Projet de POO 2022

Marwan NZIZI, Ilian CRAGUE

January 6, 2023

1 Représentation du projet

1.1 Représentation des fichiers

Le code constituant notre projet est divisé entre 4 répertoires :

- common/: contenant tous les fichiers responsables des parties du modèle communes aux deux jeux
  - Cote.java: le côté d'une tuile
  - Joueur.java: un joueur avec sa main, son score, et si c'est une IA
  - Partie.java: un plateau, une liste de joueurs et les méthodes déroulant le jeu
  - Plateau.java: un tableau à deux dimensions de tuiles
  - Sac.java: une pile de tuiles qui agit comme une pioche
  - Tuile.java: un tableau de côtés
- carcarssonne/: contenant tous les fichiers responsables des parties du modèle spécifiques au Carcassonne
  - Parcelle.java: étend Tuile
  - Abbaye.java
  - Champs.java
  - Route.java
  - Ville.java
  - PartieDeCarcassonne.java: étend Partie

  - SacDeParcelle.java: étend Sac
  - Terrain.java: étend Cote

- domino/ : contenant tous les fichiers reponsables des parties du modèle spécifiques au Domino Carré
  - Domino.java: étend Tuile
  - DominoTextuel.java: contient toutes les méthodes nécéssaires à une partie console
  - PartieDeDomino.java: étend Partie
  - Rangee.java: étend Cote
  - SacDeDomino.java: étend Sac

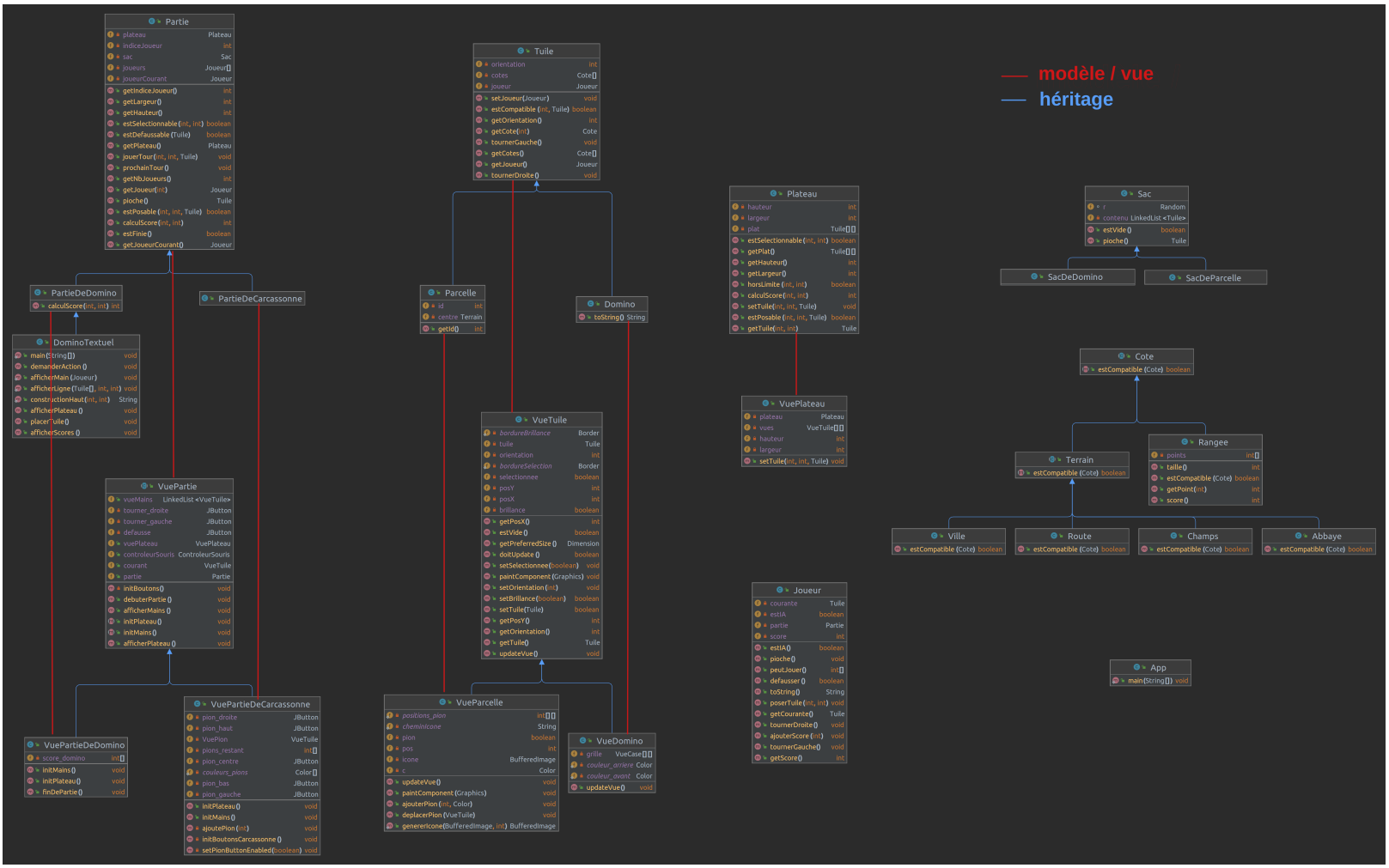
- ui/: contenant les fichiers responsables des vues ainsi que les icones pour le Carcassonne, calqués sur le modèle, ainsi que le fichier App, pour lancer les jeux graphiquement
  - App.java
  - icons/: les 24 tuiles de Carcassonne
  - vues/
    * VueDomino.java
    * VueParcelle.java
    * VuePartie.java
    * VuePartieDeCarcassonne.java
    * VuePartieDeDomino.java
    * VuePlateau.java
    * VueTuile.java

On remarque un découpage structuré. common/ contient et mutualise les fonctionnalités communes. carcassonne/ et domino/ proposent chacun une implémentation indépendante de manière complètement analogue: les deux jeux ont été pensés de la même manière et sont réalisés identiquement.
Ce découpage nous permet d'ordonner le code et nous a aidé à garder un travail propre tout au long de la période de développement. On peut y noter l'architecture Model-view, où la vue est uniquement chargée d'afficher le modèle, sans intéferer dans ses décisions.

1.2 Représentation graphique du projet

Dans la manière dont nous avons développé notre projet, les fichiers peuvent être liés par deux différentes relations : une relation d'héritage ou bien une relation modèle/vue.

Par exemple, nous voyons sur le graphique suivant que les classes Partie, Tuile, Sac et Cote, communes aux deux jeux, sont héritées chacune par une



classe implémentant les spécificités liées à chaque mode de jeu. Nous voyons aussi que les différents fichiers associés aux Parties, aux Tuiles ou au Plateau sont en lien avec un fichier responsable de la représentation de leur vue. Ce graphique montre de plus que la classe Joueur est commune aux deux jeux : du point de vue du joueur, jouer au Carcassonne ou au Domino revient donc à effectuer les mêmes actions.

2 Parties traitées

Nous avons traité tout le contenu du cahier des charges minimal :

1. Un environnement de jeu, réalisant l'accueil de l'utilisateur, et permettant de procéder au paramétrage du jeu :

   (a) choisir le jeu (dominos ou Carcassonne) ;
   (b) choisir le nombre de joueurs ;
   (c) choisir le nombre d'IA ;

2. L'implementation complète du Domino, géré graphiquement par awt et swing.

3. L'implementation partielle du Carcassonne : gestion de la pose de tuiles tournables, de la première jusqu'à ce que le sac soit vide, ainsi que de la pose de 8 pions par joueurs. géré graphiquement par awt et swing.

4. Une version texte du Domino, jouable dans le terminal.

3 Problèmes connus

3.1 modélisation

- identifier les parties communes et celles qui diffèrent entre les deux jeux, pour implémenter un maximum dans des classes communes aux deux jeux, dont hériteront des sous-classes responsables des spécificités de chaque jeu.

- choix du plateau fini/infini : plateau infini est fidèle au fonctionnement du jeu dans la vraie vie mais pose des soucis, notamment au niveau de l'affichage dans la console. Nous avons choisi de garder une implémentation uniforme peu importe la façon dont le jeu est affiche. Un plateau fini n'est pas parfaitement fidèle au jeu mais le nombre de pièces étant limité n'enlève pas grand chose au gameplay.

- choix du nom des variables et méthodes ainsi que de l'implémentation de certaines fonctionnalités. Sur un long développement (plusieurs mois) il est très difficile de rester constant dans son approche, d'où des implémentations et des noms pas toujours très cohérents entre eux.

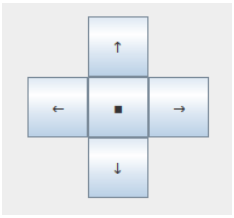
- l'abus de setBounds. L'interface principale du jeu n'utilise pas de layout. Nous voulions que les cases soient toujours de la même dimension pour rester adaptées à leur icône. Cela s'est fait au dépend d'une interface non responsive qui marche uniquement sur des écrans standards.

- la gestion des pions dans la vue. En effet, les pions rélèvent du fonctionnement du jeu, et donc du modèle, puisqu'ils servent au calcul des points et dépendent de certaines règles relatives à la partie. Pourtant, ils sont gérés dans la vue et sont complètement absents du modèle.

- pas d'utilisation de package. On se limite aux restrictions public / private.

3.2 implémentation

Lors de la phase de développement, la pose des pions a été un soucis : il a d'abord fallu choisir comment représenter ces derniers, ainsi que du système à adopter pour permettre à l'utilisateur de poser un pion sur la dernière tuile posée. Nos IA posant leur tuile dès que le joueur a posé la sienne, on aurait pu faire en sorte que cette action soit suivie de l'apparition d'une fenetre pop-up nous demandant si l'on souhaitait poser un pion.



En premier lieu nous avons choisi de placer le pion avant la pose de la tuile : la tuile apparaissant dans la main, il devenait possible d'y apposer un pion. Cette méthode nous a posé de nombreux soucis, premièrement car le pion était simplement un rond présent au même endroit que la tuile, il fallait donc le déplacer quand on plaçait la tuile ou qu'on la tournait, le supprimer quand la tuile était défaussée. Nous en avons finalement décidé autrement : nous avons implémenté un pavé directionnel, utilisable dès qu'un joueur a posé une tuile, jusqu'à qu'un autre joueur en ait posé une. Il permet de poser un pion sur la dernière tuile placée par un joueur, à l'endroit correspondant. Une fois la méthode choisie, il a été compliqué de coordonner la vue et le modèle.

Une partie de notre réflexion s'est aussi portée sur "comment faire fonctionner les jeux avec leurs propres règles en conservant le maximum de parties communes ?". Une première approche a été de travailler dans VuePartie avec un booléen "estCarcassonne" qui indique le jeu actuel. Cette approche, à peine pire qu'utiliser instanceof sur la partie actuelle, a vite été abandonnée au profit d'un strategy desgin pattern. VuePartie et son contrôleur ont alors été séparés en classes selon le jeu qu'elles implémentent. Ce refactor a été profond et couteux, mais à grandement améliorer la clarté du code et son extensionnabilité.

4 Pistes d'amélioration

4.1 implémenter les fonctionnalité avancées proposées

- terminer à 100% le Carcassonne, en mettant en place les contraintes de placement des pions, ainsi que le calcul final du score. Il faudrait ajouter les pions au modèle, puis parcourir les tuiles récursivement pour calculer le score.
- mettre en place la possibilité de sauvergarder une partie, en implémentant l'interface Serializable
- améliorer les IA, qui pour l'instant placent leur tuile au premier endroit où elles le peuvent. Elles pouraient parcourir le plateau de différentes directions, pour mieux répartir les tuiles. Aussi, et surtout, elles pour-

raient prendre en compte le score, voire même les tuiles déjà piochées et défaussées pour optimiser le placement.

- changer les tuiles carrées par des tuiles hexagonales. Les tuiles sont assez générales, elles peuvent avoir autant de côtés qu'on veut. Pour autant changer la forme des tuiles demanderait de changer le plateau. Il faudrait revoir la façon dont on choisit les voisins (pour déterminer la compatibilité) et toutes les méthodes relatives à l'affichage.

4.2 implémenter des fonctionnalités avancées auxquelles on a pensé

- rendre le tout plus joli, les menus ainsi que les jeux, par exemple en mettant des textures adaptées à chaque jeu : un fond d'écran, des skins pour les boutons.
- faire plusieurs niveaux de difficultés pour les IA, selon les principes évoqués ci-dessus, sélectionnables au début de la partie par le joueur.
- faire une version en ligne, pour jouer avec des amis à distance.