Auteur : Castillo Thaïs

Groupe 11

Titre : Playmobil présente : The Bedroom Exploration !

https://app.leonardo.ai/

Sommaire :

IV) Mode d'emploi

I) Projet Zuul

A. Auteur

CASTILLO Thaïs

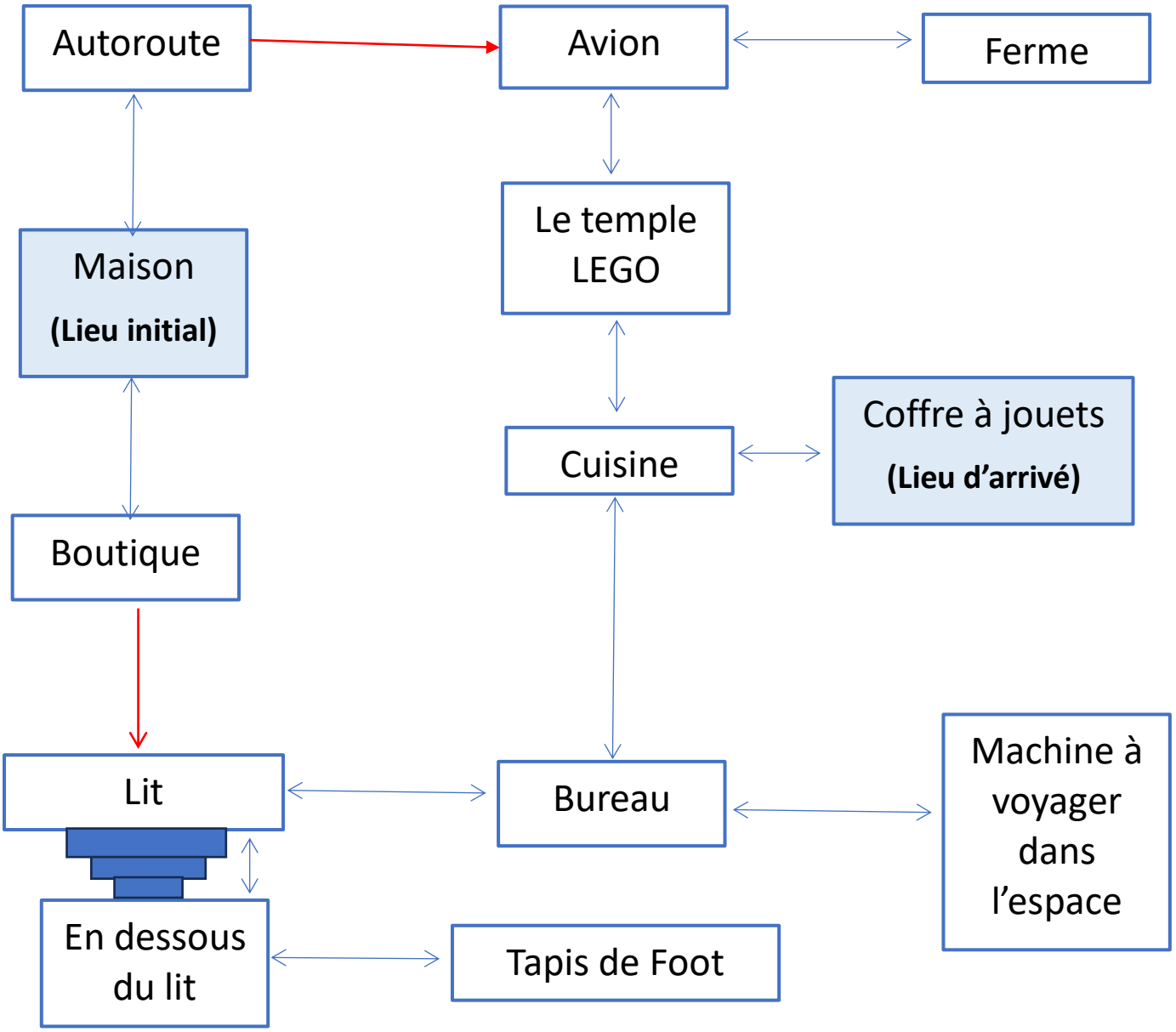
B. Thème

Dans une chambre d'enfant : aider un Playmobil à retrouver son chemin vers sa boîte à jouets.

C. Résumé du scénario

Nous sommes dans une chambre d'enfant et notre personnage est un Playmobil aventurier dont le principal rôle est d'explorer la chambre ou encore partir à la recherche de ses compagnons. Alors qu'il se trouve dans la Maison des Playmobil, celui-ci va chercher à retrouver son chemin vers la boîte à jouets afin de retrouver ses compagnons. Cependant, le trajet se révèle plus périlleux qu'il n'en a l'air : en effet, il va devoir éviter de se faire éliminer par les différents obstacles qu'il va rencontrer tout le long de son trajet en se rendant dans plusieurs lieux.

D. Plan



**Entre le Lit et En dessous du lit nous avons un escalier qui nous permet de descendre.*

E. Scénario détaillé

Andy, petit garçon, possède la plus belle et la plus grande des chambres jamais rêvé pour un enfant. Surtout si celle-ci est rempli à ras bord de jouets de toutes sortes : LEGO, peluches, petites voitures, mais surtout des Playmobils ! Et parmi eux nous retrouvons le très célèbre Playmobil aventurier, celui qui serait prêt à tout pour sauver ses compagnons et qui aime explorer la chambre d'Andy.

Cependant, un jour, alors que Playmobil aventurier explorait la Maison des Playmobil, une urgence l'oblige à rentrer le plus rapidement possible dans la boîte à jouets. Mais attention aux nombreux obstacles qui l'empêcheront d'accéder à son objectif. En effet, il s'avère que la chambre d'Andy soit le plus dangereuse qui puisse exister pour un jouet qui doit entièrement la traverser par lui-même. Ainsi, il va par exemple devoir faire face à la machine à voyager dans l'espace, les Traps Doors et surtout trouver la clé qui lui permettra d'ouvrir le coffre à jouet, lieux d'arrivé.

Enfin, si le Playmobil n'arrive pas à bon port, il sera perdu pour toujours et Andy sera triste de ne plus retrouver son jouet !

F. Détails des lieux, items, personnages

- **Maison :**

Description : Lieu initial, il s'agit de la maison des Playmobils.

Personnages, items : Une épée.

- **Boutique :**

Description : Boutique Playmobil qui se trouve juste à côté de la Maison, elle vend pleins de choses mais surtout des vies.

Personnages, items : Un cœur (qui sera le magic Cookies) et une autre épée.

- **Lit :**

Description : Il y a des peluches (au moins deux), qui vont lui raconter beaucoup de choses pas forcément utiles.

Personnages, items : Une échelle.

- **Bureau :**

Description : Il y a des feuilles, des cahiers partout et il faut résoudre un pendu (le jeu) pour passer à une prochaine pièce, si on perd une fois, on perd une vie.

Personnages, items : Rien.

- **Tapis de Foot :**

Description : Tapis où il ne faut pas se faire manger par le chat.

Personnages, items : Rien.

- **Ferme :**

Description : Il s'agit d'une ferme Playmobil où il y a toute sorte d'animaux, mais il faut aider un fermier à retrouver sa fourche.

Personnages, items : Fourche, Pelle, Seau, Râteau.

- **Autoroute :**

Description : Il y a pleins de voitures et le Playmobil ne doit pas se faire écraser.

Personnages, items : Rien.

- **Avion :**

Description : On est dans une fusée.

Personnages, items : Rien.

- **Boite à jouets :**

Description : Lieu d'arrivé dans la situation gagnante.

Personnages, items : Rien.

- **Cuisine :**

Description : Dernière pièce avant d'accéder à la boite à jouets, il faut retrouver une clé dans un labyrinthe.

Personnages, items : Le clé.

G. Situations gagnantes et perdantes

Situation gagnante :

Arrive sain et sauf dans la boite à jouets.

Situation perdante :

Perd toutes ses vies et n'arrive pas dans la boîte à jouets.

H. Eventuellement énigmes, mini-jeux, combats, etc…

Il n'y a pas de mini jeu.

I. Commentaires

Par la suite j'ai décidé d'ajouter plus de boutons (à voir dans la version finale).

II) Déclaration obligatoire anti-plagiat

Afin de pouvoir ajouter plus de boutons par la suite (donc après avoir ajouté le bouton look) j'ai copié un fragment de code de la classe UserInterface d'une ancienne élève de E1 de l'année dernière (Lana Cheam qui faisait cette fonction.

Aussi, je me suis aidé pour l'exercice 7.46 de ChatGBT pour convertir une chaine de caractère sous forme de chiffre en int (en utilisant la fonction *parseInt()).* Car je ne voyais pas comment faire autrement.

III) Réponses aux exercices

1) Exercices 7.5

J'ai dû simplifier un fragment de code dans la classe Game afin d'éviter la duplication dans deux méthodes différentes (*printWelcome*() et *goRoom*()).
C'est pourquoi j'ai dû créer la procédure *printLocationInfo*() que j'appel donc dans les deux méthodes concernés.

```
/**
 * Procédure qui affiche les sorties
 */
private void printLocationInfo(){
    System.out.println("Tu es "+ this.aCurrentRoom.getDescription());
    System.out.println("Exits:");
       if(this.aCurrentRoom.aNorthExit!=null){
               System.out.println(" north ");
       }
       if(this.aCurrentRoom.aSouthExit!=null){
               System.out.println(" south ");
           }
       if(this.aCurrentRoom.aEastExit!=null){
               System.out.println(" east ");
       }
       if(this.aCurrentRoom.aWestExit!=null){
               System.out.println(" west ");
       }

}
```

2) Exercice 7.6

La façon dont la version *setExits(…)* teste si chaque paramètre est null n'est pas indispensable car un paramètre de type Room peut très bien valoir null (c'est-à-dire qu'il n'existe pas d'objet).

L'accesseur ajouté dans la classe Room nous permet en effet par la suite de simplifier et d'encapsuler une grande partie de code dans la classe Game au moment où nous cherchons à déterminer quel sera le prochain lieu courant.

```
/**
 * Fonction qui retourne la pièce dans la direction demandée
 * par l'utilisateur
 * @param pD String correspondant à la direction
 * @return Retourne la room qui existe dans cette direction
 */
public Room getExit(final String pD)
{
    if(pD.equals("north")){
        return aNorthExit;
    }
    if(pD.equals("east")){
        return aEastExit;
    }
    if(pD.equals("south")){
        return aSouthExit;
    }
    if(pD.equals("west")){
        return aWestExit;
    }
    return null;
}
```

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

Voici comment je décide de traiter du cas « Unknown Direction », même si cela ne reste pas très précis pour l'utilisateur :

```
if (vNextRoom.equals(null)){
    System.out.println("There is no door or unknowm direction !");
     return;
}
```

3) Exercice 7.7

Encore une fois nous rendons la méthode *printLocationInfo()* plus courte en encapsulant une grande partie de code que nous avons écrit dans la fonction *getExitsString()* ce qui permet à la classe Game de seulement afficher les sorties sans savoir comment elles sont construites, puisque c'est la classe Room qui possède cette information.

```
/**
 * Fonction qui renvoie les sorties d'une pièce
 * @return String correspondant aux directions possibles
 * où il existe une pièce
 */
public String getExitsString(){
    System.out.println("Exits:");
    String vSorties="";
    if(aNorthExit!=null){
                vSorties = vSorties+" north ";
        }
    if(aSouthExit!=null){
                vSorties = vSorties+" south ";
            }
    if(aEastExit!=null){
                vSorties = vSorties+" east ";
        }
    if(aWestExit!=null){
                vSorties = vSorties+" west ";
        }
    return vSorties;

}
```

```java
/**
 * Procédure qui affiche les sorties
 */
private void printLocationInfo(){
    System.out.println("Tu es "+ this.aCurrentRoom.getDescription());
    System.out.println(this.aCurrentRoom.getExitsString());

}
```

4) Exercice 7.8

```java
/**
 * Procédure qui définit les sorties de cette pièce. Chaque direction
 * soit conduit à une autre pièce, soit est null.
 * @param pN Room pour initaliser aNorthExit
 * @param pE Room pour initialiser aEastExit
 * @param pS Room pour initialiser aSouthExit
 * @param pW Room pour initialiser aWestExit
 */
public void setExits(final Room pN, final Room pE, final Room pS, final Room pW)
{
    if(pN!=null){
        exits.put("north", pN);
    }
    if(pE!=null){
        exits.put("east", pE);
    }
    if(pS!=null){
        exits.put("south", pS);
    }
    if(pW!=null){
        exits.put("west", pW);
    }
}
```

```java
 //Attributs
private String aDescription;
private HashMap<String,Room> exits;
```

On modifie une première fois la procédure *setExits(…)* qui va définir toutes les sorties possibles d'une pièce.

On change également les attributs au début de la classe.

Puis on remplace la procédure *setExit**s**(..)* par la procédure *setExit(..)* qui va prendre seulement 2 paramètres et qui va en fait définir les sorties d'une pièce une par une et non d'un gros groupe comme le faisait *setExits(…).*

```java
/**
 * Procédure qui définit les sorties de cette pièce.
 * Les sorties de chaque pièce peuvent être définit une par une.
 * @param pD String qui permet d'avoir la direction
 * @param pS Room qui permet d'avoir la pièce où se situe la sortie
 */
public void setExit(final String pD ,final Room pS)
{
    exits.put(pD, pS);
}
```

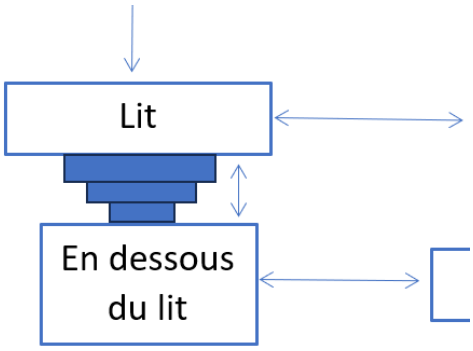
On modifie également la méthode *getExit(…)*

```java
/**
 * Fonction qui retourne la pièce atteinte si nous nous déplaçons
 * dans la direction pD
 * @param pD String correspondant à la direction
 * @return Retourne la room qui existe dans cette direction (peut valoir null)
 */
public Room getExit(final String pD)
{
    return aExits.get(pD);
}
```

Et on modifie également la méthode *createRoom()* dans la classe Game en appelant non pas *setExits(…)* mais *setExit(..).* Nous allons donc attribuer pour chaque pièce une sortie vers une autre pièce.

```java
vMaison.setExit("north", vAutoroute);
vMaison.setExit("south",vBoutique);
vAutoroute.setExit("east", vAvion);
vAutoroute.setExit("south",vMaison);
vBoutique.setExit("north", vMaison);
vBoutique.setExit("south", vLit);
vLit.setExit("north", vBoutique);
vLit.setExit("east", vBureau);
vLit.setExit("south", vSousLeLit);
vAvion.setExit("east", vFerme);
vAvion.setExit("south", vTempleLego);
vAvion.setExit("west", vAutoroute);
vFerme.setExit("west", vAvion);
vCoffre.setExit("west", vCuisine);
vCuisine.setExit("north", vTempleLego);
vCuisine.setExit("east", vCoffre);
vCuisine.setExit("south", vBureau);
vTapisDeFoot.setExit("west", vSousLeLit);
vBureau.setExit("north", vCuisine);
vBureau.setExit("west", vLit);
vTempleLego.setExit("north", vAvion);
vTempleLego.setExit("south", vCuisine);
vSousLeLit.setExit("north", vLit);
vSousLeLit.setExit("east", vTapisDeFoot);
```

5) Exercice 7.8.1



6) Exercice 7.9

La méthode *keySet()* de la classe HashMap permet de mettre toutes les clés, qui correspondent donc aux sorties/directions, de aExits dans une certaine variable (ici vKeys) qui va ainsi toutes les posséder.

```
/**
 * Fonction qui renvoie les sorties d'une pièce
 * @return String correspondant aux directions possibles
 * où il existe une pièce
 */
public String getExitsString(){
    System.out.println("Exits:");
    String vSorties="";
    Set<String> vKeys=aExits.keySet();
    for(String vExits: vKeys){
        vSorties+=" "+vExits;
    }
    return vSorties;

}
```

7) Exercice 7.10

D'abord on affiche le message « Exits :», puis on déclare et initialise une String vSorties. Puis on met toutes les clés (donc les sorties) dans une variable vKeys, qui est finalement une sorte de tableau contenant un ensemble de clés, grâce à la méthode *keySet().* Enfin, on parcours ce « tableau » vKeys grâce à la boucle for each, qui va en fait attribuer à une variable vExits, chaque clé (=sortie) à chaque nouveau tour de boucle. Dans le bloc d'instruction du for, on concatène dans la String vSorties, chaque clé qui sont elles-mêmes de type String, pour obtenir toutes les sorties possibles à la suite.

Finalement on retourne la String vSorties.

8) Exercice 7.10.1&2

JavaDoc complétée et générée.

file:///C:/Users/thado/OneDrive/Documents/IPO/TP3/zuul-v4/doc/Game.html

9) Exercice 7.11 :

On ajoute une méthode *getLongDescription()* qui permet d'afficher en détail la description de la pièce avec éventuellement des objets et les sorties.

```
/**
 * Renvoie une description détaillée de cette pièce
 * @return String description de la pièce avec les sorties
 */
public String getLongDescription(){
    return " Vous êtes "+this.aDescription+".\n"+this.getExitsString();
}
```

Cela permet de simplifier ensuite le code de la méthode *printLocationInfo()* dans la classe Game.

```
/**
 * Procédure qui affiche les sorties possibles et la description quand on est dans une pièce.
 */
private void printLocationInfo(){
    System.out.println(this.aCurrentRoom.getLongDescription());
}
```

10) Exercice 7.14 :

On ajoute la commande « look » dans la classe CommandsWords :

```
//Attribut
private final String[] aValidCommands={"quit", "help", "go","look"};
```

Puis on y ajoute sa fonctionnalité dans la classe Game qui est d'avoir à nouveau une description de la pièce où se trouve l'utilisateur.

```
/**
 * Procédure qui affiche les sorties possibles et la description quand on est dans une pièce
 * si l'utilisateur le demande à nouveau
 */
private void look(){
    System.out.println(this.aCurrentRoom.getLongDescription());
}
```

Et on modifie ProcessCommand :

```
//Demander la description de la pièce
else if(pD.getCommandWord().equals("look")){
    this.look();
    return false;
}
```

11) Exercice 7.15 :

On ajoute la commande « eat » dans la classe CommandsWords :

```
//Attribut
private final String[] aValidCommands={"quit", "help", "go","look", "eat"};
```

Puis on y ajoute sa fonctionnalité dans la classe Game.

```
/**
 * Procédure qui permet à l'aventurier de se nourrir
 */
private void eat(){
    System.out.println("Tu viens de manger, tu n'as maintenant plus faim.");
}
```

Et on modifie ProcessCommand :

```
//Se nourrir
else if(pD.getCommandWord().equals("eat")){
    this.eat();
    return false;
}
```

12) Exercice 7.16 :

On parcours le tableau contenant toutes les commandes disponibles grâce à la boucle « for each » dans la procédure *showAll()* de la classe CommandWords et on les affiche les unes à la suite des autres.

```
/**
 * Affiche toutes les commandes valides du System.out.
 */
public void showAll(){
    for(String command : aValidCommands){
        System.out.print(command+ " ");
    }
    System.out.println();
}
```

On créer une méthode *showCommands()* dans Parser qui va appeler la méthode *showAll()* dans CommandWords et va être ensuite elle-même appelé dans la méthode *printHelp()* de la classe Game. La méthode *showCommands()* joue un rôle de relai entre la classe CommandWords et Game.

```
/**
 * Affiche la liste des commandes valides
 */
public void showCommands(){
    aValidCommands.showAll();
}
```

```
/**
 * Procédure qui sera exécutée lorsque l'on tapera "help".
 */
private void printHelp(){
    System.out.println("You are lost. You are alone.");
    System.out.println("Your command words are:");
    aParser.showCommands();
}
```

13) Exercice 7.18 :

On modifie la méthode *showAll()* de la classe CommandWords, en la renommant *getCommandList()* dont le rôle va non pas d'afficher les sorties mais seulement de la retourner.

```
/**
 * Affiche toutes les commandes valides du System.out.
 */
public String getCommandList(){
    String vCommands="";
    for(String command : aValidCommands){
        vCommands+=command+" ";
    }
    return vCommands;
}
```

On appelle la méthode précédemment créé dans la méthode de la classe Parser *showCommands()* que l'on va renommé *getCommands()*, et modifié de telle sorte qu'elle retourne la liste des commandes (et ne les affiches pas).

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

Enfin, on ajoute dans la procédure *printHelp()* de la classe Game, l'appel de la méthode précédente afin que la procédure s'occupe de seulement afficher les informations.

```
private void printHelp(){
    System.out.println("You are lost. You are alone.");
    System.out.println("Your command words are:");
    System.out.println(aParser.getCommands());
}
```

14) Exercice 7.18.1 :

Comparaison avec zuul-better faite.

15) Exercice 7.18.3 :

Maison playmobil

Ensuite remplacé par

Autoroute(voiture en bois)

Ensuite remplacé par :

Boutique playmobil

Ensuite remplacé par

Lit enfant

Avion lego

Ferme playmobil

Boite à jouet

Cuisine playmobil

Tapis de foot

Bureau

Temple lego

Sous le lit

16) Exercice 7.18.4 :

Titre : “ Playmobil présente : The bedroom exploration ! “ et incorporé dans le jeu.

17) Exercice 7.18.6 :

Les classes GameEngine et UserInterface ont bien été ajouté au code. La classe Room a bien été modifiée (méthode+attribut).

```
 //Attributs
private String aDescription;
private HashMap<String,Room> aExits;
private String aImageName;
```

```java
/**
 * Return a string describing the room's image name
 */
public String getImageName()
{
return this.aImageName;
}
```

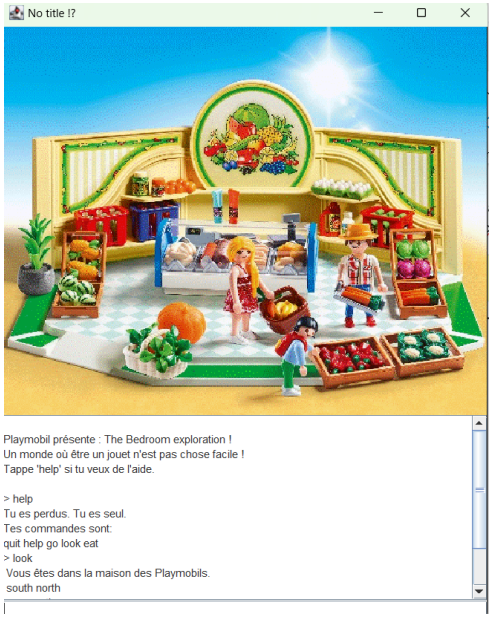
C'est la classe UserInterface qui permet d'afficher du texte, des images, activer/désactiver la zone de texte… La classe GameEngine reprend une grande partie du code de l'ancienne classe Game, mais on ajoute un nouvel attribut (aGui) :

```java
private Parser        aParser;
private Room          aCurrentRoom;
private UserInterface aGui;
```

La méthode *play()* a disparu. Aussi on renomme la méthode processCommand() par *interpretCommand()* en la modifiant (dans la classe GameEngine).

```java
public void interpretCommand( final String pCommandLine )
{
    this.aGui.println( "> " + pCommandLine );
    Command vCommand = this.aParser.getCommand( pCommandLine );
```

Le jeu est testé :



18) Exercice 7.18.7 :

ActionListener est une interface qui sert à contrôler les événements liés aux actions des composants graphiques tels que les boutons par exemple.

Ainsi, cette interface possède deux méthodes : *addActionListenner()* qui doit « écouter » les événements associé à un composant, et *actionPerformed()* qui est appelé quand on doit définir le

comportement à exécuter quand un événement se produit. En effet, son corps comporte les instructions à exécuter dans chaque situation.

19) Exercice 7.18.8 :

Pour ajouter un bouton, on déclare un nouvel attribut en appelant la classe JButton

```
private JButton    aButton;
```

Ensuite, on définit l'emplacement du bouton, et on y applique la méthode *addActionListener()* .

Puis on veut savoir dans la méthode *actionPerformed()* si l'évennement provient d'un clic de bouton ou non. Pour cela on teste la méthode *getSource*() sur un ActionEvent. Si oui, alors on va exécuter les instructions qui correspondent à l'utilité du bouton. Sinon, *actionPerformed*() va faire appel à *processCommand*().

```
@Override public void actionPerformed( final ActionEvent pE )
{
    // no need to check the type of action at the moment
    // because there is only one possible action (text input) :

    if(pE.getSource()==this.aButton){
        this.aEngine.interpretCommand( "look" );
    }
    this.processCommand(); // never suppress this line
} // actionPerformed(.)
```

```
this.aImage = new JLabel();
this.aButton = new JButton("look");

JPanel vPanel = new JPanel();
vPanel.setLayout( new BorderLayout() ); // ==> only five places
vPanel.add( this.aImage, BorderLayout.NORTH );
vPanel.add( vListScroller, BorderLayout.CENTER );
vPanel.add( this.aEntryField, BorderLayout.SOUTH );
vPanel.add( this.aButton,BorderLayout.EAST);

this.aMyFrame.getContentPane().add( vPanel, BorderLayout.CENTER );

// add some event listeners to some components
this.aEntryField.addActionListener( this );
this.aButton.addActionListener( this );
```

20) Exercice 7.19.2 :

Répertoire « images » créé à la racine du projet et les images ont bien été incorporées dans le projet.

```
//On donne la description de chaque pièce créées
Room vMaison = new Room("dans la maison des Playmobils", "images/maison.gif");
Room vAutoroute = new Room("sur l'autoroute des voitures en bois", "images/autoroute.gif");
Room vBoutique = new Room("dans la boutique Playmobil", "images/boutique.gif");
Room vLit = new Room("sur le lit", "images/lit.gif");
Room vAvion = new Room("dans un avion LEGO", "images/avion.gif");
Room vFerme = new Room("dans la ferme Playmobil", "images/ferme.gif");
Room vCoffre = new Room("dans la boite à jouets", "images/boite.gif");
Room vCuisine = new Room("dans la cuisine Playmobil", "images/cuisine.gif");
Room vTapisDeFoot = new Room("sur le tapis de foot", "images/tapi.gif");
Room vBureau = new Room("sur le bureau", "images/bureau.gif");
Room vTempleLego = new Room("dans le temple LEGO", "images/temple.gif");
Room vSousLeLit = new Room("en-dessous du lit", "images/lit.gif");
```

21) Exercice 7.20 :

On créé la classe Item qui créera tous les Items qui possiblement existe dans une pièce. Aussi on ajoute deux getters et deux setters (description de l'item+poids).

```
public class Item
{
    private String aItemDescription;
    private int aItemWeight;

    /**
     * Constructeur naturel. Créer un Item décrit par la chaîne.
     * @param pD String pour intialiser aItemDescription
     * @param pW Entier pour initialiser aItemWeight
     */
    public Item(final String pD, final int pW){
        this.aItemDescription=pD;
        this.aItemWeight=pW;
    }
```

Ensuite on ajoute 3 nouvelles méthodes dans la classe Room afin de pouvoir ajouter un Item dans chaque pièce mais aussi ajouter la description de l'item dans *getLongDescription*()

```
 /**
  * Procédure qui ajoute un Item à la pièce
  * @param Item pI de la pièce
  */
public void setItem(final Item pI){
    this.aItem=pI;
}

/**
 * Retourne l'Item de la pièce
 */
public Item getItem(){ return this.aItem;}

/**
 * Fonction qui renvoie la description de l'Item présent dans la pièce
 */
public String getItemString(){
    if(this.aItem.getItemDescription()==null){
        return "Il n'y a pas d'Item ici";
    }
    else{
        return "Il y a "+this.aItem.getItemDescription();
    }
}
```

```
/**
 * Renvoie une description détaillée de cette pièce
 * @return String description de la pièce avec les sorties
 */
public String getLongDescription(){
    return " Vous êtes "+this.aDescription+".\n"+this.getExitsString()+"\n"+this.getItemString();
}
```

22) Exercice 7.21 :

Chaque Item est créé dans la classe GameEngine (comme les pièces) avec leur description et leur poids. Ensuite chaque Item est mis dans les pièces qui leur correspondent grâce à la fonction *setItem*(), précédemment créé dans la classe Room. Ainsi la classe GameEngine s'occupe d'à la fois de la création des Items mais aussi de l'affichage.

```
//Création des Items
Item vEpee=new Item("une belle épée te permettant d'affronter le danger !", 0);
Item vCoeur=new Item("un coeur qui signifit une nouvelle vie...", 10);
Item vEchelle=new Item("une échelle te permettant d'escalader des peluches", 10);
Item vFourche=new Item("une fourche appartenant au fermier !", 15);
Item vClee=new Item("une clée te permettant de rentrer chez toi !", 30);
//Ajout des Items
vMaison.setItem(vEpee);
vBoutique.setItem(vCoeur);
vLit.setItem(vEchelle);
vFerme.setItem(vFourche);
vCuisine.setItem(vClee);
```

23) Exercice 7.22 :

Je choisi d'ajouter un attribut aNomItem dans la classe Item par soucis de praticité. Je modifie donc le constructeur de cette classe :

```
/**
 * Constructeur naturel. Créer un Item décrit par la chaîne.
 * @param pD String pour intialiser aItemDescription
 * @param pW Entier pour initialiser aItemWeight
 */
public Item(final String pN,final String pD, final int pW){
    this.aNomItem=pN;
    this.aItemWeight=pW;
    this.aItemDescription=pD;
}
```

Puis dans la classe Room, je créer la méthode *ajouteItem*() qui va simplement ajouter l'item dans l'attribut hashmap aItems afin de pouvoir créer autant d'Item que l'on souhaite.

Puis, je modifie la méthode *getItemString*() pour qu'elle parcours chaque item de l'attribut HashMap de la pièce en question et prenne sa description, afin d'en faire une chaîne contenant toutes les descriptions de tous les items de la pièce. Enfin, on utilise cette méthode dans la méthode *getLongDescription*(). J'ajoute une autre fonction qui affichera le nom des items que l'utilisateur peut prendre en plus de la description :

```
public void ajouteItem(final String pNom, final String pD, final int pW) {
     Item vItem = new Item(pNom, pD, pW);
     this.aItems.put(pNom, vItem);
 }
```

```
public String getItemStringDescription() {
     String vD = "";
     if (this.aItems.isEmpty()) {
         return vD + "Il n'y a pas d'Item ici";
     }

     for (Item vItem : this.aItems.values()) {
         vD += vItem.getItemDescription();
     }
     return "Il y a " + vD;

 }
```

```
public String getItemStringNom() {
     String vD = "";
     if (this.aItems.isEmpty()) {
         return vD;
     }

     for (Item vItem : this.aItems.values()) {
         vD += vItem.getNomItem() + "\n";
     }
     return "Tu peux prendre : " + vD;
 }
```

24) Exercice 7.22.1 :

Pour l'exercice précédent, j'ai choisi d'utiliser une HashMap car elle permet de facilement manipuler une collection d'objets (ici des Items). De plus, elle est avantageuse pour différents paramètres comme sa taille extensible ou encore la facilité d'ajout et de suppression des éléments.

25) Exercice 7.22.2 :

Items du scénario ajouté

```
//Ajout des Items
vMaison.ajouteItem("Epée","une belle épée te permettant d'affronter le danger",0);
vBoutique.ajouteItem("Coeur","un coeur qui signifit une nouvelle vie", 10);
vBoutique.ajouteItem("Epée2","une belle épée te permettant d'affronter le danger",0);
vLit.ajouteItem("Echelle","une échelle te permettant d'escalader des peluches", 10);
vFerme.ajouteItem("Fourche","une fourche appartenant au fermier", 15);
vFerme.ajouteItem("Pelle", "une pelle appartenant au fermier", 15);
vFerme.ajouteItem("Seau","un seau appartenant au fermier", 15);
vFerme.ajouteItem("Râteau", "un râteau appartenant au fermier", 15);
vCuisine.ajouteItem("Clée","une clée te permettant de rentrer chez toi", 30);
```

26) Exercice 7.23 :

On ajoute un attribut aPreviousRoom dans la classe Room et on l'initialise dans la méthode *goRoom*() avec la pièce courante, juste avant de changer de pièce:

```
// Try to leave current room.
Room vNextRoom = this.aCurrentRoom.getExit( vDirection );
Room vPreviousRoom=this.aCurrentRoom;
this.aPreviousRoom=vPreviousRoom;
```

Puis on ajoute la méthode back, toujours dans la classe Room

```
private void back(final Command pD){
    this.aCurrentRoom=this.aPreviousRoom;
    printLocationInfo();
    if ( this.aCurrentRoom.getImageName() != null )
            this.aGui.showImage( this.aCurrentRoom.getImageName() );
}
```

27) Exercice 7.24 :

On veut en fait que l'utilisateur ne rentre pas de second mot, donc nous allons utiliser la méthode *hasSecondWord()* pour le vérifier. Ensuite, nous avons le cas où l'utilisateur veut revenir en arrière alors qu'il est dans la pièce initiale (maison). Pour cela on veut savoir si l'attribut aPreviousRoom a été initialisé ou non.

```
if(pD.hasSecondWord()){
    this.aGui.println("Tu ne peux pas retourner en arrière dans un direction précise !");
    return;
}
if(this.aPreviousRoom==null){
    this.aGui.println("Il n'existe pas de pièce précédente !");
    return;
}
```

28) Exercice 7.25 :

Nous pouvons seulement retourner en arrière qu'une seule fois et non deux ni trois. En effet, je reste coincé dans la même pièce, alors que j'entre deux fois d'affiler back.

29) Exercice 7.26 :

Dans la méthode *goRoom*(), on ajoute chaque nouvelle pièce courante dans la stack après l'avoir importé dans classe GameEngine

```
// Try to leave current room.
Room vNextRoom = this.aCurrentRoom.getExit( vDirection );
Room vPreviousRoom=this.aCurrentRoom;
this.aStack.push(vPreviousRoom);
```

Puis on modifie la méthode *back*() : pour cela, on visualise le dernier élément ajouté à la pile et correspondant à la pièce précédente, grâce à la méthode *peek*(), puis on supprime cet élement grâce à la procédure *pop*(), après l'avoir attribué à l'attribut aCurrentRoom.

```
private void back(final Command pD){
    if(pD.hasSecondWord()){
        this.aGui.println("There is no previous direction !");
        return;
    }
    this.aPreviousRoom=this.aStack.peek();
    this.aCurrentRoom=this.aPreviousRoom;
    this.aStack.pop();
    printLocationInfo();
}
```

30) Exercice 7.26.1 :

Les 2 javadoc ont bien été généré en appliquant d'abord :

```
DIR "C:\Program Files\BlueJ\jdk\bin" (la javadoc.exe existe bien)
```

Puis,

```
SET PATH="C:\Program Files\BlueJ\jdk\bin"
```

Ensuite en faisant javadoc --version, j'obtiens :

javadoc 11.0.10

Je copie le chemin :

C:\Users\thado\OneDrive\Documents\IPO\projet zuul\Version intermédiaire Zuul Thaïs Castillo\zuul-v4

C:\Users\thado\OneDrive\Documents\IPO\projet zuul\version-intermediaire-zuul-thais-castillo\zuul-v4

Et enfin :

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

31) Exercice 7.28.1 :

Tout d'abord, j'ajoute une nouvelle commande dans la classe CommandWords (« test »), puis je vais créer une méthode *test*() dans la classe GameEngine. Cette méthode va prendre le second mot de la commande entrée en paramètre pour obtenir le nom du fichier (avec *getSecondWord*()).

```
public void test(final Command pD){
    if(!pD.hasSecondWord()){
        this.aGui.println("Test what ?");
        return;
    }
    String vFichier=pD.getSecondWord();
    this.lecture(vFichier);
}
```

Celle-ci va elle-même contenir une méthode qui va lire le nom d'un fichier et le traiter (*lecture*())

```
private void lecture( final String pNomFichier )
{
    Scanner vSc;
    try { // pour "essayer" les instructions suivantes
        String vNomFichier=pNomFichier+".txt";
        vSc = new Scanner( new File( vNomFichier ) );
        while ( vSc.hasNextLine() ) {
            String vLigne = vSc.nextLine();
            interpretCommand(vLigne);
            // traitement de la ligne lue
        } // while
        vSc.close(); //fermeture du fichier
    } // try
    catch ( final FileNotFoundException pFNFE ) {
        this.aGui.println("Fichier non trouvé");
    } // catch
} // lecture
```

On traite également du cas où l'utilisateur entre un nom de fichier sans l'extension « .txt ». Ensuite, grâce aux instructions try/catch qui nous permettent de gérer les exceptions, on va lire ligne par ligne le fichier entré par l'utilisateur pour ensuite traiter les instructions qui sont entrés dans le fichier, grâce à la méthode *interpretCommand*(). Enfin on n'oublie pas de fermer le fichier après avoir lu la dernière ligne.

Dans le cas où l'on rencontre une exception, on affiche un message d'erreur.

32) Exercice 7.28.2

Fichier 1 (court) test1.txt :

go south

go south

go down

Fichier 2 (pour gagner) test-ideal.txt :

go north

go east

go south

go south

go east

Fichier 3 (pour tout tester) test-tout.txt :

go north

go east

go east

go west

go south

go south

go south

go south

go west

go down

go east

go west

go up

go north

back

eat

look

33) Exercice 7.29 :

On réparti le code de *goRoom*() et *back*() entre les deux classes Player et GameEngine. De ce fait, nous modifions les attributs en premier lieux :

Pour la classe GameEngine :

```
private Parser        aParser;
private UserInterface aGui;
private Player        aPlayer;
```

Pour la classe Player :

```
private String        aNamePlayer;
private Room          aCurrentRoom;
private Stack<Room>   aStack;
private Room          aPreviousRoom;
```

En effet, toutes les attributions de pièces précédentes/actuelles se retrouvent dans la classe Player pour s'adapter à chaque joueur.

On ajoute le constructeur :

```
public Player(final String pN, final Room pCurrentRoom){
    this.aNamePlayer=pN;
    this.aStack=new Stack<>();
    this.aCurrentRoom=pCurrentRoom;
}
```

Où l'on va donner un nom au joueur et la salle où il se trouve initialement

```
/**
 * @return String aNamePlayer
 */
public String getNamePlayer(){ return this.aNamePlayer;}
```

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

```
/**
 * @return Stack<Room> aStack
 */
public Stack<Room> getLastRoom(){ return this.aStack;}
```

Puis on ajoute des getters qui nous seront utile dans la classe GameEngine

On ajoute les méthodes *back*() et *goRoom*() dans la classe Player :

```
public void goRoom(final String pD){
    Room vNextRoom = this.aCurrentRoom.getExit( pD );
    this.aStack.push(this.aCurrentRoom);
    this.aCurrentRoom=vNextRoom;
}

/**
 * Procédure permettant de revenir sur ses pas
 */
public void back(){
    this.aPreviousRoom=this.aStack.peek();
    this.aCurrentRoom=this.aPreviousRoom;
    this.aStack.pop();

}
```

On modifie donc *goRoom*() dans GameEngine :

```
public void goRoom( final Command pCommand )
{
    if ( ! pCommand.hasSecondWord() ) {
        this.aGui.println( "Go where?" );
        return;
    }
    String vDirection = pCommand.getSecondWord();

    if(this.aPlayer.getCurrentRoom().getExit(vDirection)==null){
        this.aGui.println( "There is no door!" );
        return;
    }
    this.aPlayer.goRoom(vDirection);
    printLocationInfo();
}
```

Puis *back*() :

```
private void back(final Command pD){
    if(pD.hasSecondWord()){
        this.aGui.println("Tu ne peux pas retourner en arrière dans un direction précise !");
        return;
    }

    if(this.aPlayer.getLastRoom().empty()){
        this.aGui.println("Il n'existe pas de pièce précédente !");
        return;
    }
    this.aPlayer.back();
    printLocationInfo();

}
```

Etant donné que nous avons supprimé aCurrentRoom de GameEngine, nous devons également modifié les méthodes *look*() et *printLocation*() :

```
private void look(){
    this.aGui.println( this.aPlayer.getCurrentRoom().getLongDescription());
}
```

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

Enfin, on initialise aPlayer dans *createRoom*() :

```
this.aPlayer=new Player("Joueur", vMaison);
```

Ainsi l'utilité de la classe Player est seulement d'effectuer les déplacements du joueur dans le jeu et GameEngine va afficher les informations et appeler les fonctions correspondantes de la classe Player pour effectuer les déplacements.

34) Exercice 7.30 :

Tout d'abord on ajoute les nouvelles commandes dans la classe CommandWords et on ajoute à la méthode *interpretCommand*() :

```
else if (vCommandWord.equals("take") )
    this.take(vCommand);
else if (vCommandWord.equals("drop") )
    this.drop(vCommand);
```

Pour faire ces deux fonctions, on répartit le code à la fois dans la classe Player et dans la classe GameEngine. En effet, la classe GameEngine, comme précédemment s'occupera seulement d'afficher les messages alors que la classe Player s'occupera d'exécuter les instructions.

Ainsi, dans la classe Player, la méthode take est une boolean car on veut tester si l'item entré est valide ou non, pareil pour drop. Dans le cas de take, on ajoute l'objet dans une HashMap préalablement déclaré en attribut, et on le supprime de la pièce courante. Puis dans le cas de drop, on supprime l'objet de la HashMap et on l'ajoute à la pièce courante.

Ainsi avant, on créé ces deux méthodes qui vont permettre d'ajouter ou supprimer des Items de la HashMap contenant tous les Items dans la classe Room.

```
public void removeItem(final Item pI){
    this.aItems.remove(pI.getNomItem());
}

public void addItem(final Item pI) {
     this.aItems.put(pI.getNomItem(), pI);
 }
```

Dans la classe Player :

```
public boolean take(final String pI){

    if(pI==null){
        return false;
    }
    Item vItem=this.aCurrentRoom.getItem(pI);
    this.aItems.put(pI,vItem);
    this.aCurrentRoom.removeItem(vItem);
    return true;
}
```

```
public boolean drop(final String pI){
    Item vItem=this.aItems.get(pI);
    if(vItem==null){
        return false;
    }

    this.aItems.remove(pI);
    this.aCurrentRoom.addItem(vItem);
    return true;
}
```

Dans la classe GameEngine :

```
private void take(final Command pD){
    if(pD.hasSecondWord()==false){
        this.aGui.println("take what ?");
        return;
    }
    String vItem = pD.getSecondWord();
    if(this.aPlayer.getItems().size()==1){
        this.aGui.println("Tu ne peux pas prendre d'Item supplémentaire !!");
        return;
    }
    if(this.aPlayer.take(vItem)){
        this.aGui.println("Tu possède maintenant "+vItem);
        return;
    }
    this.aGui.println("Tu n'as rien pu prendre !!");
}
```

```
private void drop(final Command pD){
    if(pD.hasSecondWord()==false){
        this.aGui.println("drop what ?");
        return;
    }
    String vItem = pD.getSecondWord();
    if(this.aPlayer.getItems().isEmpty()){
        this.aGui.println("Tu ne possède pas d'Item !!");
        return;
    }
    if(this.aPlayer.drop(vItem)){
        this.aGui.println("Tu ne possède plus "+vItem);
        return;
    }
    this.aGui.println("Tu n'as rien pu jeter !!");
}
```

Aussi, je souhaite afficher tous les Items que l'utilisateur porte sur lui à chaque fois qu'il change de pièce pour vérifier que le programme marche je créer donc la méthode *getAllItemStringNom*() dans la classe Player et que je vais ajouter à *printLocationInfo*().

```
/**
 * Procédure permettant d'afficher les éléments de la hashmap
 * @param pH HashMap<String, Item>
 */
public String getAllItemsString(final HashMap<String, Item> pH){
Set<String> vKey = pH.keySet();
String vD="";
for ( String pI : vKey ){
    vD=vD+pI+" ";
}
return vD;
}
```

35) Exercice 7.31 :

On va venir retirer l'accesseur à la HashMap et on va supprimer un if dans la méthode drop et take de la classe GameEngine, qui permettait de savoir si l'utilisateur pouvait prendre un autre item:

```
private void take(final Command pD){
    if(pD.hasSecondWord()==false){
        this.aGui.println("take what ?");
        return;
    }
    String vItem = pD.getSecondWord();
    if(this.aPlayer.take(vItem)){
        this.aGui.println("Tu possède maintenant "+vItem);
        return;
    }
    this.aGui.println("Tu n'as rien pu prendre !!");
}
```

```
private void drop(final Command pD){
    if(pD.hasSecondWord()==false){
        this.aGui.println("drop what ?");
        return;
    }
    String vItem = pD.getSecondWord();
    if(this.aPlayer.drop(vItem)){
        this.aGui.println("Tu ne possède plus "+vItem);
        return;
    }
    this.aGui.println("Tu n'as rien pu jeter !!");
}
```

Aussi, on va modifier la méthode *getAllItemStringNom*() de sorte à ce qu'elle ne prenne aucun paramètre et renvoi directement une String qui indique le nombre d'items que possède l'utilisateur.

```
public String getAllItemsString(){
if(this.aItems.isEmpty()){
    return "Tu ne possède pas d'Item";
}
Set<String> vKey = this.aItems.keySet();
String vD="";
for ( String pI : vKey ){
    vD=vD+pI+" ";
}
return "Tu possède "+vD;
}
```

On l'ajoute à nouveau dans les deux méthodes *look*() et *printLocationInfo*().

Les commandes test ont bien été modifiée en conséquence :

test-ideal :

take epee

go north

go east

go south

go south

take clee

go east

test-tout :

take epee

go north

go east

go east

take fourche

take pelle

take seau

take rateau

go west

go south

go south

take clee

go south

go south

go west

go down

go east

go west

go up

go north

take coeur

take epee

back

drop epee

drop epee

drop coeur

drop fourche

drop pelle

drop seau

drop clee

drop rateau

eat

look

36) Exercice 7.31.1 :

On vient dupliquer les 2 méthodes *removeItem*() et *addItem()* vers la classe ItemList. On aura alors à la fois ces deux méthodes présentes dans la classe ItemList et dans la classe Room. On fait toutes les modifications nécessaires dans la classe Room notamment au niveau des méthodes *ajouteItem*(). Aussi pour la méthodes *getItem*(), j'ai fait la même modification que pour *removeItem*() et *addItem*() (il y a donc une méthode *getItem()* dans ItemList et Room). Ensuite, j'ai ajouté un attribut ItemList dans les classes Room et Player. Je modifie légèrement l'appel des méthodes *removeItem*() et *addItem*() dans les méthodes take et drop de Player. Aussi je modifie ma méthodes d'affichage *getAllItemsString*() dans Player que je vais mettre dans la classe Item List.

Dans la classe Room :

```
/**
 * Procédure permettant de supprimer des Items
 * @param pI Item
 */
public void removeItem(final Item pI){
    this.aItems.removeItem(pI);
}
```

```
/**
 * Procédure permettant d'ajouter des Items
 * @param pI Item
 */
public void addItem(final Item pI) {
    this.aItems.addItem(pI);
}
```

Dans la classe ItemList :

```
/**
 * Procédure permettant d'ajouter des Items
 * @param pI Item
 */
public void addItem(final Item pI) {
    this.aItems.put(pI.getNomItem(), pI);
}

/**
 * Procédure permettant de supprimer des Items
 * @param pI Item
 */
public void removeItem(final Item pI){
    this.aItems.remove(pI.getNomItem());
}
```

37) Exercice 7.32 :

J'initialise 2 attributs dans la classe Player, dont un que je vais initialiser à 0 et l'autre représentera le poids max que le joueur ne devra pas dépasser. Dans la fonction *take*(), je rajoute un if pour savoir si

l'attribut que l'on incrémente du poids de l'item prit, dépasse le max que l'utilisateur peut transporter. En fonction de si nous sommes dans la fonction *take*() ou *drop*(), on additionne les poids des items ou on soustrait.

```
this.aWeightPlayer=100;
this.aWeightTotal=0;
```

```
public boolean drop(final String pI){
    Item vItem=this.aItems.getItem(pI);
    if(vItem==null){
        return false;
    }
    int vW=vItem.getItemWeight();
    this.aWeightTotal=this.aWeightTotal-vW;

    this.aItems.removeItem(vItem);
    this.aCurrentRoom.addItem(vItem);
    return true;
}
```

```
public boolean take(final String pI){
    if(pI==null){
        return false;
    }
    Item vItem=this.aCurrentRoom.getItem(pI);
    int vW=vItem.getItemWeight();
    this.aWeightTotal+=vW;
    if(this.aWeightTotal>this.aWeightPlayer){
        this.aWeightTotal=this.aWeightTotal-vW;
        return false;
    }
    this.aItems.addItem(vItem);
    this.aCurrentRoom.removeItem(vItem);
    return true;
}
```

38) Exercice 7.33 :

Pour faire cet exercice, je me sert de ma méthode précédemment crée (*getAllItemsString*()) qui était initialement créé dans ItemList. Je vais simplement légèrement la modifier de telle sorte qu'elle affiche le poids de chaque objet.

```
/**
 * Fonction permettant d'afficher les éléments de la hashmap
 * @return String
 */
public String getAllItemsString(){
if(this.aItems.isEmpty()){
    return "Tu ne possède pas d'Item";
}
Set<String> vKey = this.aItems.keySet();
String vD="";
for ( String pI : vKey ){
    Item vI=this.aItems.get(pI);
    int vW=vI.getItemWeight();
    vD=vD+pI+" "+":"+vW+"\n";
}
return "Tu possède :"+"\n"+vD;
}
```

Cette méthode est utilisé dans une autre méthode *getAllItemsString*() dans la classe Player :

```
/**
 * Fonction permettant d'afficher les éléments de la hashmap
 * @return String
 */
public String getAllItemsString(){
return this.aItems.getAllItemsString();
}
```

Enfin, j'appelle cette dernière dans la méthode inventaire dans GameEngine (après avoir bien entendu ajouter une commande inventaire).

```
/**
 * Procédure qui affiche tous les items que possède l'utilisateur et leur poids
 */
private void inventaire(){
    this.aGui.println(this.aPlayer.getAllItemsString());
}
```

39) Exercice 7.34 :

On reprend le fonctionnement de take/drop pour créer la commande eat.

Je créé une méthode eat qui sera booléenne dans la classe Player et qui vérifiera si le second mot que l'utilisateur tapera après eat sera bien « cœur », qui représente dans mon jeu le magic cookies. Aussi, lorsque l'utilisateur aura mangé le cœur, celui disparaîtra de l'inventaire.

```
public boolean eat(final String pD){
    if(pD.equals("coeur")){
        if(this.aItems.coeur()){
            this.aWeightPlayer=this.aWeightPlayer*2;
            Item vItem=this.aItems.getItem(pD);
            this.aItems.removeItem(vItem);
            return true;
        }
        return false;
    }
    return false;
}
```

Sans oublier de tester si l'utilisateur possède ou non l'objet cœur. Pour cela, je créé une méthode cœur dans la classe ItemList :

```
/**
 * Fonction permettant de savoir si l'utilisateur possède l'objet coeur dans son inventaire
 * @return boolean
 */
public boolean coeur(){
    if(this.aItems.get("coeur")==null){
        return false;
    }
    return true;
}
```

On utilise ainsi cette méthode dans l'autre méthode *eat()* présente dans la classe GameEngine :

```
/**
 * Procédure qui permet à l'aventurier de se nourrir
 */
private void eat(final Command pC){
    if ( ! pC.hasSecondWord() ) {
        this.aGui.println( "Eat what ?!" );
        return;
    }
    String vE= pC.getSecondWord();
    if(this.aPlayer.eat(vE)){
        this.aGui.println( "Tu viens de manger un coeur, ton inventaire a été doublé.");
        return;
    }
    this.aGui.println( "Tu n'as rien pu manger !");
}
```

40) Exercice 7.34.1 :

Les commandes test ont bien été modifiée en conséquence :

test-tout :

take epee

go north

go east

go east

take fourche

take pelle

take seau

take rateau

go west

go south

go south

take clee

go south

go south

go west

go down

go east

go west

go up

go north

eat coeur

take coeur

eat coeur

take epee

back

drop epee

inventaire

drop epee

drop coeur

drop fourche

drop pelle

drop seau

drop clee

drop rateau

look

test-ideal :

take epee

go south

take coeur

eat coeur

inventaire

go north

go north

go east

go east

take fourche

take pelle

take seau

take rateau

inventaire

drop fourche

drop pelle

drop seau

drop rateau

back

go south

go south

take clee

go east

41) Exercice 7.34.2 :

Les 2 javadoc ont bien été à nouveau généré.

42) Exercice 7.42 :

J'initialise un attribut de type int dans la classe UserInterface que j'appel aNbreTouch et que j'incrémente à chaque fois que la procédure *actionPerformed()* est appelé (puisque que son rôle est de géré tout les événements générés par l'utilisateur).

```
@Override 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) :
    this.aNbreTouch++;
    if(pE.getSource()==this.aButton){
        this.aEngine.interpretCommand( "look" );
    }
    this.processCommand(); // never suppress this line
} // actionPerformed(.)
```

Puis toujours dans la même classe, je fais le test avec trois commandes entrées (ensuite remplacé par 50). Si l'attribut à dépasser le nombre limite, alors j'appel une méthode *perdu*() de la classe GameEngine qui va arrêter le jeu.

```
private void processCommand()
{
    if(this.aNbreTouch>3){
        this.aEngine.perdu();
        return;
    }
    String vInput = this.aEntryField.getText();
    this.aEntryField.setText( "" );

    this.aEngine.interpretCommand( vInput );

} // processCommand()
```

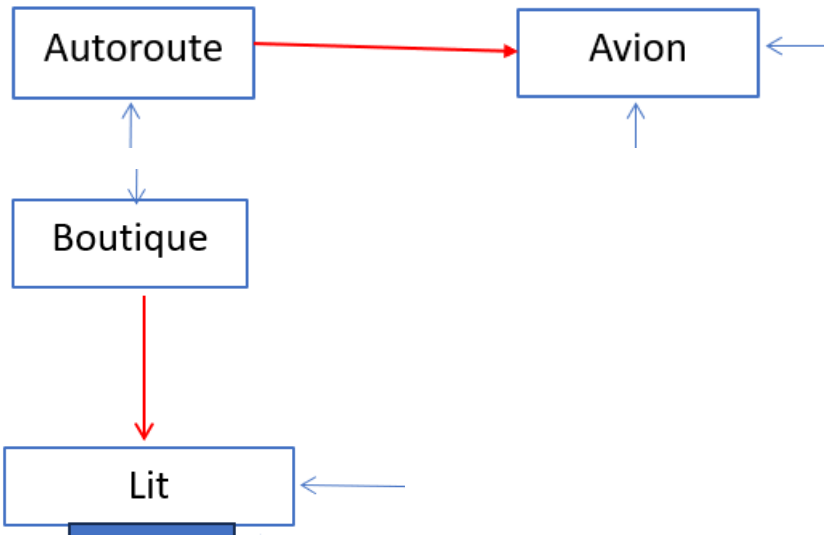
```
public void perdu(){
    this.aGui.println( "Tout ton temps s'est écoulé. Tu as perdu, ton playmobil n'as pas pu retourner dans sa boîte." );
    this.aGui.enable( false );
}
```

43) Exercice 7.42.2 :

Je vais me contenter de l'IHM actuelle.

44) Exercice 7.43 :

Mes deux TrapDoor sont



Je mets en commentaire les sorties que je veux supprimer

```
//vLit.setExit("north", vBoutique);
vLit.setExit("east", vBureau);
//On ajoute la direction "down"
vLit.setExit("down", vSousLeLit);
vAvion.setExit("east", vFerme);
vAvion.setExit("south", vTempleLego);
//vAvion.setExit("west", vAutoroute);
```

On créé une nouvelle méthode dans la classe Room, en utilisant *containsValue(),* une méthode de type boolean, fournie par la classe HashMap et qui retourne true si la clée correspondant à la valeur mise en paramètre existe, sinon false.

```
public boolean isExit(final Room pR){
    if(this.aExits.containsValue(pR)){
        return true;
    }
    return false;
}
```

On utilise ensuite cette méthode dans *back*() de Player, que l'on va d'ailleurs modifier (la procédure devient une fonction booléenne). Si *isExit*() retourne false, on vide la Stack et on retourne false, sinon on effectue les instructions de la fonction back normalement et on retourne true. On veut savoir lorsque nous sommes dans la pièce courante si l'accès à la pièce précédente existe toujours.

```
public boolean back(){
    this.aPreviousRoom=this.aStack.peek();
    if(!this.aCurrentRoom.isExit(this.aPreviousRoom)){
        this.aStack.clear();
        return false;
    }
    this.aCurrentRoom=this.aPreviousRoom;
    this.aStack.pop();
    return true;
}
```

Puis, je modifie la procédure *back*() présente dans la classe GameEngine. En effet, si *back*() de Player retourne false, j'affiche un message en conséquence, sinon j'effectue les instructions normalement.

```
private void back(final Command pD){
    if(pD.hasSecondWord()){
        this.aGui.println("Tu ne peux pas retourner en arrière dans un direction précise !");
        return;
    }

    if(this.aPlayer.getLastRoom().empty()){
        this.aGui.println("Il n'existe pas de pièce précédente !");
        return;
    }

    if(this.aPlayer.back()){
        printLocationInfo();
        return;
    }
    this.aGui.println("Tu ne peux pas retourner en arrière");
```

45) Exercice 7.43.1 :

Les 2 javadoc ont bien été à nouveau généré.

46) Exercice 7.44

Je créé la classe Beamer qui va contenir toutes les méthodes suivantes :

```
private Room    aChargeRoom;
private boolean aCharger;
private boolean aUsed;

public Beamer(final String pN,final String pD, final int pW){
    super(pN,pD,pW);
    this.aCharger=false;
    this.aUsed=false;
}

public boolean getCharger(){ return this.aCharger;}

public boolean getUsed(){ return this.aUsed;}

public Room getChargeRoom(){ return this.aChargeRoom;}

public void charge(final Room pR){
    this.aChargeRoom=pR;
    this.aCharger=true;
}

public void use(){
    this.aUsed=true;
}
```

- Il y a donc des getters qui vont être utile dans la classe GameEngine
- L'attribut aCharger permet de savoir si le téléporteur est chargé ou non
- L'attribut aUsed va me permettre d'interdire à l'utilisateur d'utiliser plusieurs fois le téléporteur
- Enfin, je vais créé le Beamer dans la classe GameEngine :

```
this.aBeamer=new Beamer("teleporteur","Un téléporteur qui te permet de te ramener dans une pièce que tu peux charger"+"\n ",50);
vTapisDeFoot.ajouteBeamer(this.aBeamer);
```

J'ajoute les commandes « charge » et « use »dans la classes Command et complète la méthode *interpretCommand()* dans la classe GameEngine.

Aussi, j'ajoute la méthode *teleporteur()* dans la classe itemList pour savoir si l'utilisateur a bien le teleporteur dans son inventaire

```
public boolean teleporteur(){
    if(this.aItems.get("teleporteur")==null){
        return false;
    }
    return true;
}
```

Je l'utilise donc ensuite dans la classe Player et j'ajoute en plus une méthode qui me permettra de changer de pièce quand j'utiliserai le téléporteur.

```
public boolean teleporteurPresent(){//on veut
    if(this.aItems.teleporteur()){
            return true;
    }
    return false;
}

public void changeRoom(final Room pR){
    this.aCurrentRoom=pR;
}
```

Enfin, voici les deux méthodes dans GameEngine qui vont à la fois me permettre de charger et d'utiliser mon téléporteur

La méthode charge :

```
private void charge(final Command pD){
    if(pD.hasSecondWord()){
        this.aGui.println("Tu ne peux pas charger une pièce en particulier !");
        return;
    }

    if(this.aBeamer.getUsed()){
        this.aGui.println("Tu ne peux pas utiliser ton téléporteur plusieurs fois !!");
        return;
    }

    if(!this.aPlayer.teleporteurPresent()){
        this.aGui.println("Ton téléporteur n'est pas dans ton inventaire");
        return;
    }

    if(this.aBeamer.getCharger()){
        this.aGui.println("Ton téléporteur est déjà chargé !!");
        return;
    }

    if(this.aPlayer.teleporteurPresent()){
        this.aBeamer.charge(this.aPlayer.getCurrentRoom());
        this.aGui.println("Ton téléporteur a pu être recharger");
        return;
    }
    this.aGui.println("Tu n'as pas pu charger ton téléporteur");
}
```

La méthode use :

```
if(pD.hasSecondWord()){
    this.aGui.println("Tu ne peux pas retourner dans une pièce en particulier !");
    return;
}

if(!this.aPlayer.teleporteurPresent()){
    this.aGui.println("Ton téléporteur n'est pas dans ton inventaire");
    return;
}

if(this.aBeamer.getUsed()){
    this.aGui.println("Tu ne peux pas utiliser ton téléporteur plusieurs fois !!");
    return;
}

if(!this.aBeamer.getCharger()){
    this.aGui.println("Ton téléporteur n'as pas été rechargé !!");
    return;
}

if(this.aBeamer.getCharger()){
    this.aPlayer.changeRoom(this.aBeamer.getChargeRoom());
    this.aBeamer.use();
    this.aGui.println("Tu t'es téléporté vers "+this.aBeamer.getChargeRoom());
    this.printLocationInfo();
    return;
}
this.aGui.println("Tu n'as pas pu utiliser ton téléporteur !!");
```

47) Exercice 7.45.1 :

test-tout :

take epee

go south

take coeur

go north

go north

go east

go east

eat coeur

take fourche

take pelle

take seau

take rateau

drop fourche

drop pelle

drop seau

drop rateau

go west

go south

go south

take clee

go south

go west

go down

go east

take teleporteur

inventaire

go west

go up

go east

go north

charge

go south

go west

go down

back

use

inventaire

look

test-ideal :

take epee

go south

take coeur

eat coeur

inventaire

go south

go down

go east

take teleporteur

go west

go up

go east

go north

charge

go north

go north

go east

take fourche

take pelle

take seau

take rateau

inventaire

drop fourche

drop pelle

drop seau

drop rateau

back

use

take clee

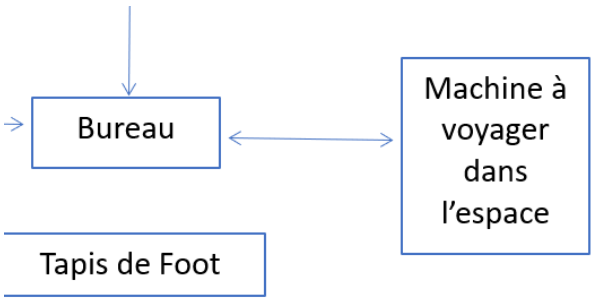
go east

48) Exercice 7.45.2 :

Les 2 javadoc ont bien été à nouveau généré.

49) Exercice 7.46 :

Tout d'abord, on créer une nouvelle pièce qui correspond à la TransporterRoom (machine à voyager dans l'espace).



Ensuite, on créé un nouvel attribut static de type List qui sera en fait une liste qui contiendra toutes les pièces du jeu (sauf la machine à voyager dans l'espace). Aussi, on associe un indexe à chaque pièce du jeu dans la liste et on créer un accesseur. On choisi de mettre l'attribut et l'accesseur en static pour éviter de créer de devoir créer un objet GameEngine dans la classe RoomRandomizer (où on utilise l'accesseur).

```
private static List<Room>   aTabRooms= new ArrayList<Room>();
```

```
        //Ajoute les pièces dans une hashmap
        this.aTabRooms.add(0,vMaison);
        this.aTabRooms.add(1,vAutoroute);
        this.aTabRooms.add(2,vBoutique);
        this.aTabRooms.add(3,vLit);
        this.aTabRooms.add(4,vAvion);
        this.aTabRooms.add(5,vFerme);
        this.aTabRooms.add(6,vCoffre);
        this.aTabRooms.add(7,vCuisine);
        this.aTabRooms.add(8,vTapisDeFoot);
        this.aTabRooms.add(9,vBureau);
        this.aTabRooms.add(10,vTempleLego);
        this.aTabRooms.add(11,vSousLeLit);

    }

    public static List<Room> getTabRooms(){
        return aTabRooms;
    }
```

On créer les deux nouvelles classes :

```
public class TransporterRoom extends Room
{
    private RoomRandomizer aRandom;

    /**
     * Constructeur naturel
     * @param pD String pour initialiser aDescription dans la classe Room.
     * @param pImage String pour ajouter une image dans la classe Room.
     */
    public TransporterRoom(final String pD, final String pImage){
        super(pD,pImage);
        this.aRandom= new RoomRandomizer();
    }

    /**
    * Fonction qui retourne une pièce aléatoire si on se déplace dans n'importe quelle direction pD.
    * @param pD String correspondant à la direction
    * @return Room qui existe dans cette direction (peut valoir null si elle n'existe pas).
    */
    @Override
    public Room getExit(String pD){
        return this.aRandom.findRandomRoom();
    }
}
```

```
public class RoomRandomizer
{
    private int aRandomNumber;

    /**
     * Constructeur naturel.
     * Il va initialiser l'attribut aRandomNumber en générant un chiffre au hasard entre et 0 et 11 inclus.
     */
    public RoomRandomizer(){
        Random random=new Random();
        this.aRandomNumber=random.nextInt(11);
    }

    /**
     * Fonction servant à générer une Room au hasard.
     * @return Room correspondant à une pièce aléatoire
     */
    public Room findRandomRoom(){
        List<Room> vR=GameEngine.getTabRooms();
        return vR.get(aRandomNumber);
    }
}
```

La première (TransporterRoom) est une sous classe de Room et va permettre de redéfinir la méthode getExit() dont on veut donner une autre fonction que l'on va d'ailleurs définir spécifiquement dans la seconde classe (RoomRandomizer). En effet, celle-ci possède un seul attribut que l'on va initialiser grâce à la classe Random qui va générer un chiffre entre 0 et 11 inclus (le nombre de pièce). Cela va nous permettre ensuite de retrouver la pièce correspondant à l'indexe aléatoire dans la méthode findRandomRoom().

```
    TransporterRoom vMachine =new TransporterRoom("dans la machine à voyager dans l'espace", "images/machine.jpg");
```

Evidemment, on créer l'objet TransporterRoom dans createRooms().

50) Exercice 7.46.1

Tout d'abord, on créer deux attribut : aTestMode va nous servir à tester si nous sommes dans un fichier test ou non et aRoomTestMode va stocker la String correspondant à la pièce que l'on veut mémoriser. Également, on créé des accesseurs.

```
private static boolean        aTestMode;
private static String           aRoomTestMode;
```

```
/**
 * Accesseur de la String aRoomTestMode
 * @return String
 */
public static String getRoomTestMode(){
    return aRoomTestMode;
}

/**
 * Accesseur du booléen aTestMode
 * @return boolean
 */
public static boolean getTestMode(){
    return aTestMode;
}
```

On va donc initialiser aTestMode à false au début puis dans la méthode test on va le mettre à true pour signaler que nous sommes effectivement dans un fichier test. Ensuite on ajoute la commande dans la classe CommandWords et on l'ajoute dans *interpretCommand()* en vérifiant d'abord si l'attribut aTestMode est à false ou true. Si c'est à false on renvoie un message d'erreur, sinon on utiliser la méthode alea.

```
else if (vCommandWord.equals("alea") )
    if (!this.aTestMode){
        this.aGui.println( "Commande seulement utilisable dans un fichier test." );
        return;
    }
    else
        this.alea(vCommand);
```

Dans la méthode alea, nous allons initialiser aRoomTestMode : soit à null s'il n'y a pas de second mot, soit on mémorise la valeur.

```
/**
 * Procédure permettant de savoir à l'avance où sera téléporter l'utilisateur ou pas
 *  @param pD Command
 */
private void alea(final Command pD){
    if(!pD.hasSecondWord()){
        this.aRoomTestMode=null;
        return;
    }
    this.aRoomTestMode=pD.getSecondWord();
}
```

Puis on modifie la méthode *findRandomRoom()* dans RoomRandomizer où l'on va en fonction de la valeur de l'attribut aRoomTestMode, soit générer une pièce au hasard, soit convertir la String en int grâce à la fonction *parseInt()* de la classe Integer (je me suis aidé de ChatGBT pour cette partie) et ainsi donc aller à la pièce voulue.

```java
/**
 * Fonction servant à générer une Room au hasard.
 * @return Room correspondant à une pièce aléatoire
 */
public Room findRandomRoom(){
    List<Room> vR=GameEngine.getTabRooms();
    if (GameEngine.getRoomTestMode()==null){
        return vR.get(aRandomNumber);
    }
    int vNbre=Integer.parseInt(GameEngine.getRoomTestMode());
    return vR.get(vNbre);

}
```

On change les fichiers test (seulement test-tout) :

alea 2

take epee

go south

take coeur

go north

go north

go east

go east

eat coeur

take fourche

take pelle

take seau

take rateau

drop fourche

drop pelle

drop seau

drop rateau

go west

go south

go south

take clee

go south

go west

go down

go east

take teleporteur

inventaire

go west

go up

go east

go north

charge

go south

go west

go down

back

use

inventaire

look

go south

go east

go north

51) Exercice 7.46.3

Tous les commentaires javadoc ont bien été réalisé

52) Exercice 7.46.4

Les 2 javadoc ont bien été à nouveau généré.