Rapport Projet Zhuul

site disponible ici : https://perso.esiee.fr/~adamb/A3P/index.html

Auteur

Baptiste Adam
Groupe 8
Elève en première année à l'ESIEE.

Scenario

Vous êtes un héros sans nom en recherche de gloire et de fortune. De ce fait, vous avez accepté une quête d'un vieux mage. Il vous a chargé de récupérer la statuette sacrée d'Anbrodéma.

Celle-ci a été perdue il y a bien longtemps mais les légendes racontent qu'elle sera enfermée au fin fond du terrible donjon de Xaar-Saroth.

Votre histoire commence alors que vous vous apprêtez à pénétrer dans le dit donjon…

Détails du jeu

Item : épée rouillée, armure rouillée, épée, armure, cookie, steak, champignon, potion, sacoche d'or, coffre d'or.

Salles :

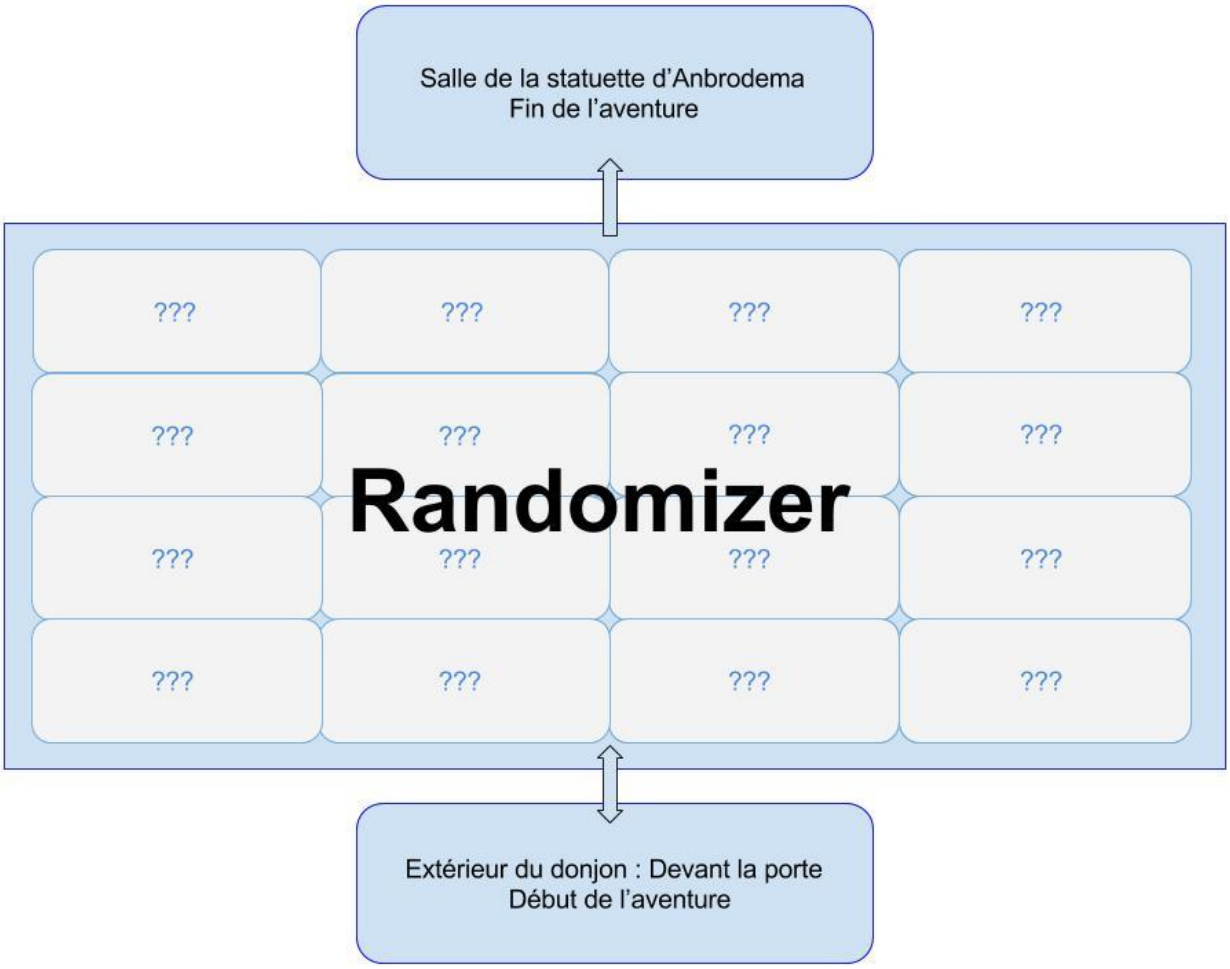
- Salle de la statuette
- Salle avec un item aléatoire (cookie / steak / mushroom / potion)
- Salle avec un miroir (affiche les caractéristiques)
- Salle avec un magicien (efet correspondant au magicien)
- Salle d'où on ne peux pas revenir
- Salle avec un orbe (beamer)
- Salle avec une fontaine (soigne le joueur)
- Salle avec un piege : fleche dans le dos
- Salle avec un piege : fosse piégée
- Salle avec une rune vous transformant en poulet (dure 5 Rooms)
- Salle avec un dragon (se déplace dans le donjon)
- Salle avec un crapaud géant
- Salle avec un « poulet monté »
- Salle avec une horde de zombies
- Salle avec un poulet (monstre faible, mais peut vous reserver une surprise s'il rencontre le dragon)
- Salle avec un spectre
- Trésorerie (contient tout les items rares)
- Salle avec un cadavre d'aventurier (looter de l'equipement)
- TransporterRoom (téléporte aléatoirement)

Monstres : Dragon, Spectre, horde de zombie, crapaud géant, squelette, Aventurier.

Gagner : Trouver la Statuette d'Anbordema et la prendre.
Perdre : Mourir (de fatigue, tomber à zero PVs, tomber dans un piège / fuir avec d'Agi, se suicider)

Combats : contre tous les monstres.

Plan du jeu



Exercice 7.5 :

Comme il y a duplication de code dans les méthodes *goRoom(.)* et *printWelcome()*, nous avons créé une fonction *printLocationInfo()* regroupant ce morceau de code utilisé dans les deux fonctions. Cette nouvelle méthode est alors appelée dans *goRoom(.)* et *printWelcome().*

Exercice 7.6 :

Création d'une méthode permettant d'obtenir la sortie (aNorthExit, aSouthExit, …) en fonction de la string rentrée en paramètre (« North », « South », …).

Modification de la classe Game pour utiliser cette nouvelle méthode.

Exercice 7.7 :

Ajout de la méthode *getExitString()* permettant d'obtenir dans une string la totalité des sorties de la pièce dans laquelle on se situe.

Exercice 7.8 :

Introduction aux HashMap.
Modification de la classe Room pour les utiliser.
Modification de la classe Game pour utiliser les nouvelles méthodes utilisant les HashMap.

Exercice 7.8.1 :

Ajout avec succès d'un déplacement vers le bas, puis d'un déplacement vers le haut pour revenir sur ses pas.

Exercice 7.9 :

Modification de la méthode *getExitString()* pour utiliser les keySet.
Compréhension des keySet.

Exercice 7.10 :

La méthode *getExitString()* avec le keySet fonctionne de cette facon :

Set<Type_des_clés> ma_variable = ma_HashMap.keySet();

met dans ma_variable l'ensemble des clés de ma_HashMap

Exercice 7.10.1 :

Javadoc faite à la création de chaque méthode elle est donc déjà complète.

Exercice 7.10.2 :

Javadoc générée.

Exercice 7.11 :

Ajouts de la méthode *getLongDescription()* qui retourne une String contenant toutes les informations sur la pièce où l'on se trouve. Simplifie les actions à faire lors d'éventuels ajouts d'information sur une salle.

Exercice 7.14 :

Ajout de la méthode *look()* qui permet d'afficher les sorties de la pièce où l'on se trouve sans avoir à en sortir puis a y re-rentrer.

Exercice 7.15 :

Ajout de la méthode *eat()* qui affiche le message *"You have eaten now and you are not hungry any more."*

Possibilité d'ajout d'aliments dans le jeu permettant de gagner des bonus a ses statistiques.

Exercice 7.16 :

Ajout des méthodes *showAll()* dans la classe CommadWord et de *showCommands()* dans la classe Parser.

Modification de la méthode *printHelp()* pour utiliser ces nouvelles méthodes.

Exercice 7.18 :

Modification de la méthode *showAll()* en *getCommandList()*. Celle-ci n'affiche plus les commandes valides, mais les retourne sous forme d'une string. C'est donc à la classe Game d'afficher ces commandes en appelant cette méthode (via la classe Parser).

Exercice 7.18.1 :

Comparaison avec le projet Zuul-better. Le seul changement effectué est d'avoir créé les Room sur une seul ligne dans la méthode *createRoom().*

Exercice 7.18.3 :

Image temporaires trouvée sur internet. Projet de dessiner sois même les images définitives.

Exercice 7.18.4 :

Titre du projet : *« Xaar-Sharoth : La quête d'Anbrodema »*

Exercice 7.18.5 :

Ajouts d'un tableau *aAllRooms* en attribut de la classe Game pour pouvoir accéder à chaque pièce de n'importe quelle méthode de la classe.

Dans *createRoom()*, un tableau de String *aAllDescriptions* contenant les descriptions est créé manuellement. Et dans un boucle for parcourant chaque case du tableau *aAllRooms* les initialise de cette façon : `aAllRooms[i] = new Room(aAllDescriptions[i])`

Première tentative d'intégration d'une génération aléatoire des Rooms.

Des méthodes statiques créent diffèrent types de Room (différenciable par leur contenu futur). Des méthodes statiques permettant de générer un des différents types de Room et des directions aléatoire sont également créées.

Chaque Room créée initialise tout d'abord un moyen de revenir sur ses pas grâces à la room précédente et la direction pour y retourner passés en paramètres. Puis, elle initialise ses sorties vers une autre Room générée aléatoirement qui initialisera ses sorties à son tour, etc…

Certaines salles ont (sans compter le retour sur ses pas) une sortie, d'autres deux, d'autres aucunes.

Problème : aucune possibilité de gérer le nombre de salle et donc problème d'overflow.

NON INTEGRE DANS LA SUITE DU PROJET

Exercice 7.18.6 :

Etude du projet `Zuul-with-images`.

Ajout de la classe UserInterface.

Les * des imports de classe ont été remplacé par les classes utilisées.

Modification de la classe Parser.

Transfert de la classe Game vers la classe GameEngine.

Ajout des éléments manquant dans la classe GameEngine.

Ajout du nouveau contenu de la classe Game.

Modification de la classe Room (ajout des images dans le constructeur et des méthodes associées).

Modification de la méthode *creatRoom()* pour initialiser les images dans la boucle for.

Exercice 7.18.8 :

Ajout de 7 boutons : « go north », « go south », « go east », « go west », « go down », « go up », « look ».

Existence du bouton « quit » même s'il n'est pas encore utilisé.

Utilisation de nombreux JPanel pour une organisation de la fenêtre design et pratique.

Exercice 7.19.2 :

Ajout du répertoire « Images » à la racine du projet et transfert des images dans ce répertoire.

Modification des strings du tableau *aAllImages* pour aller chercher les images dans le répertoire.

Deuxième tentative d'intégration d'une génération aléatoire des Rooms.

Revue de la méthode de création de la map : passage d'une génération procédurale en début de partie à une génération progressive pendant la partie.

Les méthodes statiques créant les Rooms sont gardés. Mais elles ne créent plus les sorties (seul le retour sur ses pas en conservé). Maintenant, une sortie est ajouté sur la Room dans laquelle ont ce rend lors de l'utilisation de *goRoom(.)*.

Améliorations : Maintenant capable de limiter le nombre de Rooms.

Problèmes : Des sorties sont aussi générées sur des salles déjà existante lors d'un retour en arrière, cela pose des problèmes de consistance de la map.

NON INTEGRE DANS LA SUITE DU PROJET

Exercice 7.20 :

Ajout de la classe Item.

Ajout d'un item dans une Room : La statuette d'Anbrodema dans la cinquième salle.

Exercice 7.21 :

Ajout de la description de l'item dans *getlongDescription()*.

Exercice 7.22 :

Ajout d'une HashMap d'Item permettant d'avoir autant d'Item que l'on veut dans une Room.
C'est une HashMap<String, Item> - La String correspond à un mot clef permettant d'identifier rapidement l'item en question, elle n'as pas pour vocation d'être montrée au joueur.

Exercice 7.22.2 :

Ajout d'une branche d'arbre (*branch*) et d'une épée rouillée (*rusty_sword*) dans la Room de départ.

Exercice 7.23 :

Ajout de la méthode back, celle-ci permet de retourner dans la salle que l'on vient de quitter. Si l'on se trouve dans la première salle, affiche `"You should move foreward first !"`

Exercice 7.24 :

Modification de la méthode *interpretCommand(.)* pour afficher `"Back what ?"` si il y a un deuxième mot après back.

Exercice 7.25 :

Modification de la méthode *interpretCommand()* pour malgré tout effectuer back une fois si la commande entrée est `"back back"` (ou `"back back back"`, …)

Exercice 7.26 :

Ajout d'un Stack sur allPreviousRoom, permet de stocker toutes les salles déjà visité dans l'ordre de passage. Permet ainsi d'utiliser back successivement pour revenir sur ses pas sur plusieurs salles grâce au *push()* qui ajoute la salle quittée sur le haut du Stack (à chaque utilisation de *goRoom()*) et au *pop()* qui supprime l'élément le plus haut du Stack tout en le retournant pour être utilisé (à chaque utilisation de back).

Si on retourne au début, état vérifié grave au *empty()*, affiche **"***You should move foreward first !***".**

Exercice 7.26.1 : Les deux javadoc sont générées.

Exercice 7.28.1 :

Ajout d'une méthode *test(.)*. Elle permet de vérifier que les fonctionnalités du jeu fonctionnent bien. En entrant « test [nom du fichier correspondant aux actions que l'on veut faire] », la méthode va chercher le fichier correspondant au bon endroit et effectue chaque action ligne par ligne.

Exercice 7.28.2 :

Deux fichiers de commandes créés.

- win : pour gagner en un minimum d'action
- explore : fait toute les actions suivantes : aller dans toutes les rooms, look, eat, back, suicide. Seul le quit n'est pas testé (mais suicide fait la même chose)

Exercice 7.29 :

Création de la classe Player. Migration de aCurrentRoom dans Player.

Migration d'une partie de *goRoom(.)* dans Player sous le nom de *changeRoom(.).* Est appelé dans *goRoom(.).*

Création de *getLocationInfo()*, retournant la string contentant toutes les info sur aCurrentRoom. *printLocationInfo*() n'as plus qu'à l'appeler et l'afficher.

Création de *getImage(.)*, retournant la string nécessaire pour afficher l'image. Les méthodes concernées n'ont plus qu'à appeler cette méthode et afficher l'image.

Ajout d'une HashMap<String, Room> aInventory pour stocker les Items que le joueur ramasse.

Ajout des méthodes *take(.)* et *drop(.*) permettant de prendre et de poser un item passé en paramètre.

Ajout de la méthode *getIventoryString()* qui retourne le contenu de l'inventaire sous la forme d'une string.

Ajout dans GameEngine d'une méthode *inventory()* qui affiche la string retournée par *getIventoryString().*

Ajout des méthodes *takeTest(.)* et *dropTest(.)* permettant de vérifier si les actions concernées se sont effectué et retourne le message correspondant (gameEngine l'affiche alors).

Migration d'une partie de la méthode *back()* concernant le déplacement de du player.

Ajout d'un fichier de test « takedrop ». Il teste le bon fonctionnement des actions suivantes :

- prendre un objet qui n'existe pas (dans aCurrentRoom)
- prendre un objet qui existe
- vérifier que l'objet n'est plus dans aCurrentRoom (*look()*)
- vérifier que l'objet est dans l'inventaire (*inventory()*)
- poser un objet qui n'existe pas (dans l'inventaire)
- poser un objet qui existe
- vérifier que l'objet est dans aCurrentRoom (*look()*)
- vérifier que l'objet n'est plus dans l'inventaire (*inventory()*)

Exercice 7.30 :

Cf exercice 7.29.

Ajout des méthodes *take(.)* et *drop(.*) permettant de prendre et de poser un item passé en paramètre.

Exercice 7.31 :

Cf exercice 7.29.

Ajout d'une HashMap<String, Room> aInventory pour stocker les Items que le joueur ramasse.

Exercice 7.31.1 :

Ajout d'une classe ItemList regroupant toutes les méthodes agissant sur les HashMaps d'Item pour éviter la duplication de code.
(*getItem(.), addItem(.), removeItem(.), getItemDescription(), getItemListString()* [anciennement *getInventoryString()*]).

Exercice 7.32:

Ajout d'une comparaison avant d'ajouter l'item dans l'inventaire pour savoir s'il est possible de le prendre. Si ce n'est pas le cas, le message `"You cant' take this Item"` est affiché.

Exercice 7.33 :

Cf exercice 7.29.

Ajout de la méthode *getIventoryString()* qui retourne le contenu de l'inventaire sous la forme d'une string. Le poids porté actuellement par rapport au poids total qu'il est possible de porter est aussi affiché.

Exercice 7.34 :

Ajout de l'item « cookie ». Celui-ci augmente le poids maximum que le joueur peut porter de 10 kg.

De ce fait, ajout d'une méthode *eat(.)* dans Player qui fait l'action de manger l'Item passé en paramètre (ou non s'il n'est pas comestible) et renvois la string correspondant à l'action effectuée.

Modification de la méthode *eat()* de GameEngine pour afficher ce que renvois *eat(.)* de Player.

De plus :

Modification de *goRoom(.)* pour appeler la méthode *endGameWin()* si le texte affiché par *takeTest(.)* correspond au texte du ramassage de la statuette.

Exercice 7.34.1 :

Cf exercice 7.29.

Ajout d'un fichier de test « takedrop ». Il teste le bon fonctionnement des actions suivantes :

- prendre un objet qui n'existe pas (dans aCurrentRoom)
- prendre un objet qui existe (branch)
- vérifier que l'objet n'est plus dans aCurrentRoom (*look()*)
- vérifier que l'objet est dans l'inventaire (*inventory()*)
- poser un objet qui n'existe pas (dans l'inventaire)
- poser un objet qui existe (branch)
- vérifier que l'objet est dans aCurrentRoom (*look()*)
- vérifier que l'objet n'est plus dans l'inventaire (*inventory()*)

Ajout d'un fichier « eat ». il teste le bon fonctionnement de la méthode *eat(.)*.

- prendre un objet non comestible (branch)
- se rendre dans la salle du cookie
- manger le cookie avant qu'il ne soit ramassé
- prendre le cookie
- manger l'objet non comestible
- regarder quel est le poids maximum que le joueur peut porter (afficher l'inventaire)
- manger l'objet comestible
- vérifier que le poids maximum que le joueur peut porter a augmenté de 10 (afficher l'inventaire)

De plus :

Modification du fichier « win ». Il ramasse maintenant la statuette, ce qui finit le jeu sur une victoire.

Troisième tentative d'intégration d'une génération aléatoire des Rooms.

Ajout d'une classe Map : C'est elle qui s'occupera de créer le réseau de Rooms. Transfert des méthodes statiques e Room vers Map. Transfert de *createRooms()* vers Map.

Retour sur une génération procédurale en début de partie. Les Rooms sont toujours crées sans sortie (toujours sans compter le moyen de revenir sur ses pas). Les deux première Rooms sont créées manuellement (l'entrée du donjon et sa première salle qui est forcément au Nord de la l'entrée.) Puis une boucle for ajoute une sortie à la dernière Room créée ainsi que la Room suivante. Un réseau linéaire de Room est alors créé.

Ensuite, une méthode auxiliaire *createRoomRamification(.)* qui prend en paramètre aléatoirement l'indice d'une des Rooms déjà créée, permet de rajouter sur celle-ci une autre sortie (s'assure qu'elle ne crée pas de sortie sur les sorties déjà créée de la Room). Et ainsi de suite jusqu'à ce qu'il y est le nombre voulu de Room.

Le processus est répété depuis le début tant qu'il n'y pas exactement une Room de type 5 (celle où se trouve la Statuette)

Ajout d'un getter pour accéder aux Room dans l'ArrayList depuis l'extérieur de la classe.

Ajout d'une méthode *goRandom()* dans GameEngine qui appelle *goRoom(.)* en lui passant une direction générée aléatoirement. Action répétée tant que le Player ne se trouve pas dans une Room de type 5. **N'AS PAS VOCATION A ETRE UTILISE PAR LE JOUEUR**. De ce fait, le mot-clef « gdkfjbfsdl » a été retenu pour lancer cette méthode.

Modification du fichier tes « win » puisque la map changeant à chaque fois, il ne peut y avoir de trajet prédéfinit. Il fait maintenant :

1) go gdkfjbfsdl [appelle *goRandom()*]
2) 2) take Statuette [ce qui finit le jeu sur une victoire]

Exercice 7.35 :

Etude du projet `zuul-with-enums-v1`.

Incorporation des enum et des modifications sur les classes Parser, CommandWords, Command et GameEngine.

Exercice 7.35.1 :

Ajout d'un switch dans *interpretCommand(.)* ainsi que dans *eat(.)* de Player en vue de futurs ajouts d'Item comestibles.

NOTE : Le switch fonctionne avec les Strings.

Exercice 7.41.1 :

Etude du projet `zuul-with-enums-v2`.

Modification des enum et des méthodes le nécessitant.

Exercice 7.41.2 :

Javadoc à jour et générée.

Exercice 7.42 :

Ajout d'une « limite de temps ». Celle-ci se trouve sur le poids portable maximum : à chaque changement de Room, il diminue, lorsqu'il atteint 0, le joueur ne peut plus porter d'objet. Or son but et de ramasser la Statuette, il est donc dans l'impossibilité de finir le jeu. Le joueur meurt alors « de fatigue ». Le joueur peut bien entendu augmenter son poids maximum portable en mangeant des cookies avant que celui-ci ne tombe à 0.

Ajout des boutons « look », « inventory », « suicide » à l'interface graphique.

Ajout d'un attribut booléen statique ENCOMBREMENT ayant pour but d'empêcher le Player de se déplacer lorsqu'il porte trop d'items. Il est immédiatement initialisé à false.

Il est impossible de prendre un Item si cela doit encombrer le joueur. Mais comme le poids portable maximum diminue graduellement, il est possible de se retrouver encombré. Modifications des méthodes *changeRoom(.)* et *back()* de Player pour faire le nécessaire si cette situation arrive.

Lorsque l'on drop un item, il y a maintenant comparaison entre le poids portable maximum et le poids porté, si le second est inférieur au premier, ENCOMBREMENT est repassé a false.

Exercice 7.42.2 :

Exercice fait au fur et à mesure des exercices précédents.

Exercice 7.43 :

Ajout d'une méthode *RoomType6(.)* dans Map qui crée une Room où on ne peut pas revenir sur ses pas. Celle-ci est utilisé dans la méthode *createRoomNoBack(.)* qui crée une Room de type 6 sur une salle dont l'indice est passé en paramètre.

Dans *changeRoom(.)* de Player, une fois que le changement de Room a été effectué, un test est fait pour savoir si la nouvelle aCurrentRoom est de type 6, si c'est le cas, le stack aAllPreviousRoom est vidé : on ne peut plus retourner en arrière.

Exercice 7.44:

Ajout d'une nouvelle classe Beamer, elle hérite de Item et a pour attribut supplémentaire aChargedRoom qui correspond à la Room dans laquelle elle a été chargé. Il sera utilisé pour s'y téléporter.

Dans Room, ajout d'un attribut aBeamer (si la Room en contient un).

Dans Player, ajout d'un attribut aBeamer initialisé a null (ne le sera plus quand le joueur obtiendra un beamer), des méthodes permettant de le prendre et de le poser et des méthodes pour le charger et l'activer (fire).

Dans Map, ajout d'une méthode *RoomType7(.)* qui possède un beamer, elle est utilisé par *genererRoom(.)*.

Ajout du commandWord BEAMER permettant de l'utiliser (`"beamer charge"` pour le charger, `"beamer fire"` pour l'activer et `"beamer"` pour savoir les actions que l'on peut faire avec.)

Modification de *interpretCommand(.)* pour implémenter les actions décrite ci-dessus.

Beamer a été renommé en « orb » pour coller avec l'univers du jeu. Les commandes sont donc altérées.

Exercice 7.45:

Ajout d'un fichier test « beamer ». Etant très difficile d'aller dans une salle possédant un beamer, une commande a été rajoutée : `"orb hrthgrtlgh"` permet d'obtenir un beamer directement, N'A PAS POUR VOCATION A ETRE UTILISE PAR LE JOUEUR.

Les tests suivant sont donc réalisés :

- orb fire [renvois qu'il faut un orbe chargé]
- orb charge [renvois qu'il faut un orbe]
- orb hrthgrtlgh [obtient un orbe]
- go north
- orb fire [renvois qu'il faut charger l'orbe]
- orb charge [charge l'orbe]
- back
- orb fire [active l'orbe]
- drop orb
- take orb
- orb hrthgrtlgh [renvois que le joueur a déjà une orbe]

Exercice 7.45.2:

Javadoc générée.

Exercice 7.46:

Ajout de la classe RoomRandomizer. Elle possède un attribut ArrayList de Room et une méthode *getRandomRoom()* qui renvois une Room de cette ArrayList aléatoirement.

Ajout de la classe TransporterRoom qui hérite de Room, elle a en plus un attribut RoomRandomizer. Son constructeur demande en plus un paramètre ArrayList de Room qui sera utilisé pour créer le RoomRandomizer. Elle Override la méthode *getExit(.)* de Room pour renvoyer une Room aléatoirement grâce à *getRandomRoom()* de RoomRandomizer.

Ajout d'une méthode *RoomType8(.)* dans Map générant une TransporterRoom. Celle-ci est générée après toutes les autres et est raccrochée sur une Room choisie aléatoirement. Il n'y a qu'une par génération.

Exercice 7.46.1:

Ajout du CommandWord « ALEA » permettant de fixer le Random du RoomRandomizer.

Ajout d'un attribut Integer dans RoomRandomizer qui est initialisé a null. S'il vaut null, le random est activé. Sinon un setter le met a 0. Dans ce cas, il sera utilisé par *getRandomRoom()* pour retourner une Room (qui n'est donc plus aléatoire, c'est la Room d'indice 0)

Ajout des getters nécessaire dans TransporterRoom et Map pour accéder au RoomRandomizer.

Ajout d'un attribut booléen aTest dans GameEngine qui est initialisé a false. *Test(.)* le passe à true avant d'effectuer le test puis le repasse a false une fois le test fini. Toutes les méthodes qui n'ont pas vocation à être utilisé par le joueur ne peuvent maintenant pas être activée si aTest = false.

En raison de la génération aléatoire de la Map, et de la méthode *goRandom()* utilisée pour arriver à la salle de la statuette, cette commande n'est pas utile.

Ajout d'un miroir dans la Room de Type 3, il permet e voir ses statistiques.

Ajout des effets de trois des magiciens de la Room de Type 2.

Tout ceci est géré dans le GameEngine grâce à une méthode *RoomAction()* qui est lancée juste après avoir affiché les infos de la Room dans *goRoom(.)* et *back().*

Exercice 7.46.2:

Le beamer utilise déjà l'héritage.

La Trap Door ne l'utilise pas, mais c'est n'est pas utile au vue de la classe Map.

Exercice 7.46.3:

Commentaires Javadoc déjà écris pour la plupart et rajouté pour les autres.

Exercice 7.46.4:

Javadoc générée.

Exercice 7.47:

Incorporation des Abstracts Commands.

Modification du Parser, du CommandWords, du GameEngine et de toutes les classes le nécessitant pour le bon fonctionnement des Abstracts Commands.

Exercice 7.47.1:

Création de packtages :

- pkg_Engine : comporte le GameEngine et le UserInterface.
- pkg_Commands : comporte CommandWord, CommandWords, Parser et toutes les Commands
- pkg_Componnants : comporte tout le reste

Exercice 7.47.2:

Javadoc générée

Exercice 7.48:

Ajout de la classe Character (renommée en NPC pour éviter des conflits avec la Classe Character de java). Elle comporte un nom, une image, une ligne de dialogue, et les statistiques du NPC. Le dialogue est choisi parmi un tableau de String passé en paramètre. Les statistiques sont choisies grâce à la race et la classe passée en paramètre.

Exercice 7.49:

Ajout de la classe MovingCharacter (extends NPC). Ajout de la CurrentRoom du NPC dans les attributs et des méthodes nécessaires. La méthode *move(.)* permettant de déplacer le MovingCharacter , elle est appelée dans le Command GoCommand et BackCommand.

Exercice 7.49.1:

Ajout de nombreuses salles, Items et NPCs.

Exercice 7.53:

Ajout de la methode public void static main(final String[] pArgs) dans la classe Game. Elle crée un nouveau Game.

Exercice 7.54:

Jeu exécutable sans BlueJ.

Exercice 7.58:

Fichier.jar exécutable créé.

Exercice 7.58.2:

Jeu téléchargeable sur la page web.

Exercice 7.60.2:

IHM graphique terminée.

Exercice 7.60.3:

Commentaires javadocs ajoutés.

Exercice 7.60.4:

Javadoc générée et mise sur le site.

CERTIFIE AUCUN PLAGIAT

NOTE : TOUS LES TESTS DOIVENT ETRE FAIT DEPUIS LE TOUT DEBUT (LE MENU DE CHOIX DES RACES)