Stranded

Rapport d’avancement

Florian NELCHA
A3P 2024/2025 - Groupe J3

Table des matières

I.A) Auteur

J’ai quasi intégralement réalisé ce projet, comprenant la conception du jeu, le scénario, ainsi que le développement en Java; le tout, bien évidemment, avec l’aide de Denis BUREAU et des intervenants de l’ESIEE Paris.

I.B) Thème

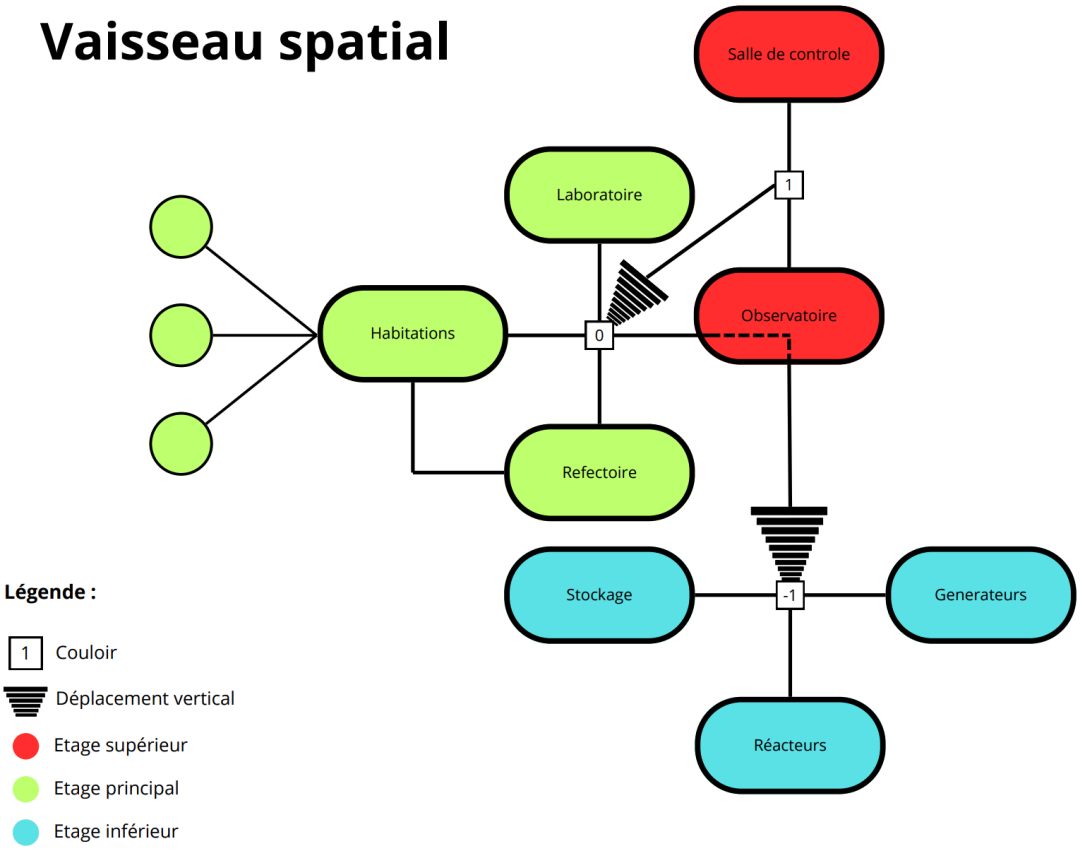
Dans un immense vaisseau spatial abandonné en orbite près d’une planète inconnue, une jeune exploratrice doit trouver et assembler les pièces nécessaires pour réparer les réacteurs et reprendre le contrôle avant qu’il ne soit attiré par la gravité de la planète.

I.C) Résumé du scénario

Le joueur incarne un personnage qui se réveille seul à bord d'un vaisseau spatial après un mystérieux incident. L’endroit semble désert, et des indices disséminés dans différentes pièces laissent entendre qu'une catastrophe s'est produite. Le joueur doit donc explorer le vaisseau pour découvrir la nature de l'incident.

L'objectif final est de réparer les systèmes essentiels et les réacteurs du vaisseau, et d’en reprendre le contrôle à temps avant qu’il ne soit attiré par la gravité de la planète à proximité.

I.D) Plan



Description de la map

- Vaisseau spatial abandonné de 3 étages
- Les couloirs permettent de monter ou descendre, ou bien de se rendre vers d'autres salles.
- L'étage supérieur du vaisseau se trouve en réalité juste au dessus de l'étage principal (Compliqué à représenter en 2d)
- La salle des réacteurs est verrouillée par code au début. La serrure de la chambre d'Ana l'est également (clé).
- **Le joueur apparaît dans la chambre du personnage principal. Il y en a deux autres qui sont celles des autres membres de l'équipage.**

I.E) Scénario détaillé (peut encore changer)

Contexte
Eden, ingénieure spécialisée en systèmes de propulsion, se réveille seule à bord du vaisseau “Voidwalker”, un vaisseau de recherche interstellaire, après un mystérieux incident. L’atmosphère est pesante : des alarmes de faible intensité clignotent, le silence est oppressant, et certaines sections du vaisseau sont plongées dans l’obscurité et d’autres, totalement inaccessibles.
Le dernier souvenir d’Eden est flou : des flashs d’une réunion d’urgence avec l’équipage, suivis d’un sentiment d’urgence et de chaos. Pourquoi est-elle la seule à bord ? Où sont passés Nathan et Ana, les deux autres membres de l’équipage ? Et surtout, que s’est-il réellement passé pour que le vaisseau soit à la dérive près d’une planète inconnue, gravitant dangereusement vers sa destruction ?

Les raisons de l’incident
Le Voidwalker était en mission scientifique pour étudier des anomalies gravitationnelles dans un secteur reculé de la galaxie. L’objectif de la mission était de récolter des données sur une étrange entité cosmique : une sorte de micro-trou noir en formation. Ce phénomène, vite devenu instable, dégageait une énergie titanesque mais également des interférences électromagnétiques qui ont commencé à affecter les systèmes du vaisseau : une surcharge du réacteur principal a conduit à un début d’incendie et à une panne critique. De plus, l’exposition prolongée à ce phénomène à également eu des effets étranges sur les membres d’équipage, notamment des troubles cognitifs et des hallucinations.

Au fil de la partie, en reconstituant petit à petit les événements, on finit par découvrir ce qu’il s’est passé.
Il semblerait que Nathan ait ordonné une tentative d’évacuation lorsque la situation devenait critique, mais des dysfonctionnements dans le système des capsules d’évacuation auraient épargné Ana et Eden de l’éjection chaotique du vaisseau.
Par la suite les deux voyageuses ont fait ce qu’elles pouvaient pour contenir la situation, et ont réussi à stopper l’incendie dans la pièce des réacteurs.
Sans nouvelles de Nathan, elles sont parvenues à isoler les sections endommagées pour faciliter les réparations et éviter d’autres incidents.
Elle n’étaient désormais plus que deux, sans possibilité de communiquer avec qui que ce soit à l'extérieur et à proximité d’un micro-trou noir ultra dangereux.
Les jours ont passé, et alors qu’elles travaillaient ensemble pour rétablir les systèmes de contrôle du Voidwalker, nos deux protagonistes ont fait face à une étrange série d’événements :
Une intense distorsion gravitationnelle a traversé le vaisseau comme une vague invisible, le faisant vaciller dans son intégralité. Le courant s’est coupé brutalement et les lumières se sont éteintes, mais ce n’était pas seulement le noir… C'était un silence absolu, angoissant, comme si le son lui-même avait été aspiré hors de l'air.
Prise de panique et ne recevant aucun signe de vie d’Ana, Eden se mit à la chercher frénétiquement.

En tâtonnant dans l’obscurité, elle s’empresse de se rendre dans sa chambre pour aller chercher sa lampe torche mais, désorientée à cause de la distorsion, elle trébuche et s'évanouit avec le choc de la chute.

Lorsque Eden reprend connaissance, le silence est toujours omniprésent. Le bourdonnement habituel des systèmes du vaisseau, même en mode secours, est absent. Les systèmes électriques ayant laché, l’ensemble du vaisseau est plongé dans une quasi obscurité.

Ne se souvenant plus de rien, Eden saisit sa lampe torche et se met à explorer le vaisseau : c’est là que le jeu débute.

I.F) Détail des lieux, items, personnages

Les différents lieux

Salle de contrôle : Pièce cruciale du vaisseau. Permet d’observer la carte, l’état des machines et d’en relancer certaines, et permet également de terminer le jeu une fois les objectifs réalisés.

Observatoire : Permet d’observer l’extérieur, et de voir le temps qu’il reste avant que le vaisseau soit attiré par la planète.

Laboratoire : Comporte des éléments et produits qui seront utiles à l’avancée.

Réfectoire : Une des pièces de vie du vaisseau. On y trouve des rations de nourriture que le joueur peut manger et des traces laissant remarquer que plusieurs individus ont occupé le vaisseau.

Habitations : Chambres de l’équipage, où des indices peuvent être trouvés.

Stockage : Contient des pièces et outils qui seront nécessaires aux réparations.

Générateurs : Systèmes permettant d’avoir de l'électricité et de l'oxygène dans le vaisseau. Ceux-ci doivent impérativement être réactivés pour la suite du jeu.

Réacteurs : Zone technique où les réparations principales doivent avoir lieu. Verrouillée par un digicode trouvable dans la chambre de Nathan.

Personnages

Le personnage principal se nomme Eden. On apprendra son nom en fouillant les différentes chambres du vaisseau, à la recherche d'informations sur la situation et ce qu'il s'est passé.

On apprendra également l'existence de deux autres personnages, Nathan et Ana, qui sont les deux autres membres disparus de l'équipage :

Nathan, le commandant du vaisseau, était chargé de prendre les décisions critiques pendant la mission.
Ce qu'on apprend sur lui : En explorant le vaisseau, Eden découvre que Nathan a tenté d'organiser l'évacuation. Les journaux de bord révèlent une lutte acharnée pour maintenir l'équipage en vie malgré la dégradation des systèmes. Évacuer le vaisseau semblait être la meilleure des solutions, mais il sera le seul à avoir pu s'éjecter en raison des dysfonctionnements de divers systèmes. Eden et Ana n'auront plus de nouvelles de lui par la suite.
On pourrait penser que Nathan a été aspiré dans l'espace peu de temps après l'éjection. Sa disparition reste un mystère.

Ana est une scientifique brillante et compatissante. Elle travaillait sur l'analyse des radiations émises par le trou noir.
Ce qu'on apprend sur elle : Ana semble avoir été plus affectée que les autres par les effets cognitifs de l'anomalie. En explorant sa chambre et le laboratoire, Eden trouve des notes de recherche incohérentes mêlées à des dessins frénétiques. Ces indices montrent qu'elle commençait peu à peu à perdre la raison.

Items

Fusible : Petit fusible métallique qui servira à faire fonctionner le levier pour activer les générateurs. Se situe dans la salle de stockage.

Lampe torche : Se situe dans la chambre d'Eden.

Bidon d'essence : Bidon rempli de carburant pour les réacteurs. Se situe dans la salle de stockage.

Pilule "magique" : Pilule compacte pleine de bienfaits et contenant énormément de nutriments. Celle-ci double la capacité de l'inventaire du joueur lorsqu'elle est consommée. Se situe dans le laboratoire.

Paquet de chips : Peut-être consommé par le joueur, mais ne fait rien de particulier pour l'instant. Se situe dans la cantine.

Beamer (téléporteur) : Rechargeable à l'infini, cet appareil permet de mémoriser une salle et de s'y téléporter depuis n'importe où dans le vaisseau. Se situe dans le laboratoire.

Montre : Permet de voir le temps restant avant la fin de la partie. Se situe dans l'observatoire.

Boîte à outils : Simple boîte à outils, indispensable pour les réparations. Se situe dans la salle de contrôle.

Capteur : À placer sur le réacteur pour les réparations. Se situe dans le laboratoire.

Journaux de bord : Consultables mais ne peuvent être ramassés, contiennent des informations sur les personnages, etc... Se situent dans les chambres.

I.G) Situations gagnantes et perdantes

Le joueur gagne la partie si il rétablit le courant pour les systèmes électroniques, répare les réacteurs du vaisseau, et reprend le contrôle de la trajectoire, évitant ainsi la collision avec la planète. Cela mène à une fin réussie où il échappe au danger.
Plus vulgairement, pour gagner, le joueur doit d'abord se rendre dans la salle des générateurs avec le fusible pour rétablir le courant ; puis il doit ensuite aller dans la salle des réacteurs avec tous les objets nécessaires (outils, pièces, carburant…), et les utiliser pour réparer les réacteurs. Enfin il doit se rendre dans le cockpit pour reprendre le contrôle du vaisseau et terminer le jeu.

Si le joueur ne parvient pas à réunir les pièces nécessaires, ou prend trop de temps à effectuer les réparations, le vaisseau se retrouvera attiré par la gravité de la planète. Cela mène à une fin tragique et, par conséquent, perdante.

Le joueur peut aussi se retrouver bloqué en ne parvenant pas à rétablir le courant dans le vaisseau, ce qui engendrerait des complications (ex. : impossibilité d'accéder à certains endroits verrouillés par code…) menant ainsi à une fin perdante.

I.H) Énigmes et épreuves

Tout d'abord, pour en savoir plus sur le mystérieux incident ayant touché le vaisseau, le joueur devra explorer celui-ci, à la recherche de notes ou de traces laissées par les autres voyageurs.
Pour ce qui est de la salle des réacteurs, son accès est verrouillé par un code. Cette salle ne sera donc pas disponible en début de partie et il faudra d'abord rétablir le courant pour pouvoir activer le verrou électronique de la porte. Ensuite, le joueur devra retrouver le code en cherchant dans les chambres.
Une fois la salle déverrouillée, le joueur pourra l'explorer pour constater les dégâts et, par la suite, se mettre à chercher toutes les pièces disséminées dans le vaisseau pour effectuer les réparations.

I.I) Commentaires

Pour l'instant, le scénario est terminé mais reste un peu bancal pour l'instant, je l'améliorerai par la suite.
J'ajouterai également à l'avenir de vraies images (reproduction du vaisseau en 3D par exemple) afin d'immerger un peu mieux le personnage dans le jeu.
Mettre un système de verrouillage par code pour la salle des réacteurs me prendrait peut-être un peu de temps, et comme c'est un exercice optionnel, je préfère mettre cet aspect de côté pour l'instant. La salle des réacteurs est donc déjà ouverte dans cette version du jeu.
La fin du jeu est assez simple, je vais la développer à l'avenir en apportant un peu plus d'interactivité (dans les réparations par exemple).

II. Réponses aux exercices

Exercices 7.5, 7.6 et 7.7

Le but de ces exercices était d'éviter la duplication de code, en adaptant certaines méthodes des classes Game et Room.
En bref, on a fait en sorte que la classe Room produise toutes les informations relatives aux sorties de la pièce courante avec les méthodes getExit() et getExitString(), puis on les utilise dans la classe Game pour afficher ces informations.
Cela permet d'alléger la classe Game, et de rendre le processus plus logique.

Voici le code correspondant dans les deux classes avant les exercices.

```
private void goRoom(final Command pC)
{
    Room vNextRoom = null;
    String vDirection = pC.getSecondWord();

    if (!pC.hasSecondWord())
    {
        System.out.println("go where ?");
        return;
    }

    if (vDirection.equals("north"))
    {
        vNextRoom = this.aCurrentRoom.aNorthExit;
    }
    else if (vDirection.equals("east"))
    {
        vNextRoom = this.aCurrentRoom.aEastExit;
    }
    else if (vDirection.equals("south"))
    {
        vNextRoom = this.aCurrentRoom.aSouthExit;
    }
    else if (vDirection.equals("west"))
    {
        vNextRoom = this.aCurrentRoom.aWestExit;
    }
    else
    {
        System.out.println("Unknown direction!");
        return;
    }

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

```
    {
        System.out.println("Unknown direction!");
        return;
    }

    if (vNextRoom == null)
    {
        System.out.println("There is no door!");
        return;
    }
    else
    {
        this.aCurrentRoom = vNextRoom;
        System.out.println(this.aCurrentRoom.getDescription());
        System.out.print("Exits : ");
        currentRoomExits();
    }
}

private void currentRoomExits()
{
    if (this.aCurrentRoom.aNorthExit != null)
    {
        System.out.print("north ");
    }
    if (this.aCurrentRoom.aEastExit != null)
    {
        System.out.print("east ");
    }
    if (this.aCurrentRoom.aSouthExit != null)
    {
        System.out.print("south ");
    }
    if (this.aCurrentRoom.aWestExit != null)
    {
        System.out.print("west ");
    }
    System.out.println("");
}
```

```
/**
 * Classe Room - un lieu du jeu d'aventure Zuul.
 *
 * @author votre nom
 */
public class Room
{
    private String aDescription;
    public Room aNorthExit;
    public Room aEastExit;
    public Room aSouthExit;
    public Room aWestExit;

    public Room(final String pD)
    {
        this.aDescription = pD;
    }

    public String getDescription()
    {
        return "You are " + this.aDescription;
    }

    public void setExits(final Room pN,
                         final Room pE,
                         final Room pS,
                         final Room pW)
    {
        this.aNorthExit = pN;
        this.aEastExit = pE;
        this.aSouthExit = pS;
        this.aWestExit = pW;
    }
} // Room
```

Et voici maintenant à quoi ressemble le code (il y a déjà la hashmap car je n'avais pas pris de capture avant de faire l'exercice 7.8).

```
private void goRoom(final Command pC)
{
    String vDirection = pC.getSecondWord();

    if (!pC.hasSecondWord())
    {
        System.out.println("go where ?");
        return;
    }

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

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

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

```
public Room getExit(final String pDirection)
{
    if (aExits.containsKey(pDirection))
    {
        return aExits.get(pDirection);
    }
    return unknownDirection;
}
```

```
public String getExitString(){
    String vExitString = "";
    if (aExits.get("north") != null)
    {
        vExitString += "north ";
    }
    if (aExits.get("east") != null)
    {
        vExitString += "east ";
    }
    if (aExits.get("south") != null)
    {
        vExitString += "south ";
    }
    if (aExits.get("west") != null)
    {
        vExitString += "west ";
    }
    if (aExits.get("back") != null)
    {
        vExitString += "back ";
    }
    if (aExits.get("upstairs") != null)
    {
        vExitString += "upstairs ";
    }
    if (aExits.get("downstairs") != null)
    {
        vExitString += "downstairs ";
    }
    if (aExits.get("southeast") != null)
    {
        vExitString += "southeast ";
    }
    if (aExits.get("northwest") != null)
    {
        vExitString += "northwest ";
    }

    return vExitString;
}
```

Désormais, la classe Room construit et gère toutes les informations relatives aux pièces, tandis que la classe Game se contente juste de les afficher.

Exercices 7.8 et 7.8.1

Ces exercices ont pour objectif d'implémenter la notion de HashMap dans notre projet. Un peu comme un dictionnaire en python, une HashMap est un ensemble de couples <clé,valeur> que l'on va utiliser ici pour stocker les différentes sorties d'une pièce en fonction de la direction qui mène à celle-ci.
Cela nous permettra d'ajouter autant de directions que l'on veut, y compris des déplacements verticaux.

Voici les parties modifiées du code :

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


public Room(final String pD)
{
    this.aDescription = pD;
    aExits = new HashMap <String,Room>();
}
```

On initialise ici la HashMap de la pièce courante, sous la forme d'un attribut

```
public void setExits(final String pDirection,
                     final Room pNeighbor)
{
    aExits.put(pDirection, pNeighbor);
}

public Room getExit(final String pDirection)
{
    if (aExits.containsKey(pDirection))
    {
        return aExits.get(pDirection);
    }
    return unknownDirection;
}
```

On adapte ensuite les méthodes de la classe Room, afin de pouvoir utiliser notre HashMap :

- *setExits() prend en paramètre la direction et la pièce qui correspond, puis les ajoute dans la HashMap*
- *getExit() permet d'accéder à la pièce correspondant à la direction passée en paramètre.*

```
vLowerDeck.setExits("east",vGenerators);
vLowerDeck.setExits("south",vReactors);
vLowerDeck.setExits("west",vStorage);
vLowerDeck.setExits("upstairs",vMainDeck);

vMainDeck.setExits("north",vLab);
vMainDeck.setExits("south",vCanteen);
vMainDeck.setExits("west",vQuarters);
vMainDeck.setExits("downstairs",vLowerDeck);
vMainDeck.setExits("upstairs",vUpperDeck);

vUpperDeck.setExits("north",vControlRoom);
vUpperDeck.setExits("south",vObservatory);
vUpperDeck.setExits("downstairs",vMainDeck);
```

Utilisation de la méthode setExits() dans la méthode createRooms() de Game, et ajout de déplacements verticaux

Exercices 7.9 et 7.10

Cet exercice sur l'utilisation du keySet permet d'alléger la méthode getExitString(), en créant un ensemble de toutes les clés présentes dans la HashMap de la pièce courante.

Avant ça, la méthode vérifiait pour chaque direction si celle-ci est présente ou non dans la HashMap, ce qui pouvait vite devenir redondant si on en a beaucoup…
Maintenant, la méthode crée un ensemble de toutes les clés de la hashmap, puis les concatène dans une chaîne de caractères par le biais d'une boucle *for each*.

```
public String getExitString(){
    String vExitString = "";
    if (aExits.get("north") != null)
    {
        vExitString += "north ";
    }
    if (aExits.get("east") != null)
    {
        vExitString += "east ";
    }
    if (aExits.get("south") != null)
    {
        vExitString += "south ";
    }
    if (aExits.get("west") != null)
    {
        vExitString += "west ";
    }
    if (aExits.get("back") != null)
    {
        vExitString += "back ";
    }
    if (aExits.get("upstairs") != null)
    {
        vExitString += "upstairs ";
    }
    if (aExits.get("downstairs") != null)
    {
        vExitString += "downstairs ";
    }
    if (aExits.get("southeast") != null)
    {
        vExitString += "southeast ";
    }
    if (aExits.get("northwest") != null)
    {
        vExitString += "northwest ";
    }

    return vExitString;
```

avant

```
public String getExitString(){
    String vExitString = "Exits :";
    Set<String> vKeys = aExits.keySet();
    for(String vExit : vKeys) {
        vExitString += " " + vExit;
    }

    return vExitString;
}
```

après

Exercices 7.10.1 et .2

La javadoc est consultable dans le fichier.
Je précise que je l'ai faite en anglais pour que tout le code soit dans la même langue.
On remarque que la javadoc n'affiche pas les méthodes privées, car elle n'affiche que ce qui est utilisable à l'extérieur de la classe.
Cela explique pourquoi on ne voit pas les méthodes de la classe Game.

Exercice 7.11

```java
/**
 * Prints the player's current location and available exits.
 */
private void printLocationInfo()
{
    System.out.println("You are in " + this.aCurrentRoom.getDescription());
    System.out.println(this.aCurrentRoom.getExitString());
} // printLocationInfo()
```

avant (classe Game)

```java
/**
 * Returns a description of the current room, of the form :
 *      You are in the Control Room.
 *      Exits : south
 *
 * @return A description of the room, and its exits
 */
public String getLongDescription()
{
    return "You are" + this.aDescription + ".\n" + getExitString();
} // getLongDescription()
```

après (classe Room)

```java
/**
 * Prints the player's current location and available exits.
 */
private void printLocationInfo()
{
    System.out.println(this.aCurrentRoom.getLongDescription());
} // printLocationInfo()
```

après (classe Game)

Exercice 7.14

```
public CommandWords()
{
    this.aValidCommands = new String[4];
    this.aValidCommands[0] = "go";
    this.aValidCommands[1] = "look";
    this.aValidCommands[2] = "help";
    this.aValidCommands[3] = "quit";
} // CommandWords()
```

Ajout de “look” dans les commandes

```
/**
 * Makes the player look around.
 */
private void look()
{
    printLocationInfo();
}
```

```
else if (pC.getCommandWord().equals("look"))
{
    if(pC.hasSecondWord() == true)
    {
        System.out.println("I don't know how to look at something in particular yet.");
        return false;
    }
    else
    {
        look();
        return false;
    }
}
```

Ajout d’une méthode look() dans Game, ainsi que le traitement de la commande dans processCommand()...

```
/**
 * Prints help instructions for the player, listing available command words.
 */
private void printHelp()
{
    System.out.println("You woke up alone in a big spaceship.\nThe place is totally deserted.");
    System.out.println("You must gather the information and components needed to repair critical systems.");
    System.out.println("\nAvailable commands:");
    System.out.println("  go [direction]  - Move to the room in the specified direction");
    System.out.println("  look            - Look around");
    System.out.println("  quit            - Exit the game");
    System.out.println("  help            - Display this help message");
} // printHelp()
```

…sans oublier les modifications dans printHelp()

Exercice 7.15

Pour ajouter la commande eat, il fallait procéder de la même manière que pour “look”.
Seulement, pour ajouter un peu plus de logique, j’ai voulu y ajouter une condition : celle qui est qu’on ne peut manger que si on se trouve dans le réfectoire (il serait un peu bizarre de trouver de la nourriture n’importe où dans le vaisseau).
Le problème était que je ne pouvais pas accéder à la pièce “vCanteen” car elle était déclarée dans createRooms(), une méthode privée. Donc je l’ai déclarée comme une variable d’instance afin de pouvoir l’utiliser ou je veux dans la classe.
Voici le code correspondant à cet exercice.

```
public class Game
{
    // ### Attributes ###

    private Room aCurrentRoom;
    private Parser aParser;
    private Room vCanteen = new Room("in the canteen, empty tables and chairs give the impression of a recently abandoned space. You can also find food in the fridge.");
```

```
/**
 * Makes the player eat food.
 */
private void eat()
{
    if (this.aCurrentRoom == vCanteen)
    {
        System.out.println("You have eaten now and you are not hungry anymore.");
    }
    else
    {
        System.out.println("There's no food here");
    }
}
```

```
/**
 * Prints help instructions for the player, listing available command words.
 */
private void printHelp()
{
    System.out.println("You woke up alone in a big spaceship.\nThe place is totally deserted.");
    System.out.println("You must gather the information and components needed to repair critical systems.");
    System.out.println("\nAvailable commands:");
    System.out.println("  go [direction]  - Move to the room in the specified direction");
    System.out.println("  look            - Look around");
    System.out.println("  eat             - Eat (only if you're in the canteen)");
    System.out.println("  quit            - Exit the game");
    System.out.println("  help            - Display this help message");
} // printHelp()
```

Exercice 7.16

Jusqu'à maintenant, la procédure printHelp() devait être mise à jour à chaque fois que l'on ajoutait une commande, ce qui pouvait très facilement laisser la place aux oublis et aux erreurs.
On a donc créé une méthode dans la classe CommandWords qui va afficher automatiquement chaque commande présente dans le tableau de commandes autorisées.
Comme il n'y a pas de lien entre les classes Game et CommandWords, il faut passer par la classe Parser.

Voici les modifications effectuées :

```java
/**
 * Prints all valid commands to System.out.
*/
public void showAll()
{
    for(String vCommand : this.aValidCommands)
    {
        System.out.print(vCommand + " ");
    }
    System.out.println();
} // showAll()
```

…dans CommandWords,

```java
/**
* Calls showAll() method from CommandWords
*/
public void showCommands()
{
    aValidCommands.showAll();
} // showCommands()
// Parser
```

…dans Parser,

```java
/**
 * Prints help instructions for the player, listing available command words.
 */
private void printHelp()
{
    System.out.println("You woke up alone in a big spaceship.\nThe place is totally deserted.");
    System.out.println("You must gather the information and components needed to repair critical systems.");
    System.out.println("\nAvailable commands:");
    this.aParser.showCommands();
} // printHelp()
```

…et dans la classe Game.

Exercice 7.18

Cet exercice consiste à modifier la méthode showAll() de la classe CommandWords afin de renvoyer une chaîne de caractère contenant toutes les commandes, plutôt que de les afficher une par une.

```java
/**
 * Prints all valid commands to System.out.
*/
public String getCommandList()
{
    String vCommandList = "Commands : ";
    for(String vCommand : this.aValidCommands)
    {
        vCommandList += (vCommand + " ");
    }
    return vCommandList;
} // showAll()
```

Après modifications

Exercice 7.18.8

On souhaite ajouter un bouton sur notre interface, afin d'appeler une certaine fonction.
Pour l'instant, j'ai décidé d'associer la procédure eat() à ce bouton : il a donc fallu la rendre publique dans GameEngine pour pouvoir y accéder depuis la classe UserInterface.

```java
/**
 * Makes the player eat food.
 * Works only if the player is in the Canteen.
 */
public void eat()
{
    if (this.aCurrentRoom == vCanteen)
    {
        this.aGui.println("You have eaten now and you are not hungry anymore.");
    }
    else
    {
        this.aGui.println("There's no food here");
    }
} // eat()
```

```java
this.aEatButton = new JButton("Button1");

JPanel vPanel = new JPanel();
vPanel.setLayout( new BorderLayout() ); // ==> only five places
vPanel.add( this.aImage, BorderLayout.NORTH );
vPanel.add( vListScroller, BorderLayout.CENTER );
vPanel.add( this.aEntryField, BorderLayout.SOUTH );
vPanel.add( this.aEatButton, BorderLayout.WEST );


this.aMyFrame.getContentPane().add( vPanel, BorderLayout.CENTER );

// add some event listeners to some components
this.aEntryField.addActionListener( this );
this.aEatButton.addActionListener ( this );
```

```java
/**
 * Actionlistener interface for entry textfield.
 */
@Override public void actionPerformed( final ActionEvent pE )
{
    if (pE.getSource() == this.aEatButton)
    {
        this.aEngine.eat();
    }
    else if (pE.getSource() == this.aEntryField)
    {
        this.processCommand();
    }
} // actionPerformed(.)
```

Exercice 7.20

Dans cet exercice, on souhaite apporter la possibilité de mettre un objet dans une pièce.
J'ai donc d'abord commencé par créer une classe Item qui a comme attributs une description et un poids.
Ensuite, dans la classe Room, j'ai ajouté un attribut Item (null par défaut, car il n'y aura pas d'objets dans chaque pièce pour l'instant) avec un setter.
Il ne faut pas oublier de modifier la méthode getLongDescription pour que le joueur sache s'il y a des objets dans la salle.

```java
public class Item
{
    private String aDesc;
    private double aWeight;

    /**
     * Constructeur d'objets de classe Item
     */
    public Item(final String pDesc, final double pWeight)
    {
        this.aDesc = pDesc;
        this.aWeight = pWeight;
    }

    public String getDescription()
    {
        return this.aDesc;
    }

    public double getWeight()
    {
        return this.aWeight;
    }

    public String getItemString()
    {
        return this.aDesc + "(Weight : " + this.aWeight + ")";
    }
}
```

Nouvelle classe Item

```java
public void setItem(final Item pI)
{
    this.aItem = pI;
}
```

```java
public String getItemString()
{
    if (this.aItem != null)
    {
        return "There's " + this.aItem.getDescription() + " here.";
    }
    else
    {
        return "No item here.";
    }
}
```

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

(classe Room)

```java
Item vFuse = new Item("a fuse", 1.0);
vStorage.setItem(vFuse);
```

Exemple de création d'objet dans une Room (dans createRooms())

```
You are in the storage room, shelves are stacked with supplies and components that may be useful for repairs.
Exits : east
There's a fuse here.
```

Description de la salle Storage

Exercices 7.21.1 et 7.22

Ici on modifie la méthode look de GameEngine pour pouvoir observer un objet précis, ou autour de soi si on ne spécifie rien.

```java
public Item getItem(final String pName)
{
    if (pName.equals(this.aItem.getName()))
    {
        return this.aItem;
    }
    else
    {
        return null;
    }
}
```

Ajout d'une fonction getItem() dans Room

J'ai ajouté un attribut "nom" à la classe Item pour pouvoir les retrouver plus facilement

```java
/**
 * Makes the player look around or at a specific item.
 */
private void look(final Command pC)
{
    if (!pC.hasSecondWord())
    {
        printLocationInfo();
        return;
    }

    String vItemName = pC.getSecondWord();
    Item vItem = this.aCurrentRoom.getItem(vItemName);

    if (vItem != null)
    {
        this.aGui.println(vItem.getItemString());
    }
    else
    {
        this.aGui.println("There is no '" + vItemName + "' here.");
    }
} // look()
```

look() après modifications

Pour résumer :

- La procédure prend désormais la commande du joueur en paramètre
- Vérifie s'il y a un second mot. Si non, elle affiche la description de la position du joueur et les sorties.
- Si il y'en a un, elle cherche dans la pièce si un Item portant le nom spécifié existe, et retourne sa description et son poids si c'est le cas (par le biais de la fonction getItem() dans Room).
- Si le nom fourni ne désigne pas un Item présent dans la pièce, un message informatif est affiché.

```
> look fuse
This is a small metallic fuse, it could help reactivating the generators. ( Weight: 0.5)
```

Utilisation de look sur l'Item 'fuse'

Exercices 7.22 et 7.22.1

Pour cet exercice, j'ai décidé, comme pour les salles, de stocker les Items dans une HashMap<Nom_De_l'Item, Item> de la classe Room. Cela me permet donc de pouvoir ajouter autant d'Items que je le souhaite dans une salle, et de pouvoir les retrouver juste avec leur nom.
Il m'a fallu donc créer une procédure addItem pour ajouter l'item spécifié dans la HashMap, et modifier les méthodes qui utilisent des objets, telles que look, ou encore getItemString()

```java
/**
 * Adds an Item in the room
 *
 * @param pItem Item to add in the room.
 */
public void addItem(final Item pItem)
{
    this.aItems.put(pItem.getName(), pItem);
} //addItems()
```

```java
Item vFuse = new Item("fuse","a
Item vKey = new Item("key","a k
vQ1.addItem(vFuse);
vQ1.addItem(vKey);
```

Procédure addItem et son utilisation dans createRooms()

```java
public String getItemString()
{
    String vItems = "Items in the room : ";

    if (this.aItems.isEmpty())
    {
        return "No items here.";
    }

    Set<String> vKeys = this.aItems.keySet();
    for (String vI : vKeys)
    {
        vItems += vI + " ";
    }

    return vItems;
} //getItemString()
```

Modifications apportées à la fonction getItemString(), même principe que getExitString()

```
You are in your quarters, you awaken surrounded by sparse, familiar belongings and a sense of urgency to uncover what happened.
Exits : out
Items in the room : fuse key
> look key
This is a key that could open something (Weight: 0.5)
```

Affichage

Exercice 7.26

Il faut implémenter une commande back dans notre jeu, permettant au joueur de revenir en arrière autant de fois qu'il le souhaite, jusqu'à ce qu'il revienne au début.
J'ai donc utilisé une pile (Stack) pour accomplir cette tâche : suivant un modèle Last In First Out (un peu comme une pile d'assiettes), c'est la structure la plus adaptée au problème.
L'idée ici est de stocker chaque pièce visitée dans la pile, l'une après l'autre afin de constituer une sorte d'historique. Puis lorsque la commande est appelée, le programme prend la dernière Room ajoutée à la pile (qui est donc la dernière pièce visitée) et celle-ci redevient la salle actuelle.

```
private Stack<Room> aVisitedRooms;

/**
 * Constructor for objects of class GameEngine
 */
public GameEngine()
{
    this.aParser = new Parser();
    this.createRooms();
    this.aVisitedRooms = new Stack<>();
}
```

Création d'un attribut de type Stack dans GameEngine

```
private void goRoom( final Command pCommand )
{
    if ( ! pCommand.hasSecondWord() ) {
        // if there is no second word, we don't know where to go...
        this.aGui.println( "Go where?" );
        return;
    }

    String vDirection = pCommand.getSecondWord();
    Room vNextRoom = this.aCurrentRoom.getExit( vDirection );

    if ( vNextRoom == null )
        this.aGui.println( "There is no door!" );
    else {
        this.aVisitedRooms.push(this.aCurrentRoom);
        this.aCurrentRoom = vNextRoom;
        this.aGui.println( this.aCurrentRoom.getLongDescription() );
        if ( this.aCurrentRoom.getImageName() != null )
            this.aGui.showImage( this.aCurrentRoom.getImageName() );
    }
}

private void goBack()
{
    if (this.aVisitedRooms.empty())
    {
        this.aGui.println("You cannot go back from here! No previous room.");
    }
    else
    {
        Room vPreviousRoom = this.aVisitedRooms.pop();
        this.aCurrentRoom = vPreviousRoom;
        this.aGui.println( this.aCurrentRoom.getLongDescription() );
        if ( this.aCurrentRoom.getImageName() != null )
            this.aGui.showImage( this.aCurrentRoom.getImageName() );
    }
}
```

Modifications dans goRoom() et création de la procédure goBack, appelée par la commande 'back'

```
else if (vCommandWord.equals("back"))
    this.goBack();
else
```

```
public CommandWords()
{
    this.aValidCommands = new String[6];
    this.aValidCommands[0] = "go";
    this.aValidCommands[1] = "look";
    this.aValidCommands[2] = "help";
    this.aValidCommands[3] = "quit";
    this.aValidCommands[4] = "eat";
    this.aValidCommands[5] = "back";
} // CommandWords()
```

Sans oublier les modifications dans processCommand() et CommandWords()

```
> go east
You are on the main deck, you stand at the crossroads of several hallways leading to different parts of the ship.
Exits : upstairs south north west downstairs
No items here.
> go south
You are in the canteen, empty tables and chairs give the impression of a recently abandoned space. You can also find food in the fridge.
Exits : northwest north
No items here.
> back
You are on the main deck, you stand at the crossroads of several hallways leading to different parts of the ship.
Exits : upstairs south north west downstairs
No items here.
> back
You are in the crew quarters and find yourself facing three rooms.
Exits : east room3 room1 room2 southeast
No items here.
> back
You are in your quarters, you awaken surrounded by sparse, familiar belongings and a sense of urgency to uncover what happened.
Exits : out
No items here.
> back
You cannot go back from here! No previous room.
```

Utilisation de ‘back’ jusqu'à revenir au début de l’historique

Exercices 7.28.1 et 7.28.2

On souhaite ajouter une commande test, permettant d’ouvrir un script avec des commandes et exécutant successivement toutes les commandes lues dans celui-ci.
J’ai donc implémenté ça dans une procédure, vérifiant d’abord si la commande comporte un nom de fichier, puis ajoute ensuite l’extension .txt a la fin de ce dernier (si ce n’est pas déjà le cas).
Ensuite, le programme exécute chaque commande du script passé en paramètre.

```
else if (vCommandWord.equals("test"))
    this.test(vCommand.getSecondWord());
else
```

Modifications dans interpretCommand()

```java
private void test(String pFileName)
{
    if (pFileName == null)
    {
        this.aGui.println("Please specify a filename to test.");
        return;
    }

    if (!pFileName.endsWith(".txt"))
    {
        pFileName += ".txt";
    }

    Scanner vSc;
    try
    {
        vSc = new Scanner(new File(pFileName));
        while(vSc.hasNextLine())
        {
            String vLine = vSc.nextLine();
            this.interpretCommand(vLine);
        }
    }
    catch (final FileNotFoundException pFNFE)
    {
        this.aGui.println("File not found.");
    }
}
```

Procédure test()

```
> test court
> go out
You are in the crew quarters and find yourself facing three rooms.
Exits : east room3 room1 room2 southeast
No items here.
> look
You are in the crew quarters and find yourself facing three rooms.
Exits : east room3 room1 room2 southeast
No items here.
> go east
You are on the main deck, you stand at the crossroads of several hallways leading to different parts of the ship.
Exits : upstairs south north west downstairs
No items here.
> go downstairs
You are on the lower deck, the dim lighting and narrow passages create a tense atmosphere.
Exits : east upstairs south west
No items here.
> back
You are on the main deck, you stand at the crossroads of several hallways leading to different parts of the ship.
Exits : upstairs south north west downstairs
No items here.
> eat
There's no food here
```

Affichage

Il y a donc 4 fichiers de test dans mon jeu :

- court.txt qui teste quelques commandes
- all.txt qui teste absolument toutes les commandes et possibilités du jeu
- allitems.txt qui va chercher tous les items un à un
- win.txt qui exécute le chemin le plus court vers la victoire

Exercice 7.29

Dans cet exercice, on ajoute à notre programme une nouvelle classe Player, qui va gérer toutes les informations propres à un joueur (ex. : son nom, la salle ou il se situe, les salles visitées…).

```
// ### Attributes ###
private String        aName;
private Room          aCurrentRoom;
private double        aMaxWeight;
public Stack<Room>    aVisitedRooms;

// ### Constructor ###

public Player(final String pName, final Room pStartRoom)
{
    this.aName = pName;
    this.aCurrentRoom = pStartRoom;
    this.aVisitedRooms = new Stack<>();
}

// ### Getters ###

public String getPlayerName()
{
    return this.aName;
}

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

// ### Other methods ###

public void goRoom(final Room pR)
{
    this.aVisitedRooms.push(this.aCurrentRoom);
    this.aCurrentRoom = pR;
}

public void goBack()
{
    Room vPreviousRoom = this.aVisitedRooms.pop();
    this.aCurrentRoom = vPreviousRoom;
}
```

Classe Player, avec une partie de certaines méthodes de GameEngine

```
this.aPlayer = new Player("Player1",vQ1);
```

Initialisation du joueur dans createRooms(),
avec le nom et la salle de départ en paramètres

A noter que, pour l'instant, je n'ai pas touché aux méthodes look() et eat() car :

- look() ne modifie pas l'état du joueur, donc il est inutile de la mettre dans cette classe.
- Pareil pour eat() pour l'instant.

Exercice 7.30

Ajout de deux méthodes, take() et drop() dans la classe Player, permettant de ramasser et de déposer un objet.
Cela impliquait d'abord de créer dans Player un attribut contenant l'item lorsque le joueur le porte, ainsi qu'une méthode removeItem() dans Room (car si on ramasse un objet, il n'est donc plus présent dans la pièce).

```java
/**
 * Remove the specified Item from the room
 *
 * @param pItemName Name of the Item to remove from the room.
 */
public void removeItem(final String pItemName)
{
    if (this.aItems.containsKey(pItemName))
        this.aItems.remove(pItemName);
} //removeItem()
// Room
```

Nouvelle méthode removeItem()

```java
public void take(final String pItemName)
{
    this.aItemInHand = this.aCurrentRoom.getItem(pItemName);
    this.aCurrentRoom.removeItem(pItemName);
}

public void drop()
{
    this.aCurrentRoom.addItem(this.aItemInHand);
    this.aItemInHand = null;
}
```

take() stocke l'Item souhaité dans l'attribut ItemInHand et le retire de la pièce, drop() fait l'inverse.

```
Items in the room : fuse
> take fuse
> look
You are in the storage room, shelves are stacked with supplies and components that may be useful for repairs.
Exits : east
No items here.
> go east
You are on the lower deck, the dim lighting and narrow passages create a tense atmosphere.
Exits : east upstairs south west
No items here.
> go south
You are in the reactors room, you hear loud clicking noises, and the machinery looks unstable and damaged.
Exits : north
No items here.
> drop
> look
You are in the reactors room, you hear loud clicking noises, and the machinery looks unstable and damaged.
Exits : north
Items in the room : fuse
> look fuse
This is a small metallic fuse, it could help reactivating the generators. (Weight: 0.5)
```

Test des deux méthodes, en déposant l'objet récupéré dans une autre pièce.

Exercices 7.31, 7.32 et 7.33

Désormais, le joueur peut porter plusieurs objets à la fois, tant que la limite de poids n'est pas dépassée, grâce à une Hashmap de la forme <nomObjet, objet>.
Voici les modifications apportées :

```
public double getInvWeight()
{
    double vTotal = 0.0;
    for (Item vI : this.aInventory.values())
    {
        vTotal += vI.getWeight();
    }
    return vTotal;
}

public String getInvString()
{
    String vInventory = "Inventory :";

    Set<String> vKeys = this.aInventory.keySet();
    for(String vI : vKeys)
    {
        vInventory += " " + vI;
    }

    vInventory += "\nTotal weight : " + this.getInvWeight() + " (max "+ this.aMaxWeight + ")";

    return vInventory;
}
```

Deux nouvelles méthodes, la première permettant de retourner le poids total de l'inventaire, et la 2e retournant une chaîne contenant la liste des objets, ainsi que le poids total (un peu comme getLongDescription() dans Room)

```
public void take(final String pItemName)
{
    Item vItem = this.aCurrentRoom.getItem(pItemName);
    if (vItem == null)
    {
        this.aGui.println("There is no item with that name here.");
        return;
    }

    double vNewWeight = this.getInvWeight() + vItem.getWeight();

    if (vNewWeight > this.aMaxWeight)
    {
        this.aGui.println("Maximum weight exceeded. You cannot carry that item.");
        return;
    }
    this.aInventory.put(pItemName, vItem);
    this.aCurrentRoom.removeItem(pItemName);
    this.aGui.println(pItemName + " picked up.");
}

public void drop(final String pItemName)
{
    if (!this.aInventory.containsKey(pItemName))
    {
        this.aGui.println("You are not carrying that item.");
    }
    Item vItem = this.aInventory.remove(pItemName);
    this.aCurrentRoom.addItem(vItem);
    this.aGui.println(pItemName + " dropped.");
}
```

Modification des méthodes take() et drop()

D’ailleurs, pour afficher les messages dans l’UI depuis la classe Player, j’ai dû y créer un attribut GUI (comme dans GameEngine) ainsi qu’une méthode setGUI que j’appelle dans la méthode setGUI de GameEngine. C’est le seul moyen que j’ai trouvé pour relier Player a l’interface correctement.

```
public void setGUI( final UserInterface pUserInterface )
{
    this.aGui = pUserInterface;
} // setGUI()
```

Dans Player

```
/**
 * Sets the game graphical user interface.
 *
 * @param pUserInterface instance of the userInterface class.
 */
public void setGUI( final UserInterface pUserInterface )
{
    this.aGui = pUserInterface;
    this.aPlayer.setGUI(pUserInterface);
    this.printWelcome();
} // setGUI()
```

Dans GameEngine

Exercice 7.34

Ajout d'un item spécial qui, lorsqu'il est consommé, permet de doubler le poids maximum de l'inventaire du joueur.
J'ai choisi de mettre un nom et une description plus appropriés au contexte de l'histoire.

```
Item vMagicPill = new Item("boost_pill", "a pill of nano-engineered proteins.\nEnhances your strength instantly when consumed.", 0.5);
vLab.addItem(vMagicPill);
```

Initialisation de l'item dans GameEngine

```
public void eat(final String pItemName)
{
    Item vItem = this.aInventory.getItem(pItemName);
    if (vItem == null)
    {
        this.aGui.println("You are not carrying that item.");
        return;
    }

    if (pItemName.equals("boost_pill"))
    {
        this.aMaxWeight *= 2;
        this.aInventory.removeItem(pItemName);
        this.aGui.println("You ate the Boost Pill! Your carrying capacity has doubled to " + this.aMaxWeight + ".");
    }
    else if (pItemName.equals("chips"))
    {
        this.aInventory.removeItem(pItemName);
        this.aGui.println("You ate a pack of Orbitos ! You are not hungry anymore.");
    }
    else
    {
        this.aGui.println("You cannot eat this.");
    }
}
```

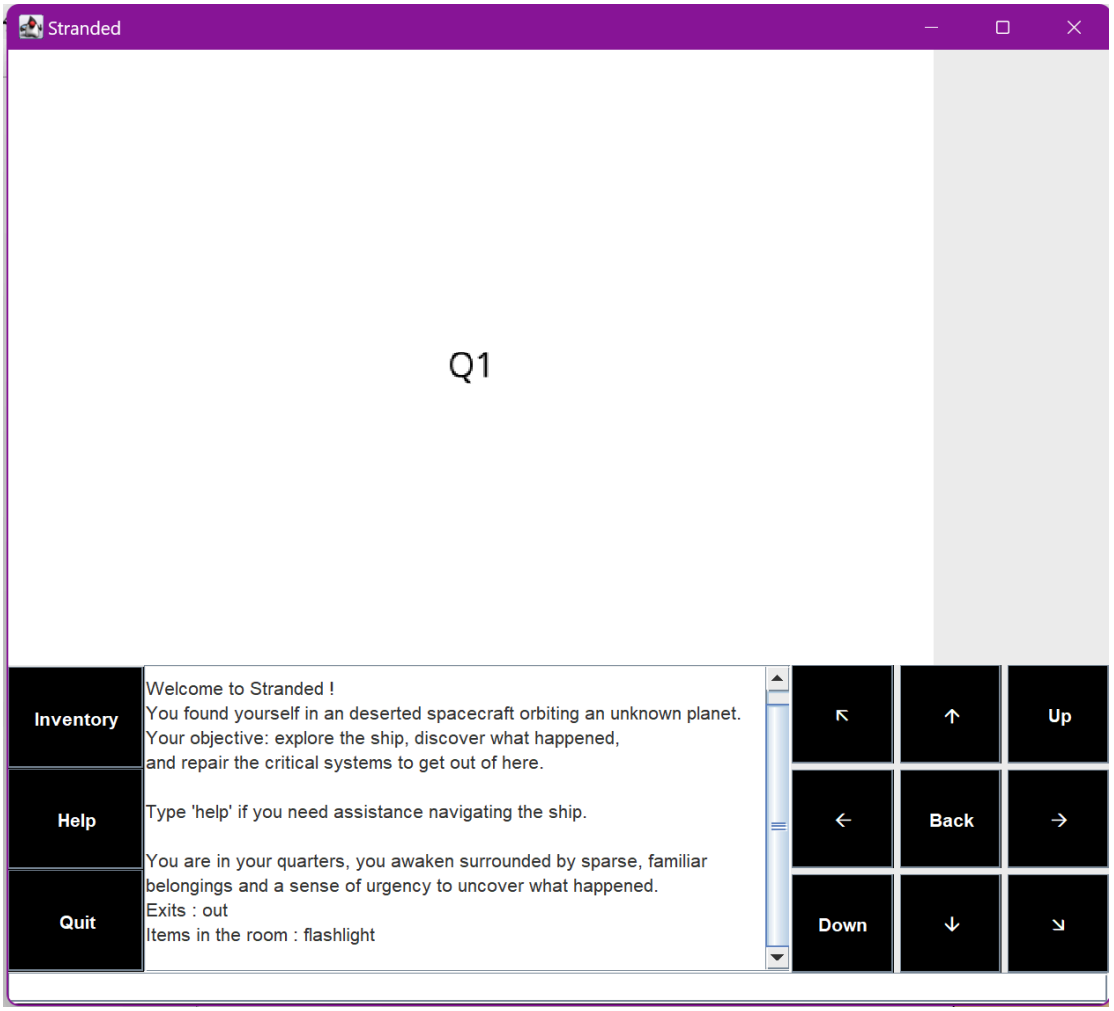
Modifications de la méthode eat(), déplacée de GameEngine à Player
(J'avais également créé un item "paquet de chips" trouvable dans la cantine : pour l'instant celui-ci ne fait rien mais pourrait avoir un effet positif sur la santé du joueur à l'avenir par exemple.)

```
> inventory
Inventory : fuse chips boost_pill flashlight
Total weight: 5.5 (max 10.0)
> eat boost_pill
You ate the Boost Pill! Your carrying capacity has doubled to 20.0.
> inventory
Inventory : fuse chips flashlight
Total weight: 5.0 (max 20.0)
```

Affichage

Modifications de l’interface (boutons)

Le bouton Eat ne servant plus à rien (puisque la méthode prend désormais un second mot), il m’a fallu le remplacer. J’ai donc décidé d’ajouter des boutons pour les déplacements, ainsi que pour les autres commandes ne prenant pas de second mot comme “inventory”, “help” ou encore “quit”.
Pour cela, j’ai utilisé d’autres layouts que celui proposé dans le code d’UserInterface qui nous a été fourni. Ces derniers sont GridLayout, et GridBagLayout et permettent d’ajouter des composants dans une disposition de grille.



Voila pour l’instant à quoi ça ressemble, il reste bien évidemment quelques ajustements à faire !

Exercices 7.42 et 7.42.1

On souhaite ajouter une limite de temps à notre jeu, obligeant le joueur à gérer son temps et instaurant ainsi une dimension stratégique dans son aventure.

Comme le scénario l'indique, le vaisseau est en dérive, s'approchant dangereusement d'une planète environnante.
J'avais déjà eu l'idée de limiter le temps disponible pour accomplir toutes les tâches et terminer le jeu, et de donner au joueur un objet permettant de suivre le temps restant.

```
Item vWatch = new Item("watch", "a high-tech gravimetric watch designed for space missions.\nDisplays the time remaining before the ship crashes into the planet.\n", 2.0);
vObservatory.addItem(vWatch);
```

Cet objet sera donc une montre, que le joueur pourra trouver dans l'observatoire

Ensuite, il m'a fallu apporter pas mal de modifications dans UserInterface, afin de pouvoir créer un timer à 15min qui tourne en fond, invisible tant que le joueur ne porte pas la montre sur lui.

```
public JLabel     aTimerLabel;
private Timer     aGameTimer;
private int       timeRemaining = 900;
```

Le panel du timer est mis en public pour que la classe Player puisse y accéder

```
this.aImage = new JLabel();
this.aImage.setBounds(0, 0, 800, 600);
vImagePlusTimer.add(this.aImage,0);

this.aTimerLabel = new JLabel("00:00", JLabel.CENTER);
this.aTimerLabel.setFont(new Font("Arial", Font.BOLD, 20));
this.aTimerLabel.setForeground(Color.WHITE);
this.aTimerLabel.setOpaque(true);
this.aTimerLabel.setBackground(Color.BLACK);
this.aTimerLabel.setBounds(350, 570, 100, 30);
this.aTimerLabel.setVisible(false);
vImagePlusTimer.add(this.aTimerLabel,1);

this.aGameTimer = new Timer(1000, new ActionListener()
{
    @Override
    public void actionPerformed(ActionEvent e)
    {
        timeRemaining--;
        int vMinutes = timeRemaining / 60;
        int vSeconds = timeRemaining % 60;
        aTimerLabel.setText(String.format("%02d:%02d", vMinutes, vSeconds));

        if (timeRemaining <= 0)
        {
            println("\nCritical alert !!! The ship has entered the atmosphere of the nearby planet.");
            println("Hull integrity failing... Brace for impact !");
            println("\nYou tried your best, but time has run out.\nThe ship succumbs to gravity, plummeting towards the surface.");
            println("Your journey ends here. Game Over.");
            enable( false );
        }
    }
});
this.aGameTimer.start();
```

```
vPanel.add(vImagePlusTimer, BorderLayout.NORTH);
```

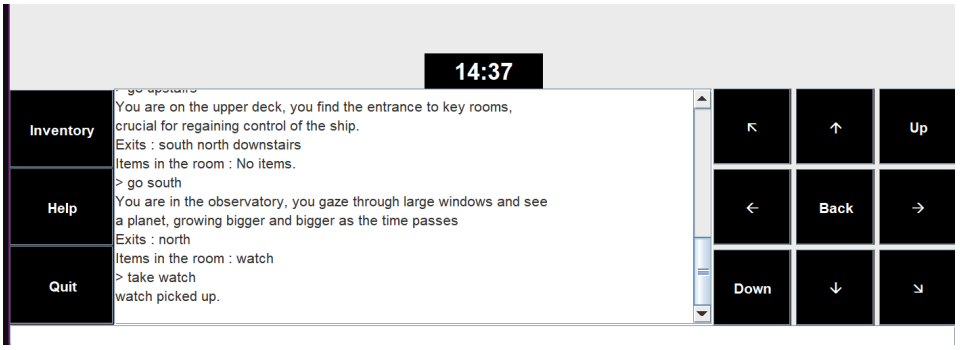
Méthode createGUI(), avec ajout d'un panel pour le timer, et l'utilisation de JLayeredPane pour pouvoir le superposer sur l'image

```
    if(pItemName.equals("watch"))
    {
        this.aGui.aTimerLabel.setVisible(true);
    }
} // take()

/**
 * Drops an item from the player's inventory into the current room.
 *
 * @param pItemName The name of the item to be dropped.
 */
public void drop(final String pItemName)
{
    Item vItem = this.aInventory.getItem(pItemName);
    if (vItem == null)
    {
        this.aGui.println("You are not carrying that item.");
        return;
    }
    this.aInventory.removeItem(pItemName);
    this.aCurrentRoom.addItem(vItem);
    this.aGui.println(pItemName + " dropped.");

    if(pItemName.equals("watch"))
    {
        this.aGui.aTimerLabel.setVisible(false);
    }
} // drop()
```

Méthodes take() et drop() de la classe Player



Voici l'affichage pour l'instant

Exercice 7.42

J'ai décidé de ne pas me lancer dans une IHM trop compliquée, et de garder celle que j'ai actuellement.

Exercice 7.43

Ici, on souhaite ajouter une trap door dans notre map, c'est-à-dire une sortie de Room qui n'est franchissable que dans un sens.
J'ai décidé de la placer au niveau du Réfectoire : on peut y accéder depuis les Quartiers ou le Couloir principal, mais une fois dedans, on ne peut qu'aller vers le Couloir principal.

```
vLab.setExits("south", vMainDeck);
//vCanteen.setExits("northwest", vQuarters); (trap door)
vCanteen.setExits("north", vMainDeck);
```

Implémenter la trap door en elle-même est très simple : il suffit juste de retirer la sortie que l'on souhaite rendre inaccessible

Reste la commande back : en effet, si on n'apporte pas les modifications nécessaires, celle-ci pourrait permettre au joueur de revenir en arrière et ainsi franchir la trap door à l'envers.

```
/**
 * Checks if the given room is one of the exits of this room.
 *
 * @param pRoom The room to check.
 * @return True if the room is an exit, false otherwise.
 */
public boolean isExit(final Room pRoom)
{
    return this.aExits.containsValue(pRoom);
} // isExit()
// Room
```

Création d'une fonction qui vérifie si la salle fournie en paramètre est une sortie de la Room courante.

```
/**
 * Moves the player back to the previous room (if it's accessible)
 * using the "visited rooms" history.
 */
public void goBack()
{
    Room vPreviousRoom = this.aVisitedRooms.pop();
    if (!this.aCurrentRoom.isExit(vPreviousRoom))
    {
        this.aGui.println("You cannot go back this way.");
        return;
    }
    this.aCurrentRoom = vPreviousRoom;
} // goBack()
```

Modifications de goBack() dans la classe Player

Exercice 7.44

Cet exercice a pour but d'implémenter un Item téléporteur (beamer) qui doit pouvoir être ramassé dans une première pièce, pour ensuite être chargé dans une autre pièce (ou la même), puis déclenché dans une troisième. Lors de l'utilisation, le joueur est téléporté dans la salle qui était mémorisée dans le téléporteur.

Étant un objet qui n'a pas du tout le même comportement que les autres, j'ai donc créé une classe Beamer, sorte d'Item. J'ai ensuite, dans CommandWords, implémenté deux nouvelles commandes "charge" et "fire" afin de pouvoir l'utiliser en jeu. Enfin, j'ai réalisé les modifications nécessaires dans GameEngine et Player.

```java
public class Beamer extends Item
{
    // ### Attributes ###

    private Room aChargedRoom;
    private boolean aIsCharged;

    // ### Constructor ###

    public Beamer(final String pN, final String pD, final double pW)
    {
        super(pN,pD,pW);
        this.aIsCharged = false;
    }
```

Extrait de la classe Beamer

```java
public void chargeBeamer(final String pItemName)
{
    Item vItem = this.aInventory.getItem(pItemName);
    if (vItem == null)
    {
        this.aGui.println("You are not carrying the beamer.");
        return;
    }

    Beamer vB = (Beamer)vItem;

    vB.charge(this.aCurrentRoom);
    this.aGui.println("Beamer charged.");
}

/**
 * Fires the beamer in the player's inventory to teleport them to the charged
 * room, if the player is carrying it.
 * If its not charged or not in the player's inventory, a message is displayed.
 */
public void fireBeamer(final String pItemName)
{
    Item vItem = this.aInventory.getItem(pItemName);
    if (vItem == null)
    {
        this.aGui.println("You are not carrying the beamer.");
        return;
    }

    Beamer vB = (Beamer)vItem;

    Room vFiredRoom = vB.fire();
    if(vFiredRoom == null)
    {
        this.aGui.println("The beamer is not charged.");
        return;
    }
    this.aCurrentRoom = vFiredRoom;
    this.aGui.println("Teleported "+ this.aCurrentRoom.getLongDescription());
}
```

Classe Player

```java
Item vBeamer = new Item("beamer", "a sophisticated device capable of instant teleportation\nto a memorized location.", 3.0);
vLab.addItem(vBeamer);
```

Ajout de l'Item dans GameEngine

```java
else if (vCommandWord.equals("charge"))
{
    if ( !vCommand.hasSecondWord() || !vCommand.getSecondWord().equals("beamer"))
        this.aGui.println( "Charge what ?" );
    else
        this.aPlayer.chargeBeamer(vCommand.getSecondWord());
}else if (vCommandWord.equals("fire"))
{
    if ( !vCommand.hasSecondWord() || !vCommand.getSecondWord().equals("beamer"))
        this.aGui.println( "Fire what ?" );
    else
        this.aPlayer.fireBeamer(vCommand.getSecondWord());
}
```

interpretCommand()

N.B : Pour une raison que j'ignore actuellement, même après avoir essayé de mettre des print un peu partout dans mes méthodes pour voir ce qui ne se déclenche pas, mon beamer ne réagit pas aux commandes "charge beamer" et "fire beamer", qui sont pourtant bien prises en compte par le jeu.
Ce problème n'impactant pas la progression du joueur dans le jeu, j'ai décidé de le mettre de côté et de revenir dessus avant l'oral.

Commande "use" et fin de jeu

Pour que le joueur puisse gagner la partie, il doit :

- retrouver les trois objets nécessaires pour les réparations, en plus de la boîte à outils
- les utiliser dans leurs salles
- aller dans la salle de contrôle et gagner la partie.

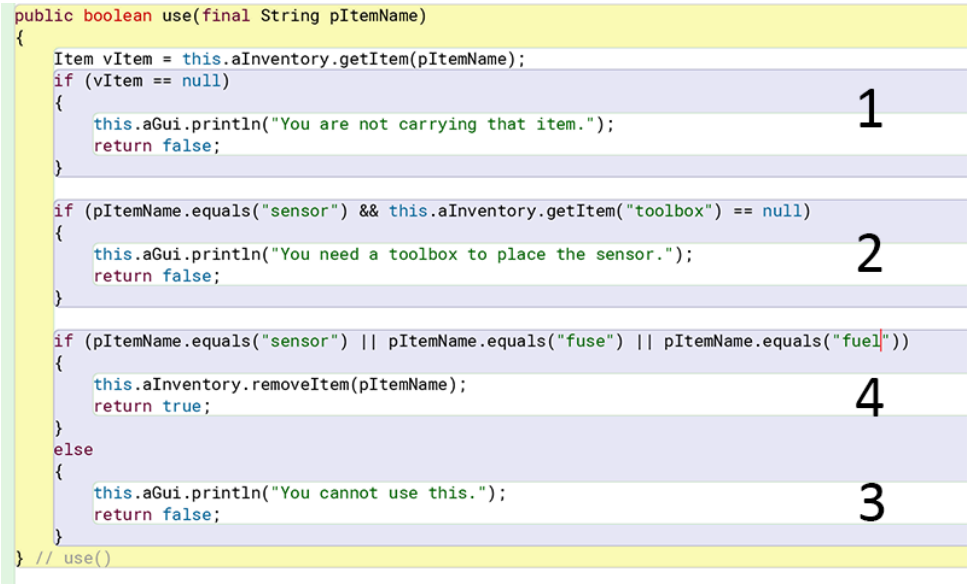
Cela nécessitait donc de mettre en place une commande "use" (qui m'a pris énormément de temps car beaucoup de conditions) et d'implémenter les méthodes correspondantes.
Celles-ci étant longues, je ne vais pas les capturer dans leur intégralité.

```
public void use(final String pItemName)
{
    switch(pItemName)
    {
        case "fuse" :
            if (this.aPlayer.getCurrentRoom().equals(this.aGenerators))
            {
                if (this.aPlayer.use(pItemName) == true)
                {
                this.aUsedItems.add(pItemName);
                this.aGui.println("Used " + pItemName + ". All the electronic systems came back to life !");
                this.aGenerators.updateDescription("in the generator room, the equipment is now operational\nand full of electricity.");
                }
            }
            else
            {
                this.aGui.println("You cannot use this here.");
            }
            break;
```

```
            default :
                this.aPlayer.use(pItemName);
        }

        if (this.aUsedItems.containsAll(this.aWinItems))
        {
            this.aGui.println("\nEverything seems to be working again !");
            this.aGui.println("Head up quickly to the control room and\nregain control of the ship.");
        }
    } // use()
```

Extraits de la méthode use()



use() dans Player

Fonctionnement : le joueur entre la commande “use [item]” en jeu, le programme la récupère et appelle la méthode use() de GameEngine avec le deuxième mot en paramètre.
Celle-ci le lit, entre dans le cas correspondant, et appelle l’autre partie se situant dans Player.
La méthode use() de Player étant une fonction booléenne, si celle-ci retourne false, c’est que l’Item n’est pas utilisable, ou alors qu’il n’est pas dans l’inventaire du joueur. Si elle retourne true, dans ce cas la on peut utiliser l’Item et l’ajouter dans la liste des objets déjà utilisés.
A chaque utilisation, on regarde si tous les objets nécessaires ont été utilisés : si oui, un message s’affiche, indiquant au joueur qu’il peut terminer la partie en se rendant dans le cockpit.

Gestion des différentes possibilités :

- 1 : Le joueur veut utiliser un objet qu’il n’a pas
- 2 : Le joueur veut poser le capteur, mais n’a pas la boite a outils sur lui
- 3 : Le joueur veut utiliser un objet autre que les trois objets utilisables (capteur, fusible, carburant)
- 4 : Si toutes les conditions sont remplies, on peut exécuter les instructions et renvoyer true

J’ai, en parallèle, également implémenté une simple méthode updateDescription() dans la classe Room pour pouvoir actualiser les descriptions des pièces après les réparations.

III. Déclaration anti-plagiat

Cette déclaration est là pour attester que je n'ai pas plagié la moindre ligne de code pour mon jeu.

Cependant, j'ai utilisé quelques notions et packages qui étaient extérieurs au cours/TP/TD et aux exercices du projet.
J'ai donc consulté pas mal de pages internet et demandé de l'aide a ChatGPT, non pas pour trouver les réponses et lignes de code dont j'avais besoin, mais surtout pour savoir comment utiliser les packages par exemple.
J'ai passé beaucoup de temps sur l'interface, et plus particulièrement sur le timer, que je n'arrivais pas à placer et faire fonctionner correctement sur celle-ci : j'ai donc cherché sur internet quel layout serait le plus adapté pour superposer et placer le timer sur l'image. J'ai ensuite posé une question à ChatGPT pour savoir comment l'utiliser.
Par rapport au scénario, j'ai également demandé à ChatGPT de développer mes phrases (descriptions des salles) afin de gagner du temps et avoir une narration un peu plus réaliste et immersive.

Je ne sais pas si c'est considéré comme de la triche mais je préfère quand même le déclarer au cas où.

Pour ce qui est des sites consultés en dehors de ChatGPT, en voici quelques uns :

Overview (Java Platform SE 8)
https://borntocode.fr/
https://openclassrooms.com/
[Résolu] 2 panels qui se superposent par Zazou - page 1 - OpenClassrooms

Les images (non finales) présentes dans le jeu viennent d'Internet, principalement de Pinterest.