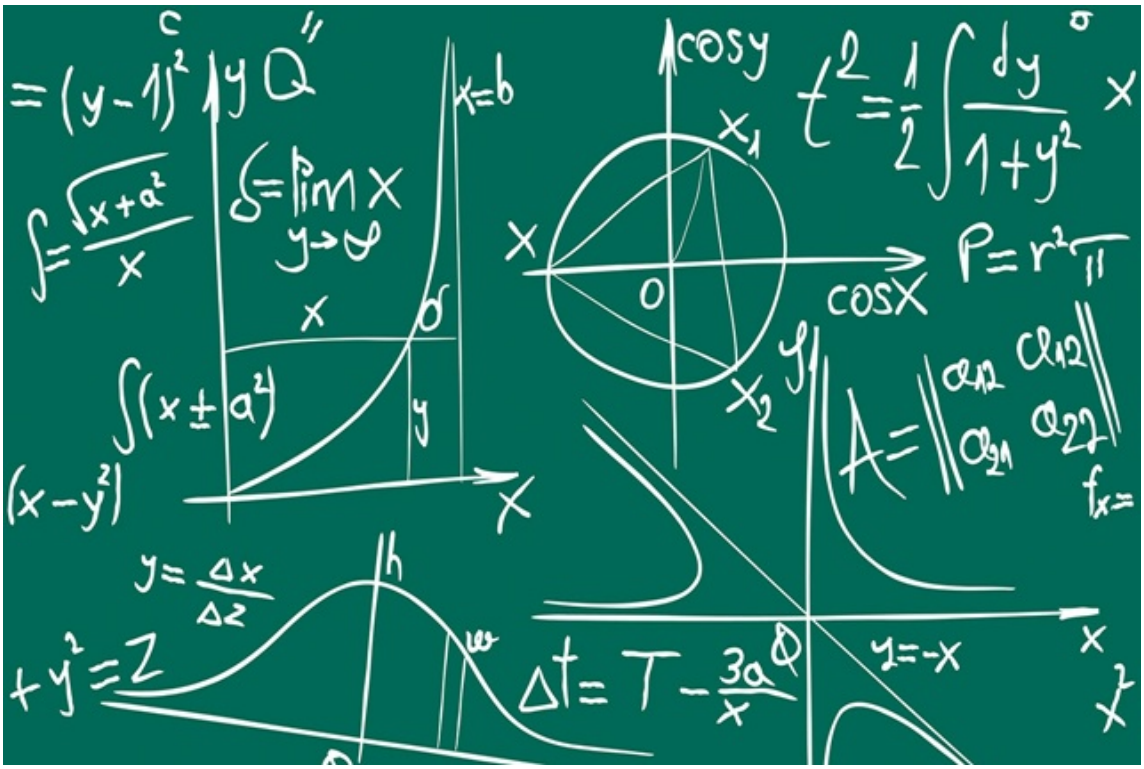
Rapport : Projet Zuul

Définir f.



Emilie CHEN

Groupe 5s

Table des matières

I) Présentation du projet

I.A) Auteur

Je suis Emilie CHEN du Groupe 5s. Ravie de vous présenter mon projet zuul, un jeu d'venture sur le thème des mathématiques.

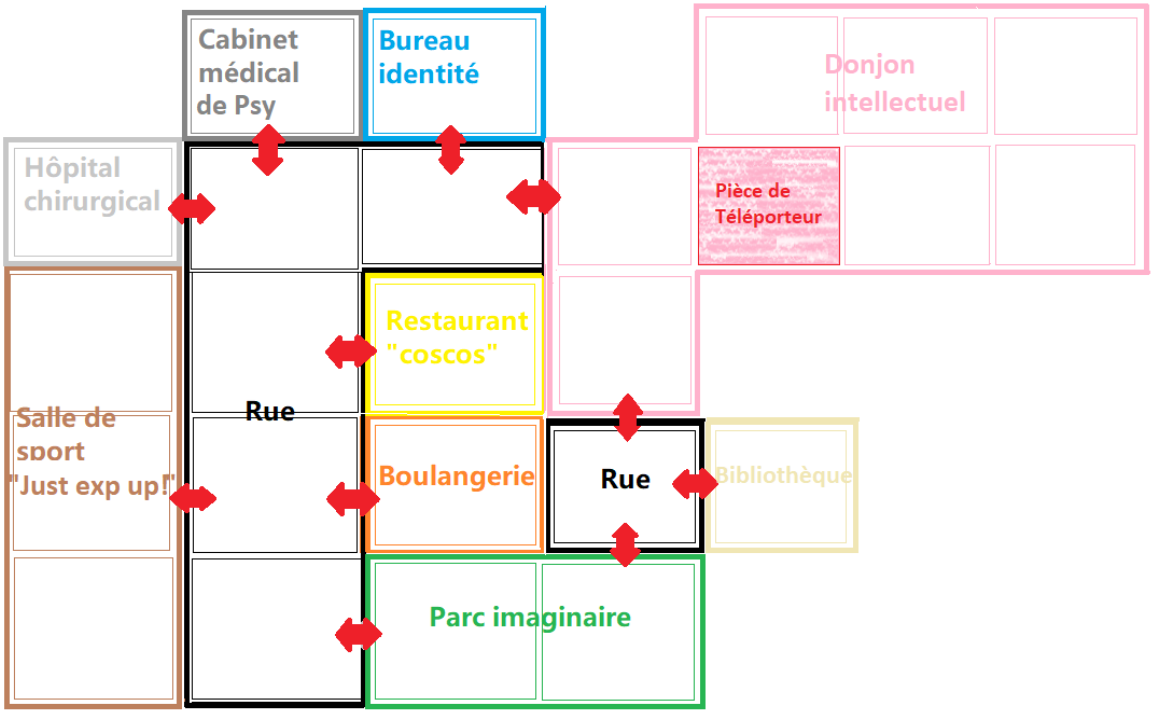
I.B) Thème

Dans le monde des mathématiques, une fonction doit retrouver sa carte d'identité.

I.C) Résumé

Dans le monde des mathématiques, une fonction perd sa carte d'identité, si et seulement s'il a perdu sa mémoire peu après.
Vous incarnez une fonction qui vient de perdre sa mémoire. Vous devez donc retrouver votre carte d'identité pour vous définir.

I.D) Plan



I.E) Scénario complet

Dans le monde des mathématiques, une fonction perd sa carte d'identité, si et seulement s'il a perdu sa mémoire peu après.

Le jeu commence lorsque vous vous êtes rendu au bureau identité pour avoir de l'aide après avoir perdu votre mémoire. Serait-elle dans le parc, la boulangerie, la salle de sport, dans un coin de la rue ou trouvée et déposée au bureau administratif par une fonction ?

Vous n'avez qu'une calculatrice sur vous, prouvant que vous êtes bel et bien une fonction. Mais vous n'avez pas de carte de stockage, votre capacité actuelle ne vous permet pas de ramasser la majeure partie des items.

Tout d'abord, vous devez vous rendre au donjon intellectuel afin de remporter cette fameuse carte de stockage. Une fois insérée dans la calculatrice, vous devez partir à la quête d'indice. Explorez les alentours pour trouver des items et des personnages pouvant vous renseigner. Les indices qu'elles donnent dépendent parfois du nom de leur métier.

Pour une fonction, perdre sa carte d'identité témoigne d'une certaine stupidité. Il sera exigé que vous limitez vos pas et que vous évitez de faire des erreurs.

I.F) Détail des lieux, items, personnages

lieux :

- Parc imaginaire (2 pièces)
- Restaurant « coscos »
- Bureau identié
- Salle de sport « Just exp it » (3 pièces)
- Cabinet médical
- Hôpital Chirurgical
- Donjon intellectuel (2 pièces)
- Boulangerie
- Rue (6 pièces)

personnages :

- le coach Exponentielle
- le boulanger fonction Carré
- la voyante Échelon
- le maître du donjon Valeur absolue
- le chef cuisinier Cosinus
- l'employé Identité
- le chirurgien fonction Nulle
- la fonction inconnu perdue
- la grand-mère Logarithme népérien

items :

- une carte de dérivation
- une carte de coatching de salle de sport
- une pièce de 1i (équivalent 1euro)
- une calculatrice
- une pièce d'identité
- une carte de stockage de 1Go
- une paire de lunettes
- un papier recensant la liste des pièces d'identité retrouvées
- un beamer

I.G) Situations gagnante et perdantes :

Situation gagnante :

- Vous avez retrouvé votre carte d'identité dans les temps.

Situations perdantes :

- Vous avez mal répondu à trois reprises au donjon intellectuel.
- Vous n'avez pas réussi pas valider un numéro au bout de 50 déplacements.
- Vous n'avez pas validé le bon numéro.
- Vous avez été trop naïf et avez donné 1i à la « voyante »

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

Une petite énigme est proposée au donjon intellectuel pour le moment.
Il faut deviner tout au long du jeu car les indices se cachent dans la plupart des noms de lieux. Elles donnent une idée des indices pouvant être trouvées.
Les items voulus par les personnages ne sont pas toujours exposés clairement. Il faut faire le lien avec leur métier.

I.I) Commentaires

Le jeu est complet.

II) Réponses aux exercices

7.5)

Les méthodes *goRoom* et *printWelcome* de la classe *Game* ont un segment de code en commun. En effet, ils affichent tout deux les informations concernant la salle courante soient les sorties disponibles. Pour éviter la duplication du code et garantir une bonne cohésion, il convient de créer une méthode *printLocationInfo* dont l'unique tâche est de renvoyer ces informations. Il suffira ensuite aux méthodes *goRoom* et *printWelcome* de l'appeler si nécessaire.

7.6)

Le couplage entre la classe *Game* et la classe *Room* est trop forte. Si nous souhaitons ajouter une nouvelle direction (objet *Room*), nous devrons aussi faire des modifications dans la classe *Game*. Ce qui peut s'avérer très fastidieux donc il est important de respecter le principe d'encapsulation. La classe *Game* ne doit pas utiliser les attributs de la classe *Room* pour s 'exécuter correctement.

Pour réduire le couplage, nous ajoutons l'accesseur *getExit* dans la classe *Room* qui renvoie les sorties en fonction de la direction.

Pour implémenter le nouveau lieu, il suffira d'écrire :

```
vNextRoom = this.aCurrentRoom.getExit(vDirection);
```

7.7)

De la même manière, nous ajoutons un accesseur *getExitString* dans la classe *Room* qui affichent les sorties disponibles. Il semble plus naturel de demander à la classe *Room* de produire ses informations, car le code fait intervenir ses attributs. *Game* gère essentiellement les affichages. Il affichera les sorties avec la méthode *printLocationInfo* qui, elle, appellera *getExitString* avec :

```
this.aCurrentRoom.getExitString();
```

7.8)

Pour faciliter l'ajout des directions, on enlève les attributs *aDirectionExit* et on ajoute dans les déclarations d'attributs.

```
private HashMap <String, Room> aExits;
```

On ajoute aussi dans le constructeur naturel:

```
aExits = new HashMap <String, Room>();
```

Ainsi, nous n'avons pas besoin de changer les implémentations de tout les objets *Room* de la méthode *createRoom* pour ajouter une direction.

7.9)

La façon de stocker les sorties a été modifiée donc il faut modifier les méthodes *getExit* et *getExitString :*

On a pour *getExit* :

```
{ return this.aExits.get(pdirection);}
```

On a pour *getExitString* :

```
if (this.aExits.get("Direction")!= null)
    {System.out.println("Direction");}
```

7.10)

On peut simplifier la méthode *getExitString* qui sert à indiquer les sorties (directions) disponibles. Les sorties sont stockés dans un objet *HashMap*, on peut prendre comme motif *String*

```
Set <String> keys = aExits.keySet() ;
```

parcourir les sorties de la salle et ajouter les directions disponibles dans un objet *String* prédéfini ;

```
for (String vExit : keys) {
    returnString += ‘ ‘ + vExit ;}
```

puis renvoyer le *String* à la fin.

7.10.1) 7.10.2)

Normalement, la Javadoc a été complété et généré. Lorsque l’on visualise la Javadoc, on remarque que la classe *Game* contient beaucoup moins de méthodes que celle de la classe *Room* (2 méthodes pour *Game* et 4 pour *Room*), or 8 méthodes ont été implémentées dans *Game* et 4 dans *Room.*

En fait, la Javadoc ne montre que les méthodes du champ public. Elle montre qui est utilisable à l'extérieur de la classe donc les services qu’elle peut rendre. *Game* utilise les services rendus par les autres classes, mais ne rend pas beaucoup services, Ils rassemblent les informations pour afficher et gérer le fonctionnent du jeu.

7.11)

Actuellement, *printLocationInfo* affiche une description simple de la salle

```
"Vous êtes dans"+ this.aCurrentRoom.getDescription()
```

Si l’on ajoute des pnj, des objets ou autres, cette description sera modifiée. Le code de *printLocationInfo* le devra aussi. Le couplage entre les classes *Room* et *Game* peut encore être diminué pour corriger ce problème.

On crée un accesseur *getLongDescription* dans la classe *Room*. Il contient des informations descriptive de la pièce (lieu, objets, pnj,…). Ainsi, on n'a plus besoin de modifier *printLocationInfo* à cause d'une modification de la description d'une salle.

Pour que *printLocationInfo* affiche la description détaillée, il suffit d'écrire :

```
System.out.println(this.aCurrentRoom.getLongDescription()) ;
```

7.12) 7.13)

Ces questions sont optionnelles et n'ont pas été traitées.

7.14)

La commande *regarder* a été ajouté. Pour cela, on l'implémente dans la classe *CommandWords*, on ajoute un *else if* dans la méthode *processCommand* de la classe *Game* pour s'occuper de la commande *regarder. Si la* commande *regarder* est tapée, cette méthode appellera une procédure *look* sans paramètre qui affiche la description détaillée de la pièce courante via *getLongDescription*.

7.14.1)

La procèdure *look* prend maintenant une commande en paramètre. Si un second mot est détecté. La procédure affichera un message :

```
if (pC.hasSecondWord)

System.out.println ("Je ne sais pas regarder quelque chose en particulier") ;
```

7.15)

De même, la commande *manger* et la procédure eat ont été ajoutées. Pour le moment, la commande *manger* appellera la procédure *eat* qui affichera seulement :

```
"Vous avez mangé assez et nous n'avez plus faim. "
```

7.16)

L'ajout des nouvelles commandes nécessite la modifications de la méthode *printHelp* de la classe *Game.* En effet, *printHelp* renvoyait : "

```
Vous êtes perdu, vous êtes tout seul, tes commandes sont : aller quitter aide.
```

Il n'y a pas de problème de compilation apparent, mais des modifications sont effectivement nécessaires. Ce qui n'est pas aisé à identifier. On a affaire à un couplage implicite. Le code de *printHelp* dépend implicitement des commandes du jeu. Ajoutons une

méthode *showAll* dans la classe *CommandWords* qui affichera les commandes possibles. Cette classe gère les commandes donc ce choix paraît naturel.

Dans la procédure *showAll*, une boucle *for* est utilisée pour stocker les commandes possibles.

```
for (String command: aValidCommands){
    System.out.println (command + " ");
```

En fait, aucun lien n'est établi entre *Game* et *CommandWords*. Mais la classe Parser, liée aux deux, peut servir d'intermédiaire. Ainsi, le couplage faible est mieux assuré. On ajoute alors l'accesseur *showCommands* à *Parser* qui appellera la procédure *showAll*. *printHelp appellera* alors *showCommands* si besoin.

7.17)

Si on ajoute une nouvelle commande, il nous modifier la méthode processCommand de la classe Game afin qu'elle puisse l'interpréter. En revanche, la méthode showAll affiche tout les commandes valides de la classe CommandWords et n'a donc pas besoin de correction.

7.18)

Pour faciliter le passage d'affichage textuel par un terminal à l'affichage par intermédiaire d'une interface graphique à l'avenir, on modifie la méthode *showCommands* de la classe *CommandWord* de sorte qu'elle ne sert qu'à la production de la chaîne de caractères des commandes disponibles. On le renomme donc comme un accesseur soit *getCommandList*. La méthode *showCommand* se nommera alors *getCommandString* et appellera *getCommandList*.

7.18.1)

La comparaison avec le projet zuul-better a permis de corriger certaines erreurs (oubli, type, confusion…).

7.18.3)

Des images provisoires ont été enregistrées dans un dossier prévu à cet effet.

7.18.4)

Le titre du jeu est : Définir f. En effet, le but du jeu est de trouver l'expression de la fonction incarnée par le joueur.

7.18.5)

Dans la classe *Game*, on a fait

```
import java.util.ArrayList ;
private ArrayList<Room> aRoomList ;
```

La méthode create*Room*s stockera les *Room* dans une liste après l'avoir déclaré dans le constructeur.

Par exemple vRue2 :

vList.add (vRue2);

7.18.6)

Game : Cette classe va dorénavant seulement contenir des attributs et un constructeur.

Parser :Cette classe lit les commandes du joueur pour les interpréter.

La nouvelle version de Parser a été incorporée. Celle ci n'a plus besoin de Scanner. En effet, la méthode getCommand de Parser décodera la commande en le prenant comme un paramètre *String*.

Room : L'attribut *String aImageName* ainsi que l'accesseur *String* getImageName() ont été ajoutés à la classe *Room*. La méthode create*Room*s a donc aussi été modifiée.

```
String returnString = "Sorties :";
    Set <String> keys = exits.keySet();
    for (String exit : keys) {
       returnString += " " + exit;
    }
    return returnString;
```

La méthode de getExitString a été modifiée :

```
StringBuilder returnString = new StringBuilder( "Exits:" );
    for ( String vS : exits.keySet() )
       returnString.append( " " + vS );
    return returnString.toString();
```

GameEngine : Cette classe reprend la quasi-totalité des méthodes de *Game*. L'attribut *aGui* de type *UserInterface* a été ajouté.

La méthode play a été enlevé au profit de l'interface graphique. Une méthode end*Game* qui affiche un message et quitte l'interface a été ajoutée ; ainsi qu'une procédure *setGUI*. La méthode *processCommand* servait à lire et à interpréter les commandes.

Dorénavant, la méthode *processCommand* sera dans la classe *UserInterface* et ne fera que lire les commandes. Une procédure *interpretCommand* de *GameEngine* qui prend la commande sous forme de *String* en fera l'interprétation.

La méthode *printLocationInfo* appelle de plus la méthode *showImage* pour afficher l'image :

```
this.aGui.showImage(this.aCurrentRoom.getImageName());
```

UserInterface : Cette classe permet d'implémenter une interface graphique permettant d'afficher une image et les entrées/sorties des textes.

Les lignes import... ont été remplacés par tous les imports de classes nécessaires.

Tous les System.out.println ont été remplacés par this.aGui.println().

7.18.7)

optionnel

7.18.8)

1. La classe servant à créer un bouton est JButoon :

```
import javax.swing.JButton;
```

On ajoute un attribut à *UserInterface* :

```
private JButton aButton;
```

2. Le texte choisi est la commande regarder

```
this.aButton = new JButton("regarder");
```

3. Le bouton sera positionné à gauche du panel.

```
vPanel.add (this.aButton, BorderLayout.WEST);
```

4. Pour savoir quand on a cliqué sur le bouton, on ajoute un écouteur d'action :

this.aButton.addActionListener(this);

5. Si un composant bénéficie d'un écouteur d'action la méthode *actionPerformed* est appelé. Si la source est un bouton, il lancera la commande regarder.

```
if (pE.getSource() == aButton){
    this.aEngine.interpretCommand("regarder");
```

7.19)

optionnel

7.19.1)

optionnel

7.19.2)

J'ai déplacé tous les images dans un dossier dédié.

Dans la méthode showImage de *UserInterface* :

```
String vImagePath = "Images/" + pImageName;
```

pour préciser le nom du répertoire.

7.20)

Pour un programme avec une bonne cohésion, une classe Item a été créée. Celle-ci comporte deux attributs *aDescriptionItem* et *aWeightItem*, ainsi qu'un constructeur naturel à deux paramètres. Un attribut *Item aItem* est ajouté à *Room*. Les items seront crées dans create*Room*s, cette méthode appellera une méthode *setItem* de *Room*, préalablement créée, servant à associer un item à un *Room* dont la signature est

```
public void setItem( final Item pItem)
```

Enfin, deux accesseurs *getDescriptionItem* et *getWeightItem* ont été créés dans la classe Item. La méthode *getLongDescription* est complétée pour appeler *getItemString* et ainsi afficher la description de l'Item s'il y en a un dans le *Room*.

7.21)

Les informations concernant un Item sont produites dans la classe Item. Le *String* décrivant l'Item est produit par la classe *Room*. Son affichage est géré par la classe *GameEngine*.

7.21.1)

À ce stade là, un Room n'a qu'un seul item. On pourrait simplement créer un accesseur getItem qui renvoie l'attribut aItem de la Room. Si un second mot est détecté pour la commande regarder, la méthode look vérifiera qu'il y a bien un item dans la pièce avec :

```
vItemRoom = this.aCurrentRoom.getItem() ;

if (vItemRoom == null) this.aGui.println ("Il n'y a pas d'item dans la pièce'')
```

puis vérifiera si le joueur a tapé le nom de l'item qui est dans la pièce avec :

```
if (vItemRoom.getNameItem() == pC.hasSecondWord())
```

Si c'est le cas, il affichera la description de l'item avec :

```
this.aGui.println (vItemRoom.getDescriptionItem()) ;
```

Sinon il affichera que “cet item n’est pas dans la pièce’’.

La méthode *look* a été modifié une nouvelle fois un peu plus tard (après l’exercice 7.46) pour :

- afficher la présentation de personnage s’il y en a.

- respecter le fait qu’il n’y a plus forcément un Item dans la pièce.

- S’occuper du cas où cet item est en possession du joueur. La description sera aussi affichée dans ce cas.

Pour cela, on testera si *vItemPlayer* et/ou *vItemRoom* sont null.

```
String vS = pC.getSecondWord();

Item vItemPlayer = this.aPlayer.getItemPlayer(vS);

Item vItemRoom = this.aPlayer.getCurrentRoom().getItem(vS);
```

7.22)

On ajoute un *HashMap* en attribut de *Room* qu’on nommera *aItems* et on le déclare dans le constructeur. On crée une procédure *addItem* dans la classe *Room* prenant un Item en paramètre que la méthode *createRooms* appelera pour affecter les Items aux *Room*. On ajoute aussi une méthode *String getItemString* dans *Room* semblable à *getExitString* qui renvoie la chaîne de caractère des items d’une *Room*. La méthode *getLongDescription* est modifiée pour appeler *getItemString* au lieu de *this.aItem.getDescriptionItem.*

7.23)

Pour ajouter une commande *retour*. On ajoute l’attribut *Room aLastRoom* à *GameEngine*. Celui-ci mémorisera la pièce précédente dans *goRoom*. Une procédure *back* est ajoutée. Cette procédure analysera si un second mot est tapé et s’il n’y a pas de pièce précédente, dans ces cas là la commande n’est pas valide. Si la commande est validée, back inversera la salle courante et la salle précédente puis appelera *printLocationInfo*. Ainsi, le joueur retournera à la pièce précédente. Bien évidemment, la commande retour est ajoutée parmi les commandes valides et son traitement est effectué par *interpretCommand*.

7.24)

Lorsque un second mot est tapé après retour, la commande fonctionne encore. Lorsque *retour* est tapé au lieu de départ, cela ne fonctionne pas. Le système ne répond plus à moins qu'une deuxième commande *retour* soit tapée pour revenir au jeu. Il faut que la procédure *back* fasse des vérifications pour éviter ce type de problèmes. Ces corrections ont été effectuées.

7.25)

Lorsque l'on tape *retour* deux fois de suite, on ne change pas de lieu. Cela s'explique par une double transposition. Si l'on effectue trois déplacements et on tape trois fois *retour*, on se retrouve au lieu précédent et non pas au lieu avant les trois déplacements. La commande *retour* telle quelle n'est pas satisfaisante. Il nous faut le modifier, c'est l'objet de la question suivante.

7.26)

La fonctionnalité de *retour* n'est pas satisfaisante. Une solution est de représenter *aLastRooms* comme une pile et l'ajouter dans le constructeur.

```
import java.util.Stack;
private Stack<Room> aLastRooms;
```

```
this.aLastRooms = new Stack<Room>();
```

Ainsi à chaque appel à *goRoom*, la pièce courante sera ajoutée à la pile avant que le joueur ne se déplace.

```
this.aLastRooms.push(this.aCurrentRoom);
```

La méthode *back* changera la pièce courante en celle se situant au sommet de la pile.

```
this.aCurrentRoom = this.aLastRooms.pop();
```

Ainsi, la commande *retour* est correctement implémentée.

7.28.1)

Trois imports ont été fait dans *GameEngine*.

```
import java.io.FileNotFoundException;

import java.io.File;

import java.util.Scanner;
```

La commande test a été ajoutée dans les commandes valides, la méthode *interpretCommand* pourra l'interpréter. Une méthode test a été ajoutée à *GameEngine*, celle-ci vérifie si la commande *test* contient un second mot désignant le fichier et si ce fichier existe. Puis, elle ouvre le fichier texte et le traite ligne par ligne.

```
String vWord2 = pC.getSecondWord();

Scanner vS = new Scanner (new File (vWord2+".txt"));

while ( vS.hasNext()) {

        this.interpretCommand (vS.nextLine());

} // if

 return;
```

7.28.2)

(Une commande *valider* a été ajoutée à ce stade. En se rendant en bureau administratif, un tableau avec une correspondance numéro et expression de fonction sera donnée. Le joueur doit trouver l'expression de f, c'est-à-dire le bon numéro de la liste de l'item *papier*. Une commande *liste* a été ajoutée permettant de regarder le *papier*. Ce choix d'utiliser les numéros est un soucis de facilité.)

Le moyen idéal de gagner est d'avoir le minimum d'indices, de déplacements nécessaires pour être sûr de sa réponse. Une méthode *valid* de *GameEngine* a été ajouté et affichera un message pour féliciter le joueur avant d'appeler *endGame*. Le moyen exploitant tous les possibilités était dans un premier temps une exploitation de tous les lieux, le fichier se nomme exploration. Maintenant, elle est conforme aux attentes.

7.29)

Un attribut *aNameItem* donnant le nom de l'item a été ajouté. Les modifications dans les classes *Room*, *GameEngine* et Item ont été apportées.

Nous allons procéder à la réingénierie en créant une classe *Player* afin de contrôler les items portées par celui-ci. On déplacera des méthodes pouvant être gérées par *Player* et des champs *aLastRooms* et *aCurrentRoom* dans la classe *Player*. Ainsi, nous pensons également au futur, si nous voulons par la suite implémenter plusieurs joueurs, il sera beaucoup plus aisé.

Tous les appellations à *aCurrentRoom* dans *GameEngine* sont fait par l'accesseur *getCurrentRoom* de *Player*. Pour modifier la pièce courante, la classe *GameEngine* appelera des méthodes de *Player*. La méthode *goRoom* appelera setNext pour répondre à la commande aller. La méthode back appelera *lastRoomIsEmpty* qui est une méthode boolean testant si la pile des pièces précédentes est vide, ainsi que *setLast* pour répondre à la commande *retour*.

Les tests ont été effectués. Tout fonctionne bien.

7.30)

Les commandes *prendre* et *poser* ont été implémentées. Après avoir vérifié que ces commandes ont un second mot et que l'item est bien dans la pièce/sur le joueur, celui-ci est pris/posé.

L'attribut Item aItem*Player* a été ajouté à *Player* l'autorisant à avoir un item. Une méthode *getItem* semblable à *getExit* a été écrite dans *Room*. Celui-ci renvoie l'item correspondant au nom *String* entré en paramètre. Il servira dans les méthodes *drop* et *take* de *GameEngine* qui prend une commande en paramètre à sélectionner l'item à prendre/poser. Deux méthodes *pickItemPlayer* et *dropItemPlayer* de *Player* ont été créées pour modifier *aItemPlayer*. Bien sûr, la liste des items de la pièce courante est modifiée. Pour cela, on crée une méthode dans *Room* nommée *removeItem s*emblable à *addItem* mais en prenant en paramètre un *String* pour la clé.

```
String vS = pC.getSecondWord();

Item vItem = this.aPlayer.getCurrentRoom().getItem(vS);
```

pour take :

```
this.aPlayer.getCurrentRoom().removeItem(vS);

this.aPlayer.pickItemPlayer(vItem);
```

avec public void pickItem*Player*(final Item pItem) {this.aItem*Player* = pItem;}

pour drop:

```
this.aPlayer.getCurrentRoom().addItem(vItem);

this.aPlayer.dropItemPlayer(vItem);
```

avec

```
public void dropItemPlayer(final Item pItem) {this.aItemPlayer = null;}
```

7.31)

Pour permettre au joueur de prendre plusieurs Item. On remplace l'attribut Item aItem*Player* en *HashMap <String, Item> aInventary*. On modifie *pickItemPlayer* et dropItem*Player* et on change vS en vItem pour l'appellation à *dropItemPlayer* car celui-ci prend maintenant un *String*.

```
public void pickItemPlayer (final Item pItem)

        {this.aInventary.put(pItem.getNameItem(), pItem);}

public void dropItemPlayer (final String pS)
```

{this.aInventary.remove(pS);}

7.31.1)

On crée une classe ItemList qui gère une liste d'items HashMap <*String*, Item> aItemMap et ainsi éviter la duplication de code dans *Player* et *Room*. Un objet ItemList nommé aItemList sera en attribut de *Player* et de *Room* et crée dans leurs constructeurs.

On déplace la méthode *getItemString* de *Room* dans la classe ItemList pour un potentiel affichage de l'inventaire du joueur sous cette forme.

On crée un accesseur *getIntemMap* qui renvoie aItemmap. Les modifications nécessaires des méthodes *Player* et *Room* pour sont apportées.

La méthode *GameEngine* n'a effectivement pas eu besoin de modifications.

7.32)

On ajoute deux attributs int *aWeightMax* et aWeight*Player* à *Player*. On initialise aWeightMax à 10 (Mo). Le joueur ne pourra que prendre deux pièces de 1i (monnaie) qui lui permettront de participer au donjon pour remporter une calculatrice et une carte mémoire (magic cookie). Un acceseur *getWeightPlayer* a été ajouté. Deux procédures addWeight*Player* et removeWeight*Player* prenant en paramètre un Item ont été écrites dans *Player*.

Pour *addWeightPlayer* :

this.aWeight*Player* += pItem.getWeightItem();

Pour *removeWeightPlayer* :

this.aWeight*Player* -= pItem.getWeightItem();

Une méthode boolean *iCanTake* de *Player* prenant un Item en paramètre a aussi été ajoutée. Celle-ci renvoie true si le joueur est en mesure de prendre l'item.

return (pItem.getWeightItem() + this.aWeight*Player* <= this.aWeightMax);

Si le joueur ne peut pas le prendre, un affichage le signalera dans la méthode take.

Pour *take* :

```
else if ( this.a Player .iCanTake(vItem)){
    this.a Player .getCurrent Room ().removeItem(vS);
    this.a Player .pickItem Player (vItem);
    this.a Player .addWeight Player (vItem);
    this.aGui.println ("Vous avez pris l'item " + vS);
```

```
this.aGui.println (vItem.getDescriptionItem());
}
else this.aGui.println("Nous n'avez plus assez de mémoire dans la carte!");
```

drop s'écrit de manière analogique.

7.33)

La commande *inventaire* a été ajoutée et bien définie. Une méthode *items* qui renvoie la liste des items en possession et la capacité de stockage utilisé par la fonction a été implémentée.

```
this.aGui.println ("Items en possession : "+ this.a Player.getItemList().getItem String());
this.aGui.println ("Vous avez utilisé " + this.a Player.getWeight Player() + "Mo");
```

7.34)

la commande *manger* a été remplacée par *insérer*. Le joueur n'aura à le taper qu'une fois suivi de carte_stockage après avoir une carte de stockage au donjon intellectuel. Cette commande appelera la méthode *addCapacity* qui vérifie que le joueur est bien en possession d'une calculatrice et qu'il cherche bien à insérer la carte mémoire dans celle-ci. Une méthode *setWeightMax* prenant un int est implémentée dans *Player*, celui-ci change la valeur de *aWeightMax* en celle du paramètre.

Pour *addCapacit*y :

```
String vS = pC.getSecondWord();
Item vItem = this.aPlayer.getCurrentRoom().getItem(vS);
if (this.aPlayer.getItemPlayer("Calculatrice") != null)
        if (vS.equals( "Carte_stockage")) {
                this.aPlayer.setWeightMax (1000);
                this.aGui.println ("Vous avez à présent une capacité de stockage de 1000
Mo");
}
        else this.aGui.println ("Cet item n'est pas une carte mémoire!");
else this.aGui.println ("Vous n'avez pas de calculatrice!");
```

7.34.1)

On a écrit un fichier *test magic.txt* qui consiste à tester uniquement le « magic cookie ». le fichier *exploration.txt* a été complétée.

7.34.2)

Les deux javadoc sont générés comme convenu.

7.35)

Actuellement, l'interface est très dépendant de la langue. Un changement de langue de la liste des commandes dans la classe CommandWords entraîne des modifications nécessaires dans la méthode interpretCommand de GameEngine.

Pour éviter ce couplage implicite, on crée une classe enum CommandWord qui stocke la liste des commandes en prenant soin d'ajouter un CommandWord UNKNOWN. On utilise la syntaxe suivante dans interpretCommands pour découpler :

```
if ( vCommandWord == CommandWord.HELP)
```

L'attribut tableau aValidCommands de CommandWords est remplacé par une Hashmap<String, CommandWord> auquel on associe chaque CommandWord à une chaîne de caractère. On change l'attribut String aFirstWord de la classe Command par CommandWord aCommandWord. On effectue les changements nécessaires au sein de la classe. On ajoute un accesseur getCommandWord dans la classe CommandWords qui prend un String en paramètre pour renvoyer le CommandWord. Cet acesseur sert notamment dans la méthode getCommand de Parser pour créer une nouvelle commande à partir des chaînes de caractères tapées sur l'interface.

7.35.1-7.41.2)

optionnel

7.42)

Pour l'instant, on a crée un « time limit » imposant au joeur de gagner avant 50 déplacements aller. On crée pour cela un attribut int aCount dans *GameEngine*. À la fin de chaque déplacement, on ajoute 1 à aCount et si *aCount* est égal à 50, la méthode end*Game* sera appelé. De plus si le joueur se trompe à trois reprises à l'énigme du donjon, il perd aussi.

7.42.2)

Je vais me contenter de l'interface graphique actuelle pour le moment.

7.43)

Le jeu comportera comportera un trap door qui est le passage entre l'entrée du donjon et la salle 1 du donjon.

L'*ArrayList aRoomList* nous permet d'accèder aux *Room* du jeu par les indices. Une méthode exhaustive a été utilisé pour modifier *goRoom* afin qu'il vide la pile des pièces précédentes s'il détecte le passage.

Toutes fois, cette manière de procéder ne respecte pas les conseils données pour la réingénierie. La question 7.45 améliorera cete partie du code.

7.44)

Les commandes *charger* et *déclencher* ont été bien implémentées.

On a crée une classe Beamer en précisant que *Beamer* hérite de la classe *Item*. Il possède un attribut *Room aRoomBeamer* permettant la sauvegarde d'une *Room*.

Il possède aussi un acceseur permettant d'avoir accès à cet attribut. Enfin une procédure *chargeBeamer* est écrite, celle-ci impose à a*RoomPlayer* la *Room* passé en paramètre.

Deux méthodes *charge* et *fire* ont été crée dans *GameEngine*. La méthode *charge* vérifie que le joueur possède bien un Beamer et appelle *chargeBeamer* sur la pièce courante si c'est le cas. La méthode fire appelle *setCurrentRoom* pour lui imposer *vBeamer.getRoomBeamer,* puis vide la pile des pièces précédentes avant d'appeller *printLocationInfo.*

7.45)

La classe Door codée comporte 4 attributs :

private Room aFirstRoom; // pouvant accèder à la porte

private Room aSecondRoom; // pouvant accéder à la porte

private boolean aIsUnlocked; // renvoie true si la porte est dévérouillé, false sinon

private Item aKey; // permet de dévérouiller une porte avec la commande dévérouiller suivie d'une direction.

Deux accesseurs pour aIsUnlocked et aKey sont codés. Une procédure setDoorToRoom qui appelle deux fois addDoor pour ajouter les portes aux deux Room a été écrite, ainsi qu'une procédure setLocked imposant false à aIsUnlocked.

Une HashMap<Room,Door> aDoors qui se superposent aux Rooms a été ajoutée. La méthode addDoor ajoute une association Room et Door dans la HashMap. Les portes et la mise en place des portes sur les Room avec setDoorToRoom sont faites dans le createRoom de GameEngine. On a aussi un accesseur Door getDoor qui renvoie la porte entre la Room courante et la Room en paramètre.

```
return this.aDoors.get(pRoom);
```

Ainsi que la méthode boolean isExit qui vérifie si aller dans une direction est possible. Si une porte se présente, isExit renverra l'État de la porte.

```
return this.aDoors.get(pRoom).getIsUnlocked() ;
```

sinon il renvoie constamment *true*.

Cela est juste car *goRoom* traîte le cas où le lieu suivant est *null* en premier temps. La méthode goRoom vérifiera ensuite que le passage est possible avec isExit avant d'effectuer le processus de changement de pièce.

La commande dévérouiller est ajoutée, ainsi que son traitement dans interpretCommand et une méthode unlock. La méthode *unlock* vérifiera dans un premier temps qu'il y a bien une porte.

```
Room vRoomToGo = this.aPlayer.getCurrentRoom().getExit(pC.getSecondWord());

if  (this.aPlayer.getCurrentRoom().getDoor(vRoomToGo)
```

puis que la porte est bien vérouillée, sinon il affichera juste un message disant que la tâche est déjà faite.

```
Door vDoor= this.aPlayer.getCurrentRoom()

if (vDoor.getIsUnlocked())
```

Puis il regardera si le Player possède bien l'item clé permettant de le dévérouiller.

```
if ( this.aPlayer.getItemPlayer  (vDoor.getKey().getNameItem())
```

si les conditions sont vérifiées,

```
vDoor.setUnlocked();
```

et un message de confirmation sera affiché.

En possession de notre classe Door, on peut maintenant améliorer les TrapDoor. Un TrapDoor est une sorte de Door, donc il a été décidé de créer une classe TrapDoor sans attribut supplémentaire et sans méthodes qui hérite de Door. Son constructeur ne prend que les deux paramètres Room, et on impose *null* à la clé et *false* à l'état de la porte.

La méthode *goRoom* après avoir vérifier que le passage est possible, fermera le passage inverse si la porte était une *TrapDoor*. Pour cela elle convertira le type *Door* en *TrapDoor*, si le revoie n'est pas null, il s'agit d'un *TrapDoor*. Un message s'affichera, la porte se refermera à jamais (Il n'y aura qu'un TrapDoor qui servira à accéder à la salle 1 du Donjon, où le joueur s'aventura qu'une fois), la pile des pièces précédentes se videra avant d'afficher les informations de la nouvelles pièce.

```
TrapDoor vTrap = (TrapDoor) vCurrentRoom.getDoor(vNextRoom);
if (vTrap != null) {
    this.aGui.println ("La porte derrière vous s'est refermée.");
    vCurrentRoom.getDoor(vNextRoom).setLocked();
    this.aPlayer.setNext(vNextRoom);
    this.aPlayer.clearLastRooms();
}
```

7.45.1)

Les fichiers de test sont mises à jour.

7.45.2)

Les javadocs ont été générés avec succès.

7.46)

On a crée une nouvelle classe TransporterRoom qui hérite de *Room*. Cette façon de faire réduit le couplage. Une nouvelle classe RoomRandomizer a aussi été créée, elle sert à générer un entier au hasard avec la classe Random et possède la méthode *findRandomDonjonRoom* que *TransporterRoom* appelera. La liste choisie est *aDonjonEnigmeRoom*. Le transporteur permet de téléporter le joueur au début de donjon. Quand le joueur gagne il sera téléporté à l'entrée du donjon. Il a été décidé aussi de placer un trapdoor sur le passage au *TransporterRoom* afin qu'il ne puisse plus y accèder.

7.46.1)

Non traité

7.46.3)

Les commentaires javadoc ont été visualisés.

7.46.4)

Les javadocs ont été regénérés.

7. 48)

J'ai ajouté des personnages car il me semble indispensable d'en avoir dans mon jeu d'aventure. La méthode pour implémenter des personnages se rapproche de celle des items.

Dans mon jeu, il y aura un personnage par pièce. Si la commande parler est tapée, le personnage parlera pour se présenter.

5 attributs ont été crées :

private String aNameCharacter;

private String aPresentation*Character*; //s'affiche si la commande parler est tapée

private Item aItemForPlayer; //possède un item d'échange

private Item aItemWanted; //exige de recevoir cet item pour donner aItemForPlayer au joeur ou/et une indice *String* aIndice.

private String aIndice;

La classe *Character* possède 5 acceseurs correspondant à ces attributs ainsi qu'une méthode *changePresentation* qui change la présentation du *character* en un encouragement et éventuellement le rappel de l'indice.

Un attribut *Hashmap <String, Character> aCharacters* est ajoutée à *Room*, ainsi que des méthodes add*Character*, *getCharacterString* semblable ax méthodes gérant les items.

La méthode *createRooms* de la classe *Game* se charge aussi de créer les *Character*s et de les ajouter dans des *Room*s à l'aide de la méthode add*Character*. Une commande *donner* est bien implémentée. *InterpretCommand* appelera la méthode *give.* Si les conditions de lieux et d'item sont vérifiées. Celle-ci va appeler la méthode *CharacterHelpPlayer* qui vérifie si le *Character* a un item ou/et un indice et le donner au joueur puis va changer la présentation du *Character* en appelant *changePresentation.*

Un personnage qui est la voyante demande effectivement quelque chose mais ne donnera rien en retour (ni item, ni indice). C'est le seul Character qui agit ainsi, car il s'agit d'un examinateur. Si le joueur lui donne une pièce d'1i en espèrant que la voyante le définit à sa place. l'examinateur le considèrera trop naïf pour être une fonction et le joueur aura perdu.

Annexe

1. Tableau de fonctions validables.

	continue	Stric.decr	Coupe l'abscisse	Lim finie	Stric. convexe	Non périodique	Non paire	fonction
					NON			arctan(-x)
			NON					1/sqrt(x)
		NON		NON			NON	x^2
		NON		NON				x*sqrt(x)
					NON			-th(x)
				NON	NON			ln(-x)
								exp(- x) - 1
		NON	NON		NON	NON	NON	10
				NON	NON			- x
				NON				- ln(x)
		NON	NON		NON			xexp(-x)
		NON		NON	NON	NON	NON	4x+12