ESIEE PARIS

Projet Zuul : 404 error : PC not found

Rapport Projet Zuul encadré par Denis BUREAU

Année : 2024/2025

Par Pierre MATAR.

Table des matières :

1) Présentation du projet Zuul

L'objectif de ce projet est de développer un jeu d'aventure interactif en Java, basé sur le concept de Zuul. Le but étant de réaliser une introduction à la programmation objet en Java.

1.1) Auteur

Pierre MATAR, étudiant en B3 (3ème année du bachelor BestM) à ESIEE Paris.

1.2) Thème

Dans une entreprise de technologie, Joe doit trouver son ordinateur pour résoudre un bug bloquant sur un projet avant une présentation devant les dirigeants de l'entreprise.

1.3) Résumé du scénario

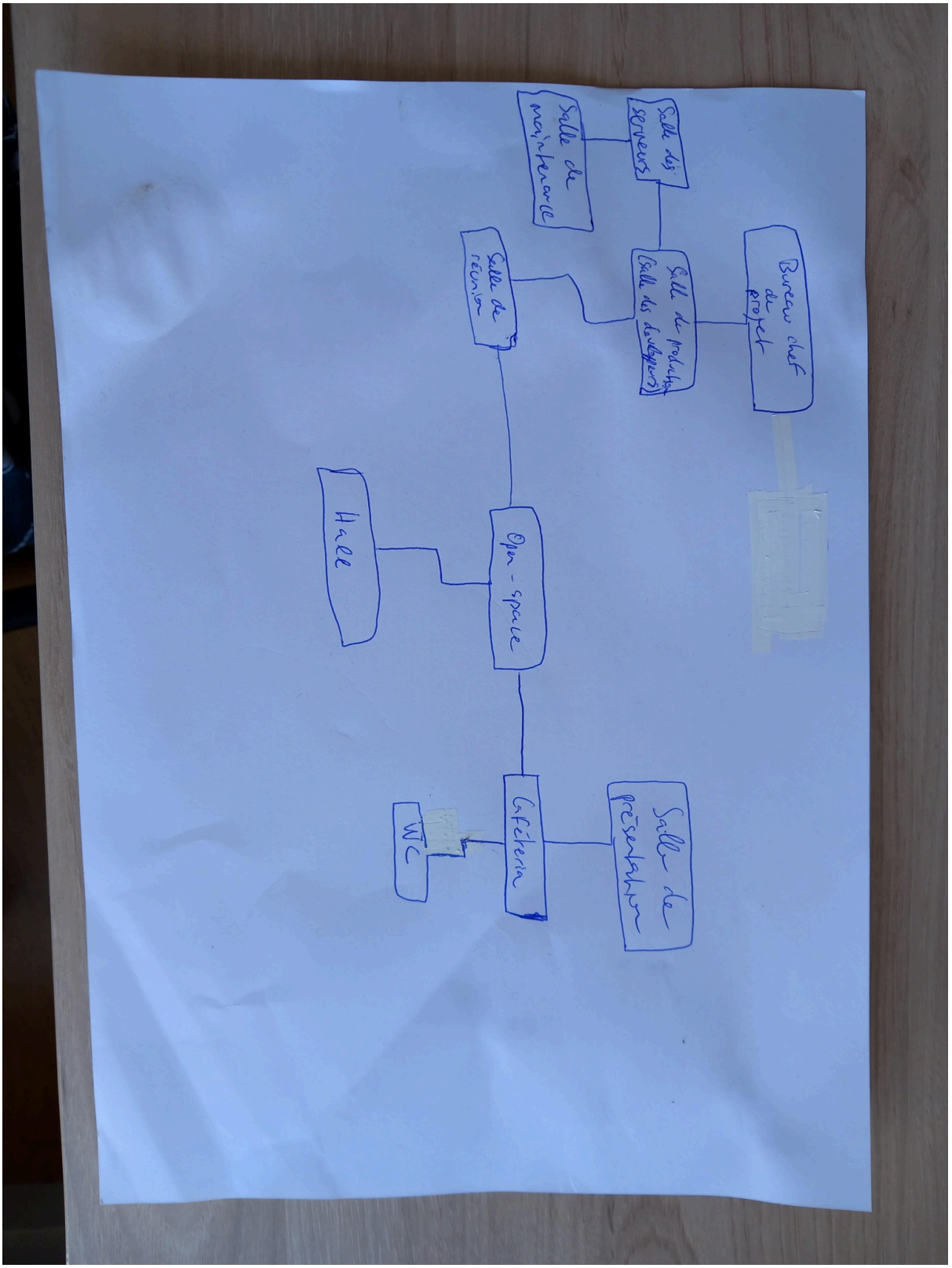
Joe, un jeune développeur de 25 ans, a toujours rêvé de travailler dans une grande entreprise de technologie. Fraîchement diplômé, il décroche son premier emploi en tant que développeur full-stack junior. Plein d'ambition, Joe voit ce travail comme l'opportunité idéale pour prouver sa valeur et gravir les échelons vers son rêve de devenir chef de projet.

Ses premiers jours se passent sans encombre, mais très vite, il est confronté à une mission d'une importance capitale : résoudre un bug bloquant dans un logiciel essentiel avant une présentation critique devant les hauts dirigeants de l'entreprise. C'est un projet sur lequel toute l'équipe travaille depuis des semaines, et Joe, désigné pour faire la dernière révision, doit prouver qu'il est à la hauteur.

Cependant, au moment de se préparer pour la tâche, Joe découvre avec stupeur que son ordinateur a disparu. Son PC, contenant les informations vitales pour corriger le bug, est introuvable. C'est ainsi que commence sa première véritable mission : retrouver son ordinateur et résoudre ce bug avant la réunion, ou risquer de compromettre l'avenir de l'entreprise... et sa carrière. Traversant les différents services, salles et énigmes de l'entreprise, Joe doit non seulement retrouver son ordinateur, mais aussi prouver ses compétences à ses collègues,

Le but du jeu étant donc de parvenir à réaliser sa première mission en entreprise, donc de pouvoir retrouver son PC et corriger le bug.

1.4) Plan


Bureau chef de projet
Salle de réunion
Salle de maintenance
Salle des serveurs
Salle de réunion
Open - space
Hall
Salle de présentation
Cafétéria
WC

Les relations en zig-zag représentent la possibilité de faire un Up-Down tandis que les relations directes représentent les directions Nord-Sud-Est-Ouest

1.5) Détail des lieux, items et personnages

Voici mes premières idées :

1. **Hall d'entrée** : Point de départ avec des instructions.
2. **Salle des développeurs** : Plusieurs PC et des post-it avec des indices utiles.
3. **Salle des serveurs** : Un serveur défectueux nécessitant une intervention technique.
4. **Salle de réunion** : Énigme de sécurité à résoudre pour accéder à l'étage supérieur.
5. **Cafétéria** : Un collègue qui détient un badge pour la salle de réunion.
6. **Bureau du chef de projet** : Contient le badge pour entrer dans la salle de présentation.
7. **Open-space** : Plusieurs personnages offrant des informations ou des distractions.
8. **Salle de présentation** : L'ordinateur à corriger pour terminer la mission.
9. **Salle de maintenance** : Cette salle contient des outils et des équipements nécessaires pour résoudre divers problèmes techniques.
10. **WC** : Rien pour l'instant

J'envisage pour la version finale du projet d'ajouter des personnages et compléter les items du scénario qui manquent actuellement dans le jeu.

1.6) Situations gagnantes et perdantes

- **Situation gagnante** : Joe parvient à corriger l'erreur dans le programme avant le début de la réunion avec les dirigeants (il doit pour cela trouver l'item nommé "thePC" et fixer le bug avec la commande fix).
- **Situation perdante** : Joe échoue à résoudre le problème à temps (avant les 25 mouvements consommés autorisés), ce qui entraîne l'échec de sa mission et donc le game over.

1.7) Énigmes

Sera fait pour la version finale du jeu.

1.8) Commandes du jeu

Légende : [ ] obligatoire, ( ) nombre indéterminé et { } optionnel.

- `go [direction]` : Se déplacer d'une salle à une autre.
- `help` : Affiche l'aide.

- `look {(items)}` : Affiche le lieu où se trouve le joueur ainsi que ses sorties possibles ou affiche un ou plusieurs items.
- `quit` : Quitte le jeu.
- `eat {items}` : Affiche que le joueur a mangé ou permet de manger le cookie magique.
- `take [item]:` Permet de prendre un item que comporte une pièce.
- `drop [item]:` Permet de lâcher un item dans une pièce.
- `(back):` Permet au joueur de revenir en arrière, à la room précédente.
- `test [nom du fichier]:` Permet d'exécuter les commandes tests d'un fichier test.
- `fix [mot secret] :` Permet de fixer le bug avec en argument la string bug.
- `Items :` Permet d'afficher les items que porte le joueur.

D'autres commandes seront ajoutées pour la version finale du jeu.

1.9) Commentaires

Le jeu est toujours en cours de développement, le scénario sera totalement incorporé pour la version finale ainsi que l'IHM améliorée (ajout de boutons).

2) Exercices

Exercice 7.5: Voici la méthode printLocationInfo(), implémenter dans la classe Game qui permet d'afficher la current room, donc la salle actuelle ou est le joueur ainsi que les exits possibles de cette salle. Elle nous a permis en particulier de mieux arranger notre code pour qu'il soit plus clair et plus facile à modifier par la suite en supprimant toute la duplication de code.

```
private void printLocationInfo() {

    System.out.println( "You are " + this.aCurrentRoom.getDescription());
    System.out.print( "Exits: " );
    if(this.aCurrentRoom.aNorthExit != null) {
        System.out.print( "north ");
    }
    if(this.aCurrentRoom.aEastExit != null) {
        System.out.print( "east " );
    }

    if(this.aCurrentRoom.aSouthExit != null) {
        System.out . print( " south ");
    }
    if(this.aCurrentRoom.aWestExit != null) {
        System.out.print( " west " );
    }
    System.out.println();
} //printLocationInfo
```

Exercice 7.6: Voici la méthode getExit() implantée dans la classe Room, prenant comme parametre la direction donnée et nous renvoyant la Room de sortie par rapport à ce paramètre. Cette méthode nous a également permis de supprimer la duplication de code dans la classe Game au sein de la méthode goRoom() en particulier.

```
public Room getExit(final String pDirection) {
    if (pDirection.equals("North"))
        return this.aNorthExit;
    if (pDirection.equals("South"))
        return this.aSouthExit;
    if (pDirection.equals("East"))
        return this.aEastExit;
    if (pDirection.equals("West"))
        return this.aWestExit;

    return null;


}
```

Ex 7.7: Nous devons créer une méthode getExitString() dans la classe Room permettant de connaitre les sorties possibles d'une salle. Elle nous permet de réduire la duplication de code notamment dans printWelcome et printLocationInfo.

```
public String getExitString() {
    String vExitString = "Exits:  ";

    if(this.getExit("North") != null) {
        vExitString += "north ";
    }
    if(this.getExit("South") != null) {
        vExitString += "south ";
    }

    if(this.getExit("East") != null) {
        vExitString += "east ";
    }
    if(this.getExit("West") != null) {
        vExitString += "west ";
    }
    return vExitString;

}
```

7.8: Méthode setExits rendu plus simple par l'importation d'une HashMap qui nous permet de stocker les exits un par un au lieu de 4 à la fois, elle nous facilite la création de nouvelles directions comme Up et Down.

7.8.1: j'ai aussi ajouté par la suite 3 nouvelles salles notamment un hall d'entrée pour faire Up pour accéder à l'étage de l'entreprise, on peut également redescendre dans cette pièce en faisant down.

```
    public void setExits(final String pDirection, final Room pNeighbor){
        aExits.put(pDirection, pNeighbor);
    }
```

7.9: Amélioration de la méthode getExitString(), en la modifiant par une boucle for (for each), qui nous permet d'itérer sur les élements (les cléfs) de la HashMap aExits afin de savoir si leur valeur est différent de null afin de concaténer la cléf a vExitString pour renvoyer la liste des directions possibles.

```
public String getExitString()
{
    String vExitString = "Exits:  ";

    for (String element : this.aExits.keySet()) {
        if (this.aExits.get(element) != null)
            vExitString += " "+ element;
    }

    return vExitString;
}
```

7.10:
La méthode getExitString permet de renvoyer une chaîne de caractère avec toutes les sorties possibles. On va tout d'abord déclarer une String vExitString avec le contenu "Exits: " puis on va itérer sur toutes les clés (avec une boucle for foreach) de la hashmap aExits grâce à la méthode keySet() qui permet de récuperer les clés de la hashmap de les mettre dans un objet type Set. On itére donc sur cet objet et si l'exit est différent de la référence null, si cette condition est true alors on concatène avec la String déclarer comme variable locale de la méthode, ensuite on retourne cette String une fois la boucle finie.

7.10.1/7.10.2 : JavaDoc compléter et générer.

7.11 : Voici la méthode getLongDescription qui retourne une String par rapport à la description de la room ainsi que les sorties de celle-ci. On a implémenté cette méthode pour que chaque classe gère ce qui lui appartient, on ne doit pas produire la String à partir de la classe Game, on doit appeler une méthode de la classe Room qui le fait car c'est relatif à une room.

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

7.14: On implémente une nouvelle commande au jeu, look : qui ne fait simplement que afficher la description de la current room ainsi que ses exits par l'appel de la méthode getLongDescription.

```
+     public void look(){
+         System.out.println(this.aCurrentRoom.getLongDescription());
+     }
```

7.15:
La méthode eat est une 2ème nouvelle commande au jeu, elle affiche que le joueur a bien mangé.

```
-     public void eat() {
-         System.out.println("You have eaten now and you are not
  hungry any more");
-     }
```

7.16 et 7.18 : Voici la méthode showAll implémenter dans la classe CommandWords et showCommands implémenter dans la classe Parser qui permettent d'obtenir la liste des commandes disponibles :

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

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

On remarque ensuite que ce sont en fait des accesseurs donc on va changer leur nom et remplacer par get :

```
public String getCommands() {
    return this.aValidCommands.getCommandList();
}
```

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

On oublie pas ensuite de faire la correspondance dans la classe Game comme ceci :

```
System.out.println(this.aParser.getCommands());
```

Cette ligne ajouté dans la méthode help de la classe Game permet d'afficher les commandes disponible à partir de son attribut aParser qui est un objet de type Parser et de pouvoir appellé sa méthode getCommands qui elle appelle la méthode getCommandList de la classe CommandWords afin d'obtenir les commandes dispoibles à partir de son attribut aValidCommands.

7.18.1 : Rien à changer après avoir fait la comparaison.

7.18.2 : Voici getExitString avec l'implémentation de StringBuilder qui est plus performant notamment lors de concaténations de String dans des boucles.

```
public String getExitString() {
    StringBuilder vSb = new StringBuilder("Exits:  ");

    for (String element : this.aExits.keySet()) {
        if (this.aExits.get(element) != null)
            vSb.append(element+" ");
    }

    return vSb.toString();

}//getExitString
```

Je l'ai implémenté dans tous les endroits nécessaires.

7.18.3 : Pas d'image encore trouvée, sera fait pour la version finale du jeu.

7.18.4 : Nom du jeu ajouté dans la méthode printWelcome dans la classe Game. Nom du jeu -> 404 error : PC not found.

7.18.5 : Exercice optionnel à faire à l'exercice 7.46.

7.18.6: Etude faite.

7.18.8 : Bouton permettant de d'appeler la commander look() ajouté à l'interface graphique.

7.19.2 : Création du dossier images fait.

7.20 : Création de la classe Item :

```
public class Item
{
    private String aDescription;
    private int aWeigth;
    private String aName;

    public Item(final String pName, final String pDescription, final int pWeigth) {
        this.aName = pName;
        this.aDescription = pDescription;
        this.aWeigth = pWeigth;
    }

    public String getItemDescription() {
        return this.aDescription;
    }

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

    public String getItemString() {
        return this.aName +" "+this.aDescription + " " + this.aWeigth;
    }
}
```

7.21 : C'est la classe GameEngine qui doit créer les informations des items donc créer les objets de type Item, cela se fait notamment dans la méthode createRoom().
C'est la classe Item qui doit produire/retourner la string décrivant l'Item, car d'après le principe d'une bonne programmation, tout ce qui est en rapport avec une classe doit se faire à l'intérieur de celle-ci et non autre part.
C'est par contre à la classe GameEngine d'afficher à l'écran cette string produite par la classe Item.

7.21.1 : La commande look peut désormais soit afficher la current room ou soit afficher un ou plusieurs item valide d'une room en même temps.

7.22: Pour cet exercice je suis parti sur une ArrayList afin de permettre la gestion de plusieurs Item pour une room. J'ai donc rajouté une procédure addItem(Item) qui permet d'ajouter un item à la room en la stockant dans son ArrayList. J'ai également changer la méthode getAllItemString() appellé dans la méthode getLongDescription() faisant cette fois ci une boucle sur tous les items de l'ArrayList afin d'ajouter à la StringBuilder le nom de l'Item par son accesseur getName() pour retourner à la fin la string des items présent dans la room.
Cependant, après réflexion une HashMap aurait été une meilleur décision car on veut pouvoir obtenir (get) un item par rapport à son nom avec la structure suivante : HashMap<String, Item>

7.22.1 : La collection HashMap est utile pour son système de clé/valeur, cette collection nous permet de rechercher un Item par son nom. Elle nous offre plusieurs méthodes et procédures pour remove, get, add (put) etc..

7.22.2 : J'ai donc intégré des items au jeu grâce à la procédure addItem() dans createRooms().

7.23 : J'ai ajouté la nouvelle commande back dans le jeu, cette dernière permet donc de revenir en arrière donc dans la room précédente grâce à une variable.

7:26 : La commande back peut être utilisée plusieurs fois répétitivement, dans la salle principale du jeu elle ne fonctionne pas, et si l'on revient le bon nombre de fois voire plus cela nous renvoie automatiquement à la salle principale du jeu.

Explication Stack : Une stack suit le principe LIFO, avec push() pour ajouter, pop() pour retirer l'élément au sommet (le dernier), empty() pour vérifier si elle est vide, et peek() pour obtenir l'élément du sommet de la stack.

7.26.1 : Les 2 javadoc ont été générées.

7.28.1 : Implémentation de la commande test ->

```
private void test(final Command pUneCommande) {

    if (!pUneCommande.hasSecondWord()) {
        this.aGui.println("Test what ?");
        return;
    }


    String vSecondWord = pUneCommande.getSecondWord();


    File vDossier = new File("./tests");
    File vFichier = new File(vDossier, vSecondWord + ".txt");


    if (!vFichier.exists()) {
        this.aGui.println("The file " + vSecondWord + ".txt' does not exist.");
        return;
    }

    try (Scanner vScanner = new Scanner(vFichier)) {
        while (vScanner.hasNextLine()) {

            this.interpretCommand(vScanner.nextLine());
        }
    } catch (IOException e) {
        System.out.println("An error occured : " + e.getMessage());
    }
}
```

.

Ici on va utiliser la classe Scanner qui permet de lire les lignes du fichier .txt qui sont des commandes du jeu et de les exécuter ensuite grâce à l'appel de la procédure interpretCommand(String)

7.28.2 : les 3 fichiers de commandes ont bien été créés.

7.29 : La classe Player a été créée, cette classe permet donc de gérer tout ce qui est en rapport avec un Player et de pouvoir séparer clairement le code, on va y stocker par exemple son nom, son poids max qu'il peut porter, son itinéraire (historique des rooms), la current room ou il se trouve… La création de getteurs et setteurs (modificateurs) est donc nécessaire notamment pour get ou set la current room à partir de GameEngine par exemple. Le nom du joueur est aussi demandé avant que la partie commence à l'aide d'un javax.swing.JOptionPane.showInputDialog("Enter your name to start the game : ") avant d'initialiser GameEngine dans le constructeur de la classe Game.

7.30 : Take et Drop sont deux nouvelles commandes implémentées. Take permet de porter un seul item grâce à une variable aItem stockée dans Player, Drop permet de lâcher cet item dans la room ou il se trouve, on a donc des setteurs et getteurs dans Player qui permettent de get ou set un Item à ce Player. Ces deux commandes fonctionnent avec un second mot qui est le nom de l'Item.

7.31 : Dans cet exercice le Player peut porter plusieurs Items grâce à l'implémentation d'une HashMap<String, Items> qui permet une structure clé : Nom de l'Item, valeur : Objet du type Item.

7.31.1 : On est amené à créer une nouvelle classe ItemList suite à la duplication de code qu'on obtient dans Room et Player. Cette classe va donc permettre de gérer la liste des Items que ce soit pour une room ou un player.

Voici une méthode de cette classe :

```
public String getAllItemString(final Object p0) {
    StringBuilder vSb;
    int index = 0;
    if (p0 instanceof Room) {
        if (this.aItemList.isEmpty()) return ".\nNo item here. ";
        vSb = new StringBuilder("\nItems available :");

    }

    else {
        if (this.aItemList.isEmpty()) return "\nYou are not carrying any items.";
        vSb = new StringBuilder("Your items -> ");

    }


    for (String itemName : this.aItemList.keySet()) {
        vSb.append(itemName +" (" + this.getItem(itemName).getItemWeigth() + ")").append(index == this.aItemList.size() - 1 ? "" : " / ");
        index++;
    }

    return vSb.toString();
}
```

getAllItemString() permet donc de produire la string des items de la liste que ce soit pour une liste d'Item d'un player ou d'une room.

7.32 : Voici la procédure de la commande take() après avoir mis en place un poids maximum que peut porter le joueur.

```
private void take(final String pItemName) {
    Item vRoomItem = this.aPlayer.getCurrentRoom().getItem(pItemName);
    if (vRoomItem != null) {

        if (vRoomItem.getItemWeigth() <= this.aPlayer.getPlayerWeigth()) {
            this.aPlayer.addItem(vRoomItem);
            this.aGui.println("You took : " + vRoomItem.getItemString());
            this.aPlayer.setPlayerWeigth(- vRoomItem.getItemWeigth());
            this.aPlayer.getCurrentRoom().removeItem(pItemName);
            this.aGui.println(this.aPlayer.getMyItemsList());
        }
        else this.aGui.println("You can carry " + pItemName + " because you weigth available is :" + this.aPlayer.getPlayerWeigth());


    }

    else
        this.aGui.println("this item does'nt exist in this room...");
}
```

7.33 : J'ai ajouté la commande items permettant de lister tous les items du player par l 'ajout de la méthode getMyItemList() dans Player qui appelle getAllItemsString() de la classe ItemList. Dans GameEngine on fait la correspondance en appelant cette méthode de la classe Player pour afficher la string dans l'interface utilisateur.

7.34 : Pour cet exercice il a fallu faire l'extension de la commande eat, qui à présent peut prendre un second mot et permet de manger le magicCookie et donc de doubler le poids max du joueur si magicCookie est dans la current room ou si ce dernier se trouve dans l'inventaire d'items du joueur.

```
private void eat(final String pItemName) {
    Room vCurrentRoom = this.aPlayer.getCurrentRoom();
    if ((vCurrentRoom.getItem(pItemName) != null || this.aPlayer.getItem(pItemName) != null) && pItemName.equals("magicCookie")) {
        this.aPlayer.setPlayerWeigth(this.aPlayer.getPlayerWeigth());
        this.aGui.println("Miam miam miammmmm, here is your weigth now : " + this.aPlayer.getPlayerWeigth());
        if (vCurrentRoom.getItem(pItemName) != null) vCurrentRoom.removeItem(pItemName);
        else this.aPlayer.removeItem(pItemName);
    }
    else this.aGui.println( this.aPlayer.getName().toUpperCase() + " has eaten now and he's not hungry any more");

}
```

7.34.1 : Mise à jour des fichiers tests fait.

7.34.2 : Génération des 2 javadoc fait.

7.35 - 7.41.2 : (Exercices optionnels)

- Mise en place d'un enum CommandWord qui nous permet d'avoir des instances de commandes du jeu, cet enum contient notamment des instances de commandes déjà initialisées au sein du enum grâce au constructeur appelé directement à la déclaration de ces dernières, chaque objet CommandWord possède donc une description, et un getteur qui permet de get la description d'une commande. Cet enum permet notamment une meilleur sécurité de typage.
- Implémentation d'un switch dans interpretCommand permettant de remplacer plus proprement la succession des if - else if.
- Les classes Command et CommandWords ont également changé après l'implémentation de la classe enum CommandWord.

7.42: Implémentation d'une limite de 25 mouvements consommés à l'issue de laquelle le joueur peut perdre. J'ajoute pour cela une procédure addOneMove qui permet d'ajouter +1 au nombre de move du joueur stocker via une variable NbMoves. A chaque passage dans interpretCommand on regarde si ce nombre de mouvement a atteint 25 alors on lance la fermeture du jeu sinon ou ajoute +1 à ce nombre de mouvement. J'ai également implémenté une possibilité de restart la game par une réponse "yes" à une confirmation avant de quitter complètement le jeu.

7.42.2 : L'IHM sera complété pour la version finale du jeu à savoir l'ajout d'images des rooms et boutons supplémentaires.

7.43 : Une Trap Door a été ajoutée entre la salle de maintenance et la salle des développeurs permettant ainsi ne plus pouvoir revenir vers la salle de maintenance une fois la direction East rentrée à partir de celle-ci. Pour l'implémentation de cette fonctionnalité j'ai ajouté une HashMap aTrapDoor avec une structure : <String, Boolean> permettant pour chaque direction d'une room d'indiquer par un boolean si c'est une trap door ou pas. Il a fallu

ensuite ajouter un paramètre boolean a setExit(), ajouter la méthode isExit(Room) qui permet de voir si la room précédente qu'à traversée le joueur est toujours une exit ou pas pour pouvoir vider la stack altinerary de ce dernier permettant de ne plus pourvoir back. Dans goRoom() ensuite il a fallu rajouter une logique pour pouvoir supprimer la direction de la current room permettant d'aller vers la direction trap door de la précédente room.

7.43.1 : Les 2 javadoc ont bien été générées.

3) Déclaration plagiat

Pas de plagiat jusqu'à présent.