AVIGNON UNIVERSITÉ

Projet Quixo

L'étudiant :
Mustapha Genouiz

09/01/2025

MASTER D'INFORMATIQUE
INTELLIGENCE ARTIFICIELLE (IA)
ECUE Application de Conception

Responsable
Mickael Rouvier

UFR
SCIENCES
TECHNOLOGIES
SANTÉ

AVIGNON UNIVERSITÉ
CERI

CENTRE
D'**ENSEIGNEMENT**
ET DE **RECHERCHE**
EN **INFORMATIQUE**
ceri.univ-avignon.fr

Sommaire

1 Introduction

Ce projet a pour objectif de développer une application de jeu Quixo. Le jeu commence avec un plateau vide de taille 5×5.

Ensuite, en implémentant une conception intégrant trois design patterns, de faire fonctionner le jeu selon les règles de Quixo pour différents modes de jeu (Joueur Vs Joueur ou Joueur Vs IA), et d'aboutir à un résultat final fonctionnel. Le développement de ce projet a été réalisé en utilisant le langage de programmation Python.

2 Analyse du Problème

Le jeu *Quixo* se déroule sur un plateau 5x5, où chaque joueur cherche à former un alignement de cinq symboles (X ou O). À chaque tour, il sélectionne un cube en bordure, vide ou déjà à son symbole, puis l'insère sur l'une des extrémités de la même ligne ou colonne, provoquant un décalage des cubes. Le premier à obtenir un alignement complet de ses symboles gagne la partie. L'objectif est de concevoir une application modulaire permettant une gestion claire du plateau, un affichage intuitif (cases disponibles, highlighted) et la possibilité d'incorporer ultérieurement une intelligence artificielle.

3 Diagramme de classe

Ci-dessous est le diagramme de classe que j'ai implémenter pour la résolution du jeu Quixo.

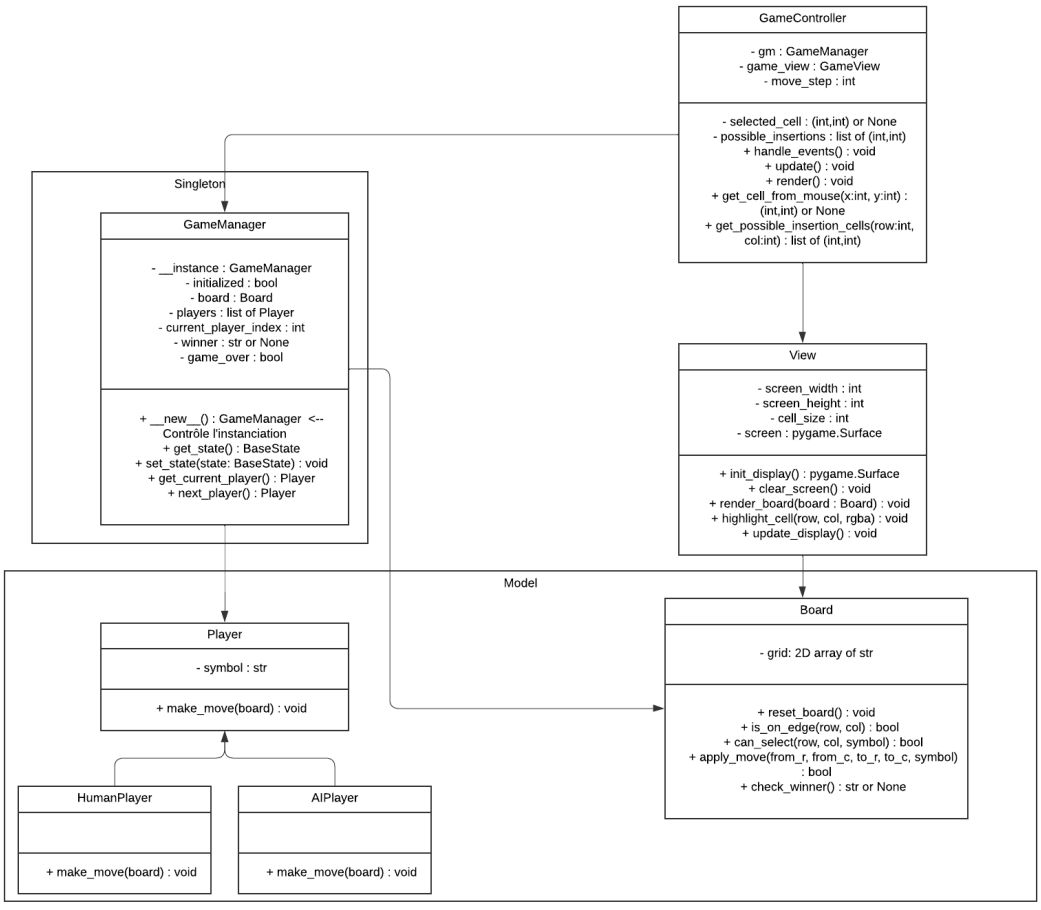

Figure 1. Diagramme de classe

- **GameManager** : assure la coordination globale du jeu (plateau, joueurs, état de la partie) et garantit l'unicité de l'instance (Singleton).
- **Board** : représente la grille Quixo 5×5 et gère les règles (sélection des cubes, insertion, vérification des alignements).
- **Player** : définit l'interface commune pour tous les joueurs, avec la méthode `make_move(board)`.
- **HumanPlayer** : implémente un joueur humain, dont les actions s'appuient sur les clics interceptés par le contrôleur.
- **AIPlayer** : illustre un joueur à logique automatisée, potentiellement géré par un algorithme IA.
- **GameView** : se charge de l'affichage PyGame (dessin du plateau, des pions, des surbrillances).
- **GameController** : intercepte les événements (souris, clavier), appelle les méthodes du `Board` et demande à la `GameView` de mettre à jour l'affichage.

4 Choix des Design Patterns

Nous avons opté pour l'utilisation de trois patrons de conception afin de structurer au mieux notre application *Quixo* :

- **Singleton** : le `GameManager` est conçu de façon à n'exister qu'en une seule instance, assurant une gestion centralisée du plateau, de la liste des joueurs et de l'état global de la partie. Ce choix simplifie l'accès aux composants clés du jeu et empêche toute duplication ou incohérence de l'état.

- **MVC (Model-View-Controller)** :
  - *Model* : la classe `Board` (et, le cas échéant, les joueurs) encapsule la logique métier (sélection, insertion, vérification de victoire).
  - *View* : la classe `GameView` se consacre à l'affichage sans manipuler directement la logique du jeu.
  - *Controller* : la classe `GameController` intercepte les événements, applique les opérations au `Board` et met à jour la `GameView`.

  Cette séparation des responsabilités facilite la maintenance et l'extension du code (par exemple, ajouter un nouvel affichage sans toucher aux règles du jeu).

- **Strategy** : la hiérarchie `Player` (classe abstraite), `HumanPlayer` et `AIPlayer` propose une interface unifiée via la méthode `make_move(board)`. Même si, dans l'état actuel du projet, seul un joueur humain est pleinement fonctionnel, l'existence de ce patron de stratégie garantit une intégration future aisée d'un joueur doté d'un algorithme d'IA. Ainsi, nous pourrons remplacer ou compléter `HumanPlayer` par `AIPlayer` sans refondre l'architecture du jeu.

Ces trois patterns se complètent et assurent une application modulaire, évolutive et simple à maintenir : `Singleton` centralise la gestion, `MVC` sépare clairement la logique métier de l'affichage et du contrôle, et `Strategy` permet de substituer ou enrichir la logique des joueurs (humain, IA) sans impacter les autres composantes.

5 Fonctionnalités Développées

- **Gestion du plateau** :
  - Initialisation d'un tableau 5×5
  - Mécanisme de sélection et d'insertion de cubes en bordure
  - Décalage automatique des cases lors de l'insertion
- **Contrôle du flux de jeu** :
  - Gestion du tour de chaque joueur
  - Possibilité de sélectionner uniquement les cubes autorisés (bordure, vide ou au symbole du joueur)
  - Détection de la fin de partie (alignement de 5 symboles)
- **Affichage et interaction** :
  - Mise en évidence (*highlight*) des cases sélectionnables et des positions d'insertion
  - Actualisation de l'interface graphique (dessin du plateau, pions, couleurs)
  - Affichage d'un message de victoire si un joueur remplit les conditions de gain

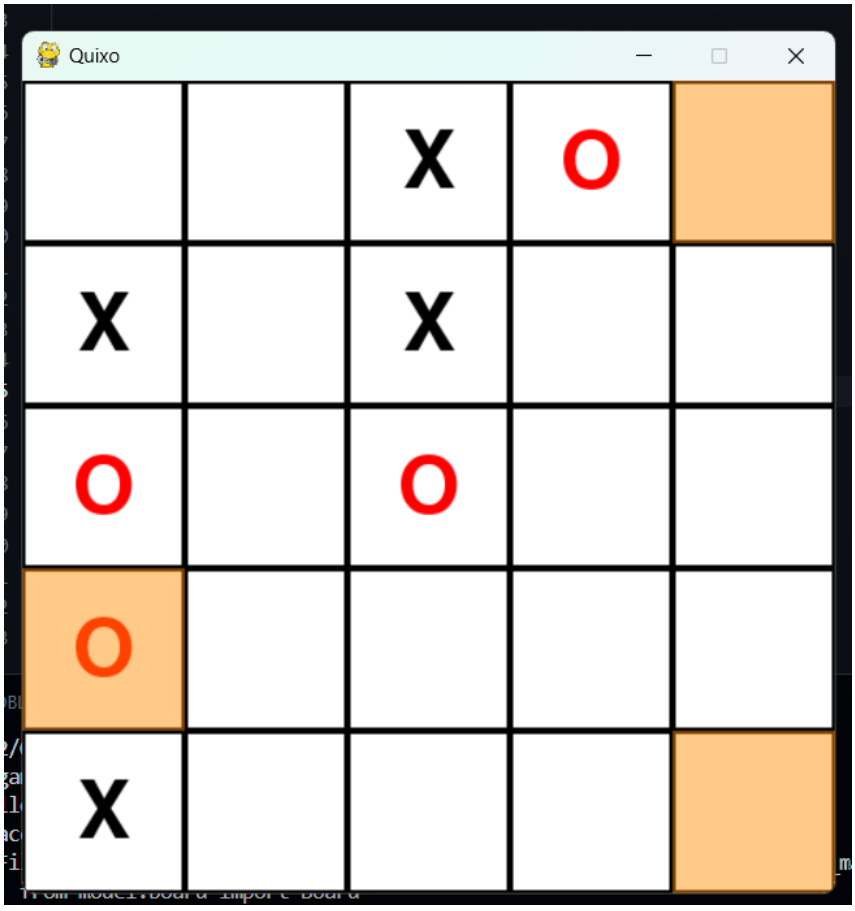

Figure 2. Affichage de réalisation

6 Conclusion

Ce projet m'a permis de réviser le cours de conception et de me remémorer l'importance des différents design patterns pour construire une architecture à la fois modulaire et optimisée. Malheureusement, je n'ai pas pu implémenter un algorithme d'IA : la principale difficulté est venue du manque de temps, car je devais aussi gérer d'autres projets, mes examens et ma mission en alternance.

Cela dit, j'ai déjà plusieurs idées pour réaliser un algorithme de type min-max, en commençant par un board pondéré, ce qui enrichirait grandement l'application. J'envisage également d'ajouter des fonctionnalités graphiques comme un bouton de step-back et différentes animations (visuelles et sonores) pour rendre l'expérience utilisateur plus agréable.