Rapport de Zuul JAVA: Wonders of the Broken Gate

I.A) Kuch William

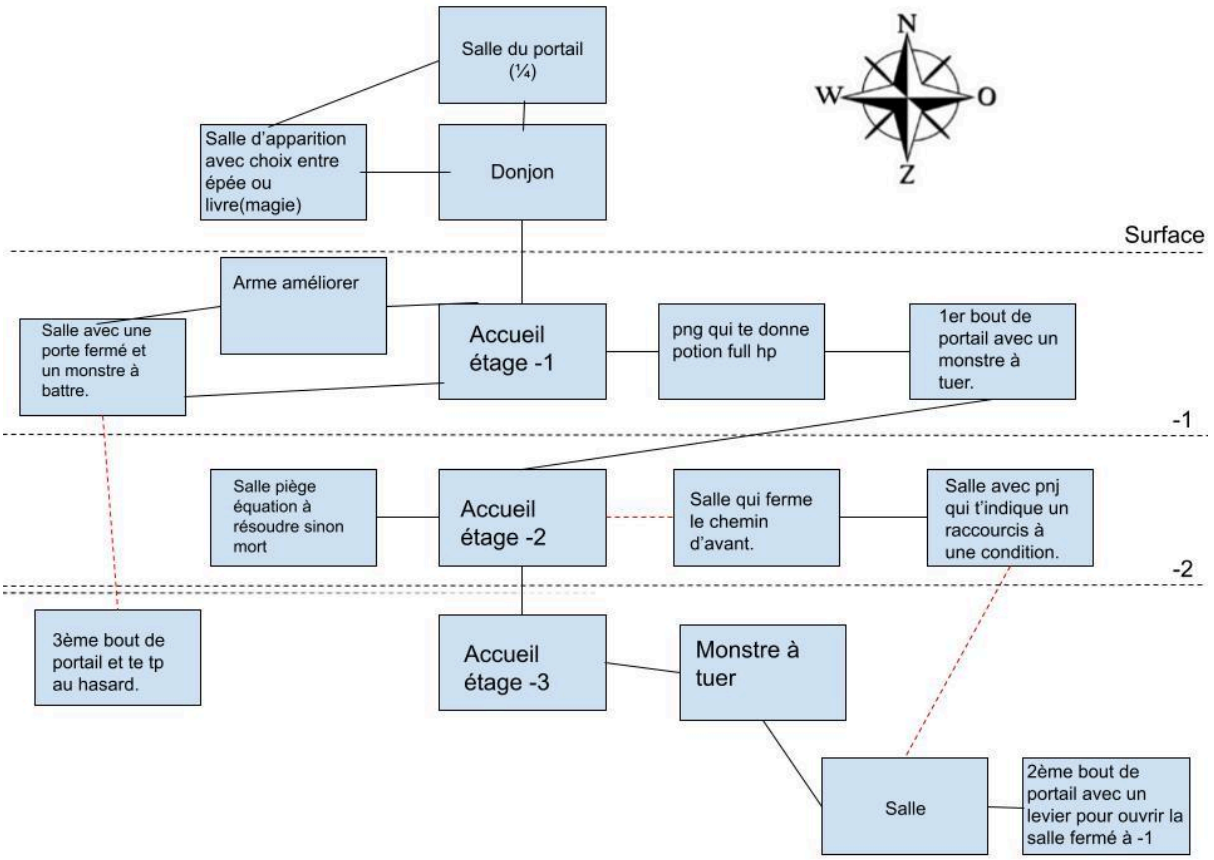
I.B) Dans un monde fantastique: Un ingénieur doit trouver les 3 parties d'un portail pour rentrer.

I.C)Vous êtes un jeune ingénieur se réveillant dans un monde fantastique après avoir été percuté par une voiture.

En faisant des recherches sur ce monde, il tombe sur une histoire.L'histoire d'un "humain venu d'ailleurs" ayant construit un portail pour exploiter Terra(le monde fantastique)mais après avoir construit le portail reliant la Terra à Terra il se fit battre par les terrarias(race de ce nouveau monde).Après cela les terrarias divisèrent le portail en 4 et les éparpillèrent dans un donjon.

L'ingénieur devra alors aller découvrir le donjon pour trouver les 4 parties du portail et les réunir.

I.D)



I.E)Vous êtes un jeune ingénieur se réveillant dans un monde fantastique après avoir été percuté par une voiture.
En faisant des recherches sur ce monde, il tombe sur une histoire. L'histoire d'un "humain venu d'ailleurs" ayant construit un portail pour exploiter Terra(le monde fantastique)mais après avoir construit le portail reliant la Terra à Terra il se fit battre par les terrarias(race de ce nouveau monde).Après cela les terrarias divisèrent le portail en 4 et les éparpillèrent dans un donjon.
L'ingénieur devra alors aller découvrir le donjon pour trouver les 4 parties du portail et les réunir.
Avant d'entrer dans le donjon vous devrez choisir entre la classe épéiste ou magicien. Puis entrant dans le donjon vous devrez parcourir le donjon en tuant des monstres, résoudre des énigmes et trouver le bon chemin.

I.F) Les lieux importants sont :

- La première salle du portail car c'est la salle qui contient le socle du portail
- L'armurerie car elle permet d'améliorer son équipements
- La salle avec une porte fermée et un monstre à battre car après avoir tué le monstre, on peut ouvrir la porte qui mène à un bout de portail
- Une salle où l'on meurt si on ne résout pas une énigme
- Une salle à sens unique après être entrée dedans
- Une salle avec un personnage qui peut te donner un raccourcis vers une autre salle
- La salle du dernier bout de portail qui te téléporte après avoir pris le portail

Les personnages sont:

- Adrien l'alchimiste qui peut te donner une potion qui te rend tes points de vie
- Albert qui peut te donner un raccourcis dans le donjon

I.G) La situation gagnante est lorsque qu'on réunit les 3 bouts de portail à la première salle.

Les situations perdantes sont:

- Mourir face à un monstre
- Rester piégé dans une salle

I.H) énigmes à résoudre et combat contre monstre.

II)

7.5)On crée une procédure printLocationInfo() qui permet d'afficher les sorties. Cette procédure empêche la duplication de code dans goRoom() et printWelcome() en l'appelant.

7.6)Afin d'éviter le couplage et de simplifier le code de goRoom(), une fonction getExit() est créée. Elle permet de retourner la salle de la direction passée en paramètre.

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

7.7)La fonction getExitString() permet de stocker les sorties possibles d'une sans les afficher.

```
public String getExitString()
{
 String vExits="Exits :";
 if(this.getExit("north" !=null)
    {
    vExits += "north ";
    }
 if(this.getExit("south" !=null)
    {
    vExits += "south ";
    }
 if(this.getExit("west" !=null)
    {
    vExits += "west ";
    }
 if(this.getExit("east" !=null)
    {
    vExits += "east ";
    }
 return vExits;
}
```

Puis pour éviter la duplication de code nous utilisons la fonction getExitString() dans la fonction printLocationInfo().

```
private void printLocationInfo()
{
    //System.out.println(this.aCurrentRoom.getLongDescription());
    //System.out.println("You are " + this.aCurrentRoom.getDescription())
    System.out.println(this.aCurrentRoom.getExitString());
    /*System.out.println("Exits: ");
```

7.8)Dans la classe Room, on importe HashMap.

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

/**
 * Classe Room - un lieu du jeu d'aventure Zuul
 * @Kuch
 */
public class Room
{
 private String aDescription;
 private HashMap<String, Room> aExits;
```

HashMap permet de créer une sorte de bibliothèque comme sur python qui associe une clé avec une valeur. On peut alors associer nord sud ouest est bas haut aux salles que l'on veut. HashMap nous permet d'enlever les attributs tels que aNorthexit etc... On remarque alors que setExits() n'est plus d'aucune utilité, on l'a remplace alors par setExit() et il faut aussi modifier getExit().

```
/**
 * Définit les sorties de la pièce selectioné.
 * Chaque direction conduit à une autre pièce ou null(pas de sortie dans cette direction).
 * @param pDirection est la clé permettant de trouver la salle.
 *        pVoisin est la salle voisine à la salle sélectioné.
 */
public void setExit(final String pDirection,final Room pVoisin)
{
    this.aExits.put(pDirection,pVoisin);
}//setExits()
```

Or pour modifier getExit() il nous faut utiliser set. Grâce à set on peut avoir un tableau des clés de aExits.

```
/**
 * Permet d'avoir la salle associé à la direction demander.
 * @param pDirection est la clé liée à une salle. Il est une direction(exemple: "sud").
 * @return renvoie la pièce atteinte si nous nous déplaçcons
 * dans la direction "pDirection". S'il n'y a pas de pas de pièce
 * dans cette direction, renvoie une room UNKNOWN qui a comme description "Unknow direction!"
 */
public Room getExit(final String pDirection)
{
   Set<String> vVoisin = this.aExits.keySet();
   for(String vDirection: vVoisin){
       if(vDirection .equals(pDirection)){
           return this.aExits.get(pDirection);
       }
   }
   return aUNKNOWN_DIRECTION;
}//getExit()
```

Set<String> permet de créer un tableau qui prend des String en valeur.
for(String vDirection : vVoisin) permet à vDirection un string de prendre pour valeur les valeurs du tableau vVoisin un par un.

7.8.1) On modifie createRoom() en ajoutant up et down.

```
//Positionnement des salles
vDepart.setExit("east",vEntreDonjon);
vDepart.setExit("north-east",vPortail);

vPortail.setExit("south-west",vDepart);
vPortail.setExit("south",vEntreDonjon);

vEntreDonjon.setExit("west",vDepart);
vEntreDonjon.setExit("north",vPortail);
vEntreDonjon.setExit("down",vAccueil1);

vAccueil1.setExit("up",vEntreDonjon);
vAccueil1.setExit("north-west",vArmurerie);
vAccueil1.setExit("west",vSallePorteF);
vAccueil1.setExit("east",vSallePnj);

vArmurerie.setExit("south-east",vAccueil1);
vArmurerie.setExit("south-west",vSallePorteF);

vSallePorteF.setExit("north-east",vArmurerie);
vSallePorteF.setExit("east",vAccueil1);

vSallePnj.setExit("west",vAccueil1);
vSallePnj.setExit("east",vPortail1);

vPortail1.setExit("west",vSallePnj);
vPortail1.setExit("down",vAccueil2);

vAccueil2.setExit("up",vPortail1);
vAccueil2.setExit("west",vPiege);
```

7.9)On modifie getExitString() en utilisant la même méthode que dans getExit().

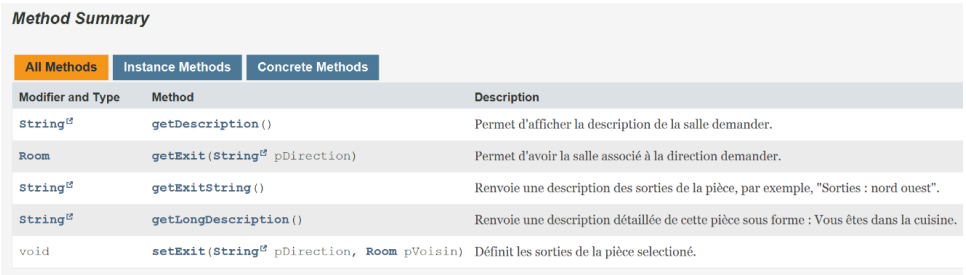
```
/**
    * Renvoie une description des sorties de la
    * pièce, par exemple, "Sorties : nord ouest".
    * @return Une description des sorties possibles.
    */
   public String getExitString()
   {
      String vSortie = "Sorties :";
      Set<String> vKeys = this.aExits.keySet();
      for(String exit :vKeys){
          vSortie +=" "+exit;
      }
      return vSortie;
   }//getExitString()
```

7.10.1) JavaDoc complété.

7.10.2) Après avoir généré la javadoc, on remarque que la classe contient beaucoup moins de méthodes que celle de la classe Room car game ne contient qu'une seule méthode public contrairement à Room qui en a plusieurs public.

Constructors

Constructor	Description
`Game()`	Constructeur par défaut de game,il permet de créer les salles et de lire le clavier.



Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`String`	`getDescription()`	Permet d'afficher la description de la salle demander.
`Room`	`getExit(String pDirection)`	Permet d'avoir la salle associé à la direction demander.
`String`	`getExitString()`	Renvoie une description des sorties de la pièce, par exemple, "Sorties : nord ouest".
`String`	`getLongDescription()`	Renvoie une description détaillée de cette pièce sous forme : Vous êtes dans la cuisine.
`void`	`setExit(String pDirection, Room pVoisin)`	Définit les sorties de la pièce selectioné.

7.11)Pour éviter du couplage, on crée la fonction getLongDescription().

```java
/**
 * Renvoie une description détaillée de cette pièce sous forme :
 *     Vous êtes dans la cuisine.
 *     Sorties : nord ouest
 * @return Une description de la pièce, avec les sorties.
 */
public String getLongDescription()
{
    return "You're " + this.aDescription + "\n" + this.getExitString();
}
```

7.14/15)On veut créer une nouvelle commande, il faut alors changer la taille du tableau contenant les noms des commandes.

```java
// a constant array that will hold all valid command words
private final String[] aValidCommands;

/**
 * Constructor - initialise the command words.
 */
public CommandWords()
{
    this.aValidCommands = new String[6];
    this.aValidCommands[0] = "go";
    this.aValidCommands[1] = "help";
    this.aValidCommands[2] = "quit";
    this.aValidCommands[3] = "look";
    this.aValidCommands[4] = "eat";
    this.aValidCommands[5] = "weapon";
} // CommandWords()
```

Après cela on peut ajouter la commande eat et look.

```
private void look(final Command pCommandLook)
{
    if(pCommandLook.getSecondWord()!=null)
    {
        System.out.println("I don't know how to look at something in particular yet.");
    }
    else{
        System.out.println(this.aCurrentRoom.getLongDescription());
    }
}

private void eat()
{
    System.out.println("You have eaten now and you are not hungry any more.");
}
```

```
else if(pProcessCommand.getCommandWord().equals("look"))
{
    this.look(pProcessCommand);
    return false;
}
else if(pProcessCommand.getCommandWord().equals("eat"))
{
    this.eat();
    return false;
}
```

7.16)On crée la méthode showAll() qui permet d'afficher toutes les commandes sans avoir à ajouter les nouvelles commandes à la main dans help.

```
public void showAll()
{
    for(String vCommand : aValidCommands){
        System.out.print(vCommand+' ');
    }
    System.out.println();
}
```

Cependant on ne peut pas appeler showAll() dans la classe game car ils sont pas couplé mais ceci est bien car on peut directement passer par parser qui lui est couplé à la classe CommandWords. On crée la méthode showCommands() pour que la classe game affiche les commandes dans printHelp().

```java
public void showCommands()
{
    this.aValidCommands.showAll();
}
```

```java
/***
 * Permet d'afficher un message d'aide.
 */
private void printHelp()
{
    System.out.println("You are lost. You are alone."+
    "You wander around at the university."+
    "\n"+
    "Your command words are:");
    aParser.showCommands();
}//printHelp()
```

7.18)

Pour être un bon concepteur logiciel, il faut pouvoir prévoir les évolutions futures tel qu'un ajout d'une interface graphique. On ne voudra donc plus que les informations s'affichent sur le terminal mais plutôt dans le champ de texte d'une fenêtre graphique. Pour cela il faudra essayer de garder toutes les informations de l'interface utilisateur dans une même classe. Les modifications apportées à mon code sont que la classe CommandWords produise des commandes plutôt que les afficher à l'aide de la fonction getCommandList(). Ce qui modifiera aussi la fonction getCommandString() et bien d'autres.

```java
/**
 * Permet d'avoir toutes les commandes dans une chaine de caractère.
 * @return la chaine de caractère.
 */
public String getCommandList()
{
    StringBuilder vCommands = new StringBuilder();
    for(String vCommand : aValidCommands){
        vCommands.append(vCommand+" ");
    }
    return vCommands.toString();
}
```

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

7.18.4) Titre du jeu : <u>Wonders of the Broken Gate</u>

7.18.6) En observant et comparant zuul with images et le nôtre, on remarque que plusieurs changements. CommandWords et Command ne sont pas modifiés. Cependant, on voit que Parsert utilise maintenant Tokenizer au lieu de Scanner. Il y a aussi 2 nouvelles classes, GameEngine qui comporte presque toutes les méthodes de game et UserInterface qui permet d'implémenter une interface

graphique, on aura donc des images et notre texte affiché à l'écran. L'ajout de UserInterface nous force à changer cette chose.

```
/**
 * Permet d'afficher la description de la salle actuelle avec les sorties possibles.
 * @param pCommand Commande permettant d'avoir le lieu actuel.
 */
private void printLocationInfo(final Command pCommand)
{
    look(pCommand);
    if ( this.aPlayer.getCurrentRoom().getImageName() != null )
            this.aGui.showImage( this.aPlayer.getCurrentRoom().getImageName() );
}//printLocationInfo()
```

7.18.8)Pour créer un bouton, il faut apporter des modifications dans la classe UserInterface. En observant notre fenêtre graphique,on remarque qu'il ne reste de la place pour le bouton qu' à l'est et l'ouest au niveau du champ de texte. Après avoir observé où mettre le bouton, il faut maintenant le coder. Tout d'abord, on import JButton dans UserInterface et ajoute un attribut JButton.

```
import javax.swing.JButton;
private JButton    aBoutonHelp;
```

Puis on le définit aBoutonHelp dans la méthode createGUI().

```
this.aBoutonHelp = new JButton("help");
```

Et on positionne le bouton à l'est du champ de texte.

```
vPanel.add(this.aBoutonHelp,BorderLayout.LINE_END);
```

Pour que l'objet courant réagisse au bouton lorsqu'il est activé, il faut lui ajouter à l'actionListener.

```
this.aBoutonHelp.addActionListener(this);
```

Et enfin on ajoute l'option d'utiliser la commande help dans actionPerformed().

```
/**
 * Actionlistener interface for entry textfield.
 */
@Override public void actionPerformed( final ActionEvent pE )
{
    // no need to check the type of action at the moment
    // because there is only one possible action (text input) :
    if("help".equals(pE.getActionCommand()))
    {
        this.aEngine.interpretCommand("help");
    }
    else{this.processCommand(); // never suppress this line
    }
} // actionPerformed(.)
```

7.19.2) Puisque qu'une interface graphique a été ajoutée, on peut maintenant déplacer toutes les images de nos salles dans le dossier du jeu, en faisant attention à ne pas mettre d'espace ni d'accent.

7.20) Un jeu n'est pas amusant si il n'y a pas d'item. On veut ajouter des items aux salles. En appliquant le principe de cohésion (une classe représente une entité précise), la classe Item est créée. Elle permet de ne pas avoir à attribuer des attributs à chaque salle. La classe Item contient 3 attributs : Son nom , la description de l'objet et son poids.

```
public class Item
{
    private String aItemDescription;
    private int aItemPoids;
    private String aItemNom;
```

Ensuite, il faut ajouter la fonction getItemString() qui permet d'afficher les items dans la salle.

```
private String getItemString()
{
    if(this.aItem ==null)
    {
        return "No item here";
    }
    else{
        return "Item :"+ this.aItem.getItemDescription() +"Weight :" + this.aItem.getItemPoids();
    }
}
```

Et aussi une fonction qui permet d'ajouter des items.

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

exemple d'utilisation:

```
Item vBoutPortail3 = new Item("Portal3","Last part of the broken portal",5);

vPortail3.setItem(vBoutPortail3);
```

7.21) Pour avoir la description des items, il faut créer l'accesseur dans la classe Item.

```
/**
 * Permet d'accéder à la description de l'item.
 */
public String getItemDescription()
{
    return this.aItemDescription;
}//getItemDescription()
```

Puis on pourra utiliser getItemDescription() dans gameEngine pour afficher leurs descriptions.

7.22) Avec la classe Item, on était bloqué à avoir 1 item par salle maximum, pour en avoir plusieurs il faut ajouter un attribut de type HashMap<String,Item>aInventory dans Room.

La HashMap permet de stocker le nom de l'item comme clé et l'item comme valeur.

```
{
    private HashMap<String,Item> aInventory;
```

```
/**
 * Contructeur par défaut, il permet de créer la liste d'item.
 */
public ItemList()
{
    this.aInventory = new HashMap<String,Item>();
}
```

Il faudra changer les méthodes créées à l'exercice précédent :

getItemString() retourne un String contenant tous les items.

```
/**
 * Afficher la liste d'item.
 * @return une phrase avec la liste d'items dedans.
 */
public String getItemListString()
{
    String vReturnString = "Inventory : ";
    Set<String> vKeys = this.aInventory.keySet();
    for(String vI: vKeys)
    {
        Item vItem = this.aInventory.get(vI);
        vReturnString += vItem.getItemName()+" ";
    }
    return vReturnString;
}
```

setItem() devient maintenant addItem().

```
public void addItem(final String pNameItem,final Item pItem)
{
    this.aItems.put(pNameItem,pItem);
}//addItem()
```

7.22.2) Avec les méthodes de l'exercice précédent, on peut donc ajouter plusieurs items aux salles.

```
//Initialisation des items
vDepart.addItem(vSword.getItemNom(),vSword);
vDepart.addItem(vBook.getItemNom(),vBook);

vArmurerie.addItem(vSwordUpgrade.getItemNom(),vSwordUpgrade);
vArmurerie.addItem(vBookUpgrade.getItemNom(),vBookUpgrade);

vSallePnj.addItem(vCookieMagique.getItemNom(),vCookieMagique);

vPortail1.addItem(vBoutPortail1.getItemNom(),vBoutPortail1);

vPortail2.addItem(vBoutPortail2.getItemNom(),vBoutPortail2);

vPortail3.addItem(vBoutPortail3.getItemNom(),vBoutPortail3);
```

7.23) On veut créer une commande qui permet de retourner dans la dernière salle parcourue par le joueur. Pour cela on commence par tout d'abord ajouter la commande "back" dans CommandWord.

```
this.aValidCommands[5] = "back";
```

Puis on ajoute un attribut aPreviousRoom dans la classe room qui gardera en mémoire la dernière salle parcourue:

```
private Room aPreviousRoom;
```

On code la méthode back():

```
private void back()
{
    if(this.aPreviousRoom !=null)
    {
        this.aCurrentRoom=this.aPreviousRoom;
        this.aGui.println( this.aCurrentRoom.getLongDescription() );
        if(this.aCurrentRoom.getImageName() != null)
        {
            this.aGui.showImage( this.aCurrentRoom.getImageName() );
        }
    }
    else{
        this.aGui.println("There is no previous room!");
    }
}
```

Puis il faut initialiser et garder en mémoire la salle dans goRoom():

```java
/**
 * Permet de se déplacer de room en room avec l'affichage de la description à c
 * d'erreur si la commande n'est pas reconnue ou null.
 * @param pCommande est la commande rentrer par le joueur
 */
private void goRoom(final Command pCommand)
{
    Room vNextRoom = null;
    String vDirection = pCommand.getSecondWord();
    vNextRoom = this.aCurrentRoom.getExit(vDirection);

    //Afficher Go where si il y a un premier mot
    if(pCommand.hasSecondWord() == false){
    this.aGui.println("Go where?");
    return;
    }
    //Déterminer le lieu suivant

    if (vNextRoom == null) {
    this.aGui.println("There is no door!");
    }
    else {
        this.aPreviousRoom= this.aCurrentRoom;
        this.aCurrentRoom = vNextRoom;
        this.aGui.println( this.aCurrentRoom.getLongDescription() );
        if ( this.aCurrentRoom.getImageName() != null )
            this.aGui.showImage( this.aCurrentRoom.getImageName() );
    }
}//goRoom()
```

7.26) Cependant si l'on utilise back() plusieurs fois, on tourne en boucle alors que notre but est de revenir au point de départ à l'aide de notre historique de passage. Pour cela au lieu de stocker une seule salle dans la variable aPreviousRoom, on va stocker les salles dans une pile.

On initialise aPreviousRoom dans la méthode createRoom():

```java
//initialisation du lieu de départ et de la previous room pour back()
this.aCurrentRoom = vDepart;
this.aPreviousRoom= new Stack<>();
```

Puis on change la méthode back():

```java
private void back()
{
    if(!this.aPreviousRoom.isEmpty())
    {
        this.aCurrentRoom=this.aPreviousRoom.pop();
        this.aGui.println( this.aCurrentRoom.getLongDescription() );
        if(this.aCurrentRoom.getImageName() != null)
        {
            this.aGui.showImage( this.aCurrentRoom.getImageName() );
        }
        }
    else{
        this.aGui.println("There is no previous room!");
    }
}//back()
```

pop() permet de sortir la dernière salle mise dans la pile.

Puis on enregistre les salles parcourues dans goRoom():

```
else {
    this.aPreviousRoom.push(this.aCurrentRoom);
    this.aCurrentRoom = vNextRoom;
    this.aGui.println( this.aCurrentRoom.getLongDescription() );
    this.aGui.println(this.aPreviousRoom.toString());
    if ( this.aCurrentRoom.getImageName() != null )
        this.aGui.showImage( this.aCurrentRoom.getImageName() );
}
```

7.28.1) Dans cette exercice, nous voulons créer une commande test() qui lira un fichier .txt et exécutera les commandes dans le fichier .txt. Pour cela on commence par tout d'abord ajouter le mot "test" dans CommandWord.

```
this.aValidCommands[6] = "test";
```

Ensuite, on écrit notre procédure dans la classe GameEngine:

```
 * Permet de tester des actions dans un fichier .txt
 * @param pCommand la commande sera "test nom_du_fichier"
 */
private void test(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("What do you want to test ?");
        return;
    }
    String vNomFichier = pCommand.getSecondWord();
    if(!vNomFichier.contains(".txt") )
    {
        vNomFichier+=".txt";
    }
    try{
        Scanner vScan = new Scanner( new File(vNomFichier) );
        while(vScan.hasNextLine())
        {
            String vLine = vScan.nextLine();
            interpretCommand(vLine);
        }
        }
    catch(final FileNotFoundException pException)
    {
        this.aGui.println("Sorry, the file" + vNomFichier + "not found");
    }
}
```

La procédure est divisé en 3 parties:
La première permet de tester si la commande rentré par le joueur contient un deuxième mot, si elle ne contient pas de 2ème mot on lui renvoie une phrase puis on finit prématurément la procédure.
La deuxième permet de voir si le 2ème mot contient .txt sinon on lui ajoute.
Et la troisième permet de lire le fichier.txt, de boucler si il reste des lignes et d'exécuter les commandes de chaque ligne.
7.29) Dans cet exercice, on cherche à créer une classe Player qui permet de stocker toutes les informations d'un joueur et aussi de créer des méthodes permettant d'accéder et modifier ces informations là. Cette création de nouvelle classe est un exemple de réingénierie qui nous servira dans le futur car dans notre ancienne méthode les informations liée à un joueur était gardé la classe GameEngine qui peut être utilisé par tous les joueurs mais imaginons qu'il y est plusieurs joueurs dans le jeu comment va t'on faire. Voici pourquoi la création de la classe Player est un bon exemple.

La classe Player contient alors le nom,le poids max, la salle actuelle du joueur,l'historique des salles parcourues et les items du joueur. Liée à cela il faut donc ajouter tous les getters et setters nécessaires. Et aussi modifier certaines méthodes dans la classe GameEngine.

“Profil du joueur”:

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

/**
 * Permet de stocker toutes les informations concernant le joueur
 *
 * @author KUCH
 */
public class Player
{
    private String aName;
    private int aWeightMax;
    private Room aCurrentRoom;
    private HashMap<String,Item> aItems;
    private Stack<Room> aPreviousRoom;

    public Player(final String pName,final Room pCurrentRoom)
    {
        this.aName = pName;
        this.aWeightMax = 10;
        this.aCurrentRoom = pCurrentRoom;
        this.aItems = new HashMap<String,Item>();
        this.aPreviousRoom = new Stack<Room>();
    }
```

Les getters et setters:

```java
public String getName()
{
return this.aName;
}

public int getWeightMax()
{
return this.aWeightMax;
}

public Room getCurrentRoom()
{
    return this.aCurrentRoom;
}

public HashMap<String,Item> getItems()
{
    return this.aItems;
}
```

Modifications apportées à certaines méthodes:

```
/**
 * Une pile stocke l'historique des salles parcourues.
 * Et back() permet de revenir en arrière à l'aide de la pile.
 */
private void back()
{
    if(!this.aPlayer.isEmpty())
    {
        this.aPlayer.setCurrentRoom(this.aPlayer.lastPreviousRoom() );
        this.aGui.println( this.aPlayer.getCurrentRoom().getLongDescription() );
        if(this.aPlayer.getCurrentRoom().getImageName() != null)
        {
            this.aGui.showImage( this.aPlayer.getCurrentRoom().getImageName() );
        }
        }
    else{
        this.aGui.println("There is no previous room!");
    }
}//back()
```

```
/**
 * Permet de se déplacer de room en room avec l'affichage de la description à chaque room parcourue. Avec l'options d'afficher des messages
 * d'erreur si la commande n'est pas reconnue ou null.
 * @param pCommand est la commande rentrer par le joueur
 */
private void goRoom(final Command pCommand)
{
    Room vNextRoom = null;
    String vDirection = pCommand.getSecondWord();
    vNextRoom = this.aPlayer.getCurrentRoom().getExit(vDirection);

    //Afficher Go where si il y a un premier mot
    if(pCommand.hasSecondWord() == false){
    this.aGui.println("Go where?");
    return;
    }
    //Déterminer le lieu suivant

    if (vNextRoom == null) {
    this.aGui.println("There is no door!");
    }
    else {
        this.aPlayer.moveTo(vNextRoom);
        this.aGui.println( this.aPlayer.getCurrentRoom().getLongDescription() );
        if ( this.aPlayer.getCurrentRoom().getImageName() != null )
            this.aGui.showImage( this.aPlayer.getCurrentRoom().getImageName() );
    }
}//goRoom()
```

Initialisation du joueur dans createRoom():

```
//initialisation du lieu de depart et de la previous room pour back()
this.aPlayer = new Player("William",vDepart);
createRoom()
```

7.30/31)On veut ajouter une fonction qui permet de prendre et de déposer un objet. Tout d'abord dans la classe Player on ajoute un attribut HashMap Inventaire qui permet de stocker ces items. Puis on va créer 2 procédures TakeItemPlayer() et DropItemPlayer():

```
public void takeItemPlayer(final String pNameItem,final Item pItem)
{
    this.aItems.put(pNameItem,pItem);
}

public void DropItemPlayer(final String pNameItem)
{
    this.aItems.remove(pNameItem);
}
```

et un accesseur qui nous permet d'avoir l'item passer en paramètre.

```java
/**
 * Permet d'avoir un item précis dans l'inventaire du joueur.
 * @return l'item en question.
 */
public Item getItemPlayer(final String pName)
{
    return this.aItemList.getItemList(pName);
}
```

On reproduit ceci dans la classe Room().

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

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

Room

Enfin après avoir fait ceci, on écrit une procédure take() et drop() dans GameEngine().

take() prend une commande en paramètre, puis vérifie que celui-ci contient un 2ème mot sinon il arrête la procédure et renvoie une phrase.

```java
private void take(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("What do you want to take ?\n");
        return;
    }
```

Puis vérifie que le deuxième mot de la commande soit un item contenue dans la pièce.

```java
String vItemName = pCommand.getSecondWord();
Item vItemToTake = this.aPlayer.getCurrentRoom().getItem(vItemName);
```

```
if(vItemToTake == null)
{
    this.aGui.println("This item is not here !\n");
}
```

Enfin si la l'item est dans la pièce, le joueur prend l'item puis on enlève l'item de la pièce.

```
this.aPlayer.takeItemPlayer(vItemName,vItemToTake);
this.aPlayer.getCurrentRoom().removeItem(vItemName);
this.aGui.println("You took the item: "+vItemName);
```

7.31.1)On remarque que dans la classe Room et Player on duplique du code permettant de gérer les items. Pour régler cela on crée une classe ItemList. Cette classe contient un attribut HashMap.

```
 */
public class ItemList
{
    private HashMap<String,Item> aInventory;
```

On y déplace aussi les méthodes tels que takeItem() et removeItem() dans Player et Room.

```
/**
 * Ajoute un item à la liste.
 * @param pN nom de l'item
 * @param pI l'item
 */
public void takeItemList(final String pN,final Item pI)
{
    this.aInventory.put(pN,pI);
}

/**
 * Opposé à take il supprime un item de la liste.
 * @param pN nom de l'item
 */
public void removeItemList(final String pN)
{
    this.aInventory.remove(pN);
}

/**
 * Permet d'accéder à un item de la liste.
 * @param pN nom de l'item
 */
public Item getItemList(final String pN)
{
    return this.aInventory.get(pN);
}
```

Ce qui modifiera aussi les mêmes méthodes. Plus de duplication de code. exemple dans takeItemPlayer().

```
this.aItemList.takeItemList(pNameItem,pItem);
```

7.32)On ajoute un restriction de poids, c'est à -dire un attribut aMaxWeight est ajouté à la classe Player ainsi que les getters et setters. Il y aura aussi des changements dans la procédure take().

```
private int aWeightMax;
private int aCurrentWeight;
```

```
/**
 * Renvoie le poids maximal que le joueur peut transporter.
 * @return Poids max
 */
public int getMaxWeight()
{
return this.aWeightMax;
}

/**
 * Permet de définir le poids maximal transportable par le joueur
 * @param pW poids
 */
public void setMaxWeight(final int pW)
{
    this.aWeightMax = pW;
}
```

Enfin pour savoir s' il est possible de prendre l'objet on crée une fonction canBeTake() qui vérifie s' il y a assez de "place" dans l'inventaire pour prendre l'objet. La fonction fait la somme du poids de l'objet et du poids du joueur afin de voir si cette somme dépasse le poids max.

```
public boolean CanBeTake(final Item pI)
{
    return (this.aCurrentWeight+ pI.getItemWeight() <= this.aWeightMax );
}
```

Lorsque tout cela est fait, il faut maintenant ajouter cette condition à la procédure take() dans Game Engine.

```
else if(this.aPlayer.CanBeTake(vItemToTake))
{
    this.aPlayer.takeItemPlayer(vItemName,vItemToTake);
    this.aPlayer.getCurrentRoom().removeItem(vItemName);
    this.aGui.println("You took the item: "+vItemName);
}
```

7.33) On souhaite pouvoir afficher notre inventaire avec toutes les informations des items et le poids total de notre inventaire. Pour cela ajoute “inventory” dans la classe CommandWord.

```
this.aValidCommands[9] = "inventory";
```

Puis dans la classe ItemList on écrit la fonction getItemListDescriptionString()

```
/**
 * Similaire à getItemListString mais avec plus d'informations.
 * @return pareil sauf avec la description et le poids en plus.
 */
public String getItemListDescriptionString()
{
    String vReturnString = "Inventory : \n";
    Set<String> vKeys = this.aInventory.keySet();
    for(String vI: vKeys)
    {
        Item vItem = this.aInventory.get(vI);
        vReturnString += vItem.getItemLongDescription()+"\n";
    }
    return vReturnString;
}
```

Et donc dans Game Engine, on écrit inventory() qui va utiliser getItemListDescriptionString() et getCurrentWeight().

```
/**
 * Affiche l'inventaire du joueur.
 */
public void inventory()
{
    this.aGui.println(this.aPlayer.PlayerInventory()+"\n"+"Weight :" +this.aPlayer.getCurrentWeight() );
}
```

7.34)On veut ajouter un item qui permet d’agrandir la poids max de l’inventaire lorsqu'on le mange. Pour cela on écrit la procédure eat() similaire à la procédure

take() juste en ajoutant un condition que l'item que l'on choisie doit être mangeable.

```
/**
 * Permet de manger le cookie magique qui augmente la taille de son inventaire.
 * @param pCommand action donner par l'utilisateur.
 */
private void eat(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("Eat what ?");
        return;
    }
    String vItemName = pCommand.getSecondWord();
    Item vItemEat = this.aPlayer.getCurrentRoom().getItem(vItemName);
    if(vItemEat == null)
    {
        this.aGui.println("You can't eat this");
    }
    else if(vItemName.equals("VitalisCookie"))
    {
        this.aPlayer.getCurrentRoom().removeItem(vItemName);
        this.aPlayer.setMaxWeight(this.aPlayer.getMaxWeight() + 5);
        this.aGui.println("You've eaten the "+vItemName+".\n"+"You are stronger,you can now lift: "+this.aPlayer.getMaxWeight());
    }
    else{
        this.aGui.println("You can't eat this");
    }
}//eat()
```

7.42)Le but de time limit est de limiter le joueur a un certain nombre d'actions sinon il perd. Pour cela il faut tout d'abord créer 2 attributs dans la classe joueur. Une qui est le nombre maximum d'actions et l'autre son compteur d'actions.

```
/**
 * Le nombre d'actions que le joueur peut faire.
 */
private int aMaxMoves= 40;

/**
 * Le nombre d'actions que le joueur a fait.
 */
private int aCurrentMoves;
```

Puis on ajoute les accesseurs et une fonction CanMove() qui regarde si aCurrentMoves est inférieur à aMaxMoves.

```java
/**
 * Permet de savoir si le joueur a le droit de bouger.
 * @return retourne vrai tant que le nombre d'action que le joueur a fait est inférieur au max.
 */
public boolean CanMove()
{
    return this.aCurrentMoves < this.aMaxMoves;
}

/**
 * Permet de connaître le nombre d'actions que le joueur a fait.
 * @return nombre d'actions en cours.
 */
public int getCurrentMoves()
{
    return this.aCurrentMoves;
}

/**
 * Permet de connaître le nombre maximum d'actions que le joueur a.
 * @return nombre d'actions maximum.
 */
public int getMaxMoves()
{
    return this.aMaxMoves;
}
```

Et les actions comptées sont bouger d'une salle et prendre/lâcher un objet. Donc dans les fonctions MoveTo() , TakeItemPlayer() et DropItemPlayer() j'ajoute un +1 au compteur.

```java
this.aCurrentMoves +=1;
```

Maintenant pour faire perdre le joueur, on doit vérifier lorsque le joueur veut bouger s' il a le droit sinon il perd. Pour cela on ajoute une condition dans GoRoom().

```java
if(!this.aPlayer.CanMove())
{
    this.aGui.println("You have no more action left !! GAME OVER");
    this.endGame();
}
```

7.42.2) J'ai juste changé la couleur du bouton Help.

```java
this.aBoutonHelp.setBackground(Color.RED);
this.aBoutonHelp.setForeground(Color.WHITE);
```

7.43)Pour créer une trapdoor, il faut tout d'abord initialiser une sortie que d'un côté.

```java
vAccueil3.setExit("up",vAccueil2);
vAccueil3.setExit("east",vSalleMonstre);

vSalleMonstre.setExit("west",vAccueil3);
vSalleMonstre.setExit("south-east",vSortieRaccourcis);
```

vAccueil a comme sortie vSalleMonstre mais pas l'inverse.

Puis on doit créer une fonction isTrapDoor() qui nous permet de savoir si la dernière salle dans la pile de back est une des sorties de la salle actuelle.
Pour cela, on écrit cette fonction dans la classe Player.

```
/**
 * Permet de savoir si la dernière salle dans la pile de back
 * @return vrai si la dernière salle parcourue est une des so
 */
public boolean isTrapdoor()
{
    return this.lastPreviousRoom().isExit(this.aCurrentRoom);
}
```

```
 */
public boolean isExit(final Room pRoom)
{
    return !pRoom.aExits.containsValue(this);
}
```

Puis on doit vérifier lorsque l'on veut se déplacer en utilisant la fonction back() que ce n'est pas une trapdoor.

```
if(this.aPlayer.isTrapdoor())
{
    this.aGui.println("You cannot go back the door is closed");
}
else{
this.aPlayer.setCurrentRoom(this.aPlayer.getPreviousRoom().pop());
this.aGui.println( this.aPlayer.getCurrentRoom().getLongDescription() );
if(this.aPlayer.getCurrentRoom().getImageName() != null)
{
    this.aGui.showImage( this.aPlayer.getCurrentRoom().getImageName() );
}
}
```

7.44)Beamer est une sorte d'item mais avec des capacités en plus. Il permet d'enregistrer une salle et lorsqu'on tire on se téléporte à la salle. Alors on doit créer une classe Beamer qui est une sorte d'item. Après l'avoir créée on ajoute 2 attributs, le premier qui est son nombre de charge et le deuxième la salle enregistrer. On crée aussi les fonctions qui permettent de charger, tirer, de savoir si on peut tirer, si le beamer a été chargé et avoir la salle chargé.

```java
public class Beamer extends Item
{
    private int aCharge;
    private Room aRoomSave;

    public Beamer(final String pNom,final String pDescription,final int pPoids)
    {
        super(pNom,pDescription,pPoids);
        Room aRoomSave;
        this.aCharge =1;
    }

    public boolean HasCharge()
    {
        return !(this.aCharge==0);
    }

    public void charge(final Room pRoom)
    {
        this.aRoomSave = pRoom;
    }

    public void fire()
    {
        this.aCharge -=1;
    }

    public Room getRoomSave()
    {
        return this.aRoomSave;
    }
}
```

```java
/**
 * Permet de savoir si on a sauvgardé la salle.
 * @return vrai si oui.
 */
public boolean isCharge()
{
    return !(this.aRoomSave==null);
}
```

Maintenant il ne reste plus qu'à ajouter les commandes charge et fire dans processCommand et dans commandWord. Puis de les écrire dans GameEngine.

```java
/**
 * Permet d'enregistrer la salle dans le beamer.
 * @param pCommand la commande.
 */
private void charge(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("What do you want to charge ?");
        return;
    }

    String vStringBeamer = pCommand.getSecondWord();
    Beamer vBeamer =(Beamer)this.aPlayer.getItemPlayer(vStringBeamer);

    if(vBeamer == null)
    {
        this.aGui.println("You dont have the beamer !");
    }
    else{
        vBeamer.charge(this.aPlayer.getCurrentRoom());
        this.aGui.println("The beamer is charged !");
    }
}//charge(.)
```

```java
public void fire(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("What do you want to fire ?");
        return;
    }

    String vStringBeamer = pCommand.getSecondWord();
    Beamer vBeamer =(Beamer)this.aPlayer.getItemPlayer(vStringBeamer);

    if(vBeamer == null)
    {
        this.aGui.println("You dont have the beamer !");
    }
    else{
        if(vBeamer.HasCharge() && vBeamer.isCharge() )
        {
            vBeamer.fire();
            this.aPlayer.setCurrentRoom(vBeamer.getRoomSave());
            this.aGui.println( this.aPlayer.getCurrentRoom().getLongDescription());
            if(this.aPlayer.getCurrentRoom().getImageName() != null)
            {
            this.aGui.showImage( this.aPlayer.getCurrentRoom().getImageName() );
            }
        }
        else{
            this.aGui.println("You cannot fire because you dont any charge left or you didn't charge the beamer!");
        }
```

7.45)Pour créer une porte fermée, il faut tout d'abord créer une nouvelle classe Door qui a 2 attributs , le premier pour savoir si la porte est fermée ou pas et le

deuxième est l'item qui permet de l'ouvrir. On ajoute aussi des fonctions qui permettent de savoir si la porte est ouverte et d'ouvrir la porte.

```java
/**
 * Ouvre la porte.
 */
public void setDoorOpen()
{
    this.aLock=false;
}

/**
 * Permet de savoir si la porte est fermée.
 * @return vrai si la porte n'est pas fermée
*/
public boolean CanBeOpen()
{
    return !(this.aLock);
}
```

Puis dans la classe Room, on ajoute un nouvel attribut qui se superpose sur la hashmap des sorties. Il prend pour clé la direction et comme valeur une porte.

```java
private HashMap<String,Door> aDoor;
```

```java
/**
 * Permet de créer une porte pour la salle précisé.
 * @param pDirection La direction où est la porte.
 * @param pDoor La porte.
 */
public void setDoor(final String pDirection,final Door pDoor)
{
    this.aDoor.put(pDirection,pDoor);
}

/**
 * Permet d'avoir la porte lié à la salle.
 * @param pDirection La direction de la porte.
 * @return la porte liée à la salle.
 */
public Door getDoor(final String pDirection)
{
    return this.aDoor.get(pDirection);
}//getExit()
```

On ajoute les portes nécessaires dans createRoom().

```java
//Initialisation des portes
Door vPortal = new Door();
vPortail.setDoor("home",vPortal);

Door vDoor= new Door();
vSallePorteF.setDoor("south",vDoor);
```

Puis dans goRoom() on vérifie si la direction où l'on va a une porte liée.

```java
vDoor =this.aPlayer.getCurrentRoom().getDoor(vDirection);
```

```java
else if(vDoor != null)
{
```

Et si il a une porte on regarde si elle est ouverte,sinon on regarde si la salle actuelle où se situe le joueur contient l'item demandé si oui on ouvre la porte sinon on refuse l'accès à la salle.

```
if(vDoor.CanBeOpen())
{
    this.aGui.println("The door is open.");
    this.aPlayer.moveTo(vNextRoom);
    this.aGui.println("\n");
    this.aGui.println( this.aPlayer.getCurrentRoom().getLongDescription() );
    if ( this.aPlayer.getCurrentRoom().getImageName() != null )
    this.aGui.showImage( this.aPlayer.getCurrentRoom().getImageName() );
}
else
{
    if(this.aPlayer.getCurrentRoom().getItemList().containsItem("Key"))
    {
    vDoor.setDoorOpen();
    this.aGui.println("The door is now open you can go.");
    }
    else{
    this.aGui.println("The door is close.Try to drop a key in the room then retry.");
    }
```

Pour gagner:

J'utilise le même concept de locked door avec une salle appelée aWinningRoom et si on arrive dans cette salle on gagne. Pour gagner il faut déposer les 3 bouts du portail dans la salle du portail.

Pour cela, j'ai créé un attribut qui est la salle gagnante. Puis je l'ai ajouté en tant que sortie possible à la salle du portail mais avec une porte entre les deux qui a comme conditions d'ouverture que la salle doit avoir tous les morceaux de portail.

```
 */
private Room aWinningRoom =new Room("You're Home, You Won.","image/home.png");
```

```
/**
 * Permet de savoir si la salle actuel du joueur contient les 3 bouts de portail.
 * @return vrai si la salle du joueur contient les 3 bouts du portail.
 */
private boolean CanOpenPortal()
{
    return this.aPlayer.getCurrentRoom().getItemList().containsItem("Portal1") && this.aPlayer.getCurrentRoom().getItemList().containsItem("Portal2") &&
}
```

```
if(vNextRoom==aWinningRoom)
{
    if(vDoor.CanBeOpen())
    {
        this.aPlayer.moveTo(vNextRoom);
        this.aGui.println("\n");
        this.aGui.println("YOUU WOONNN !!");
        if ( this.aPlayer.getCurrentRoom().getImageName() != null )
        this.aGui.showImage( this.aPlayer.getCurrentRoom().getImageName() );
        this.endGame();
        return;
    }
    else
    {
        if(this.CanOpenPortal())
        {
            vDoor.setDoorOpen();
            this.aGui.println("The portal is now open you can go home.");
            return;
        }
        else{
            this.aGui.println("The portal is close.You need all the pieces of the portal");
            return;
        }
    }
}
```

Declaration antiplagiat

J’ai été aidé par des camarades mais n'ai pas copié.