RAPPORT

DEV4 - IMPLÉMENTATION DU JEU ABALONE

Membres : BUTUC Andrian 54254

FILALI MAFHOUM Amine 55349

Tuteur : ABSIL Romain

TABLE DES MATIÈRES

1. Introduction

1.1 Présentation du jeu

Abalone est un jeu de stratégie combinatoire abstrait où s'affrontent deux joueurs. Un joueur joue avec des billes blanches, l'autre avec des billes noires.

Le but du jeu est d'être le premier à faire sortir 6 billes adverses du plan de jeu en les poussant avec ses propres billes.

Les règles du jeu ainsi que les spécifications du système de notation ABA-PRO sont trouvables dans les annexes.

2. Analyse du projet

2.1 Répartitions des responsabilités

Le modèle de données théorique est formé par les classes suivantes :

A) La classe *HexCell* (cellule) représente un hexagone du *tablier* de jeu

Les responsabilités de cette classe sont :

- S'assurer de la cohérence des coordonnées du système cartésien a 3 axes.
- Fournit des méthodes servant à la manipulation des objets *HexCell*

B) La classe *HexBoard* représente le tablier de jeu et son stockage, ses accès sont en ABA-PRO.

Les responsabilités de cette classe sont :

- Convertir et valider les coordonnes de type ABA-PRO vers le système cartésien a 3 axes.
- Fournir l'état d'une cellule donnée à un moment donné.
- Exécuter les changements d'état des cellules qui le composent

C) La classe *Game* représente l'« arbitre » du jeu.

Les responsabilités de cette classe sont :

- Lire et transmettre les données entrées par l'utilisateur au *HexBoard* afin d'effectuer un mouvement sur le plateau.
- Vérifier la validité des mouvements selon les règles d'Abalone.
- Gestion des joueurs, attribution des tours, attribution des points et vérifier s'il y a un gagnant.

D) La classe *State* représente l'état d'une cellule du plateau
Les différentes valeurs possibles sont :
- « BLACK » si une boule noire est présente sur cette cellule.
- « WHITE » si une boule blanche est présente sur cette cellule.
- « EMPTY » si une aucune boule n'est présente sur cette cellule.

E) La classe *Player* représente un joueur, on lui assigne une des deux couleurs des billes (blanc ou noir).
Un joueur connait le nombre de billes adverse qu'il a éjecté du tablier de jeu.

2.2 Modélisation et stockage du jeu

2.2.1 Plateau

Afin de stocker un tablier hexagonal, nous avons étudié plusieurs possibilités :

A) Un graphe

C'est une représentation qui parait naturelle, de par la nature d'un graphe et de ses nœuds interconnectés, le parallèle avec un tablier de jeu hexagonal paraissait instinctif.

Malheureusement, cette modélisation a un cout en ressource élevé, car les parcours, les accès à un nœud spécifique nécessitent beaucoup de temps de calcul par rapport aux autres possibilités envisagées.
D'autre part, le stockage en mémoire d'un graphe est plus difficile à implémenter dans le langage de programmation proposé que la plupart des autres possibilités envisagées.

B) Tableau bidimensionnel

Cette représentation est relativement simple à implémenter.

Malheureusement, les parcours, nécessaires à l'accès à une case spécifique de ces structures ont un cout en ressource élevé de par la complexité élevée des algorithmes de parcours d'un tableau bi dimensionnel.

Afin de respecter le caractère hexagonal du tablier de jeu. Il faut désigner certaines cases du tableau bidimensionnel comme n'étant pas valide.

Il n'y a donc jamais de bille sur la case d'indice 0 du tableau. Ainsi l'ensemble de ces cases qui ne sont jamais utilisées représente un déficit quant à l'optimisation de l'utilisation de la mémoire.

Il est possible de mitiger cet effet en décalant notre tableau vers un bord, mais cela entraine une légère diminution de l'optimisation, car il faudra

pouvoir passer du système d'indice du tableau et le système d'indice du tablier de jeu.

Un autre avantage de cette représentation c'est qu'il est possible de modéliser une cellule du tablier comme étant une case du tableau. Ainsi il ne serait pas nécessaire d'instancier 61 objets *HexCell* ni de les avoir en mémoire à tout moment.

C) Conteneur associatif (map)

Cette représentation est relativement simple à implémenter. La difficulté d'implémentation se situe entre un tableau bidimensionnel et un graphe.

Afin d'afficher le contenu d'un conteneur associatif, les parcours nécessaires sont comparables aux parcours des tableaux unidimensionnels.

Les conteneurs associatifs apportent une simplicité dans l'insertion, la recherche et la suppression d'un de ses membres en particulier : la complexité de ces opérations est de O(n).

Les performances obtenues par l'utilisation d'une map vont dépendre de la fonction de hachage utilisée.

Tous les membres du conteneur associatif sont utilisés lors du déroulement du jeu, aucune allocation de mémoire inutile n'est faite.

2.3 Systèmes de coordonnes et conversion

Les deux systèmes de coordonnées utilisés sont :

- Le système ABA-PRO décrit au point **1.1.2** utilise dans la logique interne de *Game*.
- Le système cartésien à 3 axes utilisé dans la logique interne de *HexBoard* et *HexCell*.

Le jeu reçoit les données de l'extérieur dans le format *ABA-PRO* et la conversion en système cartésien à 3 axes s'effectue par *HexBoard* lors d'un mouvement.

2.4 Modélisation des mouvements des billes

Afin d'effectuer un mouvement, on effectue dans un premier temps un certain nombre de validations selon les règles du jeu vu au point 1.1.1 et seulement si toutes les validations sont passées, on effectue le mouvement.

La responsabilité de ces validations est repartie entre deux classes : *HexBoard* et *Game.*

Game lit les données entrées par l'utilisateur et les stocke dans un tableau de String de longueur 3. La dernière case pouvant ne rien contenir dans le cas d'un mouvement en ligne.

Les vérifications concernant la validité des données entrées est demandé à *HexBoard* qui les effectue selon sa logique interne en convertissant les coordonnées ABA-PRO en *HexCell* avec ses 3 coordonnées x, y et z.

Si les premières vérifications ont été effectuées avec succès, Game demande ensuite à *HexBoard* de vérifier la validité du mouvement en respectant les règles du jeu décrites en annexes.

Si le mouvement est validé, *Game* demande a *HexBoard* d'effectuer le mouvement selon sa logique interne (modification de la map). La distinction entre les deux types de mouvement s'effectue en fonction de la troisième case du tableau de String contenant les données entrées par l'utilisateur.

Si une bille adverse est éjectée du terrain lors du mouvement, la méthode move de *Game* le signale en renvoyant un booléen ce qui est utilisé pour mettre à jour le score.

2.5 Design Pattern

Dans le respect des consignes de ce projet, le design pattern qui sera utilisé sera le design pattern « Observable/Observer »

Ce dernier sera mis en place lors d'une remise ultérieure.

2.6 Conclusion

Au cours de l'analyse effectuée pour ce projet-ci, les points soulevés les plus problématiques ont été les suivants :

La prise de décision concernant la représentation du tablier de jeu

Plusieurs possibilités ont été envisagées : un graphe, un tableau bidimensionnel et un conteneur associatif.

Chacune de ces possibilités a été savamment analysée, les avantages et inconvénients de chacune ont été contrepesés et une décision a été prise en tenant compte ces facteurs primordiaux :

- *Optimisation de la mémoire*
- *Vitesse d'accès aux données*
- *Difficulté de l'implémentation*

Le choix d'implémentation d'une map a été également soutenu par les sources citées dans les consignes de ce projet. (Voir point *4.2.1*)

Lors de la suite du déroulement de notre analyse, nous n'avons pas eu à nous confronter à des complications insurmontables.

3. Implémentation

NB : L'utilisation du mot « state » lorsqu'on fait référence au code est équivalent pour nous à l'utilisation du mot « couleur » lorsqu'on fait référence au jeu Abalone.

3.1 Algorithmes non-triviaux

HexCell

L'opérateur < comporte un algorithme non trivial : l'idée principale derrière cet algorithme est de comparer la composante y des HexCell.

Au sein de notre HexBoard, la HexCell aux composantes {-4,4,0} est considérée comme la plus petite et la HexCell aux composantes {4,-4,0} est la plus grande.

Si les composantes y sont égales, les deux autres composantes x et z sont évalués.

Si elles sont différentes, les HexCell sont donc sur des lignes différentes du HexBoard et on peut en déduire la plus petite.

HexBoard

Afin de ne pas transmettre toutes les données à la view, on stocke uniquement les données utiles dans un vecteur StateVector qui contient donc juste l'état de chaque case du HexBoard, dans l'ordre décrit dans la section précédente.

IsAbaProValid

Cet algorithme vérifie le format ABA-PRO de l'input. Le premier caractère doit être une lettre entre A et I et le deuxième caractère doit être un chiffre entre 0 et 9.

Ces vérifications sont faites en soustrayant les valeurs ASCII de ces caractères.

NextHexCell

Cette méthode a besoin d'une HexCell et d'une direction afin de déterminer la case suivante à cette HexCell.

Nous avons stocké les 6 directions possibles dans un vecteur *CellDirections* contenant 6 HexCell dont chacune représente une direction.

Ex : la direction Nord-Est est stockée sous forme d'une *HexCell* {0,1,-1}

Ainsi la direction passée en paramètre est utilisée comme index afin de nous donner la HexCell qui représente la direction. La *HexCell* passée en paramètre est également validée.

Pour trouver la case suivante, il suffit d'additionner la *HexCell* de départ avec la *HexCell* qui représente la direction souhaitée.

CountHexCell

Le but est de compter le nombre ce billes d'une couleur donnée, à partir d'une bille de départ et dans une direction donnée.

On vérifie que la bille de départ est bien de même couleur que la couleur donnée.

On compte ensuite les billes de mêmes couleurs en comptant la bille de départ comme la première bille. (Compteur commence à 1)

L'on ne prend pas en compte les cases vides, ni les cases d'une couleur différente, ni les cases ayant des coordonnées qui sont en dehors de notre plateau.

IsValidMove

Le but est de vérifier la validité d'un mouvement de billes.

L'ensemble des éléments du vecteurs contenant les coordonnes du mouvement sont vérifiés grâce à la méthode isAbaProValid.

En fonction de la taille du vecteur on peut déterminer le type de move à effectuer.

Si c'est un mouvement linéaire, on transforme les 2 string en HexCell et on vérifie que la première HexCell et la seconde HexCell sont adjacentes.

Si c'est un mouvement latéral, on transforme les 3 string en HexCell et on vérifie que :

-La première HexCell et la HexCell d'arrivée sont adjacentes

-Le mouvement latéral ne porte que sur 3 boules au maximum

-La première HexCell et la deuxième HexCell ne sont séparées que d'une HexCell maximum.

-La deuxième HexCell n'est pas adjacente à la HexCell d'arrivée.

-La première HexCell et la seconde HexCell partagent un axe en commun.

DetermineDirection

Pour déterminer la direction entre deux HexCell adjacentes on soustrait la HexCell destination à la HexCell de départ et on compare cette HexCell à celles présentes dans notre vecteur de direction CellDirection afin d'identifier l'indice auquel on trouve cette direction dans le vecteur CellDirection.

Move

En fonction de la taille du vecteur on peut déterminer le type de move à effectuer.

Si c'est un mouvement linéaire, on vérifie (après transformation des strings en HexCell) que la HexCell de départ contienne bien une bille de même couleur que la couleur passée en paramètre.

Si c'est un mouvement latéral, on vérifie que la première et la deuxième HexCell contiennent bien une bille de même couleur que la couleur passée en paramètre.

Cette méthode fait appel aux méthodes privées *inLineMove* et *sideStepMove* afin d'effectuer les mouvements.

Si le mouvement à effectuer entraine l'éjection d'une bille du joueur adverse, cette méthode renvoie un booléen vrai.

InLineMove

Le premier cas géré est celui où l'on pousse une bille vers une case vide

On compte le nombre de billes du joueur courant impliquées dans le mouvement.

Si maximum 3 billes, on déduit la case suivante et si elle n'est pas vide ou en dehors du tableau on vérifie si nous sommes dans un cas de sumitos.

Si la case est vide et a l'intérieur du plateau, on pousse simplement les billes du joueur courant.

Si nous sommes dans un cas de sumitos, si derrière la dernière bille adverse il y a une case libre on pousse les billes, s'il n'y a pas d'espace on n'effectue pas le mouvement et si c'est en dehors du plateau on pousse les billes et on en éjecte une.

SideStepMove

On détermine le nombre de billes impliquées dans le mouvement latéral, ensuite on vérifie la couleur de chacune des billes impliquées.

L'étape suivante est de déterminer les HexCell d'arrivée pour chaque bille et de vérifier que ces HexCell soient libres. Si ces vérifications sont validées, on déplace chaque bille.

Game

Move

On utilise *readPlayerMove* qui divise l'input en sous-parties afin de récupérer un vecteur de string sur lequel on appelle *isInputValid* pour s'assurer que le vecteur d'input contient bien des strings au format « Lettre-Chiffre ».

Il appelle également depuis le HexBoard la méthode isValidMove qui vérifie la validité du mouvement en fonction de l'état du HexBoard.

Si les vérifications sont passées on appelle le move du HexBoard pour effectuer le mouvement, si une bille a été éjectée on incrémente le score du joueur courant.

Ensuite, on change de joueur et on notifie l'observateur que l'état du jeu a changé afin qu'il puisse se mettre à jour.

Si l'une des validations effectuée ne passe pas, on notifie l'observateur avec un message d'erreur.

View Console

PrintAbaloneBoard

Cette méthode affiche le HexBoard, ligne par ligne, afin d'obtenir une vue pour un plateau de 61 cases.

Console Controller

Play

Au départ, le controller effectue les appels pour afficher les donnes du jeu mais ensuite l'Observable reprend cette responsabilité en notifiant son changement d'état pour que l'Observer se mette à jour.

Après les appels initiaux d'affichage, le rôle du controller est de fournir les inputs récupérés par la View_Console et les passer au modèle.

3.2 Design Pattern

Nous avons distribué les rôles d'*Observable* / *Observer* comme suit :

La classe *Game* est un *Observable* tandis que la classe *View_Console* est un Observer.

Lorsque le modèle est modifié, la classe Game notifie ses Observers qui peuvent donc se mettre à jour.

Nous utilisons deux différentes méthodes de notification, la première lorsque le modèle est modifié, et la seconde lorsqu' une erreur survient, cette dernière comportant un message d'erreur.

Afin de transmettre les données du jeu devant être mises à jour aux observateurs, nous utilisons une structure GameData. Elle contient la map qui stocke le plateau de jeu ainsi que les joueurs.

Le contenu de cette structure à varié au cours du développement.

3.3 Graphical User Interface

Concernant l'affichage graphique du plateau de jeu, nous avons décidé d'utiliser un grille contenant 61 boutons cliquables. La forme de ces boutons a été modelée afin de répondre à nos besoins.

Nous avons arrangé la position des boutons afin qu'ensembles ils forment un hexagone de même forme que le tablier de jeu.

Afin de faire ressortir le cadre du tableur de jeu, nous avons utilisé une image en arrière-plan, afin de rendre le visuel plus agréable.

Un espace latéral est réservé à chacun des joueurs afin d'y afficher les informations relatives à leurs scores respectifs.

Sous le tablier de jeu, nous avons judicieusement positionné deux boutons de contrôles du jeu :

Le premier sert à valider le mouvement après avoir cliqué sur les boules que l'on veut faire bouger.

Le second est utile si l'on veut corriger une sélection de boules incorrecte.

On peut également y trouver les informations sur le joueur courant ainsi qu'un espace réservé à l'affichage d'erreurs éventuelles.

Finalement, il existe un dernier bouton servant à démarrer une nouvelle partie.

Lorsqu'une partie est terminée, une fenêtre contextuelle apparaît afin de féliciter le joueur gagnant et de lui laisser le choix entre une nouvelle partie ou l'arrêt du jeu.

L'apparence globale de la vue graphique est épurée, l'accent est mis sur les éléments principaux du jeu afin de les mettre en évidence. Ainsi le tablier de jeu, les

boules et les boutons ont bénéficié de styles CSS qui ont été appliqués au sein de l'outil *QT Designer* et qui n'apparaissent donc pas à première vue lors de la lecture du code, mais il est visible dans le fichier *mainwindow.ui* qui contient du code XML.

Ci-dessous vous trouverez les explications relatives aux différentes classes impliquées dans notre interface graphique.

MainWindow

MainWindow représente la fenêtre principale de notre application.

Etant donné que nous utilisons l'outil QT Designer, la mise en page et la création des éléments de la vue est réalisé par une classe *Ui::MainWindow* qui est générée à partir du fichier *mainwindow.ui* qui contient le code XML.

Au sein de notre classe MainWindow, il y a un premier pointeur vers un objet de type *Ui::MainWindow*, un vecteur de pointeurs vers les 61 boutons cliquables, un vecteur de pointeurs vers un objet de type *Button_Handler* qui gère les clics sur les boutons au sein du plateau.

Finalement, cette classe contient une variable de type String à laquelle on rajoute le nom du bouton du plateau qui a été cliqué, ce nom représentant la position ABA-PRO de ce bouton au sein du plateau.

Cette classe contient des méthodes permettant de trier les pointeurs de boutons dans un ordre qui nous est utile (*sortButtons*), de mettre à jour le style spécifique d'un bouton (*updateButton*), de vider le string contenant le mouvement à effectuer afin de corriger ce mouvement en cas d'erreur de la part du joueur (*clear_move*) et de lier les boutons de contrôles avec leurs actions respectives (*addActionsOnButtons*).

Quelques méthodes nécessitent de plus amples explications, notamment les méthodes de mises à jour ainsi que le constructeur de cette classe.

Au sein du constructeur, on récupère les références vers les 61 boutons qui nous sont utiles, on crée les 61 objets de types *Button_Handler* et on lie les 61 boutons chacun à leur *Button_Handler* respectif. On trie les pointeurs de boutons dans un ordre utile lors de la mise à jour du plateau. La première mise à jour du plateau s'effectue au sein du constructeur en utilisant un objet *GameData* passé en paramètre.

La première méthode de mise à jour est appelée lorsqu'un mouvement à bien été effectuée. Cette méthode cache la zone attribuée aux erreurs, mets à jour chacun des boutons de la grille en partant de celui en haut a gauche et en terminant par celui en bas a droite, ensuite on met à jour les informations relatives à chaque joueur.

La seconde méthode de mise à jour est appelée lorsqu'une erreur est survenue dans le mouvement. Elle affiche un message d'erreur dans la zone désignée.

Button_Handler

Un objet *Button_Handler* possède une référence vers le bouton pour lequel il gère le clic, une référence vers un string qui stocke le mouvement ainsi qu'un booléen qui représente l'état cliqué ou non du bouton.

Cette classe contient des méthodes permettant lors d'un clic, de rajouter ou retirer le nom du bouton cliqué du string contenant le mouvement courant, en fonction de son état préalablement cliqué ou non.

Dans un souci de clarté, l'apparence visuelle des boutons est différente pour chaque cas afin de différencier un bouton non cliqué, un bouton cliqué une fois et un bouton cliqué deux fois (décliqué).

Gui_Controller

Ce contrôleur effectue la liaison entre le jeu (*Game*) et l'affichage graphique (*MainWindow*) en utilisant les méthodes suivantes :

Play qui récupère le mouvement auprès de la vue et le fourni au modèle afin d'effectuer le mouvement et vérifie si le joueur courant est gagnant.

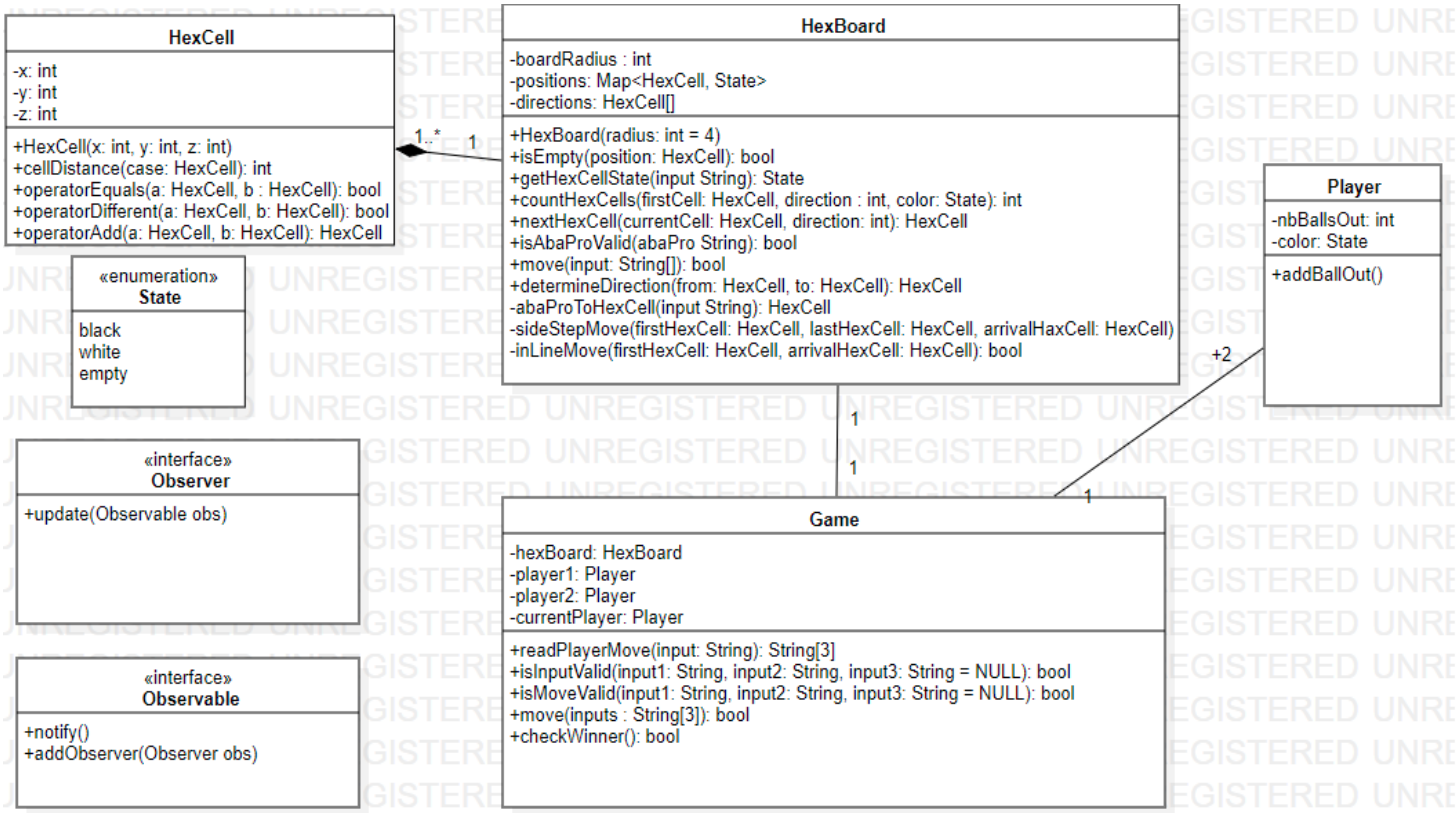
Reset réinitialise le jeu et mets à jour la vue.

Exit quitte le jeu.

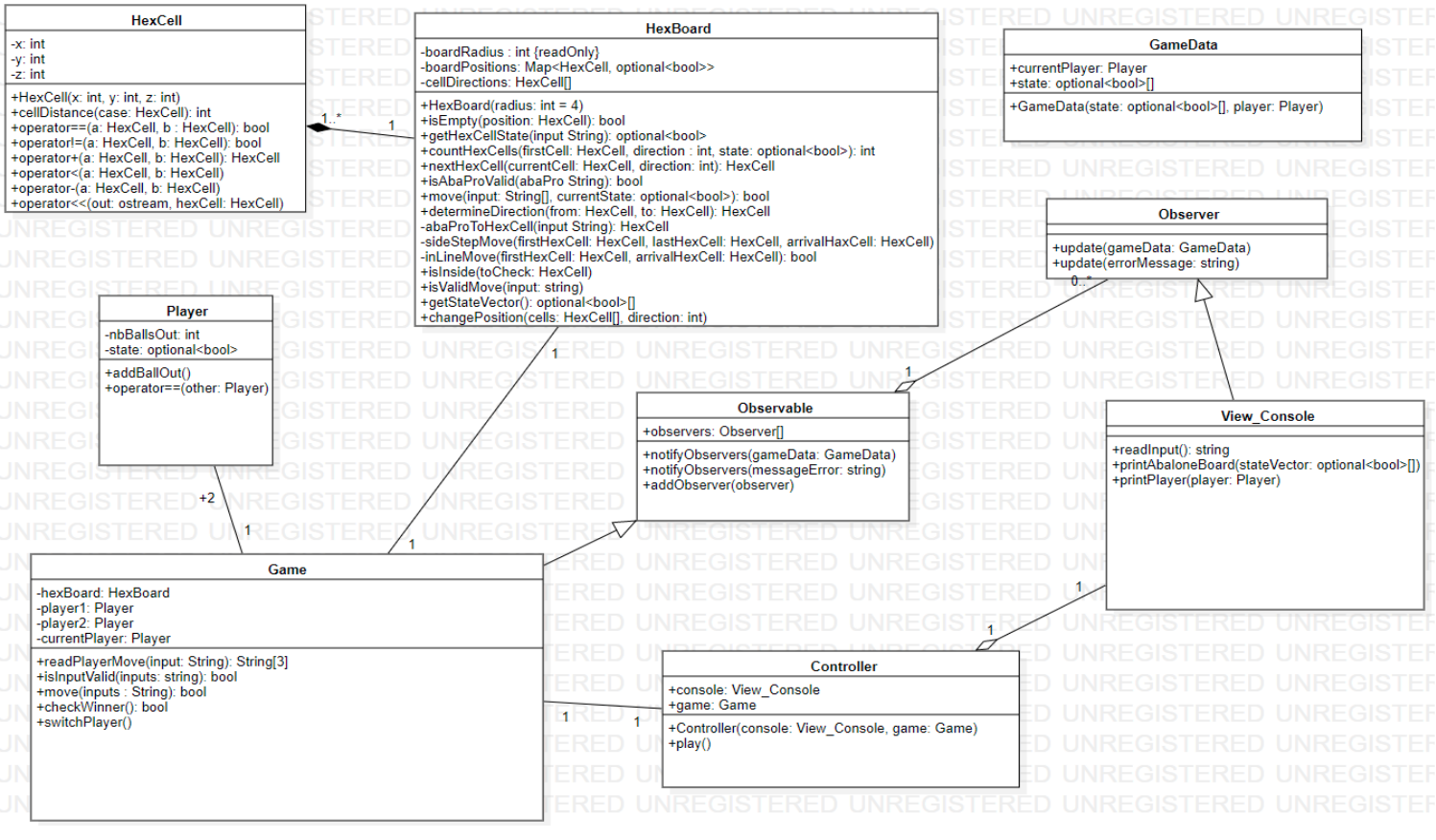
DisplayPopup affiche une fenêtre contextuelle apparaît afin de féliciter le joueur gagnant et de lui laisser le choix entre une nouvelle partie ou l'arrêt du jeu.

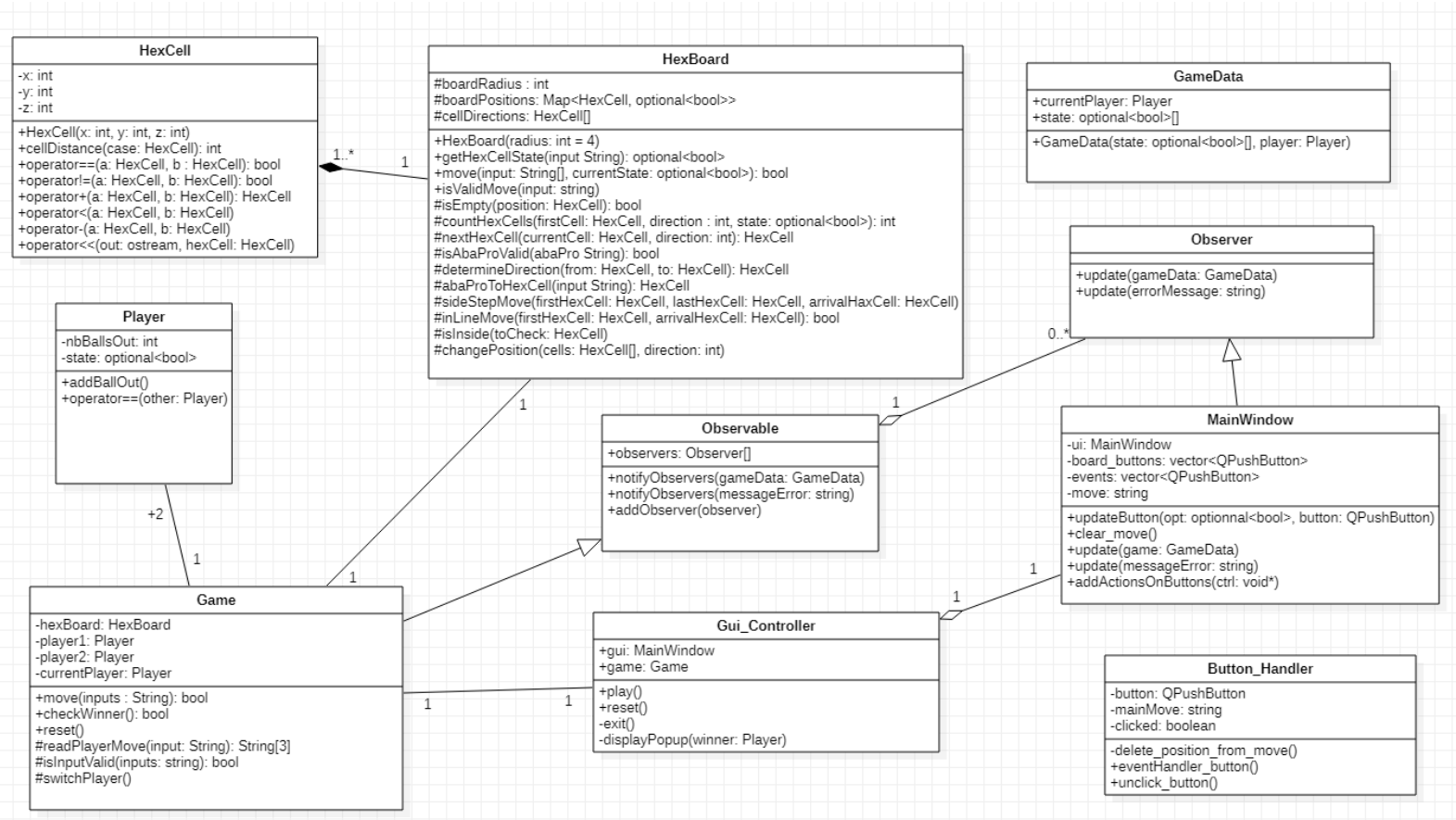
4.Annexes

4.1 Diagrammes de classe



NB : *Ces versions du diagramme de classe diffère légèrement de la version initiale. Ces changements sont expliqués par des approches différentes sur certains détails d'implémentations*





4.2 Inspirations

1. https://www.redblobgames.com/grids/hexagons/
2.