Rapport Projet Zuul

The Search For Satoshi



FELTRIN Kévin – Groupe 5

IPO 2022-2023

FELTRIN Kévin Groupe 5

Sommaire du rapport :

I.A) Auteur(s)
I.B) Thème (phrase-thème validée)
I.C) Résumé du scénario (complet)
I.D) Plan complet
I.E) Scénario détaillé (complet)
I.F) Détail des lieux, items, personnages
I.G) Situations gagnantes et perdantes
I.H) Eventuellement énigmes, mini-jeux, combats, etc.
I.I) Commentaires (ce qui manque, reste à faire, ...)
II. Réponses aux exercices **(à partir de l'exercice 7.5 inclus)**
III. Mode d'emploi (si nécessaire, instructions d'installation ou pour démarrer le jeu)
IV. Déclaration obligatoire anti-plagiat **(*)**

FELTRIN Kévin Groupe 5

I. Mon jeu

A) Auteur

Kévin FELTRIN

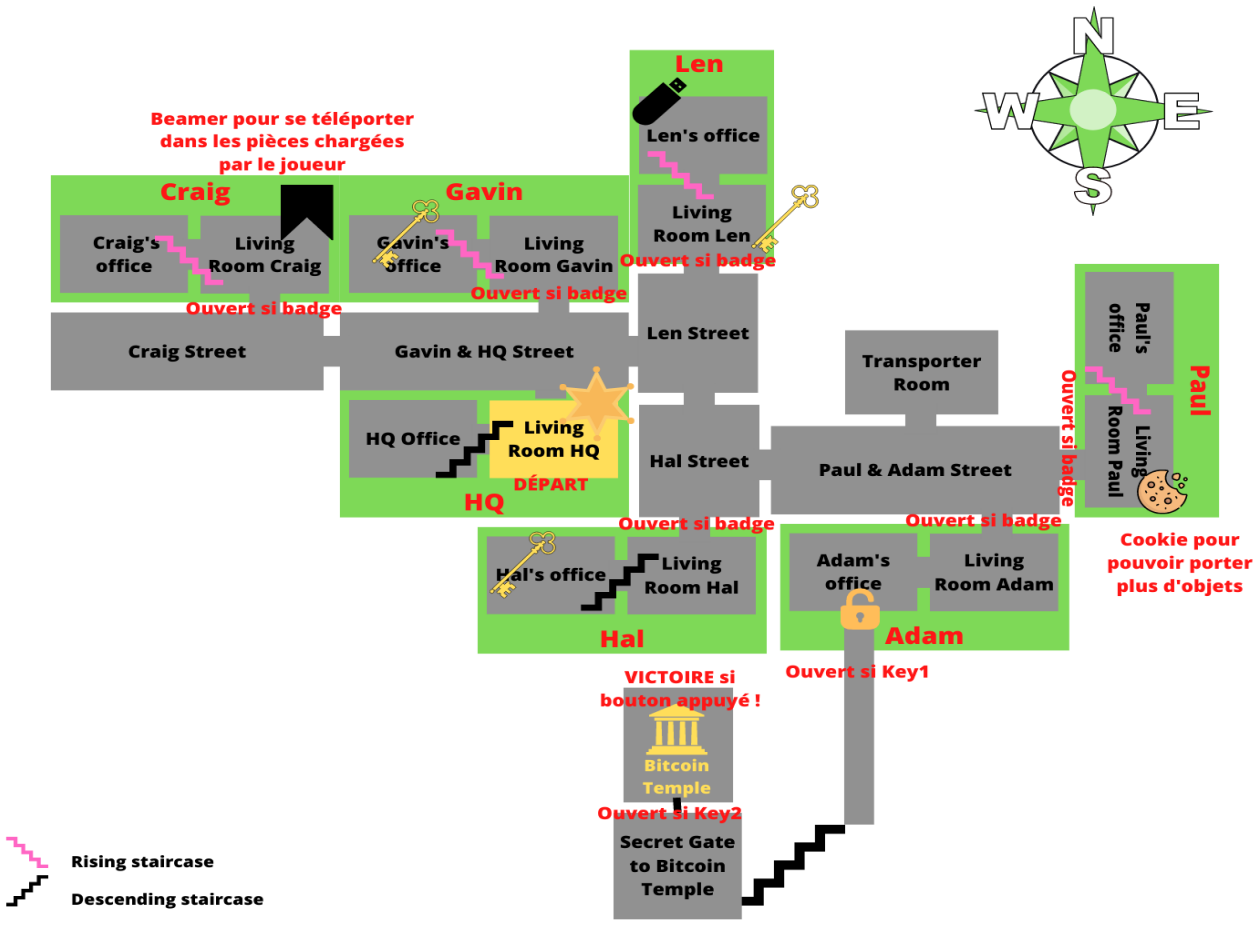
B) Thème (phrase-thème validée)

Dans le village où résident les principaux suspects de l'identité de Satoshi Nakamoto (le créateur du Bitcoin) : un enquêteur mandaté par le FMI doit retrouver le QG de Satoshi Nakamoto et arrêter le bitcoin qui menace l'économie mondiale.

C) Résumé du scénario (complet)

Un enquêteur est chargé par le FMI de découvrir l'identité de Satoshi Nakamoto (le créateur du Bitcoin) pour faire cesser les échanges internationaux de Bitcoin, car c'est un système parallèle qui fait concurrence aux grandes devises internationales. L'enquêteur doit résoudre l'enquête le plus vite possible sinon l'économie mondiale va chuter. Au début du jeu, l'enquêteur est dans son QG d'observation localisé au milieu du village où résident les principaux suspects de l'identité de Satoshi Nakamoto. Il faut que l'enquêteur ramasse des indices pour savoir qui est Satoshi parmi l'ensemble des suspects pour trouver le QG. Pour cela, il va devoir trouver le QG de Satoshi Nakamoto et y entrer pour stopper le Bitcoin et sauver l'économie mondiale.

D) Plan complet



E) Scénario détaillé

Le but du jeu est de trouver le QG de Satoshi Nakamoto et d'appuyer sur un bouton pour arrêter le Bitcoin.

Au début du jeu, l'enquêteur est dans son QG d'observation localisé au milieu du village où résident les principaux suspects de l'identité de Satoshi Nakamoto. Il faut que l'enquêteur ramasse des indices pour savoir qui est Satoshi parmi l'ensemble des suspects pour trouver le QG. Pour cela, il va devoir trouver le QG de Satoshi Nakamoto et y entrer.

Tout d'abord, le joueur (enquêteur) se trouve dans le salon de son QG. Ensuite, un minuteur s'active, il doit effectuer le moins de mouvements possibles. Le joueur doit parcourir les pièces du jeu et se débrouiller comme un vrai enquêteur.

Pour pouvoir entrer chez les suspects, il devra être muni de son badge d'agent du FMI. Le joueur doit trouver les 3 clés permettant d'ouvrir la pièce secrète dans le bureau d'Adam, un suspect.

Ces trois clés sont chez Gavin, Hal et Len.

Le joueur doit donc aller chez Gavin. Il trouve également une clé qu'il peut prendre ou non car pour l'instant on ne sait pas à quoi elle sert.

Chez Craig, le joueur peut récupérer un beamer(téléporteur). Un téléporteur est un appareil qui peut être chargé et déclenché. Lorsqu'on charge le téléporteur, il mémorise la pièce actuelle. Lorsqu'on déclenche le téléporteur, il nous ramène immédiatement dans la pièce dans laquelle il a été chargé

Il doit ensuite aller chez Len pour trouver une autre clé mais il s'aperçoit qu'il ne peut pas la prendre. Il a une capacité de poids limité. Dans ce cas, il doit chercher un cookie magique chez Paul.

Puis il doit aller chez Hal pour trouver la dernière clé. Une fois que toutes les clés sont trouvées, il faut trouver la porte qui est fermée. Celle-ci est dans le bureau d'Adam. Une fois la porte ouverte et le joueur entré, il ne peut plus sortir. Il doit donc continuer son chemin et aller dans le QG de Satoshi pour appuyer sur le bouton qui permet de stopper le bitcoin.

F) Détail des lieux, items, personnages

Détail des lieux :

- Living Room HQ : Salon du QG de l'enquêteur. Ensuite, un minuteur s'active.
  Sorties : HQAndGavinStreet, HQOffice

- HQ Office : Bureau du QG de l'enquêteur. Le joueur doit récupérer le badge du FMI.
  Sorties : HQLivingRoom

- Living Room Gavin : Salon de Gavin.
  Sorties : HQAndGavinStreet, GavinOffice

- Gavin's Office : Bureau de Gavin dans lequel l'enquêteur trouve une clé.
  Sorties : GavinLivingRoom

- Living Room Craig : Salon de chez Craig. Le joueur doit récupérer le téléporteur pour faciliter sa mission.
  Sorties : CraigStreet, CraigOffice

- Craig's Office : Bureau de Craig.
  Sorties : CraigLivingRoom

- Living Room Len : Salon de Len dans lequel l'enquêteur trouve une clé.
  Sorties : LenStreet, LenOffice

- Len's Office : Bureau de Len dans lequel l'enquêteur trouve une clé USB.
  Sorties : LenLivingRoom

- Living Room Hal : Salon de Hal.
  Sorties : HalStreet, HalOffice

- Hal's Office : Bureau de Gavin dans lequel l'enquêteur trouve une clé.
  Sorties : HalLivingRoom

- Living Room Paul : Salon de Paul. Il y a un cookie que l'enquêteur peut manger pour lui permettre de porter plus de clés car il ne peut en porter que deux pour l'instant.
  Sorties : PaulAndAdamStreet, PaulOffice

- Paul's Office : Bureau de Paul dans lequel Paul dit à l'enquêteur qu'il pense que Satoshi est Len Sassaman.
  Sorties : PaulLivingRoom

- Living Room Adam : Salon d'Adam. Adam dit à l'enquêteur que Satoshi est Gavin.
  Sorties : PaulAndAdamStreet, AdamOffice

- Adam's Office : Bureau d'Adam dans lequel l'enquêteur trouve une trappe qu'il faut ouvrir avec 3 clés.
  Sorties : AdamLivingRoom, SecretGate

- Secret Gate to Bitcoin Temple : Pièce secrète dans laquelle il y a une Grande porte. Pour l'ouvrir, il faut donner les noms des 3 principaux suspects.

Sorties : AdamOffice, BitcoinTemple

- Bitcoin Temple : QG de Satoshi. Il y a un bouton sur lequel appuyer pour arrêter le Bitcoin.
  Sorties : SecretGate

- Gavin & HQ Street : Rue qui se situe en face du QG de l'enquêteur et du domicile de Gavin.
  Sorties : GavinLivingRoom, LenStreet, HQLivingRoom, CraigStreet

- Craig Street : Rue qui se situe en face du domicile de Craig.
  Sorties : CraigLivingRoom, HQAndGavinStreet

- Len Street : Rue qui se situe en face du domicile de Len.
  Sorties : LenLivingRoom, HalStreet, HQAndGavinStreet

- Hal Street : Rue qui se situe en face du domicile de Hal.
  Sorties : LenStreet, PaulAndAdamStreet, HalLivingRoom

- Paul & Adam Street : Rue qui se situe en face des domiciles d'Adam et Paul.
  Sorties : PaulLivingRoom, AdamLivingRoom, HalStreet, TransporterRoom

Personnages :

- L'enquêteur (le personnage principale)

Items :

- 1 clé USB,
- 3 clés,
- Badge d'agent du FMI,
- Cookie pour porter plus d'objets,
- Beamer(téléporteur)

G) Situations gagnantes et perdantes

Situation gagnante : Appuyer sur le bouton pour désactiver le Bitcoin.

Situation perdante : Faire trop de mouvements avant de gagner.

H) Énigmes, mini-jeux, combats, etc.

Il n'y en aura pas.

I) Commentaires (ce qui manque, reste à faire, ...)

Ajouter des personnages et des énigmes.

II. Réponses aux exercices

Exercice 7.5 :

On constate un problème de duplication de code de l'affichage des sorties de la pièce courante dans les tâches : printWelcome() et goRoom() de la classe Game. Voici le code concerné :

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

Pour remédier à ce problème, je crée dans la classe Game une méthode printLocationInfo() qui contiendra le code dupliqué. Voici ce que nous obtenons :

```
/**
 * Affiche les détails de l'endroit dans lequel on est
 * et où nous pouvons aller.
 */
private void printLocationInfo()
{
    System.out.println("You are "+aCurrentRoom.getDescription());
    System.out.print("Exits: ");
    if (this.aCurrentRoom.aNorthExit != null){
        System.out.print("north ");
    }
    if (this.aCurrentRoom.aSouthExit != null){
        System.out.print("south ");
    }
    if (this.aCurrentRoom.aEastExit != null){
        System.out.print("east ");
    }
    if (this.aCurrentRoom.aWestExit != null){
        System.out.print("west ");
    }
    System.out.println("");
}//printLocationInfo()
```

Ainsi, je peux appeler cette méthode dans printWelcome() et goRoom(), comme ceci :

```
private void printWelcome()
{
  System.out.println("Welcome to the village of Satoshi Nakamoto, the creator of Bitcoin! \n" +
  "The Search for Satoshi Nakamoto is an incredible investigation to discover the identity of the creator of bitcoin! \n" +
  "Type 'help' if you need help.");
  System.out.println();
  printLocationInfo();
} // printWelcome()
```

```
    this.aCurrentRoom = vNextRoom;
    printLocationInfo();
} // goRoom()
```

Exercice 7.6 :

Le livre nous propose une version de setExits qui si chaque paramètre est null avant de l'affecter à l'attribut correspondant, mais ce n'est pas indispensable car nous affectons déjà les paramètres adéquats aux situations.

Ensuite, il nous est demandé d'utiliser l'encapsulation pour réduire le couplage afin d'avoir une meilleure visibilité sur les classes.

Pour faire cela, j'écris l'accesseur getExit(final String pDirection) dans la classe Room qui retourne la pièce à laquelle correspond une direction. Voici ce qu'on obtient :

```
public Room getExit(final String pDirection)
{
    if(pDirection.equals("north")){
        return aNorthExit;
    }
    if(pDirection.equals("east")){
        return aEastExit;
    }
    if(pDirection.equals("south")){
        return aSouthExit;
    }
    if(pDirection.equals("west")){
        return aWestExit;
    }
    return null;
}
```

On peut ainsi faire un appel de l'accesseur getExit() dans la méthode goRoom() de la classe Game. Je remplace donc cela :

```
String vDirection = pInstruction.getSecondWord();
if (vDirection.equals("north")){
    vNextRoom = this.aCurrentRoom.aNorthExit;
}else if (vDirection.equals("south")){
    vNextRoom = this.aCurrentRoom.aSouthExit;
}else if (vDirection.equals("east")){
    vNextRoom = this.aCurrentRoom.aEastExit;
}else if (vDirection.equals("west")){
    vNextRoom = this.aCurrentRoom.aWestExit;
}else{
    System.out.println("Unknown direction!");
    return;
}
```

Par cela :

```
Room vNextRoom = this.aCurrentRoom.getExit(pInstruction.getSecondWord());
```

Cependant avec ce remplacement, le message "Unknown direction" n'apparaît plus lorsque la direction est inconnue.

J'ai donc effectué la solution suggérée par Monsieur Bureau.

Au début de la calsse Room, j'ai ajouté :

```
public static final Room UNKNOWN_DIRECTION = new Room( "nowhere" );
```

Ensuite, à la fin de l'accesseur getExit() de la classe Room:

```
// Si la direction est inconnue, renvoie une constante définie comme étant une direction inconnue
return UNKNOWN_DIRECTION;
```

Enfin, dans la méthode goRoom() de la classe Game:

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

Cette condition vérifie si la pièce suivante est inconnue, c'est-à-dire si la méthode getExit() renvoie la constante UNKNOWN_DIRECTION. Si c'est le cas, alors la direction est inconnue et un message d'erreur est affiché sur la console. La méthode s'arrête ensuite prématurément avec l'instruction return, ce qui empêche le code suivant d'être exécuté.

Exercice 7.7 :

Il nous est demander d'apporter une modification similaire à la méthode printLocationInfo() de la classe Game afin que les détails des sorties soient désormais préparés par la classe Room plutôt que par la classe Game. Pour faire ceci, il faut définir la méthode getExitString() dans la classe Room. Cette méthode vérifie si une sortie existe pour chaque direction avant de l'ajouter à la chaîne de caractères. La chaîne de caractères complétée est ensuite retournée. Si aucune sortie n'existe, la chaîne de caractères sera simplement "Exits : ". J'obtiens ceci :

Ensuite, il faut que la méthode printLocationInfo() de la classe Game appelle la méthode getExitString() de la classe Room. Voici ce que cela donne :

```
/**
 * Affiche les détails de l'endroit dans lequel on est
 * et où nous pouvons aller.
 */
private void printLocationInfo()
{
    System.out.println("You are "+aCurrentRoom.getDescription()+"\n"+
    this.aCurrentRoom.getExitString());
}//printLocationInfo()
```

Cela allège le code et devient plus cohérent par rapport au rôle de la classe Game qui doit être le moteur du jeu.

Exercice 7.8 :

On nous demande d'implémenter dans notre projet une HashMap. Cela permet de faciliter l'aboutissement à l'ajout de direction (ex : étages) dans la map du projet.

Pour ce faire, j'importe HashMap avant son utilisation dans la classe Room comme ceci :

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

```
public class Room
{
    private String aDescription;
    private HashMap <String, Room> aExits;
```

Ensuite, j'initialise la HashMap dans le constructeur Room() de la classe Room pour qu'elle dépende de la direction et de la pièce.

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

Puis je modifie la méthode getExit() de la classe Room afin qu'elle appelle les attributs de la HashMap.

```
public Room getExit(final String pDirection)
{
  return aExits.get(pDirection);
}//getExit()
```

Cependant, cela n'est pas compatible avec la procédure setExits() de la classe Room qui n'accède pas encore à la HashMap :

```
public void setExits( final Room pNorthExit, final Room pEastExit,
final Room pSouthExit, final Room pWestExit)
{
  this.aNorthExit = pNorthExit;
  this.aEastExit = pEastExit;
  this.aSouthExit = pSouthExit;
  this.aWestExit = pWestExit;
} // setExits()
```

Je la modifie donc afin qu'elle puisse définir la pièce dans laquelle on peut aller en fonction de la direction (pDirection). Je la renomme setExit() car elle ne définira qu'une sortie à chaque fois :

```
/**
 * Define an exit from this room.
 * @param pDirection The direction of the exit.
 * @param pNeighbor The room in the given direction.
 */
public void setExit( final String pDirection, final Room pNeighbor)
{
  aExits.put(pDirection, pNeighbor);
} // setExit()
```

Cela permet de ne plus indiquer les sorties « null » dans la méthode createRooms () de la classe Game mais uniquement les sorties existantes. On passe donc de ça :

```
//positionner les sorties pour créer le "réseau" de lieux
//Nord, Est, Sud, Ouest
vHQLivingRoom.setExits(vHQAndGavinStreet,null,null,null);
vHQOffice.setExits(null,null,null,null);
vHQAndGavinStreet.setExits(vGavinLivingRoom,vLenStreet,vHQLivingRoom,vCraigStreet);
vGavinLivingRoom.setExits(null,null,vHQAndGavinStreet,null);
vGavinOffice.setExits(null,null,null,null);
vCraigStreet.setExits(vCraigLivingRoom,vHQAndGavinStreet,null,null);
vCraigLivingRoom.setExits(null,null,vCraigStreet,null);
vCraigOffice.setExits(null,null,null,null);
vLenStreet.setExits(vLenLivingRoom,null,vHalStreet,vHQAndGavinStreet);
vLenLivingRoom.setExits(null,null,vLenStreet,null);
vLenOffice.setExits(null,null,null,null);
vHalStreet.setExits(vLenStreet,vPaulAndAdamStreet,vHalLivingRoom,null);
vHalLivingRoom.setExits(vHalStreet,null,null,null);
vHalOffice.setExits(null,null,null,null);
vPaulAndAdamStreet.setExits(null,vPaulLivingRoom,vAdamLivingRoom,vHalStreet);
vPaulLivingRoom.setExits(null,null,null,vPaulAndAdamStreet);
vPaulOffice.setExits(null,null,null,null);
vAdamLivingRoom.setExits(vPaulAndAdamStreet,null,null,vAdamOffice);
vAdamOffice.setExits(null,vAdamLivingRoom,null,null);
vSecretGate.setExits(null,null,null,null);
vBitcoinTemple.setExits(null,null,null,null);
```

À ça (exemple) :

```
//positionner les sorties pour créer le "réseau" de lieux
//Nord, Est, Sud, Ouest
vHQLivingRoom.setExit("north",vHQAndGavinStreet);

vHQAndGavinStreet.setExit("north",vGavinLivingRoom);
vHQAndGavinStreet.setExit("east",vLenStreet);
vHQAndGavinStreet.setExit("south",vHQLivingRoom);
vHQAndGavinStreet.setExit("west",vCraigStreet);

vGavinLivingRoom.setExit("south",vHQAndGavinStreet);
```

On voit donc que l'on peut créer de nouvelles directions sans beaucoup d'efforts car le code ne dépend plus de seulement 4 directions du fait de la HashMap.

Cependant, en ajoutant la Hashmap, j'ai pu effacer certains « if » de go Room

Je ne peux donc plus laisser le return UNKNOWN_DIRECTION à la fin, ce qui est problèmatique.

Maintenant, quand le 2e mot est inconnu il m'affiche seulement "There's no door !".

Pour remédier à ce problème, je crée la classe DirectionWords comportant le tableau des directions reconnues et une fonction isDirection(). Comme ceci :

```
public class DirectionWords {
    private String[] aDirection = new String[6];

    public DirectionWords()
    {
        this.aDirection[0] = "north";
        this.aDirection[1] = "east";
        this.aDirection[2] = "south";
        this.aDirection[3] = "west";
        this.aDirection[4] = "up";
        this.aDirection[5] = "down";
    }

    public boolean isDirection(final String pDirection) {
        for (int i=0; i< aDirection.length; i++) {
            if (pDirection.equals(this.aDirection[i])){
                return true;
            }
        }
        return false;
    }
}
```

Ensuite, je crée un objet de cette classe dans la classe Game, puisque c'est là qu'on doit traiter le problème. Enfin, je crée deux boucles dans la méthode goRoom() afin de vérifier si la direction est valide en utilisant la méthode isDirection() de la classe DirectionWords. Comme ceci :

```
// On crée une variable Room appelée vNextRoom et on l'initialise à null.
Room vNextRoom = null;

// On récupère le deuxième mot de l'instruction et on le stocke dans une variable appelée vDirection.
String vDirection = pInstruction.getSecondWord();

// On vérifie si la direction est une direction valide en utilisant la méthode isDirection de l'objet aDirection de la classe actuelle.
if (!this.aDirection.isDirection(vDirection)) {
    System.out.println("Unknown direction!");
    return;
}

vNextRoom = this.aCurrentRoom.getExit(vDirection);

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

Exercice 7.8.1 :

Dans mon plan de jeu, j'avais prévu 4 escaliers pour monter et 3 pour descendre dans d'autres pièces.

Voici un exemple d'une des modifications que j'ai dû effectuer (up pour monter et down pour descendre) :

```
vHQLivingRoom.setExit("north",vHQAndGavinStreet);
vHQLivingRoom.setExit("down",vHQOffice);

vHQOffice.setExit("up",vHQLivingRoom);
```

Exercice 7.9 :

Avec la HashMap, on fait face à un problème d'indications des sorties et les erreurs de directions ne sont pas indiquées. En effet, cela vient principalement de la méthode getExitString qui ne prend pas en compte la HashMap :

```
public String getExitString()
{
  String exitString = "Exits: "; // On initialise la chaîne de caractères avec la mention "Exits : "
  // Si une sortie existe pour la direction nord, on l'ajoute à la chaîne de caractères
  if (this.getExit("north") != null) {
  exitString += "north ";
  }

  // Si une sortie existe pour la direction est, on l'ajoute à la chaîne de caractères
  if (this.getExit("east") != null) {
      exitString += "east ";
  }

  // Si une sortie existe pour la direction sud, on l'ajoute à la chaîne de caractères
  if (this.getExit("south") != null) {
  exitString += "south ";
  }

  // Si une sortie existe pour la direction ouest, on l'ajoute à la chaîne de caractères
  if (this.getExit("west") != null) {
      exitString += "west ";
  }

  // On retourne la chaîne de caractères complétée
  return exitString;

}//getExitString()
```

On doit donc la mettre à jour en important Set :

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

public class Room
{
```

Et en modifiant la méthode :

```
public String getExitString()
{
  String exitString = "Exits: "; // On initialise la chaîne
  Set<String> vKeys = this.aExits.keySet();
  for(String vExit : vKeys){
      exitString +=' '+vExit;
  }
  return exitString;
}//getExitString()
```

Ce code est une méthode publique qui renvoie une chaîne de caractères représentant les sorties d'une pièce. La chaîne est construite en itérant sur un ensemble de clés représentant les sorties de la pièce, et en ajoutant chaque sortie à la chaîne exitString.

La méthode utilise un objet Set<String> pour stocker les clés, qui sont des chaînes de caractères représentant les noms de chaque sortie de la pièce. La boucle for itère sur chaque clé dans l'ensemble de clés et ajoute chaque clé à la chaîne de sortie exitString en utilisant l'opérateur de concaténation de chaîne "+=".

La méthode retourne la chaîne exitString qui contient toutes les sorties de la pièce.

Exercice 7.10 :

La méthode getExitString() est une méthode qui retourne une chaîne de caractères représentant les sorties disponibles à partir de l'objet courant. Elle est probablement utilisée dans le contexte d'une simulation de jeu ou d'un programme qui modélise un environnement à explorer, où les sorties représentent les différents endroits que l'utilisateur peut visiter.

La méthode commence par initialiser une variable exitString avec la chaîne de caractères "Exits: ", qui sera le préfixe de la chaîne de sortie. Ensuite, la méthode obtient l'ensemble des clés de la carte des sorties de l'objet courant à l'aide de la méthode keySet() de l'interface Map. Cette carte des sorties est probablement une Map où chaque clé représente une direction et chaque valeur représente l'objet ou l'endroit correspondant à cette direction.

Ensuite, la méthode boucle sur l'ensemble des clés obtenues à partir de la carte des sorties et pour chaque clé, elle ajoute à la chaîne exitString une chaîne de caractères représentant la direction de sortie, avec un espace devant, en utilisant l'opérateur de concaténation +.

Finalement, la méthode retourne la chaîne exitString qui contient la liste des sorties disponibles à partir de l'objet courant. La chaîne de caractères finale aura la forme "Exits: direction1 direction2 direction3 ...", où "direction1", "direction2", "direction3", etc. sont les différentes directions de sortie disponibles à partir de l'objet courant.

Exercice 7.11 :

Nous utilisons la procédure printLocationInfo() afin d'avoir une description du lieu où nous sommes. Cependant, on ne peut pas avoir une description plus détaillée de la pièce où on se trouve, donc on peut ajouter la méthode getLongDescription() dans la classe Room :

```
/**
 * Retourne une description complète de la pièce, incluant sa description courte
 * ainsi que les sorties disponibles.
 *
 * @return La description complète de la pièce.
 */
public String getLongDescription()
{
  // Concatène la description courte avec une chaîne contenant la liste des sorties.
  // On utilise la méthode getExitString() pour générer cette liste.
  String longDescription = "You are " + this.aDescription + "\n" + getExitString();

  // Retourne la description complète de la pièce.
  return longDescription;
}
```

Nous pouvons ainsi modifier printLocationInfo() dans la classe Game:

```
/**
 * Affiche les détails de l'endroit dans lequel on est
 * et où nous pouvons aller.
 */
private void printLocationInfo()
{
    System.out.println(aCurrentRoom.getLongDescription());
}//printLocationInfo()
```

Exercice 7.12 :

Cf Plan complet : I) D).

Exercice 7.14 :

Notre objectif est de développer une méthode permettant de visualiser le contenu de la pièce. Afin de minimiser l'impact sur les autres classes, nous allons utiliser la localisation des informations pour modifier la classe. Tout d'abord, nous allons inclure la commande "look" dans le tableau de commandes valides de la classe CommandWords, afin que le programme puisse la reconnaître. Ensuite, nous allons créer une nouvelle procédure "look" dans la classe Game qui affichera le contenu de la pièce. Enfin, nous allons ajouter la possibilité pour le joueur d'entrer la commande "look" dans la méthode processCommand de la classe Game, qui exécutera alors la méthode "look".

Pour ce faire, on remplace cela :

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

Par ce tableau déclaré dans la classe CommandWords:

```
public class CommandWords
{
    // a constant array that will hold all valid command words
    private static final String aValidCommands[] = {"go","quit","help","look",
```

Voici la procédure look créée dans la classe Game :

```
/**
 * Affiche la description de la pièce et ses sorties
 */
private void look()
{
    System.out.println(this.aCurrentRoom.getLongDescription());
}
```

On ajoute donc une condition dans la fonction booléenne processCommand() :

```
/**
 * Traite l'instruction entrée par l'utilisateur.
 * @param pInstruction2 L'instruction à traiter.
 * @return true si l'instruction est de quitter le jeu, sinon false.
*/
private boolean processCommand(final Command pInstruction2)
{
    // Si la commande est inconnue, on affiche un message d'erreur et on retourne false.
    if(pInstruction2.isUnknown()){
    System.out.println("I don't know what you mean...");
    return false;
    }

    // Si la commande est "quit", on appelle la méthode quit et on retourne sa valeur de retour.
    if(pInstruction2.getCommandWord().equals("quit")){
    return quit(pInstruction2);
    }

    // Si la commande est "go", on appelle la méthode goRoom et on retourne false.
    if(pInstruction2.getCommandWord().equals("go")){
    goRoom(pInstruction2);
    return false;
    }

    // Si la commande est "help", on appelle la méthode printHelp et on retourne false.
    if(pInstruction2.getCommandWord().equals("help")){
    printHelp();
    return false;
    }
```

```
    // Si la commande est "look", on appelle la méthode look et on retourne false.
    if(pInstruction2.getCommandWord().equals("look")){
    this.look();
    return false;
    }

    // Si la commande n'est pas reconnue, on affiche un message d'erreur et on retourne false.
    System.out.println("Erreur du programmeur : commande non reconnue !");
    return false;
} // processCommand( )
```

Exercice 7.15 :

Nous ajoutons une commande « eat » qui affichera « You have eaten now and you are not hungry any more ». Pour ce faire, on ajoute « eat » dans le tableau aValidCommands de la classe CommandWords.

```
public class CommandWords
{
    // a constant array that will hold all valid command words
    private static final String aValidCommands[] = {"go","quit","help","look","eat"};
```

Ensuite, on crée une procédure privée eat() dans la classe Game qui affichera le message indiquant que le joueur a mangé et n'a plus faim.

```
/**
 * Affiche un message indiquant que le joueur a mangé et n'a plus faim.
 */
private void eat()
{
    System.out.println("You have eaten now and you are not hungry any more");
}
```

Enfin, on ajoute la condition de la procédure eat() dans la fonction processCommand() de la classe Game.

```
// Si la commande est "eat", on appelle la méthode eat et on retourne false.
if(pInstruction2.getCommandWord().equals("eat")){
this.eat();
return false;
}
```

Exercice 7.16 :

Après l'ajout de deux nouvelles commandes, il est apparu que le message d'aide existant n'était plus complet. Malheureusement, cela implique toujours un couplage implicite. Pour éviter cette situation, nous devons mettre à jour la méthode "printHelp()" pour qu'elle affiche toutes les commandes disponibles sans avoir à être modifiée ou rajoutée manuellement dans le "System.out.println()".

Une solution possible consiste à créer une procédure statique showAll() dans la classe CommandWords. Cette procédure inclura une boucle for each qui affichera chaque commande du tableau de commandes valides (aValidCommands). En utilisant cette méthode, nous obtiendrons une liste complète de toutes les commandes disponibles.

Voici ce que j’ai ajouté :

```java
/**
 * Affiche toutes les commandes valides.
 */
public static void showAll() {
    // Pour chaque commande valide dans le tableau aValidCommands,
    for (String vCommand : aValidCommands) {
        // Affiche la commande suivie d'un espace.
        System.out.print(vCommand + " ");
    }
    // Saut de ligne final.
    System.out.println();
}// showAll()
```

On appel ensuite cette méthode dans printHelp(). Or, comme les classes Game et CommandWords ne sont pas liées, on appel la méthode showAll() dans la classe Parser grâce à showCommand() :

```java
/**
 * Affiche la liste de toutes les commandes valides.
 */
public static void showCommands()
{
    // Utilisation de la méthode showAll() de la classe CommandWords pour afficher toutes les commandes valides
    CommandWords.showAll();
}// showCommands()
```

Cette nouvelle méthode est donc appelée par la méthode printHelp() de la classe Game :

```java
/**
 * Affiche les commandes disponibles pour le joueur.
 */
private void printHelp()
{
    System.out.println("You are lost. You are alone. \n" +
    "You wander around at the village of Satoshi. \n" +
    "Your command words are:");
    Parser.showCommands();
} // printHelp( )
```

Exercice 7.18 :

Nous souhaitons que la classe CommandWords produise des commandes plutôt que de les afficher. Pour cela, nous allons remplacer la procédure showAll() par une fonction nommée getCommandList(). Cette fonction retournera la liste complète des commandes possibles, en adoptant une approche similaire à celle de la méthode getExitString() de la classe Room.

Voici la nouvelle méthode getCommandList() de la CommandWords :

```java
/**
 * Retourne une chaîne de caractères représentant la liste des commandes valides.
 *
 * @return La chaîne de caractères représentant la liste des commandes valides.
 */
public String getCommandList() {
    String vResult = "";
    // Itération à travers la liste des commandes valides.
    for (String vCommand : aValidCommands) {
        // Ajout de chaque commande suivie d'un espace à la chaîne de résultats.
        vResult += vCommand + " ";
    }
    return vResult;
}// getCommandList()
```

Ce remplacement de méthode entraîne la modification de la méthode showCommands() dans la classe Parser afin d'accéder à getCommandList(). Voici la nouvelle méthode showCommands() de la classe Parser :

```java
/**
 * Affiche la liste de toutes les commandes valides.
 *
 * @return La chaîne de caractères représentant la liste des commandes valides.
 */
public String showCommands() {
    // Utilisation de la méthode getCommandList() de la classe CommandWords pour obtenir la liste des commandes valides.
    return this.aValidCommands.getCommandList();
}// showCommands()
```

Il faut ensuite modifier la procédure printHelp() de la classe Game en remplaçant Parser.showCommands() par System.out.println(aParser.showCommands()) à cause du fait que la méthode showCommands n'est plus static, comme ceci :

```java
/**
 * Affiche les commandes disponibles pour le joueur.
 */
private void printHelp()
{
    System.out.println("You are lost. You are alone. \n" +
    "You wander around at the village of Satoshi. \n" +
    "Your command words are:");
    System.out.println(aParser.showCommands());
} // printHelp( )
```

Exercice 7.18.1 :

En comparant mon code avec zuul-better, j'ai constaté qu'il fallait remplacé les if() de la méthode processCommand() de la classe Game par des else if. Voici la méthode après modification.

```
private boolean processCommand(final Command pInstruction2)
{
    // Si la commande est inconnue, on affiche un message d'erreur et on retourne false.
    if(pInstruction2.isUnknown()){
    System.out.println("I don't know what you mean...");
    return false;
    }

    // Si la commande est "quit", on appelle la méthode quit et on retourne sa valeur de retour.
    else if(pInstruction2.getCommandWord().equals("quit")){
    return quit(pInstruction2);
    }

    // Si la commande est "go", on appelle la méthode goRoom et on retourne false.
    else if(pInstruction2.getCommandWord().equals("go")){
    goRoom(pInstruction2);
    return false;
    }

    // Si la commande est "help", on appelle la méthode printHelp et on retourne false.
    else if(pInstruction2.getCommandWord().equals("help")){
    printHelp();
    return false;
    }

    // Si la commande est "look", on appelle la méthode look et on retourne false.
    else if(pInstruction2.getCommandWord().equals("look")){
    this.look();
    return false;
    }
```

```
    // Si la commande est "eat", on appelle la méthode eat et on retourne false.
    else if(pInstruction2.getCommandWord().equals("eat")){
    this.eat();
    return false;
    }

    // Si la commande n'est pas reconnue, on affiche un message d'erreur et on retourne false.
    System.out.println("Erreur du programmeur : commande non reconnue !");
    return false;
} // processCommand( )
```

Exercice 7.18.2 :

J’ai apporté des modifications à la fonction getExitString() de la classe Room en utilisant un StringBuilder pour ajouter des caractères à une chaîne de caractères existante. Voici la fonction modifiée :

```java
/**
 * Retourne une chaîne de caractères représentant les sorties possibles depuis la pièce courante.
 *
 * @return La chaîne de caractères représentant les sorties possibles depuis la pièce courante.
 */
public String getExitString() {
    // On utilise StringBuilder pour construire la chaîne de caractères de sortie.
    StringBuilder exitString = new StringBuilder("Exits:");

    // On récupère l'ensemble des clés (directions) de la table de hachage des sorties.
    Set<String> vKeys = this.aExits.keySet();

    // On boucle sur l'ensemble des clés pour les ajouter à la chaîne de caractères de sortie.
    for (String vExit : vKeys) {
        exitString.append(" "+vExit);
    }

    // On retourne la chaîne de caractères de sortie.
    return exitString.toString();
}//getExitString()
```

Exercice 7.18.3 :

Images trouvées sur Pinterest.

Exercice 7.18.4 :

Le titre de mon jeu est : « The Search for Satoshi ».

Exercice 7.18.5 (fait à partir de l'exercice 7.46):

À ce stade, les objets Room du jeu ne sont accessibles que dans createRooms().

Pour y avoir accès depuis n'importe quelle méthode ou classe, nous allons créer une HashMap contenant toutes les Room (associées à leur nom). Pour ce faire, on créé l'attribut privé aRooms de type HashMap dont la clé est de type String et la valeur est de type Room. Cette variable va être utilisée pour stocker toutes les pièces du jeu et permettre leur accès via leur nom. On l'initialise ensuite dans le constructeur dans la classe GameEngine.

```java
public class GameEngine
{
    private Parser aParser; // Le parser utilisé pour interpréter les commandes du joueur.
    private UserInterface aGui; // Interface utilisateur.
    private Player aPlayer;
    private HashMap <String, Room> aRooms; // Déclare une variable de classe "aRooms" de type HashMap qui stocke des objets Room.

    /**
     * Constructeur de la classe GameEngine.
     * Initialise les pièces du jeu, le parser et les mots de direction.
     */
    public GameEngine() {
        this.createRooms();// On crée les pièces du jeu
        this.aRooms = new HashMap <String, Room> ();// On initialise la variable aRooms avec un nouveau HashMap vide
        this.aParser = new Parser();// On crée un nouveau parser pour interpréter les commandes du joueur
    }
```

Puis, on utilise la méthode put() de l'objet aRooms pour ajouter des instances de la classe Room dans la HashMap. Chaque instance de Room représente une pièce différente dans le jeu, avec son propre nom unique. La clé pour chaque pièce est une chaîne de caractères qui correspond au nom de la pièce. Par exemple, la première ligne de code ajoute l'objet vHQLivingRoom à la HashMap, avec la clé "investigator's living room". De cette façon, on peut accéder à chaque pièce en utilisant son nom comme clé pour la HashMap

```
aRooms.put("investigator's living room", vHQLivingRoom);
aRooms.put("HQ office",vHQOffice);
aRooms.put("street of the HQ and Gavin",vHQAndGavinStreet);
aRooms.put("Gavin's living room", vGavinLivingRoom);
aRooms.put("Gavin's office", vGavinOffice);
aRooms.put("the street of Craig", vCraigStreet);
aRooms.put("Craig's living room", vCraigLivingRoom);
aRooms.put("Craig's office", vCraigOffice);
aRooms.put("street of Len", vLenStreet);
aRooms.put("Len's living room", vLenLivingRoom);
aRooms.put("Len's office", vLenOffice);
aRooms.put("street of Hal", vHalStreet);
aRooms.put("Hal's living room", vHalLivingRoom);
aRooms.put("Hal's office", vHalOffice);
aRooms.put("street of Paul and Adam", vPaulAndAdamStreet);
aRooms.put("Paul's living room", vPaulLivingRoom);
aRooms.put("Paul's office", vPaulOffice);
aRooms.put("Adam's living room", vAdamLivingRoom);
aRooms.put("Adam's office", vAdamOffice);
aRooms.put("Secret Gate", vSecretGate);
aRooms.put("Satoshi Nakamoto's HQ", vBitcoinTemple);
```

Exercice 7.18.6 :

À ce stade, nous devons ajouter des images au jeu. Pour ce faire, nous étudions le fichier zuul-with-images.jar pour comprendre son fonctionnement global et apporter des modifications au jeu :

On constate qu'il n'y a pas de changement pour Command et CommandWords. Nous avons décidé d'ajouter un attribut supplémentaire à notre classe Room : le nom de l'image de la pièce.

```
//Le nom de l'image utilisé pour chaque pièce;
private String aImageName;
```

Cela, nous amène à modifier le constructeur de la classe pour y inclure une image correspondante à la pièce.

```
/**
 * Crée une nouvelle pièce avec une description donnée.
 *
 * @param pDescription la description de la pièce
 * @param pImageName le nom de l'image utilisé pour chaque pièce
 */
public Room(final String pDescription, final String pImageName)
{
    // Initialise la variable de description avec la valeur donnée en paramètre
    this.aDescription = pDescription;

    // Initialise la table de hachage des sorties avec une nouvelle instance de HashMap
    aExits = new HashMap <String, Room>();

    this.aImageName = pImageName;
}//Room()
```

On ajoute un accesseur qui retourne le nom de l'image aImageName.

```
public String getImageName()
{
    return this.aImageName;
}
```

Désormais, on ajoute le paramètre image pour chaque pièce de la méthode createRooms() de la classe Game :

```
Room vHQLivingRoom = new Room ("in the investigator's HQ lounge","HQLivingRoom.png");
```

Ensuite, on modifie la classe Parser qui n'a plus besoin de Scanner car elle sera remplacée par la classe StringTokenizer qui sera importée.

```
import java.util.StringTokenizer;
```

Puis, on remplace l'objet Scanner par un objet CommandWords pour que ce soit compatible et que le programme se concentre sur les mots plutôt que sur la lecture.

```
public class Parser
{
    private CommandWords aValidCommands;  // (voir la classe CommandWords)
    private CommandWords aCommandWords;  // permettra de lire les commandes au clavier

    /**
     * Constructeur par defaut qui cree les 2 objets prevus pour les attributs
     */
    public Parser()
    {
        this.aValidCommands = new CommandWords();
        this.aCommandWords = new CommandWords();
        // System.in designe le clavier, comme System.out designe l'ecran
    } // Parser()
```

Cela nous amène à modifier getCommand() pour ne plus qu'il y ait de problèmes de compilation. Il faut supprimer les anciens attributs. De plus hasNext() n'existe plus lorsqu'on supprime Scanner. On le remplace donc par hasMoreTokens() et next() par nexToken().

```
/**
 * Get a new command from the user. The command is read by
 * parsing the 'inputLine'.
 */
public Command getCommand( final String pInputLine )
{
    String vWord1;
    String vWord2;

    StringTokenizer tokenizer = new StringTokenizer( pInputLine );

    if ( tokenizer.hasMoreTokens() )
        vWord1 = tokenizer.nextToken();      // get first word
    else
        vWord1 = null;

    if ( tokenizer.hasMoreTokens() )
        vWord2 = tokenizer.nextToken();      // get second word
    else
        vWord2 = null;

    // note: we just ignore the rest of the input line.

    // Now check whether this word is known. If so, create a command
    // with it. If not, create a "null" command (for unknown command).

    if ( this.aCommandWords.isCommand( vWord1 ) )
        return new Command( vWord1, vWord2 );
    else
        return new Command( null, vWord2 );
} // getCommand(.)
```

Ensuite, on doit créer deux classes : UserInterface et GameEngine. Il suffit de copier ces classes de zuul-with-images.jar puis de les coller dans les classes créées. Cependant, la classe GameEngine reprend l'essentiel de l'ancienne classe Game mais avec quelques modifications. Au final, on ne laisse que le constructeur dans la classe Game et on ajoute deux nouveaux attributs aEngine et aGui. La méthode play a disparu au profit de l'interface graphique.

Voici la classe Game :

```
public class Game
{
    private GameEngine aEngine; // Moteur de jeu
    private UserInterface aGui; // Interface utilisateur

    /**
     * Initialise une nouvelle partie de jeu.
     * Cette méthode crée les pièces, le parser et les mots de direction utilisés dans le jeu.
     */
    public Game() {
        this.aEngine = new GameEngine(); // Créer un nouveau moteur de jeu
        this.aGui = new UserInterface(this.aEngine); // Créer une nouvelle interface utilisateur pour le moteur de jeu
        this.aEngine.setGUI(this.aGui); // Définir l'interface utilisateur pour le moteur de jeu
    } // Game()
} // Game
```

Quant à la classe GameEngine, les System.out.print() sont remplacés par aGui.print().

Exemple :

```
this.aGui.println( "I don't know what you mean..." );
```

On ajoute ensuite la méthode setGUI permet de modifier la valeur de l'interface graphique :

```
public void setGUI( final UserInterface pUserInterface )
{
    this.aGui = pUserInterface;
    this.printWelcome();
}
```

On modifie printLocationInfo() pour qu'elle retourne une image :

```
/**
 * Affiche les détails de l'endroit dans lequel on est
 * et où nous pouvons aller.
 */
private void printLocationInfo()
{
    this.aGui.println(aCurrentRoom.getLongDescription());
    this.aGui.println(aCurrentRoom.getImageName());
}//printLocationInfo()
```

Dans le cas de la classe UserInterface, on commence par importer toutes les classes nécessaires (qui sont déjà utilisées dans zuul-images).

```
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JTextArea;
import javax.swing.JLabel;
import javax.swing.ImageIcon;
import javax.swing.JScrollPane;
import javax.swing.JPanel;
import java.awt.Dimension;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.event.ActionEvent;
import java.net.URL;
import java.awt.image.*;
```

Le reste de cette classe reste inchangé.

Exercice 7.18.8 :

Nous devons ajouter au moins un bouton qui déclenche une des commandes du jeu. Pour ce faire, il faut importer la classe JButton :

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

Et créer 8 attributs de type JButton dans la classe UserInterface, qui correspondent à nos 8 actions possibles :

```
private JButton aButtonNorth;
private JButton aButtonEast;
private JButton aButtonSouth;
private JButton aButtonWest;
private JButton aButtonUp;
private JButton aButtonDown;
private JButton aButtonHelp;
private JButton aButtonQuit;
```

Pour la suite, il est préférable d'utiliser un panneau de commandes/boutons qui doit être incorporer dans le jeu. Pour cela, d'après le site https://perso.telecom-paristech.fr/hudry/coursJava/interSwing/grilleSimple.html, il faut utiliser le gestionnaire de répartition GridLayout et donc l'importer.

```
import java.awt.GridLayout;
```

Ensuite, il faut créer le panneau aPanel qui sera utilisé pour afficher les boutons. On l'initialise ensuite dans createGUI() avec une configuration 3 x 3.

```
private JPanel aPanel;
```

Dans createGUI() :

```
this.aPanel = new JPanel();
this.aPanel.setLayout(new GridLayout(3,3));
```

Puis, il faut initialiser chaque bouton dans createGUI() de manière à qu'il s'affiche avec le nom du bouton correspondant. Et pour chaque bouton, il est nécessaire de le faire fonctionner avec ActionListener(this).

```
this.aButtonNorth = new JButton("North");
this.aButtonNorth.addActionListener(this);
this.aButtonEast = new JButton("East");
this.aButtonEast.addActionListener(this);
this.aButtonSouth = new JButton("South");
this.aButtonSouth.addActionListener(this);
this.aButtonWest = new JButton("West");
this.aButtonWest.addActionListener(this);
this.aButtonUp = new JButton("Up");
this.aButtonUp.addActionListener(this);
this.aButtonDown = new JButton("Down");
this.aButtonDown.addActionListener(this);
this.aButtonHelp = new JButton("Help");
this.aButtonHelp.addActionListener(this);
this.aButtonQuit = new JButton("Quit");
this.aButtonQuit.addActionListener(this);
```

Pour que chaque bouton soit ajouté au panneau, il faut faire ainsi :

```
this.aPanel.add(this.aButtonNorth);
this.aPanel.add(this.aButtonEast);
this.aPanel.add(this.aButtonSouth);
this.aPanel.add(this.aButtonWest);
this.aPanel.add(this.aButtonUp);
this.aPanel.add(this.aButtonDown);
this.aPanel.add(this.aButtonHelp);
this.aPanel.add(this.aButtonQuit);
```

Quant à cette ligne de code, elle ajoute le panneau "aPanel" sur le côté EST (East) du panneau "vPanel". Le BorderLayout est un gestionnaire de disposition qui divise l'espace d'un conteneur en cinq régions : NORTH, SOUTH, EAST, WEST et CENTER. Chaque région peut contenir un seul composant (component). Dans ce cas-ci, le panneau "aPanel" est ajouté à la région EAST du panneau "vPanel".

```
vPanel.add(this.aPanel, BorderLayout.EAST);
```

Enfin, pour que chaque bouton lance l'action qu'il doit exécuter, on implémente le code ci-dessous qui utilise la méthode actionPerformed() de l'interface ActionListener pour gérer les interactions utilisateur avec les boutons de l'interface utilisateur. Chaque bouton est associé à une action de déplacement ou à une commande, et la méthode actionPerformed() exécute l'action appropriée en fonction du bouton cliqué.

```
/**
 * Actionlistener interface for entry textfield.
 */
public void actionPerformed( final ActionEvent pE )
{
    if(pE.getSource() == this.aButtonNorth)
    {
        this.aEngine.interpretCommand("go north");
    }else{
        this.processCommand();
    }
} // actionPerformed(.)
```

Exercice 7.19.2 :

On déplace toutes les images dans un répertoire Images créé à la racine du projet. On doit donc modifier dans la méthode createRoom () les déclarations des pièces de manière à indiquer le chemin d'où se trouvent les images. Pour cela, on modifie le nom de chaque image en rajoutant « Images/ » devant pour indiquer le chemin d'où se trouvent les images.

```
Room vHQOffice = new Room ("in the HQ office","Images/HQOffice.png");
```

Exercice 7.20 :

Ici, nous cherchons à intégrer un item dans l'une des pièces de notre jeu. Ainsi, il est nécessaire de compléter les descriptions de ces pièces en créant une nouvelle classe Item() qui possède deux attributs distincts : une String décrivant l'objet et un réel représentant son poids. Cette classe se compose d'un constructeur naturel initialisant les attributs et trois accesseurs retournant la description de l'item, son poids et un qui retourne ces deux informations d'un coup. Voici la classe Item obtenue :

```java
public class Item
{
    private String aItemDescription;
    private double aItemWeight;

    public Item(final String pItemDescription, final double pItemWeight)
    {
        this.aItemDescription = pItemDescription;
        this.aItemWeight = pItemWeight;
    }//Item()

    public String getDescription()
    {
        return this.aItemDescription;
    }//getDescription()

    public double getItemWeight()
    {
        return this.aItemWeight;
    }//getWeight()

    public String getItemDescription()
    {
        return this.aItemDescription + " and it weighs " + this.aItemWeight + " kg.";
    }
}
```

Désormais, pour ajouter un item dans une pièce, il faut ajouter des méthodes et un attribut de type Item à la classe Room().

```java
private Item aItem;
```

Ensuite, on initialise cet attribut dans un modificateur setItem(), puis on ajoute un accesseur qui retourne l'item et un qui retourne « No item here. » s'il n'y a pas d'item dans une pièce, sinon, il retourne la description de l'item.

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

public Item getItem()
{
    return this.aItem;
}//getItem()

public String getItemString()
{
    if(this.aItem == null){
        return "No item here.";
    }
    else{
        return "There is : " + this.aItem.getItemDescription();
    }
}
```

Puis, on modifie getLongDescription() de manière à qu'elle retourne l'item disponible dans une pièce et son poids en plus.

```java
public String getLongDescription()
{
  // Concatène la description courte avec une chaîne contenant la liste des sorties.
  // On utilise la méthode getExitString() pour générer cette liste.
  String longDescription = "You are " + this.aDescription + "\n" + getExitString() + "\n" + this.getItemString();

  // Retourne la description complète de la pièce.
  return longDescription;
}
```

Enfin, dans la procédure createRooms() de la classe GameEngine, on crée un objet Item auquel on attribut un nom et un poids, puis on associe l'item instancié à la pièce dans laquelle il doit se trouver avec la procédure setItem(). Comme ceci :

```
Item vKey1 = new Item ("Key",1.0);

vGavinOffice.setItem(vKey1);
```

Exercice 7.21 :

La classe Item est responsable de générer les informations relatives à un objet spécifique, tandis que la classe Room est chargée de créer la chaîne de caractères décrivant cet objet, qui est ensuite affichée par GameEngine. Pour que cette description puisse être affichée correctement, il est faut appeler la méthode getLongDescription() de la classe Item.

Exercice 7.21.1 :

Pour améliorer la commande look() de la classe GameEngine et prendre en compte un second mot, on ajoute une condition dans la méthode look() pour vérifier si un deuxième mot est présent. Si c'est le cas, la méthode doit afficher les détails de l'item dont le nom est fourni en second mot. Si le nom fourni ne correspond à aucun item présent dans la pièce, la méthode doit afficher un message d'erreur. Voici look() :

```
private void look(Command command) {
    // Vérifier s'il y a un deuxième mot dans la commande d'observation.
    if (command.hasSecondWord()) {
        String itemName = command.getSecondWord();
        // Trouver l'objet dans la pièce courante.
        Item item = this.aPlayer.getCurrentRoom().getRoomItems().getItem(itemName);
        if (item == null) {
            // Si l'objet n'est pas dans la pièce ou n'existe pas, afficher un message d'erreur.
            this.aGui.println("This item is not in this room or does not exist!");
        } else {
            // Afficher la description de l'objet.
            this.aGui.println(item.getItemDescription());
        }
    } else {
        // S'il n'y a pas de deuxième mot, afficher la description complète de la pièce.
        this.aGui.println(this.aPlayer.getCurrentRoom().getLongDescription());
    }
}
```

Dans la méthode getItem, on modifie le code d'origine pour qu'elle prenne en paramètre un nom d'item pItemName. Ensuite, on vérifie si l'attribut aItem de la pièce correspond au nom de l'item fourni en paramètre et retourne l'item s'il y a correspondance.

Cette modification permet de chercher un item spécifique en fonction de son nom, plutôt que de simplement retourner l'item de la pièce sans distinction. Cela permet de mieux gérer les cas où il y a plusieurs items dans une pièce et où l'utilisateur cherche des informations spécifiques sur un item en particulier. Voici getItem() :

```
public Item getItem(final String pString)
{
    return this.aItems.get(pString);
}
```

Exercice 7.22 :

Dans cet exercice, nous cherchons à avoir plusieurs items dans chaque pièce. Pour ce faire, nous utilisons une HashMap qui contiendra les items. On crée donc un nouvel attribut dans la classe Room :

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

Puis, on l'initialise dans le constructeur :

```
aItems = new HashMap <String, Item>();
```

Ensuite, on crée une procédure addItem() qui permet d'ajouter un item dans une pièce en utilisant la méthode put d'une HashMap.

```
public void addItem(final String pName, final Item pItems)
{
    this.aItems.put(pName, pItems);
}
```

On modifie getItemString() en utilisant une boucle for each afin d'ajouter plusieurs items à la String de retour.

```
public String getItemString()
{
    String vReturnItems = new String("Items : ");
    for(String vItem : this.aItems.keySet()){
        vReturnItems += vItem + "\n";
    }

    if(vReturnItems.equals("Items : ")){
        return "No item here.";
    }
    else{
        return vReturnItems;
    }
}
```

Dans la classe GameEngine, on crée 2 objets Item avec leur nom et leur poids.

```
Item vKey1 = new Item ("Key",1.0);
Item vPicture = new Item ("Picture",1.0);
```

On utilise ensuite addItem pour ajouter les items à la pièce souhaitée.

```
vGavinOffice.addItem("Key",vKey1);
vGavinOffice.addItem("Picture",vPicture);
```

Exercice 7.22.1 :

Pour accéder facilement à un objet Item, la meilleure option est d'utiliser une HashMap. Cette structure de données permet d'ajouter et de récupérer facilement des éléments, sans avoir besoin de changer la taille comme c'est le cas avec les tableaux, par exemple, lorsque l'on souhaite rajouter un item. Ainsi, il est possible de retirer des items de la HashMap afin de les placer dans un inventaire.

Exercice 7.22.2 :

Tous les items sont intégrés dans le jeu.

Exercice 7.23 :

Dans cet exercice, on souhaite ajouter une commande back permettant de revenir dans la dernière pièce où l'on était. Pour ce faire, on ajoute « back » dans la liste des commandes valides de la classe CommandWords :

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

Ensuite, on crée un attribut aLastRoom de type Room dans la classe GameEngine :

```
private Room aLastRoom;
```

On l'initialise dans la méthode goRoom() :

```
this.aLastRoom = this.aCurrentRoom;
```

Puis on crée la méthode "back" qui permet au joueur de revenir à la pièce précédente qu'il a visitée dans le jeu. La méthode prend en paramètre une commande, représentée par un objet de la classe "Command".

La première chose que la méthode fait est de vérifier si la commande a un deuxième mot. Si oui, cela signifie que le joueur a saisi un nom de pièce après la commande "back", ce qui n'est pas autorisé, donc la méthode affiche un message d'erreur et retourne.

Ensuite, la méthode vérifie si la variable "aLastRoom" est nulle. Cette variable stocke la pièce précédente, donc si elle est nulle, cela signifie que le joueur n'a pas encore visité une pièce avant la pièce actuelle. Dans ce cas, la méthode affiche un message d'erreur et retourne.

Si tout se passe bien jusqu'à présent, la méthode procède à l'échange des pièces courante et précédente. Elle stocke d'abord la pièce courante dans une variable temporaire "vCurrentRoom", puis met la pièce précédente dans la variable "aCurrentRoom" et la pièce courante dans la variable "aLastRoom". Enfin, la méthode affiche des informations sur la nouvelle pièce courante en appelant la méthode "printLocationInfo".

```
private void back(final Command pCommand)
{
    if(pCommand.hasSecondWord()){
        aGui.println("Return where ?");
        return;
    }

    if(aLastRoom == null){
        aGui.println("There is no previous room !");
        return;
    }

    Room vRoom = this.aCurrentRoom;
    this.aCurrentRoom = this.aLastRoom;
    this.aLastRoom = vRoom;

    printLocationInfo();
}
```

Enfin, on ajoute le cas où la commande est back dans la méthode interpretCommand() afin de vérifier si elle est demandé ou non.

```
case "back":
    this.back(vCommand);
    break:
```

Exercice 7.26 :

Dans cet exercice, on souhaite implémenter la commande back afin que son utilisation répétée nous ramène plusieurs pièces en arrière, et jusqu'au début du jeu si elle est utilisée assez souvent. Pour ce faire, nous devons utiliser une pile (Stack).

Tout d'abord, on importe la classe Stack dans la classe GameEngine.

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

Ensuite, on crée un attribut de type Stack qu'on initialise dans le constructeur.

```
private Stack<Room> aPreviousRooms;

/**
 * Constructor for objects of class GameEngine
 */
public GameEngine()
{
    // On crée les pièces du jeu
  this.createRooms();

  // On crée un nouveau parser pour interpréte
  this.aParser = new Parser();

  // On crée un nouveau DirectionWords pour in
  this.aDirection = new DirectionWords();
  this.aPreviousRooms = new Stack <Room> ();
}
```

La procédure goRoom() doit ainsi être modifiée en utilisant la méthode push, pour ajouter à la pile contenant la pièce courante, l'ancienne pièce.

```
    if (vNextRoom == null) {
        this.aGui.println("There is no door!");
        return;
    }else{
        this.aLastRoom = this.aCurrentRoom;
        // On met à jour la pièce courante avec la piè
        this.aCurrentRoom = vNextRoom;
        this.aPreviousRooms.push(aCurrentRoom);
        printLocationInfo();
    }
} // goRoom()
```

Nous devons ensuite modifier la méthode back(). La partie à modifier vérifie si la pile aPreviousRooms est vide et affiche un message d'erreur si c'est le cas. Si la pile n'est pas vide, elle retire la dernière salle visitée de la pile et la définit comme la nouvelle salle actuelle.

```
private void back(final Command pCommand)
{
    if(pCommand.hasSecondWord()){
        this.aGui.println("Return where ?");
        return;
    }

    if(this.aPreviousRooms.empty()){
        this.aGui.println("There is no previous room !");
        return;
    }

    this.aCurrentRoom = this.aPreviousRooms.pop();
    printLocationInfo();
}
```

Exercice 7.26.1 :

J'ai généré la javadoc sur Windows.

Voici les lignes à écrire dans l'invite de commande dans l'ordre :

cd [dossier local du projet]

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

PATH

javadoc -d userdoc -author -version *.java

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

Exercice 7.27 :

Exercice fait.

Exercice 7.28 :

Exercice fait.

Exercice 7.28.1 :

Dans cet exercice, on crée une nouvelle commande test acceptant un second mot représentant un nom de fichier (supposé se trouver dans le répertoire courant), et exécutant successivement toutes les commandes lues dans ce fichier de texte. Pour ce faire, on importe dans la classe GameEngine, les classes Scanner, File et FileNotFoundException qui nous seront utiles pour la méthode test.

```
import java.io.File;
import java.util.Scanner;
import java.io.FileNotFoundException;
```

Ensuite, on ajoute « test » à la liste des commandes valides de la classe CommandWords.

```
private static final String aValidCommands[] = {"go","quit","help","look","eat","back", "test"};
```

Puis, on ajoute la procédure test() à la classe GameEngine. La méthode test() vérifie si l'utilisateur a entré un mot après "test". Si c'est le cas, elle récupère le deuxième mot qui est supposé être le nom d'un fichier, puis elle lit le fichier ligne par ligne à l'aide de la classe Scanner. Pour chaque ligne lue, elle appelle la méthode interpretCommand(). Si le fichier n'est pas trouvé, elle envoie un message d'erreur.

```
private void test(final Command pCommand)
{
    if (!pCommand.hasSecondWord()){
        this.aGui.println("What do you want to test? (indicates a word after writing test)");
        return;
    }

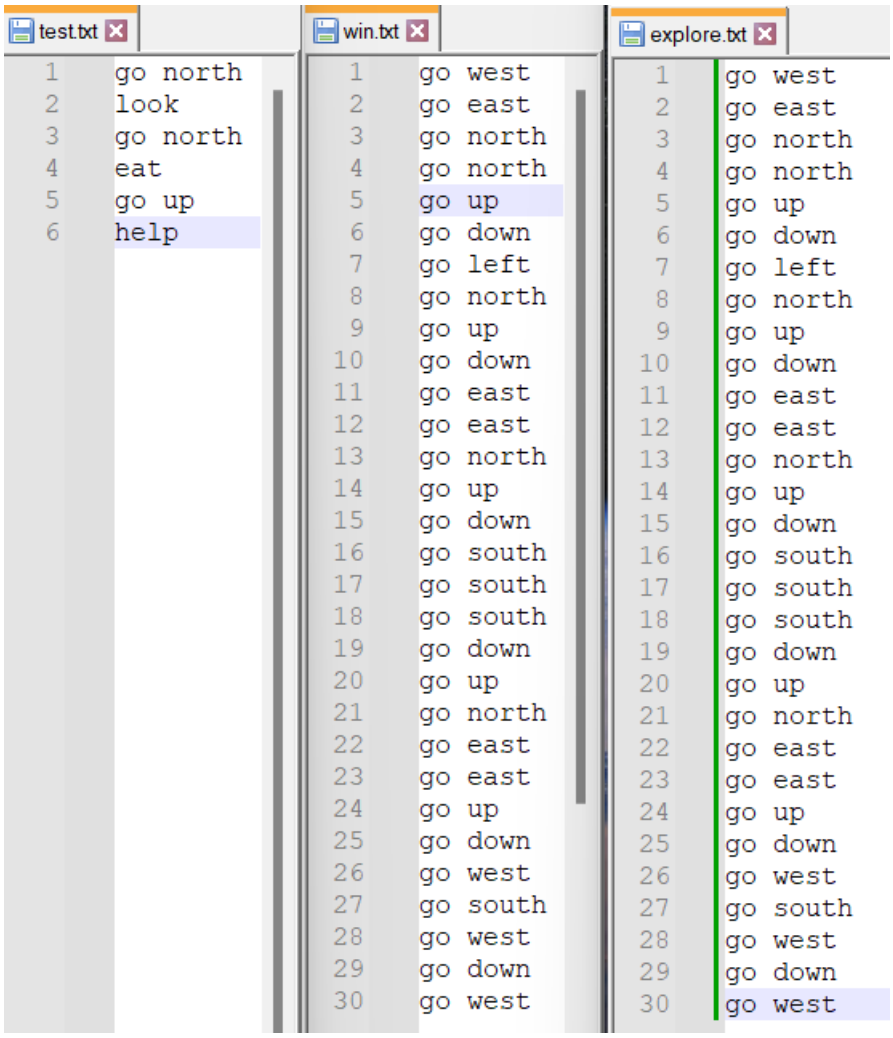
    String vFile = pCommand.getSecondWord();
    try{
        Scanner vScan = new Scanner (new File (vFile));
        while (vScan.hasNextLine()){
            interpretCommand(vScan.nextLine());
        }
    }
    catch (final FileNotFoundException pE){
        this.aGui.println("This file was not found. Try again.");
    }
}
```

Cela nous amène finalement à ajouter « test » dans la méthode interpretCommand de la classe GameEngine :

```
case "test":
    this.test(vCommand);
    break;
```

Exercice 7.28.2 :

On crée 3 fichiers de commandes : pour explorer la carte du jeu, pour gagner et pour tester simplement les commandes :



test.txt

```
go north
look
go north
eat
go up
help
```

win.txt

```
go west
go east
go north
go north
go up
go down
go left
go north
go up
go down
go east
go east
go north
go up
go down
go south
go south
go south
go down
go up
go north
go east
go east
go up
go down
go west
go south
go west
go down
go west
```

explore.txt

```
go west
go east
go north
go north
go up
go down
go left
go north
go up
go down
go east
go east
go north
go up
go down
go south
go south
go south
go down
go up
go north
go east
go east
go up
go down
go west
go south
go west
go down
go west
```

Exercice 7.29 :

Le but de cette classe Java est de créer une nouvelle classe appelée "Player" pour stocker les informations sur le joueur, y compris la pièce actuelle où se trouve le joueur et la pile des pièces précédentes visitées par le joueur. L'objectif de cette classe est de réduire la quantité de code dans la classe "GameEngine" et de rendre la gestion des déplacements du joueur plus cohérente.

La classe "Player" dispose de trois attributs : un pseudonyme pour le joueur, la pièce actuelle et la pile des pièces précédentes visitées. Ces attributs sont initialisés dans le constructeur de la classe.

```java
public class Player {

    private String aPseudo; // Pseudonyme du joueur
    private Room aCurrentRoom; // Pièce actuelle
    private Stack<Room> aPreviousRooms; // Pile des pièces précédentes

    /**
     * Constructeur naturel de la classe Player.
     *
     * @param pseudo     le pseudonyme du joueur.
     * @param currentRoom la pièce de départ du joueur.
     */
    public Player(String pseudo, Room currentRoom) {
        this.aPseudo = pseudo;
        this.aCurrentRoom = currentRoom;
        this.aPreviousRooms = new Stack<>();
    }
```

La classe Player dispose également de deux méthodes goRoom() et back() pour gérer les déplacements du joueur dans les pièces. La méthode goRoom() permet au joueur de se déplacer d'une pièce à une autre en fonction de la direction indiquée. La méthode récupère la prochaine pièce en utilisant la méthode getExit() de la pièce actuelle, ajoute la pièce actuelle à la pile des pièces précédentes, puis change la pièce courante par la pièce suivante.

```java
public void goRoom(String direction) {
    // On utilise la méthode getExit() de la pièce actuelle pour récupérer la prochaine pièce.
    Room nextRoom = this.aCurrentRoom.getExit(direction);
    // On ajoute la pièce actuelle à la pile des pièces précédentes.
    this.aPreviousRooms.push(this.aCurrentRoom);
    // On change la pièce courante par la pièce suivante.
    this.aCurrentRoom = nextRoom;
}
```

La méthode back() permet au joueur de revenir sur ses pas en se déplaçant vers la dernière pièce visitée. Elle change simplement la pièce courante par la dernière pièce visitée stockée dans la pile des pièces précédentes.

```java
public void back() {
    // On change la pièce courante par la dernière pièce visitée.
    this.aCurrentRoom = this.aPreviousRooms.pop();
}
```

Enfin, la classe Player dispose de getters pour accéder aux attributs de la classe en dehors de la classe. Le getter getPseudo() permet d'obtenir le pseudonyme du joueur, le getter getCurrentRoom() permet d'obtenir la pièce actuelle où se trouve le joueur et le getter getPreviousRooms() permet d'obtenir la pile des pièces précédentes visitées par le joueur.

```java
/**
 * Retourne le pseudonyme du joueur.
 *
 * @return le pseudonyme du joueur.
 */
public String getPseudo() {
    return this.aPseudo;
}
```

```java
/**
 * Retourne la pièce actuelle du joueur.
 *
 * @return la pièce actuelle du joueur.
 */
public Room getCurrentRoom() {
    return this.aCurrentRoom;
}
```

```java
/**
 * Retourne la pile des pièces précédentes visitées par le joueur.
 *
 * @return la pile des pièces précédentes visitées par le joueur.
 */
public Stack<Room> getPreviousRooms() {
    return this.aPreviousRooms;
}
```

La classe GameEngine devra être modifiée pour inclure la classe Player. Tout d'abord, il faut créer un attribut de type Player dans la classe GameEngine pour représenter le joueur.

```java
private Player aPlayer;
```

Dans la méthode createRooms(), il faudra alors créer un joueur avec la pièce de départ en tant que chambre.

```java
// Initialisation du jeu
this.aPlayer = new Player ("Agent Smith", vHQLivingRoom);
```

Pour que les méthodes goRoom et back fonctionnent avec cette nouvelle organisation, il faudra également les modifier en utilisant les méthodes correspondantes de l'objet Player.

```
private void goRoom(final Command pInstruction)
{
    // Vérifie si un deuxième mot est présent dans l'instruction, si
    if (!pInstruction.hasSecondWord()){
        this.aGui.println("Go where ?");
        return;
    }

    String vDirection = pInstruction.getSecondWord();// Récupère le

    // Vérifie si la direction est valide en utilisant la méthode is
    if (!CommandWords.isDirection(pInstruction.getSecondWord())) {
        this.aGui.println("Unknown direction!");
        return;
    }

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

    this.aPlayer.goRoom(vDirection);
    printLocationInfo();
} // goRoom()
```

```
private void back(final Command pCommand)
{
    // Vérifie si un deuxième mot a été fourni dans la co
    if(pCommand.hasSecondWord()){
        this.aGui.println("Return where ?");
        return;
    }

    // Vérifie si la pile des pièces précédentes est vide
    if(this.aPlayer.getPreviousRooms().empty()){
        this.aGui.println("There is no previous room !");
        return;
    }

    // Récupère la dernière pièce visitée depuis la pile
    this.aPlayer.back();
    this.printLocationInfo();
}
```

Exercice 7.30 :

L'objectif de cet exercice est de permettre à un joueur de ramasser un seul objet, plusieurs étapes doivent être suivies. Tout d'abord, on doit ajouter les commandes "take" et "drop" à la liste des commandes valides dans la classe CommandWords. Cette étape permettra au joueur de saisir ces commandes pour prendre ou déposer des objets.

```
private static final String aValidCommands[] = {"go","quit","help","look","eat","back", "test","take","drop"};
```

Ensuite, dans la classe Player, on doit ajouter une HashMap (on doit donc l'importer) pour contenir les objets dans le sac du joueur. Cette HashMap doit être initialisée dans le constructeur de la classe Player.

```
public class Player {

    private String aPseudo; // Pseudonyme du joueur
    private Room aCurrentRoom; // Pièce actuelle
    private Stack<Room> aPreviousRooms; // Pile des piè
    private HashMap <String,Item> aInventory;

    /**
     * Constructeur naturel de la classe Player.
     *
     * @param pseudo     le pseudonyme du joueur.
     * @param currentRoom la pièce de départ du joueur.
     */
    public Player(String pseudo, Room currentRoom) {
        this.aPseudo = pseudo;
        this.aCurrentRoom = currentRoom;
        this.aPreviousRooms = new Stack<>();
        this.aInventory = new HashMap<>();
    }
```

On ajoute aussi une méthode getInventory() dans la classe Player qui retournera l'item de l'inventaire :

```
public Item getInventory(final String pItem)
{
    return this.aInventory.get(pItem);
}
```

Ensuite, il faut ajouter deux méthodes dans la classe Player : "take()" et "drop()". La méthode "take()" permettra de prendre un objet de la pièce et de le placer dans le sac du joueur, tandis que la méthode "drop()" permettra de déposer un objet du sac dans la pièce. Ces méthodes doivent également mettre à jour la HashMap du sac et la liste d'objets de la pièce pour refléter les changements.

```
public void take (final String pStringItem, final Item pItem)
{
    this.aInventory.put(pStringItem,pItem);
}// take ()

public void drop (final String pStringItem)
{
    this.aInventory.remove(pStringItem);
}// drop ()
```

Dans la classe Room, on crée une procédure removeItem() qui permet de supprimer les items de l'inventaire du joueur :

```
public void removeItem(final String pName)
{
    this.aItems.remove(pName);
}
```

On ajoute ensuite les méthodes take() et drop() dans la classe GameEngine :

```
private void take(final Command pCommand)
{
    // Vérifie si la commande a un deuxième mot.
    if(!pCommand.hasSecondWord()){
        this.aGui.println("Take what ?");
        return;
    }
    // Récupère l'objet de la pièce actuelle.
    Item vItem = this.aPlayer.getCurrentRoom().getItem(pCommand.getSecondWord());
    // Si l'objet n'est pas dans la pièce, affiche un message à l'utilisateur.
    if(vItem==null){
        this.aGui.println("This item is not here.");
    }else{
        // Si l'objet est dans la pièce, le prend et le retire de la pièce.
        this.aPlayer.take(pCommand.getSecondWord(),vItem);
        this.aPlayer.getCurrentRoom().removeItem(pCommand.getSecondWord());
        this.aGui.println("You picked up " + pCommand.getSecondWord());
    }
}
```

La méthode "take" prend un objet dans la pièce où se trouve le joueur. Elle vérifie si la commande a un deuxième mot (le nom de l'objet) et si l'objet est présent dans la pièce. Si l'objet est trouvé, il est retiré de la pièce et ajouté à l'inventaire du joueur, sinon un message est affiché à l'utilisateur.

```
private void drop(final Command pCommand)
{
    // Vérifie si la commande a un deuxième mot.
    if(!pCommand.hasSecondWord()){
        this.aGui.println("Drop what ?");
        return;
    }
    // Récupère l'objet de la pièce actuelle.
    Item vItem = this.aPlayer.getInventory(pCommand.getSecondWord());
    // Si l'objet n'est pas dans l'inventaire, affiche un message à l'utilisateur.
    if(vItem==null){
        this.aGui.println("You don't have this item.");
    }else{
        // Si l'objet est dans l'inventaire, le jette et le retire de l'inventaire.
        this.aPlayer.drop(pCommand.getSecondWord());
        this.aPlayer.getCurrentRoom().addItem(pCommand.getSecondWord(), vItem);
        this.aGui.println("You droped " + pCommand.getSecondWord());
    }
}
```

La méthode "drop" permet au joueur de déposer un objet qu'il possède dans la pièce où il se trouve. Elle vérifie si la commande a un deuxième mot (le nom de l'objet) et si l'objet est présent dans l'inventaire du joueur. Si l'objet est trouvé, il est retiré de l'inventaire du joueur et ajouté à la pièce actuelle, sinon un message est affiché à l'utilisateur.

Enfin, dans la méthode interpretCommand() de la classe GameEngine, on doit ajouter la possibilité d'interpréter les commandes "take" et "drop".

```
case "take":
    this.take(vCommand);
    break;
case "drop":
    this.drop(vCommand);
    break;
```

Exercice 7.31 :

Le but de cet exercice est d'étendre l'implémentation d'un jeu existant pour permettre au joueur de transporter un nombre illimité d'objets. Cela a été fait dans l'exercice précédent. Les méthodes "take" et "drop" permettent au joueur de prendre et de déposer des objets respectivement. Elles n'ont pas de limite explicite au nombre d'objets que le joueur peut transporter. La méthode "take" ajoute l'objet à l'inventaire du joueur en utilisant la méthode "take" de l'objet Player, qui ajoute l'objet à une liste d'objets sans limite de capacité. De même, la méthode "drop" supprime l'objet de l'inventaire du joueur et l'ajoute à la pièce actuelle.

Exercice 7.31.1 :

L'objectif de cet exercice est de gérer une liste d'items dans une classe ItemList afin de supprimer la duplication de code présente dans les classes Room et Player. Cette classe ItemList aura deux attributs : aItems qui stockera les items sous forme d'une HashMap, et aLocation qui stockera l'emplacement de la liste d'items. Pour gérer cette liste, on utilisera une collection d'objets, mais elle ne sera pas manipulable depuis l'extérieur d'ItemList pour respecter les contraintes. De même, ItemList ne sera pas manipulable depuis l'extérieur de Room ou de Player. Dans le constructeur de la classe ItemList, on initialisera les deux attributs.

```
public class ItemList
{
    private HashMap <String, Item> aItems; // HashMap d'
    private String aLocation; // La où les items sont (i

    /**
     * Constructeur naturel
     *
     * @param pLocation Localisation de la liste d'items
     */
    public ItemList (final String pLocation)
    {
        this.aItems = new HashMap <String, Item> ();
        this.aLocation = pLocation;
    }// ItemList ()
```

Pour accéder aux items de la liste, on ajoutera des accesseurs getItem() qui permettront de récupérer un item à partir de sa clé, getItems() qui permettra de récupérer la liste entière sous forme de HashMap et un dernier accesseur qui renverra une chaîne de caractères des objets présents dans la pièce ou le sac, spécifiant à chaque fois ce dont il s'agit.

```
public Item getItem (final String pItem)
{
    return this.aItems.get(pItem);
}// getItem ()

/**
 * Accesseur de la liste d'items
 *
 * @return HashMap contenant tous les items
 */
public HashMap <String, Item> getItems ()
{
    return aItems;
}// getItems ()
```

Les procédures addItem() et removeItem() seront déplacées de la classe Player et Room à la classe ItemList. Dans la classe Player, on retirera les accesseurs en lien avec les items et l'attribut HashMap concernant les items sera remplacé par un attribut de type ItemList.

```
private ItemList aItemList;

this.aItemList = new ItemList ("your inventory");
```

Cela apporte donc des modifications à take(), getInventory() et drop() :

```
public Item getInventory(final String pItem)
{
    return this.aItemList.getItem(pItem);
}

public void take (final String pStringItem, final Item pItem)
{
    this.aItemList.addItem(pStringItem,pItem);
}// take ()

public void drop (final String pStringItem, final Item pItem)
{
    this.aItemList.removeItem(pStringItem,pItem);
}// drop ()
```

Dans la classe Room, on apportera les mêmes modifications en supprimant tout ce qui a été reporté dans la classe ItemList et en changeant l'attribut, l'initialisant et ajoutant un accesseur qui permet d'accéder à addItem() et removeItem() de la classe ItemList.

Exercice 7.32 :

Dans cet exercice, on souhaite ajouter une restriction qui permet au joueur de transporter des objets uniquement jusqu'à un poids maximum spécifié.

Pour ce faire, on ajoute une méthode getTotalWeight() dans la classe ItemList. On commence par initialiser la variable "vWeight" à zéro. Ensuite, on récupère toutes les clés de la map aItems et on les stocke dans un ensemble vKeys. Ensuite, on parcourt chaque élément de l'ensemble vKeys en utilisant une boucle for-each. À chaque itération de la boucle, on récupère l'élément de la map correspondant à la clé en cours, puis on ajoute le poids de cet élément à la variable vWeight. Finalement, on retourne la valeur de vWeight, qui est le poids total de tous les éléments dans la map aItems.

```
public double getTotalWeight()
{
    double vWeight=0;
    Set<String> vKeys = this.aItems.keySet();
    for(String vItem: vKeys)
    {
        vWeight+=getItem(vItem).getItemWeight();
    }
    return vWeight;
}
```

Ensuite, on ajoute les méthodes totalWeight() et canWear dans la classe Player. On déclare un attribut permettant de connaître le poids porté par le joueur et un autre indiquant le poids maximal portable par le joueur.

```
private double aWeight;
private double aMaxWeight;
```

La méthode totalWeight() calcule le poids total de tous les articles dans le sac à dos. Pour cela, on récupère le poids total de tous les articles dans la liste d'articles aItemList en appelant la méthode getTotalWeight() de cette liste. On stocke cette valeur dans la variable vTotalWeight. Ensuite, on met à jour la variable de poids aWeight avec la valeur de vTotalWeight. Finalement, on retourne la valeur de vTotalWeight, qui est le poids total de tous les articles dans le sac à dos.

La méthode canWear(Item vItem) détermine si un article peut être ajouté au sac à dos sans dépasser le poids maximum autorisé. Pour cela, on calcule le poids actuel du sac à dos en appelant la méthode totalWeight(). On stocke cette valeur dans la variable vCurrentWeight. Ensuite, on calcule le poids proposé en ajoutant le poids de l'article proposé vItem à vCurrentWeight, et on stocke cette valeur dans la variable vProposedWeight. Finalement, on retourne un booléen qui indique si l'article peut être ajouté ou non en comparant vProposedWeight avec le poids maximum autorisé aMaxWeight. Si vProposedWeight est inférieur ou égal à aMaxWeight, cela signifie que l'article peut être ajouté, donc on retourne true. Sinon, cela signifie que le poids total dépasserait le poids maximum autorisé, donc on retourne false.

```
public double totalWeight()
{
    double vTotalWeight = this.aItemList.getTotalWeight(); // Calcul
    this.aWeight = vTotalWeight; // Met à jour la variable de poids
    return vTotalWeight; // Retourne le poids total
}

/**
 * Cette méthode détermine si un article peut être ajouté au sac à d
 *
 * @param item l'article proposé à ajouter
 * @return true si l'article peut être ajouté, sinon false
 */
public boolean canWear(final Item vItem)
{
    double vCurrentWeight = totalWeight(); // Calcule le poids actue
    double vProposedWeight = vCurrentWeight + vItem.getItemWeight();
    return vProposedWeight <= this.aMaxWeight; // Retourne un boolée
}
```

Enfin, on ajoute dans la méthode take() de la classe GameEngine la condition qui bloque la commande take si la capacité de stockage du joueur est atteinte ou risque d'être dépassée:

```
else if(this.aPlayer.canWear(vItem)==false){
   this.aGui.println("Sorry, this Item is too heavy.");
```

Exercice 7.33 :

Le but de cet exercice est d'implémenter une commande d'items qui retourne tous les items actuellement transportés et leur poids total. Pour ce faire, on ajoute « inventory » dans les commandes valides dans la classe CommandWords. Ensuite, on crée la méthode getInventory() permettant de retourner la liste des items du joueur dans la classe Player.

```
public ItemList getInventory()
{
    return this.aItemList;
}
```

Enfin, on crée la méthode inventory() dans la classe GameEngine. Cette méthode Java affiche l'inventaire du joueur ainsi que son poids total et son poids maximal possible dans une interface graphique. Elle utilise les propriétés et les méthodes de l'objet aPlayer pour obtenir l'inventaire du joueur, le poids total de l'inventaire et le poids maximal possible.

La méthode getInventory() retourne l'objet Inventory qui représente l'inventaire du joueur. La méthode getItemString() de cet objet retourne une chaîne de caractères qui contient les noms et les quantités d'objets dans l'inventaire. La méthode totalWeight() retourne le poids total des objets dans l'inventaire. La méthode getMaxWeight() retourne le poids maximal que le joueur peut porter.

Toutes ces valeurs sont combinées en une chaîne de caractères avec le texte "Total weight : " et "Maximum possible weight : ", puis affichées dans l'interface graphique en utilisant la méthode println() de l'objet aGui.

```
private void inventory()
{
    this.aGui.println(this.aPlayer.getInventory().getItemString() + "Total weight : " + this.aPlayer.totalWeight()
    + "\n"+ "Maximum possible weight : " + this.aPlayer.getMaxWeight());
}
```

Exercice 7.34 :

Dans cet exercice, on souhaite intégrer un cookie magique permettant d'augmenter le poids que peut porter le joueur dans son inventaire.

Pour ce faire, on ajoute la méthode removeWeight() dans la classe Player qui permet de retirer le poids d'un objet de l'inventaire du joueur et est utilisée pour mettre à jour le poids total de l'inventaire en conséquence.

```
/**
 * Méthode pour retirer le poids d'un objet de l'inventaire du joueur.
 * @param pWeight Le poids de l'objet à retirer.
 */
public void removeWeight(final double pWeight) {
    // Soustraire le poids de l'objet du poids total de l'inventaire du joueur
    this.aWeight -= pWeight;
}
```

On peut donc ensuite modifier la méthode eat(). Cette méthode permet de consommer un cookie, augmentant ainsi la capacité d'inventaire du joueur. La méthode commence par vérifier si la commande passée en paramètre contient un deuxième mot, c'est-à-dire le nom de l'aliment à manger. Si le deuxième mot est manquant, le message "Eat what?" est affiché et la méthode se termine. Si le deuxième mot est présent mais ne correspond pas à "Cookie", le message "You don't have this food in your inventory!" est affiché et la méthode se termine.

Ensuite, la méthode vérifie si le joueur a un cookie dans son inventaire. Si le joueur ne possède pas de cookie, le message "You don't have a cookie in your inventory!" est affiché et la méthode se termine. Si le joueur possède un cookie, le poids du cookie est retiré de l'inventaire du joueur en appelant la méthode "removeWeight" de la classe Player. Ensuite, la capacité d'inventaire du joueur est augmentée de 10 unités en appelant la méthode "setMaxWeight" de la classe Player avec le nouveau poids

maximal en paramètre. Le cookie est ensuite retiré de l'inventaire du joueur en appelant la méthode "removeItem" de la classe Inventory avec le nom de l'objet en paramètre. Enfin, un message de confirmation est affiché pour informer le joueur que son inventaire a été augmenté et que le cookie a été consommé.

```
private void eat(final Command command) {
    // Vérifier si la commande contient un deuxième mot (nom de l'aliment à manger)
    if (!command.hasSecondWord()) {
        this.aGui.println("Eat what?");
        return;
    }

    // Récupérer le nom de l'aliment à manger
    final String foodName = command.getSecondWord();

    // Vérifier si l'aliment est un cookie
    if (!"Cookie".equals(foodName)) {
        this.aGui.println("You don't have this food in your inventory!");
        return;
    }

    // Vérifier si le joueur possède un cookie dans son inventaire
    final Item cookieItem = this.aPlayer.getInventory().getItem("Cookie");
    if (cookieItem == null) {
        this.aGui.println("You don't have a cookie in your inventory!");
        return;
    }

    // Retirer le poids du cookie de l'inventaire du joueur
    final double cookieWeight = cookieItem.getItemWeight();
    this.aPlayer.removeWeight(cookieWeight);
    // Augmenter la capacité d'inventaire du joueur de 10 unités
    final double newMaxWeight = this.aPlayer.getMaxWeight() + 10.0;
    this.aPlayer.setMaxWeight(newMaxWeight);
    // Retirer le cookie de l'inventaire du joueur
    this.aPlayer.getInventory().removeItem("Cookie");
    // Afficher un message de confirmation de la consommation du cookie
    this.aGui.println("You ate the magic cookie and increased your inventory capacity!");
}
```

Exercice 7.34.1 :

Fichiers tests mis à jour.

Exercice 7.34.2 :

Les 2 javadoc sont régénérée.

Exercice 7.35 :

J’ai effectué les modifications de la classe ComandWords de mon jeu en m’inspirant de zuul-with-enums-v1.

Exercice 7.35.1 :

Voici la modification de la méthode interpreteCommand() de la classe GameEngine avec les switch :

```
public void interpretCommand(final String pCommandLine) {
    // Affichage de la commande entrée
    this.aGui.println("\n" + "> " + pCommandLine );

    // Conversion de la commande en une commande utilisable par le jeu
    Command vCommand = this.aParser.getCommand(pCommandLine);

    if (vCommand.isUnknown()){
        this.aGui.println("I don't know what you mean ...");
    }else{
        // Vérification de la commande entrée et exécution de la méthode correspondante
        switch (vCommand.getCommandWord()) {
            case "go":
                this.goRoom(vCommand);
                break;
            case "look":
                this.look(vCommand);
                break;
            case "eat":
                this.eat(vCommand);
                break;
            case "quit":
                if (vCommand.hasSecondWord()) {
                    this.aGui.println("Quit what?");
                } else {
                    this.endGame();
                }
                break;
            case "help":
                this.printHelp();
                break;
```

Exercice 7.42 :

Dans cet exercice, nous devons ajouter une certaine forme de limite de temps à notre jeu. J'ai donc choisi une limite de temps mise en place en comptant le nombre de mouvements du joueur. Pour ce faire, on ajoute la fonctionnalité du comptage des déplacements du joueur dans la classe Player, le nombre de mouvements maximum du joueur sera ainsi une de ses caractéristiques. On initialise un attribut entier aTime qu'on initialise ensuite dans le constructeur naturel.

```
private int aTime;
```

```
public Player(final String pPseudo, final Room pCurrentRoom, fin
    this.aPseudo = pPseudo;
    this.aCurrentRoom = pCurrentRoom;
    this.aPreviousRooms = new Stack <Room> ();
    this.aItemList = new ItemList ("your inventory");
    this.aMaxWeight=1;
    this.aWeight=0;
    this.aTime = pTime;
}
```

Puis on ajoute un accesseur getTime() permettant d'être utilisé pour retourner les déplacements restants du joueur.

```
/**
 * Cette méthode renvoie les déplacements restants.
 * @return L'heure actuelle sous forme d'un entier.
 */
public int getTime(){
    return this.aTime;
}
```

Ensuite, on ajoute la méthode timer() dans la classe GameEngine permettant de vérifier si le joueur a encore des mouvements pour jouer. Si les mouvements sont écoulés, la partie se termine.

```
/**
 * Cette méthode vérifie si le joueur a encore des mouvements pour jouer. Si les mouvements sont écoulés, la partie se termine.
 */
private void timer()
{
    // Vérifie si les mouvements restants du joueur sont égals à zéro
    if(this.aPlayer.getTime() == 0){
        // Si les mouvements sont écoulés, affiche un message de fin de partie et termine la partie
        this.aGui.println("Game over! You have reached the maximum number of movements allowed by the IMF.");
        this.endGame();
    }
}
```

Cette méthode est ensuite appelée dans les méthodes goRoom() et back() de manière à que la condition soit vérifiées après chaque déplacement du joueur.

Grâce à l'ajout de la caractéristique time à la classe Player, on peut modifier la déclaration du joueur dans la méthode createRooms de la classe GameEngine.

```
// Initialisation du jeu
this.aPlayer = new Player ("Agent Smith", vHQLivingRoom,2);
```

Pour finir, j'ai ajouté l'information du nombre de déplacements restants dans printLocationInfo() afin de faciliter l'orientation du joueur.

```
private void printLocationInfo()
{
    this.aGui.println(this.aPlayer.getCurrentRoom().getLongDescription());
    if (this.aPlayer.getCurrentRoom().getImageName() != null){
        this.aGui.showImage(this.aPlayer.getCurrentRoom().getImageName() );
    }// if ()
    this.aGui.println("You still have "+this.aPlayer.getTime()+" mouvements.");
}//printLocationInfo()
```

Exercice 7.42.2 :

J'ai créé un bouton Inventory que j'ai positionné à gauche de l'interface, en suivant la même procédure que l'exercice 18.8.

Exercice 7.43:

Ici, nous cherchons à intégrer une trap door dans notre jeu. La pièce concernée doit avoir au moins deux sorties, mais l'une d'entre-elles ne doit être franchissable que dans un seul sens. J'ai décidé que cette pièce serait la « secret gate », elle est donc accessible par le bureau d'Adam mais on ne peut pas retourner dans ce bureau. Ensuite, on crée une méthode isExit() dans la classe Room qui vérifie si la pièce donnée en argument est une sortie de la pièce courante. Elle prend en paramètre un objet Room (pExit) et retourne un booléen. La méthode vérifie si la pièce donnée en argument est contenue dans la liste des sorties de la pièce courante (this.aExits.values()). Si la pièce est présente dans la liste, la méthode retourne true, indiquant que la pièce donnée est bien une sortie de la pièce courante. Dans le cas contraire, la méthode retourne false, indiquant que la pièce donnée n'est pas une sortie de la pièce courante.

```
public boolean isExit(final Room pExit) {
    // Vérifie si la pièce donnée en argument est contenue dans la liste des sorties de la pièce courante.
    if (this.aExits.values().contains(pExit)) {
        return true; // Si la pièce est une sortie, retourne true.
    }
    return false; // Si la pièce n'est pas une sortie, retourne false.
}
```

Puis, on modifie la méthode back() de la classe Player afin qu'elle permette au joueur de revenir sur ses pas en se déplaçant vers la dernière pièce visitée. La méthode retourne un booléen, true si le joueur peut revenir en arrière et se déplacer vers la dernière pièce visitée, false sinon. La méthode vérifie d'abord si la dernière pièce visitée est bien une sortie de la pièce courante en appelant la méthode isExit() de l'objet Room avec comme argument la dernière pièce visitée stockée dans une pile (this.aPreviousRooms.peek()). Si la pièce est une sortie, la méthode change la pièce courante par la dernière pièce visitée en utilisant la méthode pop() de la pile pour la retirer et l'assigner à la pièce courante (this.aCurrentRoom). La méthode diminue ensuite le temps restant du joueur de 1 unité et retourne true. Si la dernière pièce visitée n'est pas une sortie de la pièce courante, la méthode retourne false, indiquant que le joueur ne peut pas revenir en arrière et se déplacer vers la dernière pièce visitée.

```
public boolean back() {
    // Vérifie si la dernière pièce visitée est bien une sortie de la pièce courante.
    if (this.aCurrentRoom.isExit(this.aPreviousRooms.peek())) {
        // Change la pièce courante par la dernière pièce visitée.
        this.aCurrentRoom = this.aPreviousRooms.pop();
        // Diminue le temps restant du joueur de 1 unité.
        this.aTime -= 1;
        return true;
    }
    // La dernière pièce visitée n'est pas une sortie de la pièce courante.
    return false;
}
```

Enfin, on modifie la méthode back() de la classe GameEngine afin qu'elle puisse vérifier si le joueur ne peut pas revenir en arrière. Pour ce faire, elle essaie de retourner à la pièce précédente en appelant la méthode back() de l'objet Player (this.aPlayer.back()). Si le joueur ne peut pas revenir en arrière, la méthode affiche un message d'erreur "Error: You can't go back from a trapdoor.".

```
private void back(final Command pCommand)
{
    // Vérifie si un deuxième mot a été fourni dans la commande de retour.
    if(pCommand.hasSecondWord()){
        this.aGui.println("Return where ?");
        return;
    }

    // Vérifie si la pile des pièces précédentes est vide.
    if(this.aPlayer.getPreviousRooms().empty()){
        this.aGui.println("Error: There is no previous room to return to.");
        return;
    }

    // Essaye de retourner à la pièce précédente, sinon affiche un message d'erreur.
    if(!this.aPlayer.back()){
        this.aGui.println("Error: You can't go back from a trapdoor.");
    }

    // Affiche les informations sur la nouvelle pièce courante.
    this.printLocationInfo();
}
```

Exercice 7.43.1:

Javadoc régénérée.

Exercice 7.44:

Dans cet exercice, on souhaite ajouter un téléporteur (beamer) au jeu. Un téléporteur est un appareil qui peut être chargé et déclenché. Lorsqu'on charge le téléporteur, il mémorise la pièce actuelle. Lorsqu'on déclenche le téléporteur, il nous ramène immédiatement dans la pièce dans laquelle il a été chargé. Le téléporteur (beamer) est un Item qui doit pouvoir être ramassé dans une première pièce, peut ensuite être chargé dans une deuxième pièce, puis déclenché dans une troisième.

Pour ce faire, on crée la classe représentant le Beamer, héritant de la classe Item. Cette classe permet de créer le Beamer en stockant une référence vers une pièce du jeu. Le Beamer possède un attribut privé de type Room nommé aSavedRoom, qui représente la pièce chargée par le Beamer. Le constructeur de la classe Beamer prend en paramètre le nom de l'objet, la description de l'objet et le poids de l'objet, et appelle le constructeur de la classe mère Item avec ces mêmes paramètres. Cela permet de créer un objet Beamer avec un nom, une description et un poids.

La classe Beamer contient également une méthode getter getChargedRoom() qui permet d'accéder à l'attribut privé aSavedRoom et de récupérer la référence à la pièce chargée. La méthode setter setChargedRoom() permet de charger une pièce dans le Beamer en prenant en paramètre une référence à un objet de type Room et en affectant cette référence à l'attribut privé aSavedRoom.

La méthode isCharged() indique si le Beamer est chargé (c'est-à-dire s'il contient une référence à une pièce) ou non. Elle vérifie si l'attribut privé aSavedRoom contient une référence ou non et retourne vrai si c'est le cas, faux sinon. Cette méthode est utile pour savoir si le Beamer peut être utilisé pour projeter une pièce ou non.

```
public boolean isCharged ()
{
    return this.aSavedRoom != null;
}// isCharged ()
```

Puis, on crée deux nouvelles méthodes charge() et fire(). La première méthode charge() de la classe GameEngine permet de charger le Beamer dans l'inventaire du joueur. Elle prend en paramètre une commande pCommand qui doit avoir un deuxième mot. Si ce deuxième mot n'est pas "Beamer", un message d'erreur est affiché. Si le joueur ne possède pas le Beamer dans son inventaire, un autre message d'erreur est affiché. Si le Beamer est déjà chargé, un dernier message d'erreur est affiché. Si toutes ces conditions sont satisfaites, le Beamer est chargé, c'est-à-dire que sa référence est stockée dans le joueur.

La deuxième méthode fire() de la classe GameEngine permet de décharger avec le Beamer. Elle prend en paramètre une commande pCommand qui doit avoir un deuxième mot. Si ce deuxième mot n'est pas "Beamer", un message d'erreur est affiché. Si le joueur ne possède pas le Beamer dans son inventaire, un autre message d'erreur est affiché. Si le Beamer n'est pas chargé, un dernier message d'erreur est affiché. Si toutes ces conditions sont satisfaites, le joueur est transporté à la pièce chargée par le Beamer, c'est-à-dire que la référence de cette pièce est récupérée depuis le Beamer et le joueur est déplacé vers cette pièce. Un message indiquant que le Beamer a été déchargé est affiché, ainsi que les informations sur la nouvelle pièce dans laquelle se trouve le joueur.

```
private void charge (final Command pCommand)
{
    if (!pCommand.hasSecondWord()){
        // Vérifie si la commande saisie possède un deuxième mot
        this.aGui.println("Charge what ?");
        return;
    }// if ()

    String vSecondWord = pCommand.getSecondWord();
    if (!vSecondWord.equals("Beamer")){
        // Vérifie si le deuxième mot saisi est "beamer"
        this.aGui.println("You can't charge this !");
        return;
    }// if ()

    // Vérifie si le joueur possède un Beamer dans son inventaire
    Item vItem = this.aPlayer.getInventory().getItem(vSecondWord);
    if (vItem == null){
        this.aGui.println("You don't have the beamer in your inventory !");
        return;
    }// if ()

    Beamer vBeamer = (Beamer)vItem;
    // Vérifie si le Beamer est déjà chargé
    if (vBeamer.isCharged()){
        this.aGui.println("Your beamer is already charged !");
        return;
    }// if ()

    // Charge le Beamer
    this.aPlayer.charge(vBeamer);
    this.aGui.println("Your beamer has been charged !");
}// charge ()
```

```
private void fire (final Command pCommand)
{
    if (!pCommand.hasSecondWord()){
        // Vérifie si la commande saisie possède un deuxième mot
        this.aGui.println("Fire what ?");
        return;
    }// if ()

    String vSecondWord = pCommand.getSecondWord();
    if (!vSecondWord.equals("Beamer")){
        // Vérifie si le deuxième mot saisi est "beamer"
        this.aGui.println("You can't fire this !");
        return;
    }// if ()

    // Vérifie si le joueur possède un Beamer dans son inventaire
    Item vItem = this.aPlayer.getInventory().getItem(vSecondWord);
    if (vItem == null){
        this.aGui.println("You don't have the beamer in your inventory !");
        return;
    }// if ()

    Beamer vBeamer = (Beamer)vItem;
    // Vérifie si le Beamer est chargé
    if (!vBeamer.isCharged()){
        this.aGui.println("First you need to charge your beamer !");
        return;
    }// if ()

    // Décharger avec le Beamer
    this.aPlayer.fire(vBeamer);
    this.aGui.println("Your beamer has been fired !");
    printLocationInfo();
```

Ensuite, dans interpretCommand () de la classe GameEngine, on ajoute les commandes charge et fire de manière à qu'elles soient reconnues.

```
case "charge":
    this.charge(vCommand);
    break;
case "fire":
    this.fire(vCommand);
    break;
```

Enfin, on ajoute les méthodes charge() et fire() dans la classe Player car celle-ci représente le joueur qui se déplace dans un jeu à travers des pièces.

La méthode charge(Beamer pBeamer) permet de charger le Beamer. Pour cela, elle sauvegarde la pièce actuelle comme destination pour le Beamer en appelant la méthode setChargedRoom() de la classe Beamer. Elle diminue également le temps restant du joueur d'une unité en diminuant la valeur de la variable aTime.

Quant à la méthode fire(Beamer pBeamer), elle permet d'utiliser le Beamer pour se téléporter. Tout d'abord, elle ajoute la pièce actuelle à la pile de pièces précédentes en appelant la méthode push() de la classe Stack. Ensuite, elle change la pièce actuelle pour la pièce de destination du Beamer en appelant la méthode getChargedRoom() de la classe Beamer. Elle retire également le Beamer de l'inventaire du joueur en appelant la méthode removeItem() de la classe ItemList, ce qui diminue le poids de l'inventaire en conséquence en diminuant la valeur de la variable aWeight. Enfin, elle diminue le temps restant du joueur d'une unité en diminuant la valeur de la variable aTime.

```
public void charge(final Beamer pBeamer) {
    // Sauvegarde la pièce actuelle comme destination pour le Beamer
    pBeamer.setChargedRoom(this.aCurrentRoom);
    // Décrémente le temps restant
    this.aTime -= 1;
}

/**
 * Téléporte le joueur à la pièce où le Beamer a été chargé.
 * Ajoute également la pièce actuelle à la pile de pièces précédentes.
 * Retire le Beamer de l'inventaire du joueur et diminue le poids de l
 * Diminue également le temps restant d'une unité.
 *
 * @param pBeamer Le Beamer à utiliser pour la téléportation.
 */
public void fire(final Beamer pBeamer) {
    // Ajoute la pièce actuelle à la pile de pièces précédentes
    this.aPreviousRooms.push(this.aCurrentRoom);
    // Change la pièce actuelle pour la pièce de destination du Beamer
    this.aCurrentRoom = pBeamer.getChargedRoom();
    // Retire le Beamer de l'inventaire du joueur
    this.aItemList.removeItem(pBeamer.getItemName());
    // Diminue le poids de l'inventaire en conséquence
    this.aWeight -= pBeamer.getItemWeight();
    // Décrémente le temps restant
    this.aTime -= 1;
}
```

Exercice 7.45:

Ici, on souhaite ajouter des portes verrouillées au jeu. Le joueur doit trouver une clé pour ouvrir une porte. Une pièce peut avoir certaines de ses portes ouvertes, et d'autres fermées. Une clé doit être prise par le joueur avant de pouvoir ouvrir ou fermer une porte. Elle fonctionne des 2 côtés de la porte.

Dans le cas de mon jeu, j'ajoute une porte fermée par 3 clés que le joueur devra trouver.

Pour ce faire, on ajoute une classe Door qui représente une porte avec des clés nécessaires pour l'ouvrir. Elle permet de définir l'état de verrouillage de la porte, de spécifier les clés nécessaires, de récupérer les clés nécessaires et de tenter de déverrouiller la porte en fonction des clés disponibles dans un inventaire donné. La porte a les attributs suivants : un tableau d'objets « aKeys » de type Item

(élément). Cela représente les clés nécessaires pour ouvrir la porte. Et un booléen « aLocked » qui indique si la porte est verrouillée ou non.

Le constructeur de la classe Door initialise la variable locked à false, ce qui signifie que la porte est initialement déverrouillée.

La méthode setNecessaryKeys permet de définir les clés nécessaires pour ouvrir la porte en prenant en paramètre un tableau d'objets de type "Item".

La méthode getKeys renvoie le tableau de clés nécessaires pour ouvrir la porte.

La méthode isLocked renvoie un booléen qui indique si la porte est verrouillée ou non.

La méthode setLocked permet de définir l'état de verrouillage de la porte en prenant en paramètre un booléen.

La méthode tryToUnlock tente de déverrouiller la porte en prenant en paramètre une HashMap nommée pInventory, qui associe des clés (chaînes de caractères) à des objets de type "Item". Cette méthode parcourt toutes les clés nécessaires pour ouvrir la porte (keys) et vérifie si chaque clé est présente dans l'inventaire (pInventory). Si toutes les clés nécessaires sont présentes, la porte est déverrouillée (locked est défini sur false) et la méthode renvoie true. Sinon, si une ou plusieurs clés nécessaires sont manquantes, la méthode renvoie false.

```
public boolean tryToUnlock(HashMap<String, Item> pInventory) {
    for(Item key : keys) {
        boolean isCorrect = false;
        for(Item inventoryKey : pInventory.values()) {
            if(key == inventoryKey) isCorrect = true;
        }

        if(!isCorrect) return false;
    }

    this.locked = false;
    return true;
}
```

Puis, on ajoute un attribut aDoor de type Door dans la classe Room afin que la porte devienne une caractéristique des pièces. On l'initialise dans le constructeur par un paramètre pDoor, puis on crée une méthode getDoor() qui retourne les portes des pièces.

```
public Door getDoor() {
    return this.aDoor;
}
```

Ensuite, on configure une porte dans le bureau de "Adam" dans la méthode createRooms () de la classe GameEngine. On ajoute une première ligne qui indique que la sortie "down" du bureau de "Adam" possède une porte. On spécifie ensuite que cette porte nécessite trois clés spécifiques représentées par les objets vKey1, vKey2 et vKey3. Puis on ajoute une deuxième ligne qui indique que la porte est verrouillée en la définissant sur l'état "true".

```
vAdamOffice.getExit("down").getDoor().setNecessaryKeys(
    new Item[] {vKey1, vKey2, vKey3});
vAdamOffice.getExit("down").getDoor().setLocked(true);
```

Désormais, il faut que le jeu retourne un message d'erruer lorque le joueur essaye de franchir la porte fermée sans avoir les clés et un message de suucès en cas de franchissement de porte. Pour ce faire, on ajoute cette boucle dans goRoom() de la classe GameEngine. Cette boucle doit vérifier si la porte de la prochaine pièce (représentée par l'objet nextRoom) est verrouillée. Si la porte est effectivement

verrouillée, la méthode tryToUnlock est appelée sur cette porte en utilisant les objets contenus dans l'inventaire du joueur (this.aPlayer.getInventory().getItems()).

Si la tentative de déverrouillage réussit (c'est-à-dire que toutes les clés nécessaires sont présentes dans l'inventaire du joueur), le message "The door has been unlocked" est affiché via l'objet this.aGui.

Dans le cas où la tentative de déverrouillage échoue (c'est-à-dire qu'au moins l'une des clés nécessaires est manquante), le message "You can't pass this door! It's locked by 3 Keys!" est affiché via this.aGui, et la fonction se termine immédiatement avec un return, empêchant ainsi le joueur de passer à travers la porte.

Exercice 7.45.1:

Fichiers test mis à jour.

Exercice 7.45.2:

Javadoc régénérée.

Exercice 7.46:

Dans cet exercice, on souhaite ajouter une salle de téléportation. Chaque fois que le joueur entre dans cette pièce, il est transporté au hasard dans l'une des autres pièces.

Pour ce faire, on crée la classe RoomRandomizer qui sert à sélectionner aléatoirement une pièce parmi une liste de pièces. Elle utilise un objet Random pour générer un nombre aléatoire entre 0 et le nombre de pièces dans la liste, et renvoie la pièce correspondante dans la liste. Pour la créer, l'import de trois classes est effectué : List, ArrayList et Random. La classe List et la classe ArrayList sont des interfaces et des implémentations de listes respectivement, qui permettent de stocker des éléments dans une séquence ordonnée. La classe Random permet de générer des nombres aléatoires. Ensuite, la classe RoomRandomizer est définie avec une liste de pièces et un objet Random comme attributs. Le constructeur de la classe RoomRandomizer est défini pour initialiser la liste des pièces avec une liste vide et l'objet Random. La méthode addRoom est définie pour ajouter une pièce à la liste des pièces disponibles pour la sélection aléatoire. Elle prend en entrée un objet Room et ajoute cet objet à la liste. La méthode getRandomRoom est définie pour sélectionner une pièce aléatoire parmi les pièces de la liste. Elle génère un nombre aléatoire entre 0 et le nombre de pièces dans la liste, puis renvoie la pièce correspondante dans la liste.

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Random;

/**
 * La classe RoomRandomizer est utilisée pour sélectionner aléatoirement une pièce parmi une liste de pièces.
 * Elle utilise un objet Random pour fonctionner.
 *
 * @author K.FELTRIN
 * @version 30/04/2023
 */
public class RoomRandomizer {

    // La liste des pièces à choisir aléatoirement
    private List<Room> aRooms;

    // L'objet Random utilisé pour sélectionner aléatoirement une pièce
    private Random aRandom;

    /**
     * Construit un nouvel objet RoomRandomizer avec une liste de pièces vide et un nouvel objet Random.
     */
    public RoomRandomizer() {
        // Initialisation de la liste des pièces avec une liste vide
        this.aRooms = new ArrayList<Room> ();

        // Initialisation de l'objet Random
        this.aRandom = new Random ();
    }// RoomRandomizer ()

    /**
     * Ajoute une pièce à la liste des pièces pouvant être choisies aléatoirement.
     *
     * @param pRoom La pièce à ajouter à la liste
     */
    public void addRoom(final Room pRoom) {
        // Ajout de la pièce à la liste
        this.aRooms.add(pRoom);
    }// addRoom ()

    /**
     * Renvoie une pièce choisie aléatoirement dans la liste des pièces.
     *
     * @return Une pièce choisie aléatoirement
     */
    public Room getRandomRoom() {
        // Génère un nombre aléatoire entre 0 et le nombre de pièces dans la liste
        int vRandom = this.aRandom.nextInt(this.aRooms.size());

        // Renvoie la pièce correspondante dans la liste
        return this.aRooms.get(vRandom);
    }// getRandomRoom ()
}// RoomRandomizer
```

Ensuite, on crée la classe TransporterRoom qui hérite de la classe Room. Cette classe représente une pièce dans le jeu qui transporte aléatoirement le joueur vers une autre pièce lorsqu'il tente de partir. Elle utilise un objet RoomRandomizer pour sélectionner aléatoirement la pièce suivante. Dans le constructeur de la classe TransporterRoom, on appelle le constructeur de la classe Room avec les paramètres pDescription et pImageName pour initialiser la description et le nom de l'image associée à la pièce. On crée ensuite un nouvel objet RoomRandomizer et on l'assigne à la variable aRoomRandomizer. La méthode getExit de la classe TransporterRoom remplace la méthode getExit de la classe Room. Cette méthode renvoie une pièce aléatoire en tant que sortie pour la direction donnée en utilisant l'objet RoomRandomizer de la pièce. Enfin, la méthode getRoomRandomizer renvoie l'objet RoomRandomizer utilisé par la pièce TransporterRoom pour sélectionner aléatoirement la pièce suivante.

```java
public class TransporterRoom extends Room {

    // L'objet RoomRandomizer utilisé pour sélectionner la pièce suivante
    private RoomRandomizer aRoomRandomizer;

    /**
     * Construit un objet TransporterRoom avec la description et le nom d'image donnés.
     *
     * @param pDescription La description de la pièce
     * @param pImageName Le nom de l'image associée à la pièce
     */
    public TransporterRoom(final String pDescription, final String pImageName) {
        super(pDescription, pImageName);
        this.aRoomRandomizer = new RoomRandomizer();
    }// TransporterRoom ()

    /**
     * Renvoie une pièce aléatoire en tant que sortie pour la direction donnée.
     * Remplace la méthode getExit de la classe Room.
     *
     * @param pDirection La direction dans laquelle le joueur souhaite quitter la pièce
     * @return Un objet pièce sélectionné aléatoirement
     */
    @Override
    public Room getExit(final String pDirection) {
        return this.aRoomRandomizer.getRandomRoom();
    }// getExit ()

    /**
     * Renvoie l'objet RoomRandomizer utilisé par la pièce TransporterRoom pour sélectionner aléatoirement la pièce suivante.
     *
     * @return L'objet RoomRandomizer
     */
    public RoomRandomizer getRoomRandomizer() {
        return this.aRoomRandomizer;
    }// getRoomRandomizer ()
}// TransporterRoom
```

Puis, on crée un nouvel objet de la classe TransporterRoom dans la méthode createRooms() de la classe GameEngine. Cet objet est nommé vTransporterRoom. La méthode TransporterRoom utilisée pour créer cet objet prend trois arguments : une chaîne de caractères qui décrit la pièce, une autre chaîne de caractères qui donne le nom de l'image associée à la pièce et une troisième chaîne de caractères qui indique les directions disponibles pour quitter la pièce. La chaîne de caractères "You have entered the teleportation room. This room has the ability to randomly teleport you to different locations." + "\n" + "To activate the teleportation, simply leave the room." décrit la pièce en question, expliquant qu'il s'agit d'une pièce de téléportation qui peut téléporter le joueur de manière aléatoire et comment l'activer. Le caractère spécial \n est utilisé pour insérer une nouvelle ligne dans la description. La chaîne de caractères "Images/transporter_room.png" est le nom de l'image associée à cette pièce.

```java
TransporterRoom vTransporterRoom = new TransporterRoom ("You have entered the teleportation room. This room has the ability to randomly teleport yo
```

J’ajoute donc une sortie dans mon jeu qui mène ver cette pièce :

```java
vPaulAndAdamStreet.setExit("north",vTransporterRoom);
```

On ajoute ensuite dans la méthode createRooms() de la classe GameEngine des objets pièce à l'objet RoomRandomizer de la pièce TransporterRoom. Cela signifie que lorsque le joueur quitte la pièce TransporterRoom, il sera choisi au hasard l'une de ces pièces ajoutées comme la pièce suivante à laquelle le joueur sera téléporté. Par exemple, la première ligne "vTransporterRoom.getRoomRandomizer().addRoom(vHQLivingRoom);" ajoute l'objet pièce vHQLivingRoom à l'objet RoomRandomizer de la pièce vTransporterRoom. Lorsque le joueur quittera la pièce vTransporterRoom, il sera choisi au hasard entre toutes les pièces ajoutées à l'objet

RoomRandomizer, y compris vHQLivingRoom, comme la pièce suivante vers laquelle le joueur sera téléporté.

```
vTransporterRoom.getRoomRandomizer().addRoom(vHQLivingRoom);
vTransporterRoom.getRoomRandomizer().addRoom(vHQOffice);
vTransporterRoom.getRoomRandomizer().addRoom(vHQAndGavinStreet);
vTransporterRoom.getRoomRandomizer().addRoom(vGavinLivingRoom);
vTransporterRoom.getRoomRandomizer().addRoom(vGavinOffice);
vTransporterRoom.getRoomRandomizer().addRoom(vCraigStreet);
vTransporterRoom.getRoomRandomizer().addRoom(vCraigLivingRoom);
vTransporterRoom.getRoomRandomizer().addRoom(vCraigOffice);
vTransporterRoom.getRoomRandomizer().addRoom(vLenStreet);
vTransporterRoom.getRoomRandomizer().addRoom(vLenLivingRoom);
vTransporterRoom.getRoomRandomizer().addRoom(vLenOffice);
vTransporterRoom.getRoomRandomizer().addRoom(vHalStreet);
vTransporterRoom.getRoomRandomizer().addRoom(vHalLivingRoom);
vTransporterRoom.getRoomRandomizer().addRoom(vHalOffice);
vTransporterRoom.getRoomRandomizer().addRoom(vPaulAndAdamStreet);
vTransporterRoom.getRoomRandomizer().addRoom(vPaulLivingRoom);
vTransporterRoom.getRoomRandomizer().addRoom(vPaulOffice);
```

Exercice 7.46.1:

Pas fait car manque de temps.

Exercice 7.46.2:

Il est inutile de modifier ou ajouter des héritages.

Exercice 7.46.3:

Javadoc modifiée.

Exercice 7.46.4:

Javadoc générée.

Ajout de fonctionnalités :

Afin que le jeu soit gagnable et perdable de manière à correspondre au scénario, on ajoute des fonctionnalités.

Tout d’abord, on ajoute une méthode Strop Bitcoin qui permet de gagner.

La méthode stopBTC() de la classe Game vérifie si le joueur se trouve dans le QG de Satoshi Nakamoto. Si le joueur est dans cette pièce, un message de félicitations est affiché indiquant qu'il a arrêté Bitcoin et sauvé l'économie mondiale. Une image de victoire est également affichée. Ensuite, la méthode endGame() est appelée pour terminer le jeu. Si le joueur n'est pas dans la pièce "Satoshi Nakamoto's HQ", un message est affiché dans l'interface graphique indiquant que le joueur n'est pas dans la bonne pièce. Aucune autre action n'est effectuée. Voici la méthode stopBTC.

```
private void stopBTC()
{
    Room vCurrentRoom = this.aPlayer.getCurrentRoom();

    if(vCurrentRoom == aRooms.get("Satoshi Nakamoto's HQ") ){
        this.aGui.println("\n");
        this.aGui.println("Congratulations!" +"\n"+ "You stopped Bitcoin and saved the world economy!" +"\n"+"The IMF thanks you!");
        this.aGui.showImage("Images/win.png");
        this.endGame();
    } else{ this.aGui.println("You are not in Satoshi Nakamoto's HQ!");
    }
}
```

Cette méthode prendra la forme d'un bouton dans le jeu. C'est ainsi qu'il est ajouté dans la classe UserInterface.

Ensuite, on ajoute une image Game Over qui apparaît lorsque le joueur a dépassé le nombre de mouvements possibles. Pour ce faire, on ajoute l'affichage de l'image dans la méthode timer() de la classe GameEngine.

```
private void timer()
{
    // Vérifie si les mouvements restants du joueur sont égals à zéro
    if(this.aPlayer.getTime() == 0){
        // Si les mouvements sont écoulés, affiche un message de fin de partie et termine la partie
        this.aGui.println("Game over! You have reached the maximum number of movements allowed by the IMF.");
        this.aGui.showImage("Images/lost.png");
        this.endGame();
    }
}
```

III. Mode d'emploi (si nécessaire, instructions d'installation ou pour démarrer le jeu)

Commandes du jeu :

- eat : manger.
- back : revenir en arrière.
- take : prendre item.
- drop : lâcher item.
- inventory : accéder à son inventaire.
- charge Beamer : charger le téléporteur.
- fire Beamer : utiliser le téléporteur.
- stopBTC : arrêter le Bitcoin dans le QG de Satoshi Nakamoto.

IV. Déclaration obligatoire anti-plagiat

Scénario inspiré de faits réels. Voici les articles servant de base au jeu :

- https://www.forbes.com/advisor/investing/cryptocurrency/who-is-satoshi-nakamoto/
- https://www.investopedia.com/tech/three-people-who-were-supposedly-bitcoin-founder-satoshi-nakamoto/
- https://cointelegraph.com/learn/who-is-satoshi-nakamoto-the-creator-of-bitcoin
- https://cointelegraph.com/news/7-people-who-could-be-bitcoin-creator-satoshi-nakamoto

Images provenant de Pinterest, Dalle-2 et deco.fr.

Utilisation des fichiers et aides fournies par Monsieur Denis Bureau et le forum de l'unité.

Utilisation d'une ressource sur le GridLayout pour les boutons, du site https://perso.telecom-paristech.fr/hudry/coursJava/interSwing/grilleSimple.html.

Je déclare qu'il n'y a aucun plagiat.