

Rapport :

Titre provisoire + [lien page web](#)

(A3P(AL) 2015/2016 GQ)

I.A) Auteur(s)

I.B) Thème (phrase-thème validée)

I.C) Résumé du scénario (complet)

I.D) Plan (complet, avec indication de la partie "réduit" si exercice 7.3.3)

I.E) Scénario détaillé (complet, avec indication de la partie "réduit" si exercice 7.3.3)

I.F) Détail des lieux, items, personnages

I.G) Situations gagnantes et perdantes

I.H) Eventuellement énigmes, mini-jeux, combats, etc.

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

II. Réponses aux exercices (**à partir de l'exercice 7.5 inclus**)

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

IV. Déclaration obligatoire anti-plagiat (*)

V, VI, etc. [...](#) : tout ce que vous voulez en plus

Rapport

Samouraï on Moon

I.A)

Le jeu a été créé par PHAM Jimmy, élève en E3S à l'ESIEE Paris.

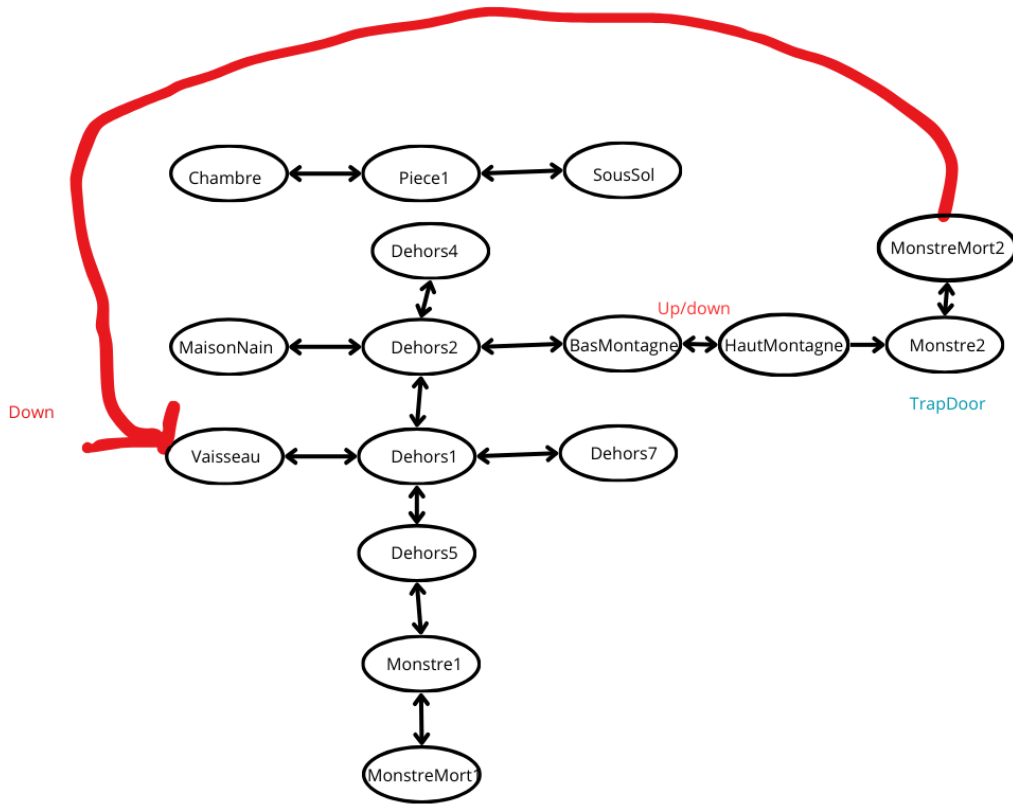
I.B)

Sur la lune Titan, Gintoki Hanma doit récupérer toutes les pièces mécaniques nécessaires afin de retourner sur Terre.

I.C)

Gintoki Hanma a endommagé le vaisseau pendant sa mission d'exploration sur Titan, il doit donc remplacer les pièces défectueuses en s'infiltrant au QG des Titanesques. Il y retrouvera des monstres et des défis qui pourront au mieux l'aider à trouver les pièces plus vite, au pire lui retirer des points de vie, jusqu'à arriver au Game Over.

I.D)



I.E)

Contexte

Gintoki Hanma, un mercenaire spatial et ancien soldat d'élite, fait partie d'une mission d'exploration pour récupérer des artefacts précieux cachés dans les profondeurs de Titan, la plus grande lune de Saturne. Cependant, lors de son arrivée, son vaisseau subit un grave accident à cause de conditions météorologiques extrêmes et des champs magnétiques anormaux de la planète. L'impact endommage les systèmes critiques de son vaisseau, le laissant immobilisé en territoire hostile. Coincé sur Titan, Gintoki doit trouver des pièces de rechange pour réparer son vaisseau avant que ses réserves d'oxygène et de ressources ne s'épuisent.

Objectif principal

Gintoki doit s'infiltrer dans le **QG des Titansques**, une ancienne civilisation extraterrestre qui domine la lune. Cette forteresse souterraine abrite non seulement les pièces dont il a besoin pour réparer son vaisseau, mais aussi de nombreux secrets technologiques et pièges mortels.

Environnement

Titan est une lune mystérieuse et dangereuse, avec des paysages variés allant des déserts glacés aux forêts de méthane liquide. L'intérieur du QG des Titanesques est un véritable labyrinthe de salles mécaniques, de couloirs étroits et de temples antiques. Chacune de ces zones est peuplée par le

Cerbère, et le titan originel créées par la civilisation Titanesque pour défendre leur savoir et leurs ressources.

Mission finale

Le but ultime de Gintoki est de récupérer 2 moteurs à propulsion pour réparer le vaisseau :

- 1 moteur à propulsion, caché au sous-sol du Monstr'Hotel. Il sera récupéré juste après avoir réussi le défi du nain.
- 1 dans la salle de trésor après avoir tué le titan originel

Conclusion

Une fois le module récupéré et le Gardien vaincu, Gintoki retourne à son vaisseau et décolle de la lune.

I.F) Détails des lieux, items, personnages

Détail des lieux :

- Le vaisseau est le lieu où il faudra ramener tous les objets de la quête principale. Il y aura également la lampe torche à disposition au tout début du jeu
- le sous-sol du monstr'Hotel sera le lieu où l'on pourra récupérer le moteur à propulsion et une épée titanesque.
- La maison du Nain abrite des chaussures boostées qui permettront d'escalader la montagne, là où se trouve le QG des Titanesques.
- Au niveau de dehors2 se trouve le nain en question.
- La pièce2 une chambre du Monstr'Hotel
- Dehors7 est la pièce où Gintoki pourra trouver sa première arme, une épée émoussée.
- Monstre1 se trouve être la pièce du premier monstre, qui ne peut seulement être tué avec une épée améliorée par le nain
- MonstreMort1 est la salle de trésors après avoir tué le Cerbère. On y trouve un MagicCookie et un livre d'incantation.
- Monstre2 est l'antre du titan originel. Il faut une épée titanesque pour pouvoir le tuer.
- MonstreMort2 est la salle de trésors après avoir tué le titan originel. On y trouve un le moteur à propulsion cassé.

Items et amélioration

Gintoki commence l'aventure avec aucun item.

Les items recensés sont :

- 2 Moteur à propulsion
- Epée émoussée
- épée titanesque
- Chaussures boostées
- Livre d'incantation
- lampe
- magic Cookie

Les personnages :

- Héros : Gintoki Hanma
- Nain
- Cerbère
- Titan Originel

I.G) Situation gagnante, perdante

Gagné : revenir au vaisseau avec les 2 moteurs à propulsion et le livre d'incantation

Perdu : plus d'oxygène ou manque les items nécessaires pour tuer les monstres

I.H) Enigmes, mini jeux, combats

- Pour éliminer les monstres, il faudra récupérer les items nécessaires.

Il faut une épée pour tuer le Cerbère, et l'épée titanesque pour tuer le titan originel.

I.I) je n'arrive pas à enlever les messages « i don't know what you mean.. » après l'appui d'un des boutons autre que down

II.)

EXO 7.5

On va créer une nouvelle procédure privée printLocationInfo dans la classe Game pour pouvoir les appeler dans les procédures goRoom et printWelcome afin d'éviter la duplication de code .

EXO 7.6

On écrit la procédure getExit dans la classe Room, on l'appelle dans la procédure goRoom de la classe Game après avoir vérifié que l'attribut aSecondWord de pCommand existe, et on la stocke dans vNextRoom.

EXO 7.7

Dans printLocationInfo, on ne peut plus accéder aux attributs de direction puisqu'ils sont privés, donc au lieu de vérifier si les attributs ne sont pas initialisés, on appelle directement la fonction getExit avec la direction associée en paramètres pour vérifier s'il existe une sortie de la Room courante.

Pour la création de la fonction getExitString, on crée instancie une room et une string a chaque direction de la room courante, et on retourne la concaténation des 4 strings, qui valent elles-mêmes la concaténation d'une string vide (et de la room associée si elle est non null).

il est logique de demander à Room de produire les informations sur ses sorties (et ne pas lui demander de les afficher), et de demander à Game d'afficher ces informations (et ne pas lui demander de les produire) car Room est la classe la mieux placée pour donner une information sur un de ces objets. Il ne faut pas que Room et Game s'échangent les rôles.

EXO 7.8

Désormais, la fonction getExitString devient inutile car

EXO 7.8.1

Dans cette partie, on va rajouter dans la procédure setExits de la classe Room les paramètres pDownExit et pUpExit de la manière suivante :

```

public void setExits(
    final Room pNorthExit,
    final Room pEastExit,
    final Room pSouthExit,
    final Room pWestExit,
    final Room pUpExit,
    final Room pDownExit)
{
    if (pNorthExit!=null){
        exits.put("North", pNorthExit);
    }
    if (pEastExit!=null){
        exits.put("East", pEastExit);
    }
    if (pSouthExit!=null){
        exits.put("South", pSouthExit);
    }
    if (pWestExit!=null){
        exits.put("West", pWestExit);
    }
    if (pUpExit!=null){
        exits.put("Up", pUpExit);
    }
    if (pDownExit!=null){
        exits.put("Down", pDownExit);
    }
}

```

De plus, on va rajouter les sorties « up » et « down » dans la fonction getExitString :

```

public String getExitString()
{
    String vExits = "Exits: ";
    if (this.getExit("North") != null) {
        vExits+="North ";
    }
    if (this.getExit("East") != null) {
        vExits+= "East ";
    }
    if (this.getExit("South") != null) {
        vExits+= "South ";
    }
    if (this.getExit("West") != null) {
        vExits+="West ";
    }
    if (this.getExit("Up") != null) {
        vExits+="Up ";
    }
    if (this.getExit("Down") != null) {
        vExits+="Down ";
    }
    return vExits;
}

```

// Room

Enfin, on va modifier dans la classe Game les Exits des room :

```

vVaisseau.setExits(null, vDehors1, null, null,null,null);
vDehors1.setExits(vDehors2, vDehors7, vDehors5, vVaisseau,null,null);
vDehors2.setExits(vDehors4, vBasMontagne,vDehors1 , vMaisonNain,null,null)
vMaisonNain.setExits(null, vDehors2, null, null,null,null);
vBasMontagne.setExits(null, null, null, vDehors2,vHautMontagne,null); |
vDehors4.setExits(vPiece1, null, vDehors2, null,null,vBasMontagne);
vPiece1.setExits(null, vSousSol, vDehors4, vChambre,null,null);
vChambre.setExits(null, vPiece1, null, null,null,null);
vSousSol.setExits(null, null, null, vPiece1,null,null);
vDehors7.setExits(null, null, null, vDehors1,null,null);
vDehors5.setExits(vDehors1, null, vMonstre1, null,null,null);

this.aCurrentRoom=vVaisseau;

```

EXO 7.9

Dans la classe Room, la fonction keySet appelée dans la méthode getExitString permet d'encenser toutes les sorties de la room courante sous forme de clés, dont chacune d'elles est associée à une valeur et n'apparaît qu'une seule fois. Ici, keySet prendra pour afin de pouvoir créer une boucle qui va concaténer la variable locale String composée de « Exits : » suivie de chaque indice de la liste.

EXO 7.10

Dans la méthode « getExitString », on déclare et initialise une variable nommée « vReturnString » pour stocker une chaîne de caractères finale qui sera retournée. On déclare ensuite une nouvelle variable « vKeys » qui contiendra l'ensemble des clés de la HashMap, où chaque clé ne peut apparaître qu'une seule fois. Ici les clés correspondent aux sorties. Ainsi grâce à la boucle « for each », on parcourt la collection de sorties contenue dans « vKeys », à chaque boucle une variable « vExit » va contenir une sortie différente qui viendra s'ajouter à « vReturnString ». A la fin de la boucle, on obtient la chaîne de caractères finale dans « vReturnString » qui correspondra à l'ensemble des sorties de la pièce actuelle.

```

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

```

EXO 7.11

Dans cet exercice nous avons ajouté la méthode « getLongDescription » dans la classe « Room » pour pouvoir l'appeler dans « printLocationInfo » de la classe « Game » qui nous permet d'alléger le code.

EXO 7.14

La méthode « look » appelle la fonction « getLongDescription » et permet de connaître notre position à n'importe quel moment dans le jeu en donnant la salle dans laquelle on se situe en plus des différentes sorties disponibles.

```
/**
 * affiche la description de la room courante.
 */
private void look()
{
    System.out.println(this.aCurrentRoom.getLongDescription());
}
```

Nous pouvons ensuite ajouter cette commande à la liste des commandes valides dans la classe « CommandWords ».

Enfin, nous ajouterons un nouveau cas dans la méthode « processCommand » qui appelle « look » pour que l'on puisse utiliser cette méthode lorsqu'on la tape.

EXO 7.15

Dans cet exercice, on implémente une nouvelle commande « eat », qui nous permet seulement d'afficher l'état de faim du personnage.

```
/**
 * Cette méthode permet d'afficher que le joueur a mangé.
 */
private void eat()
{
    System.out.println("Vous venez de manger et vous n'avez plus faim dès à présent.");
}
```

On ajoute également la commande « eat » dans la liste de commande de la classe « CommandWords » et dans la méthode processCommand de la classe Game.

EXO 7.16

Dans cet exercice, on implémente une nouvelle méthode « showAll » dans la classe « CommandWords » qui nous permet d'afficher toutes les commandes valides.

```
/**
 * Retourne un mot contenant toute les commandes
 */
public String getAll()
{
    String vCommand= "";
    for(String command : this.aValidCommands)
    {vCommand=vCommand+command+" ";}
    return vCommand;
}
```


On implémente ensuite une méthode « ShowCommands » dans « Parser » :

```
/**
 * Affiche les commandes utilisables.
 */
public void showCommands()
{
    System.out.println(this.aValidCommands.getAll());
}
```

EXO 7.18

Pour que « CommandWords » n'ait plus à afficher les commandes valides, il faut remplacer « showAll » par « getCommandList » :

```
/**
 * Retourne un String contenant toute les commandes.
 */
public String getCommandList()
{
    String vCommandList= "";
    for(String vCommand : this.aValidCommands){
        vCommandList+=vCommand+" ";
    }
    return vCommandList;
}
```

Dans la classe « Parser », la procédure « showCommands » devient la méthode « getCommands » qui retourne une String en faisant appel à la fonction « getCommandList » :

```
/**
 * retourne les commandes utilisables.
 */
public String getCommands()
{
    return this.aValidCommands.getCommandList();
}
```

Il faut ensuite implémenter les modifications dans « printHelp » de la classe « Game ».

```

/**
 * Affiche le message d'aide lorsqu'on tape help.
 */
private void printHelp()
{
    System.out.println("Vous êtes perdu. Vous êtes seul.");
    this.printLocationInfo();
    System.out.println("");
    System.out.println("Vos commandes sont:" + this.aParser.getCommands());
}

```

Ainsi, nous avons ajouté toutes ces modifications pour pouvoir produire l'information au plus près et l'utiliser ailleurs.

La classe « CommandWords » n'est donc plus responsable de l'affichage de la liste des commandes mais seulement de sa production. La classe « Parser » va la transmettre pour permettre à « Game » de l'afficher.

J'ai ajouté les images suivantes à mon jeu :



Le titre de mon jeu est « Samourai on Moon ».

Dans la suite de l'exercice (7.18.6), on ajoute l'attribut `alimageName` dans la classe `Room` qui représente le nom de l'image de la room courante, qu'on initialise dans le constructeur et on y ajoute son accesseur :

```

private String aDescription;
private HashMap<String, Room> aExits;
private String aImageName;
// constructeur qui initialise les attributs
/**
 * constructeur qui initialise les attributs
 */public Room(final String pDescription, final String pImage)
{
    this.aDescription=pDescription;
    this.aExits = new HashMap<String, Room>();
    this.aImageName = pImage;
}

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

```

Ainsi, on n'oublie pas de modifier la méthode « createRooms » dans GameEngine lorsqu'on crée les différentes pièces du jeu.

On modifie la méthode getCommand de la classe CommandWords :

```
public Command getCommand(final String pInputLine)
{
    String vWord1;
    String vWord2;

    StringTokenizer tokenizer = new StringTokenizer( pInputLine );

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

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

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

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

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

On incorpore aux codes :

- la classe UserInterface qui correspondra à l'interface graphique du jeu.
- La classe GameEngine qui reprend la majorité de la classe Game. On y modifie la méthode processCommand qui devient interpretCommand et on ajoute un attribut aGui et son accesseur qui permettront d'afficher sur l'interface ce que l'on souhaite. Ainsi, on peut remplacer tous les « System.out » en « this.aGui » :

```

public void interpretCommand(final String pCommandLine)
{
    this.aGui.println( "> " + pCommandLine );
    Command vCommand = this.aParser.getCommand( pCommandLine );

    if(vCommand.isUnknown())
    {
        this.aGui.println("I don't know what you mean...");
        return;
    }
    String vCom=vCommand.getCommandWord();
    if(vCom.equals("help"))
    {
        this.printHelp();
    }
    if(vCom.equals("go"))
    {
        this.goRoom(vCommand);
    }
    if(vCom.equals("quit"))
    {
        if(vCommand.hasSecondWord())
        {
            this.aGui.println("Quit what?");
        }
        else
        {
            this.endGame();
        }
    }
}

```

```

private UserInterface aGui;

```

```

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

```

```

private void printLocationInfo()
{
    this.aGui.println(this.aCurrentRoom.getLongDescription());
    if(this.aCurrentRoom.getImageName()!=null)
    {
        this.aGui.showImage(this.aCurrentRoom.getImageName());
    }
}

```

Ici, la méthode printLocationInfo a été modifiée afin d'afficher l'image de la room courante.

Pour pouvoir automatiser le redimensionnement des images, j'ai modifié la méthode showImage de la classe UserInterface en mettant les lignes 82-83 comme conseillé dans les réponses aux exercices.

```
70  /**
71  * Show an image file in the interface.
72  * @param pImageName correspond au nom de l'image
73  */
74  public void showImage( final String pImageName )
75  {
76      String vImagePath = "Photo/" + pImageName; // to change the directory
77      URL vImageURL = this.getClass().getClassLoader().getResource( vImagePath );
78      if ( vImageURL == null )
79          System.out.println( "Image not found : " + vImagePath );
80      else {
81          ImageIcon vIcon = new ImageIcon( vImageURL );
82          this.aImage.setIcon( new ImageIcon(
83              vIcon.getImage().getScaledInstance(640,400,java.awt.Image.SCALE_SMOOTH))
84              this.aMyFrame.pack();
85      }
86  } // showImage(.)
```

Finalement, la classe Game ne garde seulement son constructeur :

```
public class Game
{
    private UserInterface aGui;
    private GameEngine aEngine;

    /**
     * Create the game and initialise its internal map. Create the interface
     */
    public Game()
    {
        this.aEngine = new GameEngine();
        this.aGui = new UserInterface( this.aEngine );
        this.aEngine.setGUI( this.aGui );
    }

    } // Game
```

On importe les librairies suivantes dans la classe interface :

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

On crée 6 nouveaux attributs qui correspondent au déplacement du joueur :

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

On les crée par la suite dans la méthode createGUI :

```
private void createGUI()
{
    this.aButtonNorth = new JButton("North");
    this.aButtonSouth = new JButton("South");
    this.aButtonEast = new JButton("East");
    this.aButtonWest = new JButton("West");
    this.aButtonUp = new JButton("Up");
    this.aButtonDown = new JButton("Down");
}
```


On doit à présent les ajouter dans un panel :

```
JPanel vPanelButton = new JPanel();
vPanelButton.setLayout( new BorderLayout() );
JPanel vPanel2 = new JPanel();
vPanel2.setLayout( new BorderLayout() );

vPanel.add(vPanelButton, BorderLayout.EAST);
vPanelButton.add( this.aButtonNorth, BorderLayout.NORTH );
vPanelButton.add( this.aButtonSouth, BorderLayout.SOUTH );
vPanelButton.add( this.aButtonEast, BorderLayout.EAST );
vPanelButton.add( this.aButtonWest, BorderLayout.WEST );
vPanelButton.add( vPanel2, BorderLayout.CENTER );
vPanel2.add( this.aButtonUp, BorderLayout.CENTER );
vPanel2.add( this.aButtonDown, BorderLayout.SOUTH );
```

On finit par les ajouter dans les instructions addActionListener :

```
this.aEntryField.addActionListener( this );
this.aButtonNorth.addActionListener( this );
this.aButtonSouth.addActionListener( this );
this.aButtonEast.addActionListener( this );
this.aButtonWest.addActionListener( this );
this.aButtonUp.addActionListener( this );
this.aButtonDown.addActionListener( this );
```

Pour que les boutons soient bien fonctionnels, on utilise la méthode actionPerformed, qui en fonction de l'action qu'il reçoit en paramètre va la comparer avec l'action que provoque un bouton pour ainsi interpréter une commande et l'exécuter comme si l'on écrivait cette commande.

```

// no need to check the type of action at the moment
// because there is only one possible action (text input
if (this.aButtonNorth.equals(pE.getSource()))
    this.aEngine.interpretCommand("go North");
if (this.aButtonSouth.equals(pE.getSource()))
    this.aEngine.interpretCommand("go South");
if (this.aButtonEast.equals(pE.getSource()))
    this.aEngine.interpretCommand("go East");
if (this.aButtonWest.equals(pE.getSource()))
    this.aEngine.interpretCommand("go West");
if (this.aButtonUp.equals(pE.getSource()))
    this.aEngine.interpretCommand("go Up");
if (this.aButtonDown.equals(pE.getSource()))
    this.aEngine.interpretCommand("go Down");
else
    this.processCommand(); // never suppress this line
} // actionPerformed(.)

```

EXO 19

On commence l'exercice par déplacer toutes nos images dans un dossier « Photo » placé à la racine du projet. Dans la méthode showImage de UserInterface, on modifie la ligne 76 pour mettre le chemin du répertoire Photo :

```

74 public void showImage( final String pImageName )
75 {
76     String vImagePath = "Photo/" + pImageName; // to change the directory
77     URL vImageURL = this.getClass().getClassLoader().getResource( vImageName );
78     if ( vImageURL == null )
79         System.out.println( "Image not found : " + vImagePath );
80     else {
81         ImageIcon vIcon = new ImageIcon( vImageURL );
82         this.aImage.setIcon( new ImageIcon(
83             vIcon.getImage().getScaledInstance(640,400,java.awt.Image.SCALE_SMOOTH) );
84         this.aMyFrame.pack();
85     }
86 } // showImage(.)

```

Exo 20

Dans cet exercice on va rajouter des items dans le jeu : on commence par créer la classe Item, dotée de 3 attributs :

```

private String aName;
private String aDescription;
private double aWeight;

```


On les initialise dans le constructeur, on crée les 3 accesseurs et 2 getters permettant d'avoir la description d'un item :

```
/**
 * @return le nom, la description et le poids de l'item.
 */
public String getLongItemDescription()
{
    // initialisation des variables d'instance
    return this.aName+ " (" +this.aDescription+", "+this.aWeight+");"
}

/**
 * @return le nom et le poids de l'item.
 */
public String getItemDescription()
{
    // initialisation des variables d'instance
    return this.aName+ " (" +this.aWeight+");"
}
```

On ajoute un attribut Item aItem dans la classe Room pour pouvoir placer des items dans des pièces :

```
private Item aItem;
```

On initialise l'attribut dans une méthode setItem et non dans le constructeur, car cela imposerait que la pièce soit caractérisé par un item alors que l'on veut seulement placer l'item dans la pièce. On crée également l'accesseur associé :

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

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

On crée une méthode getItemString qui nous renvoie la présence ou nom d'un item dans la pièce :

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

On appelle getItemString dans getLongDescription de la manière suivante :

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

Exo 21

Cet exercice permet de modifier la méthode look de GameEngine : il aura un paramètre Command pCommand. On vérifie si pCommand possède un deuxième mot : si oui, on vérifie que l'item demandé se trouve bien dans la pièce actuelle sinon on retourne la description longue de la room courante dans ce cas et dans le cas où il n'y a pas de second mot.

```
public void look(final Command pCommand)
{
    if (pCommand.hasSecondWord())
    {
        String vSecondWord=pCommand.getSecondWord();
        Item vItem=this.aCurrentRoom.getItem(vSecondWord);
        if (vItem==null)
        {
            this.aGui.println("This item is not in this room or does not exist");
            return;
        }
        else
        {
            this.aGui.println(vItem.getLongItemDescription());
            return;
        }
    }
    else
    {
        this.aGui.println(this.aCurrentRoom.getLongDescription());
    }
}
```

On modifie par la suite `interpretCommand` pour que la commande soit transmise à la méthode `look` :

```
if(vCom.equals("look"))
{
    this.look(vCommand);
}
```

EXO 22

On aimerait qu'il y ait désormais plusieurs items dans une pièce. On peut alors utiliser une `HashMap` alternem qui sera un attribut de la classe `Room`, qu'on vient initialiser dans le constructeur naturel.

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

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

On modifie les méthodes suivantes comme montré ici :

```
/**
 * Initialise l'item de la pièce
 */
public void addItem(final String pName, final Item pItem)
{
    this.aItems.put(pName, pItem);
}
```

```

/**
 * retourne l'item de la pièce
 */
public Item getItem(final Item pItem)
{
    return this.aItems.get(pItem);
}

```

On modifie maintenant les méthodes getItemString et look :

```

/**
 * retourne la description de l'item dans la pièce s'il y en a un
 */
public String getItemString()
{
    if (this.aItems.isEmpty())
        return "No item here";
    String vReturnString= "Items :";
    Set<String> vKeys = this.aItems.keySet();
    for (String vName : vKeys)
        vReturnString+= ' ' + vName;
    return vReturnString;
}

public void look(final Command pCommand)
{
    if (pCommand.hasSecondWord())
    {
        String vSecondWord=pCommand.getSecondWord();
        Item vItem=this.aCurrentRoom.getItem(vSecondWord);
        if (vItem==null)
        {
            this.aGui.println("This item is not in this room or does not exist");
            return;
        }
        else
        {
            this.aGui.println(vItem.getLongItemDescription());
            return;
        }
    }
    else
    {
        this.aGui.println(this.aCurrentRoom.getLongDescription());
    }
}

```

J'ai créé les items de mon jeu puis inséré dans différentes Room dans la méthode createRoom comme ceci :

```

Item vLampe = new Item("Lampe", "Pour éclairer les chemins sombres",1);
Item vChaussures = new Item("Chaussures boostées", "permet de gravir"+
" des montagnes", 1);
Item vSabreNul = new Item("Sabre émoussé", "permet de tuer le " +
"premier monstre seulement", 1);
Item vSabreFort = new Item("Sabre Titanesque", "permet de tuer le "+
"gardien Titanesque", 2);
Item vLivre = new Item("Livre d'incantation", "permet de donner "+
"un second souffle à un moteur de propulsion cassé", 2);
Item vMoteur1 = new Item("Moteur à propulsion cassé", "Moteur qui"+
" aurait permis à Gintoki de repartir, mais le moteur est cassé !" +
" Peut-être y a-t-il un item qui lui rendrait la vie..", 2);
Item vMoteur2 = new Item("Moteur à propulsion boosté", "Bravo ! " +
"le premier moteur a été trouvé ! trouvez le deuxième pour pouvoir"+
" partir de la lune", 2);

vVaisseau.addItem("Lampe", vLampe);
vDehors7.addItem("Sabre émoussé", vSabreNul);
vMaisonNain.addItem("Livre d'incantation", vLivre);
vMonstreMort.addItem("Chaussures boostées", vChaussures);
vSousSol.addItem("Sabre Titanesque", vSabreFort);
vSousSol.addItem("Moteur à propulsion boosté", vMoteur2);
vMonstreMort2.addItem("Moteur à propulsion cassé", vMoteur1);

```

EXO 23

On souhaite ajouter une nouvelle commande « back » qui nous permet de revenir dans la pièce précédente.

On ajoute « back » dans le tableau de commandes valides :

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

On crée un nouvel attribut Room aPreviousRoom dans la classe GameEngine, initialisée dans goRoom :

```
private Room aPreviousRoom;

this.aPreviousRoom=this.aCurrentRoom;
```

On crée alors la méthode back dans GameEngine :

```
private void back(final Command pCommand)
{
    if (pCommand.hasSecondWord()){
        this.aGui.println("You can't back in a particular room !");
        return;
    }

    if(this.aPreviousRoom == null){
        this.aGui.println("There is no previous room");
        return;
    }
    this.aCurrentRoom = this.aPreviousRoom;
    this.printLocationInfo();
}
```

La méthode fonctionne de la manière suivante : On ne peut pas retourner dans une Room désignée, donc on ne veut pas de 2^e mot, le cas contraire on affiche un message d'erreur ; si la pièce précédente

n'existe pas on affiche un autre message d'erreur sinon on assigne à la room courante la room précédente et on appelle printLocationInfo.

On termine par rajouter la condition suivante dans interpretCommand :

```
public void interpretCommand( final String pCommandLine )
{
    this.aGui.println( "> " + pCommandLine );
    Command vCommand = this.aParser.getCommand( pCommandLine );

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

    String vCommandWord = vCommand.getCommandWord();
    if ( vCommandWord.equals( "help" ) )
        this.printHelp();
    else if ( vCommandWord.equals( "go" ) )
        this.goRoom( vCommand );
    else if( vCommandWord.equals("back"))
        this.back(vCommand);
    else if ( vCommandWord.equals( "quit" ) ) {
        if ( vCommand.hasSecondWord() )
            this.aGui.println( "Quit what?" );
        else
            this.endGame();
    }
}
```

EXO 26

On aimerait améliorer la méthode back pour pouvoir l'utiliser plusieurs fois d'affilée. On utilise pour ce faire un Stack, correspondant à une pile d'objets.

Dans la classe GameEngine, on importe la librairie, on modifie le type de l'attribut aPreviousRoom et on l'initialise comme ceci :

```
import java.util.Stack;

/**
 *
 * This class creates all rooms, creates the parser and
 * the game. It also evaluates and executes the commands
 * the parser returns.
 *
 * @author Pham Jimmy
 * @version 7.26
 */
public class GameEngine
{
    private Parser      aParser;
    private Room        aCurrentRoom;
    private UserInterface aGui;
    private Stack<Room> aPreviousRoom;

    /**
     * Constructor for objects of class GameEngine
     */
    public GameEngine()
    {
        this.aParser = new Parser();
        this.createRooms();
        this.aPreviousRoom = new Stack<Room>();
    }
}
```

On peut maintenant modifier les méthodes back et goRoom :

```

private void goRoom( final Command pCommand )
{
    if ( ! pCommand.hasSecondWord() ) {
        // if there is no second word, we don't know where to go...
        this.aGui.println( "Go where?" );
        return;
    }

    String vDirection = pCommand.getSecondWord();

    // Try to leave current room.
    Room vNextRoom = this.aCurrentRoom.getExit( vDirection );

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

```

```

private void back( final Command pCommand )
{
    if ( pCommand.hasSecondWord() ) {
        this.aGui.println( "You can't back in a particular room !" );
        return;
    }

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

    this.aCurrentRoom = this.aPreviousRoom.pop();
    this.printLocationInfo();
}

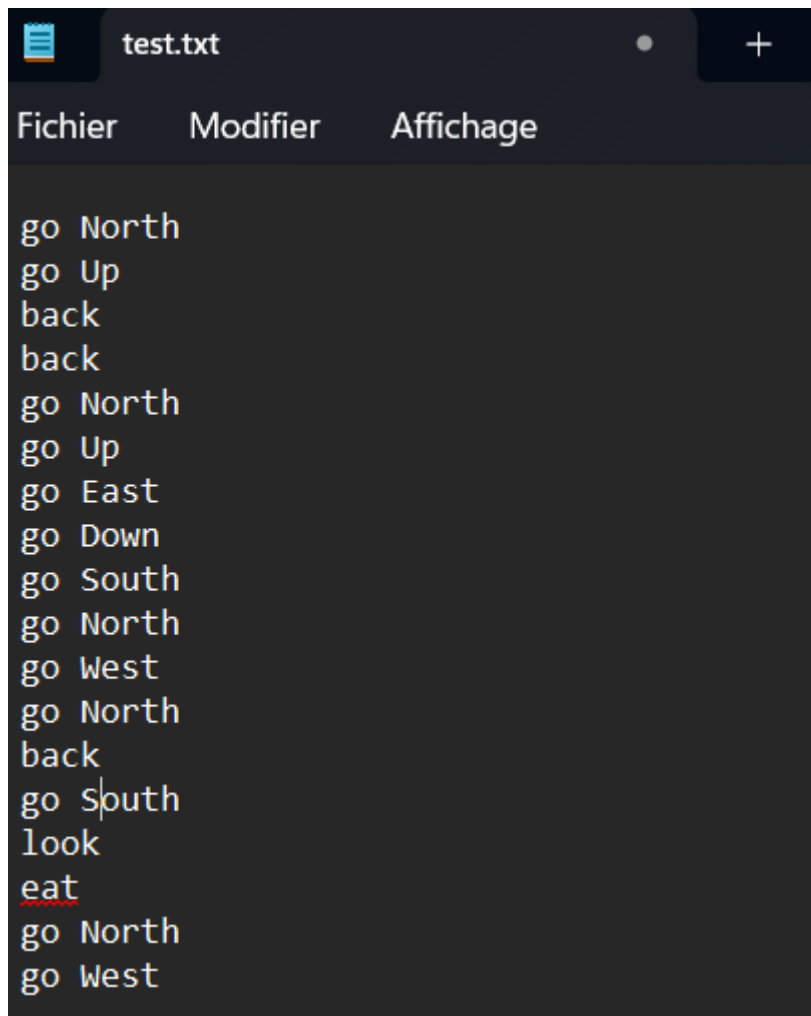
```


EXO 28

On crée une nouvelle méthode test dans la classe GameEngine, qui teste une succession de commande enregistrée dans un fichier. On importe les librairies suivantes :

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

private void test(final Command pCommand){
    if (!pCommand.hasSecondWord()){
        this.aGui.println("What do you want to test ?");
        return;
    }
    String vFile = pCommand.getSecondWord();
    try{
        Scanner vScan = new Scanner(new File(""+vFile+".txt"));
        this.aGui.println("Testing "+vFile+" ...");
        while (vScan.hasNextLine())
            this.interpretCommand(vScan.nextLine());
    }
    catch(final FileNotFoundException pE){
        this.aGui.println("There is no such file of testing");
    }
}
```



```
test.txt
Fichier  Modifier  Affichage

go North
go Up
back
back
go North
go Up
go East
go Down
go South
go North
go West
go North
back
go South
look
eat
go North
go West
```

EXO 29

On crée une nouvelle classe Player qui contiendra tout ce qui concerne le joueur et les différentes actions possibles. Ça permettra de décharger la classe GameEngine en déplaçant quelques attributs dans Player.

On y ajoute d'ailleurs le nom du personnage, la pièce courante et les pièces précédentes.

```
private String aPseudo;
private Room aCurrentRoom;
private Stack <Room> aPreviousRoom;
```

On initialise les attributs dans le constructeur :

```
public Player(final String pPseudo, final Room pCurrentRoom)
{
    this.aPseudo = pPseudo;
    this.aCurrentRoom = pCurrentRoom;
    this.aPreviousRoom = new Stack<Room>();
}
```

On crée à chaque attribut leurs getters correspondants, puis on implémente les méthodes de déplacement `executeGoRoom` et `executeGoBack` :

```
/**
 * procédure qui donne la prochaine pièce dans laquelle le joueur se déplace
 * @param pRoom correspond à la prochaine pièce
 */
public void executeGoRoom(final Room pRoom)
{
    this.aPreviousRoom.push(this.aCurrentRoom);
    this.aCurrentRoom = pRoom;
}
```

```
/**
 * procédure qui déplace le joueur en arrière
 */
public void executeGoBack()
{
    Room vBackRoom = this.aPreviousRoom.pop();
    this.aCurrentRoom = vBackRoom;
}
```

On ajoute à `GameEngine` l'attribut `aPlayer` de type `Player`, qu'on initialise dans `createRooms` :

```
this.aPlayer = new Player("Gintoki Hanma", vVaisseau);
```

On peut alors appeler ces nouvelles méthodes dans `GameEngine` :

```
private void goRoom( final Command pCommand )
{
    if ( ! pCommand.hasSecondWord() ) {
        // if there is no second word, we don't know where to go...
        this.aGui.println( "Go where?" );
        return;
    }

    String vDirection = pCommand.getSecondWord();
    Room vNextRoom = this.aPlayer.getCurrentRoom().getExit( vDirection );

    if ( vNextRoom == null )
        this.aGui.println( "There is no door!" );
    else {
        this.aPlayer.executeGoRoom(vNextRoom);
        this.printLocationInfo();
    }
}
```

d

```
/**
 * procédure qui permet de revenir dans la pièce précédente
 * @param pCommand correspond à la commande entrée par l'utilisateur
 */
private void back(final Command pCommand)
{
    if (pCommand.hasSecondWord()){
        this.aGui.println("You can't back in a particular room !");
        return;
    }

    if(this.aPlayer.getPreviousRoom().empty()){
        this.aGui.println("There is no previous room");
        return;
    }
    this.aPlayer.executeGoBack();
    this.printLocationInfo();
}
```

On modifie le reste du code pour appeler les accesseurs des room de l'attribut aPlayer comme dans printLocationInfo ici :

```
private void printLocationInfo()
{
    this.aGui.println(this.aPlayer.getCurrentRoom().getLongDescription());
    if(this.aPlayer.getCurrentRoom().getImageName()!=null)
    {
        this.aGui.showImage(this.aPlayer.getCurrentRoom().getImageName());
    }
}
```

EXO 30

On souhaite créer les commandes take et drop pour récupérer ou lâcher un item.

On commence par ajouter les commandes dans le tableau de commandes valides. On crée un nouvel attribut aPickedItem dans Player qui correspond à l'item qu'on souhaite récupérer, puis on l'ajoute dans le constructeur.

On implémente alors la méthode take :

```
private void take(final String pItemName)
{
    Item vItem = this.aPlayer.getCurrentRoom().getItem(pItemName);
    if(vItem==null)
    {
        this.aGui.println("No such Item in current room");
    }
    else{
        // if((this.aPlayer.pickedItemsWeight() + vItem.getItemWe
        // {
        this.aPlayer.executeTake(vItem);
        this.aPlayer.getCurrentRoom().removeItem(pItemName);
        this.aGui.println("You Picked "+pItemName);
    }
}

public void executeTake(final Item pItem)
{
    this.aPickedItems.addItem(pItem);
}
```

La méthode drop suit le même principe :

```
private void drop(final String pItemName)
{
    Item vItem = this.aPlayer.getPickedItem(pItemName);
    if(vItem==null)
    {
        this.aGui.println("You didn't pick up such Item");
    }
    else{
        this.aPlayer.executeDrop(pItemName);
        this.aPlayer.getCurrentRoom().addItem(vItem);
        this.aGui.println("You dropped "+pItemName);
    }
}
```

```
public void executeDrop(final String pItemName)
{
    this.aPickedItems.removeItem(pItemName);
}
```

On termine par ajouter les instructions take et drop dans interpretCommand.

EXO 31

On aimerait posséder plusieurs items à la fois. L'utilisation d'une HashMap est donc pertinente : on crée alors l'attribut aItemsBag dans la classe Player :

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

On y crée également les getters suivant, permettant de retourner un item de aItemsBag ou une String listant tous les objets de ce dernier :

```
public Item getItemsBag(final String pItemName)
{
    return this.aItemsBag.get(pItemName);
}

/**
 * @return qui donne tous les items trouvables dans le sac
 */
public String getBagString()
{
    if(this.aItemsBag.isEmpty())
    {
        return "Your bag is empty";
    }
    String vReturnString = "ItemsBag :";
    Set<String> vKeys = this.aItemsBag.keySet();
    for(String vName : vKeys)
    {
        vReturnString+=' ' + vName;
    }
    return vReturnString;
}
```


On peut alors modifier les méthodes executeTake et executeDrop de la classe Player :

```
public void executeTake(final Item pItem)
{
    this.aItemsBag.put(pItem.getName(), pItem);
    this.aCurrentRoom.removeItem(pItem.getName(), pItem);
}

public void executeDrop(final Item pItem)
{
    this.aCurrentRoom.addItem(pItem.getName(), pItem);
    this.aItemsBag.remove(pItem.getName(), pItem);
}
```

On modifie les méthodes take et drop de la classe GameEngine :

```
**
* procédure qui permet de récupérer l'item
*/
private void take(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("What do you want to take ?");
    }
    else if (pCommand.hasSecondWord())
    {
        String vTake=pCommand.getSecondWord();
        Item vItem=this.aPlayer.getCurrentRoom().getItem(vTake);
        if (vItem==null)
        {
            this.aGui.println("This item is not in this room or does not exist");
            return;
        }
        else if (vItem.getWeight()+this.aPlayer.getInventoryWeight()>this.aPlayer.getMaxWeight())
        {
            this.aGui.println("this item is too heavy for the bag");
        }
        else
        {
            this.aPlayer.executeTake(vItem);
            this.aGui.println("You have taken "+vTake);
        }
        this.aGui.println(this.aPlayer.getBagItems().getBagString());
    }
}
```

```

/**
 * procédure qui permet de poser l'item
 */
private void drop(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("What do you want to drop ?");
    }
    else if (pCommand.hasSecondWord())
    {
        String vDrop=pCommand.getSecondWord();
        Item vItem=this.aPlayer.getBagItems().getItem(vDrop);
        if (vItem==null)
        {
            this.aGui.println("This item is not in your inventory or does not exist");
            return;
        }
        else
        {
            this.aPlayer.executeDrop(this.aPlayer.getBagItems().getItem(vDrop));
            this.aGui.println("You have dropped "+vDrop);
        }
        this.aGui.println(this.aPlayer.getBagItems().getBagString());
    }
}

```

On crée une nouvelle classe ItemList qui nous permet de gérer plus facilement les items, puisqu'ils étaient dupliqués dans les classes Room et Player.

La classe est dotée des 2 attributs suivants :

```

private HashMap <String, Item> aItemsBag;
private String aLocation;

```

aItemsBag correspond à l'inventaire des items, et aLocation l'endroit où il est stocké.

On initialise les attributs de la manière suivante :

```

public ItemList(final String pLocation)
{
    this.aItemsBag = new HashMap<String, Item>();
    this.aLocation = pLocation;
}

```


On crée les différents getters :

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

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

public String getBagString ()
{
    if (this.aItems.isEmpty()){
        return "There is no items in " + this.aLocation;
    }// if ()

    StringBuilder vChaine = new StringBuilder ("Items in " + this.aLocation + " :");
    Set <String> vItemsNames = this.aItems.keySet();
    for (String vName : vItemsNames){
        vChaine.append(" " + vName);
    }// for ()
    return vChaine.toString();
} // getItemString ()
```

Et on déplace les méthodes addItem et removeItem dans ItemList :

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

/**
 * Supprime un item de la HashMap
 *
 * @param pName Clef de la HashMap (nom de l'item)
 */
public void removeItem (final String pName, final Item pItem){
    this.aItems.remove(pName);
} // removeItem ()
```

On peut maintenant remplacer le type de l'attribut aItems en ItemList :

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

```

/**
 * @return l'item qu'on veut récupérer
 * @param pItem correspond à l'item demandé par l'utilisateur
 *
 */
public ItemList getItems()
{
    return this.aItems;
}

```

Il suffit maintenant de modifier les méthodes executeTake et executeDrop puisqu'on n'a plus accès aux items directement par les rooms :

```

/**
 *permet au joueur de prendre un item
 *@param pItem l'item qu'on veut prendre
 *
 */
public void executeTake(final Item pItem)
{
    this.aInventItems.getItems().put(pItem.getName(),pItem);
    this.aCurrentRoom.aRoomItems.removeItem(pItem.getName(),pItem);
}

```

```

/**
 *permet au joueur de poser un item
 *@param pItem l'item qu'on veut poser
 *
 */
public void executeDrop(final Item pItem)
{
    this.aCurrentRoom.aRoomItems.addItem(pItem.getName(), pItem);
    this.aInventItems.getItems().remove(pItem.getName(),pItem);
}

```

On termine par modifier les placements des items dans les pièces de la manière suivante :

```

vVaisseau.getItems().addItem("lamp", vLampe);
vDehors7.getItems().addItem("blunt sword", vSabreNul);
vMaisonNain.getItems().addItem("incantation book", vLivre);
vMonstreMort.getItems().addItem("boosted shoes", vChaussures);
vSousSol.getItems().addItem("titanic sword", vSabreFort);
vSousSol.getItems().addItem("propulsion motor", vMoteur2);
vMonstreMort2.getItems().addItem("propulsion motor", vMoteur1);

```

EXO 32

On ajoute des restrictions sur le nombre d'items transportables : on ajoute alors les attributs suivants dans la classe Player :

```
private double aMaxWeight;  
private double aInventoryWeight;
```

On les initialise :

```
public Player(final String pPseudo, final Room pCurrentRoom, final double pMaxWeight)  
{  
    // initialisation des variables d'instance  
    this.aPseudo = pPseudo;  
    this.aCurrentRoom = pCurrentRoom;  
    this.aPreviousRoom = new Stack<Room>();  
    this.aInventItems = new ItemList("the inventory");  
    this.aMaxWeight = pMaxWeight;  
    this.aInventoryWeight = 0;  
}
```

On y ajoute également leurs getters :

```
/**  
 * @return le poids max des items qu'on peut porter dans l'inventaire  
 */  
public double getMaxWeight()  
{  
    return this.aMaxWeight;  
}
```

```
/**  
 * @return le poids de l'inventaire  
 */  
public double getInventoryWeight()  
{  
    return this.aInventoryWeight;  
}
```

On modifie les méthodes executeTake et executeDrop pour ajouter/supprimer du poids de l'inventaire :

```
/**
 *permet au joueur de prendre un item
 *@param pItem l'item qu'on veut prendre
 */
public void executeTake(final Item pItem)
{
    this.aInventItems.getItems().put(pItem.getName(),pItem);
    this.aCurrentRoom.aItems.removeItem(pItem.getName(),pItem);
    this.aInventoryWeight+=pItem.getWeight();
}
```

```
/**
 *permet au joueur de poser un item
 *@param pItem l'item qu'on veut poser
 */
public void executeDrop(final Item pItem)
{
    this.aCurrentRoom.aItems.addItem(pItem.getName(), pItem);
    this.aInventItems.getItems().remove(pItem.getName(),pItem);
    this.aInventoryWeight-=pItem.getWeight();
}
```

On termine par modifier la méthode take dans GameEngine puisqu'on a maintenant besoin de la condition du poids :

```
/**
 * procédure qui permet de récupérer l'item
 * @param pCommand la commande entrée par l'utilisateur
 */
private void take(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("What do you want to take ?");
    }
    else if (pCommand.hasSecondWord())
    {
        String vTake=pCommand.getSecondWord();
        Item vItem=this.aPlayer.getCurrentRoom().getItem(vTake);
        if (vItem==null)
        {
            this.aGui.println("This item is not in this room or does not exist");
            return;
        }
        else if (vItem.getWeight()+this.aPlayer.getInventoryWeight()>this.aPlayer.getMaxWeight()){
            this.aGui.println("this item is too heavy for the bag");
        }
        else
        {
            this.aPlayer.executeTake(vItem);
            this.aGui.println("You have taken "+vTake);
        }
        this.aGui.println(this.aPlayer.getBagItems().getBagString());
    }
}
```

EXO 33

On souhaite créer une commande pour ouvrir l'inventaire :

On commence par ajouter « inventory » dans le tableau de commandes de CommandWords, on ajoute la condition dans interpretCommand et on crée la méthode inventory comme ci-dessous :

```
/**
 * affiche l'inventaire du joueur
 * @param pCommand la commande entrée par l'utilisateur
 */
private void inventory(final Command pCommand)
{
    if (pCommand.hasSecondWord()){
        this.aGui.println("if you want to see your inventory write 'inventory'");
        return;
    }
    this.aGui.println(this.aPlayer.getBagItems().getBagString());
    this.aGui.println("Total weight: " + this.aPlayer.getInventoryWeight());
    this.aGui.println("Max weight: " + this.aPlayer.getMaxWeight());
}
```

Ici, on vérifie qu'il n'y ait pas de second mot sinon on retourne un message d'erreur, puis une fois ceci fait on affiche l'inventaire de cette manière.

EXO 34

On crée un nouvel item MagicCookie qui permet d'augmenter la taille de l'inventaire.

On le crée et le place dans une pièce du jeu dans la méthode createRooms.

```
Item vMagicCookie = new Item("MagicCookie", "allows you to expand your inventory", 2);
vMonstreMort.getItems().addItem("MagicCookie", vMagicCookie);
```

On ajoute des setters dans Player qui modifient le poids max et le poids de l'inventaire :

```
/**
 * change le poids max des items qu'on peut porter dans l'inventaire
 * @param pMaxWeight le nouveau poids max de l'inventaire
 */
public void setMaxWeight(final double pMaxWeight)
{
    this.aMaxWeight = pMaxWeight;
}
```

```
/**
 * enlève du poids de l'inventaire
 * @param pWeight le poids qu'on enlève de l'inventaire
 */
public void removeInventWeight(final double pWeight)
{
    this.aInventoryWeight-=pWeight;
}
```

On peut dès à présent modifier executeTake et executeDrop pour éviter la duplication de code :

```
/**
 *permet au joueur de prendre un item
 *@param pItem l'item qu'on veut prendre
 *
 */
public void executeTake(final Item pItem)
{
    this.aInventItems.getItems().put(pItem.getName(),pItem);
    this.aCurrentRoom.aItems.removeItem(pItem.getName(),pItem);
    this.removeInventWeight(-pItem.getWeight());
}

/**
 *permet au joueur de poser un item
 *@param pItem l'item qu'on veut poser
 *
 */
public void executeDrop(final Item pItem)
{
    this.aCurrentRoom.aItems.addItem(pItem.getName(), pItem);
    this.aInventItems.getItems().remove(pItem.getName(),pItem);
    this.removeInventWeight(pItem.getWeight());
}
```

On termine par modifier la méthode eat dans GameEngine :

```
/**
 * procédure qui permet au joueur de manger
 */
public void eat(final Command pCommand)
{
    if (!pCommand.hasSecondWord()){
        this.aGui.println("What do you want to eat ?");
        return;
    }
    String vFood = pCommand.getSecondWord();

    if (!vFood.equals("MagicCookie")){
        this.aGui.println("You can't eat that !");
        return;
    }

    if (this.aPlayer.getBagItems().getItem("MagicCookie")==null){
        this.aGui.println("You don't have cookie in your bag");
        return;
    }

    this.aPlayer.removeInventWeight(this.aPlayer.getBagItems().getItem("MagicCookie").getWeight());
    this.aPlayer.setMaxWeight(10);
    this.aPlayer.getBagItems().removeItem("MagicCookie",this.aPlayer.getBagItems().getItem("MagicCookie"));
    this.aGui.println("You have eaten now and you are not hungry anymore");
    this.aGui.println("Your inventory is bigger !");
}
```

EXO 42

On implémente dans cet exercice une limite de temps, ou plutôt un nombre de déplacements restreint avant le Game Over.

On ajoute un attribut `aMovesLeft` dans la classe `Player`, et l'initialise dans le constructeur :

```
private int aMovesLeft;
```

```
public Player(final String pPseudo, final Room pCurrentRoom, final double pMaxWeight, final int pMovesLeft)
{
    // initialisation des variables d'instance
    this.aPseudo = pPseudo;
    this.aCurrentRoom = pCurrentRoom;
    this.aPreviousRoom = new Stack<Room>();
    this.aInventoryItems = new ItemList("the inventory");
    this.aMaxWeight = pMaxWeight;
    this.aInventoryWeight = 0;
    this.aMovesLeft = pMovesLeft;
}
```

On y crée ses accesseur et modificateur :

```
/**
 * @return le nombre de mouvement restant
 */
public int getMovesLeft()
{
    return this.aMovesLeft;
}
```

```
/**
 * procédure qui ajoute un nombre de déplacements
 * @param pAdd le nombre de déplacements ajoutés
 */
public void setMovesLeft( final int pMovesLeft)
{
    this.aMovesLeft = pMovesLeft;
}
```

On ajoute l'instruction suivante à la fin de `executeGoRoom` et `executeGoBack` :

```
this.aMovesLeft -= 1;
```

Dans la classe `GameEngine`, on crée la procédure suivante pour l'appeler à la fin de `interpretCommand` :

```
private void timerEnd()
{
    if(this.aPlayer.getMovesLeft() == 0) {
        this.aGui.println("You have no more oxygen and you died");
        this.endGame();
    }
}
```

Dans notre jeu, on aimerait que le joueur retourne au vaisseau à chaque fois qu'il est à court d'oxygène pour se recharger. Pour se faire, on ajoute un attribut aName à la classe Room et son accesseur.

```
public class Room
{
    private String aName;
    private String aDescription;
    private HashMap<String,Room> aExits;
    private String aImageName;
    public ItemList aItems;

    /**
     * constructeur de la classe
     * @param pDescription correspond à la description de la pièce
     * @param pImage correspond au nom de l'image de la pièce
     */
    public Room(final String pName, final String pDescription, final String pImage)
    {
        this.aName = pName;
        this.aDescription= pDescription;
        this.aExits = new HashMap<String,Room>();
        this.aImageName = pImage;
        this.aItems = new ItemList("The room");
    } //Room
}
```

```
/**
 * @return le nom de la room
 */
public String getName()
{
    return this.aName;
}
```

On ajoute une méthode oxyMoves dans GameEngine pour vérifier si le joueur est bien retourné au vaisseau, si oui le compte à rebours est relancé :

```
/**
 * vérifie si le joueur est retourné au vaisseau, si oui le temps est réinitialisé à 15 moves
 */
private void oxyMoves()
{
    if(this.aPlayer.getCurrentRoom().getName().equals("Spaceship"))
        this.aPlayer.setMovesLeft(10);
}
```

On appelle alors cette méthode à la fin d'interpretCommand.

On termine par ajouter à la fin de printHelp un message pour savoir combien de tours il nous reste avant de mourir :

```
/**
 * Print out some help information.
 * Here we print some stupid, cryptic message and a list of the
 * command words.
 */
private void printHelp()
{
    this.aGui.println( "You are lost. You are alone. You wander" );
    this.printLocationInfo();
    this.aGui.println( "Your command words are: " + this.aParser.getCommandString() );
    this.aGui.println("You have " + this.aPlayer.getMovesLeft() + " moves left before "+
        "dying of asphyxiation");
}
```

EXO 43

On souhaite enlever la possibilité de faire back lorsqu'on passe par une trap door (que j'ai mis à la pièce vMonstre2).

On crée la fonction booléenne isExit dans la classe Room qui vérifie si la pièce vers laquelle on souhaite aller dispose de la pièce actuelle en sortie.

```
/**
 * @return true si la pièce vers laquelle on souhaite aller n'est pas une trap door,
 * false sinon
 * @param pRoom la room vers laquelle on souhaite se diriger
 */
public boolean isExit(final Room pRoom)
{
    return this.aExits.containsValue(pRoom);
}
```

Ici, la méthode containsValue permet de savoir si une HashMap contient bien un élément (pRoom) grâce à une clé(aExits).

De plus, on ajoute à Player la fonction possibleGoBack qui permet de savoir si le joueur est passé par une trapdoor :

```
/**
 * @return True si le joueur est passé par une trap door
 */
public boolean possibleGoBack()
{
    return this.aCurrentRoom.isExit(this.aPreviousRoom.peek());
}
```

Enfin, on ajoute la condition dans la méthode back de GameEngine :

```

/**
 * procédure qui permet de revenir dans la pièce précédente
 * @param pCommand correspond à la commande entrée par l'utilisateur
 */
private void back(final Command pCommand)
{
    if (pCommand.hasSecondWord()){
        this.aGui.println("You can't back in a particular room !");
        return;
    }

    if(this.aPlayer.getPreviousRoom().empty()){
        this.aGui.println("There is no previous room");
        return;
    }
    if (!this.aPlayer.possibleGoBack()){
        this.aGui.println("this was a trap door !");
        return;
    }
    this.aPlayer.executeGoBack();
    this.printLocationInfo();
}

```

EXO 44

Dans cet exercice on souhaite créer un téléporteur qui chargera une pièce et permettra de se téléporter dans cette pièce chargée à n'importe quel moment. Pour cela on va créer une nouvelle classe « Beamer » car cet item a un fonctionnement particulier. Cette classe possède un attribut correspondant à la pièce chargée et étant donné qu'un beamer est un item on fait hériter cette classe de la classe « Item ». On initialise également les attributs dans le constructeur naturel.

```

public class Beamer extends Item
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private Room aChargedRoom;

    /**
     * Constructeur d'objets de classe Beamer
     */
    public Beamer(final String pName, final String pDescription, final double pWeight)
    {
        // initialisation des variables d'instance
        super(pName, pDescription, pWeight);
    }
}

```

On ajoute un accesseur et un modificateur pour l'attribut qu'est la pièce chargée et une fonction booléenne qui nous dit si le beamer est chargé.

```
public Room getChargedRoom()  
{  
    // Insérez votre code ici  
    return this.aChargedRoom;  
}
```

```
public void setChargedRoom(final Room pRoom)  
{  
    this.aChargedRoom = pRoom;  
}
```

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

On ajoute les deux commandes « charge » et « fire » du beamer dans le tableau de commandes valides et on ajoute l'interprétation de ces commandes dans « interpretCommand » de « GameEngine ».

Désormais, on crée d'abord la méthode « charge » dans « GameEngine ». Dans cette méthode, on vérifie la présence d'un second mot, si le second mot est bien le beamer que l'on souhaite charger, ensuite on vérifie si le beamer est bien dans notre inventaire et finalement on vérifie si le beamer n'est pas déjà chargé.

```

private void charge(final Command pCommand)
{
    if (!pCommand.hasSecondWord()){
        this.aGui.println("What do you want to charge ?");
        return;
    }
    String vSecondWord= pCommand.getSecondWord();
    if (!vSecondWord.equals("Beamer")){
        this.aGui.println("I don't know how to charge this..");
        return;
    }
    Item vItem = this.aPlayer.getBagItems().getItem(vSecondWord);
    Beamer vBeamer = (Beamer)vItem;
    if (vItem==null){
        this.aGui.println("You don't have the beamer in your inventory");
        return;
    }
    if (vBeamer.isCharged()){
        this.aGui.println("Your beamer is already charged");
        return;
    }
    this.aPlayer.chargeBeamer(vBeamer);
    this.aGui.println("Your beamer has been charged");
}

```

Une fois tous ces tests passés on appelle la procédure « chargeBeamer » de la classe « Player » qui gère les déplacements du joueur.

```

public void chargeBeamer(final Beamer pBeamer)
{
    pBeamer.setChargedRoom(this.aCurrentRoom);
}

```

La méthode « fire » est très semblable à celle de « charge » car elle doit passer les mêmes tests.

```

private void fire(final Command pCommand)
{
    if (!pCommand.hasSecondWord()){
        this.aGui.println("What do you want to fire ?");
        return;
    }
    String vSecondWord= pCommand.getSecondWord();
    if (!vSecondWord.equals("Beamer")){
        this.aGui.println("I don't know how to fire this..");
        return;
    }
    Item vItem = this.aPlayer.getBagItems().getItem(vSecondWord);
    Beamer vBeamer = (Beamer) vItem;
    if (vItem==null){
        this.aGui.println("You don't have the beamer in your inventory")
        return;
    }
    if (!vBeamer.isCharged()){
        this.aGui.println("Your beamer isn't charged yet !");
        return;
    }
    this.aPlayer.fireBeamer(vBeamer);
    this.aGui.println("Your beamer has been fired");
    this.printLocationInfo();
}

```

Dans cette méthode, on appelle « fireBeamer » dans « Player » :

```

public void fireBeamer(final Beamer pBeamer)
{
    this.aPreviousRoom.push(this.aCurrentRoom);
    this.aCurrentRoom = pBeamer.getChargedRoom();
    this.aInventItems.removeItem(pBeamer.getName(), pBeamer);
    this.aInventoryWeight-=pBeamer.getWeight();
}

```

Finalement on crée le beamer dans « createRooms » de la classe « GameEngine » et on le place dans un pièce.

```

Beamer vBeamer = new Beamer("Beamer", "to teleport you somewhere",1);
Item vCheuesure = new Item("beasted chess" "allows you to climb mount
vVaisseau.getItems().addItem("Beamer", vBeamer);
vObjets.getItems().addItem("Beamer", vBeamer);

```

Remarque : on remplace l'item lampe par le Beamer à la première pièce du jeu.

Modifications du jeu :

En lançant le jeu, on remarque un problème : à chaque fois qu'un bouton est actionné sauf go Down, on remarque à la fin que le message « I don't know what you mean.. » est affiché, malgré le fait qu'on se déplace tout de même.

Dans la méthode actionPerformed de la classe UserInterface, on avait mis des conditions indépendantes les unes par rapport aux autres (voir exo 18 page 16). On modifie de la manière suivante pour régler ce problème :

```
@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 (this.aButtonNorth.equals(pE.getSource()))
        this.aEngine.interpretCommand("go North");
    else if (this.aButtonSouth.equals(pE.getSource()))
        this.aEngine.interpretCommand("go South");
    else if (this.aButtonEast.equals(pE.getSource()))
        this.aEngine.interpretCommand("go East");
    else if (this.aButtonWest.equals(pE.getSource()))
        this.aEngine.interpretCommand("go West");
    else if (this.aButtonUp.equals(pE.getSource()))
        this.aEngine.interpretCommand("go Up");
    else if (this.aButtonDown.equals(pE.getSource()))
        this.aEngine.interpretCommand("go Down");
    else
        this.processCommand(); // never suppress this line
} // actionPerformed()
```

Je modifie également le message de bienvenue :

```
/**
 * Print out the opening message for the player.
 */
private void printWelcome()
{
    this.aGui.print( "\n" );
    this.aGui.println( "Welcome in Samourai on Moon !" );
    this.aGui.println( "You are Gintoki Hanma, a samourai on Titan missionning for"+
        " science purposes" );
    this.aGui.println( "The spaceship is broken, you need to change the 2 broken"+
        " motors to get back on Earth" );
    this.aGui.println( "Type 'help' if you need help." );
    this.aGui.print( "\n" );
    this.printLocationInfo();
}
```

Je modifie les noms des items pour pouvoir les utiliser plus simplement :

```
Beamer vBeamer = new Beamer("Beamer", "to telepor  
Item vChaussures = new Item("BoostedShoes", "allc  
Item vSabreNul = new Item("BluntSword", "only all  
Item vSabreFort = new Item("TitanicSword", "allow  
Item vLivre = new Item("IncantationBook", "gives  
Item vMoteur1 = new Item("BrokenPropulsionMotor",  
"an item that would make it work again..", 2);  
Item vMoteur2 = new Item("PropulsionMotor", "Brav  
Item vMagicCookie = new Item("MagicCookie", "allc
```

EXO 45

Je modifie les tests pour utiliser toutes les commandes possibles :

```
1 take Beamer
2 charge Beamer
3 go East
4 go North
5 back
6 back
7 go East
8 go East
9 take BluntSword
10 back
11 go South
12 drop BluntSword
13 go North
14 go West
15 go East
16 go North
17 look
18 go North
19 go North
20 go West
21 take MagicCookie
22 eat MagicCookie
23 go East
24 go East
25 take PropulsionMotor
26 take TitanicSword
27 fire Beamer
28 drop PropulsionMotor
29 drop TitanicSword
30 help
31 look
32 back
33 go East
34 go North
35 go East
36 go Up
37 go Down
38 quit
```


Modifications du code :

On va maintenant faire le scénario gagnant du jeu : il faut drop les 2 moteurs à propulsion et le livre enchanté au vaisseau pour pouvoir repartir sur Terre. On crée alors la procédure winSituation dans GameEngine :

```
private void winSituation()
{
    if (this.aPlayer.getCurrentRoom().getName().equals("Spaceship")){
        Room vRoom = this.aPlayer.getCurrentRoom();
        if (vRoom.getItems().verifItem("IncantationBook")){
            if (vRoom.getItems().verifItem("PropulsionMotor")){
                if (vRoom.getItems().verifItem("BrokenPropulsionMotor")){
                    this.printWin();
                    this.endGame();
                }
            }
        }
    }
    else{
        this.printMiss();
    }
}
```

On vérifie que les 3 items sont bien posés dans la room Spaceship, et si c'est le cas on affiche le message de la victoire, puis on appelle endGame() pour terminer le jeu. Sinon, on affiche le message de ce qu'il faut pour gagner dans le cas où on est dans le vaisseau sans avoir drop ces items.

```
private void printWin()
{
    this.aGui.println("\n Congratulations ! You gathered all the items needed"+
        " to get back to Earth \n");
    this.aGui.println("Your adventure ends here. Thank you for playing !");
    this.endGame();
}
```

```
private void printMiss()
{
    this.aGui.println("\n Hey ! You need to DROP two propulsion motors"+
        " to get back to Earth. \n");
    this.aGui.println("Maybe there is an item to DROP to repair a broken motor..");
}
```

On appelle winSituation à la fin de interpretCommand() :

```

        if ( vCommand.hasSecondWord() )
            this.aGui.println( "Quit what?" );
        else
            this.endGame();
    }

    this.oxyMoves();
    this.timerEnd();
    this.winSituation();

```

De plus, on ajoute les situations perdantes : le joueur meurt quand il rentre dans la pièce d'un monstre sans l'arme nécessaire, ou quand il tente de gravir la montagne sans avoir les chaussures boostées dans son inventaire.

On commence par ajouter un attribut booléen aStatus dans la classe Player qui indique si le joueur est vivant ou non :

```

private boolean aStatus;
/++

```

Bien sûr, il est initialisé à true :

```

public Player(final String pPseudo, final Room pCurrentRoom, final double pMaxWe
{
    // initialisation des variables d'instance
    this.aPseudo = pPseudo;
    this.aCurrentRoom = pCurrentRoom;
    this.aPreviousRoom = new Stack<Room>();
    this.aInventItems = new ItemList("the inventory");
    this.aMaxWeight = pMaxWeight;
    this.aInventoryWeight= 0;
    this.aMovesLeft = pMovesLeft;
    this.aStatus = true;
}

```

On lui crée son accesseur :

```

public boolean getStatus()
{
    return this.aStatus;
}

```

On n'a besoin que de créer setDead, puisqu'on ne peut pas ressusciter dans le jeu !

```
public void setDead()
{
    this.aStatus = false;
}
```

A la fin de interpretCommand, on vérifie si le player est toujours vivant : si non, on affiche « you died » puis on termine le jeu :

```
if (!this.aPlayer.getStatus()){
    this.aGui.println("You died");
    this.endGame();
}
```

On ajoute 2 attributs à la classe Room : un de type int pour compter le nombre de passages dans une room et un autre pour savoir quel Item est requis pour arriver à une pièce :

```
public class Room
{
    private String aName;
    private String aDescription;
    private HashMap<String, Room> aExits;
    private String aImageName;
    public ItemList aItems;
    private int aPassage;
    private String aItemNeeded;
```

On modifie alors le constructeur naturel et on ajoute un constructeur par défaut comme ceci :

```

public Room(final String pName, final String pDescription, final String pImage,
final String pItemNeeded)
{
    this.aName = pName;
    this.aDescription= pDescription;
    this.aExits = new HashMap<String,Room>();
    this.aImageName = pImage;
    this.aItems = new ItemList("The room");
    this.aPassage = 0;
    this.aItemNeeded = pItemNeeded;
} //Room

```

```

public Room(final String pName, final String pDescription, final String pImage)
{
    this(pName, pDescription, pImage, "None");
} //Room

```

On leur crée les accesseurs :

```

public int getPassage()
{
    return this.aPassage;
}

```

```

public String getItemNeeded()
{
    return this.aItemNeeded;
}

```

On modifie la méthode createRooms de GameEngine pour ajouter les items nécessaires dans les room Cerberus et Titan :

```

private void createRooms()
{
    Room vVaisseau = new Room("Spaceship","in the spaceship", "vaisseau.jpg");
    Room vDehors1 = new Room("FSpaceship","in front of the spaceship, you stand on the frozen land.",
    "dehors1.jpg");
    Room vDehors2 = new Room("FDwarfShelter","in front of the dwarf's shelter", "maisonNain.png");
    Room vMaisonNain = new Room("DwarfShelter", "in the dwarf's shelter","dwarfhous.jpg");
    Room vBasMontagne = new Room("BotMountain","at the foot of the mountain","botmountain.jpg");
    Room vHautMontagne = new Room("TopMountain","in the valley's mountain; you may face a big monster.. a blunt sword won't be enough","topmountain.jpg", "BoostedShoes");
    Room vDehors4 = new Room("Dehors4","in the winding mud path","mud.jpg");
    Room vPiece1 = new Room("HotelMonster","in the Hotel Monster","hotel.jpg");
    Room vChambre = new Room("Room29","in room 29","room.jpeg");
    Room vSousSol = new Room("SousSol","in the basement","basement.png");
    Room vDehors7 = new Room("Grave","in front of a samourai's grave","grave.jpg");
    Room vDehors5 = new Room("Dehors5","in a sinister path, you need a blunt sword to be ready", "sinister.jpg");
    Room vMonstre1 = new Room("Cerberus", "in the Cerberus' lair","cerberus.jpg", "BluntSword");
    Room vMonstreMort = new Room("CerbMort","in the treasure room", "treasure.jpg", "TitanicSword");
    Room vMonstre2 = new Room("Titan","in the original Titan's lair", "titan.jpg", "TitanicSword");
    Room vMonstreMort2 = new Room("TitanMort","in the treasure room", "treasure2.jpg");
}

```

On peut maintenant créer les situations perdantes : on crée 2 méthodes canGoMountain et canGoMonster dans la classe Player qui donne les conditions pour rester en vie dans une pièce :

```

public boolean canGoMonster()
{
    if (this.aCurrentRoom.getPassage() == 0){
        if (this.aCurrentRoom.getItemNeeded().equals("None")){
            return true;
        }
        else if (this.aCurrentRoom.getName().equals("TopMountain")){
            return true;
        }
        else if (this.aInventItems.getItem(this.aCurrentRoom.getItemNeeded()) == null){
            this.setDead();
            return false;
        }
    }
    return true;
}

public boolean canGoMountain()
{
    if (this.aCurrentRoom.getItemNeeded().equals("None")){
        return true;
    }
    else if (this.aCurrentRoom.getName().equals("TopMountain")){
        if (this.aInventItems.getItem(this.aCurrentRoom.getItemNeeded()) == null){
            this.setDead();
            return false;
        }
    }
    return true;
}

```

On comprend maintenant l'utilité de l'attribut aPassage dans la Room : si on n'est jamais arrivé dans une room avec un monstre (Cerberus ou Titan), il nous faut alors les items pour les vaincre. Cependant, si on les a déjà vaincus, il n'y a plus d'intérêt à avoir une épée dans ces room.

La méthode canGoMountain diffère de canGoMonster du fait qu'on aura toujours besoin des chaussures boostées pour gravir la montagne.

IV déclaration anti plagiat

Je soussigné PHAM Jimmy

- déclare que ce rapport est un document original fruit d'un travail personnel ;
- suis au fait que la loi sanctionne sévèrement la pratique qui consiste à prétendre être l'auteur d'un travail écrit par une autre personne ;
- atteste que les citations d'auteurs apparaissent entre guillemets dans le corps du mémoire ;
- atteste que les sources ayant servi à élaborer mon travail de réflexion et de rédaction sont référencées de manière exhaustive et claire dans la bibliographie figurant à la fin du mémoire ;
- déclare avoir obtenu les autorisations nécessaires pour la reproduction d'images, d'extraits, figures ou tableaux empruntés à d'autres œuvres.