RAPPORT

PROJET JAVA ZUUL :

L'ÎLE DE JAVA

Table des matières

PARTIE I : L'Île de Java

I.A) Auteur :

TADRES Nicolas

I.B) Thème :

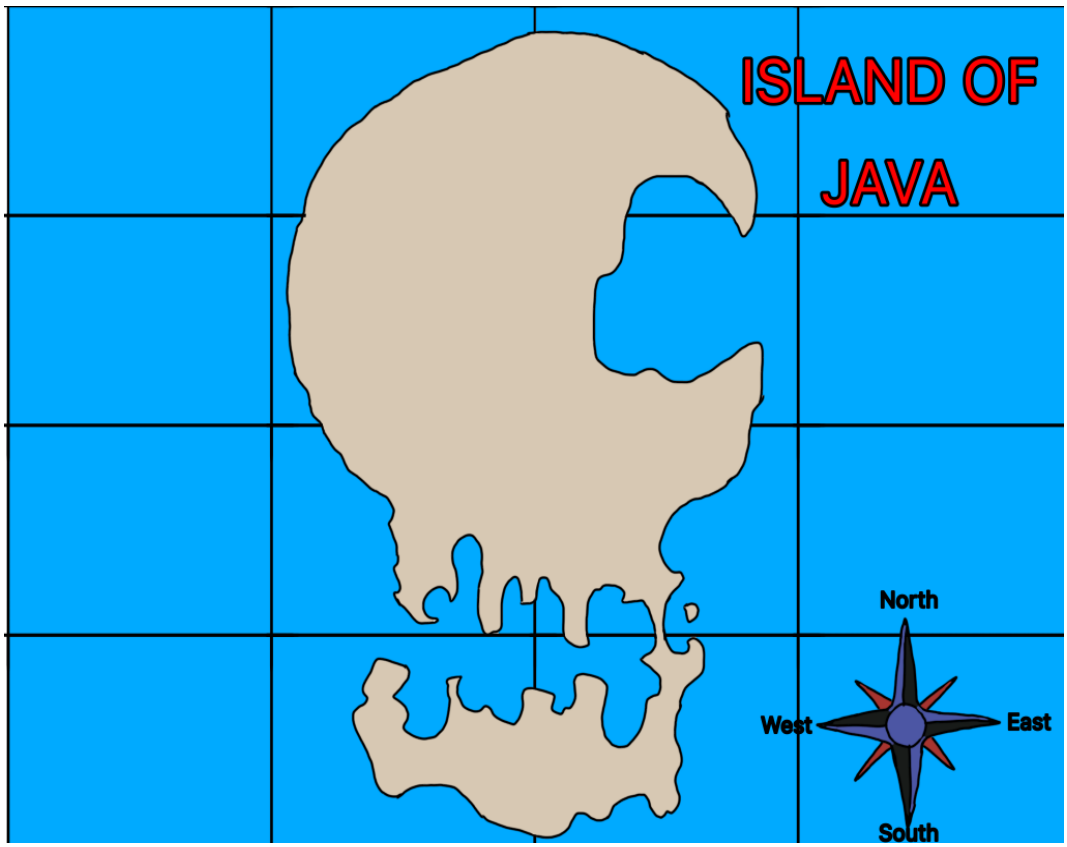
Sur une île, un jeune explorateur doit découvrir l'histoire d'une ancienne civilisation ayant vécu sur cette île mais qui a subitement disparue.

I.C) Résumé du scénario :

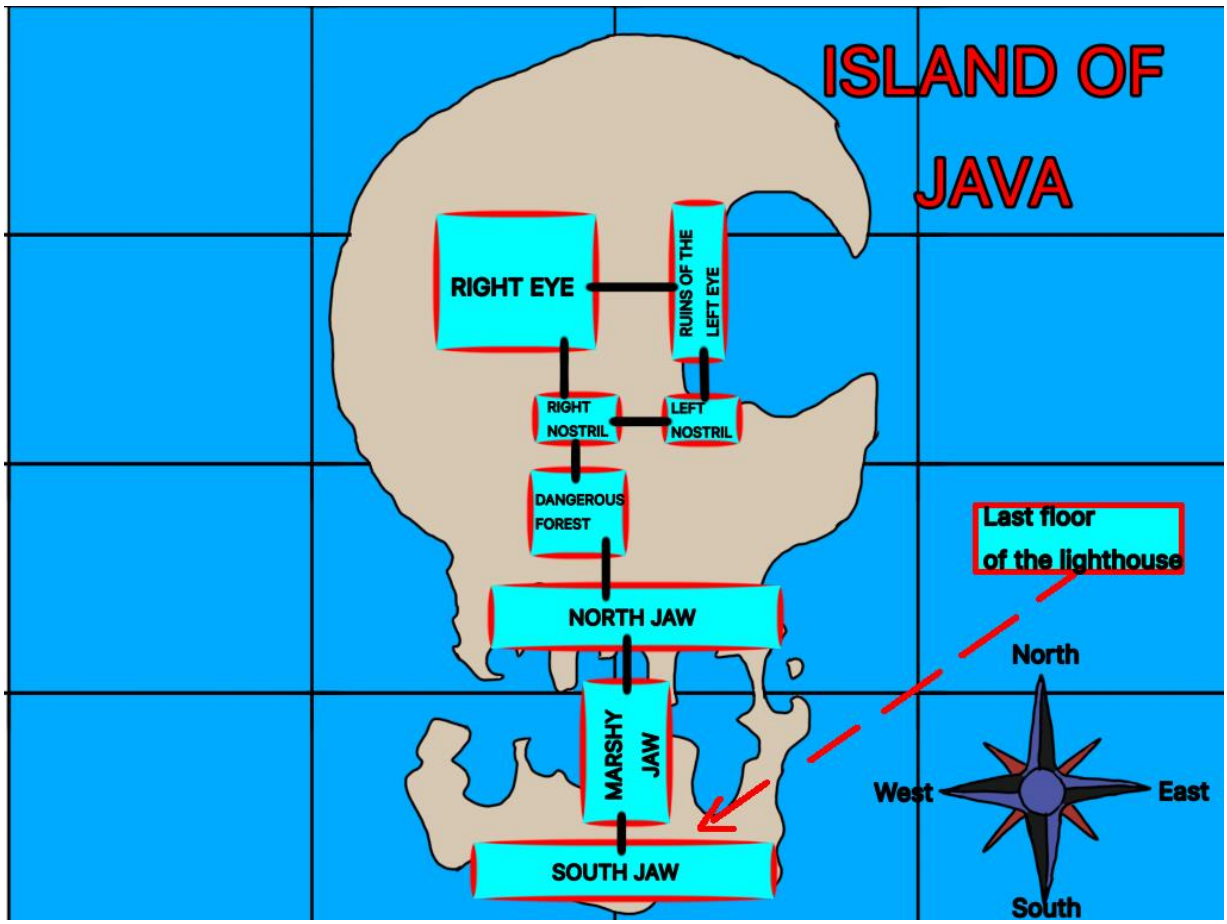
Il y'a plus d'un millénaires une civilisation antique prit demeure sur l'île de Java. Cependant, toute la civilisation disparue à l'exception d'une personne qui gravit les mémoires de sa civilisation sur une stèle. En tant que jeune explorateur, vous devriez trouver cette stèle grâce à des énigmes et en faisant attention aux dangers.

I.D) Plan :

Carte de l'île de Java sans les lieux :



Carte de l'île avec les lieux :



I.E) Scénario détaillé :

Il y'a plus d'un millénaires une civilisation antique prit demeure sur l'île de Java à la curieuse forme d'un crâne.
Cependant la civilisation entière, à l'exception d'une personne, disparu du jour au lendemain sans laisser de trace. Cette dernière personne, avant de mourir, gravit les mémoires de sa civilisation sur une stèle afin que le monde découvre leur histoire.
Malheureusement nulle ne sait où se trouve cette stèle. Toutefois certains indices et certaines énigmes laissés par le survivant semble indiquer le lieux où se trouve la stèle.
De nombreux archéologues et historiens ont tentés de découvrir l'histoire de cette civilisation en y laissant parfois leurs peaux...
Cependant vous, qui êtes un courageux archéologue et historien en quête d'aventure et en soif de connaissance, souhaitez trouver cette stèle afin de découvrir l'histoire de cette civilisation antique ainsi que la cause de leur disparition soudaine.
Votre but est donc de résoudre les énigmes afin de trouver la fameuse stèle.
Mais attention la forme de l'île n'est sûrement pas anodine et elle semble indiquer certains danger et certains pièges qui peuvent vous être mortel.

I.F) Détail des lieux, items, personnages

Il y'a qu'un seul personnage qui est le joueur (l'explorateur de l'île).

South Jaw:

- Salle de départ du jeux
- Items : lettre sur laquelle il est écrit un indice

Last Floor of the lighthouse :

- Items: présence d'une clé utile pour la suite, d'un magicCookie et d'un beamer

Marshy Jaw :

- Une sorte de marécage où il y'a un gaz empoisonné dans l'air le joueur devra quitter la salle en moins de 20 secondes.

North Jaw:

- Items : 1ère pierre qui sera utile pour une énigme
- couteau qui sera utile pour une prochaine salle

Dangerous Forest :

- Salle dans laquelle il faudra vaincre un animal (ou un monstre) grâce au couteau

Right Nostril :

- Items : 2ème pierre (qui sera également utile pour une énigme) trouver à condition d'avoir la clé

Left Nostril :

- Items : 3ème pierre (aussi utile pour une énigme)

Ruins of the Left Eye :

- Items : 4ème pierre (aussi utile pour une énigme) + feuille où il est écrit un indice

Right Eye :

- Salle où se trouve la fameuse stèle accessible seulement si on effectue une action précise
- Présence d'une cloche

I.G) Situations gagnantes et perdantes

Situation gagnante :

- réussir à trouver la stèle en évitant les pièges et les dangers

Situations perdantes :

- Rester dans la salle Masrshy Jaw plus de 20 secondes
- Utiliser plus de 6 fois la commandes « go north »
- Oublier le couteau et donc ne pas réussir à vaincre le monstre

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

Mini-jeux : Sauter d'une pierre à une autre dans la salle Marshy Jaw.

Combat : Vaincre un monstre grâce au couteau dans Dangerous Forest.

1ère énigme écrite sur la 1ère lettre :

« le croisement de 4 pierres vous indiquera le chemin »

Ici il faudra trouver 4 pierres qui, une fois assemblées, vous indiqueront le lieu où est caché la stèle.

2ème énigme écrite sur la 2ème lettre :

« l'histoire surgit au son de la cloche »

Cette énigme indique l'action précise a effectué pour avoir accès à la stèle (qui se trouve à Right Eye). Il faudra sonner une cloche qui « déverrouillera » la stèle.

I.I) Commentaires

J'aurai aimé faire tout ce que j'avais imaginé dans mon scénario cependant je n'ai pas pu tout faire. Il y a donc des éléments que j'ai cité, ci-dessus dans mon scénario, qui ne sont pas présent.

PARTIE II : Réponses aux exercices

Exercice 7.5 :

Question 3 :

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

Question 4 :

Cette procédure est conçu pour éviter la duplication de code et nous facilite le travail si on aimerai changer quelque chose plus tard.

Exercice 7.6 :

Question 1 :

Il n'est pas indispensable de tester si les paramètres sont null car un attribut est null si il n'est pas initialisé par le constructeur.

Question 2 et 3 :

Au début de Room (pour le cas Unknown direction) :

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

Modification dans la classe Room :

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

Modification dans la procédure goRoom de la classe Game :

```
private void goRoom(final Command pCommand)
    {
        if (!pCommand.hasSecondWord()) {
            System.out.println("Go where ?");
            return;
        }
        String vDirection;
        vDirection = pCommand.getSecondWord();
        Room vNextRoom = aCurrentRoom.getExit(vDirection);
       if (vNextRoom == Room.UNKNOWN_DIRECTION) {
         System.out.println("Unknown direction ! ");
         return;
       }
        if (vNextRoom == null) {
            System.out.println("There is no door !");
          return;
        }
        else {
            this.aCurrentRoom = vNextRoom;
            printLocationInfo();
        }
     }
```

Exercice 7.7 :

Fonction getExitString:

```
public String getExitString()
{
     String vS = "Exits : ";
     if (this.aNorthExit != null) {
          vS = vS + "north ";
     }
```

```
        if (this.aEastExit != null) {
            vS = vS + "east ";
        }
        if (this.aSouthExit != null) {
            vS = vS + "south ";
        }
        if (this.aWestExit != null) {
            vS = vS + "west";
        }
        return vS;
    }
```

Modification de la procédure printLocationInfo :

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

C'est la classe Room qui doit produire les informations sur ses sorties car c'est cette classe qui s'occupe des salles (c'est son devoir).

Exercice 7.8 :

Modification dans la classe Room :

- Ajout d'un attribut qui remplace les 4 autres attributs on a donc ceci :

```
    private String aDescription;
    private HashMap<String, Room> aExits;
```

- Modification dans le constructeur Room :

```
public Room(final String pDescription)
    {
        this.aDescription = pDescription;
```

```
            this.aExits = new HashMap<String, Room>();
        }
```

- On remplace setExits par setExit (ceci a conduit à des modifications dans la classe Game) :

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

- Modification de getExit :

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

- La modification de getExitString est dans l'exercice 7.9.

Exercice 7.8.1:

J'ai ajouté une pièce se situant tout haut d'un phare que l'on peut apercevoir sur South Jaw.

Exercice 7.9:

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

Exercice 7.10:

On commence par créer une chaîne de caractères qui contiendra toute les sorties possibles.

Puis nous déclarons une collection de String qui contiendra les clés de notre HashMap « aExits » . Or nous savons que les clés de notre HaspMap sont les sorties de la salle.

Donc il suffit de parcourir cette collection grâce à une boucle *for each* pour récupérer chaque sortie et la rajouter à notre chaîne de caractère.

Enfin on renvoie la chaîne de caractère.

Exercice 7.11:

Ajout de la fonction getLongDescription :

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

Modification de printLocationInfo :

```
private void printLocationInfo()
    {

        System.out.println(this.aCurrentRoom.getLongDescr
        iption());
    }
```

Exercice 7.14:

Dans la classe CommandWords on ajoute ceci :

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

Dans la classe Game on ajoute la procédure suivante (avec la partie optionnelle) :

```
   private void look(final Command pCommand)
```

```
    {
        if (pCommand.hasSecondWord()) {
         System.out.println("I don't know how to look at
         something in particular yet.");
        }
        else {

              System.out.println(this.aCurrentRoom.getLong
              Description());
        }
    }
```

La méthode processCommand devient :

```
private boolean processCommand(final Command pCommand)
    {
       if (pCommand.isUnknown()) {
         System.out.println("I don't know what you
         mean...");
         return false;
       }

       else {
           if (pCommand.getCommandWord().equals("quit")) {
             return this.quit(pCommand);
             }
            else {
             if (pCommand.getCommandWord().equals("help"))
                 {
                     this.printHelp();
                     return false;
                  }
                  else {
              if (pCommand.getCommandWord().equals("go"))
```

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

Exercice 7.15:

J'ai ajouté une commande « take » qui permet de prendre un objet (cette commande ne sert qu'à afficher un message pour l'instant, elle permet juste de répondre à l'exercice 7.15, je sais qu'elle sera complété à l'exercice 7.30).

Pour cela j'ai modifier le tableau :

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

J'ai ajouté la procédure take dans la classe Game et j'ai modifié la fonction processCommand :

```
   private void take()
    {System.out.println("You took an object ! ");}
```

Exercice 7.16:

Procédure showAll dans la classe CommandWords :

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

Procédure dans la classe Parser :

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

Modification de la procédure printHelp dans la classe Game :

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

Exercice 7.18:

Modification de showAll en getCommandList dans la classe CommandWords :

```
    public String getCommandList()
    {
        String vS = " ";
```

```
        for (String pCommand : VALIDCOMMANDS){
            vS = vS + pCommand + " " ;
        }
        return vS;
    }
```

Modification de showCommands en getCommands dans la classe Parser :

```
public String getCommands()
    {
        return aValidCommands.getCommandList();
    }
```

Modification de printHelp dans la classe Game :

```
    private void printHelp()
    {
        System.out.println("You are lost. You are alone."
+ "\n" +
         "You wander around at the South Jaw." + "\n" +
"\n" + "Your command words are:"
          + aParser.getCommands());
    }
```

Exercice 7.18.5 :

J'ai ajouté un attribut `private HashMap<String, Room> aRooms;` dans la classe Game en ayant importé `java.util.HashMap ;` .

J'ai ensuite ajouté ceci dans la procédure createRooms :

```
aRooms = new HashMap<String, Room>();

aRooms.put("in the South Jaw", vSouthJaw);
aRooms.put("in the Marshy Jaw", vMarshyJaw);
aRooms.put("in the North Jaw", vNorthJaw);
aRooms.put("in the Dangerous Forest", vDangerousForest);
```

```
aRooms.put("in the Right Nostril", vRightNostril);
aRooms.put("in the Left Nostril", vLeftNostril);
aRooms.put("in the Right Eye", vRightEye);
aRooms.put("in the ruins of the Left Eye",
vRuinsOfTheLeftEye);
aRooms.put("in the last floor of the lighthouse",
vLastFloor);
```

Exercice 7.18.6:

J'ai modifié le projet selon le modèle zuul-with-images :

- Ajout de la classe GameEngine qui reprend les éléments de la classe Game + quelques modifications
- Ajout de la classe UserInterface
- Et des modifications dans les autres classes.

Exercice 7.18.7:

On a : `this.aEntryField.addActionListener( this );`

- Dans ce cas, addActionListener() permet d'enregister que c'est la classe UserInterface qui réagira à une action survenant sur aEntryField (c'est-à-dire la zone de texte)
- actionPerformed est invoqué à chaque fois qu'une action se produit. Par exemple dans notre cas, si un texte est tapé sur la zone de texte (aEntryField) et « validé » grâce à la touche ENTREE alors on appelle la procédure processCommand() qui traite la ligne de commande tapée.

Exercice 7.18.8:

Pour la création des boutons j'ai ajouté un nouveau panel étant donné que j'ajoute 10 boutons (car je ne peux ajouter que 2 boutons dans le panel vPanel).

Pour cela j'ai importé `import java.awt.GridLayout;` qui me permet de disposer les 10 boutons dans une sorte de grille rectangulaire grâce à la classe GridLayout.

J'ai également importé import `javax.swing.JButton;` qui me permet de créer les boutons.

Voici ci-dessous la déclaration des attributs du panel et des 10 boutons :

```
    private JPanel aPanelButton;
```

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

Dans createGUI() j'ai créé la grille rectangulaire 5x2 qui contiendra les boutons :

```
this.aPanelButton = new JPanel();
aPanelButton.setLayout(new GridLayout(5,2));
```

J'ai également initialisé les 10 boutons :

```
this.aButtonQuit = new JButton("quit");
this.aButtonHelp = new JButton("help");
this.aButtonNorth = new JButton("go north");
this.aButtonEast = new JButton("go east");
this.aButtonSouth = new JButton("go south");
this.aButtonWest = new JButton("go west");
this.aButtonUp = new JButton("go up");
this.aButtonDown = new JButton("go down");
this.aButtonLook = new JButton("look");
this.aButtonTake = new JButton("take");
```

Puis j'ai ajouté les boutons dans le nouveau panel aPanelButton :

```
aPanelButton.add(this.aButtonQuit);
aPanelButton.add(this.aButtonHelp);
```

```
aPanelButton.add(this.aButtonNorth);
aPanelButton.add(this.aButtonEast);
aPanelButton.add(this.aButtonSouth);
aPanelButton.add(this.aButtonWest);
 aPanelButton.add(this.aButtonUp);
aPanelButton.add(this.aButtonDown);
aPanelButton.add(this.aButtonLook);
aPanelButton.add(this.aButtonTake);
```

aPanelButton a lui-même été ajouté dans le panel vPanel :

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

Enfin j'ai ajouté les écouteur d'action pour chaque bouton :

```
this.aButtonQuit.addActionListener(this);
this.aButtonHelp.addActionListener(this);
this.aButtonNorth.addActionListener(this);
this.aButtonEast.addActionListener(this);
this.aButtonSouth.addActionListener(this);
this.aButtonWest.addActionListener(this);
this.aButtonUp.addActionListener(this);
this.aButtonDown.addActionListener(this);
this.aButtonLook.addActionListener(this);
this.aButtonTake.addActionListener(this);
```

Dans la procédure actionPerformed j'ai ajouté les actions à réaliser en fonction des boutons cliqué :

```
if (pE.getSource() == this.aButtonQuit) {
    this.aEngine.interpretCommand("quit");
    EnableButtons();
}
else if (pE.getSource() == this.aButtonHelp) {
```

```
        this.aEngine.interpretCommand("help");
    }
    else if (pE.getSource() == this.aButtonNorth) {
        this.aEngine.interpretCommand("go north");
    }
    else if (pE.getSource() == this.aButtonEast) {
        this.aEngine.interpretCommand("go east");
    }
    else if (pE.getSource() == this.aButtonSouth) {
        this.aEngine.interpretCommand("go south");
    }
    else if (pE.getSource() == this.aButtonWest) {
        this.aEngine.interpretCommand("go west");
    }
    else if (pE.getSource() == this.aButtonUp) {
        this.aEngine.interpretCommand("go up");
    }
    else if (pE.getSource() == this.aButtonDown) {
        this.aEngine.interpretCommand("go down");
    }
    else if (pE.getSource() == this.aButtonLook) {
        this.aEngine.interpretCommand("look");
    }
    else if (pE.getSource() == this.aButtonTake) {
        this.aEngine.interpretCommand("take");
    }
    else {
        this.processCommand();
        }
```

La procédure enableButtons() qui est présente dans la procédure actionPerformed ci-dessus est une procédure que j'ai créé pour éviter la duplication de code et pour désactiver les boutons une fois que l'on clique sur le bouton « quit » . Je l'ai également ajouté dans la procédure interpretCommand() de la classe GameEngine pour que les boutons soit désactivé lorsque l'on tape « quit ».

enableButtons() utilise la procédure setEnabled() :

```
public void enableButtons()
    {
            this.aButtonQuit.setEnabled(false);
            this.aButtonHelp.setEnabled(false);
            this.aButtonNorth.setEnabled(false);
            this.aButtonEast.setEnabled(false);
            this.aButtonWest.setEnabled(false);
            this.aButtonSouth.setEnabled(false);
            this.aButtonUp.setEnabled(false);
            this.aButtonDown.setEnabled(false);
            this.aButtonLook.setEnabled(false);
            this.aButtonTake.setEnabled(false);
    }
```

<u>Remarque:</u> le bouton take a été remplacé par le bouton back dans l'exercice 7.30 (voir remarque exercice 7.30)

<u>Exercice 7.19.2:</u>

J'ai déplacé les images dans un dossier Images à la racine du projet et j'ai modifié l'adresse des images dans le string de GameEngine sous la forme suivante :

```
vSouthJaw = new Room("in the South Jaw",
"Images/SouthJaw.gif");
```

<u>Exercice 7.20 :</u>

J'ai tout d'abord crée la classe Item avec 2 attributs (le nom qui est enfaite la description et le poids), le constructeur et les 2 accesseurs :

```
public class Item
{
  private String aNom;
  private double aPoids;
```

```
    public Item(final String pNom, final double pPoids)
    {
        this.aNom = pNom;
        this.aPoids = pPoids;
    }

    public String getNom()
    {
        return this.aNom;
    }

    public double getPoids()
    {
        return this.aPoids;
    }
}
```

Puis dans la classe Room j'ai créé un attribut aItem : `private Item aItem;`

Ainsi qu'une nouvelle procédure setItem pour initialisé l'item :

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

Enfin une fonction getItemString a été créé, dans la classe Item, permettant d'obtenir une string affichant l'item :

```
    public String getItemString()
    {
        return "Item :" + this.getNom();
    }
```

La fonction getLongDescription() a également été modifié en tenant compte de la fonction getItemString() :

```
  public String getLongDescription()
    {
        if (this.aItem == null) {
        return "You are " + this.aDescription + "\n" +
this.getExitString()
        + "\nThere is no item here.";
       }
       else {
        return "You are " + this.aDescription + "\n" +
this.getExitString()
        + "\n" + aItem.getItemString();
       }
    }
```

Pour ajouter l'item dans la Room il suffit donc de déclarer et initialiser l'item dans la procédure createRooms() de la classe GameEngine et de l'ajouter de la manière suivante :

```
Item vLettre1 = new Item("First letter", 0.5);
vSouthJaw.setItem(vLettre1);
```

Exercice 7.21 :

La string produisant toutes les informations sur un item doivent être produite par la classe Item (il s'agit de getItemString()).

La string produisant toutes les information sur une Room (c'est-à-dire les sorties et les items) est produite dans la classe Room (il s'agit de la fonction getLongDescription()).

Enfin c'est la classe GameEngine qui s'occupe d'afficher les informations.

Exercice 7.22 :

Pour ajouter plusieurs items j'ai modifié l'attribut aItem en un attribut de type HashMap dans la classe Room :

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

Puis je l'ai initialisé dans le constructeur de la classe Room :

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

Enfin la procédure setItem a été modifié en une procédure addItem qui permet d'ajouter les items dans les Rooms :

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

J'ai aussi modifié la fonction getItemString à l'image de getExitString qui permet d'obtenir la string contenant tous les items d'une salle et je l'ai déplacé dans la classe Room (car ce sont les items de la Room):

```
public String getItemString()
    {
        String vS = "Items :";
        Set <String> vKeys = this.aItems.keySet();
        for (String vItem : vKeys) {
            vS += " " + vItem;
        }
        return vS;
    }
```

La fonction getLongDescription est modifié de la manière suivante en prenant en compte la nouvelle fonction getItemString :

```
public String getLongDescription()
    {
        if (this.aItems.isEmpty()) {
        return "You are " + this.aDescription + "\n" +
this.getExitString()
        + "\nThere is no item here.";
       }
       else {
```

```
        return "You are " + this.aDescription + "\n" +
this.getExitString()
        + "\n" + getItemString();
       }
    }
```

A présent il ne reste plus qu'à déclarer et initialiser les items dans la procédure createRoom() de la classe GameEngine puis de les ajouter aux Rooms (voir exercice 7.22.2).

Exercice 7.22.1:

J'ai décidé d'utiliser une HashMap car cela permettra d'accéder ou de récupérer plus facilement l'item en fonction de son nom (string).

Exercice 7.22.2:

J'ai déclaré et initialisé les items dans la procédure createRoom() de la classe GameEngine :

```
        Item vLettre1 = new Item("First letter", 0.5);
        Item vCle = new Item("Key", 1);
        Item vPierre1 = new Item("First rock", 1);
        Item vCouteau = new Item("Knife", 1);
        Item vPierre2 = new Item("Second rock", 1);
        Item vPierre3 = new Item("Third rock", 1);
        Item vPierre4 = new Item("Fourth rock", 1);
        Item vLettre2 = new Item("Second letter", 0.5);
        Item vStele = new Item("Stele", 10);
        Item vCloche = new Item("Bell", 10);
```

Puis je les ai rajouté aux Rooms :

```
        vSouthJaw.addItem(vLettre1);
        vLastFloor.addItem(vCle);
        vNorthJaw.addItem(vCouteau);
        vNorthJaw.addItem(vPierre1);
        vRightNostril.addItem(vPierre2);
```

```
        vLeftNostril.addItem(vPierre3);
        vRuinsOfTheLeftEye.addItem(vPierre4);
        vRuinsOfTheLeftEye.addItem(vLettre2);
        vRightEye.addItem(vStele);
        vRightEye.addItem(vCloche);
```

Exercice 7.23:

Pour ajouter une nouvelle commande «back» j'ai commencé par déclarer un attribut aPrecedRoom dans la classe GameEngine qui contiendra la Room précédente :

```
private Room aPrecedRoom;
```

Puis j'ai modifié la procédure goRoom de la manière suivante pour que aPrecedRoom contienne la Room actuelle avant de passer à la prochaine Room :

```
  private void goRoom(final Command pCommand)
    {
        if (!pCommand.hasSecondWord()) {
            this.aGui.println("Go where ?");
            return;
        }
        String vDirection;
        vDirection = pCommand.getSecondWord();
   Room vNextRoom = this.aCurrentRoom.getExit(vDirection);
        if (vNextRoom == null) {
            this.aGui.println("There is no door !");
        }
        else {
            this.aPrecedRoom = this.aCurrentRoom;
            this.aCurrentRoom = vNextRoom;
            printLocationInfo();
        }
    }
```

Ensuite j'ai créé la procédure back(). J'ai ajouté le cas où le joueur souhaiterai revenir en arrière or qu'il vient tout juste de commencer le jeu et qu'il n'a jamais changer de Room pour cela il suffit d'afficher un message en expliquant la situation:

```
    private void back()
    {
        if (this.aPrecedRoom == null) {
                this.aGui.println("You have just started
the game and you didn't move so there is no previous
room.");
        }
        else {
        Room vCurrentRoom = this.aCurrentRoom;
        this.aCurrentRoom = this.aPrecedRoom;
        this.aPrecedRoom = vCurrentRoom;

        printLocationInfo();
        }
    }
```

Dans la classe CommandWords j'ai ajouté, dans le tableau contenant toute les commandes valide, la commande back :

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

Enfin j'ai ajouté dans la procédure interpretCommand de la classe GameEngine l'interpretation de la commande « back » :

```
        else if (vCommandWord.equals("back")){
                this.back();
            }
```

Exercice 7.24:

Dans l'exercice 7.23 j'avais déjà pris en compte le fait d'utiliser back alors qu'on n'a pas encore bouger par contre je n'avais pas pris en compte le second mot donc j'ai modifié l'interprétation de la commande « back » de la manière suivante :

- j'ai ajouté la condition suivante dans la procédure back et back prend désormais un paramètre de type Command :

```
    if (pCommand.hasSecondWord()) {
    this.aGui.println("The back command doesn't
  accept a second word.");
      return;
       }
```

Donc dans interpretCommand on a :

```
else if (vCommandWord.equals("back")){
                this.back(vCommand);
            }
```

Exercice 7.25:

Si je me déplace d'une Room A vers une Room C en passant par la Room B et que je tape 2 fois la commande « back » je devrai normalement être dans la Room A. Cependant, la commande back de l'exercice 7.23 ne fait pas cela, en effet en tapant 2 fois la commande « back » je me retrouve dans la Room C : car une première commande back permet de revenir dans la Room B puis une seconde commande « back » permet de revenir dans la Room C.

Exercice 7.26:

La classe Stack représente une pile qui suit le principe LIFO (Last In First Out). C'est grâce à cette classe que nous pourrons modifier la commande « back ».

Pour cela on commence par l'importer : `import java.util.Stack;`

Puis notre attribut aPrecedRoom qui était auparavant une Room devient une pile de Room : `private Stack<Room> aPrecedRooms;`

Ensuite on initialise cette attribut dans la procédure createRoom :

```
this.aPrecedRooms = new Stack<Room>();
```

Dans la procedure goRoom on remplace :

```
this.aPrecedRoom = this.aCurrentRoom;
```

par : `this.aPrecedRooms.push(this.aCurrentRoom);`

La méthode push permet d'ajouter un élément tout en haut de la pile.

Enfin on modifie la procédure back en utilisant la méthode empty qui permet de tester si la pile est vide et la méthode pop qui renvoie et supprime l'élément tout en haut de la liste :

```
    private void back(final Command pCommand)
    {   if (pCommand.hasSecondWord()) {
```

```
            this.aGui.println("The back command doesn't
accept a second word.");
            return;
        }
        if (aPrecedRooms.empty()) {
                this.aGui.println("You are in the first
room you were in. Going back is therefore not possible.");
        }
        else {
        this.aCurrentRoom = this.aPrecedRooms.pop();
        printLocationInfo();
        }
    }
```

Exercice 7.27:

Pour s'assurer que le jeu fonctionne correctement je pense qu'il faut tester toutes les commandes du jeu avec toutes les possibilités et s'assurer qu'il n'y ait aucune erreur ou bogue.

Exercice 7.28:

Les tests peuvent être automatisés en écrivant les commandes dans un fichier qui sera par la suite exécuter grâce à une commande « test ». Les classes à changer sont donc les classes CommandWords et GameEngine.

Exercice 7.28.1:

J'ai tout d'abord importer les classes suivantes dans la classe GameEngine :

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

Puis j'ai écrit la procédure test, présente ci-dessous. Ici la commande `try` va permettre « d'essayer » les instructions entre accolades, si l'instruction « fonctionne » on interprète la ligne lu tant qu'il y'a une prochaine ligne. La commande `catch` permet de traiter le cas où il n'y a pas de fichier (il va « attraper » l'exception FileNotFoundException). Enfin grâce à la clause `finally` on place un test pour fermer le fichier si le fichier a été ouvert auparavant :

```
    private void test(final Command pCommand)
    {
        if (!pCommand.hasSecondWord()) {
            this.aGui.println("Test what ?");
            return;
        }
        String vNomFichier = pCommand.getSecondWord();
        Scanner vSc = null;
        try {
            vSc = new Scanner(new File("./" + vNomFichier
+ ".txt"));

            while (vSc.hasNextLine()) {
                String vCommand = vSc.nextLine();
                interpretCommand(vCommand);
            }
        }
        catch (final FileNotFoundException pFNFE) {
            this.aGui.println("File not found");
        }
        finally {
            if (vSc != null) {
                vSc.close();
            }
        }
    }
```

J'ai rajouté la commande « test » dans le tableau des commandes valides de la classe CommandWords :

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

Enfin j'ai ajouté l'interprétation de la commande « test » dans la procédure interpretCommand de la classe GameEngine :

```
else if (vCommandWord.equals("test")){
            this.test(vCommand);
        }
```

Exercice 7.28.2:

J'ai créé 3 fichiers : (le fichier parcours_gagnant n'est pas totalement fini)

court	parcours_gagnant	toutes_possibilites
help	take	help
go up	go up	take
go down	take	look
go north	go down	go up
look	go north	go down
take	go north	go north
quit	take	go north
	take	go north
	go north	go north
	go north	go east
	take	go north
	go east	go west
	take	go south
	go north	back
	take	quit
	take	
	go west	

Exercice 7.29:

Chaque player a un nom, une capacité maximum d'Items, et des déplacements différents. Donc j'ai créé une classe Player avec 4 attributs :

- aName pour le nom
- aPoidsMax pour la capacité maximum d'items
- aCurrentRoom pour indiquer la room dans laquelle il se trouve
- aPrecedRoom qui est une pile qui contient tous les déplacements précédents

et un constructeur contenant deux paramètre (room et nom) :

```
private Room aCurrentRoom;
private String aName;
private double aPoidsMax;
```

```
    private Stack<Room> aPrecedRooms;

    public Player(final Room pRoom, final String pName)
    {
        this.aCurrentRoom = pRoom;
        this.aName = pName;
        this.aPoidsMax = 7;
        this.aPrecedRooms = new Stack<Room>();
    }
```

Ensuite, j'ai créé un getteur pour aCurrentRoom, aPoidsMax et aName, un setteur uniquement pour aCurrentRoom et aPoidsMax, une fonction getPrecedRoom qui renvoie et supprime le dernier élément de la pile aPrecedRooms, une procédure ajoutePrecedRooms qui ajoute une room passer en paramètre dans la pile aPrecedRooms et enfin une fonction precedRoomsEmpty qui nous dit si la pile aPrecedRooms est vide :

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

public void setCurrentRoom(final Room pRoom)
{this.aCurrentRoom = pRoom; }

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

public double getPoidsMax() {return this.aPoidsMax;}

public void setPoidsMax(final double pPoidsMax)
{this.aPoidsMax = pPoidsMax;}

public Room getPrecedRoom()
{return this.aPrecedRooms.pop();}

public void ajoutePrecedRooms(final Room pPrecedRoom)
{this.aPrecedRooms.push(pPrecedRoom);}
```

```
public boolean precedRoomsEmpty()
{return this.aPrecedRooms.empty();}
```

Enfin dans la classe GameEngine j'ai supprimer l'attribut aCurrentRoom et aPrecedRooms et je les ai remplacé par l'attribut aPlayer :

```
private Player aPlayer;
```

Que j'ai par la suite initialisé dans la méthode createRooms :

```
this.aPlayer = new Player(vSouthJaw, "Mugiwara");
```

Et j'ai modifié chaque méthode utilisant des informations sur la aCurrentRoom ou sur la pile aPrecedRooms en remplaçant par exemple :

`this.aCurrentRoom` par `this.aPlayer.getCurrentRoom`

ou encore :

```
this.aCurrent = this.aPrecedRooms.pop();
```

par:

```
this.aPlayer.setCurrentRoom(this.aPlayer.getPrecedRoom());
```

Les méthodes modifiés sont donc printLocation, goRoom, look et back.

Exercice 7.30:

Pour permettre au joueur de ramasser un seul item j'ai commencé par créé dans la classe Player un attribut aItem : `private Item aItem;`

Puis j'ai créé trois méthodes :

- une fonction getItemPlayer qui est l'accesseur de l'attribut aItem :

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

- une procédure takeItem qui permet de ramasser l'item passé en paramètre :

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

- une procédure dropItem qui permet de jeter l'item passé en paramètre :

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

Ensuite, dans la classe Room j'ai créé deux méthodes :

- une procédure removeItem qui, à l'inverse de la procédure addItem, permet de supprimer l'item passé en paramètre de la HashMap :

```
public void removeItem(final Item pItem)
{this.aItems.remove(pItem.getNom());}
```

- une fonction getItemRoom qui retourne l'item portant le nom passé en paramètre :

```
public Item getItemRoom(final String pNom)
{return this.aItems.get(pNom);}
```

Une fois ces méthodes écrite j'ai pu écrire les procédures take et drop :

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

        String vNom = pCommand.getSecondWord();

        Item vItem =
this.aPlayer.getCurrentRoom().getItemRoom(vNom);

        if (vItem==null){
            this.aGui.println("This item is not here !");
            return;
        }
        if (this.aPlayer.getItemPlayer() != null){
            this.aGui.println("You already have an item
!");
        }
        else {
```

```
            this.aPlayer.takeItem(vItem);

this.aPlayer.getCurrentRoom().removeItem(vItem);
            this.aGui.println("You took the " +
vItem.getNom());
        }
    }

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

        String vNom = pCommand.getSecondWord();

        Item vItem = this.aPlayer.getItemPlayer();

        if (!(vNom.equals(vItem.getNom()))) {
            this.aGui.println("You don't have this item
!");
        }
        else {
            this.aPlayer.getCurrentRoom().addItem(vItem);
            this.aPlayer.dropItem(vItem);
            this.aGui.println("You drop the item " +
vItem.getNom());
        }
    }
```

Enfin il suffit d'ajouter les commande « take » et « drop » dans le tableau des commande de la classe CommandWords (j'avais déjà ajouté la commande « take » dans l'exercice 7.15):

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

Et d'ajouter leur interprétation dans interpretCommand de la classe GameEngine :

```
else if (vCommandWord.equals("take"))
{this.take(vCommand);}

else if (vCommandWord.equals("drop"))
{this.drop(vCommand);}
```

<u>Remarque :</u> Etant donné que la commande take prend un second mot, le bouton take que j'ai créé dans l'exercice 7.18.8 ne sert plus à rien j'ai donc décidé de le remplacer par un bouton back (qui lui ne prend pas de second mot).

<u>Exercice 7.31:</u>

Pour permettre au joueur de ramasser plusieurs items il suffit de transformer l'attribut aItem de la classe Player en une HashMap. Pour cela on importe les classe suivantes :

```
import java.util.HashMap;

import java.util.Set;
```

Et on a : `private HashMap<String,Item> aItemsPlayer;`
(je l'ai appelé aItemsPlayer pour ne pas confondre avec la HashMap qui est dans la classe Room)

Les méthodes getItemPlayer, takeItem et dropItem sont donc modifiés :

```
   public Item getItemPlayer(final String pNom)
   {return this.aItemsPlayer.get(pNom);}
public void takeItem(final Item pItem)
{this.aItemsPlayer.put(pItem.getNom(), pItem);}


public void dropItem(final Item pItem)
{this.aItemsPlayer.remove(pItem.getNom());}
```

Les procédures take et drop ont eux aussi subit quelques changements :

- pour la procédure take la seconde condition suivante est supprimé (ainsi que le `return;` à la fin de la première condition) :

```
 if (this.aPlayer.getItemPlayer() != null)
{this.aGui.println("You already have an item !");
```

- pour la procédure drop :

  le `Item vItem = this.aPlayer.getItemPlayer();` est remplacé par

```
Item vItem = this.aPlayer.getItemPlayer(vNom);
```

  et le `if (!(vNom.equals(vItem.getNom())))` est remplacé par `if (vItem == null)`

Exercice 7.31.1:

La classe ItemList va permettre de rendre la gestions des items dans Room et Player indépendante du choix de la collection.

Pour cela j'ai créé la classe ItemList avec un attribut aItemList qui est une HashMap (que j'initialise dans le constructeur) puis je crée les méthodes `getItemList` (qui retourne l'item portant le nom passé en paramètre), `putItemList` (qui permet d'ajouter l'item passé en paramètre dans la ItemList), `removeItemList` (qui permet de retirer l'item passé en paramètre de la ItemList), `getItemListString` (qui retourne les items sous forme de String) et `ItemListIsEmpty` (qui vérifie si la ItemList est vide):

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

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


    public ItemList()
    {this.aItemList = new HashMap<String, Item>();}
```

```
    public Item getItemList(final String pNom)
    {return this.aItemList.get(pNom);}

    public void putItemList(final Item pItem)
    {this.aItemList.put(pItem.getNom(), pItem);}

    public void removeItemList(final Item pItem)
    {this.aItemList.remove(pItem.getNom());}

    public String getItemListString()
    {   String vS = "Items :";
        Set <String> vKeys = this.aItemList.keySet();
        for (String vItem : vKeys) {
            vS += " " + vItem;
        }
        return vS;
    }

    public boolean itemListIsEmpty()
    {return this.aItemList.isEmpty();}
}
```

Il suffit à présent de remplacer les deux attributs HashMap de Room et Player en ItemList et d'adapter certaines méthodes présente dans ces classes.

Exercice 7.32:

Dans l'exercice 7.29 j'avais déjà ajouté un attribut aPoidsMax dans la classe Player qui indique le poids maximum qu'un joueur peut porter.

À présent j'ai ajouté dans la classe Player un attribut aPoids qui indique le poids actuelle du joueur : `private double aPoids;`

Je l'initialise à 0 dans le constructeur : `this.aPoids = 0;`

Ensuite j'ai écrit une fonction canTake qui indique si le joueur peut prendre l'item passé en paramètre :

```
  public boolean canTake(final Item pItem)
```

```
{return this.aPoids + pItem.getPoids() <= this.aPoidsMax;}
```

J'ai modifié la procédure takeItem de la manière suivante pour qu'elle modifie le poids actuelle lorsqu'il prend un item :

```
public void takeItem(final Item pItem)
{this.aItemsPlayer.putItemList(pItem);
 this.aPoids = this.aPoids + pItem.getPoids();}
```

De même que pour dropItem :

```
public void dropItem(final Item pItem)
{this.aItemsPlayer.removeItemList(pItem);
 this.aPoids = this.aPoids - pItem.getPoids();}
```

Enfin dans la procédure take de GameEngine j'ai ajouté une condition pour que le joueur puisse prendre ou non l'item :

```
if (this.aPlayer.canTake(vItem)) {
            this.aPlayer.takeItem(vItem);

this.aPlayer.getCurrentRoom().removeItem(vItem);
            this.aGui.println("You took the " +
vItem.getNom());
            }
            else{
                this.aGui.println("Your weight limit
doesn't allow you to take this item !");
            }
```

Exercice 7.33:

J'ai tout d'abord commencé par écrire deux méthodes dans la classe Player qui me seront utile par la suite :

- Première j'ai écrit la fonction getItemPlayerString qui permet d'obtenir la string de tous les items du joueur

```
public String getItemPlayerString()
{return this.aItemsPlayer.getItemListString();}
```

- Puis j'ai écrit la fonction itemIsEmpty qui dit si la ItemList du joueur est vide :

```
public boolean itemIsEmpty()
```

```
        {return this.aItemsPlayer.itemListIsEmpty();}
```

Puis j'ai écrit la procédure inventaireItem dans la classe GameEngine qui permet d'obtenir l'inventaire des items du joueur :

```
    private void inventaireItem(final Command pCommand)
    {
        if (pCommand.hasSecondWord()) {
            this.aGui.println("Inventaire doesn't accept a
second word !");
            return;
        }

        if (this.aPlayer.itemIsEmpty()){
        this.aGui.println("You don't have item and your
weight is " + this.aPlayer.getPoids());
        }
        else{

this.aGui.println(this.aPlayer.getItemPlayerString() +
"\nWeight : " + this.aPlayer.getPoids());
        }
    }
```

Enfin j'ai rajouté la commande « inventaire » dans le tableau des commandes de la classe CommandWords ainsi que son interprétation dans la procédure interpretCommand.

<u>Exercice 7.34:</u>

Je n'avais pas ajouté la commande eat dans l'exercice 7.15 car j'avais ajouté la commande take à la place mais je ne savais pas que l'on aurai besoin de cette commande plus tard. De plus, je ne trouve pas d'autres solutions pour répondre à cette exercice tout en respectant mon scénario. J'ai donc décidé d'ajouté la commande eat maintenant.

Pour cela j'ai écrit la procédure eat dans la classe GameEngine de la façon suivante :

```
    private void eat(final Command pCommand)
    {
       if (!pCommand.hasSecondWord()) {
```

```
            this.aGui.println("Eat what ?");
            return;
        }
        String vNom = pCommand.getSecondWord();
        Item vItem = this.aPlayer.getItemPlayer(vNom);

        if (vItem == null) {
          this.aGui.println("You don't have this item !");
        }
        else {
            if (vNom.equals("magicCookie")) {
this.aPlayer.setPoidsMax(this.aPlayer.getPoidsMax()*2);
this.aPlayer.dropItem(vItem);
this.aGui.println("You ate the magic cookie !");
         }
         else {
          this.aGui.println("You can't eat this item. So
you can carry heavier items !");
         }
        }
    }
```

(Remarque : Il faut noter que une fois le magic cookie manger on le supprime de la liste des items du joueur, pour que le joueur ne puisse pas répéter l'opération plusieurs fois.)

Puis j'ajoute la commande eat dans le tableau des commandes de la classe CommandWords et son interprétation dans interpretCommand.

Enfin je crée l'item magic cookie et je l'ajoute dans l'une des salles du jeu.

Exercice 7.34.1:

Les fichiers ont été modifié de la manière suivante :

court	parcours_gagnant	toutes_possibilites
help go up	take Letter1 go up	help take

go down go north look take Letter1 quit	take Key go down go north go north take Knife take Rock1 go north go north take Rock2 go east take Rock3 go north take Letter2 take Rock4 go west	take Key take Letter1 look go up drop drop Key drop Letter1 inventaire take Letter1 inventaire take magicCookie inventaire eat eat Letter1 eat Key eat magicCookie inventaire go down go north go north go north go north go east go north go west go south back quit

Exercice 7.42 :

J'ai décidé de limité le nombre de fois qu'un joueur peut utiliser la commande « go north ».

Pour cela j'ai créé un attribut `aTimeCommand` de type `int` initialisé à zéro dans le constructeur et qui calcule le nombre de fois qu'on utilise la command « go north ».

J'ai ensuite écris la procédure timeCommand qui incrémente `aTimeCommand` de 1 et qui crée une variable `vCompteur` de type `int` qui nous donne le nombre de fois restant pour utiliser la commande « go north ».

Ensuite en fonction de la valeur de `aTimeCommand` soit en arrête le jeu soit on affiche le message indiquant le nombre de fois restant où l'on peut utiliser la commande.

```
    private void timeCommand()
    {
        this.aTimeCommand = this.aTimeCommand + 1;
        int vCompteur = 6 - this.aTimeCommand;
        if (this.aTimeCommand > 6) {
            this.aGui.println("Game Over ! \nYou used the
go north command more than 6 times so you lost !");
            this.interpretCommand("quit");
            return;
        }
        else {
            if (vCompteur == 0){
            this.aGui.println("You can't use anymore the
go north command !");
            return;
            }
            else {
            this.aGui.println("You can use the go north
command " + vCompteur + " more times !");
            }
        }
    }
```

Ensuite dans la procédure goRoom() j'ai ajouté la condition suivante :

```
  if (vDirection.equals("north")) {timeCommand();}
```

qui permet de faire appel à la procédure timeCommand si la commande « go north » est tapé.

Exercice 7.42.1 :

J'ai décidé que l'on ne peut rester dans la salle Marshy Jaw plus de 20 secondes à cause d'un gaz empoisonné présent dans l'air.

Pour cela j'ai importé dans la classe UserInterface la classe Timer :

```
import javax.swing.Timer;
```

J'ai déclaré un attribut `aTimer` de type `Timer` que j'ai initialisé dans le constructeur de la classe UserInterface comme suit :

```
this.aTimer = new Timer(20000, this);
```

Puis j'ai écrit l'accesseur de cette attribut :

```
public Timer getTimer() {return this.aTimer;}
```

Puis dans la classe GameEngine dans la procédure goRoom() j'ai rajouté la condition suivant :

```
if
(this.aPlayer.getCurrentRoom().getDescription().equals("in
the Marshy Jaw")){
            this.aGui.getTimer().restart();
            this.aGui.println("You have 20 seconds to
leave the Marshy Jaw where there is a poisoned gas !");
        }
        else {
            if (this.aGui.getTimer().isRunning()){
                this.aGui.getTimer().stop();
            }

        }
```

Cette condition permet d'activer le timer et d'afficher un message si le joueur se trouve dans la salle Marshy Jaw sinon elle permet de stopper le timer si jamais il fonctionne.

Enfin, dans la procédure actionPerformed de la classe Userinterface j'ai rajouté la condition suivante qui permet de quitter le jeu si jamais le timer est fini :

```
        else if (pE.getSource() == this.aTimer) {
```

```
            this.println("Game Over !\nYou stayed more
than 20 seconds in the Marshy Jaw where there was poisoned
gas,so you lost!");
            this.aTimer.stop();
            this.aEngine.interpretCommand("quit");
        }
```

Exercice 7.42.2 :

Je me contente de l'IHM actuelle.

Exercice 7.43 :

J'ai créé la trap door entre la salle Right Eye et Right Nostril. On peut aller de Right Eye vers Right Nostril mais pas l'inverse.

Pour cela j'ai tout d'abord supprimé dans la procédure createRooms la ligne suivante : `vRightNostril.setExit("north", vRightEye);`

Puis j'ai écrit la fonction isExit dans la classe Room qui renvoie vrai ou faux selon que la room passée en paramètre est une des sorties de la pièce courante ou pas :

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

J'ai aussi écrit les méthodes suivantes dans la classe Player qui seront utiles pour la suite :

- Cette fonction permet de renvoyer l'élément en tête de la pile sans la supprimer :

```
public Room peekPrecedRooms(){return
this.aPrecedRooms.peek();}
```

- Cette procedure permet de supprimer tout les éléments de la pile :

```
public void clearPrecedRooms()
{this.aPrecedRooms.clear();}
```

Enfin dans la procédure back de la classe GameEngine j'ai ajouté cette condition qui permet de retourner dans la salle précédente si il y'a une sortie vers celle-ci sinon elle supprime toutes les rooms de la pile et affiche un message :

```
if
(this.aPlayer.getCurrentRoom().isExit(this.aPlayer.peekPre
cedRooms())){

this.aPlayer.setCurrentRoom(this.aPlayer.getPrecedRoom());
              printLocationInfo();
            }
            else {
                this.aPlayer.clearPrecedRooms();
                this.aGui.println("You have passed through
a trap door you cannot go back !");
            }
```

Exercice 7.44 :

Pour créer des beamers j'ai tout d'abord crée une classe Beamer qui est « une sorte » d'item donc : `public class Beamer extends Item`

Cette classe contient deux paramètres (en plus de ceux de la classe Item) aCharge qui indique si le beamer est chargé et aRoomEnregistre qui indique la room enregistré lors de la charge:

```
private boolean aCharged;
private Room aRoomEnregistre;
```

On a le constructeur ainsi que les acceseurs et seulement le modificateur de l'attribut aCharge :

```
public Beamer(final String pName,final double pPoids,final
Room pRoomEnregistre)
{super(pName, pPoids);
 this.aCharged = false;
 this.aRoomEnregistre = pRoomEnregistre;}

public boolean getCharged(){return this.aCharged;}

public Room getRoomEnregistre()
{return this.aRoomEnregistre;}
```

```
public void setCharged(final boolean pBoolean)
{this.aCharged = pBoolean;}
```

Il y'a aussi un procédure qui permet de charger le Beamer :

```
public void chargeBeamer(final Room pRoomEnregistre)
{this.aCharged = true;
this.aRoomEnregistre = pRoomEnregistre;}
```

Ensuite dans la classe GameEngine j'ai ajouté deux procédures charge et fire qui permettent respectivement de charger et déclencher le beamer :

```
private void charge(final Command pCommand)
{if (!pCommand.hasSecondWord()){
    this.aGui.println("Charge what ?");
    return;
}
String vNom = pCommand.getSecondWord();
Item vItem = this.aPlayer.getItemPlayer(vNom);
if (vItem == null){
this.aGui.println("You don't have this in your inventory
!");
return;
}
if (vItem.getClass() != Beamer.class){
this.aGui.println("This item is not a Beamer !");}
else {
Beamer vBeamer = (Beamer) vItem;
vBeamer.chargeBeamer(this.aPlayer.getCurrentRoom());
this.aGui.println("The beamer is charged !");}
}

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

```
      return;}
String vNom = pCommand.getSecondWord();
Item vItem = this.aPlayer.getItemPlayer(vNom);
if (vItem == null){
this.aGui.println("You don't have this in your inventory
!");
return;}
if (vItem.getClass() != Beamer.class){
this.aGui.println("This item is not a Beamer !");}
else {
     Beamer vBeamer = (Beamer) vItem;
     if (!vBeamer.getCharged()){
     this.aGui.println("Your beamer is not charged !");}
     else {
     this.aPlayer.setCurrentRoom(vBeamer.getRoomEnregistre
     ());
     vBeamer.setCharged(false);
     printLocationInfo();
     this.aGui.println("You are right now in the room
     where the beamer was charged !");}
        }
    }
```

J'ai ensuite rajouté les commandes « charge » et « fire » dans le tableau des commandes valide de la classe CommandWords.

Enfin j'ai créé un beamer que j'ai ajouté dans la room vLastFloor (dans la procédure createRooms) :

```
Beamer vBeamer = new Beamer("Beamer",1,vSouthJaw);
```

Ici le beamer est initialisé avec la room vSouthJaw mais cela importe peu car il n'est pas chargé (en effet l'attribut aCharge est initialisé à false).

Remarque : Après avoir été téléporté dans une room, on ne peut plus utilisé back. Plus précisément, dans ce cas la commande back se comporte comme si on avait traversé une trap door. En effet, une fois le beamer chargé, on ne peut se téléporter que dans un sens (c'est donc bien une trap door). Si l'on souhaite se téléporter dans l'autre sens il faut recharger le beamer.

Exercice 7.45.1 :

court	parcours_gagnant	toutes_possibilites
help go up go down go north look take Letter1 quit	take Letter1 go up take Key go down go north go north take Knife take Rock1 go north go north take Rock2 go east take Rock3 go north take Letter2 take Rock4 go west	help take take Key take Letter1 look go up take Beamer charge Beamer drop drop Key drop Letter1 inventaire take Letter1 inventaire take magicCookie inventaire eat eat Letter1 eat Key eat magicCookie inventaire go down go north go north go north go north go east go north go west go south back fire Key fire Beamer quit

Partie IV : Déclaration obligatoire anti-plagiat

Les codes ont été rédigés par moi-même.

En général je me suis énormément aidé des échanges de discussion présent sous les exercices qui m'ont aidé à résoudre plusieurs problèmes que j'ai rencontré.

De plus je me suis aidé de certains site pour quelques exercices :

- Pour l'exercice 7.18.7, je me suis aidé du lien suivant : https://docs.oracle.com/javase/10/docs/api/java/awt/event/ActionListener.html
- Pour l'exercice 7.18.8, je me suis aidé des liens suivant :

  https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/swing/JButton.html

  https://docs.oracle.com/javase/10/docs/api/java/awt/GridLayout.html

  https://docs.oracle.com/javase/7/docs/api/java/awt/Component.html#setEnabled%28boolean%29

  https://www.jmdoudoux.fr/java/dej/chap-awt.htm
- Pour l'exercice 7.26, je me suis aidé du lien suivant : https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Stack.html
- Pour l'exercice 7.28.1, je me suis aidé du lien suivant :

  https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html

  https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/File.html

  https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/FileNotFoundException.html
- Pour l'exercice 7.42.1 j'ai utilisé les liens suivant :

  https://web.archive.org/web/20190304162253/http://www-mips.unice.fr/Doc/Java/Tutorial/uiswing/misc/timer.html#runningapi

  https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/swing/Timer.html
- Pour l'exercice 7.44 j'ai utilisé le lien suivant :

  http://blog.paumard.org/cours/java-api/chap04-introspection-classe-class.html

L'image de la carte du jeu a également été dessiné par moi-même grâce à une application, je me suis juste aidé d'une image trouver sur ce lien :
https://onepiece.fandom.com/fr/wiki/Jaya

Les images du jeu ont était pris sur ces liens :

Right Eye:

https://onepiece.fandom.com/fr/wiki/Shandora?file=Shandora_Infobox.png

Marshy Jaw :
http://ekladata.com/BcQAvXIrR3X-WcAa5jRaNtizW1o.jpg

North Jaw:
http://ekladata.com/v8AHWOc8_HCctyy_iKLTNXxc_tg.png

Dangerous Forest :
http://ekladata.com/hlwFEeJHrx85BnAMI_2uF_OR_cc.png

Left Eye:
https://onepiece.fandom.com/wiki/Shandora?file=Shandora_Ruins.png

Right Nostril :
http://ekladata.com/1AZ8t1H0HKKKkQtNoCaZuiICbnw.jpg

South Jaw:
http://nakamaonepiece.weebly.com/uploads/6/0/8/9/6089740/4396619.jpg

Last Floor:
https://onepiece.fandom.com/fr/wiki/Thousand_Sunny?file=Thousand_sunny_bibliothèque.png

Left Nostril :
http://ekladata.com/5htXBjw7Ka8Jwny8evIIJ3GL9Do.png