Bilal Brahimi
Yanis Laghmouri

Morpion Solitaire

Documentation Développeur

Sommaire

I) Architecture choisie

1) Le modèle MVC

La première des étapes fut de réfléchir à une architecture pour notre projet. Suite aux raisonnements que nous avons menés, nous avons jugé nécessaire l'utilisation d'une architecture étudiée durant nos précédentes séances de cours : l'architecture MVC (Modèle – Vue – Contrôleur).

Le choix de ce Design Pattern nous permettrait de bien organiser notre code source, tout d'abord en trois parties distinctes.
Une couche **Modèle** qui contiendrait le code relatif au développement du jeu et ses **données**.
Le modèle offrira des méthodes pour mettre à jour les données (*exemple : insertion de point sur la grille de jeu*).
La **Vue** représente l'interface utilisateur, c'est avec cette dernière que l'utilisateur communiquera. Elle jouera principalement un rôle d'affichage. C'est une sorte d'intermédiaire avec l'utilisateur. Elle affichera les données que notre couche modèle lui fournira.
Le **Contrôleur** recevra les actions faites par l'utilisateur (fournies par la Vue) en enclenchera les actions à effectuer sur notre Modèle.

Schéma classique du modèle MVC lorsque l'utilisateur jouera un coup :

- L'utilisateur joue un coup sur la grille (clique sur un point jouable).
- La Vue envoie cet évènement au Contrôleur.
  L'action est captée par le Contrôleur, qui peut la transformer avant de la transmettre au modèle.
- Le modèle reçoit ces données et met à jour son état (le point est joué, place au coup suivant).
- Ensuite, le modèle notifie la vue pour qu'elle se mette à jour.
- L'affichage (le point jouable se transforme en point joué) suite au nouvel état du modèle.

2) Organisation des classes : Les Packages

Le packetage est important car il sert à structurer l'ensemble de nos classes. Nous avons décidé de définir trois packages, en cohérence avec l'architecture MVC choisie : l'un nommé **Front** (Vue), **Back** (Modèle), et le dernier **Controller**.

a) Front

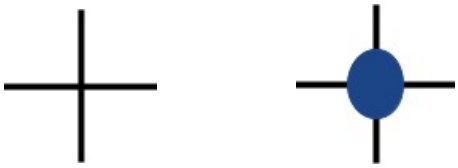
Dans ce package nous avons regroupé les classes dédiées à l'interface graphique de notre projet java. On compte **PrincipalScene** qui représente l'écran de déroulement d'une partie.

Aussi la classe **App** qui constitue l'écran principal de notre interface graphique.
La classe **Main**, elle, a pour rôle de lancer notre jeu, c'est notre « Launcher ».

b) Back

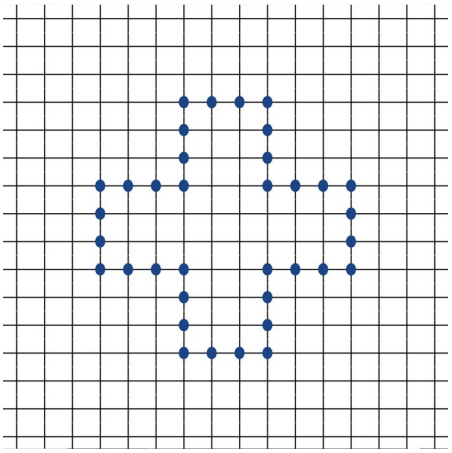
Ce package est organisé de sorte à contenir toute les classes contenant le code relatif au back end de notre jeu.
On y retrouve les classes suivantes :

- La classe **Point** :
Elle représente un point, l'élément élémentaire du projet.
Un point est défini par ses deux coordonnées, respectivement x (abscisse) et y (ordonnée).
Il possède un statut nommé state, dont la valeur détermine son apparence.



State vaut -1 state vaut 0

- La classe **Grid** :
Cette classe crée la grille du jeu.
Elle initialise les bordures du jeu de départ en modifiant le statut de certains points à 0 de sorte à créer le plateau suivant :



- La classe **OrientationLine** :
Cette classe est une **Enumeration** qui définit les différentes orientations que peut prendre une ligne, à savoir l'horizontale, la verticale, une première diagonale D1 et une seconde diagonale D2.

- La classe **Line** :
Elle représente une ligne.
Une ligne est caractérisée par son **point de départ p_start**, son **point**

d'arrivée p_end ainsi que par son **orientation** définie par la classe **OrientationLine**.

- La classe **GameEvolution** :
A chaque action de l'utilisateur, le jeu évolue et passe par différents **états.**
Cette classe représente l'évolution de la partie au fil des actions de l'utilisateur.
La méthode list_of_playable_lines est le cœur du jeu car elle propose les lignes jouables pour un point. Elle parcourt la grille 4 fois, et voici le fonctionnement pour ces 4 itérations :
Pour la premier parcours, elle cherche les lignes horizontales et les stocke.
Pour le second parcours, elle cherche les lignes verticales et les stocke.
Pour le troisième parcours, elle cherche les lignes dans la première diagonale et les stocke.
Pour le dernier parcours, elle cherche les lignes dans la seconde diagonale et les stocke.

- L'interface **Version** :
C'est une interface qui contient une méthode à implémenter lorsque l'on souhaite ajouter un mode de jeu (ici 5D, 5T, 4T et 4D).
Cette méthode, nommée *is_point_usable* définit si un point est jouable en fonction du mode de jeu (notion de chevauchement entre les lignes).

- La classe **Tversion** :
Implémente la méthode *is_point_usable* au mode de jeu 5T et 4T.
En version T, elle s'assurera que le point p soit en dehors de toute ligne ou à une extrémité d'une ligne pour autoriser le chevauchement et que le point soit jouable.

- La classe **Dversion** :
Implémente la méthode *is_point_usable* au mode de jeu 5D et 4D.
En version D, elle s'assurera que le point p soit en dehors de toute ligne, en d'autres termes, pour que le chevauchement entre lignes ne soit pas possible

- La classe **RandomGame** :
Contient les algorithmes relatifs à la recherche aléatoire utilisée par l'ordinateur.
L'algorithme cherche tous les points jouables et en choisit un aléatoirement.
Cette action est réalisée de manière récursive jusqu'à ce que n'y ait plus de point jouable.

- La classe **PlayablePoint** :
Elle définit les différents points qui sont jouables.
Ses attributs sont un point, ainsi que la liste des lignes traçables à partir de celui-ci.

Apparence d'un point jouable

- La classe **PlayablePointRandom** :
Point jouable joué par l'ordinateur.
Représente le point qui est jouable et joué par l'IA de manière aléatoire.

- La classe **GlutonGame** :

Contient les algorithmes relatifs à un second type de recherche appelé « Glouton » utilisée par l'ordinateur dans le mode de jeu Gluton Game.
Il parcourt tout les points jouables, crée une copie du jeu pour chaque point. Pour chacune de ces copies, il analyse le nombre de points jouables qui résultent du coup joué. Il récupère les coordonnées de ce point où ce nombre est le plus grand, et le joue dans le jeu original.

c) Controller

- La classe **GameController :**
Elle permet au joueur de contrôler la partie, d'effectuer des actions (des coups) sur le jeu.

- La classe **RandomGameController :**
Permet à l'ordinateur de lancer une partie Random Game, et utilise comme algorithme de recherche celui implémenté dans la classe RandomGame.

- La classe **GlutonGameController** :
Permet à l'ordinateur de lancer une partie Gluton Game, et utilise comme algorithme de recherche celui implémenté dans la classe Gluton Game.

II) Difficultés rencontrées

1) Architecture

Le choix d'une architecture permettant d'optimiser notre code n'a pas été facile. Tout les choix que nous avons fait ont été travaillé, certains ont été gardés, d'autres non.
Par exemple, le choix d'une interface Version accompagnée de classes Tversion Dversion l'implémentant (afin de ne pas tout réinventer si l'on veut ajouter un nouveau mode de jeu) a été un choix issu d'un travail réfléchi.
C'est de cette façon que nous avons pu éviter des redondances de code quand nous le pouvions.

Autre exemple de choix d'architecture qui a été travaillé au fil du temps : l'utilisation d'une variable *lineSize* différenciant à elle seule les modes de jeux 4D/4T et 5D/5T (dans le mode de jeu 4D/4T, ce sont quatre points consécutifs qui suffisent à créer une ligne). Ainsi, on évite la création de classes supplémentaires.

2) La classe GameEvolution

Cette classe GameEvolution a été la plus dure à implémenter : elle constitue le moteur du jeu.
Nous avons rencontré les plus grandes difficultés sur la conception de

l’algorithme visant à trouver la liste des lignes traçables à partir d’un point P. Cette difficulté vient du fait qu’on doit parcourir tous les points de toutes les directions afin de voir toutes les lignes qui peuvent être possiblement tracées.

III) Récapitulatif des fonctionnalités et partage des tâches

1) Fonctionnalités apportées

Au départ du projet, les fonctionnalités que nous visions étaient tout d’abord bien sûr les trois attendues au minimum, à savoir :

- Un moteur de jeu qui permet `a un utilisateur de jouer une partie en 5D ou en 5T.

- Une méthode de recherche de solution automatique aléatoire.

- Une interface graphique pour interagir et observer.

En plus de celles-ci, nous nous sommes fixés comme objectif de remplir quelques extensions.

Nous avons apporté toutes les fonctionnalités fondamentales du projet.
Concernant les extensions, nous avons proposé **l’affichage du score** au cours de la partie.
Ce score est **sauvegardé dans un fichier**, de sorte à ce que le meilleur score s’affiche sur l’écran principal.

Aussi, un **2ème algorithme de recherche de solution** a été implémenté : un algorithme Glouton.

En plus de toutes ces extensions, nous avons développé les modes de jeux **4D et 4T**.
Dans ces modes de jeux, il faut placer quatre points consécutifs pour tracer une ligne.

2) Points non abordés

En ce qui concerne les autres extensions facultatives comme l’implémentation du NMCS ou le multi-threading. Bien que nous voulions réaliser ces points, nous n’avons au final pas pu les aborder par faute de temps.

3) Répartition travail/groupe

Le projet a débuté avec une réflexion commune autour du choix de l’architecture que nous allions choisir.

Nous avons défini comment communiquer au fur et à mesure du projet, c’est ainsi que nous avons mis en place des points réguliers (tous les deux jours). Ces points étaient des réunions Teams, dont le but était de définir les objectifs à atteindre avant la prochaine réunion.
En dehors de celles-ci, il faut compter aussi des réunions informelles, des échanges par messages pour mieux s’organiser et avancer de façon efficace.
De manière générale, Bilal s’est occupé de la partie Back du projet, Yanis de la partie Front.
Nous organisions aussi des sessions (à l’occasion des points réguliers) des sessions où l’on réfléchissait ensemble. Ces sessions ont eu lieu notamment lors du développement des algorithmes de recherche (tels que les algorithmes de recherche de solutions Random et Glouton) car ils nécessitaient une réflexion collective et que chacun soit au courant du code développé.