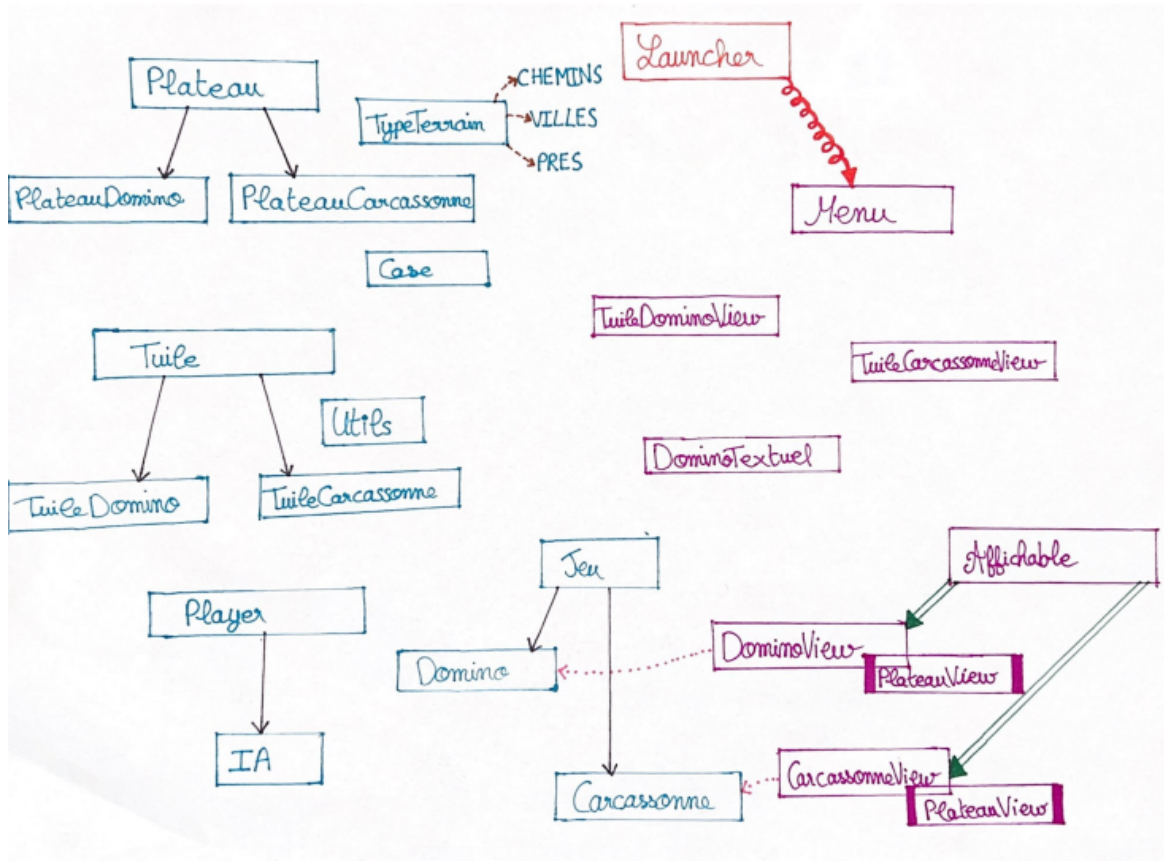


Rapport Projet de Programmation Orientée Objet

Ayadi Jana & Singh Navdeep, MI1

I. Représentation graphique de l'organisation de notre code:



Légende:

- : relation classe → sous-classe
- > : constante de l'énumération
- ⇒ : interface implémentée
-> : contenant dans son constructeur un objet de cette classe
- ☐ : classe interne
- ~~~~> : appel du Menu

A l'annonce du sujet et des consignes à suivre nous avons certaines craintes étant donné que nous n'avions jamais réalisé un projet comme celui-ci auparavant. Aborder ce sujet nous paraissait mission impossible. Mais au fur et à mesure du temps, des échecs et des réussites, nous avons pu nous approprier le sujet et dépasser nos appréhensions.

II. Réflexion dans la conception du modèle

a) Conception global du modèle

En premier lieu, nous avons essayé de faire des classes générales, abstraites ou non, dans le but d'avoir des classes qui seront communes à Domino et Carcassonne. Nous avons vite été perdus car nous n'avions pas en tête le pattern de l'implémentation du jeu Carcassonne, et les éventuelles similitudes entre les deux jeux. Nous avons donc décidé de nous concentrer tout d'abord uniquement sur le jeu Domino. Puis en commençant l'implémentation de Carcassonne la refactorisation nous est apparue naturelle.

b) Premiers pas dans la conception

Lorsque nous avons abordé la réalisation de Carcassonne nous avons réfléchi aux différentes possibilités d'implémentation pour les types de terrains. Étant donné qu'il n'était pas forcément très judicieux de faire plusieurs sous classes étendant TypeTerrain, qui n'auraient pas eu de corps, nous avons fait des recherches et approfondi nos connaissances sur l'utilité et l'utilisation des énumérations.

Ensuite, lors de la création de tuiles pour Carcassonne nous avons imaginé un système aléatoire.

```

public TuileCarcassonne(){
    this.nord=terrainAlea();
    this.sud=terrainAlea();
    this.est=terrainAlea();
    this.ouest=terrainAlea();
}

public TypeTerrain terrainAlea(){
    int a=rng.nextInt(bound: 4);
    switch(a){
        case 0: return TypeTerrain.CHEMINS;
        case 1: return TypeTerrain.VILLES;
        case 2: return TypeTerrain.PRES;
        case 3: return TypeTerrain.ABBAYES;
        default : return null;
    }
}
}

```

Plus tard dans la réalisation du code, nous avons remarqué que cette implémentation posait problème puisque les tuiles doivent respecter des contraintes. *Exemple: s'il y a une route, elle doit être soit reliée à une ville ou une abbaye, soit connectée à une autre route, ...* Et l'adaptation de notre constructeur à ces conditions nous donnait un code assez long et avec de nombreux *if*, le rendant peu ergonomique. C'est pour cela que nous nous sommes penchées vers l'idée de créer nos propres tuiles directement dans le constructeur de Carcassonne. Cela nous permettant de rajouter selon notre vouloir le nombre et le type de tuiles souhaités.

Ensuite, au moment de l'implémentation de la fonction de rotation, nous avons pris la décision de ne pas manipuler directement les références des Types de terrain mais de passer par le biais d'une fonction auxiliaire et des copies, ce qui rend le code plus long mais aussi moins ambigu.

Pour tester le bon fonctionnement de notre code avant d'implémenter la vue de l'interface graphique, nous avons créé, en amont, une fonction facultative d'affichage dans le terminal:

```

public void affiche(){
    System.out.println(x: "nord");
    if(nord==TypeTerrain.CHEMINS)System.out.println(x: "chemins");
    if(nord==TypeTerrain.PRES)System.out.println(x: "pres");
    if(nord==TypeTerrain.ABBAYES)System.out.println(x: "Abbayes");
    if(nord==TypeTerrain.VILLES)System.out.println(x: "villes");
    System.out.println(x: "");
    //Idem pour sud, est et ouest
}

```

c) Un peu plus loin dans l'implémentation...

Concernant l'intervalle des chiffres dans Domino, nous avons initialement choisi de générer des Dominos contenant des chiffres entre 0 et 5. Nous avons néanmoins remarqué que le début du jeu était un peu latent : les joueurs étaient souvent amenés à défausser car peu de probabilités de poser ; nous avons donc arbitrairement réduit l'intervalle de 0 à 2 pour une soutenance plus dynamique.

Pour gérer le déroulement d'une partie, nous avons dans le jeu un attribut `actualPlayer`. Pour le faire évoluer, nous nous basions sur l'id des joueurs, attribué lors de leur création. Mais ces id ne s'adaptent pas à la taille de la liste des participants, posant problème en cas d'abandon. Nous nous sommes ensuite basées sur l'index du joueur dans l'ArrayList participants et avons implémenté une fonction `auSuivant()` que nous appelons à toutes les possibilités de finir un tour : placer une tuile, passer son tour, abandonner.

Pour le bon fonctionnement d'une partie, particulièrement dans le cas où un unique joueur joue contre une ou des IA, nous avons créé une IA qui finira forcément par poser ses tuiles si elle en a la possibilité, en parcourant toutes les cases du plateau et pour toutes les rotations possibles.

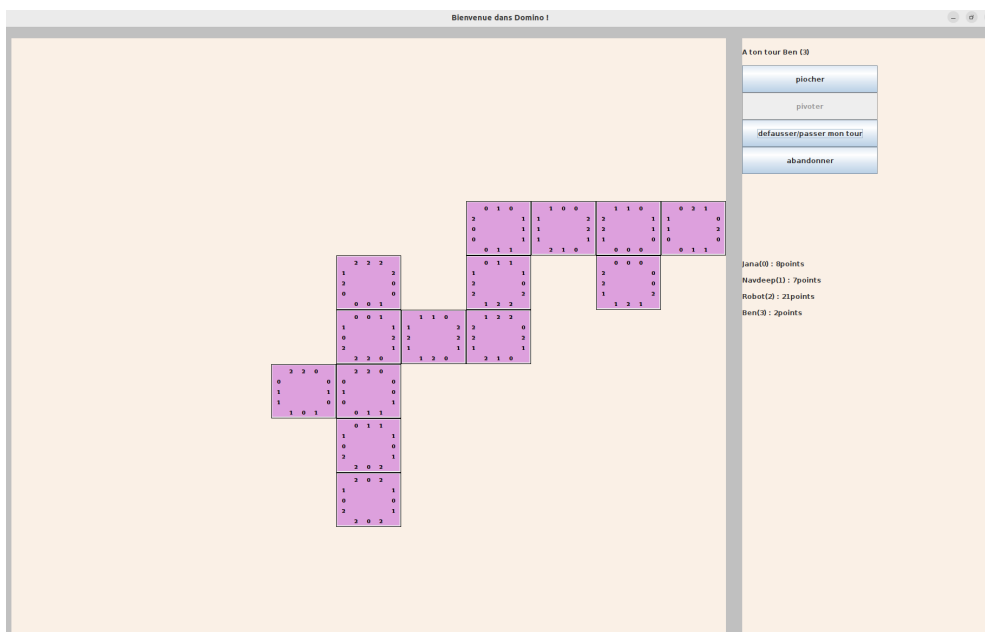
Notons que la partie peut aussi se jouer en solitaire, pour s'adapter au vouloir de chacun.

III. Les interfaces graphiques

```

----- Au tour de Robot (2) -----
  0   1   2   3   4   5   6   7   8
-----
1  .....
  .....
  .....022.....
  .....1 2.....
  .....0 0.....
  .....2 2.....
  .....222.....
  .....121,222,221.....
  .....0 11 22 2.....
  .....0 00 22 0.....
  .....1 00 00 2.....
  .....201,101,112.....
  .....112,011,220.....
  .....0 22 00 1.....
  .....1 11 11 1.....
  .....1 22 11 1.....
  .....021,112,111.....
  .....102,101,021.....
  .....2 22 11 1.....
  .....0 22 00 1.....
  .....1 11 22 2.....
  .....110,122,200.....
  .....112,110,200,101,022.....
  .....0 11 1,2 22 00 2.....
  .....1 00 0,0 00 22 2.....
  .....2 00 2,0 11 11 2.....
  .....112,211,100,221,020.....
  .....000,112,100.....
  .....2 00 0,2 1.....
  .....0 00 1,0 2.....
  .....1 22 1,2 1.....
  .....122,212,111.....
  .....200,111,101.....
  .....0 22 22 2.....
  .....2 22 22 0.....
  .....2 22 11 2.....
  .....010,120,112.....
  .....010.....
  .....2 2.....
  .....0 1.....
  .....2 1.....
  .....021.....
  .....
  .....
  .....
10 .....
-----
Robot (0) : 23 point(s)
Robot (1) : 22 point(s)
Robot (2) : 21 point(s)
Voici le(s) vainqueur(s) : Robot (0)
jane@jane-VirtualBox: /Documents/L2/SinghNavdeep-AyadiJansS

```





a) La vue générale

En bref, la vue textuelle du Domino est implémentée de sorte que seule la surface utile du plateau soit affichée à l'écran : on utilise pour cela les méthodes `getMaxI()`, `getMinI()`... qui déterminent les indexs des tuiles aux bornes de la surface utile. L'affichage du plateau est construit ligne par ligne, et non pas tuile par tuile. D'où la présence, dans tuile, de 5 fonctions d'affichage, une pour chaque ligne de nos tuiles.

Les vues graphiques respectives pour le Carcassonne et le Domino (`CarcassonneView` et `DominoView`) sont organisées assez simplement : une classe pour la vue des tuiles et une classe pour la vue générale du jeu.

Dans cette dernière, nous avons ajouté différents liens vers des objets du modèle, afin que l'exercice d'une action sur le plateau ou sur les boutons (piocher, passer son tour, pivoter, abandonner...) par l'utilisateur puisse mettre à jour les données du modèle.

On y trouve également deux attributs `modelT` et `viewT`, faisant référence au modèle et à la vue de la tuile qui est en train d'être jouée.

Les vues implémentent toutes deux une interface `Affichable` qui regroupe l'ensemble des fonctions qu'elles doivent impérativement satisfaire. Notons que les différences dans ces méthodes sont minimales mais empêchent de les rassembler complètement sous une seule méthode définie qui serait héritée des deux jeux.

b) Le plateau de jeu

Penchons nous sur la construction graphique du plateau.

Nous avons essayé d'implémenter la vue du plateau sous forme d'une classe externe à la vue générale mais cela nous a posé un problème conséquent que nous n'avons su résoudre :

- les « `MouseListener` » ne fonctionnaient plus sur l'attribut plateau

Nous avons donc choisi de définir l'objet plateau comme une classe interne de `DominoView` et de `CarcassonneView`, décision également motivée par le fait que l'existence d'un plateau n'a de sens que dans le contexte d'un jeu.

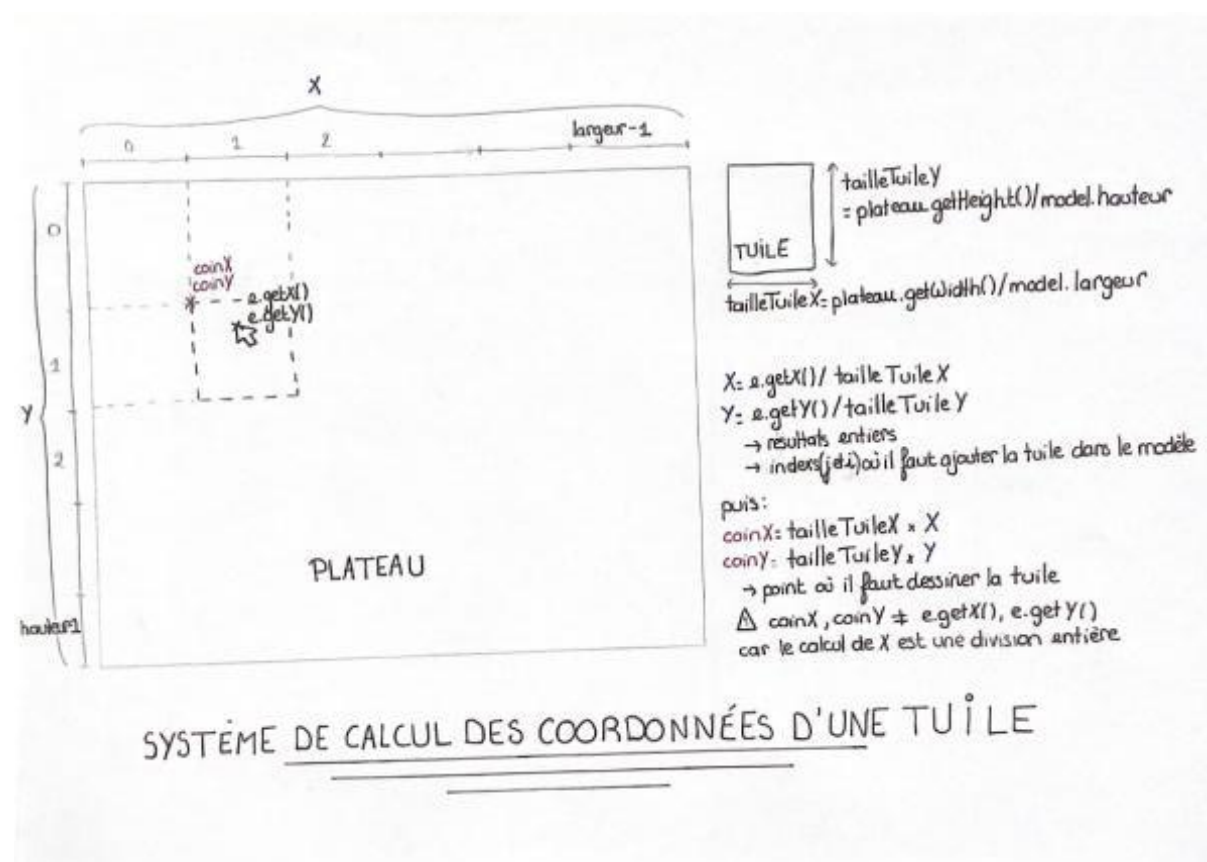
Concernant les principes de son fonctionnement nous avons tout d'abord songé à utiliser un `GridBagLayout` ou un `GridLayout`.

Nous voulions, en bref, que les cases du layout implémentent l'interface `MouseListener` et affichent une tuile au moment d'un clic de la souris.

Cependant, dans notre raisonnement, nous devons aussi faire un calcul de coordonnées pour savoir où ajouter la tuile dans le double tableau du modèle.

Nous avons donc décidé d'adopter une stratégie "2 en 1".

Voici un schéma explicatif du calcul des coordonnées et du placement des tuiles sur le plateau.



Notre raisonnement est orienté mathématiques, il s'appuie sur la proportionnalité et tire partie de la division entière de java.

Remarquons que pour le tour d'une IA dans la vue graphique, c'est autre chose : il n'y a pas de mouseClicked, on utilise d'abord les index du tableau transmis par l'IA et ensuite plaçons la tuile au bon endroit sur le plateau.

Les tailles du plateau et des tuiles, ainsi que les coordonnées calculées sont proportionnelles entre elles, de sorte qu'un changement de la taille du plateau ou des dimensions du double tableau du modèle n'impacte pas le fonctionnement du système.

Dans cette implémentation nous avons cependant été confrontées à des difficultés :

- Par dépit, nous nous étions résiliées à donner une taille fixe au plateau et ce n'est que très tard que nous avons réussi à faire un plateau dont la taille s'adapte à la fenêtre qui le contient.
- Il fallait en fait impérativement faire attention à ce que les dimensions soient initialisées avant leur utilisation dans les calculs (pour éviter une division par 0 et que l'affichage soit conforme aux attentes).

c) Les tuiles

Si le visuel des tuiles du Domino laissait peu de choix, il en est autre pour celles de Carcassonne.

Notre première pensée était de réutiliser les photos des tuiles originales fournies dans les règles pour que le visuel soit conforme à l'original.




Mais cela impliquait de saisir "à la main" les informations de la tuile modèle correspondante et surtout, cela limitait la flexibilité.

En effet, nous n'aurions pu créer d'autres types de tuiles que les 12 originaux ou instancier des tuiles aux paysages aléatoires avec une vue qui s'y adapte.

Nous avons donc utilisé les méthodes à disposition dans la classe java.awt.Graphics pour reproduire au mieux un visuel explicite.

IV. Récapitulatif de notre projet

Cahier des charges :

Consigne à implémenter	si  ou  nous avons réussi à l'implémenter
choisir un jeu	

choisir un nombre de joueurs	✓
sélectionner “humain” ou “IA” pour chaque joueur	✓
l’implémentation de toutes les règles du Domino	✓
view terminal pour le jeu Domino	✓
implémentation partielle du jeu Carcassonne	✓
interface graphique pour Domino	✓
interface graphique pour Carcassonne	✓

De plus, notre conception nous permet d’imaginer d’autres TypeTerrain, comme modéliser des rivières sur nos tuiles de Carcassonne en ajoutant seulement un TypeTerrain *rivières* et en ajoutant la gestion du cas rivière dans la construction de la vue d’une tuile. Cela aurait pu constituer une extension si nous avions trouvé le temps de l’implémenter.

V. Conclusion

Nous avons essayé de faire des jeux qui soient le plus proche possible de la réalité en fonction de nos capacités et s’adaptant aux désirs de chacun. Ce projet nous a permis de nous dépasser, en nous obligeant à prendre des initiatives et rester le plus organisé possible.