Ainsi, lorsque l'on rentre « retour » une deuxième fois, on met quelque chose de *null* dans l'attribut de la pièce courante *aCurrentRoom*. On se retrouve donc avec une *nullPointerException* qui induit alors le fait que plus certaines commandes tel que « regarder » ne fonctionnent.

### Exercice 7.26 :

Pour cet exercice, on améliore la commande de retour pour pouvoir stocker dans une pile toutes les salles parcourues, et ainsi pouvoir retourner en arrière jusqu'à la salle de départ si l'envie nous vient. On va donc pour cela importer la classe Stack :

```
import java.util.Stack;
```

Puis modifier l'attribut *aPreviousRoom* de type Room en pile de pièce, *aPreviousRooms*, qui sera initialiser dans le constructeur de la classe GameEngine avec

```
this.aPreviousRooms = new Stack<Room>();
```

Pour continuer, on modifie dans *goRoom()* la façon de mettre une pièce dans la pile en utilisant la méthode *push()* de la classe Stack.

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

Viens ensuite la modification de la procédure *back()* où l'on va prendre une pièce et l'enlever de la pile, puis la retourner et la mettre dans *aCurrentRoom*.

```
private void back()
{
    this.aCurrentRoom = this.aPreviousRooms.pop();
    this.printLocationInfo();
} // back()
```

Enfin, on modifie dans *interpretCommand()* la partie concernant le retour pour afficher un message si la ligne de commande possède un second mot, empêcher le retour si la pièce courante est la première pièce du jeu, sinon effectuer le retour.

```
else if ( vCommand.getCommandWord().equals("retour") ){
    if ( vCommand.hasSecondWord() ){
        this.aGui.println("La commande 'retour' n'accepte pas de second mot !");
        return;
    }
    if ( this.aCurrentRoom == this.aRooms.get("Hall") ){
        this.aGui.println("Il est impossible de revenir en sur ses en étant pas dans la pièce de départ");
        return;
    }
    this.back();
}
```

### A Savoir Expliquer :

- Stack : Stack est une classe qui représente des objets sous formes de pile d'objets classés du premier rentré dans cette dernière au dernier rentré.
- push() : Il s'agit d'une méthode qui permet d'ajouter un objet dans la pile, en particulier en haut de celle-ci. Elle prend donc en paramètre l'objet à ajouter.
- pop() : Cette méthode de la classe Stack prend le dernier objet de la pile qui est celui en haut de celle-ci, le retire de cette dernière et retourne l'objet en question.

- ✚ `empty()` : Méthode de type de retour booléen qui permet de vérifier si la pile est vide ou non.
- ✚ `peek()` : Méthode qui permet de regarder l'objet en haut de la pile sans le supprimer, au contraire de `pop()`.

#### Exercice 7.26.1 :

On génère les java docs utilisateur et programmeur en veillant à ne pas avoir de messages d'erreurs ou de warnings.

#### Exercice 7.28.1 :

On veut ajouter dans cet exercice une commande « test » qui lira un fichier de texte .txt qui se trouve dans le répertoire du jeu, et qui exécutera les commandes de ce-dernier. Pour cela, comme pour toutes les autres commandes, on ajoute dans le constructeur naturel de `CommandWords` une nouvelle case dans notre tableau de commandes valides contenant la petite dernière : `test`.

Pour continuer on passe dans `GameEngine` ou on va importer les classes qui nous seront utiles :

```
//pour la méthode test
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
```

Ensuite, on crée une procédure privé `test()` qui prend en paramètre une `String` correspondant au nom du fichier. Dans cette procédure, on va créer une variable de type `Scanner` qui représente le fichier de texte. On va ensuite traiter dans `try` est `catch` les exceptions. Dans la première, on va créer un nouveau `Scanner` contenant le fichier .txt qui sera lu, puis tant qu'il y a une ligne, on mettra celle-ci dans un variable de type `String` pour pouvoir exécuter la commande dans `interpretCommand()`. Dans le cas où le fichier de texte n'existe pas ou est introuvable dans le répertoire du jeu `zuul-v4`, on va afficher à l'aide du `catch` un message dans la console indiquant que le fichier est introuvable.

```

private void test( final String pFileName)
{
    //création d'une variable qui représente le fichier de texte
    Scanner vFileText;

    //Association du fichier a la variable vFileText
    //try : pour essayer les instructions
    try {
        vFileText = new Scanner( new File(pFileName + ".txt") );
        //while : pour traiter la ligne lue
        while ( vFileText.hasNextLine() ){
            String vLine = vFileText.nextLine();
            this.aGui.println(vLine);
            this.interpretCommand( vLine );
        }
    }
    catch ( final FileNotFoundException pFNFE ){
        System.out.println(pFNFE);
    }
} // test()

```

Pour terminer, dans *interpretCommand()*, on ajoute un cas à notre if : si le mot de commande est « test ». Dans ce cas, s'il y a un second mot, on appelle la nouvelle méthode, le cas échéant, on affiche un message :

```

} else if ( vCommand.getCommandWord().equals("test") ){
    if ( vCommand.hasSecondWord() ){
        this.test(vCommand.getSecondWord());
    } else {
        this.aGui.println("Je ne sais que tester des fichiers .txt");
        return;
    }
}

```

#### A Savoir Expliquer :

- ✚ lecture simple de fichiers de texte : dans cette section de *Plus de technique*, dans une classe de lecture de fichier, une procédure pratique à utiliser est créée possédant un paramètre : le nom du fichier. Dans cette procédure est créé un objet de type Scanner pour qui représente le fichier de texte dans lequel on veut lire les instructions. Cette variable est initialisée dans *try*, qui crée un nouveau fichier avec le nom du fichier pour pouvoir le lire. Ensuite, tant que ce fichier possède une prochaine ligne, on met celle-ci dans une variable de type String, puis on traite cette ligne (dans notre cas avec *interpretCommand()*). Ensuite, on va traiter une exception dans *catch* : dans le cas où le fichier est introuvable on doit afficher un message.



- ✚ File : il s'agit d'une classe qui représente les chemins d'accès des fichiers
- ✚ Scanner : la classe Scanner permet de lire les données issues d'une entrée pour pouvoir traiter les données lues lignes par lignes. A la différence de la version de Scanner qui était utiliser dans l'ancienne version de Parser, cette nouvelle version lis les données non pas issues de l'entrée standard qui est le clavier, mais plus tôt ceux issues d'un fichier de texte.
- ✚ hasNextLine() : c'est un booléen de la classe Scanner qui permet de vérifier la présence d'une prochaine ligne dans l'entrée du Scanner. Elle retourne vrai s'il y en a une sinon faux.
- ✚ nextLine() : cette méthode de la classe Scanner permet de récupérer la prochaine ligne de l'entrée du Scanner et de la retourner sous forme de String.
- ✚ traitement d'une exception du JDK : une exception est un événement exceptionnel, une erreur qui peut survenir lors de l'exécution d'instructions. Pour la traiter, on peut utiliser des blocs d'instructions. Un bloc *try* qui teste les instructions susceptibles de générer une exception. Dans le cas où une exception intervient, elle est attrapée puis traiter dans un bloc *catch*. Il existe plusieurs types d'exceptions, par exemple la *FileNotFoundException* qui indique qu'un fichier à essayer d'être ouvert sans succès en suivant le chemin d'accès.

#### Exercice 7.28.2 :

On ajoute des fichiers de texte dans cet exercice : un fichier **court** pour tester la nouvelle commande, un fichier **gagner** pour tester les commandes minimales pour gagner, et enfin un fichier **exploration** pour explorer les lieux est tester les autres commandes.

#### Exercice 7.29 :

Dans le cadre de cet exercice, on effectue une réingénierie. On veut donc ajouter une classe Player qui s'occupera de tout ce qui concerne le joueur, laissant ainsi au moteur de jeu le rôle des affichages et de la gestion de commandes. On crée donc une classe Player possédant deux champs : *aPlayerName* et *aMaxWeight*, le nom et le poids maximum d'objets que le joueur pourra transporter. De plus, on déplace de *GameEngine* vers la nouvelle classe tout ce qui concerne le Joueur. On commence donc par déplacer les attributs *aCurrentRoom* ainsi que *aPreviousRoom* et l'import de la *Stack*.

```

2 import java.util.Stack;
3 public class Player
4 {
5     private String aPlayerName;
6     private int aMaxWeight;
7     private Room aCurrentRoom;
8     private Stack<Room> aPreviousRooms;
9     private UserInterface aGui;
10
11     /**
12      * Constructeur qui initialise le nom du joueur et le poids maximum
13      * qu'il peut porter
14      * @param pPlayerName le nom du joueur
15      * @param pMaxWeight le poids maximum à emporté par le joueur
16      */
17     public Player( final String pPlayerName, final int pMaxWeight )
18     {
19         this.aPlayerName = pPlayerName;
20         this.aMaxWeight = pMaxWeight;
21         this.aPreviousRooms = new Stack<Room>();
22     } // Player(..)

```

Après avoir initialiser les attributs (sauf celui de la pièce courante dans le constructeur) on ajoute des accesseurs à ces derniers, ainsi qu'un modificateur *setCurrentRoom()* pour mémoriser et modifier la pièce courante.

```

public void setCurrentRoom( final Room pCurrentRoom )
{
    this.aPreviousRooms.push(this.aCurrentRoom);
    this.aCurrentRoom = pCurrentRoom;
} // setCurrentRoom(.)

```

En déplaçant les attributs du moteur vers la classe Player, il a fallu corriger toute les erreurs, et séparer certaines méthodes qui utiliser ces attributs comme *back()* ou *eat()* de GameEngine. On ajoute donc dans la classe Player une méthode *eat()* qui retourne un chaine de caractère indiquant que le joueur à manger, et une méthode *back()* pour retirer une salle de la pile et l'attribuer à *aCurrentRoom*. Ces deux méthodes seront donc appeler dans GameEngine lorsque nécessaire.

```

private void eat()
{
    this.aGui.println( this.aPlayer.eat() );
} // eat()

/**
 * Procédure qui change la salle courante en re
 * dans une salle précédemment visitée, et qui
 * affiche la description de la nouvelle salle
 */
private void back()
{
    this.aPlayer.back();
    this.printLocationInfo();
} // back()

```

Dans la classe `GameEngine`, on ajoute donc un attribut de type `Player` `aPlayer` pour pouvoir utiliser toute les informations produites concernant le joueur dans le moteur de jeu. On remplace donc dans `createRoom()`, `printLocationInfo()` et partout où l'on modifie la pièce courante par un appel du modificateur `setCurrentRoom()`.

Par exemple dans `createRoom()` : `this.aPlayer.setCurrentRoom(vHall);`

On remplace enfin tous les anciens appels à l'ex attribut `aCurrentRoom` de `GameEngine` par un appel de l'accessor qui se trouve dans la nouvelle classe, par exemple dans `printLocationInfo()` :

```

private void printLocationInfo()
{
    this.aGui.println( this.aPlayer.getCurrentRoom().getLongDescription() );
    if ( this.aPlayer.getCurrentRoom().getImageName() != null ) {
        this.aGui.showImage( this.aPlayer.getCurrentRoom().getImageName() );
    }
} // printLocationInfo()

```

### Exercice 7.30 :

Le but de cet exercice est de pouvoir permettre au joueur de porter un seul item, et de le déposer. Pour cela, on commence par rajouter les nouveaux mots de commandes dans `CommandWord`.

Après cela, on crée dans la classe `Player` un attribut de type `Item` : `itemTaken`. On ajoute ensuite une procédure `take()` qui prend en paramètre le nom de l'item à prendre.

```

public void take( final String pItemName )
{
    Item vItem = this.aCurrentRoom.getItem(pItemName);
    if ( vItem != null ){
        this.aItemTaken = vItem;
        this.aCurrentRoom.removeItem(pItemName);
        this.aGui.println("Vous avez pris l'item : " + pItemName);
        this.aGui.println("Vous ne pouvez plus prendre d'item !");
    } else {
        this.aGui.println("L'item que vous essayer "
            + "de prendre n'existe pas dans cette pièce !");
    }
} // take(.)

```

Elle va vérifier que l'item possédant ce nom existe bien dans la HashMap de la pièce, puis ajouter l'item en question à l'attribut *aItemTaken*. Par la suite, il faut retirer de la pièce l'item qui vient d'être pris par le joueur par l'appel d'une procédure de Room que l'on s'assure de créer dans cette classe : *removeItem()*.

```

public void removeItem( final String pItemName )
{
    this.aItems.remove(pItemName);
} // removeItem(.)

```

Pour continuer, on ajoute dans la classe *GameEngine* une procédure *take()* qui va-elle prendre en paramètre une commande et vérifier que la commande possède bien un second mot pour pouvoir appeler la méthode *take()* de *Player*.

```

public void take( final Command pCommand )
{
    if ( pCommand.hasSecondWord() ){
        this.aPlayer.take( pCommand.getSecondWord() );
    } else {
        this.aGui.println("Vous devez spécifier l'item " +
            "que vous voulez prendre.");
    }
} // take(.)

```

On ajoute enfin dans *interpretCommand()* un test qui vérifie que le mot de commande est « prendre » pour pouvoir appeler la nouvelle méthode.

On suit le même schéma pour drop : on commence par ajouter une procédure dans la classe *Player* qui prend en paramètre une String correspondant au nom de l'item à déposer. On vérifie que le joueur possède bien cet item, puis on l'ajoute dans la liste des items de la pièce courante, et on passe l'attribut *aItemTaken* à null.

```

public void drop( final String pItemName )
{
    if ( pItemName.equals( this.aItemTaken.getItemName() ) ){
        this.aCurrentRoom.addItem(this.aItemTaken);
        this.aItemTaken = null;
        this.aGui.println("Vous avez déposer l'item : " + pItemName);
        this.aGui.println("Vous pouvez à nouveau prendre un item.");
    } else {
        this.aGui.println("Vous ne possédez pas l'item : " + pItemName);
    }
} // drop(.)

```

Pour continuer, on réalise là aussi une procédure *drop()* dans *GameEngine* qui prend en paramètre une commande, et vérifie l'existence d'un second mot correspondant à l'item que le joueur décide de déposer.

```

public void drop( final Command pCommand )
{
    if ( pCommand.hasSecondWord() ){
        this.aPlayer.drop(pCommand.getSecondWord());
    } else {
        this.aGui.println("Vous ne pouvez déposer que l'item " +
            "que vous possédez !");
    }
} // drop(.)

```

Enfin on rajoute donc dans *interpretCommand()* le test qui vérifie que le mot de commande est « déposer » pour appeler la méthode *drop()*.

#### Exercice 7.31 :

On veut ajouter dans cet exercice la possibilité cette fois de permettre au joueur de porter plusieurs items. Pour cela, on modifie dans la classe *Player* l'attribut *aItemTaken* en un attribut *aInventory* de type *HashMap* qui associe le nom de l'item à l'item en question, que l'on va initialiser dans le constructeur de cette classe. Pour continuer, on ajoute trois méthode : une pour retourner un item contenue dans l'inventaire du joueur, *getInventoryItem()*, une deuxième pour pouvoir ajouter un item à cette inventaire, et une dernière pour pouvoir retirer un item de l'inventaire.

```

public Item getInventoryItem( final String pItemName )
{
    return this.aInventory.get(pItemName);
}

/**
 *
 */
private void addInventoryItem( final Item pItem )
{
    this.aInventory.put(pItem.getItemName(), pItem);
}

/**
 *
 */
private void removeInventoryItem( final String pItemName )
{
    this.aInventory.remove(pItemName);
}

```

Enfin, on modifie donc dans les procédure *take()* la ligne d'ajout de l'item à l'ancien attribut par un appel de la méthode *addInventoryItem()* qui prend en paramètre l'item à ajouter. Puis dans *drop()*, la mise à null de l'ancien attribut par un appel de la méthode *removeInventoryItem()*.

#### Exercice 7.31.1

Dans cet exercice, on crée une classe *ItemList* qui s'occupe de la gestion de la liste d'items. Pour cela, on crée la classe qui possède un attribut de type *HashMap* qui associe le nom de l'item à l'item : *altemList*. Puisque l'on veut que c'est cette classe qui s'occupe de la gestion de la liste d'items, après avoir initialisé l'attribut dans le constructeur, on ajoute les méthodes *addItemList()* et *removeItemList()* pour ajouter et retirer un item de la liste. On ajoute de plus un accesseur *getItem()* pour retourner l'item associer au nom passer en paramètre, et un accesseur *getItemListString()* qui retourne sous forme de chaine de caractère les items présent dans la liste, ainsi que leur description s'ils existent.

Pour continuer, on se dirige dans les classes *Room* puis *Player*, ou on modifie respectivement les attributs *altems* et *alInventory* pour qu'ils aient pour type de retour *ItemList*. Enfin, on modifie les méthodes qui utilisaient les anciennes versions par l'appel des bonnes méthodes dans *ItemList*.

#### Exercice 7.32 :

On nous demande dans cet exercice d'ajouter une restriction de poids au joueur pour qu'il ne puisse porter des items que jusqu'à une certaines limites de poids. Pour cela on

ajoute d'abord une méthode *getItemListWeight()* dans la classe *ItemList* pour nous retourner l'entier correspondant à la somme des poids des items que le joueur possède déjà.

```
public int getItemListWeight()
{
    int vItemsWeight = 0;
    Set<String> vKeys = this.aItemList.keySet();
    for(String vItemName : vKeys){
        Item vReturn = this.aItemList.get(vItemName);
        vItemsWeight += vReturn.getItemWeight();
    }
    return vItemsWeight;
}
```

On se dirige par la suite dans la classe *Player* où l'on rajoute un booléen *canBeTake()*, qui nous retourne 'false' si en faisant la somme du poids des objets de l'inventaire du joueur et le poids du nouveau item à prendre, elle est inférieure à l'attribut *altemWeight*. Sinon le cas échéant, il nous retourne 'true'.

```
public boolean canBeTake( final Item pItem )
{
    int vItemsWeight = this.aInventory.getItemListWeight() + pItem.getItemWeight();
    if ( vItemsWeight > this.getMaxWeight() ) return false;
    return true;
} // canBeTake()
```

Enfin, on ajoute un if qui vérifie donc dans *take()* si l'objet peut être pris en appelant le booléen.

### Exercice 7.33 :

On ajoute dans cet exercice une commande « inventaire » pour permettre au joueur d'afficher les objets qu'il possède et leur poids. Pour cela on commence par ajouter la commande dans la classe *CommandWord*. Puis on ajoute dans la classe *ItemList* qui gère la liste des items une méthode *getItemList()* qui a pour but de nous retourner une chaîne de caractère contenant le nom de chaque item, ainsi que leur poids.

```
public String getItemList()
{
    if( this.aItemList.size() > 0 ){
        String vInventory = "";
        Set<String> vKeys = this.aItemList.keySet();
        for(String vItemName : vKeys){
            Item vReturn = this.aItemList.get(vItemName);
            vInventory += "- " + vReturn.getItemName() + " (Poids : " + vReturn.getItemWeight() + " grammes)\n";
        }
        return "Vous possédez :\n" + vInventory;
    }
    return "L'inventaire est vide.";
}
```

On ajoute ensuite une méthode *getInventoryList()* dans la classe *Player* qui nous retourne la chaîne de caractère de l'inventaire *aInventory* en appelant la méthode de *ItemList* :

```
public String getInventoryList()
{
    return this.aInventory.getItemList();
}
```

Enfin, on passe dans le *GameEngine* où l'on crée une méthode *inventory()* qui affiche la chaîne de caractère retourner par l'appel de la méthode *getInventoryList()* sur l'attribut *aPlayer*. On rajoute pour terminer le test de vérification dans le cas où le mot de commande est 'inventaire' dans *interpretCommand()*.

#### Exercice 7.34 :

Dans cet exercice, on veut donner la possibilité au joueur de manger un item qui double le poids maximum qu'il peut porter. Ne s'accordant pas avec mon scénario où le joueur doit utiliser un *Parchemin d'augmentation* pour augmenter la capacité de stockage dans son bracelet, j'ai eu l'autorisation par Monsieur Bureau d'implémenter une commande « utiliser ». Cette commande a donc pour but lorsque le second mot est le Parchemin de l'utiliser pour doubler le poids maximum du joueur.

Pour cela, on commence par ajouter dans la classe *CommandWords* le mot de commande *utiliser* à notre liste de mots. Pour continuer, on se place dans la classe *Player* où on ajoute une procédure *doubleMaxWeight()* qui double le poids maximum de l'attribut.

```
private void doubleMaxWeight()
{
    this.aMaxWeight = this.getMaxWeight()*2;
}
```

Ensuite, on ajoute une procédure *use()* dans cette classe qui lorsque l'item rentré en paramètre appartient à l'inventaire, et que cet item est le Parchemin, la méthode *doubleMaxWeight()* est appelée pour doubler le poids. Puis l'item est "détruit" (retiré de l'inventaire sans être remis dans la pièce).



```

public void use( final String pItemName )
{
    Item vItem = this.getInventoryItem(pItemName);

    if ( vItem != null ){
        if ( pItemName.equals("Parchemin") ){
            this.doubleMaxWeight();
            this.removeInventoryItem(pItemName);
            this.aGui.println("Le Parchemin entre en résonnance avec votre Bracelet de Stockage."
                             + "\nUne lueur vous aveugle ! La capacité de votre Bracelet viens d'être doubler !!");
        } else {
            this.aGui.println("Vous pouvez seulement utiliser l'item : Parchemin");
        }
    } else {
        this.aGui.println("L'item que vous essayer d'utiliser n'existe pas dans votre inventaire.");
    }
}

```

Enfin, on se dirige dans `GameEngine` où on crée une procédure qui vérifie que la commande est à un second mot, puis appelle le `use()` de `Player`. Puis on ajoute le test dans `interpretCommand()` dans le cas où le mot de commande est utiliser.

```

public void use( final Command pCommand )
{
    if ( pCommand.hasSecondWord() ){
        this.aPlayer.use(pCommand.getSecondWord());
    } else {
        this.aGui.println("Vous devez spécifier l'item à utiliser.");
        return;
    }
}

```

#### Exercice 7.34.1 :

On améliore les fichiers de test avec les nouvelles commandes.

#### Exercice 7.34.2 :

On génère à nouveau les java docs.

#### Exercice 7.42 :

On veut ajouter dans cet exercice une sorte de limite de temps pour terminer le jeu. Pour cela, on crée dans la classe `GameEngine` un attribut de type `int` `aLimitMouv` qui correspond au nombre de déplacements maximum que le joueur peut faire. On l'initialise donc dans le constructeur de cette classe, puis on ajoute une procédure `timeLimit()` qui lorsque le nombre de déplacements permis au joueur atteint 0, le jeu le fait automatiquement quitter, le faisant ainsi perdre.

```
private void timeLimit()
{
    if ( this.aLimitMouv == 0 ){
        this.aGui.println("Vous avez dépassé le nombre de déplacements autorisé.\nVous avez PERDU !!!");
        this.endGame();
    } else {
        this.aGui.println("Il vous reste " + this.aLimitMouv + " déplacements avant de perdre.");
    }
}
```

Pour continuer, on ajoute dans les méthodes de déplacement *goRoom()* et *back()* du moteur une ligne qui fait décrémenter la valeur de l'attribut de 1 à chaque déplacement :

*this.aLimitMouv = this.aLimitMouv - 1;*

Dans le cas de *back()*, pour éviter de réduire le compteur même lorsque le joueur rentre la commande 'retour' alors qu'il se trouve dans la pièce de départ, donc il ne fait aucun déplacement, on décide de faire passer le type de retour de *back()* de *Player* d'un *void* à un booléen.

```
public boolean back()
{
    if ( this.aPreviousRooms.empty() ){
        this.aGui.println("Il est impossible de revenir sur ses pas en étant dans le Grand Hall.");
        return false;
    }
    this.aCurrentRoom = this.aPreviousRooms.pop();
    return true;
} // back()
```

Cela permettant ainsi de vérifier dans le *back()* de *GameEngine* qu'un déplacement a bien été fait avant de réduire la valeur de l'attribut avec cette ligne :

*if ( this.aPlayer.back() ) this.aLimitMouv = this.aLimitMouv - 1;*

Enfin, on ajoute dans *printLocationInfo()* l'appel de la méthode *timeLimit()* pour tenir le joueur informé du nombre de déplacements restant.

#### Exercice 7.42.1 :

On veut ajouter un timer dans le jeu qui se base sur du temps réel. Pour cela, on importe les classes *ActionEvent* ainsi que *ActionListener* puis on implémente la dernière dans le *GameEngine*. On ajoute/modifie ensuite l'attribut *aLimitMouv* qui devient *aLimitTime* que l'on initialise dans le constructeur du *GameEngine* à 20 minutes en milliseconde. On importe de plus la classe *Timer*, puis on modifie la méthode *timeLimit()*. Elle va initialiser un timer *vTimeLimit* avec le temps et avec l'action à réaliser lorsque le temps sera écoulé.

```
private void timeLimit()
{
    Timer vTimeLimit = new Timer(this.aLimitTime, this);
    vTimeLimit.start();
} // timeLimit()
```

Ensuite, on ajoute une procédure `actionPerformed` pour pouvoir indiquer que si la source de l'action à la même classe que le `Timer`, qu'elle est une instance de la classe `Timer`, alors on affiche un message, puis on mets fin au jeu, et on stop le timer.

```
public void actionPerformed( final ActionEvent pE )
{
    if ( pE.getSource() instanceof Timer) { // insta
        this.aGui.println("Vous avez dépassé le nomb
        this.endGame();
        ((Timer)(pE.getSource())).stop();
    }
} // actionPerformed(.)
```

#### Exercice 7.42.2 :

J'ai ajouté en plus du `BorderLayout` un `BoxLayout` (que j'ai finalement modifier en `GridLayout`) dans un nouveau panneau que j'ai créé : `vButtonPanel`. Ce panneau se trouve à l'ouest de `vPanel`, et permet de garder tous les boutons du jeu sous forme de liste type « cuisine ». Le fait de rajouter ce panneau dans le premier panneau principal permet non seulement de pouvoir gagner de la place dans un autre panneau en utilisant une place de `vPanel`. De plus, cela me laisse ainsi plus de liberté, dont la possibilité de choisir par moi-même comment je vais positionner mes objets (ici mes boutons) à l'intérieur du panneau. De plus à l'aide du JDK, j'ai modifié la couleur de mes boutons et des panneaux ainsi que de l'écriture en important la classe `Color` et en utilisant les méthodes `setBackground()` ainsi que `setForeground()`.

J'ai importer la classe `Font` pour modifier la police d'écriture que j'ai mis en Calibri pour le moment.

J'ai ajouter un nouveau panneau lié a l'inventaire et à sa commande qui est fait pour afficher les détails de l'inventaire. Pour ce faire, j'ai d'abord crée ma zone de texte puis je l'ai mise dans un panneau scrollable. Ensuite, j'ai créer un panneau d'affichage qui sera fait pour l'affichage des textes (du jeu et de l'inventaire). J'ai ajouté le panneau au centre du `vPanel` à la place de celui de l'affichage du texte, puis j'ai modifier le layout pour initialier dans mon panneau d'affichage `vAffichagePanel` un `GridLayout`. Ensuite, j'ajoute les panneaux scrollable de l'inventaire et du texte du jeu. Pour continuer, j'ai voulu que mon panneau d'inventaire apparaissent puis écris les données de l'inventaire lorsque la commande est tapé, ou lorsqu'un appuis sur le bouton associé est effectué. Et qu'il disparaisse au profit de l'autre panneau d'affichage lorsqu'une commande est rentrée, ou lorsque'un appuis sur le bouton inventaire est fait. Pour cela, j'ai ajouter une procédure `visibilityInventory()` dans `UserInterface` qui prend en paramètre un booléen et qui rend visible l'un des panneaux scrollables, et rend invisible l'autre. J'utilise cette méthode à la fin de `creatGUI()` pour rendre par défaut le panneau d'inventaire invisible, ainsi que dans la méthode `inventory()` de `GameEngine` et dans `interpretCommand()` pour le rendre invisible à l'appel des commandes.

De plus, j'ai ajouter une méthode *inverseVisibility* qui inverse la visibilité des panneaux en appelant *visibilityInventory()*, que j'utilise aussi dans *inventory()*, pour rendre le panneau visible lorsque la commande inventaire est effectué.

```
*/
public void visibilityInventory( final boolean pVisibility )
{
    this.aInventoryScroller.setVisible(pVisibility);
    this.aListScroller.setVisible(!pVisibility);
}
```

```
/**
 *
 */
public void inverseVisibility()
{
    this.visibilityInventory( !this.aInventoryScroller.isVisible() );
}
```

Enfin, j'ai ajouter une procédure *writeInventory()* qui permet d'écrire dans l'inventaire du texte en remplaçant le texte déjà écrit par le nouveau test en paramètre.

```
public void writeInventory( final String pText )
{
    this.aInventory.setText(pText);
    this.aMyFrame.pack();
}
```

J'utilise donc cette procédure dans la méthode *inventory()* de *GameEngine* pour pouvoir écrire le texte de l'inventaire dans le nouveau panneau.

J'ai par la suite ajouter une procédure *setEnabledInventoryB()* prend en paramètre un booléen et qui rend le bouton d'inventaire utilisable ou non. Par défaut, elle désactive le bouton d'inventaire, mais je l'appelle dans *take()* de *Player* pour l'activer lorsque les conditions sont remplies.

```
public void setEnabledInventoryB( final boolean pEnable )
{
    this.aButtonTab[2].setEnabled(pEnable);
}
```

J'ai ajouter un procédure dans la classe *UserInterface* qui vérifie que le panneau d'inventaire est visible, et dans ce cas modifie la police d'écriture (qui n'est pas définitif)

```

public void fontInventoryPanel()
{
    if ( this.aInventoryScroller.isVisible() ){
        this.aInventory.setFont(new Font("Calibri", Font.PLAIN, 12));
    }
} // fontInventoryPanel()

```

J'appelle ensuite cette procédure sur le aGui de la classe GameEngine lorsque je rend le panneau visible dans la méthode *inventory()*

#### Exercice 7.43 :

Dans cet exercice, on veut implémenter des trap door à notre jeu. Pour cela on doit avoir des pièces dont une sortie ne peut être prise que dans un seul sens. Concernant la commande *retour*, on veut empêcher de franchir les portes à sens unique dans l'autre sens. Pour cela, on ajoute dans la classe Room une fonction booléenne *isExit()* qui prend en paramètre une Room et qui vérifie que cette pièce passée en paramètre est une sortie de la pièce courante. Dans le cas où il s'agit d'une sortie, la fonction retournera vrai, sinon faux. Pour cela, on parcourt la HashMap *this.aExits* des directions et salles de sorties de la pièce courante. Si on trouve qu'une salle est égale à la pièce passée en paramètre, la fonction retourne vrai.

```

public boolean isExit( final Room pRoom )
{
    //vrai ou faux selon que la Room passée en
    Set<String> vKeys = this.aExits.keySet();
    for( String vExits : vKeys ){
        Room vRoom = this.aExits.get(vExits);
        if ( pRoom.equals(vRoom) ){
            return true;
        }
    }
    return false;
} // isExit(.)

```

Ensuite, on se place dans la procédure *back()* de la classe Player ou dans le cas où la pile n'est pas vide, on vérifie que la pièce se trouvant en haut de la pile est une sortie de la pièce courante avant de la retirer et de faire le déplacement :

```

public void back()
{
    if ( this.aPreviousRooms.empty() ){
        this.aGui.println("Il est impossible de revenir sur ses pas en étant dans le Grand Hall.");
    } else {
        if ( this.aCurrentRoom.isExit(this.aPreviousRooms.peek()) ){
            this.aCurrentRoom = this.aPreviousRooms.pop();
        } else {
            this.aPreviousRooms.clear();
            this.aGui.println("Vous avez franchi une porte à sens unique. Il vous est impossible de retourner en arrière.");
        }
    }
} // back()

```

### Exercice 7.44 :

Le but de cet exercice est d'ajouter un téléporteur au jeu qui mémorise la pièce dans laquelle il est chargé, puis nous y ramène une fois activé dans une autre pièce. Pour cela, on implémente à notre jeu une classe *Beamer* qui hérite de la classe *Item*. Elle possède un attribut de type *Room* *aBeamerRoom* qui permettra de mémoriser la pièce dans laquelle le beamer est chargé. Comme il s'agit d'une classe qui hérite de la classe *Item*, on passe en paramètre du constructeur de *Beamer* les attributs de la classe mère que l'on initialise avec un appel de *super*. On ajoute de plus un accesseur et un modificateur à cette attribut pour pouvoir vérifier plus tard qu'aucune pièce n'est chargée avant de charger le beamer.

```
public Beamer( final String pItemName, final String pItemDescription,
               final int pItemWeight, final int pItemPrice )
{
    super(pItemName, pItemDescription, pItemWeight, pItemPrice);
    this.aBeamerRoom = null;
} // Beamer(...)

/**
 * Accesseur à la valeur qui se trouve dans l'attribut aBeamerRoom
 * @return la valeur stocker dans l'attribut aBeamerRoom
 */
public Room getBeamerRoom()
{
    return this.aBeamerRoom;
} // getBeamerRoom()

/**
 * Modificateur qui modifie la valeur de l'attribut aBeamerRoom
 * @param pBeamerRoom la pièce utilisé pour modifier la valeur de l'attribut
 */
public void setBeamerRoom( final Room pBeamerRoom )
{
    this.aBeamerRoom = pBeamerRoom;
} // setBeamerRoom(.)
```

On se place ensuite dans la classe *Player* où on crée une procédure *chargerBeamer()* qui prend en paramètre le nom du beamer à charger. On vérifie dans cette procédure que le joueur possède l'item dans son inventaire, et que cet item est bien une instance de la classe *Beamer*. Si c'est le cas, si l'attribut de mémorisation d'une pièce du téléporteur est *null*, on charge le téléporteur.

```

public void chargeBeamer( final String pBeamerName )
{
    Item vItem = this.getInventoryItem(pBeamerName);

    if ( vItem != null ){ //le joueur possède l'item dans son inventaire
        //le joueur possède le téléporteur
        if ( vItem instanceof Beamer ){ // vérifie que vItem est une instance de 1
            Beamer vBeamer = (Beamer)(vItem);
            if ( vBeamer.getBeamerRoom() == null ){
                vBeamer.setBeamerRoom(this.getCurrentRoom());
                this.aGui.println("Vous avez chargé le téléporteur, il a mémorisé");
            } else {
                this.aGui.println("L'item " + pBeamerName + " est déjà chargé !");
            }
        } else {
            this.aGui.println("Vous pouvez seulement charger un téléporteur");
        }
    } else {
        this.aGui.println("Vous ne possédez pas l'item : " + pBeamerName);
    }
} // chargeBeamer(.)

```

Pour continuer, on se dirige dans le procédure *use()* de notre classe *Player*, où l'on va vérifier que l'item à utiliser est notre téléporteur appelé « Rune ». S'il s'agit bien de lui, si on veut l'utiliser en étant dans la pièce où il a été chargé, on bloque son utilisation. Si on se trouve bien dans une autre pièce, on fait le changement de salle en modifiant directement l'attribut *aCurrentRoom*, et non en appelant *setCurrentRoom()* car nous avons décidé qu'après une téléportation, le retour dans la salle précédente devient impossible, donc on ne veut pas mémoriser la pièce courante avant le déplacement. On décide donc après une téléportation de vider la pile. Enfin, pour empêcher une autre utilisation de notre beamer, on le supprime après utilisation de l'inventaire du joueur.

```

else if ( pItemName.equals("Rune") ){
    Beamer vBeamer = (Beamer)(vItem);
    if ( this.getCurrentRoom().equals(vBeamer.getBeamerRoom()) ){
        this.aGui.println("Vous devez effectuer un déplacement avant de vous téléporter au r");
        return;
    }
    this.aCurrentRoom = vBeamer.getBeamerRoom();
    this.aPreviousRooms.clear();
    this.aGui.println("Vous avez été téléporter dans la pièce mémorisé lors du chargement."
        + "\nLa Rune de téléportation perd de son éclat, elle vient de se briser !!!");
    this.removeInventoryItem(pItemName);
    this.aGui.println(this.inventory());
}

```

Pour finir, on se place dans la classe *GameEngine* où l'on crée une procédure privée *chargeBeamer()* qui prend en paramètre une commande et qui vérifie qu'elle possède un second mot avant d'appeler le *chargeBeamer()* de la classe *Player*. On ajoute ensuite le nouveau mot de commande dans la classe *CommandWord* ainsi qu'un test de son utilisation dans *interpretCommad()*.

#### Exercice 7.45 :

Dans cet exercice, on veut ajouter la possibilité d'avoir des portes dans certaines directions d'une salle qui sont verrouillées. Pour commencer, on crée une classe `Door` qui possède deux attribut : un booléen *aCanBeCrossed* qui indique si la pièce peut être traversé ou non, et un item *aKeyDoor* qui représente la clé de la porte, nécessaire pour l'ouvrir. On ajoute donc un constructeur naturel qui initialise ses deux attributs, ainsi que des accesseurs aux attributs et un modificateur pour modifier la valeur booléenne contenu dans *aCanBeCrossed*. On se dirige ensuite dans la classe `Room` ou on crée une `HashMap` qui associe une direction à une porte : elle possède les même direction que la `HashMap` des sorties, donc se superpose avec elle. On ajoute par la suite un setter *setDoor()* qui prend une direction et une porte en paramètre pour pouvoir ajouter des portes au direction de la pièce, ainsi qu'un getter *getDoor()* qui prend en paramètre une direction, et qui retourne la porte qui se trouve dans cette direction

```
public void setDoor( final String pDirection, final Door pDoor )
{
    this.aDoors.put(pDirection, pDoor);
} // setDoor(..)
```

```
public Door getDoor( final String pDirection )
{
    return this.aDoors.get(pDirection);
} // getDoor(..)
```

Pour continuer, on va dans la classe `GameEngine` pour ajouter une vérification dans *goRoom()* avant un déplacement : si la pièce actuelle possède une prochaine salle, et qu'il y a une porte dans la direction de cette prochaine salle, on regarde si la porte peut être franchie. Dans le cas où elle ne peut pas l'être, on prévient le joueur qu'il doit trouver la clé nécessaire. Si au contraire elle peut être franchie, on effectue le déplacement du joueur, en veillant à refermer la porte après son passage.

On continue notre implémentation en se dirigeant dans la classe `CommandWord` où on ajoute un mot de commande « ouvrir », puis dans la classe `Player`, où l'on crée une procédure *openDoor()* qui prend en paramètre une string correspondant à une direction. Cette méthode vérifie que s'il y a une porte dans la direction demandée, si le joueur possède la clé de la porte, on autorise le fait de pouvoir traverser la porte en modifiant l'attribut de celle-ci.



```

public void openDoor( final String pDirection )
{
    Door vDoor = this.getCurrentRoom().getDoor(pDirection); // la porte dans la direc

    if ( vDoor != null ){ // si il y a une porte dans la direction demandé
        Item vKeyDoor = vDoor.getKeyDoor(); // la clé de la porte
        Item vKeyInventory = this.getInventoryItem(vKeyDoor.getItemName()); // la clé

        if ( vKeyInventory != null ){ // le joueur possède la clé de la porte dans sa
            vDoor.setCanBeCrossed(true);
            this.aGui.println("La porte dans la direction " + "\"" + pDirection + "\"");
        } else {
            this.aGui.println("Vous devez récupérer la clé " + "(" + vKeyDoor.getItemName() + ")");
        }
    } else {
        this.aGui.println("Il n'y a pas de porte verrouillée dans cette direction.");
    }
} // openDoor(.)

```

Enfin, on retourne dans la classe `GameEngine` où l'on ajoute une procédure privée de même nom que la précédente, mais qui prend cette fois en paramètre une commande, et qui vérifie qu'elle possède un second mot pour pouvoir appeler le `openDoor()` situé dans `Player`.

Puis on fini l'exercice en rajouter dans notre série de `if` de `interpretCommand` le test si le mot de commande est « ouvrir ».

#### Exercice 7.45.1 :

Mise à jour des fichiers de test.

#### Exercice 7.45.2 :

Génération de la javadoc.

#### Exercice 7.46 :

On veut dans cet exercice ajouter une `Transporter Room`, une pièce qui téléporte le joueur aléatoirement lorsqu'il rentre une direction pour sortir de la pièce. Pour cela, on décide de modifier l'exercice 18.5 pour ajouter les pièces dans une `ArrayList` dans la classe `GameEngine` en important l'important, l'initialisant dans le constructeur et en ajoutant dans notre cas les pièces du premier étage seulement.

```

this.aRoomList.add(0, vHall);
this.aRoomList.add(1, vIntermediate);
this.aRoomList.add(2, vEquipments);
this.aRoomList.add(3, vTransition);

```

On fait cela car notre `Transporter Room` sera la salle dimensionnelle située au premier étage, et on veut qu'elle transporte le joueur seulement dans les autres pièces de cet étage. On crée ensuite une classe `RoomRandomizer` possédant deux champs : un pour l'`ArrayList` des pièces

dans lequel le joueur sera transporté *aRoomList*, et un de type *Random* *aRandom* pour générer des nombres aléatoires. On ajoute donc un constructeur qui prend en paramètre une *ArrayList* et qui initialise les attributs. Ensuite, on ajoute une méthode *findRandomRoom()* dans laquelle on va mettre le prochain nombre aléatoire compris entre 0 et la taille de la liste dans un entier, puis on prend la pièce associée à cet entier, qui sera donc retournée.

```

2 import java.util.Random;
3 import java.util.ArrayList;
4 public class RoomRandomizer
5 {
6     private ArrayList<Room> aRoomList;
7     private Random aRandom;
8
9     /**
10      *
11      */
12     public RoomRandomizer( final ArrayList pRoomList )
13     {
14         this.aRoomList = pRoomList;
15         this.aRandom = new Random();
16     }
17
18     /**
19      *
20      */
21     public Room findRandomRoom()
22     {
23         int vRandomNumber = this.aRandom.nextInt(this.aRoomList.size());
24         return this.aRoomList.get(vRandomNumber);
25     }

```

On crée pour continuer une classe *TransporterRoom* qui hérite de la classe *Room*, et qui possède un attribut de type *RoomRandomizer* *aRandomRoom* pour pouvoir avoir une salle aléatoire. On crée donc un constructeur prenant en paramètre les attributs de la super classe ainsi qu'un *ArrayList* pour pouvoir initialiser l'attribut *aRandomRoom* et ceux de la classe mère par un appel de *super()*. On redéfinit comme dit dans le chapitre 9.11 le *getExit()* qui sera utilisé dans la méthode *goRoom()* dans le cas où la pièce est une *TransporterRoom*, puis la méthode *findRandomRoom()* qui retourne la room choisie aléatoirement en appelant la méthode de la classe *RoomRandomizer*.

Enfin, on se place dans la classe *GameEngine* où on modifie notre salle *Dimensionnelle* pour qu'elle devienne une *TransportRoom* :

```

TransporterRoom vDimension = new TransporterRoom("dans la salle dimensionnelle",
"Dimension.png", this.aRoomList);

```

On décide que lorsque le joueur se déplace en rentrant une commande pour aller dans une autre pièce comme « aller NordOuest », lorsqu'il se trouve dans la pièce aléatoire, la commande retour lui est interdite lorsqu'il la rentre. Pour cela, on ajoute dans la méthode *back()* de la classe *Player* une restriction. On crée d'abord une méthode booléenne

wasTransporterRoom qui retourne vrai si la précédente salle était une TransporterRoom et faux dans le cas contraire. Dans la méthode *back()*, si la pièce est une sortie et même si elle ne l'ai pas, si la pièce précédente était une TransporterRoom, on vide la pile et on prévient le joueur :

```
if ( this.aCurrentRoom.isExit(this.aPreviousRooms.peek()) ){
    if ( this.wasTransporterRoom() ){
        this.aPreviousRooms.clear();
        this.aGui.println("Vous venez d'une pièce transporteuse, vous n'avez pas mémorisez de trajet par conséquent !");
        this.aGui.setEnableBackB(false);
        return;
    }
    this.aCurrentRoom = this.aPreviousRooms.pop();
} else {
    if ( this.wasTransporterRoom() ){
        this.aGui.println("Vous venez d'une pièce transporteuse, vous n'avez pas mémorisez de trajet par conséquent !");
    } else this.aGui.println("Vous avez franchi une porte à sens unique. Il vous est impossible de retourner en arrière.");
    this.aPreviousRooms.clear();
    this.aGui.setEnableBackB(false);
}
}
```

De plus, si le joueur rentre dans la pièce transportreuse, et rentre la commande « retour » il va être déplacé aléatoirement au lieu de revenir sur ses pas pour garder l'effet d'un pièce qui déplace aléatoirement lorsqu'on décide d'en sortir, et pour piéger le joueur. Bien sûr un message prévient le joueur. Pour cela, on ajoute une autre restriction dans le *back()* de Player :

```
return;
}
if ( this.getCurrentRoom() instanceof TransporterRoom ){
    TransporterRoom vRoom = (TransporterRoom)(this.getCurrentRoom());
    this.setCurrentRoom(vRoom.getExit("SudOuest"));
    this.aGui.println("Vous étiez dans une pièce transporteuse, sortir de cette pièce vous déplace aléatoirement");
    return;
}
this.aCurrentRoom = this.aPreviousRooms.pop();
```

On réalise cela car le « back » dans notre jeu est le fait pour le joueur de mémoriser son trajet. Or entrant dans la Transporter Room, le joueur est déplacé aléatoirement dans une autre pièce de l'étage, il n'a donc pas mémoriser le trajet nécessaire pour aller dans la nouvelle pièce. D'où la restriction.

A savoir expliquer :

- ✚ Random : la classe Random permet d'instancier un générateur de séquences de nombres aléatoire. Lorsqu'il prend en paramètre une seed, il génère toujours la même séquence
- ✚ nextInt() : est une méthode de la classe Random qui permet d'avoir le prochain nombre pseudo aléatoire générer par le générateur de nombre aléatoire
- ✚ Seed : le seed permet lorsque passé en paramètre d'un constructeur de Random d'avoir toujours le même séquence de nombres pseudo aléatoires, ce qui peut être très utile pour corriger des bugs

### Exercice 7.46.1 :

Dans cet exercice, on cherche à ajouter une commande « alea » qui lorsqu'elle prend une string en paramètre, la prochaine salle dans laquelle la transporter room déplacera le joueur sera la salle choisie par le second mot de la commande. Et lorsqu'elle n'a pas de second mot, elle rend le caractère aléatoire de la transporter room. Pour ce faire, on ajoute dans la classe `TransporterRoom` un attribut entier *aAleaInt* qui permettra de récupérer une pièce de notre liste, ainsi qu'un attribut de type `ArrayList` *aRoomList* pour pouvoir accéder à notre liste partout dans la classe. On initialise la liste avec celle passée en paramètre du constructeur, puis *aAleaInt*. On ajoute pour continuer un setter pour pouvoir modifier la valeur de l'attribut lorsque la commande alea String sera utilisée :

```
public void setAleaInt( final int pAleaInt )
{
    this.aAleaInt = pAleaInt;
}
```

Ensuite, on ajoute des conditions dans le *getExit()* de cette classe pour pouvoir désactiver le caractère aléatoire de la transporter room lorsque l'on se trouve en mode test, donc si l'attribut *aAleaInt* possède une autre valeur dans la limite de la taille de la liste.

```
public Room getExit( final String pDirection )
{
    if ( this.aAleaInt == 6 ) return this.findRandomRoom();
    else return this.aRoomList.get(this.aAleaInt);
} // getExit(.)
```

La suite se passe dans la classe `GameEngine`, où on ajoute une procédure privée *alea()* qui prend en paramètre une commande, et qui teste si la commande a un second mot ou pas. Si elle en a un, elle modifie l'attribut de la transporter room par le nombre passé en second mot sous forme de String pour pouvoir choisir la pièce du prochain déplacement associé à ce nombre dans la liste des pièces lorsqu'on se trouve en mode test. Pour faire cela, puisque notre attribut est un entier et la commande prend un second mot (String), on doit extraire l'entier de la string avec *Integer.valueOf()* :

```

private void alea( final Command pCommand )
{
    if ( pCommand.hasSecondWord() ){
        if ( this.aTestMode ){
            Room vRoom = this.aRoomList.get(4);
            TransporterRoom vNextRoom = (TransporterRoom)(vRoom);
            int vInt = Integer.valueOf(pCommand.getSecondWord());
            vNextRoom.setAleaInt(vInt);
        } else this.aGui.println("Cette commande ne peut être utilisée qu'en mode test.");
    } else {
        if ( this.aTestMode ){
            Room vRoom = this.aRoomList.get(4);
            TransporterRoom vNextRoom = (TransporterRoom)(vRoom);
            vNextRoom.setAleaInt(6);
        } else this.aGui.println("Cette commande ne peut être utilisée qu'en mode test.");
    }
}
} // alea(.)

```

Enfin, si la commande ne prend pas de second mot, et si on se trouve en mode test, on remet l'attribut à la valeur initiale déclarée dans la classe `TransporterRoom`, soit 6.

Pour finir, on ajoute le mot de commande dans la liste des mots dans `CommandWords`, puis on ajoute le test dans `interpretCommand()` de `GameEngine`.

#### Exercice 7.46.3 :

On réalise les commentaires javadocs de nos classes.

#### Exercice 7.46.4 :

On génère de nouveaux les javadocs en corrigeant les erreurs et warnings.

#### Exercice 7.47 :

On veut dans cet exercice implémenter la conception du `zuul-even-better` dans notre jeu de sorte à avoir une classe par commande. Pour ce faire, on commence par rendre notre classe `Command` abstraite, et à lui supprimer tout ce qui concernait l'attribut `aCommandWord` puisque désormais il y aura des sous classes de la classe `Command`. Puis on ajoute une fonction booléenne abstraite `execute()` qui prend en paramètre non pas un `Player` comme suggérer dans le `zuul-even-better`, mais plus tôt une `GameEngine`. On décide de faire cela pour pouvoir accéder à tout ce dont nous avons besoin que l'on peut facilement récupérer avec un objet de type `GameEngine`, plus tôt qu'un `player`. On redéfinira ce booléen dans chacune des sous classes.

On se place ensuite dans la classe `GameEngine` où l'on ajoute un modificateur ainsi qu'un accesseur à l'attribut correspondant au mode test. On ajoute de plus des accesseurs au `player` pour pouvoir appeler nos méthodes de commandes qui s'y trouvent dans les classes filles, ainsi qu'un accesseur au `Gui` pour pouvoir afficher des messages lorsque nécessaire. On prévoit par ailleurs un accesseur au `Parser` pour la commande « aide » qui en nécessite pour récupérer les mots de commandes, un pour récupérer de la liste des pièces une pièce en particulier, et enfin un pour notre `Talisman` (voir [IV](#)).

On crée ensuite nos différentes classe en suivant le modèle proposer dans le projet fourni par l’exercice en migrant nos méthodes du GameEngine vers la classe associée.

Pour continuer, on modifie la classe CommandWords qui prend les mots de commandes dans une HashMap maintenant au lieu du tableau. On supprime le booléen *isUnknow()* , puis on ajoute un accesseur qui retourne à la place le mot de commande associé au nom passé en paramètre. On modifie aussi le *getCommandList()* pour qu’elle parcourt la HashMap et ajoute chaque mot de commande dans la chaine de caractère avec un for each.

Pour finir, on retourne dans la classe GameEngine dans la méthode *interpetCommand()* pour vérifier que le mot de commande existe bien avant d’exécuter la commande.



#### A Savoir Expliquer :

Le polymorphisme : lorsqu’une méthode est définie dans une classe A, et qu’une autre classe B ainsi qu’une classe C hérite de la première, en redéfinissant une même méthode qui se trouve dans la classe mère dans B et C, mais avec des instructions différentes, lorsque l’on va appeler la méthode sur un objet, le polymorphisme permet d’exécuter les bonnes instructions en fonction de quel objet il s’agit.

#### Exercice 7.47.1 :

On ajoute des paquetages. Puis on importe les classes d’autre paquetages nécessaire au fonctionnement de chaque classe.

#### A Savoir Expliquer :

-  Paquetage anonyme : les classes qui se trouvent dans le même repertoire sont appelé classe du paquetages anonyme : on a pas besoin de les importer pour pouvoir les utiliser
-  Paquetage par défaut : représente l’ensemble des classes que l’on peut accéder par défaut avec l’instruction d’import et ne faisant pas parties du repertoire courant

#### Exercice 7.47.2 :

On ajoute la javadoc des nouvelles classes, puis on les génères.

#### Exercice 7.48 :

On veut dans cet exercice ajouter des personnages non joueur à notre jeu. Qui pourront parler et même aider le joueur en échange d’un item.

Pour cela, on commence par créer un paquetage character en prévision du prochain exercice, dans lequel on crée une classe Character. On ajoute dans cette classe 3 champs : un pour le nom du PNJ, un pour son inventaire, et un dernier pour la phrase qu'il va dire.

On initialise donc les attributs dans un constructeur ayant pour paramètre le nom et la phrase du PNJ. On ajoute par la suite des accesseurs aux 3 attributs, un modificateur de la phrase dites par le personnage, ainsi que deux méthodes : l'une pour ajouter des items dans l'inventaire du joueur, et l'autre pour en retirer de son inventaire.

On se place ensuite dans la classe Room où on ajoute une HashMap qui associe un PNJ à son nom. Elle représente les personnages qui se trouvent dans la pièce courante. On crée pour continuer un accesseur prenant en paramètre une String correspondant au personnage de la liste à retourner après avoir initialisé cette liste dans le constructeur. On ajoute ensuite une *getCharacterString()* qui retourne le nom des personnages présents dans la pièce lorsqu'il y en a.

```
public String getCharacterString()
{
    if ( this.aCharacterList.size() > 0 ){
        String vCharacter = "Vous voyez la/les personne(s) suivante(s) :\n";
        Set<String> vKeys = this.aCharacterList.keySet();
        for( String vCharacterName : vKeys ){
            vCharacter += "-> " + vCharacterName + "\n";
        }
        return vCharacter;
    } else return "Il semble n'y avoir aucun habitant(s) de la Tour dans cette pièce.";
} // getCharacterString()
```

On ajoute de plus une *addCharacter()* qui prend en paramètre un Character, et qui permet d'ajouter des personnages dans la HashMap de la pièce, donc dans la pièce. On peut après cela mettre un appel de la nouvelle fonction dans la *getLongDescription()* de la classe. On va ensuite dans *creatRoom()* de la classe GameEngine pour ajouter des PNJ à notre jeu puis les ajouter dans des pièces.

Pour continuer, on veut que le joueur puisse parler au personnage dans la pièce. Pour cela, on ajoute donc une nouvelle classe TalkCommand qui permet de parler au personnage dont le nom est spécifié en second mot.

Enfin, on veut aussi pouvoir donner un objet au joueur. On décide donc de modifier notre Parser pour qu'il puisse lire non plus 2 mots, mais 3 maintenant. On ajoute donc un attribut *aThirdWord* dans la classe Command ainsi qu'un accesseur, modificateur et un booléen qui vérifie l'existence de ce troisième mot. Dans la classe Parser, on ajoute le même test que pour le second mot mais cette fois pour un troisième mot dans la méthode *getCommand()*. Enfin, on crée une classe GiveCommand pour permettre de donner un objet à un personnage en spécifiant le nom de l'objet en second mot et le nom du personnage en troisième mot.

#### Exercice 7.49 :

On veut dans cet exercice ajouter des personnages non joueur a notre jeu qui se déplace aléatoirement dans les pièces adjacentes à leur pièce initiale. Pour cela, on commence par ajouter une nouvelle classe `MovingCharacter` qui hérite de la précédente classe `Character`. On lui ajoute en plus comme attribut une `ArrayList`, pour la liste des pièces dans laquelle il pourra se déplacer, un `aCurrentRoom` qui représente la pièce actuelle du moving pnj, ainsi qu'un attribut de typer `RoomRandomizer` pour prendre des salles aléatoirement. On crée un constructeur qui initialise les attributs de la classe mère ainsi que ceux de la classe. On crée de plus un accesseur et un modificateur au champ `aCurrentRoom`, ainsi qu'une méthode `findRandomRoom()` qui retourne un pièce de la liste de manière aléatoire. Pour cela on avait dû ajouter dans la classe `RoomRandomizer` un accesseur `getRandomNumber()` qui retourne un nombre aléatoire dans la limite de la taille de la liste.

```
private Room findRandomRoom()
{
    int vRandomNumber = this.aRandomRoom.getRandomNumber();
    return this.aRoomList.get(vRandomNumber);
} // findRandomRoom()
```

On ajoute enfin un `moveCharacter()` qui permettra de déplacer le Moving Character en veillant en l'enlever de la liste des pnj de la pièce, puis en le rajoutant dans celle des pnj de la nouvelle pièce.

```
public void moveCharacter()
{
    this.aCurrentRoom.removeCharacter(this.getCharacterName());
    this.setCurrentRoom(this.findRandomRoom());
    this.aCurrentRoom.addCharacter(this);
}
```

On se dirige ensuite dans la classe `GameEngine` ou on crée un tableau de `MovingCharacter` qui contiendra la liste des personnages pouvant se déplacer, puis un procédure `moveCharacter()` qui déplace chaque personnages de la liste.

```
public void moveCharacter()
{
    for( int i = 0; i < this.aMovingCharacterList.length; i++ ){
        this.aMovingCharacterList[i].moveCharacter();
    }
}
```

On veut que lorsque le joueur se déplace, les pnj qui le peuvent se déplace aléatoirement aussi dans les pièces ou ils sont autorisé a bouger. Pour cela, on ajoute un appel de cette procédure dans les classe `GoCommand` ainsi que dans `BackCommand` pour que les personnages et le joueur effectuent leur déplacement en même temps.



#### Exercice 7.49.2 :

On complète le jeu.

#### Exercice 7.49.3 :

On génère les javadocs.

#### Exercice 7.53 :

On ajoute une méthode principale dans la classe Game de notre jeu.

### III) Mode d'emploi

J'ai modifié le « fonctionnement » de l'inventaire en ajoutant une sorte de restriction : l'inventaire du joueur étant stocké dans une autre dimension à l'aide de son Bracelet, il ne peut prendre des items que lorsqu'il possède cet objet se trouvant dans la salle de départ (voir **Changement** au début de du II)).

Comme on peut le voir dans le plan du jeu au 3<sup>e</sup> étage de la tour, il n'y a pas de porte (flèches) reliant les salles d'où le joueur vient, et celles où se trouve l'objet à récupérer pour terminer le jeu. Pour cela, il se trouve dans le Hall du troisième étage un téléporteur incomplet (Plastron), et des pièces dans les deux salles voisines (Fiole et Vanadinite). Le joueur doit prendre le Plastron ainsi que ces deux pièces, et l'assembler avec la commande de même nom pour que le Plastron devienne un Talisman de téléportation. Un fois cela fait, il pourra utiliser le Talisman pour se téléporter dans la salle du Gardien et ainsi accéder à la salle finale.

### IV) Ajouts

J'ai modifié le « fonctionnement » de l'inventaire en ajoutant une sorte de restriction comme dit précédemment. Pour permettre cela, j'ai donc ajouté un *setMaxWeight()* puis j'ai rajouté des conditions dans *take()* et dans *drop()*. Pour que le joueur ne puisse prendre des objets que s'il a déjà pris le 'Bracelet' qui modifiera son poids max, et dans *drop()* pour l'empêcher de déposer son inventaire.

Pour continuer, j'ai ajouté une nouvelle commande inspecter pour regarder les détails d'un item de l'inventaire. Pour cela, j'ai ajouté dans la classe Player une procédure *inspectItem()* qui prend en paramètre le nom de l'item à regarder, vérifie que le joueur possède l'item, et dans ce cas affiche la longue description de ce dernier. Ensuite, dans la classe GameEngine, j'ajoute une procédure *inspectItem()* qui prend en paramètre une commande, et qui vérifie que cette dernière possède un second mot pour appeler *inspectItem()* de Player.

Concernant ma commande inventaire, j'ai ajouté un panneau qui apparaît puis disparaît dès lors que la commande inventaire ou inspecter est rentrée afin de permettre

l'écriture dans ce panneau d'inventaire. J'ai de plus importer la classe Font du jdk pour pouvoir modifier la police d'écriture des panneaux d'écritures.

J'ai ajouté afin de pouvoir accéder aux salles finales ( du Gardien et de l'Illusion) la possibilité d'assembler un téléporteur qui est prédéfini sur une pièce lors de sa création. Pour cela, j'ai ajouter une méthode *build()* dans Player ainsi qu'une classe depuis l'exercic 47 pour construire le téléporteur nommé talisman (voir !)) lorsque le joueur possède les pièces nécessaire à sa fabrication.

J'ai ajouté une carte du jeu qui change en fonction de l'étage, j'ai d'abord créer dans UserInterface un méthode qui change l'image en fonction du booléen passé en paramètre pour afficher soit l'image de la pièce soit la carte. J'ai créer un classe MapCommand qui permet par un appel de la méthode de UserInterface d'ouvrir la carte, et donc de change l'image de la pièce, ainsi qu'une classe CloseCommand qui permet de fermer la carte. J'ai de plus ajouter un bouton map qui lorsqu'en possession de celle-ci, le joueur peut afficher la carter par un appuie sur le bouton. Enfin, le bouton est bloqué lorsque le joueur ne possède pas la carte : il doit donner le bon item au bon personnages pour gagner la carte.

Suite à l'exercice 48, j'ai modifier mon GoCommand pour qu'il puisse lire 3 mots, et ainsi rentrer une commande du style « aller Sud Est » au lieu de « aller SudEst » pour se déplacer au Sud Est.

## V) Déclaration anti-plagiat

Je déclare ne pas avoir fait de plagiat dans mon code ou dans mes images dont les sources sont indiquées.

De plus, je tiens à remercier Benjamin LAMBERT, un super camarade de classe qui m'a permis de faire mes images en me faisant un tutoriel sur Unity3D. Ce même camarade m'a aussi aidé pour générer la javadoc avec un fichier de commande .BAT.

## VI) Sources

Sources des Tilesets qui ont permis de créer mes images :

<https://www.pixelart.fr/2019/02/21/ladresse-donjon/>

<https://www.pinterest.fr/pin/53339576821684646/>

<https://www.pinterest.fr/pin/587016132665073191/>

<https://www.pinterest.fr/pin/15973773666052524/>

<https://i.pinimg.com/564x/15/6b/72/156b72c873ac0e4436a6d1c60a662e73.jpg>

<https://www.pinterest.fr/pin/21040323246857966/>

<https://www.rpgmakervx-fr.com/t21735-bibliotheque-des-ressources-mv-tilesets>

[https://www.google.com/imgres?imgurl=http%3A%2F%2Fdata.over-blog.com%2F4%2F08%2F89%2F09%2Fresource%2Ftileset-et-iconset%2FStatues---Autels.png&imgrefurl=http%3A%2F%2Fpg-maker-vx-resource.over-blog.com%2Farticle-tileset-et-icone-set-vx-73723838.html&tbnid=sU\\_ICm-Pp72wdM&vet=12ahUKEwjpXrmb\\_5zwAhXxAmMBHdEsB28QMygAegQIARA8..i&docid=Q24nSa65ladeOM&w=512&h=512&q=tileset%20autel&ved=2ahUKEwjpXrmb\\_5zwAhXxAmMBHdEsB28QMygAegQIARA8](https://www.google.com/imgres?imgurl=http%3A%2F%2Fdata.over-blog.com%2F4%2F08%2F89%2F09%2Fresource%2Ftileset-et-iconset%2FStatues---Autels.png&imgrefurl=http%3A%2F%2Fpg-maker-vx-resource.over-blog.com%2Farticle-tileset-et-icone-set-vx-73723838.html&tbnid=sU_ICm-Pp72wdM&vet=12ahUKEwjpXrmb_5zwAhXxAmMBHdEsB28QMygAegQIARA8..i&docid=Q24nSa65ladeOM&w=512&h=512&q=tileset%20autel&ved=2ahUKEwjpXrmb_5zwAhXxAmMBHdEsB28QMygAegQIARA8)

[https://www.google.com/imgres?imgurl=https%3A%2F%2Fi.servimg.com%2Fu%2Ff49%2F19%2F50%2F44%2F78%2F9ggdnm10.png&imgrefurl=https%3A%2F%2Fwww.rpgmakervx-fr.com%2Ft21735-bibliotheque-des-ressources-mv-tilesets&tbnid=fODLyVo6l1Z7RM&vet=12ahUKEwibw7vcg5\\_wAhUNwYUKHTeLALeQMyglegQIARbk..i&docid=tkHAHncf7onuyM&w=768&h=768&q=tilesets%20biblioth%C3%A8que&ved=2ahUKEwibw7vcg5\\_wAhUNwYUKHTeLALeQMyglegQIARbk](https://www.google.com/imgres?imgurl=https%3A%2F%2Fi.servimg.com%2Fu%2Ff49%2F19%2F50%2F44%2F78%2F9ggdnm10.png&imgrefurl=https%3A%2F%2Fwww.rpgmakervx-fr.com%2Ft21735-bibliotheque-des-ressources-mv-tilesets&tbnid=fODLyVo6l1Z7RM&vet=12ahUKEwibw7vcg5_wAhUNwYUKHTeLALeQMyglegQIARbk..i&docid=tkHAHncf7onuyM&w=768&h=768&q=tilesets%20biblioth%C3%A8que&ved=2ahUKEwibw7vcg5_wAhUNwYUKHTeLALeQMyglegQIARbk)

[https://www.google.com/imgres?imgurl=https%3A%2F%2Fi.servimg.com%2Fu%2Ff62%2F19%2F50%2F44%2F78%2Finside10.jpg&imgrefurl=https%3A%2F%2Fwww.rpgmakervx-fr.com%2Ft21735-bibliotheque-des-ressources-mv-tilesets&tbnid=AWtj2QqWgSnoDM&vet=12ahUKEwibw7vcg5\\_wAhUNwYUKHTeLALeQMygAegQIARBU..i&docid=tkHAHncf7onuyM&w=768&h=768&q=tilesets%20biblioth%C3%A8que&ved=2ahUKEwibw7vcg5\\_wAhUNwYUKHTeLALeQMygAegQIARBU](https://www.google.com/imgres?imgurl=https%3A%2F%2Fi.servimg.com%2Fu%2Ff62%2F19%2F50%2F44%2F78%2Finside10.jpg&imgrefurl=https%3A%2F%2Fwww.rpgmakervx-fr.com%2Ft21735-bibliotheque-des-ressources-mv-tilesets&tbnid=AWtj2QqWgSnoDM&vet=12ahUKEwibw7vcg5_wAhUNwYUKHTeLALeQMygAegQIARBU..i&docid=tkHAHncf7onuyM&w=768&h=768&q=tilesets%20biblioth%C3%A8que&ved=2ahUKEwibw7vcg5_wAhUNwYUKHTeLALeQMygAegQIARBU)

<https://www.pinterest.fr/pin/390124386479670117/>

<https://www.pinterest.fr/pin/458311699586808478/>

[https://www.pinterest.fr/pin/3096293484398682/?rcpt=1117174388723032610&utm\\_campaign=homefeednewpins&et=cdb37ac9fdec48698eb073e4c9422666&utm\\_source=31&utm\\_medium=2025&utm\\_content=3096293484398682&news\\_hub\\_id=5156861351060097354](https://www.pinterest.fr/pin/3096293484398682/?rcpt=1117174388723032610&utm_campaign=homefeednewpins&et=cdb37ac9fdec48698eb073e4c9422666&utm_source=31&utm_medium=2025&utm_content=3096293484398682&news_hub_id=5156861351060097354)

<https://www.pinterest.fr/pin/331577591293444247/>

<https://www.pinterest.fr/pin/331577591292394970/>