Réalisation d'application: Jeux d'aventure

Groupe 6:

- BERRAI Dyhia N°15608005

- HUARD Alicia N°15601834

Année 2019 ~ 2020

Introduction

Suite aux évènements du virus COVID-19, nous avons eu quelques soucis dans notre groupe (le groupe 6). Nous n'avons plus de nouvelles de deux de nos camarades CASTANER Ophélie et OULD OULHADJ Lisa, nous nous sommes donc retrouvées à deux.
Nous avons donc décidé de reprendre les projets depuis le début, pour avoir une meilleure compréhension des sujets.

Itération 1 :

Exercice 7.1

- What does this application do?

Cette application est implémenté en mode textuelle, elle permet au joueur de se balader dans des pièces à travers des commandes textuelles.

- What commands does the game accept?

Le jeu accepte les commandes : go, quit, help

- What does each command do?

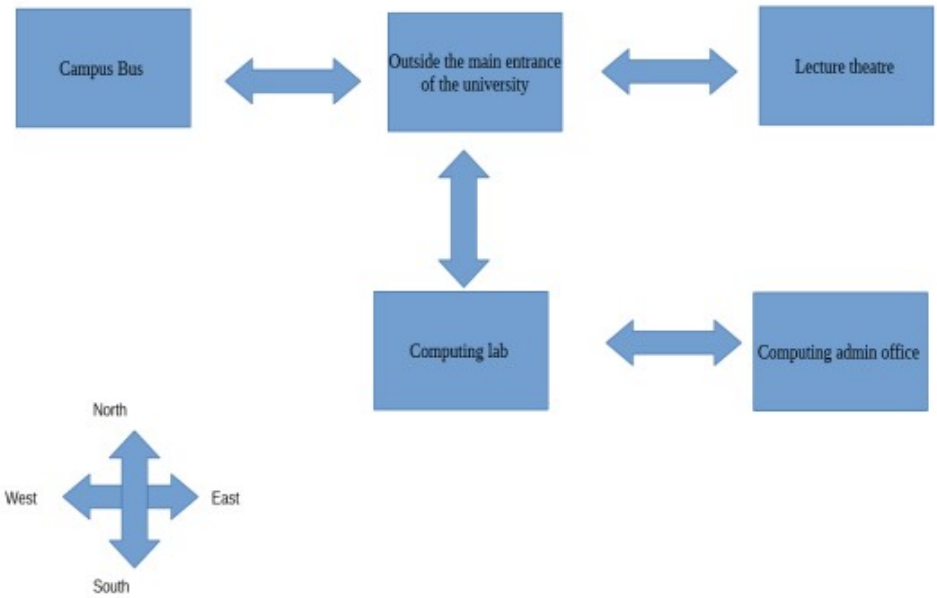
- **go :** permet au joueur de se déplacer dans quatre direction (north, south, east, west)
- **help :** pour le moment cette commande permet de localiser vaguement l'endroit où se situe le joueur.
- **quit :** permet de quitter le jeu.

- How many rooms are in the scenario?

Il y a cinq pièces:

- Computing lab
- Outside the main entrance of the university
- lecture theater
- the campus pub
- the computing admin office

Voici la map.



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 via la méthode isCommand qui renvoie «true» dans le cas où les chaînes sont équivalentes sinon «false».

Parser :

La classe récupère la chaîne de caractère, si c'est un mot, ça le met dans la variable word1, si c'est deux mots, le second mot, sera dans word2. Cette classe fait appelle à la fonction IsCommand(). Pour récupérer les mots, on utilise Scanner(System.in).

Command :

Cette classe permet la vérification individuel des commandes saisies de la part de l'utilisateur. Le premier mots est une commande et le second un mot clé.

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, elle permet d'exécuter le jeu en boucle tant que on «quit» pas le jeu toute en lisant les commandes entrées par l'utilisateur et faire les déplacements dans les salles selon les envies du joueur.

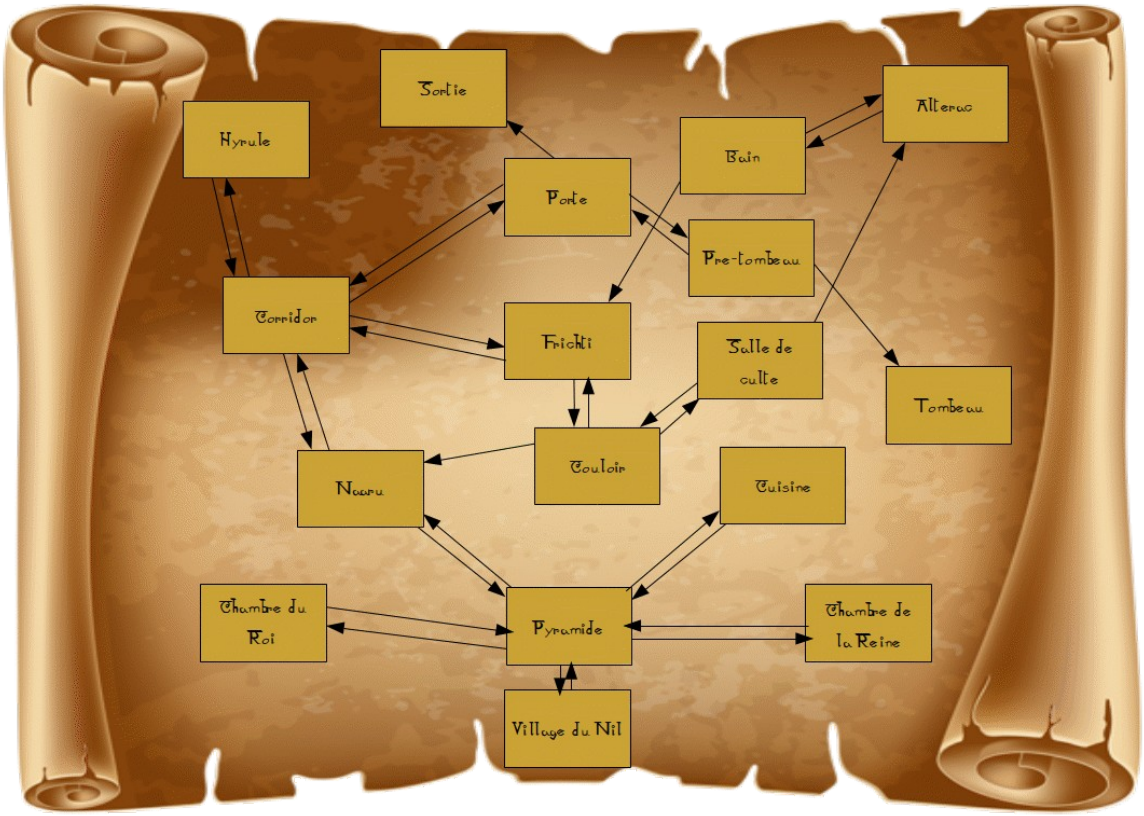
Exercice 7.2.1 :

- 1er étape : déclaration privée d'une variable reader de type Scanner
- 2ème étape : Dans le constructeur Parser() la variable reader récupère 1 ou 2 mots à l'aide de la fonction Scanner(System.In)
- 3ème étape : Méthode Command getCommand() Scanner Tokenizer = new Scanner(inputLine)

En conclusion, la classe Scanner permet de lire les saisies clavier que l'on écrit. Ceci ici, ce sera les commandes autorisées/comprises par le jeu.

Exercice 7.3 (le scénario):

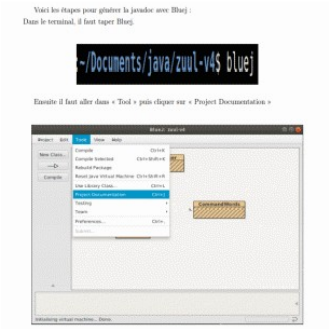
Vous êtes en pleine période de l'Égypte antique. Vous êtes un archéologue et vous vous êtes mis en tête que vous étiez l'élu pour récupérer une célèbre amulette, celle de l'œil Oudjat. Cette amulette est séparée en 4 morceaux et se trouve dans la pyramide d'Amaithor. Personne jusqu'alors n'y est parvenu.

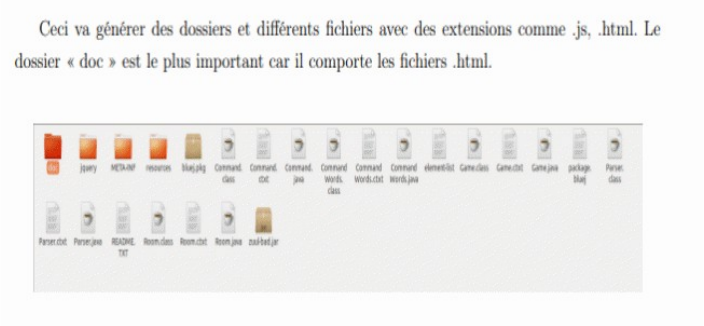


Exercice 7.9:

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

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





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.equal ( ”newcommand ” ) )
        dosomething ( ) ;
```

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

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

Itération 2:

Exercice 7.18:

Comparer son projet à celui de (gestion des sorties et des descriptions du projet zuul-better). La version actuel du projet est compatible avec zuul-better. Il y a aucun warning.

Exercice 7.18.5:

Jusqu’à présent, nous savons que les pièces du jeu sont créées dans la méthode "CreateRoom". Cette méthode se trouve dans la class "Game". Nous savons également que toutes les pièces du jeu ne seront disponibles qu’à l’intérieur de cette même classe. Il faut alors y ajouter une "HashMap" à cette classe "Game" pour accéder à toutes les pièces depuis n’importe quelle classe. De plus cette "HashMap" devra être privée "private" de préférence.

Exercice 7.18.7:

Cette partie concerne la gestion d'événements. " JTextField" permet entrer un texte sur une seule ligne. Par défaut, on ne peut écrire que trente-quatre caractères. Ce nombre pourra être modifier pour nos besoins. La méthode addActionListener() permet de préciser la classe "UserInterface" qui va gérer l'événement , ici se sera d'écrire du texte (les directions et les commandes). La méthode actionPerformed( ) reçoit les évènements de l'interface ActionListener.

Exercice 7.19 (MVC):

On ne garde pas la méthode MVC, on reste en standard.

Exercice 7.19.2:

Toutes les pièces ont des images à elles. Nous n'avons pas besoin de fabriquer des "image de mots".

Exercice 7.20:

On crée d'abord une classe "Item".

Exercice 7.21:

La classe Room 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 .

```
" public void addItem(final String pName,final Items pItem)
        Inventaire.put(pName,pItem) ; "
```

Exercice 7.23 (back):

Nous allons rajouter une nouvelle commande "back"

Exercices 7.24 (back test):

Dans cette partie, il faut tester si le jeu est jouable. Le mieux sera donc de créer un fichier "test.txt" et d'écrire des instructions comme si on jouait. Pour l'instant il faut surtout écrire la fonction de test.

Ainsi pour pouvoir essayer un fichier, il suffit de faire : test nomdufichier et de faire Entrer.

Itération 3:

Exercice 7.27:

Les tests unitaires et les tests de non-régression.

Exercice 7.28:

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 chaine de caractère test dans la liste des commandes valides de CommandWord.

Ensuite nous avons créé la méthode test : Nous avons eu la charge de réaliser une nouvelle commande test pour effectuer des test 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 à échouer, 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 pas pour l'érituré).

Exercice 7.36:

Commande «look»

Exercice 7.37:

Do you only have to edit the CommandWords class to make this change work?

NON

Itération finale :

Exercice 7.43:

Dans cet exercice, nous avons créé une pièce franchissable dans qu'un seul sens, la solution de cet exercice consiste à créer une direction permettant d'aller dans une room, mais de ne pas créer la direction inverse. Par exemple il serait possible de rentrer dans une salle avec go nord mais ne pas pouvoir faire go sud.

Exercice 7.44:

Il nous a été demandé d'ajouter un téléporteur qui doit pouvoir être ramassé dans une première pièce, puis pouvoir être chargée dans une deuxième pièce et enfin déclenché dans une 3e, il est réutilisable, mais doit être rechargement au préalable.

Puis, dans la classe GameEngine, nous avons ajouté une méthode charge() et une méthode fire(), si la première nous permet de nous souvenir de notre pièce donc de la charger, la deuxième elle nous permet d'effectuer le télé portement.

Exercice 7.45.1:

Fichier test

Apprentissage

La Classe Random Java permet de générer un flux de nombres pseudo aléatoires.

La classe utilise une graine de 48 bits, qui est modifiée à l'aide d'une formule congruentielle linéaire. La méthode nextInt(int n) est utilisée pour obtenir une pseudo-aléatoire, uniformément distribuée entre 0 (inclus) et la valeur spécifiée (exclusive), tirée de la séquence de ce générateur de nombres aléatoires. 18 Le seedest la valeur initiale de l'état interne du générateur de numéro pseudo-aléatoire qui est maintenu par la méthode next(int).

Donc est un point de départ, à partir duquel quelque chose grandit. Dans ce cas, une séquence de nombres. Ceci peut être utilisé soit pour générer toujours la même séquence (en utilisant une semence constante connue), ce qui est utile pour avoir un comportement déterministe. C'est bon pour le débogage, pour certaines applications réseau, la cryptographie, etc.

Exercice 7.46:

Nous avons changer le téléporter en une salle de téléportation. Chaque fois que le joueur entre dans cette pièce, il est transporté au hasard dans l'une des autres pièces. Pour notre part, nous avons créé une classe TransporterRoom héritant de la classe Room, celle-ci contient un constructeur, une méthode findRandomRoom() retournant une pièce aléatoire en utilisant le Scénario donné dans le constructeur de cette classe retourne une pièce aléatoire en utilisant la méthode findRandomRoomRoom().

Exercice 7.48:

Ajout des personnages au jeu. Les personnages sont similaires aux objets, mais ils peuvent parler. Ils parlent un peu de texte quand nous les rencontrons pour la première fois, et ils peuvent nous donner de l'aide si nous leur donnons le bon article. La classe Character a été créée dans le jeu. Elle crée des personnages caractérisés par un nom, la room actuelle, une String faisant référence à son dialogue et l'item qu'il peut recevoir de l'utilisateur.

Conclusion:

Pour une meilleure organisation nous avons décidé de séparer les fichiers par package. Vous pouvez lancer le jeu avec BlueJ ou directement sur le terminal avec la commande:

```
~/Jeu_aventure_BH$ javac Main.java
~/Jeu_aventure_BH$ java Main
```