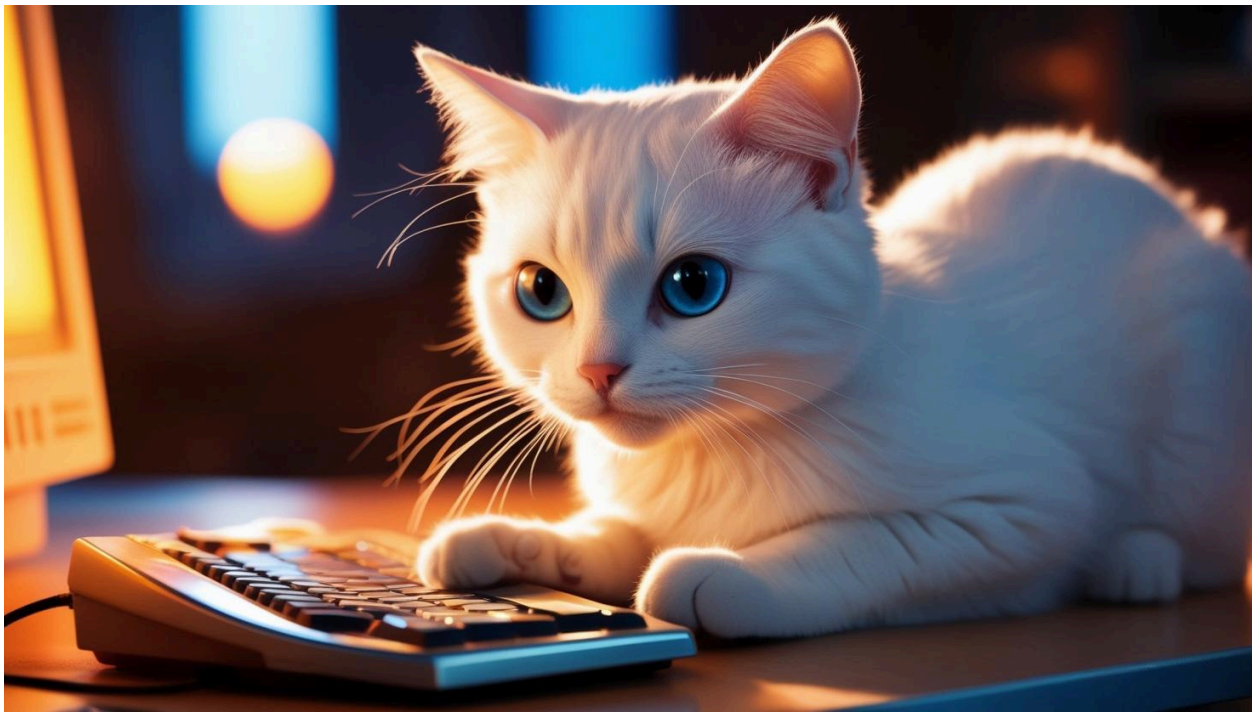
Rapport :

Un jeu d'aventure : Le secret du chat (A3P 2024/2025 J3)

I.A) Auteur

LIN Xingtong, Groupe E3T



ESIEE PARIS

I.B) Thème

Dans un laboratoire secret, un ancien soldat d'élite doit sauver sa petite sœur.

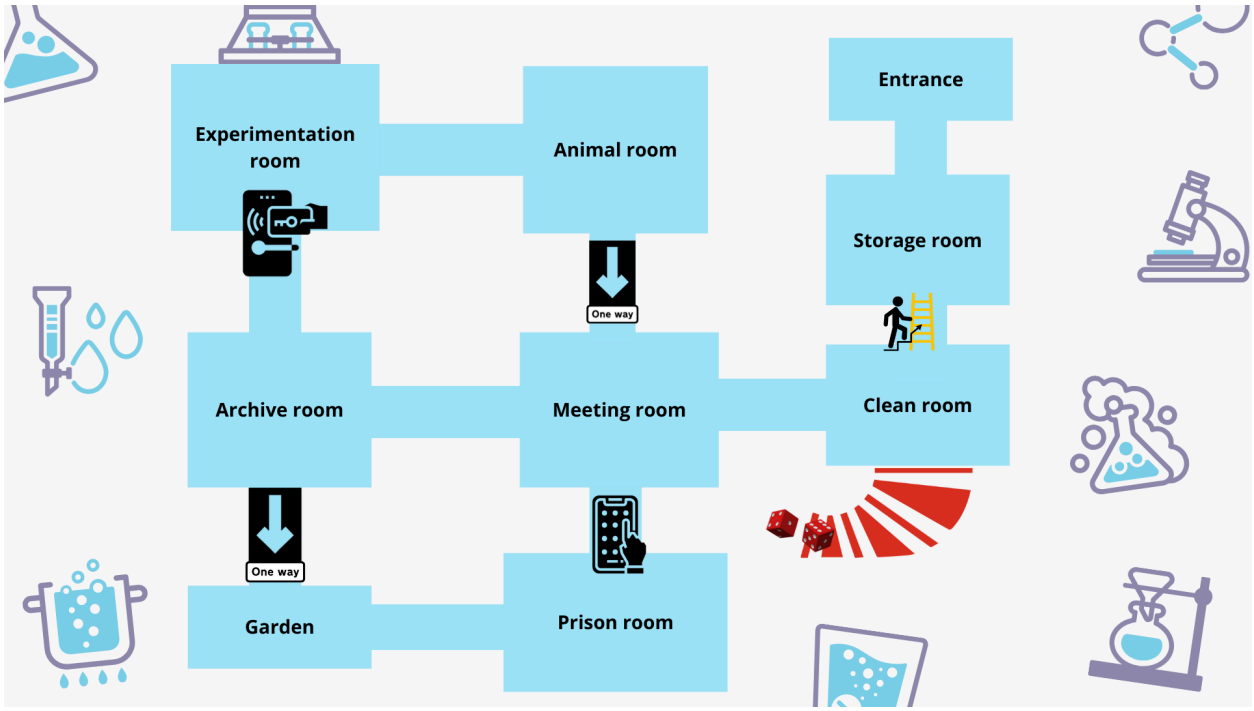
I.C) Résumé du scénario

Vous êtes Alex, un ancien soldat d'élite qui a combattu dans de nombreuses batailles, mais vous n'avez jamais pu oublier la disparition de votre sœur survenue il y a quelques années. Un jour, alors que vous sauvez accidentellement un chat, vous découvrez une bague qui vous semble familière – la bague que votre petite sœur portait avant de disparaître. Choqué, vous ressentez un lien particulier entre ce chat et votre sœur.

Pendant que vous soignez le chat, vous vous rendez compte qu'il est incroyablement intelligent. Jusqu'au jour où il saute sur votre ordinateur que vous aviez laissé allumé par inadvertance et commence à taper sur le clavier. Vous entamez alors une conversation avec lui, découvrant son histoire : il était autrefois humain avant d'être transformé en chat par un laboratoire secret, qui cherche à offrir aux riches des animaux de compagnie exceptionnellement intelligents, capables de comprendre tout ce que leur maître dit. Plus inquiétant encore, il se pourrait que votre sœur soit retenue dans ce même laboratoire.

Vous avez observé discrètement le laboratoire et découvert que la plupart des scientifiques et des gardes ne reviendront pas avant sept jours. Après avoir fait vos calculs, vous décidez de sauver votre sœur en utilisant un maximum de 100 pas. Votre mission est désormais claire : infiltrer le laboratoire, trouver des indices et sauver votre sœur sans dépasser cette limite.

I.D) Plan



I.E) Scénario détaillé

Vous êtes à l'extérieur du laboratoire secret. Après avoir discuté avec le chat, vous décidez d'infiltrer toutes les pièces du laboratoire. Pour y parvenir, vous devez traverser discrètement la salle de rangement et la salle blanche en trouvant la carte d'accès à la salle d'expérimentation, le bracelet d'accès pour la salle des prisonniers ainsi que la clé de l'armoire dans la salle des archives. Vous devrez également trouver des indices dans la salle des archives et la salle de réunion pour pouvoir identifier votre petite sœur.

C'est un moment stratégique : le laboratoire est temporairement isolé, car son système de communication a été désactivé, et la surveillance est contrôlée par votre ami Paul, qui vous couvre. Votre objectif est de retrouver votre sœur et de vous échapper avec elle du laboratoire sans dépasser la limite des 100 pas. Si vous y parvenez, vous avez gagné.

Si vous n'arrivez pas à sortir du laboratoire avec votre sœur en moins de 100 pas, vous avez perdu.

I.F) Détail des lieux, items, personnages:

lieux

Entrée :

Vous êtes à l'extérieur du laboratoire secret situé dans une grotte. Sur le sol, il y a une paire de lunettes.

Salle de rangement :

Lieu où sont entassés des objets divers, notamment la clé pour l'armoire de la salle des archives et un tissu d'invisibilité. Vous trouvez également un morceau de papier déchiré portant le nom de votre sœur – Alice, ainsi qu'un plan détaillé de tout le laboratoire.

Salle de désinfection :

Où l'on peut trouver la carte d'accès pour le laboratoire et un gâteau magique, qui ne devient visible qu'avec les lunettes spéciales. Cette salle possède une sortie qui offre un accès aléatoire à toutes les pièces, sauf la salle d'expérimentation, la salle des animaux, le jardin, la prison et l'entrée.

Salle de réunion :

Vous y apercevez un plan au tableau concernant le transfert du sujet 2566, accompagné d'un logo, ainsi qu'un slogan affiché en évidence : « Pour chaque fin, il y a toujours un nouveau départ. » (Antoine de Saint-Exupéry). Un bracelet permettant d'accéder à la prison est tombé au sol.

Salle des prisonniers :

Cellule où sont détenus les sujets humains. Dans un coin, vous trouvez une porte portant les inscriptions "Alice" et "2566".

Salle des animaux :

Cellule où sont détenus les animaux issus des expériences. Vous y trouvez une cage vide marquée du numéro "2566".

Salle des archives :

Bibliothèque contenant tous les dossiers et enregistrements d'expériences. Vous découvrez le prénom Alice accompagné de sa photo et de son passé. Vous y rencontrez également un NPC nommé Sophie.

Salle d'expérimentation :

Laboratoire où les expériences sont menées, équipé de plusieurs lits, parmi lesquels un est marqué du numéro 2566. Il y a également plusieurs animaux : un chat, un lapin et un chien. Sur le lit 2566 repose votre sœur Alice. Au sol, vous apercevez un beamer.

items

Carte d'accès pour la salle d'expérimentation.

Clé pour consulter les dossiers dans l'armoire de la salle des archives.

Bracelet pour accéder à la salle des prisonniers.

Morceau de papier déchiré avec le prénom Alice.

Tissu d'invisibilité peut aider le joueur à cacher.

"Bonne nuit" : un gaz endormant dans une bouteille de verre.

Gâteau magique : Manger ce gâteau magique permet d’augmenter la capacité à porter des objets (de 100 % à 120 %).

Beamer : Un objet qui peut charger une salle, puis, lorsqu'il est activé (fire), permet de se déplacer directement dans la salle chargée. Il ne peut être utilisé qu'une seule fois (charge et activation).

personnages

Alex : Le joueur.

Alice : La petite sœur à sauver.

Paul : Ami et assistant technique, il peut vous aider.

Sophie : Fille d'un scientifique, elle fait ses devoirs dans la salle des archives. En discutant avec elle et en gagnant sa confiance, vous pouvez obtenir un "Bonne nuit", ainsi qu'un indice vous avertissant de ne pas vous faire repérer par les membres de la salle d'expérimentation.

Louis : Un scientifique. Si vous êtes découvert par lui, il désactivera le brouillage du signal, ce qui ramènera la majorité du personnel au laboratoire secret. Cela réduira votre nombre de pas restants à 70 (au lieu de 100).

I.G) Situations gagnantes et perdantes

Gagnant : Vous réussissez à sortir du laboratoire avec Alice.

Perdant : Vous échouez à sortir du laboratoire avec Alice dans la limite des pas autorisés.

I.H) Éventuellement énigmes, mini-jeux, combats, etc.

Mini-jeux (pas fait) :

Jouer à cache-cache avec les personnes en vous cachant derrière différents objets.

I.I) Commentaires (ce qui manque, reste à faire, ...)

Ce qui manque :

1. Intégrer des mini-jeux pour enrichir l'expérience.

Ce qui reste à faire dans la future :

1. Corriger la structure des codes en créant une classe distincte pour chaque commande, afin d'améliorer la lisibilité et la maintenance du code.
2. Ajouter un NPC nommé Louis, capable de se déplacer avec le joueur, réduisant ainsi le nombre de pas restants pour le joueur tout en augmentant la tension du jeu, car il cherche à attraper le joueur.
3. Ajouter dans les fichiers de test la commande "talk" pour tester l'interaction avec les NPC.

II. Réponses aux exercices (à partir de l'exercice 7.5 inclus)

7.4 :

3. b) J’ai enregistré ce projet dans mon github et j'ai modifié la méthode createRooms() pour qu'elle crée les salles avec des descriptions correspondant à mon idée de jeu. Pour la sortie vers une salle aléatoire, j'ai créé une variable de type Room et j’ai affecté l'objet Room animal à cette variable pour le moment.

4) J'ai suivi les instructions comme indiqué dans le sujet.

7.5 :

Comme j’ai déjà créé une méthode similaire pour éviter la duplication de code, j'ai modifié ma méthode en la renommant et en ajoutant l'affichage de la localisation actuelle, car mon ancienne méthode n'affiche que les sorties possibles. J'ai également apporté des modifications pour que mon code soit exactement conforme à celui du livre. Par exemple, j'ai ajouté un println à la fin et remplacé le println de "Exits: " par un print, afin d'afficher "Exits: " et les sorties possibles sur la même ligne.

7.6 :

J'ai suivi les instructions du livre en modifiant les attributs pour les rendre privés, puis en utilisant des getters dans la classe Game. Pour résoudre le problème du message "Unknown direction", après avoir lu les échanges, j'ai combiné les trois cas dans mon code. Le premier cas est "There is no door!", qui signifie que la direction est valide mais qu'il n'y a pas de salle dans cette direction. Le deuxième cas est lorsque la direction (le mot après go) est un mot non-valide (pas défini dans notre code), ce qui fait que la méthode getExit() retourne UNKNOWN_DIRECTION, et elle affiche "Unknown direction!". Si ce n'est aucun de ces deux cas, elle affecte this.aCurrentRoom = vNextRoom pour définir la salle suivante et ainsi éviter les erreurs.

7.7 :

J'ai ajouté la méthode getExitString() avec la signature demandée. J'ai ensuite défini une variable locale vString, qui contient les noms des sorties disponibles en fonction des conditions (vérification avec if que la sortie de la salle n'est pas vide). J'ai ensuite appelé cette méthode pour afficher les sorties dans la classe Game.

7.8 :

J'ai modifié mon code comme sur le livre et, en plus de cela, j'ai corrigé la méthode getExitString() en utilisant la boucle for (typeKey nomKey : nomHashMap.keyset()).

7.8.1 :

J'ai ajouté les directions up et down entre la salle de rangement et la salle de désinfection.

7.9 :

Je pense que le changement est à peu près similaire à ce que j'ai fait en 7.8, sauf que je n'ai pas créé de variable locale de type Set<String>. Donc, je n'ai pas modifié mon code, j'ai juste importé les dépendances pour Set et Iterator en commentaire au cas où.

7.10 :

Ma fonction getExitString (sans variable locale de type Set<String>) :

Elle définit d'abord la valeur de retour avec un début fixe ("Exits:"). Après, Grace à la boucle (j'ai dit en 7.8), elle parcourt toutes les clés (de type String dans notre cas) de HashMap et ajoute les directions qui ont une sortie à la valeur de retour (séparées par un espace). A la fin, fonction retourne une chaîne de caractères qui commence par "Exits: " puis suivie des sorties possibles.

7.10.1 :

J'ai utilisé /** */ pour faire des commentaires de description pour toutes les méthodes dans les classes Room, Game et Command. Et pour les paramètres, j'ai ajouté des annotations @param nomParam pour les décrire.

7.10.2 :

j’ai visualisé ma javadoc dans BlueJ et tout semble correct.

7.11 :

J'ai créé la méthode getLongDescription() comme dans le livre, puis j'ai appelé cette méthode dans la classe Game à l'intérieur de la méthode printLocationInfo().

7.14 :

J'ai modifié la déclaration de aValidCommands[] en ajoutant le mot static. J'ai aussi changé sa manière d'affectation et ajouté "look" dans aValidCommands[]. Avant, je l'affecte dans le constructeur de CommandWords, mais maintenant j'ai fait comme dans le livre, en déclarant et créant tout en une seule ligne. Comme c'est statique, j'ai supprimé le “this.” dans la classe CommandWords. Dans la classe Game, j'ai ajouté la méthode look(), qui affiche la description de la salle actuelle. En cas de présence d'un deuxième mot après "look", il affiche un message indiquant qu'il n'y a pas encore de fonction pour observer un objet spécifique. J'ai aussi ajouté un cas dans méthode processCommand pour appeler la méthode look().

7.15 :

J'ai ajouté la commande eat avec une méthode qui affiche "You have eaten now and you are not hungry any more." Comme pour 7.14, j'ai ajouté eat dans aValidCommands[], puis j’ai ajouté un cas eat dans méthode processCommand() qui appelle la méthode eat().

7.16 :

J'ai ajouté la méthode showAll() dans la classe CommandWords comme le livre. Cette méthode utilise une boucle for pour parcourir toutes les commandes de type String dans le tableau validCommands et les afficher sur la même ligne, puis affiche une ligne vide à la fin. Pour utiliser cette méthode en réduisant le couplage, je l'appelle d’abord dans classe Parser, puis classe Game appelle la méthode qui est dans Parser.

7.18 :

J'ai changé la procédure showAll() en une fonction String getCommandList() qui retourne la liste des commandes valides. J'ai fait le même type de changement dans la classe Parser, puis dans la classe Game, j'ai affiché cette liste de commandes au lieu d'appeler la méthode directement.

7.18.1 :

J'ai comparé le code en mode de lecture et je n'ai rien changé.

7.18.3 :

J'ai trouvé des images pour mes 8 pièces, au format 800*600.

7.18.4 :

Le titre de mon jeu est "Le Secret du Chat". J'ai modifié le message de bienvenue dans la classe Game.

7.18.5 :

J'ai ajouté une ArrayList pour stocker toutes les salles que j'ai créées, à l'exception de la salle transportée.

7.18.6 :

J'ai modifié mes codes pour qu'ils soient pareils avec les fichiers zuul-with-images.jar :

1) J'ai ajouté un attribut aImgName (avec son initialisation dans le constructeur) et son getter dans la classe Room.

2) Dans la classe Parser, nous n'utilisons plus Scanner pour lire les entrées depuis le clavier. Maintenant, cela se fait dans la classe UserInterface via un JTextField. J'ai donc supprimé toutes les parties liées à Scanner dans la classe Parser. J'ai ajouté un paramètre String dans la méthode getCommand(). Ensuite, j'ai utilisé StringTokenizer pour diviser la chaîne en mots séparés.

3) J'ai créé une nouvelle classe GameEngine, avec tout l'ancien code de la classe Game. J'ai aussi ajouté un nouvel attribut aGui avec son setter. Tous les System.out.println ont été remplacés par this.aGui.println. Dans la procédure createRooms(), j'ai passé un nouveau paramètre pour chaque new Room : le nom du fichier d'image correspondant. J'ai également modifié la procédure printLocationInfo() pour afficher l'image via aGui.showImage(). J'ai transformé la fonction quit() en procédure quit() et ajouté aGui.enable(false) lorsque la commande est quit. La fonction processCommand() est devenue la procédure interpretCommand() qui prend un paramètre String au lieu de Command. J'ai aussi ajouté du code pour afficher notre commande envoyée avec signe <, et pour transformer le paramètre String en Command en utilisant aParser.getCommand(). Enfin, j'ai supprimé la méthode play().

4) J'ai copié le code des classes Game et UserInterface à partir de zuul-with-images.jar.

5) Concernant les imports dans la classe UserInterface, j'ai supprimé tous les ...* génériques, puis ajouté les imports un par un. Parfois, j'utilisais Quick Fix dans VSCode, et parfois, lorsque Quick Fix ne fonctionnait pas, je survolais le mot dans les fichiers de zuul-with-images.jar pour identifier son package, puis j'ajoutais les imports manuellement.

Et bien sûr, j'ai ajouté les images trouvées dans le dossier du projet.

Explication des composants graphiques :

JFrame : la fenêtre principale d'une interface graphique (GUI) en Java.

JTextField : un champ de saisie de texte à une seule ligne.

JTextArea : permet la saisie ou l'affichage (dans notre cas) de texte sur plusieurs lignes. Utilisable avec un JScrollPane.

JLabel : permet d'afficher du texte non éditable ou des images (dans notre cas).

JScrollPane : permet d'ajouter des barres de défilement horizontales et/ou verticales autour de composants.

JPanel : permet d'organiser les composants graphiques dans une fenêtre.

7.18.8 :

Comme les commandes de déplacement sont les plus souvent utilisées, j'ai ajouté toutes les directions disponibles sous forme de boutons. Pour avoir suffisamment de place pour les six boutons, j'ai créé un nouveau JPanel et utilisé un BoxLayout.Y_AXIS afin d'aligner les éléments en colonnes, contrairement au BorderLayout qui impose des limites d'espace. Pour éviter que mon code ne devienne trop long, j'ai utilisé un tableau de JButton, un tableau de String, et une boucle for pour créer les boutons, les ajouter au panneau, et leur assigner un ActionListener. Dans la fonction actionPerformed, j'ai ajouté une vérification : si pE.getActionCommand() est une direction, la commande cmd go est utilisée ; sinon, processCommand() est appelé pour le texte saisi. J'ai également ajouté removeActionListener pour tous les boutons de direction dans la procédure enable en cas de désactivation. Et j'ai utilisé getMaximumSize et setMaximumSize pour mettre tous les boutons à la même longueur.

7.19.2 :

J'ai suivi les instructions de la question et modifié, dans la procédure showImage de UserInterface.java, le répertoire d'images pour qu'il pointe vers "Images/".

7.20 :

J'ai ajouté un nouvel attribut Item dans la classe Room. Dans la fonction getLongDescription(), j'ai inclus l'utilisation de getItemDescription(), qui renvoie la description de l'objet. Si cet attribut est vide, un message indique qu'il n'y a pas d'objet. J'ai ajouté la procédure setItem(final String pDescription, final int pWeight), qui crée et ajoute un objet dans Room. J'ai placé un objet dans la salle de stockage et dans la salle de défectation. J'ai également créé la classe Item avec deux attributs (description et weight) et des méthodes getter pour chacun. La classe comporte aussi une fonction qui renvoie une chaîne de caractères décrivant l'objet avec sa description et son poids.

7.21 :

How should all the information about an Item present in a Room be produced ?
- Dans la classe GameEngine, j'ai utilisé setItem() pour fournir ces informations.

Which class should produce the String describing the Item ?
- Dans la classe Item, j'ai créé la fonction getLongDescription(), qui renvoie une chaîne de caractères combinant la description et le poids de l'objet.

Which class should display it ?
- Dans la classe Room, la fonction getItemDescription() appelle Item.getLongDescription() lorsque l'objet n'est pas nul. Ensuite, Room.getLongDescription() appelle getItemDescription().

7.22 :

J'ai remplacé la procédure setItem() par addItem(), et l'attribut aItem est devenu un HashMap nommé aItems. La méthode getItemDescription a été renommée en getItemsDescription, car j'ai ajouté une boucle for (comme 7.8) pour afficher tous les objets.

7.22.2 :

Pour répondre à la demande 1, j'ai ajouté un trognon de pomme abandonné dans l'entrée (ma pièce initiale). Quant à la demande 2, j'ai déjà des pièces qui contiennent plusieurs objets.

7.23 :

J'ai implémenté la commande back en ajoutant la méthode public Room leaveRoom(String direction), qui utilise Room getExit(String direction). Je n'ai pas utilisé String getLongDescription() directement dans leaveRoom(), car dans GameEngine, nous avons déjà printLocationInfo(), qui affiche le résultat de getLongDescription() dans l'interface utilisateur.

À mon avis, dans le livre, cette intégration est prévue pour afficher les informations dans le terminal, mais cela n'est pas faisable dans notre cas (version graphique).

7.26 :

J'ai remplacé l'ancien attribut String aStockDirection par Stack<Room> aPreviousRooms. J'ai utilisé push() pour stocker une pièce au sommet(le 1er position) de la pile, empty() pour vérifier si la pile est vide puis afficher notification, et pop() pour supprimer le premier élément de la pile, peek() pour retourner le premier de la pile.

7.26.1 :

J'ai exécuté les deux commandes dans la racine de mon dossier projet (où se trouvent mes fichiers .java). Cela m'a donné plusieurs erreurs et avertissements. J'ai corrigé ses fautes, et j'ai bien compris javadoc accepte que le format /** ... */ et non /* ... */ ou //

7.28.1 :

J'ai ajouté le nv commande test et nv procédure test en utilisant Scanner, hasNextLine(), nextLine() et try catch, j'ai aussi utilisé finally pour fermer Scanner dans tt les cas.

7.28.2 :

J'ai créé 3 fichiers de commandes en type txt, le 1er pour le plus court chemin aller dans salle de laboratoire et le 2eme chemin pour pouvoir parcourir toutes les salles et trouver des indices et le dernier qui teste toutes les commandes. Et j'ai testé les 3 fichiers et ils fonctionnent correctement.

7.29 :

J'ai ajouté 2 nouvelles commandes, take et drop, et créé la nouvelle classe Player pour stocker des informations personnelles comme sa salle actuelle, son poids autorisé, son prénom, ainsi que des fonctions pour modifier la pile "salles précédentes". J'ai également ajouté un dialogue au tout début du jeu pour permettre au joueur d'entrer son prénom. J'ai découvert cette idée en lisant les échanges après la question.

J'ai ajouté plusieurs nouveaux objets dans certaines salles pour permettre au joueur de trouver des indices. De plus, j'ai ajouté aussi la commande pour manger le gâteau magique

(eat cake) et une commande pour observer un objet avec son nom (look itemName) afin d'obtenir une description détaillée.

J'ai inclus une gestion du poids, avec l'affichage du poids autorisé et du poids porté en pourcentage, ainsi que le poids des objets. J'ai également ajouté de nouveaux attributs aux objets, comme leur nom, s'ils peuvent être ramassés, et s'ils sont visibles. Enfin, j'ai modifié l'objet trognon de pomme en une monture de lunettes qui permet de voir des choses invisibles, comme un gâteau magique ou un tissu invisible.

7.30 :

J'avais déjà réalisé cette étape dans l'exercice 7.29 à cause d'une mauvaise compréhension de la question 29.

7.31 :

Toujours en raison de cette confusion, j'avais déjà ajouté dans l'exercice 7.29, dans la pièce initiale, une monture de lunettes qui peut être ramassée sans aucune condition et qui permet au joueur de visualiser des objets magiques. J'avais également intégré une collection d'objets dans la classe Player en utilisant HashMap. Et pendant le jeu, en tapant la commande look, le joueur peut visualiser tous les objets qu'il possède.

7.31.1 :

J'ai créé une nouvelle classe ItemList pour y regrouper toutes les fonctions liées à la gestion des objets. Pour la classe GameEngine peut rester inchangée, j'ai conservé les méthodes liées à ItemList (dans classe Player et Room) en les faisant simplement appeler les méthodes correspondantes de la classe ItemList.

Pour la méthode getItemsNames(), j'ai ajouté un paramètre String nommé type afin de différencier les cas. Lorsque type="room", les objets sont affichés selon une condition if. Pour les autres valeurs de type, tous les objets sont affichés en utilisant une condition qui est toujours vraie (if(true)), ce qui est rendu possible grâce à la expression ternaire : condition == true ? actionSiVrai : actionSiFaux.

7.32:

Toujours à cause d'une mauvaise compréhension de la question 29, j'ai déjà implémenté en question 29 la notion de poids en utilisant une unité basée sur des pourcentages (%). Par défaut, le joueur commence avec une capacité de 100 %. Après avoir mangé le gâteau magique, sa capacité augmente à 120 %.

7.33:

Dans la question 7.29, j'avais intégré la fonctionnalité d'affichage des objets dans la commande look. Pour mieux organiser le code, j'ai créé une nouvelle commande items et déplacé cette fonctionnalité dans une nouvelle procédure appelée items().

7.34:

J'avais déjà implémenté cette fonctionnalité dans la question 7.29. J'en ai également précisé certains détails dans ma réponse à la question 7.32.

7.34.1:

J'ai mis à jour mes fichiers de test en ajoutant toutes mes nouvelles commandes et j'ai testé le jeu. Tout fonctionne correctement.

7.34.2:

J'ai mis à jour mon Javadoc en complétant les commentaires pour toutes les fonctions et attributs.

7.42:

J'ai ajouté une méthode dans la classe GameEngine ainsi qu'un nouvel attribut dans la classe Player pour compter le nombre de commandes réussies en utilisant go, back, take, et drop. Le maximum est une constante à 100. Une fois ce nombre atteint, un message Game Over s'affiche, et l'interaction de l'interface utilisateur est désactivée.

7.42.2:

Je veux garder l'IHM actuelle.

7.43:

J'ai ajouté une nouvelle salle qui est jardin pour y placer la deuxième trapdoor. J'ai également ajouté le comportement suivant : si la salle actuelle n'est pas une sortie de la prochaine salle, tous les éléments de la pile aPreviousRooms sont supprimés. Dans tous les cas, la pile aPreviousRooms stocke/ajoute la prochaine salle afin que la notification relative au one-way s'affiche correctement.

7.43.1:

J'ai fait marcher les deux commandes précédentes pour générer la JavaDoc.

7.44:

J'ai ajouté une nouvelle classe appelée Beamer, qui est un type spécifique d'Item. Beamer est lié à deux commandes, avec une limite imposée : le Beamer peut uniquement être chargé et utilisé une seule fois. Après avoir utilisé (fire) le Beamer, la commande back n'est plus disponible (la pile aPreviousRooms est entièrement vidée après son utilisation). J'ai également ajouté une méthode dans la classe Room pour afficher le nom de la salle actuelle pour rendre les notifications plus claires.

7.45.1

J'ai mis à jour les fichiers long.txt et ideal.txt pour inclure les nouvelles commandes liées aux nouvelles salles (garden) et items (beamer et ses deux commandes) que j'ai ajoutées. Tout fonctionne correctement.

7.45.2

J'ai généré la javadoc en utilisant les mêmes commandes que précédemment.

7.46

J'ai ajouté la classe RoomRandomizer, qui contient une méthode statique findRandomRoom permettant de retourner une salle parmi quatre salles que j'ai définies (index de 0 à 3).

J’ai également créé la classe TransporterRoom, qui est une sorte de Room. Elle contient un attribut List<Room> en plus et elle redéfinit les méthodes : 1. getExit, qui utilise méthode statique findRandomRoom pour retourner une sortie aléatoire ; 2. getExitString, qui indique que la sortie peut être "n’importe où que l’utilisateur veut".

Dans la classe GameEngine, j’ai modifié pour la commande back, si la salle précédente est une TransporterRoom, le message suivant est affiché : "You can't back on the one-way road".

7.46.1

J’ai d’abord ajouté un bouton back pour faciliter les tests, car j’ai tapé cette commande trop souvent pendant les essais passés.

Ensuite, j’ai ajouté la commande alea dans aValidCommands[].

J’ai également inclus la salle TransporterRoom (que j’ai créée) dans l’ArrayList<Room> afin d’y accéder dans la procédure alea. J’ai ajouté un nouvel attribut aInTestMode et modifié sa valeur dans la procédure test : 1. Quand test est appelé, aInTestMode est défini à true. 2. À la fin de test, dans la section finally, je remets aInTestMode à false.

Dans la procédure alea, j’ai décidé d’utiliser alea + indexDeSalle comme commande avec 2ème mot. Pour cela, j’ai utilisé un Integer afin de pouvoir passer null à TransporterRoom puis à RoomRandomizer.findRandomRoom. Cela facilite le processus : Si indexDeSalle est null, la méthode retourne une salle aléatoire. Sinon, elle retourne la salle correspondant à l’index que j’ai passé. Pour modifier l’attribut spécifique dans TransporterRoom, je sais que j’ai placé cette salle en dernière position dans l’ArrayList<Room>. J’ai donc utilisé (TransporterRoom) this.aRooms.get(aRooms.size() - 1) pour l’indiquer.

Enfin, j’ai modifié uniquement le fichier long.txt pour tester les commandes alea + indexDeSalle et alea.

7.46.3

J'ai ajouté les informations relatives aux auteurs (moi), à la version (mois/année) et une courte description pour chaque classe que j'avais codée.

7.46.4

J'ai généré la javadoc comme 7.26.1.

7.48

J'ai créé une nouvelle classe appelée CharacterNPC (j'ai utilisé ce nom pour éviter le conflit avec une classe existante nommée Character). Les dialogues des NPC sont gérés à l'aide d'un tableau de String. Ce tableau est organisé de la manière suivante :
- Les premières phrases correspondent aux questions/phrases posées par le NPC, celles qui attendent une réponse du joueur (ordre : de première à la dernière).
- Ensuite, la phrase de fin de conversation est variable en fonction des réponses du joueur. Elles sont indexées en fonction du nombre de bonnes réponses données (de la phrase pour aucune bonne réponse à la phrase pour toutes les réponses sont bonnes).
Pour structurer les conversations et leurs choix de réponses correspondants, je me suis inspiré de l'algorithme de tri par tas (où les enfants sont aux positions 2i+1 et 2i+2, tandis que le parent est à la position i). Chaque index de tableau est une option de réponse, et le joueur utilise la commande talk + l'index pour faire son choix.
Le NPC va donner un objet au joueur si, et seulement si, celui-ci a sélectionné toutes les réponses correctes (réponses gentilles).
Concernant l'interaction avec NPC, deux types de conversations sont possibles : Le joueur donne sa réponse / Le NPC donne sa parole. Pour cela, j'ai créé deux méthodes talk avec des paramètres différents selon le type d'interaction. De plus, le NPC fournit des indices (la phrase de fin de conversation) au joueur avec un degré de nuance ou de précision variable, en fonction du nombre total de bonnes réponses données par le joueur.

Dans la classe CommandWords , j'ai ajouté une nouvelle commande talk.

Dans la classe GameEngine, j'ai intégré un NPC nommé Sophie, avec son objet appelé "Bonne nuit". J'ai également implémenté la commande qui permet de voir la description d'un NPC, look + nameNpc. J'ai codé la commande talk, qui traite le type de second mot fourni (String ou entier) pour appeler la méthode talk correspondante. Et j'ai ajouté l'affichage de Npc pour la salle actuelle.

Dans la classe Room, j'ai ajouté une liste de NPCs présents dans la salle, ainsi que des méthodes permettant d'ajouter un NPC, de chercher un NPC spécifique à un nom, et de décrire les NPCs présents dans la salle (ces informations retourné sont utilisées dans la classe GameEngine).

à la fin

J'ai ajouté une notification pour informer le joueur qu'il a gagné. J'ai également intégré la possibilité d'examiner les objets dans l'inventaire du joueur avec la commande look + nomObjet, ce qui permet désormais d'examiner les objets présents dans la salle et ceux dans l'inventaire du joueur.

J'ai ajouté des salles et objets verrouillés : une porte entre la salle d'expérimentation et la salle des archives, qui nécessite une carte d'accès ; une porte entre la salle de réunion et la salle des prisonniers, qui nécessite un bracelet ; et une armoire dans la salle des archives, qui nécessite une clé pour être ouverte.
Lorsque les objets requis sont récupérés par le joueur (placés dans son inventaire), il peut accéder à ces endroits ou examiner ces objets. Sinon, il n'est pas possible de voir leurs descriptions ni d'entrer dans les salles verrouillées.

J'ai également modifié mes fichiers de test. Le fichier "court" est le plus rapide, conçu uniquement pour accéder à la salle d'expérimentation. Le fichier "ideal" est conçu spécifiquement pour gagner. Enfin, le fichier "long" teste toutes les commandes, à l'exception de la commande "alea", car ce fichier mène lui aussi jusqu'à la victoire.

J'ai aussi amélioré les indices pour aider le joueur à trouver sa sœur Alice, qui s'est transformée en chat. Ainsi, le seul moyen de réussir le jeu est de sortir avec ce chat.

Enfin, j'ai généré les javadocs comme auparavant.

III. Mode d'emploi (commandes disponibles dans ce jeu)

Toutes les commandes disponibles dans ce jeu :

- ➢ go + direction
- ➢ help
- ➢ quit
- ➢ look [+ NomNpc/NomObjet]
- ➢ eat + NomObjetComestible
- ➢ back
- ➢ test + nomFichierSansSuffixe
- ➢ take + NomObjet
- ➢ drop + NomObjet
- ➢ items
- ➢ charge
- ➢ fire
- ➢ alea (uniquement utilisé dans les fichiers de test, donc les joueurs peuvent l'ignorer)
- ➢ talk + NomNpc

IV. Déclaration obligatoire anti-plagiat (*)

pour mettre les boutons à la même longueur : j'ai trouvé la solution sur ligne.

J'ai utilisé ChatGPT pour corriger et améliorer mes phrases dans ce rapport.

J'ai aussi utilisé ChatGPT pour avoir l'idée de BoxLayout.Y_AXIS. (voir la capture d'écran à la fin)

V: Capture d’écran

jenvie d'avoir 2 place pour west et 2 place pour east

vPanel.add(this.aBouttonWest, BorderLayout.WEST);
vPanel.add(this.aBouttonEast, BorderLayout.EAST);

2/2

Pour ajouter deux boutons pour la direction **West** et deux pour **East**, vous pouvez utiliser un sous-`JPanel` pour chaque côté (**West** et **East**) dans le panneau principal. Ensuite, vous placez les boutons dans ces sous-panneaux avec un `BoxLayout` ou un `GridLayout` pour les organiser verticalement.

Voici comment faire :

Exemple de code

```java
import javax.swing.*;
import java.awt.*;

public class ButtonLayoutDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Direction Buttons");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

        // Panneau principal avec BorderLayout
        JPanel vPanel = new JPanel(new BorderLayout());

        // Sous-panneaux pour les boutons West et East
        JPanel westPanel = new JPanel();
        westPanel.setLayout(new BoxLayout(westPanel, BoxLayout.Y_AXIS));
        JPanel eastPanel = new JPanel();
        eastPanel.setLayout(new BoxLayout(eastPanel, BoxLayout.Y_AXIS));

        // Création des boutons West
        JButton buttonWest1 = new JButton("Go West 1");
        JButton buttonWest2 = new JButton("Go West 2");
        westPanel.add(buttonWest1);
        westPanel.add(Box.createVerticalStrut(10)); // Espace entre les boutons
        westPanel.add(buttonWest2);

        // Création des boutons East
        JButton buttonEast1 = new JButton("Go East 1");
        JButton buttonEast2 = new JButton("Go East 2");
        eastPanel.add(buttonEast1);
        eastPanel.add(Box.createVerticalStrut(10)); // Espace entre les boutons
        eastPanel.add(buttonEast2);

        // Ajouter les sous-panneaux au panneau principal
        vPanel.add(westPanel, BorderLayout.WEST);
        vPanel.add(eastPanel, BorderLayout.EAST);

        // Ajouter le panneau principal au cadre
        frame.add(vPanel);
        frame.setVisible(true);
    }
}
```

Explication du code

1. **Sous-panneaux** `westPanel` et `eastPanel` :
   - Ces sous-`JPanel` utilisent un `BoxLayout` vertical (`BoxLayout.Y_AXIS`), ce qui permet de placer les boutons en colonne.
   - Vous pouvez également utiliser un `GridLayout(2, 1)` si vous préférez une disposition plus stricte avec deux rangées.
2. **Boutons West et East** :
   - `buttonWest1` et `buttonWest2` sont ajoutés au panneau `westPanel`, et `buttonEast1` et `buttonEast2` au `eastPanel`.
   - `Box.createVerticalStrut(10)` est utilisé pour ajouter un espacement vertical de 10 pixels entre les boutons pour une meilleure lisibilité.
3. **Ajout au** `BorderLayout` :
   - Les sous-panneaux sont ajoutés dans le `BorderLayout.WEST` et `BorderLayout.EAST` du panneau principal `vPanel`.

Résultat attendu

Ce code organise deux boutons verticalement de chaque côté (West et East) de la fenêtre, chacun avec un espace entre eux.