Death’s Realm

2018-2019 G11c

I-

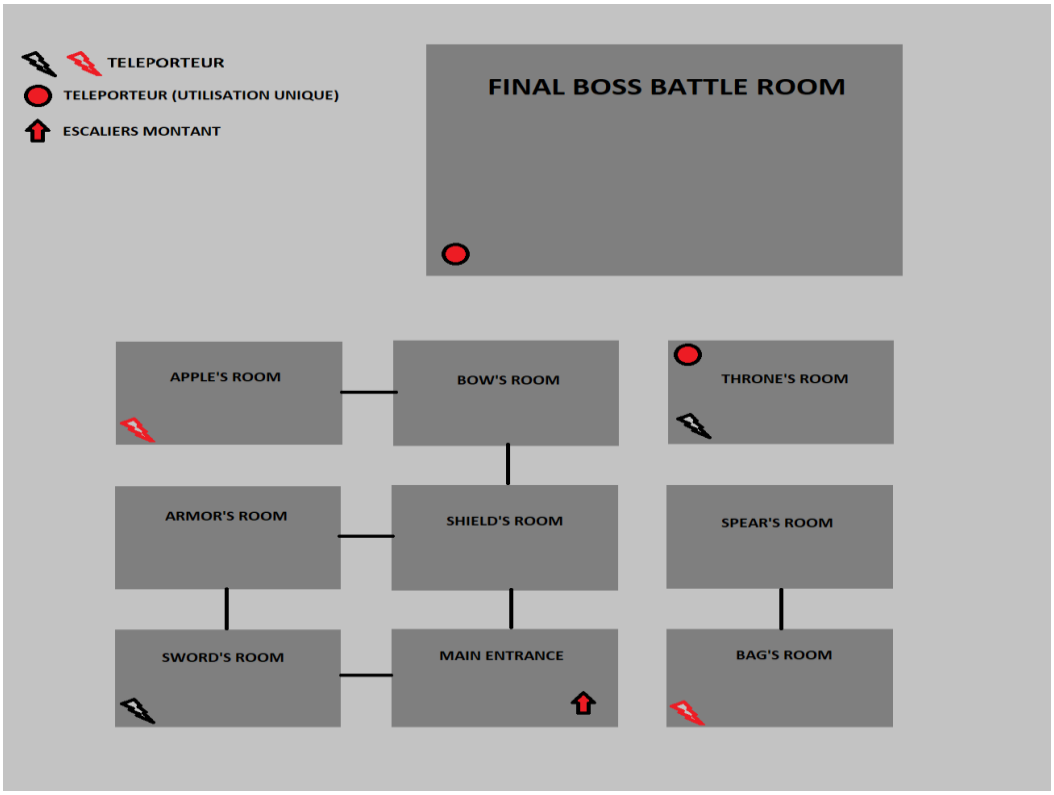
Auteur : Théo Gueuret

Thème : Dans un temple d’Hadès un spartiate tente de récupérer des armes surpuissantes.

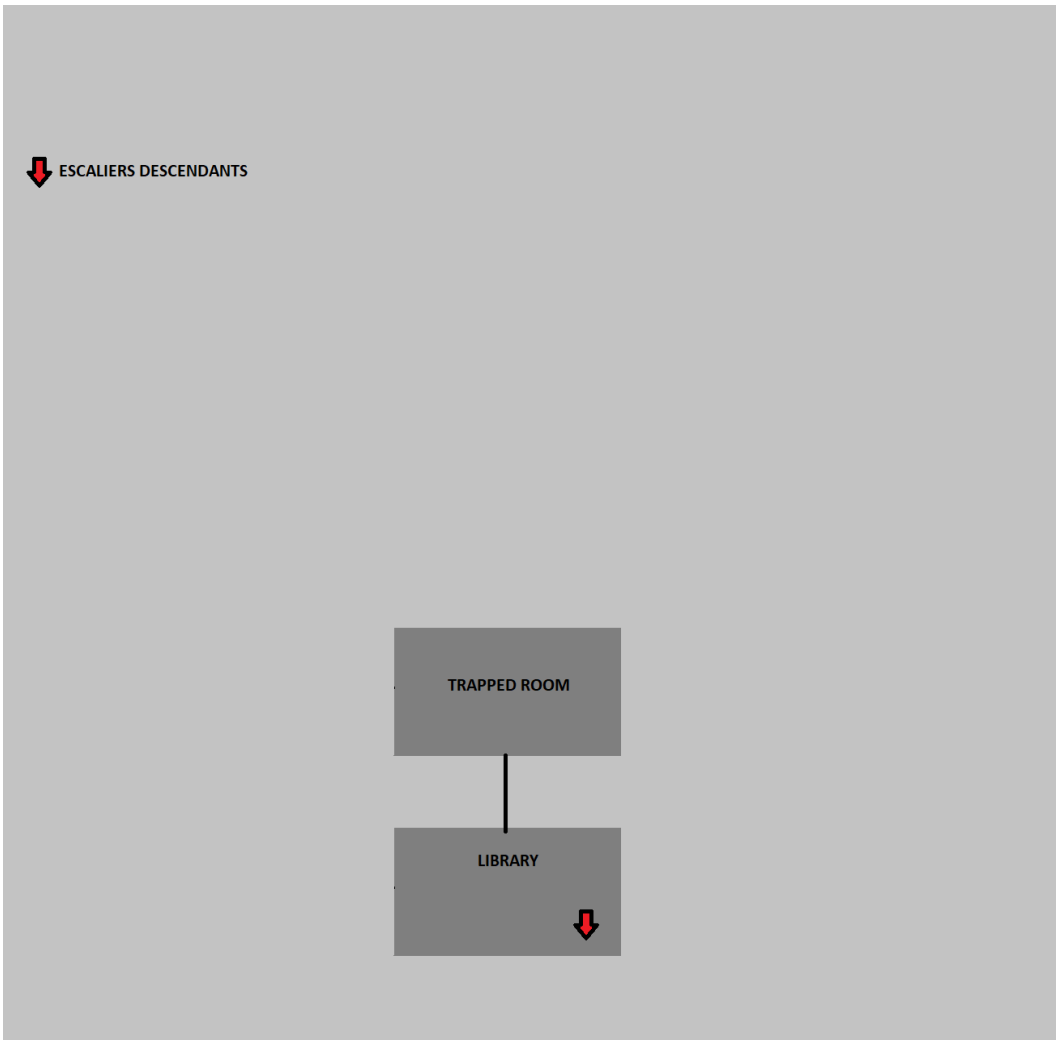
Résumé du scénario : Dans la Grèce antique, un spartiate, Léonidas, 25 ans se rend dans un temple d’Hadès pour récupérer des armes ancestrales surpuissantes afin de tuer un dangereux fils d’Arès.

Plan :

1er étage :



2eme étage :



<u>Scénario détaillé</u> :

Nous incarnons Léonidas, un hoplite (ou spartiate) Grec durant l'ère de la Grèce antique.

Un demi-dieu, Térée, fils d'Arès provoque des ravages un peu partout en Grèce, afin de l'arrêter vous vous mettez en quête d'armes capables d'abattre ce monstre de puissance.

Votre quête vous mène dans un temple d'Hadès qui se révèle être un immense labyrinthe rempli d'artéfacts puissant.

Votre but est d'explorer les différentes pièces du temple, de choisir les objets que vous voulez et de rencontrer Hadès dans la salle du trône en personne afin de lui demander de l'aide pour combattre Térée.

<u>Lieux, objets et personnages</u> :

Le jeu comporte 11 **pièces** différents, celles-ci sont :

Main entrance : endroit où le joueur commence son aventure.

Sword's Room : endroit où le joueur peut trouver une épée et un téléporteur vers '**Final Room**'.

Armor's Room : endroit où le joueur peut trouver une armure

Apple's Room : endroit où le joueur peut trouver une pomme et un téléporteur vers « **Bag's Room** ».

Shield's Room : endroit où le joueur peut trouver un bouclier.

Bow's Room : endroit où le joueur peut trouver un arc magique.

Bag's Room : endroit où le joueur peut trouver un sac et un téléporteur vers « **Apple 's Room** ».

Spear's Room : endroit où le joueur peut trouver une lance.

Throne's Room : endroit où le joueur peut trouver un téléporteur vers « **Sword's Room** », et Hadès qui peut le téléporter vers la salle finale du jeu : '**Final Boss Battle Room**'.

Final Boss Battle Room : Salle finale du jeu où le joueur doit affronter Térée, aucune possibilité de fuite.

Library : Cette pièce est située au-dessus de Main Entrance, le joueur peut y trouver un cookie magique et le beamer.

Trapped Room : Une salle piégée située au nord de **Library** si vous tentez de sortir de cette Room vous serez téléporté dans une Room aléatoire. Exception faites pour les salles : **Trapped Room** elle-même et **Final Boss Batlle Room**.

Le jeu comporte 10 **objets** différents, dont :

- 3 qui font des **dégâts** :

    Une épée, une lance, un arc.

- 2 qui servent à **bloquer** les dégâts :

    Un bouclier et une armure.

- 5 objets **utilitaires** :

    Une pomme, un livre (inutile), un cookie, un beamer et un sac.

Il faut cependant savoir que certains objets peuvent avoir une fonction double (l'épée par exemple confère quelques points de défense).

Le jeu comporte 3 **personnages,** dont :

Le personnage principal, **Léonidas.** Le roi des enfers, **Hadès**. Et enfin le fils d'Arès, **Térée**.

Nous avons **2 situations possibles à la fin du jeu** : une situation **gagnante** et une situation **perdante** :

-Si vous arrivez à battre Térée, le boss final, vous êtes dans la situation gagnante.

-Cependant si vous mourrez pendant le combat contre Térée, vous êtes dans la situation perdante et vous devez recommencer le jeu.

II-

<u>Réponses aux exercices</u> :

7.5 : J'ai modifié les méthodes printWelcome() et goRoom() afin de supprimer toute duplication de code. Pour ce faire nous avons donc crée une méthode printLocationInfo() qui est appelé dans les deux fonctions printWelcome() et goRoom(), j'ai ainsi supprimé des lignes de code dupliquées. La situation était inacceptable car nous avions de fois les mêmes lignes de code à deux endroits différents.

7.6 : J'ai simplifié le code de façon à réduire le couplage dans le code, le but étant de faire le moins de modifications possibles lorsque l'on veut ajouter quelque chose (une direction par exemple).

7.7 : Il est en effet logique de demander à Room de produire les informations sur les sorties de ses propres objets, en effet, en tant que classe Room, tout ce qui incombe des informations sur les pièces revient à Room.

7.8./7.8.1 : Je comprends l'intérêt des Hashmap, on associe des keys (dans notre cas des String) à des values (dans notre cas des Room) afin d'avoir une corrélation entre nos directions et les pièces adaptées. Nous avons ajouté une direction 'up' avec une pièce « HiddenOne ».

7.9 : La méthode keySet() permet de retourner un tableau de même type que les 'keys' (donc dans notre cas un tableau de String)

7.10 : Dans la méthode getExitString nous rentrons à l'aide de Set un tableau de String contenant les String des directions des sorties de la pièce voulue (on appelle la méthode par : maPiece.getExitString()). Ensuite nous utilisons un 'for each', c'est-à-dire que pour chaque 'exit' de notre tableau 'keys', nous ajoutons cette String à la grande String que nous allons renvoyer. Par exemple si notre tableau 'keys' est composé de : {« north », « south »}, la grande String finale va retourner « Exits : north south ».

7.11 : J'ai ajouté dans la classe Room une méthode permettant de renvoyer la String adéquate lorsque l'on veut les différentes informations quant à notre pièce actuelle, donc sa description et ses sorties. Cette fonction à été ajoutée dans le but de réduire le couplage entre les classes Room et Game.

7.14 / 7.15 : Ces exercices nous permettent de comprendre le couplage implicite. En effet il advient assez souvent que le lien soit totalement implicite. Comme pour le cas des commandes dans notre jeu. En effet il y a un couplage entre la classe CommandWords et la classe Game. En effet si nous ajoutons une nouvelle commande sans pour autant avoir gérer le cas dans Game, nous aurions un problème d'exécution : la commande sera valide mais la rentrer n'effectuera aucune action. Nous devons donc bien identifier ses couplages implicites et ne pas les oublier. Lors de ces exercices, j'ai ajouté 2 nouvelles commandes : look() et eat() . J'ai bien sur crée de nouvelles méthodes dans la classe Game afin de pouvoir les utiliser.

7.16 : J'ai cette fois effectué une correction, en effet la liste de commandes n'était pas complète. Le programme affiche désormais une String contenant toutes les commandes valides actuelles. Cette méthode est appelée depuis la classe Game par la classe Parser.

7.18 : J'ai modifié la méthode showAll() en getCommandList(), au lieu d'afficher la liste depuis CommandWords nous l'affichons désormais dans la classe Game. getCommandList() se contente de retourner la String contenant les commandes, et Game de l'afficher.

7.18.4 : Le titre du jeu est : Death's Realm.

7.18.5 : Création d'une HashMap afin de pouvoir accéder à toutes les pièces. La HashMap a été faite en tant que qu'attribut de la classe Room.

7.18.6 : Mise en place de l'interface graphique en vue d'avoir des images pour chaque pièce du jeu. De très nombreuses modification ont été effectuées lors de cet exercice.

7.18.7 : addListenerAction() : Permet d'ajouter une zone de texte au lecteur d'évènements

actionPerformed() : Permet de faire la transition entre l'évènement (action) et la méthode liée à cette action.

7.18.8 : Deux boutons ont été ajoutés, un pour la commande eat et un autre pour la commande look, j'ai dû mettre des détecteurs d'événements sur ces boutons et associer le clic du bouton au lancement de la fonction associée. La fonction createGui() a donc été modifiée.

7.19.2 : Un répertoire Images à ainsi été crée et toutes les images du jeu y ont été déposées.

7.20 : Cet exercice marque la création de la classe Item. Chaque instance de cette classe est caractérisée par 3 attributs : leurs noms, leurs poids et leurs descriptions. Des accesseurs ont ainsi été crées ainsi que des modificateurs.

Avec cette fonction les différents Item du jeu ont été crées dans la fonction createRooms() de la classe GameEngine. De plus la classe Room accepte maintenant le nouvel attribut aItem et les Items sont placés aux bonnes places dans la focntion createRooms() de GameEngine.

7.21 : La classe Item est aussi pourvue d'une fonction getItemDescr() qui a pour but de retourner une String décrivant complètement l'item souhaité de la forme : « this item, 'nom de l'item ', is 'description de l'item' » suivi de (si son poids n'est pas nul) «, its weight is 'poids de l'item' kg. » .

7.22 : Mise en place d'une HashMap afin de pouvoir avoir plusieurs items dans une seule salle. La classe Room est donc adaptée de sorte à ce qu'on puisse ajouter, enlever des items de la salle. Que l'on puisse obtenir tous les items de la salle ou un seul en fonction de son nom. De même la fonction printLocationInfos() à été modifiée afin d'afficher les items présent dans une salle à chaque qu'on rentre dans celle-ci. Par le même raisonnement une fonction permettant de retourner une String décrivant tous les items de la salle.

7.22.2 : La pièce Main Entrance (la pièce de base du jeu) contient désormais l'item 'book', cet objet est inutile et sert juste à répondre aux critères de l'exercice. De plus la pièce Library contient désormais 2 items différents.

7.23 : La commande 'back' est implémentée, elle permet de revenir à la pièce ou l'on était précédemment. Cette commande est faite de telle sorte que lorsqu'on effectue plusieurs fois la commande back à la suite nous revenons de plus en plus (voir exercice 7.26), répétant ainsi nos actions précédentes jusqu'à revenir au début du jeu (où la commande back) n'aura plus aucun effet. La fonction back enlève donc l'instance la plus haute de la pile d'appelle Stack et repositionne le joueur dans la room qui est de nouveau la plus haute dans la pile d'appel. Une fois le changement effectué, l'image de la salle est mise à jour de même que la description de la salle et des items présents dedans. Bien sur la nouvelle commande 'back' à été implanté dans la classe CommandWords, et dans la fonction interpretCommand de GameEngine.

7.26 : La pile d'appel Stack est mise en place dans cet exercice. Dans notre jeu le but du Stack est d'enregistrer tous les changements de salles du joueur, de telle sorte à ce que nous puisons revenir en arrière à tout moment (commande back). L'importation de la classe Stack (import java.util.Stack) est nécessaire afin d'utiliser la Stack. La Stack ajouté est de type Room, c'est-à-dire qu'elle contient des objets de type Room et seulement ceux-là.

7.28.1 / 7.28.2 : Création de la commande test avec intrégration classique via CommandWords et dans interpretCommand. Cette commande qui fonctionne avec un second mot (nom du fichier sans le .txt) permet d'effectuer une batterie de test très rapidement. 3 fichier ont ainsi été concus : idéal qui permet d'arriver le plus rapidement final au boss final (sans forcer pouvoir gagner contre lui), court qui permet de tester 3 commandes basique rapidement et enfin le fichier 'long' qui effectue un passage dans toute les salles et qui effectue quelques commandes à chaque fois.

7.29 : Implémentation de la classe Player, la classe est une classe très importante ayant amené de nombreux changements dans notre jeu. Tout d'abord la classe Player est composée de nombreux attributs tels que aCurrentRoom qui à été transféré depuis GameEngine vers Player pour des raisons de cohésion des classes. Les autres attributs de Player sont aCurrentWeight, aMaxWeight, aName, aCarriedItems et aPreviousRoom qui a aussi été transféré depuis gameEngine. Player contient donc de nombreux accesseurs et modificateurs. Avec ces transferts d'attributs Player devient en quelques sorte le centre de l'information du programme, la quasi-totalité de l'information est traitée ou au moins regroupée ici ce qui engendre de nombreuses nouvelles fonctions tel que getItem, getItemFromPlayer/Room, getRoomImage , getRoomDescription. Cette implantation à permis de réduire considérablement le nombre de lignes dans GameEngine. En effet en plus des fonctions transférées, de nombreuses fonctions étant restées dans GameEngine on été simplifiée par le travail de Player avec de l'information collectée directement depuis player.

7.30 : Les fonctions drop et take sont en quelques sortes les suites logiques des précédents exercices (Player, Items), en effet drop et take permettent une interaction entre les pièces, les objets et le joueur. Le joueur peut désormais prendre des items de salle sur lui ou encore déposer un item qu'il porte dans la salle. Les 2 fonctions associées aux nouvelles fonctions de la classe GameEngine ont été ajouté et associé pour comme fait précédemment. Afin de répondre aux besoins de cohésions 2 nouvelles fonctions ont aussi été ajoutées dans la classe Player afin de d'effectuer les changements sur les attributs concernés.

7.31 : Cet exercice permet de porter plusieurs items sur soi, ceci se traduit par une modification de l'attribut Item aCarriedItem en HashMap<String, Item> aCarriedItems. Le joueur peut maintenant porter plusieurs items sur lui. Ce qui permet encore plus d'interaction entre joueur, pièce et objet.

7.31.1 : Création de la classe ItemList les objets de cette classe dont le seul attribut est une HashMap remplaceront les HashMap qui sont attributs des classes Player et Room (donc aCarriedItems et aItems). Nous ne dupliquons ainsi plus de code et l'utilisation des fonctions pour ces HashMap est également simplifiée.

7.32 : Mise en place de l'attribut poids max (aMaxWeight) dans la classe Player, elle permet d'appliquer une certaine limite aux items que le joueur peut porter, dans le sens où il ne peut tous les prendre dû à sa nouvelle limite de poids. Le joueur devra donc choisir entre plusieurs choix d'items pour son aventure. Concrètement cet ajout apporte un nouvel attribut dans Player, un accesseur et un modificateur. Ainsi que d'autres nouvelles conditions concernant le fait de prendre un item (fonction pickItem()).

7.33 : Ajout de la commande inventory, elle permet d'afficher à l'écran des informations concernant les objets portés par le joueur ainsi que leur poids total. Cette commande requiert surtout des informations avec un traitement si le joueur n'a aucun item. Une fonction inventory a donc été créée dans la classe GameEngine.

7.34 : Ajout de l'item magic cookie, qui permet une fois manger de gagner 5 kg sur le poids maximum porté par le joueur. Deux nouveaux attributs ont dû être crées aEffect et aIsEatable dans la classe Item, les items peuvent donc maintenant être mangés, ou non et ont des effets ou non. La fonction eat à été remodeler dans GameEngine de telle sorte à ce que les principales données et actions s'effectuent dans la fonction eatItem, qui vérifie si cela est possible et qui applique les effets souhaités.

7.34.1 : Les fichiers de tests ont été modifiés, le chemin le plus rapide pour arriver à la fin du jeu est toujours le même mais cependant le fichier « long.txt » à été grandement modifié. En effet dans ce test le joueur va dans chaque pièce, utilise look, prend l'item ou les items présents dans la salle, affiche l'inventaire et repose l'item avant de se diriger dans la dernière salle.

7.42 : Ajout d'un nouvel attribut de type integer aTimeLeft dans la classe GameEngine, celui-ci est mit à 30 dans le constructeur de la classe. A chaque action ou commande entrée par le joueur (à condition que celle-ci soit valide) le compteur diminue de 1. Lorsque le joueur atteint 15, un avertissement est envoyé et lorsque celui-ci atteint 0, le joueur meurt et la partie est terminée.

7.42.2 : Mise à jour de l'interface graphique, maintenant nous avons 11 boutons différents : pour aller au Nord, Sud, Est, Ouest, en haut (Up), en bas (Down) et passage secret (secret). Mais aussi pour utiliser les commandes look, help, inventory et back. Toutes les possibilités ont été prévues (désactivation lors de la fin du jeu ect).

7.43 : Mise en place des portes piégées. Pour ce faire la fonction isExit à été crée dans la classe Room, elle permet de vérifier en retournant un boolean si la Room passée en paramètre est une des sorties de la Room actuelle. Avec cette fonction il suffit de rajouter une condition dans la fonction back() de GameEngine(). Ainsi si la Room voulue n'est pas dans les sorties de la salle actuelle le back est bloqué, le joueur ne peut plus retourner en arrière.

7.44 : Création de la classe Beamer, héritée de la classe Item. La classe Beamer est composée de 2 attributs : aIsCharged et aMemorizedRoom. En soit l'attribut aIsCharged peut ne pas être utilisé mais pour des raisons de simplicité et de visibilité j'ai préféré le créer et l'utiliser. La classe Beamer à aussi deux autres fonctions : charge() et fire(). charge() sert a mémoriser la Room actuelle et fire() sert à se téléporter. Deux commandes supplémentaires ont donc été ajoutées : fire et charge ! Toutes les exceptions ont été prises en compte.

7.46 : Modification de la fonction getExit() de Room, celle-ci renvoi maintenant une Room aléatoire si la salle actuelle est la TrappedRoom. La classe Random à été utilisée ainsi que la fonction nextInt() de Random. La TrappedRoom peut nous envoyer n'importe ou sauf dans la salle finale ou dans TrappedRoom elle-même.

III-

Instructions de démarrage :

-Compiler la classe Game.

-Clic droit sur la classe Game et sélectionner « new Game() ».

IV-

Déclaration anti-plagiat :

Moi, Théo Gueuret certifie que je n'ai pas recopié la moindre ligne de code, en dehors de celles fournies dans les fichiers zuul-*.jar .