10/05/2013

Rapport : L'enlèvement du Père Noël

(A3P 2012/2013)

I. A) Auteurs

Célia PRIOL - Benoît CHAUVEAU - Groupe 4G

I.B) Thème

Le Père Noël a été enlevé, il faut le retrouver.

I.C) Résumé du Scénario

Le Père Noël est capturé par des enfants dans une vieille maison en Ecosse, alors qu'il, entame sa tournée de cadeaux le soir du 24 décembre. Il se retrouve prisonnier, incapable de continuer à distribuer les cadeaux

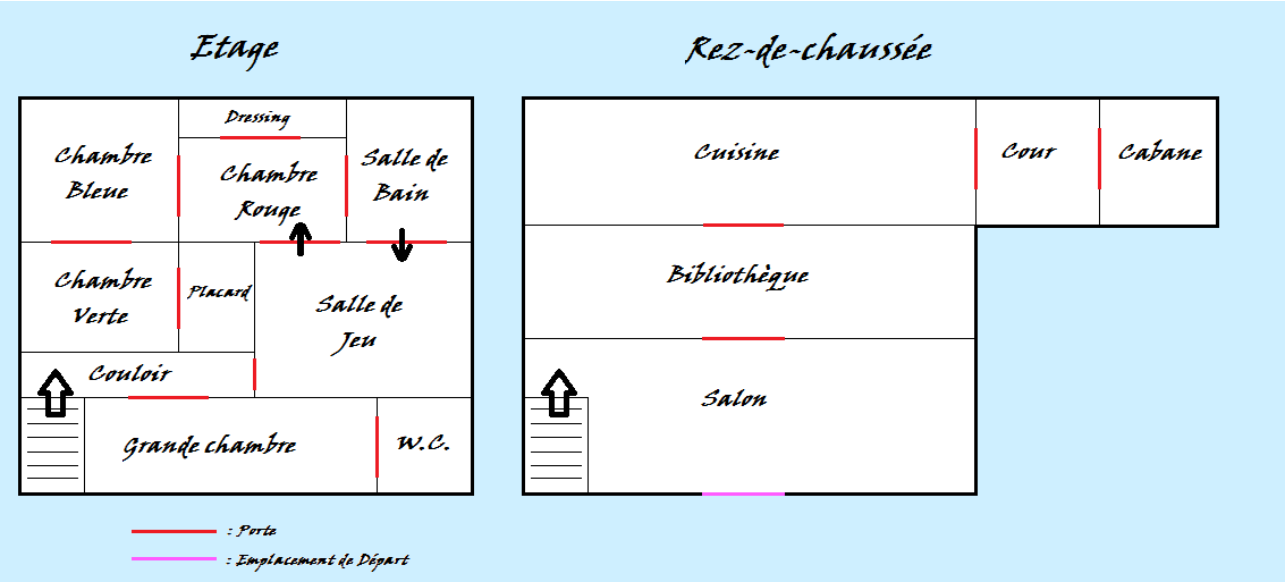
Bernie, son fidèle renne a tout vu et est sous le choc. Il parvient tout de même à joindre secrètement un détective privé afin de retrouver son maître et les cadeaux.

Vous êtes ce détective.

Bernie vous informe de la maison où est retenu le Père Noël et vous vous y rendez immédiatement, seulement il est incapable de savoir dans quelle le Père Noël est séquestré. Les enfants ne sont plus sur les lieux. En plus du Père Noël, vous devez retrouver 4 cadeaux volés par les enfants qui sont cachés dans la maison.

Il faut faire vite car les cadeaux doivent tous être distribués avant le lever du jour !

I.D) Plan



I.E) Scénario détaillé

Le joueur commence dans le salon.
Il faut d'abord se diriger vers la cuisine, manger le cookie, puis aller vers la cabane pour récupérer deux cadeaux, puis trouver la clé, récupérer le cadeau dans les toilettes, récupérer le cadeau dans la salle de bain, et enfin arriver jusqu'au Père Noël, le tout en un nombre de déplacements limités.

I.F) Détail des lieux, items, personnages

Le Père Noël est dans le placard.
Il y a deux cadeaux dans la cabane.
Il y a un cadeau dans les toilettes.
Il y a un cadeau dans la salle de bain.
Consommer le Cookie (obtenu dans la cuisine) augmente le nombre de déplacements autorisés.
La Clé (obtenue dans la salle de jeux) permet d'entrer dans les toilettes.

I.G) Situations gagnantes et perdantes

Victoire :

- Trouver la pièce où est caché le Père Noël.
- Retrouver les 4 Cadeaux dissimulés dans les différentes pièces de la maison.

Défaite :

- Ne pas retrouver le Père Noël et atteindre la limite de déplacements autorisés.
- Ne pas retrouver les 4 Cadeaux et atteindre la limite de déplacements autorisés.

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

Aucunes

I.I) Commentaires

Site Internet : http://www.esiee.fr/~chauveab/pere-noel/

7.1 Découverte zuul-bad.

L’application permet d'explorer les différentes pièces d'un jeu d'aventure dans un campus universitaire.
Au commencement du jeu, le joueur se trouve dans « the outside main entrance of the university »

- Il y a 3 commandes dans ce jeu : « go », « quit » et « help ».
Pour se déplacer il suffit de taper « go » suivi de la direction souhaitée parmi celle sproposée.

On a ainsi :
go north qui permet d'aller dans la pièce située au nord de celle où se trouve le joueur.
go south qui permet d'aller dans la pièce située au sud de celle où se trouve le joueur.
go east qui permet d'aller dans la pièce située à l'est de celle où se trouve le joueur.
go west qui permet d'aller dans la pièce située à l'ouest de celle où se trouve le joueur.

Certaines pièces ne possèdent pas de porte vers toutes les directions et l'application renvoie le message : « There is no door »

La commande « quit » permet de quitter le jeu

La commande « help » permet de donner le contexte du jeu qui est : « You are lost. You are alone. You wander around at the university. »

Il y a au total 5 pièces auquel le joueur peut accéder dans ce jeu : outside, campus pub, lecture theatre, computing lab et computing admin office.

7.2 Rôle des classes

Il y a 5 classes dans le jeu : Game, Parser, Command, CommandWord, Room.

Classe Game : initialise les autres classes, démarre, affiche et contrôle le jeu.

Classe Room : crée des pièces pouvant avoir des sorties sur d'autres pièces.

Classe Command : correspond aux « ordres » donnés par l'utilisateur. Elle vérifie que les commandes demandées sont valides. Les commandes sont composées soit d'un unique mot (QUIT, HELP), soit pour GO de deux mots, (GO suivi de la direction).Si le deuxième mot n'est pas marqué ou est invalide, il sera marqué comme null.

Classe CommandWords : liste toutes les commandes valides dans ce jeu.

Classe Parser : lit et interprète chaque ligne de commande en deux mots, et les transforme en commandes Si la commande entrée n'est pas connue alors Command retourne unknownCommand.

7.2.1 Classe Scanner

La classe Scanner est une classe du JDK (qu'il faut importer avec « import java.util.Scanner »)
Elle sert à lire les mots et lignes tapés au clavier.

7.5 PrintLocationInfo

Pour faciliter la cohésion entre les classes, on crée une méthode permettant d'afficher des informations sur la pièce. Cela indique au joueur l'endroit où il se trouve et dans quelle(s) direction(s) il peut aller.

```
private void printLocationInfo()

{
System.out.println("You are " + currentRoom.getDescription());
System.out.print("Exits: ");

        if (currentRoom.northExit != null)
        {
            System.out.print("north ");
        }

        if (currentRoom.eastExit != null)
        {
        System.out.print("east ");
        }

        if (currentRoom.southExit != null)
        {
        System.out.print("south ");
        }

        if (currentRoom.westExit != null)
        {
        System.out.print("west ");
        }
System.out.print()
}
```

Cette méthode est appelée par PrintWelcome et goRoom.

7.6 GetExit

Nous déclarons attributs de la classe Room privés pour réduire le couplage et pour respecter le principe d'encapsulation comme demandé dans l'exercice.
Nous devons créer un accesseur getExit pour les utiliser :

```
public Room getExit(String pdirection)
{
        if (direction.equals("north")
        {
        return northExit;
        }

        if (direction.equals("east")
        {
        return eastExit;
        }

        if (direction.equals("west")
        {
        return westExit;
        }

        if (direction.equals("south")
        {
        return southExit;
        }
return null;
}
```

7.7 GetExitString

La méthode getExitString(), créée dans la classe Room, retourne une String qui indique les sorties de la pièce dans laquelle se trouve le joueur.

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

        if (this.getExit("east") != null)
        {
        Exits+="east ";
        }

        if (this.getExit("south")!= null)
        {
        Exits+="south ";
        }

        if (this.getExit("west") != null)
        {
        Exits+="west ";
        }
return Exits;
}
```

On modifie ensuite la méthode prinLocationInfo() en appelant la méthode getExitString() avec l'attribut de type Room, currentRoom.

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

7.8 HashMap

Dans cet exercice nous allons utiliser une Hashmap pour stocker des sorties plutôt que des variables comme précédemment. En procédant ainsi nous sommes capables de gérer plusieurs sorties.
Chaque entrée aura donc une clé de type « chaîne de caractères » et un objet de type « Room » comme valeur
Nous importons donc une Hashmap dans la classe Game et créons un attribut de type HashMap.

Pour chaque lieu nous avons donc :

- La création des pièces

Ex :

```
salon = new Room("dans le salon",);
bibliotheque = new Room("dans la bibliotheque");
```

- L'initialisation des pièces

Ex :

```
salon.setExits("nord",bibliotheque);
bibliotheque.setExits("sud",salon);
```

7.8.1 Déplacement vertical

Nous ajoutons deux nouvelles directions à notre jeu ; haut et bas. Ainsi, le joueur pourra taper « go haut » pour monter d'un étage et « go bas » pour descendre.

7.9 keySet

La méthode keySet retourne toutes les clés d'une Hashmap sous forme d'une collection de type Set (c'est à dire dans laquelle chaque élément n'existe qu'une seule fois.)

7.10 GetExitsString

Importer la classe Set : importjava.util.Set

GetExitString() retourne les sorties disponibles d'un lieu sous forme d'un String, tout d'abord en créant une chaîne de caractères contenants les exits déterminées via keySet et retournant des Strings caractéristiques des sorties.

```
public String getExitString()
{
        String returnString="Exits:";
        Set<String> keys=exits.keySet(); // C'est l'iterator
        for (String exit : keys) // Boucle for…each
        returnString +=""+exit;
        return returnString;
}
```

7.11 GetLongDescription

Nous ajoutons dans notre programme la méthode GetLonDescription(), qui donne des informations au joueur sur la pièce dans laquelle il se trouve.

```
/**
* Return a long description of this room, of the form :
* You are in the kitchen.
* Exits: north west
* @return A description of the room, including exits.
*/

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

7.14 Méthode Look

Nous souhaitons ajouter à notre jeu une commande nous permettant de réafficher à la demande du joueur la description de la pièce et de ses sorties.

Pour cela nous ajoutons tout d'abord une nouvelle commande dans la classe CommanWords().

Un tableau constant conserve les commandes valides.

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

Nous créons ensuite dans la classe Game, une méthode look() :

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

Nous ajoutons ensuite dans la méthode processCommand() dans la classe Game : un nouveau « cas », qui appellera la méthode look() quand la commande look sera reconnue.

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

        String commandWord = command.getCommandWord();
        if (commandWord.equals("help"))
        { printHelp(); }

        else if (commandWord.equals("go"))
        { goRoom(command); }

        else if (commandWord.equals("look"))
        { look(); }

        else if (commandWord.equals("quit"))
        { wantToQuit = quit(command);
        return wantToQuit; }
}
```

7.14 Méthode Eat

Nous faisons exactement pareil que pour la méthode look. Nous ajoutons une commande eat() dans la classe CommanWords().

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

Nous créons dans la classe Game une méthode eat() :

```
public String eat ()
{
        return " Vous avez mangé et vous n'avez plus faim du tout"
}

private boolean processCommand(Command command)
{
        boolean wantToQuit = false;

        if (command.isUnknown())
        {
        System.out.println("I don't know what you mean...");
        return false;
        }

        String commandWord = command.getCommandWord();

        if (commandWord.equals("help"))
        { printHelp(); }

        else if (commandWord.equals("go"))
        { goRoom(command); }

        else if (commandWord.equals("look"))
        { look(); }

        else if (commandWord.equals("eat"))
        { eat(); }

        else if (commandWord.equals("quit"))
        { wantToQuit = quit(command);
        return wantToQuit; }
}
```

7.16 ShowAll

Lorsque nous utilisons la commande help, la commande look n'est pas listée. Pour règler ce problème « proprement », nous allons tout d'abord nous intéresser à la classe Commandword. En effet comme cette gère les commandes, elle doit gérer leur affichage.

Nous ajoutons ainsi la méthode showAll()

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

System.out.println() ;
}
```

Mais il s'avère plus facile d'aller chercher les commandes là où elles sont listées à savoir dans la classe Parser.

Nous ajoutons ainsi à la méthode printhelp :

```
System.out.println("Your command words are:");
parser.showCommands();
```

Puis nous créons la méthode showCommands dans la classe Parser :

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

7.17 Pour créer une nouvelle commande nous n'avons pas besoin de toucher à la classe Game,

7.18 Nous modifions la méthode showCommands en getCommandList :

```
public String getCommandList ()
{
return commands.getCommandList();
}
```

7.18.6 Zuul with images

Nous étudions le projet zuul with images qui permet de mettre des images dans son jeu.
3 nouvelles classes sont apparues

GameEngine : La classe GameEngine fait appel à toutes les classes, crée toutes les Rooms ainsi que du texte grace à l'attribu gui, de type UserInterface. La pluspart des méthodes de la classe Game sont déplacées dans cette classe.

UserInterface : Cette classe gère l'interface graphique et les événements.

Game : Cette classe a pour rôle de lancer le jeu : elle crée l'interface graphique, puis le jeu grâce à GameEngine.

7.18 Ajout des boutons

Ils permettent de déclencher automatiquement une des commandes du jeu.

7.20 à 22 Ajout des Items

Pour ajouter des objets, nous créons une classe Item. Chacun de ces objets possède une caractéristique, un poids, et un nom.
Nous avons donc créé les attributs suivant : aItemName, aDescription, aPoids.
Il faut donc créer un accesseur pour chacun de ces attributs dans cette classe, ainsi qu'une méthode GetDescription(), qui permet de retourner le descriptif de l'objet et son poids.

Les objets sont créés dans la méthode createRooms de GameEngine.

```
public class Item
{
        private String aItemName;
        private String aDescription ;
        private double aPoids ;
        private ItemList aItemcurrentRoom ;


/**
*Constructeur pour les objets de la classe Item.
*/

public Item (final String pItemName,final String pDescription, final double pPoids)
{
        this.aDescription = pDescription ;
        this.aPoids = pPoids ;
```

```
        this.aItemName =pItemName;
}


public Item() // constructeur par défaut.
{
        this.aDescription = "Il n'y a pas d'objet";
        this.aPoids = 0 ;
}


/**
* Renvoie la description et le poids de l'objet
*/

public double getItemPoids()
{
        return aPoids;
}

public String getItemDescription()
{
        return aDescription;
}

public String getLongItem1Description()
{
        return "vous avez " + aDescription + " " + "son poids est" + " " +  aPoids;
}

public String getName()
{
        return aItemName;
}
```

7.23 Commande Back

Implémentation d'une commande permettant de revenir dans la pièce précédente.

7.26 Stack

Stack est une structure de donnée, fonctionnant de la même manière qu'une pile. Les derniers éléments ajoutés à la pile seront les premiers à être récupérés.
Ceci correspond parfaitement à ce dont nous avons besoin pour notre commande back. En effet nous allons pouvoir « empiler » Les différentes Rooms par lesquelles le joueur est

passé et ainsi les restituer dans l'ordre inverse jusqu'à la première pièce du jeu.

Stack peut appeler plusieurs méthodes :

- empty() : Effectue un test pour savoir la pile est vide ou non (renvoie un boolean)
- push() : Ajoute un objet en haut de la pile
- pop() : Retourne l'objet en haut de la pile et le supprime
- peek() : retourne le dernier élément de la collection sans le supprimer

Dans la classe GameEngine, nous commençons par importer Stack :

```
import java.util.Stack;
```

Ensuite on crée un attribut de type Stack, que nous avons nommé Stack: private Stack<Room> stack, que nous initialisons dans le constructeur, puis nous créons la méthode :

```
public void backRoom()
{
        if (stack.empty()==true)
        {
        gui.println("il n'y a pas de piece");
        }
        else
        {
        gui.println("vous êtes dans la piece précédente ");
        }

        currentRoom=stack.pop();
        if(currentRoom.getImageName() != null)
        {
        gui.showImage(currentRoom.getImageName());
        }
}
```

La méthode Empty permet de vérifier qu'il y a bien une pièce à renvoyer et ainsi éviter un problème au cas où le joueur appellerait la commande back dans la pièce initiale.

La méthode renvoie la pièce en haut de la pile (la dernière pièce visitée par le joueur).

7.27 Test

Nous créons une nouvelle commande, la commande test.

```
private void test( Command command )
{
        if ( stack.empty() == false )
        {
        gui.println( "il est nécessaire de se trouver sur le palier pour pouvoir tester le jeu");
        return;
        }

        if ( !command.hasSecondWord() ) // if there is no second word, we don't know where to go
        {
        gui.println( "Que voulez-vous tester ?" );
        return;
        }

        try
        {
        File fichier= new File( command.getSecondWord() );
        Scanner scan= new Scanner( fichier );
        while ( scan.hasNextLine() )
        {
        interpretCommand( scan.nextLine() );
        }
        }

        catch ( Exception e )
        {
        gui.println( "le fichier n'existe pas" );
        }
}
```

Et nous ajoutons dans la méthode interpretCommand :

```
if ( commandWord.equals( "test" ) )
{
        test(command);
}
```

7.29 Classe Player

Nous créons une classe Player afin de stocker tous les attributs et les méthodes spécifiques du joueur. Nous aurions pu stocker toutes ces informations dans la classe GameEngine, mais celle-ci est déjà bien chargée.

Nous avons donc :

Classe Player :

```
public class Player
{
        private String aName;
        private Room currentRoom;
        private int aPoidsJoueur ;


/**
* Constructeur d'objets de classe Player
*/

public Player(Room currentRoom,String anName)
{
        this.aName= pname
        this.currentRoom = currentRoom;
}

public String getName()
{
        return aName;
}

public void changeRoom( Room nextRoom)
{
        currentRoom = nextRoom ;
}

public Room getcurrentRoom()
{
        return currentRoom;
}
```

L'attribut Room currentRoom est spécifique au joueur puisqu'il correspond à la pièce dans laquelle il se trouve. Il est donc logique de le déplacer dans la classe Player et de le supprimer de la classe GameEngine. Seulement de nombreuses méthodes de cette classe on a besoin de cet attribut. C'est pour cela que nous avons créé l'accesseur getcurrentRoom() et le la méthode changeRoom dans la classe Player.

Nous créons ensuite un attribut de type Player dans la classe GameEngine : Player aPlayer et il suffit de remplacer tous les currentRoom par aPlayer.getcurrentRoom().

7.30 Take et Drop

Nous souhaitons créer deux commandes permettant au joueur de prendre un objet, de le stocker dans son inventaire (donc de le garder sur lui lorsque passe de pièce en pièce) et de le déposer quand il le souhaite.

Pour cela, nous créons dans la classe Player la méthode suivante :

```
public String getInventory()
{
        String returnString = "\n"+"Vos objets sont :";
        Set<String> keys = aInventaire.getkeySet();
        for(String clé : keys)
        {
        Item trueItem = aInventaire.getItem(clé);
        returnString += " "+trueItem.getLongItem1Description();
        }

return returnString;
}
```

Elle permet de stocker les objets pris par le jouer dans son inventaire.

Nous créons ensuite les méthodes Take et Drop dans GameEngine.

```
/**
* Permet de prendre un objet
*/

private void take(Command command)
{
        if (!command.hasSecondWord())
        {
        gui.println("Prendre quoi ? ");
        return;
        }

        String vName  = command.getSecondWord();
        Item vItem = this.aPlayer.getcurrentRoom().getItem(vName);
        int vPoids = this.aPlayer.getPlayerPoids();
        if (vItem == null)
        {
        gui.println(" Il n'y a pas cet objet dans la piece ! ");
        return;
        }
```

```
        else if (vItem.getItemPoids()+vPoids > this.aPlayer.maxPoids())
        {
        gui.println("Vous ne pouvez pas prendre cet objet vous n'avez pas assez de force
        pour tout porter !");
        return;
        }

        else
        {
        this.aPlayer.addItem(vName,vItem);
        this.aPlayer.getcurrentRoom().removeItem(vName);
        gui.println("Vous avez pris"+ vName);
        gui.println(aPlayer.getInventory());
        }
        gui.println("");
}


/**
* Command Drop
*/

public void drop(Command command)
{
        if (!command.hasSecondWord()) // if there is no second word, we don't know what
to do...
        {
        gui.println("Poser quoi ?");
        return;
        }

        String vName = command.getSecondWord();
        Item vItem = this.aPlayer.getItem(vName);

        if (vItem == null)
        {
        gui.println("Vous n'avez pas cet objet dans votre inventaire");
        }

        else
        {
        aPlayer.getcurrentRoom().addItem(vName, vItem );
        aPlayer.removeItem(vName);
        gui.println("Vous avez déposé "+vName);
        }
}
```

Nous avons dans la classe Room, les méthodes removeItem et getItem

7.31.1 Classe ItemList

Nous pouvons constater qu'à ce stade du jeu, les clases Room et Player possèdent toutes les deux des méthodes de la classe et possède toutes deux une HashMap <String,Item>. Il est donc cohérent de créer une classe ItemList réunissant ces méthodes communes.

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


**
* Constructeur de la liste d'objets avec l'outil HashMap.
*/

public ItemList()
{
        ItemList = new HashMap<String,Item>();
}


/**
* Ajoute un objet a la liste.
*/

public void addItem(String ItemName, Item Objet)
{
        ItemList.put(ItemName, Objet);
}


/**
* Renvoie l'objet grâce à son nom.
*/

public Item getItem(String ItemName)
{
        return ItemList.get(ItemName);
}


/**
* Supprime l'objet.
* @param nom Représente le nom de l'objet à supprimer.
*/

public void removeItem(String ItemName)
{
```

```
        ItemList.remove(ItemName);
}

/**
* Teste si l'objet existe.
* @param nom Represente le nom de l'objet a tester.
*/

public boolean existItem(String ItemName)
{
        return ItemList.containsKey(ItemName);
}


public Set<String> getkeySet()
{
        return ItemList.keySet();
}


public HashMap<String,Item> getItemPlayer()
{
        return this.ItemList;
}

}
```

7.32 définir un poids maximal

Nous ajoutons un poids maximal que le joueur puisse porter.

Pour cela il suffit de définir un attribut que l'on initialise au poids maximum et l'on vérifie que ce dernier est toujours inférieur au poids que porte le joueur. (Nous avions dès le départ ajouté un poids aux Items)

7.33 Création d'un inventaire

Nous avions déjà créé un inventaire au Player.

7.34 Magic cookie

Nous devons créer un objet, qui une fois mangé par le joueur, augmente le poids maximal qu'il peut porter.

Nous créons la méthode suivante dans Game Engine :

```
public void eat(Command command)
{
        if(!command.hasSecondWord())
        {
        gui.println("Que voulez vous manger ?");
        }

        String vName = command.getSecondWord();

        if(aPlayer.getItem(vName) == null)
        {
        gui.println("Vous ne pouvez pas manger cet objet !");
        }

        else
        {
        gui.println(" Vous avez mangé " + vName + ".");
        if (vName.equals("cookie"))
        {
        gui.println("Vous pouvez porter plus d'objets maintenant. Le poids maximal que
        vous pouvez porter a été augmenté de 20 kg.");
        aPlayer.setMaxPoids(50);
        }
        else
        {
        gui.println("Vous avez assez mangé.");
        }
        }
}
```

Si ce cookie n'a pas été pris par le joueur, et n'est par conséquent pas dans son inventaire, alors la commande renvoie « Vous ne pouvez pas manger cet objet »
Si le cookie est dans l'inventaire et que le joueur le mange alors le poids maximal augmente.

7.35 Zuul with enum

Nous avons étudié le projet zuul with enum et réalisé les modifications nécessaires.

7.35.2 Switch

Nous utilisons un Switch dans méthode interpretCommand de façon à éviter la répétition de la structure « else if ». Le Switch prend une variable de type CommandWord, qui correspondra à la commande donné par le joueur.

Nous avons donc :

```
public void interpretCommand(String commandLine)
{
	gui.println(commandLine);
	Command command = parser.getCommand(commandLine);
	CommandWord commandWord = command.getCommandWord();

	switch(commandWord)
	{
	case UNKNOWN : gui.println("Je ne vois pas ce que tu veux dire..");
	break;
	case HELP : printHelp();
	break;
	case QUIT : endGame();
	break;
	case GO : goRoom(command);
	break;
	case LOOK : look();
	break;
	case EAT : eat(command);
	break;
	case BACK : backRoom();
	break;
	case DROP : drop(command);
	break;
	case TAKE : take(command);
	break;
	case USE: utiliser ();
	break;
	case LOAD : charger () ;
	break;
	case TEST : test(command);
	break;
}
```

7.40 zuul-with-enums-v2

Nous avons étudié le projet zuul-with-enums-v2, et effectué les modifications nécessaires dans notre jeu.

7.42 TimeLimit

Dans cet exercice nous avons instauré un nombre de pas limité.

Pour cela nous avons créé dans la classe GameEngine deux attributs de type int :
L'un correspond au nombre de pas du joueur et est initialisé à 0 :

```
private int aMove = 0
```

L'autre correspond au nombre de pas maximal que le joueur ai le droit de faire :

```
private int aMaxMove = 2;
```

Nous ajoutons ensuite cette partie de code à la méthode goRoom :

```
if(aMove >= aMaxMove  )
{
        this.Dead();
        this.endGame();
}

else
{
this.aMove++;
}
```

Ainsi que la méthode Dead () :

```
private void Dead()
{
        gui.println("\n Trop tard, vous n'avez pas réussi a retrouver le Père Noel a temps !
        ");
}
```

Ainsi quand le nombre de pas du joueur est supérieur au nombre de pas maximum, goRoom renvoie la méthode Dead() ainsi que la méthode engGame(), qui met fin au jeu.

Nous nous sommes rendus compte que nous aurions dû déplacer ces méthodes dans Player (car le nombre de pas est relatif au jouer) et les appeler dans GameEngine grâce à l'attribut de type Player.

7.43 TrapDoor

Dans cet exercice nous allons réaliser une porte à sens unique. Ainsi, un joueur pourra passer une porte dans un sens et ne plus pouvoir la franchir dans l'autre sens.
Les sortis de nos pièces sont initialisées dans la méthode createRoom() de GamEngine
Nous avons normalement :

```
chambrerouge.setExits("sud", salledejeu);
salledejeu.setExits("nord",chambrerouge);
```

Il suffit tout simplement de marquer :

```
chambrerouge.setExits("sud", null);
salledejeu.setExits("nord",chambrerouge);
```

Ainsi, nous pouvons entrer par la porte nord de la salle de jeu dans la chambre rouge, mais pas ressortir par cette dernière.

7.44 Beamer

Nous avions pensé à créer une classe Beamer (ou téléporteur), car ce dernier doit être considéré comme un objet, mais nous avions rencontré un problème.
Nous avons du coup créé un Beamer que le joueur possède en permanence sur lui.

Le Beamer doit être chargé dans une pièce puis utilisé pour transporter directement le joueur dans cette pièce malgré les déplacement qu'il a effectué et la « distance » qui le sépare de cette pièce .

Nous devons ajouter deux commandes dans la classe commandWords et interpretCommand : Utiliser et charger.

Dans la classe Player, après avoir créé un attribut de type Room (Room aBeamer ;) (qui permet de « stocker » la pièce chargé), nous avons créé les méthodes suivantes

```
/**
* Charger the beamer to the current room
*/
public void chargeBeamer()
{
        aBeamer = currentRoom;
}

/**
* Fires the beamer
*/
public boolean fireBeamer()
{
        if(aBeamer != null)
        {
        changeRoom(aBeamer);
        return true;
        }
return false;
}
```

Nous créons ensuite les méthodes « utiliser » et « charger » dans GameEngine.

```
private void charger()
{
        aPlayer.chargeBeamer();
        gui.println("Vous venez de charger votre beamer.");
}

private void utiliser()
{
        if(aPlayer.fireBeamer())
        {
        gui.println("Vous venez d'utiliser votre beamer avec succes ! ");
        gui.println(aPlayer.getLongDescription());
        gui.showImage(aPlayer.getcurrentRoom().getImageName());
        }

        else
        {
        gui.println("Vous n'avez pas charge votre beamer! Charger le dans une piece, puis
utilser le.");
        }
}
```