

Rapport :

Titre provisoire + lien page web

(A3P(AL) 2015/2016 Ga)

I.A) Auteur(s)

I.B) Thème (phrase-thème validée)

I.C) Résumé du scénario (complet)

I.D) Plan (complet, avec indication de la partie "réduit" si exercice 7.3.3)

I.E) Scénario détaillé (complet, avec indication de la partie "réduit" si exercice 7.3.3)

I.F) Détail des lieux, items, personnages

I.G) Situations gagnantes et perdantes

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

I.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 (*)

V, VI, etc... : tout ce que vous voulez en plus

(*) 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).

Thème (phrase-thème validée):

Dans un vaisseau spatial en perdition un membre d'équipage doit réparer les systèmes vitaux pour gagner le jeu dans un temps limité.

Résumé du scénario (complet)

Noms:

Personnage principale(Nous): Mike

À la radio: Lara

Personne qui nous aide James

On est dans un vaisseau spatial en perdition est sur notre tableau de bord on voit que les moteurs sont défaillants.

Cette casse est sûrement liée à l'attaque d'ennemis située sur l'aile droite que nous apercevons à la caméra de notre tableau de bord.

Il faut donc aller les vaincre afin qu'il ne fasse pas de dégâts supplémentaires et aller réparer nos moteurs en toute sécurité.

Aller à l'aile droite vaincre les ennemis.

Aller réparer le moteur droit.

On entend un explosions "BOOM" à l'aile gauche du vaisseau.

Se rendre à l'aile gauche, puis arrivé nous voyons un mini-boss à tuer.

Alors le vaincre.

Après cela Lara nous contacte et nous demande d'aller absolument réparer le moteur droit:

"Il faut absolument réparer les moteurs sinon nous allons nous écraser"

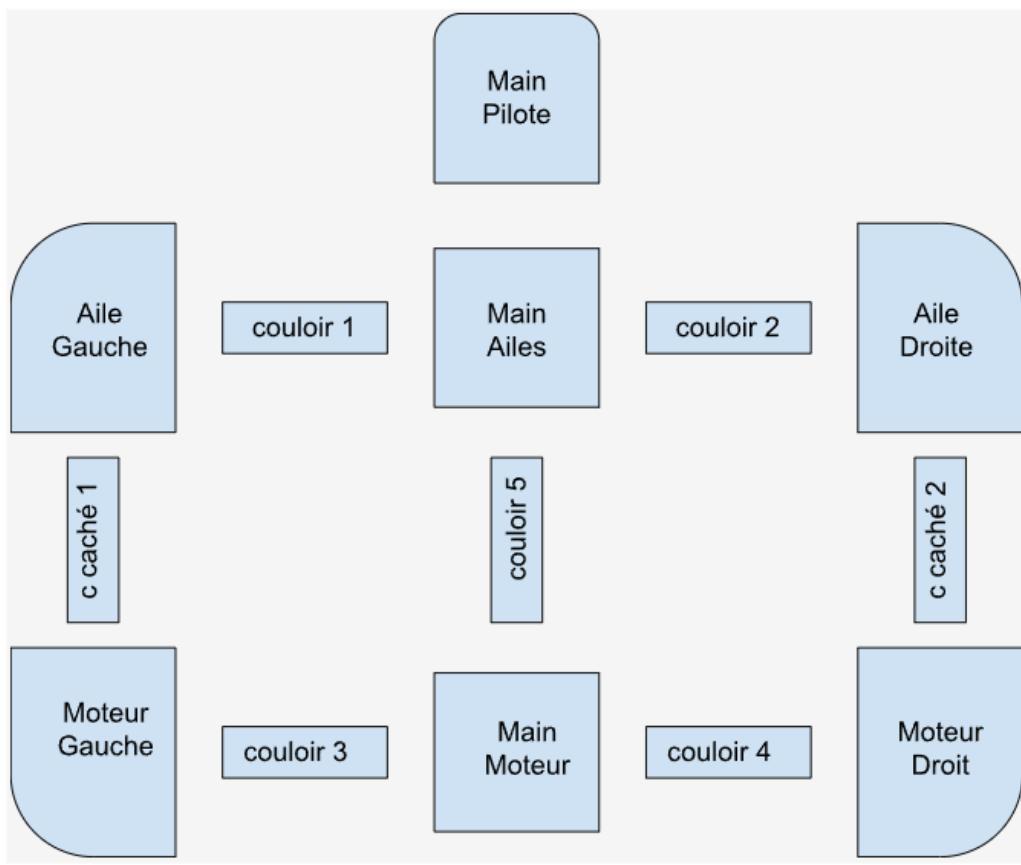
Aller au moteur droit, il y a un Boss, le vaincre et réparer le moteur gauche.

Vaincre le Boss Final.

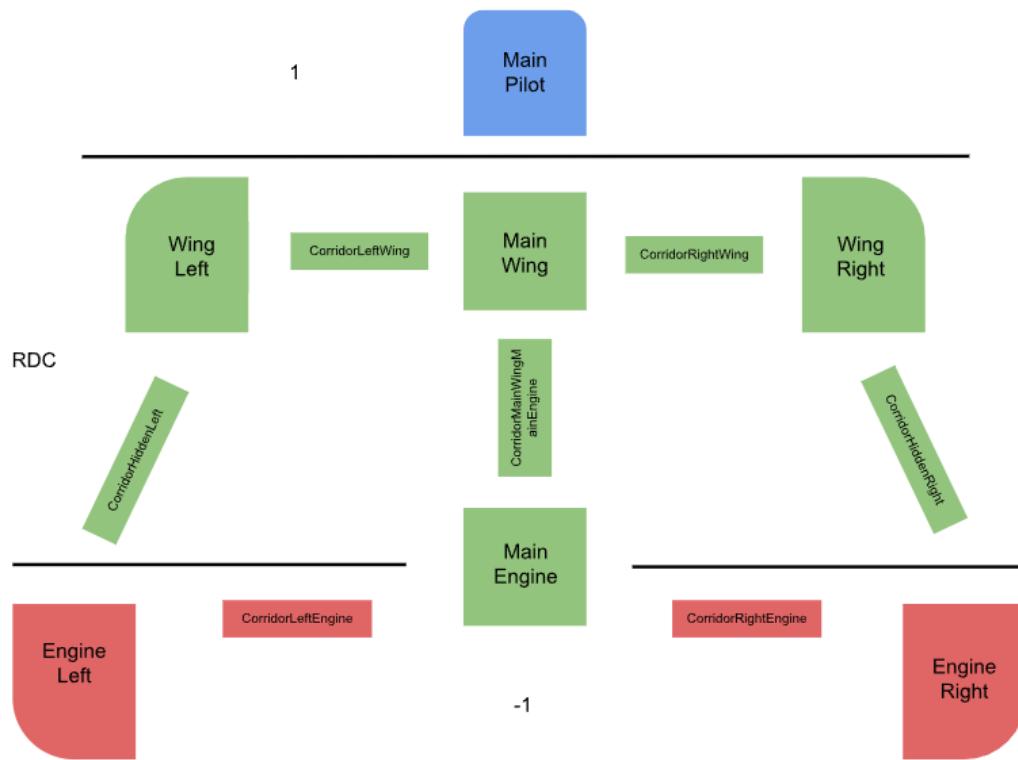
C'est la victoire.

Plan (complet):

- V1:



- V2:



Exercice 7.5:

Question 7.5:

Avez-vous compris pourquoi on vous demande de créer et utiliser cette procédure, et pourquoi la situation était "inacceptable" avant cette amélioration ?

Oui car dans printWelcome() donne le message de bienvenue et donne l'information sur notre emplacement au début du jeu, et dans goRoom() nous donne les différentes possibilités de sortir de la room actuel avec la description de la room actuel. On demande de vérifier la même chose(s'il y a une sortie)

Et on était obligé de le faire comme ça car les deux méthodes données des informations sur l'emplacement actuel mais elles ne pouvaient s'appeler car elles sont aussi différentes.

C'est donc pour cela qu'on créer une autre méthode(procédure) qui donne juste l'information de l'emplacement de la room actuel. Et donc elle l'appel dans printWelcome() pour l'emplacement de la première room et dans goRoom() à chaque fois que l'on change de room.

Modifications de code apportés:

Game : createRooms() :

```

//Direction : pNorthExit, pSouthExit, pWestExit, pEastExit
vMainPilot.setExits(null, vMainWing, null, null);
vMainWing.setExits(vMainPilot, vCorridorMainWingMainEngine, vCorridorLeftWing, vCorridorRightWing);
vMainEngine.setExits(vCorridorMainWingMainEngine ,null, vCorridorLeftEngine, vCorridorRightEngine);

vRightWing.setExits(null, vCorridorHiddenRight, vCorridorRightWing, null);
vLeftWing.setExits(null, vCorridorHiddenLeft, null ,vCorridorLeftWing);
vRightEngine.setExits(vCorridorHiddenRight, null, vCorridorRightEngine, null);
vLeftEngine.setExits(vCorridorHiddenLeft, null, null, vCorridorLeftEngine);

vCorridorRightWing.setExits(null, null, vMainWing, vRightWing);
vCorridorLeftWing.setExits(null, null, vLeftWing, vMainWing);

vCorridorMainWingMainEngine.setExits(vMainWing, vMainEngine, null, null);

vCorridorRightEngine.setExits(null, null, vMainEngine, vRightEngine);
vCorridorLeftEngine.setExits(null, null, vLeftEngine, vMainEngine);

vCorridorHiddenRight.setExits(vRightWing, vRightEngine, null, null);
vCorridorHiddenLeft.setExits(vLeftWing, vLeftEngine, null, null);

this.aCurrentRoom = vMainPilot;

} // createRooms()

```

Game : printLocationInfo() :

```

/**
 * give: the current room in which the user is located, the room description, its different exits
 * it was created to avoid code duplication between printWelcome() and goRoom()
 * @method procedure printLactionInfo()
 *
 */
private void printLocationInfo() {
    System.out.println("You are " + this.aCurrentRoom.getDescription());
    System.out.print("Exits: ");

    if(this.aCurrentRoom.aNorthExit != null) {
        System.out.print("north ");
    }

    if(this.aCurrentRoom.aSouthExit != null) {
        System.out.print("south ");
    }

    if(this.aCurrentRoom.aWestExit != null) {
        System.out.print("west ");
    }

    if(this.aCurrentRoom.aEastExit != null) {
        System.out.print("east ");
    }
    System.out.println();
} //printLocationInfo()

```

Game : printWelcome()

```
private void printWelcome() {  
    System.out.print("Welcome to the world of Zuut!\nWorld of Zuul is an incredibly boring new  
    System.out.println("\nYou are in the pilot room of the ship you can exit to: South");  
    System.out.println();  
    printLocationInfo();  
} //printWelcome()
```

Game : end goRoom()

Permet de donner définir la room actuelle et d'appeler la méthode printLocationInfo();

```
if (vNextRoom == null) {  
    System.out.println("There is no door !");  
    return;  
} else {  
    this.aCurrentRoom = vNextRoom;  
    printLocationInfo();  
}  
} // goRoom()
```

Exercice 7.6:

Question 7.6:

La version de setExits proposée dans le livre teste si chaque paramètre est null avant de l'affecter à l'attribut correspondant, mais ce n'est pas indispensable. Comprenez-vous pourquoi ?

Oui cela n'est pas indispensable car dans tous les cas une deuxième vérification se fait dans la classe Game. En effet la méthode printLocationInfo() vérifie chaque sortie aussi pour connaître s'il existe ou pas des directions(sorties) pour chaque room.

Modifications de code apportés:

Room : setExits() :

```
/**  
 * define the exits for the all rooms. All direction if its possible, if that rooms egal null no exits here |  
 * @method setExits(Room pNorthExit,Room pSouthExit,Room pWestExit,Room pEastExit)  
 */  
public void setExits(final Room pNorthExit, final Room pSouthExit, final Room pWestExit, final Room pEastExit) {  
    if(pNorthExit != null) {  
        this.aNorthExit = pNorthExit;  
    }  
  
    if(pSouthExit != null) {  
        this.aSouthExit = pSouthExit;  
    }  
  
    if(pWestExit != null) {  
        this.aWestExit = pWestExit;  
    }  
  
    if(pEastExit != null) {  
        this.aEastExit = pEastExit;  
    }  
} //setExits()
```

Room : getExit() :

A la base les attributs de cette classe étaient publics, cela pose donc un problème car la classe Room explose dans son interface le fait qu'elle possède des sorties mais aussi comment les informations de sorties sont stockées. C'est donc un principe de base en programmation qui est brisé: l'encapsulation. Je dois donc les rendre privés. Mais je dois toujours pouvoir les utiliser en dehors de classe (dans la classe Game). Pour cela j'ai donc utilisé un getter: getExits() (accesseur) qui va donc retourner les attributs que si la direction est égale à celle qui existe.

```

/**
 * return all attribute relative to the direction
 * @method |getExit(String pDirection)
 */
public Room getExit(final String pDirection) {
    if(pDirection.equals("north")) {
        return this.aNorthExit;
    }

    if(pDirection.equals("south")) {
        return this.aSouthExit;
    }

    if(pDirection.equals("west")) {
        return this.aWestExit;
    }

    if(pDirection.equals("east")) {
        return this.aEastExit;
    }

    return null;
} //getExit()

```

Maintenant je dois utiliser cette méthode dans notre classe Game car je n'ai plus accès aux attributs publics.

Dans la méthode goRoom() qui permet de changer de room qui si la sortie existe bien dans la room.

```

Room vNextRoom = null;
String vDirection = pCommand.getSecondWord();

if(vDirection.equals("north")) {
    vNextRoom = this.aCurrentRoom.getExit("north");

} else if( vDirection.equals("south")) {
    vNextRoom = this.aCurrentRoom.getExit("south");

} else if( vDirection.equals("west")) {
    vNextRoom = this.aCurrentRoom.getExit("west");

} else if( vDirection.equals("east")) {
    vNextRoom = this.aCurrentRoom.getExit("east");

} else {
    System.out.println("Unknown Direction !");
    return;
}

```

Je dois aussi l'utiliser dans la méthode printLocationInfo() qui nous permet de connaitre les prochaines directions disponibles de notre room actuelle.

```
/**  
 * give: the current room in which the user is located, the room description, its different exits  
 * it was created to avoid code duplication between printWelcome() and goRoom()  
 * @method procedure printLocationInfo()  
 */  
  
private void printLocationInfo() {  
    System.out.println("You are " + this.aCurrentRoom.getDescription());  
    System.out.print("Exits: ");  
  
    if(this.aCurrentRoom.getExit("north") != null) {  
        System.out.print("north ");  
    }  
  
    if(this.aCurrentRoom.getExit("south") != null) {  
        System.out.print("south ");  
    }  
  
    if(this.aCurrentRoom.getExit("west") != null) {  
        System.out.print("west ");  
    }  
  
    if(this.aCurrentRoom.getExit("east") != null) {  
        System.out.print("east ");  
    }  
    System.out.println();  
} //printLocationInfo()
```

Exercice 7.7:

Question 7.7:

Comprenez-vous pourquoi il est logique de demander à Room de produire les informations sur ses sorties (et ne pas lui demander de les afficher), et pourquoi il est logique de demander à Game d'afficher ces informations (et ne pas lui demander de les produire) ?

Il est logique de demander à Room de produire les informations sur ses sorties car ce sont les siennes et cela est plus logique de comparer ses propres sorties au sein de sa classe que de la faire dans une autre classe. Et donc il est logique de demander à Game d'afficher ces informations car Game est responsable de l'interaction utilisateur et de l'affichage. Cela est aussi logique car chaque classe doit avoir une responsabilité bien définie.

Ici la classe Game est responsable de l'affichage et de l'interaction user. Alors que la classe Room est responsable de la gestion des sorties.

Si plus tard dans le projet j'ai besoin d'utiliser les informations de sorties de Room alors je pourrais le faire en appelant les getters de la classe Room sans passer par la classe Game, cela est donc plus modulable et Room ne dépend plus de Game.

Aussi, si je veux ajouter d'autres sorties je n'aurai qu'à le faire dans la classe Room, plus besoin de modifier la classe Game.

En somme, cela rend le code plus modulaire, maintenable et extensible pour la suite du projet.

Room : getExitString() :

```
/*
 * print different output calls in printLocationInfo() in Game class if output is not null
 * @method
 */
public String getExitString() {
    if(this.aNorthExit != null) {
        System.out.print("north ");
    }

    if(this.aSouthExit != null) {
        System.out.print("south ");
    }

    if(this.aWestExit != null) {
        System.out.print("west ");
    }

    if(this.aEastExit != null) {
        System.out.print("east ");
    }

    return null;
} //getExitString
```

Game : printLocationExit() :

Dans cette méthode nous n'avons plus besoin de vérifier si la sortie est égale à null car nous le faisons déjà dans getExitString() je remplace cela par un simple appel de la méthode getExitString()

```
private void printLocationInfo() {  
    System.out.println("You are " + this.aCurrentRoom.getDescription());  
    System.out.print("Exits: ");  
    this.aCurrentRoom.getExitString();  
    System.out.println("\n");  
} //printLocationInfo()
```

Exercice 7.8, 7.8.1:

Pour cet exercice je dois utiliser la notion de HashMap dans la classe Room afin de pouvoir créer nos propres descriptions exemple(up, down)

Grâce au HashMap la direction de la Room sera créée lorsque l'on créer la room dans createRooms(), il n'y aura donc plus de direction pré définie, Cela veut dire que je pourrais avoir n'importe quelle direction.

Room : Constructeur Room :

```
import java.util.HashMap;

/**
 * Classe Room - un lieu du jeu d'aventure Zuul.
 *
 * @author Hautin Matthias
 */
public class Room
{
    private HashMap<String, Room> aExits;
    private String aDescription;

    /**
     * initialize all attribute
     * @Constructor Room() use for create new room
     * @param String pDescription intialize Room description
     */
    public Room(final String pDescription) {
        this.aDescription = pDescription;
        this.aExits = new HashMap<String, Room>();
    } // Room()
}
```

Room : getter getDescription() :

Celui-ci permet de retourner la chaîne de caractères de la description

```
/**
 * return aDescription attribute, String, getter
 * @method getDescription()
 */
public String getDescription() {
    return this.aDescription;
} //getDescription()
```

Room : setter getExits() :

Permet de définir les sorties possibles de chaque Room avec le paramètre pDirection de type String ce qui nous permet de créer n'importe quelles directions , et donne la prochaine room de cette direction avec le paramètre pNeighbor(voisin) de type Room car c'est une room

```
/**
 * define the exits for the all rooms. All direction if its possible, if that rooms egal null no exits here, procedure, setter
 * @method setExits()
 * @param pDirection Exit direction
 * @param pNeighbor the given direction room
 */
public void setExits(final String pDirection, final Room pNeighbor) {
    this.aExits.put(pDirection,pNeighbor);
} //setExits()
```

Room : getter getExit() :

Permet de retourner toutes les directions disponibles pour la prochaine room, si la direction est nulle alors cette direction n'existe pas

```
/**
 * return all directions relating to room , Room, getter
 * @method getExit()
 * @params String pDirection given all direction for next room else null if direction is null
 */
public Room getExit(final String pDirection) {
    return this.aExits.get(pDirection);
} //getExit()
```

J'ai donc dû recréer et redéfinir toutes mes sorties pour chaque room:

```
VLaRoom.setExits("String pDirection", NextRoom en relation avec la direction);
```

```
//setExits : String pDirection, Room pNeighbor
// for vMainPilot:
vMainPilot.setExits("down", vMainWing);

//for vMainWing
vMainWing.setExits("up", vMainPilot);
vMainWing.setExits("south", vCorridorMainWingMainEngine);
vMainWing.setExits("west", vCorridorLeftWing);
vMainWing.setExits("east", vCorridorRightWing);

//for vMainEngine
vMainEngine.setExits("north", vCorridorMainWingMainEngine);
vMainEngine.setExits("downwest", vCorridorLeftEngine);
vMainEngine.setExits("downeast", vCorridorRightEngine);

//for vRightWing
vRightWing.setExits("south", vCorridorHiddenRight);
vRightWing.setExits("west", vCorridorRightWing);

//for vLeftWing
vLeftWing.setExits("south", vCorridorHiddenLeft);
vLeftWing.setExits("east", vCorridorLeftWing);

//for vRightEngine
vRightEngine.setExits("upnorth", vCorridorHiddenRight);
vRightEngine.setExits("west", vCorridorRightEngine);

//for vLeftEngine
vLeftEngine.setExits("upnorth", vCorridorHiddenLeft);
vLeftEngine.setExits("east", vCorridorLeftEngine);
```

```
//vCorridorRightWing
vCorridorRightWing.setExits("west", vMainWing);
vCorridorRightWing.setExits("east", vRightWing);

//vCorridorLeftWing
vCorridorLeftWing.setExits("west", vLeftWing);
vCorridorLeftWing.setExits("east", vMainWing);

//vCorridorMainWingMainEngine
vCorridorMainWingMainEngine.setExits("north", vMainWing);
vCorridorMainWingMainEngine.setExits("south", vMainEngine);

//vCorridorRightEngine
vCorridorRightEngine.setExits("upwest", vMainEngine);
vCorridorRightEngine.setExits("east", vRightEngine);

//vCorridorLeftEngine
vCorridorLeftEngine.setExits("west", vLeftEngine);
vCorridorLeftEngine.setExits("upeast", vMainEngine);

//vCorridorHiddenRight
vCorridorHiddenRight.setExits("north", vRightWing);
vCorridorHiddenRight.setExits("downsouth", vRightEngine);

//vCorridorHiddenLeft
vCorridorHiddenLeft.setExits("north", vLeftWing);
vCorridorHiddenLeft.setExits("downsouth", vLeftEngine);

this.aCurrentRoom = vMainPilot;
```

Pour la méthode getExitString je n'ai pas réussi à la faire dans le 7.8 je l'ai donc fait dans le 7.9.

Exercice 7.9:

Pour cet exercice je dois redéfinir la fonction(getter) getExitString() afin qu'elle retourne toutes les sorties possibles pour la room actuelle. Pour cela on importe la classe Set.

Room : getter getExitString() :

```
2 import java.util.Set;
3
4 /**
5  * return all directions that exist for all exits that exist in the room
6  * @method getExitString()
7  */
8 public String getExitString() {
9     String vReturnString = "Exits: ";
10    Set<String> vKeys = this.aExits.keySet();
11    for(String vExit : vKeys) {
12        vReturnString += " " + vExit;
13    }
14    return vReturnString;
15}
16 //getExitString
```

Game : printLocationInfo():

On l'appel dans la méthode printLocationInfo() qui permet d'afficher la description de la room actuelle et toutes les sorties de cette room aussi

```
/*
* give: the current room in which the user is located, the room description, its different exits
* it was created to avoid code duplication between printWelcome() and goRoom()
* @method procedure printLocationInfo()
*
*/
private void printLocationInfo() {
    System.out.println("You are " + this.aCurrentRoom.getDescription());
    System.out.println(this.aCurrentRoom.getExitString());
    System.out.println("\n");
} //printLocationInfo()
```

Exercice 7.10,7.10.1,7.10.2:

- **HashMap**

Le hashmap permet d'associer deux mots clé à leurs définitions.

Dans mon cas pour l'attribut aExits j'associe une direction à une pièces voisine. Cela permet de définir les sorties possibles de chaque room vers d'autres room.

Par exemple pour la variable vMainPilot nous avons "down" qui est associé à la salle vMainWing ce qui fait que lorsque je print "go down" je me retrouve dans la salle vMainWing

- **Set**

Set nous permet de stocker plusieurs valeurs mais jamais 2 fois la même valeur, il peut stocker plein d'éléments mais jamais 2 fois le même.

Par exemple lorsque je vais dans la salle vMainWing les sorties qui sont associé à ces salles sont: Exits: | east | | south | | west | | up |

- **keySet()**

Permet d'obtenir l'ensemble des sets. Elle permet de retourner l'ensemble des valeurs stockées dans le HashMap.

Dans mon cas j'extrais la valeur de chaque sorties "String" de la room actuelle pour pouvoir les afficher.

- **Explication de getExitString() :**

getExitString() est dans la classe Room elle permet d'afficher une chaîne de caractère "String" dans laquelle est continue la liste de toutes les sorties de la Room actuelle. C'est une fonction qui retourne donc une "String".

Elle prend aucun paramètre mais a plusieurs variables locales:

- vReturnString qui contient "Exits: "
- vKeys qui contient toutes les directions associés à la room actuelle: c'est donc les futurs directions possibles pour se déplacer.

Une boucles for:

- Elle permet de stocker chaque direction dans les variables vExit, et ajoute toutes ses directions à la variables vReturnString.

Et pour retourne vReturnString avec tous les éléments qu'elle dispose pour connaître toutes les directions de la room actuelle.

```
/**
 * return all directions that exist for all exits that exist in the room
 * @method getExitString()
 */
public String getExitString() {
    String vReturnString = "Exits: ";
    Set<String> vKeys = this.aExits.keySet();
    for(String vExit : vKeys) {
        vReturnString += " " + vExit;
    }
    return vReturnString;
} //getExitString
```

Exercice 7.11:

- Explications de getLongDescipcion():

Elle return getDescription() et getExitString() et est appelée dans printLocationInfo()

Elle return la description de la salle actuelle et les sorties de cette sont qui y sont associées.

```
/**  
 * @return getDescription() and getExitString() to avoid calling its 2 getters in printLocationInfo()  
 * print for example:  
 * You are in main pilot room.  
 * Exits: | down |  
 * @method getLongDescription()  
 */  
public String getLongDescription() {  
    return "You are " + getDescription() + ".\n" + getExitString() + "\n";  
} //getLongDescription
```

```
private void printLocationInfo() {  
    System.out.println(this.aCurrentRoom.getLongDescription());  
} //printLocationInfo()
```

Exercice 7.14:

Modification de Command Word:

Ajout d'un attribut static et suppression des anciens attributs comme ça plus besoin d'initialiser les attributs dans le constructeur. Plus flexible pour la suite.

```
// a constant array that will hold all valid command words
private static final String aValidCommands[] = {"go", "help", "quit", "look"};
```

- Explications de look():

Cette méthode fonctionne comme quit(),

Elle affiche "I don't know how to look at something in particular yet." si le joueur tape un second mot après la commande "look".

Sinon elle affiche la longue description en appelant getLongDescription() créer juste avant.

```
/*
 * print "I don't know how to look at something in particular yet." if there have a second word after look
 * else give: the current room in which the user is located, the room description, its different exits
 * @method procedure look()
 */
private void look(final Command pSecondMot) {
    if(pSecondMot.hasSecondWord() == true ) { //si l'utilisateur tap un second mot après "quit" (exemple: "quit program")
        System.out.println("I don't know how to look at something in particular yet.");
    } else {
        System.out.println(this.aCurrentRoom.getLongDescription());
    }
} //look()
```

Exercice 7.15:

- Explication de getEat():

Elle return une chaîne de caractère "You have eaten now you are not hungry any more."

```
/**  
 * @return "You have eaten now and you are not hungry any more.\n"  
 * @method getEat()  
 */  
public String getEat() {  
    return "You have eaten now and you are not hungry any more.\n";  
} //getEat()
```

- Explication eat():

Elle fonctionne comme look() et quit()

Affiche "Just on thing at a time." si le joueur saisi un deuxième mot après la commande eat

Sinon elle affiche le message que return getEat()

```
/**  
 * print "Just one thing at a time." if there have a second word after eat  
 * else print: "You have eaten now and you are not hungry any more."  
 * @method procedure eat()  
 */  
private void eat(final Command pSecondMot) {  
    if(pSecondMot.hasSecondWord() == true ) { //si l'utilisateur tape un second mot apres "eat" (exemple: "eat fish")  
        System.out.println("Just one thing at a time.\n");  
    } else {  
        System.out.println(this.aCurrentRoom.getEat());  
    }  
} //eat()
```

Désormais voici donc toutes les commandes qui existent dans le jeu stocké dans un attribut static aValidCommands[]:

```
// a constant array that will hold all valid command words  
private static final String aValidCommands[] = {"go", "help", "quit", "look", "eat"};
```

Exercice 7.16:

- Explication de showAll():

Cette procédure utilise une boucle for et permet d'afficher toutes les commandes qui sont stockées dans le tableau aValidCommands[] espacé avec de " | " en chaque commande.

```
/**  
 * print all possible command + " | "  
 * @method procedure showAll()  
 */  
public void showAll() {  
    for(String Command : this.aValidCommands){ //pour toutes les commandes qu'il existe dans aValidCommands  
        System.out.print(Command + " | ");  
    }  
    System.out.println();  
} //showAll()
```

- Explication de showCommands():

Permet de récupérer les commandes valide de showAll(), fait l'intermédiaire entre showAll() et printHelp().

```
/**  
 * allows you to display commands with this.aValidCommands object  
 * @method procedure showCommands()  
 */  
public void showCommands() {  
    this.aValidCommands.showAll();  
}
```

- Explication de printHelp():

Est appelé lorsque le joueur tape "help" et permet d'afficher toutes les commandes récupérées dans showCommands. Toutes les commandes valides qui existent.

```
/**  
 * printHelp to print if you write "help"  
 * call showCommands() for give you all possible commands  
 * @method printHelp()  
 */  
private void printHelp() {  
    System.out.println("You are lost. You are alone.\nYou wander around at the ship. \n");  
    System.out.println("Your command words are:");  
    this.aParser.showCommands();  
} //printHelp()
```

Exemple:

```
> help  
You are lost. You are alone.  
You wander around at the ship.  
  
Your command words are:  
go | help | quit | look | eat |  
>
```

Exercice 7.18/7.19:

Modification de toutes les méthodes relatives à l'affichage des Commandes:

CommandWords : Création du getter getCommandList():

Créer pour ne plus gérer l'affichage dans CommandWords mais seulement la production des commandes.

```
/**  
 * @return all commands that exist in aValidCommands() in a board  
 * @method getCommandList()  
 */  
public String[] getCommandList() {  
    return this.aValidCommands;  
} //getCommandList()
```

Parser : Création du getter getShowCommands():

Return toutes les commandes qui existent dans getCommandList() dans un tableau.

Je ne pas passer cette étape car aValidCommands(dans CommandWords, c'est ici que je stock toutes les commandes) est privé et il doit le rester.

```
/**  
 * @return all commands that exist in getCommandList() in a board  
 * call getCommandList()  
 * @method getShowCommands()  
 */  
public String[] getShowCommands() {  
    return this.aValidCommands.getCommandList();  
} //getShowCommands()
```

Game : Modification de printHelp():

Gère l'affichage des commandes stockées dans vCommands qui sont les commandes de getShowCommands()

Affiche toutes les commandes séparées de: " | " + Command + " | "

```
/**  
 * printHelp to print if you write "help"  
 * call showCommands() for give you all possible commands  
 * @method printHelp()  
 */  
private void printHelp(final Command pAppelCommandWords) {  
    System.out.println("You are lost. You are alone.\nYou wander around at the ship. \n");  
    System.out.println("Your command words are:");  
  
    String[] vCommands = this.aParser.getShowCommands(); //création d'une variable locale pour stocker toutes les commandes de getShowCommands() de type Parser  
  
    for(String Command : vCommands){ //pour toutes les commandes qu'il existe dans vCommands  
        System.out.print(" | " + Command + " | ");  
    }  
    System.out.println("\n");  
} //printHelp()
```

Exemple:

> **help**

You are lost. You are alone.

You wander around at the ship.

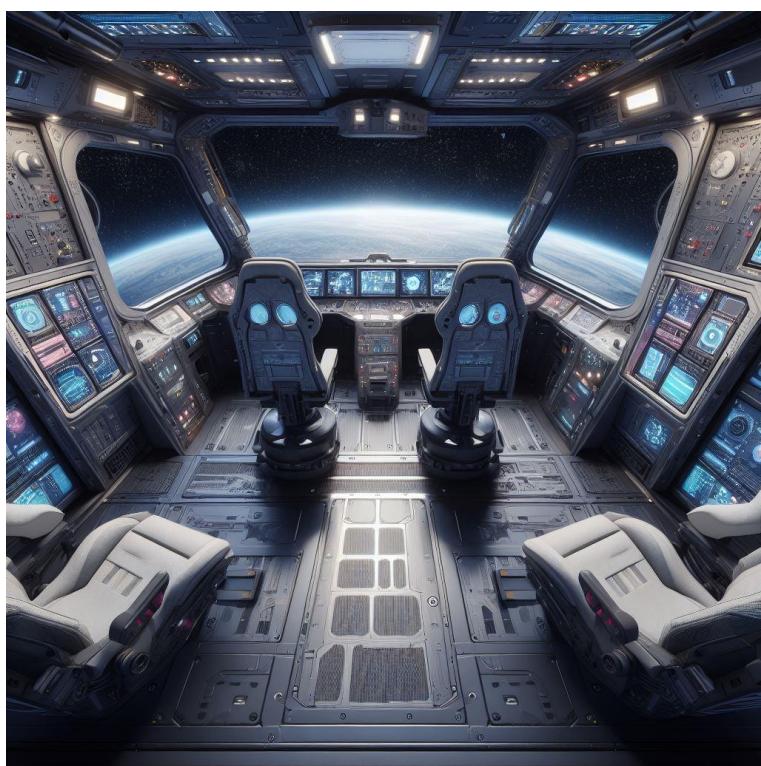
Your command words are:

| go | | help | | quit | | look | | eat |

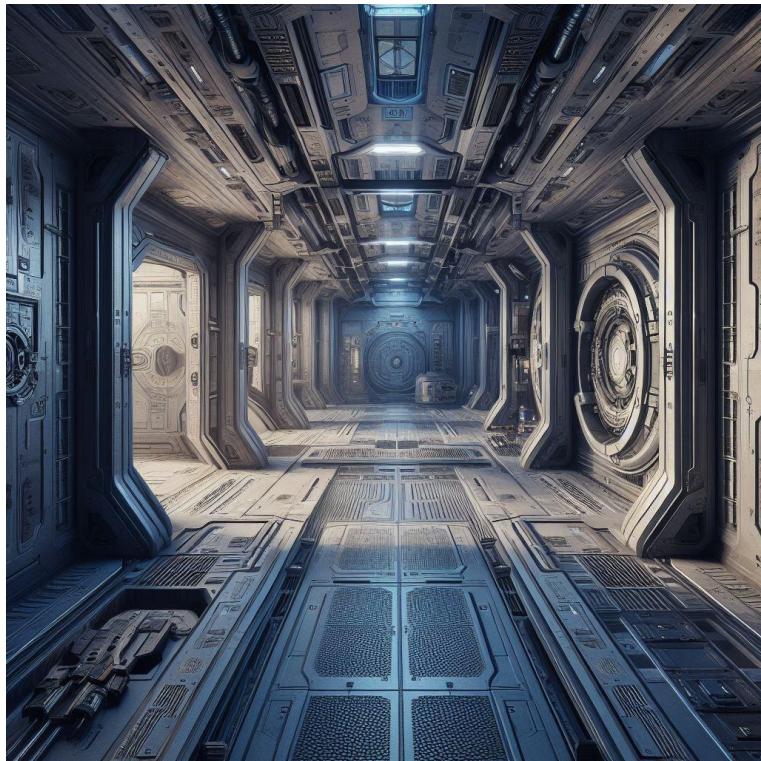
>

Images validées:

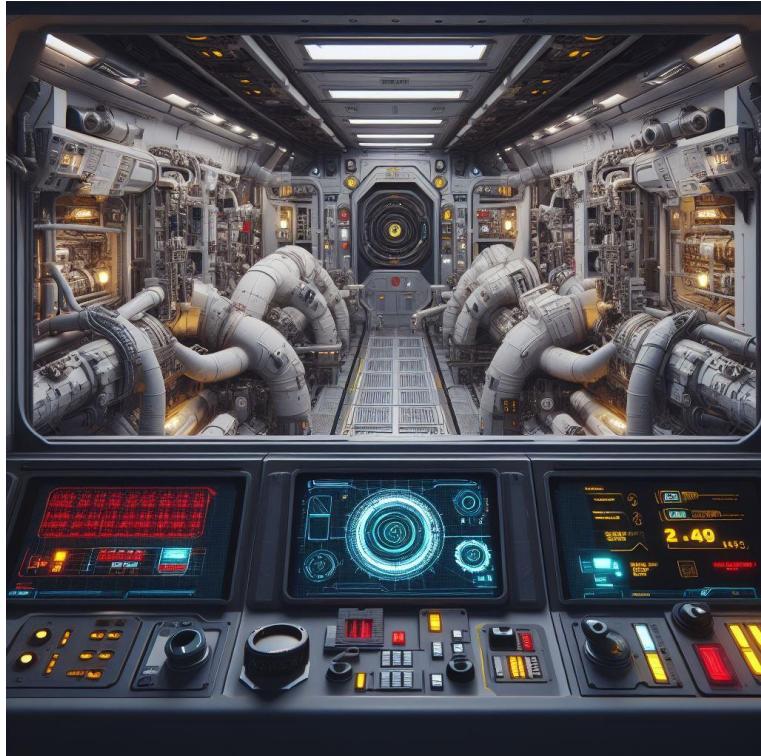
- Main Pilot:



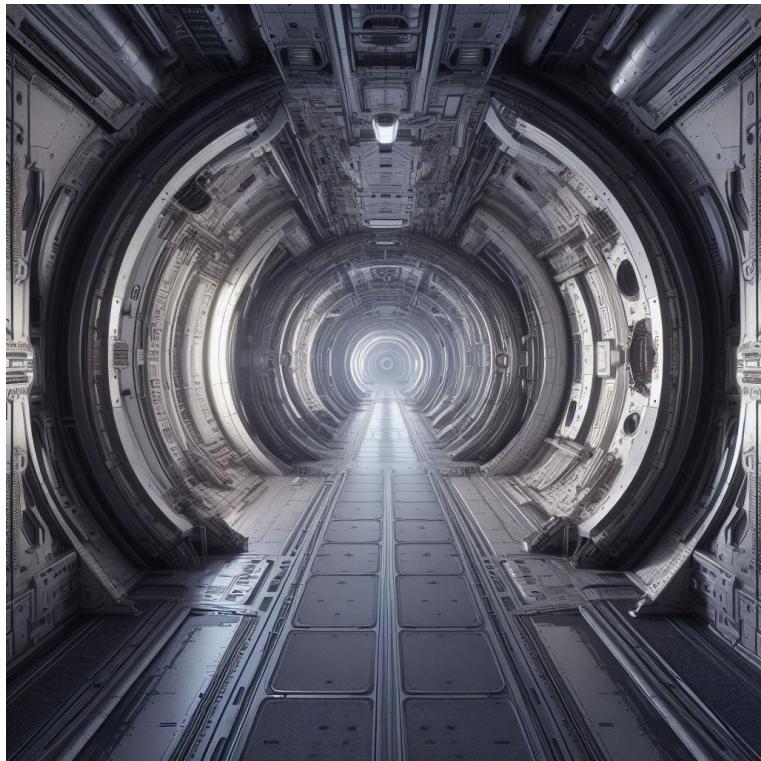
- Main Wing:



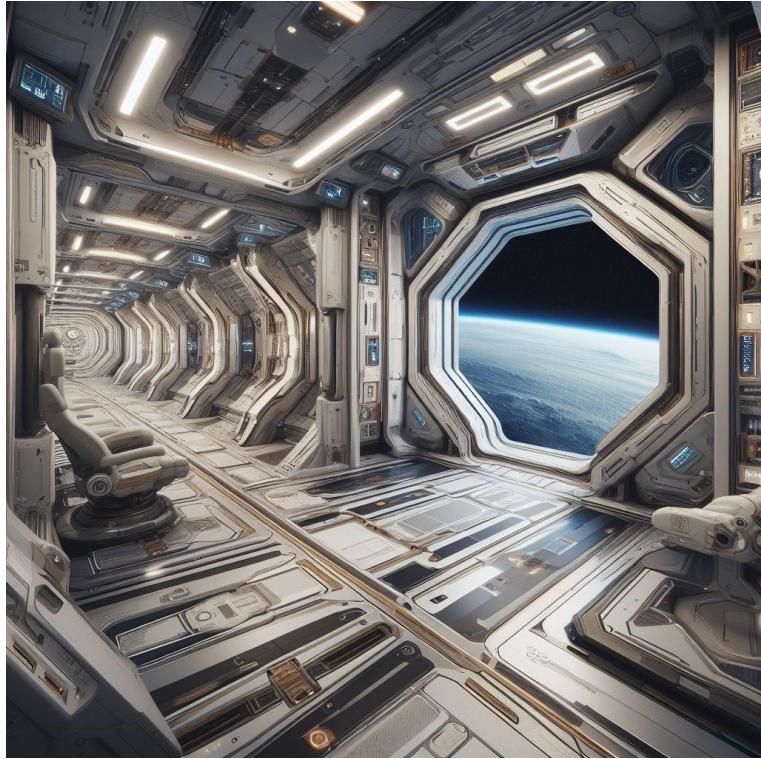
- Main Engine:



- Corridor Main Wing Main Engine:



- Corridor Left Wing:



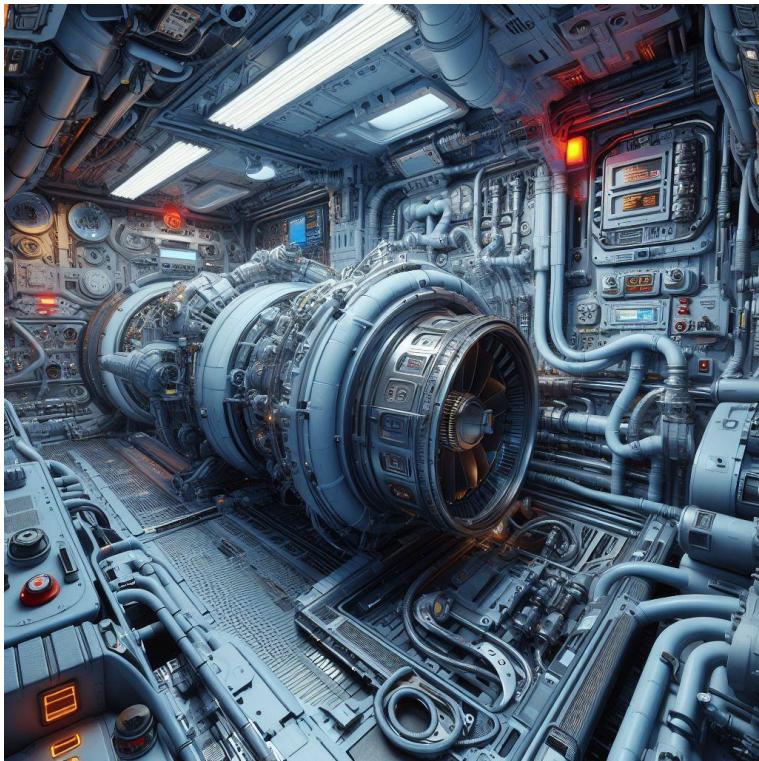
- Corridor Right Wing:



- Left Room Engine:



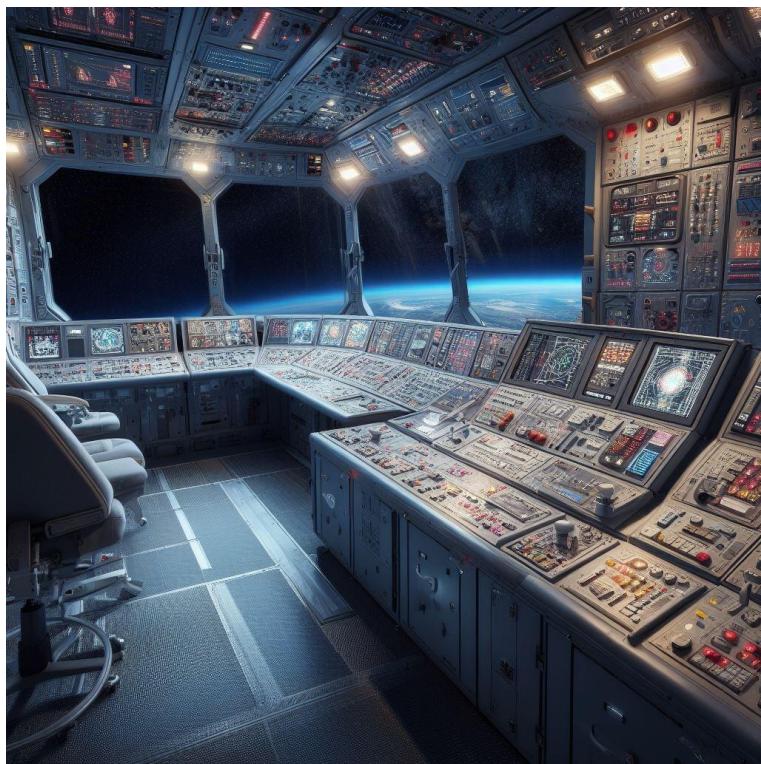
- Right Room Engine:



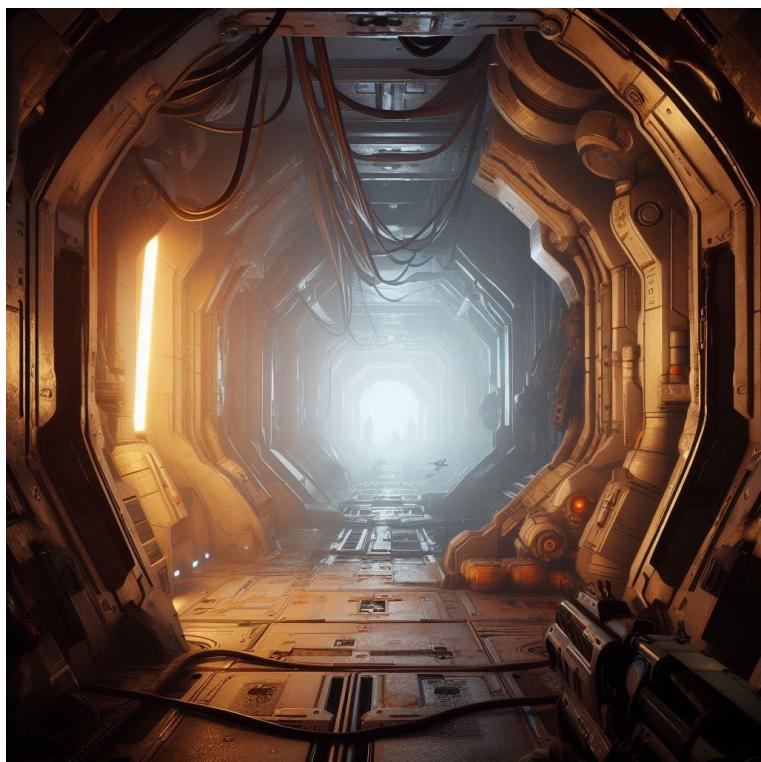
- Right Room Wing:



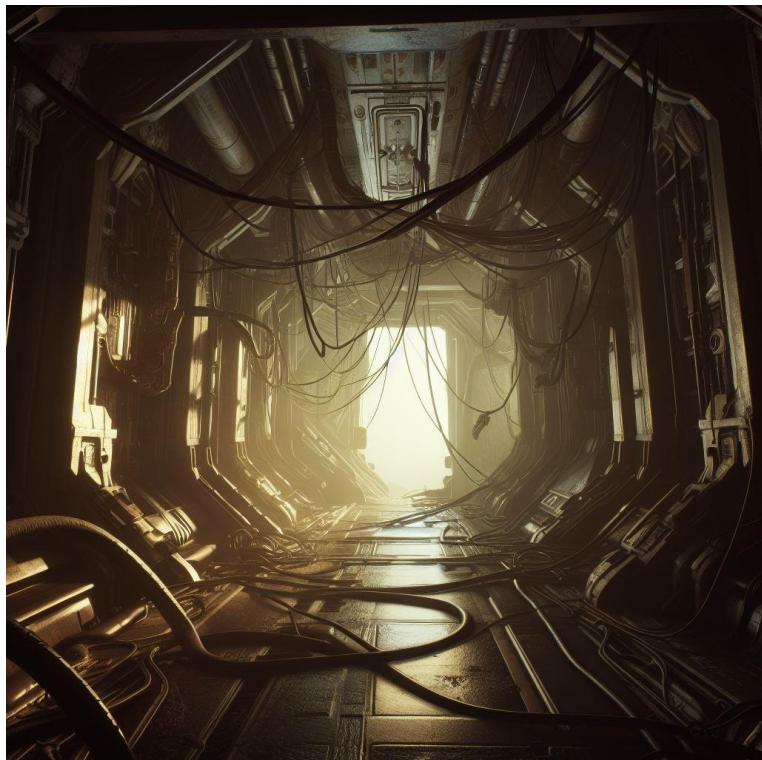
- Left Room Wing:



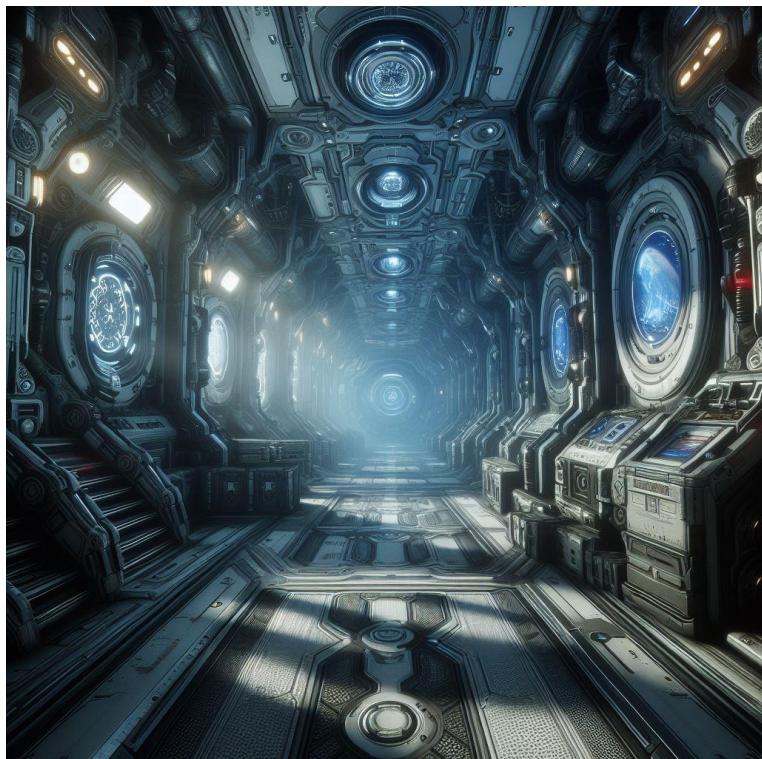
- Corridor Hidden Left:



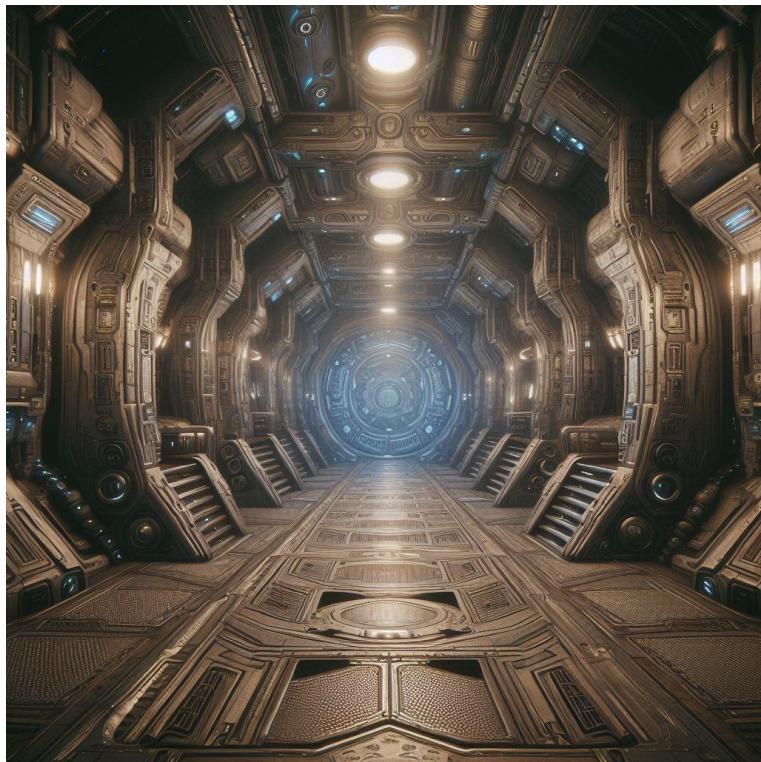
- Corridor Hidden Right:



- Corridor Engine Right:



- Corridor Engine Left:



L'affichage des images dans le jeu:

Ajout des Classes : UserInterface et GameEngine.

Pour l'ajout des images j'ai donc modifié la classe Room en ajoutant un attribut et donc un paramètre dans le constructeur.

Voici ma nouvelle classe Room :

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

/**
 * initialize all attribute
 * @Constructor Room() use for create new room
 * @param String pDescription intialize Room description
 */
public Room(final String pDescription, final String pImage) {
    this.aDescription = pDescription;
    this.aExits = new HashMap<String, Room>();
    this.aImageName = pImage;
} // Room()
```

J'ai aussi ajouté un getter qui retourne une image :

```
/**
 * @return aImageName
 * @method getimageName()
 */
public String getimageName() {
    return this.aImageName;
} //getimageName()
```

On utilise donc showImage() pour afficher notre image sur l'interface GUI.

```
/**  
 * Show an image file in the interface.  
 */  
public void showImage( final String pImageName )  
{  
    String vImagePath = "" + pImageName; // to change the directory  
    URL vImageURL = this.getClass().getClassLoader().getResource( vImagePath );  
    if ( vImageURL == null )  
        System.out.println( "Image not found : " + vImagePath );  
    else {  
        ImageIcon vIcon = new ImageIcon( vImageURL );  
        this.aImage.setIcon( vIcon );  
        this.aMyFrame.pack();  
    }  
} // showImage()
```

Puis j'appelle showImage ici pour afficher les images de chaque si celle-ci existe.

```
/**  
 * give: the current room in which the user is located, the room description, its different exits  
 * and displays the image of the current room  
 * it was created to avoid code duplication between printWelcome() and goRoom()  
 * @method procedure printLocationInfo()  
 */  
private void printLocationInfo() {  
    this.aGui.println(this.aCurrentRoom.getLongDescription());  
    if ( this.aCurrentRoom.getImageName() != null )  
        this.aGui.showImage( this.aCurrentRoom.getImageName() );  
} //printLocationInfo()
```

J'ai donc modifié la classe createRooms() :

Et j'ajoute pour Room() un nouveau paramètre, par ailleurs j'ai aussi ajouté la possibilité d'avoir un son pour chaque room donc j'ai utilisé la même méthode que pour les Images:

Voici par exemple à quoi ressemble la room vMainWing :

```
vMainWing = new Room("in main wing room", vLienImages + "mainwing.jpg" , vLienAudios + "mainwing.wav");
```

Et en image :



Puis maintenant je vais ajouter les boutons :

J'ai donc pour l'instant créé différent bouton grâce à la librairie JButton

```
//Btn Command
private JButton aBtnAudio;
private JButton aBtnQuit;
private JButton aBtnHelp;
private JButton aBtnEat;
private JButton aBtnLook;
//Btn Basic Direction
private JButton aBtnNorth;
private JButton aBtnSouth;
private JButton aBtnEast;
private JButton aBtnWest;
//Btn Other Direction
private JButton aBtnUp;
private JButton aBtnDown;
private JButton aBtnDownWest;
private JButton aBtnDownEast;
private JButton aBtnUpNorth;
private JButton aBtnUpEast;
private JButton aBtnUpWest;
private JButton aBtnDownSouth;
```

Cela fait beaucoup de boutons...

C'est donc dans createGUI qui permet de créer notre interface utilisateur, que je défini la valeur des boutons, de la fenêtre, la gestion des panels.

Par exemple pour le bouton **aBtnNorth**:

D'abord je créer le bouton et lui affecte son nom. « North »

Et grâce à setPerredSize lui donne une taille de « (X , Y) »

```
this.aBtnNorth = new JButton("North");
this.aBtnNorth.setPreferredSize(new Dimension(100,25));
```

Ajouter se bouton à un JPannel :

Ici je créer donc un Panel et grâce à setLayout défini le nombre de casse que j'aurai dans mon panel
ici 1casse = 1 bouton.

```
JPanel vPaneDirection = new JPanel();
vPaneDirection.setLayout(new GridLayout(3, 4) );
vPaneDirection.add(this.aBtnNorth, BorderLayout.NORTH );
```

Pour ma part j'ai mon panel principal qui donc affiche l'image, l'affiche du texte, la zone d'écriture,
mon panel pour les command, mon panel pour les directions :

```
JPanel vMainPanel = new JPanel();
vMainPanel.setLayout( new BorderLayout()); // ==> only five places
vMainPanel.add( this.aImage, BorderLayout.NORTH );//le placer au Nord
vMainPanel.add( vListScroller, BorderLayout.CENTER );//le placer au centre
vMainPanel.add( this.aEntryField, BorderLayout.SOUTH );//le placer au sud
vMainPanel.add( vPaneCommand, BorderLayout.EAST); //le placer à l'est
vMainPanel.add( vPaneDirection, BorderLayout.WEST); //le placer à l'ouest
```

Pour définir mon panel principal celui que j'affiche dans ma fenêtre :

```
this.aMyFrame.getContentPane().add( vMainPanel, BorderLayout.CENTER );
```

Puis après addActionListener(). Qui permet de préciser la classe qui va gérer l'événement utilisateur
de type ActionListener du bouton :

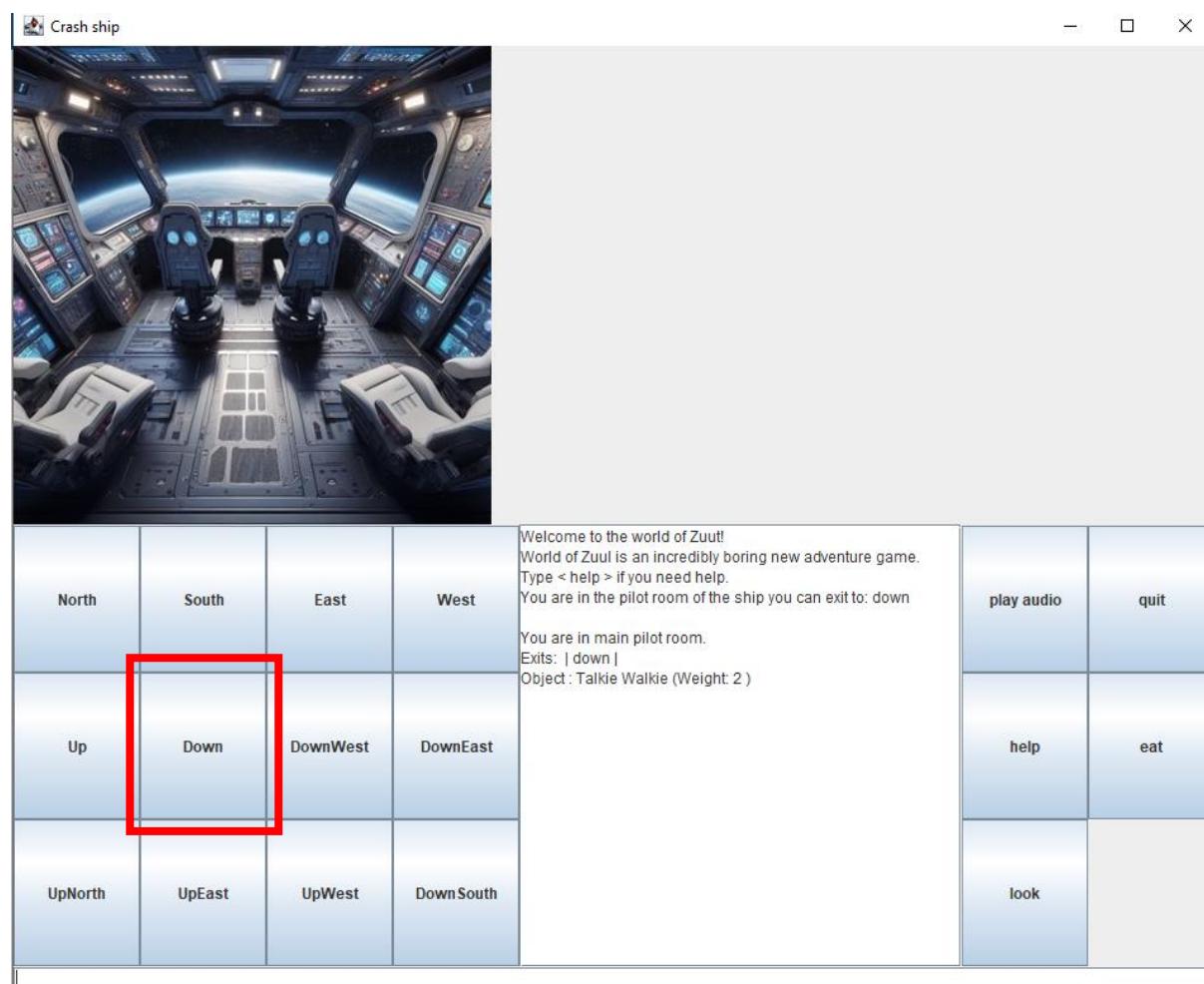
```
this.aBtnNorth.addActionListener( this );
```

Puis après dans actionPerformed() qui a pour paramètre pE et qui est de type ActionEvent :

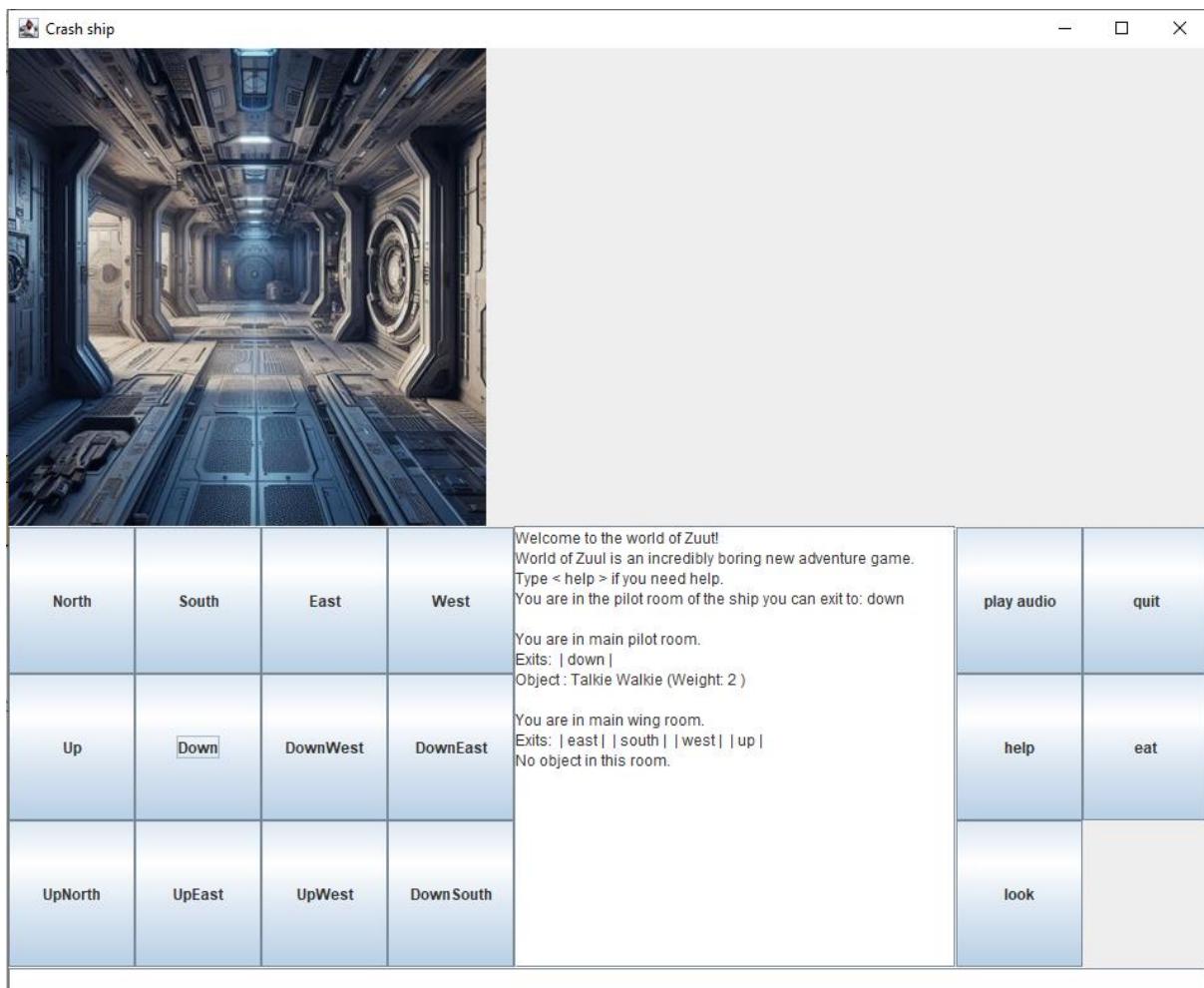
Lorsque je clique sur le bouton aBtnNorth alors j'appelle la fonction goRoom() situé dans GameEngine donc avec mon attribut aEngine créé de type GameEngine. Et applique la Command « go north ».

```
String vGo = "go";  
  
}else if(pE.getSource() == this.aBtnNorth ){  
    this.aEngine.goRoom(new Command(vGo, "north"));
```

Voici ma fenêtre :



Je clique sur Down :



On peut l'ajout des Item que je présente dans la partie d'après.

Avant cela je vais expliquer :

- ActionListener :

L'interface ActionListener fait partie du package java.awt.event. Elle est utilisée pour écouter les événements d'action, tels que les clics de bouton ou les actions de champ de texte. Pour écouter ces événements, la classe doit utiliser la méthode actionPerformed.

- addActionListener()

La méthode addActionListener() permet d'écouter une action à un composant qui génère des événements d'action. Par exemple, pour un bouton ou un champ de texte, vous pouvez ajouter un ActionListener pour écouter les événements d'action sur ce composant.

- actionPerformed()

La méthode actionPerformed(). C'est dans cette méthode qu'on donne l'ordre à exécuter par rapport à l'événement d'action. On doit implémenter cette méthode lorsque qu'on créer l'interface ActionListener.

- ActionEvent

ActionEvent est une classe qui représente un événement d'action. Lorsque qu'on cliquer sur un bouton, un objet ActionEvent est créé et est passé à la méthode actionPerformed. On peut donc utiliser cet objet pour obtenir des informations sur l'événement, comme la source de l'événement.

- getActionCommand()

La méthode getActionCommand() est utilisée pour obtenir la commande associée à l'action. Cela est utile lorsque plusieurs composants utilisent le même écouteur d'action, et quand on doit déterminer quelle action spécifique a déclenché l'événement.

- getSource()

La méthode getSource() de l'objet ActionEvent renvoie la source de l'événement, c'est-à-dire l'objet qui a déclenché l'événement. Cela peut être utile si on a plusieurs composants qui écoutent des actions avec le même écouteur, et on doit déterminer lequel a généré l'événement.

Exercice 7.20/7.21:

Pour cet exercice je dois créer un item de n'importe qu'elle room:
 Dans un premier temps voici ma classe Item :

```
/**  
 * Décrivez votre classe Item ici.  
 *  
 * @author Matthias Hautin  
 * @version 25/11/23  
 * this class allows you to create an item from any room  
 */  
public class Item  
{  
    private String aNameItem;  
    private int aPoids;  
    private String aLongDescription;  
  
    //constructor  
    public Item(final String pNameItem, final int pPoids, final String pLongDescription) {  
        this.aNameItem = pNameItem;  
        this.aPoids = pPoids;  
        this.aLongDescription = pLongDescription;  
    } //Item()
```

Avec les différents getters :

```
/**  
 * @return aNameItem  
 * @method getNameItem()  
 */  
public String getNameItem() {  
    return this.aNameItem;  
} //getImageName()  
  
/**  
 * @return aPoids  
 * @method getPoids()  
 */  
public int getPoids() {  
    return this.aPoids;  
} //getPoids()  
  
/**  
 * @return aLongDescription  
 * @method getLongDescrptionItem()  
 */  
public String getLongDescriptionItem() {  
    return this.aLongDescription;  
} //getLongDescription()  
  
} //Item()
```

Puis après ajout du setter dans room car c'est dans une room qu'il y a un Item:

```
/**  
 * define the items that are given to a room.  
 * @method setItem()  
 * @param pItem (string, int, string) of Item  
 */  
public void setItem(Item pItem) {  
    this.aItem = pItem;  
} //setItem()
```

J'appelle donc le setter lors de la création d'un Item dans GameEngine , createRooms() :

```
Item vTW = new Item("Talkie Walkie", 2);  
vMainPilot.setItem(vTW);
```

Pour l'affiche et la partie graphique il suffit de créer un getter qui retourne le nom ainsi que le poids de la room :

```
/**  
 * @return "Object : " + this.aItem.getNameItem() + " (Weight: " + this.aItem.getPoids() + " )" if exist an Item  
 * @method getItemString()  
 */  
public String getItemString() {  
    if(this.aItem != null) {  
        return "Object : " + this.aItem.getNameItem() + " (Weight: " + this.aItem.getPoids() + " )";  
    }else {  
        return "No object in this room.";  
    }  
} //getItemString
```

Maintenant comme pour le reste j'ajoute aussi une longue description à mon Item :

Voici le getter :

```
/**  
 * @return "Item " + this.aItem.getNameItem() + " description : " + this.aItem.getLongDescriptionItem();  
 * @method getItemLongDescription()  
 */  
public String getItemLongDescription() {  
    if(this.aItem != null) {  
        return "Item " + this.aItem.getNameItem() + " description : " + this.aItem.getLongDescriptionItem();  
    }else {  
        return "";  
    }  
} //getItemLongDescription
```

J'appelle donc le setter lors de la création d'un Item dans GameEngine , createRooms() :

Voici la modif :

```
Item vTW = new Item("Talkie Walkie", 2, "Le talkie Walkie vous permettra de toujours être connecté avec Lara pour v  
vMainPilot.setItem(vTW);
```

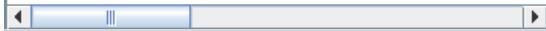
J'ajoute donc getItemString() et getItemLongDescription à mon getLongDescription() :

```
public String getLongDescription() {  
    return "You are " + getDescription() + ".\n" + getExitString() + "\n" + getItemString() + "\n" + getItemLongDescription();  
} //getLongDescription
```

Et donc voici au moment de l'affichage :

```
Welcome to the world of Zuul!
World of Zuul is an incredibly boring new adventure game.
Type < help > if you need help.
You are in the pilot room of the ship you can exit to: down

You are in main pilot room.
Exits: | down |
Object : Talkie Walkie (Weight: 2 )
Item Talkie Walkie description : Le talkie Walkie vous permetra de faire des appels radio.
```



Exercice 7.22 :

Dans cette partie je vais devoirs créer plusieurs Item dans une Room:

Pour cela dans la classe Room() l'attribut altem deviendra aItems et sera un HashMap comment pour aExits.

Création de l'attribut aItems :

```
private HashMap<String, Item> aItems;
```

Initialisation de aItems dans le Constructeur :

```
this.aItems = new HashMap<String, Item>();
```

Puis modification de setItem en setItems :

Il permettra comme l'ancien de créer un ou plusieurs Item dans une pièce.

Avec deux paramètre (String, Item) car l'attribut est un hashMap avec 2 paramètres qui sont un String et un Item :

```
/*
 * define the items that are given to a room.
 * @method setItem()
 * @param pItem (string, int, string) of Item
 */
public void setItems(final String pItemName, final Item pItem) {
    this.aItems.put(pitemName, pItem);
} //setItem()
```

Puis après je modifie getItemString() en utilisant un SringBuilder.

Si attribut aItems n'est pas vide alors pour tous les items dans la room j'écris un StringBuider avec le nom des objets et leur poids respectif. Ensuite je return le StringBuider en retirant les 2 dernier caractères avec substring.

Sinon j'écris qui n'y a pas d'objet dans cette room.

```
/*
 * @return "Object : " + this.aItem.getNameItem() + " (Weight: " + this.aItem.getPoids() + ")" if exist an Item
 * @method getItemString()
 */
public String getItemString() {
    if(!this.aItems.isEmpty()){
        StringBuilder vItemsString = new StringBuilder("Objects: ");
        for (Item vItem : this.aItems.values()) {
            vItemsString.append(vItem.getNameItem()).append(" (Weight: ").append(vItem.getPoids()).append("), ");
        }
        return vItemsString.substring(0, vItemsString.length() - 2); // Pour enlever la virgule finale
    } else {
        return "No objects in this room.";
    }
} //getItemString()
```

Ensuite j'ai aussi modifié getItemLongDescriptiton() de la même manière :

```
/**
 * @return "Item " + this.aItem.getNameItem() + " description : " + this.aItem.getLongDescriptiton(); if exist an Item
 * @method getItemLongDescription()
 */
public String getItemLongDescription() {
    StringBuilder vItemsDescription = new StringBuilder("Item descriptions:\n");
    if (!this.aItems.isEmpty()) {
        for (Item vItem : this.aItems.values()) {
            vItemsDescription.append(vItem.getNameItem()).append(": ").append(vItem.getLongDescriptiton()).append("\n");
        }
    } else {
        vItemsDescription.append("No objects in this room.");
    }
    return vItemsDescription.toString();
} //getItemLongDescription
```

Puis après je peux créer dans GameEngine plusieurs Items :

```
Item vTW = new Item("Talkie Walkie", 2, "Le talkie Walkie vous permettra de toujours être connecté avec Lara");
Item vTest = new Item("Test", 2, "un item en plus");
vMainPilot.setItems("Test", vTest);
vMainPilot.setItems("Talkie Walkie", vTW);
```

```
Welcome to the world of Zuul!
World of Zuul is an incredibly boring new adventure game.
Type < help > if you need help.
You are in the pilot room of the ship you can exit to: down

You are in main pilot room.
Exits: | down |
Objects: Test (Weight: 2), Talkie Walkie (Weight: 2)
Item descriptions:
Test: un item en plus
Talkie Walkie: Le talkie Walkie vous permettra de toujours être connecté avec Lara
```

Exercice 7.23/7.26:

Le but dans cette partie est de créer une commande back afin de revenir dans la room précédente.
Je vais donc utiliser Stack qui permet de faire des piles d'objet.

Pour cela tout d'abord je créer un attribut de type Stack<Room> dans GameEngine():

```
private Stack<Room> aPreviousRoom;
```

Après je l'initialise dans le constructeur :

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

Ensuite je modifie goRoom() en ajoutant la ligne 159 en utilisant la méthode push.

Cette ligne permet d'ajouter la pièce actuelle sur la pile avant que l'on change de room. Comme ça je peux garder toutes les pièces précédentes.

```
145 public void goRoom(final Command pCommand) {
146
147     if(! pCommand.hasSecondWord()) {//si le second mot n'existe pas
148         this.aGui.println("Go Where ?");
149         return;
150     }
151
152     String vDirection = pCommand.getSecondWord();//initialise une variable de type String
153     Room vNextRoom = this.aCurrentRoom.getExit(vDirection);//donne à la variable vNextRoom
154
155     if (vNextRoom == null) {//si la room suivante n'a pas de direction associée
156         this.aGui.println("There is no door !");
157         return;
158     } else {//si on change de room on donne la valeur de aCurrentRoom à notre nouvelle room
159         this.aPreviousRoom.push(this.aCurrentRoom); // mettre à jour la pièce précédente
160         this.aCurrentRoom = vNextRoom;
161         printLocationInfo();
162     }
163 } // goRoom()
```

Ensute j'ai créé une procédure goBack qui permet dans un premier temp de vérifier que l'utilisateur écrit seulement la commande « back » sans second mot.

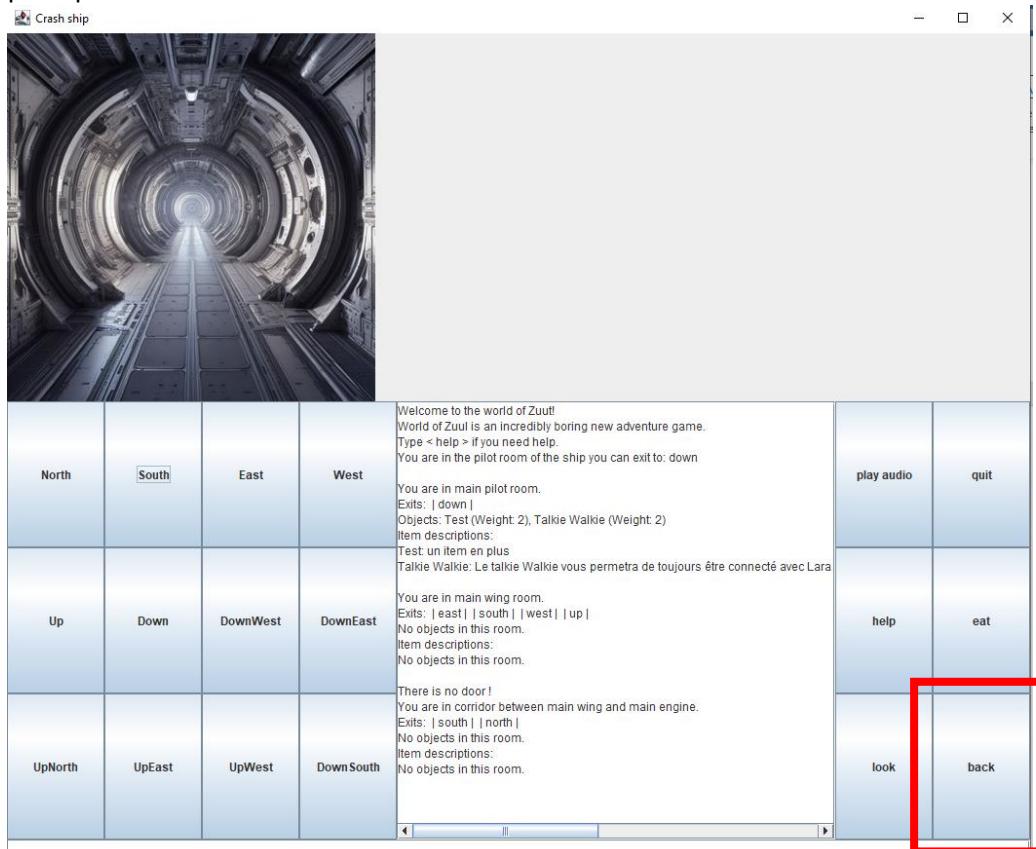
Puis elle donne une nouvelle valeur à aCurrentRoom grâce à pop, cette valeur est simplement la room qui est placée tout en haut de la pile cette room est aussi supprimée.

```
/*
 * This method allows the user to go to the previous room if they enter just "back"
 * @param pSecondMot to check if there is a second word
 * @method procedure goBack()
 */
private void goback(final Command pSecondMot) {
    if(pSecondMot.hasSecondWord() == true ) {//si l'utilisateur tape un second mot apres "back" (exemple: "back south")
        this.aGui.println("Just back.\n");
    } else if (!this.aPreviousRoom.isEmpty()) {
        this.aCurrentRoom = this.aPreviousRoom.pop(); //retire la room au dessus de la pile avec pop et donne la valaeur de cette room à aCurrentRoom
        printLocationInfo();
    } else {
        this.aGui.println("You can't go back!");
    }
} //goback()
```

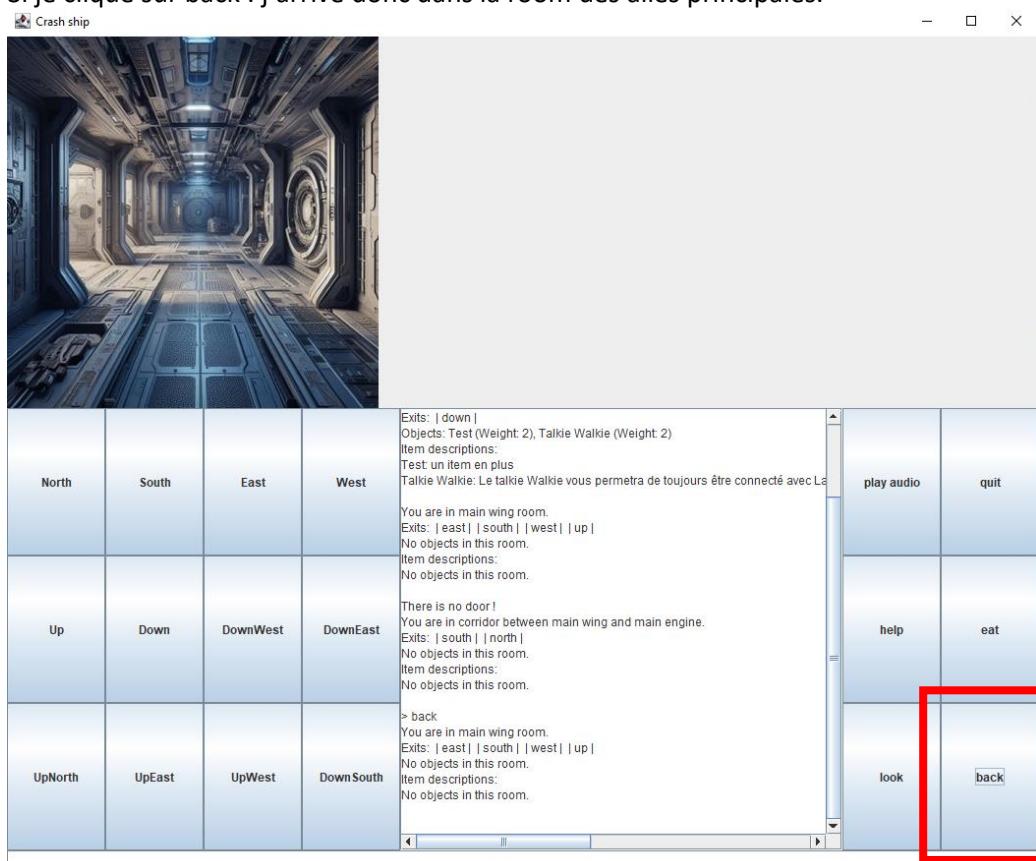
J'ai donc aussi créé un nouveau bouton comme pour les autres.

Utilisation de « back » :

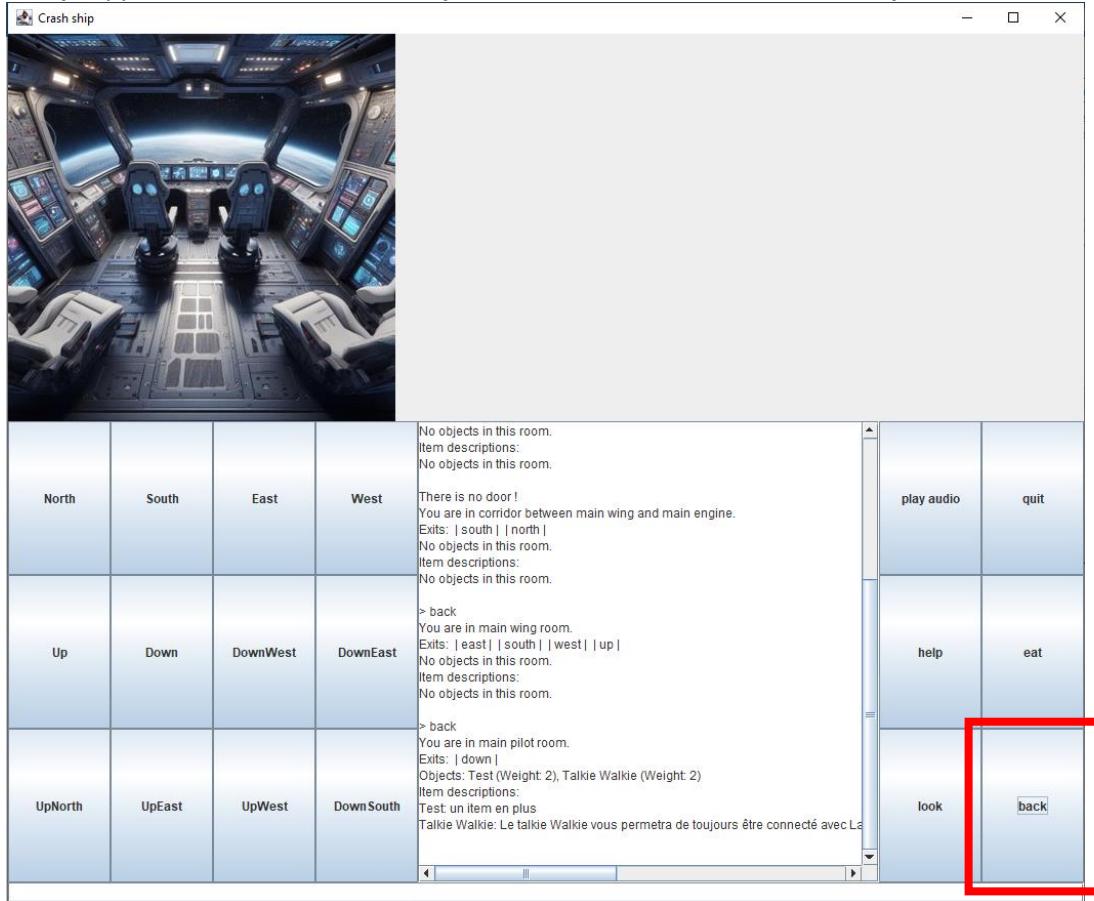
Je suis actuellement dans le couloir qui se situe entre la room principale des ailes et la room principale des moteurs.



Si je clique sur back : j'arrive donc dans la room des ailes principales.



Et si je appui à nouveau sur « back » j'arrive dans le ma room du début du jeu.



Si je veux de nouveau appuyer sur « back » alors voici le message :

```
> back
You are in main pilot room.
Exits: | down |
Objects: Test (Weight: 2), Talkie Walkie (Weight: 2)
Item descriptions:
Test: un item en plus
Talkie Walkie: Le talkie Walkie vous permettra de toujours être connecté avec La

> back
You can't go back!
```

Avant de passer à la suite je vais expliquer :

- Stack (pile) :

Une pile est une structure de données qui permet de contenir des éléments. Avec toujours le dernier ajouter en haut. Cela signifie aussi que le dernier élément ajouté à la pile est le premier à être retiré.

- push() :

Push permet d'ajouter un élément à une pile donc automatiquement l'élément ajouté se retrouve en haut de la pile.

- pop() :

Pop est une opération qui retire l'élément en haut de la pile. Après l'opération pop, l'élément situé en dessous de celui retiré devient le nouveau sommet de la pile.

- empty() :

Empty est une méthode qui vérifie si la pile est vide. Elle renvoie true si la pile ne contient aucun élément, sinon elle renvoie false.

- peek() :

Peek est une opération qui permet d'examiner l'élément en haut sans modifier la pile ou sans le retirer.

Exercice 7.28:

Dans cet exercice je vais créer une commande test qui permet de tester les différentes commandes créées dans le jeu via un fichier .tx

Voici donc ma classe test() qui se trouve dans GameEngine():

```
/*
 * test allows you to automatically test the game
 * @param pCommand is a command typed by the player
 */
private void test(final Command pFileName){
    if (!pFileName.hasSecondWord()){
        this.aGui.println( "tester quoi? (court, exploration, ideal)"); //si le joueur tape seulement test
        return; //sortir directement de la fonction
    }

    try { //faire toutes les commandes même si erreur
        File vTest = new File("./Commands/" + pFileName.getSecondWord() + ".txt"); //récupération du fichier pour le test
        Scanner vScanner = new Scanner(vTest); //scanner le fichier

        while(vScanner.hasNextLine()){ //tant qu'on détecte une ligne passer à la suivante
            interpretCommand(vScanner.nextLine()); //appelle de interpretCommand()
        }
        vScanner.close();
    }
    catch(final java.io.FileNotFoundException pE){ //si le fichier demandé n'existe pas
        this.aGui.println("il n'y a pas le fichier demandé.");
    }
} //test()
```

J'ai donc ensuite créé une commande test voici ce qui se produit :

Par exemple en tapant « *test court* » :

```
> test court
> go down
You are in main wing room.
Exits: | east| | south| | west| | up|
No objects in this room.

> look
You are in main wing room.
Exits: | east| | south| | west| | up|
No objects in this room.

> help
You are lost. You are alone.
You wander around at the ship.

Your command words are: | go | help | quit | look | eat | audio | back | test |
> go south
You are in corridor between main wing and main engine.
Exits: | south| | north|
No objects in this room.

> eat
You have eaten now and you are not hungry anymore.
```

Exercice 7.29:

Dans cet exercice je vais créer la classe Player qui permettra de créer un nouveau joueur qui a pour paramètres :

- Un Nom (String)
- La Room actuelle (Room)
- La Room précédente (Room)
- Un Poids Maximum (int)
- La Vie (int)

Un Player est aussi associé à une room c'est pour cela qu'il a un paramètre aCurrentRoom() et aPreviousRoom(). Cela veut dire que la gestion des Room se fera désormais depuis Player.

Donc désormais il n'y a plus de paramètre de type Room dans GameEngine. GameEngine ne fera plus que de l'affichage avec this.aGui.

```
public class Player
{
    private String      aNamePlayer;
    private Room        aCurrentRoom;
    private Stack<Room> aPreviousRoom;
    private int          aPoidsMax;
    private int          aVie;

    public Player(final String pNamePlayer, final Room pCurrentRoom , final int pPoidsMax, final int pVie) {
        this.aNamePlayer = pNamePlayer;
        this.aCurrentRoom = pCurrentRoom;
        this.aPoidsMax = pPoidsMax;
        this.aVie = pVie;
        this.aPreviousRoom = new Stack<Room>();
    } //Player()
}
```

Pour cette classe je vais créer beaucoup de getter pour récupérer chaque information que j'utiliserais plus tard. Pour obtenir la valeur actuelle d'un champ :

```
/** getter
 * return this.aPoidsMax
 */
public int getPoidsMax() {
    return this.aPoidsMax;
} //getPoidsMax()

/** getter
 * return this.aVie
 */
public int getVie() {
    return this.aVie;
} //getVie()

/** getter
 * return this.aCurrentRoom
 */
public Room getCurrentRoom() {
    return this.aCurrentRoom;
} //getCurrentRoom()

/** getter
 * return this.aPreviousRoom
 */
public Stack<Room> getPreviousRoom() {
    return this.aPreviousRoom;
} //getPreviousRoom()
```

Puis je créer un setter qui me permettra de modifier la valeur d'un champ ici c'est setCurrentRoom() pour modifier la room actuelle au début du jeu.

```
/** setter
 * @param pCurrentRoom
 */
public void setCurrentRoom(final Room pCurrentRoom) {
    this.aCurrentRoom = pCurrentRoom;
} //setCurrentRoom()
```

Ensuite je modifie printWelcome() pour afficher les différents paramètres d'un player au début du jeu.

```
private void printWelcome() {
    this.aGui.print("Welcome to the world of Zuut!\nWorld of Zuul is an incredibly boring new adventure game. \nType <
    this.aGui.println("\nYou are in the pilot room of the ship you can exit to: down");
    this.aGui.print("\n");
    this.aGui.println("----- Your name : " + this.aPlayer.getNamePlayer() + " -----");
    this.aGui.println("----- Your life : " + this.aPlayer.getVie() + " -----");
    this.aGui.println("----- Your Maximum possible weight : " + this.aPlayer.getPoidsMax() + " -----");
    printLocationInfo();
} //printWelcome()
```

Toujours dans l'optique de permettre à GameEngine de juste gérer l'affichage je modifie aussi d'autre fonction qui feront des appels de Player :

```
public void goRoom(final Command pCommand) {
    if(! pCommand.hasSecondWord()) {//si le second mot n'existe pas
        this.aGui.println("Go Where ?");
        return;
    }

    String vDirection = pCommand.getSecondWord(); //initialise une variable de type String qui est le second mot de notre
    Room vNextRoom = this.aPlayer.getCurrentRoom().getExit(vDirection); //donne à la variable vNextRoom la valeur de tout

    if (vNextRoom == null) {//si la room suivante n'a pas de direction associée
        this.aGui.println("There is no door !");
        return;
    } else {//si on change de room on donne la valeur de aCurrentRoom à notre nouvelle room
        this.aPlayer.getPreviousRoom().push(this.aPlayer.getCurrentRoom()); // mettre à jour la pièce précédente
        this.aPlayer.setCurrentRoom(vNextRoom);
        printLocationInfo();
    }
} // goRoom()
```

Dans goRoom() je modifie vNextRoom qui récupère désormais le this.aCurrentRoom de Player avec ses directions associées, pareille pour le this.aPreviousRoom qui se trouve maintenant dans Player.

Pour printLocationInfo() je dois également modifier la fonction :

J'appelle donc getCurrentRoom() qui se trouve dans Player c'est donc pour cela que j'utilise un attribut de type Player. Il y a aussi getLongDescription() qui se trouve dans Room et donc j'utilise getCurrentRoom() pour l'utiliser.

Pour afficher l'image cela change aussi et le principe reste le même.

```
private void printLocationInfo() {
    this.aGui.println(this.aPlayer.getCurrentRoom().getLongDescription());

    if ( this.aPlayer.getCurrentRoom().getImageName() != null )
        this.aGui.showImage(this.aPlayer.getCurrentRoom().getImageName());
```

J'ai aussi modifié look() qui utilise aPlayer pour afficher la description de la Room actuelle.

```
private void look(final Command pSecondMot) {
    if(pSecondMot.hasSecondWord() == true ) {//si l'utilisateur tape un second mot apres
        this.aGui.println("I don't know how to look at something in particular yet.\n");
    } else {
        this.aGui.println(this.aPlayer.getCurrentRoom().getLongDescription());
    }
} //look()
```

Aussi pour goback qui permet au joueur de revenir dans le room précédente grâce à Stack. Vérifie si la pile de la room précédentes n'est pas vide et alors si elle n'est pas vide déplace le joueur dans le room précédente et retire-le room de la pile grâce à pop. Puis appel printLocationInfo() qui affiche les infos de la nouvelle room. Si la room d'avant est vide donc inexistant alors j'écris que on ne peut pas faire back.

```
private void goback(final Command pSecondMot) {
    if(pSecondMot.hasSecondWord() == true ) {//si l'utilisateur tape un sec
        this.aGui.println("Just back.\n");

    } else if (!this.aPlayer.getPreviousRoom().isEmpty()) {
        this.aPlayer.setCurrentRoom(this.aPlayer.getPreviousRoom().pop());
        printLocationInfo();
    } else {
        this.aGui.println("You can't go back!");
    }
} //goback()
```

Pour finir et après je crois que j'ai dit l'essentiel.

Pour eat voici la nouvelle méthode :

```
private void eat(final Command pSecondMot) {
    if(pSecondMot.hasSecondWord() == true ) {//si l'utilisateur tape
        this.aGui.println("Just one thing at a time.\n");

    } else {
        this.aGui.println(this.aPlayer.getCurrentRoom().getEat());
    }
} //eat()
```

Elle fonctionne comme look() sauf que j'appelle getEat() qui affiche que l'utilisateur a mangé.

Exercice 7.30:

Ici le but sera de créer take et drop, qui sont 2 méthodes qui permettent de prendre seulement un Item(pour take) d'une Room et de le remettre(pour drop) dans une Room.

Voici alors ma première version de la fonction takeItem() dans GameEngine qui créer 2 variables locales :

- vItemName qui est un String de l'item
- vSuccessTake qui return un boolean par rapport à la réponse de take dans Player

```
private void takeItem(final Command pCommand) {
    if (pCommand.hasSecondWord()) {
        String vItemName = pCommand.getSecondWord();
        boolean vSuccessTake = this.aPlayer.take(vItemName);

        if (vSuccessTake) {
            this.aGui.println("You took the " + vItemName + ".");
            this.aGui.println("Your Maximum possible weight now is " + this.aPlayer.getPoidsMax() + ".");
        } else {
            this.aGui.println("Unable to take the " + vItemName + ".");
        }
    } else {
        this.aGui.println("Take what?");
    }
} //takeItem
```

Voici take() dans Player :

Take retourne donc donc un boolean. Cette fonction permet de vérifier si l'Item que l'on veut prendre existe et si le poids maximum que le joueur peut porter est inférieur à celui de l'Item qui veut prendre, avec getTotalWeight().

Take met aussi à jour le poids maximum du joueur quand il récupère l'Item. Si l'Item n'existe pas alors return false et si l'Item est trop lourd alors le remet dans le Room.

```
public boolean take(final String pItemName) {
    Item vItem = this.aCurrentRoom.removeItem(pItemName);

    if (vItem != null) {
        if (getTotalWeight() + vItem.getPoids() <= getPoidsMax()) {
            this.aItem = vItem;
            this.aPoidsMax = this.aPoidsMax - this.aItem.getPoids();
            return true;
        } else {
            // L'objet est trop lourd pour être porté.
            this.aCurrentRoom.setItems(pItemName, vItem); // Remettre l'objet dans la pièce.
        }
    }
    return false; // L'objet n'a pas été trouvé dans la pièce.
} //take()
```

Voici donc removeItem() qui permet de supprimer un Item d'une Room en fonction de son nom :

```
public Item removeItem(final String pItemName) {
    return this.aItems.remove(pItemName);
} //removeItem()
```

Voici setItems() qui remet l'item dans la room avec put si le poids de l'item est supérieur à ce que peut porter un joueur en fonction du Nom de l'Item de l'Item :

```
public void setItems(final String pItemName, final Item pItem) {
    this.aItems.put(pItemName, pItem);
} //setItem()
```

Et donc getTotalWeight() Qui return le poids de l'Item que le joueur veut take :

```
private int getTotalWeight() {
    if (this.aItem != null) {
        return this.aItem.getPoids();
    } else {
        return 0;
    }
} //getTotalWeight()
```

Maintenant voici dropItem() dans GameEngine qui permet de remettre un Item dans Une salle et de le supprimer de player, cette méthode fonctionne comme takeItem() finalement

```
private void dropItem(final Command pCommand) {
    if (pCommand.hasSecondWord()) {
        String vItemName = pCommand.getSecondWord();
        boolean vSuccessDrop = this.aPlayer.drop(vItemName);

        if (vSuccessDrop) {
            this.aGui.println("You drop the " + vItemName + ".");
            this.aGui.println("Your Maximum possible weight now is " + this.aPlayer.getPoidsMax() + ".");
        } else {
            this.aGui.println("Unable to drop the " + vItemName + ".");
        }
    } else {
        this.aGui.println("Drop What?");
    }
} //dropItem
```

Voici drop() dans Player qui permet de drop et Item de Player et le mettre dans une room. Pour cela je vérifie que l'item existe et qu'il est égale au String passé en paramètre. Je le remet dans la room actuelle grâce à setItems et donc ajoute au poids maximum que peut porter le joueur le points de l'Item que le joueur vient de drop.

```
public boolean drop(final String pItemName) {
    if (this.aItem != null && this.aItem.getNameItem().equalsIgnoreCase(pItemName)) {
        this.aCurrentRoom.setItems(pItemName, this.aItem);
        this.aPoidsMax = this.aPoidsMax + this.aItem.getPoids();
        this.aItem = null;
        return true;
    }

    return false; // L'objet n'a pas été trouvé dans l'inventaire du joueur.
} //drop()
```

Exercice 7.31.1/7.32/7.33:

Le but de cet exercice et de pouvoir prendre et poser plusieurs Items dans une room et un player. En créant une classe ItemList qui est la liste de tout les Items du Jeu. Le but est donc que la gestion des Item se fasse dans ItemList et plus dans Player et Room.

Donc dans ma Classe ItemList j'ai seulement un attribut HashMap qui comporte un String et un Item.

```
public class ItemList
{
    private HashMap<String, Item> aItemList;

    public ItemList() {
        this.aItemList = new HashMap<String, Item>();
    } //ItemList()
```

J'ai donc déplacé mon takeItem() et removeItem() dans ItemList :

```
public void takeItem(final String pItemName, final Item pItem) {
    this.aItemList.put(pItemName, pItem);
} //takeItem()

public void removeItem(final String pItemName, final Item pItem) {
    this.aItemList.remove(pItemName, pItem);
} //removeItem
```

J'ai aussi décalé mes deux affichages dans ItemList getItemString() et getItemLongDescription().

J'ai aussi modifié mon takeItem et dropItem dans player :

Ici j'utilise un appel à takeItem() de ItemList, et je mets toujours à jour le poidsMax que peut porter un joueur, je fais la même chose pour dropItem sauf que j'appelle removeItem() de ItemList.

```
public void takeItem(final String pItemName, final Item pItem) {
    this.aInventoryItems.takeItem(pItemName, pItem);
    this.aPoidsMax = this.aPoidsMax - pItem.getPoids();
}

public void dropItem(final String pItemName, final Item pItem) {
    this.aInventoryItems.removeItem(pItemName, pItem);
    this.aPoidsMax = this.aPoidsMax + pItem.getPoids();
}
```

Et ensuite voici la modification dans GameEngine() de take() et de drop() :

Ici take permet toujours de prendre un Item d'une Room et de le mettre dans l'inventaire du joueur.

Donc je vérifie que la commande tapée est bien un secondMot si on arrête. Puis je créer 2 variables locales : la première permet de récupérer le nom de l'item que le joueur écrit. Et la deuxième permet de récupérer l'item de la pièce actuelle.

Puis je vérifie si l'item existe et si le poids de l'item est inférieure à ce que peut porter un joueur. Maintenant après avoir fait toutes ces vérifications je peux ajouter l'item à l'inventaire du joueur et le retirer de la room actuelle. Et je print que j'ai pris l'item et je mets à jour le poids actuel que le joueur peut porter.

```
private void take(final Command pCommand) {
    if (!pCommand.hasSecondWord()) {
        this.aGui.println("Take what?"); //si la commande est incomplète sans second mot
        return;
    } else {
        String vItemName = pCommand.getSecondWord(); // Obtient le nom de l'objet à partir de la commande
        Item vItem = this.aPlayer.getCurrentRoom().getItem(vItemName); //Récupère l'objet de la pièce actuelle

        if(vItem != null ) {
            if(this.aPlayer.getTotalWeight(vItem)) { // Vérifie si le poids de l'objet peut être ajouté à l'inven
                this.aPlayer.takeItem(vItemName, vItem); // ajoute l'item à l'inventaire du joueur
                this.aPlayer.getCurrentRoom().getItemList().removeItem(vItemName, vItem); // retire l'item de la
                this.aGui.println("You took the " + vItemName + "."); //print l'item
                this.aGui.println("Your Maximum possible weight now is " + this.aPlayer.getPoidsMax() + ".\n");
            } else {
                this.aGui.println("The item is too heavy."); //print si l'objet est supérieur à ce que peut port
            }
        } else {
            this.aGui.println("Unable to take the " + vItemName + ".\n"); //print si l'item n'existe pas
        }
    }
} //takeItem()
```

Pour drop c'est quasiment la même chose mais je remets l'item dans la room, et je supprime l'item d'inventaire de Player.

```
private void drop(final Command pCommand) {
    if (!pCommand.hasSecondWord()) {
        this.aGui.println("Drop what?");
        return;
    } else {
        String vItemName = pCommand.getSecondWord();
        Item vItem = this.aPlayer.getItem(vItemName);

        if (vItem != null) {
            this.aPlayer.getCurrentRoom().getItemList().takeItem(vItemName, vItem);
            this.aPlayer.dropItem(vItemName, vItem);
            this.aGui.println("You dropped the " + vItemName + ".");
            this.aGui.println("Your Maximum possible weight now is " + this.aPlayer.getPoidsMax() + ".\n");
        } else {
            this.aGui.println("Error: The " + vItemName + " is not in your inventory.\n");
        }
    }
} //drop()
```

Exercice 7.34/7.34.1:

Ici le but est de créer un Item qui permet au joueur d'avoir plus de vie. Pour cela dans mon jeu il doit d'abord take() l'item puis après le manger avec eat(). On veut manger un « Cookie ». Et ajouter 20 de vie pour Cookie.

Premièrement je créer une fonction qui permet de donner de la vie dans Player :

donneVie() :Cette procédure permet simplement d'ajouter de la vie au joueur en utilisant un int donné en paramètre.

```
public void donneVie(final int pVie) {
    this.aVie += pVie;
} //donneVie()
```

Après je modifie eat() :

Pour commencer je vérifie qu'il y a un secondMot sinon je print. Ensuite je vérifie que le second mot et égale à Cookie sinon je print que le joueur ne peut pas manger ça. Et pour finir je vérifie dans l'inventaire de player qu'il y a bien l'item Cookie sinon je print que l'item n'est pas présent dans l'inventaire. Si Cookie et dans l'inventaire à je drop Cookie de l'inventaire il sera supprimer et j'ajoute au joueur 20 de vie. Et je pris en mettant à jour pour le voir visuellement.

```
private void eat(final Command pSecondMot) {
    if (pSecondMot.hasSecondWord()) {
        String vItemName = pSecondMot.getSecondWord();
        Item vItem = this.aPlayer.getItemInInventory(vItemName);

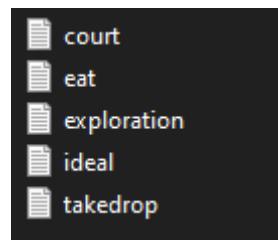
        // Check if the item is in the player's inventory
        if (vItem != null && vItem.getNameItem().equals("Cookie")) {
            // Check if the player has the item in the inventory
            if (this.aPlayer.getItemInInventory("Cookie") != null) {
                // Remove the Cookie from the player's inventory
                this.aPlayer.dropItem("Cookie", vItem);
                // Update player's health
                this.aPlayer.donneVie(20);
                this.aGui.println("You have eaten the Cookie. Your health is now " + this.aPlayer.getVie() + ".");
            } else {
                this.aGui.println("You don't have a " + vItemName + " in your inventory.");
            }
        } else {
            this.aGui.println("You can't eat that!");
        }
    } else {
        this.aGui.println("Just one thing at a time.");
    }
} // eat()
```

Voici mon jeu maintenant quand je le lance :

```
Welcome to the world of Zuut!
World of Zuul is an incredibly boring new adventure game.
Type < help > if you need help.
You are in the pilot room of the ship you can exit to: down

----- Your name : Mike -----
----- Your life : 100 -----
----- Your Maximum possible weight : 20 -----
You are in main pilot room.
Exits: | down |
Items: Cookie (Weight: 1), Talkie-Walkie (Weight: 2), Test (Weight: 15), Masse (We
Item descriptions:
Cookie: Un cookie magique qui ajoute de la vie
Talkie-Walkie: Le talkie Walkie vous permettra de toujours être connecté avec Lara
Test: un item en plus
Masse: Une masse lourde
```

J'ai donc créé des nouveaux tests qui se situent dans le dossier Commands pour take et drop et eat :



Si j'écris test :

```
> test
tester quoi? (court, exploration, ideal, takedrop, eat)
```

Exercice 7.42:

Le but de cet exercice est de créer un temps limite de jeu qui sera pour moi le minimum de salle que le joueur doit faire pour gagner.

Pour faire le temps je vais donc me servir des salles pour faire un compteur, plus précisément quand le joueur passe d'une salle à une autre.

Pour cela j'ai donc créé deux nouveaux attributs dans ma classe GameEngine() de type int :

- aMaxMoves : (int) pour savoir le temps max
- aCurrentMoves : (int) pour savoir ou en est le joueur

D'abord, j'ai initialisé aMaxMoves dans mon constructeur à 14. Car je dois parcourir 13 salles en tous.

```
public GameEngine() {
    this.aParser = new Parser();
    this.createRoomsAndPlayers();
    this.aIsAudioEnabled = false; //l'audio est désactivé par défaut
    this.aMaxMoves = 14;
    this.aCurrentMoves = 0;
}
```

Ensuite, je crée CheckTimeLimit() qui permet de savoir quand le joueur est à une salle de perdre est qui vérifie si aCurrentMoves = aMaxMoves auxquels cas le joueur perds :

```
private void checkTimeLimit() {
    if(this.aCurrentMoves == this.aMaxMoves - 1) {
        this.aGui.println("Il vous reste seulement un seul Mouvement. Attention vous avez atteint votre limite de temps");
    }

    if (this.aCurrentMoves == this.aMaxMoves) {
        System.out.println("Temps écoulé ! Vous avez dépassé le nombre maximal de mouvements. Fin du jeu.");
        this.endGame();
        return;
    }
} //checkTimeLimit()
```

J'ai aussi modifié goRoom() qui incrémente de 1 l'attribut aCurrentMves à chaque fois que le joueur change de room et j'appelle CheckTimeLimit() :

```
public void goRoom(final Command pCommand) {
    if(! pCommand.hasSecondWord()) {//si le second mot n'existe pas
        this.aGui.println("Go Where ?");
        return;
    }

    String vDirection = pCommand.getSecondWord(); //initialise une variable de type String qui est le second mot de Room
    vNextRoom = this.aPlayer.getCurrentRoom().getExit(vDirection); //donne à la variable vNextRoom la valeur de

    if (vNextRoom == null) {//si la room suivante n'a pas de direction associée
        this.aGui.println("There is no door !");
        return;
    } else { //si on change de room on donne la valeur de aCurrentRoom à notre nouvelle room
        this.aCurrentMoves++; // Incrémente le compteur de mouvements à chaque commande entrée
        checkTimeLimit(); // Vérifie si le joueur a dépassé la limite de temps
        this.aPlayer.getCurrentRoom().stopAudio(); //arrete l'audio de la pièce actuelle avant de change de piece
        this.aPlayer.getPreviousRoom().push(this.aPlayer.getCurrentRoom()); // mettre à jour la pièce précédente
        this.aPlayer.setCurrentRoom(vNextRoom);
        printLocationInfo();
    }
} // goRoom()
```

Exercice 7.42.2:

Pour cet exercice j'ai seulement ajouté une minimap que j'ai mise en haut à droite de la frame :

Dans UserInterface() :

Création d'un nouvel attribut qui est un JLabel donc une image :

```
private JLabel aImageMap;
```

Ensute je l'initialise et je modifie la taille :

```
// Créez un nouveau JLabel pour la minimap
this.aImageMap = new JLabel();
this.aImageMap.setPreferredSize(new Dimension(469, 350));
```

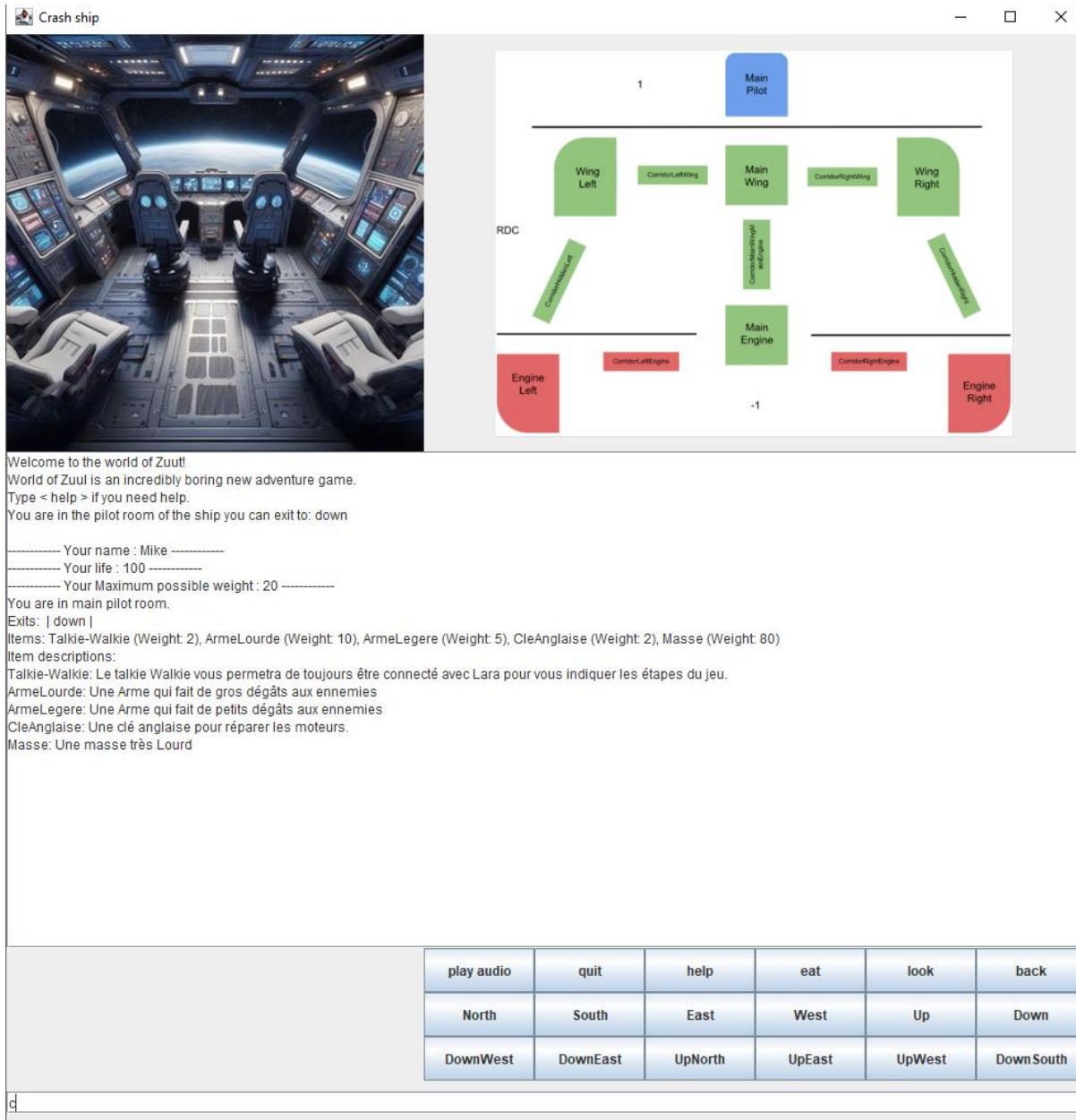
Ensute je créer une classe pour afficher l'image :

```
public void showMinimapImage(String imagePath) {
    URL imageURL = this.getClass().getClassLoader().getResource(imagePath);
    if (imageURL == null) {
        System.out.println("Image not found: " + imagePath);
    } else {
        ImageIcon icon = new ImageIcon(imageURL);
        this.aImageMap.setIcon(icon);
    }
}
```

J'appelle cette classe sur l'emplacement sur ma machine physqie dans le constructeur:

```
public UserInterface(final GameEngine pGameEngine ) {
    this.aEngine = pGameEngine;
    this.createGUI();
    this.aBool = true;
    showMinimapImage("./Images/Map.png");
} // UserInterface(.)
```

Voici à quoi ressemble le Jeu au démarrage :



Exercice 7.43:

Le but de cet exercice est de créer une fonction trapdoor qui lorsque le joueur fait back, il ne peut pas revenir en arrière si la salle qui le précède est équipée d'une trapdoor.

Voici donc ma classe trapdoor :

Trapdoor crée une variable locale qui est la pièce précédente, puis vérifie si il existe une room précédente et si il y a une trapdoor dans cette pièce, s'il y en a une on reste dans la même pièce et on affiche un message sinon le joueur est déplacé dans la room précédente et on retire la pièce du dessus de la pile (stack).

```
private void trapdoor() {
    Room vPreviousRoom = this.aPlayer.getPreviousRoom().peek(); // Récupère la pièce précédente

    if (vPreviousRoom != null && !vPreviousRoom.getTrapDoor()) { // Vérifie si la pièce précédente a une trapdoor
        this.aPlayer.setCurrentRoom(this.aPlayer.getPreviousRoom().pop()); // Déplace le joueur dans la pièce précédente
        printLocationInfo(); // Affiche les informations de la room
    } else {
        aGui.println("You can't go back through the trapdoor!");
    }
}
```

Ensuite j'appelle cette fonction dans goback() qui vérifie si il y a une salle précédente et si oui alors exécute trapdoor() :

```
private void goback(final Command pSecondMot) {
    if (pSecondMot.hasSecondWord() == true) { // Vérifie si un deuxième mot est présent
        this.aGui.println("Just back.\n");

    } else if (!this.aPlayer.getPreviousRoom().isEmpty()) { // Si la pièce précédente n'est pas vide
        trapdoor();
    } else {
        this.aGui.println("You can't go back!");
    }
} // goback()
```

Pour finir j'ai mis à jour le test ideal :



ideal - Notepad

```
File Edit Format View Help
take Talkie-Walkie
take CleAnglaise
take ArmeLourde
go down
go east
go east
attaque ArmeLourde
go south
go downsouth
repare motor1
go west
go upwest
take Cookie
eat Cookie
go north
go north
go west
go west
attaque ArmeLourde
attaque ArmeLourde
go south
take MegaCookie
eat MegaCookie
go downsouth
attaque ArmeLourde
attaque ArmeLourde
attaque ArmeLourde
attaque ArmeLourde
repare motor2
```

Liste anti-plagiat :

J'ai emprunté la partie pour faire le son à :

<https://www.delftstack.com/fr/howto/java/play-sound-in-java/>

Ainsi que les différentes ia pour faire le son et les images :

- Images :

<https://www.blogdumoderateur.com/tools/bing-image-creator/>

- Son :

<https://murf.ai/studio/project/2/P017048208829567LJ?workspaceId=WORKSPACEID017048208816586XO&>