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, 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 Plan

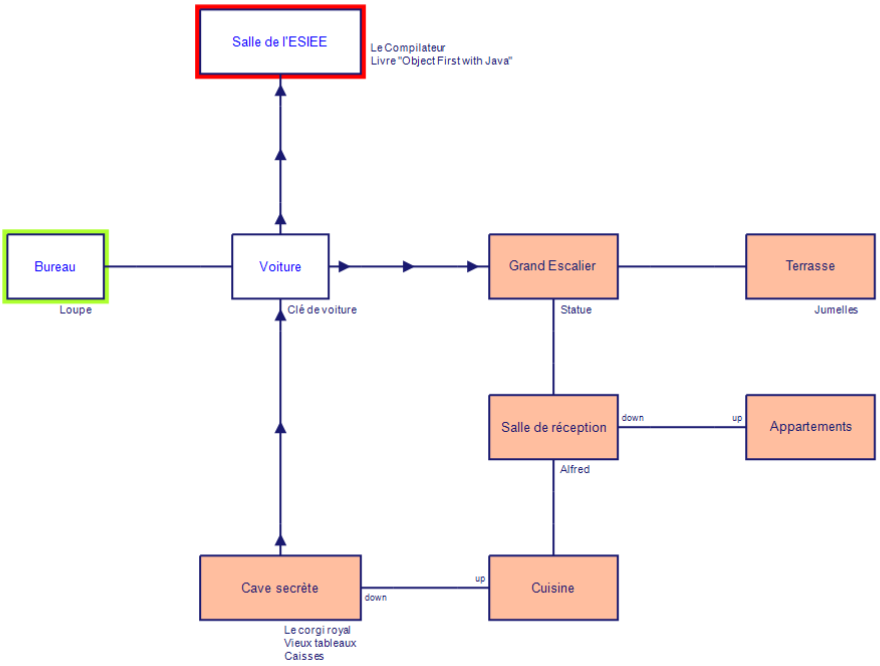


FIGURE 1.1 – Plan du jeu

## 1.6 Détail des lieux, items, personnages

### 1.6.1 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.

### 1.6.2 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.

## 1.7 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.

## 1.8 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 réécrit `printLocationInfo`.

```
public String getExitString() {
    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 minimisé, 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 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 fondamentale 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, pour la première, seul `play` est publique, tandis que la seconde 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

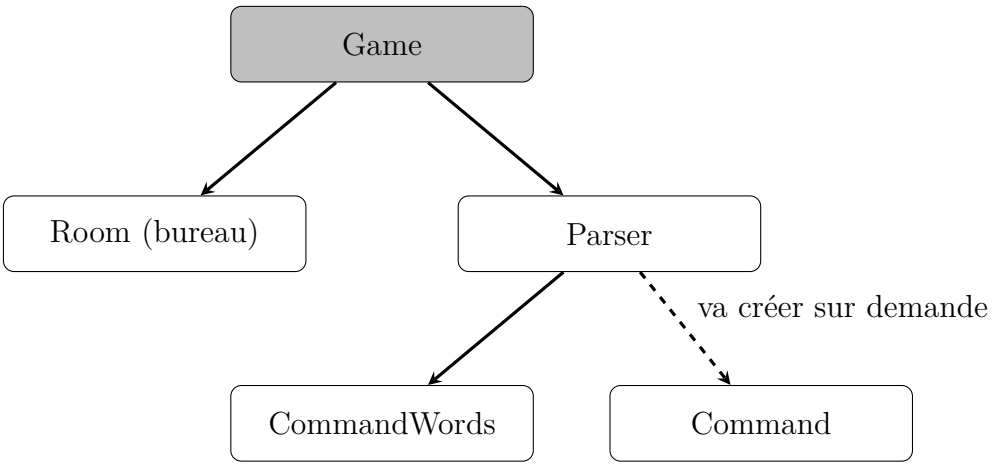


FIGURE 2.1 – Diagramme de relation entre les objets

## Exercice 7.13 Diagramme après go

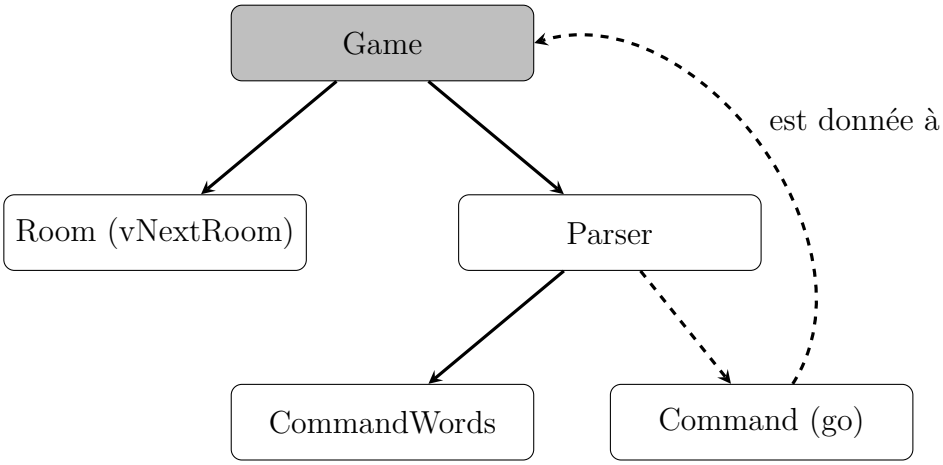


FIGURE 2.2 – Diagramme de relation entre les objets après un go

## 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.

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.

```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();
}
```

## 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`.

```
1  private void createGUI() {
2      // code omitted for brevity
3
4      // we create a new Button instance...
5      JButton vButton = new JButton();
6      // ...and we set a special text for it
7      vButton.setText("Indice");
8
9      // then, we add an action listener for the click event
10     vButton.addActionListener(new ActionListener() {
11         @Override
12         public void actionPerformed(ActionEvent actionEvent)
           ↪  {
13             this.giveIndice();
14             vButton.setVisible(false); // we allow only one
               ↪  press for this button
15         }
16     });
17
18     vPanel.setLayout(new BorderLayout());
19     // finally, we set this button to the east of the main
       ↪  panel
20     vPanel.add(vButton, BorderLayout.EAST);
21 }
```

## 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 décidé de ne pas implémenter le modèle MVC dans mon projet Zuul, préférant me concentrer sur le modèle de programmation réactive proposé par `JavaFX`.

### 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`.

```
1  public String getLongDescription() {
2      String vText = String.format(
3          "Vous êtes actuellement dans la salle \"%s\"\n%s.\n",
4          this.aName,
5          this.aDescription
6      );
7
8      if (this.aItem == null) vText += "Il n'y a pas
       ↪  d'objet.\n";
9      else vText += "Il y a un objet : \"" +
       ↪  this.aItem.getDescription() + "\".\n";
10
11     vText += this.getExitString();
```

```
12
13      return vText;
14  }
```

### Améliorer la commande look

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

```
1   private void look(final Command pCommand) {
2       String vToDisplay;
3
4       if (pCommand.hasSecondWord()) {
5           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`.

```
1   /**
2    * 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`.

```
1  private void back(final Command pCommand) {
2      if (pCommand.hasSecondWord()) {
3          this.aGui.println("retourner où ??");
4          return;
5      }
6
7      this.changeRoom(this.aPreviousRoom);
8  }
```

## 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`.

```
1  private void back(final Command pCommand) {
2      if (this.aPreviousRoom == null) {
3          this.aGui.println("Aucune salle dans laquelle
           ↪  retourner.");
4          return;
5      }
6
7      // code omitted for brevity
8  }
```

## 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` et `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.

```
1  private void go(final Command pCommand) {
2      // code omitted for brevity
3
4      this.aPreviousRooms.push(this.aCurrentRoom);
5      this.changeRoom(vNextRoom);
6  }
```

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 pCommand) {
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  }
```

## Exercice 7.27 Réflexion sur les tests

Pour le projet Zuul, les tests devraient porter sur les fonctionnalités suivantes :

**commandes** s'assurer que la logique des commandes est fonctionnelle
**traitement** s'assurer que le traitement de l'entrée est fonctionnel

## Exercice 7.28 Fonctionnement des tests

Une idée serait de mettre en place des tests unitaires et fonctionnels. On peut faire cela en créant des fichiers contenant un scénario détaillé à tester pour s'assurer qu'il n'y a pas de régression.

On crée donc une nouvelle commande `test` dont le but est de charger un scénario (spécifié en second mot) et de l'exécuter dans l'interface. Pour ce faire, on crée la méthode test de la façon suivante :

```
1   private void test(final Command pCommand) {
2       if (!pCommand.hasSecondWord()) {
3           System.out.printf("La commande test doit avoir un
            ↪  second mot.%n");
4           return;
5       }
6
7       this.aGui.println("--- DEBUT MODE TEST ---");
8
9       String vPath = "./test/" + pCommand.getSecondWord();
10
```

```
11      Scanner vScanner;
12      try {
13          vScanner = new Scanner(new File(vPath));
14      } catch (FileNotFoundException vError) {
15          String vMessage = String.format("Le fichier \"%s\" est
            ↪  introuvable.%n", vPath);
16          this.aGui.println(vMessage);
17
18          return;
19      }
20
21      while (vScanner.hasNextLine())
        ↪  this.interpretCommand(vScanner.nextLine());
22
23      this.aGui.println("--- FIN MODE TEST ---");
24  }
```

## Exercice 7.29 Player

On va réduire de taille `GameEngine` en la déchargeant de certaines responsabilités, que l'on va transférer vers une nouvelle classe `Player`, qui représentera l'état d'un joueur durant notre jeu. Les attributs qui sont relatifs au joueur sont immédiats et doivent donc être transférés.

```
1   public class Player {
2       /**
3        * The history of the rooms were the player was.
4        */
5       private final Stack<Room> aPreviousRooms;
6
7       /**
8        * The room where the player is.
9        */
10      private Room aCurrentRoom;
11
```

```
12      // constructor omitted for brevity
13  }
```

Fort de ce changement, on déplace le code dont la responsabilité va directement à la classe `Player` et non plus à `GameEngine`, par exemple, le changement de salle, maintenant entièrement géré par `Player` comme on peut le voir :

```
1   /**
2    * Goes back to the previous room.
3    *
4    * @throws EmptyStackException if there is no previous room.
5    */
6   public void toPreviousRoom() throws EmptyStackException {
7       this.aCurrentRoom = this.aPreviousRooms.pop();
8   }
9
10  /**
11   * Sets the new current room where the player will be.
12   *
13   * @param pRoom the new current room.
14   */
15  public void setCurrentRoom(final Room pRoom) {
16      if (this.aCurrentRoom != null)
        ↪  this.aPreviousRooms.push(this.aCurrentRoom);
17      this.aCurrentRoom = pRoom;
18  }
```

## Changement d'architecture

A ce niveau de projet, il est clairement apparu une nécessité d'agencer différemment l'organisation et l'architecture du Zuul. De plus, j'ai proposé une idée à Monsieur Bureau pour faire cohabiter trois types d'interfaces pour jouer au jeu : `JavaFX`, `Swing` et l'interface `Console`.

Pour plus détails concernant l'implémentation de JavaFX dans le projet Zuul, l'annexe A.1 est disponible dans ce rapport.

## Exercice 7.30 Take et Drop

On commence par se poser la question suivante : quelle classe devrait être en charge de gérer l'inventaire du joueur ainsi que la logique de gestion des items d'une salle ? En prenant appui sur les modification de l'exercice précédent, `Player` semble être un bon candidat. On rajoute donc l'attribut nécessaire `private Item aCarriedItem;` et on la modifie en conséquence.

```
1  /**
2   * Takes an item in the current room.
3   *
4   * @param pItemName the item to be taken.
5   * @return the item that has been taken.
6   * @throws CannotManageItemException if the requested object
    ↪  cannot be took.
7   */
8  public Item takeItem(final String pItemName) throws
    ↪  CannotManageItemException {
9      if (this.aCurrentRoom == null) throw new
       ↪  CannotManageItemException("Aucune salle dans laquelle
       ↪  prendre l'objet.");
10
11     // remove the item from the current room...
12     Item vItem = this.aCurrentRoom.removeItem(pItemName);
13     if (vItem == null) throw new
       ↪  CannotManageItemException("L'objet à prendre n'existe
       ↪  pas dans la salle.");
14
15     // ...and add it to the player's inventory
16     this.aCarriedItem = vItem;
17
18     return vItem;
19 }
20
21 /**
```

```java
 * Drops an item in the current room.
 *
 * @return the item that has been dropped.
 * @throws CannotManageItemException if the requested object
 ↪  cannot be dropped.
 */
public Item dropItem() throws CannotManageItemException {
    if (this.aCurrentRoom == null) throw new
     ↪  CannotManageItemException("Aucune salle dans laquelle
     ↪  prendre l'objet.");

    // remove the item from the player inventory...
    Item vItem = this.aCarriedItem;
    if (vItem == null) throw new
     ↪  CannotManageItemException("L'objet a déposer n'existe
     ↪  pas dans votre inventaire.");

    this.aCarriedItem = null;

    // ...and add it to the room
    this.aCurrentRoom.addItem(vItem);

    return vItem;
}
```

**Notes :** on utilise ici le système de gestion d'erreurs pour alerter la méthode appelante que certaines actions ne peuvent pas être effectuées. Pourquoi ne pas utiliser un simple if / else pour vérifier que la méthode ne retourne pas `null` ? Simplement car l'impossibilité de manipuler l'item peut venir de différentes raisons (pour l'instant, il y en a deux, soit le joueur n'est pas dans une salle, soit l'item n'existe pas).

L'ajout des commandes et leurs implémentations dans `GameEngine` découlent directement de ces modifications.

```java
private void drop(Command pCommand) {
    if (pCommand.hasSecondWord()) {
        this.aGui.println("Déposer quoi ???");

```

```
5           return;
6       }
7
8       Item vItem;
9       try {
10          vItem = this.aPlayer.dropItem();
11      } catch (Player.CannotManageItemException vError) {
12          this.aGui.println("Aucun item déposé.\n" +
            ↪  vError.getMessage());
13          return;
14      }
15
16      this.aGui.println("Vous avez bien déposé " +
        ↪  vItem.getName() + ".");
17  }
18
19  private void take(Command pCommand) {
20      if (!pCommand.hasSecondWord()) {
21          this.aGui.println("Prendre quoi ???");
22          return;
23      }
24
25      Item vItem;
26      try {
27          vItem =
            ↪  this.aPlayer.takeItem(pCommand.getSecondWord());
28      } catch (Player.CannotManageItemException vError) {
29          this.aGui.println("Impossible de prendre l'objet.\n" +
            ↪  vError.getMessage());
30          return;
31      }
32
33      this.aGui.println("Vous avez bien récupéré " +
        ↪  vItem.getName() + ".");
34  }
```

## Exercice 7.31 Plusieurs items et ItemList

Pour porter plusieurs objets, il suffit de faires plusieurs modifications simple sur notre code. Premièrement, dans `Player`, changeons notre attribut et réécrivons nos méthodes manipulant les objets.

```
1  public class Player {
2      private final HashMap<String, Item> aInventory;
3
4      public Item takeItem(final String pItemName) throws
       ↪  CannotManageItemException {
5          if (this.aCurrentRoom == null) throw new
           ↪  CannotManageItemException("Aucune salle dans
           ↪  laquelle prendre l'objet.");
6
7          // remove the item from the current room...
8          Item vItem = this.aCurrentRoom.removeItem(pItemName);
9          if (vItem == null) throw new
           ↪  CannotManageItemException("L'objet à prendre
           ↪  n'existe pas dans la salle.");
10
11         // ...and add it to the player's inventory
12         this.aInventory.put(vItem.getName(), vItem);
13
14         return vItem;
15     }
16
17     public Item dropItem(final String pItemName) throws
       ↪  CannotManageItemException {
18         if (this.aCurrentRoom == null) throw new
           ↪  CannotManageItemException("Aucune salle dans
           ↪  laquelle poser l'objet.");
19
20         // remove the item from the player inventory...
21         Item vItem = this.aInventory.remove(pItemName);
22         if (vItem == null) throw new
           ↪  CannotManageItemException("L'objet a déposer
           ↪  n'existe pas dans votre inventaire.");
23
24         // ...and add it to the room
```

```
25          this.aCurrentRoom.addItem(vItem);
26
27          return vItem;
28      }
29
30      // code omitted for brevity
31  }
```

## Exercice 7.32 Poids maximum

Arbitrairement, spécifions le poids maximum pouvant être porté par `Player` comme étant 100. On crée une constante contenant cette information pour pouvoir la changer plus facilement si le besoin se fait sentir. On crée aussi un attribut initialisé à 0 pour suivre le poids actuel de l'inventaire du joueur.

```
1   class Player {
2       public final static int MAX_WEIGHT = 100;
3
4       private int aWeight;
5   }
```

On doit aussi modifier `takeItem` et `dropItem` pour qu'ils mettent à jour le poids porté par le joueur. Évidemment, `takeItem` augment le poids tandis que `dropItem` le diminue. Il ne faut pas oublier pour `takeItem` les conditions nécessaires pour déterminer si le joueur peut prendre ou non un item de la salle.

```
1   public Item takeItem(final String pItemName) throws
    ↪  CannotManageItemException {
2       if (this.aCurrentRoom == null)
3           throw new CannotManageItemException("Aucune salle dans
            ↪  laquelle prendre l'objet.");
4
5       // get the item from the current room...
```

```
6       Item vItem = this.aCurrentRoom.getItem(pItemName);
7       if (vItem == null) throw new
        ↪  CannotManageItemException("L'objet à prendre n'existe
        ↪  pas dans la salle.");
8
9       // if the difference is less than zero, the total weight
        ↪  of the items exceeds the allowed limit
10      if (this.aWeight + vItem.getWeight() > Player.MAX_WEIGHT)
        ↪  {
11          throw new CannotManageItemException("Vous ne pouvez
            ↪  pas prendre cet objet, votre inventaire est trop
            ↪  lourd !");
12      }
13
14      // then we definitely remove it from the room
15      this.aWeight += vItem.getWeight();
16      this.aCurrentRoom.removeItem(vItem);
17
18      // ...and add it to the player's inventory
19      this.aInventory.addItem(vItem);
20
21      return vItem;
22  }
23
24  public Item dropItem(final String pItemName) throws
    ↪  CannotManageItemException {
25      if (this.aCurrentRoom == null) throw new
        ↪  CannotManageItemException("Aucune salle dans laquelle
        ↪  poser l'objet.");
26
27      // remove the item from the player inventory...
28      Item vItem = this.aInventory.removeItem(pItemName);
29      if (vItem == null) throw new
        ↪  CannotManageItemException("L'objet a déposer n'existe
        ↪  pas dans votre inventaire.");
30
31      // subtract the item weight
32      this.aWeight -= vItem.getWeight();
33
34      // ...and add it to the room
```

```
35      this.aCurrentRoom.addItem(vItem);
36
37      return vItem;
38  }
```

## Exercice 7.33 Inventaire

On a besoin de récupérer tous les noms des items d'un inventaire, on implémente donc une nouvelle méthode dans `ItemList`.

```
1  public String getAllNames() {
2      return String.join(", ", this.aItems.keySet());
3  }
```

Et, dans `Player`, on va adjoindre à cette description le poids de tous les items.

```
1  public String getInventoryDescription() {
2      if (this.aInventory.isEmpty()) return "L'inventaire est
       ↪  vide !";
3
4      return String.format("Inventaire : %s (poids %d)",
       ↪  this.aInventory.getAllNames(), this.aWeight.get());
5  }
```

Et on finit par implémenter une nouvelle commande `inventaire` dans `GameEngine`.

```
1  private void item(final Command pCommand) {
2      if (pCommand.hasSecondWord()) {
3          this.aGui.println("Pas de second mot pour items.");
4          return;
```

```
5       }
6
7       this.aGui.println(this.aPlayer.getInventoryDescription());
8   }
```

On peut aussi en profiter pour changer le comportement de la commande quand un second mot est donné. On affchera la description de l'item demandé s'il est dans l'inventaire.

```
1       */
2   private void item(final Command pCommand) {
3       if (pCommand.hasSecondWord()) {
4           Item vItem =
            ↪  this.aPlayer.getItem(pCommand.getSecondWord());
5
6           if (vItem == null) this.aInterface.println("Vous ne
            ↪  possédez pas cet item.");
7           else
            ↪  this.aInterface.println(vItem.getLongDescription());
8
9           return;
10      }
11
12
        ↪  this.aInterface.println(this.aPlayer.getInventoryDescription());
13  }
```

## Exercice 7.34 Cookie magique

Pour implémenter le cookie magique, on va créer une nouvelle commande `use` plutôt que de créer une commande `eat`. La raison est qu'une commande `eat` servirait uniquemment une fois pour le cookie magique tandis que, grâce à sa sémantique plus large, `use` est réutilisable dans d'autre situation, avec d'autres items.

Une cookie magique est donc une sorte d'`Item` mais possédant la capacité

de modifier certaines caractéristiques de notre joueur quand celui-ci l'utilise. On va donc créer dans `Item` une méthode `use` lançant une exception et qui pourra être redéfinie dans les classes définissant un item spécial tel que `Beamer` ou `Cookie`.

```
/**
 * Uses the current item.
 *
 * @param vPlayer player that is using the item.
 */
public void use(final Player vPlayer) {
    throw new UnsupportedOperationException("Cet item ne peut
    ↪  pas être utilisé.");
}
```

Ensuite on peut définir notre cookie magique assez facilement.

```
public class Cookie extends Item {
    public final static int WEIGHT_ADDED = 10;

    public Cookie() {
        super("cookie", "Un cookie qui semble...
        ↪  particulier.", 2);
    }

    /**
     * Uses the cookie and modify the maximum weight that
     * the player can be carried.
     *
     * @param pPlayer player that is using the item.
     */
    @Override
    public void use(final Player pPlayer) {
        int vNewMax = pPlayer.getMaxWeight() + WEIGHT_ADDED;

        pPlayer.setMaxWeight(vNewMax);
        pPlayer.deleteItem(this.getName());
    }
}
```

Et on implémente la logique de notre commande `use`.

```
1  private void use(final Command pCommand) {
2      if (!pCommand.hasSecondWord()) {
3          this.aInterface.println("Vous devez spécifier un objet
           ↪  à utiliser.");
4          return;
5      }
6
7      Item vItem =
       ↪  this.aPlayer.getItem(pCommand.getSecondWord());
8      if (vItem == null) {
9          this.aInterface.println("Vous ne possédez pas cet
           ↪  item.");
10         return;
11     }
12
13     try {
14         vItem.use(this.aPlayer);
15     } catch (Exception pError) {
16         this.aInterface.println(pError.getMessage());
17         return;
18     }
19
20     this.aInterface.printf("Vous avez bien utilisé %s.",
       ↪  vItem.getName());
21     this.aInterface.println();
22 }
```

## Exercice 7.35 zuul-with-enums-v1

On utilisait déjà un `switch` pour traiter plus efficacement les commandes. Après implémentation de la nouvelle classe `CommandWord`, on le change juste pour être utilisé avec des `enum`.

```java
public void processCommand(final String pInput) {
    this.aInterface.println("> " + pInput);

    Command vCommand = Parser.parseCommand(pInput);

    switch (vCommand.getCommandWord()) {
        case GO:
            this.go(vCommand);
            break;

    // code omitted for brevity
}
```

## Exercice 7.40 enum/look

On va ré-implémenter `look` en utilisant le système d'`enum` établit à l'exercise précédent. On commence par modifier `CommandWord`.

```java
public enum CommandWord {
    GO("go"),
    BACK("back"),
    LOOK("look"),

    // code omitted for brevity
}
```

Puis on ajoute la gestion de la commande dans le switch de `GameEngine`.

```java
switch (vCommand.getCommandWord()) {
    case GO:
        this.go(vCommand);
        break;

    case LOOK:
```

```
7           this.look(vCommand);
8           break;
9
10      // code omitted for brevity
11  }
```

On remarquera un énorme gain de temps et de maintenabilité pour l'ajout d'une nouvelle commande, seulement deux classes ont eu besoin d'être modifiés, de plus de façon minime. Ceci est rendu possible grâce au chargement automatique de la `Map` de `CommandWords` et lors du bouclage des valeurs de l'enum.

**Notes :** en implémentant la logique de `zuul-with-enum-v2`, j'ai trouvé que la syntaxe pour le chargement de la `Map` de correspondance commande-enum était assez lourde. J'ai préféré ajouté un nouveau attribut dans mon `enum` comme ceci :

```
1   public enum CommandWord {
2       GO("go"),
3       BACK("back"),
4       // code omitted for brevity
5       UNKNOWN("???", true);
6
7       private String aCommandName;
8       private boolean aIsHidden;
9
10      CommandWord(final String pCommandName, final boolean
        ↪ pIsHidden) {
11          this.aCommandName = pCommandName;
12          this.aIsHidden = pIsHidden;
13      }
14
15      CommandWord(final String pCommandName) {
16          this(pCommandName, false);
17      }
18
19      public boolean getIsHidden() {
20          return this.aIsHidden;
21      }
```

```
22
23      @Override
24      public String toString() {
25          return this.aCommandName;
26      }
27  }
```

Avec cette modification, on peut modifier le chargement de la `Map` dans `CommandWords` comme ceci :

```
1   public class CommandWords {
2       public static final Map<String, CommandWord>
        ↪   aValidCommands;
3
4       static {
5           Map vMap = new HashMap<String, CommandWord>();
6           for (CommandWord command : CommandWord.values()) {
7               if (command.getIsHidden()) continue;
8               vMap.put(command.toString(), command);
9           }
10
11          aValidCommands = Collections.unmodifiableMap(vMap);
12      }
```

## Exercice 7.41 enum/help

Changeons la constante associé à `help` dans l'enum de `CommandWord` pour le faire passer de `HELP("help")` à une version comme ceci `HELP("aide")`. Lançons le jeu et observons.

```
> help
Commande inconnue.
Essayez une des commandes suivantes:
take item test drop aide use go back quit
```

La liste des commandes à bien été mise-à-jour sans aucun changement de code, ce qui démontre encore une fois l'utilité énorme des `enum`.

## Exercice 7.42 Limite de temps

TODO

## Exercice 7.43 Trap Door

Tout l'interêt de cet exercice repose sur la supression du contenu du `Stack`, gérant notre historique, au bon moment. Pour cela, on définit une porte piégée comme étant une salle A donnant accès à une salle B tandis que la salle B ne donne pas accès à la salle A (on peut voir ça comme une relation asymétrique). Si tel est le cas, on supprime notre historique pour ne pas permettre au joueur d'utiliser `back`.

On pourrait vérifier que si l'on va par exemple au nord, la nouvelle salle possède bien une sortie au sud qui rammène à la salle initiale, mais essayons autre chose en exploitant toutes les possibilités offertes par une `Map`.

Pour implémenter ceci, on va créer dans `Room` une méthode `hasExit` qui vérifira si l'on peut rejoindre une salle depuis la salle actuelle, peut-importe la direction (on ne s'intéresse qu'à la valeur de la `Map`, pas la clé).

```
1 public boolean hasExit(final Room vRoom) {
2     return this.aExits.containsValue(vRoom);
3 }
```

Ensuite, on se place dans `Player` et dans la méthode `goToExit` qui permet au joueur de changer de salle.

```
1 public void goToExit(final String pDirection) throws
  ↪  RoomNotFoundException {
2     Room vCurrent = this.aCurrentRoom.get();
3
```

```
4       Room vNext = vCurrent.getExit(pDirection);
5       if (vNext == null) throw new RoomNotFoundException();
6
7       this.setCurrentRoom(vNext);
8
9       // check if the door was a trapdoor
10      if (!vNext.hasExit(vCurrent)) {
11          this.clearRoomHistory();
12      }
13  }
```

## Exercice 7.44 Beamer

Pour implémenter le `Beamer`, on va s'appuyer sur la même structure que le cookie magique, c'est-à-dire créer une nouvelle classe `Beamer` étendant `Item`.

```
1   public class Beamer extends Item {
2       private Room aSavedLocation;
3
4       public Beamer() {
5           super("beamer", "Pratique pour revenir sur vos pas
            ↪  rapidement !", 50);
6       }
7
8       @Override
9       public void use(final Engine pEngine) {
10          UserInterface vInterface = pEngine.getInterface();
11          Player vPlayer = pEngine.getPlayer();
12
13          if (this.aSavedLocation == null) {
14              this.aSavedLocation =
                ↪  pEngine.getPlayer().getCurrentRoom();
15              vInterface.println("Beamer chargé !");
16          } else {
17              vPlayer.clearRoomHistory();
```

```
18              vPlayer.setCurrentRoom(this.aSavedLocation);
19              vPlayer.deleteItem(this.getName());
20
21              vInterface.println("Beamer utilisé !");
22              vInterface.println();
23
24              pEngine.printLocationInfo();
25          }
26      }
27  }
```

On utilisera la commande `use` pour chargé le beamer puis une deuxième fois `use` pour l'utiliser.

**Note :** On peut remarquer que le beamer se supprime de lui-même de l'inventaire du joueur après utilisation.

## Exercice 7.45 locked door

TODO

## Exercice 7.46 Transporter

On commence par définir une classe `RoomRandomizer` dont le but sera de gérer et de retourne une pièce aléatoire parmis un ensemble de pièce du jeu.

```
1   public class RoomRandomizer {
2       private final Map<String, Room> aRooms;
3       private final Random aRandom;
4
5       public RoomRandomizer() {
6           this.aRooms = new HashMap<>();
7           this.aRandom = new Random();
8       }
```

```
9
10      public void add(final Room pRoom) {
11          this.aRooms.put(pRoom.getName(), pRoom);
12      }
13
14      public Room getRoom(final String pName) {
15          return this.aRooms.get(pName);
16      }
17
18      public Room getRandomRoom() {
19          if (this.aRooms.isEmpty()) return null;
20
21          Object vRoom =
            ↪  this.aRooms.values().toArray()[this.aRandom.nextInt(this.aRooms.size
22          return (Room) vRoom;
23      }
24  }
```

Et on défini notre `TransporterRoom` comme étant un type de `Room` particulier.

```
1   public class TransporterRoom extends Room {
2       private final RoomRandomizer aRandomizer;
3       private String aForcedExit;
4
5       public TransporterRoom(final RoomRandomizer pRandomizer)
        ↪  {
6           super("Salle de téléportation", "Pratique pour se
            ↪  déplacer dans le vaisseau");
7
8           this.aRandomizer = pRandomizer;
9       }
10
11      @Override
12      public Room getExit(final String pDirection) {
13          return this.aForcedExit == null
14                  ? this.aRandomizer.getRandomRoom()
15                  : this.aRandomizer.getRoom(this.aForcedExit);
16      }
```

```
17
18      public void setForcedExit(final String pExitName) {
19          this.aForcedExit = pExitName;
20      }
21  }
```

Ainsi, chaque `TransporterRoom` recevra, lors de son initialisation, une instance de `RoomRandomizer` pour pouvoir téléporter le joueur. On passera cette instance dans `GameEngine`.

```
1  private void createRooms() {
2      // code omitted for brevity
3
4      Room vTeleporter = new TransporterRoom(this.aRandomizer);
5  }
```

**Note :** Pour le code de `getRandomRoom`, on peut simplifier le corps de la fonction en utilisant un `Stream`[1] de Java.

```
1  public Room getRandomRoom() {
2      if (this.aRooms.isEmpty()) return null;
3
4      int vSize = this.aRooms.size();
5
6      return this.aRooms.values()
7              .stream() // converts the collection to a stream
               ↪  of rooms
8              .skip(this.aRandom.nextInt(vSize)) // skip a
               ↪  random number of rooms
9              .findFirst() // selects only one room
10             .orElse(null); // if there is no room (an empty
               ↪  stream), returns null
11 }
```

1. Un Stream est une séquence d'éléments prenant en charge les opérations d'agrégation séquentielles et parallèles.

## Commande alea

Ayant fait l'exercice sur Command abstract avant cet exercice, on va juste créer une nouvelle classe pour une nouvelle commande `alea`.

```
1  public class AleaCommand extends Command {
2      public AleaCommand() {
3          super("alea", true);
4      }
5
6      /**
7       * Forces the exit of the current transporter room.
8       *
9       * @param pEngine    The game engine.
10      * @param pPlayer    The player using the command.
11      * @param pInterface The user interface used by the
    ↪   game.
12      * @throws UnsupportedOperationException If player is
    ↪   not using this command in a transporter room.
13      */
14     @Override
15     public void execute(Engine pEngine, Player pPlayer,
        ↪  UserInterface pInterface) throws
        ↪  UnsupportedOperationException {
16         if (!this.hasSecondWord()) {
17             pInterface.println("Vous devez spécifier la sortie
                ↪  forcée.");
18
19             return;
20         }
21
22         Room vRoom = pPlayer.getCurrentRoom();
23
24         if (!(vRoom instanceof TransporterRoom))
25             throw new UnsupportedOperationException("alea ne
                ↪  peut être utilisé uniquement que dans un
                ↪  transporter.");
26
27         TransporterRoom vTransporter = (TransporterRoom)
            ↪  vRoom;
```

```
28          vTransporter.setForcedExit(this.getSecondWord());
29
30          pInterface.printf("Sortie de %s mise sur %s.",
            ↪  vRoom.getName(), this.getSecondWord());
31          pInterface.println();
32      }
33  }
```

## Exercice 7.47 Commande abstraite

On va modifier la manière dont nos commandes sont gérés par le jeu. A la place d'avoir notre logique centralisée dans `GameEngine`, on va les éclater en classe indépendant telle que `GoCommand`, `HelpCommand`... implémentant la classe abstraite `Command`.

```
1   public abstract class Command {
2       /**
3        * Name of the command.
4        */
5       private final String aName;
6
7       /**
8        * Indicate if the command is only intended to be used
    ↪   in test.
9        */
10      private final boolean aIsTestCommand;
11
12      /**
13       * The second word provided with the command.
14       */
15      private String aSecondWord;
16
17      /**
18       * Creates a new command object.
19       *
20       * @param pName Name of the command.
21       * @param pIsTestCommand true if the command is only
    ↪   intended to be used in test context, false otherwise.
```

```
22          */
23         public Command(final String pName, final boolean
            ↪  pIsTestCommand) {
24             this.aName = pName;
25             this.aIsTestCommand = pIsTestCommand;
26 
27             this.aSecondWord = null;
28         }
29 
30         /**
31          * Creates a new command object.
32          *
33          * @param pName Name of the command.
34          */
35         public Command(final String pName) {
36             this(pName, false);
37         }
38 
39         /**
40          * Gets the second word.
41          *
42          * @return the second word.
43          */
44         public String getSecondWord() {
45             return this.aSecondWord;
46         }
47 
48         /**
49          * Define the second word of this command (the word
50          * entered after the command word). Null indicates that
51          * there was no second word.
52          *
53          * @param pSecondWord The second word.
54          */
55         public void setSecondWord(final String pSecondWord) {
56             this.aSecondWord = pSecondWord;
57         }
58 
59         /**
60          * Checks if a command has a second word
```

```
61          *
62          * @return true if the command has a second word, false
    ↪   otherwise.
63          */
64         public boolean hasSecondWord() {
65             return this.aSecondWord != null;
66         }
67
68         /**
69          * Execute the command.
70          *
71          * @param pEngine     The game engine.
72          * @param pPlayer     The player using the command.
73          * @param pInterface The user interface used by the
    ↪   game.
74          * @throws Exception If the command execution fail.
75          */
76         public abstract void execute(final Engine pEngine, final
            ↪  Player pPlayer, final UserInterface pInterface)
            ↪  throws Exception;
77
78         /**
79          * Gets the command name.
80          *
81          * @return The command name.
82          */
83         public String getName() {
84             return this.aName;
85         }
86
87         /**
88          * Checks if the command be executed in the current
    ↪   context.
89          *
90          * @param pTestMode true if the test environment is
    ↪   enabled, false otherwise.
91          * @return true if the command can be used, false
    ↪   otherwise.
92          */
93         public boolean isExecutable(boolean pTestMode) {
```

```
94          return !this.aIsTestCommand || pTestMode;
95      }
96  }
```

Par exemple, `GoCommand` sera implémenté comme ceci :

```
1   public class GoCommand extends Command {
2       public GoCommand() {
3           super("go");
4       }
5
6       /**
7        * Moves the player in another room by taking an
    ↪   available exit.
8        *
9        * @param pEngine    The game engine.
10       * @param pPlayer    The player using the command.
11       * @param pInterface The user interface used by the
    ↪   game.
12       */
13      @Override
14      public void execute(Engine pEngine, Player pPlayer,
        ↪   UserInterface pInterface) throws
        ↪   Room.RoomNotFoundException {
15          if (!this.hasSecondWord()) {
16              pInterface.println("Cette direction est
                ↪   inconnue.");
17              pInterface.printf("Vous pouvez aller : %s",
                ↪   pPlayer.getExitsDescription());
18              pInterface.println();
19              return;
20          }
21
22          pPlayer.goToExit(this.getSecondWord());
23
24          pEngine.printLocationInfo();
25      }
26  }
```

Maintenant, on doit modifier notre `CommandWord` pour prendre en comptre notre modification de comment s'organise les classes. On va utiliser ces enums pour relier les noms de commandes aux objets commandes.

```
/**
 * Representation for all the valid commands of the game.
 */
public enum CommandWord {
    GO(new GoCommand()),
    BACK(new BackCommand()),
    ITEM(new ItemCommand()),
    QUIT(new QuitCommand()),
    TEST(new TestCommand()),
    TAKE(new TakeCommand()),
    DROP(new DropCommand()),
    USE(new UseCommand()),
    HELP(new HelpCommand()),
    ALEA(new AleaCommand()),
    UNKNOWN(new UnknownCommand());

    /**
     * The command object used to process the command.
     */
    private final Command aCommand;

    /**
     * Constructor for enum's command.
     *
     * @param pCommand The command object associated.
     */
    CommandWord(final Command pCommand) {
        this.aCommand = pCommand;
    }

    /**
     * Gets the command object associated with the constant
     *
     * @return The command associated.
     */
    public Command getCommand() {
```

```
37          return this.aCommand;
38      }
39
40      /**
41       * Gets if the command is hidden.
42       *
43       * @return true if the command is hidden, false
    ↪  otherwise.
44       */
45      public boolean isHidden() {
46          return this.aCommand.getName() == null;
47      }
48
49      @Override
50      public String toString() {
51          return this.aCommand.getName();
52      }
53  }
```

## Exercice 7.48 Character

## Exercice 7.49 Moving character

## Exercice 7.50 maximum

La classe `Math` du paquetage `java.lang` possède plusieurs méthode statiques pour comparer deux valeurs et retourner la plus grande. Ces méthodes, définies par surcharge, sont nommées `max` et accessible statiquement. Pour deux entiers, la signature est la suivante :

```
public static int min(int a, int b)
```

## Exercice 7.51 static

Les méthodes `max` de `Math` ne s'appuient que sur ses deux paramètres pour déterminer le résultat. En ça, ce sont des fonctions pures car pour les mêmes arguments, le même résultat sera donné. Il est évidemment possible de les écrire sous une forme de méthode d'instance en enlevant le mot clé `static`, mais cela ne représente par d'intérêt.

## Exercice 7.52 currentTimeMillis

Pour tester le comptage de 1 à 100 par une boucle, on récupère le temps écoulé entre le début et la fin.

**Note :** Dans pratiquement tous les programmes, les perfomances sont limités par l'IO [2] notamment `System.out.*` qui effectue des opérations d'écriture. Pour ne pas biaiser le test, on ne va pas logger le comptage fait par le boucle.

```
1  public static void countFrom0to100() {
2      long vStart = System.currentTimeMillis();
3
4      for (int i = 1; i <= 100; i++) { }
5
6      long vEnd = System.currentTimeMillis();
7      System.out.printf("From 1 to 100 in %dms.", vEnd -
       ↪  vStart);
8      System.out.println();
9  }
```

Le résultat est le suivant :

```
From 1 to 100 in 0ms.
```

Ce qui n'est absolument pas étonnant étant donné la simplicité de l'opération demandée.

2. *Input Output*, les entrées et sorties.

## Exercice 7.53 Main

On crée une classe `Main` à la racine du projet

```java
import zuul.ui.javafx.JavaFX;

public class Main {
    public static void main(String[] args) {
        JavaFX.play();
    }
}
```

# 3 A savoir expliquer

**Note :** Plusieurs des exemples présentés ci-dessous sont extrait de la documentation Java.

## 3.1 Scanner

`Scanner` est une classe se trouvant dans `java.util` et qui permet d'analyser et de manipuler du texte selon plusieurs stratégies. Par exemple ligne par ligne avec `nextLine` ou encore mot par mot avec `next`. Lors de sa création, on doit préciser dans le constructeur la source des données à analyser (soit un stream comme `System.in`, soit un fichier ou encore une chaîne de caractères).

## 3.2 HashMap

Une `HashMap` est une structure de données de type dictionnaire. C'est une implémentation de l'interface `Map`. Elle permet à partir d'une clé de trouver la valeur associée.

## 3.3 Set

Un `Set` est une structure de données permettant de créer une liste d'éléments uniques, le `Set` nous donnant la garantie qu'un même élément ne peut pas être présent plus d'une fois. Il est aussi à noter qu'un `Set` n'est qu'une interface et peut donc être utilisé avec l'implémentation faite par `HashSet`.

### 3.3.1 keySet()

`keySet()` est une méthode pouvant être appelée sur une `Map`. Elle permet de récupérer un `Set` contenant les clés utilisées dans une `Map`.

## 3.4 La boucle for each

Une boucle for each s'écrit de la façon suivante :

```
1 for ( TypeElement vElement : enumerable ) {
2   // do something
3 }
```

Elle permet de parcourir les élements contenus dans un certain objet comme un `Set` ou un tableau.

## 3.5 ActionListener

`ActionListener` est une interface qui permet à une classe d'être à l'écoute de certains évènements. La classe implémentant cetter interface doit posséder une méthode `actionPerformed` et peut être enregistré comme écouteur d'évènements à l'aide de la méthode `addActionListener`.

### 3.5.1 addActionListener()

`addActionListener()` est une méthode permettant d'enregistrer la classe implémentant `ActionListener` sur laquelle appeler `actionPerformed` quand un évènement se produit.

### 3.5.2 actionPerformed()

`actionPerformed()` est la méthode qui va recevoir en paramètre un objet de type `ActionEvent`. Cette méthode sera appelée pour chaque évenement qui se produira sur les éléments graphiques auxquels elle est attachée.

### 3.5.3 ActionEvent

`ActionEvent` est l'objet qui définit un évènement s'étant produit sur l'interface graphique.

### 3.5.4 getActionCommand()

`getActionCommand()` est une méthode disponible sur un `ActionEvent` et permet de récupérer une chaîne de caractère correspondant à la commande d'action pouvant être définit sur un élement par `setActionCommand(String command)`.

### 3.5.5 getSource()

`getSource()` est une méthode disponible sur un `ActionEvent` et donne une référence de l'objet qui a émit l'évènement.

## 3.6 Stack

`Stack` est une structure de données qui fonctionne comme une pile de type LIFO [1]. Il permet d'empiler des objets d'un certain type et de conserver l'ordre d'insertion.

1. Last In First Out

### 3.6.1 push()

`push()` permet de rajouter un objet en le positionnant en haut du `Stack`.

### 3.6.2 pop()

`pop()` permet de récupérer l'élément le plus en haut du `Stack` et de l'enlever par la même occasion.

### 3.6.3 empty()

`empty()` permet de tester si le `Stack` ne contient pas d'objet et est donc vide. Il est a noter que `Stack` contient aussi une méthode `isEmpty()` faisant exactement la même chose. La raison à cette duplication est que `Collection` n'existait pas dans la version 1 du JDK, ainsi chaque classe comme `HashMap`, `Stack`, `Vector`... ont tous une méthode `empty()`. `Collection` requiert la méthode `isEmpty()` d'être implementé et `emtpy()` fût garder pour des raisons de compatibilité.

### 3.6.4 peek()

`peek()` permet de récupérer l'objet sur le haut du `Stack` sans le supprimer de ce dernier.

## 3.7 switch

Un `switch` est une instruction permettant de définir plusieurs chemins d'éxécutions en fonction de la valeur testé. La syntaxe est la suivante :

```
1 int val = 12;
2 String result;
3
4 switch (val) {
```

```
5       case 6:
6           result = "six";
7           break;
8
9       case 10:
10          result = "dix";
11          break;
12
13      case 12:
14          result = "douze";
15          break;
16
17      default:
18          result = "autre chose"
19  }
```

### 3.7.1 case

`case` permet de spécifier un nouveau chemin d'éxécution si la valeur testée correspond à la valeur attendue par le `case`.

### 3.7.2 default

`default` permet de spécifier un chemin à prendre par défaut si aucun `break` n'a été rencontré pendant l'éxécution du `switch`

### 3.7.3 break

`break` arrête l'éxécution de l'instruction du `switch`.

## 3.8 enum

Un `enum` liste toutes les valeurs possibles que pourra prendre une variable du type de l'enum, par exemple :

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}

Day today = Day.MONDAY;
```

### 3.8.1 values()

Apeller `values()` sur un enum (dans notre cas précédent `Day.values()` retourne toutes les constantes possibles de l'enum (toujours dans notre cas, nos constantes sont `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` et `SUNDAY`).

### 3.8.2 toString()

`toString()` renvoit par défaut le nom de la constante (par exemple pour le code `Day.MONDAY.toString()` on obtient "MONDAY"). L'implémentation de `toString()` peut être supplanté pour redéfinir sa valeur de retour.

### 3.8.3 attributs / constructeur

Chaque constante de l'enum peut prendre des paramètres, définit par le constructeur de l'enum. Chaque attribut est propre à chaque constante, par exemple :

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
```

```
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;   // in kilograms
    private final double radius; // in meters

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    private double mass() { return mass; }
    private double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }

    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

Ici, chacune des constantes de l'enum `Planet` contiendra une masse et un rayon qui lui seront propre.

## 3.9 Random

`Random` est une classe se trouvant dans `Java.util` permettant de créer un générateur de nombres pseudo-aléatoire.

### 3.9.1 nextInt()

`nextInt()` permet de récupérer un entier généré pseudo-aléatoirement. Sans paramètre, le résultat sera dans l'intervalle $[\![0, 2^{32}]\!]$ avec tous les nombres ayant une probabilité égal d'être généré. Si un paramètre `x` est précisé, alors cet intervalle devient $[\![0, x]\!]$.

### 3.9.2 seed

Dans un générateur pseudo-aléatoire, la sortie n'est pas réellement aléatoire mais déterminée par une formule mathématique. La seed (ou graine en français) est l'élement qui va permettre d'initialiser cette formule. Ainsi, pour reproduire deux fois la même séquence de nombres, il suffit de passer deux fois la même graine.

## 3.10 Polymorphisme

Le polymorphisme est la capacité de mettre en place des liens entre objets et de rassembler certaines caractéristiques communes, notamment avec le procédé d'héritage en Java.

## 3.11 Paquetages

Le système de paquetage en Java permet de compartimenter le code en sections logiques (un paquet pour l'interface, une pour les items, une pour les rooms etc.) plutôt que de laisser les fichiers s'accumuler à un seul endroit sans organisation.

### 3.11.1 le paquetage par défaut

Le paquetage anonyme permet d'utiliser les classes du paquet courant sans avoir à préciser son nom (il peut même ne pas en avoir, c'est alors le paquetage anonyme).

# A L'Interface Utilisateur

## A.1 JavaFX

JavaFX est un framework [1] pour créer des interfaces graphiques qui remplace historiquement le framework Swing, utilisé dans la première partie de ce projet pour construire l'IHM. JavaFX peut-être utilisé procéduralement [2] comme Swing pour créer l'interface graphique ou via la création d'un fichier à la syntaxe spécifique pour décrire la structure que prendra l'IHM.

On explorera dans cette annexe les concepts clés de JavaFX comme les notions de *bindings* par exemple. On verra aussi comment utiliser le FXML avec l'outil Scene Builder.

### A.1.1 Organisation

L'organisation d'une interface JavaFX repose sur quelques concepts particuliers que l'on retrouvera dans notre code.

**Stage** Un stage représente une fenêtre d'application. Autant de stage que nécessaire pourront être crées. Il est a noter que lors de l'initialisation de l'interface, JavaFX créera pour nous un stage nommé stage primaire.

**Scene** Un stage doit contenir une unique scène qui pourra être changé au cours de l'exécution du programme. Une scène englobe ce qui sera affiché dans un stage.

**Node** Les nodes sont tous les éléments composant l'interface graphique (boutons, zone de texte etc.) sont attachés a une scène. On nomme cet ensemble le graphe de la scène.

1. Un framework est un ensemble d'outils permettant de développer plus facilement certaines partie d'une application, ici notre interface graphique.
2. via une série d'appels de fonctions.

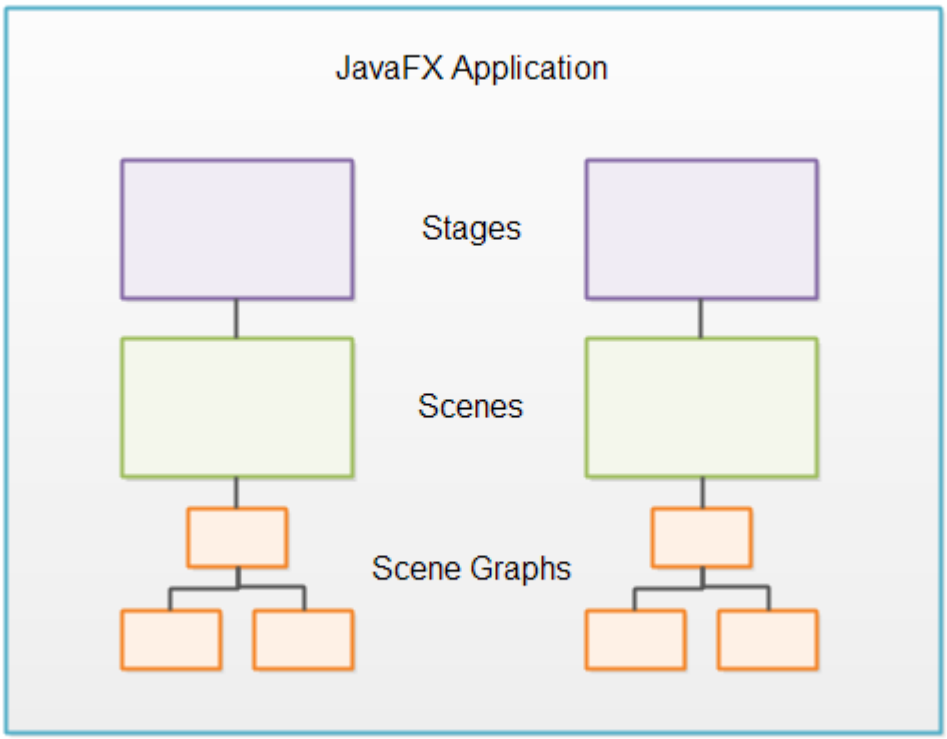


FIGURE A.1 – Architecture d'une interface JavaFX

## A.2 Base

Pour créer une interface graphique de base en JavaFX, on doit créer une nouvelle classe qui étend `Application`. Ensuite, la construction de l'interface passe par la redéfinition de la méthode `start` qui fournira en paramètre le stage primaire (discuté un peu plus haut).

Voici un code très basique montrant la création d'un simple "Hello World" en JavaFX.

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Label;
4  import javafx.stage.Stage;
5
6  public class HelloWorld extends Application {
7
8      @Override
9      public void start(Stage primaryStage) throws Exception {
10         primaryStage.setTitle("My First JavaFX App");
11
12         Label label = new Label("Hello World !");
13         Scene scene = new Scene(label, 400, 200);
14         primaryStage.setScene(scene);
15
16         primaryStage.show();
17     }
18
19     public static void main(String[] args) {
20         HelloWorld.launch(args);
21     }
22 }
```

Code A.1 – "Hello World" en JavaFX

Code qui, une fois exécuté dans BlueJ (qui intègre nativement le support de JavaFX), donne cette fenêtre.

Le code ci-dessus ne fait rien de plus que changer le titre de notre fenêtre puis crée un nouveau Label qui contient le texte que nous voulons afficher.

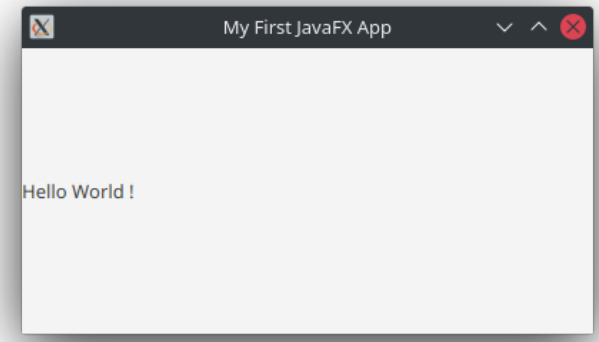


FIGURE A.2 – Application JavaFX basique.

Ensuite, il nous faut une `Scene` pour structurer notre interface. Une fois que tous les éléments sont en place, on affiche l'objet `Stage` via l'appel de `show`.

Évidement, pour agencer plusieurs éléments (une zone de texte, des boutons, des images etc.), comme pour avec `Swing`, on devra se servir des différents conteneurs de JavaFX. Voici une liste (non-exhaustive) des conteneurs que vous pouvez utiliser avec `JavaFX`. Un conteneur permet d'imposer une certaines logiques de disposition dans l'interface à ses éléments enfants (notre fenêtre de tout à l'heure est le parent de notre `Scene` qui est lui-même parent de notre `Label`).

**BorderPane** Vous avez certainement dû l'utiliser avec `Swing` dans les premières version de l'IHM. Ce conteneur permet d'agencer les élements en 5 zones distinctes : le haut, le bas, la gauche, la droite et le centre.

**HBox** Le conteneur `HBox` (pour Horizontal Box) agencera ses enfants à la suite horizontalement.

**VBox** Le conteneur `VBox` (pour Vertical Box, vous ne vous y attendiez pas) agencera ses enfants à la suite verticalement.

**GridPane** Ce conteneur permet une très grande liberté du placement des enfants. Il permet de diviser son espace en une grille et de donner à chaque enfants une position dans cette grille.

**StackPane** Ce conteneur permet la disposition de ses enfants l'un par dessus l'autre. Le premier élement sera l'élement le plus "loin" de l'écran, le dernier ajouté sera au premier plan.

Pour plus de matière sur les conteneurs, n'hésitez pas à jouer avec dans Scene Builder où de regarder la documentation de JavaFX.

## A.3 FXML

La question pouvant se poser à ce niveau est la différence entre JavaFX et Swing, utilisé précédemment pour construire l'IHM, c'est ce que nous allons discuter ici.

Vous avez pu remarquer en jouant un peu avec Swing que très vite, pour complexifier l'interface du Zuul (imaginons un ajout de boutons, d'images, de zones de d'affichage voir d'autres fonctionnalités plus ou moins avancés etc.), le nombre de lignes augmentait et le code devenait de plus en plus difficile à lire et la maintenance compliquée. On peut imaginer séparer l'interface graphique en plusieurs composants mais cela demande du temps et beaucoup de rigueur.

JavaFX vient avec sa propre solution pour faciliter la construction d'interface graphique en découplant fortement la logique (le code Java donc) de l'interface graphique (qui sera définit dans un format de balisage nommé FXML). L'avantage de définir son interface en FXML est que la hiérarchie est tout de suite visible à la lecture du fichier. Regardons un exemple de code FXML pour s'en convaincre.

Code A.2 – "Hello World" en FXML

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <?import java.lang.*?>
4  <?import java.util.*?>
5  <?import javafx.scene.*?>
6  <?import javafx.scene.control.*?>
7  <?import javafx.scene.layout.*?>
8
9  <AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320"
   ↪  xmlns:fx="http://javafx.com/fxml/1"
   ↪  fx:controller="com.example.DocumentController">
10     <children>
11         <Button layoutX="126" layoutY="90" text="Click Me!"
           ↪  onAction="#handleButtonAction" fx:id="button" />
```

```
12          <Label layoutX="126" layoutY="120" minHeight="16"
            ↪  minWidth="69" fx:id="label" />
13      </children>
14  </AnchorPane>
```

Il apparaît directement que cette interface est basé sur un parent qui est un `AnchorPane` dont on repère les dimensions. Ce parent contient en enfant un bouton et une zone d'affichage de texte.

Pour créer notre propre fichier FXML avec notre interface, on peut utiliser un outil de type drag & drop nommé Scene Builder pour concevoir facilement notre interface. Jouez un peu avec les différents élements et conteneurs.

## A.4 Lien entre FXML et code Java

## A.5 Implémentation dans le projet Zuul

# B Les dialogues

La plus grande particularité de ce jeu est qu'il est pensé pour exploiter le principe de méta-narration : le fait que le joueur soit inclus dans le jeu en tant que lui-même et qu'il soit conscient d'être dans un jeu est la base même du scénario.

Pour ce faire, j'avais pensé dès le début à l'inclusion de dialogues permettant de faire progresser l'histoire. L'objectif de cette appendice est d'explorer la mise en place d'un système de dialogue modulaire.