Hes·so VALAIS WALLIS
School of Engineering

MOULIN C++

Jean Nanchen

JUNE 21, 2020

HES-SO – 2ÈME ANNÉE

Prog cpp

Table des matières

1. INTRODUCTION

Ce projet est réalisé dans le cadre du cours de programmation C++. Le but de ce projet de semestre est d'appliquer la théorie vue en cours. J'ai choisi de développer le jeu du Moulin, jeu très connu en Europe, aussi connu sous le nom « marelle ». Ce jeu est composé de trois phases, la pose, le mouvement et le saut.
Durant la première phase, faut poser à tour de rôle ses 8 pions. Si l'on parvient à aligner 3 de ses pions (moulin), on peut manger un pion de l'adversaire (ne doit pas être un moulin).
Une fois tous les pions posés, il faut déplacer à tour de rôles ses pions sur des cases voisines. C'est la deuxième phase. Il faut donc chercher à faire des moulins pour pouvoir manger les pions adverses.
Une fois qu'un des joueurs n'a plus que 3 pions, il se déplacer toutes les cases vides, même des cases non-voisine. Le jeu se termine si un adversaire bloque un joueur ou n'a plus que 2 pions.

2. CAHIER DES CHARGES

Les objectifs à remplir sont :

- Réaliser un « Moulin » (jeu de plateau) en C++
  - Version v0.0 : deux joueurs humains, aucun contrôle des règles du jeu.
  - Version v1.0 : deux joueurs humains, contrôle des règles du jeu.
  - Version v2.0 : un joueur humain contre l'ordinateur, contrôle des règles du jeu.
- Utiliser un pattern MVC (Model View Controller)

L'utilisation de la souris dans ce projet n'a pas été permis. Il faut utiliser le clavier et entrer des coordonnées à la main.

3. ANALYSE

DIAGRAMME PHYSIQUE

Le diagramme n'est pas de grande aide dans notre cas puisque tout notre projet est uniquement un software. Néanmoins nous disposons de composant physiques, le clavier utilisateur, l'écran et le PC.

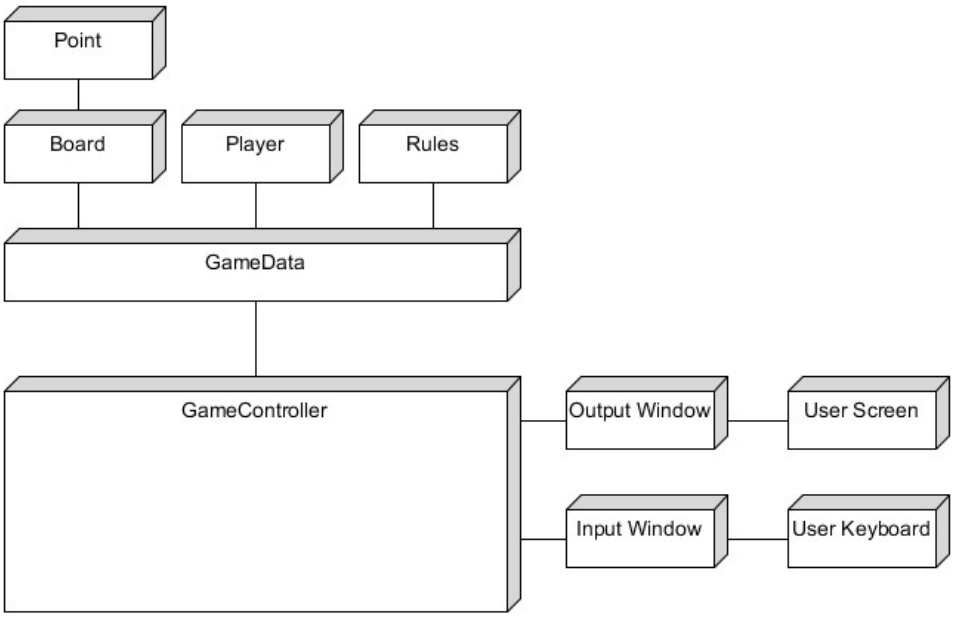


Figure 1 - Diagramme physique

USE CASE

Le diagramme Use case nous permet de savoirs quelles sont les interactions que peuvent avoir l'utilisateur avec le programme. L'utilisateur pourra donc, grâce à la fenêtre « input window » placer un pion, le déplacer, manger un pion ennemi, ainsi que démarrer la partie.

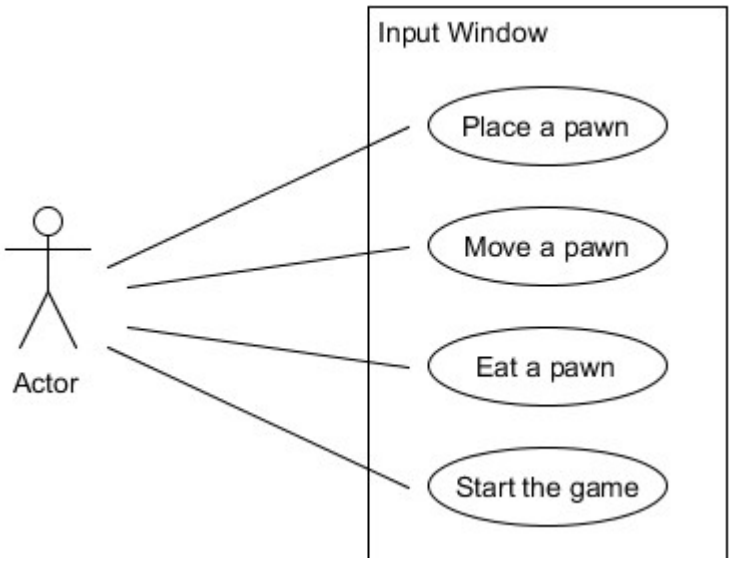


Figure 2 - Diagramme Use Case

DIAGRAMME DE SÉQUENCE

Le diagramme de séquence nous permet de connaitre les interactions entre les différents objets. Ci-dessous sont décrit plusieurs scénarios, comme le placement d'un pion sur le plateau de jeu, la suppression d'un pion sur le plateau ainsi que déplacer un pion sur le plateau.

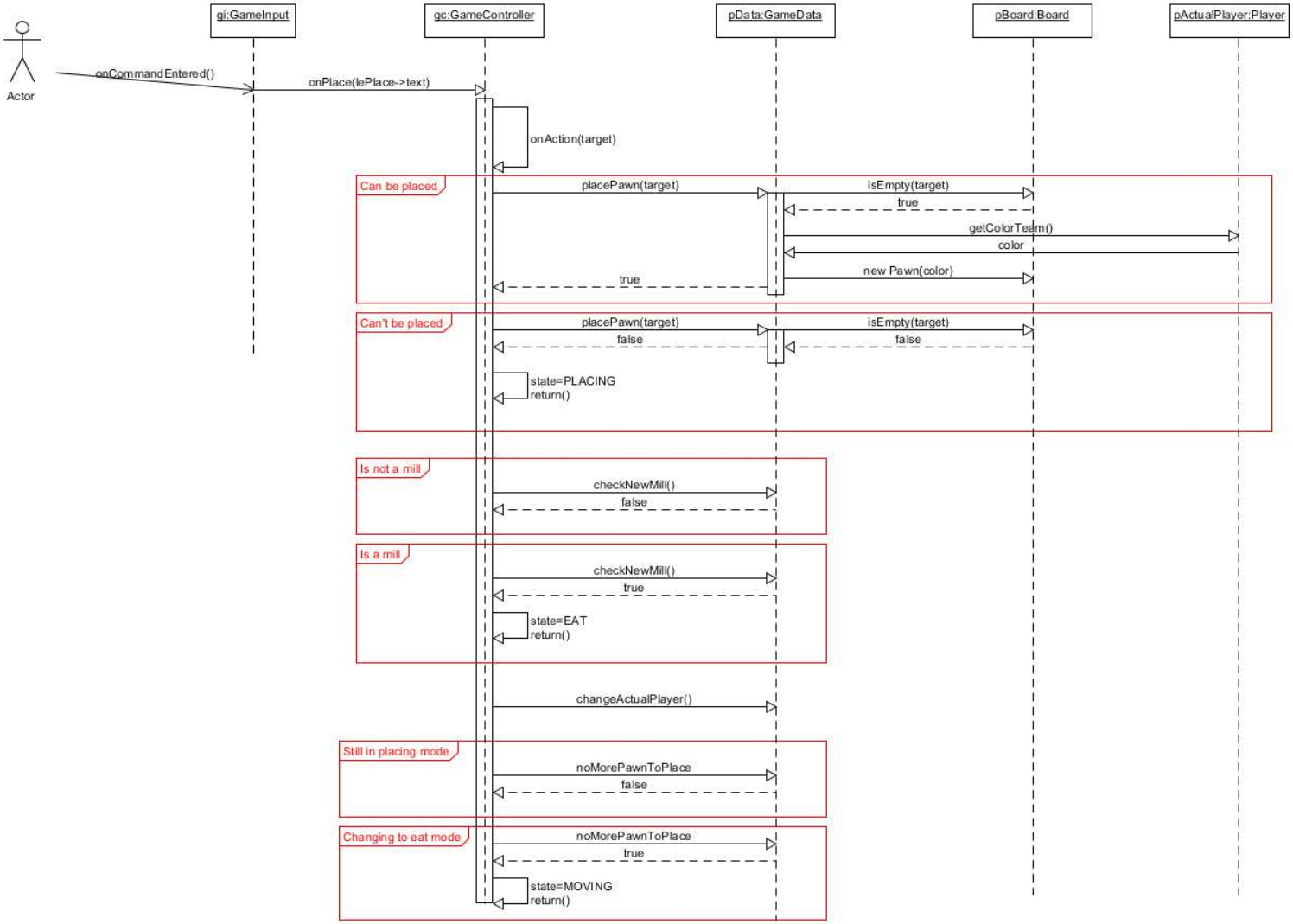


Figure 3 - Diagramme de séquence pour placer un pion

Sur ce premier diagramme de séquence décrivant le placement d'un pion, on peut y voir un utilisateur qui rentre une coordonnée d'un point dans un QLineEdit. Cette coordonnée est transmise au contrôleur avec la fonction « onPlace(QString target) ». Le contrôleur va demander à pData de créer un pion. pData va contrôler avec pBoard si le point est libre, demander la couleur du joueur actuel, pour créer un pion dans le vecteur vectPoint de pBoard. Si le point possède déjà un pion, on effectue un return() et le joueur devra sélectionner un point vide.

Dès qu'un pion est placé, on contrôle si 3 pions sont alignés avec checkNewMill(). Si cette fonction est vraie, il faut passer en EATING. Si cette fonction est fausse, il faut changer de joueur.

Pour finir, on regarde s'il reste des pions dans la main des joueurs avec la fonction noMorePawnToPlace() de pData. S'il y a encore des pions à placer, nous ne changeons pas de mode. Sinon, on passe en mode MOVING.

Le fonctionnement est similaire pour les diagrammes de séquences suivants.

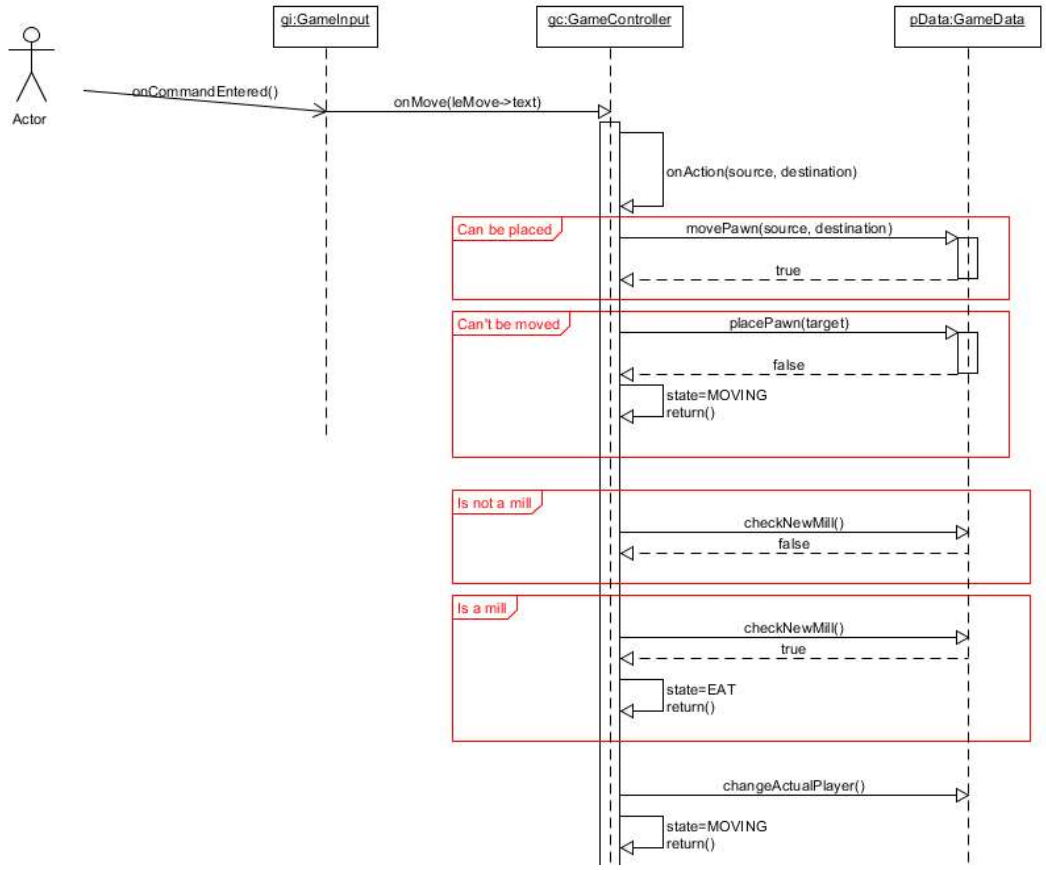


Figure 4 - Diagramme de séquence pour bouger un pion

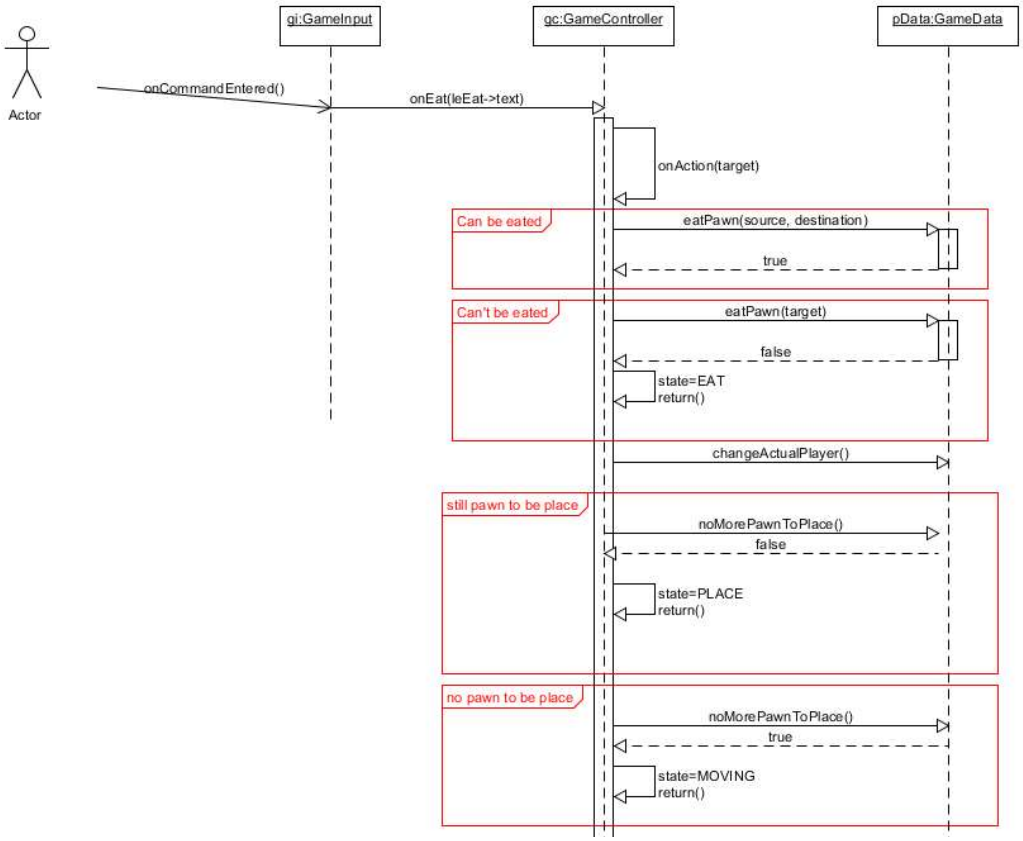


Figure 5 - Diagramme de séquence pour manger un pion

CONCEPTION

MODÈLE

Le modèle est une classe qui contient les données de notre programme, ici elle se nomme GameData. Cette classe permet d'accéder ou d'interagir avec ses données. Elle contient les joueurs, pHuman1 et pHuman2 qui sont des pointeurs vers la classe HumanPlayer. GameData contient le plateau de jeu (Board). Le plateau de jeu (Board), à un QVector d'objets Point. Les objets Points ont des coordonnées x et y, un numéro, des voisins (Point), ainsi qu'un pointeur de Pawn (pion). GameData à aussi accès à la classe Rules (avec un pointeur), possède des informations sur la positions des moulins, la position en x et en y des points, ainsi que les voisins d'un point. GameData met à jour l'output view si on change modifies ses données pour pouvoir les afficher au joueur.

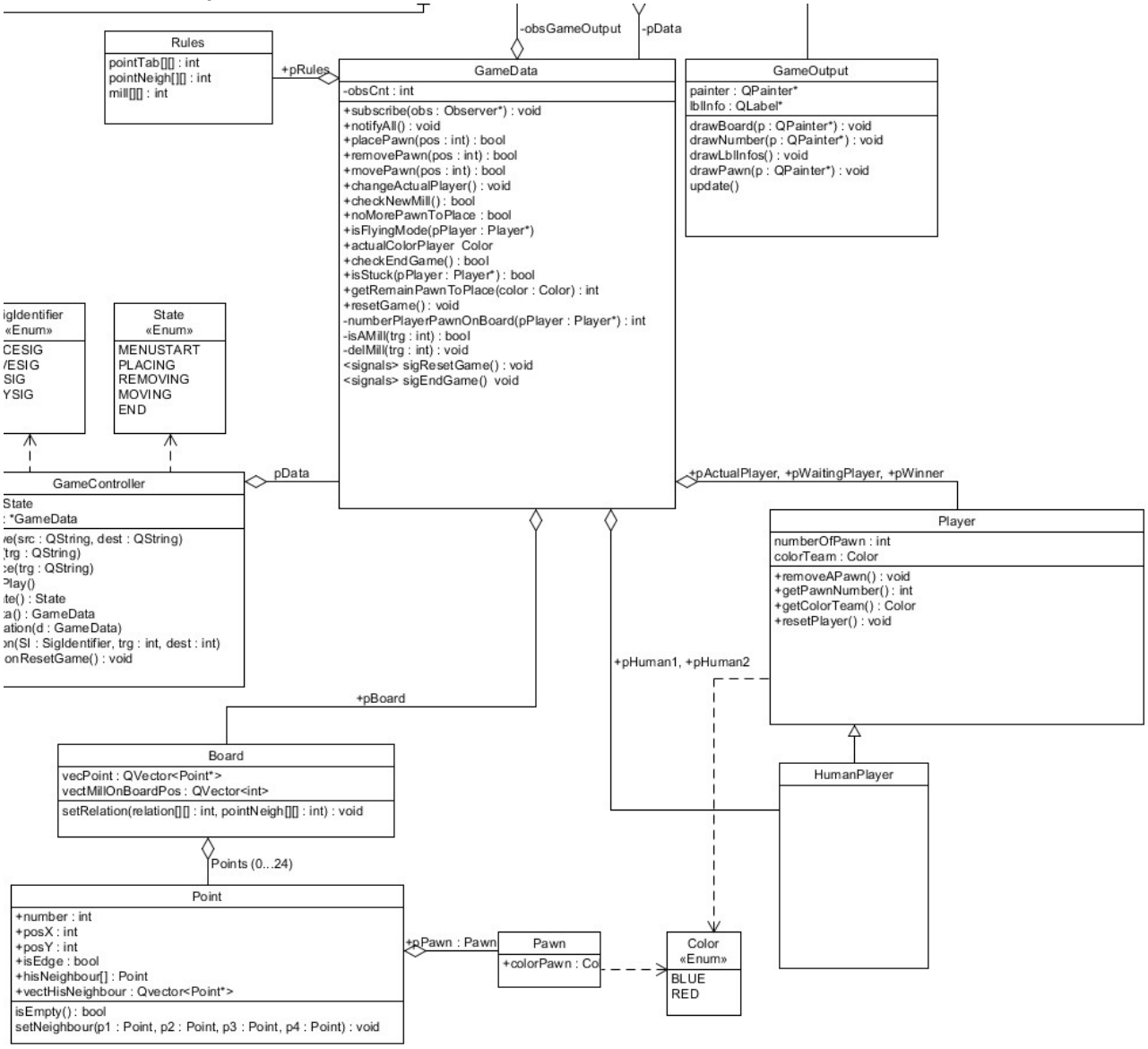


Figure 6 - Extrait de l'UML (Model)

VIEW

La view de notre programme s'appelle dans l'UML GameOutput (hérite de View). Cette classe s'occupe de l'affichage de notre jeu. La classe possède un pointeur vers GameData, ce qui lui permet de pouvoir accéder aux data de notre programme et les afficher (par ex. si des points ont des pions sur eux, etc.).
GameOutput possède 4 fonctions permettant l'affichage du jeu. La fonction drawBoard dessine la grille. La fonction drawNumber dessine les numéros des points. Les pions sont dessinés avec la fonction drawPawn. Le label qui donne les instructions aux joueurs sur la phase du jeu est actualisé avec la fonction drawLblInfos.

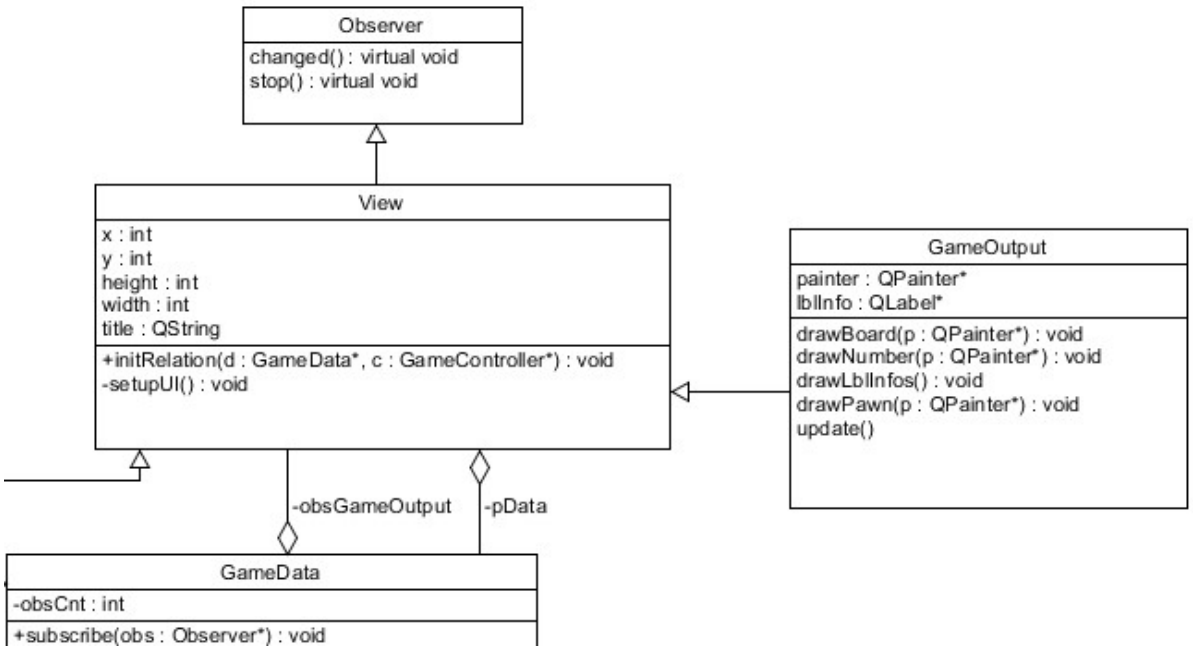


Figure 7 - Extrait de l'UML (View)

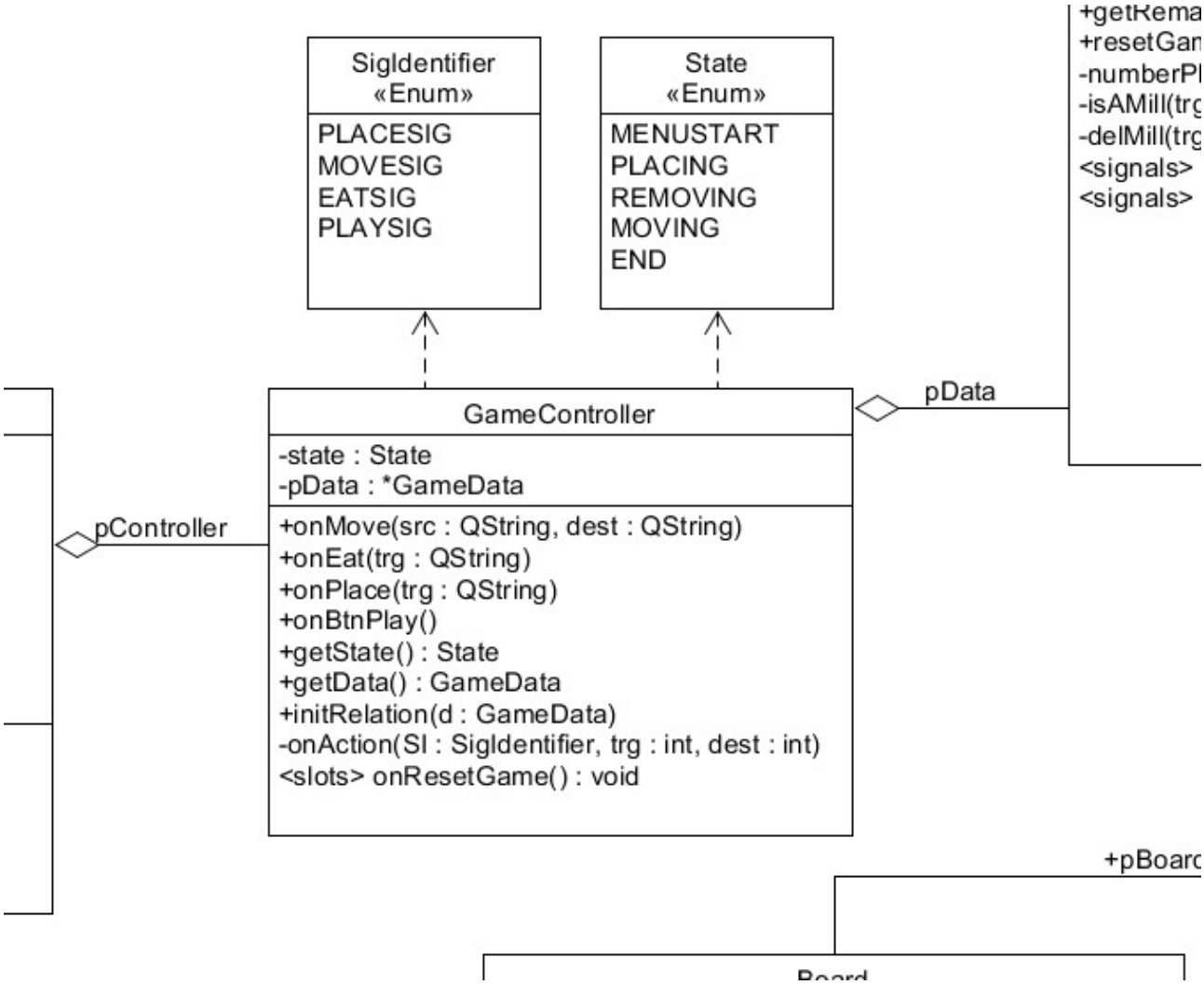
CONTROLER

Le contrôleur contient la logique de notre programme. C'est lui qui réceptionne les entrées clavier du joueur, traite ces informations, et demande au modèle de changer ses valeurs.

C'est lui qui demande typiquement au modèle d'ajouter un pion sur un point, ou d'en déplacer un. Il possède deux énumérations. C'est lui aussi qui change la phase du jeu en fonction des données du modèle.

L'énumération State est utilisée pour déterminer l'état / la phase de notre jeu. State contient l'état PLACING, qui veut dire que l'on doit placer des pions sur le plateau. REMOVING signifie que l'on doit « manger » un pion adverse. MOVING veut dire que nous sommes dans une phase où l'on doit déplacer nos pions. Etc.

L'énumération SigIdentifier permet de connaître quel signal nous a été transmis.



Voici ci-dessous la machine d'état du contrôleur. On peut y voir les 4 phases de jeu.

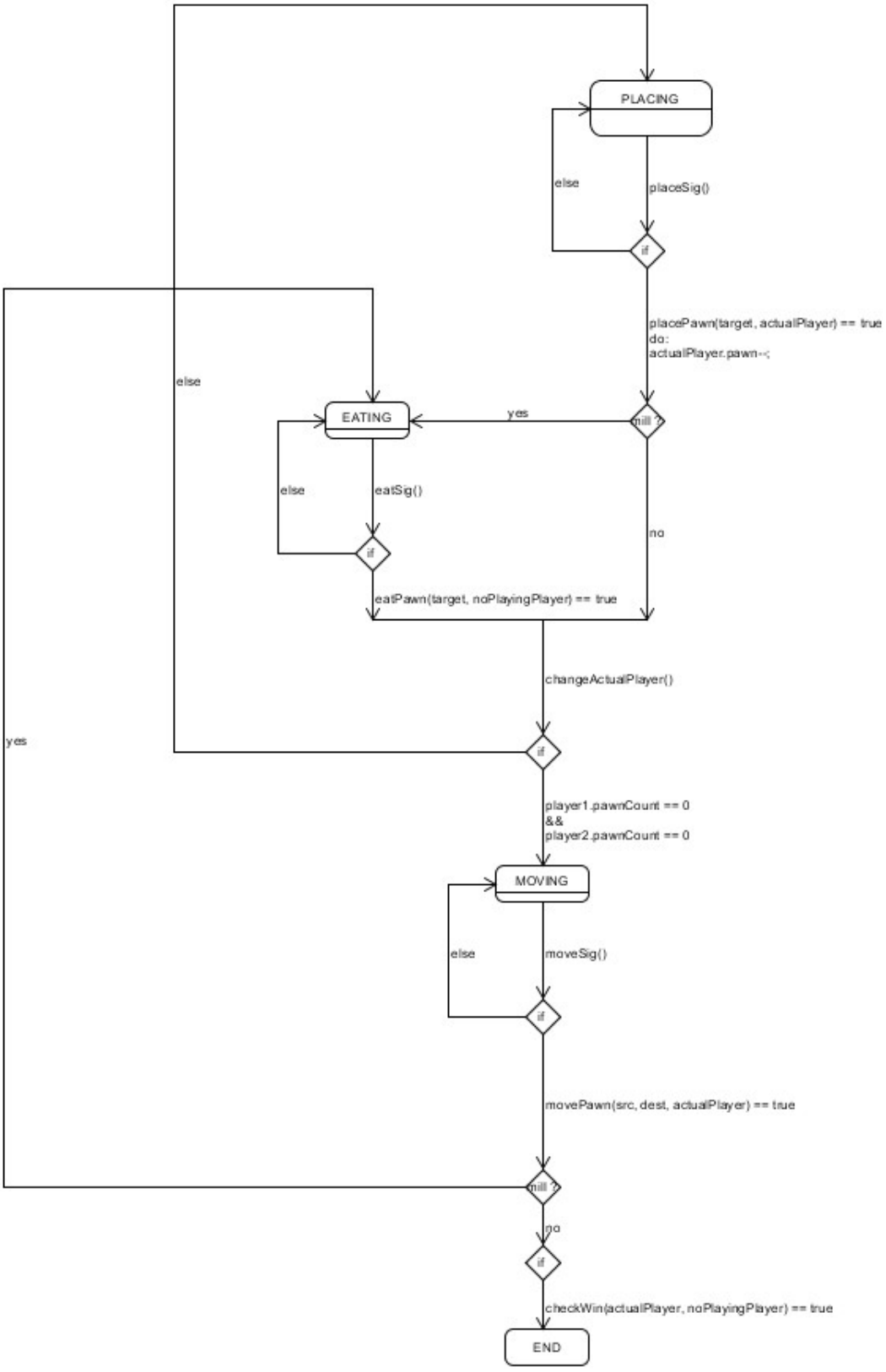


Figure 8 - Machine d'état du contrôleur

CONTRÔLE DE MOUVEMENT

Lors de la phase de jeu MOVING, il est important de contrôler que le joueur déplace son pion à une case voisine.

Pour ce faire, la classe Rules contient un énorme tableau avec les voisins de chaque case.

Chaque case peut avoir 2 à 4 voisins. La valeur -1 n'est pas un voisin, les valeurs > = 0 sont des voisins.

Par exemple le point numéro 0, possède le point 1 et le point 9 comme voisin (pointNeigh[0][] = {1,9,-1,-1}). Grâce à ce tableau, les voisins sont donc stockés dans un QVector du point. Il est donc très facile de savoir si un point est voisin de l'autre. Car quand un pion doit être déplacé, nous possédons le pointeur sur le point source et le point de destination. Il suffit donc de regarder si le point de destination est le voisin du point source.

```
int pointNeigh[MAXPOINTS][4]={{1,9,-1,-1},{0,4,2,-1},{1,14,-1,-1},{4,10,-1,-1},{1,3,5,7},{4,13,-1,-1},{7,11,-1,-1},
                      {4,6,8,-1},{7,12,-1,-1},{0,10,21,-1},{3,9,11,18},{6,10,15,-1},{8,13,17,-1},{5,12,14,20},{2,13,23,-1},
                      {11,16,-1,-1},{15,17,19,-1},{12,16,-1,-1},{10,19,-1,-1},{16,18,22,20},{13,19,-1,-1},{9,22,-1,-1},{19,21,23,-1},{14,22,-1,-1}};
```

Figure 9 - Tableau contenant les voisins de chaque point

CONTRÔLE DES MOULINS

Pendant le jeu, il faut pouvoir détecter si un joueur forme un moulin. Pour ce faire, la classe Rules contient toutes les possibilités de moulins (avec le numéro des points).

La logique pour détecter cela n'est pas très digestible graphiquement. Mais pour résumer, il faut parcourir avec trois boucles « for » les points. Ce qui permets de regarder toutes les combinaisons possibles, de 0,0,0 à 24,24,24. Il faut ensuite comparer ceci au tableau mill[][] de la classe Rules. Si l'on détecte un moulin, on enregistre dans un QVector la position du moulin. De façon à pouvoir ne pas le détecter la prochaine fois. Quand on ouvre un moulin, on supprime du QVector le moulin.

```
int mill[MAXMILL][3] ={{0,1,2}, {3,4,5}, {6,7,8}, {9,10,11}, {12,13,14}, {15,16,17}, {18,19,20},{21,22,23}, {0,9,21},
                       {3,10,18}, {6,11,15}, {1,4,7}, {16,19,22}, {8,12,17}, {5,13,20}, {2,14,23}};
```

Figure 10 - Tableau contenant les moulins

TESTS

Test	Résultat attendu	Résultat obtenu	Rem
Placement d'un pion sur un point libre	Le pion est placé	Le pion est placé	OK
Placement d'un pion sur un point occupé	Le pion n'est pas placé	Le pion n'est pas placé	OK
Placement d'un pion sur la case 99 (n'existe pas)	Le pion n'est pas placé	Le pion n'est pas placé	OK
Moulin créer pendant la phase placement	Eating mode	Eating mode	OK
Essayer de manger son propre pion	Le pion n'est pas mangé	Le pion n'est pas mangé	OK
Essayer de manger une case vide	Rien ne se passe, toujours en eating mode	Rien ne se passe, toujours en eating mode	OK
Essayer de manger un pion ennemi	Le pion est mangé, on change de joueur	Le pion est mangé, on change de joueur	OK
Déplacer son pion sur un point non-voisin (> 3 pions)	Rien ne se passe	Rien ne se passe	OK
Déplacer son pion sur un point voisin (< 3 pions)	Le pion se déplace	Le pion se déplace	OK
Ouvrir et fermer un moulin	Passe en Eating mode quand on le referme	Passe en Eating mode quand on le referme	OK
Manger un moulin ennemi	Rien ne se passe	Rien ne se passe	OK
Déplacer son pion sur un point non-voisin (3 pions)	Le pion se déplace (flying mode)	Le pion se déplace (flying mode)	OK
Déplacer son pion sur un point voisin (3 pions)	Le pion se déplace (flying mode)	Le pion se déplace (flying mode)	OK
Bloquer son ennemi, ennemi > 3 pions	Fin du jeu, vous avez bloqué votre ennemi	Fin du jeu, vous avez bloqué votre ennemi	OK
Bloquer son ennemi, ennemi a 3 pions	Pas de fin du jeu, car l'ennemi peut sauter	Pas de fin du jeu, car l'ennemi peut sauter	OK
2 pions restant	End game + detection vainqueur	End game + detection vainqueur	OK
Relancer une partie	Relance la partie	Relance la partie	OK

Le jeu fonctionne à 100%.

RÉSUMÉ DES FONCTIONS IMPLÉMENTÉES

Fonction	Résultat	Remarque
Eat	Implémenté et fonctionnel	
Place	Implémenté et fonctionnel	
Move	Implémenté et fonctionnel	
Détection de moulins	Implémenté et fonctionnel	
Gestion des déplacements	Implémenté et fonctionnel	
Fin du jeu (bloquage)	Implémenté et fonctionnel	
Fin du jeu (= 2 pions)	Implémenté et fonctionnel	
Humain VS HUmain	Implémenté et fonctionnel	
Humain vs ai	Non implémenté	Manque de temps
Pattern mvc	Implémenté	
Petit menu	Implémenté	Uniquement « Play »

Il ne reste plus qu'a implémenter une intelligence artificielle avec un petit algorithme. Mais par manque de temps, je n'ai pas pu implémenter cette fonction. Le menu est aussi un peu trop léger à mon goût, mais sans AI, il n'y a qu'une façon de jouer.

4. CONCLUSION

Dans ce projet, j'ai donc dû développer un Moulin en C++. Personnellement je ne pensais pas que ce jeu était à ce point complexe. J'ai rencontré beaucoup de difficulté pendant ce projet, mais elles ont été résolues assez rapidement.

Le développement avec un pattern MVC est très intéressant à mon goût. Il offre une bonne lisibilité et une bonne répartition des classes.

Le jeu quant à lui est jouable, mais il est moins intuitif de jouer avec des coordonnées qu'avec la souris. Le jeu fonctionne très bien, toutes les gestions des règles fonctionnes. Une fois la partie terminée il est affiché le gagnant sur l'output view, et l'on peut recommencer la partie en cliquant sur le Button PLAY. Le jeu n'a aucune erreur.

Je n'ai pas eu le temps d'implémenter une AI, qui m'aurait permis par la même occasion d'agrandit le menu avec des sous-menus.

D'un point de vue personnel, j'ai apprécié travailler sur ce projet. Malgré les conditions exceptionnelles (COVID19), mes professeurs ont toujours été la si nous avions des questions et ont tout fait pour que l'on puisse travailler dans de bonnes conditions.

5. ANNEXES

Github : https://github.com/73jn/Moulin

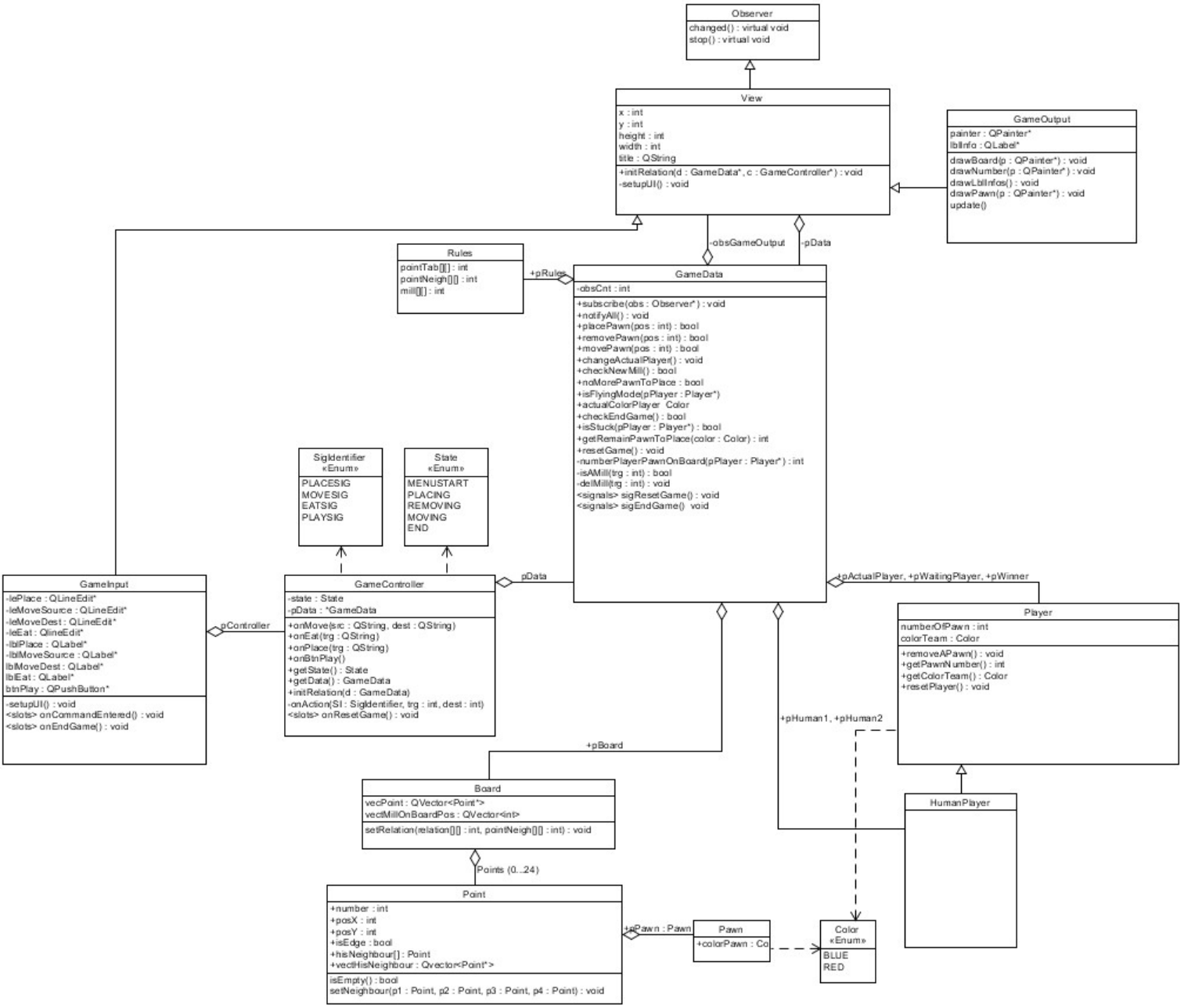


Figure 11 - UML complet

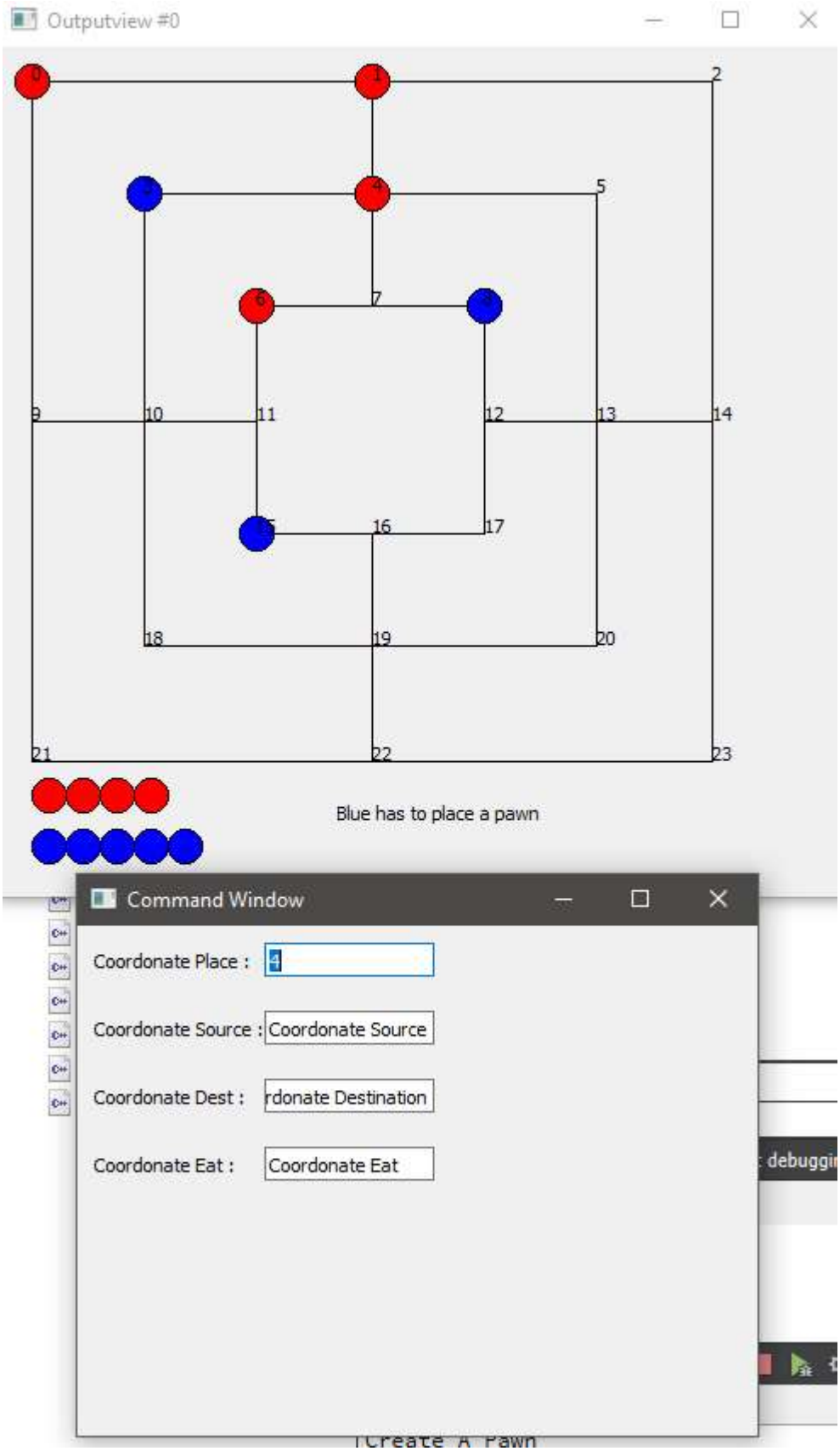


Figure 12 - Screenshot du jeu