AVIGNON UNIVERSITÉ

Rapport Projet – Application de conception

Sarra Bensafi

6 janvier 2022

Master 1
Master Intelligence Artificielle
UE Application de conception

Responsables
Mickael Rouvier

UFR
SCIENCES
TECHNOLOGIES
SANTÉ

AVIGNON UNIVERSITÉ
CERI

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

Sommaire :

1. Analyse du problème

1.1. Définition et information sur le jeu Quixo

Chaque cellule de la grille aussi appelée tuile, peut être vide, ou marquée par le symbole d'un joueur : soit X, soit O.

Le quixo est une grille également appelée plateau de 25 cases. Cette grille de jeu est remplie avec des pions " vide" et chaque joueur choisit un symbole : rond ou croix.

A tour de rôle pour chaque tour, le joueur prend une tuile (vide ou avec son symbole) de la bordure, puis l'insère, avec son symbole, dans la grille en déplaçant les tuiles existantes vers le vide créé par le retrait précédent.

Le joueur gagnant est le premier à créer une ligne de tuiles portant toutes son symbole, horizontalement, verticalement ou en diagonale.

1.2. Objectif principal

L’ objectif principal est de résoudre le Quixo en considérant 2 type de jours un humain, et un algorithme d'IA, avec une plus grande probabilité que le deuxième joueur IA soit le gagnant

1.3. Quelques questionnement soulevées

Le plateau est dynamique, le X ou un O dans le plateau peuvent changer d'emplacement au cours des tours suivants. Cet aspect de dynamicité rend le jeu assez difficile pour un joueur humain de planifier plus d'un tour.

Un problème qui peut aussi être soulevé c'est qu'on peut ne pas terminer le jeu, un problème de non déterminisme. Par exemple, prendre et déplacer toujours les mêmes tuiles et n'avoir qu'une ou deux tuiles.

2. Diagramme de classe et choix de conception

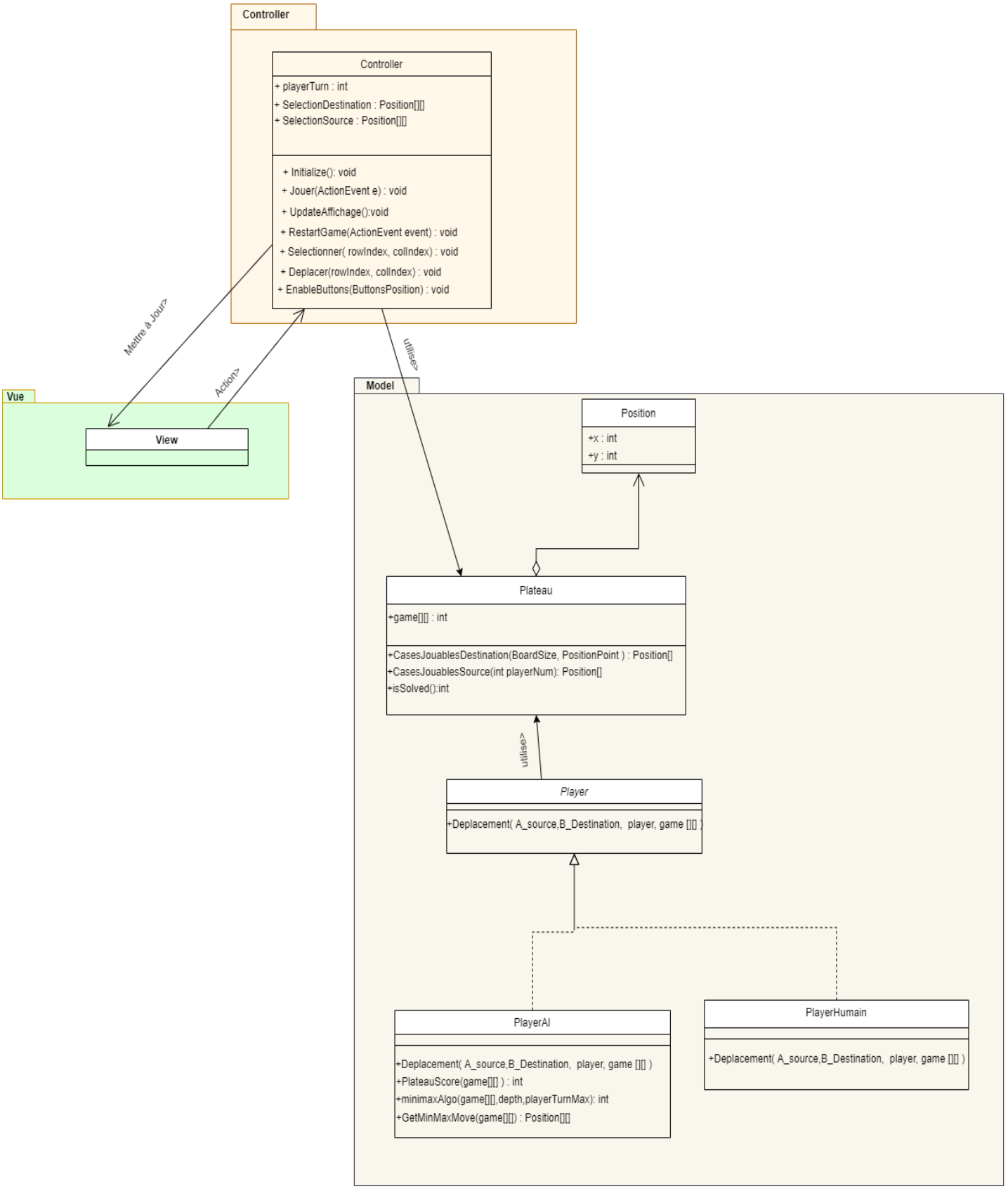

Figure 2.1 Diagramme de Classe

2.1 Choix de Conception

Pour rendre ce projet plus organisé et mieux structuré nous avons opté pour l'utilisation d'un ensemble de design pattern qui sont les suivants :

1) Design Pattern : MVC

Comme il existe une division du code entre les trois niveaux(Model View Controller)séparant ainsi la logique métier et l'interface utilisateur, il est extrêmement facile de diviser et d'organiser la logique de l'application.

MVC donne la possibilité de modifier facilement l'ensemble de l'application. L'ajout/mise à jour de nouvelles fonctionnalités.

On peut aussi aisément gérer et afficher plusieurs vues du même modèle(dans notre cas ce n'est pas très utile car nous n'avons qu'une seule vue).

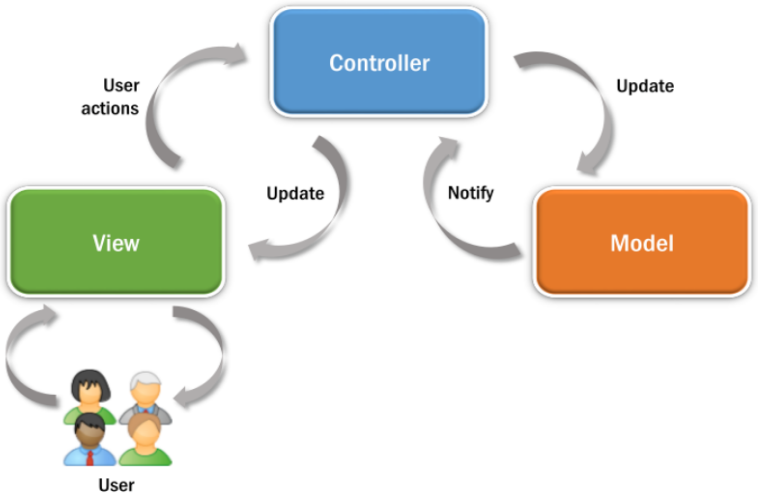

Figure 2.1 schéma MVC

Mise en ouvre MVC :

➔ View

Le view s'occupe de l'affichage du Plateau et n'effectue aucun calcul.

Il se contente de récupérer le tableau **game** a chaque état du jeu pour savoir quoi afficher sur les tuiles représentant les **mise à jour** envoyer par le controller, les tuiles sont les boutons qui font appel à la fonction **Jouer** représentant ainsi **les actions de l'utilisateur**.

➔ Modèle

Le Modèle c'est l'ensemble de classes et méthodes procurant des **fonctionnalités**, qui permettent d'accéder aux données dans notre cas c'est le tableau "**game**" qui est manipulé et modifié par un ensemble de méthodes.

Dans la classe Plateau je définis la méthode qui nous permet de donner les tuiles que le joueur actif peut sélectionner(**source**), la méthode qui définit les emplacements de **destination** pour cette case sélectionnée par le joueur et la méthode qui vérifie si le jeu est **fini** et qui a **gagné**.

Les deux classes **`PlayerAI`** et **`PlayerHumain`** qui implémente la classe Player.

Dans PlayerHumain on s'appuie sur les intéractions de l'utilisateur avec notre UI mais dans la Classe PlayerAI j'ai utilisé l'algorithme **`MiniMax`** afin de définir le meilleur coup

➔ **Controller**

C'est une sorte d'intermédiaire entre le modèle et la vue, le contrôleur va reçevoir **les signaux en provenance de la vue(bouton pressé)** qui modifient les données et renvoie le résultat à afficher à la vue.

2) Design Pattern : Singleton

Le design le mieux adapté est le singleton, puisque nous n'avons besoin que d'une seule instance de notre classe Controller qui vas être appelé tout au long de notre exécution, grâce à cette partie du code

dans view.fxml :

```
<AnchorPane . . . .
fx:controller="com.example.quixo.Controller">
```

C'est aussi le seul gestionnaire de configuration et mise à jour dans le View et modèle.

3) Design Pattern : stratégie

Comme on a besoin de deux variantes d'un algorithme, c'est le patron stratégie qui vas permettre de modifier le comportement du traitement durant l'exécution du logicielle, puisque l'interface **Player** définit la nature du sous traitement qui peuvent être modifiées pendant l'exécution, **PlayerHumain** contiendra la méthode simple de déplacement, mais la class **PlayerAI** contiendra une méthode déplacement, mais avec en plus des méthodes comme calcul score, minimaxAlgo et GetMiniMaxAlgo.

3. Les différentes fonctionnalités logicielles développées

A. Controller

- **`La classe controller:`**

Il contient les **fonctions** responsables de **la séquence des interactions** avec l'utilisateur, il communique avec la vue en lui envoyant les modifications après chaque coup d'un joueur.

Fonction	Rôle
`Initialize`	permet au contrôleur d'instancier les objets essentiels au démarrage de la partie.
`Jouer`	correspond au clic que fait l'utilisateur dans le board, alternant ainsi entre les deux étapes d'un coup du joueur, avec *Selectionner* et *Déplacer*.
`Restart`	permet de recommencer le jeu et réinitialiser les composants.
`sélectionner`	permet de sélectionner une case afin de faire le déplacement

Déplacer	Correspond au deuxième clique de même joueur sur un bouton ce qui correspond au mouvement d'une case sélectionnée.

B. Modèle

- **La classe Plateau:**

Cette classe est la responsable de traitement d'un plateau de jeu et de différents calcules de jeu. Elle contient l'ensemble des fonctions suivants :

Fonction	Rôle
CasesJouables Source	Détermine les cases qu'un joueur peut jouer en comparant les cases avec le même symbole, ou les cases vides se trouvant en bordure.
CasesJouables Destination	La fonction qui détermine les cases destinations où il est possible de mettre la tuile sélectionnée au premier coup.
IsSolved	La fonction qui détermine si le jeu est fini et si oui qui a gagné, en vérifiant les lignes colonne et les diagonales.

- **La classe Position:** détermine les coordonnées **x** et **y** qui aident ainsi à définir l'emplacement des tuiles et à les déplacer plus facilement.
- **La classe PlayerHumain:** la méthode **déplacement** qui selon le joueur actif déplace la tuile sélectionnée vers la case choisie par le joueur.
- **La classe PlayerAI:**

Le but est de **simuler tous les coups et de récupérer leur score**. Le meilleur coup à jouer sera celui avec le meilleur score. Elle contient un ensemble de fonctions :

Fonction	Rôle
MiniMaxMove	Détermine le meilleur coup à jouer, en augmentant ainsi la probabilité que le joueur **“Max”** gagne.
GetPlateauScore	Calcule le score courant de plateau.
minimaxAlgo	prends en paramètre l’état d’un plateau, la profondeur de calcule et quel joueur va jouer (min ou max) et elle retourne le meilleur score finale après un nombre de mouvement données (profondeur).
cloneGame	une fonction pour re-copier le plateau de jeu afin d’éviter le problème de référencement dans les tableau en java (et pour ne pas créer une fonction unplay)
deplacement	une fonction selon les méthode de l’IA

Le pseudocode de l’algorithme **minimaxAlgo** est le suivant :

```
fonction minimaxAlgo(plateau, profondeur, playerTurnMax){
    si(depth = 0 or plateau présent un gagnant)  alors
        return score de plateau.
    si (playerTurnMax) alors {
        valeur := Integer.MIN_VALUE;
        // générer les cases jouables et leurs destinations.
        pour chaque case jouable et destination {
           déplacer dans temp
           valeur := max(valeur,minimax(temp, profondeur-1,MIN))
        }
    }
    sinon(playerTurnMin) {
       valeur := Integer.MAX_VALUE
       // générer les cases jouables et leurs destinations.
       // faire ce déplacement
       pour chaque case jouable et destination{
          déplacer dans 123265temp
          valeur:=min(valeur,minimaxAlgo(temp, profondeur-1,MAX))
```

```
        }
    }
    return valeur
}
```

C. Vue

Dans l'application on a une seule vue, qui se constitue de 25 boutons représentant les tuiles du jeu et un bouton à fin de recommencer "Restart Jeu".
Un Label affichant le joueur courant et un autre pour annoncer le gagnant du jeux.

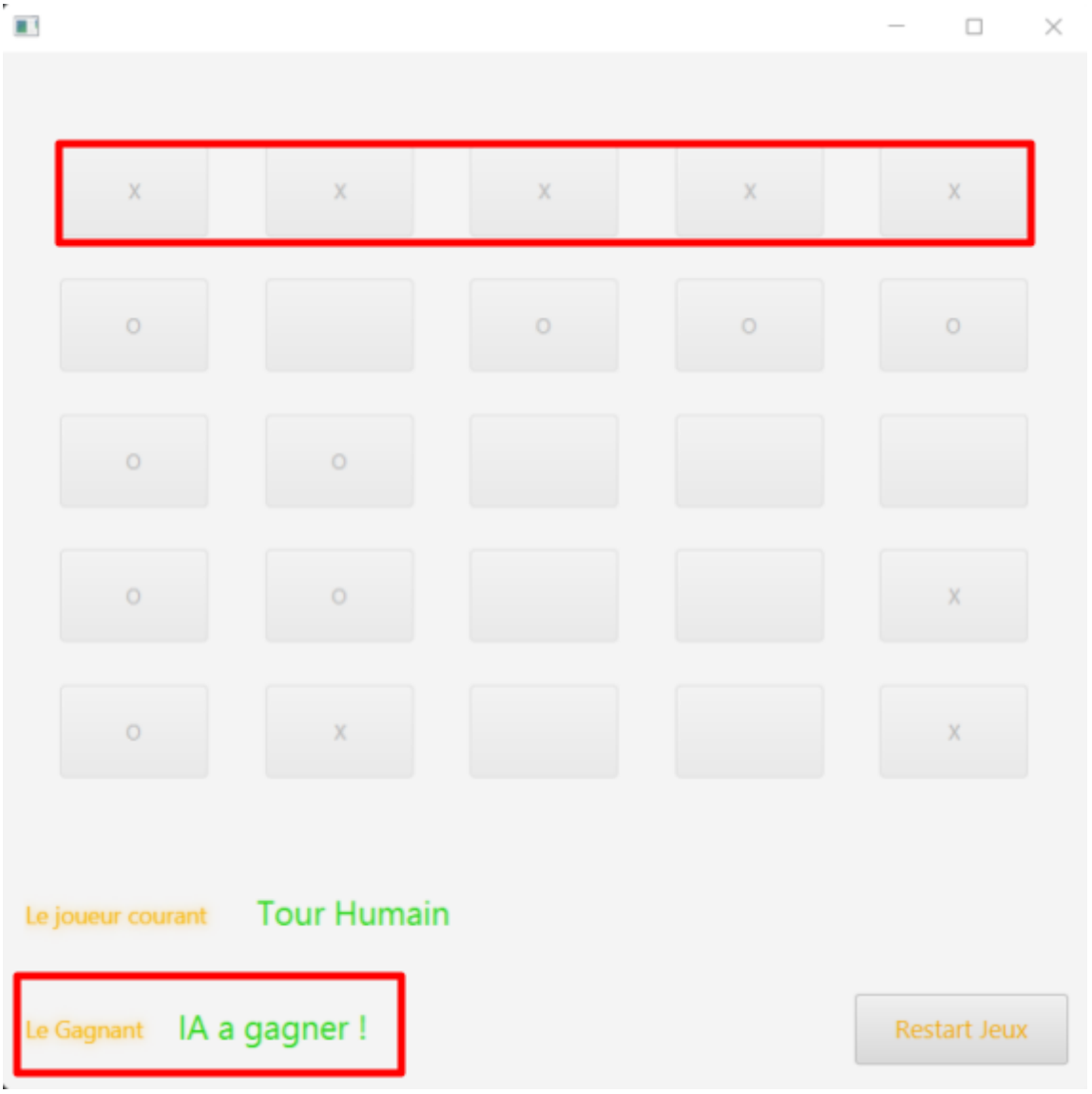

Figure 3.1 Exemple Vue

4. Les difficultés rencontrées et les améliorations possibles:

Parmi les améliorations possibles sur notre application :

- **Optimiser la recherche** du meilleur mouvement et indirectement, le calcule du meilleur score pour savoir si un déplacement est gagnant ou pas.

  Nous avons essayé de créer cette dernière en suivant plusieurs principes par exemple :

  - ❖ Le nombre de cases sur la même ligne, colonne et diagonal d'un joueur.
  - ❖ Si je sélectionne une case occupée je soustrai 2 de score. Cette condition nous a permis de bien choisir des cases libres dans un premier temps et d'éviter de déplacer la case qu'on a déjà jouée ce qui réduit la chance de gagner une partie.

La performance de minimax est liée principalement de la fonction d'évaluation des leafs (nodes finaux), donc pour avoir des résultats satisfaisants il faut chercher une fonction de calcule de score plus optimal.

- Parmi les **améliorations possibles aussi**, l'utilisation de l'algorithme **alpha beta** qui est plus optimal que minimax en termes de temps de calcul.
- Ajouter un menu déroulant pour **choisir le niveau de difficulté** qui change la profondeur passée à l'algorithme.
- La **complexité** de notre algorithme (sélectionner une case et la déplacer) est grande et **dépendante** de la profondeur précise dans l'algorithme. Donc si nous voulons calculer pour une grande profondeur **l'algorithme met un grand temps** pour faire ça. Ce qui n'est pas vraiment optimal.