Rapport :

No'mad,
8 months for a lifetime memorie...

IPO-1104 2018/2019 groupe 2A

I/ Description du jeu

a) Informations générales

Auteur : Denoëla Guennoc

Phrase thème : Autour du monde, un bagpacker passe de pays en pays jusqu'à revenir à son point de départ.

Résumé du jeu : Un jeune étudiant d'une vingtaine d'années part faire un tour du monde pendant son année de césure avec un simple sac sur le dos. En chemin il va devoir faire preuve de beaucoup de qualités pour surmonter les défis que lui prépare ce voyage. Entre budget réduit et rencontres, il devra négocier pour monter à bord de moyens de transport divers et variés afin de retrouver la France et la suite de ses études. Pour cela il va selon les lieux devoir trouver des objets à échanger contre un moyen de transport, répondre à des énigmes, apprendre de nouvelles compétences (langue, navigation...).

b) Détails du scénario

Scénario détaillé :

Août 2019, cette fois-ci ça y est, après une dernière soirée entre amis, Danaël vérifie qu'il ne lui manque rien. Son sac est prêt, son réveil est programmé, ses billets d'avions et son passeport tout neuf sont sur la table dans l'entrée, tous les ingrédients d'une grande aventure sont réunis dans ce petit appartement qu'il s'apprête à quitter pour une année de césure autour du monde. Le lendemain matin sac sur le dos, direction le RER puis l'aéroport et c'est parti ! En 8 mois il a prévu de parcourir autant de pays que possible avec le petit budget que représente ses économies d'étudiant. Danaël n'est pas le genre à se prendre la tête, il n'a pour l'instant qu'une vague idée des lieux qu'ils souhaitent visiter. Hormis quelques point sur une carte par lesquels il espère passer, le reste de son voyage s'écrira au fur et à mesure de ses rencontres et de ses envies. Néanmoins, avec un budget comme le sien, il le sait il faudra se montrer ruser, trouver des solutions économiques là où on ne les attend pas toujours et surtout faire avec ce qu'on lui offre sans être trop regardant. Attention cependant à ne pas tomber dans des pièges qui pourraient mettre en péril son projet. Quoi qu'il en soit, dans 8 mois il devra être rentré en France pour pouvoir procédé à sa réinscription dans l'école pour l'année à suivre...

Le héro : Jeune étudiant français de 20 ans, Danaël est un aventurier débrouillard à qui le monde ne fait pas peur. Il passe son année de césure à voyager autour du monde avec un budget pour ses trajets extrêmement réduit.

Dans son sac : Une carte, un dictionnaire électronique, son budget de départ pour les trajets, son passeport et une bonne dose d'huile de coude.

Les lieux traversés : Au moins 13 lieux différents modélisés par des salles (peut-être plus en fonction de mon avancée dans la création du jeu).

Pour le moment chaque lieu est un pays :

- France (point de départ et d'arrivée du jeu)
- Nouvelle Zélande
- Australie
- Indonésie
- Inde
- Jordanie
- Kenya
- Afrique du Sud
- Costa Rica
- Canada
- Groenland
- Islande
- Écosse

Les liens entre les pays (salles) seront modélisés par des voies de transport (comme des portes). Par exemple, la voie aérienne, la voie maritime, la voie terrestre. Ou de manière plus détaillée par un mode de transport (exemple : avion, voilier, cargo, voiture, dos d'animal, vélo, pieds...)

Modes de transports possiblement emprunté : (première liste d'idées à peaufiner par la suite)

Air : avion de compagnies aériennes, avions privés, dirigeables, montgolfière, parapente, parachute, planeur, deltaplane, hydravion

Mer : ferry, voilier, rame, rafting, canoë

Terre : pieds, voiture, animal (dromadaire, cheval, âne), ski, traîneau, quad, moto, touk-touk, car, bus, train, vélo

Stéréotype des personnages possiblement rencontrés : d'autres voyageurs (échangent leurs bons plans), le nomade (le héro peut voyager avec lui), l'agriculteur (possible prêt d'un moyen de locomotion véhicule ou animal), les locaux du pays, le bandit (mets des bâtons dans les roues du héro). (Sûrement d'autres à venir)

c) Issues et péripéties possibles

Situation gagnante : parvient à retourner en France avant la reprise des cours (en étant passé par l'ensemble des pays obligatoires sur sa route)

Situation perdante : est forcé de faire appel au consulat français pour rentrer en France car il n'a plus les moyens de continuer son voyage, ne trouve pas de moyens de transports ou s'est associé aux mauvaises personnes

Possibles défis à surmonter : trouver un objet pour quelqu'un afin qu'il conduise le héro dans un nouveau lieu, effectuer une tache pour obtenir de l'argent ou un moyen de transport en échange, répondre à une énigme pour avoir accès à un nouveau lieu, réaliser des échanges avec des personnages contre des objets récoltés auparavant, apprendre quelques mots dans la langue du personnage rencontré pour lui demander de l'aide. (liste à compléter et préciser)

II/ Exercices Zuul

7.5

Cette exercice vise à optimiser notre code et à rendre un travail de maintenance plus simple. En effet, on a constaté que les procédures goRoom() et printWelcome() possédaient un certains nombre de lignes de code strictement identiques :

```
System.out.println("You are in the " + this.aCurrentRoom.getDescription());
// Liste des sorties disponibles
System.out.print("Possible exits : ");
if (this.aCurrentRoom.getExit("north") != null)
{
    System.out.print("North ");
}
if (this.aCurrentRoom.getExit("east") != null)
{
    System.out.print("East ");
}
if (this.aCurrentRoom.getExit("south") != null)
{
    System.out.print("South ");
}
if (this.aCurrentRoom.getExit("west") != null)
{
    System.out.print("West ");
}
System.out.println();
```

On a donc supprimé cette partie de code des deux procédures précédentes et créé la procédure printLocationInfo() telle que ci-dessous, qui sera appelée chaque fois que nécessaire comme dans goRoom() et printWelcome(), plutôt que de dupliquer son code.

```
private void printLocationInfo()
{
    System.out.println("You are in the " + this.aCurrentRoom.getDescription());
    // Liste des sorties disponibles
    System.out.print("Possible exits : ");
    if (this.aCurrentRoom.getExit("north") != null)
    {
        System.out.print("North ");
    }
    if (this.aCurrentRoom.getExit("east") != null)
    {
        System.out.print("East ");
    }
    if (this.aCurrentRoom.getExit("south") != null)
    {
        System.out.print("South ");
    }
    if (this.aCurrentRoom.getExit("west") != null)
    {
        System.out.print("West ");
    }
    System.out.println();
}
private void printWelcome ()
{
    System.out.println("Welcome to the World of Zuul!");
    System.out.println("World of Zuul is a new, incredibly boring adventure game.");
    System.out.println("Type 'help' if you need help.");
    System.out.println();
    this.printLocationInfo();
}
```

7.6

Dans cet exercice, on travail l'encapsulation des données. Pour cela, on passe tous les attributs de la classe Room, les sorties, en private.

Néanmoins, puisqu'il faudra quand-même que l'on puisse accéder à ces sorties, on créé également un accesseur pour ces attributs. Cet accesseur retourne la valeur de la sortie souhaitée.

```
public class Room
{
    private String aDescription;
    private Room aNorthExit;
    private Room aEastExit;
    private Room aSouthExit;
    private Room aWestExit;

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

7.7

Dans la classe Game, la procédure printLocationInfo() doit accéder aux attributs de la classe Room. Cela été possible directement tant que ces attributs étaient publics, cependant, il faut maintenant passer par l'accesseur de ces attributs. On effectue donc les modifications suivantes dans la classe Game :

```
private void printLocationInfo()
{
    System.out.println("You are in the " + this.aCurrentRoom.getDescription());
    // Liste des sorties disponibles
    System.out.print("Possible exits : ");
    if (this.aCurrentRoom.getExit("north") != null)
    {
        System.out.print("North ");
    }
    if (this.aCurrentRoom.getExit("east") != null)
    {
        System.out.print("East ");
    }
    if (this.aCurrentRoom.getExit("south") != null)
    {
        System.out.print("South ");
    }
    if (this.aCurrentRoom.getExit("west") != null)
    {
        System.out.print("West ");
    }
    System.out.println();
}
```

On crée également une méthode getExitString() dans la classe Room.

7.8

On s'intéresse ici aux HashMap afin de simplifier considérablement la façon de créer des sorties dans nos pièces. Pour pouvoir utiliser ce nouvel outil, il faut tout d'abord aller le chercher dans la base de données Java en ajoutant avant la classe Room : `import java.util.HashMap;`

Maintenant que l'on peut utiliser les HashMap, on change les attribut de Room et on les remplace par une HashMap. Cela nous permettra de créer de nouvelles sorties sans avoir à les créer à trop d'endroits différents dans notre code. On modifie également en conséquence le constructeur naturel de Room.

```
public class Room
{
    private String aDescription;
    private HashMap<String, Room> exits;

    public Room (final String pDescription)
    {
        this.aDescription = pDescription;
        exits = new HashMap<String, Room>();
    }
```

Enfin, pour créer plus facilement de nouvelles sorties et initialiser les sorties d'une salle sans avoir à apporter des modifications à toutes les autres salles à chaque nouveauté, on corrige le modificateur setExit() de la classe Room et en conséquence la procédure createRoom() de la classe Game.

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

setExit() devient :

Cela permettra de ne plus avoir à lister les valeurs de toutes les sorties de chaque salle comme précédemment.

Ainsi, createRoom() que l'on connaissait comme ceci :

```
private void createRooms ()
{
    // Déclaration des différents lieux
    Room vOutside = new Room("main entrance");
    Room vTheatre = new Room("lecture theatre");
    Room vPub = new Room("campus pub");
    Room vLab = new Room("computing lab");
    Room vOffice = new Room("computing admin office");
    Room vPremier = new Room("first floor");

    // Positionnement des sorties
    // Ordre : Nord, Est, Sud, Ouest, Haut, Bas
    vOutside.setExits(null, vTheatre, vLab, vPub, null, null);
    vTheatre.setExits(null, null, null, vOutside, null, null);
    vPub.setExits(null, vOutside, null, null, vPremier, null);
    vLab.setExits(vOutside, vOffice, null, null, null, null);
    vOffice.setExits(null, null, null, vLab, null, null);
    vPremier.setExits(null, null, null, null, null, vPub);

    // Initialisation du lieu courant
    this.aCurrentRoom = vOutside;
}
```

devient de façon plus simple :

```
private void createRooms ()
{
    // Déclaration des différents lieux
    Room vOutside = new Room("main entrance");
    Room vTheatre = new Room("lecture theatre");
    Room vPub = new Room("campus pub");
    Room vLab = new Room("computing lab");
    Room vOffice = new Room("computing admin office");
    Room vPremier = new Room("first floor");

    // Positionnement des sorties
    vOutside.setExit ("east", vTheatre);
    vOutside.setExit ("south", vLab);
    vOutside.setExit ("west", vPub);
    vTheatre.setExit ("west", vOutside);
    vPub.setExit ("east", vOutside);
    vPub.setExit ("up", vPremier);
    vLab.setExit ("north", vOutside);
    vLab.setExit ("east", vOffice);
    vOffice.setExit ("west", vLab);
    vPremier.setExit ("down", vPub);

    // Initialisation du lieu courant
    this.aCurrentRoom = vOutside;
```

7.8.1

Il s'agit ici d'ajouter aux pièce une troisième dimension en rendant possible un changement d'étage (up et down).

(cf : code de l'exercice 7.8)

7.9

Dans cet exercice, on utilisera la méthode keySet(), qui nous permet d'avoir accès aux clés des valeurs connues de notre HashMap.

La méthode getExitString() est modifiée et devient alors :

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

7.11

Il s'agit ici de pouvoir apporter à chaque salle une description plus longue que la simple liste de ses sorties. Pour cela, on crée une méthode getLongDescription() telle que :

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

On modifie également l'instruction d'impression dans la procédure printLocationInfo() de la classe Game afin que ce soit dorénavant la description longue de la pièce qui soit retournée.

7.14

On souhaite ajouter ici une nouvelle commande, « look », qui permettrait à l'utilisateur d'avoir accès à la description longue de la salle dans laquelle il se trouve. Pour cela, il faut d'abord que « look » soit reconnu par notre programme comme une commande valide. Il faut donc l'ajouter à la liste des commande de la classe CommandWords :

```
public class CommandWords
{
    // tableau constant qui contient tous les mots de commande valides
    private static final String[] sValidCommands = {
        "go", "quit", "help", "look"
    };
```

Une fois que la commande est valide, il faut quelle se rapporte à une méthode et que cette méthode soit exécutée lorsque l'utilisateur entre cette commande. On effectue donc les modifications suivantes :

```
private void look ()
{
    System.out.println(aCurrentRoom.getLongDescription());
}
private boolean processCommand (final Command pCom)
{
    boolean vB = false;

    if (pCom.getCommandWord().equals("go"))
    {
        goRoom(pCom);
    }
    else if (pCom.getCommandWord().equals("quit"))
    {
        vB = quit(pCom);
    }
    else if (pCom.getCommandWord().equals("help"))
    {
        printHelp();
    }
    else if (pCom.getCommandWord().equals("look"))
    {
        look();
    }
```

7.15

Dorénavant, il suffit de reproduire le même processus que dans l'exercice précédent (7.14) pour ajouter une nouvelle commande à notre jeu.

7.16

Dans les exercices précédents, de nouvelles commandes ont été crée. Cependant, lorsque l'on fait appelle à la commande help, elles n'apparaissent pas dans la liste des commandes possibles. Pour régler ce problème, on crée dans la classe CommandWords, qui gère déjà l'ensemble des commandes possible une nouvelle méthode chargée de retourner les commandes valides.

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

Par la suite, il faut faire apparaître cette nouvelle méthode dans la classe Game. Cependant, on souhaite créer le moins de liens possibles entre les classes par soucis d'optimisation et de maintenance. On choisi donc de lier cette méthode de CommandWords avec une nouvelle commande de Parser qui est lui-même déjà lié à la classe Game. Ainsi, on effectue les modifications suivantes :

Dans la classe Parser :

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

Dans la classe Game :

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

7.18

Afin de continuer l'optimisation de notre code, il est important d'avoir un minimum de redondances. Ainsi, pour faciliter la maintenance du jeu, on change la méthode showAll() par une méthode getCommandList() qui sera appelée chaque fois que l'on souhaitera avoir une liste des commandes valides dans le jeu.

CommandWords.java

```
@@ -20,12 +20,13 @@
20          /**
21           * Print all valid commands to System.out.
22           */
-       public void showAll ()
+       public String getCommandList ()
24          {
+           String ComList = "";
26          for (String pCommand : sValidCommands) {
-               System.out.print(pCommand + " ");
+               ComList += pCommand + " ";
28          }
-           System.out.println();
+           return ComList;
30          }
```

7.18.5

Les Rooms, jusqu'ici accessibles uniquement dans createRoom(), deviennent accessibles partout grâce à la création d'une HashMap dans laquelle elles sont répertoriées au sein de la classe gameEngine.

```
private HashMap <String, Room> aRooms;
```

Dans le constructeur de gameEngine :

```
/**
 * Constructeur naturel de la classe.
,6 +18,7 @@ public Game ()
{
    this.createRooms();
    aParser = new Parser();
    aRooms = new HashMap<String, Room>();
}
```

Dans createRoom() :

```
aRooms.put("Theatre", vTheatre);
aRooms.put("Outside", vOutside);
aRooms.put("Pub", vPub);
aRooms.put("Lab", vLab);
aRooms.put("Office", vOffice);
aRooms.put("Premier", vPremier);
```

7.18.6

Dans cet exercice le code est entièrement réorganisé et modifié pour des raisons d'optimisation. La majeure partie de la classe Game passe dans la nouvelle classe GameEngine. Une classe UserInterface est créée et les classes Room et Parser sont également modifiées.

7.18.8

Cet exercice consiste en l'ajout de boutons de commande. Le but est de pouvoir réaliser une commande sans avoir forcément besoin de la taper. Pour cela on modifie la classe UserInterface telle que :

16 UserInterface.java

```
@@ -19,6 +19,7 @@
        private JTextField aEntryField;
        private JTextArea  aLog;
        private JLabel     aImage;
+       private JButton    aHelpButton;

        /**
         * Construct a UserInterface. As a parameter, a Game Engine
@@ -82,6 +83,8 @@ private void createGUI()
        {
            this.aMyFrame = new JFrame( "Zork" );
            this.aEntryField = new JTextField( 34 );
+
+           this.aHelpButton = new JButton ("help");

            this.aLog = new JTextArea();
            this.aLog.setEditable( false );
@@ -96,6 +99,7 @@ private void createGUI()
            vPanel.add( this.aImage, BorderLayout.NORTH );
            vPanel.add( vListScroller, BorderLayout.CENTER );
            vPanel.add( this.aEntryField, BorderLayout.SOUTH );
+           vPanel.add( this.aHelpButton, BorderLayout.EAST );
```

```
        this.aHelpButton.addActionListener( this );


    public void actionPerformed( final ActionEvent pE )
    {
        // Si l'utilisateur appuie sur le bouton
        if ( pE.getActionCommand().equals("help"))
        if ( pE.getSource() == this.aHelpButton)
            this.aEngine.processCommand("help");

        else if ( pE.getSource() == this.aConsulateButton)
            this.aEngine.processCommand("go consulate");
        // Sinon
        else this.processCommand();
    } // actionPerformed(.)
```

7.19.2

Ajout des images dans chaque room. Pour cela on cela on adapte le constructeur de la classe Room puis on ajoute l'emplacement de l'image souhaitée comme deuxième paramètre à la création des rooms. Puisqu'il devient nécessaire que chaque room possède une image, ont place des images temporaires aux rooms qui n'en ont pas.

Dans la classe Room :

4 Room.java

```
@@ -14,11 +14,11 @@
14 14         /**
15 15          * Constructor of the Room class.
16 16          */
17    -       public Room (final String pDescription)
   17 +       public Room (final String pDescription, final String pImageName)
18 18         {
19 19             this.aDescription = pDescription;
20 20             exits = new HashMap<String, Room>();
21    -           aImageName = "image";
   21 +           this.aImageName = pImageName;
22 22         }
```

Dans la classe GameEngine, méthode createRoom() :

```
Room vFr = new Room("France", "Images/france.jpg");
Room vNz = new Room("New Zealand", "Images/newzealand.jpg");
Room vAu = new Room("Australia", "Images/australia.jpg");
Room vIndo = new Room("Indonesia", "Images/indonesia.jpg");
Room vInde = new Room("India", "Images/inde.jpg");
```

7.20

Création de la classe Item contenant nom, description et poids de l'item créé. On modifie aussi la classe Room afin de pouvoir ajouter jusqu'à un item dans chaque room.

Dans la classe Item :

```
public class Item
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private String aItemName;
    private String aItemDescription;
    private int aItemWeight;
```

On y ajoute également des getters et setters pour chaque attribut.

Dans la classe Room :

33 Room.java

```
@@ -10,6 +10,7 @@
10 10      private String aDescription;
11 11      private HashMap<String, Room> exits;
12 12      private String aImageName;
   13 +    private Item aItem;
13 14
14 15      /**
15 16       * Constructor of the Room class.
@@ -67,15 +68,37 @@ public String getExitString()
67 68       */
68 69      public String getLongDescription ()
69 70      {
70    -         return "You are in " + aDescription + ".\n" + getExitString();
   71 +         return "You are in " + aDescription + ".\n" + getExitString() + ".\n" + getItemDescription();
71 72      }
```

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

/**
 * Used to add an item to the current room.
 */
public void addItem (final Item pItem)
{
     this.aItem = pItem;
}

/**
 * Returns the informations about the item located in the
 * current room : description, weight
 *
 * If there's no item in the room, returns "No item here."
 */
public String getItemDescription()
{
    if (this.aItem == null)
        return "No item here" + ".\n";
    else
        return "Item : " + this.aItem.getItemName() + ".\n";
}
```

7.21

On modifie getLongDescription() dans Room pour qu'elle affiche les items disponibles à chaque Room.

C'est bien la classe Room qui doit gérer l'affichage de la liste des items et non pas la classe Item directement. Elle accèdera à chaque fois à la classe Item pour trouver les informations souhaitée.

7.22

On modifie le mode de stockage des items dans Room pour pouvoir mettre plusieurs items dans chaque room. Pour cela on utilise une HashSet permettant de créer une liste des items de chaque room.

Room.java

```
@@ -10,7 +10,7 @@
10          private String aDescription;
11          private HashMap<String, Room> exits;
12          private String aImageName;
-           private Item aItem;
13  +       private HashSet<Item> aItemList;
14
15          /**
16           * Constructor of the Room class.
@@ -20,6 +20,7 @@ public Room (final String pDescription, final String pImageName)
20              this.aDescription = pDescription;
21              exits = new HashMap<String, Room>();
22              this.aImageName = pImageName;
23  +           this.aItemList = new HashSet<Item>();
24          }
            public void addItem (final Item pItem)
            {
                this.aItem = pItem;
                this.aItemList.add(pItem);
            }

            /**
             * Returns the informations about the item located in the
             * current room : name, description, weight
             * Returns the informations about the items located in the
             * current room, returns "Item : " + the list of all the
             * name of the disponibles items.
             *
             * If there's no item in the room, returns "No item here."
             */
            public String getItemDescription()
            {
                if (this.aItem == null)
                String vItemList = "Item : ";
                if (this.aItemList.isEmpty())
                    return "No item here" + ".\n";
                else
                    return "Item : " + this.aItem.getItemName() + ".\n";
                else
                    for(Item vItem : aItemList)
                    {
                        vItemList += vItem.getItemName() + " ";
                    }
                return vItemList + ".\n";
            }
```

7.23

Création d'une commande back. Utilisée sans second mot, la commande back permet à l'utilisateur de revenir dans la pièce précédent la pièce courante. Pour cela on crée un nouvel attribut dans lequel sera stocké à chaque changement de salle la pièce n-1, et une nouvelle méthode, la méthode back(). Il faut également ajouter « back » à la liste des commandes valides dans le jeu.

Dans la méthode back() :

```
this.aCurrentRoom = this.aPreviousRoom;
this.printLocationInfo();
```

7.26

Afin de pouvoir utiliser la commande back plusieurs fois d'affiler et donc remonter de room précédente en room précédente jusqu'à revenir au point de départ du jeu, il nous faut un nouveau moyen de stockage des rooms précédentes. Pour cela, on utilise une Stack. Il s'agit de créer une pile d'éléments (ici de rooms) où le dernier ajouté sera le premier accédé. On y stocke donc nos rooms au fur et à mesure qu'on les traverse. Quand l'utilisateur tape "back", on retira le dernier nom de la liste, la dernière room dans laquelle il se trouvait, qui devient alors room courante...

Dans la classe GameEngine :

```
    private Stack<Room> aPreviousRooms;
        aPreviousRoom = aCurrentRoom;
        aPreviousRooms.push(this.aCurrentRoom);
        aCurrentRoom = vNextRoom;
        this.printLocationInfo();
    }
@@ -393,12 +393,18 @@ private void eat()
}

/**
 *
 * Allows the user to go back to the previous room by only
 * entering "back".
 */
private void back()
{
    this.aCurrentRoom = this.aPreviousRoom;
    this.printLocationInfo();
    if (aPreviousRooms.empty())
        gui.println ("You can't go back any further");
    else
    {
        this.aCurrentRoom = aPreviousRooms.pop();
        this.printLocationInfo();
    }
}
```