Projet de POOIG

Introduction

L'objectif de ce projet était d'utiliser ce que nous avons appris en POOIG ce semestre pour coder un jeu regroupant une variante des dominos et le jeu Carcassonne. Nous avons commencé par modéliser le premier, puis avons arrangé notre code pour qu'il soit générique et que l'implémentation du deuxième soit similaire. Pour ce travail, il nous a semblé essentiel d'utiliser Git afin de pouvoir suivre nos modifications respectives du code. Cela nous a aussi permis de se répartir les issues qui survenaient au fur et à mesure, sans avoir une délimitation exacte des objectifs de chacun. Nous avons donc tous les deux travaillé sur l'ensemble des éléments. Nous avons également décidé de produire un code et une documentation en anglais uniquement.

Ce rapport est construit de la manière suivante : nous rappelons le cahier des charges que nous avons suivi puis nous détaillons la manière dont nous l'avons respecté. Nous approfondissons en particulier les axes qui ont exigé une réflexion plus poussée en expliquant nos choix d'implémentation. Nous finissons par les fonctionnalités optionnelles que nous n'avons pas implémentées ou auxquelles nous avons réfléchi.

Cahier des Charges

Voici un rappel des fonctionnalités que nous avions prévues et programmées. Celles qui ne l'ont pas été figurent plus loin dans le rapport.

Fonctionnalités obligatoires :

- Menu d'accueil qui permet de:
    - Choisir entre Domino et Carcassonne
    - Choisir le nombre de joueurs et/ou de bots
    - Choisir les noms des joueurs
- Jeu Domino complet avec toutes les règles implémentées
- Jeu Carcassonne partiel, implémentation d'une partie complète, sans compter les points et sans tester si le pion peut être placé
- Mode d'affichage «Textuel» pour Domino
- Mode d'affichage «Graphique» pour Carcassonne et Domino
- Un bot pour chaque jeu.
- Séparation complète de vue et modèle
- Fichier README avec instructions de lancement et fonctionnement du jeu
- Script en bash pour simplifier la compilation et exécution du code

Fonctionnalités supplémentaires :

- Accès aux règles de chacun des jeux dans la vue (Textuel et Graphique)
- Bot intelligent pour le Domino
- Affichage du score et d'un classement des joueurs en fin de jeu pour le Domino

- Possibilité de recommencer une partie avec les mêmes paramètres pour l'affichage textuel
- Retour au menu d'accueil possible pour l'affichage graphique
- Possibilité d'abandonner pour un joueur

Modélisation

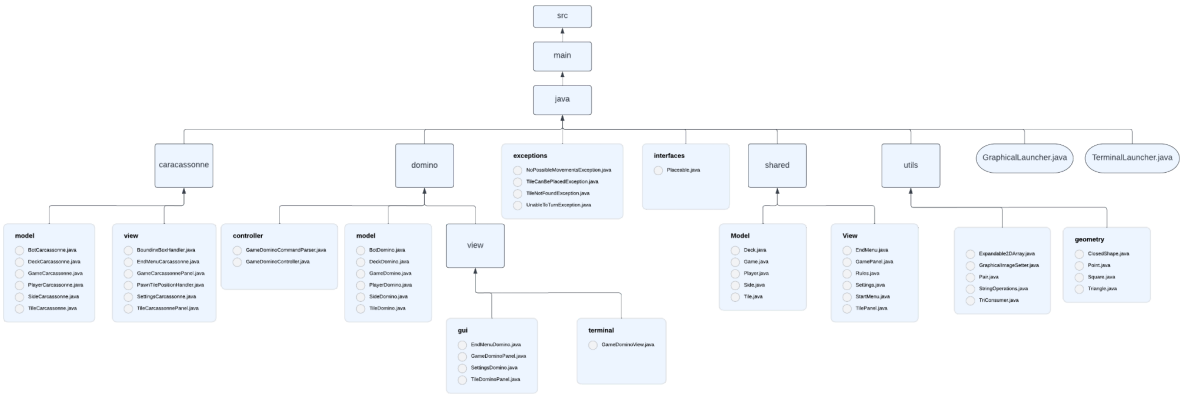


Diagramme de la hiérarchie de classes de notre code présentant les fichiers .java uniquement

- Arborescence

Notre arborescence sépare à un premier niveau les ressources utilisées (*resources/*), comme les images des tuiles de Carcassonne, du code source (*java/*). Dans ce dossier, nous avons séparé l'implémentation de Domino (*domino/*) et de Carcassonne (*carcassonne/*) en deux dossiers, avec un dossier supplémentaire pour les éléments communs (*shared/*). A ce niveau de l'arborescence, nous avons également des dossiers utiles comme *exceptions/*, *interfaces/* et *utils/,* qui regroupe différents outils. C'est à ce niveau également que les fichiers qui lancent les jeux (*GraphicalLauncher.java* et *TerminalLauncher.java*) se situent.

Comme nous avons suivi un modèle de MVC (Model-View-Controller), nos dossiers *domino/*, *carcassonne/* et *shared/* sont subdivisés en deux ou trois dossiers y correspondant. Ainsi, dans *model/*, on retrouve en général les classes concernant le jeu (**Game**), le joueur (**Player**), les tuiles (**Tile**), les côtés de ces tuiles (**Side**) et l'ensemble des tuiles d'une partie (**Deck**). Le controller n'est utilisé que pour le mode de jeu terminal du domino et le dossier correspondant (*controller/*) contient deux classes permettant de gérer les inputs du terminal avec le model. Les dossiers *view/* regroupent les différents éléments d'affichage graphique comme le panneau des règles, des paramètres ou de fin de jeu. Celui de Carcassonne contient également deux classes utiles à la gestion du placement des pions, expliquée plus loin.

Model

- Héritage

Pour modéliser les deux jeux, nous avons utilisé des classes abstraites pour chaque composante : le jeu lui-même, les tuiles, les côtés des tuiles, les joueurs et le deck. Pour arriver à ce stade, nous avons décidé de commencer par coder le jeu de Dominos pour un affichage terminal, et ensuite faire une refactorisation du code. Nous avons ainsi pu mettre dans des classes abstraites le code qui peut être partagé entre les deux jeux et entre les vues pour Domino. Grâce à ça, l'abstraction des éléments du jeu a été assez simple et le résultat est celui qu'on attendait : des classes abstraites qui font la plupart du travail et des classes héritées, plus petites, qui servent seulement à gérer les éléments qui sont différents dans les deux jeux.

Pour résumer, la structure est simple, chaque élément de jeu a une classe abstraite associée et deux classes enfant, une pour chaque jeu. Par exemple, la classe abstraite **Player** correspond à la modélisation d'un joueur. Elle regroupe des informations comme son nom et s'il est toujours en jeu, et possède des méthodes communes qui s'appliquent à un joueur, quel que soit le jeu auquel il joue. Deux classes héritent de Player : **PlayerDomino** et **PlayerCarcassonne**. Elles apportent les spécifications nécessaires pour pouvoir jouer a ces deux jeux, par exemple, le joueur de Carcassonne a une couleur de pion qui lui est attribuée et le joueur de Domino a des méthodes pour gérer son score.

La modélisation des joueurs contient également celle des joueurs IA. Nous avons donc **BotDomino** qui hérite de **PlayerDomino** et **BotCarcassonne** de **PlayerCarcassonne**.

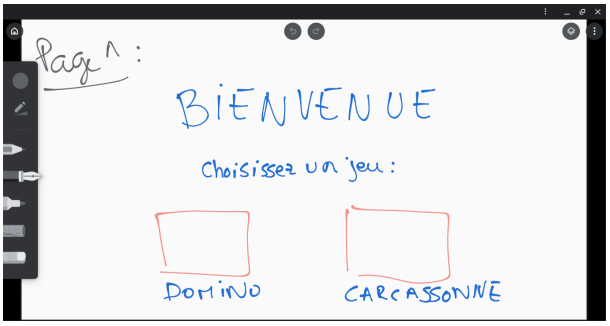
Les tuiles présentent une exception à cette généralité. La classe abstraite **Tile** existe aussi mais elle implémente une interface appelée **Placeable**. Cette interface correspond à un objet qui a 4 côtés et qui peut être placé, et c'est elle qui s'occupe de définir les méthodes qui permettent de savoir si la tuile peut être placée à côté d'une autre tuile ou non.

- Généricité

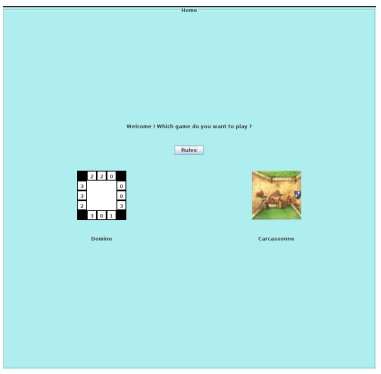
Nous avons exploité la généricité pour gérer plusieurs classes. Un exemple sont les tuiles. La classe abstraite **Tile** utilise un type qui étend un **Side**. Cela nous permet d'implémenter des méthodes qui agissent sur les côtes de la tuile sans avoir à les redéfinir dans chaque classe enfant (les tuiles de Domino et de Carcassonne). Ainsi, la classe **TileDomino** étend **Tile<SideDomino>** et la classe **TileCarcassonne** étend **Tile<SideCarcassone>**. Ce principe est utilisé aussi dans **Game** et **Deck**, paramétrées par des types qui étendent des tuiles, des joueurs ou des côtés.

Vue

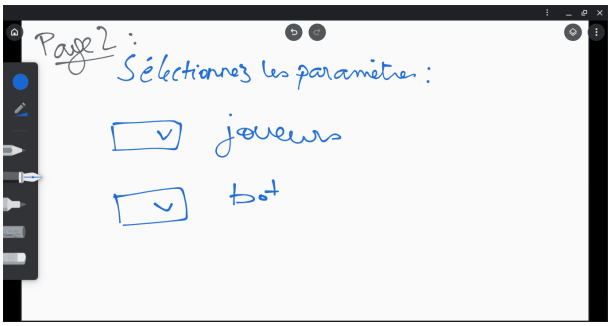
Pour la vue nous utilisons une hiérarchie similaire. Nous avons commencé par imaginer l'interface graphique en 5 parties : un menu d'accueil, une page accessible sur les règles, un affichage du choix de la configuration de jeu, le jeu en soi et une page de fin de jeu. Les images qui suivent correspondent d'une part aux designs que l'on a réalisé et d'autre part au rendu final de notre affichage graphique.



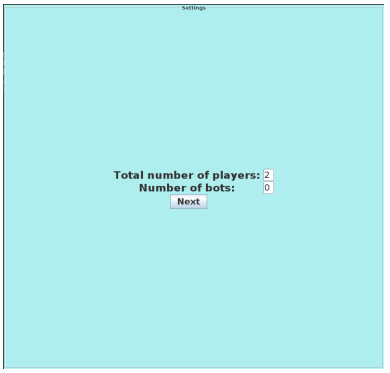
Design de la page d'accueil



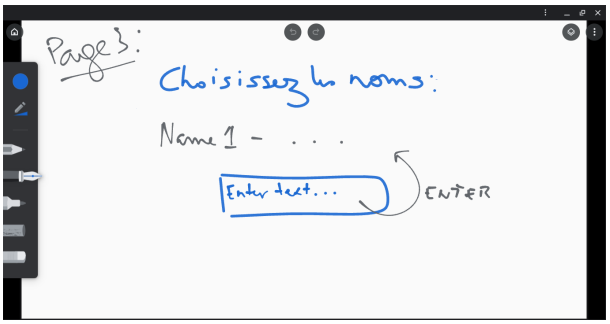
Vue finale de la page d'accueil



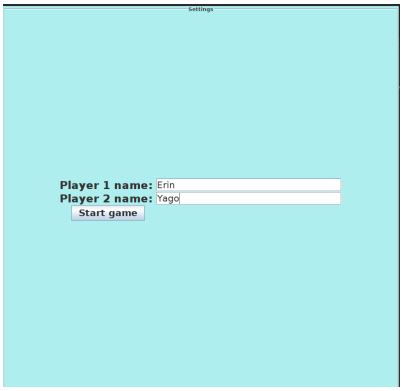
Design de la page de paramétrage



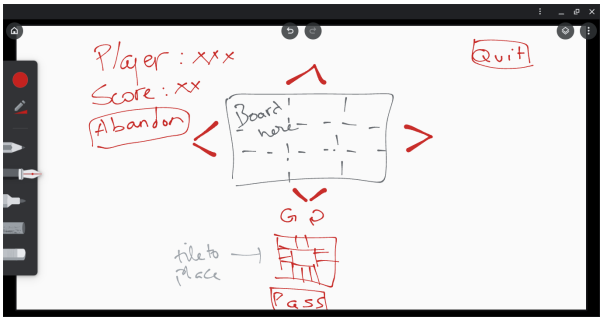
Vue finale de la page de paramétrage



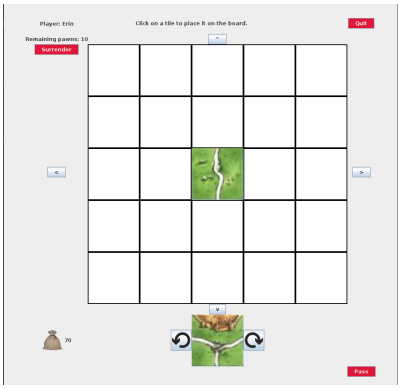
Design de la page de choix des noms de joueurs



Vue finale de la page des noms de joueurs

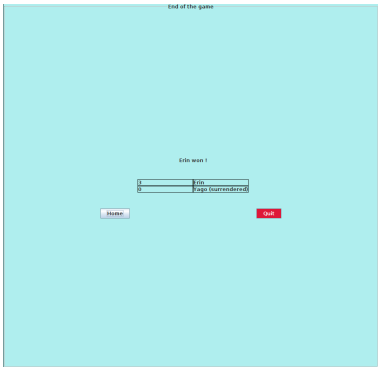


Design de la page de jeu principale

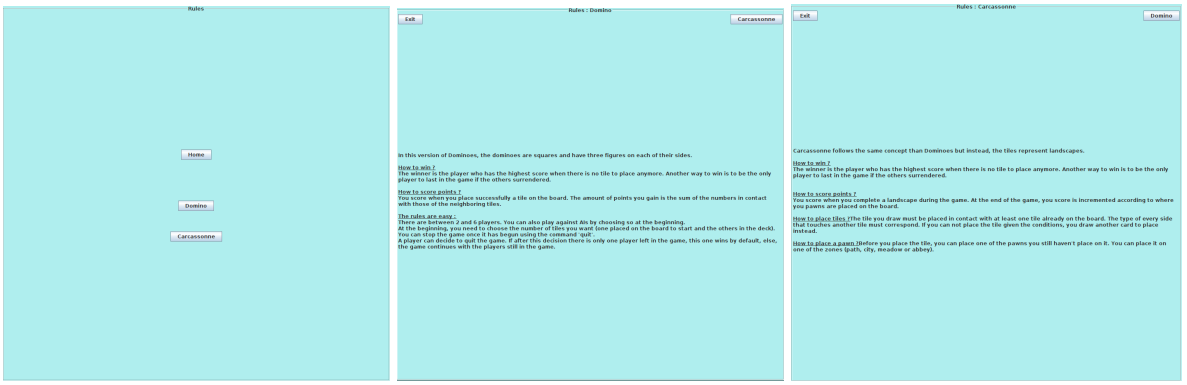


Vue finale de la page de jeu principale de Carcassonne

Design de la page de fin de jeu



Vue finale de la page de fin de jeu de Domino



Vue finale des pages de règles de jeu

Tous ces éléments étendent la classe **JPanel.** Le menu d'accueil et la page des règles sont les mêmes pour Carcassonne et Domino. En revanche, les autres éléments ont été modélisés par une classe abstraite, étendue par des classes filles pour chaque jeu, puisque la plupart des éléments graphiques sont les mêmes, quel que soit le jeu. Comme pour le modèle, nous avons décidé de commencer par l'implémentation de l'affichage des dominos avant de s'attaquer à la création de classes abstraites bien redéfinies, puisque le modèle de Carcassonne était encore en train d'être modifié. Une fois terminée, la

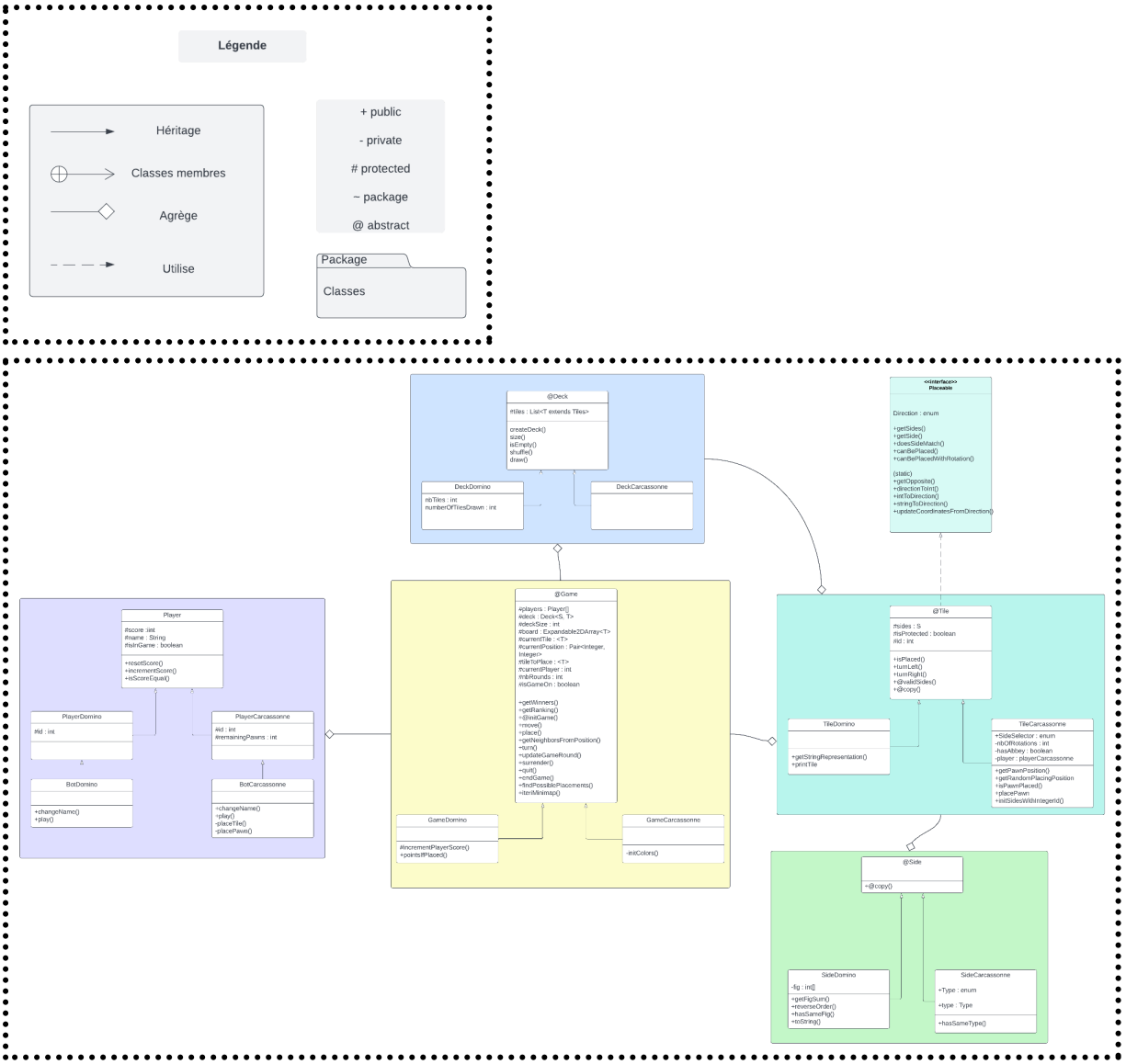
refactorisation du code en un code compatible pour Carcassonne a été simple puisque notre démarche tenait compte de la future structure du deuxième jeu.

Une attention particulière a été portée à la représentation graphique des tuiles. Pour cela nous avons aussi créé une classe abstraite, **TilePanel** qui est étendue pour chaque jeu. Cette classe est utilisée par le **GamePanel** (classe abstraite qui s'occupe de la représentation et la gestion d'une partie dans la GUI) ce qui nous a permis de simplifier énormément l'implémentation des interfaces graphiques pour les deux jeux.

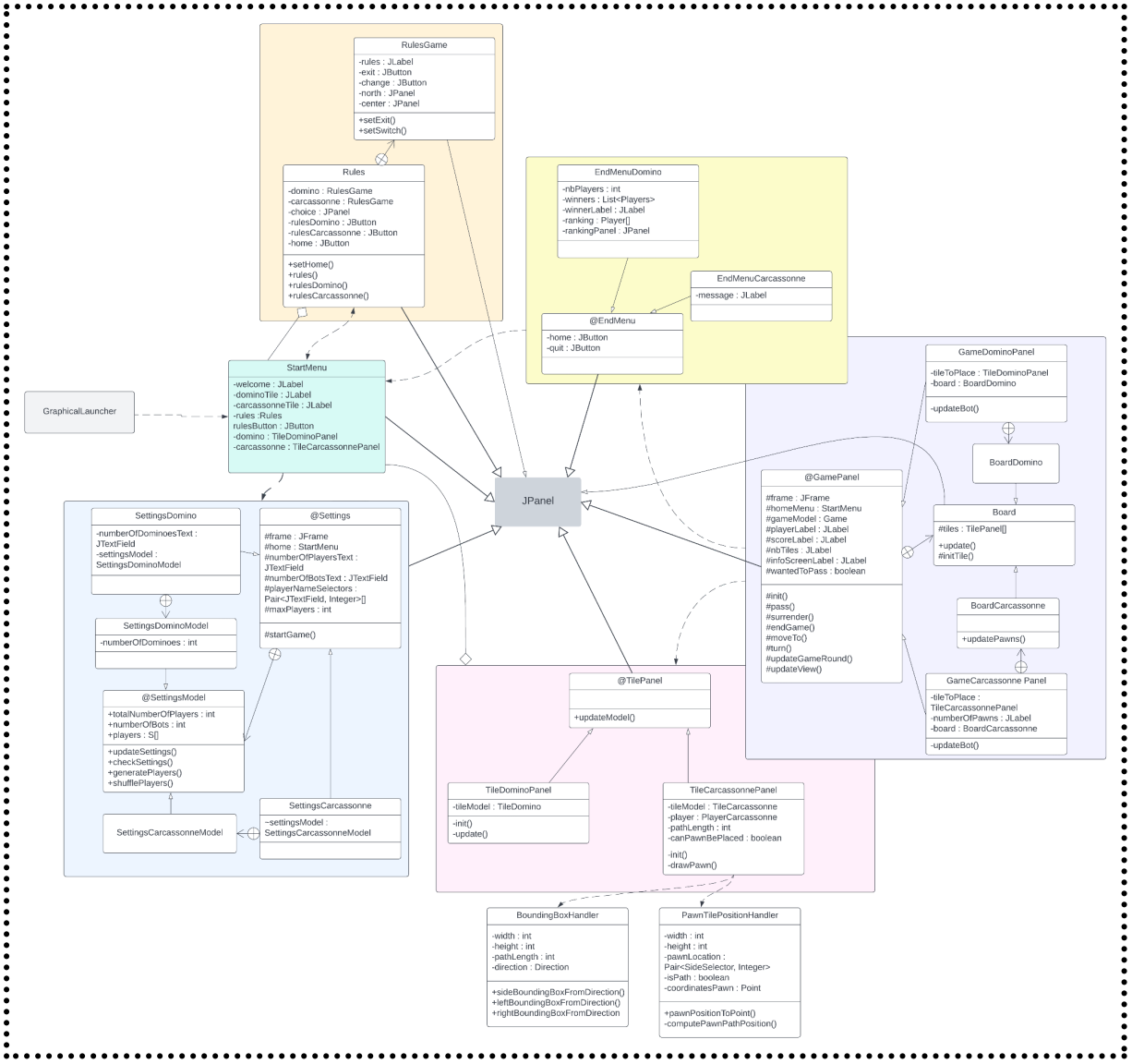
Tout comme pour le modèle, on utilise de manière régulière la généricité afin de rendre plus utiles les classes abstraites.

- Notation graphique (UML)

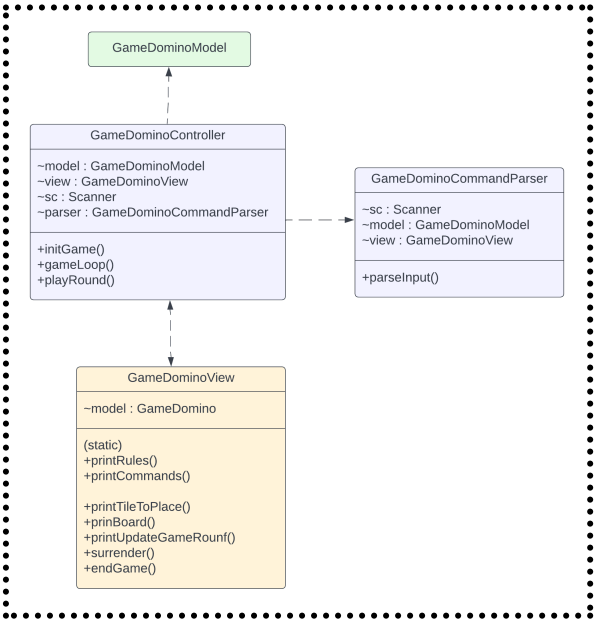
Voici les représentations graphiques des relations (non exhaustives car trop complexes pour être entièrement représentées ici) qui existent entre nos classes.



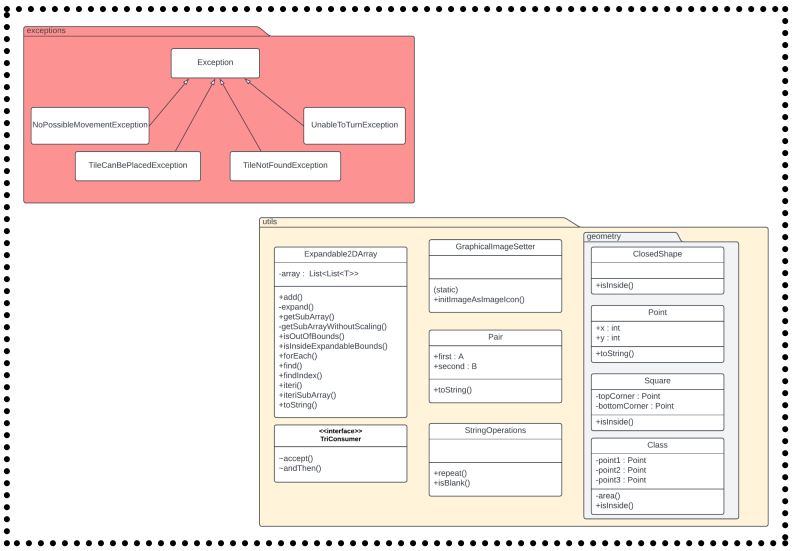
Notation graphique du modèle de notre projet



Notation graphique de la vue Graphique de notre projet



Notation graphique du MVC de la version Terminal pour Domino



Notation graphique des autres classes du projet

Axes de réflexion

Dans cette partie, nous allons approfondir la manière dont nous avons implémenté certaines fonctionnalités. Il s'agit d'éléments qui nous ont fait réfléchir à leur modélisation, avec parfois une première implémentation puis une modification pour avoir un code plus efficace par exemple.

Concernant le modèle

- Modélisation des Sides

Pour les dominos, un côté d'une tuile est représenté comme un tableau final de trois entiers, et pour Carcassonne, un côté est d'un type parmi un enum.

- Modélisation des tuiles

Pour modéliser les différentes zones dans les tuiles de Carcassonne, nous avons eu plusieurs idées. L'une était de créer 9 zones par tuile qui seraient de différents types en fonction du dessin. Chaque zone pourrait être une ville, un chemin, un champ ou une abbaye. Finalement, nous avons décidé de ne pas utiliser cette approche et de n' utiliser que les côtés (aussi des champs, villes ou chemins) et un booléen pour savoir si la tuile contient une abbaye.

- Génération des tuiles

Elles sont toutes générées à la création du Deck, et non quand on les tire. Au début l'id des tuiles était initialisé selon leur ordre de création, puis nous avons changé pour que ce soit quand elles sont piochées.

- Génération du deck (Domino)

Pour générer le deck on utilise l'algorithme suivant :

1. On génère une première tuile et on l'ajoute à la liste qui représente le deck.
2. Tant que l'on a encore des tuiles à générer, on fait les opérations suivantes :

a. On choisit un nombre aléatoire entre 0 et 3 qui représente le nombre de tuiles que l'on va générer pendant cette itération.
b. On choisit un nombre aléatoire entre 0 et 3 qui représente le côté qui va être la copie de la dernière tuile générée.
c. On génère une nouvelle tuile qui a un côté égal à la dernière tuile générée.
d. On génère le reste des tuiles à générer dans cette itération autour de la tuile générée, dans des côtés aléatoires. On fait en sorte d'éviter de générer des tuiles du même côté.

Nous souhaitions créer un algorithme pour la génération des tuiles afin d'améliorer les chances de pouvoir placer une tuile (plutôt que de générer les figures au hasard), tout en permettant d'avoir certaines tuiles qui ne peuvent pas être placées. Nous nous sommes rendu compte en testant à grande échelle que, plus une partie est longue, moins la méthode de génération des tuiles est importante, car le nombre de côtés disponibles augmente. C'est pour cela qu'on a cherché à maximiser l'expérience au début de la partie.

Cette méthode permet d'avoir un début assez bon, sans presque devoir se défausser dans les premiers tours. En effet, comme l'algorithme évite seulement la superposition des tuiles générées à chaque itération, il n'évite pas la superposition entre tuiles qui ont été générées à des itérations différentes. Ainsi, même si les chances de pouvoir placer la tuile sont élevées au début, elles ne sont pas de 100 % ce qui explique le fait de se retrouver parfois obligés de passer son tour.

- Bots
    - Domino

Ici il y a un système de points bien précis. Pour trouver l'emplacement idéal, le bot parcourt la liste de tous les emplacements possibles et cherche celui qui lui rapporterait le plus de points. Une fois trouvé, il place la tuile.

    - Carcassonne

Comme il n'y a pas de système de points, il n'y a pas une manière optimale de jouer. A cause de cela, le bot se limite à choisir aléatoirement un emplacement possible pour la tuile à poser et, s'il peut placer un pion, il le fait avec un 50% de chances.

- Le nombre maximal de joueurs

Dans Carcassonne la limite de joueurs provient des normes du jeu. Cependant, pour Domino, les normes sont plus libres. Même si en principe il pourrait ne pas y avoir de limite de joueurs, nous avons décidé de le limiter à 6. Cela a plusieurs bénéfices:

1. C'est une limite raisonnable dans le sens ou un nombre supérieur de joueurs n'améliorent pas l'expérience de jeu (trop de temps d'attente).
2. Cela permet de ne pas devoir gérer dans l'interface graphique des classements avec un nombre très élevé de joueurs, ce qui compliquerait la méthode actuelle d'affichage. A part cette limitation graphique, le reste du code pourrait permettre un nombre arbitraire de joueurs.

- Pass

Dans les deux jeux, le joueur a l'option de passer son tour s'il ne peut pas placer sa tuile.

Dans Domino, le joueur est aussi libre de passer son tour, même s'il peut placer sa tuile. Nous avons choisi de permettre de passer à volonté car cela peut être difficile de

trouver un endroit où placer la tuile. Le fait de pouvoir ignorer cette tuile et attendre le prochain tour est parfois une décision qui améliore la qualité du jeu. Elle peut éviter de bloquer la partie à un point mort où le joueur ne parvient pas à trouver l'emplacement assez rapidement.

En revanche, dans Carcassonne, les règles indiquent explicitement que toute tuile piochée doit être placée si cela est possible, donc l'option de passer à volonté n'a pas été envisagée.

- Placement des tuiles

En mode terminal, le joueur rentre les coordonnées du plateau ou bien une direction et un numéro d'une tuile déjà posée. En mode graphique, c'est avec un clic sur l'emplacement que l'on place la tuile. Il peut être intéressant de noter que nous avons remarqué plus tard qu'en tournant une tuile, les nombres sur les côtés gauche et droit des tuiles de domino s'inversaient. Ce bug a bien été résolu par la suite.

Notre idée initiale était d'utiliser une structure avec des tuiles liées. De cette manière chaque tuile connaissait ses 4 voisins et pour les afficher, on faisait un parcours récursif de tous les voisins de la tuile pour afficher celles qui étaient à une distance de 2 tuiles ou moins de la tuile du centre. Nous avons trouvé que cette méthode rendait difficile le placement des tuiles, surtout dans l'interface graphique. C'est pour cela que nous avons décidé d'utiliser une autre approche.

- Tableau redimensionnable

Nous avons alors décidé d'utiliser un tableau redimensionnable. Pour cela, Yago a créé une classe appelée **Expandable2DArray**. Cette classe est générique et représente un tableau redimensionnable à 2 dimensions, d'où le '2D'. Ce tableau modifie sa taille si on essaye de rajouter un élément aux bords, c'est-à-dire aux indices (-1) et (longueur)/(largeur) du tableau. Dans ce cas, il rajoute une ligne ou une colonne d'éléments nuls.

Ce tableau est utilisé comme plateau de jeu, ce qui permet de placer les tuiles où l'on veut (selon les règles du jeu) sans contraintes de taille du tableau. Pour afficher le plateau on utilise un minimap. Ce minimap montre un carré qui contient 25 tuiles. Ce carré est centré sur la dernière tuile placée et on peut se déplacer dans le tableau grâce à lui. On peut modifier la tuile sur laquelle le tableau est centré, en affichant ainsi un nouveau sous-tableau du plateau.

Pour obtenir ce minimap, on se sert de la fonction dans **Expandable2DArray** appelée **getSubArray(int x, int y, int size)** qui renvoie le sous tableau centré en **(x,y)** et de taille **size.** Ainsi, il suffit d'appeler cette fonction avec la position de la tuile qui est au centre du minimap pour l'obtenir.

- Modélisation des pions (Carcassonne)

Comme on implémente un jeu de carcassonne sans points, on peut simplifier la modélisation des pions sans compromettre non plus une possible expansion dans le futur. Grâce à la modélisation que nous avons choisie, on peut savoir où le pion est placé. Si les côtes sont simples (ville ou champ) alors tout placement dans cette zone est modélisé comme un placement sur ce côté en particulier. Par contre, dans un côté complexe (chemin), on doit aussi savoir si le pion se trouve sur le chemin, à gauche ou à droite.

Pour stocker la position des pions on utilise alors une liste de tableaux de booléens. Un booléen est vrai si et seulement si un pion se trouve à cette position. Chaque tableau a

une taille différente en fonction du type de côté (simple ou complexe). Dans un côté simple le tableau a une taille de 1, et dans un côté complexe, une taille de 3. La première position correspond à la gauche du chemin vue depuis le centre de la tuile, la deuxième au chemin lui-même et la dernière à la droite du chemin, vue aussi depuis le centre de la tuile. Enfin, si la tuile contient une abbaye, alors la liste est de taille 5 et le dernier tableau de booléens en contient un seul, qui correspond à l'abbaye.

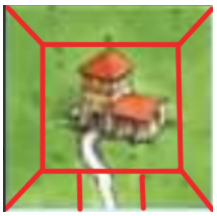
- Placement des pions

Le placement de pions sur les tuiles de Carcassonne dans l'interface graphique se fait grâce au concept des bounding boxes. On divise la tuile en différentes zones en fonction des côtés qu'elle a, et si la souris clique sur l'une de ces zones, alors on sait où placer le pion dans la tuile.

La division est simple. On commence par diviser le carré (la tuile) en quatre triangles, en suivant les diagonales :

Ici chaque triangle représente une zone différente dans laquelle on peut placer un pion.
Pour le cas des chemins, on divise la zone du côté en trois. La division se fait comme suit:

Ainsi, on peut distinguer de quel côté du chemin le pion a été placé.
Le dernier cas est celui de l'abbaye. Dans ce cas on dessine un carré au milieu qui correspond plus ou moins avec la forme des deux abbayes et on génère les autres zones après. Voici la représentation graphique:



Pour savoir si un élément appartient à la bounding box on utilise des classes qui représentent des objets géométriques clos et on cherche à voir si le point donné se trouve à l'intérieur de la forme. Ces classes se trouvent dans le package **utils.geometry**.

Pour générer les bounding boxes (des instances des classes de ce package) on utilise la classe **BoundingBoxHandler** (dans *carcassonne/view*).

Pour représenter les pions visuellement, on a décidé de les mettre à des endroits prédéterminés. Ces endroits dépendent du type de côte et de s'il s'agit d'une abbaye ou non. Il sont trouvés par la classe **PawnTilePositionHandler** (dans *carcassonne/view*).

- Tests

Tout au long du développement, nous avons testé notre code. Pour le faire, nous écrivions les tests dans le main de la classe à tester. L'ensemble du code a été testé par nous deux, en particulier pour l'interface graphique. Nous avons également réalisé plusieurs tests de compatibilité de notre code avec les machines de l'université. Nous avons fait le choix de supprimer les tests avant le rendu final afin d'avoir un code plus propre.

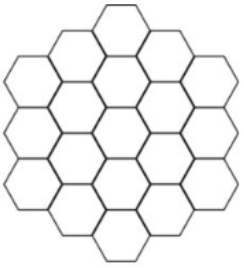
Cependant, même si nous avons bien testé le code, nous n'avons pas implémenté un environnement de test, ni utilisé des frameworks de test comme Junit.

Fonctionnalités non-implémentées

En plus du bot intelligent manquant au jeu de Carcassonne, comme expliqué plus tôt, nous avons réfléchi aux changements que notre code nécessiterait pour l'implémentation d'un jeu Hexa-Carcassonne. Nous exposons ici les pistes résultantes.

Pour implémenter le jeu Hexa-Carcassonne nous devrions modifier la hiérarchie des tuiles. Actuellement, l'interface **Placeable** ne permet pas l'utilisation de tuiles avec un nombre de côtés différent de 4. Ainsi, on devrait éviter de l'implémenter par **Tile** et plutôt le faire au niveau des classes filles de **Tile**. Pour apporter un peu plus d'ordre, on pourrait aussi créer une classe abstraite, héritant de **Tile,** qui représente une tuile avec quatre côtés et qui implémente **Placeable**. Les classes **TileDomino** et **TileCarcassonne** étendraient cette nouvelle classe. On peut imaginer une classe **TileHexaCarcassonne** qui étend **Tile** mais n'implémente pas **Placeable**. De cette façon, on conserverait l'abstraction des classes tout en permettant au reste du programme de fonctionner.

Les méthodes à implémenter dans les tuiles de Hexa-Carcassonne sont les mêmes que celles qu'il y a déjà dans la classe **TileCarcassonne.** On pourrait créer une interface qui représente une tuile de Carcassonne et qui ne dépend pas du nombre de côtes de la tuile. Ainsi la modélisation de la tuile serait faite. Pour le reste du code il ne faudrait presque rien modifier. En revanche, il faudrait modifier la méthode **getSubArray** de la classe **Expandable2DArray**, car la géométrie des tuiles hexagonales est différentes des tuiles rectangulaires. Pour cela on peut créer une nouvelle classe qui étend **Expandable2DArray** et qui apporte les modifications nécessaires afin de pouvoir obtenir un sous tableau d'une forme semblable à celle -ci :



Pour la vue, la tâche est un peu plus compliquée. Il faudrait chercher un moyen de dessiner des hexagones grâce à swing et une fois trouvé, il faudrait aussi définir la classe

interne **Board** de GamePanel afin de permettre de créer un minimap de taille différente à 25 et de la forme voulue.

Conclusion

L'objectif de ce projet a été atteint : nous avons implémenté le cahier des charges minimal qui était demandé et avons ajouté quelques fonctionnalités en plus. Pour cela, nous avons utilisé l'ensemble des notions vues au cours du semestre de POOIG. Avec davantage de temps, nous aurions pu implémenter d'autres fonctionnalités parmi celles proposées, en remaniant certaines parties de notre code.

Ce projet a été enrichissant car il a permis un entraînement supplémentaire aux TD et TP qui ont introduit des nouvelles notions. Nous avons pu les utiliser dans un contexte de projet de programmation complet et pu voir en quoi ces notions peuvent nous aider à résoudre des problèmes concrets de modélisation. Il nous a aussi donné une expérience de conduite de projet avec des objectifs à atteindre sous pression. En effet, étant tous les deux en double-licence, il a pu être difficile de consacrer du temps à l'avancée du projet durant certaines périodes chargées par les autres projets ou examens que nous avions. Nous avons pu trouver une fenêtre de travail en fin de semestre pour s'y consacrer entièrement, ce qui nous a demandé beaucoup d'efforts mais nous a apporté beaucoup.