Table des matières

Présentation du jeu

Auteur :

Baptiste Espinasse

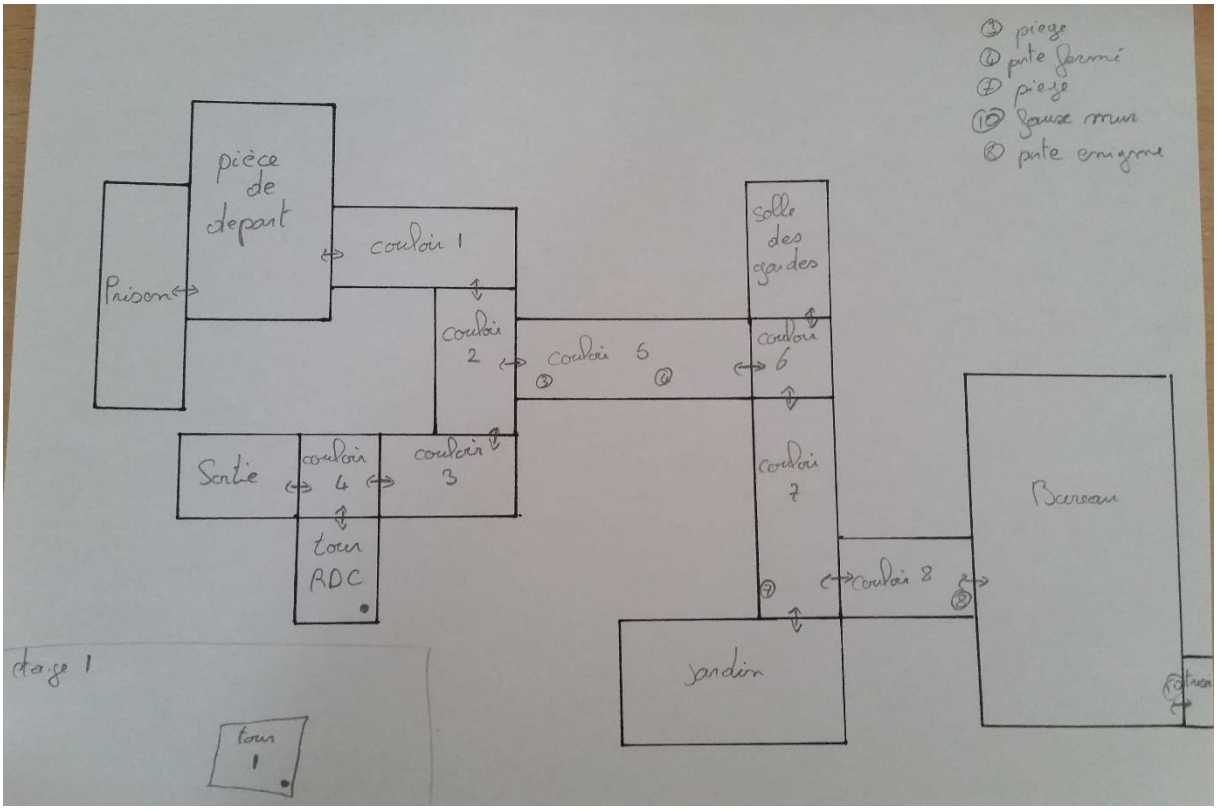
Thème :

Une équipe d'aventurier doivent 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 Objet 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).