DEV4

Rapport conception métier Abalone

Marika Winska (55047), Oscar Tison (55315)

Professeur : R. Absil

Haute École Bruxelles-Brabant

École Supérieure d’informatique

19 Février 2021

1. Introduction

Ce projet a comme 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.

Cette 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.

2. Modélisation

Le projet est divisé en 6 grandes classes. Vous pouvez trouver le diagramme de classes en annexe de ce rapport. Ces classes seront expliquées une par une ci-dessous.

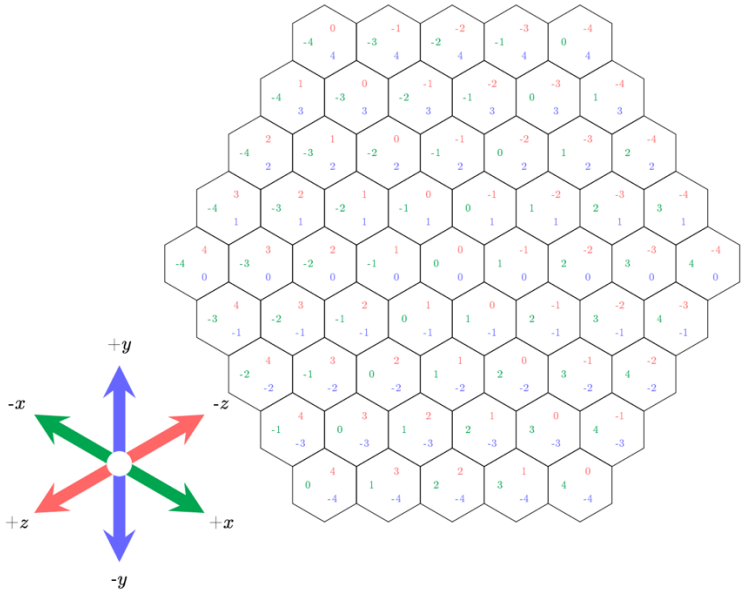
2.1. Game

Le Game est la classe qui contient l'état d'un jeu. Il a donc un attribut 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. Premièrement en regardant si c'est bien au tour du joueur de jouer, deuxièmement en contrôlant si le mouvement respecte bien les règles de jeu. Si le mouvement est possible, le Game ordonnera au tableau de jeu de changer la position des billes concernées.

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 (0,0,0). La position (0,0,0) se trouvera donc à marble[4][4][4]. Toutes les méthodes de la classe Board feront cette conversion implicitement. Les positions autour de ce point central sont calculées comme affiché ci-dessous. 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.

2.5. Player

La classe Player représente un joueur par un chiffre (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.

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 Vue qui affichera le jeu à l'image. De ce fait, la Vue sera mise à jour lorsqu'un changement sera effectué dans le Game.

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 .

5. Références

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)

Model::Main

6. Annexe