DEV4

Rapport du projet Abalone

2020-2021

Professeur : Romain ABSIL

Par : 54985 Amine-Ayoub Bigham et 54247 Zakaria Bendaimi

Table des matières

Introduction

Dans le cadre du cours de DEV4, le jeu de réflexion Abalone sera développé en C++, implémenté en trois parties, par groupe de deux, Amine-Ayoub Bigham et Zakaria Bendaimi. Le but de ce rapport est d'expliquer nos choix, nos décisions et de les motiver, mais aussi expliquer nos algorithmes non triviaux. Ce rapport accompagnera chacune des remises, à commencer par la première.

Cette remise consiste à produire une modélisation logicielle de la partie métier. Il nous fallait donc produire tous les fichiers headers, composés uniquement des prototypes de nos classes et des fonctions jugées nécessaires. Un diagramme de classe doit aussi être fourni. Ces éléments seront développés dans la section suivante.

Modélisation :

A. La structure

La structure choisie est QtCreator, respectant le modèle subdir. Ce choix a été fait par élimination, nous avions déjà l'habitude de travailler sous QtCreator. Il nous semblait plus optimal de travailler de cette façon.

B. Le design pattern

Le projet sera développé en suivant le design pattern Modèle-Vue-Controller. Ce design pattern est selon nous le plus clair, permettant une bonne découpe entre les différentes classes du métier, favorisant le principe de couplage faible et une forte cohésion. C'est aussi le design pattern que nous avons le plus utilisé jusqu'ici, il nous est donc plus familier et nous permettra de produire de meilleurs résultats.

C. Les classes métier

1. Position

Sur base du système de coordonnées présenté dans les consignes du projet et sur le tutoriel de Red Blob Games[1], nous avons décidé de créer une classe Position ayant comme attribut les trois coordonnées d'une case (Cell). Cela représente selon nous un moyen simple et efficace de représenter une grille hexagonale. L'avantage que l'on obtient en implémentant cette classe, est qu'il nous sera aisé de se repérer dans le plateau de jeu, mais surtout, les vérifications de positions qui sont nécessaires lors des déplacements de billes, mais aussi lors de la validation de l'entrée de l'utilisateur.

Une direction pourra être créée à partir de deux positions données, à l'aide de la méthode *computeDirection*. Cette méthode attend deux positions, et sur base d'un calcul simple

[1] https://www.redblobgames.com/grids/hexagons/

utilisant les coordonnées de ces positions, la direction sera déduite. La classe direction sera développée plus en détail au point suivant.

La classe position contient une méthode de conversion d'un string vers une position, nommé *toPosition.* Ce string représente une position en notation ABA-PRO, qui sera ensuite traduite en position de la classe Position.

La méthode *getNext* retourne la position adjacente à la position courante, dans la direction fournie en paramètre.

2. Direction

La classe Direction hérite de la classe Position puisque, en réalité, il s'agit d'une position comprenant des valeurs d'incrémentation ou de décrémentation de position (1, -1 et 0) sur l'axe X, Y ou Z. La direction nous permettra de diriger les billes sur la bonne voie.

3. Cell

La classe Cell représente une case du tableau de jeu. Elles sont au nombre de 61. Chaque case est logiquement composée d'une position, ainsi que d'une couleur. Cette couleur sera détaillée au point suivant. On retrouve au sein de classe Cell la méthode *isAdjacentTo*, qui prend en paramètre une autre cell. Comme son nom l'indique, elle permet de vérifier si une case est adjacente à une autre ou non.

4. Énumération Color

Pas de classe joueur, ni de classe bille, mais une énumération Color. Ses valeurs sont NONE, WHITE ou BLACK. Elles sont utilisées pour connaître la couleur de la bille sur une case, mais aussi sa non-présence. Dans la classe Game, elle permet de retenir le joueur courant et donc de permettre l'intégrité des méthodes de mouvement, ainsi que le compteur de billes perdues pour chaque joueur.

5. Board

Il s'agit du plateau de jeu, il contient toutes les cases et un compte le nombre de billes perdues pour chaque couleur. Elle permet d'exercer des mouvements (ligne ou latéral) et de tenir compte de leur intégrité. Ces mouvements sont effectués à l'aide des méthodes *move* de la classe Board. Ce qui différencie ces deux méthodes est le nombre de paramètres attendus. Pour un mouvement en ligne, seul deux positions sont requises, alors que pour un mouvement latéral, trois positions le sont.

Les méthodes *canMove*, servent à déterminer pour les deux types de mouvement, si ce dernier peut être effectué ou pas. Les conditions devant être respectées avant de bouger des billes varient en fonction du type de mouvement.

6. Game

La classe Game rassemble tous les éléments nécessaires au jeu pour présenter une façade au contrôleur. Ce dernier n'interagira qu'avec cette classe pour accéder au modèle.

La méthode *askAbaPro* sert à lire une position en notation ABA-PRO fournie par un joueur. Après avoir vérifié la validité de cette position, elle sera convertie en une instance de Position.

7. Controller

Le contrôleur orchestre les interactions sur le modèle en fonction des entrées de l'utilisateur à travers à la classe Game, qui elle-même assure aussi le bon fonctionnement du jeu. Le contrôleur mobilise aussi la classe View pour assurer le feedback visuel aux joueurs.

D. Problèmes rencontrés

Durant cette première étape de modélisation, nous avons rencontré une difficulté dans le choix d'implémentation du conteneur de cases. Notre idée initiale était d'opter pour un conteneur séquentiel, l'*array* de la librairie standard. En effet, il ne s'agissait pas du meilleur choix puisqu'à chaque mouvement de bille, le conteneur devait être parcouru du début jusqu'à tomber sur la case désirée et cela impacte fortement les performances. Nous avons remédié à cela en choisissant une *unordered_map* comme conteneur de cases. Celui-ci permet de retrouver rapidement les cases sur base de leurs positions hachée. Cette implémentation avait aussi été recommandée dans le tutoriel de Red Blob Games.

Implémentation console

A. Déroulement

Grâce à l'accent mis sur l'analyse effectuée au préalable, nous avons pu directement nous lancer dans la rédaction du code sans devoir effectuer de réflexions supplémentaires. Une ligne directrice était définie clairement et il suffisait de la suivre pour achever notre objectif dans sa totalité.

De plus, à chaque rédaction de méthode, des tests correspondants y étaient rajoutés immédiatement. C'est de cette façon sûre qu'a été conduit notre travail. Chaque faute d'inattention était donc corrigée à l'instantané, ne laissant pas la place à de futurs bugs ou mal fonctionnements dans le bon déroulement du jeu.

B. Différences face à l'analyse initiale

Les oublis

Notre analyse n'était pas pour autant infaillible. En effet, certaines méthodes n'ayant pas été prises en compte dans celle-ci, ont dû être rajoutée à la volée. Par exemple, une méthode faisant correspondre chaque lettre et chiffre de la notation ABAPRO vers son axe sur le plateau de jeu, pour ensuite en faire une position. Ou encore une méthode pouvant vérifier la correspondance d'une paire lettre-nombre entrée par les joueurs.

Les superflus

Certaines méthodes conceptualisées lors de l'analyse se sont révélées inutiles une fois arrivés à la fin du développement des classes. C'est le cas pour les méthodes hasSameColor, isAdjacentTo, et les méthodes de surcharge d'opérateurs. Elles ont alors été supprimées.

C. Problèmes rencontrés

Les problèmes que nous avons pu rencontrer étaient souvent causés par inattention, et ont pu être rapidement détectés et corrigés par les tests. Cependant, nous avons tout de même fait face à un problème lié au parcours de positions. En effet, les positions ayant des attributs constants, il nous était impossible d'assigner une position à une autre, ce qui nous empêchait d'avancer dans le parcours de positions, notamment dans la méthode chargée de compter le nombre de billes d'une couleur donnée. La solution à ce problème a été d'opter pour une méthode récursive qui attendait en paramètre la position suivante dans le parcours. Ainsi, un constructeur de positions était appelé pour atteindre la prochaine position, sans devoir passer par une assignation.

Lorsque nous avions fini de développer la partie console, nous nous sommes mis à nous questionner quant à l'utilité de la classe *Cell*. En effet, cette dernière semble être une simple enveloppe contenant une couleur et une position. Nous avons alors pensé à supprimer la classe *Cell*, et de fonctionner tout simplement avec le *unordered_map* contenant pour chaque position, une couleur. De cette manière, nous n'aurions plus besoin d'instancier des objets de type *Cell*, mais comme les *cells* sont présentes dans plusieurs classes, la supprimer demanderait de modifier une grande partie du projet. D'un autre côté, garder cette classe permet de conserver une certaine lisibilité et une meilleure compréhension du code en général. Nous avons alors décidé de conserver la classe *Cell*.

Le dernier problème survenu était les comportements inattendus de certaines billes lors des déplacements, que nos tests n'ont pas pu détecter et qui ont été découverts en jouant des parties tout en essayant de provoquer volontairement des bugs. Ces erreurs ont alors directement été corrigées et les tests renforcés.

D. Conclusion

Une leçon capitale que nous a fait apprendre cette implémentation est donc l'importance d'une analyse approfondie et de tests réguliers sur chaque aspect du programme au fur et à mesure de son développement.

Nous retenons donc beaucoup plus que ce qu'on espérait, car il s'agit ici d'une réelle application des notions vues jusqu'ici dans de divers cours théoriques.