Compte Rendu d'IPO

Informations

Nom du projet : **Les gardiens de la cité enchantée**

Auteur : **Léo Gaillet** (E1-5)

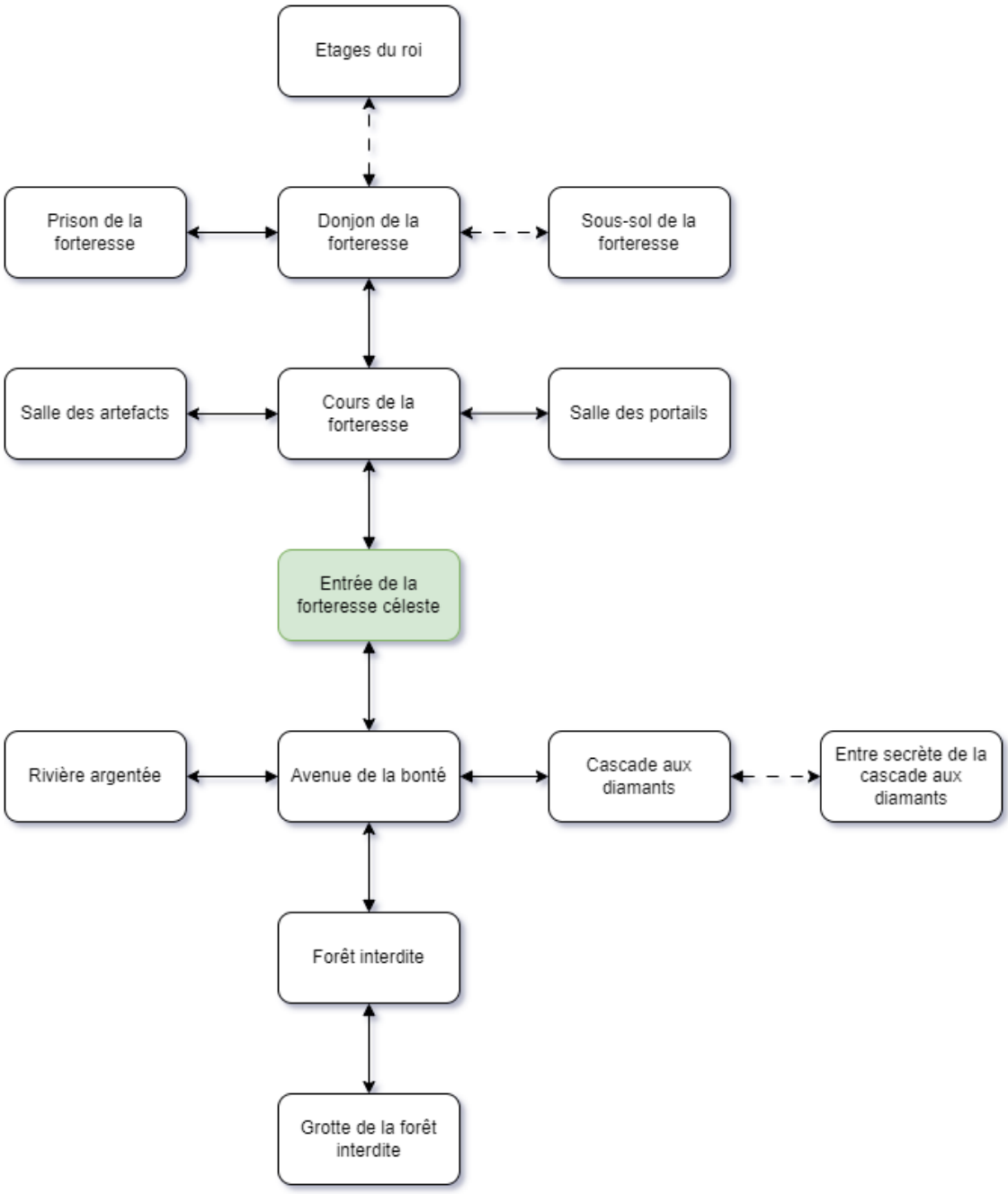
Scénario détaillé : Dans un monde fantastique peuplé de créatures magiques, la cité enchantée était autrefois un lieu de paix et de prospérité. Cependant, un groupe maléfique de sorciers a lancé un sortilège qui a plongé la cité dans l'obscurité et la destruction.

Un jeune héros a été choisi pour devenir un gardien de la cité enchantée. Les gardiens sont des guerriers magiques qui ont juré de protéger la cité et ses habitants, et de combattre les forces du mal qui menacent l'équilibre de l'univers.

Le joueur doit explorer la cité enchantée et ses environs pour trouver des alliés, des armes magiques et des artefacts anciens qui peuvent aider à défaire le sortilège maléfique. Le joueur doit également affronter des ennemis dangereux, tels que des créatures magiques corrompues, des soldats maléfiques et des sorciers puissants.

Au fil de l'aventure, le joueur découvre des indices sur l'identité des sorciers maléfiques et leur plan diabolique. Les gardiens doivent alors rassembler leurs forces et lutter pour sauver la cité enchantée et ramener la paix dans le monde fantastique.

Carte du jeu :



Détails des lieux :

- **Etages du roi** : "The King's Floor. You are certainly appreciated by the King."
- **Donjon du Fort** : "The Dungeon of the Fortress. It's a privilege to be here."
- **Prison du Fort** : "The Prison of the Fortress. It can be some prisoners in there."
- **Sous-sol du Fort** : "The Basement of the Fortress. You may see some bats."
- **Salle des Artéfacts** : "The Artefacts Room. You can feel strange things coming from this place."
- **Salle des Portails** : "The portal room. Portals may teleport you, even if the way by foot is impossible"
- **Cours du Fort** : "The Fortress Yard. See how beautiful are flowers around here."
- **Entrée du Fort** : "The Entrance of The Fortress. Why is this door so tall ?"
- **La Rivière d'Argent** : "The Silver River. Is this silver or just the snow falling on it ?"
- **L'Avenue de la Bonté** : "The Joyful Avenue. This avenue goes so far away that I can't see the end."

- **La Forêt Interdite** : "The Forbidden Forest. The only thing that I can say about this place is it's make me goosebumps"
- **La Grotte de la Forêt Interdite** : "The Cave of The Forbidden Forest. This place is filled by the Darkness"
- **Les Cascades de Diamants** : "The Cascades of Diamonds, The water is so turquoise that I can't see the difference with the color of the diamond"
- **Salle secrète de la Cascade de Diamants** : "The Secret Place of the Cascade of Diamonds, Who could pretend that under the cascade it would be an door ?"

Items disponibles : [SOON]

Personnages :

- Le héro
- [SOON]

Situations gagnantes : Quand le héros tue le grand nécromancien ET libère la cité de son sortilège

Situations perdantes : Si le joueur meurt, si la phase principale du sortilège maléfique commence, si un sorcier touche le joueur avec une attaque "attrape rêve" qui l'enferme dans un rêve sans fin.

Partie exercices

Exercice 7.6

Dans `Room.java` Nous avons cette procédure:

```
/**
 * Permet de définir les différentes sorties de la salle instanciée
 *
 * @param northRoom Salle se trouvant au Nord de cette salle
 * @param southRoom Salle se trouvant au Sud de cette salle
 * @param eastRoom Salle se trouvant à l'Est de cette salle
 * @param westRoom Salle se trouvant à l'Ouest de cette salle
 */
public void setExits(final Room northRoom, final Room southRoom, final Room
eastRoom, final Room westRoom) {
    this.northRoom = northRoom;
    this.southRoom = southRoom;
    this.eastRoom = eastRoom;
    this.westRoom = westRoom;
}
```

Ajoutons quelques éléments pour augmenter notre nombre de sorties possibles:

```
public Room {

    // [...]

    public Room upRoom;
    public Room downRoom;

    // [...]
```

```
    public void setExits(final Room northRoom, final Room southRoom, final Room
eastRoom, final Room westRoom, final Room upRoom, final Room downRoom) {
        this.northRoom = northRoom;
        this.southRoom = southRoom;
        this.eastRoom = eastRoom;
        this.westRoom = westRoom;
        this.upRoom = upRoom;
        this.downRoom = downRoom;
    }

}
```

Explications :

Nous remarquons que nous avons une procédure qui à besoin maintenant de 6 paramètres, car ces 6 paramètres représentent les 6 sorties possibles. Notre code ne permet donc pas de dynamiser le nombre de sorties possible.

Et puisque nous avons 6 paramètre dans ce programme, nous devons forcément modifier notre procédure privée `createRooms()` dans le fichier `Game.java` puisque maintenant, il n'y a plus 4 sorties mais 6 sorties.

Dans la procédure `createRooms()` dans `Game` remplaçons les instructions de la manière suivante:

```
fortressEntrance.setExits(fortressYard, joyfulAvenue, null, null);
// maSortie     .setExits(Nord,         Sud,          Est,  Ouest);
```

par:

```
fortressEntrance.setExits(fortressYard, joyfulAvenue, null,   null,  null,
null);
// maSortie     .setExits(Nord,         Sud,          Est,    Ouest, Haut, Bas);
```

Exercice 7.7

Dans la classe `Room` créons une méthode `getExitString()`

```
/**
 * Permet de récupérer l'ensemble des sorties possible de la salle instanciée
 *
 * @return Ensemble des sorties possible
 */
public String getExitString() {
   String exits = "Available Exits : ";

   if(this.northExit != null) exits += "north ";
   if(this.southExit != null) exits += "south ";
   if(this.eastExit != null) exits += "east ";
   if(this.westExit != null) exits += "west ";

   // ainsi que les deux nouvelles sorties
   if(this.upExit != null) exits += "up ";
```

```
    if(this.downExit != null) exits += "down";

    return exits;
}
```

Exercice 7.8

Dans le fichier `Room.java` Nous pouvons ajouter/remplacer :

```
import java.util.HashMap;

public class Room {

    // [...]

    private final HashMap<String, Room> exits = new HashMap<String, Room>();

    /**
     * Setter qui permet de définir une salle en fonction de la direction de
sortie
     * Cette procédure n'est pas sensible à la casse !
     *
     * @param direction Direction de la sortie qu'on souhaite modifier
     * @param exitRoom Salle qui va être définie comme sortie à la direction
indiquée
     */
    public void setExit(final String direction, final Room exitRoom) {
        // Vérifie si la salle n'est pas nul, ça reviendrait à dire
        // qu'on a pas de salle en argument
        if(exitRoom != null)
            return this.exits.get(direction.toLowerCase());
    }

    /**
     * Getter qui récupère la salle d'une sortie en fonction de la direction
     * de sortie. Cette procédure n'est pas sensible à la casse !
     *
     * @param direction Direction de la sortie qu'on cherche
     * @return Salle se trouvant à la sortie recherchée
     */
    public Room getExit(final String direction) {
        return this.exits.get(direction.toLowerCase());
    }

    // [...]
}
```

Ce qui permet de rendre plus modulable nos sorties et ainsi permettre à nos salles de ne pas dépendre uniquement de 6 sorties mais en définir une infinité et les dynamiser (*à condition que la RAM suive aussi*)

Explications :

La fonction `toLowerCase()` permet de remplacer les caractères en majuscules dans la chaîne et les mettre en minuscule, permettant ainsi d'empêcher de créer deux directions qui sont similaires (*par exemple East, EaST et eAsT ici appartiendraient à la même direction : east*)

Exercice 7.8.1

Pour ajouter dans le scénario le déplacement vertical, nous avons ajouter les deux directions nécessaire : `up` et `down`

Pour les représenter, j'ai décidé de définir leur direction verticale, j'ai représenté leurs flèches sous forme de pointiés :

- **up** : flèche en pointillés qui commence vers la droite et pars vers la gauche OU commence vers le bas pour aller vers le haut (par rapport à la position relative de la salle)
- **down** : flèche en pointillés qui commence vers la gauche et pars vers la droite OU commencer vers le haut pour aller vers le bas (par rapport à la position relative de la salle)

Exercice 7.9

Maintenant que nous avons dynamiser l'ajout de sorties dans notre classe `Room` Nous devons maintenant récupérer les sorties disponibles de ces sorties

Dans la classe `Room` nous pouvons remplacer le contenu de la méthode `getExitString`:

```
/**
 * Getter qui récupère les sorties disponibles dans la salle instanciée
 *
 * @return Chaîne comportant la liste des sorties disponibles
 */
public String getExitString() {
    String availableExits = "Available Exits : ";
    for(String keys : this.exits.keySet())
        availableExits += keys + " ";
    return availableExits;
}
```

Explications :

`this.exists.keySet()` permet de récupérer un `Set<T>` (un ensemble) avec `T` le type des variables de référence en entrée du "dictionnaire". En effet, les HashMap sont un peu comme des dictionnaires (car clé => valeur). Ici la Map à une clé de type `String` (*la direction*) et renvoie une valeur de type `Room` (la salle en sortie) car nous avons écrit `HashMap<String, Room>`.

Exercice 7.10.1

Documentation Java complètée

Exercice 7.10.2

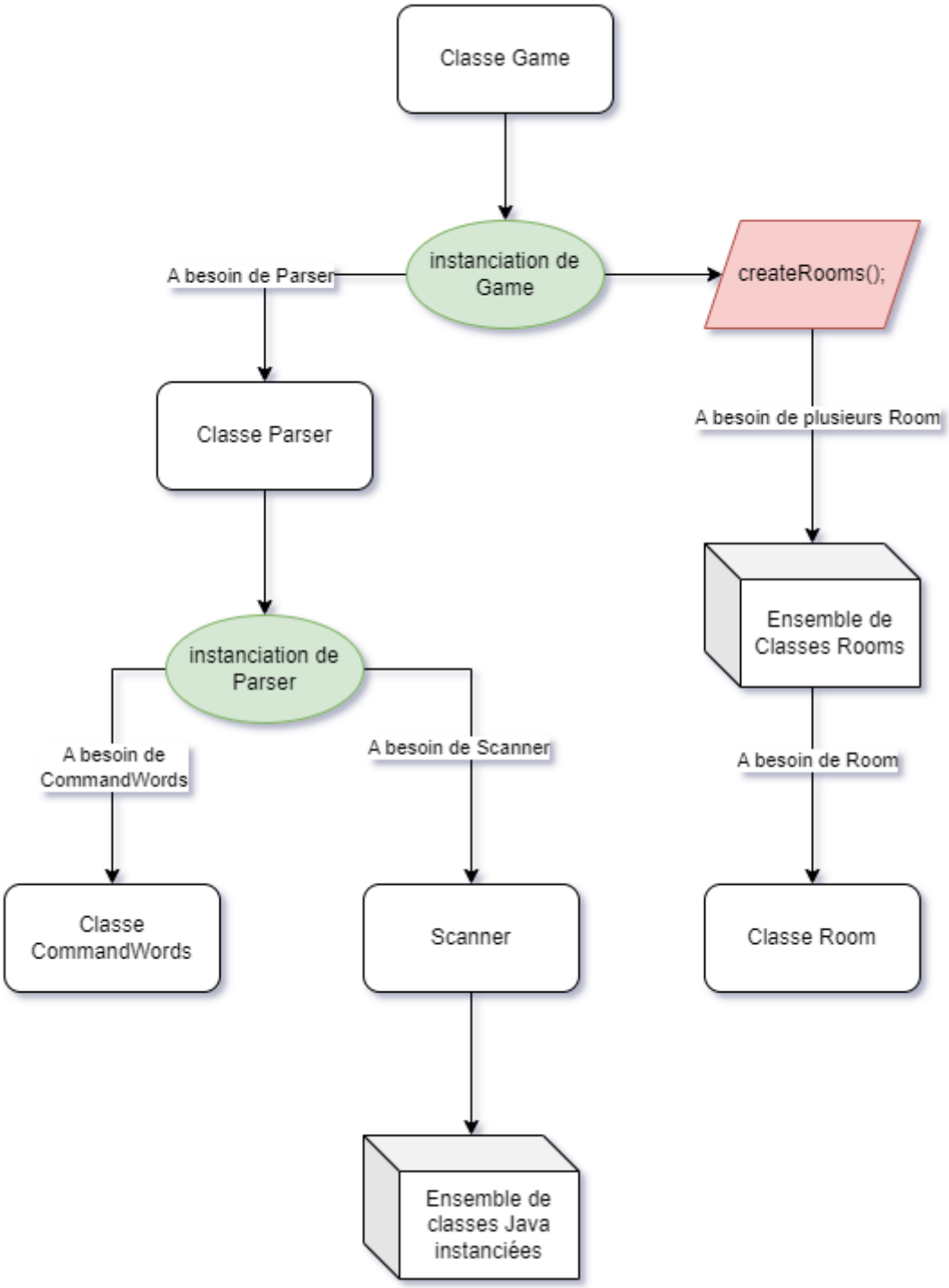
Documentation Java générée

Exercice 7.11

Dans notre classe `Room`, pour ajouter une description complète de notre salle, nous allons ajouter une méthode nommée `getLongDescription()` permettant de récupérer les informations de notre salle :

```
/**
 * Permet de récupérer l'intégralité des informations de notre salle:
 * La description de notre salle ainsi que les sorties disponible
 *
 * @return Description complète de la salle instanciée
 */
public String getLongDescription() {
    return "You are in " + this.getDescription() + "\n" + this.getExitString();
}
```

Exercice 7.12



Exercice 7.13

Comment change le diagramme des objets instanciés pendant au moment de l'exécution de la commande go ?

Lorsque nous regardons plus attentivement les étapes du programme quand on entre la commande "go" nous savons déjà que nous avons récupéré une nouvelle instance d'une commande (en effet, dans la classe `Parser` au moment où nous appelons `getCommand()`, nous observons à la fin de la méthode , un `return new Command(parametre1, parametre2);` ce qui montre bien que nous avons un nouvel objet de type `Command`.

qui dit commande "go" dit appel de la procédure `goRoom()` Nous pouvons observer que chaque fois que nous appelons `goRoom()`, nous définissons une nouvelle variable locale `vNextRoom` (celle-ci sera détruite une fois la méthode finie). Or tout ce que nous faisons c'est récupérer une salle qui se trouve stockée elle-même dans la salle où nous sommes actuellement.

Exercice 7.14

Dans la classe `Game` ajoutons une nouvelle procédure :

```
/**
 * Permet d'afficher la description complète de la salle dans laquelle nous
sommes actuellement
 */
private void look(Command command) {
    if(command.hasSecondWord())
        System.out.println("I don't know how to look at something in particular
yet");
    else
        System.out.println(this.currentRoom.getLongDescription());
}
```

Cette méthode, si nous observons bien, elle permet de récupérer la description complète de la salle dans laquelle nous somme et de nous l'afficher.

Et puisqu'il s'agit d'une nouvelle commande, il faut l'ajouter dans le tableau de commandes disponibles dans `CommandWords` et dans les commandes à vérifier dans la procédure `processCommand` dans la classe `Game`.

Nous avons donc comme procédure `processCommand`:

```
/**
 * Permet d'exécuter une commande saisir par l'utilisateur
 *
 * @param command Commande que l'utilisateur a écrit
 * @return Si la commande saisie arrête le jeu
 */
private boolean processCommand(final Command command)
{
    if (command.isUnknown()) {
        System.out.println("I don't know what you mean ...");
        return false;
    }

    switch(command.getCommandWord().toLowerCase()) {
        case "go":
            goRoom(command);
            return false;
        case "quit":
            return this.quit(command);
        case "help":
            printHelp();
            return false;
        case "look":
            look(command);
```

```
            return false;
        default:
            System.out.println("Unknown command !");
            return false;
    }
}
```

Exercice 7.15

Faisons la même chose pour la procédure `eat()`, ajoutons là dans la liste dans `CommandWords` et dans la méthode `processCommand()` :

Ecrivons la procédure :

```
/**
 * Permet d'afficher qu'on a mangé
 */
private void eat() {
    System.out.println();
}
```

Dans la méthode `processCommand()` :

```
/**
 * Permet d'exécuter une commande saisir par l'utilisateur
 *
 * @param command Commande que l'utilisateur a écrit
 * @return Si la commande saisie arrête le jeu
 */
private boolean processCommand(final Command command)
{
    if (command.isUnknown()) {
        System.out.println("I don't know what you mean ...");
        return false;
    }

    switch(command.getCommandWord().toLowerCase()) {
        case "go":
            goRoom(command);
            return false;
        case "quit":
            return this.quit(command);
        case "help":
            printHelp();
            return false;
        case "look":
            look(command);
            return false;
        case "eat":
            eat();
            return false;
        default:
            System.out.println("Unknown command !");
            return false;
```

```
    }
}
```

Pour dynamiser et rendre plus accessibles nos commandes, nous pouvons afficher l'ensemble des commandes sous forme de boucle.

Puisque les commandes enregistrées se trouvent dans la classe, nous allons l'écrire dans la classe `CommandWords` :

```
/**
 * Permet d'afficher l'ensemble des commandes enregistrées.
 */
public void showAll() {
    for(String command : this.registeredCommands) {
        System.out.print(command + " ");
    }
    System.out.println("");
}
```

Faire la liaison de cette méthode dans `Parser`

```
/**
 * Permet d'afficher les commandes disponibles
 */
public void showCommands() {
   this.commands.showAll();
}
```

Et appeler cette procédure dans `printHelp()` dans `Game`

```
/**
 * Permet d'afficher la liste d'aide pour le joueur
 */
private void printHelp() {
    System.out.println("Available Commands : ");
    this.parser.showCommands();
}
```

Explications : la boucle `foreach` est de la forme `for(T type : Collection<T>)` ou de la forme `for(T type : T[])` car il faut noter qu'une classe qui implémente l'interface générique `Collection` n'est pas un tableau de `T` mais peut représenter un tableau de `T`.

Exercice 7.17

Lorsque nous ajouterons une nouvelle commande, ce ne sera plus à `Game` de les gérer mais bel et bien à `CommandWords` qui enregistrera toutes les commandes possibles.

Donc quand nous ajouterons une nouvelle commande, nous écrirons dans la classe `CommandWords` dans son constructeur :

`this.registeredCommands[X] = "ma_nouvelle_commande";`

avec `X` qui correspond à l'indice suivant le dernier a être utilisé pour stocker une commande.

Exercice 7.18

Pour ajouter la liste des commandes, nous devons tout d'abord placer le code au bon endroit, ici, puisqu'il s'agit de récupérer des commandes, le mieux c'est de l'intégrer dans le fichier `CommandWords.java` où se trouve la classe `CommandWords`.

```
/**
 * Permet de récupérer l'ensemble des commandes disponibles dans le jeu
 * @return Commandes disponibles
 */
public String getCommandList() {
    String commands = "";
    for(String command : registeredCommands) {
        commands += command + " ";
    }
    return commands;
}
```

On se permet de modifier aussi dans le fichier `Parser`. On remplace :

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

Par

```
public CommandWords getCommandWords() {
    return this.commands;
}
```

Et dans `Game` on modifie :

```
/**
 * Affiche les commandes possibles
 */
private void printHelp()
{
    System.out.println("Your commands are :");
    // On commente cette instruction : parser.showCommands();
    System.out.println(parser.getCommandWords().getCommandList()); // On ajoute
celle-ci
}
```

Exercice 7.18.1

Les différences qu'il y a entre `zuul-better` et le `zuul-bad` sont multiples :

- Dans la classe `Room`, il y a un nouveau paramètre demandé: `String imageName` s'agit certainement d'une image qu'on va devoir charger pour représenter la salle instanciée. Il y a

aussi le getter `String getImageName()`

- Dans la classe `Room` on a accès à présent d'un nouveau getter `String getShortDescription()` qui permet probablement, comme son nom l'indique, de récupérer une petite description de la salle.
- Dans la classe `Parser`, nous n'utilisons plus l'interface `Scanner` mais un `StringTokenizer` c'est à dire qu'on aura, probablement, plus à taper sur le clavier pour entrer les différentes commandes.
- On remarque aussi que toutes les sorties (affichage de messages) c'est à dire les appels de procédure `System.out.println(msg);` sont désormais utilisées que dans la classe `Game`. Il s'agit donc d'un refactorisation des appels de différentes fonctions.

Exercice 7.18.2

Le `StringBuilder` est une classe qui est un utilitaire pour les chaînes de caractères, un peu comme `Arrays` pour les listes.
`StringBuilder` a donc, logiquement, plusieurs fonctionnalités. Nous pouvons par exemple modifier l'exercice 7.18 en ajoutant les `StringBuilder` :

On importe avec `import java.util.StringBuilder;`.

```
/**
 * Permet de récupérer l'ensemble des commandes disponibles dans le jeu
 * @return Commandes disponibles
 */
public String getCommandList() {
    StringBuilder strBuilder = new StringBuilder();
    for(String command : registeredCommands) {
        strBuilder.append(command);
        strBuilder.append(" ");
    }
    return strBuilder.toString().trim();
}
```

Exercice 7.18.3

Exercice 7.18.4

Exercice 7.18.5 OPTIONNEL TEMPORAIREMENT

Exercice 7.18.6

Les classes commençant par `J` (6) :

- `javax.swing.JFrame` correspond au cadre de la fenêtre, la barre de menu, le titre, la taille de la fenêtre, etc...
- `javax.swing.JTextField` correspond à une entrée où l'utilisateur pourra envoyer des informations au programme, un peu comme un `<input>` en HTML, dans notre situation il s'agit d'un champ pour les commandes.
- `javax.swing.JTextArea` correspond à une large zone de texte, éditable ou non. Ici, cela correspond à l'affichage de la console, les informations des différentes sorties des commandes du jeu

- `javax.swing.JLabel` correspond à un label, un objet, mais qui ne peut pas être modifiable par l'utilisateur. Ici cela correspondra à un endroit pour afficher nos images
- `javax.swing.JPanel` correspond à l'interface qui permettra de récupérer les évènements que le joueur aura appelé. Le `JPanel` peut être comparé au contenu du `JFrame` c'est à dire l'intérieur du cadre de notre fenêtre.
- `javax.swing.JScrollPane` correspond à la barre qui permet de faire défiler une zone qui possède un gros contenu. Ici ca correspondra à l'historique des entrées / sorties des commandes de l'utilisateur.

Exercice 7.18.7

`addActionListener();` est une fonction appartenant à l'interface `Component` qui permet d'ajouter un composant qui hérite de `Component` dans une liste d'éléments pouvant être appelé lors d'une action de l'utilisateur (touche tapée, clic sur un élément de l'interface, etc...) et donc d'appelé les éléments enregistrés selon les différents évènements.

Exercice 7.18.8

Pour ajouter le bouton Help nous devons importer la classe `javax.swing.JButton` qui permet d'utiliser les bouton dans notre interface et ajoutons comme attribut `private JButton helpButton;`

dans la fonction `createGUI()` dans la classe `UserInterface` ajoutons

```
private void createGUI() {
    // [...]
    this.aImage = new JLabel();

    // ajoutons ceci
    this.helpButton = new JButton("Help");

    JPanel vPanel = new JPanel();
    // [...]
    this.aMyFrame.getContentPane().add( vPanel, BorderLayout.CENTER );

    // Et ceci aussi
    this.helpButton.addActionListener(this);

    this.aEntryField.addActionListener(this);
    // [...]
    this.aEntryField.requestFocus();
}
```

Et modifions la fonction `actionPerformed()` :

```
/**
 * Actionlistener interface for entries.
 */
@Override public void actionPerformed( final ActionEvent pE )
{
    if(pE.getSource() == this.aEntryField) {
        this.processCommand();
```

```
    }
    else if(pE.getSource() == this.helpButton) {
        if(pE.getActionCommand().equalsIgnoreCase("help")) {
            this.aEngine.interpretCommand("help");
        }
    }
}
```

Exercice 7.20

Pour ajouter un item nous devons d'abord créer une classe Item :

`Item.java` :

```
/**
 * Décrivez votre classe Item ici.
 *
 * @author Gaillet Leo
 * @version 23/04/16
 */
public class Item
{
    private final String name;
    private final String description;
    private final int weight;

    /**
     * Constructeur d'objets de classe Item
     */
    public Item(final String name, final String description, final int weight) {
        this.name = name;
        this.description = description;
        this.weight = weight;
    }

    public String getName() {
        return this.name;
    }

    public String getDescription() {
        return this.description;
    }

    public int getWeight() {
        return this.weight;
    }
}
```

Ensuite dans `Room` ajoutons :

```
    private Item item;
```

```
    public Item getItem() {
        return this.item;
    }

    public void setItem(final Item item) {
        this.item = item;
    }

    public String getItemString() {
        return this.item == null ? "No item here" : "Available item: " +
this.item.getName();
    }
```

Et modifions la fonction `getLongDescription()` :

```
    public String getLongDescription() {
        //BEFORE:
        //return getDescription() + "\n" + getExitsString();
        //AFTER:
        return getDescription() + "\n" + getExitsString() + "\n" +
getItemString();
    }
```

Exercice 7.21

Pour avoir une description détaillée de l'item dans une salle, nous allons créer une nouvelle fonction dans la classe `Item` car il s'agit de la description détaillée de l'Item.

```
    public String getLongDescription() {
        return "Item name: " + this.name + " is described as \"" +
this.description + "\" and costs " + this.weight + " pounds.";
    }
```

Nous devons donc modifier `getItemString()` dans la classe `Room` :

```
    public String getItemString() {
        return this.item == null ? "No item here" : "Available item: " + /*
this.item.getName(); */ this.item.getLongDescription();
    }
```

Exercice 7.21.1

Pour faire correctement fonctionner la fonction `look()` dans `GameEngine` pour faire en sorte que si un second mot est donné, on puisse afficher l'item disponible dans la salle en question

```
/**
 * Permet d'afficher la description complete de la salle dans laquelle nous
sommes actuellement
 */
```

```
private void look(Command command) {
    if(command.hasSecondWord()) {
        String commandword = command.getSecondWord();
        if(this.currentRoom.getItem() != null) {

 if(this.currentRoom.getItem().getName().equalsIgnoreCase(commandWord)) {
                this.userInterface.println(this.currentRoom.getItemString());
            }
            else
                this.userInterface.println("Item not found");
        }
        else
            this.userInterface.println(this.currentRoom.getItemString());
    }
    else
        this.userInterface.println(this.currentRoom.getLongDescription());
}
```

Exercice 7.22

Nous allons effectuer quelques modifications dans la classe `Room`:

```
    /*
    Nous retirons l'attribut
    private Item item;

    pour le remplacer par:
    */
    private final HashMap<String, Item> items = new HashMap<String, Item>();

    public Item getItem(final String itemName) {
        return this.items.get(itemName);
    }

    /*
    On retire la fonction
    public void setItem(final Item item)
    */

    public void addItem(final Item item) {
        this.items.put(item.getName().toLowerCase(), item);
    }

    public String getItemString() {
        if(items.size() == 0)
            return "No items here";

        StringBuilder itemContent = new StringBuilder();
        for(Item item : items.values()) {
            itemContent.append("\"" + item.getName() + "\": ");
            itemContent.append("\t" + item.getLongDescription() + "\n");
        }
        return "Available items (" + items.size() + ") :";
    }
```

Et nous devons aussi modifier la fonction `look` dans la classe `GameEngine` :

```
/**
 * Permet d'afficher la description complete de la salle dans laquelle nous
sommes actuellement
 */
private void look(Command command) {
    if(command.hasSecondWord()) {
        String commandWord = command.getSecondWord();
        if(this.currentRoom.getItem(commandWord) != null)

 this.userInterface.println(this.currentRoom.getItem(commandWord).getLongDescrip
tion());
        else
            this.userInterface.println("Item not found");
    }
    else
        this.userInterface.println(this.currentRoom.getLongDescription());
}
```

Exercice 7.22.1

Pour avoir la possibilité d'ajouter plusieurs items dans la salle nous pouvons utiliser une `HashMap` ou une `ArrayList` selon la façon dont nous allons chercher les items dans une salle. Ici nous allons chercher les items selon leur nom qui a été défini, il est donc préférable d'utiliser une `HashMap<String, Item>` avec `String` qui correspond au type de la valeur que nous allons entrer dans la `HashMap` pour trouver l'`Item` que nous recherchons.

Exercice 7.22.2

Nous allons ajouter les items nécessaire à cet exercice (3 items minimum, dont 1 dans la salle de début, et 2 autres dans une seule et même salle).

Dans la classe `GameEngine` dans la fonction qui doit initialiser les différentes salles (`createRooms()`) nous allons ajouter les Items

```
private void createRooms() {
    // [...]

    // Défini la salle de départ (elle était déjà censé être là)
    this.currentRoom = fortressEntrance;

    // Définis les différents item dans le jeu
    Item fleur, chaise, epee;

    fleur = new Item("Fleur", "Il y en a une dans la salle de début", 1);
    chaise = new Item("Chaise", "Bon, même si elle ne sert à rien, il faut quand
même la mettre", 5);
    epee = new Item("Epee", "Qui dit, monde fantastique, dit aussi, épée
stylée", 2);
    this.currentRoom.addItem(fleur);

    fortressDungeon.addItem(chaise);
```

```
    fortressDungeon.addItem(epee);
}
```

Exercice 7.23

Pour ajouter la commande `back` modifions d'abord la fonction `interpretCommand()` dans `GameEngine` :

```
public void interpretCommand(final String rawCommand) {
    this.userInterface.println( "> " + rawCommand );
    Command command = this.parser.getCommand( rawCommand.toLowerCase() );

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

    switch(command.getCommandWord().toLowerCase()) {
        case "go":
            goRoom(command);
            return;
        case "quit":
            quit(command);
            return;
        case "help":
            printHelp();
            return;
        case "look":
            look(command);
            return;
        case "eat":
            eat();
            return;
        // Ajoutons-la ici
        case "back":
            back(command);
            return;
        default:
            this.userInterface.println("Unknown command !");
            return;
    }
}

private Room previousRoom;

public void back(final Command command) {
    if(command.hasSecondWord()) {
        this.userInterface.println("Back what ?");
        return;
    }

    if(this.previousRoom == null) {
        this.userInterface.println("There is any previous room !");
        return;
```

```
    }

    final Room alternativeRoom = this.previousRoom;
    this.previousRoom = this.currentRoom;
    this.currentRoom = alternativeRoom;

    printLocationInfo();
}
```

et dans `goRoom` nous devons ajouter à la toute fin de la fonction :

```
private void goRoom(final Command command) {
    if (!command.hasSecondWord()){
        this.userInterface.println("Go where ?");
        return;
    }

    Room nextRoom = this.currentRoom.getExit(command.getSecondWord());

    if (nextRoom == null){
        this.userInterface.println("There is no door !");
        return;
    }

    // Ajoutons ceci
    this.previousRoom = this.currentRoom;
    this.currentRoom = nextRoom;

    printLocationInfo();
}
```

Exercice 7.26

Pour retourner sur les anciennes salles que nous avons traversé, nous devons faire une Pile (FILO - First In Last Out) pour pouvoir accéder aux anciennes salles au fur et à mesure que nous avançons.

Pour cela nous allons modifier `goRoom` dans la classe `GameEngine` ainsi que l'attribut par défaut :

```
/*
    Remplaçons
    private Room previousRoom;

    par
 */
    private final Stack<Room> previousRooms = new Stack<Room>();

private void goRoom(final Command command) {
    if (!command.hasSecondWord()){
        this.userInterface.println("Go where ?");
        return;
    }
```

```
    Room nextRoom = this.currentRoom.getExit(command.getSecondWord());

    if (nextRoom == null){
        this.userInterface.println("There is no door !");
        return;
    }

    /*
     Remplaçons
     this.previousRoom = this.currentRoom;

     par:
    */
    this.previousRooms.push(this.currentRoom);

    this.currentRoom = nextRoom;

    printLocationInfo();
}
```

De plus nous devons modifier la fonction `back()` :

```
public void back(final Command command) {
    if(command.hasSecondWord()) {
        this.userInterface.println("Back what ?");
        return;
    }

    if(this.previousRooms.size() == 0 || this.previousRooms.peek() == null) {
        this.userInterface.println("There is any previous room !");
        return;
    }

    this.currentRoom = this.previousRooms.pop(); // On modifie ça
    printLocationInfo();
}
```

Exercice 7.26.1

Pour générer les deux Javadocs, nous avons deja besoin de savoir la différence entre ces deux là

- La javadoc dans `userdoc` fournie par la commande "*javadoc -d userdoc -author -version *.java*" est celle qui est utile aux personnes de l'exterieur qui souhaite comprendre le comportement du programme
- La javadoc dans `progdoc` fournie par la commande "*javadoc -d progdoc -author -version -private -linksource *.java*" est celle qui est utile au programmeur et aux développeurs du projet et permet ainsi de ne pas se perdre dans l'ensemble des fonctions créée.

Exercice 7.28.1

Pour créer une nouvelle commande `test` nous avons d'abord besoin d'intégrer `"test"` dans notre liste de commandes existantes (définies dans la classe `CommandWords`) :

```
public class CommandWords
{
    private final String[] registeredCommands;

    public CommandWords() {
        this.registeredCommands = new String[]  {
            "go", "help", "quit", "look", "eat", "back", "test" // <--- On
l'ajoute juste là
        };
    }

    // [...]
}
```

Ensuite nous devons l'ajouter dans le multi-bloc conditionnel (les `if`, `else if` *en boucle*) / `switch` dans la fonction `interpretCommand` dans `GameEngine`:

```
public void interpretCommand(final String rawCommand) {
    this.userInterface.println( "> " + rawCommand );
    Command command = this.parser.getCommand( rawCommand.toLowerCase() );

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

    switch(command.getCommandWord().toLowerCase()) {
        case "go":
            goRoom(command);
            return;
        case "quit":
            quit(command);
            return;
        case "help":
            printHelp();
            return;
        case "look":
            look(command);
            return;
        case "eat":
            eat();
            return;
        case "back":
            back(command);
            return;
        // Nous l'ajoutons juste là
        case "test":
            test(command);
            return;
```

```
        default:
            this.userInterface.println("Unknown command !");
            return;
    }
}
```

Or pour que cela fonctionne, nous devons créer une nouvelle fonction (dans `GameEngine` aussi) qui permettra de faire nos différents `tests` demandés :

```
public void test(final Command command) {

    // Vérifie si la commande n'a pas de second mot
    if(!command.hasSecondWord()) {
        this.userInterface.println("You need to choose a file to test commands
!");
        return;
    }

    // Dans le cas où la commande à un second mot, on poursuit directement ici

    // Défini un nouveau fichier
    File file = new File(command.getSecondWord() + ".txt");

    // Vérifie si le fichier n'existe pas
    if(!file.exists()) {
        this.userInterface.println("file named " + file.getName() + " is not
found !");
        return;
    }

    // Dans le cas où le fichier existe, on continue ici

    try {
        // Scanne chaque lignes du fichier et les exécute
        Scanner commandScanner = new Scanner(file);

        while(commandScanner.hasNextLine()) {
            String rawCommand = commandScanner.nextLine();
            this.interpretCommand(rawCommand);
        }
    }
    catch (FileNotFoundException e) {
        // Préviens le joueur que une erreur est survenue au lieu d'afficher une
exception
        this.userInterface.println("Unable to read file " + file.getName() +
"!");
        return;
    }
}
```

Exercice 7.28.2

Pour créer nos fichier de test de commandes, nous devons aller dans le dossier de notre projet et créer les fichiers nous même, `court.txt`, `visite.txt` et `ideal.txt`, ensuite rouvrir BlueJ, et normalement, nous sommes censé avoir les 3 fichiers automatiquement implémentés dans notre projet.

Exercice 7.29

Pour Refactoring notre classe `GameEngine` en deux classes distinctes, nous devons comprendre tout d'abord quelle fonction doit aller dans laquelle des deux classes, soit celle de `GameEngine` qui comporte les éventuelles actions et commande effectuées lors de la partie, et une autre, `Player` qui contient les différentes variables qui vont influencer le joueur.

Par exemple on peut noter que la fonction `goRoom(final Room room)` prend en compte un système d'affichage, un système de déplacement du joueur et un autre système d'affichage

```
private void goRoom(final Command command) {

    // Condition qui vérifie la commande POUR L'AFFICHAGE
    if (!command.hasSecondWord()){
        this.userInterface.println("Go where ?");
        return;
    }

    Room nextRoom = this.currentRoom.getExit(command.getSecondWord());

    // Conditin qui vérifie si la salle récupérer est nulle POUR L'AFFICHAGE
    if (nextRoom == null){
        this.userInterface.println("There is no door !");
        return;
    }

    // SYSTEME QUI DEPLACE LE JOUEUR et enregistre la position précédente
    this.previousRooms.push(this.currentRoom);
    this.currentRoom = nextRoom;

    // AFFICHE les informations de la position
    printLocationInfo();
}
```

Or tout ce bloc de code n'a pas forcément besoin d'être déplacé dans la classe `Player`. En effet, la classe va stocker uniquement ce qu'a besoin le joueur, c'est à dire la position actuelle `currentRoom`, ses position précédentes `previousRooms`, son nom `nom`, et peut être d'autres attributs qui dépendront du joueur. Il faut penser qu'il peut y avoir plusieurs joueurs, par exemple, en multijoueur, les joueurs auront le même affichage dans `GameEngine` mais n'auront pas le même contenu dans leurs variables, deux joueurs différents peuvent être dans deux salles différentes.

C'est donc pour cela que nous allons faire ceci :

```
public class GameEngine {
```

```
    /*
    Nous allons supprimer:

    private Room currentRoom;
    private Stack<Room> previousRooms;

    ---------

    Pour ajouter:
    */

    private final Player player;

    // [...]

    public GameEngine() {
        // [...]
        this.player = new Player("Nom de mon joueur");
        // [...]
        this.createRooms();
    }
}
```

Comme vous pouvez le voir, on initialise `this.player` avant d'appeler la procédure `createRooms();`. En effet, `createRooms` aura besoin de l'attribut `player` pour définir sa salle par défaut. On aurait pu directement définir `this.player = new Player("nom");` directement dans la fonction `createRooms()` Or, par mesure de convention et d'organisation de code, il est toujours préférable d'initialiser les variables dans le constructeur.

Dans cette situation on se retrouve quand même face à plusieurs erreurs : puisque les attributs `currentRoom`, `previousRooms` ont été déplacées et que la classe `Player` n'existe pas. Nous allons y remédier, tout d'abord en créant la classe `Player` :

```
import java.util.Stack;

/**
 * Les differents attributs et fonctions liées au Joueur, permettant
 * ainsi de se déplacer, de connaitre sa salle actuelle, de savoir ce qu'il a
sur lui
 * de savoir ses salles précédentes. Tout ce qui est lié ou influence sur le
Joueur.
 *
 * @author Leo Gaillet
 * @version 19/04/2023
 */
public class Player {

    private final String name;
    private Room currentRoom;
    private Stack<Room> previousRooms;

    /**
     * Constructeur de la classe Player
     * @param name Nom du joueur dans le jeu
```

```
     */
    public Player(final String name){
        this.name = name;
        this.previousRooms = new Stack<Room>();
    }

    /**
     * Permet de récupérer le nom du joueur
     * @return Nom du joueur
     */
    public String getName() {
        return this.name;
    }

    /**
     * Permet de récupérer la salle dans laquelle le joueur se trouve
actuellement
     * @return Salle du joueur actuelle
     */
    public Room getCurrentRoom() {
        return this.currentRoom;
    }

    /**
     * Permet de savoir si le joueur est déjà allé dans d'autres salles
précédentes
     * @return Vérifie si le joueur peut retourner en arrière
     */
    public boolean hasPreviousRoom() {
        return this.previousRooms.size() != 0;
    }

    /**
     * Permet de déplacer le joueur dans une nouvelle salle
     * @param room Nouvelle salle où le joueur sera déplacé
     * @param saveCurrentToPreviousRooms Autorise la sauvegarde de l'ancienne
salle dans la liste des salles précédentes
     */
    public void setCurrentRoom(final Room room, final boolean
saveCurrentToPreviousRooms) {
        if(saveCurrentToPreviousRooms) {
            this.previousRooms.push(this.currentRoom);
        }
        this.currentRoom = room;
    }

    /**
     * Permet de déplacer le joueur dans son ancienne salle
     * @return Salle où le joueur compte se déplacer
     */
    public Room goPreviousRoom() {
        Room previousRoom = this.previousRooms.pop();
        if(previousRoom != null)
            this.setCurrentRoom(previousRoom, false);
        return previousRoom;
```

```
    }

    public String lookForItem(final String itemName) {
        return currentRoom.getItem(itemName) != null ?
currentRoom.getItem(itemName).getLongDescription() : "Item not found";
    }

}
```

Et grâce à cette classe, nous pouvons modifier quasiment tous les `this.currentRoom` dans la classe `GameEngine` qui provoquent des erreurs par des `this.player.getCurrentRoom()`. Il y aura malgré ça quelques exceptions, comme dans la commande `back()` :

Remplaçons cet ancienne version de `back` :

```
public void back(final Command command) {
    if(command.hasSecondWord()) {
        this.userInterface.println("Back what ?");
        return;
    }

    if(this.previousRooms.size() == 0 || this.previousRooms.peek() == null) {
        this.userInterface.println("There is any previous room !");
        return;
    }

    this.currentRoom = this.previousRooms.pop();
    printLocationInfo();
}
```

Par :

```
public void back(final Command command) {
    if(command.hasSecondWord()) {
        this.userInterface.println("Back what ?");
        return;
    }

    if(this.player.hasPreviousRoom()) {
        this.userInterface.println("There is any previous room !");
        return;
    }

    this.player.goPreviousRoom();
    printLocationInfo();
}
```

Ainsi que l'ancienne version de la commande `goRoom` :

```
private void goRoom(final Command command) {
    if (!command.hasSecondWord()){
        this.userInterface.println("Go where ?");
        return;
```

```
    }

    Room nextRoom =
this.player.getCurrentRoom().getExit(command.getSecondWord());

    if (nextRoom == null){
        this.userInterface.println("There is no door !");
        return;
    }

    this.player.setCurrentRoom(nextRoom, true);
    /*
        Le deuxième argument dans la fonction (le 'true') ici, permet d'indiquer
        à la fonction `setCurrentRoom` dans la classe Player de sauvegarder la
derniere position
        du joueur pour qu'il puisse revenir en arrière.
     */
}
```

Et finalement pour la fonction `look` :

Exercice 7.30

Pour créer les nouvelles commandes `take` et `drop` pour que le joueur puisse prendre des items présents ou lâcher des items qui lui appartiennent dans la salle, nous avons tout d'abord besoin de les créer :

Comme à son habitude, nous devons aller dans la classe `CommandWords` pour définir nos nouvelles commandes :

```
public class CommandWords
{
    private final String[] registeredCommands;

    public CommandWords() {
        this.registeredCommands = new String[]  {
            "go", "help", "quit", "look", "eat", "back", "test", "take", "drop"
        };
    }

    // [...]
}
```

Ensuite, ajoutons ces deux nouvelles commande dans le bloc principale des commande à vérifier dans la fonction `interpretCommand()` :

```
public void interpretCommand(final String rawCommand) {
    // [...]

    switch(command.getCommandWord().toLowerCase()) {

         // [...]
```

```
        case "take":
            take(command);
            return;
        case "drop":
            drop(command);
            return;

         // [...]

        default:
            this.userInterface.println("Unknown command !");
            return;
    }
}
```

Maintenant, ne reste plus qu'à créer les deux commandes, la première qui sera liée à l'affichage (du coup dans la classe `GameEngine`) et la seconde partie de la commande (qui va s'occuper des contrôles du déplacement de l'item) dans la classe `Player`.

Donc dans la classe `Player` nous ajoutons ceci :

```
public String takeItem(final String itemName) {
    if(this.item != null)
        return "You already have an item on your player !";

    Item foundItem = currentRoom.getItem(itemName);
    if(foundItem == null) {
        return "Item in this room is not found !";
    }

    this.item = foundItem;
    this.currentRoom.removeItem(foundItem);
    return "You took the item " + foundItem.getName() + " !";
}

public String dropItem() {
    if(this.item == null)
        return "You have any item on your player !";

    Item item = this.item;
    this.currentRoom.addItem(this.item);
    this.item = null;
    return "You dropeed the item " + item.getName() + " in the room !";
}
```

Attention ! N'oublions pas d'ajouter un attribut de type `Item` dans la classe `Player`:

```
private Item item;
```

Et dans la classe `GameEngine` nous ajoutons ces deux procédures là :

```
/**
 * Permet d'afficher la réponse du jeu quand le joueur veut récupérer un objet
dans une pièce
```

```
     * @param command Commande "take" au moment de l'exécution
     */
    public void take(final Command command) {
        if(!command.hasSecondWord()) {
            this.userInterface.println("Take what ?");
            return;
        }
        this.userInterface.println(this.player.takeItem(command.getSecondWord()));
    }

    /**
     * Permet d'afficher la réponse du jeu quand le joueur veut déposer un objet
    dans la pièce
     * @param command Commande "drop" que le joueur a saisi
     */
    public void drop(final Command command) {
        if(command.hasSecondWord()) {
            this.userInterface.println("Wrong prompt, there is any argument !");
            return;
        }
        this.userInterface.println(this.player.dropItem());
    }
```

Exercice 7.31

La différence entre cet exercice et le précédent, c'est que dans cette situation, nous pouvons prendre plusieurs objet sur nous, ce qui va modifier pas mal notre programme :

Tout vas quasiment se passer dans la classe `Player`, en effet, le traitement se passe sur le joueur, car c'est lui qui fait l'action, donc il s'agit de la classe `Player` qui va être modifiée.

Tout d'abord retirons cet attribut

```
private Item item;
```

par:

```
private final HashMap<String, Item> items = new HashMap<String, Item>();
// Sans oublier d'importer avec "import java.util.HashMap";
```

et remplaçons nos deux fonctions `takeItem` et `dropItem` dans `Player` par ça :

```
public String takeItem(final String itemName) {
    Item foundItem = currentRoom.getItem(itemName);
    if(foundItem == null) {
        return "Item in this room is not found !";
    }

    this.items.put(foundItem.getName().toLowerCase(), foundItem);
    this.currentRoom.removeItem(foundItem);
    return "You took the item " + foundItem.getName() + " !";
}
```

```java
public String dropItem(final String itemName) {
    if(!this.items.containsKey(itemName.toLowerCase()))
        return "You have any item named like this on your player !";

    Item item = this.items.get(itemName.toLowerCase());
    this.currentRoom.addItem(item);
    this.items.remove(itemName.toLowerCase());
    return "You dropeed the item " + item.getName() + " in the room !";
}
```

Et puisque nous avons modifié la signature de `public String dropItem()` par `public String dropItem(final String itemName)` Nous devons aller dans la classe `GameEngine` pour effectuer quelques modifications (car la procédure `drop` qui se trouve dans `GameEngine` appelle la fonction `dropItem` dans la classe `Player`) :

Dans la classe `GameEngine` modifions alors `drop` :

```java
public void drop(final Command command) {
    // Nous avons inversé la condition (avec le point d'exclamation '!')
    if(!command.hasSecondWord()) {
        this.userInterface.println("Drop what ?");
        return;
    }
    // De :
    // this.userInterface.println(this.player.dropItem());
    // Par :
    this.userInterface.println(this.player.dropItem(command.getSecondWord()));
}
```

Exercice 7.31.1

Comme il nous est demandé de créer une classe commune à Player et Room, c'est un peu comme si la classe `ItemList` était la représentation d'un inventaire, un pour le joueur, et un inventaire de la salle. On remarque depuis quelques exercices que la classe Player et la classe Room possèdent plusieurs points communs :

- Ils peuvent avoir des items dans une liste d'items
- On peut leur retirer des éléments de la liste d'items
- Les items ne doivent pas être accessibles de l'exterieur

Nous pouvons créer une classe `ItemList` en faisant :

```java
import java.util.HashMap;

/**
 * ItemList est une classe dédiée à la gestion d'une collection des
 * Item présent pour dans un joueur sous forme d'inventaire ou
 * sous forme d'un ensemble de fonctionnalités permettant
 * de manipuler les Items qui se trouvent à l'interieur de celle-ci
 *
 * @author Leo GAILLET
 * @version 20/04/2023
 */
```

```java
public class ItemList {

    private final HashMap<String, Item> items = new HashMap<String, Item>();

    public Item getItem(final String itemName) {
        return items.get(itemName.toLowerCase());
    }

    public void addItem(final Item item) {
        this.items.put(item.getName().toLowerCase(), item);
    }

    public void removeItem(final Item item) {
        this.items.remove(item.getName().toLowerCase());
    }

    public boolean hasItem(final String itemName) {
        return this.items.containsKey(itemName.toLowerCase());
    }

    public boolean isEmpty() {
        return getSize() == 0;
    }

    public int getSize() {
        return this.items.size();
    }

    public Item[] getContent() {
        Item[] availableItems = new Item[items.size()];
        int i = 0;
        for(Item item : items.values()) {
            availableItems[i] = item;
            i++;
        }
        return availableItems;
    }

}
```