Jeu d'aventure projet Zull
(A3P(AL) 2023/2024 G5)

I. a. Auteur(s)
   b. Thème (phrase-thème validée)
   c. Résumé du scénario (complet)
   d. Plan (complet, avec indication de la partie "réduit" si exercice 7.3.3)
   e. Scénario détaillé (complet, avec indication de la partie "réduit" si exercice 7.3.3)
   f. Détail des lieux, items, personnages
   g. Situations gagnantes et perdantes
   h. Éventuellement des énigmes, mini-jeux, combats, etc.
   i. Commentaires (ce qui manque, reste à faire, ...)
II. Réponses aux exercices (à partir de l'exercice 7.5 inclus)
III. Mode d'emploi (si nécessaire, instructions d'installation ou pour démarrer le jeu)
IV. Déclaration obligatoire anti-plagiat (*)

(*) Cette déclaration est obligatoire :
- soit pour préciser toutes les parties de code que vous n'avez pas écrites vous-même et citez la source,
- soit pour indiquer que vous n'avez pas recopié la moindre ligne de code (sauf les fichiers zuul-*.jar qui sont fournis évidemment).

I. Première partie

a. Auteurs :

Gnamien Marie Emilienne

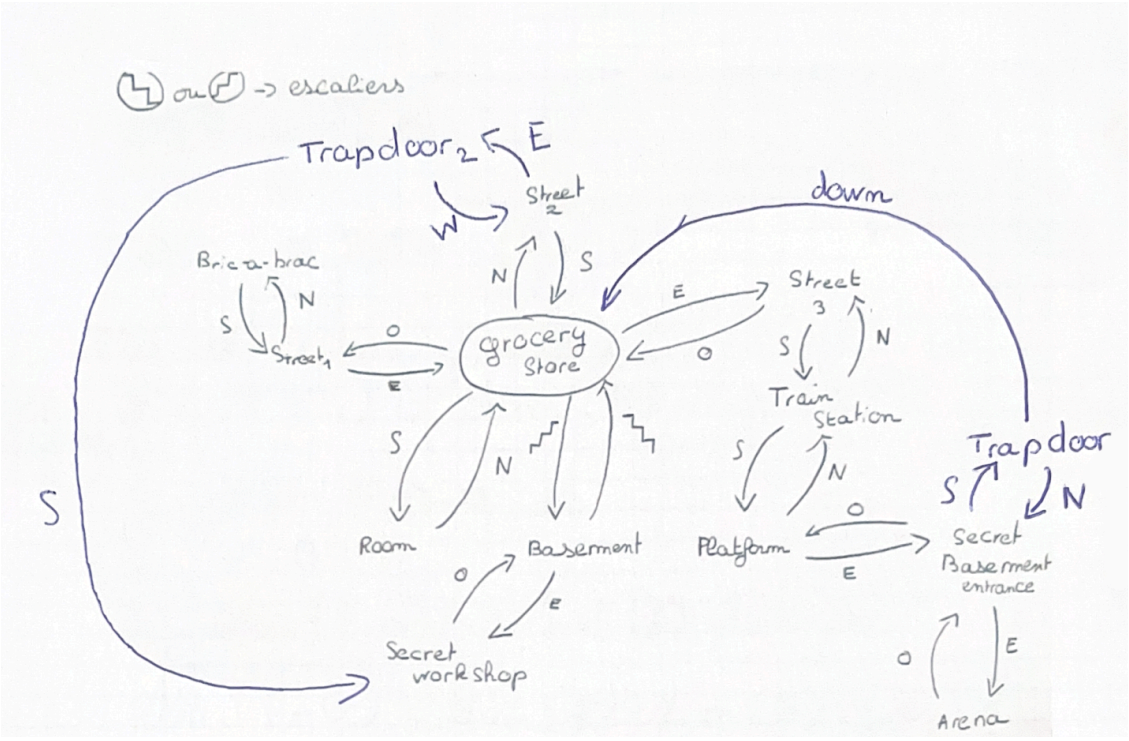
b. Thème :

En 3417, dans un univers cyberpunk : une jeune femme de 19 ans (moitié humaine/moitié androide) doit améliorer son système interne pour pouvoir participer à une compétition de combat.
Titre du jeu : 3417.

c. Résumé du scénario :

Le but de l'héroïne est donc de réussir à obtenir des composants nécessaires à l'amélioration de son système. Dans cet univers, il est très compliqué de se procurer des composants de qualitatifs. Mais il y a une autre façon de les obtenir, et cette méthode est illégale. Une fois que cela sera acquis elle pourra donc participer à une compétition de combat très connue des banlieues, malgré le fait qu'elle soit interdite par les autorités.

d. Plan du jeu :



f. Détail des lieux, items, personnages :

➔ Lieux :
- Chambre de l'héroïne
- Sous-sol de la maison de son Grand-Père
- Atelier secret du Grand-Père
- épicerie tenue par le Grand-Père
- 3 différentes rues (dont une avec un cul-de-sac)
- Gare de train/métro
- Quai
- Sous-sol secret (compétition combats)
- Brocante

➔ personnages :
- Grand-Père de l'héroïne (lui cache des secrets, l'aide à trouver des composants plus tard dans le jeu, l'avertit mais reste quand même inquiet)
- 3 amis à l'héroïne (l'un d'entre eux est un traître)
- les autorités
- passeur (pour pouvoir participer à la compétition)

➔ items :
- clés pour pouvoir entrer dans certaines pièces verrouillées
- composants utiles à l'amélioration du système de l'héroïne
-

g. Situations gagnantes ou perdantes :

➔ Situations gagnantes :
- trouve le beamer (pour la version finale)
- Réussit à se procurer les composants sans se faire remarquer (peut renoncer à son combat).
- Réussit à améliorer son système, gagne son combat et réussit à échapper aux autorités lorsqu'elles découvrent le lieu de compétition secret (si elle n'a pas démasqué son ami traître).
- démasque son ami.e traître.

➔ Situations perdantes :
- Ne réussit pas à trouver le balai. (pour la version finale)
- Se fait remarquer en se procurant les composants nécessaires à l'amélioration de son système
- Ne réussit pas à s'échapper et se fait arrêter par les autorités.
- Ne réussit pas à se procurer les composants nécessaires.

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

- réfléchir aux autres situations dans lesquelles il serait possible de perdre.
- ajouter des items en plus.
- rechercher des idées de mini-jeux et énigmes pour le jeu.
- compléter le jeu avec toutes les autres situations possibles.

II. exercices :

Exercice 7.5 : dans la méthode goRoom de la classe Room
à chaque condition : suppression de l'affichage de la Current Room.
Puis suppression du code qui nous permet d'accéder aux exits.

Exercice 7.6 :
1. Il n'est pas nécessaire de vérifier si chaque paramètre est null. Il suffit simplement de vérifier la valeur de "direction".
2. Pour ma première classe Game, j'ai donc créé mon accesseur "getExits" me permettant d'accéder à mes attributs privés Exits.
J'utilise donc cet accesseur dans ma classe "Game" pour pouvoir attribuer à vNextRoom la pièce dans laquelle l'utilisateur veut se rendre. Grâce à cette ligne, je n'ai plus besoin de vérifier la valeur de vDirection à chaque fois quelque soit la valeur de vNextRoom (dont "null").
J'ai donc supprimé tous les tests vérifiant que vNextRoom est null en les remplaçant par deux tests.

Exercice 7.7 :
J'ai modifié ma méthode printLocationInfo en utilisant mon accesseur getExits sur mes comparaisons.
Pour définir "getExitString()", on retourne juste la liste des différentes sorties de notre pièce "Room".
Il est logique de demander à Room de produire les informations sur ses sorties puisqu'elle a accès aux attributs privés Exits contrairement à Game. Avant de définir "getExitString" il fallait utiliser un accesseur, en plus d'un test pour pouvoir afficher la liste de sorties. Mais maintenant ce n'est plus nécessaire.

Exercice 7.8 :
La méthode qui est désormais inutile est setExits puisque l'on peut maintenant définir les sorties de chacune des pièces une par une en utilisant la méthode setExit. Grâce à cette méthode, il est alors possible de définir d'autres directions dont haut et bas.
Pour modifier la méthode "getExitString", j'ai utilisé la liste de clés existantes dans mon tableau associatif. Chaque clé correspond à la sortie disponible, j'ajoute cette sortie à ma chaîne de caractères vExits. À la fin de cette boucle, vExits affichera donc toutes les sorties disponibles.

Exercice 7.8.1 :
Un escalier a été ajouté de l'épicerie vers le sous-sol.

Exercice 7.9 :

(voir exo 7.8)

Exercice 7.10 :
Voici comment fonctionne la méthode "getExitString" :
Premièrement, une variable vExits contenant une chaine de charactères est déclarée.
Ensuite, on déclare un ensemble Set (le type des clés est String) contenant l'ensemble des sorties (clés) de notre HashMap exits.
Nous avons ensuite une boucle parcourant toutes les clés de ce tableau associatif.
Chaque clé est donc ajoutée à notre variable vExits déclarée auparavant.
Finalement, vExits est retournée.

Exercice 7.10.2 :
La javadoc de la classe Game est presque vide puisque la majorité des méthodes utilisées sont privées.

Exercice 7.11 :
Après avoir ajouté la nouvelle fonction getLongDescription(), il m'a été nécessaire de faire certaines modifications dans ma classe game. Dans cette classe, j'ai choisi de modifier la méthode printLocationInfo() en remplaçant toutes les lignes de codes qui affichent la pièce courante et ses sorties, par
System.out.println(aCurrentRoom.getLongDescription());

Exercice 7.14 :
Après avoir complété la classe Game avec la méthode look(), on modifie donc processCommand, en ajoutant un nouveau test. Si la condition est respectée, look() est appelé. On retourne ensuite le booléen false qui nous permettra de continuer à faire tourner le jeu.

Exercice 7.15 :
Pour ajouter une nouvelle commande, on ajoute le nom de cette commande dans notre classe CommandWords. Puis, on crée la méthode liée à cette commande, ici nous avons eat(). Finalement, nous modifions notre fonction processCommand(), pour que la commande eat() puisse fonctionner correctement.

Exercice 7.16 :
Modification de printHelp() (aParser.showCommands()) dans la classe Game.
Création de la méthode showAll() dans la classe CommandWords.
Création de la méthode showCommands() dans la classe Parser.

Exercice 7.18 :
Ajout de la méthode GetCommandList() dans la classe CommandWords qui retourne la liste entière des commandes disponibles.

Ajout de la commande GetCommandString() dans la classe Parser qui retourne la chaîne de caractères contenant toutes les commandes disponibles..
Modification de la méthode printHelp() de la classe Game, en ajoutant l'affichage des commandes disponibles.

Exercice 7.18.6 :
On utilise plus de scanner dans le Parser puisque les commandes tapées aux claviers sont prises en compte par l'attribut aEntryField de la classe UserInterface.

Exercice 7.18.6 :
Après avoir effectué tous les changements nécessaires dans nos classes, on modifie la méthode printLocationInfo() pour que le lieu actuel et son image s'affiche dans la fenêtre.

Exercice 7.18.8 :
Il a été nécessaire d'utiliser la classe Jbutton.
Ensuite, il a fallu ajouter le bouton à l'interface de l'utilisateur. Pour cela on ajoute les lignes de code suivantes :

```
this.aButton = new JButton("start!");
aMyFrame.add(this.aButton,BorderLayout.EAST);
```

Ici j'ai ajouté un bouton qui "démarre" le jeu en quelque sorte. Lorsqu'on clique sur ce bouton, il disparaît et les informations sur le lieu actuel sont affichées.

Exercice 7.20 :
On crée une classe nommée Item, constituée d'un constructeur naturel et tous les accesseurs nécessaires (description, poids). Cette classe comporte aussi une méthode getItemString qui retourne l'Item présent (ou non) de la pièce courante.
Dans notre classe Room, on crée un attribut Item altem, puis on définit notre méthode setItem qui nous permet d'assigner un Item spécifique à notre pièce.
On modifie getLongDescription(), qui affichera désormais l'item de la pièce courante s'il existe.

Exercice 7.21 :
Ici la classe Item produit toutes les informations concernant un Item.
La classe Room se charge de produire la String.
Finalement la classe GameEngine se charge de l'afficher.

Exercice 7.22 :

J'ai donc créé une HashMap me permettant d'assigner plusieurs Items à une seule et même pièce. J'ai aussi dû modifier ma méthode getItemString() qui me renvoie maintenant tous les Items de la pièce courante.

Exercice 7.24 :
Si un utilisateur tape un second mot après back, l'utilisateur ne change pas de pièce et ce message est affiché "You can't do that...". Si l'utilisateur n'a pas encore bougé et qu'il essaie d'utiliser la commande back, l'utilisateur ne change toujours pas de pièce et ce message est affiché "You haven't moved yet...".

Exercice 7.25 :
Lorsque l'utilisateur se déplace 3 fois successivement, lorsqu'il entre back une première fois il retourne dans la pièce précédente. Mais après avoir appelé back deux fois supplémentaires, l'utilisateur ne change pas de pièce. Cela est satisfaisant puisque la commande respecte ce qu'il fallait faire avec la commande back.

Exercice 7.26 :
Si l'on utilise notre nouvelle commande back qui utilise cette fois une Stack, l'utilisateur revient bien au départ du jeu. Seulement, il ne peut plus faire de back une fois arrivé au point de départ, puisque notre pile (Stack) est vide.

Exercice 7.27 :
Premièrement, il faut vérifier que notre jeu démarre correctement.
Mais ce test là n'est pas suffisant pour être sûr du bon fonctionnement de notre jeu.
Il faut vérifier que les méthodes utilisées dans nos différentes classes fonctionnent correctement sans aucune régression au niveau de notre jeu.
Il faudrait aussi s'assurer du bon fonctionnement de toutes les commandes qui peuvent être utilisées par l'utilisateur.

Exercice 7.28 :
Il serait peut-être possible d'utiliser un fichier de commandes contenant l'ensemble des commandes qui pourraient être entrées par l'utilisateur pour pouvoir tester le jeu et son bon fonctionnement.
Il faudrait changer la classe GameEngine.

Exercice 7.28.1 :
J'ai donc créé ma nouvelle commande test en plus d'une autre méthode nommée lecture qui nous permet de lire les fichiers nécessaires aux tests.
Exercice 7.28.2 :
J'ai donc créé mes trois fichiers de commandes nécessaires puis je les ai ajoutés à mon jeu.
Exercice 7.29 :

Pour la classe Player, j'y ai ajouté ce que je pense être essentiel à conserver. Il y a donc 5 attributs différents, un pour le nom du joueur et un autre pour conserver la pièce courante du joueur. Il y a ensuite un attribut qui conserve le poids maximal d'Items que peut porter un joueur ainsi qu'un attribut Stack(Room) et une HashMap pour notre futur inventaire.

J'ai modifié notre méthode back() qui n'est désormais plus une procédure mais une fonction dans player. Cette méthode retourne un booléen en fonction de s'il est possible ou non de retourner dans la pièce précédente ou non.
J'ai ajouté un assesseur nommé getCurrentRoom(), nous permettant de récupérer la pièce actuelle du joueur.
Finalement, j'ai créé un modificateur nommé changeRoom, qui permet au joueur de changer de pièce.
De plus, ma classe Player possède aussi une méthode addStackRoom() nous permettant de gérer notre attribut Stack(Room) en y ajoutant le lieu actuel "en haut de la pile".

Exercice 7.30 :
J'ai créé mes deux méthodes take et drop en m'appuyant sur les méthodes addItem et removeItem de ma classe. Si l'on prend un Item, il est donc retiré de la hashmap des Items disponibles dans la pièce. Mais si l'on dépose un item, cet item est donc ajouté à la hashmap d'items de la pièce.

Exercice 7.31 :
J'ai donc ajouté une Hashmap pour pouvoir stocker les items. Cela a donc impliqué que je modifie mes commandes take et drop.

Exercice 7.31.1 :
Dans la classe ItemList, j'ai réuni toutes les méthodes gérant les Hashmap de mes classes player et room. Puis je les ai modifiées de telle sorte qu'elles se comportent d'une façon assez "générale". De cette façon, il n'est pas nécessaire de gérer deux hashmap différentes dans ces classes là.

Exercice 7.32 :
Dans ma classe Player, j'avais déjà déclaré un attribut aMaxWeight auparavant.
J'ai donc modifié ma méthode take en utilisant une fonction verifyWeight qui nous permet de vérifier que le poids total des Items de l'inventaire du joueur est bien inférieur au poids maximal avant d'ajouter un nouvel item à l'inventaire.

Exercice 7.33 :
Pour cette méthode, j'ai utilisé le getItemString de la classe Itemlist pour afficher tous les items présents dans l'inventaire. Pour le poids, on utilise l'attribut aWeight (poids de

l'inventaire du joueur) qui est mis à jour à chaque fois que l'on prend/dépose un item.Après tout cela, le texte demandé est bien affiché.

Exercice 7.34 :
Pour cet exercice, j'ai créé mon nouvel item magicCookie.
Puis j'ai modifié ma commande eat de telle sorte que lorsque le joueur l'utilise, le nom de l'item soit pris en compte. Ensuite j'ai créé une nouvelle méthode dans ma classe player me permettant donc de doubler le poids maximal de notre inventaire. Après avoir effectué cela, l'item est retiré de l'inventaire du joueur.

Exercice 7.34.1 :
Pour cet exercice, j'ai ajouté au troisième fichier "all" toutes les nouvelles commandes et fonctionnalités ajoutées jusqu'ici à tester.

Exercice 7.42 :
Pour cet exercice, le joueur doit cliquer sur le bouton quest de la fenêtre pour pouvoir lancer la quête. J'ai ajouté de nouvelles fonctions telles que quest() qui nous permet de compter le nombre de pas effectués par le joueur. La quête principale est de pouvoir récupérer le balai qui se trouve au sous-sol en moins de 3 déplacements. Après ces 3 déplacements, si le joueur ne possède pas le balai dans son inventaire, le jeu se termine. Pour cela j'ai donc modifié ma fonction interpretCommand() dans le cas où le joueur veut se déplacer. J'ai utilisé un accesseur qui me permet de savoir si la quête a été lancée par le joueur. Si c'est bien le cas, le joueur se déplace mais les pas du joueur sont comptés grâce à la fonction quest(). à chaque déplacement on vérifie alors si le nombre de pas est égal à 3 et si le joueur possède le balai dans son inventaire. Si la condition est respectée, un message s'affiche : "You must find the broom... Try again.", et le jeu s'arrête.

Exercice 7.42.2 :
Je prends la décision de garder mon IHM graphique actuelle.

Exercice 7.43 :
Pour les deux trapdoors que j'ai créées, il suffit simplement de créer deux pièces différentes dans notre jeu et de leur attribuer deux sorties différentes. Lorsque l'on prend une des sorties et que l'on essaie de repartir dans notre trapdoor, il est affiché un message "You can't go back", puisque notre trapdoor ne peut être utilisée dans le sens inverse pour l'une de ses sorties.

Exercice 7.44 :
Pour le beamer j'ai créé une nouvelle classe héritant de la classe Item puisque c'est un Item mais qui possède ses propres fonctions. Ensuite j'ai donc placé mon Item Beamer dans mon jeu. Pour pouvoir l'utiliser il faut le prendre dans une pièce, le charger dans une

autre pièce et finalement se déplacer à nouveau pour pouvoir l'utiliser. Une fois le beamer utilisé, il faut le recharger pour pouvoir le réutiliser.

Exercice 7.45.1 : Les fichiers de commande ont été mis à jour.

IV. Déclaration obligatoire anti-plagiat (*)

Aucune ligne de code n'a été copiée/collée. J'ai seulement utilisé les lignes de codes données par Mr. D. BUREAU.