Programmation Orientée Objet
Rapport projet

Yann BERTHELOT, Vincent COMMIN et Louis LEENART

Enseignant : M. Samuel Peltier
Licence 3 Informatique, Université de Poitiers, année 2020-2021

1 Table des matières

2 Introduction

Le projet demandé consiste à réaliser un jeu d'aventure en mode texte à la manière de "Colossal Cave adventure", nous avons eu pour idée de reprendre un sujet de projet de notre année de L2 qui, lui, consistait en un petit jeu appelé "Banquise", et de le transposer dans ce mode d'utilisation.

3 Utilisation

3.1 Installation

Pour compiler (ce qui est normalement déjà fait), il faut utiliser la commande suivante :

```
javac -d ./out src/**/*.java
```

Note: la version du jdk nécessaire est obligatoirement la 15.0 ou postérieure.

3.2 Lancement

3.2.1 Windows

Pour lancer notre jeu sous Windows, il existe plusieurs méthodes pour lancer notre jeu.

- Via ligne de commande :

```
cd out # Placez vous dans le dossier "out"
java Game.Main
```

- Via raccourci :

Ouvrez le fichier `RUN_WINDOWS.bat` qui se trouve dans la racine.

3.2.2 Linux

Pour lancer notre jeu sous Linux, il existe plusieurs méthodes pour lancer notre jeu.

- Via ligne de commande :

```
cd out # Placez vous dans le dossier "out"
java Game.Main
```

- En exécutant le raccourci :

Ouvrez le fichier `RUN_LINUX.sh` qui se trouve dans la racine via `bash RUN_LINUX.sh`

3.3 Commandes du jeu

Dans un premier temps, le but du jeu est de trouver le personnage appelé Chief Scientist et de lui parler. Pour cela, l'utilisateur envoie des ordres via la console de commande. Pour se faire, il existe plusieurs commandes pour réaliser des actions. La liste des commandes est résumée dans la commande `help`. La liste des commandes est la suivante :

```
debug
take <item_index>
help ?<order>
quit
save
load
info
talk <character_index>
open <crossing_direction>
list <target>
use <item_index> <target_character_index>
go <crossing_direction>
trade
look
player
```

On note que certaines commandes ne sont pas encore disponibles pour le moment. Le détail de leur méthode d'utilisation est résumé via la commande `help.`

Par exemple , la commande GO permet de déplacer le joueur dans la direction souhaitée. Il faut qu'il existe une liaison (crossing) dans cette direction. La signature est donc la suivante :

```
go <dir_char> (ex: go N)
```

L'argument <dir_char> représente une des directions cardinales (N/S/E/W). On note qu'il est impératif qu'il existe un passage dans la direction sélectionnée.

Pour obtenir la liste des liaisons, on peut utiliser la commande `look` ou `list crossing`. Le résultat est alors possiblement le suivant :

```
> look
This tile contains;
[2] crossings:
      [N] Door - close
      [S] Door - open
[1] characters:
      [0] Player (you)
[0] items:
      No item on this tile
```

```
> go S
```

Via l'exemple précédent, on affiche le détail de la case sur laquelle le joueur se trouve. On remarque alors qu'il existe une porte ouverte vers le sud. De ce fait, on utilise la commande `go S` pour se diriger dans cette direction.

4 Documentation

4.1 Diagramme UML

Notre diagramme UML est l'élément qui nous a demandé le plus de réflexion et le plus de temps. Une fois arrivé au point où notre diagramme nous semblait à la fois cohérent et correct, nous avons décidé de commencer le développement du jeu. Cependant, nous avons remarqué quelques erreurs mineures de conception, oubli d'attributs et méthodes, que nous avons dû modifier par la suite dans notre diagramme. Notre conception est répartie dans plusieurs paquets : Crossings, Game, Interpreteur, Items, Personnages, Tiles.

- Crossings :

  Le paquet Crossings répertorie tous les différents moyen de passer d'une zone (tile) à une autre.

- Game :

  Le paquet Game regroupe toutes les classes de logique de jeu. C'est ici qu'on s'occupe de l'initialisation des différentes constantes, qui appelle la création de la carte ainsi que celle des entités présentes dans le jeu. Il est aussi chargé de la gestion des tours de jeux et de l'impact des choix de l'utilisateur (commande interprétés par l'interpréteur qui appelle les méthodes du GameManager).

- Interpréteur :

  Le paquet Interpréteur regroupe les classes nécessaires à l'interprétation des commandes entrées dans la console par le joueur. La classe "Interpréteur" lit l'entrée (via la classe scanner) et crée une instance de Request avec les entrées. Request convertit alors l'entrée en ordre. Ensuite, l'interpréteur détermine la méthode à appeler (dans GameManager) en fonction du résultat contenu dans Request.

- Items :

  Le paquet Item regroupe toutes les classes relatives aux objets avec lesquels le joueur peut interagir et porter. Chaque objet à une utilisation et des conditions qui lui sont propres.

- Personnages :

  Le paquet Personnages regroupe toutes les différentes classes liées aux personnages et à leurs caractéristiques (points de vie, faim, le froid). C'est ici que se trouve la classe Player chargée de toutes les interactions avec le joueur et l'environnement.

- Tiles :

  Le paquet Tiles regroupe toutes les informations concernant les zones dans

lesquelles le joueur évolue.

4.1.1 Package Interpreteur

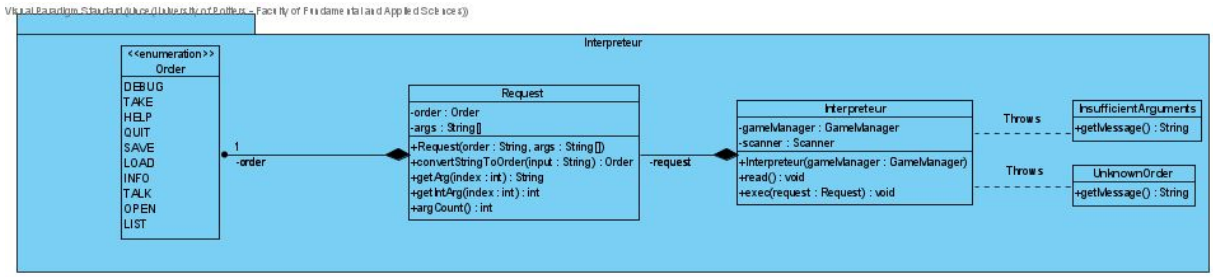

Figure 1 : Diagramme UML du package Interpréteur

4.1.2 Package Game

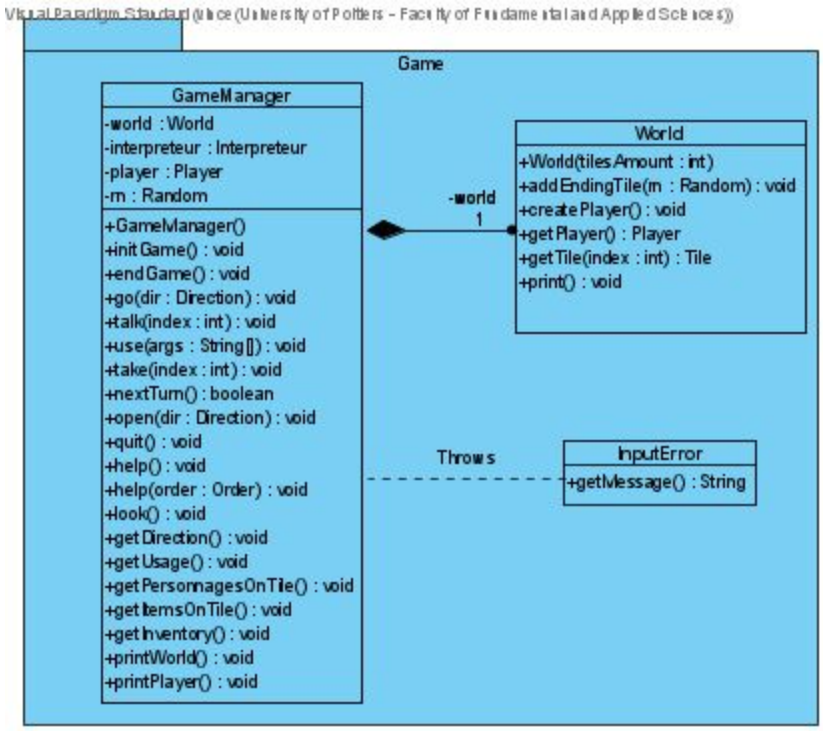

Figure 2 : Diagramme UML du package Game

4.1.3 Package Personnage

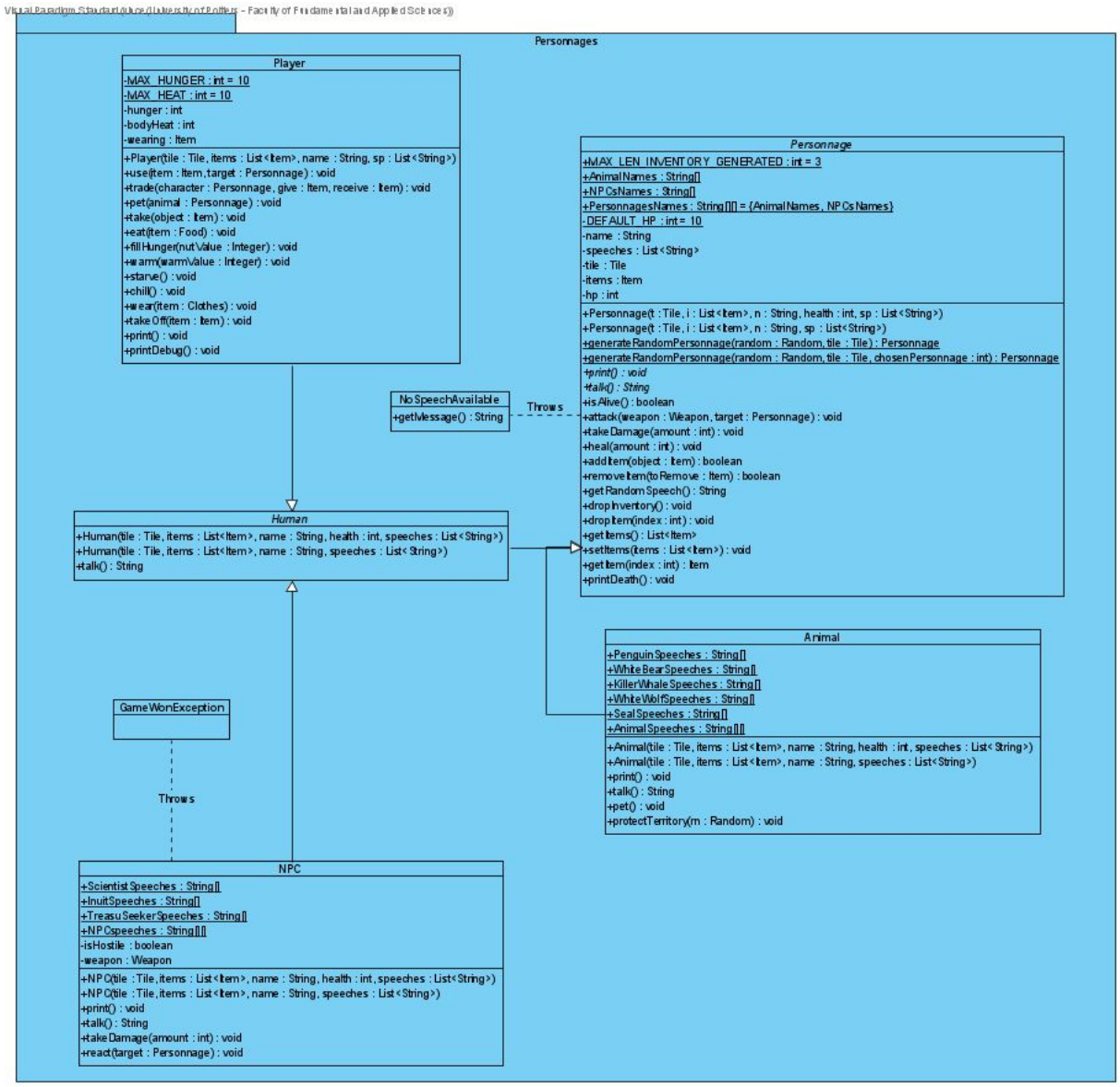

Figure 3 : Diagramme UML du package Personnage

4.1.4 Package Items

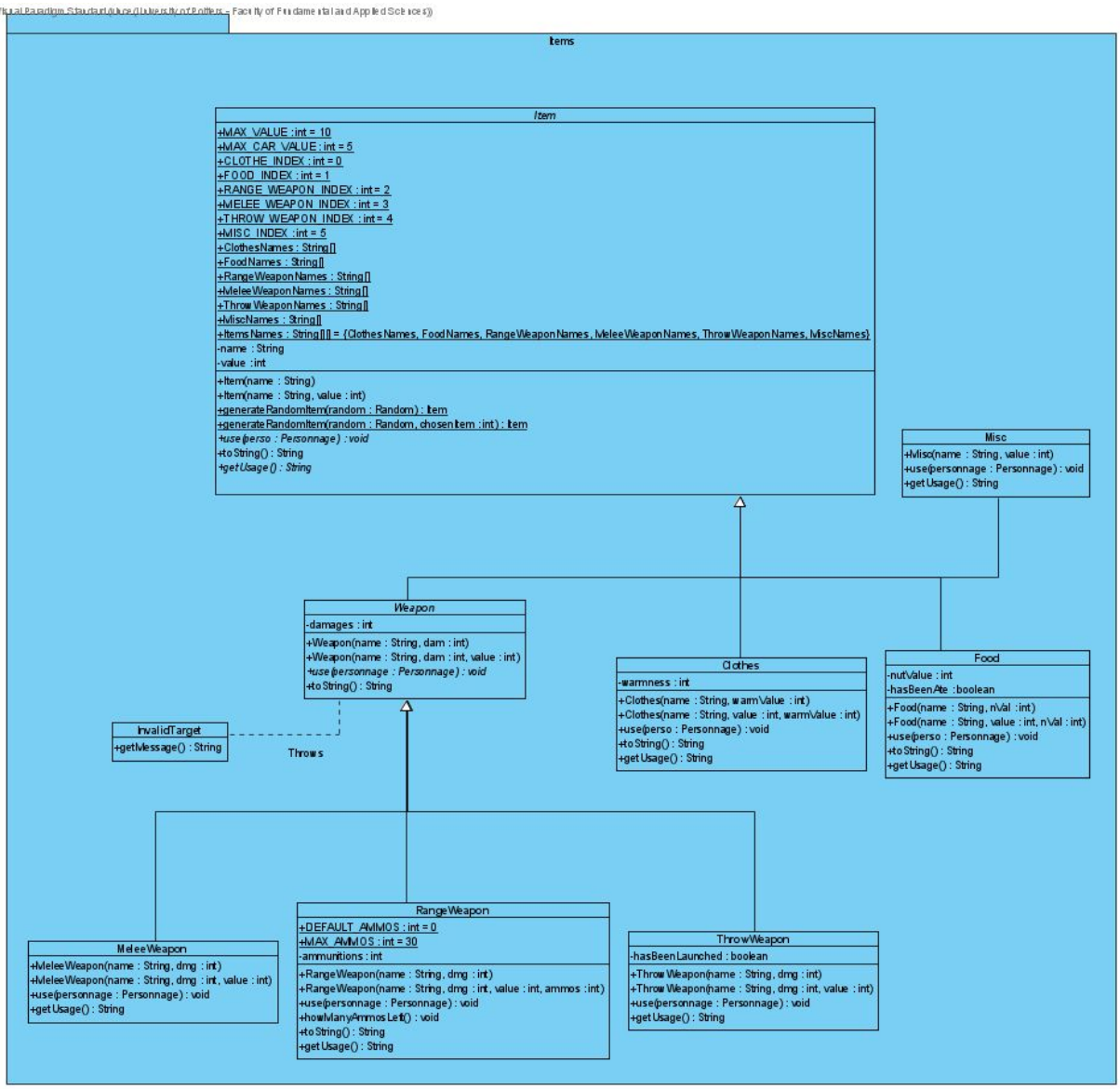

Figure 4 : Diagramme UML du package Items

4.1.5 Package Tiles

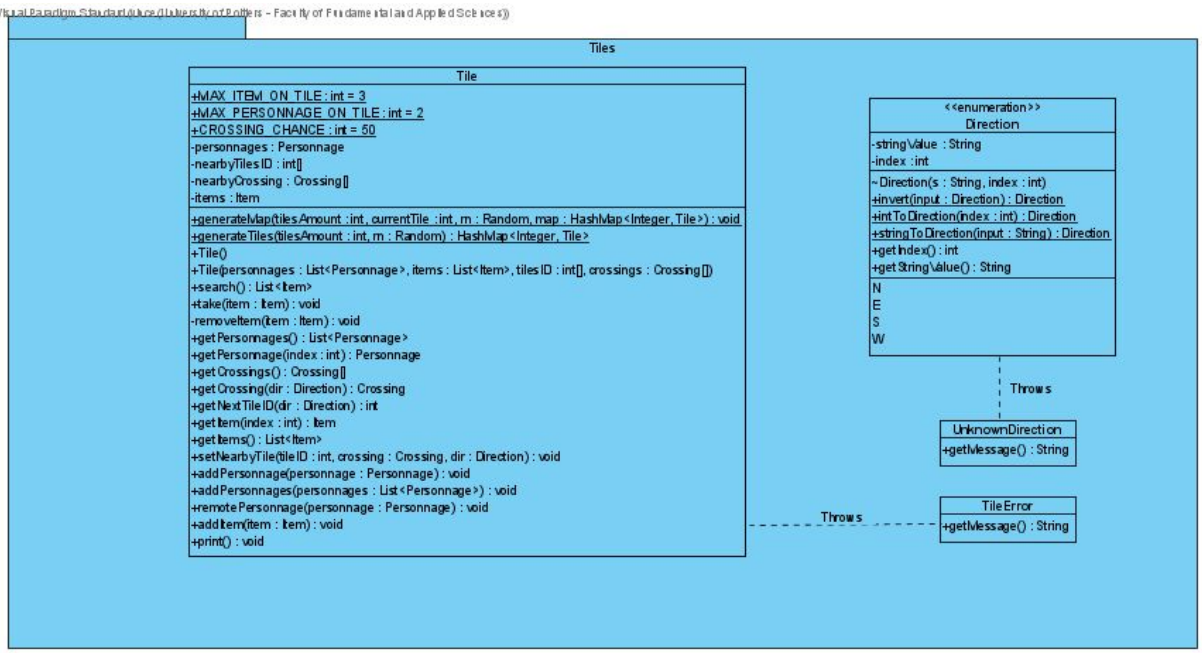

Figure 5 : Diagramme UML du package Tiles

4.1.6 Package Crossings

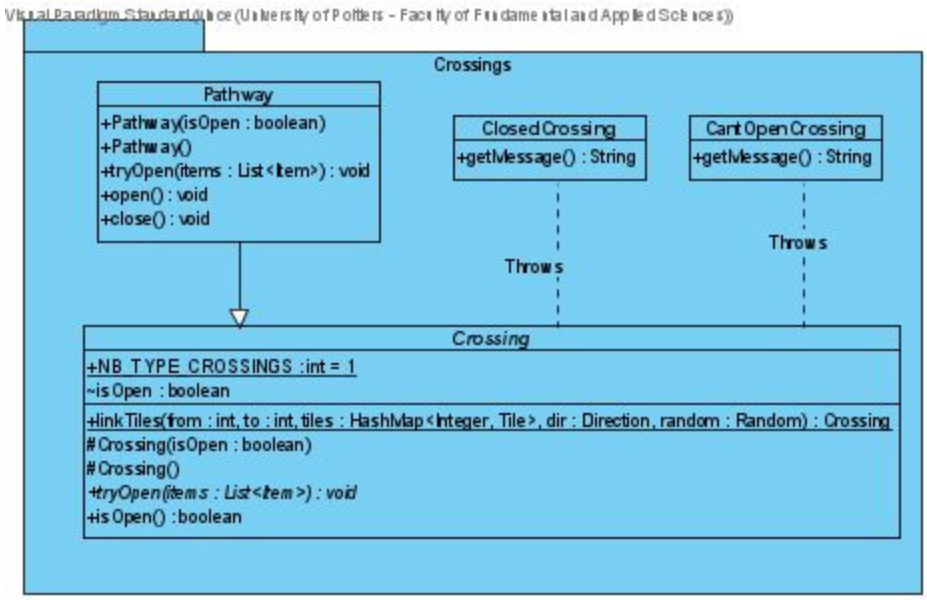

Figure 6 : Diagramme UML du package Crossings

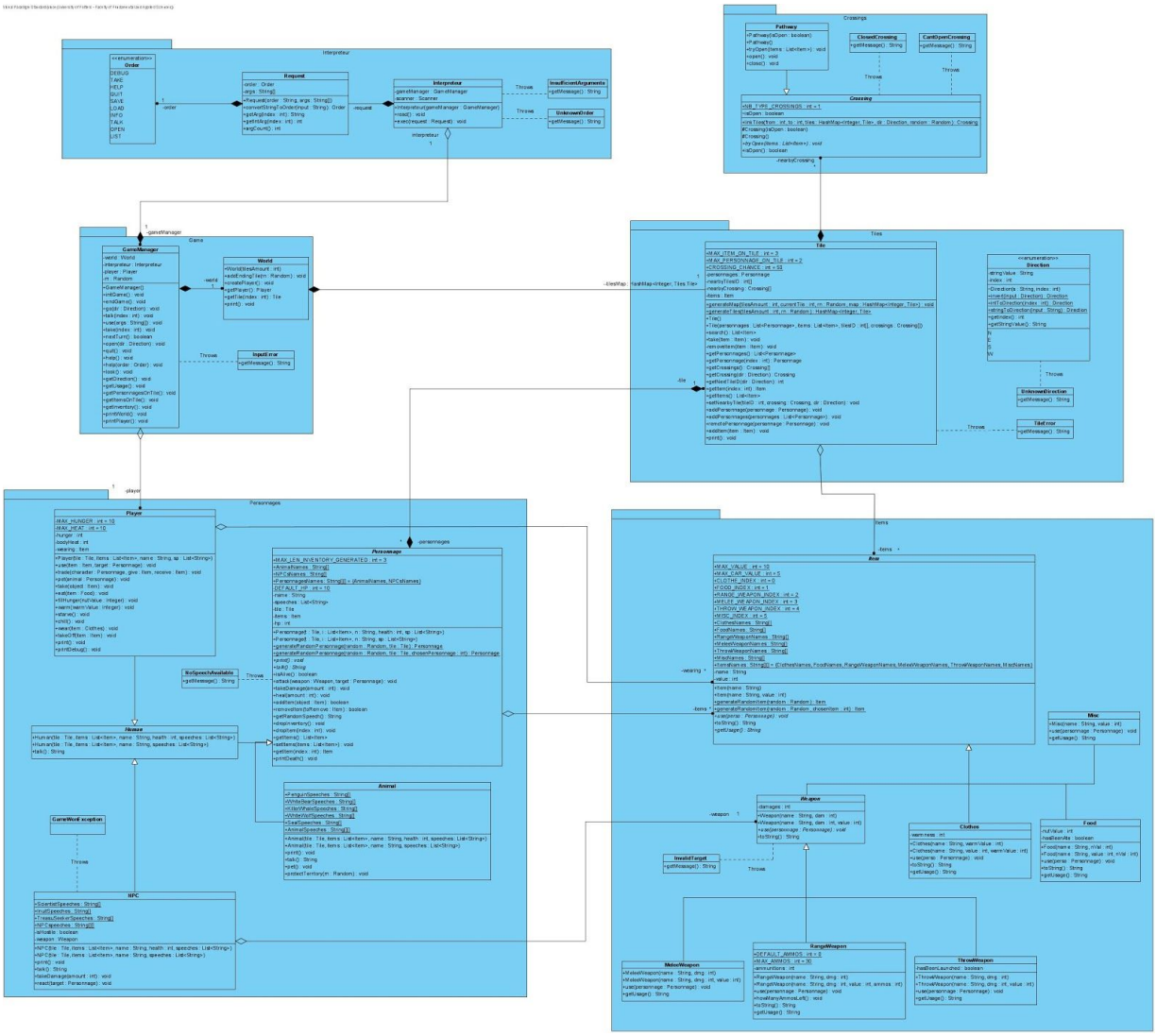

Figure 7 : Diagramme UML complet

4.2 Diagrammes d’états

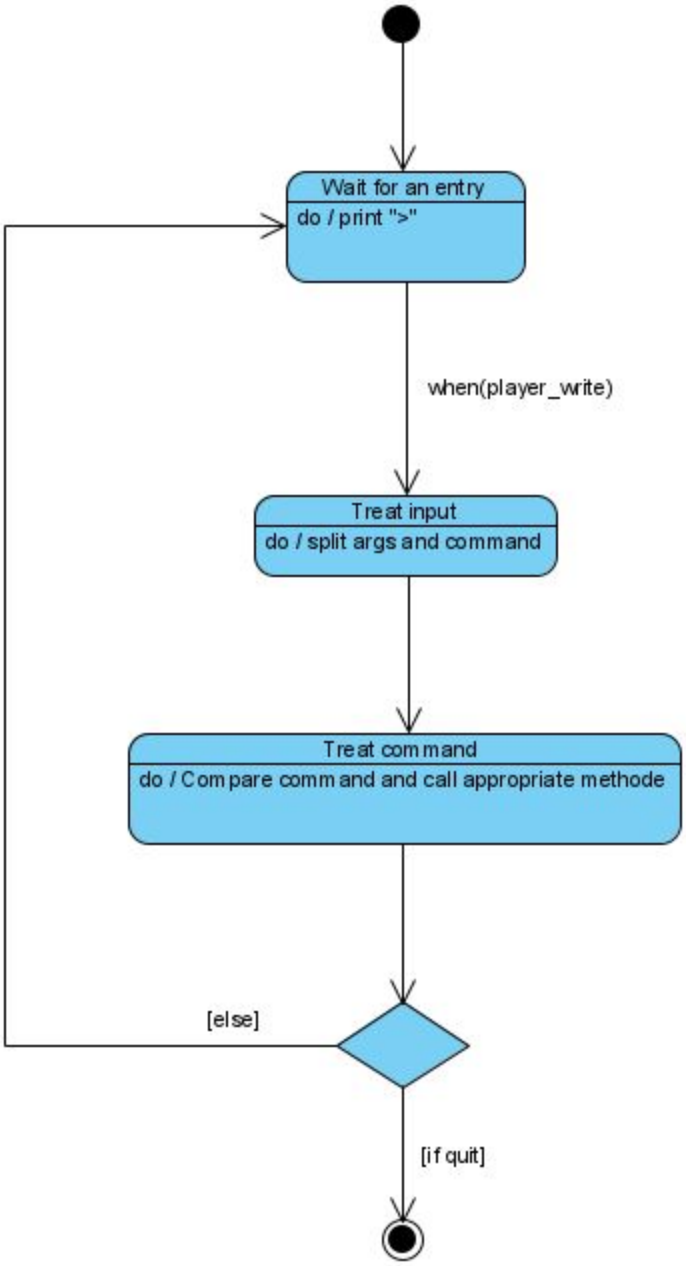

Figure 8 : Diagramme d’état de l’interpréteur

Ce diagramme représente le fonctionnement de l’interpréteur de commandes. Celui-ci attend que le joueur rentre une instruction dans la console. Une fois la ligne écrite récupérée par l’interpréteur, il la traite et appelle les méthodes demandées, ce jusqu’à la fin du programme.

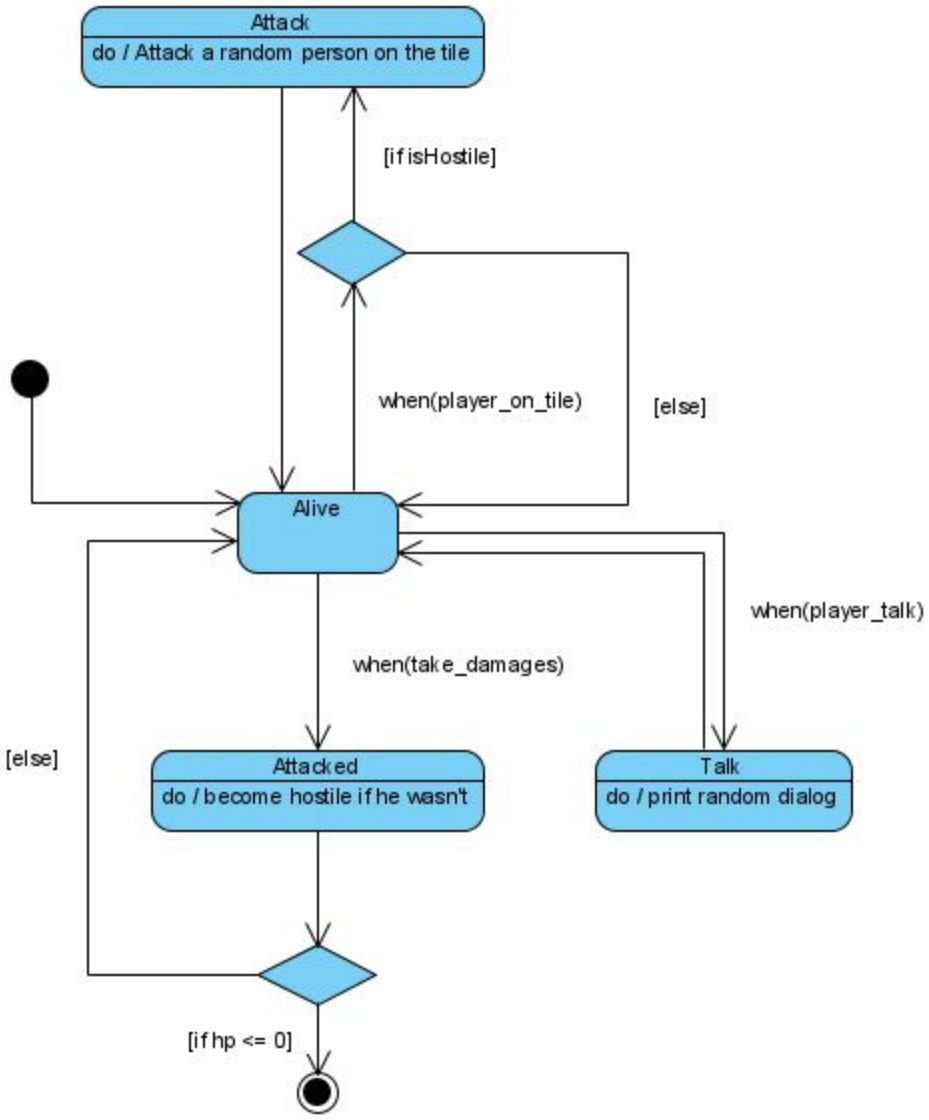

Figure 9 : Diagramme d'état d'un personnage non joueur (NPC)

Ce diagramme représente le fonctionnement d'un personnage non-joueur. Si celui-ci est en vie et que le joueur lui parle, il répond. S'il se fait attaquer, il attaque n'importe qui dans la zone, sinon il reste passif.

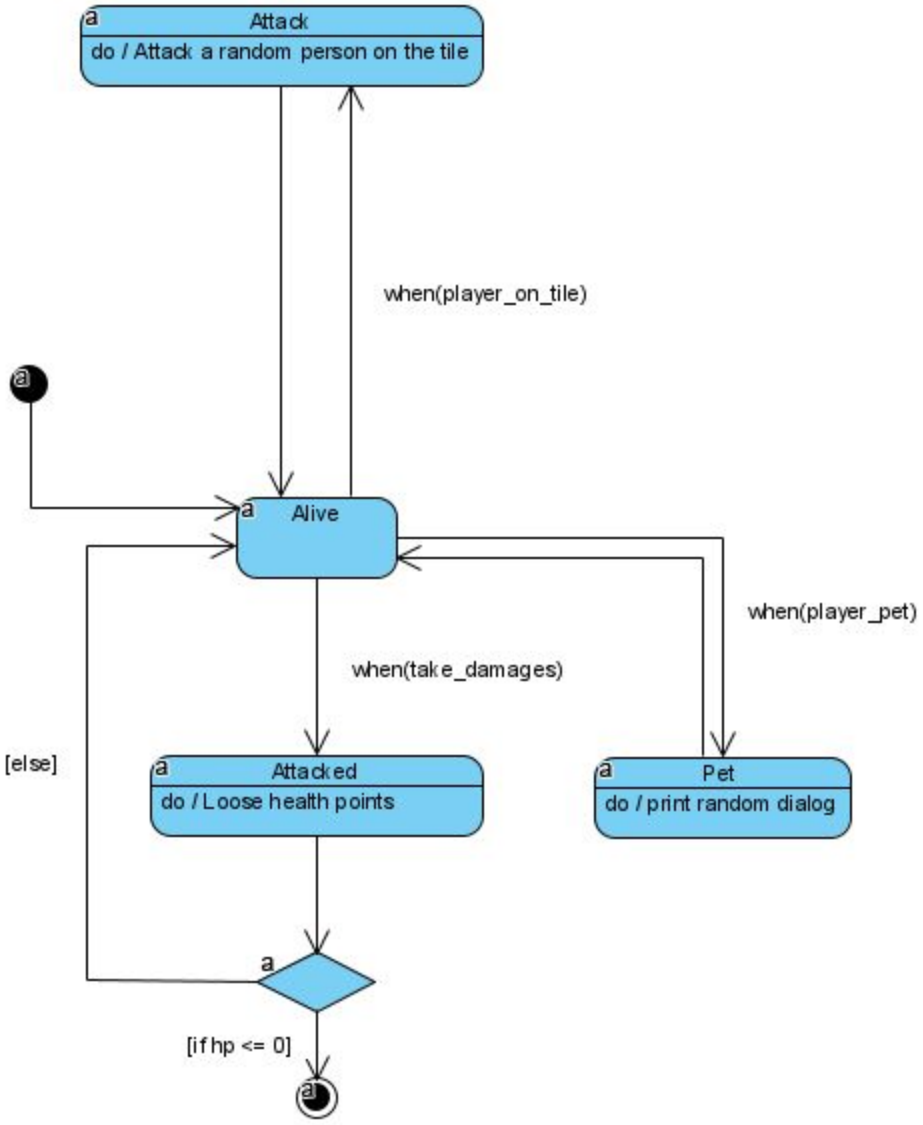

Figure 10 : Diagramme d'état d'un animal

Ce diagramme représente le fonctionnement d'un animal. Son fonctionnement est presque le même que celui d'un NPC à la différence que l'animal attaque à vue.

4.3 Diagrammes de séquences

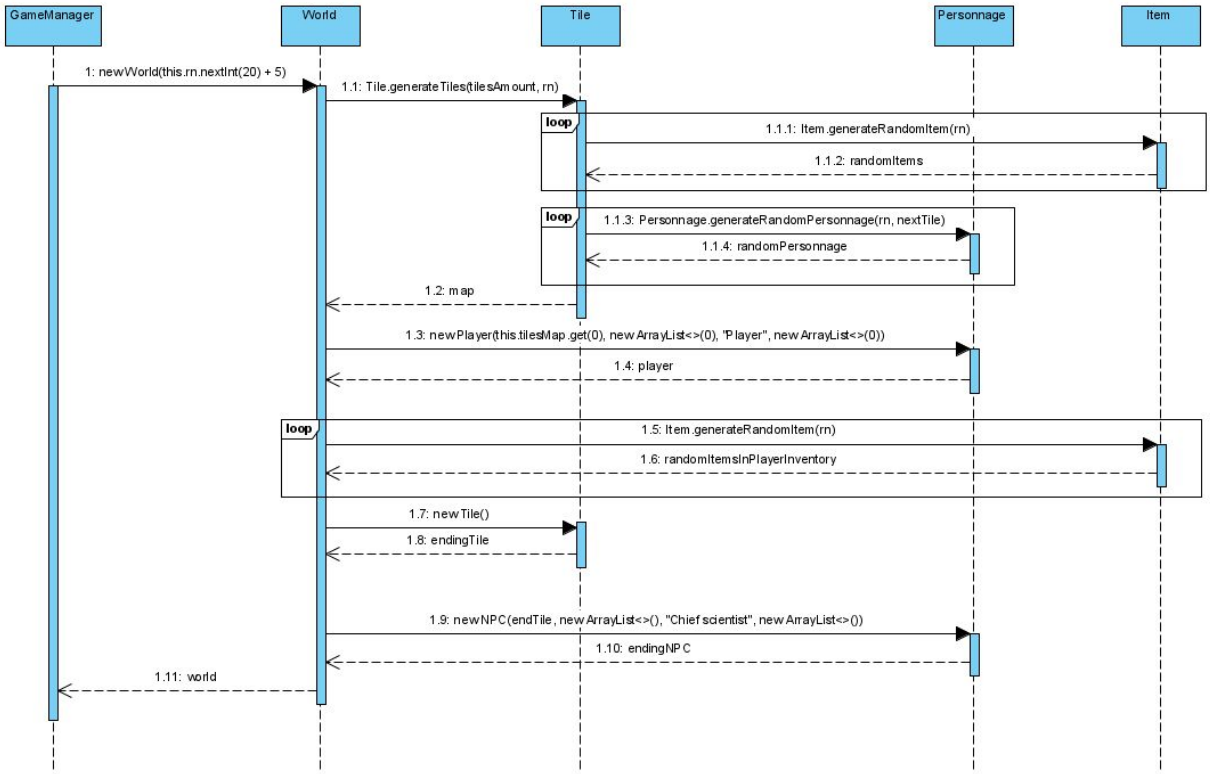

Figure 11 : Diagramme de séquence de la génération de la carte du jeu

Ce diagramme montre donc la génération du monde dans lequel le jeu prend place, il est généré procéduralement au lancement du jeu, le Game Manager créer un World d'une taille (ici) d'entre 5 et 25 tiles. Ce même World génère donc le bon nombre de tiles qui sont créées avec un nombre aléatoire d'items et de personnages. Une fois que le bon nombre de tiles est généré, le Player est ajouté au jeu sur la première Tile créée et se voit créé un inventaire par le World. Ensuite la case de fin est ajoutée en dernier et le pnj auquel il faut parler pour finir le jeu est ajouté sur cette dernière. Pour finir le World est retourné au Game Manager et le jeu peut commencer.

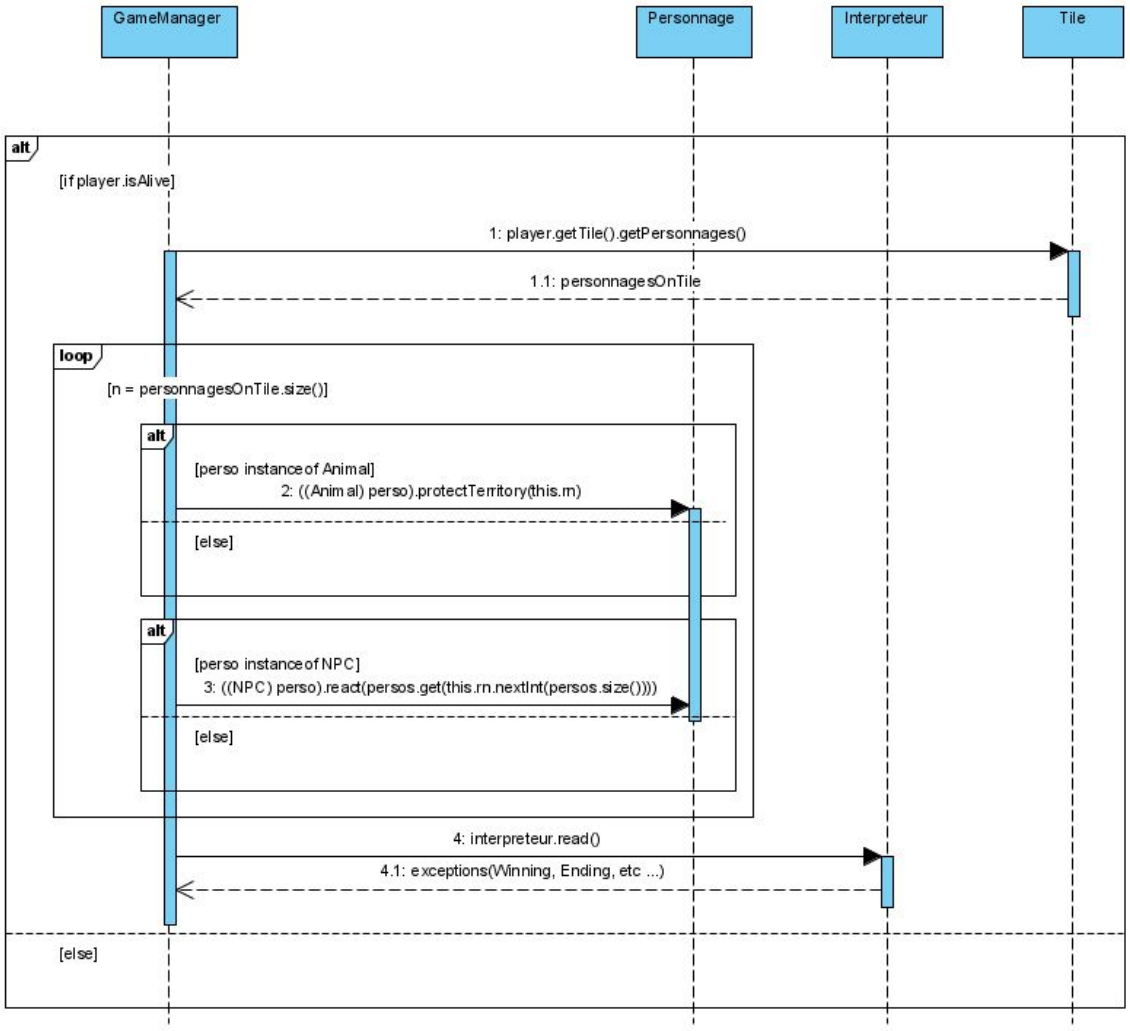

Figure 12 : diagramme de séquence d'un tour de jeu

Ce diagramme montre comment est géré chaque tour de jeu. On commence par vérifier si le joueur est encore en vie. Ensuite, nous récupérons la liste des personnages présents sur la même zone que le joueur pour gérer les différents combats qui pourraient s'y dérouler (chaque commande compte pour le joueur un tour). On regarde ensuite si les personnages présents dans la même zone un par un et on traite leur réactions avec la classe Personnage.

Une fois sorti de la boucle de combat des personnages non-joueurs, on appelle l'interpréteur pour demander la prochaine commande effectuée par le joueur.

4.4 Implémentation de fonctionnalités

Avec la structure actuelle de notre projet, il est aisé d'ajouter des fonctionnalités. On peut alors diviser les différents types de fonctionnalités que l'on peut implémenter.

4.4.1 Ajout d'ordre / commande

Pour ajouter une commande, il faut dans un premier temps ajouter un élément dans le type énuméré ORDER de la manière suivante : NOM_ORDRE("input requis", "brève description d'aide"). Par exemple, pour ajouter l'ordre MAP qui affiche toute la carte du jeu, on doit ajouter la ligne suivante : `MAP("map", "\"map\" to display the map of the world")`. Une fois la chose faite, il faut ajouter le cas le l'ordre MAP dans l'interpréteur (dans la méthode exec()). Le cas de la map pourrait alors appeler une méthode dans gameManager qui affiche la carte à l'écran.

4.4.2 Ajout de Crossing

Pour ajouter un type de crossing, il suffit d'ajouter une classe qui hérite de la classe abstraite Crossing (dans le package Crossings). La conception que nous avons adopté devrait permettre d'ajouter ce type d'objet facilement. Nous avion en premier lieu une vus plus complexe que nécessaire de la chose mais nous sommes rapidement arrivé à cette mise en place bien plus simple et tout aussi efficace

4.4.3 Ajout d'Item

Pour ajouter un item, il faut simplement ajouter une classe qui hérite du type souhaité. Il faut alors implémenter les méthodes abstraites. Il faut aussi ajouter le cas du type de l'objet ajouter dans la méthode use() de la classe player. On note que le fait de devoir vérifier le type de l'objet qu'on souhaite utiliser (utiliser la méthode use()) dénote un problème de conception que nous n'avons pas réussi à déceler lors de la mise en place de notre diagramme UML. En effet, pour avoir une conception beaucoup plus cohérente et efficace, nous ne devrions pas avoir à faire ce test de type pour l'objet.

4.4.4 Ajout de Personnage

Pour ajouter un nouveau type de personnage. Il faut créer une classe héritant de la classe "human" ou "animal" (ou de personnage si la classe rajoutée ne correspond à aucun des deux). Il faut ensuite implémenter les méthodes abstraites.

Pour pouvoir instancier cette nouvelle classe dans le monde du jeu, il faut rajouter un tableau de noms (à l'image de AnimalNames et NPCsNames) dans la classe personnage et dans le tableau à deux dimensions "PersonnagesNames". Il suffit ensuite de rajouter un cas au switch dans "generateRandomPersonnage".

5 Organisation et répartition du travail

Pour ce qui est de la répartition des tâches, nous sommes partis de notre premier jet de conception UML que nous avons mis en place ensemble et nous sommes organisés comme suit :

- Yann s'est occupé du joueur, étant l'élément central du programme, son fonctionnement changeait régulièrement .

- Vincent a instauré les NPC (personnages non-joueurs) et items, ces éléments rendent le jeu vivant et doivent interagir de la bonne manière avec le joueur et le monde qui les entoure. Il a également revu la génération du monde et de ses composantes (items, personnages, tiles, crossing).
- Louis s'est chargé du Game Manager ainsi que de l'interpréteur qui composent la structure du jeu et qui encadre tout le reste, l'interpréteur notamment, sans lequel l'utilisateur ne pourrait pas interagir avec le programme. Il s'est aussi chargé des tiles et des crossings.

Chacun a rédigé les tests liés à la partie qui lui était assignée et chacun aidait l'autre en cas de besoin, bien évidemment.
Enfin, la rédaction du compte rendu et la mise en place du diaporama est le résultat d'un travail collectif. Chacun d'entre nous a rédigé la documentation du code dont nous avions la responsabilité, et nous avons répartit également les différentes autres parties de la rédaction.

6 Conclusion

Nous sommes arrivés à bout du projet en rencontrant quelques difficultés notamment au niveau de la mise en place des passages entre les Tiles et notre conception de base n'est pas tout à fait celle que nous avons obtenue à la fin. Certaines fonctionnalités prévues ne sont pas encore intégrées mais les éléments principaux sont présents et le jeu est fonctionnel. Le projet nous a apporté encore un peu plus d'expérience dans le travail d'équipe et nous a permis d'utiliser la plupart si ce n'est toutes les notions vues en TD/TP. Il a été intéressant de constater que les choses pouvaient être faites de biens des manières et de savoir qu'il y en a encore auxquelles nous n'avons pas pensé.