Mezouar Mikael Esiee Paris IGE-3007

Rapport projet programmation JAVA (A3P(AL) 2020/2021 G3)

28/10/2021 1 ère année Cycle Ingénieur

Titre du Projet :Un joueur perdu a Anfield.

Personnage : Entraîneur

Lieu : Stade d'Anfield

Introduction :

Le but de ce projet est de créer un jeu d'aventure en Java. (à compléter)

I.A) Auteur :

Mezouar Mikael E3T

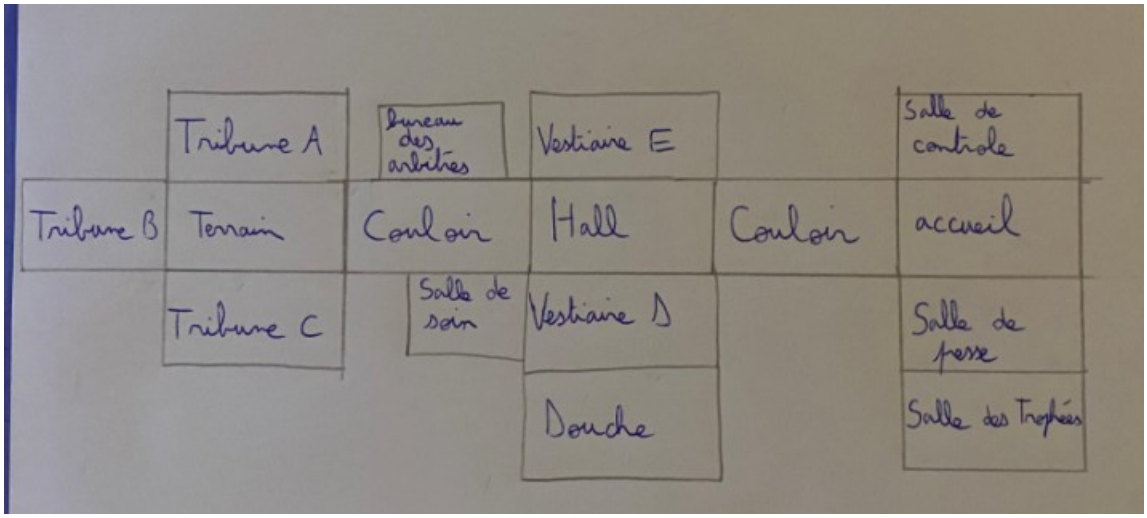
I.B) Thème (phrase-thème validée) :

Un entraîneur à la recherche de son joueur à Anfield .

I.C) Résumé du scénario (complet)

Le but du jeu sera de retrouver un joueur perdu dans le stade ,en incarnant le personnage de l'entraîneur.

I.D) Plan



Parking supprimé dans la version finale*

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

Scenario non finalisé pour l'instant . Le match débute dans moins d'1h et la star de l'équipe est introuvable , l'entraîneur doit tout faire pour le retrouver . La star de l'équipe est enfermé dans la salle des Trophée. Un code permet de l'ouvrir , ce code est composé d'une suite de numéro composé d'énigme autour du club de Liverpool dont le joueur pourra soit trouver si il a une bonne connaissance du club , soit il pourra aller chercher les réponses dans les différentes pièce du jeu.

Le code sera 8 6 19 1892 :

8- numéro maillot de Gerrard **(tribune A)**
6- nombre de ligue des champions **(mur du vestiaire domicile)**
19-nombre de championnat remporté **(dans le couloir A)**
1892-date de création du club **(de la tribune B tu peux le distinguer)**

à l'accueil il sera possible d'avoir une carte des lieux .

I.F) Détail des lieux, items, personnages

- Anfield (Liverpool)
-Entraineur personnage principal
-le joueur qui doit être retrouvé
-clés ou code pour accéder à certaine pièce.
-autre personnage à venir

I.G) Situations gagnantes et perdantes

Joueur retrouvé (porte de la salle des trophée ouverte) = situation gagnante
Joueur non retrouvé avant le coup d'envoi (nombre de coup limite) OU abandon = situation perdante

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

code de la salle ou est le joueur à trouver.

I.I) Commentaires (ce qui manque, reste à faire, …)

Scénario et Mini solution de résolution du jeu à réfléchir
Faire la documentation
Rentrer les pièces du jeu quand le plan sera définitif

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

Exercice 7.5 :

Oui la création de cette procédure permet d'éviter la duplication de code et de faire 2 action c,'est à dire ou l'on est , puis afficher les sorties possible.

```
this.printLocationInfo();
// System.out.println("\nYou are in the"+this.aCurrentRoom.getDescription());

// System.out.print("Exits :");
// // if (this.aCurrentRoom.aNorthExit!=null) System.out.println("north");
// // if (this.aCurrentRoom.aSouthExit!=null) System.out.println("south");
// // if (this.aCurrentRoom.aEastExit!=null) System.out.println("east");
// // if (this.aCurrentRoom.aWestExit!=null) System.out.println("west");
```

Exemple ici dans la procédure goRoom on a remplacé tout ce qui est en commentaire commentaire par l'appel de notre nouvelle procédure , on peut voir très nettement ici qu'on évite beaucoup de répétition de code .

Exercice 7.6 :

Nous n'avons pas besoin de tester si le paramètre est nul, car même si il est nul , nous devons quand même le stocker pour que le programme comprenne que dans la direction ou il y a un null correspond a une direction qui n'a pas de sortie .

Exercice 7.7 :

Il est logique de faire cela car on peut maintenant changer la classe Room a notre guise , sans devoir changer le code principal du jeu.

Exercice 7.8 :

On a remplacé setExits par setExit car maintenant on initialise les sortie une par une et non plus toute les sortie d'un coup.

Exercice 7.9 :

Keyset permet de récupérer les clef du dictionnaire . Ils sont stocké dans une variable de type « set » .

```
(String vexit : this.aexits.keySet()){  //parcours les directions de la Room
```

Exercice 7.10 :

Tout d'abord on crée une variable String qu'on retournera à la fin , la fonction récupère les clés du dictionnaire et renvoi la variable de retour avec toute les clés récupérer.

```
public String getExitString ()
{
   String vReturn="Exits :";
   for(String vexit : this.aexits.keySet()){  //parcours les directions de la Room
       vReturn+=' '+ vexit;
}
return vReturn;

}
```

Rappel :

HashMap = dictionnaire
set= type ensemble (qui peut pas avoir de doublon
keySet()= fonction du Hashmap qui génère un type set

Exercice 7.10.2 :

La classe Game contient beaucoup moins de méthode que la classe Room, en effet la classe Room contient beaucoup de méthodes private et donc elles se sont pas dans la javadoc.

Exercice 7.14:

```
public void look(){
    printLocationInfo();
}
```

Exercice 7.15:

```
public void eat(){
    System.out.println("tu viens de manger");
}
```

Exercice 7.18:

Pour cette question on fait en sorte que la classe CommandWords n'affiche plus les commandes valides mais les revoies sous forme de String , de sorte que l'on puisse choisir la manière de les afficher.

```
/**
 * Fonction qui return une liste des commandes qui sont valides
 */

public String getCommandList()
{
    String vCommande="";
    for(String command: this.aValidCommands){
        vCommande+=(command+" ");
    }
    return vCommande;
}
```

Exercice 7.18.1:

Mon code jusqu'ici est fonctionnel donc je n'ai pas apporté de modification particulière pour cette question .

Exercice 7.18.4:

« Un joueur Perdu à Anfield » , « you'll never walk alone » (autre idée).

Exercice 7.18.6:

2)c) on a plus besoin de scanner car maintenant on utilise une interface graphique et le scanner servait pour la version avec le terminal.

3) et 4) Après avoir compris le code de zuul with image , je l'ai incorporé au mien , en l'adaptant à mes images , avec mon titre . Ce qu'on peut dire sur la UserInterface , on a 5 attributs , un qui représente le moteur du jeu , un autre la fenêtre entière du jeu , un la zone ou l'on écrit , un autre qui représente la zone ou les informations qui s'affichent , et un dernier qui représente l'image.

```
private GameEngine aEngine; //moteur du jeu
private JFrame     aMyFrame; //fenetre entiere
private JTextField aEntryField; //zone ou on ecrit
private JTextArea  aLog; //zone ou les infos s'affiche
private JLabel     aImage;
```

Ensuite il y a eu quelque adaptation a faire dans la classe GameEngine , comme ajouter les images à chaque pièce du jeu ou en changeant tout les System.out.println(...) par des this.aGui.println(…) car on affiche plus rien dans le terminal maintenant , mais on travaille uniquement sur la fenêtre du jeu .

5)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.URL;
//import java.awt.image.*;
```

Après avoir essayer de les mettre en commentaire une par une , c'est la ligne import java.awt.image.* qui n'avait pas incidence sur la compilation donc on l'a supprime pour la suite.

Exercice 7.18.8:

Pour ajouter un bouton, j'ai crée un nouvelle attribue bouton , puis en m'inspirant du code qu'on nous a donné précédemment je lui ai affecté la position « east ». Puis dans la méthode actionPerformed() je lui assigne la commande qui correspond au bouton :

```
private GameEngine aEngine; //moteur du jeu
private JFrame     aMyFrame; //fenetre entiere
private JTextField aEntryField; //zone ou on ecrit
private JTextArea  aLog; //zone ou les infos s'affiche
private JLabel     aImage;
private JButton aBouton;
```

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

```
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) :

    Object vE=pE.getSource();
    if(vE==aBouton){
        this.aEntryField.setText("go east");
        this.processCommand();
    }
    else if(vE==aEntryField){
        this.processCommand(); // never suppress this line
    }
} // actionPerformed(.)
```

Exercice 7.20:

Pour cette question j'ai crée une classe Item avec 2 attributs (poids et description), 1 constructeur et 2 accesseurs afin de pouvoir accéder au attribut en dehors de la classe.4

Ensuite dans Room, j'ai crée un nouvel attribut Item , une classe setItem pour attribuer un item à une Room. Puis j'ai crée une fonction getItemString() afin de récupérer une description d'un item lorsque la room en possède une. J'ai donc ajouté un appel de fonction à cette fonction dans getLongDescription afin d'avoir une description d'un item lorsqu'il y en a un dans la Room dans laquel on se trouve pendant le jeu.

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

public String getItemString(){
    String vR=" Item :";
    if (this.aItem==null){
        return vR+" pas d'item ici";
    }
    else{
        return vR+this.aItem.getPoids()+"kg"+" "+this.aItem.getDescription(
    }
}
```

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

Exercice 7.21:

Je n'ai rien changé pour cette question.

Exercice 7.22:

J'ai remplacé l'attribut item par une Hashmap d'items car c'est la collection que je maitrise le mieux et qu'on l'a utilisé précédemment. J'ai donc modifié les méthodes setItem et getItemString en conséquence de cela .

```java
public void setItem(final Item pI)
{
    this.aItems.put(pI.getDescription(),pI);

}

public String getItemString(){
    // String vR=" Item :";
    // if (this.aItem==null){
        // return vR+" pas d'item ici";
    // }
    // else{
        // return vR+this.aItem.getPoids()+"kg"+" "+this.aItem.
    // }
    String vReturn="Items :";
    for(String vI : this.aItems.keySet())
    {
       vReturn=vReturn+' '+ vI;
    }
    return vReturn;

}
```

Exercice 7.22.2:

```java
//déclaration des items

Item vI=new Item(1,"clé");
Item vI3=new Item(1,"carte");
Item vI2=new Item(1,"Ballon");
Item vI4=new Item(1,"gourde");
Item vI5=new Item(1,"serviette");

vAccueil.setItem(vI3);
vHall.setItem(vI3);
vTribuneB.setItem(vI);
vTerrain.setItem(vI2);
vTerrain.setItem(vI4);

vVestiaireD.setItem(vI4);
vVestiaireD.setItem(vI2);
vVestiaireD.setItem(vI5);
```

Exercice 7.23:

Pour le bouton back , j'ai d'abord ajouté un atribut room précédente , initialisé a null au départ. Puis dans la procédure back() , on regarde si la room précédente est null (on fait rien dans ce cas) , sinon on fait les actions que l'on faisait dans goRoom pour changer de salle en allant dans la salle précédente.

```
private Room aRoomPrecedente;
```

```
this.aRoomPrecedente=null;
```

```
public void back(){
    if(this.aRoomPrecedente==null){
        this.aGui.println("pas de Room Precedente");
    }
    else{

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

Exercice 7.26:

On déclare simplement une pile de Room comme ceci :

```
//initialisation de la pile
this.aRoomPrecedente=new Stack<Room>();
```

Dans la procédure goRoom , on ajoute la room actuelle a la pile des Room précédente :

```
else {
    this.aRoomPrecedente.push(this.aCurrentRoom);
    this.aCurrentRoom = vNextRoom;
```

Enfin dans la procédure back() on vérifie si la pile est vide avec la procédure empty (cas de départ ou on arrive dans une salle sans avec de salle précédente), ensuite si ce n'est pas le cas , on effectue le back en récupérant la salle précédente comme étant l'élément en haut de la pile , avec la procédure peek(), ensuite on le

supprime de la pile avant d'effectuer le changement. Ainsi on peut retourner à chaque fois le dernière élément de la pile et on peut donc enchaîner les back jusqu'au départ du jeu .

```
public void back(){
    if(this.aRoomPrecedente.empty()){
        this.aGui.println("pas de Room Precedente");
    }

    else{

        this.aCurrentRoom = this.aRoomPrecedente.peek(); //renvoie le de
        this.aRoomPrecedente.pop();
        this.aGui.println( this.aCurrentRoom.getLongDescription() );
        if ( this.aCurrentRoom.getImageName() != null )
            this.aGui.showImage( this.aCurrentRoom.getImageName() );
    }
}
```

A retenir sur les piles :

Stack : C'est une classe qui va permettre « d'empiler » un type d'objet préciser.

Peek() :méthode de la classe stack qui retourne l'objet en haut du stack, soit le dernier ajouté

Push() : méthode de la classe stack qui permet de mettre un élément en haut de la pile.

Pop() : méthode de la classe stack qui supprime l'élément en haut de la pile.

Empty() : méthode de la classe stack qui retourne un booleen :true si la pile est vide et false sinon

Exercice 7.28:

```
private void test(final String pS) throws Exception{
    Scanner vScan= new Scanner(new File(pS));
    while(vScan.hasNextLine()){
        String vLine=vScan.nextLine();
        this.interpretCommand(vLine);
    }
}

private void testF(final String pS)
{
    try{
        this.test(pS);
       }
    catch(Exception pE){
        this.aGui.println("Aucun fichier de ce nom");
    }
}
```

```
    else if ( vCommandWord.equals( "test" ) )
        this.testF(vCommand.getSecondWord()+".txt");
}
```

Cela nous permet de faire un test de toute nos commandes marqué dans le fichier sans avoir a toute les taper a la main.

Exercice 7.29:

Pour la classe Player auquel on a donné les attributs : un nom , un poids , un poids max et on a déplacé quelque élément comme la CurrentRoom , la pile des RoomPrecedente qui était dans le gameEngine et que l'on va maintenant gérer dans la classe Player :

```java
public class Player
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private String aNom;
    private int aPoidsPlayer;
    private int aPmax;
    private Room aCurrentRoom;
    private Stack<Room> aRoomPrecedente;
    //private Item aItem;
    private ItemList aItems;
    //private ItemList aItemList;
```

Ensuite après avoir déplacer toute les prodédure qui allaient avec ces attributs, dans game engine on ajoute un attribut aPlayer (aP) et on adapte le code dans toute la classe. A noté que quelque méthode change de fonctionnement et sont géré en 2 fois , comme par exemple la méthode goRoom est séparé en 2 , une partie dans gameEngine qui gère la partie graphique et cette partie fait appel à l'autre partie dans Player qui gère les conditions pour que la méthode s'applique correctement.

Exemple d'adaptation dans gameEngine avec le this.aP… :

```java
this.printLocationInfo();
if(this.aP.getCurrentRoom().getImageName()!=null)
    this.aGui.showImage(this.aP.getCurrentRoom().getImageName() );
```

Exercice 7.30/31:

```java
public void take(final Command pCommand ){
    if ( ! pCommand.hasSecondWord() ) {
        // if there is no second word, we don't know where to go...
        this.aGui.println( "take what?" );
        return;
    }
    String vItem = pCommand.getSecondWord();
    if (aP.getCurrentRoom().getItem().estPresent(vItem)){

        aP.getItems().addItem(aP.getCurrentRoom().getItem().getItem(vItem)); //ajoute de l'élément a la liste
        aP.getCurrentRoom().getItem().removeItem(vItem); //supprision de l'item de la salle
        this.aGui.println( "take récupéré?" );
    }
}

public void drop(final Command pCommand ){
    if ( ! pCommand.hasSecondWord() ) {
        // if there is no second word, we don't know where to go...
        this.aGui.println( "drop what?" );
        return;
    }
    String vItem = pCommand.getSecondWord();
    if (aP.getItems().getItem(vItem).getDescription().equals(vItem)){

        aP.getCurrentRoom().getItem().addItem(aP.getItems().getItem(vItem)); //ajout dans de l'item dans la salle
        aP.getItems().removeItem(vItem); //suppresion de l'item de la liste
        this.aGui.println( "drop fais" );
    }
}
```

Pour cette question , on a mit en place un attribut aItems qui est une hashmap d'item.Et ensuite on teste si il y a un second mot après le mot take ou drop. Si il y en a un un :

Pour le take : on vérifie si l'item est déjà présent dans la salle, ensuite si il est bien la , on met l'item dans la hashmap et ensuite on le supprime de la salle en le supprimant de la hashmap qui contient les items de la salle.

Pour le drop : on vérifie si l'item taper en second mot est bien dans la hashmap qui contient les objet du joueur , si c'est le cas , on le met dans la salle et ensuite on le supprime de la hashmap du joueur.

Enfin on ajoute les 2 commandes à la méthode interpretCommand() :

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

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

Exercice 7.31.1:

Pour cette partie on implémente une nouvelle classe avec comme seul attribut une hashmap d'item , celle si sera utilisé pour les item que le joueur possède sur lui et pour les items qu'une salle contient . Les méthodes sur cette hashmap sont donc maintenant toute stocké dans cette classe.

```
public class ItemList
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private HashMap<String,Item> aItemList;

    /**
     * Constructeur d'objets de classe ItemList
     */
    public ItemList()
    {
        // initialisation des variables d'instance
        this.aItemList = new HashMap<String, Item>();
    }
```

Exemple de méthode :

```java
/**
 * Procedure ajout item dans hashmap
 * @param pI un Item
 */
public void addItem(final Item pI){

    this.aItemList.put(pI.getDescription(),pI);
} // addItem()

/**
 * Procedure retrait item dans hashmap
 * @param pN est le Nom de l'item de type string
 */
public void removeItem(final String pN){
    this.aItemList.remove(pN);
} // removeItem()
```

Exercice 7.33:

Pour l'inventaire on fait une méthode qui renvoie une string dans la classe Player, on vérifie si la liste d'objet que le joueur a est vide dans un premier temps , sinon on renvoie la liste des objets que contient l'inventaire avec la méthode getItemString() déjà implémenté dans la classe ItemList.

```java
/**
 * Methode qui affiche l'inventaire du joueur
 * @return l'inventaire sous forme de string
 */
public String inventaire(){
    if(this.aItems.estVide()){
        return "votre inventaire est vide";
    }
    else{ return "Vous avez: " +this.aItems.getItemString();}
} // inventaire()
```

Exercice 7.32:

Pour cet exercice , on crée d'abord une méthode dans la classe Player qui renvoie un booléen qui vérifie si un Item est trop lourd par rapport au joueur + le poids de l'item , le poids max étant un attribut de Player ici.

```
/**
 * Methode qui test si le joueur peut prendre un Item par rapport au poids
 * @param pPI poids d'un item
 * @return true si le poids est valade sinon false
 */
public boolean PoidsValable(final double pPI){
    return this.aPoidsPlayer + pPI <= this.aPoidsMax;
} // PoidsValable()
```

Puis dans la méthode take de la classe GameEngine on ajoute cette condition qui vérifie que le poids de l'objet qu'on veut ramasser est valable.

```
if(!this.aP.PoidsValable(aP.getCurrentRoom().getItem().getItem(vItem).getPoids())){
    this.aGui.println("Objet est trop lourd \n");
    return;
}
```

Exercice 7.34:

Pour cet exercice , en amont je crée un Item cookie que je place dans une de mes pièce du jeu. Ensuite on modifie la méthode eat dans la classe GameEngine, on vérifie d'abord si la commande contient 2ème mot , si c'est le cas alors on vérifie si c'est bien le cookie que l'utilisateur veut manger (et si le cookie est bien présent dans la pièce actuelle) , si c'est le cas alors on mange et on modifie le poids maximum que le joueur peut prendre. Sinon on affiche un message d'erreur . (ajoute d'un modificateur dans Player pour l'attribut aPoidsMax)

```
public void eat(final Command pCommand){
    if ( ! pCommand.hasSecondWord() ) {
        // if there is no second word, we don't know where to go...
        this.aGui.println( "eat what?" );
        return;
    }
    String vItem = pCommand.getSecondWord();
    if (aP.getCurrentRoom().getItem().estPresent(vItem) && vItem.equals("cookie")){

        aP.getCurrentRoom().getItem().removeItem(vItem); //suppresion de l'item de la salle
        aP.setPoidsMax(200);
        this.aGui.println( "tu viens de manger un cookie magique" );

    }

    else{
        this.aGui.println("tu ne peut pas manger cet objet");
    }
}
```

Exercice 7.42:

Pour mettre en place un système de temps , on ajoute un attribut compteur a la classe GameEngine que l'on initialise à 0 . Ensuite dans la fonction go Room, on incrémente le compteur de 1 a chaque commande valide de l'utilisateur, si la valeur du compteur est supérieur a 25 , alors on arrete le jeu et le joueur a donc perdu .

```
public class GameEngine
{
    private Player aP;
    // private Stack<Room> aRoomPrecedente;
    private Parser aParser;
    private UserInterface aGui;
    private int aCompteur; //compteur de commande

    public GameEngine(){
        createRooms();
        this.aParser=new Parser();
        this.aCompteur=0;
    }
```

```
else{
    this.aCompteur+=1;
    if(this.aCompteur==25){
        this.aGui.println("vous avez perdu , le match a commencé");
        this.endGame(); //mettre fin au jeu
    }
```

à noter ici que le compteur fonctionne seulement pour les commandes de type go + une direction et non pas pour le bouton ou bien pour un bouton back par exemple , on considérera que cela représente une aide pour le joueur .

Changement de scénario : Par manque de temps j'ai du changer le scénario du jeu , le scenario est donc simplifier , il faut toujours retrouver un joueur coincé dans la salle des trophée , pour gagné il suffit juste de trouver une clé qui permet d'ouvrir cette salle et donc de retrouver le joueur.

Guide du jeu :

aller dans une salle = go + direction (north , east , west ou south)
ramasser (ou lâcher) un objet= take(ou drop) + nom de l'objet
prendre les information de salle = look

manger = eat + l'objet
aide = help
quitter = quit
revenir un pas en arrière= back
afficher l'inventaire du joueur= inventaire

Conclusion :

Ce projet m'aura permis d'approfondir mes connaissances en programmation orienté objet en java en plus des heures de travaux pratiques . J'ai bien aimé le fait d'être guider tout au long du projet (pouvoir trouver de l'aide sur le site en plus d'avoir les intervenants disponibles) , même si parfois je trouvais cela dommage d'écrire du code pour le remplacer par la suite mais avec du recul cela nous permet de voir du code sous plusieurs aspect et parfois de comprendre pourquoi une manière de faire est meilleure qu'une autre , j'aurai aimé avoir plus de temps afin de mieux finir les détails du jeu. Je pense que mon organisation de travail aurait pu mieux être gérer afin de me permettre d'avoir le temps nécessaire à la fin. Venant d'une licence Mathématique-Informatique ce projet me donne une autre manière de mener un projet , en effet je n'avais jamais programmé en langage orienté objet , donc j'ai trouvé le projet pertinent car c'était une nouvelle manière de faire contrairement à mes anciens projets.