Space Hermès (titre provisoire).

I.A) Auteur : Jason GAVALDA

I.B) Thème : Rentrer sur Terre faire votre rapport sur la vie extraterrestre.

I.C) Résumé du scénario :

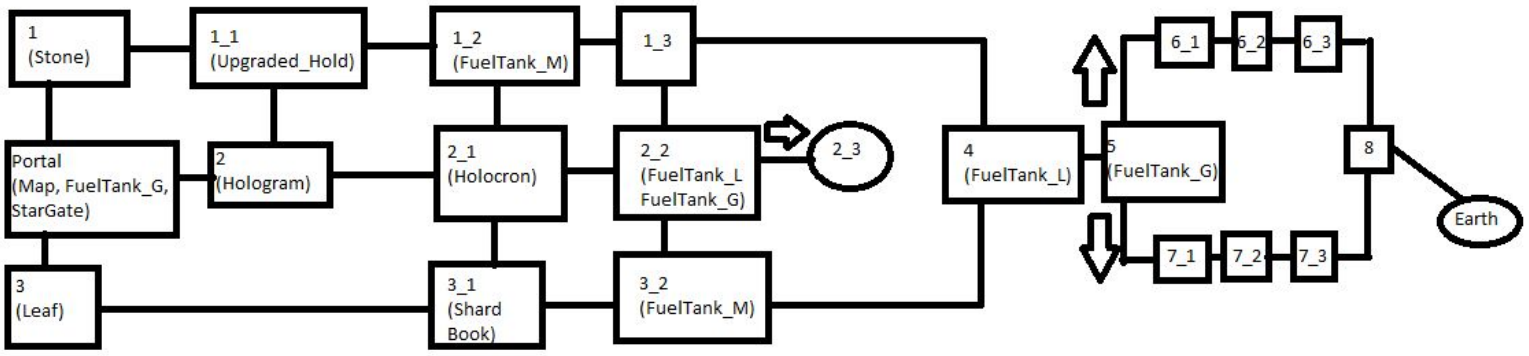
Dans un futur lointain où l'humanité a trouvé le moyen de voyager à travers la galaxie, vous êtes le commandant de la première mission chargée d'établir un contact avec la vie extraterrestre, en vous rendant dans un système stellaire dont la Terre a reçu un signal étrange quelques décennies auparavant.

Seulement, une fois arrivé, vous et votre flotte de 3 vaisseaux êtes attaqués et vous êtes contraints de fuir. Seul le vaisseau du commandant (le vôtre) en réchappe.

Vous voilà perdus, vous et votre équipage, dans l'espace avec pour seul objectif de rentrer sur Terre sain et sauf. Mais vous allez vite vous rendre compte que vous n'êtes pas seul dans la galaxie… En effet, vous êtes au beau milieu d'une guerre entre deux civilisations extraterrestres diamétralement opposées : l'une fanatiquement religieuse, l'autre douée de technologie extrêmement avancée.

Pour trouver la route pour rentrer chez vous, vous allez devoir fouiller les différentes traces laissées par ces deux races sur les différents systèmes sur votre route, mais fatalement, il vous faudra vous allier à l'un ou à l'autre camp…

I.D) Plan.



I.E) Scénario détaillé.

Introduction (pas présente dans le jeu… mais pourrait l'être dans son manuel par exemple) :

En l'an 2212, l'humanité a la surprise de recevoir un signal étrange en provenance d'un système de la constellation d'Orion. Ce signal semble avoir été émis par une civilisation extraterrestre, mais rien n'a permis de le décoder. Il a néanmoins bouleversé l'humanité entière, qui a mis de côté ses différents pour se concentrer sur l'exploration spatiale. Après 3 siècles de recherches sur le voyage interstellaire et d'exploration du système solaire dans ses moindres recoins, l'humanité a mis au point un moteur capable de voyager rapidement entre les étoiles, à des vitesses supraluminiques. En l'an 2550, l'expédition vers le système d'origine du signal est prête. Vous êtes le commandant de cette expédition, constituée de trois vaisseaux. Après près d'un an de voyage, votre expédition arrive enfin dans le système en question. Vous localisez

rapidement la seule planète habitable du système et y descendez. Vous n'y trouvez que ruines et désolation. Alors que vous vous apprêtez à repartir, vous remarquez un objet d'un éclat étrange, qui semble vous appeler. Vous vous en emparez et vous quittez la planète pour repartir vers le système solaire. Mais à peine vous sortez de l'atmosphère que vous remarquez plusieurs objets en approche rapide sur vous. A peine avez vous distingué qu'il s'agit d'autres vaisseaux spatiaux que des rayons lumineux sont tirés vers vous. N'étant pas armé, vous cherchez à prendre la fuite, mais votre système de voyage interstellaire n'est pas rechargé !
Toutes les tentatives pour semer vos assaillants sont vaines, et vous assistez impuissant à la perte des deux autres vaisseaux vous accompagnant. Alors que tout semble perdu, vous apercevez dans l'espace une sorte d'anneau flottant dont l'intérieur est illuminé. Tout comme l'artefact, il vous appelle. Ne voyant pas d'autre solution, vous vous engagez à l'intérieur, juste à temps pour éviter un tir qui devait vous détruire. Après un certain temps dans une sorte de tunnel lumineux, vous ressortez par un anneau similaire, un autre système solaire s'étendant devant vous. Vous êtes perdu, mais avec une preuve de votre découverte entre vos mains. Votre but est de retrouver votre chemin vers la Terre !

Scénario dans le jeu :

1ère Phase de jeu : l'exploration.

Vous sortez du "portail" alors que votre second vient vers vous vous briefer sur la situation (dialogue de début du jeu). Vous localisez une planète habitable dans le système stellaire et décidez de vous y rendre. Similairement, vous n'y trouvez que ruines. Mais l'artefact réagit sur un objet mystérieux. Une sorte de carte stellaire est projetée sous vos yeux : trois étoiles proches y sont indiquées.

A partir de là l'aventure commence vraiment : vous êtes libre de naviguer entre les systèmes. Chaque système dévoile un peu plus ce qui se trame dans le secteur : vous allez rapidement comprendre que des batailles ont eu lieu sur certaines planètes que vous visitez et qu'il n'y a pas de traces de survivants, ou d'autres planètes qui ont été occupées par le passé, mais qui ne le sont plus lors de votre passage.

Vous allez constater que deux espèces semblent dominer ce secteur : une race insectoïde fanatique, les Zaynites, et une race reptilienne technologiquement très avancée et rationnelle : les Tal'Xins.

Celles-ci semblent se livrer une guerre sans merci depuis plusieurs siècles, à en croire ce que vous trouverez lors de votre exploration.

NB : Certains systèmes cachent des objets à ramasser (comme des armes) ou des technologies à découvrir qui seront utiles pour passer la deuxième phase du jeu. Le joueur est donc encouragé à explorer un maximum afin d'obtenir un vaisseau capable de résister à la deuxième phase du jeu.

Après avoir avancé dans l'exploration, vous finissez par arriver au beau milieu d'une bataille spatiale entre les deux espèces, que vous devez fuir pour ne pas être touché par quelque projectile perdu. Dans votre fuite, vous apercevez une planète occupée par les Tal'Xins, mais assiégée par les Zaynites. Les deux belligérants réclament votre aide en échange de la leur pour retrouver votre route vers le système solaire.

Vous êtes confronté au choix d'aider l'une ou l'autre espèce, qui en retour vous aidera à sa manière (les deux choix permettent d'arriver à la fin du jeu).

2ème Phase du jeu (linéaire) : la guerre.

Si vous avez choisi d'aider les Zaynites, vous en apprenez un peu plus sur leur manière d'agir. Ils vouent un culte à une divinité qui en échange leur accorde d'immenses pouvoirs. Vous comprenez bien vite que toute leur civilisation est bâtie autour de cette religion et que quiconque vient remettre en question leur mode de vie ou leur croyances le paie de sa vie, ce qui explique en grande partie leur guerre avec les Tal'Xins, qui sont vus comme des hérétiques.

Après une cérémonie religieuse destinée à vous accueillir (et aussi pour décider si oui ou non vous méritez leur aide), ils vous affirment que leur Dieu leur a parlé et qu'il vous montrera la voie du retour si vous aidez les Zaynites dans leur guerre.

Votre première mission sera de détruire un chantier spatial Tal'Xin où une nouvelle arme est en construction. Ceci fait, vous allez devoir attaquer la base principale des Tal'Xins dans le secteur.

NB : Le système de combat n'ayant pas été implémenté, cette partie du jeu correspond plus à un enchaînement d'histoires qui est choisi par le joueur selon le chemin qu'il décide d'emprunter. Il n'y a pas d'objectif à accomplir ici, simplement suivre la route en observant ce qui se passe, à condition au préalable d'avoir assez de carburant pour le voyage !

En cas de succès, vous serez accueillis en héros par les Zaynites, qui via un autre rituel, vous permet de voir le chemin du retour vers la Terre. Ils vous laissent une amulette en signe d'amitié.

Vous pouvez alors revenir vers la Terre et raconter votre périple.

Si vous choisissez d'aider les Tal'Xins, vous en apprenez un peu plus sur leur manière d'agir. Ils sont dévoués corps et âme à la science et au progrès, et méprisent tous ceux qui refusent de voir l'univers de manière rationnelle. Reconnaissant en vous une espèce capable de suivre cette voie, ils acceptent de vous aider à condition que vous les aidiez à repousser les Zaynites du secteur, les Tal'Xins voyant en eux les ennemis absolus de toute rationalité.

Votre première tâche consiste à atteindre un temple des Zaynites situé dans un système proche pour enquêter sur une importante source d'énergie détectée en provenance du temple. Sur place, vous découvrez que les Zaynites accomplissent un rituel dans lequel ils sacrifient des prisonniers Tal'Xins ! Vous remarquez que pour chaque Tal'Xin exécuté, ils semblent obtenir des pouvoirs de plus en plus terrifiants ! Vous devez secourir au plus vite les Tal'Xins avant qu'il ne soit trop tard !

En cas de réussite, vous parvenez à fuir la planète avec les survivants, qui, en prenant contact avec leurs confrères, vous demandent de détruire la base principale des Zaynites dans le secteur (analogue à la quête des Zaynites si vous avez choisi l'autre camp).

En cas de victoire, les Tal'Xins vous remercient en mettant leur connaissances à votre disposition pour retrouver la Terre. Ils y parviennent sans trop de difficulté, leur cartographie de la galaxie étant très complète. En signe d'amitié, ils vous donnent un holocron contenant nombre de leur connaissances, mais vous indiquent aussi que vous et votre espèce devrez trouver vous-mêmes la plupart d'entre-elles, l'holocron ne contenant que des informations sur l'univers et ses propriétés, stipulant que bien que leur modèle est très avancé, il est sans doute imparfait, et que plutôt que de vous le fournir, ils espèrent que vous reviendrez avec des solutions différentes que celles qu'ils ont trouvées aux nombreuses questions de l'univers.

Vous pouvez alors revenir vers la Terre faire part de vos découvertes.

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

Les lieux de la 1ère partie permettent de raconter l'univers dans lequel le joueur a débarqué : une zone de la galaxie où deux espèces aliens diamétralement opposées semblent s'affronter depuis des siècles. En effet, le joueur ne découvre souvent que des vestiges des batailles entre les deux races, où de leur simple passage dans un système. La 1ère partie vous permet de comprendre à qui vous avez affaire, mais également à trouver des preuves de vos découvertes, importantes pour terminer le jeu.

Les items se divisent en 2 catégories dans le jeu : les items utilisables, comme les réservoirs de carburant, ou l'amélioration de la soute, qui aident le joueur à continuer le jeu ; et les items dits de collection, qui ne peuvent être utilisés, mais qui serviront à la fin du jeu pour déterminer la victoire ou la défaite du joueur.

L'absence de réels personnages détaillés est volontaire, car non nécessaire à la progression du scénario : en effet les personnages rencontrés ne le sont que très brièvement, et ne constituent alors pas d'intérêt à être développés.

I.G) Situations gagnantes ou perdantes.

Conditions de victoire : Atteindre le sol terrestre avec des preuves de votre rencontre avec la vie extraterrestre : c'est à dire posséder les items Stone, Leaf, Holocron ou Hologram, Book ou Shard.

Conditions de défaite :

- Tomber à court de carburant sans pouvoir ravitailler (déplacement impossible sans carburant.
- Atteindre la Terre sans rapporter une preuve de la vie extraterrestre.

I.H) Énigmes, mini-jeux, combats, etc.

Non développé (voir partie I.I).

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

Un système de combat aurait été intéressant pour la 2nde partie compte tenu de sa linéarité. Cela aurait pu ajouter une condition de victoire ou de défaite pour le joueur ainsi que de nouvelles possibilités.

Ce système de combat a été abandonné compte tenu du temps nécessaire pour le développer par rapport aux autres contraintes données par les exercices.

Le jeu est plutôt facile dans l'ensemble dans le sens où le seul moyen de perdre est de se retrouver à court de carburant, qui ne manque pas dans la 1ère partie du jeu, et qui est donné en condition suffisante avant la 2ème partie pour finir le jeu. L'autre moyen de perdre est d'arriver au système solaire sans avoir de preuve de vos découvertes, qui se comptent au nombre de 2 : une par race alien.

II) Réponses aux exercices (à partir du 7.5 inclus).

7.5) Création d'une nouvelle procédure printLocationInfo() dans la classe Game, permettant d'afficher d'une part la description de la Room courante, ainsi que les sorties disponibles pour cette Room (rappel : toutes les Rooms (ou presque) du jeu correspondent à un système stellaire).

Elle permet d'éviter des duplications de code dans les méthodes goRoom et printWelcome, qui désormais y font appel directement plutôt que d'effectuer toutes deux une même suite d'instructions pour afficher ce qui est dit ci-dessus.

7.6) Ajout de deux nouvelles directions, Up et Down, qui sont utilisées pour descendre sur les planètes des systèmes. Elle ont donc été ajoutées sur chaque room.

Création d'un accesseur getExit(String pDirection) dans la classe Room qui renvoie la pièce atteinte si un déplacement dans la direction pDirection est effectué, ou null si il n'y a pas de pièce dans cette direction.

Celui-ci permet entres autres de réduire le couplage entre les classes Room et Game, afin de remplacer les différents attributs de sortie des pièces par un objet HashMap.

Ainsi, les méthodes de la classe Game, notamment goRoom, y font directement appel, ce qui la simplifie grandement.

7.7) Ajout d'une fonction String getExitString() dans la classe Room permettant d'obtenir les descriptions des destinations possibles d'une room. Ce travail n'a donc plus à être effectué par la méthode printLocationInfo() de la classe Game, qui fait désormais appel directement à getExitString() pour cela désormais. On diminue donc encore le couplage entre les deux classes.

7.8) Remplacement des attributs représentant les sorties dans la classe Room par un seul objet de type HashMap. La méthode getExitString() a donc été modifiée en conséquence.

7.10) La méthode getExitString() renvoie une description de toutes les destinations possible d'une room donnée, l'ensemble de ces destinations étant ajouté dans la String retournée par une boucle qui ajoute une à une ces sorties dans la String, selon qu'une clé correspondante ait été trouvée par le keySet() implanté précédemment.

7.10.1 et 2) Complétion et génération de la JavaDoc pour chacune des méthodes de chaque classe.

7.11) Ajout d'une fonction String getLongDescription() dans la classe Game qui retourne une description détaillée de la pièce avec un "You are" en début de description et une retour à la ligne à la fin de celle-ci.

Elle est désormais appelée par la méthode printLocationInfo.

7.14) Ajout d'une procédure look() dans la classe Game, qui ne permet pour l'instant que de donner la description de la pièce actuelle.

7.15) Ajout d'une procédure fire() dans la classe Game, qui sera utilisée pour le combat lorsque celui-ci aura été implémenté. Elle n'affiche pour le moment qu'une chaîne de caractères.

7.16) Ajout d'une procédure showAll() dans la classe CommandWords qui a pour rôle d'afficher les commandes valides via System.out, ainsi que d'une autre procédure showCommands() dans la classe Parser qui utilise showAll() sur l'attribut aValidCommands de cette classe.

La méthode printHelp() de la classe Game utilise dorénavant showCommands() pour afficher les commandes valides. On évite ainsi un couplage entre la classe Game et la classe CommandWords en utilisant showCommands() plutôt que showAll().

7.18) Remplacement de showAll() par une fonction getCommandList() qui renvoie la même chose sous forme de String (plutôt que de les afficher directement). On modifie en conséquence les méthodes showCommands() (qui devient aussi une fonction retournant une String) de la classe Parser et la procédure printHelp() de la classe Game pour s'adapter à ce changement.

7.18.1) Aucun changement n'a eu lieu lors de la comparaison à Zuul-better.

7.18.3) Début de la recherche d'images : 4 images issues d'internet sont utilisées pour illustrer les premiers systèmes. Une recherche plus approfondie sera effectuée plus tard pour les autres systèmes. Les références vers les images sont situées dans la partie "Anti-Plagiat".

7.18.4) Le titre du jeu est ramené à "Hermes" le rôle du dieu grec étant d'être le messager des dieux, le rôle du joueur est ici de délivrer à la Terre le message "la vie extraterrestre existe, et elle ressemble à ça".

Un ajout de ligne de System.out.println est ajouté au début de la méthode printWelcome() pour annoncer au joueur qu'il est le commandant de la mission "Hermes" chargée de la découverte de la vie extraterrestre.

7.18.5) Création d'une HashMap<String, Room> aRooms en attribut de la classe Game. Elle est initialisée vide par le constructeur de la classe Game.

Un accesseur pour elle est également créé en cas de besoin pour la suite du projet.

La HashMap est remplie par la méthode createRooms() à mesure que celle-ci créé les Rooms : une ligne aRooms.put("Nom du système", Room); est placée après chaque création de Room.

7.18.6) Après étude du zuul-with-images, on prend la mesure d'ajouter des images au jeu.

Cela commence par l'ajout d'un attribut String aImageName dans Room contenant l'adresse d'un fichier image. Cela amène un nouveau paramètre pImage au constructeur qui peut alors initialiser ce nouvel attribut pour chaque Room selon le fichier passé. Un accesseur getImageName() est également créé dans Room pour avoir accès à l'attribut, notamment dans UserInterface.

Une nouvelle version du Parser issue de zuul-with-images est incorporée au projet, qui utilise un StringTokenizer à la place d'un Scanner, qui n'est donc plus utile puisque le StringTokenizer est capable de repérer les mots tapés sans lui.

Ajout de la classe UserInterface issue du zuul-with-images qui va créer une interface pour le jeu et afficher les images dessus.

La totalité du contenu de la classe Game est transféré dans une nouvelle classe GameEngine. La classe Game est dorénavent doté d'un attribut aEngine qui est initialisé par son constructeur qui se contente de créer un nouveau GameEngine.

En conséquence, avec l'apparition de l'interface, un autre attribut pour UserInterface aGui pour Game est créé et initialisé dans le constructeur par un new UserInterface initialisé par aEngine. Enfin l'interface est modélisé sur aEngine par un setGUI utilisant aGui.

Dorénavant, toute méthode utilisant des system.out.print voient ceux-ci remplacés par des this.aGui.println("String à afficher") .

La méthode play() n'existe plus puisque l'interface graphique fait son travail.

La méthode processCommand() se retrouve fondamentalement modifiée : elle devient non plus une fonction mais une procédure, qui ne prend non plus une commande en paramètre mais une String, qui est simplement la ligne de commande tapée par le joueur. Le Parser analyse alors cette ligne via la méthode getCommand.

Il n'y a plus de booléen à renvoyer pour processCommand() dans le cas où le joueur quitte le jeu, puisqu'on crée une nouvelle méthode endGame() dans GameEngine qui affiche dans l'interface le message de fin du jeu et ferme l'interface.

De plus, une méthode nommée processCommand() existant déjà dans la classe UserInterface, qui a pour but d'effectuer les commandes

sur l'interface, la méthode processCommand() de la classe GameEngine sera dorénavant renommée interpretCommand().

Les imports effectués dans UserInterface étant trop larges, j'ai choisi de n'importer que ce qui est utile pour cette classe, se qui se résume à ceux-ci :

```
1  import javax.swing.JFrame;
2  import javax.swing.JTextField;
3  import javax.swing.JTextArea;
4  import javax.swing.JLabel;
5  import javax.swing.ImageIcon;
6  import javax.swing.JScrollPane;
7  import javax.swing.JPanel;
8  import java.awt.Dimension;
9  import java.awt.BorderLayout;
10 import java.awt.event.ActionListener;
11 import java.awt.event.WindowAdapter;
12 import java.awt.event.WindowEvent;
13 import java.awt.event.ActionEvent;
14 import java.net.URL;
15
```

7.18.8) Ajout d'un import dans UserInterface : l'import de la classe JButton, afin d'ajouter un attribut JButton aButton dans cette classe.

Dans la méthode createGUI() de cette classe, on ajoute une ligne pour initialiser l'attribut aButton par un nouveau bouton "fire", qui va déclencher la commande fire. On ajoute également dans createGUI() une ligne vPanel.add(this.aButton, BorderLayout.EAST) pour afficher ce bouton sur la droite de l'interface. Toujours dans cette méthode, on ajoute une ligne this.aButton.addActionListener (this) à la suite de celle pour l'EntryField pour que l'appui sur le bouton déclenche bien la commande fire.

Il est alors nécessaire d'ajouter un test dans la procédure actionPerformed() pour détecter l'appui sur le bouton. On utilise donc un getSource() sur l'ActionEvent pE paramètre de la méthode.

7.19.2) Création du répertoire images dans le projet.

Certaines Rooms sont désormais créées avec des images issues de ce répertoire, toutes les images pour le projet n'étant pas encore

décidées. Les Rooms sans images ont une String vide comme paramètre.

7.20) Ajout d'une nouvelle classe Item définis par un nom aNom (String), une description aDescription(String) et un poids aWeight (double). Trois accesseurs pour ces attributs sont également ajoutés. Un attribut HashMap<String (Nom de l'item), Item (L'item)> est également créé dans la classe Room qui peut désormais contenir des Items dedans. Ajout d'une fonction String getItemString() qui renvoie les noms des items contenus dans la salle, d'une procédure setItem(pNom, pItem) qui place un item dans cette Room.

De plus une autre HashMap aItems est créée dans GameEngine pour contenir tous les items du jeu. C'est grâce à elle que les items vont pouvoir être créés et ajoutés dans les Rooms, via une nouvelle procédure createItems() qui crée les Items et les met dans cette HashMap. La méthode createRooms() pioche alors dans la HashMap pour ajouter les items dans les Rooms.

7.21) L'information concernant les items de Room est produite directement dans Room, puisque c'est une information propre à la Room. Les accesseurs permettent d'avoir accès à ces informations.

La String décrivant la description des items est en revanche propre à chaque Item, il est donc naturel qu'il s'agisse d'un attribut de Item.

C'est en revanche à GameEngine d'afficher cette description via son Interface.

7.22) Les Rooms pouvant déjà contenir plusieurs items (ceci ayant été anticipé via l'utilisation de HashMaps), cette partie de l'exercice est déjà traitée dans le 7.20. La méthode setItem correspond aussi à la méthode addItem demandée dans l'exercice. Enfin printLocationInfo() utilise désormais la méthode getItemString() de Room pour afficher au joueur les items présents dans la pièce.

7.22.2) Les items sont implantés selon les contraintes de l'exercice.

7.23) Création d'une méthode (naïve) back() dans la classe GameEngine qui ramène le joueur dans la pièce précédemment visitée.

Un nouvel attribut Room aPreviousRoom (initialisé null) est créé dans la classe GameEngine. Désormais la méthode goRoom() utilisée pour la commande go met dans cet attribut la Room où le joueur se trouve avant de bouger.

La méthode back() en elle-même contient un test vérifiant si l'attribut aPreviousRoom contient bien une Room (ce qui n'est pas le cas au lancement du jeu !) et renvoie un message d'erreur au joueur dans le cas où il n'y a rien dans l'attribut. Si tout se passe bien, la Room actuelle du joueur est stockée dans une variable Room vPreviousRoom, les joueur est déplacé dans aPreviousRoom et vPreviousRoom est stocké dans l'attribut aPreviousRoom. La description de la pièce et des items présents est affichée sur l'interface, ainsi que l'image si elle existe.

Le problème assez évident de cette méthode back() est qu'elle ne permet en réalité de ne bouger qu'entre 2 Rooms, en passant de l'une à l'autre. L'exercice suivant va corriger cela.

7.26) L'attribut aPreviousRoom devient une Stack aPreviousRooms contenant dans l'ordre les pièces visitées par le joueur.

La méthode back() n'a plus qu'à dépiler la Stack pour déplacer le joueur. Elle doit toujours vérifier si la Stack contient quelque chose avant de déplacer. La méthode goRoom doit donc désormais empiler la Room dans laquelle se trouve le joueur avant de le déplacer.

Ainsi, l'utilisation successive de back permet de remonter une à une les Rooms visitées par le joueur, et non plus seulement la dernière qu'il a visitée.

7.28.1) Création d'une nouvelle procédure test(String pFileName) dans GameEngine qui lit le fichier dont le nom est passé en paramètre via l'utilisation d'un Scanner (importé en conséquence). La procédure est bâtie pour gérer l'exception FileNotFound dans le cas où le fichier n'existerai pas. Elle lit le fichier, qui contient une commande par ligne et les exécute en utilisant interpretCommand(). La procédure étant une

nouvelle commande, le nécessaire est fait pour que le jeu la reconnaisse comme telle (ajout dans les validCommands, test dans interpretCommand(), ou le deuxieme mot est le nom du fichier à lire).

Le fichier essai.txt est créé avec des tests des différentes commandes du jeu.

7.28.2) Création de 3 nouveaux fichiers de commande :
court.txt, qui teste go east, fire et test.
shortcut.txt, qui teste les commandes de déplacement pour faire le chemin le plus rapide vers la fin du jeu.
slowrun.txt, qui parcourt l'intégralité des salles du jeu, et teste toutes les commandes.

7.29) Création d'une classe Player qui récupère de GameEngine les attributs aCurrentRoom et aPreviousRoom (qui nécessitent désormais des accesseurs pour pouvoir être utilisés par GameEngine.

Le dernier attribut est une String aName contenant le nom du joueur, qui est initialisé en paramètre du constructeur par une String pName. De même la Room de départ du joueur est également un paramètre du constructeur. La classe GameEngine contient désormais un nouvel attribut aPlayer qui est initialisé en tant que "Commander" et qui débute toujours le jeu au système "Portal".

Comme le mouvement du joueur semble être plus adapté à être géré par Player que par GameEngine, les parties consistant au mouvement des méthodes goRoom() et back() sont déplacées dans la classe Player, goRoom() se contentant désormais de faire appel aux procédures ainsi créées dans Player et de faire l'affichage.

7.30) Création d'une HashMap dans Player en tant qu'attribut aInventory contenant les Items portés par le joueur, les noms des items étant les clés; initialisé vide à la création du joueur. Création des commandes take et drop séparées en 2 parties comme goRoom() et back(). take(pItemName) place l'item de la Room dont le nom est passé en paramètre dans l'inventaire du joueur, tout en le retirant de la Room en question. La commande take est implémentée dans les

validCommands et le test dans interpretCommands vérifie si l'item est présent dans la Room, en renvoyant un message d'erreur au joueur si ce n'est pas le cas.

Même principe avec drop(pItemName), qui enlève l'item dont le nom est placé en paramètre dans l'inventaire du joueur pour le placer dans la Room courante du joueur. interpretCommands() teste de son côté si l'item est bien présent dans l'inventaire du joueur et renvoie un message d'erreur au joueur dans le cas contraire.

7.31) Le joueur peut déjà emporter un nombre indéfini d'item grâce à son inventaire créé dans l'exercice précédent.

7.32) Création d'une classe ItemList qui n'a comme attribut qu'une HashMap<String(Nom de l'item), Item (L'item lui-même)> aItemList.

La classe contient les méthodes getItemList(), qui retourne un Set de la HashMap; la procédure addItem, qui ajoute un item dans la HashMap dont le nom et l'item sont passés en paramètre; une fonction booléenne checkItem(pItemName), qui vérifie si l'item dont le nom est passé en paramètre est présent dans l'ItemList; la fonction getItem(pItemName) permet d'accéder à l'Item dont le nom est passé en paramètre dans l'ItemList; et enfin la fonction removeItem(pItemName) qui enlève l'item dont le nom est passé en paramètre de l'ItemList.

Dorénavant, les attributs HashMap utilisant des Items dans les classes Room et Player sont donc des ItemList et il faut utiliser les méthodes d'ItemList pour les utiliser, ce qui implique de nombreux changements et ajouts dans ces deux classes, notamment au niveau des constructeurs, des méthodes take et drop, setItem, getRoomItem (qui est l'accesseur des objets de chaque Room), removeRoomItem (qui permet d'enlever un item spécifique de Room en utilisant removeItem).

La méthode getItemString() initialement placée dans Room est déplacée dans ItemList puisque la collection d'item utilisée dans Room est désormais une ItemList, et que cette méthode pourra être utilisée plus tard pour afficher l'inventaire du joueur. Une nouvelle méthode getItemsString() faisant appel à getItemString() est donc désormais utilisée dans Room.

7.33) Ajout d'une nouvelle commande "Inventory" (ajoutée dans les validCommands). La procédure interpretCommands() se voit implémenter un nouveau test pour vérifier l'entrée de cette commande. Si la commande est reconnue, l'interface va afficher le contenu de l'inventaire grâce à une nouvelle méthode getPlayerInventory() dans la classe Player qui elle-même renvoie une String composée de "In your Hold, " suivie du renvoi de getItemString() sur l'inventaire du joueur.

7.34) Ajout d'un nouvel Item "Upgraded_Hold" qui "améliore" la soute du vaisseau du joueur pour que celle-ci puisse accueillir un poids maximum de 50. La perspective d'utiliser la commande "eat" dans le jeu ne semble plus vraiment adaptée au jeu en lui-même. Elle est donc remplacée par une commande "use" qui prend en second mot le nom d'un item, et l'utilise. En l'occurrence, le seul item utilisable à ce stade du jeu est l'Upgraded_Hold, mais une utilisation sur la Map par exemple pourrait être utile.

J'ai donc ajouté une nouvelle procédure use(final String pItemName) dans GameEngine qui vérifie si l'item dont le nom est passé en paramètre est dans l'inventaire du joueur, puis effectue l'action selon l'item à utiliser. Si l'item n'est pas présent, un message indique au joueur qu'il ne possède pas l'item. Si l'item est présent mais non utilisable, un message l'indique au joueur.

Qui dit nouvelle commande dit nouveau test pour interpretCommand() qui utilise le second mot tapé comme paramètre de use().

7.42) Ajout d'une nouvelle mécanique de jeu : la gestion du carburant du vaisseau, qui sert de "temps restant" au joueur avant de perdre. J'ai donc créé un nouvel attribut entier pour le joueur : aFuel, qui est initialisé à 5 au début du jeu. J'ai également ajouté l'accesseur et le modificateur correspondants. Chaque mouvement effectué par le joueur réduit son nombre de Fuel restant de 1. Si le joueur tombe à 0 de Fuel, il perd la partie. Pour permettre au joueur d'atteindre la fin du jeu, des items FuelTanks ont été ajoutés dans certains systèmes. Tous ne donnent pas

la même quantité de carburant : le FuelTank_G en donne 5, le FuelTank_M en donne 3 et le FuelTank_M en donne 1. La méthode goRoom() de la classe GameEngine a donc été modifiée en conséquence : si le joueur a 1 de carburant ou plus, il se déplace normalement au prix de voir son carburant baisser de 1 (cette baisse de carburant étant implémentée dans la méthode move()). Sinon, le jeu vérifie si du carburant est présent dans le système. Si c'est le cas, le jeu indique au joueur qu'il devrait regarder un peu dans le système si il peut en trouver. Si il n'y en a pas, un message indique au joueur que son moteur a cassé à cause du manque de carburant et qu'il est condamné à errer où il se trouve pour l'éternité. Le jeu se ferme en conséquence (appel à endGame()).

7.43) Le scénario était déjà prévu pour contenir ces TrapDoors puisque la deuxième partie du jeu repose sur un choix sur la race alien à aider, il semble logique qu'une fois le choix fait, on ne puisse revenir en arrière !

J'ai donc supprimé les sorties west permettant de revenir en arrière dans les Rooms de cette partie, et ajouté une méthode isExit() dans la classe Room qui prend en paramètre la Room dans laquelle le joueur souhaite se rendre et vérifie si cette Room fait partie des sorties disponible en renvoyant true ou false selon que le test est validé ou pas.

isExit() est donc dorénavant utilisé par back() qui renvoie donc un message d'erreur à l'utilisation si la Room précédente n'est pas parmi les sorties disponibles.

7.44) Ajout du "beamer" qui porte le nom de StarGate dans le jeu.

Comme décrit dans l'énoncé de l'exercice, il permet, une fois obtenu par le joueur, de retourner dans une Room déjà visitée si elle a été stockée au préalable par la StarGate, sachant que celle-ci ne se "rappelle" que d'une Room à la fois, et qu'une fois la téléportation faite, une nouvelle Room doit être chargée. Le nouvel Item est donc implémenté dans le jeu dès le premier système visité (il doit donc être ramassé par le joueur). Deux nouvelles commandes ont été ajoutées : "charge" qui permet de stocker une Room dans la StarGate, et "warp"

(fire dans l'énoncé), qui permet de se téléporter dans la Room stockée, au prix d'utiliser une charge de carburant (le joueur ne peut donc pas se téléporter sans carburant) et de vider la Stack contenant les PreviousRooms. Ces commandes sont donc ajoutées sous forme de procédures dans Player et sont ajoutées dans les validCommands. Elles manipulent une nouvelle Stack aStarGate, nouvel attribut de Player, qui permet de stocker une Room via la StarGate. Les commandes sont codées de sorte que cette Stack ne contienne au plus qu'une seule Room.

7.45.1) Mise à jour des fichiers de test pour qu'ils prennent en compte les nouvelles mécaniques de gestion de carburant et du beamer. Les test shortcut essai et slowrun sont les principaux concernés.

Abandon du système de combat, qui entraîne la disparition de la commande "fire" devenue inutile. Le bouton utilisé sur l'interface est celui utilisé pour le "warp".

7.45.2) Mise à jour des javadoc.

7.46) Ajout de la mécanique de TransporterRoom. Suite à la lecture du chapitre du livre il est décidé d'ajouter une nouvelle salle vers la fin de la 1ère partie qui renvoie aléatoirement entre 4 salles, dont l'une d'elle permet d'avancer dans le jeu, les autres étant le début des embranchements de la carte de jeu. Cependant, le caractère "aléatoire" de la salle nécessite l'ajout de deux nouvelles classes : un RoomRandomizer, qui contient en attribut une ArrayList aRoomList contenant des Rooms : les différentes possibilités de sortie de la pièce et un attribut Random aRNG permettant de générer des nombres aléatoires.
L'autre classe, TransporterRoom, hérite de la classe Room en rajoutant un objet de type RoomRandomizer comme attribut. Dans cette dernière classe, un Override de getExit() de Room permet d'utiliser le concept de sortie aléatoire via une méthode findRandomRoom() créée dans RoomRandomizer, qui donne une Room générée aléatoirement en sortie via la génération d'un entier qui doit correspondre à un index de

l'ArrayList du RoomRandomizer. Il est ajouté au préalable dans ces deux classes des procédures fondamentales, telles qu'ajouter des Rooms comme possibilité de sortie via une procédure addRoom() dans RoomRandomizer, réutilisée dans TransporterRoom ; retirer des Rooms de sortie ; et vérifier si une Room est une sortie possible d'une TransporterRoom donnée. Les accesseurs et modificateurs des attributs de ces classes sont créés pour pouvoir être manipulés par GameEngine() : en effet, une nouvelle TransporterRoom est créée dans la méthode createRooms() de GameEngine, et les méthodes fondamentales citées ci-dessus se chargent de lui donner ses sorties.

Il est intéressant de noter que cette Room fonctionne comme une trapDoor, puisque aucune sortie conventionnelle n'y existe : mais l'Override de getExit() dans TransporterRoom permet de trouver une sortie aléatoire peu importe la direction rentrée par le joueur. La commande back n'y fonctionne donc pas. En revanche, le warp fonctionne toujours, permettant au joueur qui le possède de s'échapper "avec certitude sur l'arrivée". Pas de seed utilisée dans aucun cas, puisque l'exercice suivant permet de s'en passer.

7.46.1) Ajout d'un attribut booléen aTestMode dans GameEngine toujours initialisé à false, qui ne devient vrai qu'en cas d'exécution de la commande de test. Cela permet d'ajouter une nouvelle commande uniquement utilisable en utilisant les fichiers de test : alea.

alea ne va pas constituer une méthode, mais utiliser une nouvelle propriété de RoomRandomizer. En effet, cette classe possède désormais un nouvel attribut entier aAleaInt initialisé à -1 qui peut être manipulé par un modificateur. Cet entier est désormais utilisé par findRandomRoom() pour déterminer de façon aléatoire ou non une sortie de sa TransporterRoom : si aAleaInt vaut -1, le fonctionnement est identique à précédemment. Si l'AleaInt est compris dans les limites d'index de l'ArrayList du RoomRandomizer, findRandomRoom() renverra toujours la Room indexée à cet endroit. Si l'AleaInt est hors des limites d'index de l'ArrayList, findRandomRoom() renverra toujours la dernière Room de l'ArrayList. Enfin, il ne reste plus qu'a ajouter alea à la liste des validCommands, et à décrire son fonctionnement dans

interpretCommands() de la manière suivante : si le commandWord tapé est alea ET que le jeu est bien en TestMode (booléen à true), le jeu vérifie d'abord que le joueur se situe bien dans une TransporterRoom (utilisation de instanceof). Ensuite, il vérifie si un secondWord a été tapé : si c'est le cas il cherche un entier dans la String secondWord et le converti en entier pour modifier l'attribut aAleaInt de RoomRandomizer, si un entier est trouvé dans la string. Un try catch est utilisé pour repérer le cas où il n'y a pas d'entier dans la String secondWord. Si il n'y a pas de SecondWord tapé, on réinitialise aAleaInt à -1 pour permettre un fonctionnement aléatoire du RoomRandomizer, et donc de la TransporterRoom.

7.46.2) Il aurait été intéressant de considérer l'item StarGate comme une classe à part entière héritant de Items compte tenu de son fonctionnement assez particulier. Cependant, comme un seul Item de ce type est prévu pour le jeu, ceci n'est pas nécessaire, puisque que GameEngine est parfaitement capable d'en gérer un seul dans le code déjà écrit. L'héritage aurait été utile si plusieurs exemplaires de l'item StarGate aurait été implentés dans le jeu. Il aurait été possible de penser à l'héritage pour les FuelTanks, mais cela aurait amené une profonde restrucuration de la mécanique de gestion de carburant au niveau du code.

Bonus) Ajout de la vérification des conditions de victoire via une procédure checkWinCondition() dans la classe GameEngine qui vérifie d'abord si le joueur a atteint la Terre. Si tel est le cas, cette procédure vérifie son inventaire pour vérifier si les items nécessaires à la victoire sont présents. Le message de victoire ou de défaite s'affichera selon le résultat du test.

Cette procédure sera appelée à chaque utilisation de go puisque c'est forcément par cette commande que la Terre est atteinte.

III) Mode d'emploi.

Pour lancer le jeu dans BlueJ, il faut clic-droit sur la classe Game et lancer new Game(), puis Ok.

La fenêtre du jeu apparaît. Entrez "help" pour connaître les commandes. Entrez "go" suivie d'une direction pour aller dans la direction souhaitée. Entrez "quit" pour quitter le jeu. Les boutons sont a dispositions du joueur en tant que raccourcis de certaines commandes.

IV) Déclaration anti-plagiat.

Le jeu est inspiré des rogue-like *FTL* et *Out There* (pour le style de jeu). L'univers est en partie inspiré du jeu *StarCraft* (sans réutiliser son contenu) et de *Spore*, les races aliens étant tirées de mes propres parties sur ce jeu (où on créé sa propre espèce en la faisant évoluer à travers les âges, du stade de simple cellule au stade d'empire galactique). Cependant, aucun élément des univers n'a été repris et le style de jeu est prévu pour être inspiré des deux rogue-like sans en être une recopie (les deux étant d'ailleurs eux-mêmes différents).

En ce qui concerne le code, à l'exception de la classe Parser (qui est fournie pendant le projet), d'une partie de la classe UserInterface (fournie aussi pendant le projet) et des méthodes explicitement fournies telles quelles dans les exercices, rien n'a été recopié sur un autre projet ou toute autre source.

Les images étant toutes tirées d'internet, certaines pouvant être soumises aux droits d'auteurs, un fichier texte contenant les références URL de chacune des images peut être trouvé dans le dossier images de l'archive. La plupart des images utilisées dans le jeu proviennent de la HubbleGallery de la NASA, qui regroupe différents clichés pris par le téléscope satellite Hubble, et de Wikimedia (qui elles sont libres de droits).