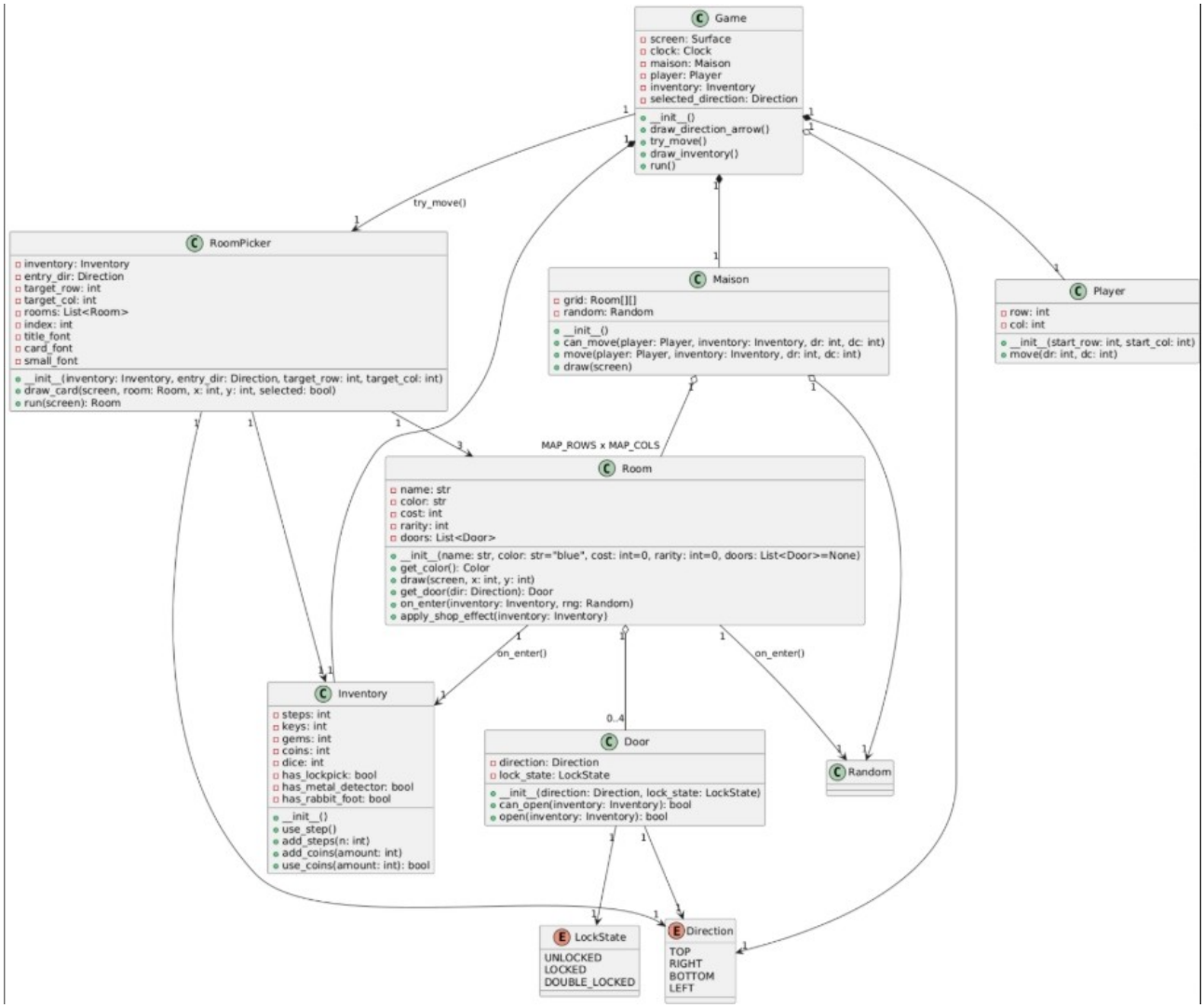
Rapport projet de Programmation

Diagramme UML

Sabrina Moulahcene
21204151
Elif Kaya
21213105
M1 Syscom 2025/2026

Accés GitHub
Id :https://github.com/elifkaya259-dotcom/projet_python.git
Mot de passe :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

Diagramme UML



Description du diagramme de classes UML et choix de conception

Le diagramme de classes ci-dessus décrit l'architecture objet du jeu que nous avons développé en Python. La classe centrale est `Game`, qui joue le rôle de contrôleur principal : elle gère la boucle de jeu, la fenêtre Pygame, les événements clavier ainsi que la coordination entre les autres objets. Pour cela, `Game` possède une composition vers `Maison`, `Player` et `Inventory`. Ce choix permet de bien séparer la logique « moteur de jeu » (boucle principale, affichage global) des données de la carte, de la position du joueur et de l'état de l'inventaire.

La classe `Maison` représente la grille de salles du manoir. Elle contient une matrice de `Room` et est responsable des règles de déplacement (`can_move`, `move`) ainsi que de la création des nouvelles salles. Nous avons choisi de regrouper cette logique dans une classe dédiée afin de pouvoir modifier la structure du manoir (taille de la grille, génération des salles) sans toucher au reste du code. Chaque `Room` modélise une salle individuelle, avec ses attributs (nom, couleur, coût, rareté, portes) et son comportement (`on_enter`, `apply_shop_effect`). Les `Room` contiennent elles-mêmes plusieurs objets `Door`, ce qui reflète bien la structure réelle : une maison contient des salles, et une salle contient des portes.

La classe `Player` est volontairement simple : elle ne stocke que la position (ligne, colonne) et fournit une méthode de déplacement. Toute la logique métier liée aux ressources est concentrée dans `Inventory`, qui gère les pas restants, les clés, gemmes, pièces, dés et objets spéciaux. Cette séparation respecte le principe de responsabilité unique : le joueur sait « où il est », l'inventaire sait « ce que le joueur possède ». Lorsque le joueur entre dans une salle, `Room` modifie l'`Inventory` (gain/perte de ressources) sans avoir besoin d'accéder à la classe `Game`.

Les classes `Door` et `RoomPicker` complètent le cœur du modèle. `Door` encapsule l'état de verrouillage d'une porte et la logique d'ouverture (`can_open`, `open`) en fonction de l'inventaire (clé, lockpick, etc.). `RoomPicker` gère l'interface de sélection d'une nouvelle salle lorsque le joueur ouvre une porte vers une case encore vide : cette responsabilité est isolée dans une classe dédiée pour ne pas surcharger `Game` ou `Maison` avec des détails d'affichage et d'interaction. Enfin, les énumérations `Direction` et `LockState` structurent les valeurs possibles pour les directions (haut, droite, bas, gauche) et les différents niveaux de verrouillage. L'utilisation d'enums rend le code plus lisible et réduit les erreurs par rapport à l'utilisation de simples constantes entières ou chaînes.

Dans l'ensemble, ce découpage en classes permet d'obtenir un code plus modulaire et plus facile à faire évoluer : la boucle de jeu, la génération du manoir, l'inventaire, les salles et les portes sont clairement séparés, tout en étant reliés par des relations de composition et d'association visibles dans le diagramme UML.