ESIEE Paris

Edgar’s Journey

Rapport Projet Zuul encadré par Denis Bureau

2022/2023. Par GHAZANFAR Aman.

Bonne lecture.

Table des matières :

1) Présentation du projet Zuul.

Tout d'abord, il est à noter qu'une documentation plus esthétique et interactive est disponible directement sur le GitBook du projet, à l'adresse Web suivante : [Lien].

L'objectif de ce projet est de réalisé une introduction à la programmation orienté objet en Java, au travers d'un jeu vidéo Zuul, issue du chapitre 7 du livre officiel Zuul.

La documentation ci-après est rédigée en français, mais le jeu ainsi que le code sont rédigés en anglais.

Site web du projet : [Lien].

1.1) Auteur

GHAZANFAR Aman, actuellement étudiant en première année dans l'école d'ingénieur ESIEE Paris, je suis développeur full-stack durant mon temps libre.

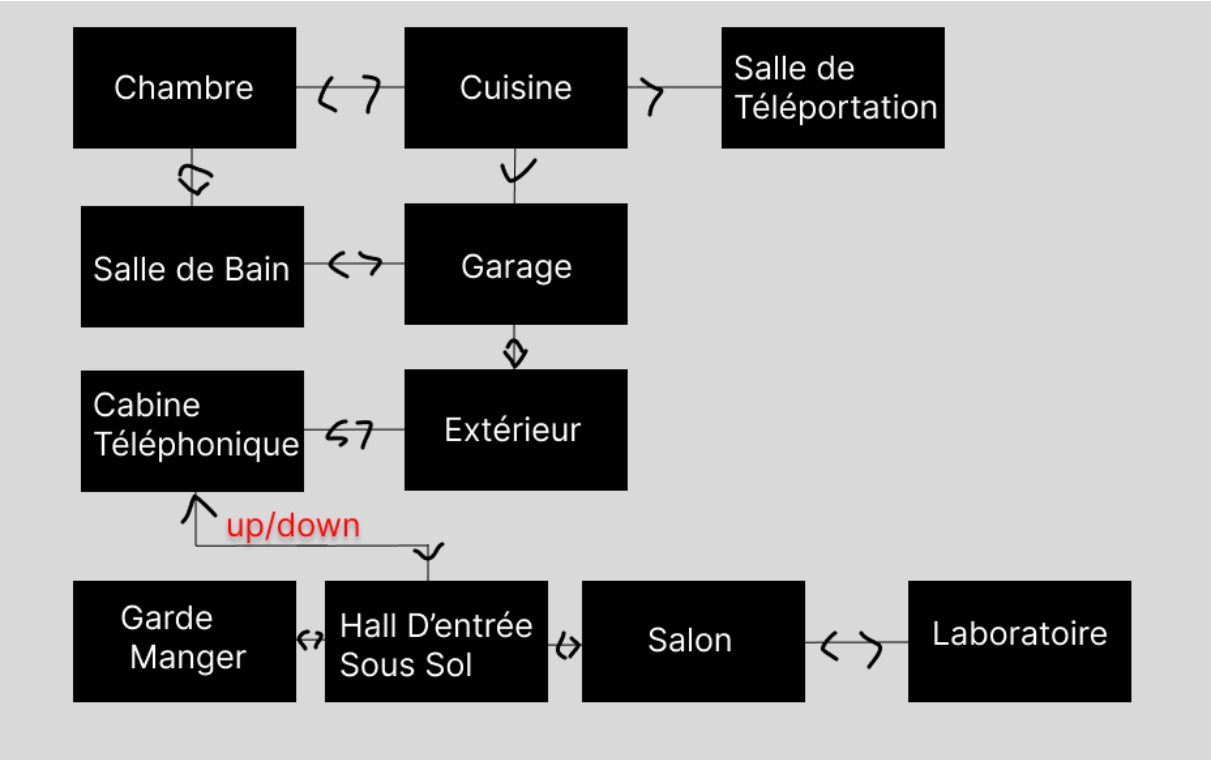
1.2) Thème

Dans sa maison, Edgar va devoir se rendre dans son laboratoire pour développer un programme Hello World !

1.3) Résumé du scénario

Un beau matin, Edgar se réveille avec un message LinkedIn, sa candidature pour développeur freelance a été acceptée, il a enfin la vie dont il rêvait tant, une vie sans pression d'employeur, qui se base sur son propre emploi du temps. Mais tout ne va pas se passer comme il le souhaitait. Il se rend compte qu'il a sa première réunion dans 15 minutes, il s'est réveillé pile à l'heure pour faire sa routine du matin, car oui Edgar a besoin de son café et de sa routine matinale, et il ne la sacrifierait pour rien au monde. Vous allez donc incarner ce personnage, et suivre son aventure, dans sa cuisine, dans sa salle de bain et même dans son sous-sol style bunker postapocalyptique. Le but du jeu ? Parvenir à réaliser votre toute première mission ! Nom de code : Hello World !

1.4) Plan



1.5) Scénario complet et détaillé

Edgar se réveille un beau matin dans sa chambre et se rend compte qu'il est en retard pour son travail. Aujourd'hui est un jour important, il va devoir réaliser sa toute première tâche en freelance en tant que développeur Java.
Mais avant cela, il faut savoir qu'Edgar est un passionné de Café, et donc il ne peut pas commencer sa journée sans son café et sa routine matinale
C'est pourquoi le joueur doit prendre ses chaussons car sinon sa mère n'est pas contente, il ne peut pas sortir de sa chambre tant qu'il n'a pas mis ses chaussons.
Ensuite il part dans la cuisine où il parle à son frère qui lui dévoile le mot de passe de son ordinateur, information précieuse pour Edgar car il doit coder dessus aujourd'hui. Il prend son café puis repart dans sa chambre et descend dans la salle de bain où il doit se brosser les dents et s'appliquer une crème au visage. Une fois tout cela fait il ouvre son tiroir dans la salle de bain pour accéder à la clé permettant d'ouvrir le garage. Une fois en possession de cette clé, il part au garage et inspecte une boîte à outils dans laquelle se trouve une pièce, cette pièce permet d'accéder au téléporteur et récupérer une carte passe, permettant de compléter le jeu a 100%. Mais même sans cet item le jeu peut être finit. Ensuite il faut qu'il se rende à l'extérieur où un chat va lui parler et lui donner le code de la cabine téléphonique. Il se rend dans cette cabine et compose 123, rien ne se passe. Il lit le manuel qui dit que le chat communique de mauvaises informations intentionnellement, et qu'en réalité c'est l'inverse, le code est 321. Une fois ce code rentré il peut accéder à son QG secret au sous sol. Il doit se rendre dans le garde mangé pour prendre un cookie qui lui permettra de soulever la porte se trouvant à droite de l'entrée. Une fois tout cela fait, il se retrouve dans le salon où un mage nommé Saro va lui poser une énigme, la réponse de cette énigme

est montre. Une fois cela fait la porte vers le laboratoire se dévoile, il doit y rentrer le mot de passe donné par son frère et le jeu est terminé. A noter qu'un robot accompagne Edgar pour lui afficher la carte du jeu a tout instant.

1.6) Détail des lieux, items et personnages

Chambre : La chambre d'Edgar, dans cette chambre se trouve des chaussons ainsi que sa mère.

Cuisine : Une machine a café et le frère d'edgar.
Salle de bain : Une brosse à dent, du dentifrice, une crème et une clé.
Garage : Une boîte à outil avec une pièce dedans.
Extérieur : Un chat.
Cabine téléphonique : Un manuel.
Hall d'entrée : Rien de spécial
Garde-Manger : Un cookie.
Salon : Un mage qui pose une question à Edgar
Laboratoire : Un ordinateur, un clavier et une souris.

1.7) Situations gagnantes et perdantes

Situations perdantes :

- Il a excéder la quantité limite de 25 (limité volontairement à si bas pour que le joueur s'y prenne a plusieurs fois et mémorise le chemin correct pour parvenir à la victoire).

Situations gagnantes :

Une seule, parvenir a écrire le mot helloworld dans le laboratoire avec la commande interact.

1.8) Enigmes

- Une seule, l'énigme de Saro : Qu'est ce qui a 2 aiguilles mais ne pique pas, réponse : montre.

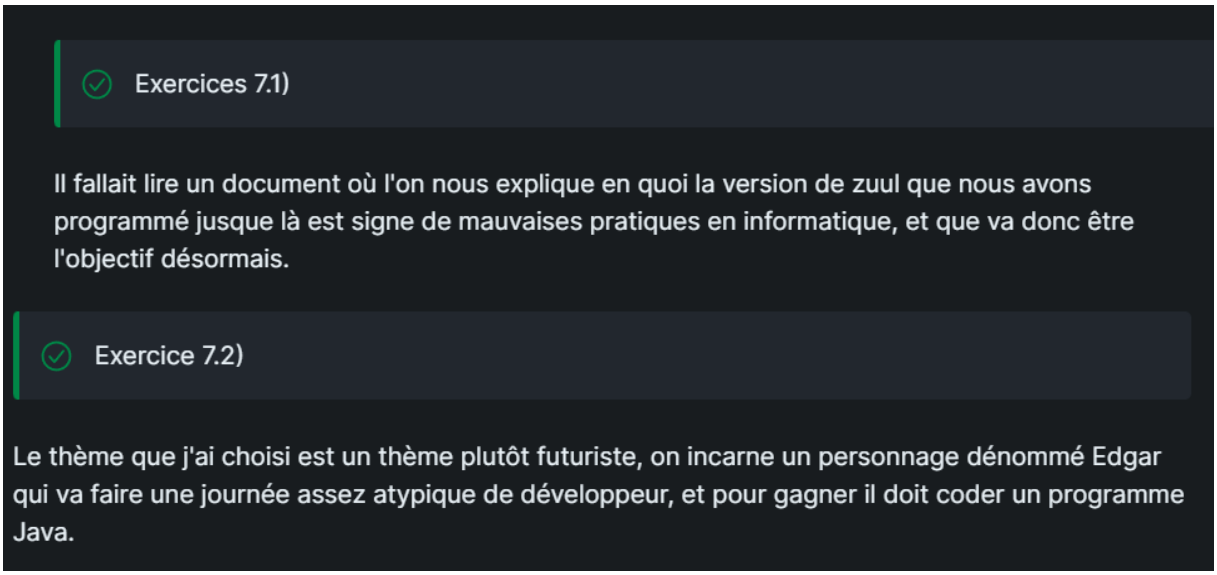
1.9) Commandes de jeu

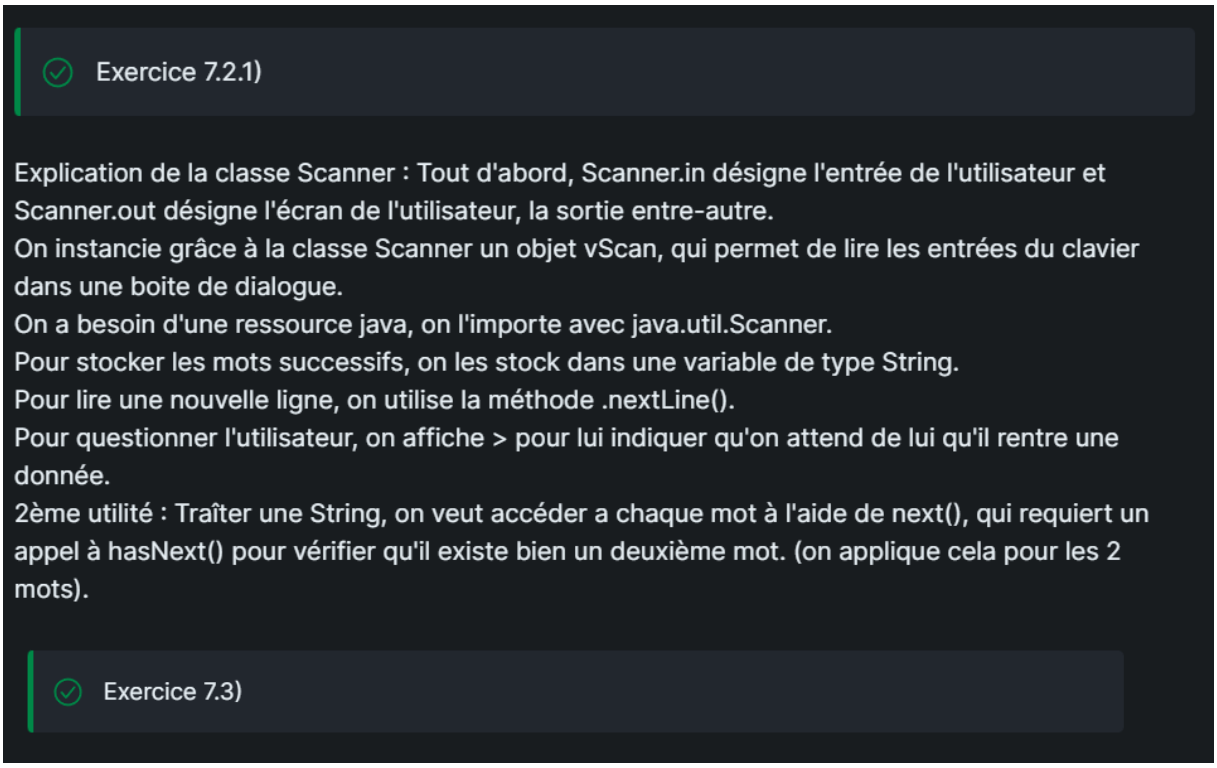
Merci de ne pas inclure les (), {} et [] éventuel dans la commande.
Légende : [] obligatoire, () optionnel et {} nombre indéterminé.
go [direction] : Permet d'aller d'une salle à une autre (se déplacer).
eat [item] : Permet de manger un item.
help : Affiche la page d'aide.
quit : Quitte le jeu.
look {item} : Affiche le lieu où se trouve le joueur ainsi que les sorties empruntables à partir de ce lieu.
take [item] : Prendre un item dans une salle
drop [item] : Jeter un item dans une room
inventory : Regarder l'inventaire du joueur
charge [nom_beamer] : Permet de charger le téléporteur dans une Room.
Fire [nom_beamer] : Permet d'utiliser le téléporteur.
test : Permet de tester le jeu a l'aide d'un fichier texte.
back : Retourner dans la Room précédente.
alea : (ne marche qu'en mode test), Permet de changer la Room vers laquelle la Room de téléportation va vous mener.
interact [argument] : Pour interagir avec Saro ou la cabine téléphonique, il suffit de mettre la réponse attendue en argument.
save [nom_sauvegarde] : Permet de sauvegarder, marche même si un fichier txt n'est pas prévu à l'effet.
load [nom_sauvegarde] : Permet de charger une sauvegarde, merci de ne l'utiliser qu'en début de partie.

1.10) Commentaires / TODO

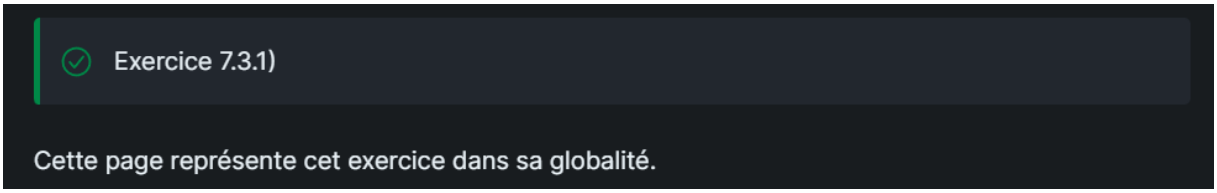
Fin du projet, plus rien à rajouter.

2) Réponse aux exercices





La partie résumée du scénario détaillé illustre cet exercice.



Exercice 7.3.2*)

Exercice 7.4.2)

La partie Plan illustre cet exercice

Exercice 7.3.3*)

Exercice 7.4.3) Exit with ^↵

Je n'ai que 10 salles donc cet exercice ne concerne pas mon projet.

Exercice 7.4)

-bad suffixe enlevé et vérification du projet zuul-bad vérifiée par l'encadrant du TD.
Les salles définie dans le plan du compte rendu ci-présent ont été transférée à la place des salles standards que nous avions définie dans zuul-bad.
La nouvelle structure sans les tests unitaires est construite et empaquetée, voici le rendu actuel (architecture du projet) :

Exercice 7.4.1)

Compte rendu tenu a jour jusqu'à ce point (29/01/2023).

Exercice 7.5)

On doit implémenter une méthode printLocationInfo(). Pour ce faire, on crée cette méthode dans la classe Game, elle va nous permettre de remplacer toute la duplication de code pour afficher les informations actuels de la salle dans laquelle est le joueur. De plus, si on veut changer plus tard les informations que l'on souhaite afficher, ça changera dans tous les endroits où l'on veut afficher les informations de la Room.
Voici la solution proposée :
Tout d'abord enlevé les 2 parties récurrentes, et les remplacer par la nouvelle méthode printLocalInfo(), qui donc affiche la salle actuelle.

```
private void printLocationInfo() {
  System.out.println("You are "+this.aCurrentRoom.getDescription());
        String vPath = "Exits:";
        if(this.aCurrentRoom.aNorthExit != null) {
            vPath+=" north";
        }
        if(this.aCurrentRoom.aEastExit != null) {
            vPath+=" east";
        }²
        if(this.aCurrentRoom.aSouthExit != null) {
            vPath+=" south";
        }
        if(this.aCurrentRoom.aWestExit != null) {
            vPath+=" west";
        }
        System.out.println(vPath);

  }
```

Exercice 7.6)

On constate que la vérification null avant d'affecter l'attribut correspondant dans setExits est innutile car cette vérification est déjà faite dans toutes les méthodes où l'on a recouvert aux exits d'une salle.
La méthode suivante à été crée :

```
public Room getExit(String pDirection) {

          if(pDirection.equals("north") ){
              return this.aNorthExit;
          }
          if(pDirection.equals("south")){
        return this.aSouthExit;
          }
          if(pDirection.equals( "west")) {
              return this.aWestExit;
          }
          if(pDirection.equals("east")) {
              return this.aEastExit;
          }
          return null;

    }
```

Méthode permettant de get les sorties d'une salle selon la direction donnée.
Pour régler le problème de Unknown Direction, j'ai opté pour une solution qui vise à renvoyer une UNKNOWN_ROOM dès lors que la direction n'est pas reconnue, donc j'ai crée une Room UNKNOWN_ROOM comme attribut publique de la classe Game. J'ai aussi crée cette méthode privée :

```
private boolean verifDirection(String pDirection) {
       String[] vDirections = {
           "north",
           "south",
           "west",
           "east"
       };
       for (String direction: vDirections) {
           if (direction.equals(pDirection)) {
               return true;
           }
       }
       return false;
   }
```

Cette méthode me permet de vérifier si le mot donné en argument de la commande go est bien une coordonnée que mon jeu supporte. Donc ensuite, à l'aide de cette méthode, je fais la vérification, de si je dois renvoyer UNKNOWN_ROOM ou alors l'exit de la Room demandée :

```
public Room getExit(String pDirection) {
        if (this.verifDirection(pDirection)) {
            return UNKNOWN_ROOM;
        } else {
            return this.exits.get(pDirection);
        }
    }
```

Si l'exit n'existe pas, this.exits.get(pDirection) vaudra null, sinon elle vaudra l'exit de la currentRoom. Sinon, on renvoie UNKNOWN_ROOM, et un traitement est fait par la suite dans la classe Game, avec le code suivant :

```
if(vNextRoom == Room.UNKNOWN_ROOM) {
            System.out.println("Unknown direction");
            return;
        }
```

Ici, on comprend la nécessité de rendre publique l'attribut UNKNOWN_ROOM de la classe Room, j'aurais pu définir un accesseur aussi.

Exercice 7.7)

Création de la méthode `getExitString()` dans Room: retourne une chaine de caractères contenant toutes les sorties disponibles de la salle sur laquelle elle a été appelée. Cette méthode nous permet d'éviter de répéter du code dans `printWelcome()` et `printLocationInfo()`.
Le code nous donne cela :

```
public String getExitString() {
          String vPath = "Exits:";
        if(this.aNorthExit != null) {
            vPath+=" north";
        }
        if(this.aEastExit != null) {
            vPath+=" east";
        }
        if(this.aSouthExit != null) {
            vPath+=" south";
        }
        if(this.aWestExit != null) {
            vPath+=" west";
        }
        return vPath;
    }
```

Exercice 7.8)

La méthode setExits ne sert plus a rien, elle a été remplacée par setExit.
Mais avant, on doit changer la façon avec laquelle on définit nos sorties, on stock toutes nos exits dans un HashMap désormais. Et on utilise la méthode setExit() pour définir nos sorties, le code :

```
public void setExit(final String direction, final Room neighbor) {
        this.exits.put(direction, neighbor);
    }
```

Exercice 7.8.1)

J'ai rajouté dans le scénario une salle (le hall d'entrée), qui pour y accéder on doit aller down, donc dans le sous sol, et on peut aussi remonter a la surface (up). Les directions sont directement ajoutés dans les directions acceptées par le projet, grâce au code suivant :

```
private boolean verifDirection(String pDirection) {
        String[] vDirections = {
            "north",
            "south",
            "west",
            "east",
            "up",
            "down"
        };
        for (String direction: vDirections) {
            if (direction.equals(pDirection)) {
                return true;
            }
        }
        return false;
    }
```

Exercice 7.9)

Il faut mettre la méthode getExitString à jour.

```
public String getExitString() {
        String vPath = "Exits: ";
        Set < String > keys = exits.keySet();
        for (String exit: keys) {
            vPath += exit + " ";

        }
        return vPath
    }
```

Exercice 7.10)

La méthode getExitString() sert a afficher toutes les sorties possible pour la room où la méthode est appelée. On déclare un String dans lequel on va stocker toutes nos sorties, on lui donne un texte de base, "Exits: ", après cet espace, on veut ajouter toutes les sorties possible depuis la salle dans laquelle on est, les uns a la suite des autres, séparés chacun par un espace.
On définit un Set, qui agit comme un dictionnaire, à une clé est associée une valeur. Donc dans notre cas, un point cardinal est associé à une Room. La méthode keySet() nous permet de get l'ensemble de ces clés, et de les stocker dans un objet de type Set. On itère ensuite sur ce Set avec une boucle foreach, et on ajoute successivement les exits trouvées dans vPath. On retourne enfin vPath.

Exercice 7.10.1) et Exercice 7.10.2)

JavaDoc générée et completée.

Exercice 7.11)

La méthode getLongDescription() permet de faire une synthèse entre la description d'une salle, et les sorties qui sont disponibles pour sortir de cette salle. Le code :

```
public String getLongDescription() {
        return "You are " + this.aDescription + ".\n" + this.getExitString();
    }
```

On fait cela car, comme il nous est expliqué dans le livre, il faut laisser a chaque classe traité ses propres attributs, encapsulé tout ça. On l'affiche ensuite en faisant un System.out.println(une_room.getDescription())

Exercice 7.12) Exercice 7.13) // a faire plus tard

Exercice 7.14)

Il faut rajouter une nouvelle commande de jeu, look, pour cela on rédige la méthode, puis sa correspondance dans la classe Game.

```
private void look() {
      System.out.println(this.aCurrentRoom.getLongDescription());
  }

//
```

Exercice 7.15)

Création de la commande eat, permet au joueur de mangé. J'ai décidé de faire en sorte que le joueur ne puisse manger que 3 ingrédients pour l'instant, avec un tableau et une vérification rapide :

```java
private void eat(final Command pCmd) {
      String[] vIngredients = {"apple", "banana", "tomato"};
      boolean verif = false;

      if(!pCmd.hasSecondWord()) {
          System.out.println("Eat what ?");
      } else {

      StringBuilder parse = new StringBuilder();
       for(String ingredient: vIngredients) {
          if(ingredient.equals(pCmd.getSecondWord())) {
              verif = true;

            }
            parse.append(" "+ingredient);

        }
      if(verif) {
           System.out.println("You ate a "+pCmd.getSecondWord()+" you are not hungry an
      }else {
            System.out.println("Wrong ingredient, you can only eat : "+parse.toString()
          }

      }

    }
```

Exercice 7.16)

J'ai implémenter les changements décrits dans le livre pour intégrer une méthode permettant d'afficher toutes les commandes.

Exercice 7.17)

Si une nouvelle commande vient à être ajoutée, on devra renseigner sa correspondance dans la classe Game, c'est-à-dire, écrire sa méthode, et dire qu'il faut l'executer lorsque le mot de commande est détecté par le Parser.

Exercice 7.18)

Changement de showAll() vers getCommandList(), fait appel à un changement dans 3 classes en réalité,
1er changement, dans CommandWords :

```
public String getCommandList() {
      StringBuilder vCommands = new StringBuilder();
      for(String command: this.aValidCommands) {
          vCommands.append(command+ " ");

      }
      return vCommands.toString();
```

2ème changement dans Game :

```
private void printHelp() {
     System.out.println("How could you be lost inside/near your own house..., anyway, I
     System.out.println(this.aParser.showCommands());

 }
```

3ème changement dans Parser :

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

Exercice 7.18.1)

Comparaison faite, rien a changer.

Exercice 7.18.2)

Implémentation de StringBuilder dans tous les endroits où il y avait une concaténation de String successifs sur une boucle.

Exercice 7.18.3)

Les images trouvées, pour l'instant, sont repertoriées ici :

[Lien].

Exercice 7.18.4)

Changement du message de bienvenue.
Le titre du jeu serait : Edgar's Journey.

Exercice 7.18.5)

On veut pouvoir avoir accès aux rooms dans d'autres classes, pour ce faire, on

```
// dans la partie initialisation des attributs de la classe Game
 private HashMap<String, Room> aRooms;

 // dans le constructeur de la classe Game
 this.aRooms = new HashMap<String, Room>();

// dans la méthode createRooms()
 aRooms.put("bedroom", vBedroom);
 aRooms.put("kitchen", vKitchen);
 aRooms.put("garage", vGarage);
 aRooms.put("cabin", vCabin);
 aRooms.put("outside", vOutside);
 aRooms.put("entry", vEntry);
 aRooms.put("pantry", vPantry);
 aRooms.put("living", vLiving);
 aRooms.put("laboratory", vLaboratory);
```

Exercice 7.18.6)

On nous a donné le projet zuul-with-images, après étude, on constate que la nouvelle classe UserInterface contient toutes les méthodes permettant d'introduire la nouvelle interface graphique, par exemple la création de boutton avec JButton, de frame avec JFrame etc ...
Les méthodes de la classe Game sont transférées dans une nouvelle classe GameEngine, a l'exception de play() car on l'a supprimé au profit d'une interface graphique. On change processCommand() en interpretCommand(), et la façon pour quitter le jeu a changer.
Rien ne change dans Parser, ni dans Command, ni dans CommandWords.
Game lance le jeu désormais. Parser lit désormais les textes de l'interface graphique

Exercice 7.18.7)

La fonction addActionListener() permet l'appel de actionPerformed() (les deux fonctions sont complémentaires). Le paramètre de addActionListener(), spécifie à actionPerformed sur quel objet il faut écouter les évènements (cliques, touches etc...). De plus, la classe UserInterface implémente ActionListener() et possède par l'occurence, actionPerformed()

Exercice 7.18.8)

J'ai rajouté qu'un seul bouton, mais dans un gridLayout() 3×3 pour ensuite rajouter les directions. Le bouton permet juste de look() dans une room. J'ai crée un panel en mode gridLayout() et je l'ai positionner à l'est de vPanel.hin

Exercice 7.20)

Création de la classe Item, qui possède pour attribut le nom de l'item, sa description et son poids. Cette classe permet la gestion d'un Item quelconque, qui est forcement présent dans une salle, une room peut contenir plusieurs Items.

```
public class Item {
    private String aName;
    private String aDescription;
    private double aWeight;
    public Item(final String pName, final String pDescription, final double pWeight) {
        this.aName = pName;
        this.aDescription = pDescription;
        this.aWeight = pWeight;
    }
      public String getName(){
        return this.aName;
    }

    public String getDescription(){
        return this.aDescription;
    }

    public double getWeight(){
        return this.aWeight;
    }

}
```

On aurait pu rajouter les setters aussi ^

Exercice 7.21)

Aucune modification spécifique.
Réponse questions :
"quelle classe devrait crée ?" : La classe GameEngine dans la méthode createRooms()
"quelle classe devrait afficher ?" : C'est GameEngine qui devrait afficher ces informations car selon notre méthode d'encapsulation du code, il y est dit que c'est la classe qui se charge de modeler les objets qui doit mettre en place des méthodes get/set/help et qui doit permettre a la classe principale d'utiliser l'engine pour afficher les informations à l'aide d'Item et de ses méthodes get

Exercice 7.22)

Les items d'une room sont désormais des HashMap<String, Item>, comme cela, on peut avoir plusieurs items dans une room. On a dès lors la nécessité de 3 nouvelles fonctions :
addItem(String, Item) → ajoute un item dans une salle.
getItem(String) → retourne l'item dont le nom a été passer en paramètre (permet aussi de vérifier si l'item est dans une room ou non)
removeItem(String) → retirer un item d'une salle (utile si jamais on veut récupérer l'item en l'occurence).

Exercice 7.22.1)

On a choisit le Collection HashMap pour son système de clé valeur, un item porte un nom (unique) qui est associer a son objet. On veut pouvoir get un item, le supprimer ainsi que de le changer, et les hashmap permettent de faire tout cela. Un Item nécessite toutes ces fonctionnalitées.

Exercice 7.22.2)

J'ai intégré les items a l'aide de la méthode addItem() dans createRooms() sur chacune des rooms qui nécessitent un ou plusieurs items. Exemple d'items rajouté : café, dentifrice, ordinateur, cookie.

Exercice 7.23)

Ajout de la commande back qui permet de revenir dans la room précedente (à l'aide d'une variable qui change a chaque fois).

Exercice 7.24)

Quand on utilise back dans la première salle, cela ne marche pas, mais cela pourrait être pallier à l'aide d'une condition.

Exercice 7.25)

Quand on utilise back plusieurs fois dans une même salle, on finit par tourner en boucle sans retourner au point de départ. Donc ce n'est pas satisfaisant (nécessité de pouvoir toutes les stockées)

Exercice 7.26 et après 7.26)

On rajoute donc ce changement, à l'aide d'une Stack (qui admet le principe de first in first out FIFO) et qui va nous permettre de dépiler les salles déjà explorées jusqu'à atteindre le point de départ. Une stack comporte des méthodes bien utiles dans notre exemple :
- à l'aide du peek() je get la dernière salle visitée sans pour autant la supprimée.
- à l'aide du pop() je get la dernière salle visitée en la supprimant.
- à l'aide du empty() je vérifie si la stack est vide ou non (la condition d'arrêt et de message d'erreur de la commande back).
- à l'aide du add() j'ajoute la room dans le goRoom()

Maintenant, on peut avancer de plusieurs rooms, et utiliser la commande back pour revenir sur ses pas, jusqu'à atteindre le point de départ (la chambre d'edgar) du jeu.

Exercice 7.26.1)

Les deux javadoc côté utilisateur et côté développeur sont générées.

Exercice 7.28)

On va s'aider de la classe Scanner pour lire les lignes d'un fichier .txt et executer les commandes les unes après les autres. Tout cela dans une commande test, qui va lire 3 fichiers :
1) court fichier pour essayer la commande de test

2) parcours idéal pour gagner (*suite minimale de commandes pour aller le plus loin possible dans le jeu*)

3) exploration de toutes les possibilités du jeu (*y compris exploration de lieux inutiles, commandes affichant des informations, ...*)

Exercice 7.29)

On crée la classe Player. Un Player a un nom (que l'on prend dès le début du jeu a l'aide de javax.swing.JObject.showPrompt(). Un poids maximal fixé à 50. Un poids actuel qui est fixé à 0. Un inventaire (pas réglé pour l'instant, sujet d'un prochain exercice, pour l'instant il s'appelle aCurrentItem vu qu'on peut en prendre qu'un). C'est maintenant cette classe qui va s'occuper de générer les Strings pour les commandes (donc j'ai déplacer les méthodes back, eat etc...) dans Player, et aussi, on crée une méthode getCurrentRoom() qui va désormais remplacer le this.aCurrentRoom de la classe GameEngine, qui n'avait pas l'utilité ni la logique de l'utilisé, il est bien plus cohérent de laisser a Player, tout ce qui qui concerne le Player.

Exercice 7.30)

Création de deux nouvelles commandes (donc ajout de leurs noms dans aCommandWords, ainsi que la correspondance dans GameEngine dans le switch). Ces commandes s'appellent take et drop.
- take permet de prendre un item quelconque dans une room.
- et drop permet de faire l'inverse, donc de lacher un item dans une room
(pour l'instant, vu qu'un item n'a pas d'id spécifique ni d'appartenance particulière distinguable a une room, on peut take un item d'une salle et le drop dans une autre).
On doit donc crée les getter et setter nécessaire dans Player, pour pouvoir get l'item actuel et le changer si besoin. Ajout aussi des méthodes take() et back() qui change l'item actuel en fonction de la demande.
Ainsi, on crée aussi une classe Item qui gère tout ce qui est relatif a un Item, on rajoute la méthode addItem() dans Room pour pouvoir rajouter un item directement dans GameEngine après déclaration des Rooms.

Exercice 7.31.1)

On remarque que les attributs items et inventory correspondent tous les deux à une `HashMap<String,Item>` On va donc crée une classe ItemList qui va permettre de regrouper les getter/setter et surtout pouvoir instancier l'inventaire nouveau du joueur, qui cette fois pourra contenir plusieurs items. Un seul attribut dans cette nouvelle classe, une HashMap<String, Item> et regroupant les méthodes de room et player concernant les items.

Exercice 7.32)

Déjà réalisé, a l'aide de l'attribut aMaxWeight présent dans la classe Player. La restriction quant à elle est fixée de 50 en terme de poids. Elle a lieu dans la méthode take de Player (qui renvoie une String car toutes les commandes relatives au Player ont été transférées dans Player :)

```
if(this.aCurrentWeight+vItem.getWeight() > this.aMaxWeight)
return "You can't carry this item";
```

Exercice 7.33)

Ajout de la méthode getInventoryString() qui permet d'afficher les items présent dans l'inventaire du joueur, en respectant la forme suivante : [nom_item](prix). Elle appelle la méthode getItemsString() de ItemList. On crée ensuite la correspondance de la commande dans GameEngine, et on ajoute le mot dans aCommandWords dans la classe CommandWords.

Exercice 7.34)

Changement de la commande eat, qui initialement permettait de manger un ingrédient quelconque, maintenant on lui passe un paramètre de type Command. Le joueur peut maintenant manger un cookie, si et seulement si ce cookie est présent dans son inventaire. On crée une liste des ingrédients mangeable par le joueur (pour une potentiel extension sur plusieurs mangeables).
On vérifie d'abord si le paramètre de eat est bien dans vIngredients (variable qui stock les mangeables). Ensuite, on vérifie si un second paramètre est passer ou non. Puis on vérifie si l'item est bien dans l'inventaire du joueur. Et dans le cas où tout les tests sont passés, on ajoute 10 au aMaxWeight.

Exercices 7.35 à 7.41)

Ces exercices étant optionnels, voici le rapport de ce qui a été changer dans le code :
- On a maintenant une liste d'énumérations (une nouvelle classe CommandWord), qui garde toutes les commandes afin de pouvoir ensuite faire en sorte que le jeu puisse avoir d'autres langues.
- On a un nouveau système de switch / case / default dans interpretCommand.
- CommandWords change donc de structure, avec maintenant une CommandWord qui est traîtée au lieu d'une String.
- On peut maintenant accéder aux commandes de façon globale, en faisant Command.HELP par exemple pour accéder à la String correspondant au nom de la commande.

Exercice 7.42)

Ajout du fait que le joueur a désormais 25 déplacements possibles dans le jeu, à l'issue desquels si il n'a pas développé le programme, il perd le jeu. Pour cela on rajoute un attribut qui va compter les déplacements, et a chaque déplacement qui n'a pas généré d'erreurs dans goRoom(), on incrémente de 1. Juste ensuite, on fait un test pour vérifier si le max est atteint, et si oui, alors on System.exit(0).

Exercice 7.42.1)

Ajout d'un temps réel à l'aide de la classe Timer() issue de javax.swing. On instancie un attribue global qui sera notre unique timer dans le jeu, qui va, a chaque changement de Room, va lancer un Timer de 2 minutes, et qui à l'issue de ce Timer va demander au joueur si il est a.f.k (away from keyboard). La classe Timer prend 2 paramètres, le temps en millisecondes ainsi qu'un ActionListener(), munie d'un actionPerformed, où l'on va traité l'action qui a été réalisée, et on affiche un dialogue pour le joueur.

Exercice 7.42.2)

Pour améliorer l'IHM graphique, pour l'instant, j'ai rajouté des boutons de sorte a faire une croix directionnelle, qui permet d'aller au nord, sud, est et ouest (up et down a rajouté). Aussi, 4 autres boutons qui permettent de manger, regarder les informations de la room, regarder l'inventaire et enfin un bouton pour quitter le jeu.

Exercice 7.43)

Ajout d'une trap door entre la cuisine et le garage, on ne peut que passer de la cuisine vers le garage et non du garage vers la cuisine. Pour ce faire, on supprime la porte entre le garage et la cuisine, mais on laisse celle menant de la cuisine vers le garage. Cela pose un problème pour la commande back qui dans ce cas, rentre en contradiction avec le système d'une trap door. Pour pallier à ce problème, on rajoute une méthode isExit() dans la classe Room. De ce fait, on peut maintenant vérifier dans la commande back, pour que avant de back dans une room, on vérifie si cette room est une sortie bien définie de la room dans laquelle la commande back a été lancée.

Exercice 7.44)

On veut crée un item qui puisse être chargé dans une Room, puis utiliser dans une autre Room afin de se téléporter dans la Room chargée précédement. On constate que le Beamer, est un sorte d'Item, mais qu'il possède des spécificitées qui lui sont propres. On crée donc la classe Beamer, qui extend la classe Item. Et on lui donne une méthode charge, isCharged et fire. Ensuite, dans Player, on crée les nouvelles commandes fire et charge, et dans GameEngine on crée un Item de type Beamer, et on le pose dans une Room. A noter qu'on peut avoir plusieurs Beamer avec cette configuration. La commande fire, test si le beamer est chargé, si oui, alors il change la currentRoom sur la Room qui était chargé, et change l'état du Beamer ainsi que la Room qui était mémorisée sur null. charge fait juste l'inverse.

Exercice 7.45)

On souhaite crée un système de clé et de portes verrouillé. Pour ce faire, on crée une clé qui est un item comme un autre. Et une classe Door qui désormais permettra de gérer les exits d'une Room. Cette classe contient la méthode isLocked, getKey et lock() et prend en paramètre 1 argument dans son constructeur naturel, l'item clé qui est nécessaire pour pouvoir l'ouvrir. Ensuite, on change dans GameEngine, en mettant maintenant l'exit qui est lock, ainsi que l'item clé qui est nécessaire pour ouvrir cette porte. Dans goRoom maintenant on vérifie si le joueur a bien la clé dans son inventaire, et on vérifie la Door considérée est une lockedDoor ou non.

Exercice 7.46)

On doit crée une nouvelle Room, qui est une TransporterRoom. Cette Room doit pouvoir être accessible depuis une autre Room, et lorsqu'on sort de cette dernière par n'importe quelle de ses sorties j'ai choisi la sortie Est comme étant la seule sortie empruntable, mais menant vers UNKNOWN_ROOM). En réalité, cette sortie UNKNOWN_ROOM, est une Room au hasard parmi les Room qui évite au joueur d'avancer trop dans le jeu, ni d'arriver a un point de non retour. On a besoin de la classe RoomRandomizer, qui ne contient qu'une seule méthode generateRandomRoom, qui prend une collection de Rooms en paramètre, et qui retourne la room au hasard. Pour ce faire, on utilise la classe Random() issue de java.util.
La TransporterRoom est une sorte de Room, donc elle a sa propre classe. Elle prend autant de paramètres que le constructeur naturel Room, mais une HashMap de Rooms (donc celles qui sont visitables par Edgar dès le début du jeu) lui sont passées en plus d'une Room classique.
On doit Override getExit dans un premier temps, car cela nous permettra d'éviter de devoir changer le goRoom, ce nouveau getExit retourne une Room au hasard en utilisant la méthode findRandomRoom, qui s'occupe a son tour d'appeler generateRandomRoom de la classe RoomRandomizer.
Désormais, quand on rentre dans cette Room, sa description lambda nous est donnée, mais quand on quitte cette Room (initialement instanciée dans GameEngine, puis correctement définie comme toute autre Room, mais avec le type TransporterRoom), on se retrouve dans une Room au hasard.

Exercice 7.46.1)

L'objectif de cet exercice est de pouvoir tester notre jeu si l'on passe dans une TransporterRoom. Un problème devient évident, on ne peut pas prévoir dû au caractère aléatoire de notre nouvelle implémentation, comment va se comporter cette TransporterRoom, on doit donc pouvoir manipuler la RNG (random number generation). Pour ce faire, on crée une commande alea, qui permet de réglé la Room vers laquelle on se retrouvera téléporter à l'issue de la sortie de la TransporterRoom par le joueur. Ainsi, on pourra tester et voir si on est bien téléporter la où l'on veut dans les fichiers test. Cela implique l'ajout d'un nouvel attribut dans la classe TransporterRoom, une String. En effet, si cette String est vide, alors on laisse nextInt() de Random prendre le relai, en revanche, si elle a un nom d'une Room en paramètre, alors c'est un nouveau morceau de code qui prend le relai. Cela implique l'ajout d'une méthode setAlea() dans TransporterRoom, et l'ajout d'un mode test surtout. En effet, on ne veut pas que le joueur

Exercice 7.46.2) JavaDoc côté utilisateur et développeur à jour.

Exercice 7.47.1)

L'objectif de cet exercice est de paqueter les classes, c'est a dire de les regroupées dans différent paquets. Chaque paquet représente une fonction qu'assure chaque classe en commun. J'ai décidé de les regroupées en 5 blocs : pkg_commands, pkg_items, pkg_rooms, pkg_gameobjects, pkg_utils. Pour crée un package, il suffit de mettre la ligne package nompackage; pour ranger une classe dans un package, et ensuite pour utiliser cette classe, si on est dans un autre package, il faut faire import pkg_name.ClassName, si on est dans le même paquetage, alors on y a directement accès.

Exercice 7.47)

L'objectif de cet exercice est d'enlever la succession de switch dans la méthode interpretCommande de la classe GameEngine. Pour ce faire, on va séparer chaque commande dans son propre fichier, en rendant la classe Command, commune a toutes ces commandes abstraite désormais, avec une méthode execute, prenant en paramètre un object Player, qui devra être re écrite dans chaque fichier de commande. Cette méthode execute contiendra la fonction qui assure la commande.
CommandWord qui tient l'ensemble de nos ENUMS, va changer en stockant un plus du nom de la commande, sa correspondance en classe, on stock directement le pointeur menant vers la classe, permettant ensuite d'executer la méthode execute.
Suppression de toutes les méthode de la classe GameEngine et Player qui concernaient des commandes.
Changement de la méthode interpretCommand, on prend maintenant la commande directement du Parser, qui nous renvoie une commande après avoir setFirstWord() et setSecondWord(), on peut ainsi la traitée.

Exercice 7.48)

L'objectif de cet exercice est de crée un PNJ dans le jeu. On crée donc la classe Character, on lui donne 3 attributs : name, description et dialog. On règle les getters ainsi que les setters de cette classe. Ensuite, on va dans la classe Room, et on crée une méthode addCharacter, ce qui implique l'utilisation d'une nouvelle HashMap<String, Character>, qui associe le nom du PNJ à son object correspondant contenant son nom, description et son unique dialogue. La méthode addCharacter ajoute le PNJ dans la hashmap, et la méthode removeCharacter fait l'inverse. Une méthode getCharactersString() est aussi nécessaire pour changer le printLocationInfo et informer le joueur des PNJ présent dans la pièce. Enfin, il faut crée l'object Character et l'ajouter dans chaque rooms où il est nécessaire d'avoir un PNJ.
Ajout donc d'une commande talk pour pouvoir interagir avec un PNJ, prenant un argument qui est le nom du PNJ. Le PNJ ne fait que afficher son dialogue tout simplement.

Exercice 7.49)

L'objectif de cet exercice est de crée un PNJ qui changera de pièce sous l'action du joueur. J'ai décider de crée un robot du nom de Recupix, qui suivra le joueur lorsqu'il change de Room, et qui lui affichera la carte du jeu lorsqu'il intéragira avec.
On crée une classe MovingCharacter, qui est une sorte de Character, donc qui hérite de toutes les propriétés, classes etc.. de la classe mère Character. On y rajoute seulement une méthode move, donc un attribut aCurrentRoom, et on fait move le robot lorsque le joueur move, en changeant la commande goCommand.

Exercice 7.50)

L'objectif de cet exercice est d'étudier la classe Math issue du paquetage java.lang
On constate dans un premier temps que toutes les méthodes de cette classe sont strict, donc elles n'ont pas besoin d'être instanciées, on peut par exemple appeler la méthode max, directement avec Math.max(2,4)
En parlant de cette méthode max(), sa signature est multiple : on remarque qu'il y en a une pour chaque types : double, float, int et long.

`static double`	**`max`**`(double a, double b)` Returns the greater of two `double` values.
`static float`	**`max`**`(float a, float b)` Returns the greater of two `float` values.
`static int`	**`max`**`(int a, int b)` Returns the greater of two `int` values.
`static long`	**`max`**`(long a, long b)` Returns the greater of two `long` values.

documentation issue du site docs.oracle.com

Exercice 7.51)

Ces méthodes (celle de la classe Math) sont toutes statiques car il n'y a pas lieu a instancier une classe qui poserait d'emblée des attributs globaux nécessaires au bon fonctionnement de la classe. En effet, a chaque méthode est associée sa signature qui prend déjà les informations nécessaires au bon fonctionnement de la méthode. Cette classe a aussi pour vocation d'être utilisée sur des échantillons différents, donc devoir l'instancier serait absurde, autant pouvoir l'appeler directement, par exemple Math.max(1,2), Math.min(3,4)
Il est possible de les écrire directement comme forme de méthodes d'instance en enlevant les static dans les signatures des fonctions.

Exercice 7.52)

L'objectif de cet exercice est de trouver le temps qu'il faudrait a java pour compter de 1 à 100, voici le programme que je propose :

```
long vStartTime = System.currentTimeMillis();
for(int i=1;i<=100;i++) {}
System.out.println("Count 1 to 100 in "+ ""+(System.currentTimeMillis()
- vStartTime)+"ms");
```

Exercice 7.53)

L'objectif de cet exercice est de permettre a l'utilisateur de lancer le jeu sans utiliser BlueJ.
On doit donc crée une classe Main dans la racine du projet, et mettre ce code dedans :

```
public class Main {
    public static void main(String[] args) {
        new Game();
    }
}
```

Exercice 7.56)

Pouvez-vous appeler une méthode statique à partir d'une méthode d'instance ?
Oui, prenons la méthode statique max(final double a, final double b) qui est issue du paquetage Math, on peut l'appeler où l'on veut.

Pouvez-vous appeler une méthode d'instance à partir d'une méthode statique ?
Non, car le mot clé this n'existe pas dans une méthode statique.

Pouvez-vous appeler une méthode statique à partir d'une méthode statique ?
Oui, il y a par exemple la suite de fibonacci qui est une fonction définie par récursion, qui s'appelle elle même a chaque tour.
A noter : Idem pour méthode d'instance dans une méthode d'instance.

Exercice 7.57)

L'objectif de cet exercice est de savoir si il est possible de compter le nombre d'instances qui ont été crée a partir d'une classe.
Il est en effet possible de réaliser cela en déclarant un attribut statique à l'intérieur, qui s'incrémente directement dans le constructeur, exemple :

```
 // Code VTC
public class VTC {
    private static int cpt;
    private String aName;
    public VTC(final String pName) {
        this.aName = pName;
        cpt++;
    }
    public int getVTCCount() {
        return cpt;
}
}
// Code Main

public class Main {
    public static void main(String[] args) {
        VTC v1 = new VTC("abc");
        System.out.println(v1.getVTCCount()); // renvoie 1
        VTC v2 = new VTC("def");
        System.out.println(v2.getVTCCount()); // renvoie 2
    }
}
```

Exercice 7.61)

L'objectif de cet exercice est de permettre au joueur de sauvegarder l'état de son jeu, j'ai opter pour la méthode 1.
Cette méthode consiste à avoir une ArrayList<String> des commandes executées, puis de savegarder cette ArrayList<String> en les séparant par des virgules. Exemple :
go=west, eat=null, etc...
Cette méthode pose néanmoins des problèmes car pour l'exercice prochain où il faut permettre de charger cette sauvegarde sous format .txt par ailleurs, lorsque l'utilisateur rentre dans la transporter Room, on ne peut pas prédire où il va, à part si il utilise la commande alea, sauf que la commande alea n'est utilisable qu'en mode test, donc on ne peut pas se permettre de changer l'état du boolean aTestMode juste pour recharger une sauvegarde faute de crée des failles de sécurité dans le jeu.
Donc pour pallier à ce soucis, lorsque l'utilisateur go east dans TransporterRoom, alors ça remplace east par le nom de la Room. On verra ensuite en quoi cela est utile.

Exercice 7.62)

Dans cet exercice, on se propose de permettre à l'utilisateur de charger une sauvegarde précédement crée. On crée une commande quasiment semblable à la commande test, mais pour le cas cité ci-dessus de la TransporterRoom, on regarde si le deuxième argument est une direction, si oui, alors on fait le comportement basique de go, sinon, on utilise aAllRooms de GameEngine pour tout simplement téléporter l'utilisateur a la String de la Room posée en second paramètre.

3) Déclaration plagiat

Les références vers les images choisies pour le jeu sont présentes ici : https://app.gitbook.com/s/M6PNII78QGETUTHCvZyE/group-1/les-images
Certains bouts de codes proviennent de la documentation java, ainsi que de sites tels que StackOverflow, GeeksForGeeks et OpenClassRooms.
Pour l'utilisation de la carte, le code d'une partie de StackOverflow : https://stackoverflow.com/questions/14353302/displaying-image-in-java
Et l'autre partie de la documentation de swing
Le nom Recupix est tiré du jeu Zelda Skyward Sword
Pour l'utilisation de JOptionPane, cela provient d'une discussion entre Monsieur Bureau et un élève sous un exercice du forum.
Pour les paquetages, je me suis en partie inspirée de la forme choisie par cette repository github : https://github.com/francois07/zuul-bad