Rapport d’A3P

fillMyClass()

I. Première partie

A. Auteurs

Wallerand Delevacq, élève de la promotion 2020 ESIEE Paris. Passionné d'informatique.
Connaissances en développement Web, HTML, CSS, PHP, JS, SQL, Java Android.
Directeur des systèmes informatique de l'association Esieespace.
Développeur de l'application android officielle de la liste BDE Hélios
Responsable Web du club Fonzy.

B. Thème

You need to write your own Class !

Vous êtes dans la peau d'un étudiant de première année qui apprend à développer en Java, votre but est de retrouver les différentes commandes à écrire pour compléter la class qui vous permettra de passer au niveau suivant.

C. Résumé du scénario

Comment gagner ?

Pour pouvoir gagner une partie de ce jeu, vous allez devoir retrouver les différentes commandes qui vont composer la class que vous allez devoir créer pour compléter le niveau. Ces commandes seront soit des mots soit des commandes simples. Les différents niveaux seront composés de salles représentants les endroits où l'on peut réellement trouver de l'aide (icampus, forum, internet, livre, etc.) chaque niveau sera un étage auquel on peut passer en entrant la bonne commande.

D. Plan

E. Détail des lieux, items, personnages

Les différents lieux sont donc tout d'abord la salle de cours (Classroom), Icampus, Internet, le livre (Book), le Forum. La pièce de " l'étage supérieur " est le niveau suivant (le lieu qui servira à la validation des items récupérés).

Les items sont le Clavier, le Cours, le Compilateur, Bluej, la Methode, la Variable.
C'est différents items sont nécessaires pour créer la Class que nous devons compléter.

L'unique personnage est l'élève qui est animé par le joueur.

F. Situations gagnantes et perdantes

La situation gagnante est lorsque le joueurs a récupéré tous les items et les a déposé dans la pièce du niveau supérieur.

La situation perdante est lorsque le joueurs n'a pu récupérer tous les items et les déposer dans la pièce supérieur. (lorsque par exemple il n'aura pas obtenu l'item en répondant mal à une énigme).

II. Réponses aux exercices

Exercice 7.0

Le premier exercice a consisté à créer la page web de notre projet sur notre page perso. l'adresse est http://perso.esiee.fr/~delevacw

Exercice 7.1

introduction aux différentes notions et termes du Java : Cohésion, couplage, réingéniere, Objet, Class, etc.

Exercice 7.1.1

Choix du thème du jeu : Un étudiant devra compléter une Class en ramenant les bons items pour passer au niveau suivant.

Exercice 7.2

Exercice 7.2.1

Découverte de la classe Parser et de l'objet Scanner afin de pouvoir saisir des informations au clavier.

Exercice 7.3

Conception du scénario du jeu (Voir section Résumé du scénario).

Exercice 7.3.1

Conception de ce rapport sur le modèle décrit par Denis Bureau.

Exercice 7.3.2

Réalisation du plan du jeu (Voir section Plan).

Exercice 7.3.3

Conception d'un sous ensemble du scénario avec seulement deux pièces afin de simplifier le développement.

Exercice 7.4

Introduction au couplage et à la cohésion.

Exercice 7.5 printLocationInfo()

On crée une méthode printLocationInfo() dans la classe Game afin d'éviter une dupication de code. En effet, ces informations sont nécessaires dans la méthode printWelcome() mais aussi dans la méthode goRoom().

Exercice 7.6 getExit()

On ajoute une méthode getExit() dans la classe Room afin de pouvoir récupérer les différentes sorties possibles dans une Room.

Exercice 7.7 getExitsString()

On modifie la méthode printLocationInfo() de la classe Game puis on ajoute une fonction getExitsString() dans la classe Room, elle renvoi les sorties au lieu de les afficher.

Exercice 7.7.1

Mise à jour du rapport.

Exercice 7.8 HashMap, setExit()

Utilisation d'une HashMap afin de facilement stocker les sorties d'une Room. La méthode setExit() permet d'ajouter une sortie à une Room.

Exercice 7.8.1

Nous pouvons maintenant ajouter dans le scénario un déplacement vertical vers la salle du niveau suivant.

Exercice 7.9 keySet()

La méthode keySet() de la classe HashMap nous permet de récupérer toutes les clefs d'un tableau associatif. Une HashMap étant un tableau associatif.

Exercice 7.10 getExitString()

```
public String getExitString()
{
        String returnString = "Sorties :";
        Set<String> keys = exits.keySet();
```

```
        for(String exit : keys){
                returnString += ' ' + exit;
        }
        return returnString;
}
```

La méthode getExitString() initialise la String de retour avec la valeur “Sorties :”. Elle appelle ensuite la fonction keySet() pour mettre toutes les clés des sorties de la pièce courante dans une variable keys. Set est un ensemble où chaque élément est unique. keys est ensuite utilisé dans une boucle foreach. Ainsi on associe à chaque itération de keys la String correspondante que l’on stocke dans exit. À chaque boucle, on ajoute exit à returnString.
Lorsque chaque clé a été examinée, on sort de la boucle et on fini par renvoyer la variable returnString qui comporte maintenant toutes les sorties formatées.

Exercice 7.10.1

Complétion de la javadoc.

Exercice 7.10.2

Génération de la javadoc.

Exercice 7.11 getLongDescription()

On ajoute la méthode getLongDescription() à la classe Room puis on modifie la méthode printLocationInfo().

Exercice 7.14 look()

Cette méthode permet de ré-afficher les informations de la pièce courante. On ajoute cette commande à la classe CommandWords.

Exercice 7.14.1

Modification de look() afin qu’elle affiche les informations d’un item passé en second mot de la commande.

Exercice 7.15 eat()

Cette méthode de la classe Game affiche une simple phrase. On ajoute le mot de commande dans CommandWords.

Exercice 7.16 showAll(), showCommands()

Résolution de problème de couplage implicite avec la méthode help() qui n’affiche pas automatiquement les nouvelles commandes créées. On crée la méthode showCommands() dans la classe Game qui permet d’appeler la méthode showAll() de la classe CommandWords. Cette dernière affiche toutes les commandes valides disponibles.

Exercice 7.17

Il faut simplement ajouter la méthode nécessaire à cette nouvelle commande ainsi que sont CommandWord associé.

Exercice 7.18 getCommandList()

La méthode getCommandList() permet à la classe CommandWords de produire la liste des commandes au lieu de l'afficher.
La méthode showAll() renvoit directement la String des commandes disponibles. Cette String est affichée dans la méthode printHelp() de la classe Game.

Exercice 7.18.1

Le projet correspond bien au projet Zuul-Better.

Exercice 7.18.3

Recherche et création des images du jeu.

Exercice 7.18.4

Choix du titre du jeu : fillMyClass()

Exercice 7.18.5

Non réalisation de cette exercice car d'après la lecture de la suite du projet, accéder aux Room créées dans createRooms() ne me semble pas utile ni pertinent.

Exercice 7.18.6

Étude du projet Zuul-with-images et modification de fillMyClass() afin d'utiliser les images.

Exercice 7.18.7

element.addActionListener(listener);

enregistre que c'est l'objet listener qui va réagir aux évènements survenant sur l'élément. Dans notre cas l'élément est la zone de saisie et le listener l'objet UserInterface.

La méthode actionPerformed d'une classe va être appelée dès qu'un évènement est détecté sur un composant qui a pour écouteur un objet de cette classe.

Exercice 7.18.8

Ajout d'un bouton en borderlayout.EAST.
Inclusion d'un gridlayout dans le borderlayout.EAST afin de pouvoir placer les boutons help, look, eat, quit.

Exercice 7.19.2

Création du dossier ./images/ afin d'y déplacer toutes les images du jeu.

Exercice 7.20

Création d'une nouvelle classe Item comportant deux attributs (aWeight, aDescription) avec leurs getters & setters respectifs.

Exercice 7.21

Création d'une méthode permettant d'obtenir la description d'un item. C'est la classe GameEngine qui se chargera de l'afficher.

Exercice 7.22

Ajout d'une HashMap dans la classe Room afin de contenir les items de la pièce.

Exercice 7.22.1

La HashMap est un choix judicieux car elle permet d'associer une clé à une valeurs ainsi on associe un nom cours à l'item de notre choix et on peut donc facilement le récupérer.

Exercice 7.22.2

Ajout des items dans les différentes pièces du jeu.

Exercice 7.23 back()

Ajout de la méthode back() à GameEngine ainsi que d'un attribut aPreviousRoom afin de conserver la pièce précédente.

Exercice 7.24

Ajout d'une condition qui empêche l'utilisation d'un second mot avec la commande back.

Exercice 7.26

On crée une Stack qui stocke progressivement les différentes Room et qui permet donc d'effectuer des back() successifs.

Exercice 7.26.1

Génération des deux javadoc.

Exercice 7.27

Il serait pertinent de tester dans notre jeu les différents déplacements, commandes, et dans un futur proche, l'interaction avec les PNJ et les objets.

Exercice 7.28

L'automatisation des tests se ferait simplement en créant une commande test qui ouvrirait un fichier texte puis passerai chacune de ses ligne en paramètre d'interpretCommand().

Exercice 7.28.1

Création d'un commande test qui va parcourir chaque ligne d'un fichier .txt et exécuter les commandes correspondantes.

Exercice 7.28.2

Création de deux fichiers de commandes test répondant à différents scénarios.

Exercice 7.29

Création de la classe Player. Un objet Player comporte donc sa salle courante, les salles visitées précédemment, le poids maximum transportable, les items portés.

Exercice 7.30

Création des méthodes pickUpItem() et dropItem() afin de pouvoir récupérer un objet dans une pièce.

Exercice 7.31

Déjà traité car utilisation d'une HashMap dès le début de l'exercice 7.29 en prévision de ce scénario.

Exercice 7.31.1

Création de la classe ItemList afin de simplifier l'utilisation des listes d'items.

Exercice 7.32

Ajout d'un attribut qui permet de stocker un nombre limité d'objets et des méthodes en conséquence afin de calculer si l'on a assez de place pour prendre un objet.

Exercice 7.33

Création de la commande "inventory" qui permet d'afficher l'inventaire du joueur.

Exercice 7.34

Création du MagicCookie qui permet de doubler la capacité de l'inventaire quand il est mangé.

Exercice 7.34.1

Mise à jour des fichiers de test.

Exercice 7.34.2

Mise à jour des javadocs.

Exercice 7.35

Intégration des enum dans le projet, modification des classes existantes pour l'adapter à cette nouvelle configuration.

Exercice 7.35.1

Remplacement des if, else if par des switch.

Exercice 7.40

Étude des sources de zuul-with-enums-v2.jar et réalisation de l'exercice.

Exercice 7.41

Changement du mot associé à la commande help et observation de la réflection automatique dans le jeu.

Exercice 7.41.1

Merge de zuul-with-enums-v2.jar et fillMyClass().jar

Exercice 7.42

Ajout d'une limite de temps sous forme de décrémentation d'un compteur à chaque changement de pièce.

Exercice 7.42.2

Étude de la pertinence d'un changement d'IHM complexe. Je remet cet exercice à plus tard afin de me concentrer sur le reste du projet. J'ai également découvert une librairie pour les IHM, javaFX. Cette librairie semble être intéressante et plus moderne que Swing. Je pense l'utilisé par la suite l'implementation se rapprochant de celle des View d'android, un portage du jeu pourrait être envisagé.

Exercice 7.43

Suppression de la porte Forum → Icampus afin de réaliser l'exercice TrapDoor. Seul le chemin Icampus → Forum reste possible. Modification de la commande back afin de tester si aLastRoom.peek() == null. Modification de goRoom() afin d'ajouter une room null à aLastRoom si c'est une TrapDoor.

Exercice 7.44

Mise en place du Beamer : création de la sous classe d'Item Beamer. Création des méthodes fire et charge dans player.

Exercice 7.45.1

Modification des fichiers de test afin de tester le Beamer.

Exercice 7.45.2

Re-génération des fichiers JavaDoc.

Exercice 7.46

Création d'une classe RoomRandomizer qui renvoi une Room au hasard. Création d'une TransporterRoom qui utilise RoomRandomizer.

Exercice 7.46.1

La commande aléa est fonctionnelle et permet de tester le jeu avec la TransporterRoom.

Exercice 7.46.2

Le choix de l'héritage avait déjà été fait auparavent.

Exercice 7.46.3

Ajout de certains commentaire javadoc dans le projet.

Exercice 7.46.4

Génération des deux javadocs pour le projet.

Exercice 7.47

Mise en place du système de Command avec héritage. Cela permet à présent de créer une nouvelle commande directement dans une classe Java. De plus la méthode interpretCommand de GameEngine est simplifiée.

Exercice 7.47.1

Création de package pour le projet :

- pkg_commands
- pkg_game
- pkg_item
- pkg_player
- pkg_room

Exercice 7.47.2

Génération des javadocs en prenant en compte les packages.

Exercice 7.48

Création de la classe Character qui hérite de Item.
Modification de Parser et Command pour accepter une commande du type “give ITEM to CHARACTER”.

Exercice 7.49

Création de la classe MovingCharacter héritant de Character afin de pouvoir avoir dans le jeu des personnages se déplaçant selon un chemin défini. Ce chemin est un attribut de type Stack<Room> dans MovingCharacter.

Exercice 7.49.1

J’ai trouvé qu’un héritage d’Item par Character était pertinent car celui-ci me permet de traiter mes characters avec des ItemLists ce qui en facilite l'implémentassions.

Exerice 7.49.2

Ajout d'éléments de scénario dans le jeu.

Exercice 7.49.3

Génération des javadocs en tenant compte des packages.

Exercice 7.50

La signature de la méthode de la classe Math du package java.lang qui permet de trouver la valeur maximale entre deux entiers est : *static int max(int a, int b).*

Exercice 7.51

Les méthodes de la classe Math sont static car cela permet de facilement les utiliser dans notre code. de plus le package java.lang est automatiquement importé. On ne peut les instancier.

Exercice 7.53

Création de la classe spécifique Main afin qu'elle appelle une méthode play() sur l'objet Game pour lancer le jeu.

Exercice 7.54

Exécution du jeu sans Bluej

Exercice 7.54.1

Codage du jeu via l'éditeur Atom et compilation en ligne de commande. J'ai de plus créé un script createJar.sh qui se charge de compiler, générer la javadoc et créer un Jar. (Voir V. Fonctions ajoutées).

Exercice 7.56

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

Oui car la méthode static existe.

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

Non car une méthode static ne peut agir que sur la classe et non sur l'instance.

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

Oui

Exercice 7.57

```
public class MaClasse {
    private static int aNumberOfInstance = 0;

    public Maclasse(){
        aNumberOfInstance++;
    }//constructeur
    public static int numberOf(){
        return aNumberOfInstance;
    }//numberOf
}//MaClasse
```

Exercice 58

Création d'un fichier Jar afin de lancer le jeu par la commande *java -jar.*

Exercice 58.1

Cette commande est fonctionnelle pour lancer le jeu sur linux et windows.

Exercice 58.2

Le jeu est maintenant téléchargeable sur http://perso.esiee.fr/~delevacw/a3p .

Exercice 60.2

Ajout de derniers boutons dans l'IHM réalisée avec Swing.

Exercice 60.3

Finalisation de la javadoc.

Exercice 60.4

Génération des javadocs finales.

Exercice 63

Complétion du scénario, ajout des fichiers de test et des différents scénarios (victoire / défaite).

Exercice 63.1

Compilation du projet sans warning dans le projet ni dans la javadoc.

Exercice 63.2

Le jeu est à présent jouable.

Exercice 63.3

Des dialogue avec les personnages ont été ajoutés.

Exercice 63.4

Les deux javadocs sont générées et disponibles sur le site ainsi que le projet et le jar du jeu.

III. Mode d'emploi

Lancer directement le .jar (une méthode Main a été ajouté afin que ce dernier soit exécutable).

test room → visite toutes les rooms
test ideal → test du parcours idéal
test full → test de toutes les fonctionalitées

give *item* to *character* → commande d'échange avec un PNJ

IV. Déclaration anti-plagiat

Aucun code autre que ceux fournis n'a été utilisé.

V. Fonctions ajoutées

Création d'un script bash nommé "createJar.sh". Ce dernier commence par compiler les sources puis demande si l'on veut générer la javadocs. Enfin, il nous demande si l'on veut créer un fichier Jar et si oui, comment veut on le nommer.