Université de Paris

L2 Informatique

Projet de POOIG

Année 2022/2023

Réalisé par ARROUS Thomas (22114626) et SOAN Tony Ly (22107803)

Plan

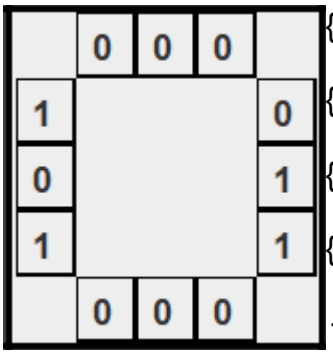
Le projet que nous avons réalisé consiste en la création de deux jeux de société, domino et Carcassonne, en Java. Nous avons suivi les consignes du cahier des charges, qui nous ont demandé d'implémenter les règles du jeu du domino, ainsi qu'une partie des règles du jeu de Carcassonne, de la pose de la première tuile jusqu'à ce que le sac soit vide, en veillant à respecter les règles de chaque jeu, avec deux exceptions majeures pour Carcassonne : la vérification des contraintes de placement des pions (partisans) et le calcul des points. Nous avons réalisé ces jeux de manière progressive et en suivant le motif d'architecture logicielle "Modèle-vue-contrôleur" (MVC). Pour commencer, nous avons réalisé un mode de jeu "texte" pour les dominos, qui permet de jouer entièrement dans le terminal. Ensuite, un mode de jeu "graphique" pour les dominos et pour Carcassonne utilisant AWT et Swing. Nous avons également ajouté la possibilité de jouer contre un robot. Nous présenterons d'abord les choix que nous avons effectués pour la modélisation des différents éléments qui constituent les jeux. Nous détaillerons comment nous avons organisé notre travail pour y parvenir. A l'aide du principe de la séparation MVC, et des différents concepts mis à notre disposition par Java. Enfin, nous expliquerons les fonctionnalités du jeu que nous avons réalisées. Nous terminerons par évoquer les perspectives d'évolution de notre projet et les enseignements que nous avons tirés de cette expérience.

I) Modélisation

a) Choix de représentation des tuiles

Pour commencer, nous avons dû réfléchir à la meilleure façon de représenter les tuiles dans notre programme. Nous avons choisi d'utiliser la classe "PieceModel" , qui contient un tableau de dimension 2 de valeurs dont le type n'est pas statique. Les valeurs inscrites sur la pièce sont représentées avec une liste d'entier pour domino et pour Carcassonne par une liste de "Terrain" qui indique quel type de terrain est présent sur chaque espace de la tuile. "Terrain" est une énumération permettant de définir le type de terrains est présent sur une pièce de Carcassonne. Il existe donc plusieurs types de terrain : les villes, les villes avec des boucliers, les prés, les chemins et les carrefours. Il peut arriver qu'aucun élément n'est indiqué sur un emplacement qui compose une pièce. Par exemple pour Domino, les angles et le centre des pièces n'ont aucune valeur indiquée. Dans ces cas-là on utilisera -1 comme valeur pour les domino et NONE pour les pièces de Carcassonne.

Exemples de représentation :



{{-1,0,0,0,-1}, | {{NONE, PRE, PRE, PRE,NONE},
{1,-1,-1,-1,0}, | {PRE, PRE, PRE, PRE,PRE},
{0,-1,-1,-1,1}, | {CHEMIN,CHEMIN,CHEMIN,PRE,PRE},
{1,-1,-1,-1,1}, | {PRE, PRE, CHEMIN,PRE,PRE},
{-1,0,0,0,-1}} | {NONE, PRE, CHEMIN,PRE,NONE}}

(Pour faire moins lourd, on a réduit la syntaxe, la vraie syntaxe étant : Terrain.(type de terrain), exemple : Terrain.PRE, qui sont les valeurs énuméré par "Terrain")

Le NONE présent dans les angles des tuiles de Carcassonne, sont là pour éviter des situations ambiguës. Il se peut que dans le coin d'une tuile, prenons par exemple le coin supérieur droit de l'image de la pièce, soit divisé en deux par une ville et un pré. Le triangle supérieur gauche du coin est une ville et le coin inférieur droit est un pré. Il y a donc ambiguïté. Grâce au NONE, il n'y a plus de situation ambiguë, les coins ne sont plus partagés pour 2 côtés, car lorsqu'il y a un terrain, il prend l'intégralité d'un côté de la pièce. Sauf pour les routes qui sont centrées. C'est pour cela qu'on a choisi d'utiliser des tuiles de 5 par 5 pour Carcassonne.

Voici un exemple :

Zone ambiguë :
le haut est une ville la droite est un pré.

Cela nous a permis de facilement implémenter les règles de pose des tuiles, en vérifiant que chaque élément de la nouvelle tuile correspond à un élément adjacent présent sur les tuiles adjacentes.

b) Génération des tuiles

Afin de générer une tuile d'un jeu spécifique, nous avons étendu notre classe "PieceModel". Nous avons maintenant "DominoPieceModel" et "CarcassonnePieceModel". Nous avons également pour le jeu domino développé une fonction de génération de tuiles aléatoires (qui ne peuvent être composé que de 0 ou de 1) lors de l'appel du constructeur de "PieceDominoModel". Nous avons choisi de créer un certain nombre de tuiles, qui s'élèvent à 10 tuiles supplémentaires par joueur. Pour le jeu Carcassonne, nous avons récupéré toutes les pièces qui existent dans le jeu. Et nous avons modélisé les 72 pièces qui composent le jeu au format static dans la classe "CarcassonnePieceModel". Afin de pouvoir facilement remplir notre sac de tuiles au début de chaque partie. Cela nous a permis de garantir une certaine variété dans les tuiles disponibles tout en facilitant l'implémentation du jeu, et en augmentant les probabilités de compatibilité entre les tuiles.

Pour générer une tuile dans Domino :

```java
/**
 * Initialise une piece de taille hauteur par largeur.
 * Pour chaque valeur qui est sur un bords, mais qui n'est pas un coin
 * initialise une valeurs.
 *
 * @param hauteur Hauteur de la piece
 * @param largeur Largeur de la piece
 */
DominoPieceModel(int hauteur, int largeur) {
    super(hauteur, largeur, -1);
    for (int i = 0; i < hauteur; i++) {
        for (int j = 0; j < largeur; j++) {
            // si c'est une valeur sur un bords de la piece, mais ce n'est pas un coins
            if (((i == 0 || i == hauteur - 1) && j != 0 && j != largeur - 1)
                    || ((j == 0 || j == largeur - 1)) && i != 0 && i != hauteur - 1) {
                // initialise la valeur
                valeurs.get(i).set(j, new Random().nextInt(bound: 2));
            }
        }
    }
}
```

Parcours l'ensemble des cases qui compose une tuile.

Si c'est une case orange alors on initialise une valeur aléatoire : 0 ou 1

	0	1	1	
1				1
0				1
0				1
	0	0	0	

Des variables static qui représente les pièces de carcassonne sont stocké dans “CarcassonnePieceModel”, voici un exemple pour la première pièce :

```
public static Terrain[][] un =
        /* */ { { Terrain.NONE, Terrain.PRE, Terrain.PRE, Terrain.PRE, Terrain.NONE },
                { Terrain.PRE, Terrain.PRE, Terrain.PRE, Terrain.PRE, Terrain.PRE },
                { Terrain.CHEMIN, Terrain.CHEMIN, Terrain.CHEMIN, Terrain.PRE, Terrain.PRE },
                { Terrain.PRE, Terrain.PRE, Terrain.CHEMIN, Terrain.PRE, Terrain.PRE },
                { Terrain.NONE, Terrain.PRE, Terrain.CHEMIN, Terrain.PRE, Terrain.NONE } };
```

c) Choix de représentation du plateau.

Pour représenter le plateau de jeu, nous avons choisi de créer un tableau de dimension 2 d'objets "PieceControleur" qui contient des éléments non static. Cela nous a permis de pouvoir y placer aussi bien des pièces de Carcassonne que des pièces de Domino. De plus, le modèle étend la classe “Extendable”, qui permet de faire en sorte qu’un tableau soit en étendable à l’infini. Si une pièce est ajoutée sur un bord du tableau actuel, alors le tableau sera agrandi de 1 ligne ou 1 colonne en direction du bord en question. Cela me permet au début de la partie d’avoir un tableau de 1 par 1. Qui sera étendu au fur et à mesure que des pièces sont ajoutées. De plus, cette représentation nous a permis de facilement vérifier si une tuile peut être placée à un emplacement donné en vérifiant simplement si toutes les tuiles adjacentes sont compatibles. De plus, lorsque le joueur se déplace, il se centre sur une pièce du plateau. Ainsi le plateau contient un argument “point” qui définit la position actuelle sur le plateau, c'est-à- dire qui permet de savoir à quelle colonne et à quel ligne est ce que nous sommes centrés sur le plateau.

Comme exemple dans la classe Extendable, on a une méthode qui ajoute une colonne à gauche du plateau.

```
private void ajouterUnCoteGauche() {
    for (ArrayList<P> ligne : tableau) {
        // ajouter un element a la fin de chaque ligne
        ligne.add(e: null);
        for (int i = getLargeur() - 1; i > 0; i--) {
            // deplacer tout les element vers la droite
            ligne.set(i, ligne.get(i - 1));
        }
        // supprimer le premiere element
        ligne.set(index: 0, element: null);
    }
}
```

Pour chaque ligne du plateau

Ajoute un élément tout à droite de à ligne.

Déplace tous les éléments de la ligne de 1 sur la droite. (écrase l’élément ajouté)

Supprime le premier élément

d) Les autres objets.

Dans notre modélisation du jeu de dominos, nous avons représenté un joueur avec une classe "Player" qui prend en compte plusieurs attributs tels que le nom du joueur, la pièce qu'il a en main et son score. Le score du joueur est un entier qui augmente chaque fois qu'on applique la méthode addScore. Nous avons également modélisé une classe "Sac" qui gère l'ensemble des tuiles disponibles pour les joueurs. Cette classe fournit une tuile au hasard à chaque joueur qui en fait la demande, et retourne “null” si le sac est vide et qu'un joueur essaie de piocher une tuile, ce qui signale la fin de la partie. Pour représenter les coordonnées du plateau, nous avons

créé une classe "Point" qui possède les attributs x pour les colonnes et y pour les lignes. De plus, afin de comprendre de quel côté du plateau et ou de la pièce on parle, on a mis en place une énumération Direction avec les valeurs “HAUT”, “BAS”, “GAUCHE”, “DROITE”, et “ACTUEL”. On a aussi ajouté une interface nommée “Demander” qui permet de demander un boolean et une direction dans le terminal, ainsi que des exceptions "DirectionInvalide" et "PositionInvalide" qui sont levées respectivement lorsque la direction entrée ne correspond pas à une réponse attendue ou lorsqu'une action est tentée sur une position qui n'est pas dans le tableau. Avec cette modélisation, nous avons mis en place une boucle de jeu qui permet à chaque joueur de jouer à tour de rôle jusqu'à la fin de la partie.

Nous avons également créé un diagramme des classes pour mieux visualiser comment chaque objet était lié aux autres dans notre programme (page 12).

II) Développement progressif et séparation modèle vue contrôleur

a) Importance de commencer par une version jouable minimal

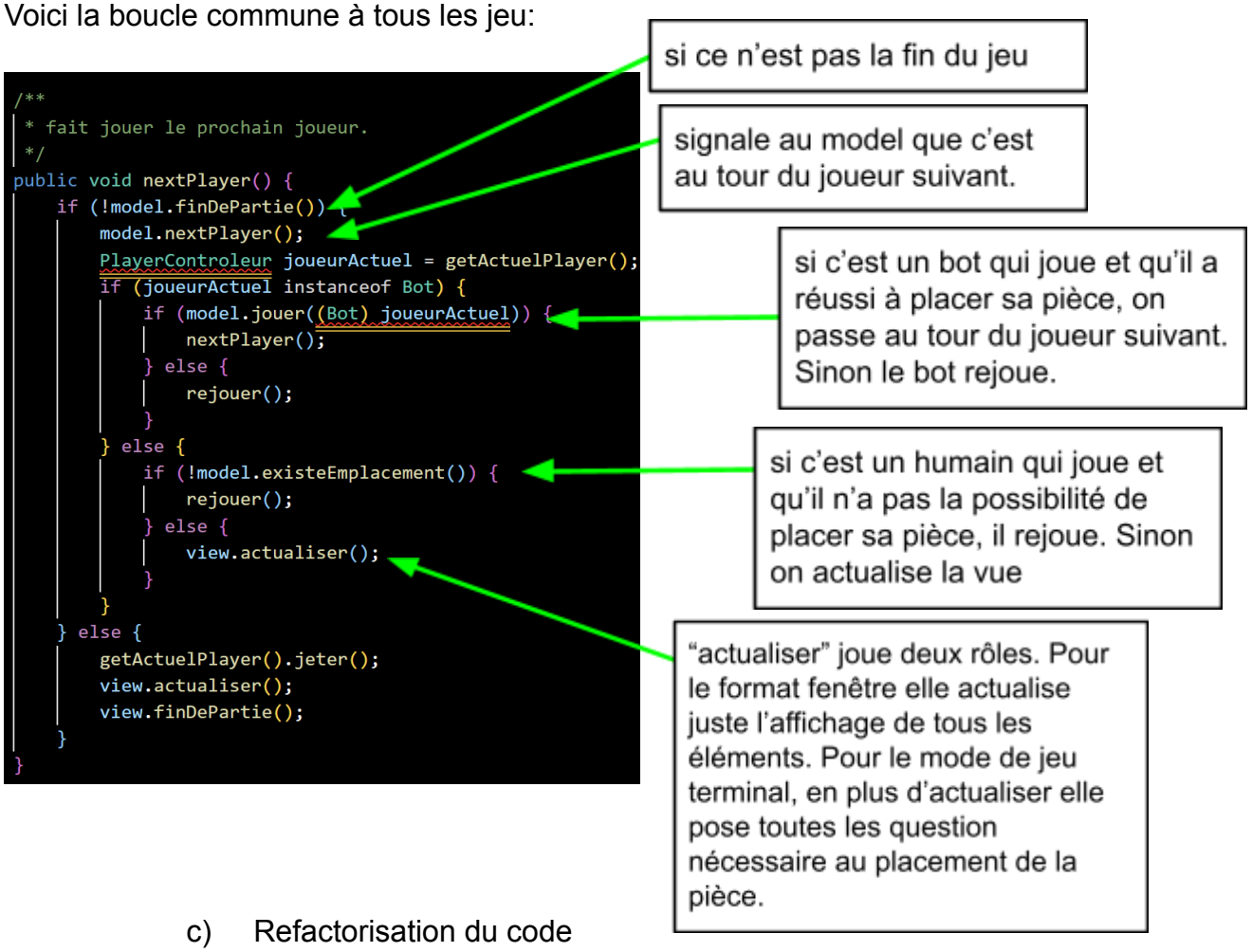
Pour développer notre projet de jeux de dominos et Carcassonne, nous avons adopté une approche de développement progressif. Nous avons commencé par créer une version minimale du jeu qui soit jouable, afin de pouvoir tester et corriger les erreurs au fur et à mesure de notre avancée. Dans un premier temps, nous avons défini les interfaces de nos classes principales. Ensuite, nous avons modélisé le jeu de dominos en généralisant au maximum les classes, comme les classes “Plateau”, "Piece", "Sac" et "Player" . Cette étape nous a permis de disposer d'une base solide pour développer le jeu de Carcassonne. Nous avons donc terminé par généraliser quelques classes restantes et finaliser les deux jeux.

b) Avantage séparation model vue contrôleur (MVC)

La séparation de la vue et du modèle a également été cruciale dans notre développement. En suivant le modèle MVC , nous avons pu séparer les aspects liés au rendu graphique de la logique du jeu, ce qui a rendu notre code plus structuré et plus facile à maintenir. En utilisant le modèle MVC, nous avons pu développer notre projet de manière modulaire, en séparant les différentes parties de notre programme en trois parties distinctes : la vue, le modèle et le contrôleur. Cela nous a permis de développer chaque partie de manière indépendante, ce qui a grandement simplifié notre processus de développement. Cela nous a permis de développer de nouvelles fonctionnalités sans avoir à toucher au code existant, ce qui a grandement simplifié notre processus de développement. Cela a également permis de réduire le risque d'erreurs et de bugs. De plus, en utilisant le modèle MVC, nous avons pu plus facilement ajouter de nouvelles fonctionnalités et évolutions au jeu, en modifiant uniquement certaines parties du code sans avoir à toucher à d'autres. Cela nous a également permis de rendre notre programme plus modulaire et réutilisable, en pouvant facilement réutiliser certaines parties du code dans les deux jeux. Enfin, le fait de séparer la vue du modèle nous a également permis d’avoir plusieurs vues différentes pour le même élément, ce qui nous a permis de proposer une expérience de jeu différente selon les préférences de l'utilisateur comme pour domino par exemple. Il peut se jouer dans le terminal, ou dans une fenêtre. Ainsi “PlayGameView”, permet de jouer au domino dans la fenêtre. Alors que “PlayGameTerminalView” le permet dans le terminal. On a aussi pu faire un Model et un

contrôleur pour Carcassonne, mais pas de Vue car on utilise “PlayGameView”. Pour les 3 modes de jeu on peut donc utiliser la même boucle de jeu.

Voici la boucle commune à tous les jeu:



c) Refactorisation du code

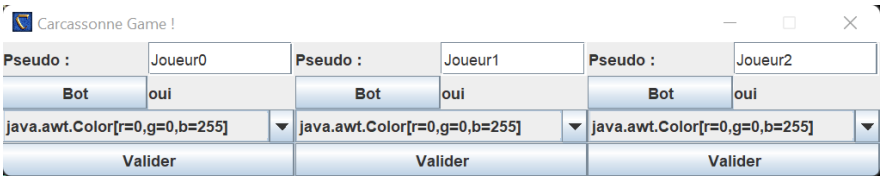
La refactorisation a été un élément clé de notre développement car elle nous a permis de nous assurer que notre code respecte les bonnes pratiques de programmation. En effet, en utilisant des noms de variables et de méthodes explicites, en organisant notre code de manière modulaire et en respectant les conventions de style, nous avons rendu notre code plus compréhensible pour nous-mêmes et pour d'autres développeurs s’ils venaient à travailler sur le projet. De plus, en procédant régulièrement à des re-factorisations, nous avons été en mesure d'identifier et de corriger les éventuelles erreurs de notre code de manière efficace. Cela a également contribué à rendre notre code plus robuste et moins sujet aux bugs. En sommes, la refactorisation a été un élément crucial de notre développement car elle nous a permis de créer un code de qualité et lisible. Même si cela a été difficile.

d) Ajout d’un menu

Le menu de notre programme permet aux utilisateurs de choisir entre les deux jeux, Carcassonne et Domino, et de définir le nombre de joueurs pour chaque partie, ainsi que d’initialiser la liste de joueurs. Le menu est organisé en utilisant le modèle MVC, avec une classe de contrôleur pour gérer les interactions avec l'utilisateur, une classe de modèle pour stocker les données et une classe de vue pour afficher le menu et les informations de la partie

en cours. Le menu est également capable de lancer des parties en mode terminal pour ceux qui préfèrent jouer sans interface graphique.

Ici, on peut choisir le nombre de joueurs ainsi que le mode de jeu

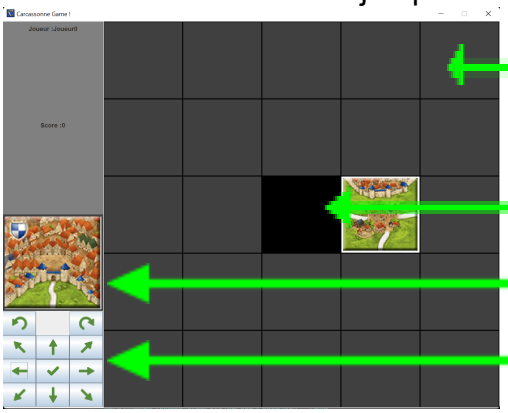


Ici on ajoute les joueurs un par un.

III) Fonctionnalités de bases

a) Implémentation d'une partie

Afin de gérer le déroulement d'une partie, on a écrit une classe qui se nomme "PlayGameControleur". En fonction du jeu auquel on joue, on utilise l'un des deux fils "PlayDominoControleur", ou "PlayCarcassonneControleur". Elles gèrent une partie du début à la fin. Voilà comment se déroule la partie. Un plateau de 1 par 1 est créé, puis un sac, puis les pièces sont ajoutées au sac en fonction du jeu. On initialise une liste de joueurs. Et on met la variable indice a 0. Elle indique c'est quel joueur de la liste qui joue son tour. Enfin on démarre le jeu. On place la première pièce du sac au milieu du plateau. Et on peut faire jouer le premier joueur. C'est le tour du joueur 1 qui s'exécute, on attend qui effectue ses actions, une fois qu'il a placé sa pièce on applique la méthode nextPlayer qui passe au joueur suivant. Puis on recommence un tour. Et ce jusqu'à ce qu'il n'y ai plus de pièce dans le sac.



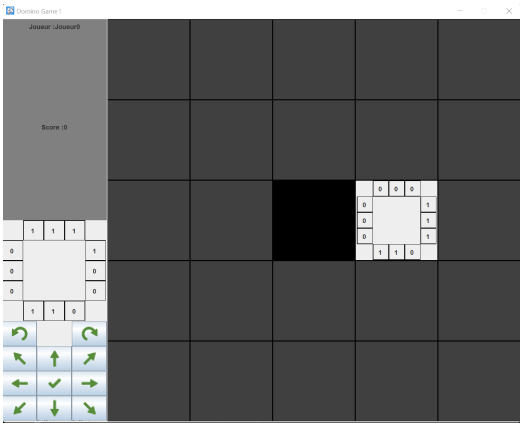
b) Mode de jeu "texte" pour le domino

Pour la fonctionnalité du mode de jeu "texte" pour les dominos, on a décidé de représenter le plateau comme tel, on affiche le tableau de 5 par 5 centré sur la case où se situe le joueur. On représente la case au centre par des "+" si il n'y a pas de pièce, sinon on affiche la pièce:

Tout ceci à été rendu possible grâce à la méthode “afficher” de la classe PlayGameView. Le mode de jeu "texte" a également été utile pour déboguer et tester rapidement notre implémentation du jeu de dominos, en affichant clairement les tuiles jouées et les actions effectuées par chaque joueur. Nous avons également créé une classe "PlayDominoControleur" qui étend la classe “PlayGameControleur” qui gère le déroulement de la partie, en demandant successivement aux joueurs les actions qu’ils souhaitent effectuer tout en vérifiant que leurs actions sont valides selon les règles du jeu.

c) Implémentation jeu Domino

Pour lancer une partie de Domino sur l’interface graphique nous n’avons pas eu à faire beaucoup de changement. Il suffit de réadapter la vue du jeu. Ainsi que de réadapter la manière de demander les actions.



d) Implémentation jeu Carcassonne

Pour l'implémentation partielle du jeu de Carcassonne, nous avons choisi de ne pas implémenter les contraintes de placement des pions et le calcul du score, afin de nous concentrer sur la pose des tuiles. Nous avons toutefois pris en compte les différents éléments du paysage présents sur les tuiles, comme les routes, les villes, les villes avec des boucliers et les champs. Nous avons décidé de représenter chaque tuile sous forme d'un objet contenant les informations sur chacun de ces éléments. Ainsi que la possibilité de placer des partisans sur les pièces. Il a alors fallu adapter le modèle d’une pièce pour pouvoir y ajouter un tableau de dimension 2 de partisan et ainsi pouvoir afficher où se trouve le partisan sur la pièce, s' il y en a un. De plus, maintenant il ne suffit pas de comparer simplement les valeurs présentes sur les pièces pour savoir si elles sont adjacentes. En effet, “Terrain.QUARTIER”, et “Terrain.QUARTIERBOUCLIER” peuvent être placées côte-à-côte. Pour cela on a associé des valeurs numériques au différents “Terrain”. On a donné la même valeur pour “QUARTIER” et “QUARTIERBOUCLIER”. Il nous suffit donc de redéfinir la méthode compare qui compare 2 côtés pour voir s' ils sont identiques. Ainsi on pourra vérifier lors du placement des pièces si les valeurs de chaque côté adjacent des pièces Terrain ont la même valeur 2 à 2, on dit alors que les côtés adjacents sont compatibles.

```java
public enum Terrain {
    ABBAYE(i: 0),
    QUARTIER(i: 1),
    QUARTIERBOUCLIER(i: 1),
    CARREFOUR(i: 2),
    CHEMIN(i: 3),
    PRE(i: 4),
    NONE(i: 5);
```

Comme QUARTIER et QUARTIERBOUCLIER ont la même valeur ils sont compatibles. Le reste ne l'est pas.

e) Jouer contre un robot

Pour représenter le robot, on utilise un interface que l'on a nommé Bot. Pour la fonctionnalité de jeu contre un robot, nous avons développé une stratégie très peu optimale. Le robot parcourt le plateau de gauche à droite, et de haut en bas, et place sa tuile dès qu'il trouve une position valide. C'est-à-dire un emplacement qui respecte les règles du jeu. Nous avons également implémenté la possibilité pour le robot de poser des partisans dans Carcassonne. Cependant, la stratégie n'est pas efficace. Il a une chance sur deux de poser un partisan sur une tuile qu'il vient de poser. Et il le place aléatoirement sur cette tuile. La méthode "jouer" de l'interface Bot permet au bot de jouer.

```java
public boolean jouer(PlateauControleur<Terrain> plateauControleur, CarcassonneBotControleur botActuelControleur)
    Point position = plateauControleur.getEmplacementPossible(getMain());
    if (position != null) {
        plateauControleur.setPiece(botActuelControleur, position);
        Random rd = new Random();
        // Veut-il placer un partisant
        if (!partisantsIsEmpty() && rd.nextInt(bound: 2) == 0) { // si 0, il place un partissant aléatoirement s
                                                                 // pièce.
            ((CarcassonnePieceControleur) getMain()).placerPartisant(
                    new Point(rd.nextInt(getMain().getLargeur()), rd.nextInt(getMain().getHauteur())),
                    botActuelControleur);
        } // si 1 il ne place pas de partisant
        return true;
    } else {
        return false;
    }
}
```

Cette méthode vérifie ainsi s' il existe une position sur laquelle on peut placer la pièce du bot. Puis, elle place la pièce en question à sa position si c'est possible. Puis il détermine aléatoirement (1 chance sur 2) s'il place un partisan. Enfin il place aléatoirement ce partisan sur la pièce. Le booléen qui est retourné indique si le bot a réussi à jouer.

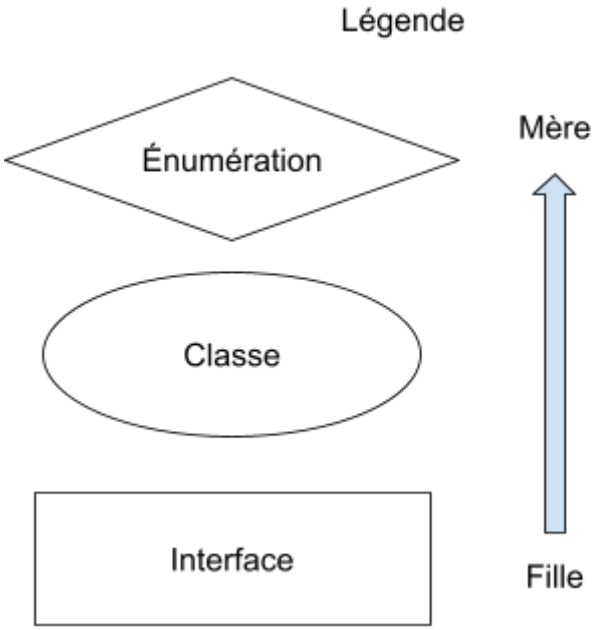
Pour conclure, pendant le développement de notre projet, nous avons rencontré plusieurs difficultés, principalement liées à une mauvaise organisation au départ. En effet, nous avons eu du mal à nous répartir le travail de manière équitable et cela a entraîné des retards dans la réalisation de certaines parties.

Nous avons également identifié plusieurs points à améliorer dans notre code actuel. Tout d'abord, nous avons manqué de temps pour implémenter les fonctionnalités avancées comme la gestion des contraintes de placement des pions et le calcul du score pour le jeu de Carcassonne. Nous avons également remarqué que notre génération de tuiles aléatoires n'était pas optimale et pourrait être améliorée.

Au cours de ce projet, nous avons acquis de nombreuses expériences en termes de développement de jeux et de gestion de projet. Nous avons appris à mieux nous organiser et à travailler en équipe de manière efficace.

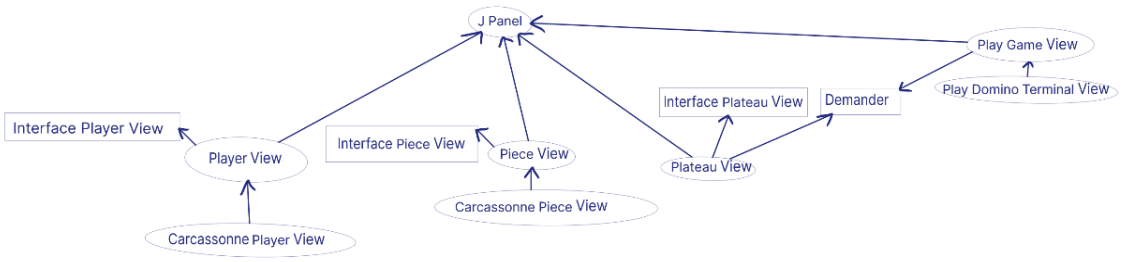
Enfin, nous sommes satisfaits du résultat final de notre projet, même s'il reste encore des points à améliorer.

Diagramme des classes

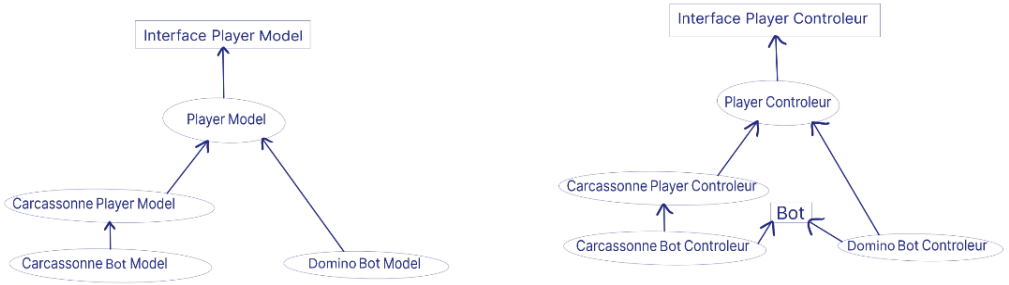


(Tournez la page SVP)

Views :



Player :

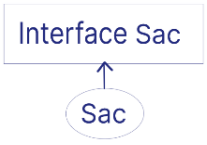


Piece :

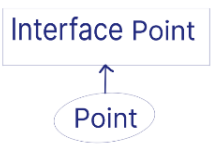
Plateau :

Play Game :

Sac :

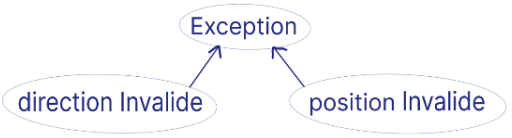


Point :



Partisan :

Exceptions :



Enumération :

Menu :