A3P 2016/2017 G8

Quentin GARRIDO
E1 Groupe 8

2016-2017

Sommaire :

I)Présentation globale du projet:

Auteur:

Quentin Garrido, étudiant en première année à l'ESIEE PARIS
Site internet : http://www.esiee.fr/~garridoq/A3P/index.html

Phrase thème:

Dans une antre médiévale fantastique Luneth doit vaincre le Bahamut.

Scénario:

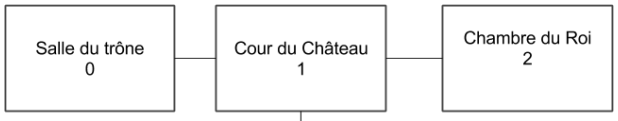
Luneth, chevalier Dragon de la lumière du village d'Ir, doit aller pourfendre Bahamut afin de récupérer son sang pour guérir le Roi du mal dont il souffre.

Bahamut se trouvant dans son antre en haut du Mont Amarth, Luneth devra trouver son antre et ensuite se frayer un chemin à l'intérieur afin de récupérer Ragnarok, la lance forgée un millénaire auparavant mais qui fut volée par le seigneur du mal Melkar et cachée avec le dragon lui même qui permettra à Luneth de vaincre le dragon.

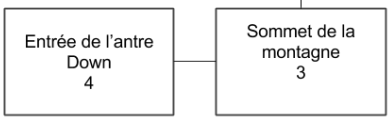
Mais le chemin sera semé d'embûches et d'énigme que Luneth devra résoudre afin d'accéder au Bahamut. Heureusement pour lui, grâce à sa formation donnée par Firio, et l'armure de chevalier dragon dans le meilleur mithril du royaume, il est prêt à parer à toute éventualité afin de triompher de sa quête d'une importance vitale pour le peuple.

Plan du jeu:

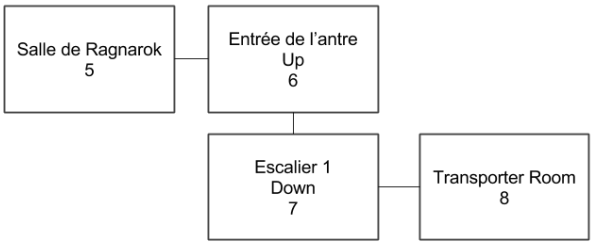
Château:



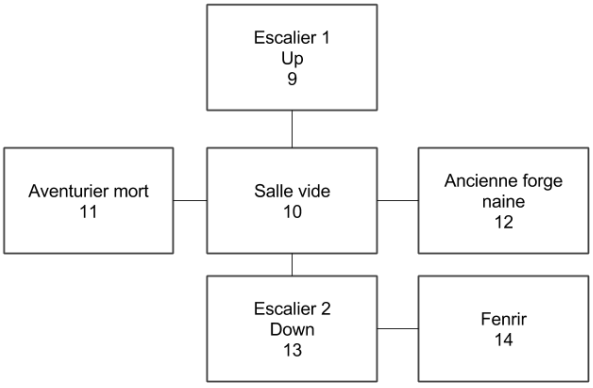
Montagne :



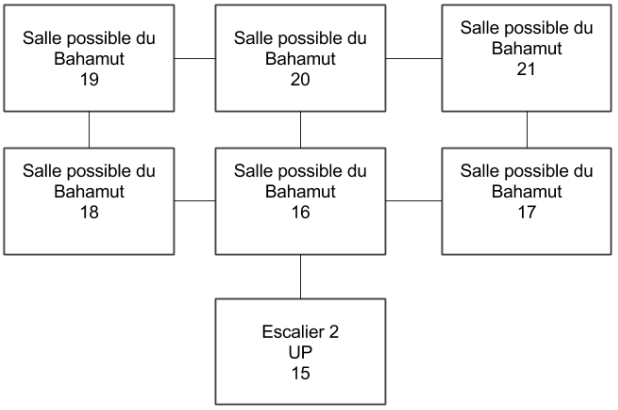
Etage 1:



Etage 2:

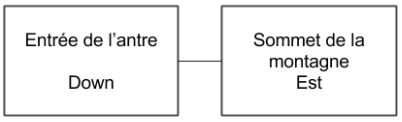


Etage 3:

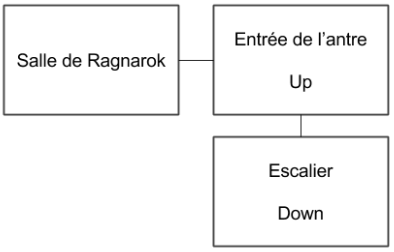


Plan Simplifié :

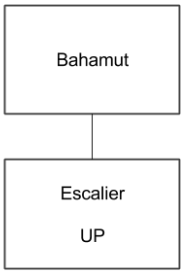
Montagne :



Etage 1:



Etage 2:



Détail des lieux, objets, personnages

Lieux: La salle du trône est le lieu de début du jeu
La chambre du Roi est le lieu de fin du jeu
La salle libellée Ragnarok contient Ragnarok
Les 6 salles Lieu possible du Bahamut sont les salles ou le Bahamut pourra apparaître aléatoirement
La salle Transporter Room téléportera aléatoirement le joueur.
Le reste des salles possède des noms assez explicites, ou avec des événement non définis

Objets: Ragnarok,Excalibur, Beamer,cotte de maille, lemba, épée rouillée, potion de vie, elixir mystérieux

Personnages: Messager du Roi dans la salle Trône
Roi dans la salle Chambre du Roi
Luneth (incarné par le joueur)
Aventurier (cadavre) dans la salle Aventurier

Fenrir dans la salle Fenrir
Bahamut
Gimli dans la salle Entrée de l’antre

Situations Gagnantes et perdantes:

Le joueur gagne si il termine le donjon en donnant le sang du bahamut au Roi dans un temps imparti.
Il perd si il meurt dans un combat, ou lors de tout autre évènement, ainsi que si il est trop lent.

II) Réponses aux exercices

7.0

Ma page internet a été créée à l'aide de Bootstrap et est accessible par tout le monde de part ses permissions et sa localisation dans mon dossier /public_html

Lien :http://www.esiee.fr/~garridoq/A3P/index.html

7.1.1

Le thème choisit est le suivant : *" Afin de sauver le Roi d'un terrible mal, Luneth doit aller chercher du sang du Bahamut au mont Amarth"*

7.2

Les TPs 3.1 et 3.2 ont été terminés et vérifiés par les professeurs en cours.

7.2.1

Scanner va nous permettre de réucpérer l'entrée *System.in* (ici le clavier) et de la stocker dans une variable afin de traiter cette entrée, ce qui est indispensable pour notre programme qui demandera à l'utilisateur d'entrer des commandes.

7.3

Voir scénario écrit précédemment

7.3.1

Fait car le rapport est écrit avec les informations demandées

7.3.2

Voir plan plus haut

7.4

Les classes ont été mises à la racine du projet en se débarrassant du package v1.

7.5

Afin d'éviter toute duplication de code et pour réduire le coupling nous avons implémenté la méthode printLocationInfo séparément pour la réutiliser dans d'autres méthodes la requérant.

7.6

Nous avons modifié la classe Room comme dans le livre, afin de créer une fonction getExit prenant en paramètre la direction afin de savoir quelle pièce est dans la direction donnée, afin de pouvoir mettre les attributs de sortie en private et non public comme avant.
Nous avons donc aussi modifié la classe Game afin d'utiliser cette méthode (accesseur) car les attributs sont désormais private.

7.7

Création d'une méthode dans la classe Room:

```
public String getExitString()
{
```

```
    String vExitString = new String("The exits are :");
    if(this.aNorthExit != null)
        vExitString += " north";
    if(this.aSouthExit != null)
        vExitString += " south";
    if(this.aEastExit != null)
        vExitString += " east";
    if(this.aWestExit != null)
        vExitString += " west";

    return vExitString;

}
```

Pour afficher les sorties on remplace dans game le bloc de conditions de sorties par un appel de la méthode getExitString (eg : this.aCurrentRoom.getExitString();)

7.7.1

Fait, voir la partie 2 du rapport jusqu'à cet endroit.

7.8

Modification dan la classe Room:

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

public void setExits(final Room pN,final Room pS,final Room pE,final Room pW)
{
    if(pN != null)
      this.aExits.put("north", pN);
    if(pS != null)
        this.aExits.put("south", pS);
    if(pE != null)
       this.aExits.put("east", pE);
    if(pW != null)
       this.aExits.put("west", pW);
}

public Room getExit(final String pD  )
{
```

```
    return this.aExits.get(pD);
}
```

7.8.1

Voir plan du jeu pour se rendre compte des différents étages, induisant des déplacement verticaux.

7.9

Modification de la méthode getExitString de Room en :

```
public String getExitString()
{
    String vReturnString = "Les sorties sont:";
    Set<String> vKeys = this.aExits.keySet();
    for(String vExit : vKeys)
    {
        vReturnString += " " + vExit;
    }

    return vReturnString;

}
```

keySet() retourne un objet Set contenant toutes les keys (clefs) de l'objet HashMap

7.10

Tout d'abord nous créons une chaîne de caractères *vReturnString* avec le texte ne dépendant pas des sorties.
Ensuite nous récupérons les clefs de *aExits* grâce à *keySet.*
Nous allons ensuite parcourir les éléments du Set retourné, correspondant au sorties de la pièce courante, puis les ajouter une par une à *vReturnString*, enfin nous retournerons *vReturnString*.

7.10.1

Voir l'onglet javadoc sur le site

7.10.2

Voir l'onglet javadoc sur le site

7.11

Ajout de la méthode suivante dans la classe Room:

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

Et utilisation de cette dernière dans la classe Game pour afficher directement toute la description, sans la créer dans la classe Game, pour réduire le coupling.

7.14

Ajout de *"look"* dans les commandes valides de CommandWords.

Création de la méthode suivante dans Game :

```
private void look()
{
    this.printLocation();
}
```

Et ajout de la condition d'exécution de la commande dans processCommands (code montré avec l'exercice suivant car code similaire).

7.15

Ajout de *"eat"* dans les commandes valides de CommandWords.

Création de la méthode suivante dans Game :

```
private void eat()
{
    System.out.println("Vous avez mangé et n'avez plus faim désormais.");
}
```

Et modification de processCommands (ici avec look et eat ajoutés) :

```
        private boolean processCommand(final Command pC){

        if(pC.getCommandWord() == null)
         {
        System.out.println("I don't understand");
        return false;

        }
        else if(pC.getCommandWord().equals("go"))
        {
        this.goRoom(pC);
        return false;
        }
        else if(pC.getCommandWord().equals("help"))
        {
        this.printHelp();
        return false;
        }
        else if(pC.getCommandWord().equals("look"))
        {
        this.look();
        return false;
        }
        else if(pC.getCommandWord().equals("eat"))
        {
        this.eat();
        return false;
        }
        else if(pC.getCommandWord().equals("quit"))
        return this.quit(pC);
        else
        return false;
}
```

7.16

Ajout de la méthode suivante dans la classe CommandWords pour afficher toutes les commandes lorsque l'on utilise la commande help :

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

 System.out.println();

}
```

Ajout dans la classe Parser d'une procédure appelant showAll() afin d'éviter de créer du coupling entre Game et CommandWords, ainsi on n'utilise que des liens déjà existants.

7.17

Dans cet exercice, nous avons modifié la methode showAll() de la classe CommandWords en :

```
public String getCommandList()
{
String vString = new String("");
for(String vCommand : sValidCommands)
vString += vCommand + " ";
return vString;

}
```

Nous avons juste dû changer après dans la classe PArser la manière d'afficher les commandes, en n'appelant plus directement showAll (renommée en getCommandList) mais en l'affichant.

7.18.1

Rien n'a eu à être modifié.

7.18.3

Les images sont toutes intégrées dans la version simplifiée du jeu et sont présentes sur la version intermédiaire.

7.18.4

Titre du jeu décidé : Bahamut's Lair.

7.18.5

Création d'une array list :

Dans le constructeur:

```
this.aRoomList = new ArrayList<Room>(7);
```

Dans createRooms():

```
        aRoomList.add(vSommetEst);
         aRoomList.add(vEntreeH);
         aRoomList.add(vEntreeB);
         aRoomList.add(vRagnarok);
         aRoomList.add(vEscalierH);
         aRoomList.add(vEscalierB);
         aRoomList.add(vBahamut);


         this.aCurrentRoom = aRoomList.get(0);
```

7.18.6

Tout a été mis en place en respectant la structure de zuul-with-images et aucune erreur n'est présente lors de la compilations.
Les principaux changements ont été l'emplacement de méthodes dans certaines classes

7.18.8

Créations de 10 boutons pour gérer les actions et déplacements:

```
this.aButtonHarakiri = new JButton("harakiri");
this.aButtonEat = new JButton("eat");
this.aButtonLook = new JButton("look");
this.aButtonAttack = new JButton("attack");

this.aButtonNorth = new JButton("go nord");
this.aButtonNorth.setPreferredSize( new Dimension(100,66));
this.aButtonSouth = new JButton("go sud");
this.aButtonEast = new JButton("go est");
this.aButtonWest = new JButton("go ouest");
this.aButtonUp = new JButton("go haut");
this.aButtonDown = new JButton("go bas");
```

Et d'une ArrayList les contenant

```
this.aButtonList = new ArrayList<JButton>(10);

this.aButtonList.add(this.aButtonHarakiri);
this.aButtonList.add(this.aButtonEat);
this.aButtonList.add(this.aButtonLook);
this.aButtonList.add(this.aButtonAttack);
this.aButtonList.add(this.aButtonNorth);
this.aButtonList.add(this.aButtonSouth);
this.aButtonList.add(this.aButtonEast);
this.aButtonList.add(this.aButtonWest);
this.aButtonList.add(this.aButtonUp);
this.aButtonList.add(this.aButtonDown);
```

Ainsi création de listener simplifiée par cette ArrayList, permettant d'avoir uniquement :

```
for(JButton vB : this.aButtonList)
    vB.addActionListener( this);
```

Et dans l'interprétation des commandes:

```
if(pE.getSource().getClass() == this.aButtonHarakiri.getClass())
    this.aEngine.interpretCommand(pE.getActionCommand());
```

```
else
        this.processCommand();
```

Ensuite nous avons empêché l'utilisation de boutons quand le joueur est mort , encore une fois simplifié par l'arrayList créée auparavant :

```
for(JButton vB : this.aButtonList)
        vB.setEnabled(pOnOff);
```

7.19.2

Toutes les images ont été mises dans un dossier images

7.20

Création de la classe Item, possédant comme attributs un poids et une description uniquement.

Pour l'affichage dans Room nous avons adapté l'affichage selon la présence ou non d'objet mis la possibilité d'afficher ou non les objets :

```
        public String getLongDescription()
        {
        if( this.aItem == null)
        return "Vous êtes " + this.aDescription + ".\n" + this.getExitString();
        else
        return "Vous etes " + this.aDescription + " \nL'objet présent dans la salle est: " +
this.aItem.getItemInformation() + ".\n" + this.getExitString();
        }
```

7.21

Ajout de la méthode suivante dans la classe Item:

```
public String getItemInformation()
```

```
        {
        return this.aDescription + " qui a un poids de " + this.aWeight;

        }
```

7.22

Nous avons choisi d'utiliser une Hashmap pour faire le travail de stockage de multiples objets, ainsi que adapté la méthode d'affichage :

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

[.....]

public String getItemString()
{
        String vReturnString = "Les objets sont :";
        Set<String> vKeys = this.aItems.keySet();
        for(String vItem : vKeys)
        vReturnString +=  "\n" + aItems.get(vItem).getItemInformation() ;

        return vReturnString;
}
```

Pour ajouter un objet dans la Room, la méthode suivant est placée dans la classe Room:

```
public void addItem(final String pN, final Item pI)
{
   this.aItems.put(pN, pI);
}
```

7.22.2

Voici les objets placés ainsi que leurs positions :

Objet:	Lieu:
torche	Sommet Est
Ragnarok	Ragnarok
Casque	Ragnarok

7.23

Pour implémenter la commande back, nous avons créé un attribut aPreviousRoom qui dès que l'on change de pièce prend pour valeur la salle où nous étions.
Ainsi lorsque nous appelons la commande back, nous nous retrouvons dans la salle précédente.

7.24

Tout marche comme il faut, si l'on tape back au début du jeu, on reste bien dans la salle de départ, si on met un second mot, on renvoie bien un message d'erreur.

7.25

Si on exécute back deux fois (ou plus) de suite, nous bouclons entre les deux salles, ce qui est problématique.

7.26

Nous utilisons un stack de Room ce qui nous donne la méthode back suivante :

```
private void back()
{
        this.aCurrentRoom = this.aRoomStack.pop();
        this.aGui.println(this.aCurrentRoom.getLongDescription());
        if(this.aCurrentRoom.getImageName() != null)
        this.aGui.showImage(this.aCurrentRoom.getImageName());
}
```

de plus nous ajoutons la ligne suivante dans goRoom quand nous changeons de pièce :

```
this.aRoomStack.push(this.aCurrentRoom);
```

7.26.1

Javadoc mises à jour et disponibles sur le site.

7.27

Nous pourrions par exemple ajouter une commande de test pour s'assurer que tout marche lors de modifications, au lieu de tester manuellement à chaque fois.

7.28

Nous pourrions utiliser des fichiers de test de la forme suivante :

```
go ouest
back
go nord
attack
go ouest
go bas
harakiri
[....]
```

7.28.1

Voici la ocmmande test, qui utilisera des fichiers comme vu au 7.28:

```
private void test(final String pNomFichier)
{
        Scanner vSc;
        try { // pour "essayer" les instructions suivantes
        vSc = new Scanner( new File( pNomFichier ) );
        while ( vSc.hasNextLine() ) {
                String vLigne = vSc.nextLine();
                this.interpretCommand(vLigne);
        } // while
        } // try
        catch ( final FileNotFoundException pFNFE ) {
        aGui.println("Aucun fichier correspondant n'a été trouvé");
   } // catch

}
```

Ainsi nous lisons ligne par ligne le fichier et exécutons les diverses commandes.

7.28.2

Les deux fichiers ont été créés mais pour plus de lisibilité ils ne seront pas mis ici.

7.29

Pour implémenter le Player, nous l'avons doté des attributs suivants :

```
private String aNom;
private Room aCurrentRoom;
private UserInterface aGui;
private Item aItem;
```

Nous l'avons donc fait communiquer avec le GUI existant grâce à la méthode suivante:

```
public void setGui(final UserInterface pUserInterface)
{
  this.aGui = pUserInterface;
}
```

De cette manière nous avons pu déplacer les méthodes eat,look etc dans Player.
Nous avons aussi mis dans Player la méthode suivante pour pouvoir déplacer le jour, le travail essentiel de goRoom restant dans GameEngine :

```
public void changeRoom(final Room pRoom)
{
  this.aCurrentRoom = pRoom;

}
```

7.30

La méthode take va regarder si l'objet se trouve dans la pièce et l'y enlever, puis l'ajouter à notre personnage.
La méthode drop va quand à elle mettre l'objet dans la pièce actuelle et ôter l'objet de notre inventaire.

7.31

Pour cet exercice, il suffit de changer aItem en Hashmap dans Player et Room :

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

7.31.1

Nous avons créé la classe ItemList contenant des méthodes similaires à celles de base du hashmap, cependant nous avons définit toString() et toStringDétail() de la manière suivante, selon l'affichage désiré :

```
    @Override
    public String toString()
    {
    if (this.aItemList.isEmpty())
    return "";
    else
    {
    String vString = "";
    Set<String> vKeys = this.aItemList.keySet();
    for(String vKey : vKeys)
    {
            vString += " " + vKey;
    }
    return vString;
    }

    }

    public String toStringDetail()
    {
    if (this.aItemList.isEmpty())
    return "";
    else
    {
    String vString = "";
    Set<String> vKeys = this.aItemList.keySet();
    for(String vKey : vKeys)
    {
            vString += "\n" + this.aItemList.get(vKey).getItemInformation();
    }
    return vString;
```

```
        }

        }
```

7.32

Ici, dès que nous ajoutons un objet à notre joueur, nous ajoutons son poids à notre poids total et SI le poids de l'objet que nous allons prendre nous fait dépasser la limite, nous ne le ramassons pas.
En jetant l'objet, nous enlevons son poids à notre poids courant.

7.33

Nous avons ajouté la commande Invetory qui parcourt l'inventaire en affichant les objets:

```
        public void inventory()
        {
        if (this.aInventory.isEmpty())
        this.aGui.println("Votre inventaire est vide");
        else
        {
        String vString = "Dans voter inventaire se trouvent : "+ this.aInventory.toString() + "
pour un poids total de " + this.aCurrentWeight;
        this.aGui.println(vString);
        }
        }
```

7.34

L'objet implémenté ici est le "Lemba", qui augmente le poids maximal du joueur de 5.
Si nous voulons manger cet objet, notre poids sera augmenté et il nous sera dit de combien.
Ensuite l'objet est enlevé de l'inventaire mais pas avec Drop, il est ici tout simplement 'détruit'.

7.34.1

Fichiers mis à jour.

7.34.2

Javadoc mises à jour.

7.35

Nous avons implémenté la même organisation, en utilisant les enums.

7.35.1

Malgré le fait que dans la version actuelle de java, le switch peut être effectué sur des String, nous avons utilisé les enums pour pouvoir remplacer le else if par un switch, qui est plus ”propre”.

le code est donc le suivant(ici il est troqué pour plus de clarté, la méthode à utiliser étant la même à chaque fois) :

```
switch(vCommandWord)
{
case UNKNOWN:
        this.aGui.println("Je ne comprends pas !");
        break;
case HELP:
        this.printHelp();
        break;
case GO:
        this.goRoom(pCommand);
        break;
case LOOK:
        this.look();
        break;
case EAT:
        if(!pCommand.hasSecondWord())
        this.aGui.println("Que vouelz vous manger ?");
        else
```

```
                this.aPlayer.eat(pCommand.getSecondWord());
                break;
        case HARAKIRI:
                this.harakiri();
                break;
        case ATTACK:
                this.attack();
                break;
        case BACK:
                if(pCommand.hasSecondWord())
                aGui.println("Reculer de quoi ?");
                else if( this.aPlayer.getCurrentRoom() == this.aRoomStack.peek())
                aGui.println("Vous n'avez pas encore bougé ! ");
                else
                this.back();
                break;
        case TEST:
                if(!pCommand.hasSecondWord())
                aGui.println("Tester quoi ?");
                else
                this.test(pCommand.getSecondWord());
                break;

       [...]


        default :
                this.aGui.println("ERROR");
                break;

        }
```

7.41.1

La nouvelle organisation des enums a été implémentée.

7.41.2

Les deux javadocs ont été régénérées.

7.42

Pour coller au scénario, si le joueur met trop longtemps, le Roi meurt et donc sa quête échoue.
Ici la limite est en nombre de salle, si le joueur se déplace de plus de N salles, le Roi meurt et la partie est finie.

Dès que le joueur se déplace le nombre de déplacement est incrémenté et ainsi quand il dépasse le maximum le jeu est terminé.

7.42.2

Je vais garder l'interface actuelle

7.43

Dans le cas de mon jeu, la trapdoor sera située entre les pièces 4 et 5. Ainsi le joueur pourra rentrer dans le donjon mais pas en ressortir (pour le moment).
Ainsi la pièce 5 n'aura pas de sortie vers la pièce 4 et dès que l'on passera de la salle 4 à 5 la pile de Room sera vidée pour que l'on ne puisse pas back.

7.44

Pour cet exercice, j'ai ajouté un attribut Porteur a chaque objet, afin de pouvoir récupérer par exemple la salle courante du joueur.
Ainsi la classe Beamer possède la méthode charge, qui va récupérer la pièce courante du joueur, et fire qui permettra de téléporter le joueur dans la Room chargée précédemment.
Evidemment pour éviter le back après avoir fire, la pile de Room est vidée.

7.45.1

Les fichiers de test ont été modifiées.

7.45.2

Les deux javadocs ont bien été générées.

7.46

Pour implémenter la Transporter Room, j'ai décidé d'override la méthode get Exit, ainsi à chaque fois que l'on demande à Transporter Room la sortie dans une direction, il nous sera retourné une pièce aléatoire.
La pièce aléatoire est choisie dans l'ArrayList de Room grâce à la fonction suivante :

```
int vIndex = (int)(Math.random() * (aRoomList.size()));
```

Toute la partie aléatoire, ainsi que le stockage de l'ArrayList est fait par la classe RoomRandomizer; classe qui n'a pour seul but que de trouver une salle aléatoire dans une ArrayList passée en paramètre.
Une Transporter Room aura bien sur un attribut Room Randomizer.

7.46.1

Ayant utilisé une ArrayList pour le stockage des Room, alea prendra comme second mot le numéro associé à la pièce et non son nom.
Ainsi la classe GameEngine a un nouvel attribut aTestMode pour savoir si alea peut être utilisée, et lors de la méthode test, cet attribut passe à true puis redevient false après coup.
Ainsi dans la méthode getExit de TransporterRoom vérifiera si oui ou non son attribut aAleaNb vaut null ou un nombre, et si il est non null, la sortie renvoyée sera celle associée à ce numéro.
Ainsi la commande "alea n" fixera aAleaNb a n et "alea" le fixera à nul.

7.46.2

Dans ce qui a été réalisé, beamer utilise déjà le concept d'héritage car hérite d'Item.
En revanche il pourrait être intéressant d'appliquer ce concept à TrapDoor, pour directement faire le nécessaire au niveau des sorties ou de la pile, sans devoir l'appliquer dans la classe Room directement.

7.46.3

Tous les commentaires ont été ajoutés/modifiéeS.

7.46.4

Les javadocs ont été régénérées.

7.47

La classe Command a été rendue abstraite et ne contient plus comme attribut que aSecondWord.
Ainsi nous avons pour chaque classe héritant de Command la structure suivante

```
public class NomCommand extends Command
{
        private CommandWords aCommandWords;

        public NomCommand(final CommandWords pCommandWords)
        {
           super(null);
           this.aCommandWords = pCommandWords;
        }

        @Override
        public void execute(final GameEngine pGameEngine)
        {

        [.....]

        }
}
```

Le stockage de la liste des commandes sera fait par une Hashmap<CommandWord,Command>, où à chaque création de nouvelles commandes nous l'y ajouterons, par exemple pour help :

```
this.aCommandes.put(CommandWord.HELP, new HelpCommand(this));
```

Ainsi une fois que tout cela est mis en place, dans GameEngine la méthode interpretCommand se simplifie grandement et va directement appeler sur la commande en paramètre sa méthode execute en ayant en paramètre le GameEngine courant.
Ainsi chaque commande est indépendante et à accès à tout ce qui lui est nécessaire.
Nous nous éliminons par la même occasion du soucis de se demander où va quelle commande car elles ont toute leur classe, ce qui simplifie la création de nouvelles commandes.

7.47.1

J’ai créé trois packages :
►pkg_core : qui contient GameEngine et UserInterface
►pkg_command : qui contient toutes les classes liées aux commandes
►pkg_mechanics : qui contient tout le reste des classes (sauf Game), donc Player,Room etc

7.47.2

Toutes les javadocs ont été générées.

7.48

Création de la classe NPC,

Chaque NPC possède une ArrayList de chaînes de caractères qui seront ses différentes phrases. La première phrase de l’ArrayList sera la phrase dite en premier puis le reste sera dit aléatoirement.
De plus les NPC auront une pièce courante, ainsi que chaque Room une HashMap de personnages.
Ils auront aussi un nom, et pour le système de combat une vie, des dégâts, une armure, le fait qu’il soit tuables ou non, et un objet qu’ils feront tomber à leur mort.
A l’entrée dans une pièce, tous les NPCs présents seront affichés.

7.49

Création de la classeMovingCharacter

Ils sont comme des NPC normaux mais avec un RoomRandomizer en attribut, afin de savoir dans quelles Room ils peuvent se déplacer.
J’ai ici opté pour un déplacement aléatoire, ainsi dès que le joueur se déplace de pièce, nous cherchons dans toutes les Room et tous les NPC de ses Rooms si il y a des MovingNPC, et si il y en a on les fera se déplacer aléatoirement dans leur liste de pièces potentielles.

7.49.2

Tout est implémenté comme dit au début du rapport.
Le personnage se déplaçant sera le Bahamut, entre les pièces 16 à 21

7.49.3

Les deux javadocs ont été générées.

7.53

Le constructeur de Game a été remplacé par la méthode statique main.
Ainsi en appelant cette méthode nous aurons le jeu qui se lancera , sans créer d'objets Game

7.54

Pour ce faire, il suffit de créer un .jar qui a pour classe principale Game.
Ainsi en lançant le .jar le jeu s'exécutera.

7.58

Déjà fait à l'exercice 7.54.

7.58.1

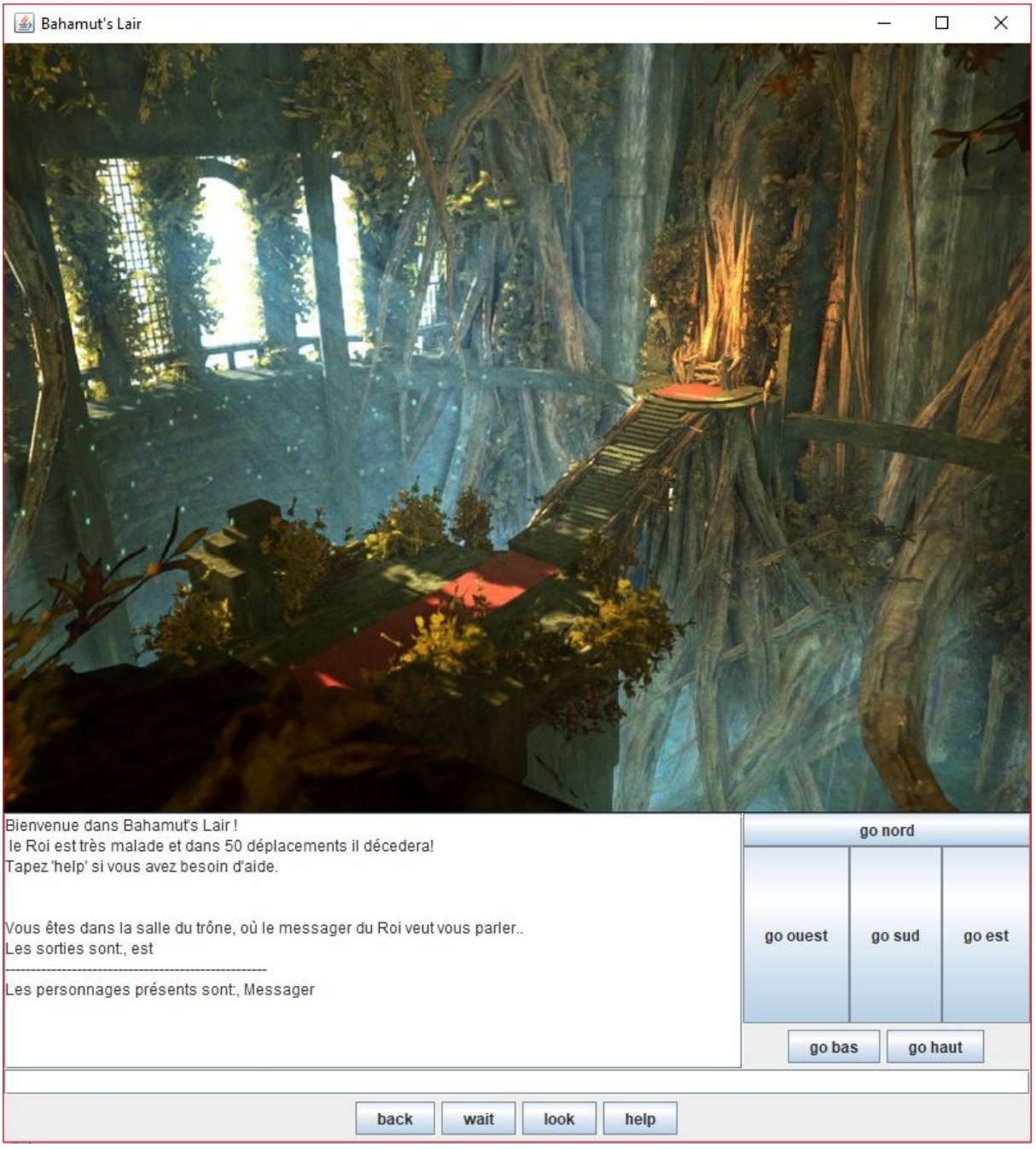
La commande marche à la fois sous Linux et Windows.

7.58.2

Lien ajouté sur le site.

7.58.3

Je me contenterai de celle déjà implémentée, visible ci dessous:



7.60.2

Aucune modification ne sera ajoutée à l'interface.

7.60.3

Toute la javadoc est terminée.

7.60.4

Les javadocs sont générées.

7.63

Les scénarios sont refaits et fonctionnent.

7.63.1

Aucun warning présent à la compilation

7.63.2

Le jeu est désormais jouable, condition de victoire et défaite ajoutées.

7.63.3

Tout le scénario est incorporé

7.63.4

Les javadocs sont générées et mises sur le site.

III) Fonctionnalités Supplémentaires

En plus des exercices obligatoires, j'ai décidé d'implémenter un système de combat et d'équipement.
Ainsi tout Player possède une Arme et une Armure, tout deux étant des classes héritant d'Item, il est possible de s'en équiper afin d'augmenter ses statistiques:
-Vie
-Dégâts
-Resistance

Les NPC possèdent les même statistiques ainsi qu'un objet qui sera déposé dans la salle à leur mort.
Ainsi le Bahamut laissera tomber une fiole de son sang à sa mort et Fenrir une arme.

Ensuite j'ai voulu implémenter des consommables pour restaurer de la vie au joueur.
J'ai donc créé une classe Food héritant de Item qui a comme attribut un nombre d'utilisation et une quantité de soin à apporter.
Ainsi lorsque l'on écrira "eat n" avec "n" un objet de type Food, on sera soigné du montant de vie de cet objet, et on en consommera un.

IV) Déclaration anti-plagiat

Je certifie n'avoir pris aucun morceau de code hormis ce qui était fourni dans les projet zuul-*.jar