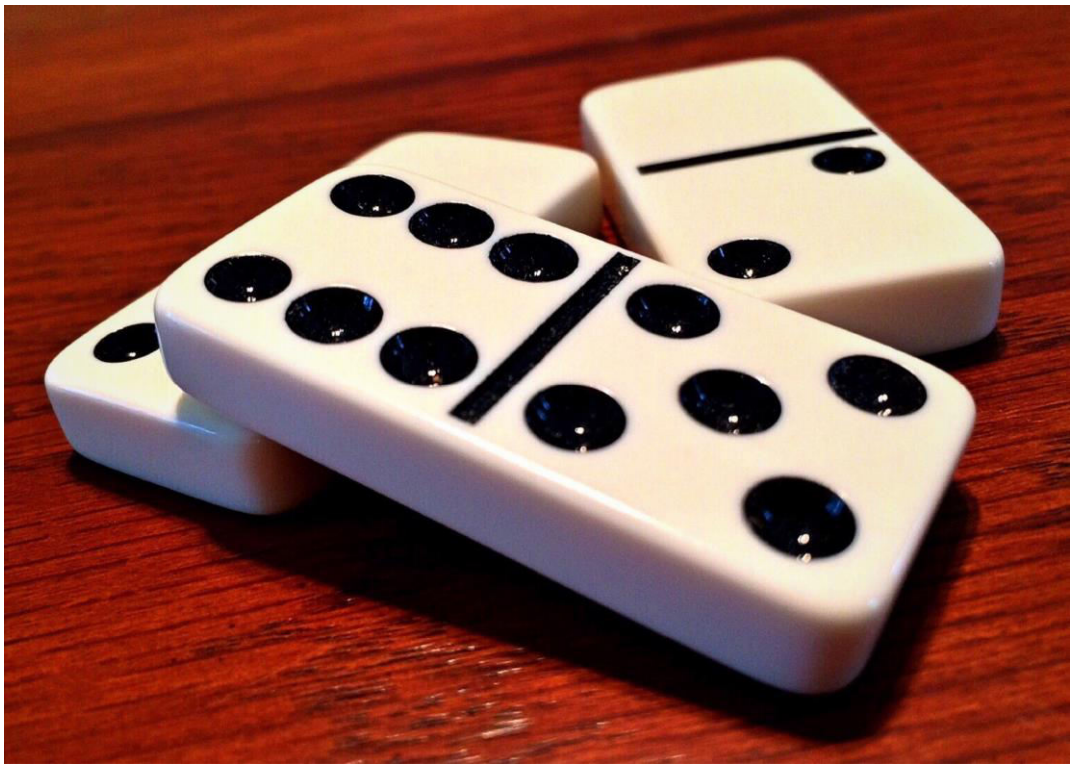


Rapport de projet

POOIG SEMESTRE 3

AGOSTINHO DA SILVA Ninoh, RALPH José | 4 Janvier 2019

Table des matières

0) Introduction

Nous rapporterons ici les stratégies adoptées pour la réalisation du projet, sans revenir ni sur les règles de fonctionnement du programme dans sa globalité ni sur celles propres à chaque jeu, qui seront détaillées dans le fichier README.

Dans un premier temps, nous expliquerons l'architecture imaginée pour répondre au problème, puis nous l'illustrerons.

Nous justifierons ensuite certains partis-pris sur la modélisation de chaque jeu, ce qui nous aidera à déterminer en quoi notre travail répond au problème posé.

1) Généricité et Héritage

1.1) VISION DE L'ARCHITECTURE

Pour l'architecture de notre code, nous avons combiné la généricité et l'héritage autant que possible. Les différentes classes « Jeu* » partagent toutes une classe parent abstraite et générique « Jeu<J j, P p> ». Le fait que cette classe soit abstraite permet à la fois d'empêcher toute instanciation d'un objet jeu non spécifique, et de laisser certaines méthodes sans corps (comme « ajouter (P p) » et…), car elles différent même pour des jeux aussi ressemblants que les dominos et domino-gommettes. Nous avons opté pour une classe abstraite au lieu d'une interface car nous avons choisi besoin de déclarer les variables communes à tous les jeux uniquement dans la classe parent Jeu.

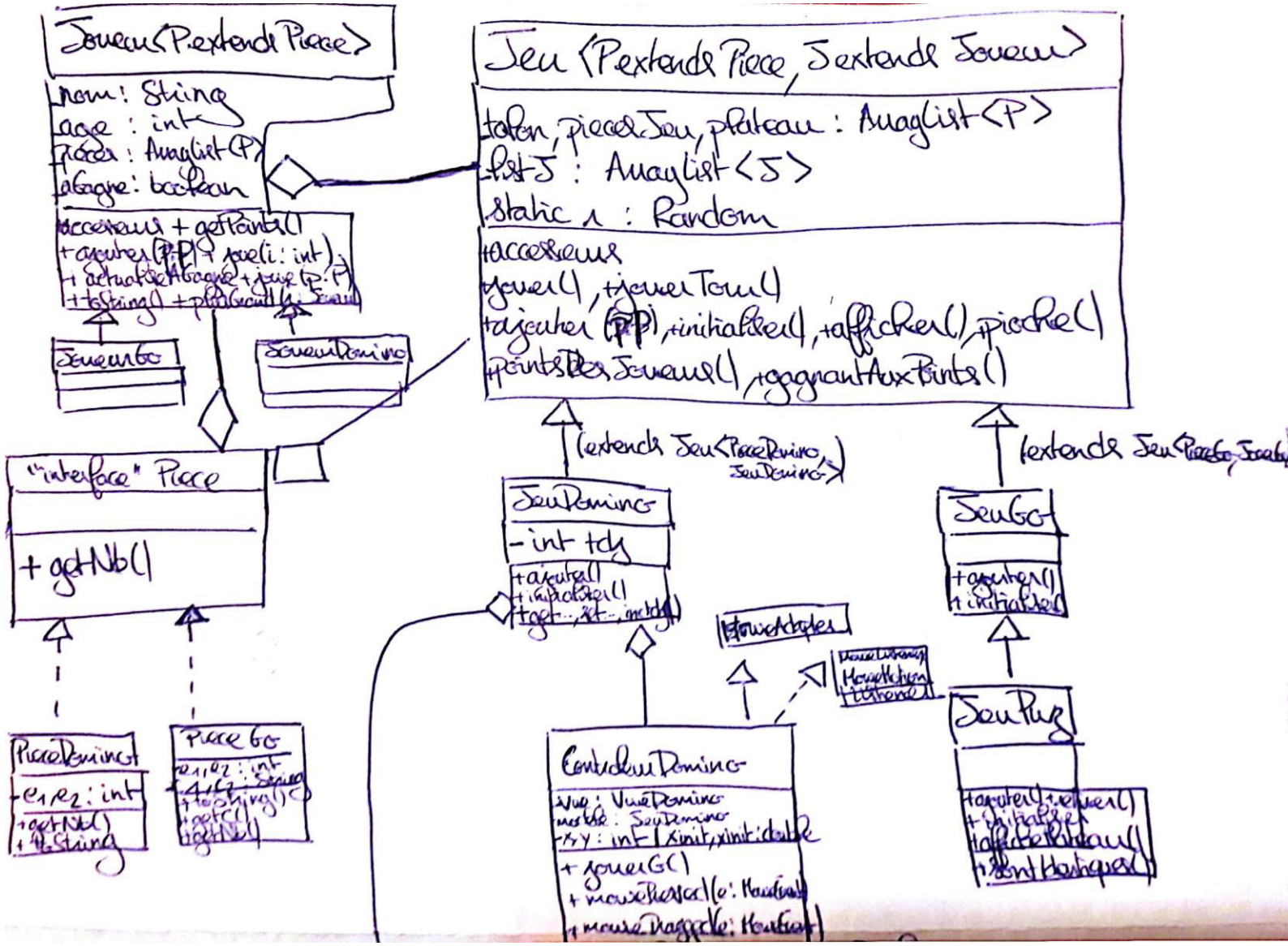
En effet, chaque jeu utilise de manière générale un plateau, modifié par un ou plusieurs joueurs, qui se répartissent une liste de pièces prédéfinie et non extensible, et cette structure nous évite de répéter les déclarations de variable, la majorité des instructions du constructeur, et certaines méthodes dans chaque classe fille de Jeu.

Cependant, le choix de centraliser les variables dans la classe Jeu abstraite complexifie la syntaxe pour l'accès aux variables, qui sont, sauf exception, au moins « private » voire « private final ». Bien qu'il soit possible d'accéder aux champs de la super-classe depuis une classe fille en utilisant l'instruction « super.nomdevariable », nous avons à chaque fois utilisé les accesseurs et mutateurs définis dans la classe mère, pour que le code ait plus de sens à la lecture, et aussi car ces accesseurs et mutateurs seraient nécessaires dans d'autres cas.

Nous avons procédé de la même façon pour la classe abstraite « Joueur <P p> » et l'interface « Piece », bien que cette dernière se soit révélée peu utile dans le code. L'idée d'une classe « PieceDominoDouble » qui étendait « Piece Domino » a été quant à elle abandonnée car, comme mentionné dans le fichier README, nous avons établi que l'âge des joueurs définirait l'ordre de jeu dans tous les jeux avec plusieurs joueurs. Par conséquent nous n'avions plus de raison de distinguer les pièces doubles des classiques, sauf dans la vue graphique du domino, sur laquelle nous reviendrons.

La suppression de cette classe est donc un des exemples d'adaptation de notre code au fil des besoins.

1.2) REPRESENTATION GRAPHIQUE DE LA STRUCTURE

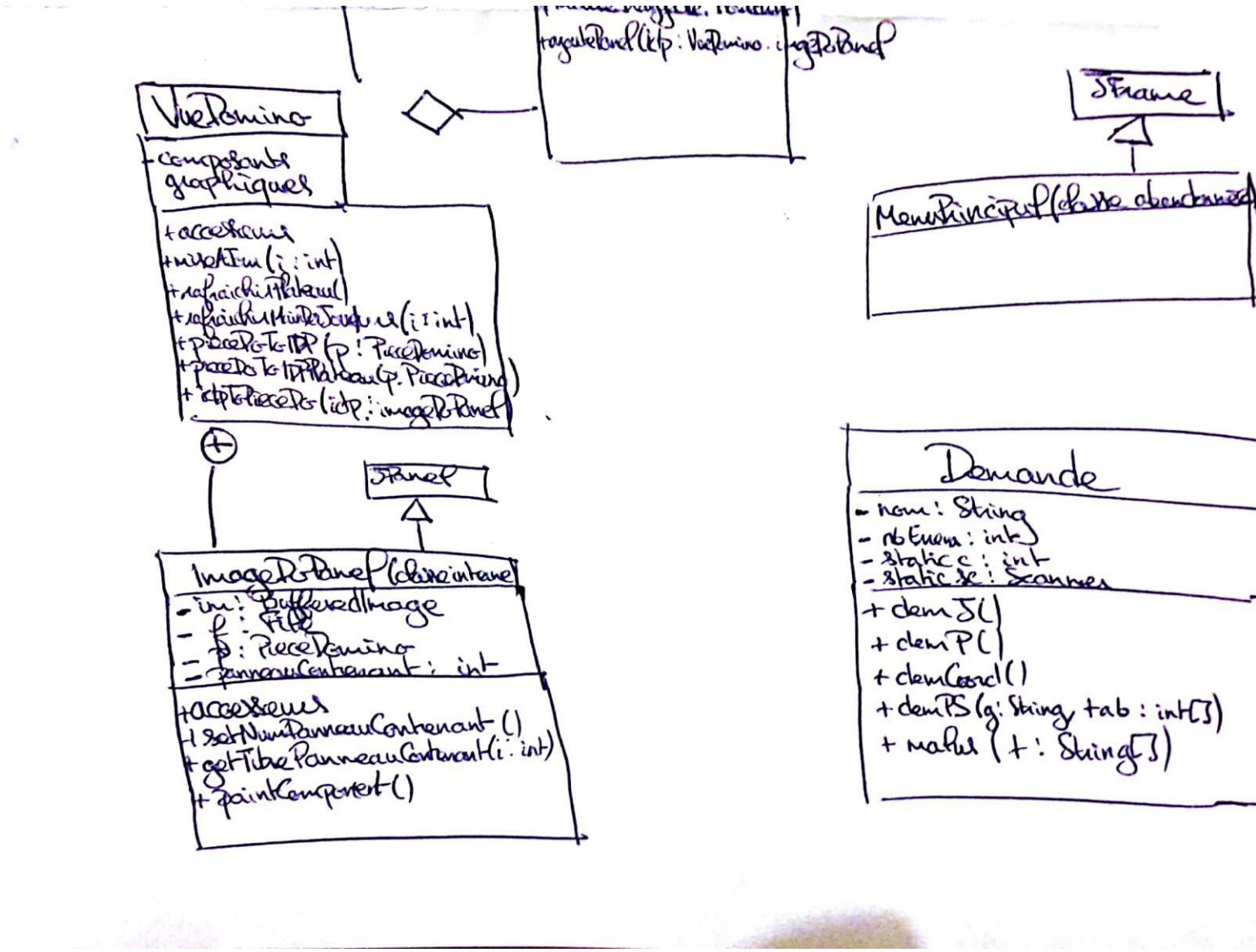


Partie 1 :

Nous avons utilisé autant que possible les règles officielles de représentation UML pour ce diagramme (relations d'héritage, d'implémentation, de dépendance, modificateurs d'accès des variables et méthodes).

Bien que ce ne soit pas forcément identifiable, certains noms de classe et noms de méthode sont écrits en italique pour marquer leur caractère abstrait).

Partie 2 :


VueDomino
- composants graphiques
JFrame
Demande
- nom : String
- nbEnum : int
- static c : int
- static sc : Scanner
+ demJ()
+ demP()
+ demCoord()
+ demPS(g: String, tab : int[])
+ maMet(t : String[])
JPanel
- im : BufferedImage
- f : File
- p : PieceDomino
- panneauContenant : int
+ accesseurs
+ setNumPanneauContenant()
+ getTitrePanneauContenant(i : int)
+ paintComponent()

2) Jeux

2.1) JEUX DE DOMINOS ET DOMINO-GOMMETTES

2.1.1) Stratégie générale choisie pour les dominos

Les jeux de dominos et de dominos-gommettes ne présentent que quelques différences au niveau de la représentation. Pour modéliser un jeu de domino-gommettes à partir d'un jeu de dominos, il suffit plus ou moins d'associer à chaque chiffre un symbole et de rajouter une information sur la couleur à chaque symbole.

Pour cette raison, les classes « JeuGo », « PieceGo » et « JoueurGo » étendent respectivement « JeuDomino », « PieceDomino » et « Joueur Domino », et ne comportent qu'un nombre limité d'ajouts par rapport à leurs classes-parent.

Dans les 3 représentations que nous avons fourni pour les dominos (vue graphique des dominos, vue textuelle des dominos, vue textuelle des domino-gommettes), chaque élément de stockage de pièces (main de chaque joueur, talon, plateau, liste de toutes les pièces du jeu) est une ArrayList, et ceci pour 2 raisons. La première est que Java ne permet pas la création de tableaux génériques, obligeant un auteur du code insistant à effectuer des « cast » considérés « unsafe », or nous avons expliqué en 1.1) que la généricité était une des composantes de notre architecture. La deuxième raison est que les mains des joueurs avaient besoin d'être extensibles, et que cela évitait de créer un tableau de PieceDomino à 2 dimensions disproportionné pour éviter d'en sortir.

Une des conséquences de ce choix est un affichage linéaire pour le plateau dans les 3 représentations. De plus, nous avons décidé, en nous inspirant de certains jeux mobiles de dominos, de placer automatiquement la pièce choisie par joueur dont c'est le tour (selon un ordre de conditions prédéfini) si son choix est correct, et de passer automatiquement le tour du joueur qui se trompe en le faisant piocher.

Ces mesures permettent de fluidifier le jeu, en comparaison avec le puzzle (dont nous parlerons plus tard) où nous ne les avons pas établies pour des raisons évidentes de préservation de l'intérêt du puzzle, et de simplifier la vérification de la validité de la pièce.

Les versions textuelles des jeux auraient pu être encore simplifiées pour le joueur, notamment en rajoutant les indices en dessous des pièces dans les affichages, surtout pour le jeu du puzzle.

2.1.2) Particularités des représentations textuelles

Comme nous l'avons déjà dit, le déroulement dans notre programme d'une partie de dominos et d'une partie de domino-gommettes est quasiment identique. Après avoir rentré les informations nécessaires à l'initialisation du jeu, le joueur le plus grand (ou bien le premier joueur dont on a rentré les informations s'il y a une égalité entre plusieurs âges) commence. Il peut placer n'importe laquelle de ses pièces puisque le plateau est vide. Puis les tours s'enchaînent avec les contraintes mentionnées plus haut. (Voir le README pour plus de détails concernant les règles et le déroulement).

La partie est gagnée lorsque la main d'un des joueurs est vide. Les représentations textuelles du jeu de dominos et de domino-gommettes diffèrent cependant de la représentation graphique des dominos en un point. En effet, lorsque le talon (la pioche) est vide, la structure des versions textuelles permet de laisser tourner le jeu pour un tour de plus (ou autant de tours qu'on veut en changeant une valeur) pour gérer le cas exceptionnel ou la partie pourrait encore se terminer.

Et si le jeu ne se résout pas dans ce dernier tour, les versions textuelles diffèrent aussi par le calcul final du gagnant aux points. Le joueur qui a le moins de points cumulés sur tous ses dominos restants gagne (un autre parti-pris parmi les multiples variantes des dominos).

2.1.3) Contraintes de la vue graphique du jeu de dominos

Le jeu de dominos classique est celui que nous avons choisi pour illustrer la séparation entre la vue et le modèle. Nous n'avons pas réalisé la structure recommandée dans l'énoncé avec le champ « VueGénérale » dans le modèle car la vue graphique ne s'y serait pas prêtée.

En effet, il nous semblait impossible de faire en sorte que le modèle (à savoir la classe « JeuDomino » dans notre cas) demande à sa vue graphique de s'actualiser, car le joueur doit d'abord essayer de déplacer une pièce avec sa souris pour que le modèle puisse décider quoi faire et que la vue s'actualise par la suite.

Pour résoudre ce problème, nous avons opté pour une structure « Modèle-Vue-Contrôleur ». Le « ControleurDomino » possède un champ « VueDomino » et un modèle « JeuDomino » qui est le même que celui de sa vue. Les méthodes de jeu pour le jeu de dominos graphique se trouvent dans le contrôleur, et diffèrent des méthodes de jeu de la version textuelle par l'absence de boucle while. Nous avons rajouté dans la classe « JeuDomino » un champ entier représentant le tour du joueur actuel, un accesseur, un mutateur, et une méthode pour incrémenter ce champ, le faisant revenir à zéro si on se prépare à sortir des limites de la liste de joueur. Ces méthodes seront ensuite appelées dans le contrôleur selon les actions que le joueur effectue, tandis que le contrôleur vérifie constamment ce que renvoie la méthode booléenne « partieGagnee() » de « JeuDomino ». Lorsque cette méthode renvoie « true », cela veut dire qu'un des joueurs a posé toutes ses pièces, et la partie s'arrête abruptement avec un message d'information indiquant le nom du gagnant. La partie s'arrête aussi selon le même procédé si le talon est vide, indiquant que la partie est nulle.

Nous aurions aimé avoir le temps de rajouter certaines fonctionnalités comme un bouton quitter, un bouton rejouer, ou encore la possibilité de rajouter un ou plusieurs joueurs contrôlés par l'ordinateur au cas où le joueur voudrait jouer seul, mais nous n'avons pas réussi à les implémenter de manière satisfaisante.

De plus, nous avons essayé, dans la classe « MenuPrincipal », abandonnée par la suite, de lancer le choix des jeux et d'initialiser complètement la représentation du jeu de dominos via une interface utilisateur, mais cette méthode produisait des fenêtres nulles pour des raisons que nous n'arrivions pas à identifier.

Malgré ces échecs, la version graphique du jeu de dominos fonctionne et permet même à des joueurs honnêtes de jouer en se relayant sur un même ordinateur, car seule la main du joueur dont c'est le tour est visible.

2.2) PUZZLE

Comme on peut le lire dans l'énoncé du projet, on peut facilement envisager un puzzle textuel comme une dérivation d'un jeu de domino-gommettes. Cependant, nous avons dû redéfinir quasiment toutes les méthodes dans la classe « JeuPuz » car les mécaniques de jeu présentent de réelles différences. Notre puzzle utilise des « PieceGo » tant dans le modèle que dans le plateau et dans la partie des pièces directement accessibles au joueur. Contrairement aux versions des dominos, on utilise deux tableaux à 2 dimensions (un pour le modèle et un pour le plateau) dans lesquels l'affichage des pièces supprime les accolades de représentation des pièces, pour se rapprocher du concept d'une image dans un vrai puzzle.

Représenter un puzzle de façon textuelle nécessite tout de même un peu d'imagination. Nous nous sommes demandé s'il fallait considérer la contrainte de devoir placer une pièce exactement à côté de la bonne pièce (comme dans un puzzle à pièces de formes différentes) pour pouvoir accepter le mouvement, ce qui nous a amenés à réfléchir aux listes chaînées. Mais nous avons finalement construit notre puzzle comme les puzzles à pièces carrées ou rectangulaires, qui laissent le joueur placer une pièce n'importe où sur le plateau pour reconstituer l'image.

Nous avons aussi réfléchi aux options que nous laisserions au joueur, et nous avons décidé de laisser uniquement le joueur placer une pièce de sa liste vers le plateau (à un endroit non occupé), ou de retirer une pièce du plateau vers sa liste. Nous avons abandonné la piste du déplacement d'une pièce sur le plateau car devoir rentrer les coordonnées de départ et d'arrivée ralentissait trop le déroulement du jeu déjà très long, avec 28 pièces à placer au total.

D'une manière assez évidente, la partie est gagnée lorsque le plateau et le modèle sont identiques (on utilise la méthode « Arrays.deepEquals() ». Si c n'est pas le cas mais que le joueur a placé les 28 pièces, on ne lui laisse plus que l'option de retirer une pièce.

2.3) SABOTEUR

Le jeu du Saboteur était clairement le plus complexe des quatre à modéliser, et c'est aussi celui dont nous trouvons le plus de choses à redire dans notre programme.

Tout d'abord, nous avons fini le projet par ce jeu et nous n'avons pas été en mesure de l'intégrer à la structure générique du reste du code. On trouve donc plusieurs classes dédiées uniquement au jeu du Saboteur, et de multiples redondances dans ces mêmes classes.

Nous pouvons aussi mentionner un problème relatif à l'orientation des cartes du jeu. Malgré l'enchevêtrement de conditions, chaque carte chemin placée se retourne automatiquement dans l'orientation inverse à celle qu'elle avait dans la main du joueur, compliquant sa tâche.

Enfin, l'affichage textuel de ce jeu est le moins lisible de tous, en fonction de sa nature et des contraintes liés au terminal à l'heure de dessiner des chemins.

3) Conclusion

Nous pouvons dire que la généricité combinée avec l'héritage était une stratégie crédible pour respecter la consigne demandant une structure modulable et la plus compacte possible. Nous déplorons les échecs du menu principal graphique et de la pleine inclusion du Saboteur dans le modèle. Néanmoins, nous pensons sincèrement que le design pattern « MVC » était beaucoup plus fiable pour illustrer la séparation entre le modèle et la vue graphique dans le jeu des dominos et que nous avons répondu au problème concernant les trois premiers jeux, grâce à la réutilisation des variables et de la majorité des méthodes.