Rapport :

The Crossing

I.A) Hadrien GRADVOHL

I.B) Le Joueur doit traverser une forêt mais celle-ci contient de nombreux monstre, l'objectif est de sortir de cette forêt indemne en faisant les bons choix.

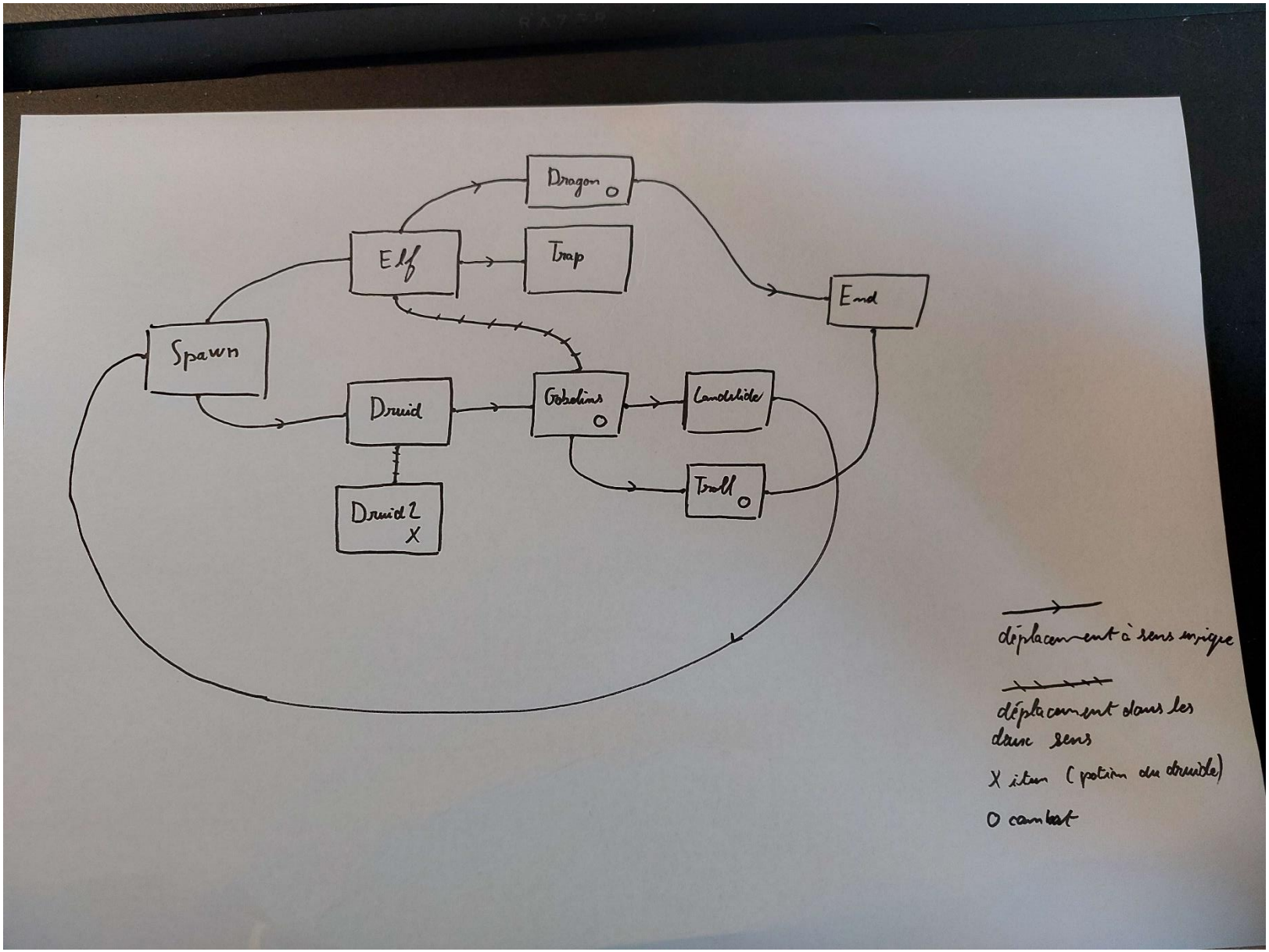
I.C)

Le personnage est le joueur lui-même dans une époque médiévale. Il rentre dans une forêt et deux chemins se présentent à lui :

-1er chemin : rencontre avec des elfes dans une clairière qui mettent en garde le joueur. Il a ensuite le choix de prendre trois chemins, celui de gauche mène à un dragon, celui du milieu mène à une route piégée qui tue le joueur qui doit ensuite recommencer le jeu, la dernière route mène à un groupe de gobelins auxquelles le joueur devra faire face (situation décrite dans la partie du deuxième chemin), si le joueur choisit le dragon et qu'il gagne le combat ce dernier gagne la partie.

-2ème chemin : le joueur arrive dans une maison, il voit un escalier, s'il décide de prendre cet escalier, il va parler à un druide : il lui fait une énigme à laquelle il doit répondre par oui ou non si la réponse est bonne, le joueur prend une potion qui l'aidera plus tard dans l'aventure chez le druide le joueur obtient également des lunettes qui n'ont pas de capacités spécifiques. Après la maison le joueur affronte les mêmes goblins que dans le premier chemin. Il a ensuite la possibilité d'aller dans deux chemins l'un d'eux mène à un éboulement qui tue le joueur et l'autre qui mène à un troll. S'il choisit de se battre le joueur meurt. Si le combat est gagné le joueur réussi traverser la forêt et gagne la partie.

I.D)


Dragon
Elf
Trap
End
Spawn
Druid
Gobelins
Cavalerie
Troll
Druid 2
X
déplacement à sens unique
déplacement dans les deux sens
X item (potion du druide)
O combat

I.E)

Le personnage est le joueur lui-même dans une époque médiévale. Il rentre dans une forêt (au spawn le joueur obtient un chapeau qui ne lui servira à rien dans l'aventure) et deux chemins se présentent à lui :

-1er chemin : rencontre avec des elfes dans une clairière qui mettent en garde le joueur. Il a ensuite le choix de prendre trois chemins, celui de gauche mène à un dragon, celui du milieu mène à une route piégée qui tue le joueur qui doit ensuite recommencer le jeu, la dernière route mène à un groupe de gobelins auxquelles le joueur devra faire face (situation décrite dans la partie du deuxième chemin), si le joueur choisit le dragon et qu'il gagne le combat ce dernier gagne la partie.

-2ème chemin : le joueur arrive dans une maison, il voit un escalier, s'il décide de prendre cet escalier, il va parler à un druide : il lui fait une énigme à laquelle il doit répondre par oui ou non si la réponse est bonne, le joueur prend une potion qui l'aidera plus tard dans l'aventure. Après la maison le joueur affronte les mêmes gobelins que dans le premier chemin. Il a ensuite la possibilité d'aller dans deux chemins, l'un d'eux mène à un éboulement qui tue le joueur et l'autre mène à un troll. S'il choisit de se battre le joueur meurt. Si le combat est gagné le joueur réussi traverser la forêt et gagne la partie.

Clairière Gobelin :

Gagne le combat si le joueur décide de lui mettre un high kick sinon le joueur meurt et repart du début.

Clairière du dragon :

Pour combattre le dragon le joueur a trois possibilités : fuir (s'il fuit il meurt), le combattre à la seul force de ses poings (n'arrivera pas à le battre sauf si il a pris la potion que lui à donner le druide), lui lancer un regard perçant (il bat le dragon). Si le joueur gagne son combat il arrive à finir le jeu.

Combat contre le troll :

Le joueur doit choisir de courir le plus vite possible pour échapper au troll. S'il choisit de se battre le joueur meurt. Si le combat est gagné le joueur réussi traverser la forêt et gagne le jeu.

I.F)

Spawn

Elf

Dragon

Druid

Druid 2

Gobelins

Landslide

Troll

End

Dragon

Trap

Potion du druide

Lunettes

Chapeau

I.G) Le joueur gagne s'il arrive à sortir de la forêt.

I.H) Enigme du druide pas encore décidé

II)

Exercice 7.5

On crée cette procédure pour afficher au joueur sa position. La situation était inacceptable car auparavant on n'affichait pas au joueur dans quelle room il se situait. Ce changement dans le code permet également de de simplifier le code en évitant les redondances.

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

}
```

Exercice 7.6

Dans cet exercice on définit les room et les sorties du jeu de la façon suivante :

```
Room vSpawn = new Room ("You are at the entrance of the forest");
Room vDruid = new Room ("You arrive in the house of a druid");
Room vElf = new Room ("You arrive in front of an elf");
Room vTrap = new Room ("You are now in a trap");
Room vGobelins = new Room ("A tribe of gobelin appears in front of
Room vDragon = new Room ("You find yourself in front of a dragon");
Room vLandslide = new Room ("Many stones fall on you chock");
Room vTroll = new Room ("You have awakened a troll who now wants to
Room vEnd = new Room ("You've won the game");


//Affectation des sorties à chaque pièce
vSpawn.setExits(vElf,vDruid,null,null);
vDruid.setExits(vElf,null,vGobelins,null);
vElf.setExits(vDragon,vDruid,vTrap,null);
vTrap.setExits(null,null,null,null);
vGobelins.setExits(vDragon,vTroll,vLandslide,null);
vDragon.setExits(null,null,vEnd,null);
vLandslide.setExits(null,null,null,null);
vTroll.setExits(null,null,vEnd,null);
vEnd.setExits(null,null,null,null);

//Initialisation du jeu
this.aCurrentRoom = vSpawn;
```

On créé également un accesseur dans la classe room (getExit).

Et on résout le problème unknown direction en modifiant de la façon suivante :

```
if(aCurrentRoom.getExit(vDirection)!=null){
     vNextRoom=aCurrentRoom.getExit(vDirection);
 } else{
     System.out.println("Unknown Direction!");
     return;
 }
```

Exercice 7.7 :

Méthode qui permet d'afficher au joueur les issues possibles dans une room :

```
ublic String getExitString() {
    String aExitString="Exits :";
    if(aNorthExit!=null)
    {aExitString+=aExitString + "North";}
    if(aSouthExit!=null)
    {aExitString+=aExitString + "South";}
    if(aEastExit!=null)
    {aExitString+=aExitString + "East";}
    if(aWestExit!=null)
    {aExitString+=aExitString + "West";}
    return aExitString;
```

Changements effectués dans la classe Game afin de faire fonctionner le programme avec les nouvelles modifications :

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

IL est logique pour la classe room de produire des informations sur ses sorties sans les afficher car le programme à besoin de ces informations pour passer d'une pièce à l'autre. La classe game doit afficher ces informations pour que le joueur puisse être au courant des sorties disponibles sans pour autant afficher deux fois ces sorties.

7.8

Changement de l'écriture des sorties pour ajouter un étage :

```
vSpawn.setExit("North",vElf);
vSpawn.setExit("South",vDruid);
vDruid.setExit("East",vGobelins);
vDruid.setExit("Up",vDruid2);
vDruid2.setExit("Down",vDruid);
vElf.setExit("North",vDragon);
vElf.setExit("South",vGobelins);
vElf.setExit("East",vTrap);
vGobelins.setExit("South",vTroll);
vGobelins.setExit("East",vLandslide);
vLandslide.setExit("Respawn",vSpawn);
vGobelins.setExit("North",vElf);
vDragon.setExit("East",vEnd);
vTroll.setExit("East",vEnd);
```

Modifications effectuées après l'ajout de hashmap dans la classe Room :

```
public void setExits (final Room pNorthExit, final Room pSouthExit, final Room pEastExit, final Room pWestExit)
{
    this.aNorthExit = pNorthExit;
    this.aSouthExit = pSouthExit;
    this.aEastExit = pEastExit;
    this.aWestExit = pWestExit;
}
public Room getExit (String direction)
{
    return aExit.get(direction);
}
public String getExitString() {
    String aExitString="Exits :";
    if(aNorthExit!=null)
    {aExitString+=aExitString + "North";}
    if(aSouthExit!=null)
    {aExitString+=aExitString + "South";}
    if(aEastExit!=null)
    {aExitString+=aExitString + "East";}
    if(aWestExit!=null)
    {aExitString+=aExitString + "West";}
    return aExitString;
}
public void setExit(String direction, Room neighbor){
    aExit.put(direction, neighbor);
}
```

7.9

Ajout de keys pour afficher le nom des sorties des room :

```java
public String getExitString() {
    String aExitString="Exits :";
    Set<String>keys=aExit.keySet();
    for (String exit : keys){
        aExitString= aExitString + " " + exit;
    }
    return aExitString;
}
```

7.10

On ajoute des commentaires du code en java doc.

7.11

Dans cet exercice on ajoute getLongDescription dans la classe Room ce qui permet. Ce qui permet de simplifier le code en déplaçant la description de la localisation du personnage dans la classe room.

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

}
```

```java
private void printLocationInfo()
{
    System.out.println(aCurrentRoom.getLongDescription());
}
```

7.14

Cette exercice nous permet d ajouter la méthode look dans le programme, et également de définir look comme une commande.

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

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

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

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

```
private boolean processCommand (final Command pCommand){
    String commandWord=pCommand.getCommandWord();
    if (pCommand.isUnknown()){
        System.out.println("I don't know what you mean...");
        return(false);
    }
    if (commandWord.equals("go")){
            goRoom(pCommand);
            return(false);
    }else if (commandWord.equals("quit")){
            return(this.quit(pCommand));
    }else if (commandWord.equals("help")){
            printHelp();
            return(false);
    }else if(commandWord.equals("look")) {
            look();
            return(false);

    }else{
            System.out.println("Unknown command!");
            return(false);
    }
}
```

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

7.15

Dans cette exercice en utilisant le même processus que l'exercice précédent on ajoute la méthode eat au programme.

```
private boolean processCommand (final Command pCommand){
    String commandWord=pCommand.getCommandWord();
    if (pCommand.isUnknown()){
        System.out.println("I don't know what you mean...");
        return(false);
    }
    if (commandWord.equals("go")){
            goRoom(pCommand);
            return(false);
    }else if (commandWord.equals("quit")){
            return(this.quit(pCommand));
    }else if (commandWord.equals("help")){
            printHelp();
            return(false);
    }else if(commandWord.equals("look")) {
            look();
            return(false);
    }else if(commandWord.equals("eat")) {
        eat();
        return(false);

    }else{
            System.out.println("Unknown command!");
            return(false);
    }
}
```

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

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

7.16

Dans cet exercice on créé une méthode permettant d'afficher toute les commandes que le joueur pourra utiliser.

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

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

On modifie également print Help pour y intégrer la méthode showCommands() pour que le joueur puisse voir quelles sont les commandes qu'il peut utiliser en tapant help.

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

7.18

On remplace la méthode showAll par la méthode getCommandList() pour ne plus avoir de problème de compilation.

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

7.18.1

Dans cet exercice on restructure le jeu en créant deux nouvelles classes qui permettront de nous faciliter la tâche dans le futur.

7.18.3

Dans cette partie j’ai généré les images de mon jeu à laide d’une IA et je les ais toutes redimensionnés.

Voici une partie de ces images :

7.18.4

Le titre du jeu est The Crossing

7.18.5

Dans cette partie on fait en sorte que les Room situés dans createRooms soient accessibles depuis les autres classes.

7.18.6

Dans cette partie on modifie le programme pour qu'il soit lancé sur une interface graphique.

7.18.8

Dans cette partie on crée les bouton que le joueur pourra utilisé

```
this.aPanelEast = new JPanel();
this.aPanelEast.setLayout(new GridLayout(2,3));

this.aButtonNorth = new JButton ("North");
this.aButtonNorth.addActionListener( this );
this.aButtonNorth.setForeground(Color.white);
this.aButtonNorth.setBackground(Color.black);
this.aPanelEast.add(this.aButtonNorth);
```

```
public void enable( final boolean pOnOff )
{
    this.aEntryField.setEditable( pOnOff ); // enable/disable
    if ( pOnOff ) { // enable
        this.aEntryField.getCaret().setBlinkRate( 0 ); // cursor blink
        this.aEntryField.removeActionListener( this ); // reacts to entry
    }
    this.aButtonNorth.setEnabled(pOnOff);
    this.aButtonSouth.setEnabled(pOnOff);
    this.aButtonEast.setEnabled(pOnOff);
    this.aButtonWest.setEnabled(pOnOff);
    this.aButtonEat.setEnabled(pOnOff);
    this.aButtonUp.setEnabled(pOnOff);
    this.aButtonDown.setEnabled(pOnOff);
} // enable(.)
```

```
public class UserInterface implements ActionListener
{
    private GameEngine aEngine;
    private JFrame     aMyFrame;
    private JTextField aEntryField;
    private JTextArea  aLog;
    private JLabel     aImage;
    private JButton aButtonNorth;
    private JButton aButtonSouth;
    private JButton aButtonEast;
    private JButton aButtonWest;
    private JButton aButtonEat;
    private JButton aButtonDown;
    private JButton aButtonUp;
    private JPanel  aPanelEast;
    private JPanel aPanelWest;
```

```
vPanel.add(this.aPanelEast, BorderLayout.EAST);
this.aMyFrame.getContentPane().add( vPanel, BorderLayout.CENTER );
```

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

7.19.2

Dans cette partie on intègre les images dans le jeu

```
 void createRooms()

m vSpawn, vDruid, vDruid2, vElf, vTrap, vGobelins, vDragon, vLandslide, vTroll, vEnd;
pawn = new Room (" at the entrance of the forest","Image/vSpawn.jpeg");
ruid = new Room (" in a house and you see a staircase","Image/vDruid1.jpeg");
ruid2 = new Room ("in th laboratory of a druid","Image/vDruid2.jpeg");
lf = new Room (" in front of an elf","Image/vElf.jpeg");
rap = new Room (" now in a trap","Image/vTrap.jpg");
obelins = new Room (" in front of a tribe of gobelins ","Image/vGobelins.jpeg");
ragon = new Room (" in front now of a dragon","Image/vDragon.jpeg");
andslide = new Room (" in a dead end and many stones fall on you","Image/vLanslide.jpeg");
roll = new Room (" you are ina clearing and you have awakened a troll who now wants to attack you","Image/vTroll.jpeg");
nd = new Room (" at the end of the forest, you've won !!!!","Image/vEnd.jpeg");
```

7.20

Création de la classe Item contenant toutes les méthodes nécessaires pour définir les items.

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


    public Item(final String pDescription, final double pPoids ) {
        this.aDescription=pDescription;
        this.aPoids=pPoids;
    }
    public void setDescription( final String pDescription) {
        this.aDescription= pDescription;
    }
    public void setPoids(final double pPoids) {
        this.aPoids=pPoids;
    }
    public String getDescriptionItem(){
        return this.aDescription;
    }
    public double getPoidsItem(){
        return this.aPoids;
    }
}
```

7.21

Affiche les items présents dans chacune des room.

```
public Room( final String pDescription, final String pImageName){
    this.aDescription=pDescription;
    this.aExits = new HashMap();
    this.aItem=new HashMap();
    this.aImageName=pImageName;
}
```

```
public String getLongDescription()
{
    if(this.aItem!=null){
      return "You are in "+this.aDescription + ". \n" + this.getExitString()+"\n"+this.getItemString();
```

```
public String getLongDescription()
{
    if(this.aItem!=null){
      return "You are in "+this.aDescription + ". \n" + this.getExitString()+"\n"+this.getItemString();

    } else {
        return "You are in"+this.aDescription+". \n" + this.getExitString()+"\n"+"There is no item in this room.";
    }
}
```

7.22

Dans cet exercice on modifie getItemString afin d voir plusieurs items dans la même dans la même salle

```
public String getItemString(){
    if (this.aItem.isEmpty()){
        return "No item here.";
    }
    String vItem = "Items are :";
    Set <String> vItemName = this.aItem.keySet();
    for (String vNom : vItemName){
        vItem+=" "+vNom;
    }
    return vItem;
}
public void setItem(final String pName, final Item pItem){
    this.aItem.put(pName, pItem);

}
```

7.22.2

On déclare tous les items : leurs noms, leurs description et leur poids.

```
Item vPotion= new Item(" this potion is  very precious and will usefull later",0.100 );
Item vGlasses= new Item(" they have no use expect making you even more beautiful",0.02);
Item vHat=new Item(" this magnificent hat is useless",0.4);
vDruid2.addItem("potion",vPotion);
vDruid2.addItem("glasses",vGlasses);
vSpawn.addItem("hat",vHat);
```

7.23)

Dans cet exercice on fait en sorte que le joueur puisse revenir à la room précédente.

```
private Room backRoom;
```

```
this.backRoom = this.aCurrentRoom;
this.aCurrentRoom = vNextRoom;
this.printLocationInfo();
```

```
private void back(){
    Room vbackRoom=this.aCurrentRoom;
    this.aCurrentRoom= this.backRoom;

    if(this.aCurrentRoom.getImageName()!=null){
        this.aGui.showImage(this.aCurrentRoom.getImageName());
    }
    this.printLocationInfo();
}
```

7.26)

On utilise stack pour enregistrer les room dans lesquels le joueur est passé ce qui permet plus facilement de retourner à la room précédente.

```
private Stack<Room> backRoom;

public GameEngine() {
    this.aParser = new Parser();
    this.createRooms();
    this.backRoom = new Stack();
```

```
else {
    this.backRoom.add(this.aCurrentRoom);
    this.aCurrentRoom = vNextRoom;
    this.printLocationInfo();
}
```

```
private void back(){
    Room vbackRoom=this.aCurrentRoom;
    if (!this.backRoom.isEmpty()) {
    this.aCurrentRoom= this.backRoom.peek();

    if(this.aCurrentRoom.getImageName()!=null){
        this.aGui.showImage(this.aCurrentRoom.getImageName());
    }
    this.printLocationInfo();
    this.backRoom.pop();
}
}
```

7.28.1)

On créé une commande test qui permet de rentrer une suite de commandes pour voir si le code fonctionne.

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

    String vFileToRead = pCommand.getSecondWord();
    try {
        Scanner vScan = new Scanner(new File("" + vFileToRead + ".txt"));
        while (vScan.hasNextLine()) {
            this.interpretCommand(vScan.nextLine());
        }
    } catch (final FileNotFoundException vE){
        this.aGui.println("There is no such file of testing !");
    }
}
```

7.28.2

Dans cette partie on créé les fichier test qui contiennent les commandes.

7.29

On créé une nouvelle classe Player

Cette classe sert à simplifier game engine à l'aide des différentes méthodes que l'on créé dans cette dernière.

```java
import java.util.Stack;

/**
 * Décrivez votre classe Player ici.
 *
 * @author (votre nom)
 * @version (un numéro de version ou une date)
 */
public class Player
{
    private String aName;
    private Room aCurrentRoom;
    private Stack<Room> backRooms;

    public Player(final String pName, final Room pCurrentRoom) {
        this.aName = pName;
        this.aCurrentRoom = pCurrentRoom;
        this.backRooms = new Stack();
    }

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

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

    public Stack<Room> getPreviousRoomStack() {
        return this.backRooms;
    }

    public void goRoom(final String pNextRoom) {
        this.backRooms.push(this.aCurrentRoom);
        Room vNextRoom = this.aCurrentRoom.getExit(pNextRoom);
        this.aCurrentRoom = vNextRoom;
    }

    public void setCurrentRoom(final Room pRoom) {
        this.aCurrentRoom = pRoom;
    }
    public void back() {
        this.aCurrentRoom = this.backRooms.pop();
    }
}
```

```
}
private void back(final Command pCommand){
    if (pCommand.hasSecondWord()){
        this.aGui.println("You can't back in a particular place !");
        return; //Arret prématuré
    }
    if (this.aPlayer.getPreviousRoomStack().isEmpty()) {
    this.aGui.println("There is no previous room");
        return;
    }
    this.aPlayer.back();
    this.printLocationInfo();
}
```

```
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.aPlayer.getCurrentRoom().getExit( vDirection );
    //this.aGui.println(this.backRoom.getShortDescription());
    if ( vNextRoom == null )
        this.aGui.println( "There is no path!" );
    else {
        this.aPlayer.goRoom(vDirection);
        this.printLocationInfo();
    }

}
```

7.30)

Dans cet exercice nous créons les commandes take et drop qui permettent au joueur de prendre et de lâcher un item.

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

    private void take(final Command vCommand) {
        String vItem = vCommand.getSecondWord();

        if (this.aPlayer.getCurrentRoom().getItem(vItem) != null) {
            this.aPlayer.take(vItem);
        }
    }
    private void drop(final Command vCommand) {
        String vItem = vCommand.getSecondWord();

        if (this.aPlayer.getInventory() != null) {
            this.aPlayer.drop(vItem);
        }
    }
```

```
public void take(final String pItemName) {
    this.aInventory = this.aCurrentRoom.getItem(pItemName);
    this.aCurrentRoom.removeItem(pItemName);
}

public void drop(final String pItemName) {
    this.aCurrentRoom.addItem(pItemName, this.aInventory);
    this.aInventory = null;
}
// a constant array that will hold all valid command words
private static final String[] validCommands = {
    "go","exit","help","look","eat", "Respawn", "back", "test", "take", "drop"};
```

7.31)

Dans cette partie on modifie le programme de façon à ce que le joueur ai plusieurs item dans son inventaire.

```
public HashMap getInventory() {
    return this.aInventory;
}


public void setInventory(final Item pItem) {
    this.aInventory.put(pItem.getName(), pItem);
}

public void take(final String pItemName) {
    this.aInventory.put(pItemName, this.getCurrentRoom().getItem(pItemName));
    this.aCurrentRoom.removeItem(pItemName);
}

public void drop(final String pItemName) {
    this.aCurrentRoom.addItem(pItemName, this.aInventory.get(pItemName));
    this.aInventory.remove(pItemName);
}
```

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

public Player(final String pName, final Room pCurrentRoom) {
    this.aName = pName;
    this.aCurrentRoom = pCurrentRoom;
    this.backRooms = new Stack();
    this.aInventory = new HashMap();
}
```

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

}
public Item getItem(final String pItemName) {
    if (this.aItem.get(pItemName) == null) {
        return null;
    } else {
        return this.aItem.get(pItemName);
    }
    }
public void removeItem(final String pItemName) {
    if (this.aItem.get(pItemName) == null) {
        return;
    } else {
        this.aItem.remove(this.aItem.get(pItemName));
        return;
    }
}
```

1.31.1

Nous créons itemList pour que le joueur est un inventaire

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

    public ItemList(){
        this.aInventory= new HashMap<>();
    }
    public void addItem(final String pName, final Item pItem){
        this.aInventory.put(pName, pItem);

    }
    public String getItemName(final String pItemName) {
        return this.aInventory.get(pItemName).getName();
    }
    public void removeItem(final String pItemName) {
        if (this.aInventory.containsKey(pItemName)) {
            this.aInventory.remove(pItemName);
        }
    }
    public String getInventory() {
        String items = "Inventaire :";
        Set<String> keys=this.aInventory.keySet();

        for(String itemKey : keys) {items += itemKey+ " "  ;
        }
        return items;
    }
    public Item getItem(final String pItemName) {
        return this.aInventory.get(pItemName);
    }
    /*
     * Vérifie si un item est dans la liste
     * @param pItemName
     * @return
     */
    public boolean hasItem(final String pItemName) {
        if (this.aInventory.containsKey(pItemName)) {
            return true;
        } else {
            return false;
        }
    }
```

```java
public boolean isEmpty() {
    if (this.aInventory.isEmpty()) {
        return true;
    } else {
        return false;
    }
}

    /**
 * Retourne la liste des items en String
 * @return String
 */
public String getItemsString() {
    String items = "Items: ";
    Set<String> keys = this.aInventory.keySet();

    for(String itemKey : keys) {
        items += itemKey + " ";
    }

    return items;
}
```

7.32

Dans cet exercice on fait en sorte que le joueur ne puisse pas porter plus d'un certains poids maximum d'item.

```
public Player(final String pName, final Room pCurrentRoom) {
     this.aName = pName;
     this.aCurrentRoom = pCurrentRoom;
     this.backRooms = new Stack();
     this.aInventory = new ItemList();
     this.poidsMaximum=100;
     this.poidsTotal=0;
 }

 public void setPoidsMaximum(final double pMax) {
     this.poidsMaximum = pMax;
 }
```

```
public void take(final String pItemName) {
 this.aInventory.addItem(pItemName, this.getCurrentRoom().getItem(pItemName));
 this.aCurrentRoom.removeItem(pItemName);
 this.poidsTotal += this.aInventory.getItem(pItemName).getPoidsItem();
}

public void drop(final String pItemName) {
    this.aCurrentRoom.addItem(pItemName, this.aInventory.getItem(pItemName));
    this.aInventory.removeItem(pItemName);

}
public double getPoidsTotal() {
    return this.poidsTotal;
}
public void setPoidsTotal(final double pPoidsTotal) {
    this.poidsTotal=pPoidsTotal;

}
```

7.33

Dans cette partie on modifie le programme pour que l'inventaire du joueur soit affiché.

```
private void inventory(final Command pCommand) {
    if (pCommand.hasSecondWord()) {
        return;
    }

    this.aGui.println(this.aPlayer.getInventory().getItemsString());
    }
```

7.34

On ajoute l'item magic cookie qui permet au joueur de porter plus d item.

```
public void eat(final String pItem) {
    Item vItem = this.aInventory.getItem(pItem);

    if (pItem.equals("magiccookie")) {
        this.poidsMaximum *= 2;
    }

    this.aInventory.removeItem(pItem);
}
```