Nael TALBI
Nicolas BERTHIER

Documentation développeur

I Architecture

Notre application doit nous permettre de jouer au Morpion Solitaire selon deux modes de jeux (5T et 5D), et ce, avec une interface graphique. Le Morpion Solitaire est un jeu où l'on place des points sur une grille pour créer des alignements[1]. Pour cela, nous avons structuré notre code avec une architecture Modèle-vue-contrôleur (MVC). Cette architecture permet de bien distinguer le modèle, à savoir le jeu et ses données, des éléments graphiques, et de la logique liés aux événements-utilisateurs, ce qui nous a paru particulièrement approprié pour notre jeu.

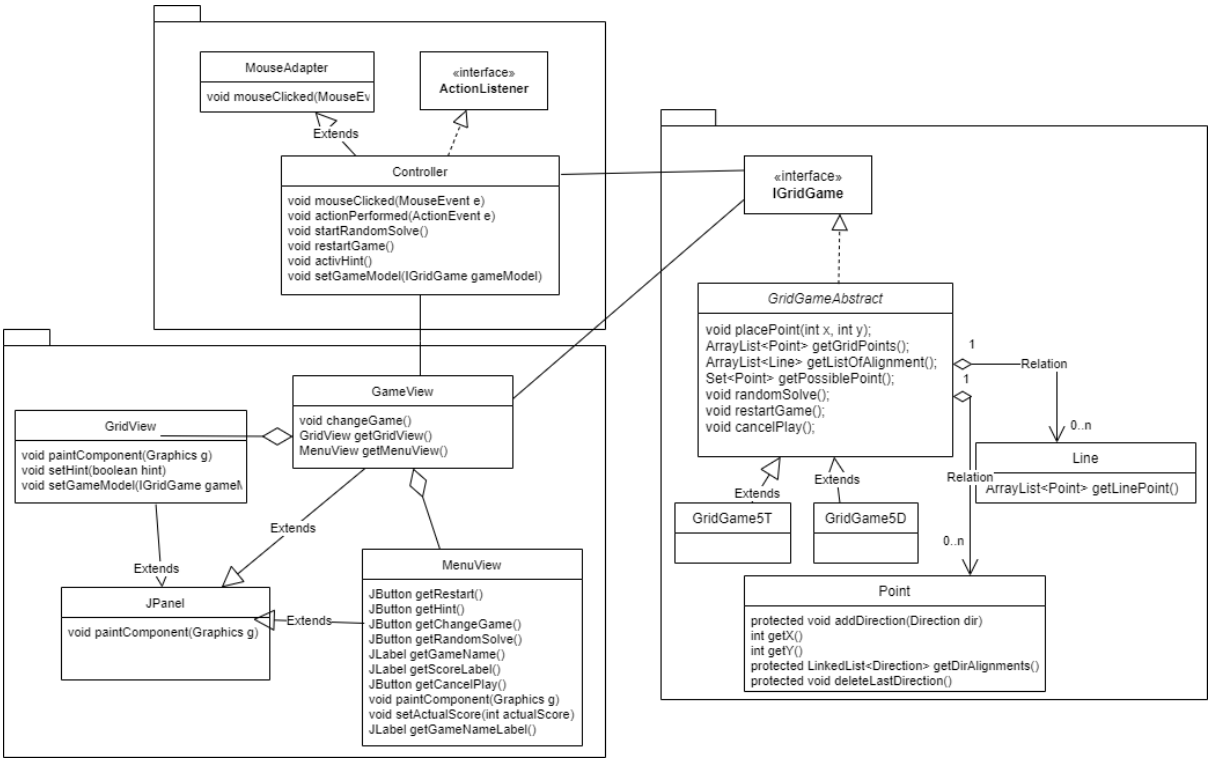

Fig. Diagramme de classe du projet Morpion Solitaire[2]

Sur le diagramme, seules les méthodes visibles à l'extérieur des classes ont été représentées. Les méthodes où la visibilité n'est pas indiquée sont des méthodes publiques. Nous ne l'avons pas indiqué par souci de lisibilité.

[1] https://fr.wikipedia.org/wiki/Morpion_solitaire

2 Les trois packages de notre MVC sont représentés. Pour faciliter la lecture de notre diagramme, nous avons volontairement omis le package "utils" qui reprend des classes utiles à l'ensemble de notre projet (classe de Comparator<Point> et une enum de directions.)

II Les packages

1) Le Modèle

Pour implémenter notre MVC nous avons structuré notre code en trois répertoires distincts (model, view et controller). Notre modèle comprend une interface IGridGame qui reprend les principales fonctions d'un jeu de grille (placer un point, savoir quels points sont placés, quels sont les alignements déjà faits, annuler un coup, recommencer une partie). Elle permet aussi de créer un contrat, de structurer les intéractions de notre modèle avec les classes extérieures.

La classe abstraite

Les modes de jeux 5T et 5D sont très proches et ne se distinguent que par la manière de placer un point (ou de créer un nouvel alignement). Pour factoriser un maximum de code, nous avons développé la plupart de nos méthodes dans la classe abstraite GridGameAbstract. Ses méthodes sont les plus génériques possibles, par exemple celles liées aux détections des alignements ou au placement des points prennent en argument la longueur des alignements attendus et le nombre de points déjà alignés dans la même direction que peut contenir un nouvel alignement. Ainsi, les classes GridGame5D et GridGame5T redéfinissent les méthodes de placement d'un point en utilisant les méthodes de leur classe mère (GridGameAbstract) avec les bons paramètres d'alignement de points (5 et 0 pour 5D, 5 et 1 pour 5T)[3].

La méthode de recherche d'alignement

Pour trouver un alignement, nous avons développé les méthodes les plus "locales" possible (AlignedPointsByDirection() et placePoint()) pour avoir une complexité faible. Lorsque le joueur propose un nouveau point, nous recherchons autour du point proposé s'il existe un autre point. S'il existe un point dans une direction donnée, nous continuons d'avancer dans la direction tant qu'il existe un point. Lorsqu'on ne trouve plus de points, on recommence dans l'autre sens. A la fin, nous nous retrouvons avec un alignement plus ou moins important de points, ce qui nous permet de valider l'alignement ou non. Cette méthode légèrement modifiée permettrait aussi de renvoyer une liste d'alignements lorsque plusieurs alignements possibles se présenteraient.

---

3 Pour constituer un alignement il faut 5 jetons, aucun point ne doit déjà être aligné dans la même direction (5D). Pour constituer un alignement, il faut 5 jetons, un point au maximum doit être aligné dans la même direction (5T).

C'est également avec cette méthode que l'on propose des résolutions automatiques (random) en testant les intersections voisines autour des points de la grille pour voir si un point placé pourrait former un nouvel alignement. Si un point non placé permet un alignement, il est mis dans une liste, liste dans laquelle on tire au sort un point aléatoirement pour le placer. Lorsqu'il est placé, on recommence avec les nouveaux voisins, et ce, tant que la liste des points potentiels est supérieure à 0.

Les Points

Les points sont contenus dans une ArrayList de la classe abstraite GridGameAbstract. Un Point se définit par sa position sur la grille. Il contient ainsi deux attributs, x et y. Ce sont ces seuls éléments qui permettent de vérifier l'égalité entre deux points tel que nous avons développé la méthode equals(). Cela permet rapidement de savoir si un point existe déjà dans la grille. Un point se définit également par les directions de ses divers alignements. Il peut concourir à un alignement dans chaque direction en 5D, et à au moins un alignement dans chaque direction en 5T. Lors de l'annulation d'un coup nous devons être en capacité de supprimer cette dernière direction pour permettre à nouveau un alignement dans cette direction. Pour cela nous utilisons la méthode removeLast() de la LinkedList contenant les directions d'alignements.

Les alignements (Line)

Les Line (lignes) représentent les différents alignements créés par les placements de points. Une liste de lignes est contenue dans la classe abstraite GridGameAbstract. Une ligne se définit par une succession de Points (5 dans le cadre de nos jeux actuels). Les points sont enregistrés dans une ArrayList et sont triés par leurs valeurs (x et y) pour distinguer les deux plus éloignés. Cette distinction nous permet de tracer les lignes plus facilement dans notre interface graphique.

2) La Vue

Pour notre vue nous avons utilisé la bibliothèque graphique Swing et plus particulièrement la classe JPanel dont nos classes héritent et redéfinissent la méthode paintComponent(). Notre vue est composée de trois classes, GameView, GridView et MenuView. GameView comprend les paramètres généraux de la vue, et les méthodes d'initialisation du jeu. GridView quant à elle, gère la vue de la grille. C'est elle qui permet d'afficher la grille, points, les alignements. Enfin, MenuView génère des boutons ajoutés en haut de la fenêtre et reprend les actions possibles au joueur, parmi elles : montrer les points encore jouables à l'utilisateur, annuler un coup, recommencer la partie, enclencher une recherche aléatoire à tout moment, ou encore changer de jeu.

3) Le Controller

Notre controller implémente les classes MouseAdapter et ActionListener. Nous avons souhaité utiliser la position du clic de la souris pour permettre à l'utilisateur de placer à point. Pour que l'usage de la souris soit agréable nous avons créé des hitbox autour des intersections de lignes et de colonnes.

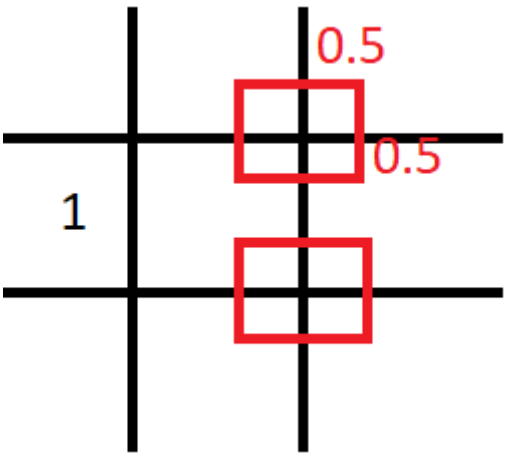

fig. Schéma représentant les zones de clique (hitbox)

Une zone représentée par des carrés de 0.5 de côté (vs 1 pour une largeur entre deux lignes ou deux colonnes) sur le schéma ci-dessus permettent de filtrer les cliques de l'utilisateur. Si un clique n'a pas lieu dans cette zone, le controller filtre et n'essaie pas de contacter le modèle pour placer un point.

Notre contrôleur gère également les cliques sur les différents boutons du menu de notre vue grâce notamment à la méthode actionPerformed() de la classe ActionListener implémentée. Nous réagissons aux cliques de l'utilisateur sur les boutons en lançant des actions en conséquence du côté de la vue ou encore du modèle. Par exemple, en cliquant sur le bouton "Restart", nous utilisons une méthode du modèle qui instancie une nouvelle grille. La vue est directement mise au courant du changement et remet la grille de jeu à zéro.

III Les fonctionnalités développées

1) Score

Le score est affiché et est continuellement actualisé au fur et à mesure de l'ajout de points. Il correspond aux nombres de lignes tracées, c'est-à-dire la taille de listOfAlignment.

2) Can you continue ?

Cette fonctionnalité permet de proposer à l'utilisateur toutes les possibilités de placement de points. Nous regardons tous les voisins des points de notre grille et vérifions s'ils permettent un alignement. Si c'est le cas, nous les ajoutons à un Set de points. En outre, lorsque l'utilisateur clique sur le bouton "Can you continue ?" et qu'aucun point n'est jouable un "Game over" apparaît dans un JLabel au niveau du menu.

3) Random solution

Permet de générer un déroulement de partie aléatoire jusqu'à ce qu'on ne puisse plus ajouter de points supplémentaires. Cette fonctionnalité s'appuie sur la fonctionnalité précédente. Un point parmi les points permettant un alignement est sélectionné aléatoirement, et est placé.

4) Cancel

Cette fonctionnalité permet à l'utilisateur d'annuler le dernier coup joué et peut être utilisé plusieurs fois (jusqu'à revenir à une grille sans ligne). De ce fait, le dernier point joué est enlevé et la ligne qui avait été tracée également. Nous faisons également attention à bien retirer la dernière direction d'alignement enregistrée pour chacun des points de la ligne pour qu'ils puissent à nouveau autoriser un alignement dans cette direction.

5) Restart

Réinitialise la grille et le score qui lui était associé.

6) Switch Game (5D/5T)

Fonctionnalité permettant de changer de grille de jeu au moment souhaité. La version du jeu en cours d'utilisation est indiquée à côté du bouton.

IV Fonctionnalités non explorées

Par manque de temps nous n'avons pas pu mettre en place d'autres fonctionnalités (Même si, nous avions entamé des réflexions à ce sujet) telles que :

- L'ajout d'autres grilles (exemple : 5D# qui pourrait être permise avec une modification de la méthode d'initialisation ou encore en permettant au constructeur de prendre une List qui contiendrait nos points de départ en argument ;
- L'algorithme NMCS ;
- Multithreading.

IV Difficultés rencontrées

Dans un premier temps, ne connaissant pas ce jeu et ses règles, nous avons eu du mal à percevoir quels étaient les enjeux derrière le fait de placer un point (le fait qu'un point puisse être aligné dans plusieurs directions mais pas deux fois dans la même, etc…). Nous avons appris à comprendre et connaître les subtilités du Morpion Solitaire, en le développant.

Lors du développement, nous avons également rencontré plusieurs erreurs de pointeurs. En effet, dans un premier temps nous avions mal instanciées certaines de nos classes. En conséquence, notre vue ne pointait pas vers la même instance que notre contrôleur par exemple.