Manon Baha et Lukas Tabouri

Projet de POOING

Domino carré et Carcassonne

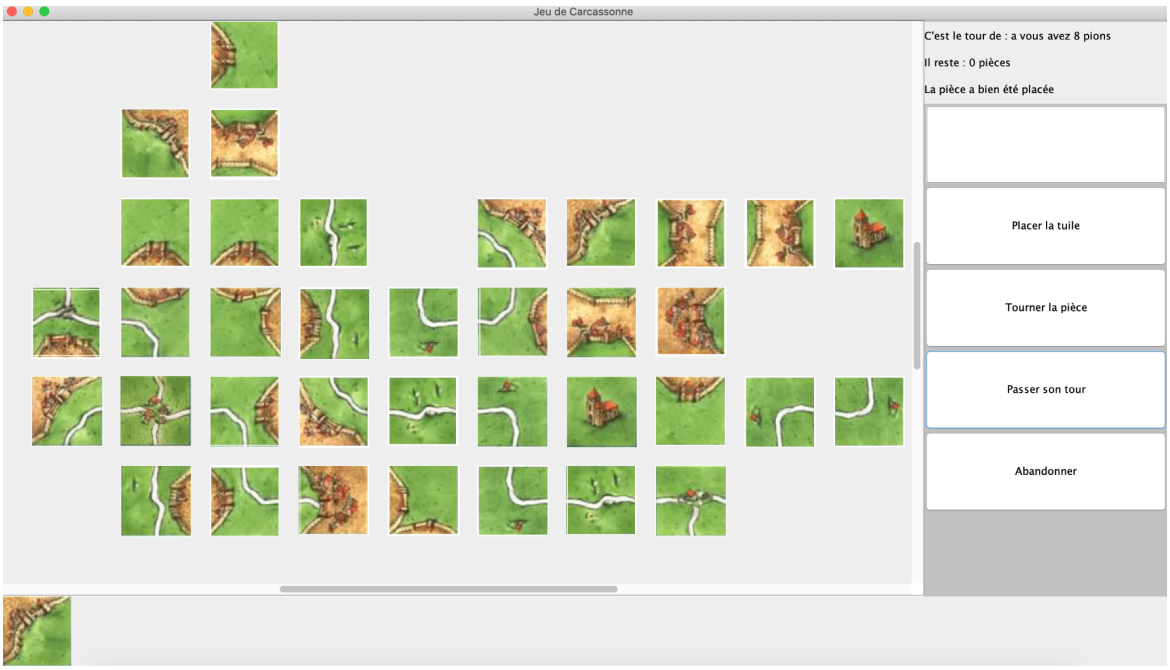

Fig 1. Interface graphique de Carcassonne

Introduction

Nous devions réaliser un projet dans le cadre du cours de programmation orientée objet de la deuxième année de licence d'informatique. Ce projet consistait à coder deux jeux : les dominos carrés et le jeu de plateau *Carcassonne*.

I. Organisation du projet

La première étape a bien évidemment été de prendre du temps pour bien appréhender ce qui nous était demandé, et de comprendre les différentes règles et subtilités des deux jeux que nous devions implémenter.

La première question que nous avons dû nous poser pour commencer le projet, et notre première problématique, fût l'organisation que nous allions développer. Nous avons réfléchi à la méthode à utiliser, la manière de se répartir les tâches, les fonctionnalités par lesquelles nous voulions commencer. Nous avons utilisé github pour travailler ensemble efficacement.

II. Cahier des charges

1. Modèle

La première partie du projet que nous avons implémentée était le Modèle du Domino. C'était une partie assez simple au niveau du code en lui-même mais c'est une partie qui nous a néanmoins demandé du temps de réflexion quant aux différentes implémentations possibles . Ce temps que nous avons consacré à la réflexion nous a beaucoup aidé sur la suite du projet. Lorsque nous avons par la suite commencé l'implémentation de carcassonne, nous avions déjà une forte idée de quelles classes devaient rester communes aux deux jeux.

a. Différentes classes du modèle :

Nous avons essayé de garder le plus de classes possibles communes aux deux jeux, tant que celà avait du sens.

- Une classe abstraite Tuile est étendue par deux classes : TuileCarcassonne et TuileDomino. Ces classes permettent de définir les pièces des deux jeux. Elles ne peuvent pas constituer une seule et même classe car elles sont différentes, mais elles ont des fonctionnalités communes : c'est pour celà que la classe abstraite Tuile

possède les méthodes abstraites pour tourner une pièce et comparer les côtés de deux pièces. Nous reviendrons plus en détail sur ces tuiles dans la suite du rapport.

- Une classe abstraite Sac est étendue par deux classes : SacCarcassonne et SacDomino. Ces classes sont différentes car elles possèdent respectivement soit des TuileCarcassonne soit des TuileDomino. Les TuileDomino sont générées aléatoirement alors que les TuileCarcassonne sont définies précisément. Sac a une méthode abstraite qui sert à piocher des pièces, puisque les deux sacs de pièces ont cette fonctionnalité en commun.
- La classe Joueur est commune aux deux jeux, les joueurs ayant les mêmes attributs d'un jeu à l'autre. L'Intelligence artificielle est aussi considérée comme un joueur, un booléen sert à déterminer si le joueur en est une ou non.
- La classe Partie est également commune aux deux jeux. Elle possède comme attributs tout selon on a besoin pour définir une partie : une liste de joueurs, un plateau de jeu, un sac de pièces, un gagnant et un booléen qui détermine qui la partie est finie.
- La classe Plateau est elle aussi commune aux deux jeux, elle a comme attribut une Grille. Nous détaillerons ces classes dans la prochaine partie.

b. <u>Cas du plateau de jeu :</u>

Nous avons commencé l'implémentation de notre classe Plateau en lui donnant comme attribut un tableau en deux dimensions d'une taille arbitraire. Mais, une fois les méthodes finies, et pour ne pas nous bloquer de possibilités dès le départ, nous avons fait le choix d'implémenter un plateau "infini". En effet, on voulait ne jamais être bloqués dans l'espace, et donc pouvoir ajouter des pièces à l'infini vers tous les côtés. Pour celà, notre plateau a un champ Grille. La classe Grille a un attribut List<List<Tuile>> et comporte deux méthodes principales : un setter et un getter. Le constructeur de Grille initialise notre liste à une taille de 5x5. Le setter a trois rôles : ajouter une pièce à la Liste normalement si celle-ci est ajoutée à une position initialisée, gérer le cas où une pièce est ajoutée en dehors de la zone initialisée (au dessus ou à droite de la pièce initiale), et enfin gérer le cas où une pièce est ajoutée en "négatif" dans la liste (en dessous ou à gauche de la pièce initiale). Si la pièce est ajoutée en haut ou à droite en dehors de la zone initialisée, le setter va tout simplement

initialiser une nouvelle ligne ou colonne puis placer la pièce. Si la pièce est ajoutée en en bas ou à gauche, on est à une position négative dans notre liste : le setter va alors déplacer toutes nos pièces vers la droite ou vers le haut en ajoutant une ligne ou une colonne de pièces null. La pièce peut alors être placée. Concrètement : si l'utilisateur veut placer une pièce en (-1,0), le setter de Grille est appelé, il va décaler toutes les pièces d'un rang vers la droite et va placer la pièce en (0,0). Nous ne voulons pour autant pas que l'utilisateur se rende compte de ce décalage. Alors, à chaque fois que l'on décale notre tableau, un vecteur de déplacement est actualisé. Dans le getter, le programme se charge de prendre en compte le déplacement. Prenons notre pièce ayant été placée en (-1,0), si aucune autre pièce n'a été placée entre temps, nous avons dx = 1 et dy = 0, respectivement les vecteurs du déplacement en x et en y. Alors, en demandant au getter la pièce placée en (-1,0), il nous retourne la pièce placée en (-1 + dx, 0 + dy), donc la pièce placée en (0,0) : c'est bien la pièce que l'on voulait ! Si on demande au getter une pièce placée en dehors de la partie initialisée du tableau, il retourne null.

c. Les Tuiles

Avant de commencer à implémenter les tuiles, nous avons réfléchi à une manière de s'y prendre, et nous avons décidé de les implémenter de la manière suivante : Nous avons créé une classe abstraite Tuile, qui sert de base pour les deux types de tuiles. La seule différence entre les tuiles des deux jeux est leurs composantes. Les TuileDomino prennent en argument un tableau à deux dimensions de longueur 4x3. En effet, chaque TuileDomino a quatre côtés qui ont chacun trois chiffres. Ces chiffres sont compris entre 0 et 2. Nous avons fait ce choix pour que les chances de pouvoir placer une tuile soient assez élevés. Les TuileCarcassonne quant à elles prennent en argument un tableau de quatre paysages. Les paysages, représentés dans la classe Paysage, sont les différents éléments qui représentent une TuileCarcassonne. Nous avons donc créé une classe abstraite Paysage qui est héritée par les classes Pre, Route, et Village. Ceux-ci représentent tous les types de paysages possibles.

2. L'interface textuelle et ses contrôleurs

Dès lors que nous avons fini le modèle, nous avons commencé à implémenter l'interface textuelle. Elle a d'abord été dans une seule classe et une seule méthode dans Terminal.java. Puis, nous avons fait un travail de décomposition pour séparer cette longue méthode en plusieurs méthodes, chacune avec sa fonctionnalité précise. Cette interface textuelle a évolué au fur et à mesure que nous avancions dans l'année, nous n'avons par exemple pas directement géré les exceptions que le programme pouvait lancer car nous ne l'avions pas encore vu en cours. Nous avons ensuite créé les deux contrôleurs de l'interface textuelle : le contrôleur de l'intelligence artificielle et le contrôleur du joueur humain. Nous n'avions pas directement envisagé l'implémentation des contrôleurs et nous avons donc dû, de nouveau, décomposer le code. Le contrôleur a une référence vers la vue et la vue a une référence vers le contrôleur et vers le modèle. Le contrôleur a accès au modèle grâce à la vue. Le contrôleur se charge des liens entre le modèle et la vue. Concrètement, nous avons une classe ControleurTerminal qui est héritée par ControleurJoueurTerminal et par ControleurIATerminal. ControleurTerminal comprend les méthodes et champs communs entre les deux contrôleurs.

3. Interface graphique

Pour le cas de l'interface graphique, nous avons décidé de l'implémenter de la sorte : une fenêtre divisée en 3 parties (voir image ci-dessous) :

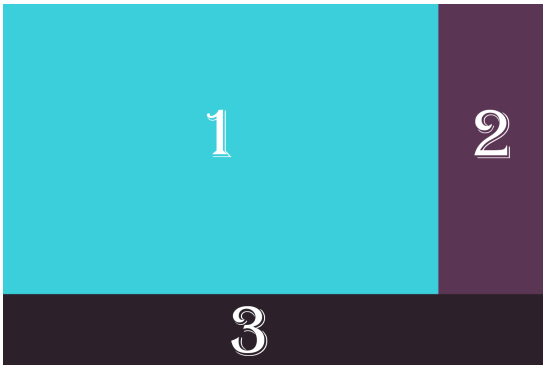

Fig 2. Plan de l'interface graphique

En 1, est affiché le plateau de jeu, en 2, les actions possibles, et en 3, la main actuelle.

Pour l'affichage de la fenêtre, nous avons utilisé un BorderLayout, avec le 1 au centre (CENTER), le 2 a droite (EAST) et le 3 en bas (SOUTH).

Pour l'affichage du plateau, le premier problème était de trouver comment placer des pièces, la manière la plus intuitive a été de penser à un GridLayout. Nous avons néanmoins rencontré un problème avec ce dernier : on ne décide pas lorsqu'on y ajoute un JPanel, de l'endroit ou on l'ajoute. Nous avons donc dû rechercher un autre moyen de le faire. En cherchant, on a appris l'existence du GridBagLayout, qui associé à un GridBagConstraint, nous permet d'être plus libre, et donc de pouvoir placer la pièce à l'endroit que l'on désire. Nous avons rencontré un nouveau problème : lorsque l'on dépassait le cadre (ce qui peut souvent arriver), les pièces s'empilaient les unes sur les autres. Pour remédier à cela, nous utilisons un JScrollPane, qui nous permet d'avoir un plateau assez grand pour pouvoir placer toutes nos pièces. Il nous permet donc de naviguer sur le plateau grâce aux barres de déplacement (verticale et horizontale). Nous avons en plus défini une limite : le joueur ne peut pas placer sa pièce en dehors du plateau. Celà n'est pour autant pas très probable d'arriver. L'une des améliorations que l'on aurait aimé implémenté est un plateau infini, qui pourrait s'agrandir autant qu'on le veut (cf. III.).

Pour l'affichage des actions actuelles nous avons décidé d'utiliser un GriLayout contenant 7 lignes et 1 colonne. Le GridLayout contient JPanel texte sur lequel est écrit le nom du joueur actuel, le nombre de pièces restantes, ainsi que l'état de l'action actuelle. Il contient aussi 4 autres boutons (Placer la tuile, Tourner la pièce, Passer son tour, Abandonner et un 5ème bouton pour Carcassonne : Placer un pion).

Et enfin, pour la main actuelle, nous avons utilisé un GridLayout pour le jeu de Carcassonne et un BorderLayout pour le jeu de Domino (a cause des contraintes d'affichage de la TuileDomino)

- Gestion graphique des tuiles

Les tuiles étant différentes pour les deux jeux du point de vue de l'affichage, nous avons donc procédé de manière différente pour les deux.

Pour les TuileDomino, nous avons créé une classe CarreChiffre qui étend JPanel. Elle crée un carré avec un chiffre au milieu. Nous avons aussi implémenté la classe TuileDominoGraphique, qui étend elle aussi JPanel, et qui a comme Layout un GridLayout et un attribut TuileDomino, avec des CarreChiffre aux endroits correspondants qui sont initialisées avec les chiffres correspondant à la TuileDomino.

Pour les TuileCarcassonne, nous avons créé une classe interne a la classe JeuCarcassonne qui s'appelle PieceCGraph, qui étend JPanel, elle contient un attribut BufferedImage et TuileCarcassonne, l'image s'initialise selon la TuileCarcassonne liée a la PieceCGraph, et va chercher dans le dossier resources le fichier .png associé.

4. Intelligences artificielles

Nos intelligences artificielles sont dans les contrôleurs de chaque affichage. L'intelligence artificielle de l'affichage textuel de domino est ControleurIATerminal, celle de l'affichage graphique de domino est ControleurIADomino et enfin celle de l'affichage graphique de Carcassonne est ControleurIACarcassonne. Nous avons commencé par implémenter l'intelligence artificielle pour la vue graphique du Domino. Lorsqu'elle est appelée, elle parcourt le plateau ligne par ligne. Pour chaque emplacement non vide sur la ligne, elle regarde chaque emplacement autour de la pièce. Si un emplacement est vide, elle va regarder si elle peut se placer dans son état actuel, puis en se tournant dans chaque position. Si elle peut se placer, elle se place directement. Les intelligences artificielles pour le jeu graphique de domino et pour Carcassonne ont le même principe. Elles ne sont pas les mêmes car les actions à effectuer sont différentes selon le jeu et l'affichage.

III. Pistes d'extension

1. Plateau

Avec un peu plus de temps nous aurions aimé pouvoir, dans la vue graphique, faire un plateau infini. Nous nous servons évidemment de la classe Grille dont nous avons parlé de l'implémentation précédemment. Mais, nous avons dû nous limiter à faire un plateau avec des limites dans l'interface graphique. Le plateau infini est cependant implémenté dans la vue textuelle.

2. Hexagonal Grids

Nous nous sommes renseignés sur l'implémentation des tuiles hexagonales. Il nous aurait fallu bien plus de temps pour pouvoir envisager leur implémentation, mais se renseigner sur les possibilités nous a permis d'avoir une idée du large champ des possibles. La première question que nous pouvons nous poser est la manière de représenter les tuiles hexagonales dans nos classes telles que nous les avons construites. Dans le modèle, rien ne change réellement. Nous définirons tout simplement nos tuiles avec 6 côtés au lieu de 4, mais les côtés seront toujours définis de la même manière. Concernant le placement, la correspondance entre deux côtés reste la même pour les deux jeux. La subtilité vient des coordonnées de placement réel dans le plateau.

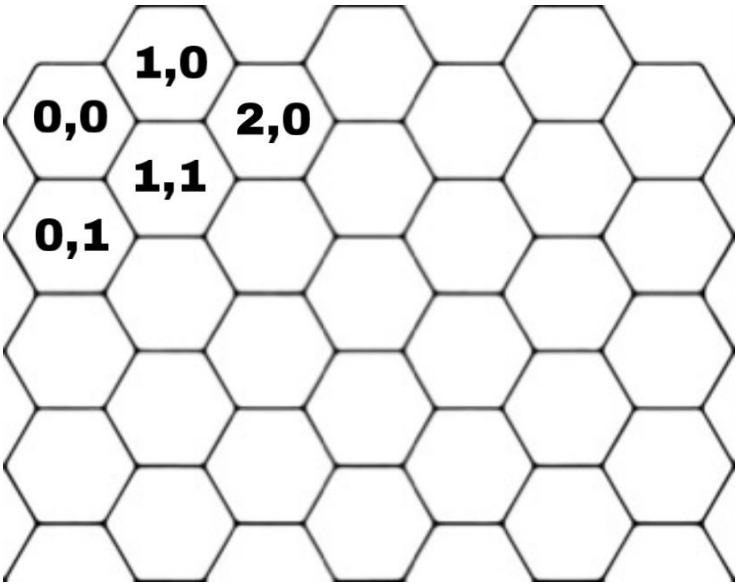

Fig 3. Exemple de plateau

La manière la plus simple que nous avons trouvé pour gérer les coordonnées de notre plateau serait de considérer comme une seule et même ligne les hexagones en quinconce tel que nous l'avons représenté sur cette image.

Ainsi, et étant donné que nous avons besoin de parcourir les contours d'une tuile dans plusieurs de nos méthodes, les pièces d'à côté seraient accessibles de la sorte :

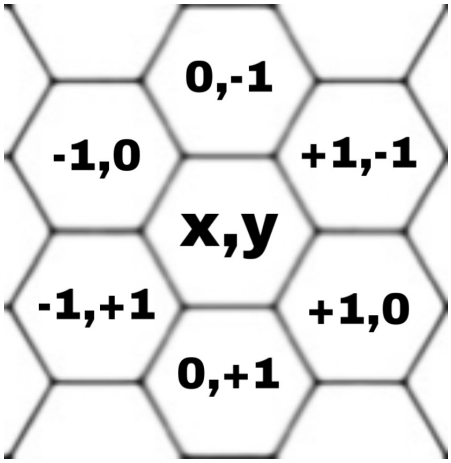

Fig 4. Contours d'une Hexagonal Grid

IV. Diagramme des classes

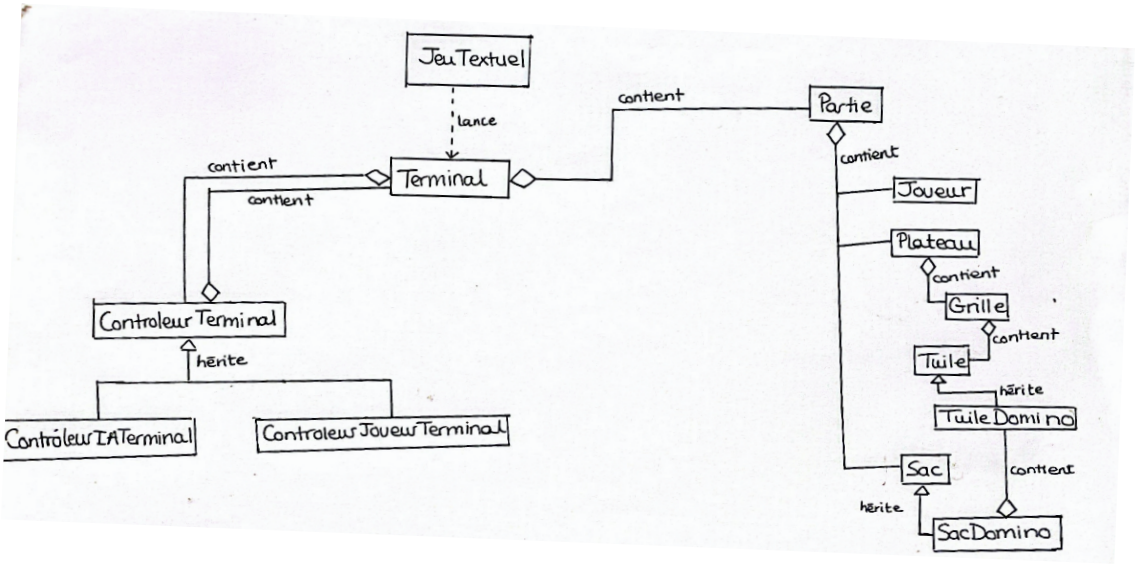

Fig 5. Diagramme de l'interface textuelle

(voir page suivante)

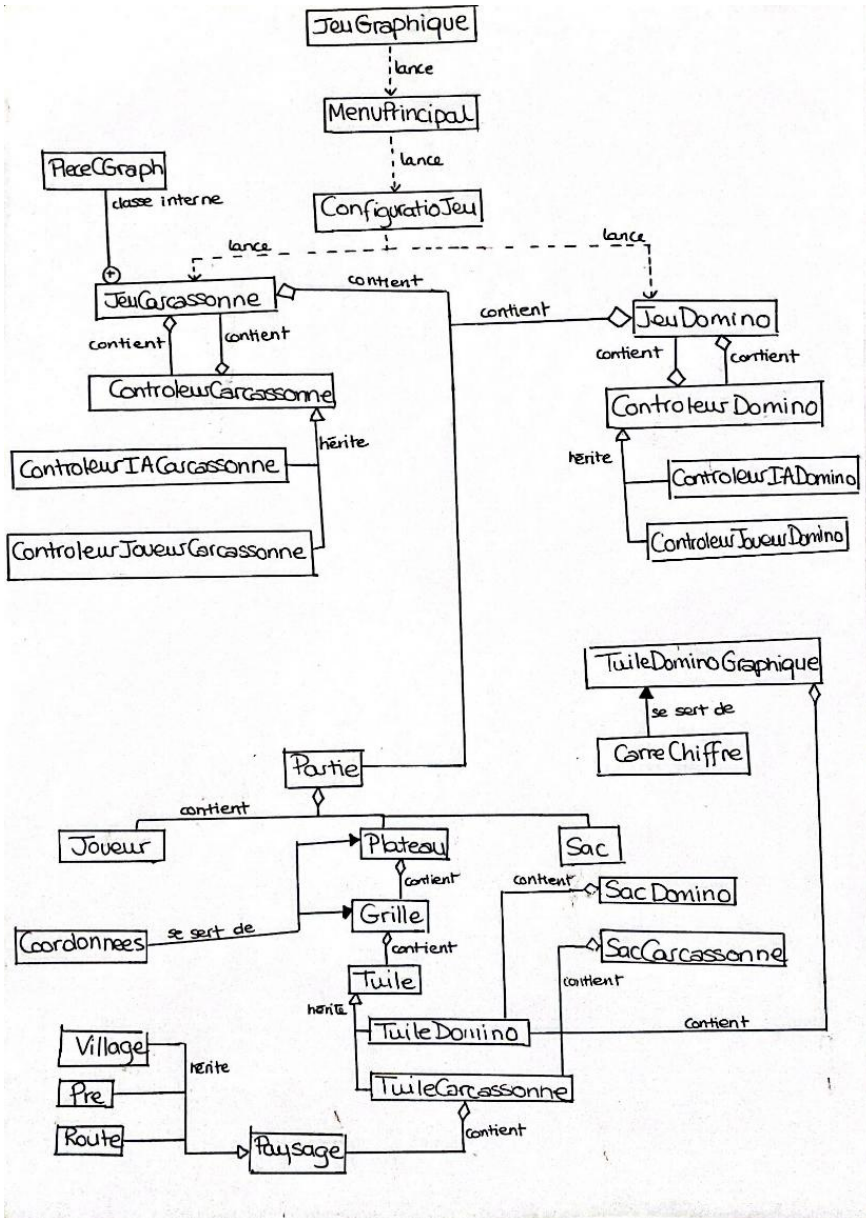

Fig 6. Diagramme de l'interface graphique