



ESIEE PARIS

Rapport projet Zuul

Corentin POUPRY

supervisé par
Denis Bureau

2020 - 2021

Table des matières

1	Présentation	4
1.1	Auteur	4
1.2	Thème	4
1.3	Résumé du scénario	4
1.4	Scénario détaillé	4
1.5	Plan	6
1.6	Détail des lieux, items, personnages	6
1.6.1	Lieux	6
1.6.2	Personnages	7
1.7	Commentaires	7
1.7.1	Important	7
2	Exercices	8
7.5	printLocationInfo	8
7.6	getExit	8
7.7	getExitString	9
7.8	HashMap et setExit	10
7.9	keySet	11
7.10	keySet et Javadoc	11
7.11	getLongDescription	12
7.12	Diagramme au lancement	12
7.13	Diagramme après go	14
7.14	Commande look	14
7.15	Commande search	15
7.16	showAll et showCommands	16
7.17	Changer Game ?	17
7.18	getCommandList	17
7.19	Modèle-Vue-Contrôleur	21
7.20	Item	22
7.21	Item description	22
7.22	Items	23
7.23	Commande back	24
7.24	Test de la commande back	25
7.25	back back	26
7.26	Stack	26

7.27	Réflexion sur les tests	27
7.28	Fonctionnement des tests	27
7.29	Player	28
7.30	Take et Drop	30
7.31	Plusieurs items et ItemList	33
7.32	Poids maximum	34
7.33	Inventaire	36
7.34	Cookie magique	37
7.35	zuul-with-enums-v1	39
7.40	enum/look	40
7.41	enum/help	42
7.42	Limite de temps	43
7.43	Trap Door	47
7.44	Beamer	48
7.46	Transporter	49
7.47	Commande abstraite	53
7.48	Character	58
7.49	Moving character	60
7.50	maximum	63
7.51	static	64
7.52	currentTimeMillis	64
7.53	Main	65
7.56	test static	65
7.57	Nombre d'instances	67
3	A savoir expliquer	68
3.1	Scanner	68
3.2	HashMap	68
3.3	Set	68
3.3.1	keySet()	69
3.4	La boucle for each	69
3.5	ActionListener	69
3.5.1	addActionListener()	69
3.5.2	actionPerformed()	70
3.5.3	ActionEvent	70
3.5.4	getActionCommand()	70
3.5.5	getSource()	70
3.6	Stack	70
3.6.1	push()	71
3.6.2	pop()	71
3.6.3	empty()	71

3.6.4	peek()	71
3.7	switch	71
3.7.1	case	72
3.7.2	default	72
3.7.3	break	72
3.8	enum	73
3.8.1	values()	73
3.8.2	toString()	73
3.8.3	attributs / constructeur	73
3.9	Random	75
3.9.1	nextInt()	75
3.9.2	seed	75
3.10	Polymorphisme	75
3.11	Paquetages	75
3.11.1	le paquetage par défaut	76
Appendices		77
A L'Interface Utilisateur		77
A.1	JavaFX	77
A.1.1	Organisation	77
A.2	Base	79
A.3	FXML	81
A.4	Lien entre FXML et code Java	82
A.5	Implémentation dans le projet Zuul	82

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 sont passés 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, les prenant pour les détectives ayant été appelés par le palais pour retrouver un bijou de la reine manquant.

Cependant, en inspectant une cave du palais, Murphy Law et Le Joueur se retrouve propulsé dans un vaisseau spatial en pleine bataille stellaire. Ils comprennent assez vite qu'il ne leur reste pas beaucoup de temps avant que le vaisseau n'explose. Une fois la sortie trouvée, ils se retrouvent encore une fois propulsé dans la voiture de Murphy Law, ce dernier bien décidé à faire toute la lumière sur les récents événements.

Murphy Law décide alors d'aller à l'ESIEE. Une fois arrivé là-bas 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 où Le Joueur avait lancé le jeu. Murphy Law se retrouve alors confronté au Compilateur, qui essayait de manipuler le jeu pour que l'étudiant n'obtienne pas une note à la hauteur de son travail. Murphy ressort gagnant de cette confrontation, ce qui lance le dialogue de fin de Planet Wars.

Note : En raison d'un manque de temps, je n'ai pas pu implémenté un système de dialogue assez complexe pour soutenir le scénario du jeu. Seul la partie technique est présente.

1.5 Plan

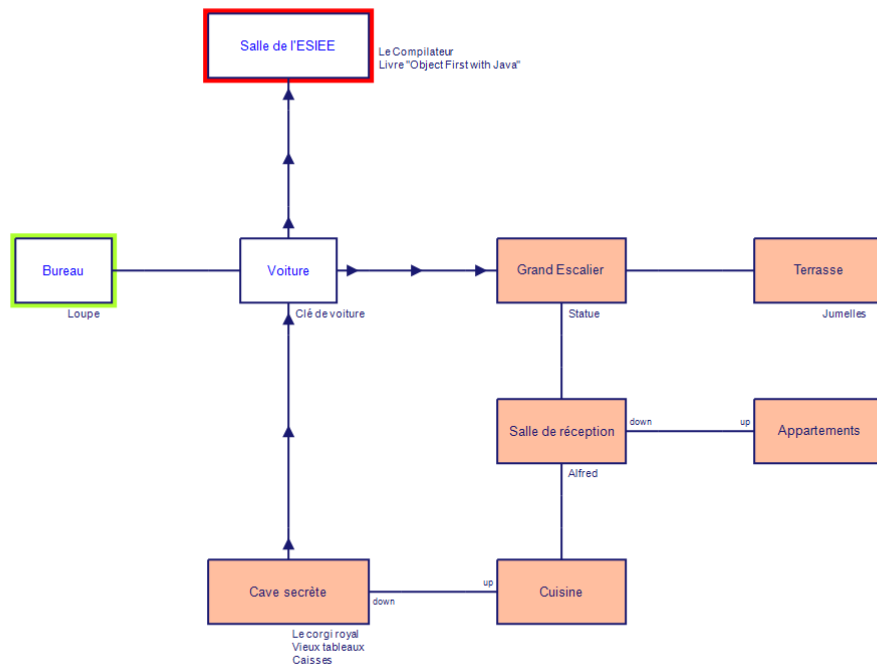


FIGURE 1.1 – Plan du jeu

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.
Palais de Buckingham	Là où loge la reine et où Murphy Law mènera une courte enquête.

Salle de l'ESIEE	C'est dans cette salle de l'ESIEE que vous avez lancé ce jeu.
Vaisseau	C'est un vaisseau spatial inspiré de l'Enterprise de Star Trek.

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 et situations sans sortir de la 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 Commentaires

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

1.7.1 Important

Pensez à compiler à la main le fichier `ZuulController` car BlueJ ne le fait pas.

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",  
4         ↪ vCurrent.getDescription());  
5  
6     System.out.print("Exits: ");  
7     if (vCurrent.aEastExit != null) System.out.print("east ");  
8     if (vCurrent.aNorthExit != null) System.out.print("north  
9         ↪ ");  
10    if (vCurrent.aSouthExit != null) System.out.print("south  
11        ↪ ");  
12    if (vCurrent.aWestExit != null) System.out.print("west ");  
13    System.out.println();  
14 }
```

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 réécrit `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",
16         ↪ vCurrent.getDescription());
17     System.out.println(vCurrent.getExitString());
18 }

```

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
2     ↪ pExit) {
3     this.aExits.put(pDirection, pExit);
4
5     return this;
6 }
7
8 public Room getExit(final String pDirection) {
9     return this.aExits.get(pDirection);
10 }

```

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 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` 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

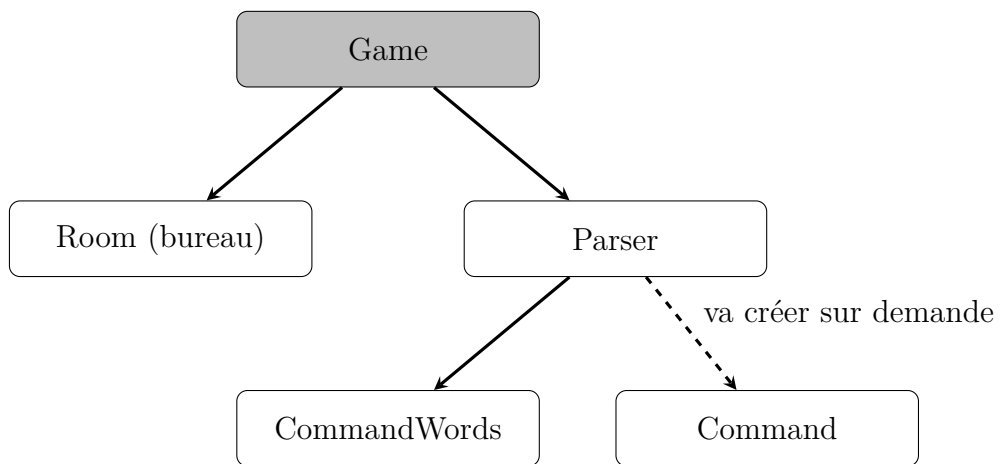


FIGURE 2.1 – Diagramme de relation entre les objets

Exercice 7.13 Diagramme après go

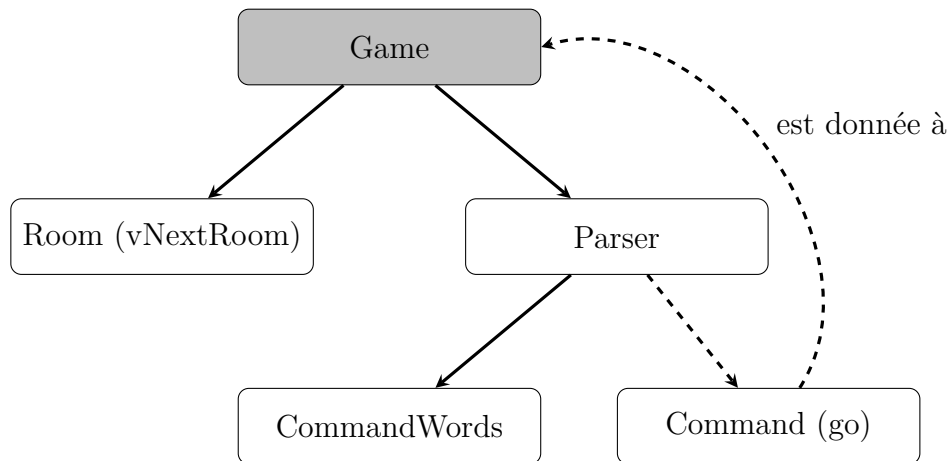


FIGURE 2.2 – Diagramme de relation entre les objets après un go

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
25                     ↪ mean...");
26                 return false;
27         }
28     }

```

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.

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

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

Exercice 7.16 showAll et showCommands

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

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 7.17 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`).

```
1 public String getCommandList() {
2     String vResult = "";
3     for (String command : CommandWords.aValidCommands) {
4         vResult += command + " ";
5     }
6
7     return vResult;
8 }
```

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

```
1 // Dans Parser.java
2 public String getCommands() {
3     return this.aValidCommands.getCommandList();
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
5         ↪ university.");
6     System.out.println("Your command words are:");
7     // "\t" représente une tabulation
8     System.out.println("\t" + this.aParser.getCommands());
9 }
```

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

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

```

4         vResult.append(vExit).append(" ");
5
6         return vResult.toString();
7     }

```

Les objets Room

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

```

1 public class Game {
2     private final HashMap<String, Room> aAllRooms;
3
4     private Room initRoom(final String pName, final String
5         ↪ pDescription) {
6         Room vCurrentRoom = new Room(pName, pDescription);
7         this.aAllRooms.put(pName, vCurrentRoom);
8
9         return vCurrentRoom;
10    }
11 }

```

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

```

1 private void createGUI() {
2     // code omitted for brevity
3
4     // add some event listeners to some components
5     this.aEntryField.addActionListener(this);
6 }

```

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 événements venant du `JTextField`. Cela est rendu possible car `UserInterface` implémente l'interface `ActionListener`.

Imaginons que nous voulions implémenter un bouton "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            ↪ {
14            this.giveIndice();
15            vButton.setVisible(false); // we allow only one
16            ↪ press for this button
17        }
18    });
19
20    vPanel.setLayout(new BorderLayout());
21    // finally, we set this button to the east of the main
22    ↪ panel
23    vPanel.add(vButton, BorderLayout.EAST);
24 }
```

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 élément 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 =
7             → (vItem.getDescription().equals(pCommand.getSecondWord()))
8                 ? vItem.getLongDescription()
9                 : "Objet inconnu. Rien a afficher\n";
10    }
11    else {
12        vToDisplay = this.aCurrentRoom.getLongDescription();
13    }
14
15    this.aGui.println(vToDisplay);
16 }

```

Exercice 7.22 Items

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

```

1 /**
2  * The items of the room.

```



```

3  */
4  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`.

```

1  private void go(final Command pCommand) {
2      if (!pCommand.hasSecondWord()) {
3          this.aGui.println("Aller où ?");
4
5          return;
6      }
7
8      Room vNextRoom =
9      ↪ this.aCurrentRoom.getExit(pCommand.getSecondWord());
10     if (vNextRoom == null) {
11         this.aGui.println("Cette direction est
12         ↪ inconnue...\n");
13         return;
14     }
15     this.changeRoom(vNextRoom);
16 }

```

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 amener 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
4         ↪ retourner.");
5         return;
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 se 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 sorte 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  
6             ↪ retourner.");  
7         return;  
8     }  
9 }
```

```

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
4              ↪ second mot.%n");
5          return;
6      }
7
8      this.aGui.println("--- DEBUT MODE TEST ---");
9
10     String vPath = "./test/" + pCommand.getSecondWord();

```

```

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
16             ↳ introuvable.%n", vPath);
17         this.aGui.println(vMessage);
18     }
19     return;
20
21     while (vScanner.hasNextLine())
22         ↳ this.interpretCommand(vScanner.nextLine());
23
24     this.aGui.println("--- FIN MODE TEST ---");
25 }

```

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)
17         ↪ this.aPreviousRooms.push(this.aCurrentRoom);
18     this.aCurrentRoom = pRoom;
19 }

```

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
7   * ↪ cannot be took.
8   */
9  public Item takeItem(final String pItemName) throws
10 ↪ CannotManageItemException {
11      if (this.aCurrentRoom == null) throw new
12 ↪ CannotManageItemException("Aucune salle dans laquelle
13 ↪ prendre l'objet.");
14
15      // remove the item from the current room...
16      Item vItem = this.aCurrentRoom.removeItem(pItemName);
17      if (vItem == null) throw new
18 ↪ CannotManageItemException("L'objet à prendre n'existe
19 ↪ pas dans la salle.");
20
21      // ...and add it to the player's inventory
22      this.aCarriedItem = vItem;
23
24      return vItem;
25  }
```

```

22  * Drops an item in the current room.
23  *
24  * @return the item that has been dropped.
25  * @throws CannotManageItemException if the requested object
    ↪ cannot be dropped.
26  */
27  public Item dropItem() throws CannotManageItemException {
28      if (this.aCurrentRoom == null) throw new
    ↪ CannotManageItemException("Aucune salle dans laquelle
    ↪ prendre l'objet.");
29
30      // remove the item from the player inventory...
31      Item vItem = this.aCarriedItem;
32      if (vItem == null) throw new
    ↪ CannotManageItemException("L'objet a déposer n'existe
    ↪ pas dans votre inventaire.");
33
34      this.aCarriedItem = null;
35
36      // ...and add it to the room
37      this.aCurrentRoom.addItem(vItem);
38
39      return vItem;
40  }

```

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.

```

1  private void drop(Command pCommand) {
2      if (pCommand.hasSecondWord()) {
3          this.aGui.println("Déposer quoi ???");
4      }

```



```

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" +
13             ↳ vError.getMessage());
14         return;
15     }
16
17     this.aGui.println("Vous avez bien déposé " +
18         ↳ vItem.getName() + ".");
19 }
20
21 private void take(Command pCommand) {
22     if (!pCommand.hasSecondWord()) {
23         this.aGui.println("Prendre quoi ???");
24         return;
25     }
26
27     Item vItem;
28     try {
29         vItem =
30             ↳ this.aPlayer.takeItem(pCommand.getSecondWord());
31     } catch (Player.CannotManageItemException vError) {
32         this.aGui.println("Impossible de prendre l'objet.\n" +
33             ↳ vError.getMessage());
34         return;
35     }
36
37     this.aGui.println("Vous avez bien récupéré " +
38         ↳ vItem.getName() + ".");
39 }

```

Exercice 7.31 Plusieurs items et ItemList

Pour porter plusieurs objets, il suffit de faire 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
5         ↳ CannotManageItemException {
6         if (this.aCurrentRoom == null) throw new
7             ↳ CannotManageItemException("Aucune salle dans
8             ↳ laquelle prendre l'objet.");
9
10        // remove the item from the current room...
11        Item vItem = this.aCurrentRoom.removeItem(pItemName);
12        if (vItem == null) throw new
13            ↳ CannotManageItemException("L'objet à prendre
14            ↳ n'existe pas dans la salle.");
15
16        // ...and add it to the player's inventory
17        this.aInventory.put(vItem.getName(), vItem);
18
19        return vItem;
20    }
21
22    public Item dropItem(final String pItemName) throws
23        ↳ CannotManageItemException {
24        if (this.aCurrentRoom == null) throw new
25            ↳ CannotManageItemException("Aucune salle dans
26            ↳ laquelle poser l'objet.");
27
28        // remove the item from the player inventory...
29        Item vItem = this.aInventory.remove(pItemName);
30        if (vItem == null) throw new
31            ↳ CannotManageItemException("L'objet a déposer
32            ↳ n'existe pas dans votre inventaire.");
33
34        // ...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
10     ↪ of the items exceeds the allowed limit
11     if (this.aWeight + vItem.getWeight() > Player.MAX_WEIGHT)
      ↪     {
12         throw new CannotManageItemException("Vous ne pouvez
      ↪         pas prendre cet objet, votre inventaire est trop
      ↪         lourd !");
13     }
14
15     // then we definitely remove it from the room
16     this.aWeight += vItem.getWeight();
17     this.aCurrentRoom.removeItem(vItem);
18
19     // ...and add it to the player's inventory
20     this.aInventory.addItem(vItem);
21
22     return vItem;
23 }
24
25 public Item dropItem(final String pItemName) throws
      ↪      CannotManageItemException {
26     if (this.aCurrentRoom == null) throw new
      ↪     CannotManageItemException("Aucune salle dans laquelle
      ↪     poser l'objet.");
27
28     // remove the item from the player inventory...
29     Item vItem = this.aInventory.removeItem(pItemName);
30     if (vItem == null) throw new
      ↪     CannotManageItemException("L'objet à déposer n'existe
      ↪     pas dans votre inventaire.");
31
32     // subtract the item weight
33     this.aWeight -= vItem.getWeight();
34
35     // ...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 affichera 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 =
5             ↪ this.aPlayer.getItem(pCommand.getSecondWord());
6
7         if (vItem == null) this.aInterface.println("Vous ne
8             ↪ possédez pas cet item.");
9         else
10            ↪ this.aInterface.println(vItem.getLongDescription());
11
12        return;
13    }
14
15    ↪ this.aInterface.println(this.aPlayer.getInventoryDescription());
16 }

```

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 uniquement 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`.

```
1  /**
2   * Uses the current item.
3   *
4   * @param vPlayer player that is using the item.
5   */
6  public void use(final Player vPlayer) {
7      throw new UnsupportedOperationException("Cet item ne peut
8          ↪ pas être utilisé.");
9  }
```

Ensuite on peut définir notre cookie magique assez facilement.

```
1  public class Cookie extends Item {
2      public final static int WEIGHT_ADDED = 10;
3
4      public Cookie() {
5          super("cookie", "Un cookie qui semble...
6              ↪ particulier.", 2);
7      }
8
9      /**
10       * Uses the cookie and modify the maximum weight that
11       * the player can be carried.
12       *
13       * @param pPlayer player that is using the item.
14       */
15      @Override
16      public void use(final Player pPlayer) {
17          int vNewMax = pPlayer.getMaxWeight() + WEIGHT_ADDED;
18
19          pPlayer.setMaxWeight(vNewMax);
20          pPlayer.deleteItem(this.getName());
21      }
22  }
```

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
4         ↪ à utiliser.");
5         return;
6     }
7     Item vItem =
8     ↪ this.aPlayer.getItem(pCommand.getSecondWord());
9     if (vItem == null) {
10        this.aInterface.println("Vous ne possédez pas cet
11        ↪ item.");
12        return;
13    }
14    try {
15        vItem.use(this.aPlayer);
16    } catch (Exception pError) {
17        this.aInterface.println(pError.getMessage());
18        return;
19    }
20    this.aInterface.printf("Vous avez bien utilisé %s.",
21    ↪ vItem.getName());
22    this.aInterface.println();
23 }
```

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`.


```

1 public void processCommand(final String pInput) {
2     this.aInterface.println("> " + pInput);
3
4     Command vCommand = Parser.parseCommand(pInput);
5
6     switch (vCommand.getCommandWord()) {
7         case GO:
8             this.go(vCommand);
9             break;
10
11         // code omitted for brevity
12     }

```

Exercice 7.40 enum/look

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

```

1 public enum CommandWord {
2     GO("go"),
3     BACK("back"),
4     LOOK("look"),
5
6     // code omitted for brevity
7 }

```

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

```

1 switch (vCommand.getCommandWord()) {
2     case GO:
3         this.go(vCommand);
4         break;
5
6     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
11         ↪ pIsHidden) {
12         this.aCommandName = pCommandName;
13         this.aIsHidden = pIsHidden;
14     }
15
16     CommandWord(final String pCommandName) {
17         this(pCommandName, false);
18     }
19
20     public boolean getIsHidden() {
21         return this.aIsHidden;
22     }
23 }

```

```

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>
3          ↪ aValidCommands;
4
5      static {
6          Map vMap = new HashMap<String, CommandWord>();
7          for (CommandWord command : CommandWord.values()) {
8              if (command.getIsHidden()) continue;
9              vMap.put(command.toString(), command);
10         }
11
12         aValidCommands = Collections.unmodifiableMap(vMap);
13     }
14 }

```

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`.

Note : Il est a noté que j'ai choisi de faire l'exercice "Commande abstraite" à ce moment là car il me semblait logique de continuer dans cette lancée avec les enums. Ceci a aussi permis de simplifier grandement certaines logiques et certains exercices.

Exercice 7.42 Limite de temps

On va ajouter la possibilité de limiter le nombre de mouvements fait par un joueur dans le jeu.

Déjà, il faut que ce système soit activable et désactivable à souhait. Je fais donc le choix de baser toute la logique du compteur de temps dans une nouvelle classe `Timer`. Aussi, j'ai fait le choix de baser la logique du limiteur de temps en utilisant des exceptions.

```
1 public class Timer {
2     /**
3      * The remaining moves of timer.
4      */
5     private int aRemainingMoves;
6
7     /**
8      * If the instance is activated or not.
9      */
10    private boolean aDisabled;
11
12    /**
13     * Creates a new disabled Timer.
14     */
15    public Timer() {
16        this(0, true);
17    }
18
19    /**
20     * Creates a new timer.
21     *
```

```

22      * @param pRemainingMoves The remaining moves that can
↳ be made.
23      * @param pDisabled      If the counter is disabled.
24      */
25      public Timer(final int pRemainingMoves, final boolean
↳ pDisabled) {
26          this.aRemainingMoves = pRemainingMoves;
27          this.aDisabled = pDisabled;
28      }
29
30      /**
31      * Indicates that a new tick has taken place and
↳ decrement the internal timer.
32      *
33      * @throws TimerLimitException If the timer does not
↳ allow any more actions.
34      */
35      public void tick() throws TimerLimitException {
36          if (this.aDisabled) return;
37
38          this.aRemainingMoves -= 1;
39
40          if (this.aRemainingMoves < 1) throw new
↳ TimerLimitException();
41      }
42
43      /**
44      * Sets the new remaining moves for the timer.
45      *
46      * @param pRemainingMoves The new remaining of moves..
47      */
48      public void setRemainingMoves(final int pRemainingMoves)
↳ {
49          this.aRemainingMoves = pRemainingMoves;
50      }
51
52      /**
53      * Gets the remaining moves.
54      *
55      * @return The remaining moves.

```

```

56     */
57     public String getRemainingDescription() {
58         switch (this.aRemainingMoves) {
59             case 0:
60                 return "Aucun mouvement restant.";
61
62             case 1:
63                 return "Un seul mouvement restant.";
64
65             default:
66                 return String.format("Il vous reste %d
67                                     ↪ mouvements.", this.aRemainingMoves);
68         }
69     }
70
71     /**
72      * Gets the state of the timer.
73      *
74      * @return true if the timer is enabled.
75      */
76     public boolean isEnabled() {
77         return !this.aDisabled;
78     }
79
80     /**
81      * Sets the activation state of the timer.
82      *
83      * @param pDisabled true if the timer should be
84      ↪ disabled, false otherwise.
85      */
86     public void setDisabled(final boolean pDisabled) {
87         this.aDisabled = pDisabled;
88     }
89
90     /**
91      * Indicates that the timer has completed its countdown.
92      */
93     public static class TimerLimitException extends Exception
94     ↪ {
95         TimerLimitException() {

```

```

93         super("La limite de mouvement est atteinte, vous
           ↪ avez perdu.");
94     }
95 }
96 }

```

L'idée est la suivante :

On va créer un nouvel attribut `aTimer` dans `Player` et dans les fonctions de déplacement tel que `toPreviousRoom` et `takeExit`, on va ajouter un simple `this.aTimer.action()`. Ces fonctions sont uniquement appelées par des commandes et les commandes sont appelées dans `GameEngine` de la manière suivante :

```

1  public void processCommand(final String pInput, final boolean
   ↪ pTestMode) {
2      this.aInterface.println("> " + pInput);
3
4      Command vCommand = Parser.parseCommand(pInput, pTestMode);
5
6      // while writing my commands, I noticed that many of them
   ↪ needed the player instance, the game engine
7      // and / or the interface used. So instead of using the
   ↪ getters I preferred to put the different
8      // elements as parameters of the `execute` method.
9
10     try {
11         vCommand.execute(this, this.aPlayer,
   ↪ this.aInterface);
12     } catch (Timer.TimerLimitException pError) {
13         // here we are in the case where the game cannot
   ↪ continue because a timer has expired
14         this.aInterface.printf(pError.getMessage());
15         this.aInterface.disable();
16     } catch (Exception pError) {
17         // a non-fatal error occurred
18         this.aInterface.println(pError.getMessage());
19     }
20 }

```

```

21     this.aInterface.println();
22 }

```

On pourra donc attraper les exceptions du `Timer` directement au niveau du `GameEngine`. Cette approche laisse une grande liberté, implémenter plusieurs `Timer`, changer la logique de gestion de l'erreur etc. Et ne complexifie pas le code de `Player` à outrance (simple rajout de `this.aTimer.action()`).

Exercice 7.43 Trap Door

Tout l'intérêt de cet exercice repose sur la suppression 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 ramène 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 vérifiera 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
6             ↪ rapidement !", 50);
7     }
8
9     @Override
10    public void use(final Engine pEngine) {
11        UserInterface vInterface = pEngine.getInterface();
12        Player vPlayer = pEngine.getPlayer();
13
14        if (this.aSavedLocation == null) {
15            this.aSavedLocation =
16                ↪ pEngine.getPlayer().getCurrentRoom();
17            vInterface.println("Beamer chargé !");
18        } else {
19

```

```

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 7.46 Transporter

On commence par définir une classe `RoomRandomizer` dont le but sera de gérer et de retourner une pièce aléatoire parmi 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 void addAll(final Room ...pRooms) {

```

```

15         for (Room room : pRooms) this.add(room);
16     }
17
18     public Room getRoom(final String pName) {
19         return this.aRooms.get(pName);
20     }
21
22     public Room getRandomRoom() {
23         if (this.aRooms.isEmpty()) return null;
24
25         int vRandom =
26             ↪ this.aRandom.nextInt(this.aRooms.size());
27         Object vRoom =
28             ↪ this.aRooms.values().toArray()[vRandom];
29         return (Room) vRoom;
30     }
31 }

```

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
8         ↪ of rooms
9         .skip(this.aRandom.nextInt(vSize)) // skip a
10        ↪ random number of rooms
11        .findFirst() // selects only one room
12        .orElse(null); // if there is no room (an empty
13        ↪ stream), returns null
14 }

```

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

Ce qui permet de se passer de l'allocation de mémoire pour un tableau à chaque appel de la méthode.

Ensuite, on définit notre nouvelle `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     ↪ {
7         super("Salle de téléportation", "Pratique pour se
8         ↪ déplacer dans le vaisseau");
9
10        this.aRandomizer = pRandomizer;
11    }
12
13    @Override
14    public Room getExit(final String pDirection) {
15        return this.aForcedExit == null
16            ? this.aRandomizer.getRandomRoom()
17            : this.aRandomizer.getRandomRoom(this.aForcedExit);
18    }
19
20    public void setForcedExit(final String pExitName) {
21        this.aForcedExit = pExitName;
22    }
23 }
```

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 }
```

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
12     ↪ game.
13     * @throws UnsupportedOperationException If player is
14     ↪ not using this command in a transporter room.
15     */
16     @Override
17     public void execute(Engine pEngine, Player pPlayer,
18     ↪ UserInterface pInterface) throws
19     ↪ UnsupportedOperationException {
20         if (!this.hasSecondWord()) {
21             pInterface.println("Vous devez spécifier la sortie
22             ↪ forcée.");
23
24             return;
25         }
26
27         Room vRoom = pPlayer.getCurrentRoom();
28
29         if (!(vRoom instanceof TransporterRoom))
30             throw new UnsupportedOperationException("alea ne
31             ↪ peut être utilisé uniquement que dans un
32             ↪ transporter.");
33
34         TransporterRoom vTransporter = (TransporterRoom)
35         ↪ 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
8       ↪ available exit.
9       *
10      * @param pEngine    The game engine.
11      * @param pPlayer    The player using the command.
12      * @param pInterface The user interface used by the
13      ↪ game.
14      */
15      @Override
16      public void execute(Engine pEngine, Player pPlayer,
17      ↪   UserInterface pInterface) throws
18      ↪   Room.RoomNotFoundException {
19          if (!this.hasSecondWord()) {
20              pInterface.println("Cette direction est
21              ↪   inconnue.");
22              pInterface.printf("Vous pouvez aller : %s",
23              ↪   pPlayer.getExitsDescription());
24              pInterface.println();
25              return;
26          }
27
28          pPlayer.goToExit(this.getSecondWord());
29
30          pEngine.printLocationInfo();
31      }
32 }

```

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

```
1  /**
2   * Representation for all the valid commands of the game.
3   */
4  public enum CommandWord {
5      GO(new GoCommand()),
6      BACK(new BackCommand()),
7      ITEM(new ItemCommand()),
8      QUIT(new QuitCommand()),
9      TEST(new TestCommand()),
10     TAKE(new TakeCommand()),
11     DROP(new DropCommand()),
12     USE(new UseCommand()),
13     HELP(new HelpCommand()),
14     ALEA(new AleaCommand()),
15     UNKNOWN(new UnknownCommand());
16
17     /**
18      * The command object used to process the command.
19      */
20     private final Command aCommand;
21
22     /**
23      * Constructor for enum's command.
24      *
25      * @param pCommand The command object associated.
26      */
27     CommandWord(final Command pCommand) {
28         this.aCommand = pCommand;
29     }
30
31     /**
32      * Gets the command object associated with the constant
33      *
34      * @return The command associated.
35      */
36     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

Note : A la place de nommer mes classes **Character**, j'ai préféré utiliser **Personage** en raison du conflit potentiel avec la classe java gérant les **char**.

On définit une class **Personage** très simpliste qui permet des opérations basiques avec les PNJ de notre jeux. On peut par exemple voir que le joueur pourra interagir avec et vice-versa via **dialog()**.

```

1 public class Personage {
2     /**
3      * Name of the personage.
4      */
5     private final String aName;
6
7     /**

```

```

8      * Description of the personage.
9      */
10     private final String aDescription;
11
12     /**
13      * Creates a new personage.
14      *
15      * @param aName      Name of the personage.
16      * @param aDescription Description of the personage.
17      */
18     public Personage(final String aName, final String
19 ↪ aDescription) {
20         this.aName = aName;
21         this.aDescription = aDescription;
22     }
23
24     /**
25      * Allows you to talk with a personage.
26      *
27      * @param pInterface GUI of the game.
28      * @param pPlayer    Player that want to talk with the
29 ↪ personage.
30      */
31     public void dialog(final UserInterface pInterface, final
32 ↪ Player pPlayer) {
33         pInterface.printf("[%s] Excusez-moi mais je n'ai rien
34 ↪ à vous dire...", this.aName);
35         pInterface.println();
36     }
37
38     /**
39      * Gets a long description about the personage.
40      *
41      * @return A description of the personage.
42      */
43     public String getLongDescription() {
44         return String.format("%s est %s.", this.aName,
45 ↪ this.aDescription);
46     }

```

```

43     /**
44      * Gets the personage's name.
45      *
46      * @return The personage's name.
47      */
48     public String getName() {
49         return this.aName;
50     }
51 }

```

Exercice 7.49 Moving character

Un `MovingPersonage` est bien évidemment une sorte de `Personage` qui a la capacité de se mouvoir. J'ai fait le choix de la simplicité et j'ai choisis une stratégie de déplacement où le PNJ va se téléporter dans des salles prédéfinis et ce, aléatoirement. J'ai pris cette décision afin que je puisse avoir un contrôle total sur les salles atteignables par le PNJ. De plus, cela me permet de réutiliser `RandomizerRoom` car son API est suffisamment abstraite pour être utilisée dans ce contexte.

L'algorithme de déplacement sera implémenté dans `move` et ce dernier est facilement remplaçable pour, par exemple, implémenté un algorithme basé sur le déplacement dans des salles adjacentes etc.

```

1  public class MovingPersonage extends Personage {
2      /**
3       * Possible rooms for the movement of the personage.
4       */
5      private final RoomRandomizer aRandomizer;
6
7      /**
8       * Current room where the personage is.
9       */
10     private Room aCurrent;
11
12     /**
13      * Creates a new personage.

```

```

14      *
15      * @param aName      Name of the personage.
16      * @param aDescription Description of the personage.
17      */
18  public MovingPersonage(String aName, String aDescription)
19      ↪ {
20      super(aName, aDescription);
21
22      this.aRandomizer = new RoomRandomizer();
23  }
24
25  /**
26   * Adds rooms to the travel possibilities.
27   * @param pRooms Rooms to be added.
28   */
29  public void addRooms(final Room... pRooms) {
30      this.aRandomizer.addAll(pRooms);
31  }
32
33  /**
34   * Moves the personage in a random room.
35   */
36  public void move() {
37      if (this.aRandomizer.size() <= 1) {
38          throw new IllegalStateException(this.getName() +
39              ↪ " doit avoir plus d'une pièce dans laquelle se
40              ↪ déplacer.");
41      }
42
43      this.aCurrent.removePersonage(this.getName());
44      Room vNextRoom = this.aRandomizer.getRandomRoom();
45      vNextRoom.addPersonage(this);
46
47      this.setCurrentRoom(vNextRoom);
48  }
49
50  /**
51   * Sets the current room of the personage.
52   * @param pRoom
53   */

```

```

51     public void setCurrentRoom(final Room pRoom) {
52         this.aCurrent = pRoom;
53         System.out.println(pRoom.getName());
54     }
55 }

```

J'en profiter pour définir un `addPersonage` sur `Room` pour pouvoir ajouter un `Personage` convenablement.

```

1  /**
2   * Adds a personage to the room.
3   *
4   * @param pPersonage personage to be added.
5   */
6  public void addPersonage(final Personage pPersonage) {
7      this.aPersonages.put(pPersonage.getName(), pPersonage);
8
9      if (pPersonage instanceof MovingPersonage)
10         ((MovingPersonage) pPersonage).setCurrentRoom(this);
11 }

```

J'aurais pu rendre le code plus simple en définissant l'attribut `aCurrent` sur `Personage` directement plutôt que `MovingPersonage` mais j'ai préféré ne pas le faire car un PNJ statique n'a pas besoin de cette donnée.

Vient maintenant un point qui m'a fait réfléchir assez longtemps. Comment déplacer tous les `MovingPersonage` à chaque fois que le joueur bouge ? L'idée que j'ai retenue qui me semble la plus élégante est de créer une liste `AllMovingPersonage` comme attribut statique de `MovingPersonage` et à chaque déplacement, une itération sera faite sur cette liste et chaque `move()` sera invoqué. Le rajout de `MovingPersonage` se fera dans le constructeur.

```

1  public class MovingPersonage extends Personage {
2      /**
3       * All instances of moving non-player.
4       */
5      private static final List<MovingPersonage>
        ↪ AllMovingPersonages = new ArrayList<>();

```

```

6
7      // code omitted for brevity
8
9      /**
10     * Creates a new personage.
11     *
12     * @param aName      Name of the personage.
13     * @param aDescription Description of the personage.
14     */
15     public MovingPersonage(String aName, String aDescription)
16     ↪ {
17         super(aName, aDescription);
18
19         this.aRandomizer = new RoomRandomizer();
20
21         AllMovingPersonages.add(this);
22     }
23
24     // code omitted for brevity
25
26     /**
27     * Moves all the non-player that can move.
28     */
29     public static void moveAll() {
30         for (MovingPersonage vPerso : AllMovingPersonages)
31             vPerso.move();
32     }
33 }

```

Il ne reste plus qu'à appeler la méthode statique `moveAll` aux endroits voulus (dans mon cas, lors d'un déplacement quelconque du joueur).

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 performances sont limitées par l'IO² notamment `System.out.println` qui effectue des opérations d'écriture. Pour ne pas biaiser le test, on ne va pas afficher le comptage fait par la boucle.

```
1 public static void countFrom0to100() {  
2     long vStart = System.currentTimeMillis();  
3  
4     for (int i = 1; i <= 100; i++) {} // noop  
5  
6     long vEnd = System.currentTimeMillis();  
7     System.out.printf("From 1 to 100 in %dms.", vEnd -  
8         ↪ vStart);  
9     System.out.println();  
10 }
```

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

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 et l'absence d'I/O qui serait le goulot d'étranglement des performances du programme.

Exercice 7.53 Main

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

```
1 import zuul.pkg_ui.pkg_javafx.JavaFX;
2
3 public class Main {
4     public static void main(String[] args) {
5         JavaFX.play();
6     }
7 }
```

Exercice 7.56 test static

Pouvez-vous appeler une méthode statique à partir d'une méthode d'instance ?

Il est évidemment possible d'appeler une méthode statique depuis une méthode d'instance, c'est notamment ce que je fais dans mon jeu quand je veux utiliser une méthode de la classe `Utils`.

```
1 private void welcome() {
2     // code omitted for brevity
3
4     String vName = this.aInterface.ask("Nom du joueur",
5     ↪ "Veuillez donner votre nom.");
```

```

5      if (!Utils.isCorrectString(vName)) { // isCorrectString
        ↳ est une méthode statique
6          throw new IllegalArgumentException("Le nom ne peut pas
          ↳ être utilisé.");
7      }
8
9      // code omitted for brevity
10 }

```

Pouvez-vous appeler une méthode d'instance à partir d'une méthode statique ?

Non, il n'est pas possible de faire ceci car, dans le contexte d'une méthode statique, `this` n'est pas défini.

```

1  class MyProgram {
2      public static void main(String[] args) {
3          this.test();
4      }
5
6      public void test() {
7          System.out.println("it's working !");
8      }
9  }

```

Ce code ne compile évidemment pas et donne le message d'erreur

error : non-static variable this cannot be referenced from a static context

Pouvez-vous appeler une méthode statique à partir d'une méthode statique ?

Il est tout à fait possible de faire ceci, imaginons une méthode calculant une factorielle par récursion.

```

1  class MyProgram {
2      public static int factorial(final int n) {
3          if (n == 1) return 1;

```

```

4
5         return n * factorial(n - 1);
6     }
7 }

```

On voit bien qu'une méthode statique peut donc appeler une méthode statique.

Exercice 7.57 Nombre d'instances

Une classe peut-elle compter le nombre d'instances qui ont été créées à partir d'elle ? Quels sont les éléments nécessaires pour cela ?

Il est tout à fait possible qu'une classe puisse faire ça, il suffit simplement que le constructeur incrémente un attribut statique qui tiendra le compte des instances créées.

```

1 public class MyProgram
2 {
3     private static int instances = 0;
4
5     public MyProgram() {
6         instances += 1;
7     }
8
9     public static int numberOfInstances() {
10         return instances;
11     }
12 }

```

Il est à noter que cet exercice m'a donné l'idée d'utiliser ce pattern pour bouger tous les `MovingPersonage` facilement et élégamment.

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 éléments 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 événements. La classe implémentant cette interface doit posséder une méthode `actionPerformed` et peut être enregistré comme écouteur d'événements 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 événement 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 événement qui se produira sur les éléments graphiques auxquels elle est attachée.

3.5.3 `ActionEvent`

`ActionEvent` est l'objet qui définit un événement 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 définie sur un élément 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 émis l'événement.

3.6 `Stack`

`Stack` est une structure de données qui fonctionne comme une pile de type LIFO¹. 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 à 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 implémenté et `empty()` fût gardé 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'exécutions en fonction de la valeur testée. 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'exécution 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'exécution du `switch`

3.7.3 break

`break` arrête l'exécution 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 :

```
1 public enum Day {  
2     SUNDAY, MONDAY, TUESDAY, WEDNESDAY,  
3     THURSDAY, FRIDAY, SATURDAY  
4 }  
5  
6 Day today = Day.MONDAY;
```

3.8.1 values()

Appeler `values()` sur un enum (dans notre cas précédent `Day.values` retourne alors 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()` renvoie 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 :

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

```

4      EARTH    (5.976e+24, 6.37814e6),
5      MARS     (6.421e+23, 3.3972e6),
6      JUPITER  (1.9e+27,   7.1492e7),
7      SATURN   (5.688e+26, 6.0268e7),
8      URANUS   (8.686e+25, 2.5559e7),
9      NEPTUNE  (1.024e+26, 2.4746e7);
10
11     private final double mass;    // in kilograms
12     private final double radius;  // in meters
13
14     Planet(double mass, double radius) {
15         this.mass = mass;
16         this.radius = radius;
17     }
18
19     private double mass() { return mass; }
20     private double radius() { return radius; }
21
22     // universal gravitational constant (m3 kg-1 s-2)
23     public static final double G = 6.67300E-11;
24
25     double surfaceGravity() {
26         return G * mass / (radius * radius);
27     }
28
29     double surfaceWeight(double otherMass) {
30         return otherMass * surfaceGravity();
31     }
32 }

```

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 $\llbracket 0, 2^{32} \rrbracket$ avec tous les nombres ayant une probabilité égal d'être généré. Si un paramètre `x` est précisé, alors cet intervalle devient $\llbracket 0, x \rrbracket$.

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'élément 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 packaging par défaut

Le packaging 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 packaging anonyme).

A L'Interface Utilisateur

A.1 JavaFX

JavaFX est un framework¹ 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 procéduralement² 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 créés. Il est à 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 à 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 parties d'une application, ici notre interface graphique.

2. via une série d'appels de fonctions.

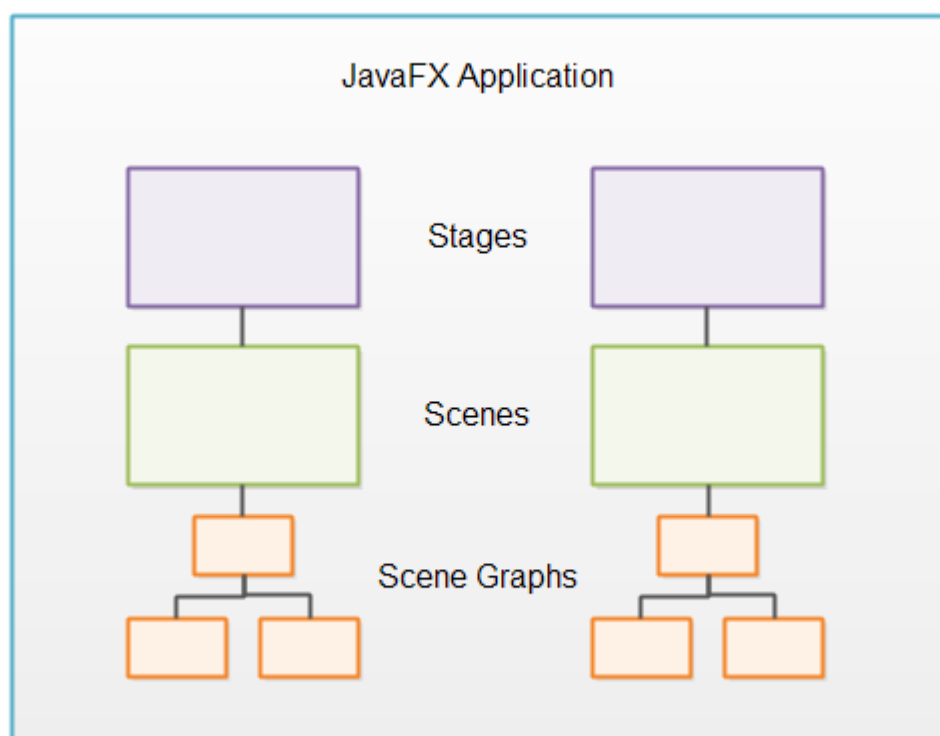


FIGURE A.1 – Architecture d’une interface JavaFX

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.

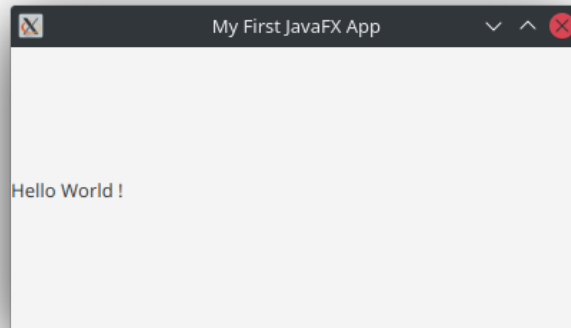


FIGURE A.2 – Application JavaFX basique.

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**.

Évidemment, 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 éléments 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 élément sera l'élément 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 défini 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"  
13             ↪ minWidth="69" fx:id="label" />  
14     </children>  
    </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 éléments et conteneurs.

A.4 Lien entre FXML et code Java

A.5 Implémentation dans le projet Zuul

En cours d’écriture, pour plus d’info, me contacter par email.