ESIEE Paris

Rapport projet Zuul

Corentin POUPRY

supervisé par
Denis Bureau

2020 - 2021

Table des matières

1 Présentation

1.1 Auteur

Corentin POUPRY, étudiant à l'ESIEE Paris en E1, promotion 2025.

1.2 Thème

Murphy Law, un détective, doit faire la lumière sur l'enquête confiée.

1.3 Résumé du scénario

Vous vous attendiez à tomber sur un super jeu de science-fiction proposé par un étudiant talentueux. Cependant, la réalité est tout autre et vous vous retrouvez au bureau d'un curieux détective. Ce détective, bien décidé à vous aider à faire la lumière sur votre cas atypique, c'est Murphy Law, et c'est lui qu'on appelle quand tout va mal.

1.4 Scénario détaillé

Le Joueur (notons la majuscule) est un personnage à part entière de l'histoire, bien que l'utilisateur joue au travers de Murphy Law. Le Joueur apparaît au début de la narration complètement perdu et à la recherche du jeu de science-fiction promis par le talentueux étudiant dans son rapport. Murphy Law, détective, se demande par quel moyen Le Joueur a pu arriver dans son agence alors que, manifestement, il ne fait même pas partie du schéma narratif du jeu. Quelque chose cloche, quelque chose ne tourne pas rond.

Murphy Law décide de partir mener l'enquête en allant voir une source pouvant l'aider dans cette enquête. Avec Le Joueur, il monte dans sa voiture (voir Plan), cependant, la structure du jeu commence à se corrompre, à changer dangereusement sans raison, provoquant la stupéfaction chez les deux protagonistes. Murphy accélère pour semer les incohérences de narration. Alors qu'ils roulent vers leur contact à toute allure, Murphy commence à perdre le contrôle de la situation jusqu'à qu'un arbre apparaisse devant la voiture provoquant un accident.

Murphy Law et Le Joueur se réveille dans un grand escalier avec des dorures et un dôme imposant surmontant la pièce. Après l'irruption d'un majordome disant qu'on les cherche partout, Murphy et Le Joueur réalisent qu'ils passé dans l'univers d'un autre jeu de la promotion, prenant place à Buckingham Palace. Très vite, Murphy Law et Le Joueur sont accostés par les employés du palais qui les accompagnent dans la salle de réception où il découvre le crise nationale qui frappe le palais : un corgi royal est manquant.

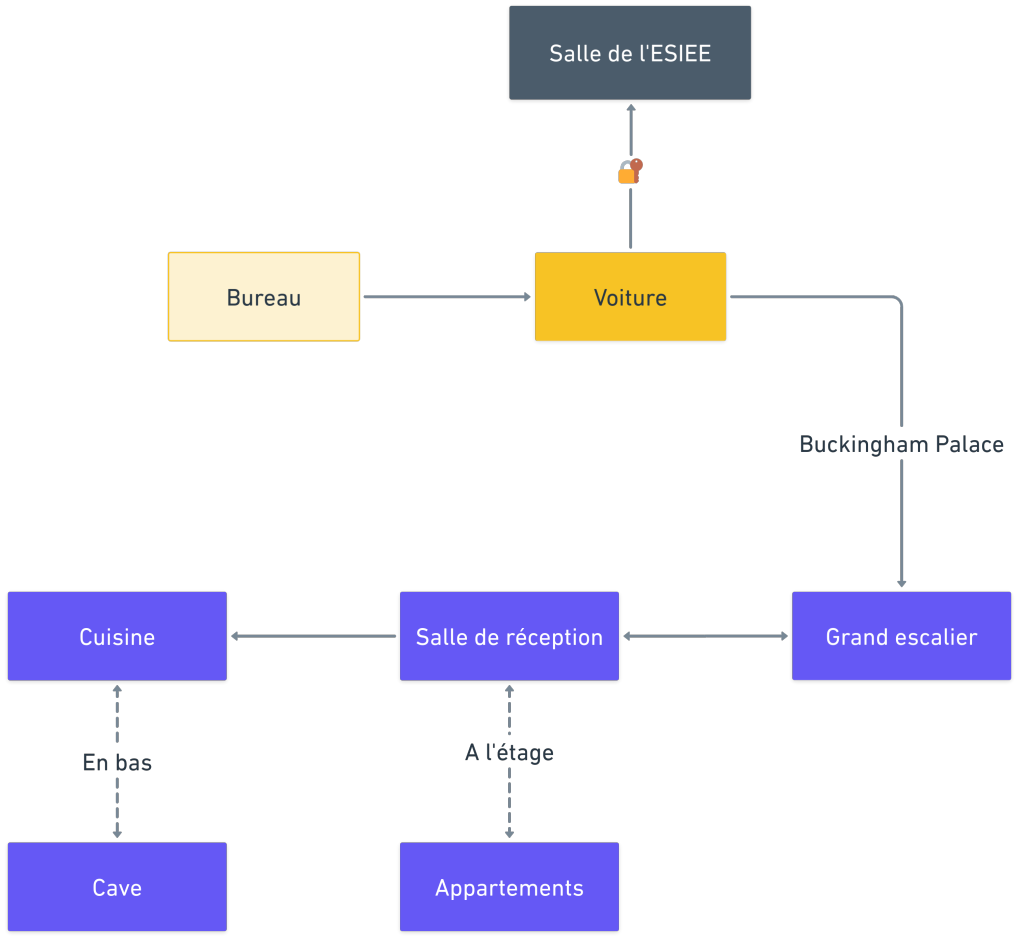
`TODO: pas encore totalement fini`.

Après avoir perdu le joueur et flairant que quelque chose se trame dans son dos, Murphy Law décide alors de se rendre là où tout a commencé : dans la salle de l'ESIEE où Le Joueur avait lancé le jeu. S'en suit une confrontation avec Le Compilateur, qui essayait de manipuler le jeu à sa guise pour que l'étudiant n'obtienne pas une note à la haute de sont travail. Murphy ressort gagnant de cette confrontation, ce qui lance le dialogue de fin de Planet Wars.

1.5 Commandes

go *direction*	Déplace le personnage vers une certaines pièce.
look	Affiche la position actuelle ainsi que la description de la pièce.
inspect *objet*	Permet d'inspecter un objet présent dans la pièce.
help	Affiche un bref résumé ainsi que la position actuelle.
quit	Quitte le jeu.

1.6 Plan



1.7 Détail des lieux, items, personnages

1.7.1 Lieux

Seuls les lieux importants à l'avancement et à l'histoire du jeu sont listés ci-dessous.

Bureau de Murphy C'est le lieu où vous commencez votre aventure.

Voiture	La voiture vous permet de rejoindre certains lieux.
Grand escalier	C'est l'endroit où vous arrivez à Buckingham Palace.
Salle de réception	Là où la crise du corgi royal sera expliquée.
Salle de l'ESIEE	C'est dans cette salle de l'ESIEE que vous avez lancé ce jeu.

1.7.2 Personnages

De la même façon, cette liste ne comprend que les personnages essentiels au scénario. Certains personnages non-joueurs ne sont pas listés ici.

Murphy Law	Le détective et aussi le personnage que vous incarnez. Son rôle est de mener l'enquête pour comprendre par quelles circonstances Le Joueur s'est retrouvé ici.
Le Narrateur	La voix que vous entendez quand vous jouez. Le Narrateur permet de décrire les scènes & situations sans sortir de la narration.
Le Joueur	C'est vous ! Cependant, Le Joueur est paradoxalement un personnage non jouable qui vous représente tout au long de l'histoire dans le cadre de la méta-narration.
Le Compilateur	Le compilateur Java est le grand méchant de l'histoire. Son rôle, manipuler la structure du jeu pour que l'étudiant n'ait pas une note à la hauteur de son travail.

1.8 Situations gagnantes et perdantes

Le Joueur et Murphy Law seront confrontés à des situations de crises où des choix et décisions devront être prises, certains pourront entraîner la mort ou l'immobilisation des protagonistes et donc faire gagner Le Compilateur.

La liste des situations perdantes est la suivante :

A Buckingham Palace
- Vous ne retrouvez pas le corgi royal.
- Vous essayez de fouiller la Reine d'Angleterre.

1.9 Commentaires

Murphy Law est une création originale de la Fondation SCP. Le personnage, les œuvres associées et ce jeu sont tous proposés sous licence *Creative Commons Attribution-ShareAlike 3.0*

2 Exercices

Exercice 7.5 *printLocationInfo*

En ré-usinant le code de l'affichage de la localisation et des sorties en une fonction, on s'assure de ne pas répéter cette partie à plusieurs endroits.

```
1  private void printLocationInfo() {
2      Room vCurrent = this.aCurrentRoom;
3      System.out.printf("You are %s%n",
       ↪  vCurrent.getDescription());
4
5      System.out.print("Exits: ");
6      if (vCurrent.aEastExit != null) System.out.print("east ");
7      if (vCurrent.aNorthExit != null) System.out.print("north
       ↪  ");
8      if (vCurrent.aSouthExit != null) System.out.print("south
       ↪  ");
9      if (vCurrent.aWestExit != null) System.out.print("west ");
10     System.out.println();
11 }
```

Exercice 7.6 *getExit*

Dans cet exercice, on se rend compte que le système "un attribut = une sortie" n'est pas le plus optimal (imaginons un hub ayant une dizaines de sorties par exemple, ce qui deviendrait problématique pour la lisibilité et la maintenance du code). L'idée étant de limiter le couplage entre la classe `Room` et les autres classes. Pour cela, on cherchera à limiter la dépendance aux attributs de Room et plutôt passer par une fonction pour récupérer les sorties, ce qui permet de ne gérer la logique des sorties que du côté de `Room`.

Notes : Notons qu'il reste un problème dans `printLocationInfo` à cause de ce changement, problème abordé dans les exercices qui suivent. On peut aussi émettre une critique quant à la façon de fonctionner de `getExit` : si jamais on passe `"North"` à la place de `"north"` par inadvertance, on aura comme valeur de retour `null`. Une solution serait d'utiliser des `enum`.

```
1  public Room getExit(final String pDirection) {
2      switch (pDirection) {
3          case "north":
4              return this.aNorthExit;
5
6          case "east":
7              return this.aEastExit;
8
9          case "south":
10             return this.aSouthExit;
11
12         case "west":
13             return this.aWestExit;
14
15         default:
16             return null;
17     }
18 }
```

Exercice 7.7 *getExitString*

Le dernier changement a impacté la façon dont `printLocationInfo` fonctionne. Pour le résoudre, on écrit une nouvelle méthode `getExitString` et, fort de ce changement, on ré-usine `printLocationInfo`.

```
1  public String getExitString() {
2      String vResult = "Exits: ";
3
4      if (this.aEastExit != null) vResult += "east ";
5      if (this.aNorthExit != null) vResult += "north ";
6      if (this.aSouthExit != null) vResult += "south ";
7      if (this.aWestExit != null) vResult += "west";
```

```
8
9       return vResult;
10  }
11
12  private void printLocationInfo() {
13      Room vCurrent = this.aCurrentRoom;
14
15      System.out.printf("You are %s%n",
        ↪  vCurrent.getDescription());
16      System.out.println(vCurrent.getExitString());
17  }
```

Exercice 7.8 *HashMap et setExit*

Maintenant que le couplage entre `Room` et `Game` est faible, on peut remplacer les détails de l'implémentation sans risquer de casser quelque chose. On utilise une structure de données `HashMap` et on doit changer les méthodes de `Room` en conséquence. On en profite aussi pour remplacer `setExits`, devenue inutile, par `setExit`.

```
1   public Room setExit(final String pDirection, final Room
    ↪  pExit) {
2       this.aExits.put(pDirection, pExit);
3
4       return this;
5   }
6
7   public Room getExit(final String pDirection) {
8       return this.aExits.get(pDirection);
9   }
```

Note : J'ai ajouté en plus de ce que l'exercice demandait la dernière ligne `return this;` pour pouvoir chaîner les appels de `setExit` comme ci-dessous :

```
office.setExit("east", car)
      .setExit("north", kitchen)
      .setExit("west", library);
```

Exercice 7.9 *keySet*

`getExitString` doit elle aussi être modifiée. Plutôt que de tester la présence de certaines valeurs dans la `HashMap` (du type "north", "east", "down", "up"...), on peut énumérer les différentes clés qui composent la `HashMap` des sorties.

```
public String getExitString() {
    String vResult = "Exits : ";
    for (String vExit : this.aExits.keySet())
        vResult += vExit + " ";

    return vResult;
}
```

Exercice 7.10 *Fonctionnement de keySet et Javadoc*

Reprenons le code intéressant de l'exercice précédent et intéressons-nous à son fonctionnement.

```
for (String vExit : this.aExits.keySet())
    vResult += vExit + " ";
```

Ce code marche car `keySet` renvoie un `Set` qui est énumérable et utilisable par la boucle for-each.

La différence entre un `Set` et une liste normale (un tableau) est que le `Set` ne peut pas contenir deux mêmes éléments.

Pour la Javadoc, la documentation de `Game` contient beaucoup moins de méthode que `Room` car l'encapsulation fait que seul `play` est publique pour `Game`, tandis que `Room` doit exposer beaucoup plus de méthodes `public` pour être utilisée par `Game`.

Exercice 7.11 *getLongDescription*

Toujours dans la logique de la conception dirigée par responsabilités, on déplace la création de la description dans la classe `Room`.

```
1 public String getLongDescription() {
2     return "Vous êtes actuellement dans la salle \"" +
3             this.aName + "\".\n" +
4             this.aDescription + ".\n" +
5             this.getExitString();
6 }
```

Et on change `Game` en conséquence.

```
1 private void printLocationInfo() {
2     Room vCurrent = this.aCurrentRoom;
3     System.out.println(vCurrent.getLongDescription());
4 }
```

Exercice 7.12 *Diagramme au lancement*

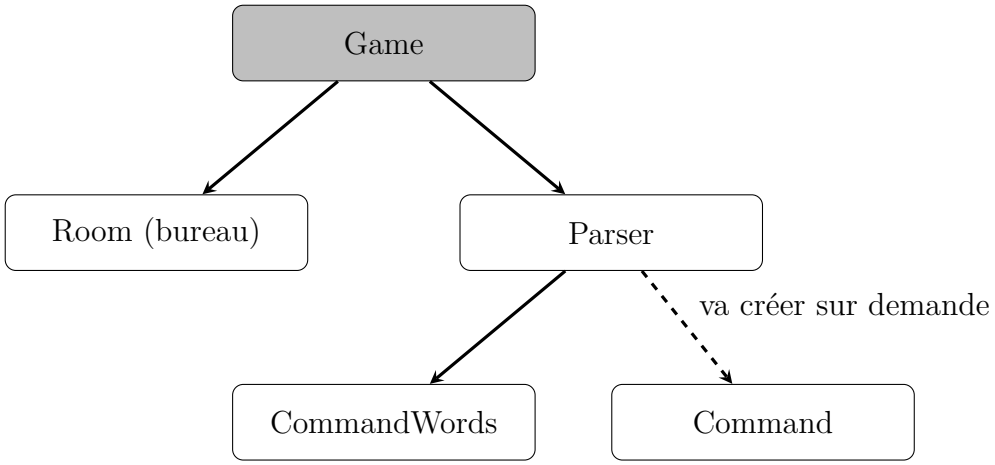


FIGURE 2.1 – Diagramme de relation entre les objets

Exercice 7.13 *Diagramme après go*

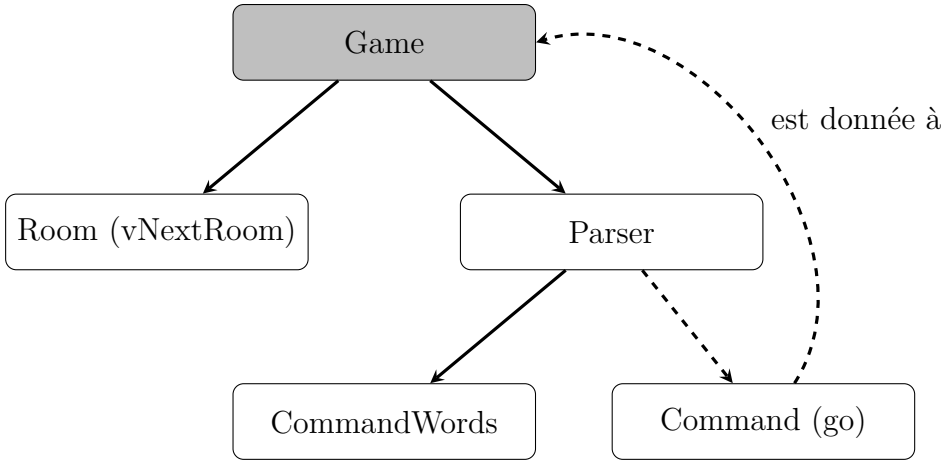


FIGURE 2.2 – Diagramme de relation entre les objets après un go

Exercice 7.14 *Commande look*

L'ajout de la commande `look` se traduit par plusieurs changements : le premier en mettant le mot de la commande dans `CommandWords`.

```
1  // a static constant array that will hold the valid commands
   ↪  words
2  private final static String[] aValidCommands =
3      { "go", "quit", "help", "look" };
```

Il faut aussi modifier `processCommand` en renseignant le nouveau mot de la commande ainsi que la fonction associée.

```
1  private boolean processCommand(final Command pCommand) {
2      if (pCommand.isUnknown()) {
3          System.out.println("I don't know what you mean...");
4          return false;
5      }
6
```

```
7       switch (pCommand.getCommandWord()) {
8           case "help":
9               this.printHelp();
10              return false;
11
12          case "quit":
13              return this.quit(pCommand);
14
15          case "go":
16              this.goRoom(pCommand);
17              return false;
18
19          case "look":
20              this.look();
21              return false;
22
23          default:
24              System.out.println("I don't know what you
                ↪  mean...");
25              return false;
26      }
27  }
```

On remarque aussi que la méthode `look()` fait, pour l'instant, la même chose que `printLocationInfo()`.

Exercice 7.15 *Commande search*

J'ai choisi d'implémenter une commande nommée `search <something>` qui permettra de fouiller des objet présent dans la pièce où se trouve le joueur. Pour ajouter cette commande, on modifie `CommandWords`.

```
1   private final static String[] aValidCommands = { "go",
    ↪  "quit", "help", "look", "search" };
```

Et on doit modifier la méthode `processCommand` de `Game` ainsi que créer la méthode pour gérer cette commande.

```
private boolean processCommand(final Command pCommand) {
    // code omitted for brevity

    switch (pCommand.getCommandWord()) {
        case "inspect":
            this.inspect(pCommand);
            return false;

        // code omitted for brevity
    }
}
```

```
private void inspect(final Command pCommand) {
    System.out.printf("Nothing to inspect here.");
}
```

Exercice 7.16 *showAll et showCommands*

Dans cet exercice, on rend l'affichage des commandes dans la méthode `printHelp` dynamique en créant un méthode d'énumération des commandes dans `CommandWords`.

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

Note : On observe que la méthode `showAll` pourrait être déclarée comme `static` car ne dépendant que de `CommandWords.aValidCommands` qui est un

attribut statique de la classe `CommandWords`.

On crée aussi un moyen de l'appeler depuis `Game` en passant par `Parser`.

```java
// Dans Parser.java
public void showCommands() {
    this.aValidCommands.showAll();
}
```

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

Exercice 7.17 *Changer Game ?*

Malgré toutes nos modifications, si l'on ajoute une nouvelle commande au jeu, il faudra quand même modifier la classe `Game`. En effet, la méthode `processCommand` contient un `switch` dont on doit ajouter une nouvelle branche pour chaque nouvelle commande.

Exercice 7.18 *getCommandList*

Dans cet exercice, on poursuit notre travail sur le modèle de la conception axée sur la responsabilité. Pour le cas de `showAll`, plutôt qu'afficher la liste des commandes disponibles, on préférera générer un `String` pour ne pas imposer un affichage spécifique (notamment via `System.out`).

```java
public String getCommandList() {
    String vResult = "";
```

```
    for (String command : CommandWords.aValidCommands) {
        vResult += command + " ";
    }

    return vResult;
}
```

On modifie la cascade crée aux exercices d'avant pour refléter ce changement.

```
// Dans Parser.java
public String getCommands() {
    return this.aValidCommands.getCommandList();
}
```

```
// Dans Game.java
private void printHelp() {
    System.out.println("You are lost. You are alone.");
    System.out.println("You wander around at the
    ↪  university.");
    System.out.println("Your command words are:");
    // "\t" représente une tabulation
    System.out.println("\t" + this.aParser.getCommands());
}
```

On en profite aussi pour changer la concaténation des `String` dans les boucles en utilisant `StringBuilder`. Par exemple, sur la boucle dans `Room`.

```
public String getExitString() {
    StringBuilder vResult = new StringBuilder("Exits : ");
    for (String vExit : this.aExits.keySet())
        vResult.append(vExit).append(" ");

    return vResult.toString();
}
```

Les objets Room

Dans l'optique de stocker les instances de `Room` crées, on ajoute un attribut `aAllRooms` qui contiendra tous les objets crées dans `createRooms`. On crée une fonction utilitaire `initRoom` pour nous faciliter la tâche.

```
public class Game {
    private final HashMap<String, Room> aAllRooms;

    private Room initRoom(final String pName, final String
     ↪  pDescription) {
        Room vCurrentRoom = new Room(pName, pDescription);
        this.aAllRooms.put(pName, vCurrentRoom);

        return vCurrentRoom;
    }
}
```

zuul-with-images

Après avoir fait les changements pour intégrer l'interface graphique voulue, on peut s'intéresser à son fonctionnement, notamment à sa gestion des évènements. En effet, dans le code de `UserInterface`, on retrouve cette ligne

```
private void createGUI() {
    // code omitted for brevity

    // add some event listeners to some components
    this.aEntryField.addActionListener(this);
}
```

Le fait de passer `this` en paramètre indique que c'est la classe courante (soit `UserInterface`) qui va interpréter au moyen d'une méthode `actionPerformed` les évènements venant du `JTextField`. Cela est rendu possible car `UserInterface` implémente l'interface `ActionListener`.

Imaginons que nous voulions implémenter un boutton "Indice" sur le côté

droit de l'interface, pour cela, on va devoir modifier `createGUI`.

```java
private void createGUI() {
    // code omitted for brevity

    // we create a new Button instance...
    JButton vButton = new JButton();
    // ...and we set a special text for it
    vButton.setText("Indice");

    // then, we add an action listener for the click event
    vButton.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent actionEvent)
        ↪ {
            this.giveIndice();
            vButton.setVisible(false); // we allow only one
            ↪ press for this button
        }
    });

    vPanel.setLayout(new BorderLayout());
    // finally, we set this button to the east of the main
    ↪ panel
    vPanel.add(vButton, BorderLayout.EAST);
}
```

Exercice 7.19 *Modèle Vue Contrôleur*

L'architecture MVC (signifiant *Modèle-Vue-Contrôleur*) permet d'organiser son code autour de trois éléments :

La vue la partie responsable sur l'affichage.

Le modèle la partie responsable de gérer les données.

Le contrôleur la partie responsable de la logique ainsi que de faire l'intermédiaire entre les données du modèle et l'affichage fait par les vues.

On retrouve cette architecture dans le web (non-exhaustivement, Laravel en PHP, Spring en Java, Express pour node.js etc.), dans la création d'application mobile et bureau...

Ce succès d'utilisation s'explique par le fait que séparer en trois grandes parties l'application permet de mieux se construire une représentation mentale de l'application : chaque élement appartient à un de ces trois groupes. L'un des autres avantages est de minimiser le nombres de modifications à effectuer lorsqu'un changement est nécessaire. On voit bien maintenant pourquoi cette modification pourrait être bénéfique au projet Zuul.

Note : J'ai, pour l'instant, décider de ne pas implémenter le modèle MVC dans mon projet Zuul. Je planifie de refaire l'interface utilisateur du jeu avec `JavaFX` qui force par design l'utilisation d'un pattern MVC.

Image

On en profite pour déplacer les images de la racine vers un nouveau dossier `Images`.

```
$ mkdir Images
$ mv *.png ./Images
$ ls Images
```

Exercice 7.20 *Item*

On crée une classe `Item` avec les accesseurs appropriés pour `description` et `weight`. Ensuite, on va attribuer un item à chaque objet `Room`. Pour cela, on modifie la classe pour ajouter une nouvelle méthode comme ceci :

```
1 public Room setItem(final String pDescription, final int
  ↪  pWeight) {
2     Item vItem = new Item(pDescription, pWeight);
3     this.aItem = vItem;
4 
5     return this;
6 }
```

Note : on rajoute aussi ici un `return this;` pour pouvoir chaîner les instructions lors de la création d'un objet `Room`.

Exercice 7.21 *Item description*

En suivant la logique de cohésion, les informations d'un objet `Item` doit-être généré par `Item` lui-même. La classe en charge d'afficher la description de l'`Item` doit-être `GameEngine`.

```
public String getLongDescription() {
    String vText = String.format(
        "Vous êtes actuellement dans la salle \"%s\"\n%s.\n",
        this.aName,
        this.aDescription
    );

    if (this.aItem == null) vText += "Il n'y a pas
    ↪  d'objet.\n";
    else vText += "Il y a un objet : \"" +
    ↪  this.aItem.getDescription() + "\".\n";

    vText += this.getExitString();

    return vText;
}
```

Améliorer la commande look

Pour améliorer la commande look, on remplace dans `GameEngine` la méthode éponyme.

```
private void look(final Command pCommand) {
    String vToDisplay;

    if (pCommand.hasSecondWord()) {
        Item vItem = this.aCurrentRoom.getItem();
```

```
6           vToDisplay =
            ↪  (vItem.getDescription().equals(pCommand.getSecondWord()))
7                   ? vItem.getLongDescription()
8                   : "Objet inconnu. Rien a afficher\n";
9       }
10      else {
11          vToDisplay = this.aCurrentRoom.getLongDescription();
12      }
13
14      this.aGui.println(vToDisplay);
15  }
```

Exercice 7.22 *Items*

On commence par remplacer la définition de la `HashMap` contenue dans chaque `Room`.

```
/**
 * The items of the room.
 */
private final HashMap<String, Item> aItems;
```

Puis ensuite on s'intéresse aux erreurs de compilation lié au changement de type.

On utilise ici une collection `HashMap` car on veut établir une relation entre le nom de l'item et l'objet représentant cet item. On n'a pas à parcourir la `HashMap` pour retrouver un item spécifique.

Exercice 7.23 *Commande back*

On va implémenter la commande `back` de façon très naïve : on rajoute un attribut `aPreviousRoom` dans `GameEngine` qui contiendra la salle qui précédait, dans le parcours du joueur, la salle actuelle.

Pour cela, on change la logique de notre commande `go`.

```
private void go(final Command pCommand) {
    if (!pCommand.hasSecondWord()) {
        this.aGui.println("Aller où ?");

        return;
    }

    Room vNextRoom =
     ↪  this.aCurrentRoom.getExit(pCommand.getSecondWord());
    if (vNextRoom == null) {
        this.aGui.println("Cette direction est
         ↪  inconnue...\n");
        return;
    }

    this.changeRoom(vNextRoom);
}
```

Et on rajoute en conséquence une méthode pour notre commande `back`.

```
private void back(final Command vCommand) {
    if (vCommand.hasSecondWord()) {
        this.aGui.println("retourner où ??");
        return;
    }

    this.changeRoom(this.aPreviousRoom);
}
```

Exercice 7.24 *Test de la commande back*

En voyant l'implémentation de la commande `back`, le cas singulier de la salle initiale saute aux yeux : en effet, si l'on a pas encore bougé, `aPreviousRoom` est évalué à `null`, pouvant ammener à un `NullPointerException`. On modifie donc `back`.

```java
private void back(final Command vCommand) {
    if (this.aPreviousRoom == null) {
        this.aGui.println("Aucune salle dans laquelle
        ↪  retourner.");
        return;
    }

    // code omitted for brevity
}
```

Exercice 7.25 *back back*

Exécuter deux commandes `back` nous fait retourner à la salle de départ, celle où ce situait le joueur lors de la première commande `back`. Ce problème est inhérent à l'implémentation faite de la commande `back`, notamment du fait que lors de l'exécution de `back`, la salle dans `aPreviousRoom` va dans `aCurrentRoom` et un second `back` fait exactement le contraire !

Exercice 7.26 *Stack*

Pour résoudre le problème évoqué à l'exercice précédent, on va utiliser une collection `Stack` dans l'attribut `aPreviousRooms`. Les deux fonctions qui nous intéressent sur `Stack` sont `push` `pop()`, permettant respectivement d'insérer un élément en haut de la pile et de récupérer l'élément le plus haut de la pile. Ainsi, on change `go` et `back` pour refléter ce changement.

```java
private void go(final Command pCommand) {
    // code omitted for brevity

    this.aPreviousRooms.push(this.aCurrentRoom);
    this.changeRoom(vNextRoom);
}
```

Pour ne pas faire de régression (et faire en sort que Java ne soulève pas

une exception `EmptyStackException`), on va tester à chaque appel de `back` si `aPreviousRooms` est vide.

```
1  private void back(final Command vCommand) {
2      // code omitted for brevity
3
4      if (this.aPreviousRooms.isEmpty()) {
5        this.aGui.println("Aucune salle dans laquelle
           ↪  retourner.");
6        return;
7      }
8
9      Room vPreviousRoom = this.aPreviousRooms.pop();
10     this.changeRoom(vPreviousRoom);
11 }
```

A L'Interface Utilisateur

A.1 JavaFX

Après avoir testé la création d'interface au moyen de `Swing` et de `awt` comme demandé dans les exercices, j'ai décidé de partir sur `JavaFX` car mon interface se complexifiait et j'avais l'impression de me répéter énormément dans le code.