Friedmann Eliot

Laïdouni Mohamed

2022-2023

Rapport du projet de POO

Dans le cadre du 1er semestre de la deuxième année de licence d'informatique, il est demandé de faire un projet dans la l'UE programmation orientée objet. Ce projet consiste en la réalisation de deux jeux : un domino et un jeu de Carcassonne. Ceux-ci ayant une structure propice à utiliser l'ensemble du programme de la matière, à savoir, l'utilisation de classes, de l'héritage ou encore des interfaces graphiques.

I. Dominos

La création du domino se divise en trois étapes majeures :

1. Conceptualisation

Dans la conception du projet cette étape a été essentielle, elle consiste en la création conceptuelle du domino et des différentes classes qui le compose. La mise sur papier de ces idées permet de faire apparaitre une première vision du code et d'organiser le travail. Ce travail de conceptualisation est aussi utile dans la création du Carcassonne étant donné les similitudes des deux jeux notamment concernant les tuiles.

On peut voir que l'organisation du code du dominos est déjà plus ou moins complète, il « suffit » maintenant de l'écrire.

2. Ecriture du code

Ayant maintenant une assez bonne idée de la structure du code, l'écriture de celui-ci est divisée selon trois classes :

a. Trio

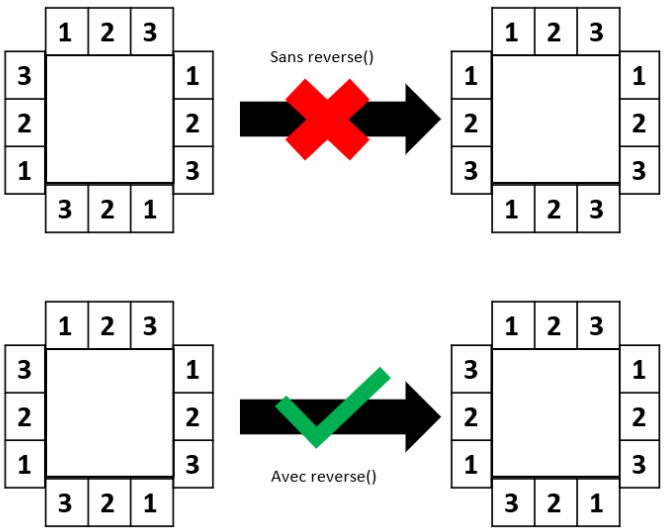
Le premier problème dans la création du domino a été de choisir si oui ou non une classe auxiliaire était utile pour définir les nombres contenus dans les tuiles du domino. On utilise la classe Trio pour définir les côtés des tuiles du domino. Elles possèdent une fonction **booelan estEgal()** qui renvoie true si deux Trio sont égaux, la méthode reverse qui échange la 1ère et la dernière case du Trio et la fonction **int sum()** qui renvoie la somme des trois nombres du Trio. Cette dernière méthode sera utilisée pour tourner les tuiles. L'implémentation de cette classe permet de ne pas avoir à manipuler les tableaux à chaque fois, elle a toutefois pour conséquence de charger un peu le code.

b. TuileDominos

Cette classe hérite de Tuile.java qui est une classe vide qui permet l'utilisation de certaines classes pour les deux jeux.

Elle possède 4 attributs Trio : haut, bas, gauche et droite. Ceux-ci représentent les quatre faces du domino. Les constructeurs sont assez simples et il y a une méthode **String toString()** qui permet de jouer dans le terminal. La méthode **void tourneTuile()** permet de tourner le domino dans le sens des aiguilles d'une montre, c'est ici que la méthode reverse de Trio est très utile, pour que la Tuile tourne

proprement il faut que les trios se comportent comme dans le schéma ci-dessous :



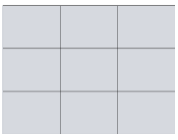
La fonction **boolean estCompatible()** est liée à la fonction **Tuile[] tuileAdjacentes()** du plateau. Celle-ci crée un tableau de tuiles qui contient les tuiles adjacentes et chacune d'entre elle a un index attribué. Dans **boolean estCompatible(),** les index servent à savoir quelle partie de la Tuile doit être vérifiée (haut, bas, gauche ou droite). Elle retourne un booléen qui vérifie, à l'aide de **boolean estEgal(),** si la tuile est compatible ou non.

c. Lanceur (il faut que je le finisse)

3. <u>Interface graphique</u>

a. CaseView

Cette classe est la représentation graphique d'une case sur les dominos. Elle hérite de la classe JLabel. Il s'agit donc d'un composant graphique. Cette classe possède seulement deux constructeurs.



Un premier constructeur sans argument, qui se contente d'afficher un JLabel vide avec une bordure noire, utilisé pour créer les cases vides du plateau.

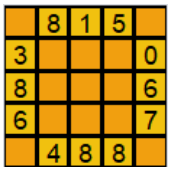
CaseView possède un deuxième constructeur, qui prend en argument une chaîne de caractère et une couleur (awt.Color). Le case retournée possède un texte bien centré, une couleur de fond, une bordure tout autour, et une forme carrée. Ce second constructeur est utilisé pour créer les dominos.

b. TuileDominosView

Cette classe est la représentation graphique d'un domino. Elle hérite de la classe JPanel. Il s'agit donc d'un conteneur. Chaque constructeur définit le layout comme GridLayout, la tuile étant une grille de 5 x 5.

Le constructeur principal prend en argument une TuileDomino. Ensuite, dans une "boucle for", on ajoute les cases les unes à la suite des autres. Comme les cases sont

ajoutées en ligne dans le JPanel, on gère différents cas : les cases vides, celles du haut, celles du bas, à droite et à gauche. Et on ajoute les bordures.



II. Carcassonne

1. Ecriture du code

Il y a une classe parent Lieu et plusieurs classe qui hérite de celle-ci et qui représentes les autres lieux possibles(abbaye, chemin…).
Les tuiles de carcassonne possèdent donc 5 attributs lieu qui sont représenté sur les 4 côtés des tuiles et en leurs centre. De plus elles ont un entier id qui permet de les liés à une tuile de l'interface graphique et un boolean symbole qui permet de savoir si oui ou non il y a pion sur la tuile.
La fonction estCompatible() est très simple car elle utilise le même système d'index que celle de tuileDominos sauf qu'elle vérifie simplement si les lieux correspondent. La fonction tourneTuile() est aussi très simple puisqu'elle consiste simplement a intervertir les positions de chacun des lieux dans le sens anti-trigonométrique.
La fonction calculId regarde chaque côté de la tuile et associe à chaque côté un chiffre. Elle sert à associer à chaque tuile un identifiant, qui fera le lien entre une tuile et son image.

2. Interface graphique

a. TuileCarcassonneView

Cette classe est la représentation graphique d'une tuile de Carcassonne. Elle hérite de la classe JPanel.

La classe BufferedImage est une sous-classe de la classe Image, utilisée pour manipuler et gérer les données d'images. L'identifiant id est propre à chaque tuile créée, pour lui associer une image.

Les constructeurs ont le même fonctionnement : ils tentent d'initialiser l'image bi. Pour cela, on utilise la méthode read() de la classe ImageIO, qui prend un fichier File() en argument et retourne une image de type BufferedImage. Cette action est capturée dans un try-catch. Si le programme n'arrive pas à lire l'image, il retourne l'erreur "Can not read the input file".

La seule grosse différence est le troisième constructeur. Il sert à placer des pions. Il s'agit juste de dessiner un cercle sur la tuile à l'aide de Graphics.fillOval().

La dernière étape consiste à afficher l'image. Pour cela, on va dessiner l'image, avec la méthode paintComponent(Graphics g) et la méthode Graphics.drawImage().

III. Classes communes

1. Class Joueur

La classe joueur est commune aux deux jeux, elle contient 6 attributs. Il y a les attributs couleur et nom qui comme leur nom indique servent a rendre l'interface lisible en attribuant à chaque joueur un nom et une couleur. Cette classe sert essentiellement a enregistré des infos et a séparé la tour. Les quelques fonction qu'elles contient sont **piocher()** et **placerTuile()** dans lesquels on appelle simplement les fonctions du même nom de la classe plateau. Il y a aussi un attribut pion propre au Carcassonne. L'attribut booléen bot permet, comme son nom l'indique, de choisir si oui ou non le joueur est joué par l'ordinateur. Enfin l'int score stock le score du joueur et la Tuile tuileCourante représente la tuile dans la main du joueur.

2. Class Plateau

La classe Plateau soulève plusieurs problèmes : comment faire pour qu'elle fonctionne avec le domino et avec le Carcassonne ? Comment faire la pioche du domino ? Comment faire un plateau dynamique ?

a. Constructeur et fonctionnement

La classe Tuile permet de faire le lien entre le domino et le Carcassonne, elle permet d'avoir une seule class plateau pour les deux types de tuile. De plus, l'utilisation de instanceof permet de faire des fonctions communes au deux jeux. La classe plateau possède dix attributs :

```
Tuile[][] plateauActuel; // Un Plateau est un tableau de tableaux de Tuiles.
ArrayList<Tuile> defausse; // Une ArrayList des tuiles qui sont défaussées.
Tuile[] pioche; // Un tableau de tuile représentant la pioche (le sac).
int[] tuilesRestantes; // Un tableau d'entiers représentant les indexes des tuiles restantes.
int[][] pionplacé; // Un tableau d'entiers représentant les indexes des pions placés.
int nbCasesCote; // Nombre de cases du plateau = nbCasesCote * nbCasesCote
String msgStatut = "Début du tour"; // Message indiquant le statut des actions
int xCentrePlateau, yCentrePlateau; // Index x,y du centre du plateau
boolean isCarcassonne; // Indique si le jeu joué est carcassonne ou pas
```

Le constructeur de plateau prend deux arguments : un booléen isCarcassonne et un entier nbTuiles. L'entier nbCasesCote est initialisée a afin que dans le cas où toutes les tuiles sont placé en ligne, il n'y est pas de soucis. Il y a deux cas : Si isCarcassonne est true le tableau pioche se rempli des tuiles du Carcassonne qui ont été manuellement entré. Dans l'autre cas, la pioche se remplit des tuiles du domino qui sont générée via la class Générateur dont le fonctionnement sera explicité plus tard. Le tableau des pions placé est rempli de -1 signifiant qu'aucun pion placée. Enfin, la fonction **void centrePlateau**() est appelée et celle-ci défini le point centrale du plateau.

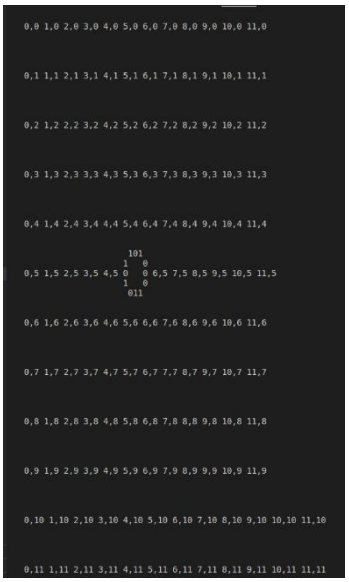
Le tableau tuilesRestantes est un tableau de taille nbTuiles qui contient les nombres de 1 à nbTuiles. Dès que la i-ème tuile de la pioche est jouée ou défaussée, le i-ème élément de tuilesRestantes est remplacé par un -1. Lorsque le tableau n'est plus composé que de -1 la partie s'arrête, cela est vérifié par la fonction **boolean resteTuile().**

b. Fonctions principales

La fonction **Tuile[] tuilesAdjacentes()** prend des coordonnées x,y et regarde les quatres case adjacentes. Pour chaque cas non nul, le tableau se rempli de l'index 1 à 4 comme expliqué précédemment. La fonction **boolean placer()** prend en argument des coordonnées x,y, un Joueur j ainsi qu'une Tuile t qu'on veut placer. Le tableau de tuilAdj est de taille quatre et contient a chaque index soit une tuile soit null. Pour chaque case vide du tableau on incrémente cntVide qui sera utile dans la suite de la fonction. Si la case n'est pas vide, il y a deux cas : la tuile est une instanceof TuileDominos ou une instanceof Tuiles Carcassonne. Cette dichotomie permet de cast le type de tuile dans les deux cas afin que le programme sache quelle fonction **boolean estCompatible()** utilisé car les deux types de tuiles n'ont pas la même. Chaque test est stocké dans la variable compatibleAvecAll. Il y a deux cas où le placement d'une tuile est possible si le plateau est vide ou si il y a au moins une tuile adjacente au coordonnées x,y et que celle(s) est(sont) compatible(nt). Dans le cas où ce sont des TuileDominos, le score est compté et stocké dans le score du joueur en argument. Enfin, si les tests sont passé, la tuile est placé sur plateauActuel et la fonction renvoie true ; sinon rien ne se passe et la fonction renvoie false.

La fonction **Tuile piocher()** initialise un nombre aléatoire nb, si le nombre l'index nb de tuileRestantes est -1, elle return **piocher()** et ce jusqu'à ce que le nombre a l'index nb soit différent de -1 (il n'y a jamais que des -1 car le jeu est arrêté avant si c'est la cas). Lorsque le test est passé, une Tuile temp est initialisée et il y a encore deux cas, Carcassonne ou domino (avec instanceof).Une fois de plus, les deux cas ne diffère que dans le type des tuiles. On stock dans une variable p, du type voulu, le nb-ème élément de la pioche qui est cast a ce même type. Une nouvelle tuile, qui est une copie en profondeur de p, du type désiré est affecté temp. Enfin, le nb-ème élément de pioche devient null et celui de tuileRestantes est maintenant -1. Et on retourne temp.

Il reste les fonctions **void defausser()** et **String toString().** La première ajoute une tuile a la défausse et la seconde affiche le tableau dans le terminal de la manière suivante :



La première tuile est au centre et les coordonnées possibles pour poser la prochaine forment le plateau.

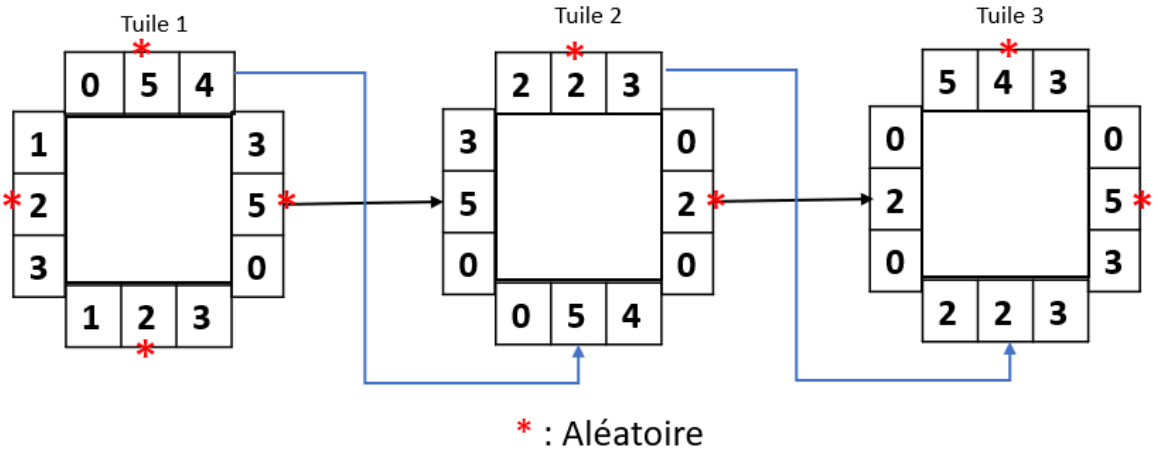
3. Lanceur (partie terminal)

Le lanceur pour la partie terminal contient trois méthodes : joueurtuile() qui demande a l'utilisateur où il veut placer sa tuile. Si c'est correct la place, sinon lui demande s'il veut faire autre chose. Defausser() qui comme son nom l'indique permet de défausser. Enfin la fonction tourner() qui permet de tourner la tuile autant de fois qu'on le veut. Le lanceur suit le déroulement d'une partie commence par demander le nombre de joueur, lesquels sont des ia (qui ne feront que de défausser), et le nombre de tuiles. Après il demande au joueur les actions qu'il veut faire et appelle les fonctions selon les choix de celui-ci. La partie se finie lorsqu'il n'y a plus de tuiles et le classement est affiché.

4. Générateur

La classe générateur est composé de deux fonctions qui renvoie toutes des Tuiles(Carcassonne ou Domino), chacune génère les tuiles d'un des deux jeux. Le générateur de Carcassonne est très simple puisqu'il a été écrit a la main car les cartes du Carcassonne ne sont pas aléatoire et ne suivent pas un paterne de création spécifique.

Le générateur du domino est plus intéressant car il répond à la problématique suivante : comment rendre le domino jouable sans le rendre trop simple ? Pour cela le générateur génère une première tuile et stock les valeurs 2 des 4 trio dans tableaux appelé temp1 et 2. Une fois que la première tuile est créée, complètement aléatoirement, la tuile suivante est composé de deux nouveaux trio aléatoire en haut et à droite. Le trio gauche et le trio bas sont eux les mêmes que les trio haut et droite de la tuile précédente :



IV. Interface graphique : classes communes, lancement des jeux et menu

1. Classe commune : PlateauView

Cette classe représente la vue du plateau. Elle hérite d'un JPane.

Ce plateauView est un peu spécial. En effet, il n'affiche que la partie du plateau qui est utilisée et utilisable. Son affichage change donc à chaque fois qu'on dépose un domino ou une tuile de Carcassonne.

Les autres attributs x et y sont les coordonnées qui délimitent l'espace du plateau complet qui sera affiché à l'écran. Cela évite d'afficher un nombre trop grand de cases, rendant les cases plus lisibles.

Le constructeur prend en argument un plateau. Au départ, les coordonnées min et max à afficher sont celles qui se situent autour du centre du plateau. Le layout est un GridLayout du nombre de lignes et de colonnes.

Dans les getter et setter, nous avons ajouté nous même les *getXView* et *getYView* qui renvoient les coordonnées x,y sur le plateauView en fonction de l'index. Cette fonction convertit en quelque sorte cet index.

Enfin, la méthode *update(int xpl, int ypl, int Plateau pl)* met à jour la vue du plateau (PlateauView). Les xpl et ypl sont les coordonnées de la tuile qui vient d'être posée, coordonnées par rapport au plateau et non au plateauView.

L'étape 1 de la mise à jour consiste à actualiser les variables min, max, plateau, ligne et colonne.

L'étape 2 consiste à actualiser l'affichage. Pour cela, on supprime tout. Puis on recrée une grille de ligne x colonne. Ensuite, on parcourt le plateau dans l'espace délimité par (min,max). S'il n'y a pas de tuiles aux coordonnées actuelles, on ajoute une case vide. Sinon, on regarde de quel type de tuile s'agit-il grâce à un *instance of* et on ajoute la tuile.

2. Classes de jeux : DominosGame et CarcassonneGame

a. DominosGame

Cette classe représente une partie de Dominos. Cette partie se joue dans une fenêtre, donc la classe hérite de JFrame.

Ses attributs sont un plateau, un plateauView, une liste des joueurs, plusieurs panels utiles, des boutons pour le jeu, des messages de statut, la tuile actuellement en pioche (c'est -à-dire dans la main) ainsi que l'index du joueur actuel.

Son constructeur prend en argument une liste de joueurs et un nombre de tuiles (si jamais nous voulons modifier facilement ce paramètre). On commence par créer la fenêtre avec l'appel de super(). On définit l'opération exécutée lorsque on ferme la fenêtre. Ici, ça revient à exécuter System.exit(0), qui ferme toutes les fenêtres et termine le programme. On définit la taille, place la fenêtre au centre de l'écran (this.setLocationRelativeTo(null), l'argument null étant le bureau). Ensuite, on crée le conteneur principal (this.getContentPane). On définit le layout comme un BorderLayout, permettant de placer les éléments à dans des zones précises de l'écran. On crée le plateau, le plateauView et le plateauView de type déclaré JPanel. On définit les variables de liste de joueur et joueur actuel. On ajoute les éléments au conteneur principal (win), c'est-à-dire le panelDroit, panelGauche et le panelPlateauView. Enfin, on active les différents gestionnaires d'événements.

Ensuite les méthodes : création de différents panels pour le jeu, événements du jeu et enfin g estion des tours du jeu.

Tout d'abord, la création du panel de droit crée un JPanel de type FlowLayout qui contient les boutons pour piocher, tourner la tuile, défausser la tuile, l'affichage de la tuile actuelle, ainsi qu'un message de statut qui indique l'action qui vient d'être effectuée ou une éventuelle erreur.

Le panelGauche est assez similaire, avec un bouton pour arrêter le jeu, un bouton pour revenir au menu principal, un message de statut qui indique le tour du joueur ainsi qu'un learderBoard (créé grâce à *createLeaderBoard()*) qui indique les scores des joueurs.

Ensuite, les événements de jeu. Tout d'abord, piocherView() : s'il reste des dominos à piocher, on regarde quel type de joueur vient de piocher. Si c'est un bot, on appelle tourDuBot(). Sinon, on supprime la pioche actuelle et on la met à jour. On met à jour le message de statut, active les boutons tourner et défausser (setVisible()) et désactive le bouton piocher (pour empêcher de piocher à nouveau). Puis on appelle *appliqueEffectForAll()*. En revanche, si on a pas pu piocher, alors le jeu est terminé !

defausserView() : on empêche le placement d'une tuile ou l'affichage du sélecteur. Cela revient à désactiver tous les mouseListener grâce à unAppliqueEffectForAll(). Puis on défausse la tuile et met l'affichage en mode début de tour (avec uniquement le bouton piocher visible). Puis on lance la méthode finDeTour().

TournerView() : on supprime la tuile en pioche avant de la tourner. Puis la tuile en pioche devient la même tuile mais tournée.

PlacerView(int index) : index équivant à une case. Si on clique sur cette case (redéfinition de la méthode mouseClicked), on commence par calculer certaines variables, puis on tente de placer le domino. Si le placement est bon et effectué, on met à jour le plateau. Puis on empêche un nouveau placement et on met l'affichage en mode début de tour, avant de lancer le gestionnaire de fin de tour. En revanche, si le placement n'a pas fonctionné, seul le message de statut est mis à jour.

La méthode stopAndMenuView() gère les boutons “abandonner” et “menu principal”. Dans les deux cas, on ouvre une boîte de dialogue qui explique à l’utilisateur son action. S’il valide l’action, le bouton Abandonner ferme toutes les fenêtres et arrête le programme avec l’exécution de System.exit(0). Et le bouton Menu principal ferme la fenêtre actuelle (this.dispose()) et lance une nouvelle fenêtre de menu (Lancement.java).

afficheSelection(int index) affiche un sélecteur. Lorsqu’on survole une case, on change la couleur des bordures par la couleur du joueur. Et lorsqu’on quitte la case, on remet la couleur et l’épaisseur originales.

AppliqueEffectForAll() applique tous les événements à toutes les cases. Cette méthode parcourt chaque case du plateauView. S’il y a déjà une tuile posée, elle s’assure que tous les événements sont désactivés pour cette case. Sinon, elle applique les événements *afficheSelection* et *placerView*

UnAppliqueEffect désactive tous les mouseListener d’une case, c’est-à-dire afficheSelection et placerView. Et UnAppliqueEffectForAll applique la méthode à toutes les cases. Elle empêche donc le placement d’une tuile sur cette case ou l’affichage du sélecteur.

Enfin, les gestionnaires de tour. Tout d’abord, finDeTour() : met à jour le joueur actuel ainsi que le message de statut de joueur, puis actualise le leaderBoard. Enfin, si le joueur actuel est un bot, on lance tourDuBot().

TourDuBot() simule un tour joué par une IA. Notre IA n’étant pas très intelligente, elle se contente de piocher puis défausser la tuile.

b. CarcasonneGame

Cette classe est essentiellement identique à DominosGame. Nous aurions même pu fusionner les deux classes. Toutes fois, quelques différences subsistent :

- Il n’y a pas de learderboard.
- La fonction PlacerView possède queques lignes en plus, qui servent à déterminer où le pion va être placé.

c. Lancement

C’est la classe principale. Elle fait office de menu. Plusieurs sous-menus s’affichent les uns après les autres permettant de paramétrer le jeu au maximum et de remplir les conditions du cahier des charges.

V. Piste d’extension et représentation graphique du modèle des classe

1. Piste d’extension

-Finalisation du Carcassonne avec l’implémentation totale des règles.

-Retouche et nettoyage graphique, notamment du domino pour que les tuiles soit forcément carré et la trop grande quantité de boîte de dialogue.

-Améliorer les ia qui sont très primitive.

-Amélioration de l'interaction avec la machine

-Fusion des classes DominoGame et CarcasonneGame, leur code est très similaire.

2. Représentation graphique du modèle des classes