Table des matières

Présentation du jeu

Auteur :

Baptiste Espinasse

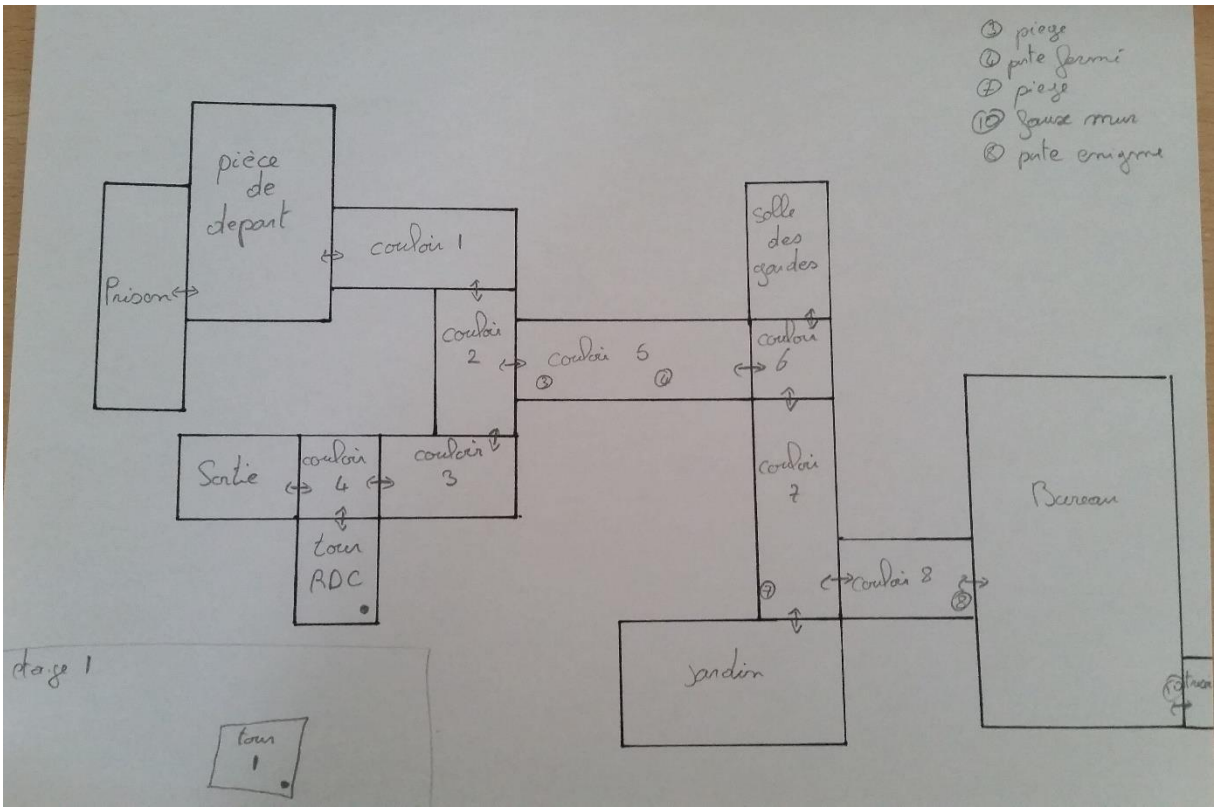
Thème :

Une équipe d'aventurier doit retrouver un antidote volé par un savant fou à l'intérieur d'un donjon

Résumé du scénario complet :

Malheureusement, la nièce de l'alchimiste a confondu son milk shake fraise avec un flacon d'un poison très rare. Elle est désormais à l'article de la mort. En temps normal Balthazar n'aurait eu aucun problème à lui inoculer l'antidote, sauf que quelques semaines plus tôt celui-ci avait un stagiaire qui se révéla un peu fou. Quand le vieil homme le renvoya, ce dernier fou de rage lui vola une partie de ses potions. Et il n'a pas les ingrédients pour refaire l'antidote. La mission des aventuriers et donc de partir sur-le-champ où le stagiaire fou réside et de lui reprendre l'antidote.

Plan complet



Scénario détaillé

La veille : Le groupe d'aventuriers venant tout juste de résoudre une quête fête cette victoire dans la taverne du village le plus proche. Une fois dans la taverne et ayant bu plus que de raison une bonne partie de la nuit, un vieil homme entra en catastrophe dans la taverne. Celui si était un ancien alchimiste. Après avoir écouté cet homme du nom de balthazar, les aventuriers partirent pour résoudre cette nouvelle quête qui venait tout juste de leur être confiée. Dans un état plus que précaire (complètement ivre), le groupe était parti en route vers le donjon. Le seul objet qu'ils ont sur cette quête est un morceau de papier se trouvant dans l'une des poches des aventuriers.

La mission : Malheureusement, la nièce de l'alchimiste a confondu son milk shake fraise avec un flacon d'un poison très rare. Elle est désormais à l'article de la mort. En temps normal Balthazar n'aurait eu aucun problème à lui inoculer l'antidote, sauf que quelques semaines plus tôt celui-ci avait un stagiaire qui se révéla un peu fou. Quand le vieil homme le renvoya, ce dernier fou de rage lui vola une partie de ses potions. Et il n'a pas les ingrédients pour refaire l'antidote. La mission des aventuriers et donc de partir sur-le-champ où le stagiaire fou réside et de lui reprendre l'antidote.

Réponses aux exercices

Exercice 7.5 (printLocationInfo)

Après avoir écrit les méthodes printWelcome() et goRoom(), on remarque que tous deux exécutent la même suite de fonctions. Ceci est une duplication de code.

Pour éviter cette duplication de code, on peut créer une méthode printLocationInfo() qui effectuera cette même suite de fonctions.

Ensuite, nous appellerons cette procédure dans printWelcome et goRoom.

```
private void printLocationInfo()
{
    System.out.println("You are in the "+ this.aCurrentRoom.getDescription());
    System.out.print("Exit(s):");
    if(this.aCurrentRoom.getExit("north") != null) System.out.print("north ");
    if(this.aCurrentRoom.getExit("south") != null) System.out.print("south ");
    if(this.aCurrentRoom.getExit("east") != null) System.out.print("east ");
    if(this.aCurrentRoom.getExit("west") != null) System.out.print("west");
}
```

Exercice 7.6 (getExit)

Nous souhaitons ajouter deux nouveaux types de direction pour sortir d'une pièce tels que "haut" et "bas ». Malheureusement, lors de la première création de la class Room les attributs de directions étaient publique aux autres class. Ce qui a permis à la class Game d'utiliser un accès à ceci très simple. Mais maintenant que nous voulons modifier la class Room cela va perturber le bon fonctionnement de Game, car ces deux class ont un couplage fort.

Pour remédier à ce problème nous allons renforcer la séparation de ces deux class en rendant les attributs prive.

Ce qui oblige de créer un getteur a la class room.

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

maintenant, il faut également modifie la class Game. Qui a maintenant besoin des getters pour accédait au champ de Room.

Au lieu d'écrire :

```
vNextRoom = this.aCurrentRoom.eastExit;
```

Il faut :

```
vNextRoom = this.aCurrentRoom.getExit("east");
```

Au premier abord rendre privé les attributs peut sembler générer une difficulté en plus, mais sur le long terme, cela facilite la modification du code.

Par exemple le code suivant :

```
        Room vNextRoom = null;
        String vDirection =pCommand.getSecondWord();

        if( vDirection.equals("north") ) vNextRoom = this.aCurrentRoom.aNorthExit;
        if( vDirection.equals("south") ) vNextRoom = this.aCurrentRoom.aSouthExit;
        if( vDirection.equals("east") ) vNextRoom = this.aCurrentRoom.aEastExit;
        if( vDirection.equals("west") ) vNextRoom = this.aCurrentRoom.aWestExit;
```

Devient beaucoup plus court, et permet d'ajouté une nouvelle direction de sortie avec aucune ligne a modifiée dans la class Game:

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

Exercice 7.7 (getExitString)

Dans la même optique que dans la question précédente, nous allons créer une méthode getExitString() dans la class Room.

Elle aura pour but de simplifier la méthode printLocationInfo, cette dernière n'aura plus qu'à afficher la String de retour de getExitString().

Ainsi si de nouvelles directions de sortie sont ajoutées cela n'aura pas d'impact sur printLocationInfo().

À noter qu'il faudra tout de même modifier getExitSting() si une nouvelle direction est ajoutée pour – l'instant.

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

```
private void printLocationInfo()
{
    System.out.println("You are in the "+ this.aCurrentRoom.getDescription());
    System.out.println(this.aCurrentRoom.getExitString());
}
```

Exercice 7.8 (HashMap, setExit)

Dans le but d'avoir de créé des sorti différente pour chaque objet Room. Nous allons remplacer les 4 attributs de direction par une table de hachage. Ainsi on ne sera plus limitée à créer des sorties correspondant aux quatre points cardinaux.

Et comme la class Room a déformé une encapsulation forte grâce au travail précédent, les modifications qui vont lui être apportées n'auraient aucune répercussion sur les autres class (correction : il faut tout de même revoir les instanciations des objets Room dans les autres class).

```
import java.util.HashMap;

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

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

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

    public Room getExit(String pDirection)
    {
        return this.aExits.get(pDirection);
    }

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

    public void setExit(final String pDirection, final Room pNeighbor)
    {
        this.aExits.put(pDirection, pNeighbor);
    }
}
```

une table de hachage simplement est un tableau ou les indices ne sont pas des entiers de 0 à N-1, mais des objets que l'on nommera "key".

Dans notre cas les "key" sont les noms de sortie de la pièce. Pour utiliser ce paquetage, il faut ajouter : `java import java.util.HashMap;`

Ainsi on peut instancier des objets de type Hashmap et utiliser les méthodes qui sont déjà créées dans le paquetage. Le constructeur naturel subit lui aussi des modifications pour correspondre aux attributs de la class.

Exercice 7.9 (keySet)

Il faut mettre la méthode getExitString à jour.

```
public String getExitString()
{
    String vReturnString = "Exits:";
    Set<String> vKeys = this.aExits.keySet();
    for(String vExit : vKeys)
    {
        vReturnString += " "+vExit;
    }
    return vReturnString;
}
```

Exercice 7.10 (getExitString CCM?)

Le but de la méthode getExitString est de retourner sous forme d'un String toutes les sorties possibles pour la commande go.

Toutes ces directions de sortie sont les "key" de la table de hachage aExits.

L'interface Set est une collection d'objets dans lequel on ne peut pas avoir de doublons.

La méthode keySet() permet de retourner un objet de type set<> représentant la liste des clés contenues dans la collection.

Donc en faisant « `java Set<String> vKeys = this.aExits.keySet();` » On stock toute les "key" (direction de sortie de la pièce this.) dans la collection de types Set<String> vKeys.

La boucle for each :

```
for(type variable : collection)
{
 /*instruction*/
}
```

Va effectuer les instructions sur les objets de la collection une par une dans l'ordre.

Exercice 7.11 (getLongDescription)

Pour encore réduire n'encapsulation de la class Room et en prévision de modification future telle que l'ajout de personnage et objet dans les pièces. Il faut ajouter une nouvelle méthode qui pourra fournir une description de la pièce et de tout ce qui s'y trouve.

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

Exercice 7.14 (look)

Depuis le début du projet nous nous jamais soucier des problèmes de couplage implicite.

Un couplage implicite est une situation ou une class dépend des informations d'une autre, mais à la différence d'un couplage normal, celui-ci ne produira pas d'erreur de compilation.

Ce problème s'illustre dans cet exercice par l'ajout d'une nouvelle commande (look) dans le jeu.

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

Si nous ajoutons seulement la méthode look dans la class Game il n'y aura pas d'erreur de compilation. En revanche l'utilisateur ne pourra jamais utiliser cette commande, car elle n'est pas connue de la class CommanWords.

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

Et de la méthode qui interprète les commandes dans la class Game.

```
public boolean processCommand(final Command pCommand)
{
    if(pCommand.isUnknown()) System.out.println("I don't know what you mean...");
    if (pCommand.getCommandWord().equals("help"))  printHelp();
    if (pCommand.getCommandWord().equals("go"))    goRoom(pCommand);
    if (pCommand.getCommandWord().equals("look"))  look();
    if (pCommand.getCommandWord().equals("quit"))
    {
        Command vCommand = new Command(pCommand.getSecondWord(), null);
        return quit(vCommand);
    }
    return false;
}
```

Exercice 7.15 (eat)

Pour cette commande il faut effectuée les mêmes modifications que pour l'exercice précèdent.

Dans la class Game

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

Ajout de l'interprétation de la commande par la méthode processCommand()

```
if (pCommand.getCommandWord().equals("eat"))    eat();
```

Et dans la class CommandWords

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

Exercice 7.16 (showAll, showCommands)

Dans les deux exercices précèdent il a été oublie d'ajouter à la méthode help() d'ajouter les commandes look et eat.

C'est un problème de couplage implicite.

Pour que ce problème n'arrive plus on va ajouter une méthode showAll() qui affichera la liste de toutes les commandes répertorier dans sValidCommands de la class CommandWords.

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

Il faut maintenant pouvoir appeler cette méthode dans printHelp(), mais comme nous ne souhaitons pas augmenter le degré de couplage dans l'application, il ne faut pas faire de lien direct en la class Game et CommandWords.

Il faut donc faire communiquer ComandWords avec Parser puis Parser avec Game

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

```
private void printHelp()
{
    System.out.println("You are lost. You are alone.");
    System.out.println("You wander around at the university.");
    System.out.println();
    System.out.println("Your command words are:");
    aParser.showCommands();
}
```

Exercice 7.18 (getCommandList)

Suppression de la méthode showAll() de ma classCommandWords.Il est nécessaire de supprimer cette procédure, car dans des évolutions futures du jeu il ne faudra plus afficher les commandes disponibles dans un terminal à l'aide de l'instruction System.out.println().

Il est donc, pour des raisons d'encapsulation, de créer une méthode qui préparera un String contenant toutes les commandes pour pouvoir ensuite les afficher.

```
public String getCommandList()
{
    StringBuilder commands = new StringBuilder();
    for(int i = 0; i < sValidCommands.length; i++) {
        commands.append( sValidCommands[i] + "   " );
    }
    return commands.toString();
}
```

Exercice 7.18.1 : Comparer son projet au projet zuul-better ...

Les seules différences présentent entre Zuul-better et mes sources, sont les méthodes eat, look qui sont en plus.

Exercice 7.18.3 : Chercher des images ...

Recherche d'images sur internet pouvant correspondre aux salles présentent dans le jeu.

Exercice 7.18.4 : Décider du titre du jeu ...

Exercice 7.18.5 : Les objets Room ...

Comme les pièces du jeu sont créées dans une des méthodes de la class Game ils ne sont pas accessible à l'extérieur de la class.
C'est pourquoi il faut ajouter un attribut prive à la class pour pouvoir y accéder depuis l'extérieur.
Pour ce faire nous allons créer une liste de type HachMap()

```
private HashMap<String, Room> aListeRoom;
```

Il sera maintenant possible après la création des objets Room de lister tous ces objets dans le HashMap() de cette manière:

```
aListeRoom = new HashMap();
this.aListeRoom.put("Piece de depart",vPieceDeDepart);
this.aListeRoom.put("couloir 1",vCouloir1);
this.aListeRoom.put("couloir 2",vCouloir2);
this.aListeRoom.put("couloir 3",vCouloir3);
this.aListeRoom.put("couloir 4",vCouloir4);
this.aListeRoom.put("couloir 5",vCouloir5);
this.aListeRoom.put("couloir 6",vCouloir6);
this.aListeRoom.put("couloir 7",vCouloir7);
this.aListeRoom.put("couloir 8",vCouloir8);
this.aListeRoom.put("Sortie",vSortie);
this.aListeRoom.put("RDC de la tour",vTourRDC);
this.aListeRoom.put("Sommet de la tour",vTourHight);
this.aListeRoom.put("Salle des gardes",vSalleDesGardes);
this.aListeRoom.put("Jardin",vJardin);
this.aListeRoom.put("Bureau",vBureau);
this.aListeRoom.put("Salle au tresor",vTresor);
```

Exercice 7.18.6 : Étudier le projet zuul-with-images ...

Exercice 7.18.8 : Ajouter au moins un bouton ...

Pour la création des boutons j'ai choisi d'ajouter un nouveau JPanel du nom de aButton que j'ajoute ensuite dans vPanel.

Pour la création de aButton et de tous les éléments qui seront dispos à l'intérieur, j'ai ajouté une procédure

makeButtonBar() qui s'occupe de la disposition et la création des boutons dans le Panel aButton.

```
    public void makeBoutonBar()
{
    aButton = new JPanel();
    aButton.setLayout(new GridLayout(0,1,3,5));

    this.aButtonN = new JButton("north");
    this.aButtonN.addActionListener(this);
    this.aButtonS = new JButton("south");
    this.aButtonS.addActionListener(this);
    this.aButtonE = new JButton("east");
    this.aButtonE.addActionListener(this);
    this.aButtonW = new JButton("west");
    this.aButtonW.addActionListener(this);
    this.aButtonEat = new JButton("eat");
    this.aButtonEat.addActionListener(this);
    this.aButtonLook = new JButton("look");
    this.aButtonLook.addActionListener(this);
    this.aButtonHelp = new JButton("help");
    this.aButtonHelp.addActionListener(this);

    aButton.add( this.aButtonN);
    aButton.add( this.aButtonS);
    aButton.add( this.aButtonE);
    aButton.add( this.aButtonW);
    aButton.add( this.aButtonEat);
    aButton.add( this.aButtonHelp);
    aButton.add( this.aButtonLook);
}
```

Ensuite j'ajoute les actions à réaliser lors de l'appui sur les boutons.

```
public void actionPerformed( final ActionEvent pE )
{
    // no need to check the type of action at the moment.
    // there is only one possible action: text entry
    if(pE.getSource() == this.aButtonN){ this.aEngine.interpretCommand("go north");}
    if(pE.getSource() == this.aButtonS){ this.aEngine.interpretCommand("go south");}
    if(pE.getSource() == this.aButtonE){ this.aEngine.interpretCommand("go east");}
    if(pE.getSource() == this.aButtonW){ this.aEngine.interpretCommand("go west");}
    if(pE.getSource() == this.aButtonEat){ this.aEngine.interpretCommand("eat");}
    if(pE.getSource() == this.aButtonLook){ this.aEngine.interpretCommand("look");}
    if(pE.getSource() == this.aButtonHelp){ this.aEngine.interpretCommand("help");}
    if(pE.getSource() == this.aEntryField) {processCommand();}
} // actionPerformed(.)
```

Et enfin j'ajoute dans vPanel

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

Exercice 7.19.2 : Déplacer toutes les images ...

Je déplace toutes les images dans un dossier images

Puis je change le chemin de mes images

Exercice 7.20 (Item)

Création d'une nouvelle class Item.

Cette class a pour but de créer des objets item.

```
public class Item
{
    private String aDescription;
    private double aPrix;
    private double aPoids;

    public Item(final String pDescription, final double pPrix, final double pPoids)
    {
        this.aDescription = pDescription;
        this.aPrix        = pPrix;
        this.aPoids       = pPoids;
    }

    public String getDescriptionItem(){return this.aDescription;}
    public double getPrixItem(){return this.aPrix;}
    public double getPoidsItem(){return this.aPoids;}

    public void setDescriptionItem(final String pDescription){this.aDescription=pDescri
    public void setPrixItem(final double pPrix){this.aPrix=pPrix;}
    public void setPoidsItem(final double pPoids){this.aPoids=pPoids;}
}
```

Exercice 7.21 (item description)

Création d'une méthode toString() qui aura pour but de retourner une description complète de l'Item sous forme d'un String.

```
@Override
public String toString()
{
    return this.aDescription + "cette objet pèse " +this.aPoids+"Kg et coute "+this.aPrix+"piece d'or.\n";
}
```

Exercice 7.22 (items)

Maintenant que la class Item es finie il faut ajoute une méthode à la class Room et un attribue. L'attribue sera une collection d'items contenue dans la Room.

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

Et la méthode addItem() aura pour but d'ajouter un Item à la collection.

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

Exercice 7.22.2 : Intégrer les objets (items) ...

Pour ajouter un Item dans une pièce il faut dans la méthode CreatRoom() ajouter après la création des Objets Room .

```
vPieceDeDepart.addItem("torche", vTorche);
```

Par exemple pour ajouter une torche dans la première pièce du jeu.

Exercice 7.23 (back)

On souhaite ajouter une nouvelle commande "back" elle aura pour but de permettre au jour de revenir dans la salle précédente sans à connaître la direction.
Pour cela rien de plus simple. Il suffit d'ajouter un attribut à la class Gameangine, cet attribut contiendra le nom de la Room précédente.

```
private Room aLastRoom;
```

On ajoute la ligne 16 à la méthode goRoom()

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

    String vDirection = pCommand.getSecondWord();

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

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

Maintenant je crée ma procédure back()

```
private void back()
{
    Room vCurrentRoom = this.aCurrentRoom;
    this.aCurrentRoom = this.aLastRoom;
    this.aLastRoom = vCurrentRoom;

    this.aGui.println(this.aCurrentRoom.getLongDescription());
    if(this.aCurrentRoom.getImageName() != null)
        this.aGui.showImage(this.aCurrentRoom.getImageName());
}
```

Il faut ajouter l'interprétation de la commande "back" dans la méthode interpretCOmmand()

```
else if (commandWord.equals("back"))    back();
```

Il reste une petite chose à faire. L'ajout du String "back" dans le tableau contenant toute les commandes connues dans la class CommandWords.

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

Faire l'exercice 7.26 (Stack) :

Stack est une collection de base dans la JDK, sont principe de fonctionnement est comme une pile, c'est-à-dire le dernier objet ajouter dans la collection est aussi le premier à en sortir.

Pour pouvoir l'utiliser il faut l'importer :

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

Nous souhaitons créée une collection contenant toute les pièces visiter dans l'ordre chronologique de la dernière a la première c'est pourquoi nous utilisons Stack

Nous ajoutons donc un attribut a la class GameEngine :

```
private Stack<Room> aLastRooms;
```

Chaque déplacement dans une nouvelle pièce, il faut ajouter le nom de l'ancienne position, cette étape se trouve dans la méthode goRoom() :

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

La méthode push de la class Stack ajoute l'objet spécifié au sommet de la pile et le retourne.

```
private void back()
{
    if(this.aLastRooms.empty() == true) //si la liste est vide(retour a la premier pc
    {
        this.aGui.println("You are all ready in your first localisation.");
    }
    else
    {
        this.aGui.println("your go back in the last room");
        this.aCurrentRoom = this.aLastRooms.pop(); // la piece courant deviens la pie
    }

    this.aGui.println(this.aCurrentRoom.getLongDescription());
    if(this.aCurrentRoom.getImageName() != null)
    this.aGui.showImage(this.aCurrentRoom.getImageName());
}
```

La méthode pop() de la class Stack récupère l'objet au sommet de la pile.
La méthode empty() teste si la pile ne contient aucun élément (return true si vide).

Exercice 7.28.1 : Créer une nouvelle commande test …

Nous souhaitons créée une commande de teste qui permettra de savoir si le jeu est jouable sans avoir à jouer au jeu. Pour cela, il faut ajouter une commande qui réalisera une routine de test à partir d'un fichier contenant toutes les commandes à réaliser.

Dans la classe CommandWords, il faut ajouter le nouveau mot:

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

La partie compliquée se trouve la classe GameEngine. Ou il faut maintenant ouvrir le fichier contenant la liste des commandes à réaliser et les recouper pour les exécuter. Ajout d'une méthode test():

```
private void test(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("test what?");
        return;
    }

    String vFile = pCommand.getSecondWord();
    Scanner vScan = null;

    try {vScan = new Scanner(new File("./"+vFile+".txt"));}
    catch ( final java.io.FileNotFoundException pException )
    {
        this.aGui.println("File not find");
    }

    while(vScan.hasNextLine())
    {
        String vLigne = vScan.nextLine();
        interpretCommand(vLigne);
    }
    if (vScan != null) {vScan.close();}
}
```

Il faut ajouter les dépendances a la class:

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

Et ajouter l'interprétation de la commande dans la méthode interpretCommand();

```
else if (commandWord.equals("test"))    test(command);
```

Exercice 7.28.2 : Créer 2 fichiers de commandes …

/todo : expliquer la création de fichiers test/

Exercice 7.29 (Player) :

Tout d'abord il faut savoir ce que l'on attend de la classe player, on sait que l'on ne doit plus enregistrer les déplacements dans gameEngine puisque que l'on pourrait imaginer un jeu avec plusieurs player et chacun pourrait faire une suite de déplacement différant. Il faut donc enregistre l'emplacement actuel du player. Et faire une liste contenant tous ses emplacements précédents. Le player a un nom pour le différencier des autres et il a une capacité de charge utile tout ceci nous permet de connaître les attributs dont la classe a besoin.

```
private String aName;
private double aWeight;
private Room aLocalisation;
private HashMap<String, Item> aInventary;
private Stack<Room> aLastRooms;
```

Maintenant il ne reste qu'a complété la classe avec son constructeur et l'ensemble des getteur et setter

```
public Player(final String pName, final Room pLocalisation)
{
    this.aName = pName;
    this.aLocalisation = pLocalisation;
    this.aWeight = 700;
    this.aInventary = new HashMap<>();
    this.aLastRooms = new Stack();
}

public String getName(){return this.aName;}
public void setName(final String pName){this.aName = pName;}

public double getWeight(){return this.aWeight;}
public void setWeight(final double pWeight){this.aWeight = pWeight;}

public Room getLocalisation(){return this.aLocalisation;}
public void setLocalisation(final Room pLocalisation){this.aLocalisation = pLocalisati

public Room getLastRoom(){return this.aLastRooms.pop();}
public void setLastRoom(final Room pLastRoom){this.aLastRooms.push(pLastRoom);}
public boolean lastRoomsIsEmpty(){return this.aLastRooms.empty();};
```

Dans Game Engine il faut ajouter la création d'un nouveau player dans le constructeur. Et supprimer la liste des Room précédente puisque c'elle ci est différente pour chaque player, il faut aussi modifier les méthodes utilisant les informations sur la Room courante et la liste des emplacements précédant. Soit Back() goRoom() et printWelcome()

```
private void printWelcome()
{
    this.aGui.print("\n");
    this.aGui.println("Welcome to the World of Zuul!");
    this.aGui.println("World of Zuul is a new, incredibly boring adventure game.");
    this.aGui.println("Type 'help' if you need help.");
    this.aGui.print("\n");
    Room currentRoom = this.aPlayer.getLocalisation();
    this.aGui.println(currentRoom.getLongDescription());
    this.aGui.showImage(this.aPlayer.getLocalisation().getImageName());
}
```

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

    String vDirection = pCommand.getSecondWord();

    Room vNextRoom = this.aPlayer.getLocalisation().getExit(vDirection);

    if (vNextRoom == null) this.aGui.println("There is no door !");
    else
    {
        this.aPlayer.setLastRoom(this.aPlayer.getLocalisation());
       this.aPlayer.setLocalisation(vNextRoom);
        this.aGui.println(this.aPlayer.getLocalisation().getLongDescription());
        if(this.aPlayer.getLocalisation().getImageName() != null)
            this.aGui.showImage(this.aPlayer.getLocalisation().getImageName());
    }
}
```

```
private void back()
{
    if(this.aPlayer.lastRoomsIsEmpty() == true)
    {
        this.aGui.println("You are all ready in your first localisation.");
    }
    else
    {
        this.aGui.println("your go back in the last room");
        this.aPlayer.setLocalisation(this.aPlayer.getLastRoom());
    }

    this.aGui.println(this.aPlayer.getLocalisation().getLongDescription());
    if(this.aPlayer.getLocalisation().getImageName() != null)
            this.aGui.showImage(this.aPlayer.getLocalisation().getImageName());
}
```

Exercice 7.30 (take, drop) :

Comme à l'habite lors de l'ajout de nouvelles commandes il faut ajouter dans CommandWords les deux nouveaux mots de commande. Ici dans ce cas "take" et "drop":

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

Ajouter les commandes à la méthode interpretCommand de la classe gameEngine

```
else if (commandWord.equals("take"))    take(command);
else if (commandWord.equals("drop"))    drop(command);
```

```
private void take(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("take what ?");
        return;
    }

    String vItem = pCommand.getSecondWord();

    Item vToTake = this.aPlayer.getLocalisation().getItem(vItem);

    if (vToTake == null) this.aGui.println("this item is not here !");
    else{
        this.aPlayer.takeItem(vItem, vToTake);
        this.aPlayer.getLocalisation().removeItem(vItem);
        this.aGui.println("I take the item");

    }
}
```

Dans player il faut ajouter une liste pour l'inventaire

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

et

```
public boolean lastRoomsIsEmpty(){return this.aLastRooms.empty();}
public void resetLastRoom(){this.aLastRooms.clear();}

public void takeItem(final String pStringItem, final Item pItem){this.aItemsList.takeItem(pStringItem,pItem);}
public void dropItem(final String pStringItem){this.aItemsList.dropItem(pStringItem);}
public Item getItem(final String pItem){return this.aItemsList.getItem(pItem);}
```

Dans Room

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

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

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

Exercice 7.31 (porter plusieurs items)

Fait avec la question précédente

Exercice 7.31.1 : Créer une nouvelle classe ItemList …

Pour diminuer le couplage des class PLayer et Room avec GameEngine il faut créer une nouvelle classe ItemsList. Cette class de gérer les inventaires des Room et des Player

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

public class ItemList
{
    private HashMap<String, Item> aInventary;

    /**
     * Constructeur d'objets de classe ItemList
     */
    public ItemList()
    {
        this.aInventary = new HashMap<>();
    }

    public String getItemString(){
        String vReturnString = "";
        Set<String> vKeys = this.aInventary.keySet();
        for(String vItem : vKeys)
        {
            vReturnString += " a "+vItem+"\n";
        }
        return vReturnString;
    }

    public void takeItem(final String pStringItem, final Item pItem){this.aInventary
    public void dropItem(final String pStringItem){this.aInventary.remove(pStringIte
    public Item getItem(final String pItem){return this.aInventary.get(pItem);}

}
```

Exercice 7.32 (poids max) :

Ajout dans la class Player un attribut de aStrong qui correspond au poids max que le joueur peut porter

```
private double aStrong;
```

Ainsi que ces getter et setter

```
public double getStrong(){return this.aStrong;}
public void setStrong(final double pStrong){this.aStrong = pStrong;}
```

Et une méthode permettant de savoir di le joueur peut ajouter à son inventaire un objet d'un certain poids

```
public boolean canITake(final double pWeight){
    return (pWeight+this.aWeight <= this.aStrong) ;
}
```

Et dans la méthode take de la classe GameEngine il faut ajouter une condition de test pour savoir si le player est en condition de prendre l'objet

```
else if(this.aPlayer.canITake(vToTake.getPoidsItem()) == false) this.aGui.println("this item is too heavy !");
```

Exercice 7.33 (inventaire) :

Ajout d'une commande pour afficher la liste des objets dans l'inventaire du joueur

```
private void items(){this.aGui.println(this.aPlayer.getItemsString());}
```

Exercice 7.34 (magic cookie) :

Modification de la commande eat, pour que si le joueur mange un magicCookie il ait une modification de la sont attribut de force.

```
private void eat(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("eat what?");
        return;
    }

    String vItem = pCommand.getSecondWord();

    Item vToEat = this.aPlayer.getItem(vItem);

    if(vToEat == null) this.aGui.println("I don't have it !");
    else if(vItem.equals("magicCookie")){
        this.aPlayer.takeItem(vItem, vToEat);
        this.aPlayer.getLocalisation().dropItem(vItem);
        this.aPlayer.setStrong(this.aPlayer.getStrong()+100);
        this.aGui.println("You have eat the magic cookie");
    }
    else
        this.aGui.println("You have eaten now and you are not hungry any more.");
}
```

Exercice 7.35 (zuul-with-enums-v1) :

Pour diminuer encore un peut plus le couplage implicite de la classe CommandWords et la classe Gameengine, il est demandé de suivre la nouvelle architecture fournie dans zuul-with-enums-v1

Tout d'abord mous créons une nouvelle class de type enum qui contiendra toutes les commandes réalisables :

```
public enum CommandWord
{
    GO, QUIT, HELP, LOOK, EAT, BACK, TAKE, DROP, ITEMS, TEST, UNKNOWN;
}
```

On enlève toutes les commandes valides de la class CommandWords en le remplacent par une collection

```
private HashMap<String,CommandWord> aValidCommands;
```

Pour le moment on remplit la collection dans le constructeur de la class

```
public CommandWords()
{
    this.aValidCommands = new HashMap<String, CommandWord>();
    this.aValidCommands.put("go", CommandWord.GO);
    this.aValidCommands.put("help", CommandWord.HELP);
    this.aValidCommands.put("quit", CommandWord.QUIT);
    this.aValidCommands.put("look", CommandWord.LOOK);
    this.aValidCommands.put("eat", CommandWord.EAT);
    this.aValidCommands.put("back", CommandWord.BACK);
    this.aValidCommands.put("test", CommandWord.TEST);
    this.aValidCommands.put("take", CommandWord.TAKE);
    this.aValidCommands.put("drop", CommandWord.DROP);
    this.aValidCommands.put("ITEMS", CommandWord.ITEMS);
}
```

Et on ajoute une méthode pour pouvoir accéder au contenue de la collection depuis l'extérieur

```
public CommandWord getCommandWord(String pCommandWord)
{
   CommandWord vCommand = this.aValidCommands.get(pCommandWord);
    if(vCommand != null) {
        return vCommand;
    }
    else {
        return CommandWord.UNKNOWN;
    }
}
```

On modifie aussi la méthode isCommand pour obtenir ceci

```
public boolean isCommand( final String pString )
{
    return this.aValidCommands.containsKey(pString);
}
```

Dans les autres class tous les attributs et paramètre ou variable qui utiliser un Type String pour utiliser une commande doivent être changer en type CommandWord

Exercice 7.35.1 : Utiliser le switch …

Dans la méthode interpretCommand de la class GameEngine il est souhaité que nous passions à une structure plus efficace. C'est à dire utiliser des Switch a la place des if et else dans notre cas

```java
public void interpretCommand(String pCommandLine)
{
    this.aGui.println(pCommandLine);
    Command command = this.aParser.getCommand(pCommandLine);

    if(command.isUnknown()) {
        this.aGui.println("I don't know what you mean...");
        return;
    }

    CommandWord commandWord = command.getCommandWord();
    switch(commandWord){
        case HELP : printHelp(); break;
        case GO   : goRoom(command); break;
        case TEST : test(command); break;
        case TAKE : take(command); break;
        case DROP : drop(command); break;
        case LOOK : look(); break;
        case EAT  : eat(command); break;
        case BACK : back(); break;
        case ITEMS: items(); break;
        case QUIT : if(command.hasSecondWord())
                        this.aGui.println("Quit what?");
                    else
                        endGame();
                    break;
    }
}
```

Exercice 7.41.1 zuul-with-enums-v2 …

On modifie la class CommandWord comme demander, on obtient ceci

```java
public enum CommandWord
{
    // A value for each command word, plus one for unrecognised
    // commands.
    GO("go"), QUIT("quit"), HELP("help"), LOOK("look"), EAT("eat"), BACK("back"), TAKE("take"), DROP("drop"), ITEMS("items"), TEST("test"), UNKNOWN("?"), CHARGE("charge"), TP("tp");

    private String commandString;

    /**
     * Initialise with the corresponding command word.
     * @param commandString commandWord The command string.
     */
    CommandWord(String commandString){this.commandString = commandString;}

    /**
     * @return The command word as a string.
     */
    public String toString(){return commandString;}
}
```

Et j'améliore le constructeur de CommandWords pour ne plus avoir à le modifier à chaque fois que j'ajoute une commande

```java
public CommandWords()
{
    this.aValidCommands = new HashMap<String, CommandWord>();
    for(CommandWord command : CommandWord.values()){
        if(command != CommandWord.UNKNOWN)
            this.aValidCommands.put(command.toString(), command);
    }
}
```

Exercice 7.42 (time limit) :

J'ai ajouté un attribut qui correspond au nombre de commande traiter par la méthode interpretCommand si le nombre de commande interpréter est supérieur a 200 il est impossible de continuer a exécuté de commande.

```
private boolean time(){
    this.aCommantInput++;
    if(this.aCommantInput>200){
        this.aGui.println("You are out of time");
        this.aGui.enable(false);
        return true;
    }
    else return false;
}
```

```
public void interpretCommand(String pCommandLine)
{
    if(time())return;
   [...]
}
```

Exercice 7.43 (trap door) :

Le but est qu'une fois la sortie emprunter on ne plus peut plus faire marche arrière. Rien de bien compliquer il suffit de ne pas mettre cette direction de sortie dans la nouvelle pièce. Le problème est qu'il faut aussi empêcher la méthode back de retourner dans la pièce précédente. J'ai donc décidé d'ajouter une collection a la class Room qui contient toutes les sorties est un boolean qui indique si elles sont à sens unique ou non.

```
private HashMap<String, Boolean> aTrapDoor;
```

Et lors de la définition des sorties de la Room, j'ai ajouté un paramètre pour connaître la nature de la sortie (sens unique ou non).

```
public void setExit(final String pDirection, final Room pNeighbor, final boolean pTrapDoor)
{
    this.aExits.put(pDirection, pNeighbor);
    this.aTrapDoor.put(pDirection, pTrapDoor);
}
```

Et ajout d'une méthode pour connaître la nature de la sortie lorsque le joueur l'emprunte

```
public Boolean isTrapDoor(String pDirection){return this.aTrapDoor.get(pDirection);}
```

Il faut ajouter une méthode dans la class player pour vider la pile des pièces précédentes pour que le joueur ne puisse pas utiliser la commande back:

```
public void resetLastRoom(){this.aLastRooms.clear();}
```

Maintenant le dernière modification ce trouve la commande goRoom de GameEngine

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

    String vDirection = pCommand.getSecondWord();

    Room vNextRoom = this.aPlayer.getLocalisation().getExit(vDirection);

    if (vNextRoom == null) this.aGui.println("There is no door !");
    else
    {
        if(this.aPlayer.getLocalisation().isTrapDoor(vDirection)==true) this.aPlayer.resetLastRoom();
        else this.aPlayer.setLastRoom(this.aPlayer.getLocalisation());
        this.aPlayer.setLocalisation(vNextRoom);
        this.aGui.println(this.aPlayer.getLocalisation().getLongDescription());
        if(this.aPlayer.getLocalisation().getImageName() != null)
            this.aGui.showImage(this.aPlayer.getLocalisation().getImageName());
    }
}
```

Exercice 7.44 (beamer) :

Nous savons que le beamer est une sorte d'Item donc, beamer hérite de Item. Les seules différences entre un Item classique et un Beamer sont que le beamer a un état de charge et une pièce enregistrer lors de la charge

```
public class Beamer extends Item
{
    private boolean aCharge;
    private Room aChargeRoom;

    public Beamer(final String pDescription, final double pPrix, final double pPoids){
        super(pDescription,pPrix,pPoids);
        this.aCharge = false;
    }

    public Beamer(final String pDescription, final double pPrix, final double pPoids, final Room pChargeRoom){
        super(pDescription,pPrix,pPoids);
        this.aCharge = true;
        this.aChargeRoom = pChargeRoom;
    }
```

Ajout d'une méthode pour connaître l'état de charge du beamer

```
public boolean isCharged(){return this.aCharge;}
```

deux pour autres pour charger l'état du beamer

```
public void charge(final Room pChargeRoom){
    this.aCharge = true;
    this.aChargeRoom = pChargeRoom;
}
public void discharge(){this.aCharge = false;}
```

Et une dernière dans la class beamer pour connaître le lieu où le beamer a été charger

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

Nous allons maintenant ajouter deux nouvelles commandes charge et téléporte charge a pour but d'enregistrer l'emplacement actuel pour pouvoir une fois la commande téléporte taper s'y déplacer.

```
private void charge(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("Charge what ?");
        return;
    }

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

    if(vBeamer == null) this.aGui.println("I don't have it !");
    else{
        vBeamer.charge(this.aPlayer.getLocalisation());
        this.aGui.println("The beamer is charged");
    }
}
```

Et la commande téléporte, si le beamer est charger fait changer le player de Room

```
private void teleport(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("Teleport with what ?");
        return;
    }

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

    if(vBeamer == null) this.aGui.println("I don't have it !");
    else if(vBeamer.isCharged()==false) this.aGui.println("The beamer is not charged");
    else{
        vBeamer.discharge();
        this.aPlayer.resetLastRoom();
        this.aPlayer.setLocalisation(vBeamer.getChargeRoom());
        this.aGui.println(this.aPlayer.getLocalisation().getLongDescription());
        if(this.aPlayer.getLocalisation().getImageName() != null)
            this.aGui.showImage(this.aPlayer.getLocalisation().getImageName());
    }
}
```

Et faire tous les ajouts habituels lors de l'ajout d'une commande