ESIEE PARIS

RAPPORT PROJET ZULL

Jeu : RPandaScape (R.P.S)

Réalisé par

Elie DUBOUX

Encadré par

Mr. Denis BUREAU

Version du rapport	Date de rendu
Finale	02/01/2022

Année universitaire 2021/2022

SOMMAIRE

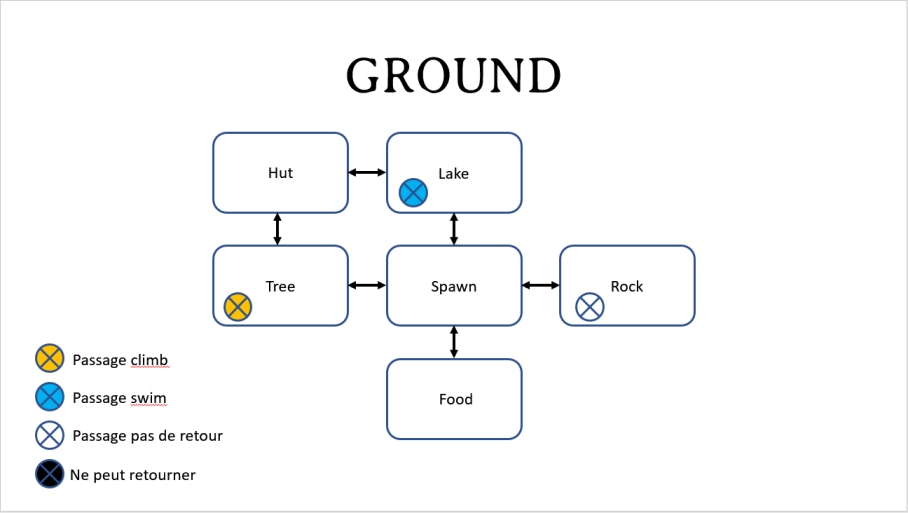
Phrase Thème :

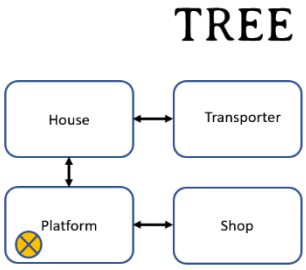
Vous êtes devenu un panda roux et tentez de vous échapper du safari où vous êtes enfermés.

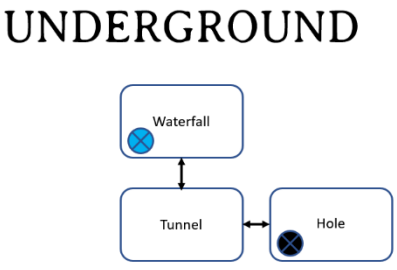
Résumé :

Vous êtes un panda roux tout juste enfermé dans un safari. En vous réveillant, vous ne vous souvenez plus rien et ne savez pas faire grand-chose. Cependant, vous ressentez un étrange pouvoir en vous. Petit à petit, vous allez vous familiariser avec ce corps en acquérant compétences et savoirs dans le but de vous échapper.

Plan :







Scénario Détaillé :

1) Le joueur apparaît au Spawn.

2) Le joueur devra récupérer le livre compétence « climb » se trouvant en dessous de l'arbre. En l'apprenant il sera capable de monter à l'arbre.

3) Le joueur peut devra se procurer les compétences « light » et « shrink » chez le marchand.

4) Il faut ensuite parler au « traveller » se trouvant dans la pièce « Transporter ».

5) Le joueur doit retrouver le traveller afin d'obtenir une mystérieuse plume.

5) En se glissant sous le gros rocher (à l'aide de shrink) le joueur trouve une pièce et y apprend la compétence « swim ».

6) Dans le tunnel noir, le joueur doit utiliser « light » afin d'obtenir la compétence « transform ».

7) Il faut ensuite parler au guide pour que le speech final s'affiche.

6) Le joueur peut désormais utiliser la compétence « transform » si il possède la plume dans son inventaire et s'enfuir du safari.

Détail des lieux :

Spawn : Pièce où apparait le joueur. Le guide présent dans cette salle aide le joueur, le joueur doit aussi lui parler avant de finir le jeu. On y trouve de la mousse.

Lake : Pièce possédant un lac et une cascade. Une pièce se trouve sous le lac, accessible avec la compétence « swim ». On y trouve des algues et des fleurs.

Hut : Une vielle cabane. On y trouve des racines.

Tree : Un grand arbre qui permet de lier le sol avec le haut de l'arbre. S'y trouve le livre de compétence « climb » et des racines.

Rock : Il y aura une pièce sous le rocher avec le livre de compétence « swim », accessible par la compétence « shrink ». S'y trouve le cookie.

Food : Il y a de la nourriture. S'y trouve des racines et du bamboo et des roots.

Platform : Une plateforme a plusieurs mètres du sol, permet de lier le haut de l'arbre au sol.

Shop : Le magasin des pandas roux. Le joueur y pourra s'y procurer des objets comme des pommes, et livres de compétences « shrink » et « light ». Le marchand s'y trouve.

House : La maison des pandas roux. S'y trouve le beamer.

Transporter : Une pièce où en partir téléporte le joueur dans une autre salle aléatoirement. S'y trouve le pnj.

Hole : Un trou sous le rocher, il est impossible de remonter. On y trouve le livre de compétence « swim ».

Tunnel : Un noir complet. (*utilisation de light*) -> Vestiges d'une ancienne civilisation. S'y trouve le livre de compétence « transform ».

Waterfall : Une cascade vraisemblablement reliée au lac. S'y trouve des algues.

Situations gagnantes / perdantes:

Situations gagnantes :

- sortir de la cage en suivant l'histoire.

Situations perdantes :

- mourir de faim.

Enigmes :

- réponses dans des dialogues de personnages.

Commentaires :

- ajouter les énigmes pour débloquer des éléments du jeu.
- système de clans (crépuscule et aurore) avec suite de quêtes.
- alchimie.
- salle d'arcade.
- système d'argent + achat d'objet au magasin.
- images sur l'exécutable.

Exercices :

Exercice 7.5 :

Cette procédure permet d'éviter la duplication de code présente dans les méthodes goRoom() et printWelcome(). Ceci améliorant par la même occasion la lisibilité du code. Aussi, il est plus simple de modifier la façon dont est affiché la description si l'on le désire.

Exercice 7.6 :

Vérifier qu'une Room donnée en paramètre est égale à null n'entraine qu'une différence mineure. Cette dernière étant que si l'on désire débloquer plus tard dans la partie une pièce du jeu qui n'était pas accessible. Nous devrons répéter les pièces déjà accessibles en plus de la nouvelle alors que le livre peut simplement noter les autres sorties comme null.

Exemple :

Une Room A donnait sur les Rooms B au sud et C à l'est. On désire désormais que l'ouest de A donne sur une Room D.

Nous devrons écrire : A.setExits(null, B, C, D) ;

Le livre : A.setExits(null, null, null, D) ;

Mais étant donné que plus tard dans les exercices, le fonctionnement de setExits() sera modifié. Il n'y aura plus aucune différence entre les 2 codes. Il n'est donc pas nécessaire de faire comme le livre.

Pour résoudre le problème de « unknown direction », la 3ème solution donnée dans les commentaires est utilisée:

Au début de Room :

```
public static final Room UNKNOWN_DIRECTION = new Room( "nowhere" );
```

A la fin de getExit() :

```
return (UNKNOWN_DIRECTION);
```

Dans goRoom() de Game :

```
if (vNextRoom == Room.UNKNOWN_DIRECTION)
{
    System.out.println("Unknown direction !");
    return;
}
```

Le problème avec cette méthode est qu'il n'y a pour l'instant pas de différences entre « unknown direction » et « there is no door ». Pour régler ce problème il faudrait vérifier, dans la méthode getExit(), si la direction rentrée par le jouer existe dans le jeu. Cela sera implémenté plus tard.

Exercice 7.7 :

Étant donné que nous utilisons les données contenues dans Room, il est plus logique qu'une méthode de Room traite l'information. De même, jusqu'à maintenant, tous nos affichages se sont faits depuis la classe Game et il n'y a pas de raison que cela change. Ainsi, l'affichage des sorties se fait après avoir récupérer les informations

Exercice 7.8 :

La méthode setExits() est désormais inutile puisqu'elle ne permettait qu'un nombre restreint de sorties. La différence de nommage avec setExit() vient du fait que setExit**S**() définis les 4 sorties avec les paramètres données alors que setExit() ne définit qu'une sortie dans une direction donnée.

POUR TESTER VOTRE COMPRÉHENSION

Pour chacune des 5 opérations sur les tableaux associatifs, pouvez vous associer la méthode de la classe HashMap qui l'implémente ?

ajout :	put
modification :	put
suppression :	remove
recherche :	get
liste :	keyset

Corriger

Exercice 7.8.1 :

Les sorties « up » et « down » ont respectivement été ajoutés aux pièces Tree et Platform. Permettant ainsi au personnage de monté et descendre de l'arbre avec les commandes de déplacement.

Exercice 7.9 :

Le keySet() permet d'obtenir toutes les clefs d'un HashMap. Dans notre cas, on souhaite les directions qui sont utilisées comme clefs de l'attribut exit de la classe Room.

keySet

```
public Set<K> keySet()
```

Returns a `Set` view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own `remove` operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the `Iterator.remove`, `Set.remove`, `removeAll`, `retainAll`, and `clear` operations. It does not support the `add` or `addAll` operations.

Specified by:

`keySet in interface Map<K,V>`

Overrides:

`keySet in class AbstractMap<K,V>`

Returns:

`a set view of the keys contained in this map`

Exercice 7.10 :

Pour résumer les commentaires de la méthode getExitString() : Celle-ci ne prend pas d'argument et doit retourner une String.

```
/**
 * Return a string describing the room's exits, for example
 * "Exits: north west".
 * @return Details of the room's exits.
 */
public String getExitString()
{
    String vExits = "Exits:"; //1
    for (String vOneExit : this.aExits.keySet()) //2
        vExits += " " +vOneExit; //3

    return vExits;//4
}//getExitString()
```

Etudions le code ligne par ligne :

1) On instancie une chaine de caractère vide où l'on concatènera les différentes sorties en les séparant d'un espace.
2) On parcourt les clefs de la HashMap une à une à l'aide du keyset() (décrit à l'exercice précédent). C'est-à-dire que à la fin de la boucle for, la variable vOneExit aura pris à tour de rôle chacune des clefs de la HashMap soit chaque direction.
3) On ajoute une des sorties à celles qui ont déjà été trouvés.
4) On retourne La chaine de caractère.

Exercice 7.10.1 :

Une grande partie des commentaires présent dans le projet sont des reprises de ceux du livre, ces derniers étant très clairs.

Exercice 7.10.2 :

N'apparaît dans la javadoc que les fonctions et attributs public. La classe Game ne possède en public que son constructeur et sa méthode play(). Alors que la classe Room possède beaucoup de méthodes public getExit(), getExitString(), setExit()…

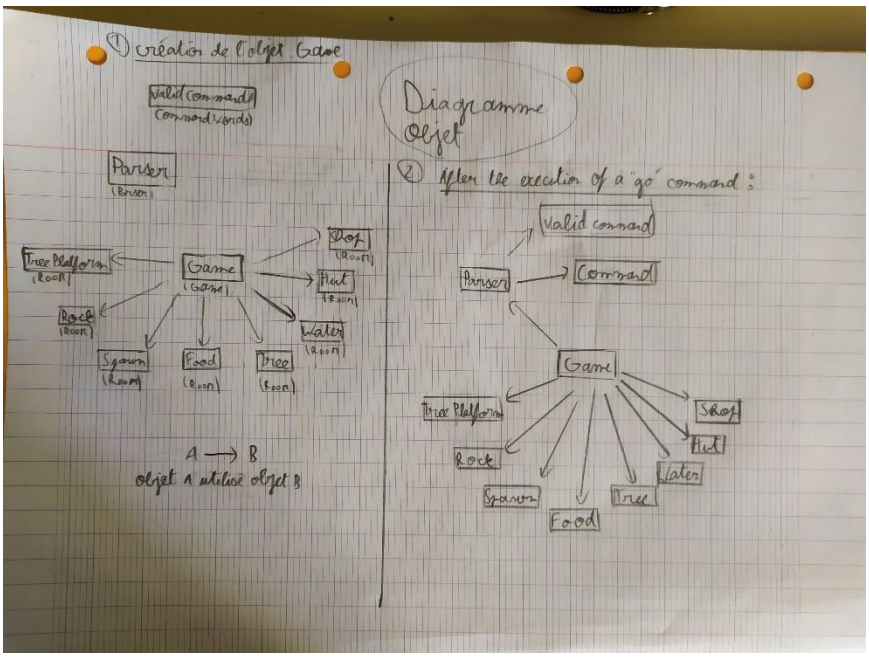
Exercice 7.11 :

Cette fonction getLongDescription est dans la continuité des changements fais jusqu'à maintenant. C'est-à-dire faire en sorte que chaque classe ait son rôle bien défini. Et ici, c'est à la Room de créer un string concernant sa description pas au Game.

Exercice 7.12 :

Voici les diagrammes objets du projet :

1) Après avoir créé un objet de la classe Game.
2) Après avoir exécuté une commande go.



Exercice 7.14 :

Il est plus simple, pour rendre la description d'une room accessible au joueur, de lui faire taper une commande. Ici il s'agit de la commande look qui affiche la valeur retournée par getLongDescription préalablement crée.

Exercice 7.15 :

Cette fonction eat n'a que peu d'utilité pour l'instant, en effet elle ne fait qu'afficher un message sans réel intérêt. Mais il s'agira par la suite de l'une des contraintes principales du jeu la créer maintenant fais donc sens.

Exercice 7.16 :

Ajout de la fonction permettant de ne plus gérer les mots de commande dans la classe Game mais là où ils sont générés, c'est-à-dire dans CommandWords. ShowAll parcours la liste de tous les mots clés déjà crées (à savoir : go, quit, help, look et eat) pour les afficher.

Après exercice 7.16 :

Le for each permet de ne pas à avoir à passer par un index (comme dans un for classique) mais par élément directement.

Par exemple dans la fonction crée à l'exercice 7.16, on a aValidCommand comme ensemble qui est un tableau de String. A chaque itération de la boucle for, on traite l'objet String suivant (c'est-à-dire l'afficher) jusqu'à que tous les éléments du tableau aient étés traités.

Exercice 7.17:

Si on ajoute une nouvelle commande il faudra toujours modifier la classe Game. C'est-à-dire qu'il faudra ajouter une nouvelle méthode qui effectuera l'action désiré. Et cette méthode est appelé dans la méthode processCommand() qui est donc aussi à modifier.

Exercice 7.18.2 :

Le String builder permet d'éviter la création d'un trop grand nombre d'objets de type String. Le changement à été effectué dans la méthode getExitString() de la classe Room et getCommandList() de la classe CommandWords.

Exercice 7.18.3 :

Les images sont toutes de dimension 600x400 pour ce projet. Valeurs choisies arbitrairement.

Exercice 7.18.4 :

Le nom du jeu est désormais RPandaScape pour Red Panda Escape.

Exercice 7.18.6 :

Réponse aux questions :

2.b) Attribut supplémentaire de la Classe Room: chemin vers l'image.

2.c) Il n'y a plus besoin du scanner puisque les données sont collectées grâce au champ texte de UserInterface (aLog).

5)

Avant

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.URL;
import java.awt.image.*;
```

Après

```
//import javax.swing.*;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JTextArea;
import javax.swing.JLabel;
import javax.swing.ImageIcon;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JButton;
//import java.awt.*;
import java.awt.Dimension;
import java.awt.BorderLayout;
//import java.awt.event.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.net.URL;
```

Tous les autres changements nécessaires ont été implémentés.

Exercice 7.18.8 :

Il existe différentes façons pour créer des boutons en java, celle que j'ai choisi ici est la classe JButton. Quant à son placement, il ne restait que EAST et WEST de disponible pour le border layout si aucun d'eux ne me plaisaient j'aurai dû songer à un autre moyen pour placer les éléments dans le panel. Ici, le bouton positionné sur EAST.

Ce bouton devra afficher une commande, pour l'instant il est réglé sur « look » mais cela sera surement changé dans le futur. Le texte à l'intérieur du bouton est généré lors de son instanciation étant donné que la classe JButton peut prendre un Icon, une String ou une Action. Ici l'argument passé est donc la chaine de caractère look.

On ajoute ensuite un ActionListener sur ce bouton à l'aide de la fonction addActionListener() pour qu'une action se produise en cliquant dessus. Il faut ensuite modifier actionPerformed pour pouvoir traiter cette action. Pour différencier le fait de taper « Enter » dans aEntryField et cliquer sur le bouton, il faut vérifier la source de l'action avec getSource. Ainsi, si c'est bien le bouton qui a été cliqué, on utilise la fonction interpretCommand de gameEngine avec la String « look » comme valeur.

Après exercice 7.18.8 :

ActionListener : C'est l'interface permettant de récupérer les ActionEvent (javadoc).

addActionListener() : Permet de placer un « écouteur » sur un objet pouvant déclencher un évènement. Dans notre projet on retrouve le bouton look et l'EntryField.

actionPerformed() : Permet d'effectuer une action (à décrire) si un certain évènement est produit.

ActionEvent : L'évènement en question, dans notre projet on a par exemple : cliquer sur le bouton ou tapper sur entrée après avoir saisie du texte.

getActionCommand() : Renvoie sous forme de texte, la commande associée à l'action.

getSource() : Renvoie l'objet sur lequel l'action a été déclenchée.

Exercice 7.19 :

Le Model View Controller est une façon d'organiser un projet et de veiller à ce que chaque classe suive un rôle bien défini.

Le Model s'occupe de générer une grande partie des données utiles pour l'application.

Le View gère l'affichage et la présentation des information.

Le Controller reçoit les informations du Modèle, les traitent, et les renvoie au View.

L'intérêt de ce design est de facilement pouvoir s'y retrouver dans le code, que ce la soit pour le programmer ou pour une tierce personne voulant ajouter quelque chose au projet.

Aussi en segmentant les rôles ainsi, on peut facilement ajuster et ajouter de nouvelles fonctionnalités sans avoir à forcément modifier tout ce qui a été précédemment codé.

Exercice 7.19.1 :

Dans le projet, on écrit une classe « GameModel » s'occupant de la création des objets du jeu (rooms, exits…) et la génération des Strings. Cette classe gère la partie Model.

Ici le « TextView » du fichier zuul-mvc.jar est remplacé par le « UserInterface » de notre projet c'est cette classe qui affiche un certain nombre de String générés par le « GameModel », il s'agit du View.

Enfin le controller représente le GameEngine qui effectue les actions .

Due à certains problèmes lors de l'implémentation, Observable et Observer ne sont pas utilisées.

Exercice 7.19.2 :

Toutes les images du projet se trouvent dans un dossier « images » à la racine du projet.

Exercice 7.20 :

Un item dans le jeu est caractérisé par son nom, sa description, son poids et son prix.

Des getters/setters sont mis en place pour pouvoir accéder à ces attributs. Et la méthode toString() est Override pour pouvoir afficher le nom, le poids et le prix de l'objet.

Ces Items sont créés et placés dans des Rooms depuis la fonction initialiseRoomsAndItems du GameEngine.

La classe Room accueil désormais un nouvel attribut, aItem (qui évoluera surement en aItems plus tard). Avec certaines méthodes en lien comme getExitItem() qui renvoie la méthode toString de Item.

Exercice 7.21 :

Les items sont créés dans la méthode create() de GameEngine.

C'est logiquement à la classe Item de procurer les informations concernant un item.

La classe affichant les messages est GameEngine, il n'y a pas de raison que cela change.

Exercice 7.21.1 :

La commande look Item regarde si l'item est présent dans la pièce. Si oui, il affiche les informations de la toString de l'Item et sur une 2ème ligne sa description.

Exercice 7.22 :

Parmi les différentes collections possibles, j'ai choisi la HashMap pour représenter les items dans une room avec comme signature HashMap<Item,Integer>.Les items font offices de clés et l'integer associé est le nombre d'occurrences de cet item dans la collection.

Ce qui est différent de la méthode suggérée dans les commentaires de l'exercice avec HashMap<String, Item>. Dans RPandaScape il y aura la possibilité d'agir sur des groupes items ce qui devient bien plus simple avec mon implémentation.

En consequence, les méthodes addItem et Remove Item ont donc comme argument un Item au lieu d'une String (le nom de l'item).

Aussi, tous les items sont affichés dans le getItemsString qui se retrouve dans le getLongDescription elle-même qui se retrouve dans le printLocationInfo (appelé à chaque fois que le joueur entre dans une pièce) .

Exercice 7.26 :

La fonction back() permet de revenir dans la pièce précédente. Ici on utilise une pile pour enregistrer ces pièces. A chaque fois que le joueur se déplace vers une autre pièce, on ajoute l'ancienne à la pile. Pour chaque appel de back(), on retire la dernière ajoutée. Il est donc possible d'enchainer les back sans problème.

Est aussi traité le cas où le joueur utilise back alors que la pile est vide.

Après exercice 7.26 :

Stack : Est une pile c'est-à-dire que le premier élément ajouté sera le dernier retiré (First In Last Out)

Push : Ajoute un élément à la pile.

Pop : Retire et retourne le dernier élément ajouté de la pile.

Empty : vaut true si la pile est vide, false sinon.

Peek : permet d'accéder au dernier élément ajouté de la pile.

Exercice 7.26.1 :

Les commentaires javadoc ont été ajoutés de sorte qu'il n'y ait ni d'erreur ni de warning sur les deux commandes.

Exercice 7.28.1 :

Création du premier fichier de test qui se traduit par l'ajout d'une constante dans CommandWord, dans GameEngine du traitement du mot de commande « test » et la création de la fonction test.

Dans cette fonction on doit ouvrir le fichier et envoyer chaque ligne dans le interpretCommand. Tout cela étant dans un try/catch pour éviter que le programme crash sur une erreur.

Après exercice 7.28.1 :

Lecture simple : En java, il est possible d'ouvrir un fichier, le lire (read) et le refermer.

File : Object java correspondant à un fichier, peut s'instancier avec le chemin vers un fichier.

Scanner : Cette fois ci on ne l'utilise pas pour lire des caractères tapés par un utilisateur mais pour lire dans un fichier.

hasNextLine() : return true si la nextLine n'est pas null.

nextLine() : correspond à la prochaine ligne d'un fichier.

Try/catch/exception : Une erreur dans la série d'instruction du try peut être intercepté par un catch dans le but de ne pas renvoyer cette erreur et ainsi fermer le programme. Il existe de très nombreuses exception comme FileNotFoundException (correspondant à la tentative d'ouverture d'un fichier inexistant), ou tout simplement Exception (qui récupère toutes les erreurs).

Exercice 7.28.2 :

Les 3 fichiers « court.txt », « long.txt » et «gagne.txt » ont été créé dans un répertoire test.

Les fichiers « long » et « gagne » seront modifiés au fur et à mesure de l'avancement du code.

Exercice 7.29 :

Si il existe plusieurs objets Player : inventaire, nom, argent, pièces précédemment parcourus, pièce actuelle, et charge maximale doivent être différentes.

On change donc CurrentRoom et PreviousRooms en conséquence.

Ce qui change le this.aCurrentRoom en this.aPlayer.getCurrentRoom() et la manipulation de PreviousRooms.

Exercice 7.30 :

Ici on ajoute 2 commandes take et drop. Elles doivent, comme toutes commandes, être ajoutées dans CommandWords, le processCommand de GameEngine et être traités dans une nouvelle méthode de Game Engine. Le fonctionnement de ces fonctions changera pour pouvoir take et drop plusieurs items à la fois.

Exercice 7.31 :

Comme pour Room, on ajoute une HashMap<Item,Integer> au Player qui présente les mêmes avantages que pour la gestion des Items de Room.

Aussi, les fonctions take et drop sont modifiés. Elles doivent ajouter ou retirer les items du Player ou de la Room.

Exercice 7.31.1 :

L'ajout de la classe ItemList permet de réduire la multiplication de code présente chez Player et Room. On y retrouvera aussi une fonction toString pour afficher la liste d'item présent. Cette liste sera considérée comme l'inventaire du Player et les objets entreposés pour une Room.

Exercice 7.32 :

On doit ici faire attention à la méthode take(), qui ne peut pas prendre un objet si le joueur n'est pas capable de supporter cette charge supplémentaire. Un attribut aWeightTotal est donc ajouté à la classe ItemList qui s'incrémente à chaque appel de take() et décrémente pour drop.

Il était ennuyeux de devoir répéter les commandes take et drop si l'on désire un nombre important d'items. Pour pallier à cela, les commandes take et drop peuvent prendre un 3ème argument qui est le nombre de fois où l'item en question est take/drop.

Cette fonctionnalité a entrainé divers changements :

- le Parser (prise en compte d'un 3ème Word)
- la classe Command vérifie que ce 3ème argument est bien un nombre entier
- Le comportement de take et drop
- Le comportement de add/remove dans Room,Player et ItemList (avec notamment les vérifications)

Exercice 7.33 :

Ajout de la fonction inventory() qui affiche le retour de getItemsString() du Player qui n'est rien de moins que la String « inventory :» concaténé avec la toString de ItemList.

Exercice 7.34 :

La méthode eat() est modifié pour prendre en argument la ligne rentrée par l'utilisateur. Le joueur pour désormais manger certains Items actuellement Bamboo, Apple, Cookie et Root.

Manger le cookie le retire de l'inventaire et la aWeightMax du Player vaut son double.

Cette méthode sera encore modifiée pour intégrer le système de barre de nourriture plus tard.

Aussi une classe héritant de « Item » appelé « EdibleItem » permet de gérer plus facilement les items mangeables et les autres.

Exercice 7.34.1 :

Il y a 4 fichiers tests :

- court : qui contient juste quelques commandes.
- commandes : qui contient toutes les commandes du jeu.
- rooms : qui fait passer le joueur par toutes les pièces du jeu .
- win : qui fait gagner le joueur.

Exercice 7.35.1 :

Je n'ai pas implémenté d'enum dans mon projet, cependant la méthode eat utilise le switch pour traiter les différentes actions effectuées en fonction de l'item mangé.

Étant donné que le nombre d'item pouvant être manger pourrait augmenter par la suite, il est plus simple de procéder ainsi.

Après exercice 7.35 :

Le switch permet de remplacer une longue suite de if/else if/else.

Le switch prend une variable en argument, et va avoir différents comportements en fonction de sa valeur.

Chaque « case » prend en compte une valeur possible de la variable. Elles doivent tous être de même type.

Le default est le cas où la variable n'est égale à aucune des valeurs des « cases ».s

Le mot clé break, désigne le passage la sortie du switch.

Exercice 7.42 :

Dans ce jeu, le joueur possède le l'énergie qu'il dépense à chaque fois qu'il se déplace (par la commande go ou back).

SI un joueur n'a plus d'énergie il meurt entrainant le game over.

Il peut regagner de l'énergie en mangeant des items comestibles (apple, bamboo, cookie et root).

Exercice 7.42.2 :

De légers changements concernant l'IHM graphique ont été effectués comme la couleur du background et la police de la zone de texte.

Exercice 7.43 :

En passant par une trapdoor, le joueur ne peut pas revenir en arrière. Par exemple dans mon jeu, après être descendu par le trou sous le rocher, le joueur ne peut pas remonter.

Il faut dans un premier temps ne pas ajouter une sortie « up » au rocher (ou faire pointer cette direction vers une autre room que la précédente).

Dans un second temps, on définit une méthode dans room « isBackPossible » qui détermine si la pièce est une trap door ou non. Si c'est le cas la liste de pièces précédemment visités par le joueur est remise à zéro et un message s'affiche.

Pour vider cette liste, la méthode « clear » de la classe Vector (mère de « Stack ») est utilisé.

Exercice 7.44 :

L'ajout du « beamer » entraine la création de 2 nouvelles commandes « fire » et « charge » et d'une nouvelle classe « Beamer » héritant de « Item ».

Les informations concernant la room sauvegardé avec le « charge » est stocké dans le beamer. Donc le beamer peut être chargé, laissé tomber, ramassé puis utilisé par potentiellement différents joueurs.

Exercice 7.45 :

Dans ce jeu, une porte ne peut être passé que si le joueur possède la compétence adéquate.

Par exemple pour grimper ou descendre de l'arbre le joueur a besoin de la compétence climb.

Une porte est représentée avec une classe Door avec un skill qui lui est associé.

Il existe 3 portes dans le jeu lié à 3 skills différents : climb, shrink et dive.

Le skill étant lié au joueur, ce concept est donc approprié pour un jeu en multi-joueurs.

Exercice 7.46 :

La pièce transporter est une pièce particulière, on créer donc une classe « TransporterRoom » héritant de « Room » qui contient toutes les pièces du jeu.

Il s'agit d'une trap door où en sortir renverrait le joueur dans une pièce aléatoire du jeu. Le jeu a d'ailleurs été conçu pour ne pas pouvoir bloquer le joueur qu'importe où il se téléporte.

Ce nombre aléatoire est généré par une classe à part « RoomRandomizer » qui après lui avoir fourni le nombre de pièces dans le jeu, renvoie un nombre aléatoire entre 0 et ce nombre -1. Il s'agit de l'un des indices dans le tableau des pièces. La pièce portant cet indice sera la pièce vers laquelle le joueur sera téléporté.

Après exercice 7.46 :

Random est une classe permettant de générer des nombre aléatoire (entiers ou réels).

La méthode nextInt renvoie un nombre entre 0 et sa valeur max (qui peut être ou non définit).

```
public int nextInt(int bound) {
  if (bound <= 0)
    throw new IllegalArgumentException("bound must be positive");

  if ((bound & -bound) == bound)  // i.e., bound is a power of 2
    return (int)((bound * (long)next(31)) >> 31);

  int bits, val;
  do {
      bits = next(31);
      val = bits % bound;
  } while (bits - val + (bound-1) < 0);
  return val;
}
```

Ici, on souhaite que le nombre généré soit compris entre 0 et le nombre de pièces -1 .

Une seed est le nombre à partir duquel l'algorithme de génération de nombre aléatoire va tourner. Si l'on fourni 2 seeds identiques on aura donc la même séquence de nombre aléatoires. Pour éviter ceci, on utilise currentTimeMilis() qui renvoie le nombre de millisecondes écoulés depuis le 1er janvier 1970. Donc dans que les deux requêtes ne sont pas générés à moins d'une milliseconde d'écart, les nombres générés seront différents.

Exercice 7.46.1 :

La commande alea change le comportement de Transporter room et ajoute un attribut au gameEngine pour vérifier si la commande est bien exécutée en mode test. Ainsi l'utilisateur ne peut utiliser cette fonction sans passer par un fichier de test.

La commande test utilise la fonction alea (sans rien derrière) à la fin pour s'assurer que la sortie de transporter room soit de nouveau aléatoire.

SI la commande alea fonctionne, elle enregistre cette room dans transporter room.

Et tant que cette pièce ne vaut pas null n'importe quelle sortie de la pièce renvoie vers cette room. La commande alea (sans rien derrière) permet de set cette room à null.

Exercice 7.46.2 :

Les classe Beamer hérite de Item étant donné qu'un Beamer est une sorte de Item.

Il hérite donc des méthodes et attributs (en public ou protected) définis dans Item + celles propre à sa classe.

Exercice 7.46.3 :

Un commentaire javadoc a été généré pour chaque classe, méthode et attribut afin d'éviter des erreurs et warnings lors de la génération des 2 docs.

Exercice 7.47 :

On crée une classe fille de « Command » pour chaque commande. Cela rendant le code plus modulable, cela demande cependant quelques modifications notamment des attributs anciennement accessibles depuis le GameEngine qui ne le sont plus désormais.

Le jeu demande des skills pour passer les portes, l'implémentation de la classe SkillBook héritant de Item permet donc une meilleure structuration du code. Chaque skill book possède une compétence qui peut être apprise avec la commande learn.

Aussi, classe « DarkRoom » représentant une pièce où aucune information (à l'exception des sorties) n'est accessible. Pour pouvoir savoir ce que contient la pièce, il faut utiliser le skill « light » qui est une nouvelle commande.

Après exercice 7.47 :

Le polymorphisme à partir de l'héritage est très pratique pour que 2 classes sœurs (héritant d'une même mère) puisse avoir une même méthode mais avec des comportements différents. En java une classe fille peut override une méthode (en ajoutant @Override) avant la définition de la méthode pour ne changer le comportement (il faut aussi que la définition soit identique).

Par exemple dans ce projet la classe Command donne à toutes ses classes filles la méthode execute qui doit être redéfini pour chaque commande. L'avantage est que en appelant execute de Command, on puisse exécuter l'action correspondant à la bonne méthode.

Exercice 7.47.1 :

Un découpage logique pour les classes et de faire par héritage et utilisation dans le code.

Le premier package permet de séparer toutes les classes (à l'exception du main) du reste des fichiers du projet (tests, javadoc…).

Ensuite un package est créé pour chaque groupe de classe fille (classes qui héritent de la même classe mère). Ainsi les packages pkg_room, pkg_command et pkg_item comprennent toutes les classes qui héritent de Room, Command et Item. Il serait aussi étrange de laisser les classes dérivées par exemple RoomRandomizer dans un package différent de TransporterRoom.

Après exercice 7.47.1 :

Lorsqu'une classe appartient à un package, il faut écrire au début du fichier « package nom_package », ainsi, si l'on veut utiliser cette classe dans une autre il faut « import chemin_packages…NomClasse ». Ce chemin de package se fait depuis la racine du projet.

Le package par défaut correspond à un package que l'on n'a pas besoin d'import, c'est le cas du package java.lang.

Deux classes d'un même package n'ont pas besoin de s'import entre eux, par exemple entre UserInterface et GameEngine il n'y a pas besoin d'import étant donné qu'elles sont dans le même package « pkg_class ».

Exercice 7.48 :

Un pnj possède un nom et une liste de dialogue, et chaque pièce du jeu contient une liste de pnj.

Quand le joueur utilise la commande « talk », le premier des speechs du pnj en question apparait dans la zone de texte. Si il y a plus d'un pnj dans la pièce, il est nécessaire d'écrire « talk 'nomPnj' ». Aussi, la commande look a été modifié pour permettre de voir quels sont les speechs d'un pnj en tapant « look 'nomPnj' ».

La commande talk peut prendre un 3ème argument qui correspond à un certain dialogue du pnj. Pour cela le joueur doit écrire « talk 'nomPnj' 'X' » avec 'X' le numéro du speech.

Exercice 7.49 :

Pour finir le jeu, le joueur doit notamment parler au traveller qui va se téléporter dans une pièce aléatoire du jeu et c'est en le retrouvant que le joueur obtient un certain objet. Il est au départ du jeu présent dans le transporter.

Une classe TravellerPnj a été créé pour représenter ce traveller, elle possède en attribut la pièce dans laquelle le pnj est présent. La commande talk vérifie si le pnj est bien le traveller, si c'est le cas, il change sa currentRoom si le pnj ne ce n'est pas encore déplacé.

Exercice 7.49.2 :

En plus du pnj traveller 2 autres ont été ajoutés.

Le guide est sensé aider le joueur et lui donner des indices si il est bloqué (non implémenté). Il est aussi nécessaire pour le joueur d'aller lui parler afin de finir le jeu.

Le marchand n'a aucune fonctionnalité mais sera utile lors de l'implémentation du système d'achat (non implémenté).

Exercice 7.53 :

La classe Game contient une méthode main en static qui peut être appelé sans avoir à créer d'instance de Game.

Exercice 7.58 :

Le .jar est un exécutable (lancement du main) cependant il y a un problème au niveau de l'affichage des images et mes recherches n'ont pas réussis à le résoudre… Ce qui n'est pas dérangeant étant donné qu'il est toujours possible d'ouvrir le jar et de le lancer dans n'importe quel IDE.

Mode d'installation :

Pour l'instant :

- télécharger le fichier .jar disponible sur le site web. (encore non disponible)
- l'ouvrir avec un IDE (par exemple blueJay).
- Instancier un objet de la classe Game.

Sources et anti-plagiat:

- Extrait du chapitre 9 du livre rédigé par Michael Kolling and David J. Barnes.
- Classes CommandWords, Parser, GameEngine, UserInterface du projet, de Michael Kolling, David J. Barnes et Denis Bureau.
- La java doc https://docs.oracle.com/javase/8/docs/api/index.html (par exemple pour la fonction HashMap).
- Les sites suivants ont été utilisé notamment pour résoudre des bugs.
  https://stackoverflow.com/
  https://waytolearnx.com/
  https://www.delftstack.com/
  https://www.developpez.net/
  https://www.geeksforgeeks.org/
- Parties de codes entièrement repris :
  https://www.developpez.net/forums/d864596/java/interfaces-graphiques-java/awt-swing/swing-ajouter-background-type-image/ - 6ème commentaire
- Images du jeu tirés d'internet, la source de ces images a été perdu.

Ne figure en ce projet nulle information ne provenant pas des sources citées ci-dessus ou de ma réflexion personnelle.