# Rapport Projet POO

# En java


BEN MOHAND Macine et HERVOUET Léo L3 informatique 2022

# Table des matières

# I - Documentation utilisateurs

## 1.Introduction du sujet

Notre projet est un jeu s'inspirant du jeu "Colossal Cave Adventure", c'est un jeu basé sur l'affichage d'un texte pour décrire une situation et ainsi de pouvoir avec ces informations avancer dans le jeu via une console. Ici nous avons un principe de salle, on se déplace de salle en salle pour avancer dans le jeu et dans celle-ci nous pouvons avoir différentes interactions pour pouvoir progresser. Nous utilisons des commandes suivies d'arguments ou pas pour pouvoir interagir avec le jeu. Notre jeu s'appelle JAMG pour "Just another Medieval Game" car finalement comme notre jeu est basé sur un univers médiéval ça faisait sens.

Au lancement du jeu, le héro que vous incarnez en quelque sorte, arrive dans une pièce et le but est de parcourir la map afin de trouver la sortie, ici le but est de terminer le jeu sur une victoire. Il faut bien sûr être attentif car certains détails permettront au joueur d'avancer dans le jeu mais aussi de perdre.

## 2.Comment lancer le jeu

Pour lancer le jeu est plutôt simple il suffit d'être sous Linux avec la version 17 de java d'installer. Ensuite depuis une console, il faut se mettre dans le répertoire /src qui contient /game et le document makeFile. Le document makeFile permet de compiler le jeu. Le makeFile a trois commandes : la première est « make » qui permet de compiler le projet, la seconde est « make run » qui permet de compiler le jeu et de lancer ce dernier. Vous pourrez donc jouer directement dans la console. Et la dernière commande permet de nettoyer les dossiers du jeu des .class qui sont générés grâce à la compilation. La commande pour faire cela est « make clean ».

# 3.Comment jouer au jeu

Comme nous l'avons dit plus haut le jeu fonctionne à base de commande rentré par l'utilisateur. Pour ce jeu il y a plusieurs commandes pour faire avancer le joueur entre les différentes pièces mais pas seulement certaines permettent également de prendre des informations sur l'environnement dans lequel le joueur se trouve. D'autres vont permettre de sauvegarder la progression du joueur, de quitter le jeu…

Les commandes sont les suivantes :

**-GO:** Permet de se déplacer d'une pièce vers une autre.

Exemple : "GO nomPiece"

**-HELP:** Permet de voir toutes les commandes utilisables.

Exemple "HELP"

**-LOOK:** Permet de voir les différentes choses qu'il y a dans la pièce et de redonner le descriptif de la pièce.

Exemple : "LOOK"

**-TAKE:** Permet de prendre un objet de la pièce.

Exemple :"TAKE nomObjet"

**-DROP:** Permet de remettre un objet dans la pièce qui vient de notre inventaire.

Exemple : "DROP nomObjet"

**-USE:** Permet de donner un objet au NPC(non-player character) qui se trouve dans la pièce ou de donner cet objet ou un code à une sortie pour la déverrouiller.

Exemples :"USE nomObjet" / "USE nomObjet nomSortie"

**-EXAMINE:** Permet d'avoir la description de l'objet qui se trouve dans la pièce.

Exemple : "EXAMINE"

**-INTERACT:** Permet de parler au NPC qui se trouve dans la pièce.

Exemple : "INTERACT"

**-INVENTORY:** Permet d'afficher tous les objets que le héro a dans son inventaire.

Exemple : "INVENTORY"

**-SAVE:** Permet de faire une sauvegarde du jeu.

Exemple :"SAVE nomDuFichierDeSauvegarde"

**-LOAD:** Permet de charger une sauvegarde du jeu.

Exemple : "LOAD nomDuFichierDeSauvegarde"

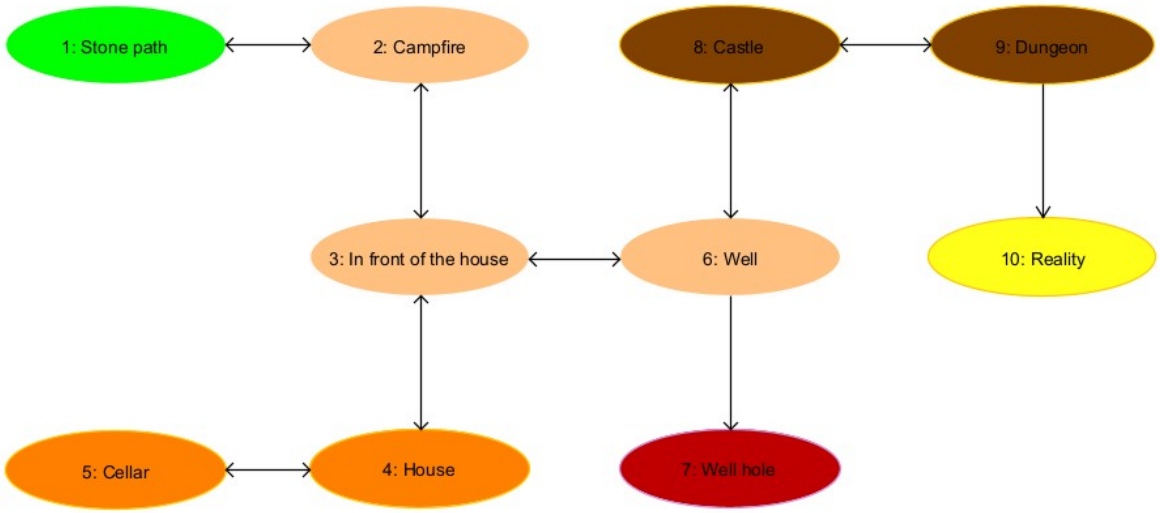
**-QUIT:** Permet de quitter le jeu.

Exemple :"QUIT"

## 4. Map et solution

Dans l'histoire du jeu vous êtes un héro qui arrive dans une nouvelle région et qui a pour but en arrivant dans cette nouvelle région de sauver la princesse qui se trouve dans un château plus précisément dans sa tour la plus haute qui est le donjon. Pour cela vous devez mener le héro jusqu'à la princesse sans mourir bêtement. Dans le jeu il y a différentes choses avec lesquelles vous pouvez interagir comme les éléments, qui sont des éléments du décor d'une pièce, les NPCs qui sont des personnages avec lesquelles vous pouvez parler, il y a aussi des objets que vous pouvez ramasser sur le sol pour pouvoir ensuite avoir accès à d'autre salle. Et enfin vous pouvez interagir avec les sorties d'une pièce. Selon la sortie, soit il n'y a pas de condition et vous pouvez la prendre ou soit cette sortie demande un objet ou un code pour la déverrouiller et pour pouvoir passer par la suite.

**Voici la map du jeu :**




Départ à Stone path et fin du jeu à Reality (ou potentiellement à Well hole).

**Solution Rapide pour aller a la fin du jeu :**

Go Campfire, Go HouseFront, Go Well, Take keys, Use Shazam Castle, Go Castle, Use Keys Dungeon, Go Dungeon, Go Reality.

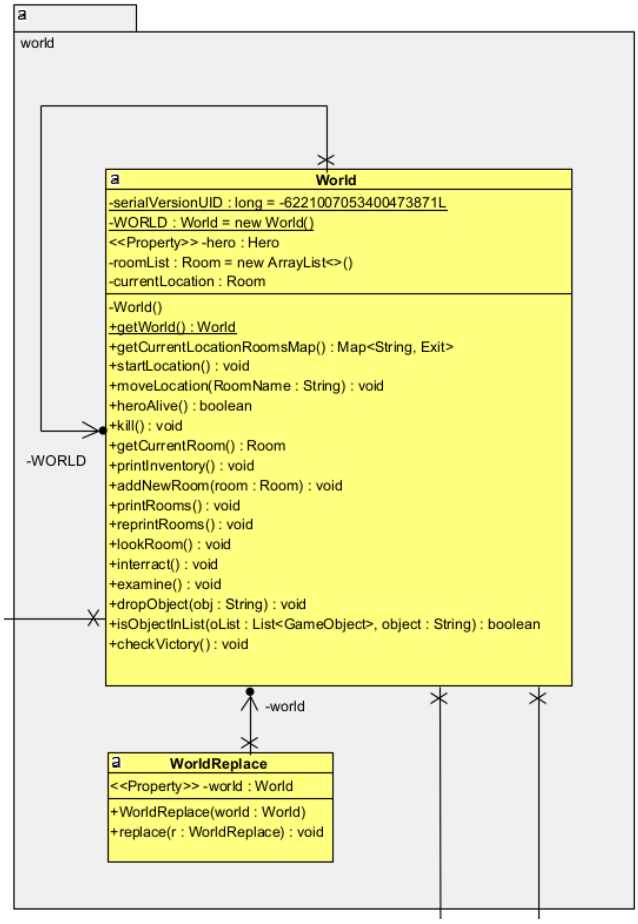
# II - Documentation développeurs

## 1.Introduction technique du jeu

Le but dans le développement du jeu était de faire un jeu modulable avec un ajout facile de pièces avec les sorties qui correspondent. Pour tous les éléments du jeu, donc les pièces, NPCs, objets, éléments, sorties et même les ordres il est facile d'en rajouter dans le code. C'était ici la priorité car sinon au fil du développement si notre histoire n'avait plus de sens ou que quelque chose nous gêner alors il aurait fallu tout refaire. Nous avons séparé tous ces éléments en différents packages comme nous allons le voir dans la suite.

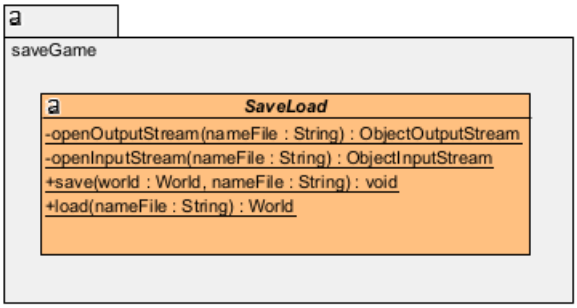
## 2.UML

Dans la fonction main nous utilisons un objet de classe World qui nous permet de faire les différentes actions du jeu. Ici un World est unique il n'y a pas de constructeur, un seul objet de la classe World se construit et nous pouvons le récupérer grâce à un getter.

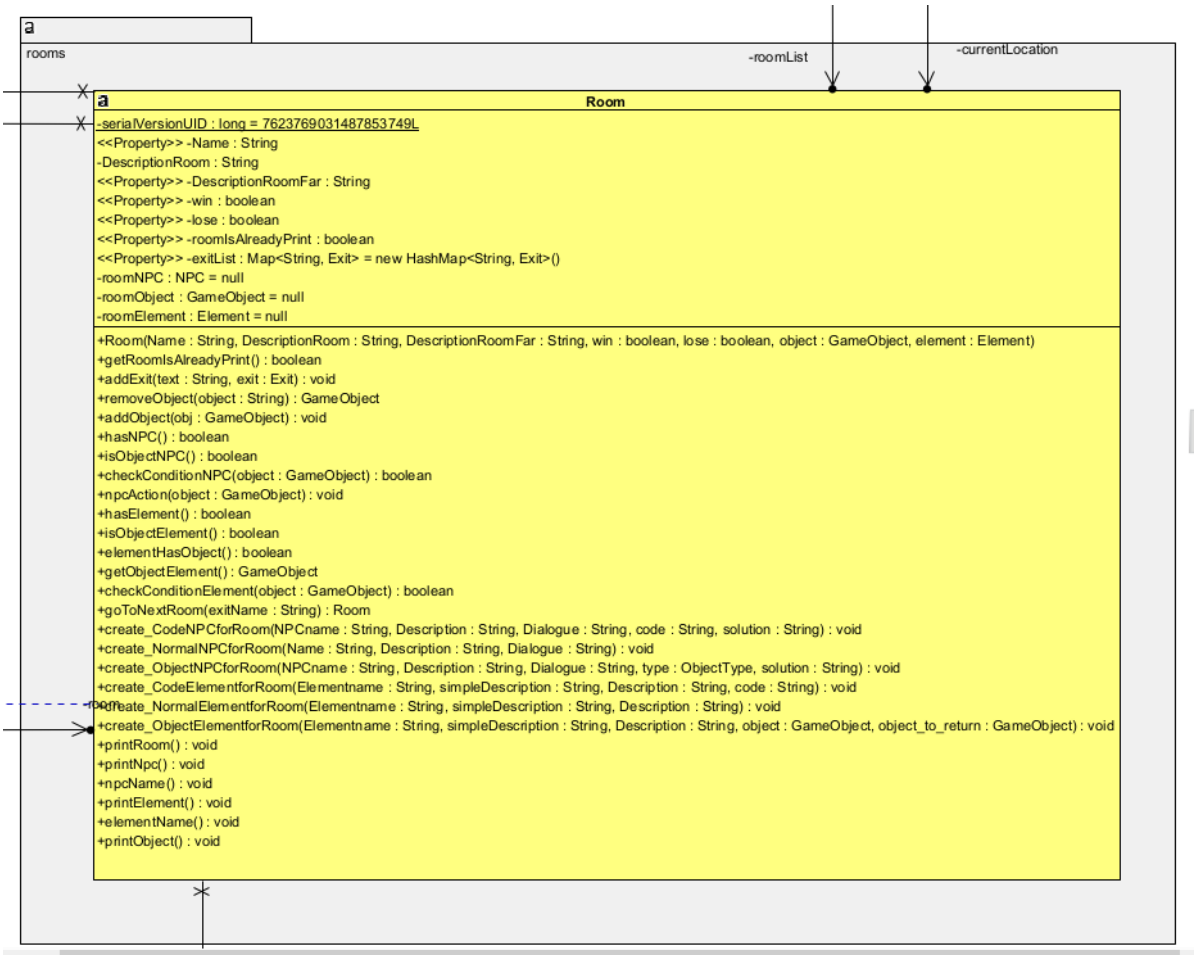


La classe WorldReplace permet de pouvoir changer entièrement un World, on me l'adresse de l'emplacement mémoire d'un World dans un WorldReplace et grâce à la fonction replace on peut mettre un nouveau World dans l'emplacement de notre objet.

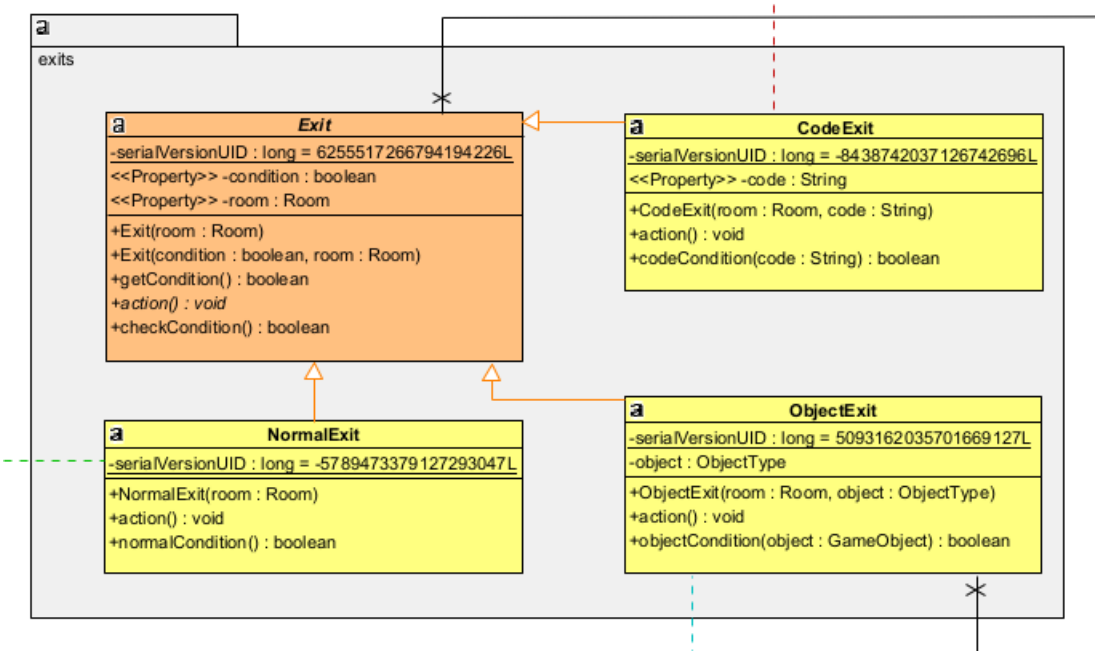
Nous utilisons cette classe principalement pour pouvoir changer le World par le World qui est donner par ma fonction de chargement d'une partie(save/load).



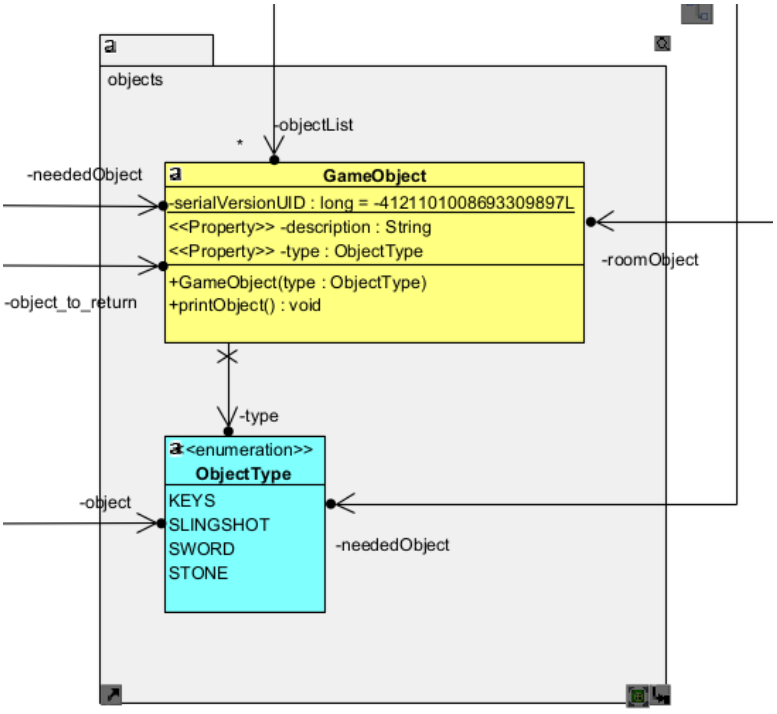

Dans un World il y a des Room et une Room courante qui nous permet de faire jouer le joueur dedans.



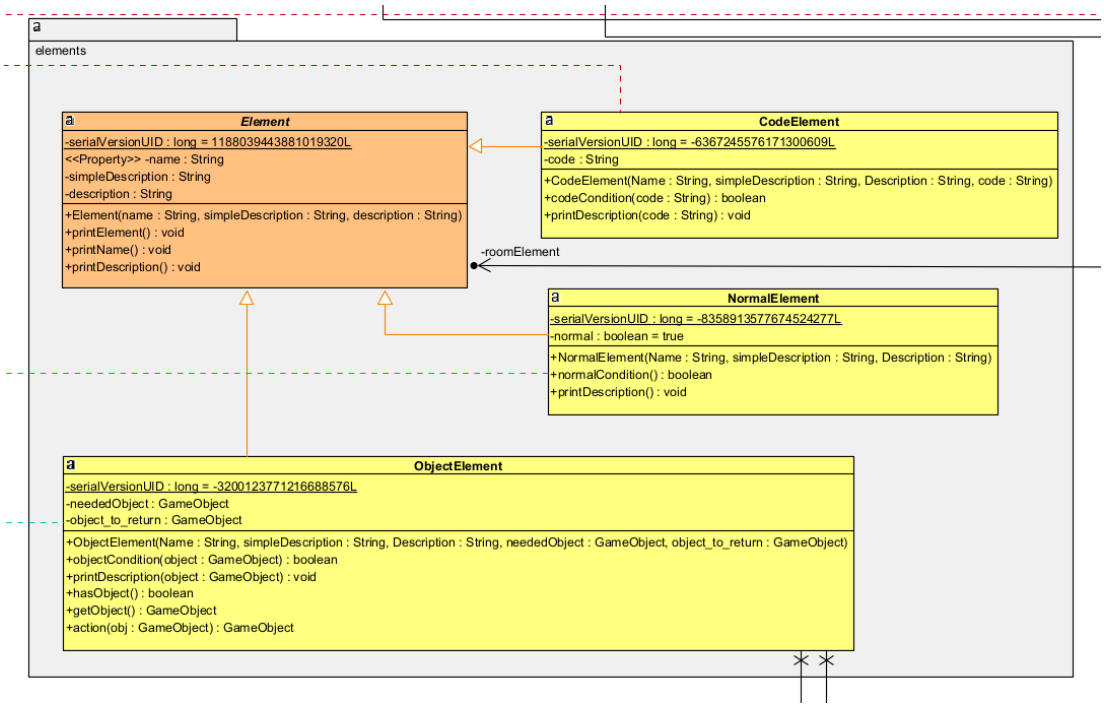
Les Rooms contiennent différentes choses comme par exemple des Exits qui permettent au joueur de se déplacer de salle en salle. Il y a différents types d'Exits. Il y a la NormalExit qui est une Exit qui ne demande pas de condition pour pouvoir sortir de la Room. Il y a ensuite la ObjectExit qui est une sortie qui demande un objet pour la déverrouiller et CodeExit est la même chose que ObjectExit sauf qu'au lieu d'utiliser un objet pour déverrouiller la sortie c'est un code qui est demandé.



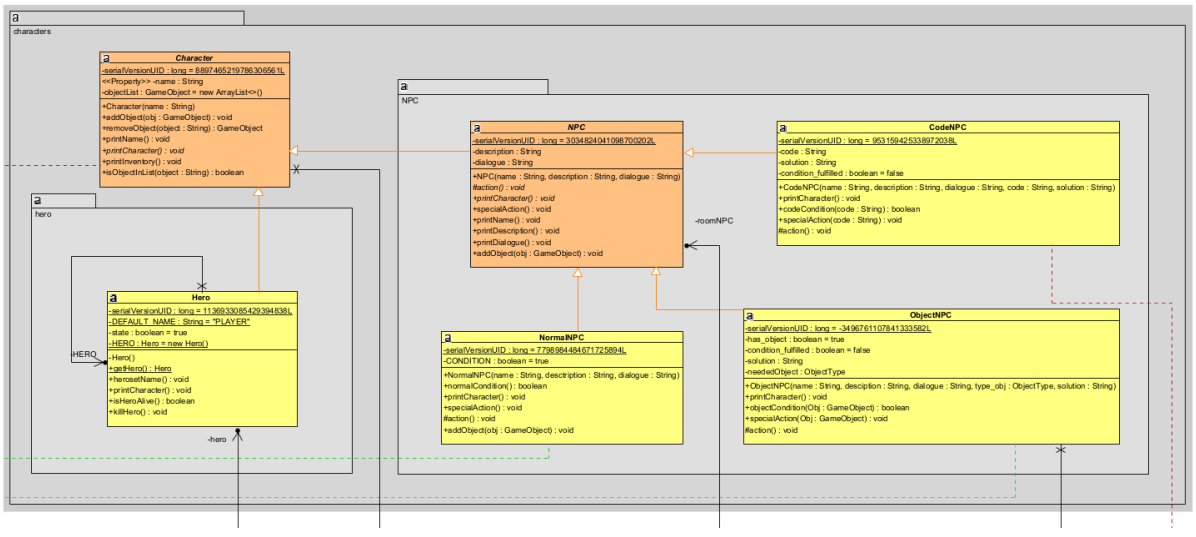

Ensuite dans une Room nous avons un objet qui permet justement d'interagir avec les ObjectExit par exemple.



Dans une Room nous avons aussi des Element qui sont en quelque sorte des objets qui font une action et que l'on peut regarder. Comme les Exit un Element peut avoir différentes réactions selon si vous lui donnez un code, un objet ou juste rien.

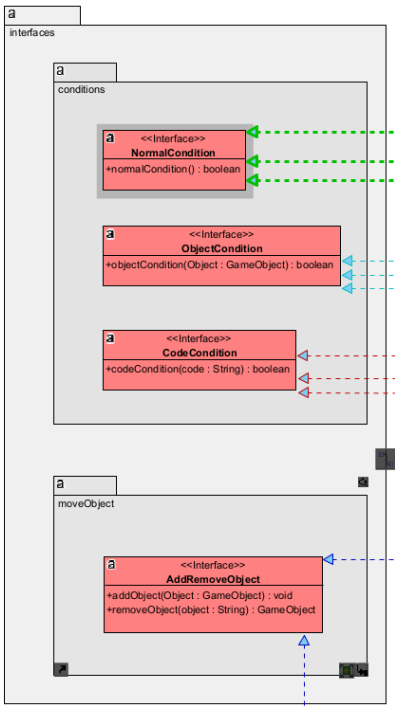



Dans une Room nous avons des NPC(non-player character) qui agissent un peu comme les Element.



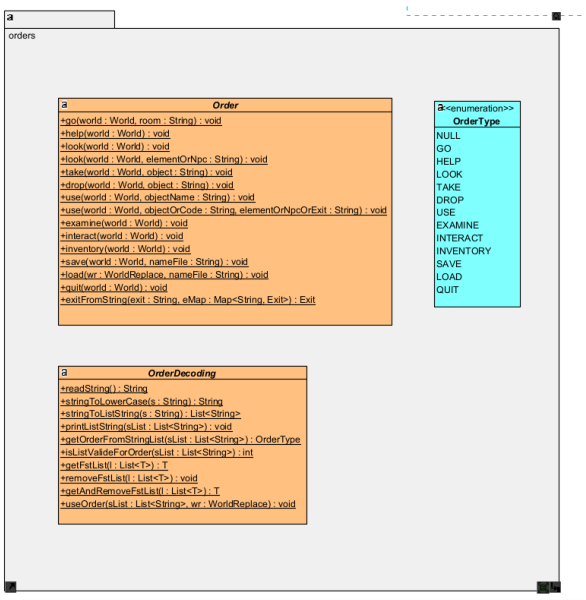

Les NPC dérivent de la classe Character qui permet à un NPC d'avoir un nom et une liste d'objet.
La classe Hero dérive aussi de la classe Character, il n'y a qu'un seul Hero dans le jeu que l'on récupère grâce à un getter. Le Hero est dans le World et interagi avec lui.

Les classes Exit, NPC et Element sont des classes dériver qui permet d'utiliser les interfaces (NormalCondition, ObjectCondition et CodeCondition) pour utiliser des actions sans paramètres et les actions avec paramètre (objet et code).

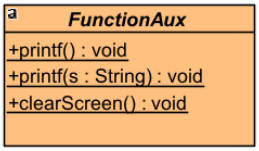


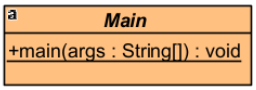

L'interface AddRemoveObject permet à un Character que pouvoir ajouter ou enlever des objets de son inventaire et elle sert également à Room pour pouvoir aussi faire la même chose.

Enfin nous utilisons la classe OrderDecoding pour pouvoir lire le texte qui est rentré par le joueur dans la console et pour pouvoir séparer la commande en morceaux pour pouvoir ainsi utiliser les ordres qui se trouve dans la classe Order. La classe Order regroupe toutes les commandes qui sont utilisables dans le jeu.

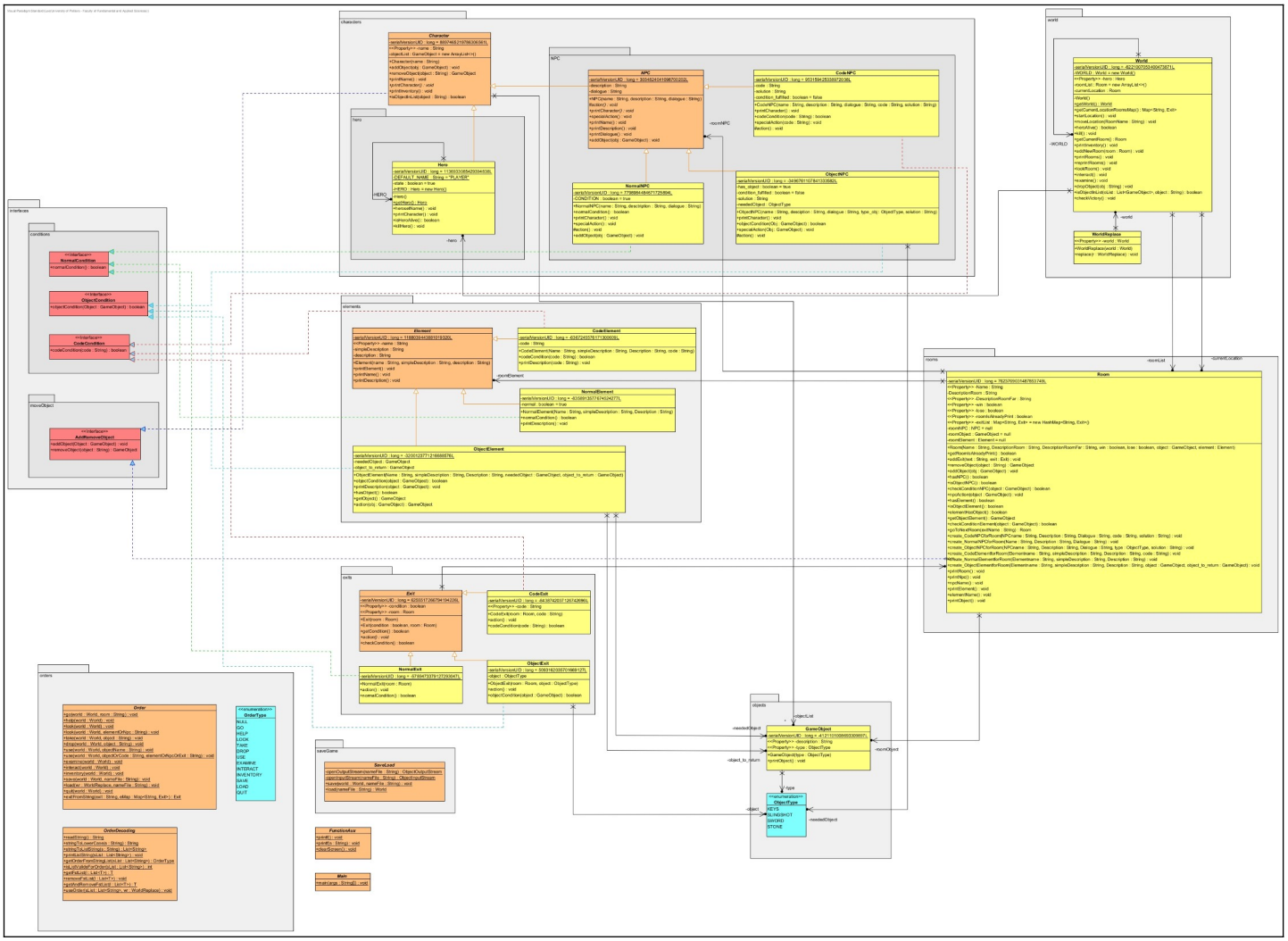


Et pour finir nous avons une classe Main ou le World est créé et ou la boucle de jeu se trouve. Quand on lance le jeu c'est le main de cette classe qui est appelé. La classe FunctionAux permet d'effacer l'écran et de pouvoir afficher du texte à l'écran.




Voici l'UML globale, ce n'est pas la version définitive de l'UML car par la suite certaines fonctions ont changé mais toute la structure évoquée juste avant reste d'actualité (Nous avons par exemple rajouté des objets, donc dans le type enum ObjectType il y a des changements mais qui ne change rien dans le fonctionnement du jeu).



# 3.Diagramme d’états

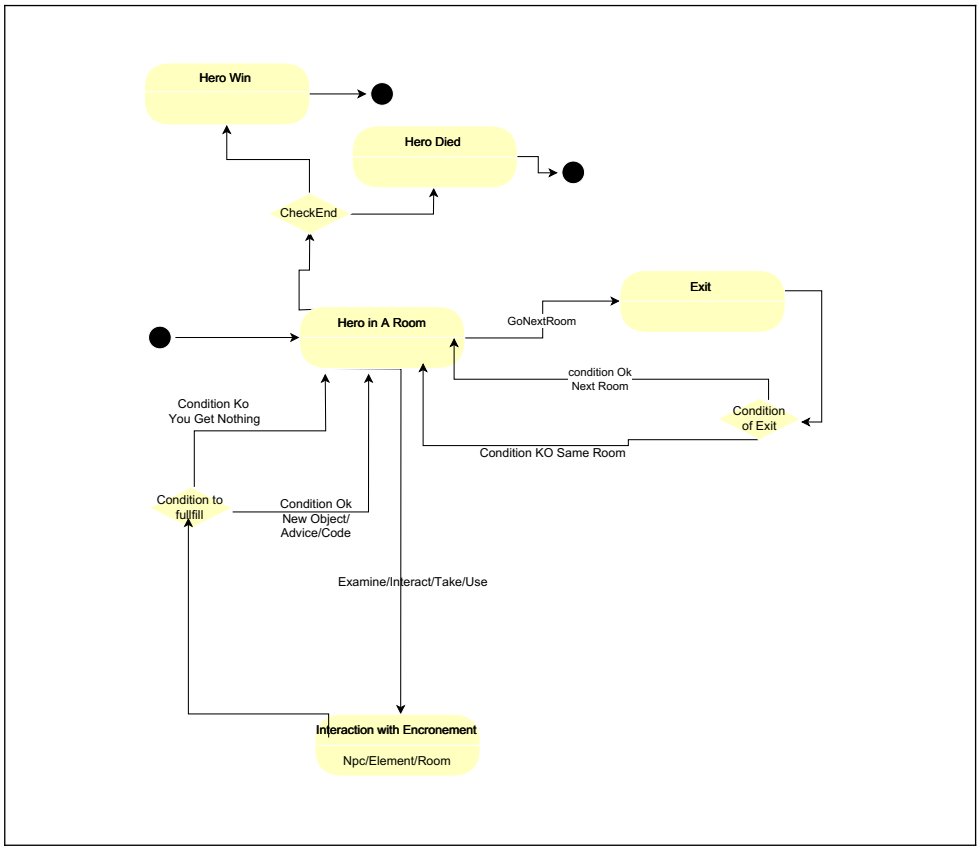
Dans le diagramme d’état nous remarquons qu’il y a 2 événements qui mènent a la fin du jeu:

-Soit le héro meurt.

-Soit il arrive dans la pièce Final.

Le reste du temps le héro est dans une pièce et attend les commandes de l’utilisateur afin d’interagir avec son environnement .

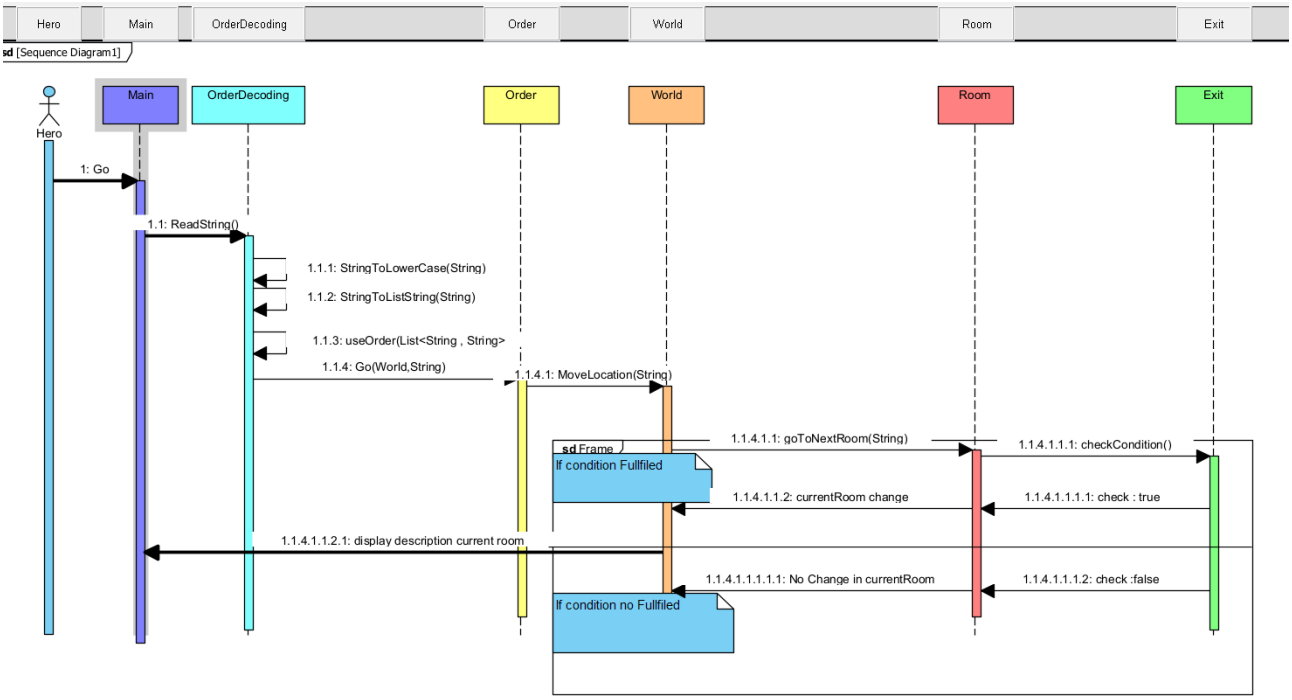
La commande Go déclenche l’événement qui permet au héro de se déplacer (ce n’est pas le héro qui se déplace mais la pièce qui change) , cependant le déplacement requiert de satisfaire une condition, si satisfaite on change de pièce, sinon on ne change pas.



# 4.Diagramme de séquence

Choix de la Commande Go, qui est la commande la plus importante du jeu, car elle permet au héro de se déplacer, est elle est aussi assez complète pour décrire le fonctionnement du jeu.

En effet, 6 différentes classes interviennent dans l'utilisation de la commande Go (Main, OrderDecoding, Order, World, Room, Exit), ce qui met en lumière les différentes interactions entre les classes.



# 5.Test

Création de Tests Fonctionnels,qui testent le fait que l'utilisateur se déplace bien dans les différentes pièces sans pour autant pouvoir se déplacer comme il veut et passer de la pièce ou le jeu commence à la fin du jeu sans effectuer les étapes nécessaire.

Pour utiliser les test sous linux il faudra ajouter des bibliothèques :

sudo apt install aptitude

sudo aptitude install junit

Puis la Commande

```
java org.junit.runner.JunitCore TestProjet.java
```

```
(Cependant la démarche n'a pas fontionné :

Erreur : impossible de trouver ou de charger la classe principale
org.junit.runner.JUnitCore

Causé par : java.lang.ClassNotFoundException:
org.junit.runner.JUnitCore)
```

# III - Organisation du groupe

Nous avons travaillé ensemble tout le long du projet et s'il nous arrivait d'avancer chacun de notre côté alors nous nous retrouvions soit le lendemain ou le surlendemain pour en discuter à la BU ou entre les cours. Le début a été un peu chaotique car nous avions un manque d'organisation mais au fil du projet nous sommes parvenus à avoir comme nous avons dit plutôt une bonne synchronisation dans notre travail.

Nous n'avons pas de manque au cahier des charges du projet, notre projet fonctionne correctement (et si nous avions trouvé des bugs avant le rendu alors nous les aurions mis ici). Le seul défaut de notre projet est que le code est relativement « fouillis » car il nous manquait un peu de temps sur la fin du projet. Et aussi nous aurions aimé que l'histoire de notre jeu dur plus longtemps mais la aussi par manque de temps nous avons raccourci l'histoire pour pouvoir rendre un jeu fonctionnel.

Organisation du travail :

**-Histoire :** Macine et Léo.

**-Programmation du jeu :**

Macine : World, Room, Element et Character(Hero et NPC).

Léo : Exit, SaveLoad, Order, OrderDecoding et GameObject.

**-Tests :** Macine.

**-Diagrammes d'états et de séquences :** Macine.

**-UML/Structure :** Léo.

**-README / makeFile :** Léo.

**-Rapport :** Macine et Léo.