

RÉALISATION D'APPLICATION

RAPPORT ITÉRATION III

Jeu D'aventure

Zuul-bad

Auteurs
Groupe 5
LOKO Loïc
GHOUIBI Ghassen
BOUCHICHA Abdelrahim

Table des matières

I Présentation	3
II Itération I	3
II.1. Exercice 7.1	3
II.2. Exercice 7.2	5
II.3. Exercice 7.3	5
II.4. Exercice 7.4	5
II.5. Exercice 7.5	5
II.6. Exercice 7.6	7
II.7. Exercice 7.7	7
II.8. Exercice 7.8	7
II.9. Exercice 7.9	7
II.10. Exercice 7.10	7
II.11. Exercice 7.11	8
II.12. Exercice 7.12	8
II.13. Exercice 7.13	8
II.14. Exercice 7.14	8
II.15. Exercice 7.15	8
II.16. Exercice 7.16	8
II.17. Exercice 7.17	8
III Itération II	9
III.1. Exercice 7.18	9
III.2. Exercice 7.18.1	9
III.3. Exercice 7.18.2	9
III.4. Exercice 7.18.3	9
III.5. Exercice 7.18.4	9
III.6. Exercice 7.18.5	9
III.7. Exercice 7.18.6	10
III.8. Exercice 7.18.7	10
III.9. Exercice 7.18.8	10
III.10. Exercice 7.19.1	10
III.11. Exercice 7.19.2	10
III.12. Exercice 7.20	11
III.13. Exercice 7.21	11
III.14. Exercice 7.22	11
III.15. Exercice 7.22.1	11
III.16. Exercice 7.22.2	11
III.17. Exercice 7.23	11
III.18. Exercice 7.24	11
III.19. Exercice 7.25	11
III.20. Exercice 7.26	11
III.21. Exercice 7.26.1	12

IV Itération III	12
IV.1. Exercice 7.27	12
IV.2. Exercice 7.28.1	12
IV.3. Exercice 7.28.2	13
IV.4. Exercice 7.29	13
IV.5. Exercice 7.30	13
IV.6. Exercice 7.31	13
IV.7. Exercice 7.32	13
IV.8. Exercice 7.33	13
IV.9. Exercice 7.34	13
IV.10. Exercice 7.35	13
IV.11. Exercice 7.35.1	13
IV.12. Exercice 7.35.2	13
IV.13. Exercice 7.36	13
IV.14. Exercice 7.37	14
IV.15. Exercice 7.38	14
IV.16. Exercice 7.39	14
IV.17. Exercice 7.40	14
IV.18. Exercice 7.41	14

I Présentation

Dans le cadre du cours "Réalisation d'application", nous avons eu la tâche de réaliser l'itération 1 du jeu d'aventure intitulé "Zuul".

Zuul-bad est une version de world-of-zuul c'est un jeu d'aventure qui était développé en 1970 par Will Crowther et étendu par Don Woods. Le jeu original est connu sous le nom Colossal Cave Adventure.

Le principe du jeu c'est trouvé un chemin dans un système compliqué de cave autrement dit une labyrinthe avec des portes et le joueur doit essayer de trouver des mots secrets et des trésors cachés et autres. Tout en visant de récolter le maximum de point possible. Dans ce rapport on se base sur Zuul-bad c'est une implémentation avec des mauvais design de classe.

Le but de cette itération est de découvrir certains facteurs qui influencent la conception d'une classe.

Dans cette itération, en répondant aux différentes questions, nous avons découvert des principes importants dans la conception d'une classe. Ce rapport contient donc les différentes réponses aux questions.

II Itération I

II.1 Exercice 7.1

7.1.1 Que fait cette application ?

Cette application est implémenté en mode textuelle permet le joueur de se balader dans des pièces à travers l'interaction en ligne de commande (ou autres).

7.1.2 Quelles commandes le jeu accepte-t-il ?

Le jeu accepte les commandes suivantes :

go quit help

7.1.3 Que fait chaque commande ?

go : permet le joueur de se déplacer dans quatre directions *north, south, east, west* et entrée dans une chambre s'il existe.

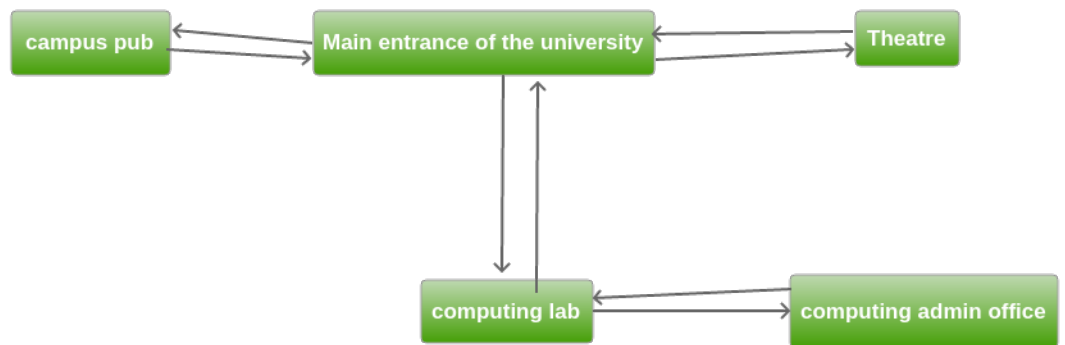
help : pour le moment cette commande permet de localiser vaguement l'endroit du joueur. *quit* : permet de quitter le jeu.

7.1.4 Combien de pièces y a-t-il dans le scénario ?

il existe 5 chambres : *outside, theatre, pub, lab, office*

7.1.5 Dessinez une carte des pièces existantes.

Plan du jeu inventé :



II.2 Exercice 7.2

Rôle des classes :

CommandWord : Cette classe permet tout simplement de tester si une chaîne est équivalente au tableau de commande qu'elle contient à travers la méthode `isCommand` qui renvoie `true` dans le cas les chaînes sont équivalentes sinon `false`.

Parser : Cette classe permet de récupérer la chaîne à partir de la ligne de commande en utilisant la bibliothèque `Scanner` à travers la méthode `getCommand` On récupère le premier mot et le deuxième et ignore le reste.

Command : Cette classe permet la vérification individuel des commandes saisies de la part de l'utilisateur décomposé en deux parties au par avant avec le `parser`.

Room : Cette classe permet la création des salles avec une petite description on constate aussi la méthode `setExits` qui permet de mettre des sorties de cette salle.

Game : C'est la classe main du jeu permet d'exécuter le jeu en boucle tant que on quitte pas le jeu tout en lisant les commandes entrées par l'utilisateur et faire les déplacements dans les salles selon les envies du joueur.

II.3 Exercice 7.3

Plan du jeu inventé :

Vous êtes le chef d'équipage d'un bateau de pirates, vous naviguez sur les mers afin de visiter différentes îles, se ravitailler en vivres ou équipements, combattre d'autres équipages et la marine afin de s'approprier et trouver les plus grands trésors dont le `One Piece`.

II.4 Exercice 7.4

Voir code.

II.5 Exercice 7.5

Voir code.

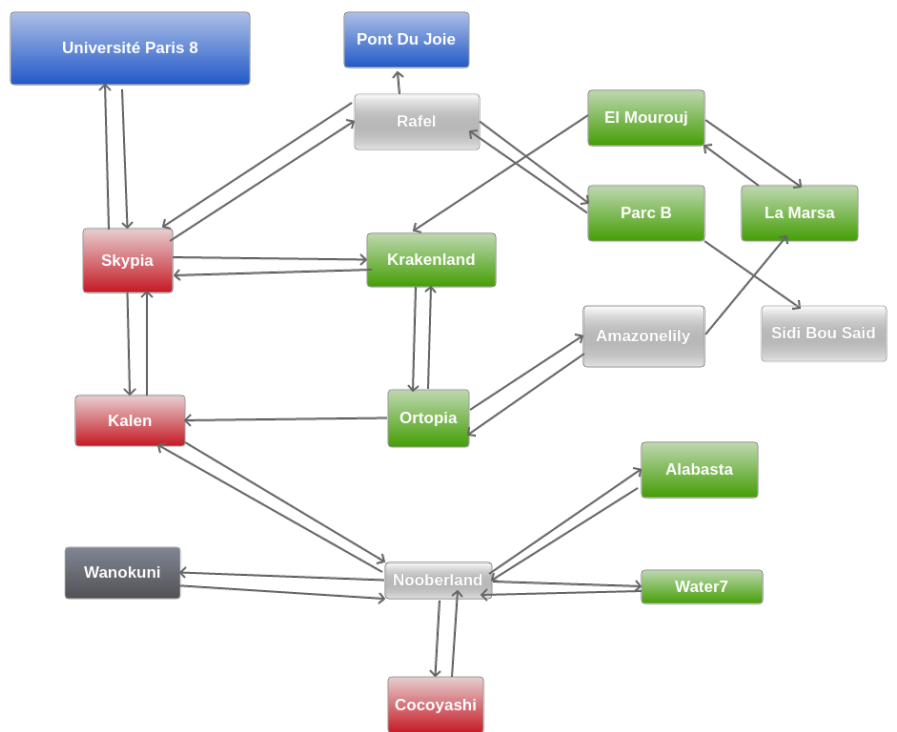


FIGURE 1 – carte du jeu

II.6 Exercice 7.6

Voir code.

II.7 Exercice 7.7

Voir code.

II.8 Exercice 7.8

Voir code.

II.9 Exercice 7.9

Recherchez la méthode `keySet` dans la documentation de `HashMap`. Qu'est ce que ça fait ?

La méthode `keySet()` est utilisée pour obtenir une vue d'un Map, Un Map est un ensemble de clé valeur, et là méthode `keySet` permet de voir nos clés. Les modifications apportées au Map sont donc reflétées dans l'ensemble, et inversement.

II.10 Exercice 7.10

Expliquez, en détail et par écrit, comment la méthode `getExitString` indiqué dans le code 7.7 fonctionne.

```
String returnString = "Exits:";
```

Cette ligne on déclare une chaîne de caractère simple.

```
Set<String> keys = exits.keySet();
```

Ici on déclare notre Set on obtient toutes les clés grâce à `keySet()` du `HashMap` `exits`

```
for (String exit : keys)
    returnString += " " + exit;
```

Ici on boucle sur ces clés et on ajoute nos clés à notre chaîne de départ `returnString`

```
return returnString;
```

Renvoie la chaîne avec les clés de `exits`.

II.11 Exercice 7.11

Voir code.

II.12 Exercice 7.12

Draw an object diagram with all objects in your game, the way they are just after starting the game

II.13 Exercice 7.13

How does the object diagram change when you execute a go command ?

II.14 Exercice 7.14

Voir code.

II.15 Exercice 7.15

Voir code.

II.16 Exercice 7.16

Voir code.

II.17 Exercice 7.17

Si vous ajoutez maintenant une nouvelle commande, avez-vous encore besoin de changer la classe de jeu ? Pourquoi ?

Oui ,dans la méthode processCommand qui permettra de tester si la commande saisie par l'utilisateur existe ou pas comme ceci :

```
if (commandWord.equals("newcommand"))  
    dosomething();
```

par contre on est plus obligé de l'écrire dans printHelp puisqu'elle s'appelle automatiquement a partir du parser :

```
public void showCommands(){  
    commands.getCommandList();  
}
```

III Itération II

III.1 Exercice 7.18

Voir code.

III.2 Exercice 7.18.1

Comparer son projet à celui-ci (gestion des sorties et des descriptions du projet zuul-better.)

La version actuelle du projet est compatible avec zuul-better. Il y a aucun warning.

III.3 Exercice 7.18.2

StringBuilder Avant d'aborder StringBuilder, rappelons nous qu'un objet de type string est non mutable, aux antipodes de ceci, StringBuffer est mutable, ce qui signifie que l'on peut changer la valeur de l'objet. Ainsi, StringBuilder sera favorable

```
public StringBuilder getExitString(){
    StringBuilder returnStringBuilder=new StringBuilder(" Exits: ");
    for(String exit : exits.keySet())
        returnStringBuilder.append(exit+" ");
    return returnStringBuilder;
}
```

Il existe aussi la méthode insert qui prend en paramètre la position et la chaîne à insérer.

Aussi delete qui permet de supprimer en revanche il prend deux arguments qui seront la position de départ et d'arrivée de la suppression.

III.4 Exercice 7.18.3

Voir dossier images.

III.5 Exercice 7.18.4

Le jeu est intitulé One Piece Treasure Cruise

III.6 Exercice 7.18.5

Voir code.

III.7 Exercice 7.18.6

Voir code.

III.8 Exercice 7.18.7

addActionListener() :

AddActionListener ajoute l'écouteur d'action spécifique pour recevoir les événements d'action de ce champ de texte. Elle prend en paramètre l'écouteur d'action à ajouter. Exemple : écouter la musique et chanter. (vouloir chanter quand on marche). écouter la musique est un événement et chanter une action on peut ajouter la marche une action, Dans ce cas on a fait addAction() si en marche on chante aussi.

actionPerformed() :

ActionPerformed prend en paramètre une action, et fait appelle à la fonction processCommand afin de créer une interface pour le champ de texte de saisie, lire la commande et faire ce qui est nécessaire pour la traiter. Ici obligatoire, car Userinterface implémente l'interface ActionListener.

Résumons :

L'interface d'écoute pour recevoir des événements d'action. La classe intéressée par le traitement d'un événement d'action implémente cette interface et l'objet créé avec cette classe est enregistré avec un composant, à l'aide de la méthode [addActionListener](#) du composant. Lorsque l'événement action se produit, la méthode [actionPerformed](#) de cet objet est appelée.

III.9 Exercice 7.18.8

voir code.

III.10 Exercice 7.19.1

voir code.

III.11 Exercice 7.19.2

voir code.

III.12 Exercice 7.20

voir code.

III.13 Exercice 7.21

voir code.

La classe Room c'est elle qui contient la fonction responsable de l'affichage
Tout simplement parce que c'est les items présent juste dans la chambre et
que cette fonction fera appel à la description de chaque item .

III.14 Exercice 7.22

voir code.

III.15 Exercice 7.22.1

L'utilisation d'un HashMap composé des clés et d'items, le choix c'est
porté sur un HashMap tout simplement pour la raison de cohérence avec les
sorties de la chambre.

III.16 Exercice 7.22.2

voir code.

III.17 Exercice 7.23

voir code.

III.18 Exercice 7.24

La commande 'back' ignore le deuxième paramètre.

III.19 Exercice 7.25

Si on tape back deux fois ça nous renvoie deux fois en arrière.

III.20 Exercice 7.26

Oui la commande back marche mais elle nous indique qu'on est au début
on peut plus reculer

III.21 Exercice 7.26.1

Documentation générer.

IV Itération III

IV.1 Exercice 7.27

Les tests unitaires et les tests de non-régression.

IV.2 Exercice 7.28.1

Créer une nouvelle commande test acceptant un second mot représentant un nom de fichier, et exécutant toutes les commandes lues dans ce fichier de texte. Avant la création de la nouvelle commande, il nous faut au préalable rajouter dans la chaîne de caractère « test » dans la liste des commandes valides de CommandWord. Ensuite nous allons créer la méthode test : Nous avons eu la charge de réaliser une nouvelle commande « test » pour effectuer des tests de commandes pour notre jeu. En effet, lorsqu'un code est changé, il est probable que des erreurs soient introduites dans celui-ci, il est donc important de procéder avec prudence et d'établir une série de tests de notre programme. La méthode test prend en paramètre un fichier texte dans lequel sera inscrit à chaque ligne les différentes commandes de notre jeu. Pour réaliser cette fonction nous avons employé les méthodes hasNext, next et exceptions qui sont utiles dans la lecture des fichiers de texte. Tout d'abord, l'usage de la méthode hasNext nous permet de vérifier si notre fichier contient une ligne suivante, elle va retourner true si notre itération contient d'autres éléments (lignes à parcourir), ici nous lirons notre fichier tant qu'il contient des lignes à parcourir. Ensuite nous découvrons la méthode next qui va tout simplement récupérer la ligne suivante afin de l'interpréter. Enfin pour vérifier que le fichier a bien été lu, nous faisons appel à exception, une exception est une erreur qui survient dans un programme, généralement elle produit l'arrêt de celui-ci. Dans notre code nous pratiquons la capture d'exception qui généralement sert à repérer un morceau de code qui pourrait générer une exception, pour notre part FileNotFoundException signalera qu'une tentative d'ouverture de fichier a échoué, cette exception sera levée par les constructeurs FileInputStream, FileOutputStream et RandomAccessFile lorsqu'un fichier avec le chemin spécifié n'existe pas et sera également lancée par eux si le fichier existe mais qu'il est cependant inaccessible (Par exemple lors d'une tentative d'ouverture en lecture seule mais pour l'écriture).

IV.3 Exercice 7.28.2

Deux fichiers créer mais vu que le scénario n'est encore claire parce que on a besoin des enemies pour ganger dans notre cas.

IV.4 Exercice 7.29

voir code.

IV.5 Exercice 7.30

voir code.

IV.6 Exercice 7.31

voir code.

IV.7 Exercice 7.32

voir code.

IV.8 Exercice 7.33

voir code.

IV.9 Exercice 7.34

voir code.

IV.10 Exercice 7.35

La modification des fichiers de tests accueil bien le take,drop,eat,check

IV.11 Exercice 7.35.1

voir code.

IV.12 Exercice 7.35.2

voir code.

IV.13 Exercice 7.36

voir code.

IV.14 Exercice 7.37

Pour le moment on a décidé de laisser les mots clés jusqu'à l'établissement des ennemis vu que notre but c'est la réalisation du jeu One Piece.

IV.15 Exercice 7.38

Oui ça marche puisque le type est enum donc on peut avoir même GO("avancer") par exemple

IV.16 Exercice 7.39

Pour nous les directions seront plus ou moins les mêmes donc pas besoin de changer en autres mots. Et tout simplement parce que c'est des directions du bateau.

IV.17 Exercice 7.40

Oui, et bien sûr ne pas oublier de l'implémenter dans le switch de interpretCommand

IV.18 Exercice 7.41

Oui ça change automatiquement.