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.

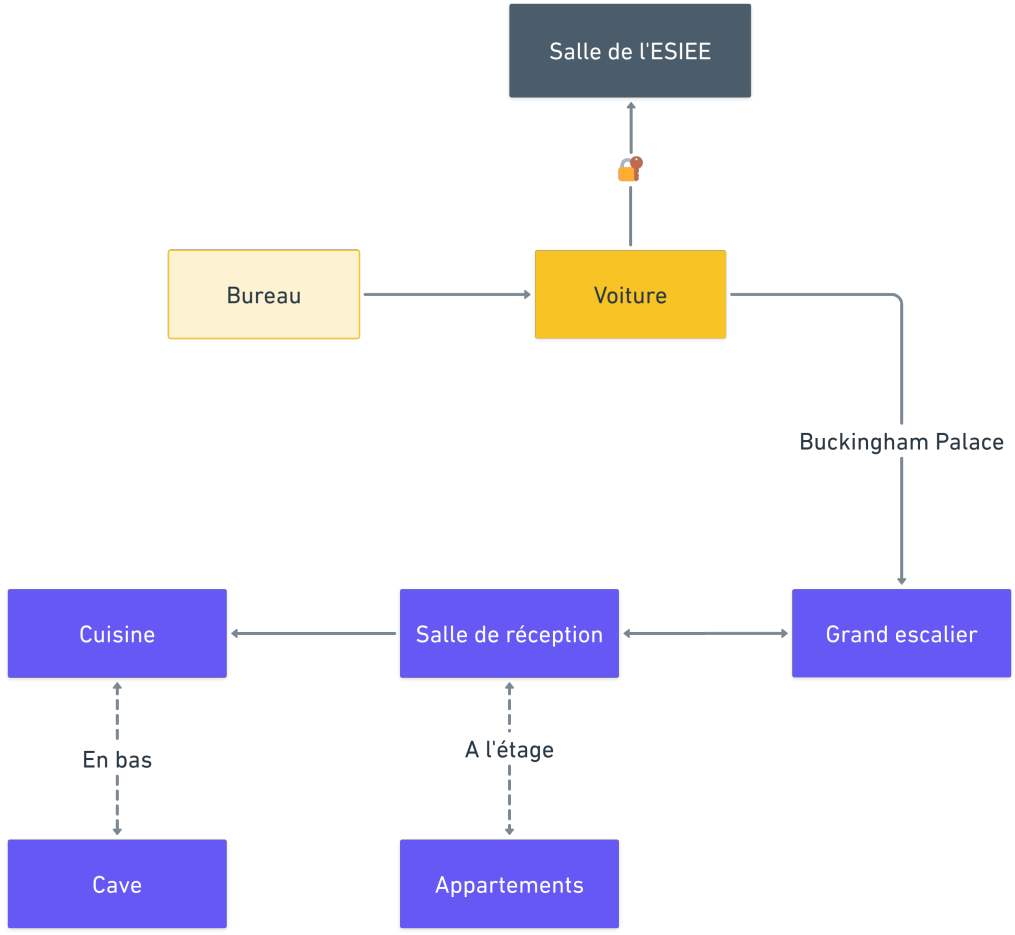
Murphy Law et Le Joueur se réveille dans un grand escalier avec des dorures et un dôme magnifique surmontant la pièce. Après l'irruption d'un majordome disant qu'on les cherche partout, Murphy et Le Joueur réalisent qu'ils passé dans l'univers d'un autre jeu de la promotion, se passant à Buckingham Palace. Très vite, Murphy Law et Le Joueur sont accostés par les employés

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. Murphy en ressort gagnant.

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`.

```
public Room getExit(final String pDirection) {
    switch (pDirection) {
        case "north":
            return this.aNorthExit;

        case "east":
            return this.aEastExit;

        case "south":
            return this.aSouthExit;

        case "west":
            return this.aWestExit;

        default:
            return null;
    }
}
```

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`.

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

    if (this.aEastExit != null) vResult += "east ";
    if (this.aNorthExit != null) vResult += "north ";
    if (this.aSouthExit != null) vResult += "south ";
    if (this.aWestExit != null) vResult += "west";

    return vResult;
}

```

```
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 :

```
1  office.setExit("east", car)
2        .setExit("north", kitchen)
3        .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.

```
1  public String getExitString() {
2      String vResult = "Exits : ";
3      for (String vExit : this.aExits.keySet())
4          vResult += vExit + " ";
5
6      return vResult;
7  }
```

Exercice 7.10 *Fonctionnement de keySet et Javadoc*

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

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

Ce code marche car `keySet` renvoit 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`.

```java
public String getLongDescription() {
    return "Vous êtes actuellement dans la salle \"" +
            this.aName + "\".\n" +
            this.aDescription + ".\n" +
            this.getExitString();
}
```

Et on change `Game` en conséquence.

```java
private void printLocationInfo() {
    Room vCurrent = this.aCurrentRoom;
    System.out.println(vCurrent.getLongDescription());
}
```

Exercice 7.12 *Diagramme au lancement*

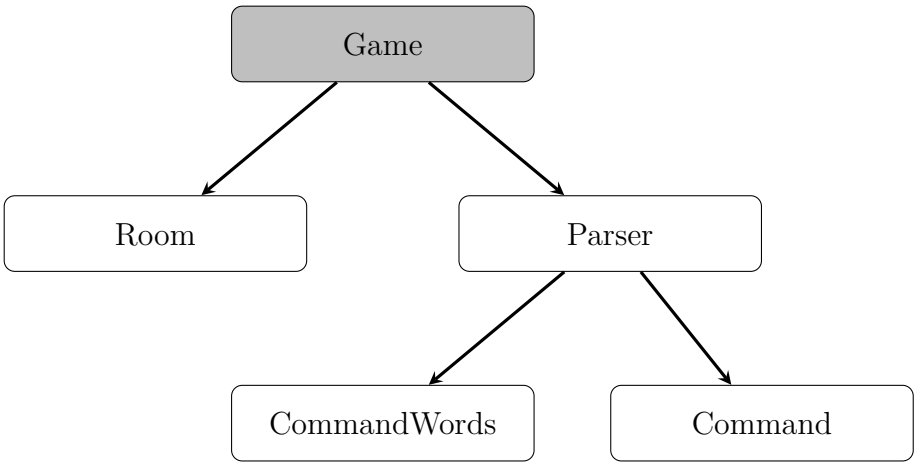


FIGURE 2.1 – Diagramme de relation entre les classes

Exercice 7.13 *Commande look*

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

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

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

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

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

        case "quit":
            return this.quit(pCommand);

        case "go":
            this.goRoom(pCommand);
            return false;

        case "look":
            this.look();
            return false;

        default:
            System.out.println("I don't know what you mean...");
            return false;
    }
}
```

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

Exercice 7.14 *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.

```
1  private boolean processCommand(final Command pCommand) {
2      ...
3
4      switch (pCommand.getCommandWord()) {
5          case "inspect":
6              this.inspect(pCommand);
7              return false;
8
9          ...
10     }
11 }
```

```
1  private void inspect(final Command pCommand) {
2      System.out.printf("Nothing to inspect here.");
3  }
```

Note : On utilise ici ... pour signaler les parties du code inutile à montrer car n'ayant pas changées entre les exercices.

Exercice 7.15 *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`.

```
1  public void showAll() {
2      System.out.print("\t");
```

```
3       for (String command : CommandWords.aValidCommands) {
4           System.out.print(command + " ");
5       }
6       System.out.println();
7   }
```

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`.

```
1   // Dans Parser.java
2   public void showCommands() {
3       this.aValidCommands.showAll();
4   }
```

```
1   // Dans Game.java
2   private void printHelp() {
3       System.out.println("You are lost. You are alone.");
4       System.out.println("You wander around at the
        ↪  university.");
5       System.out.println("Your command words are:");
6       this.aParser.showCommands();
7   }
```

Exercice 7.16 *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 7.17 *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.

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

```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:");
    // "\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`.

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

    return vResult.toString();
}
```

Exercice 7.18 *Les objets Room*

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

```
1  public class Game {
2      private final HashMap<String, Room> aAllRooms;
3
4      ...
5
6      // notre fonction utilitaire qui crée l'objet
7      // Room, l'ajoute à la HashMap et la retourne.
8      private Room initRoom(final String pName, final String
       ↪  pDescription) {
9          Room vCurrentRoom = new Room(pName, pDescription);
10         this.aAllRooms.put(pName, vCurrentRoom);
11
12         return vCurrentRoom;
13     }
14 }
```