Lamoudni Tima
El Badri Maha
Ouad Mouad
Apcher Mathéo
Laghzaoui Marwane

Projet Java : CY Path

Sommaire

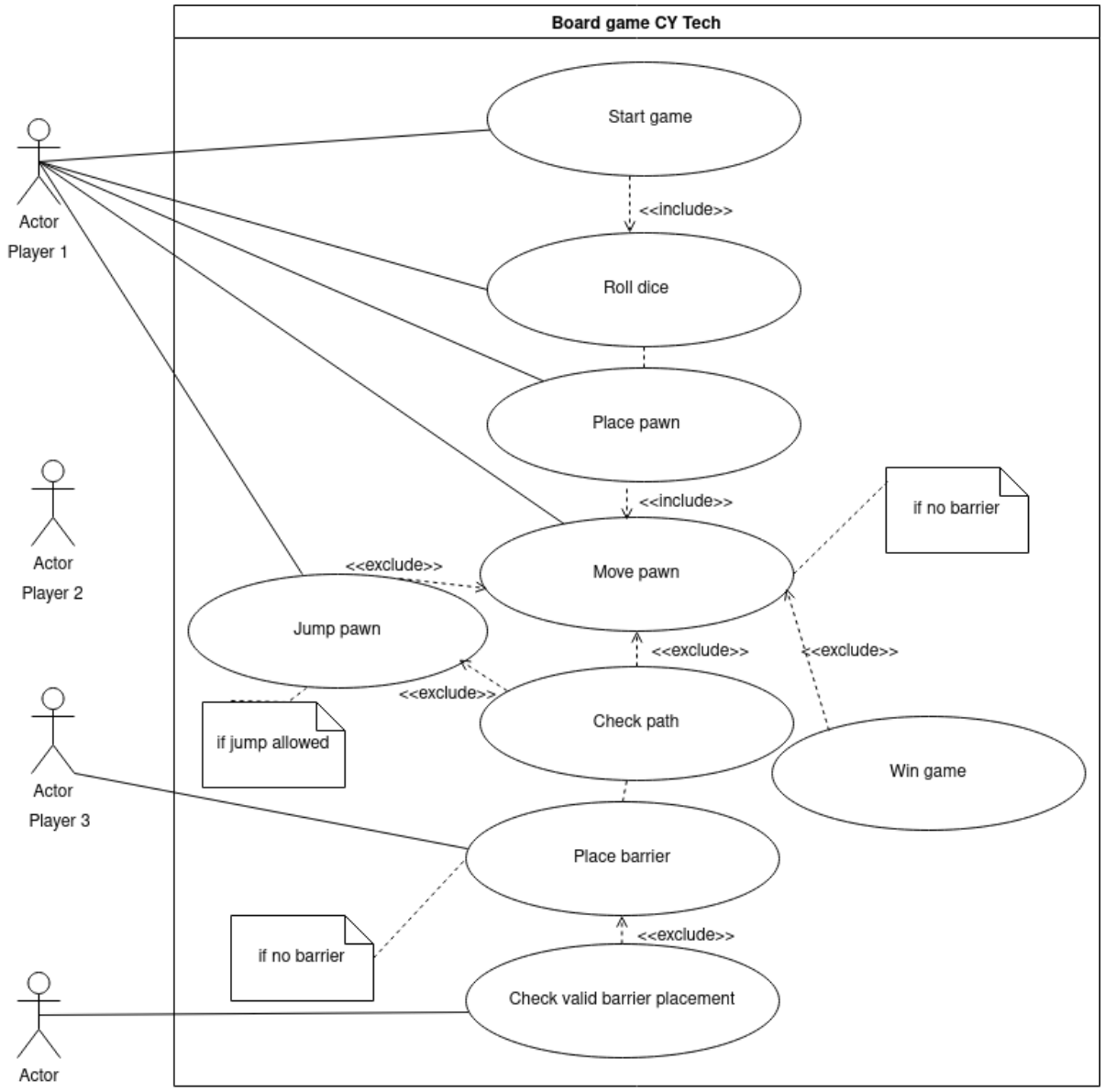
I- Conception de diagrammes

CY Path est un jeu de plateau compétitif pour 2 ou 4 joueurs. L'objectif est de faire traverser le plateau à son pion avant les autres joueurs. Les joueurs ont la possibilité de déplacer leurs pions d'une case à la fois horizontalement ou verticalement et peuvent également placer des barrières pour bloquer leurs adversaires. Cependant, les joueurs doivent placer les barrières de manière stratégique en suivant les différentes règles pour les barrières . Ils doivent faire ainsi preuve de tactique, de planification et de prise de décision pour trouver le meilleur chemin vers la ligne d'arrivée tout en évitant les obstacles. Le premier joueur à faire parvenir son pion sur n'importe quelle case du bord opposé à son point de départ est déclaré vainqueur.
Pour aboutir à dimensionner le problème, il a fallu tout d'abord élaborer 3 diagrammes :

1) Diagramme de cas d'utilisation : nous avons suivi une démarche systématique. La première étape a consisté à identifier les acteurs principaux impliqués dans le jeu, notamment Joueur 1, Joueur 2(dans le cas d'une partie de 2 joueurs), Joueur 3 et Joueur 4(si la partie comporte 4 joueurs). C'est par le biais de cette phase que nous avons déterminé les principaux participants du jeu.
   Ensuite vient l'étape d'identifier les principaux cas d'utilisation :
   - Démarrer la partie.
   - Lancer les dés.
   - Placer le pion.
   - Déplacer le pion.
   - Placer une barrière.
   - Sauter par-dessus le pion.
   - Vérifier le chemin.
   - Vérifier la validité du placement de la barrière.
   - Gagner la partie.

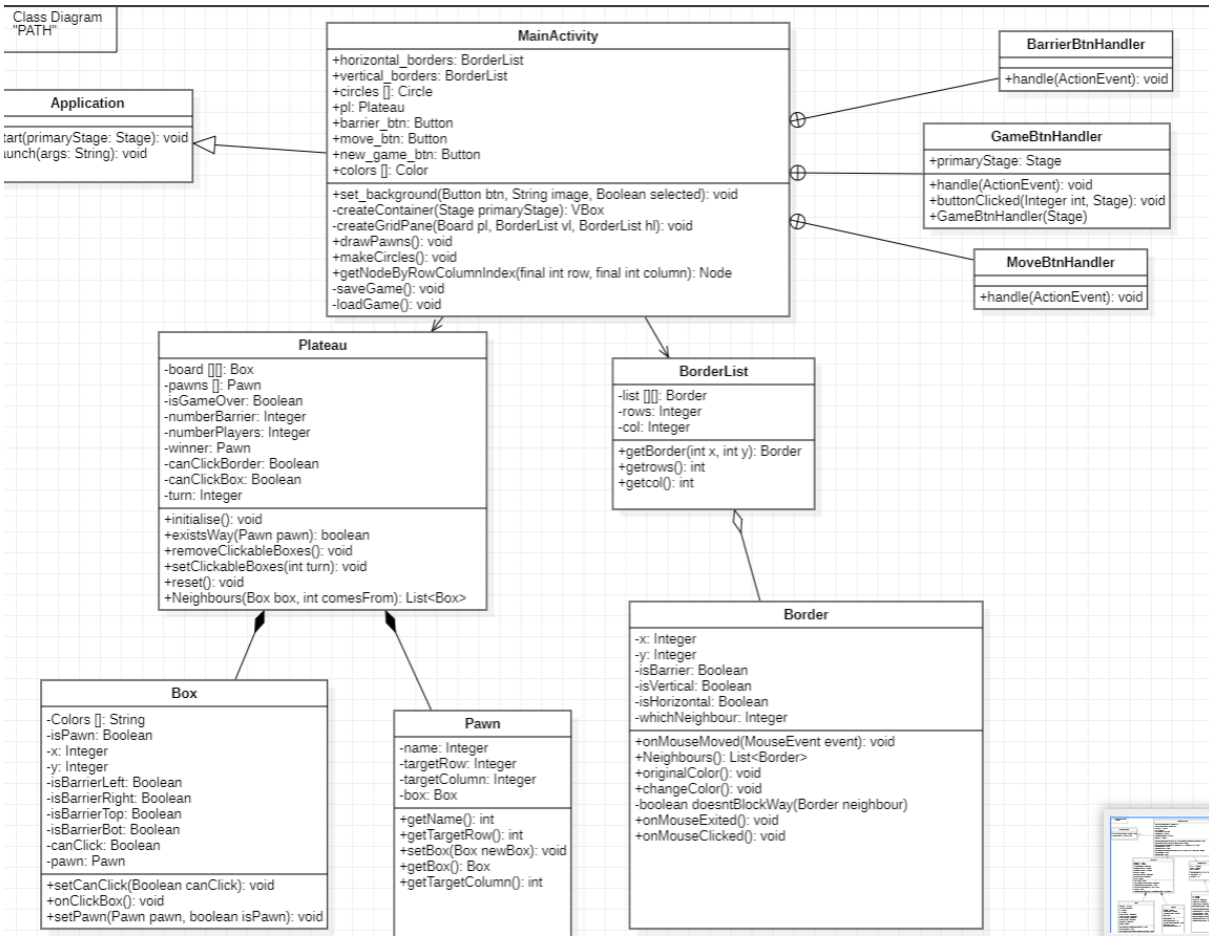
   Au cours de ce processus, nous avons rencontré quelques difficultés, parmi elles, de déterminer les relations appropriées entre les cas d'utilisation. Nous avons utilisé des relations "include" pour montrer comment certains cas d'utilisation, tels que "Démarrer la partie" et "Lancer les dés", étaient interconnectés. De plus, nous avons utilisé des relations "extend" pour montrer comment "Sauter par-dessus le pion" et "Vérifier la validité du placement de la barrière" étaient des extensions de "Déplacer le pion" et "Placer une barrière", respectivement.
   Une autre contrainte qui s'est présentée est d'identifier les exceptions pour permettre la validation des règles du jeu. Cela comprenait des cas d'utilisation tels que "Vérifier le chevauchement des barrières", "Vérifier la limite des barrières", "Vérifier le déplacement valide" et "Vérifier la disponibilité des barrières". Ces cas d'utilisation ont contribué à garantir que les règles du jeu étaient correctement suivies et que les joueurs avaient une expérience équitable et agréable.
   Dans l'ensemble, grâce à une approche étape par étape, nous avons réussi à créer un diagramme de cas d'utilisation complet pour le jeu de plateau. Nous avons surmonté les difficultés de définition des relations et d'intégration des cas d'utilisation de support pour représenter avec précision les fonctionnalités du jeu. Ce diagramme

offre une vue d'ensemble claire des fonctionnalités, des acteurs et des interactions du jeu, ce qui facilite la compréhension et l'analyse du comportement du jeu.

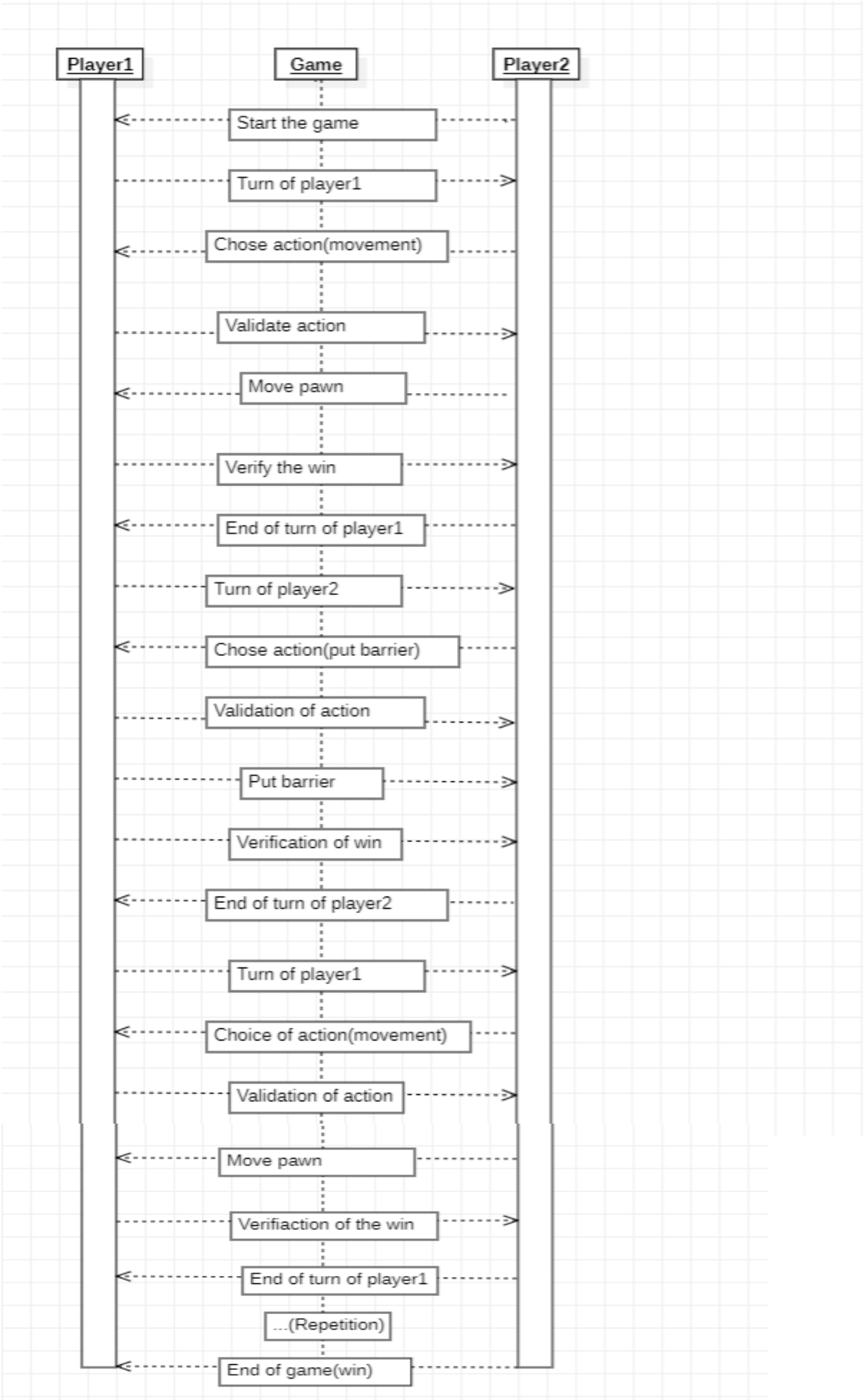


2) Diagramme de classes : Lors de la conception du diagramme de classe , nous avons suivi une approche méthodique en commençant par la classe essentielle qui joue un rôle central dans la construction du jeu , à savoir “**Main Activity**” . Cette classe utilise deux entités cruciales : “**Plateau**” et “**BorderList**” . La première classe “**Plateau”** construit le plateau du jeu qui contient les petits carreaux **“Box”** et les joueurs **“Pawn**” , d’où la relation de composition. La deuxième entité importante est **“BorderList**” qui contient une liste de barrières “**Border**” d’où la relation d’agrégation . Lors de la conception du diagramme de classe , nous avons également pris en compte la présence de certaines classes interne à la classe **“MainActivity”** comme : **“MoveBtnHandler”, “GameBt Handler” et “Barrier Btn Handler”.** Ces classes internes ont été créées pour gérer les différents événements et actions liés aux boutons présents dans l’interface utilisateur de l'application . Chacune de ces classes internes est spécialisée dans la gestion d’un type spécifique de bouton :

- Bouton BarrierBtnHandler : Bouton des barrières , une fois qu'on clique sur lui et on choisit une barrière du plateau , cette dernière se crée .
- Bouton GameBtnHandler : Bouton du nouveau jeu, qui effectue des modifications dans le jeu.
- Bouton MoveBtnHandler : Bouton de déplacement , une fois qu'on clique sur lui on peut déplacer le pion à une position choisie parmi les positions possibles.
  Voici le diagramme de classe :



3) Diagramme de séquences : Après études attentives de la description du jeu pour comprendre les actions possibles des joueurs, les règles du déplacement des pions, la position des barrières, ainsi que les conditions de victoire et les limitations du jeu, nous avons commencé à concevoir un 1e diagramme de séquences pour représenter ces éléments. Nous avons identifié les objets principaux impliqués dans le jeu qui étaient les joueurs. Pour avoir une modélisation optimale du problème, nous allons considérer l'interaction entre 2 joueurs. Le premier évènement qui aura lieu sera de commencer le jeu, ainsi le cycle de vie des 2 joueurs va démarer et sera représenté par un bloc, car les messages entre joueurs ne vont s'achever que lorsque l'un des deux sera déclaré vainqueur, et par suite la fin du jeu.
   Pendant le tour du 1e joueur, il devra choisir son action et voir si celle-ci sera validée(pas d'existence de barrière), puis procèdera à déplacer le pion. Le jeu vérifiera s'il y'a une victoire, sinon il y'aura passage au rôle du 2e joueur…


Player1
Game
Player2
Start the game
Turn of player1
Chose action(movement)
Validate action
Move pawn
Verify the win
End of turn of player1
Turn of player2
Chose action(put barrier)
Validation of action
Put barrier
Verification of win
End of turn of player2
Turn of player1
Choice of action(movement)
Validation of action
Move pawn
Verifiaction of the win
End of turn of player1
...(Repetition)
End of game(win)

II- Les outils utilisés :

Pour cela on a utilisé eclipse pour l'IDE, git pour partager le code, un serveur discord pour gérer les réunions et le planning.

III- Les grandes étapes du projet :

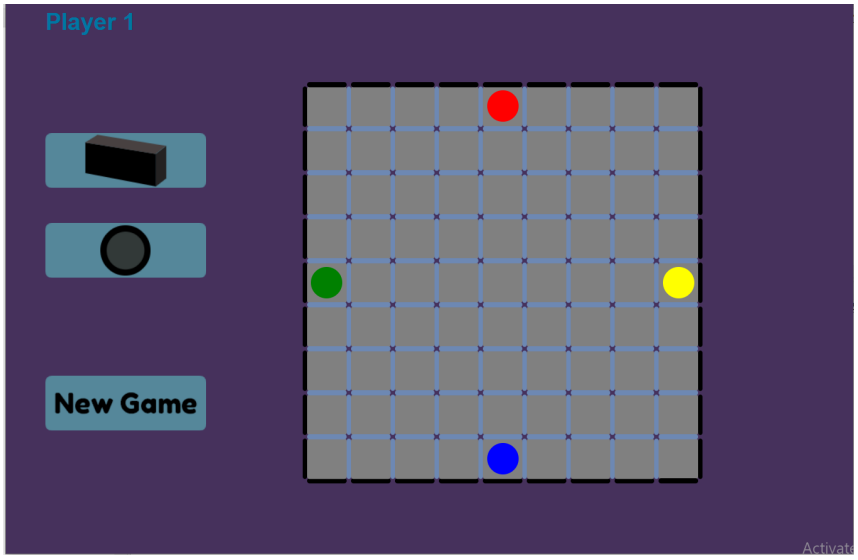
Le premier jour, nous avons principalement essayé de cerner le projet, c'est-à-dire trouver la manière la plus efficace de définir en java le plateau de jeu. Mon camarade Mathéo avait l'idée de réaliser un tableau d'entiers de taille 19x19 afin de gérer facilement les différents éléments du plateau (barrières pions etc) :

```
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
9 0 1 0 1 0 1 0 1 2 1 0 1 0 1 0 1 0 9
9 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 9
9 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 9
9 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 9
9 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 9
9 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 9
9 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 9
9 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 9
9 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 9
9 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 9
9 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 9
9 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 9
9 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 9
9 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 9
9 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 9
9 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 9
9 0 1 0 1 0 1 0 1 3 1 0 1 0 1 0 1 0 9
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
```

où les 9 représentent le bord de la map, les 0 les cases où on peut se déplacer mais sans pion dessus, les cases 1 sont les cases sans barrières, on aurait pu mettre un 5 par exemple s' il y avait une barrière. Les pions sont le 2 et 3. Le plus gros problème de ce format c'est la mise graphique. On ne peut pas gérer la taille des lignes barrières différemment des lignes déplacement, c'est-à-dire que la map ne ressemblera en rien à ce que nous cherchons à faire, n'ayant pas trouver de solution à ce problème on a décidé d'explorer une autre idée.

De mon côté (Marwane) j'ai penché vers le fait de créer un tableau 9x9 d'un objet que nous définirons en fonction des besoins du sujet, qui sera plus tard la classe Box. Ainsi le plateau de jeu était un tableau 9x9 de Box, chaque Box contenant des informations et méthodes tel que les positions des barrières (représenté par un booléen pour chaque côté), la présence d'un pion sur la case et si oui lequel etc… Donc pour former le plateau de jeu on a besoin de trois classes, Plateau, Pawn et Box. Après avoir construit ces trois classes, il nous manquait l'implémentation des barrières, pour ça on utilise une Grid, qui prend en compte un plateau de type Plateau et 2 BorderList. BorderList et Border sont les deux classes permettant la gestion des barrières, Border possède une orientation (vertical ou horizontal) et des coordonnées. Les BorderList servent à récupérer ces informations pour savoir où sont les

barrières et de mettre à jour les Box du plateau. (mettre true où il y a des barrières). Puis avec un plateau est les BorderList (verticalBorders et horizontalBorders) on peut construire une grid avec des cases où peuvent de déplacer les pions et les informations nécessaire pour placer les bordures.



La grid en question

Ensuite, on a dû faire face aux intéractions avec l'utilisateur, c'est-à-dire pouvoir se déplacer et poser des barrières¨. Pour ce faire, on doit savoir si on en a le droit, si les barrières de sortent pas du plateau, si elle ne bloque pas le chemin totalement, si elle ne se superpose pas à une autre barrières et pour le pion si une barrière ne le bloque pas ou les cas spéciaux si un pion, deux pions ou pion + barrière le bloque. Tous ces cas doivent être vérifiés avant le déplacement ou la pose d'une barrière.

Il nous fallait donc une méthode pour trouver un chemin possible. Nous connaissions deux algorithmes pour trouver le chemin le plus court pour nous amener d'un point à un autre, le A* ou l'algorithme de Dijkstra. L'algorithme de Dijkstra test chaque chemin en même temps alors que le A* test un chemin à la fois pour trouver le chemin le plus rapide. Seulement nous ne voulions pas trouver le chemin le plus court, seulement un chemin possible, donc on a utilisé l'algorithme de parcours en largeur (Breadth First Search). Qui est connu est très efficace dans notre cas, d'ailleurs la recherche d'un algorithme pour nous aider fut une recherche assez longue. On a créé une méthode Neighbours qui nous aide à produire cet algorithme. On marque les cases où on est déjà passé en utilisant une file et on cherche un chemin en itinérant.

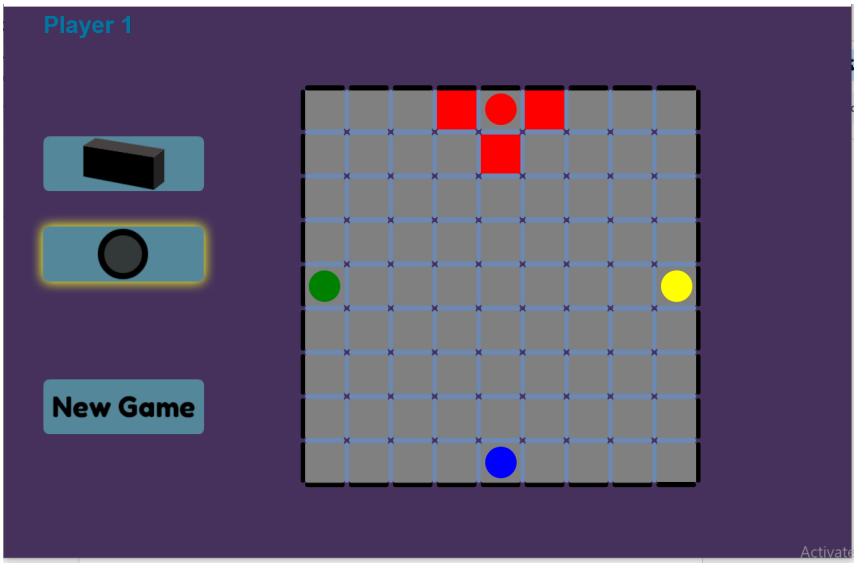
Puis pour le déplacement une suite de conditions à remplir pour pouvoir se déplacer et mettre en surbrillance les cases possibles. Lorsque nous choisissons l'option de déplacement, les cases où nous pouvons déplacer le pions dont il est le tour de jouer se mettent en surbrillance. Les cases se mettent en surbrillance après avoir vérifié quelles cases pouvaient être accédées par le pion (cas des barrières ou d'un pion collé à lui). Pour éviter les confusions par rapport à la gestion des barrières, on spécifie dans le code qu'il n'est plus possible de cliquer sur une barrière, ainsi seulement les cases pourront être

sélectionnées. Étant donné que les cases sont des boutons qui ont pour effet de modifier le plateau (en l'occurrence pour un déplacement de pion les cases correspondantes) on va actualiser le plateau. Pour déplacer un pion, on va créer une nouvelle instance du pion dans la case cible, définir la nouvelle case du pion (dans ses attributs et non dans le plateau) comme la case cible et enfin désassigner le pion à l'ancienne case.

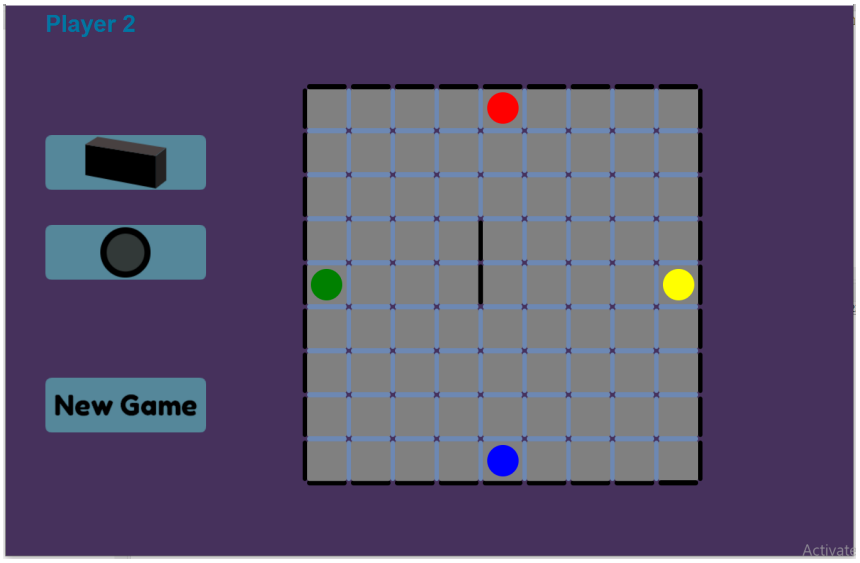
Pour les barrières, lorsque le bouton barrière est sélectionné et qu'on clique entre les cases on peut placer une barrière si certaines conditions sont bien respectées comme par exemple:
-Que le nombre de barrières soit inférieur ou égal à 20
-Que la barrière ne bloque pas complètement le chemin d'un pion
-Que le barrière ne se superpose pas avec une barrière existante
-Que la barrière ne sorte pas de la grille
Pour faciliter la lecture de l'utilisateur, on a mis en place un système de mise en surbrillance de la barrière en passant sa souris à l'endroit voulu, comme ça il n'y a pas de confusion sur l'endroit où on veut poser la barrière.
Le problème majeur de cette partie était de savoir quoi modifier lorsque que l'on clique entre les cases, et donc on change les box correspondantes celle où on a cliqué et son voisin. La méthode des voisins est une clé essentielle à la solution.

Ensuite mettre en pratique avec la partie graphique :

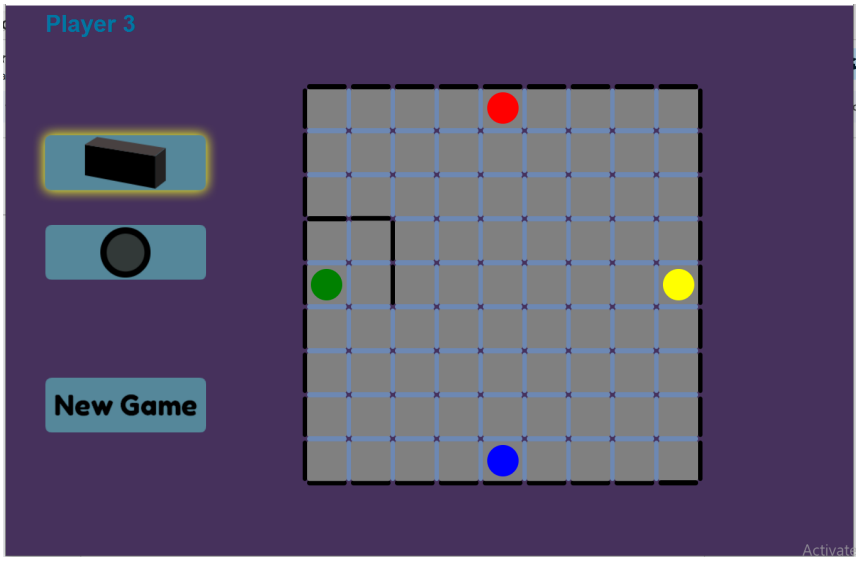
On a donc affiché le tour de jeu des joueurs en haut à gauche de la fenêtre puis créé deux boutons de déplacement et de posage de barrières et un bouton pour l'instant obsolète de nouvelle partie.



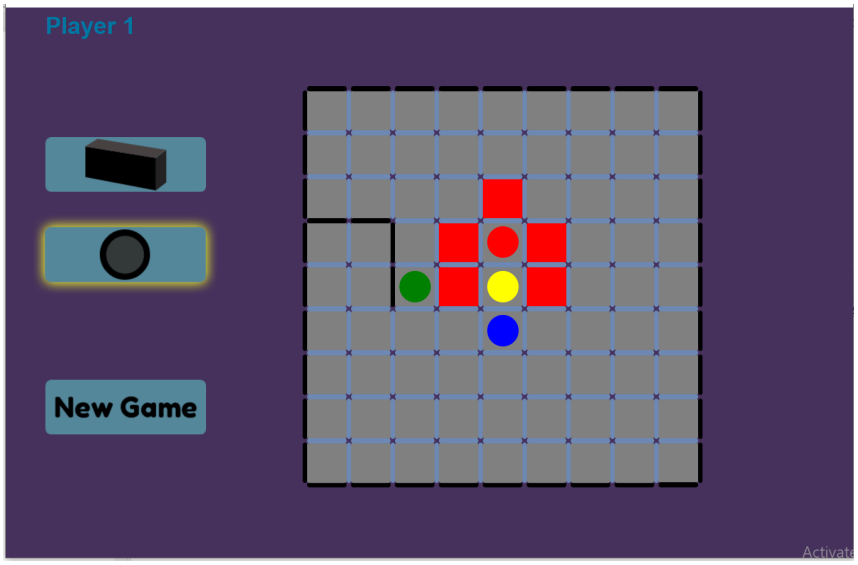
pour le déplacement de base



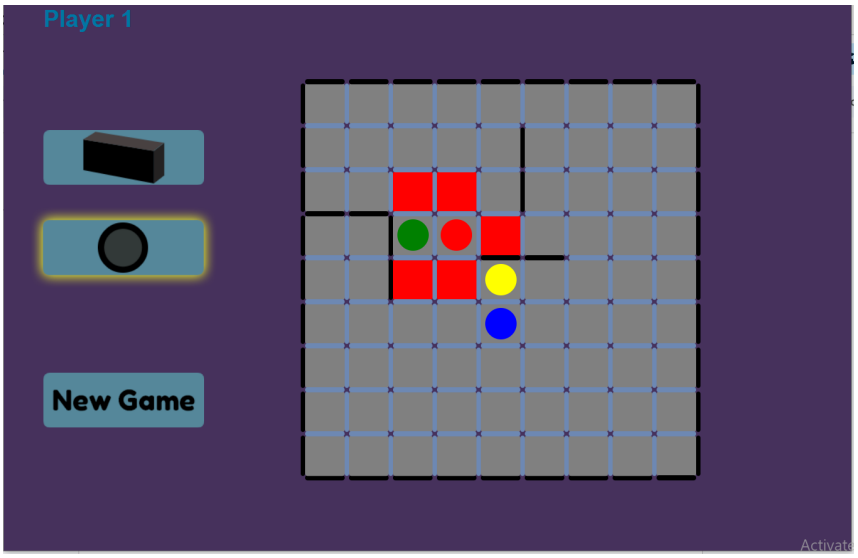
pour la pose de barrière de base



on ne peut pas poser une barrière à côté du pion vert car il bloque tout chemin possible pour celui-ci, on ne peut pas non plus superposer des barrières



ici on remarque que les spécificités de déplacement sont respectées



De même que ci-dessus

Après avoir créé une interface nous permettant de jouer au jeu voulu correctement, il nous fallait créer le système de sauvegarde, nouvelle partie et de chargement de partie. Pour cela on a commencé à créer deux boutons, un pouvant charger la partie, un autre sauvegarder et un pour créer une nouvelle partie. On a d'abord réussi à écrire dans un fichier en appuyant sur le bouton save, puis à créer une méthode pour récupérer les informations. Cependant on a changé de directive pour le côté pratique. Maintenant le jeu se sauvegarde à la fermeture de la fenêtre et se charge à l'ouverture de la fenêtre, vu que nous n'avons qu'un seul fichier save. Cependant cela ne nous convenait pas, trop de bouton sur l'interface, même si la solution d'un menu aurait pu régler ce problème on a préféré mettre un menu pour le bouton de nouvelle partie afin de choisir le nombre de joueurs. Et que les sauvegardes et le chargement se fassent indépendamment de l'utilisateur.

On a rencontré un problème majeur pour la sauvegarde des classes. Pour charger une partie on a besoin de 3 variables, une variable Plateau qui contient le plateau de la partie, 2 variables BorderList contenant les positions des bordures, une vertical et l'autre horizontal. Seulement les objets JavaFx ne peuvent pas être sérialisés ce qui fut une surprise pour nous, on ne s'y attendait vraiment pas, on a donc dû modifier l'architecture de notre code (enlever tous éléments de javafx des classes BorderList et Plateau et celle associées) afin de pouvoir sauvegarder ces classes ce qui nous a finalement pris pas mal de temps. Donc globalement faire fonctionner un code pour jouer au jeu du Path était plutôt simple comparé à son implémentation graphique et fonctionnel (gestion de fichier)

IV- Ce qu'il manque au projet pour qu'il soit parfait :

On pense que le côté graphique n'est pas et ne sera jamais parfait c'est quelque chose qui peut toujours être amélioré il faut juste du temps.
La gestion de fichier aussi, peut être qu'avoir plusieurs fichiers de sauvegarde et pouvoir choisir celui que l'on veut charger est une amélioration possible. Mais étant un jeu d'arcade c'est très peu commun d'avoir un choix de sauvegarde pour ce genre de jeu.

V- Planning durant le projet :

Notre premier objectif fixé était celui de créer une interface graphique avec notre grille de jeu, puis de mettre en place les moyens pour que l'utilisateur puisse jouer au jeu. On s'est donné une semaine voire un peu plus pour réaliser le tout, ce qu'on a respecté. Puis durant le temps restant de finir les derniers élément du cahier des charges, écriture du rapport, documentation, gestion des fichiers de sauvegardes etc...