Projet 3 – Labyrinthe

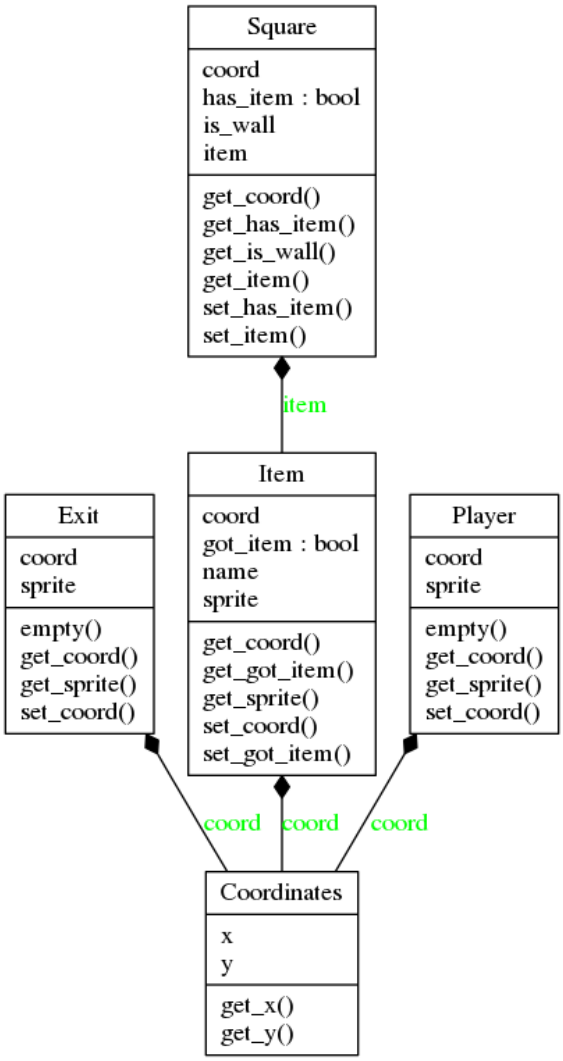
1. Raisonnement

J’ai commencé par imaginer les objets dont j’allais avoir besoin, et leurs attributs. Ensuite, la manière dont les conditions allaient se vérifier. Le plus simple pour moi était d’effectuer une vérification à chaque mouvement à l’aide de la capture d’événements de Pygame : ainsi à chaque mouvement, le programme vérifie la case sur laquelle le joueur veut se rendre. Si c’est un mur, le mouvement est bloqué. Si elle est libre, le programme vérifie si il y a un objet, dans quel cas l’objet sera considéré comme possédé, et si cette case correspond à la sortie, dans quel cas une vérification du nombre d’objets possédés est effectuée et le résultat de la partie affiché en conséquence. Finalement, à mes yeux, les parties les plus importantes sont la capture d’événements ainsi que la génération de la grille via le fichier texte.
La partie qui m’a causé le plus de soucis est certainement la partie graphique. En tant qu’utilisateur, je pense qu’on ne se rend pas forcément compte des choses que cela implique, comme par exemple l’animation du personnage qui n’est pas un simple déplacement mais un rafraîchissement de chaque élément de la fenêtre.

2. Classes et méthodes créées

J’ai fait le choix de créer 5 classes : Player, Exit, Coordinates, Item et Square

- Square : définit chaque case de la grille
- Item : définit les objets
- Player : définit le joueur
- Exit : définit la sortie
- Coordinates : stocke les coordonnées des différents éléments (cases, joueur, sortie, objets)



Méthodes

- generate_random_coordinates() : génère un set de coordonnées aléatoire, puis le retourne sous la forme d'un objet de type 'Coordinates'

- generate_grid() : convertit le fichier 'grid.txt' en une liste de caractères, crée un objet de type 'Square' par caractère, et selon la lettre lue, donne une valeur spécifique aux attributs de la case. Retourne la grille sous forme d'une liste d'objets de type 'Square'

- display_grid() : gère l'affichage de la fenêtre

- put_item_in_grid() : attribue un set de coordonnées aléatoires à un objet, puis vérifie que rien ne se trouve auxdites coordonnées. Si la case est occupée, la méthode est relancée avec de nouvelles coordonnées aléatoires.

3. Commentaires

Les classes 'Player' et 'Exit' ont exactement les mêmes attributs, mais par souci de clarté, j'ai préféré créer une autre classe au lieu d'utiliser la fonction d'héritage, car d'un point de vue pratique, ce sont deux entités différentes qui n'ont pas la même fonction.

La classe 'Coordinates' pourrait être supprimée et ses utilisations remplacées par un tuple ou une liste, mais à mes yeux, je trouve l'accès à cet élément plus facile et d'un point de vue pratique, cela fait plus de sens d'accéder aux valeurs x et y d'une coordonnée au lieu d'utiliser un indice de liste.

Certains attibuts se voient attribuer une valeur directement dans le constructeur : ces valeurs sont fixes et étant donné que certaines des classes ne seront appelées qu'une fois (comme 'Player' et 'Exit'), il m'a semblé judicieux de faire ce choix.

J'ai utilisé des getsetters pour l'accès et la modification des attributs des objets, mais après quelques recherches, il semblerait qu'en terme de sécurité, il ne soit pas nécessaire de le faire avec Python, contrairement à Java par exemple.