DEV4

Rapport remise GUI

Marika Winska (55047), Oscar Tison (55315)

Professeur : R. Absil

Haute École Bruxelles-Brabant

École Supérieure d'Informatique

7 Mai 2021

Table des matières

1. Introduction

Ce projet a pour but d'implémenter le jeu Abalone en C++ dans le cadre du cours DEV4. Abalone est un jeu de stratégie où deux joueurs s'affrontent. Chaque joueur commence avec 14 billes. Le premier joueur qui arrive à pousser 6 billes de son adversaire du plateau de jeu gagne.

La première remise contient tous les headers métier du projet avec une courte documentation des headers. Pour cette remise le design pattern « Observer / Observable » a été implémenté.

La seconde remise contient l'implémentation de tous les headers métier et de la partie console. Une version fonctionnelle du jeu a donc été implémentée.

2. Modélisation

Le projet est divisé en 7 grandes classes. Vous pouvez trouver le diagramme de classes en annexe de ce rapport. Ces classes seront expliquées une par une ci-dessous.

2.1. Game

La classe Game contient l'état du jeu. Elle a donc un attribut de type Board qui représente le plateau de jeu sur lequel les mouvements seront effectués. Il garde aussi en mémoire à quel joueur c'est au tour de jouer. On pourra contrôler si le jeu est fini en vérifiant à chaque tour si le joueur a gagné. Un Game est toujours créé avec un joueur 1 et un joueur 2. Cette classe contient toute la logique de jeu. Quand un joueur essaiera de faire un mouvement, on contrôlera dans le Game via la méthode *isMovePossible,* si c'est un mouvement licite en regardant si c'est bien au tour du joueur de jouer. Si le mouvement est possible, le Game ordonnera au tableau de jeu de changer la position des billes concernées. Une fonction a été ajoutée pour convertir un mouvement en *ABA-pro* vers un mouvement avec notre manière de représenter les positions (renseignée dans la classe Position).

La fonction *makeMove* est un peu plus complexe. Commençons par sa variante à 2 paramètres. Celle-ci pousse une bille. L'algorithme contrôle d'abord s'il n'y a pas plus de 2 billes du joueur qui joue devant celle qu'il pousse en suivant la direction du mouvement. Si la position qui suit le groupe du joueur est vide, on fait le déplacement. Si la position qui suit possède une bille du joueur adverse on contrôle dans la direction du mouvement s'il y a plus de bille du joueur adverse dans la direction ou du joueur. S'il y en a plus du joueur adverse, une exception est lancée, sinon le mouvement est effectué en poussant les billes de l'adversaire.

```
void Game::makeMove(Position posBegin, Position posEnd) {
    if (isMovePossible(posBegin) && posBegin.distance(posEnd) == 1) {
        Position direction =  posEnd - posBegin;
        Position nextPosition = posEnd;
        Position posForPlayer = posEnd;
        auto i = 0;
        auto j = 0;

        while (gameBoard_.playerAtPosition(nextPos))  {
            if (gameBoard_.playerAtPosition(nextPos)->id() !=
playerTurn()->nb()) {
                if (j==0) {
                    posForPlayer = nextPosition;
                }
                j++;
            } else {
                i++;
            }
            nextPosition = nextPosition + direction;
        }

        if (i<3 && j==0) {
            gameBoard_.changePosition(posBegin,nextPosition);
            changeTurn();
            notifyObservers();
        } else if (j<=i && i<3) {
            gameBoard_.changePosition(posForPlayer,nextPosition);
            gameBoard_.changePosition(posBegin,posForPlayer);
            changeTurn();
            notifyObservers();
        } else throw std::invalid_argument("Illegal movement !");
    } else throw std::invalid_argument("Illegal movement !");
}
```

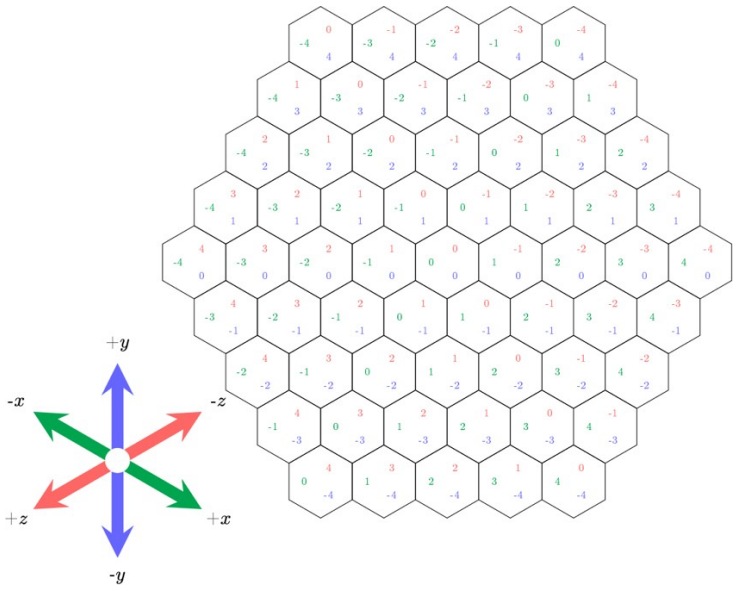
Le code ci-dessus fait comme suit : dans le while il cherche la prochaine position libre, tant que ce n'est pas le cas, on incrémente i si la position contient une bille du joueur actuel et j si c'est une bille de l'adversaire. Si i < 3 et j == 0, on pousse moins de 3 billes avec une position libre derrière, ce qui ne pose pas de souci. Pour faire ce mouvement on met la première bille à la position libre. Si j <= i et i < 3, on peut pousser un groupe de billes de l'adversaire, car l'adversaire a un plus petit groupe que le joueur actuel. Pour faire cela, on déplace la première bille de l'adversaire à la première position libre, et la première bille du joueur actuel à l'ancienne position de la première bille de l'adversaire. Si ce n'est pas le cas une exception est lancée, car le mouvement n'a pas pu être effectué.
Ceci sera le seul algorithme renseigné dans ce rapport car les autres sont assez triviaux.

Le makeMove à 3 paramètres va juste regarder si on a un groupe de maximum 3 billes et si la position où on veut les déplacer est bien libre.

La grande différence entre les 2 makeMove est que celui à 2 paramètres fait un mouvement en poussant une bille vers une direction, alors que celui à 3 paramètres déplace un groupe de billes vers une nouvelle position. On peut donc uniquement pousser un adversaire avec la variante à 2 paramètres.

2.2. Board

Le Board est la classe qui représente le plateau de jeu. Cette classe n'intervient pas dans la logique de jeu, mais uniquement dans le stockage des positions des billes. 2 solutions s'ouvraient à nous pour la représentation. La première est celle à 2 axes et la seconde à 3 axes. Nous avons choisi celle à 3 axes. En utilisant cette implémentation à 3 axes, les positions voisines seront plus facilement trouvables. Une implémentation à 2 axes aurait nécessité des calculs plus complexes pour trouver les positions voisines. Les billes seront donc stockées dans un tableau 3D selon leur position. Pour pouvoir représenter toutes les cases du plateau, chaque axe devra avoir la possibilité de prendre 9 valeurs différentes. Vu que nous ne pouvons pas représenter des index négatifs dans un tableau, il faudra ajouter 4 à chaque axe d'une position pour la stocker dans le tableau. Le centre du plateau sera la position (4,4,4), représenté ci-dessous par la position (0,0,0). Les positions autour de ce point central sont calculées comme spécifié sur l'image. On garde aussi la taille du Board en mémoire pour faciliter sa représentation pendant les prochaines remises.



2.3. Marble

La classe Marble représente une bille d'une couleur (blanche ou noire) définie par le joueur donné en paramètre du constructeur. La position est définie dans le tableau dans le Board. La classe Marble a un attribut Player, cela permettra de décrémenter le nombre de billes d'un joueur quand celle-ci est poussée du tableau.

2.4. Position

La classe Position représente une position jeu d'une bille sur le plateau de jeu grâce à trois axes. Le système de coordonnées sur 3 axes renseigné dans les consignes est utilisé pour représenter les positions. Ce choix a déjà été renseigné dans la classe Board. La fonction *abaToPosition* a été ajoutée. Celle-ci traduit un

string en ABA-pro en une position utilisant le système de coordonnées renseigné. Pour faire cela le string est comparé au centre du tableau E5 qui correspond à la position (4,4,4). Quand on a par exemple D5, on constate que cette position est à 1 char du centre. On ajoute donc (0,-1,1) à la position centrale. D5 correspond donc à (4,3,5). Et ainsi de suite.

2.5. Player

La classe Player représente un joueur par un id (1 pour les billes noires, 2 pour les billes blanches). Le Player aura un nombre de billes. Le Game va vérifier combien de billes chaque joueur a pour voir si un des joueurs a déjà perdu.

2.6. View

Cette classe observe le Game. A chaque fois que le Game notifie la classe, la méthode *printGame* est appelée pour imprimer le Game dans la console. Et si le jeu est gagné la fonction *printWin* est appelée, elle affichera un petit message pour féliciter le joueur qui a gagné. Pour afficher le jeu une triple boucle sur le gameBoard du Game est lancée. A chaque fois qu'on rencontre une position avec une bille, on imprime celle-ci. Avant chaque position de début d'un rang et après chaque position de fin d'un rang, des caractères sont imprimés pour délimiter le tableau de jeu.

2.7 HexaCell

Cette classe représente chaque case du tableau de jeu qui sera dessiné dans la partie gui du jeu. Chaque Hexacell a des attributs *selected* et *mouseover.* Ces attributs permettent de savoir comment dessiner le HexaCell a l'écran. Quand une case est sélectionnée, elle est affichée de couleur différente.

2.8 MainWindow

Cette classe observe la classe Game et l'affiche à l'écran. Dans le constructeur nous dessinons 1 fois tous les HexaCells. A chaque mouvement nous remettons les attributs mouseover et selected à false des HexaCells. Pour les mouvements dans le GUI nous avons décidé de réutiliser les mouvements comme en aba-pro pour ne pas devoir modifier l'implémentation métier du jeu. Lorsque nous cliquons sur une case, la position de la case se met dans un vecteur de Positions. Pour arriver à cela, chaque case est un observable de MainWindow. Quand une case est cliquée, elle notifie le MainWindow. Cela permet au MainWindow de récupérer la position de la case sélectionnée. Quand on appuie sur « Make move », la méthode *makeMove* du Game est appelée avec les éléments de ce vecteur. Quand le mouvement est effectué, le Game notifie le MainWindow pour qu'il se mette à jour. Quand le MainWindow se met à jour via sa méthode *update,* chaque Marble est à nouveau dessiné à l'écran et les informations de jeu sont mises à jour. En fin de jeu, les joueurs pourront décider s'ils veulent rejouer une partie ou s'ils veulent quitter le jeu.

3. Design pattern

3.1. Observer / observable

Nous avons implémenté le design pattern « observer / observable ». Le Game sera l'observable. Lorsqu'un mouvement sera effectué avec succès soit le jeu est gagné par un joueur soit on change de tour, le Game avertira ses observers que son état a changé. Ses observers devront à leur tour se mettre à jour. Comme observer du Game, nous avons choisi la View qui affichera le jeu à l'image. De ce fait, la View sera mise à jour lorsqu'un changement sera effectué dans le Game.

Dans la partie GUI du projet, ce sera le MainWindow qui observera le Game ainsi que les HexaCells représentants les cases du plateau de jeu. Le MainWindow se mettra à jour à chaque fois que le Game change d'état.

4. Conclusion

Pour cette première remise de la partie métier, nous avons décidé d'implémenter le design pattern « observer / observable ». Une version avec les headers de la partie métier est disponible sur git au lien suivant : https://git.esi-bru.be/55047/projet-dev4-55315-55047.git. Pour la seconde remise, une version console du jeu a été implémentée. Les joueurs doivent renseigner leurs coups en ABA-pro.

5. Référence

1. Refactoring, URL : https://refactoring.guru/fr/design-patterns/observer/cpp/example, (consulté le 18/02/2021)

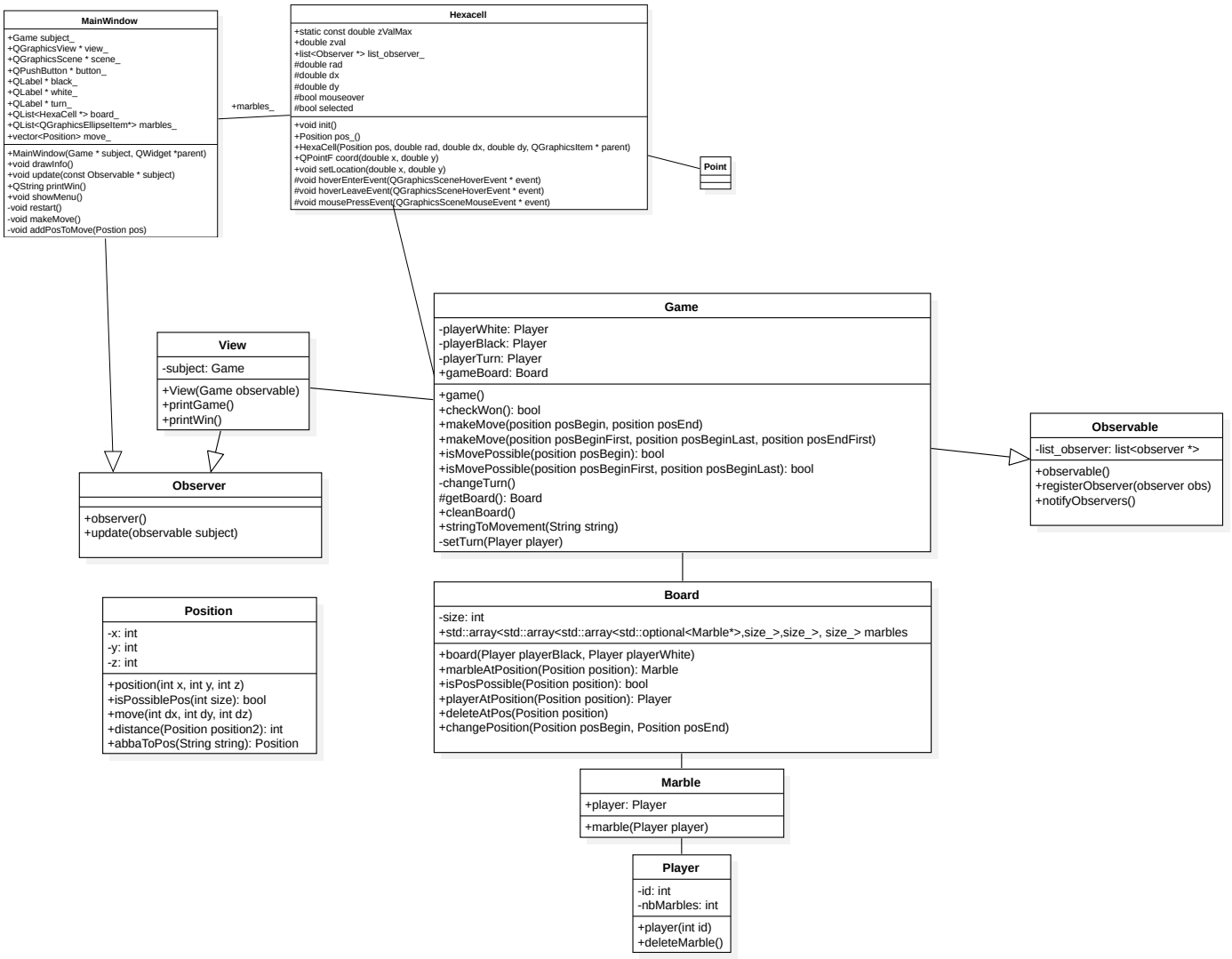
2. Stack Overflow, URL : https://stackoverflow.com/questions/318064/how-do-you-declare-an-interface-in-c , (consulté le 17/02/2021)

3. Geeks for Geeks, URL : https://www.geeksforgeeks.org/pure-virtual-functions-and-abstract-classes/, (consulté le 19/02/2021)

4. Doxygen, URL : https://www.doxygen.nl/manual/docblocks.html, (consulté le 18/02/2021)

5. Nicolas Vansteenkiste, l'implémentation de l'Observer / Observable trouvée sur la page du cours sur poESI.

6. Annexe


MainWindow
+Game subject_
+QGraphicsView * view_
+QGraphicsScene * scene_
+QPushButton * button_
+QLabel * black_
+QLabel * white_
+QLabel * turn_
+QList<HexaCell *> board_
+QList<QGraphicsEllipseItem*> marbles_
+vector<Position> move_
+MainWindow(Game * subject, QWidget *parent)
+void drawInfo()
+void update(const Observable * subject)
+QString printWin()
+void showMenu()
-void restart()
-void makeMove()
-void addPosToMove(Postion pos)
+marbles_
Hexacell
+static const double zValMax
+double zval
+list<Observer *> list_observer_
#double rad
#double dx
#double dy
#bool mouseover
#bool selected
+void init()
+Position pos_()
+HexaCell(Position pos, double rad, double dx, double dy, QGraphicsItem * parent)
+QPointF coord(double x, double y)
+void setLocation(double x, double y)
#void hoverEnterEvent(QGraphicsSceneHoverEvent * event)
#void hoverLeaveEvent(QGraphicsSceneHoverEvent * event)
#void mousePressEvent(QGraphicsSceneMouseEvent * event)
Point
Game
-playerWhite: Player
-playerBlack: Player
-playerTurn: Player
+gameBoard: Board
+game()
+checkWon(): bool
+makeMove(position posBegin, position posEnd)
+makeMove(position posBeginFirst, position posBeginLast, position posEndFirst)
+isMovePossible(position posBegin): bool
+isMovePossible(position posBeginFirst, position posBeginLast): bool
-changeTurn()
#getBoard(): Board
+cleanBoard()
+stringToMovement(String string)
-setTurn(Player player)
View
-subject: Game
+View(Game observable)
+printGame()
+printWin()
Observer
+observer()
+update(observable subject)
Observable
-list_observer: list<observer *>
+observable()
+registerObserver(observer obs)
+notifyObservers()
Position
-x: int
-y: int
-z: int
+position(int x, int y, int z)
+isPossiblePos(int size): bool
+move(int dx, int dy, int dz)
+distance(Position position2): int
+abbaToPos(String string): Position
Board
-size: int
+std::array<std::array<std::array<std::optional<Marble*>,size_>,size_>, size_> marbles
+board(Player playerBlack, Player playerWhite)
+marbleAtPosition(Position position): Marble
+isPosPossible(Position position): bool
+playerAtPosition(Position position): Player
+deleteAtPos(Position position)
+changePosition(Position posBegin, Position posEnd)
Marble
+player: Player
+marble(Player player)
Player
-id: int
-nbMarbles: int
+player(int id)
+deleteMarble()