Puzzle Edition

Rapport de projet

VINCENT Leo 21805239
BELLEBON Alexandre 21808613
LEROY Clementine 21800424
DEROUIN Auréline 21806986

Table des matières

1 Présentation du projet **1**

A Présentation de l'application 1

2 Arborescence **2**

A Arborescence du projet 2

B Présentation src jeuAssemblages 2

3 Aspects techniques **3**

A Librairie piecesPuzzle 3

B Modèle : Le plateau 3

C Controleur 5

i Controleur : Play 6

D Vue 7

i InterfaceGraphique 7

ii ActionGraphique et MouseClicke 8

4 Conclusion **9**

1 Présentation du projet

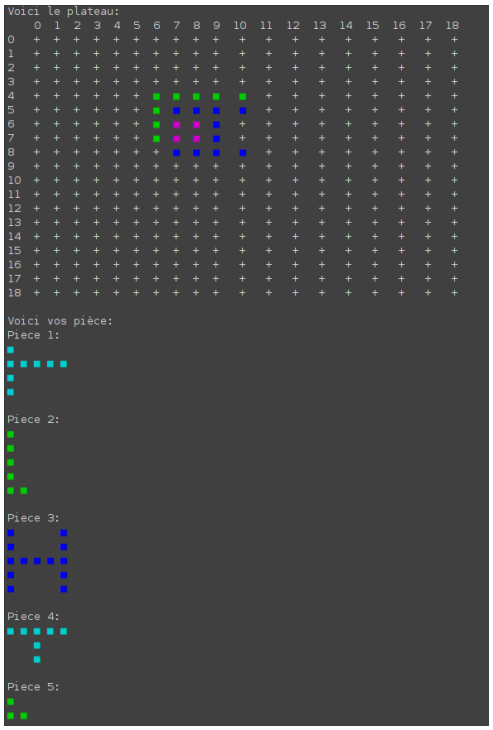
A Présentation de l'application

Puzzle Edition est une application de jeu, dotée d'une interface graphique, qui consiste à assembler des formes de sorte qu'elles occupent le moins de place possible. Cela s'apparente au tetrix, mais avec des modalités différentes. D'ailleurs, le joueur retrouvera certaines pièces de ce jeux. Ces pièces peuvent être placer, déplacer ou supprimer du plateau ou encore tourner.

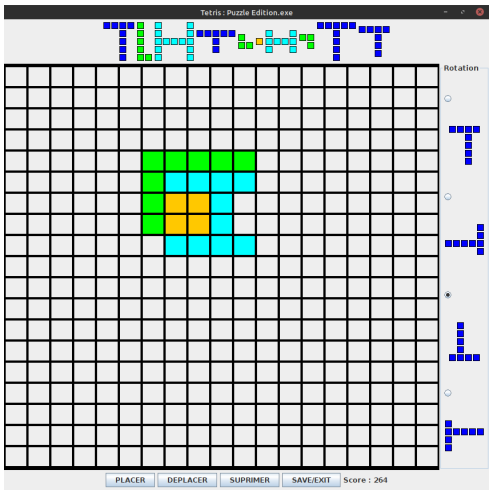
Le joueur peut créer ou charger une partie. La difficulté de la partie est mesurer en fonction de la taille du plateau choisis ainsi que le nombre de pièces générés. En effet, plus il y a de pièces généré, plus la partie est difficile, mais plus le score sera élevé. C'est pour cela qu'à chaque partie, il est possible soit de demander à obtenir une nouvelle configuration de départ, soit charger une configuration déjà créée et sauvegardée.

Le but du joueur est de minimiser l'espace occupé par l'ensemble des pièces. Plus précisément, la fonction d'évaluation sera l'aire du plus petit rectangle (parallèle aux axes) recouvrant l'ensemble des pièces. Lorsque le joueur considère avoir terminé (ou lorsque le nombre maximum d'actions autorisées est atteint), il clique sur un bouton est son score est alors calculé.

Une option permet de faire jouer l'ordinateur sur une nouvelle partie, ou une partie déjà sauvegarder, permettant ainsi de comparer son score avec celui de l'ordinateur.



Vue console



Vue graphique

2 Arborescence

A Arborescence du projet

repertoire : répertoire contenant le rapport et le diagramme de classes
dis : répertoire contenant les fichiers .jar
  gson-2.3.1.jar : Permet de faire fonctionner le Json pour la sauvegarde/changement de partie
  piecesPuzzle-0.1.jar : Librairie créer de pièces
src : répertoire contenant le code du pojet
  jeuAssemblage : Projet contenant le jeu
  PiecePuzzle : Projet contenant le jeu
doc : répertoire contenant le javadoc du projet
build.xml Fichier permettant de compiler et lancer le projet
README.txt Fichier contenant les commandes/A propos

B Présentation src jeuAssemblages

controleur : Classes de controleur
  EnumAction : Enumération des actions
  InterfacePlay : Interface de joueur
  Play : Console global
  PlayIA : Choix ia
  PlayJoueur : Choix joueur pour console
  PlayMenu : Choix de la vue
file : Classes de gestion de fichiers
  ChargerPartie :]Permet de charger une partie
  DeleteFile : Supprime un fichier
  SauvegardeFichier : Sauvegarde d'un fichier
  ScoreFile : Sauvegarde/affichage des scores
    partie : répertoire contenant les fichier de parties
modele : Classes de modèle
  PlateauPuzzle : Plateau en lien avec la librairie PiecesPuzzle
util : répertoire contenant les Patern Listener
  Listenable : Patern listener
  Listener : Patern listener
vue : Affichage graphique
  ActionGraphique : Réalise des actions en fonction des actions du joueurs sur la vue
  InterfaceGraphique : Vue graphique
  MouseClicker : Prend en compte les cliques

3 Aspects techniques

A Librairie piecesPuzzle

Pour la conception de ce projet, une librairie a été créer, réferencant ainsi toutes les pièces possibles. Cette librairie comporte 4 pièces différentes qui ont toutes une grille créer a partir de la rotation défini. Cette création se fait à l'aide de la méthode *createGrid()* (présent dans la classe abstraite **AbstractPiece**) qui prend en paramètre un numéro (de 0 à 3) qui correspond a la rotation dans le sens horaire. Cette méthode change la largeur/longueur de la pièce, en prenant en compte la rotation ainsi que la largeur/-longueur donnée à la pièce dans le constructeur. Ainsi, cela permet la création d'une nouvelle grille en prenant en compte la rotation. En effet, si la pièce tourne, sa largeur et longueur sont inverser, puis les valeurs sont utiliser dans la méthode *pieceGrid()* de la pièce afin de créer une nouvelle grille.

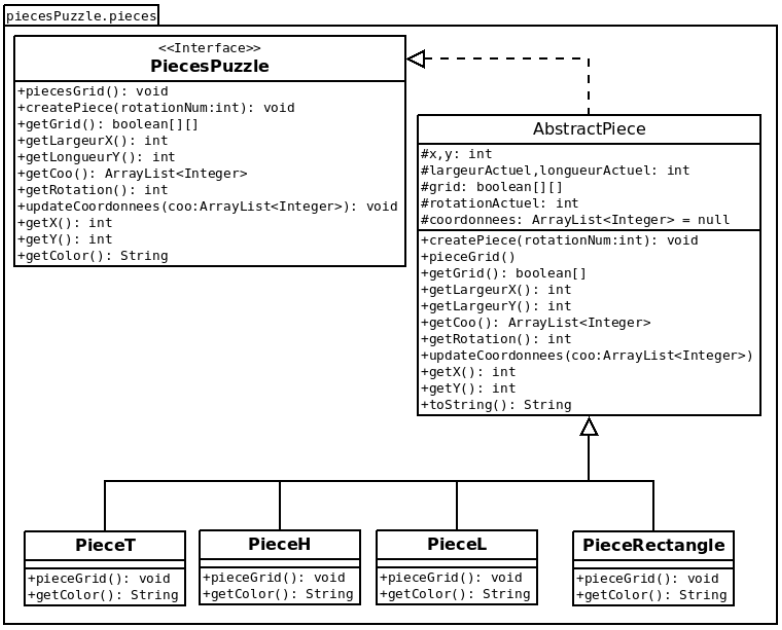


FIGURE 3.1 – Librairie piecesPuzzle

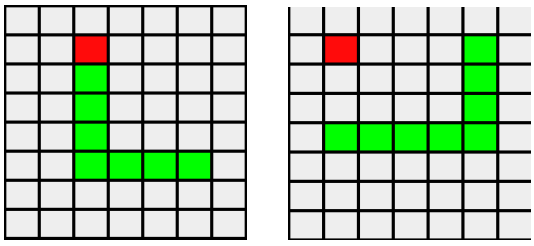
Chaque pièces a une grille qui est composé de valeur boolean, permettant ainsi de créer la pièce voulu (la valeur *true* permet de "dessiner" la pièce) en fonction de sa rotation, largeur, longeur. Elle est aussi composée de coordonnées, permettant, en fonction du jeu, de pouvoir la situé dans sur le plateau. Mais ces coordonnées n'ont qu'une valeur (exemple : 2,3), ce qui est une case de référence, et non les coordonnées au complet.

B Modèle : Le plateau

La librairie piecesPuzzle est utilisée par notre modèle. La classe **PlateauPuzzle** permet de créer un plateau ainsi que d'utiliser les pièces pour les ajouter, supprimer, déplacer, ou encore, tourner. Pour la création du plateau, une HashMap est utilisée, comprenant des coordonnées (ArrayList de *Integer*) en clé et d'une **PiecePuzzle** s'il y en a une, ou "*null*" en valeur. Cela permet d'avoir une lisibilité absolut sur le plateau, ainsi que sur les différentes pièces présentes, sans être obligé d'avoir une Map trié. La HashMap est directement créée dès l'instanciation de la classe.

Pour que le plateau puisse créer, placer, supprimer, déplacer, tourner une pièce, plusieurs méthode sont appelées. Et pour toutes ces méthodes, la plupart appel la méthode *validePlacement()* permettant de savoir si la pièce, reçu par la méthode, peut être placer dans la HashMap. Pour cela, elle parcourt la grille de la pièce grâce à une boucle "for". Dès que la valeur de la grille parcouru est de boolean "*true*", elle regarde si la valeurs de la clé de coordonnées, reçu en pamaètre, du HashMap est libre. Si oui, elle continue la boucle, sinon, elle retourne la valeur booelan "*false*" qui indique que le placement n'est pas possible. Cela permet donc de pouvoir placer des pièces cote a cote, en ne placant que les case "true" de la pièce dans la HashMap.

Les pièces ont donc des cases "*true*" et "*false*" en fonction de comment elles sont créer. Nous avons choisit une case de référence afin d'avoir les coordonnées de la pièce s'il faut, par exemple, la déplacer ou la faire tourner, ou encore, si on sauvegarde la partie. Cette case de référence, se situe en haut à gauche de la pièce. Sauf que cela posait problème si le joueur voulait poser la pièce à l'endroit où il a cliqué/indiqué les coordonnées (voir image**??**). En effet, la case est toujours en haut a gauche, même si cette case est de valeur boolean "*false*".



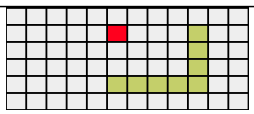
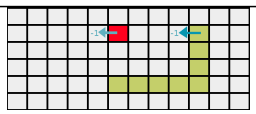
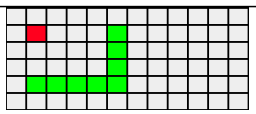
Légende :
Rouge : coordonnées où le joueur veut placer la pièce, et indique aussi la case de référence
Vert : pièce L en rotation 0 (img 1) et rotation 3 (img 2)

Pour régler ce problème, si la méthode *ajoutPiece())* ou *movePiece()* est appelé, les coordonnées ("*coo*") reçu en paramètre (clique ou indication de coordonnées en console) sont modifiés. Pour cela, une boucle while est utiliser : tant que les coordonnées Y de la grille de la pièce est de boolean "*false*", les coordonnées ("*coo*") en position Y est diminué de 1, permettant ainsi, si la prochaine case de la grille est de boolean "*true*", de placer la grille de la pièce a partir des coordonnées reçu en paramètre moins 1 .

Algorithm 1: Boucle while

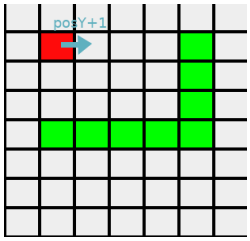
```
1 posY ← 0;
2 xx ← 0;
3 yy ← 0;
4 while !piece.getGrid()[xx][yy + posY] do
5 |   coo.set(1, coo.get(1) − 1);
6 |   posY ← posY + 1;
7 end
```

Cela permet de changer les coordonnées indiquées par le joueur, jusqu'à ce que la valeur des coordonnées de la grille de la pièce est égal a la valeur boolean "*true*".

		
Coordonnées pris en compte	Boucle while	Application de l'ajout/déplacement de la pièce

Légende :
Rouge : Case de référence et variable "*coo*" qui, sur l'image 1, sont les coordonnées renseignées par le joueur (modifier par la suite dans la while)
Flèche : Paramètre *coo* (coordonnées) modifié dans la while
Vert/jaune : Pièce "fantome" où la pièce devrait être normalment sans cette while

Pendant que la valeur des coordonnées Y de la variable *coo* diminuent, le parcours de la grille de la pièce se fait jusqu'à la case la plus en haut a gauche possible. Donc elle ajoute +1 a la variable *posY*.



Légende :
Rouge : Case de référence
Flèche : vérification de la prochaine case si sa valeur est "*true*"
Vert : pièce L rotation 3

C Controleur

Le controleur est séparé en plusieurs classes :

classe *PlayMenu* : Permet de choisir entre la vue console et la vue graphique.

classe *Play* : Fait appel aux méthodes du modèle ainsi qu'aux méthodes de choix, en fonction de la vue/joueur actuel.

classe *InterfacePlay* : Méthodes communes aux choix des joueur : ia ou joueur console.

classe *PlayJoueur* : Méthodes de choix pour joueur Console, grâce aux Scanner.

classe *PlayIa* : Méthodes de choix pour ia.

enum *EnumAction* : Enumère les actions possibles en jeu.

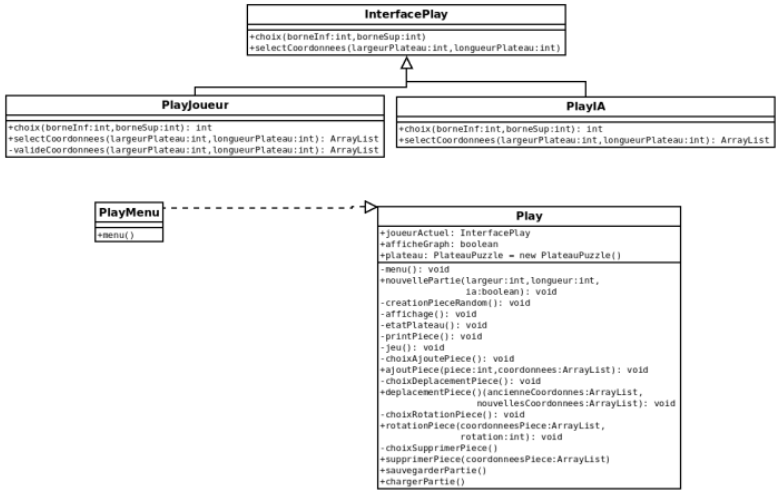


FIGURE 3.2 – Controleur

Cette séparations permet simplifier et séparer les choix de jeu. Cela évite la redondance de code.

i Controleur : Play

En vue console, les choix se font grâce à la variable *joueurActuel* qui est une instance de **InterfacePlay**. Cela permet, en fonction du choix du joueur (option ia ou non), d'appeller les même méthodes de choix, permettant ensuite d'appeler les méthodes communes à toutes les vues pour accomplir l'action prévu (placer, déplacer...). Tout cela est controlé par la méthode *jeu()*.

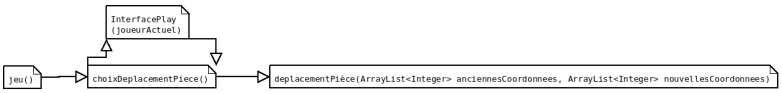


FIGURE 3.3 – Explication choix en fonction du joueur - vue console

En vue graphique, les choix se font grâce à des actions sur des cases/boutons (cf : voiri).

Controleur : Joueur Ia

Une méthode *jeuIa()* est appelé dès que le joueur choisis de regarder l'ia jouer une partie chargé ou une nouvelle partie (en vue graphique ou console). Cette option permet d'avoir une des meilleurs composition de jeu. Cette méthode sauvegarde d'abord la partie actuel dans une sauvegarde propre a l'ia. Puis, une boucle "for" lance des *fourmis* (joueur ia qui joue une partie avec des choix random et guidées) afin de faire un certain nombre de parties. A chaque partie, la partie sauvegardée est chargée. L'ia joue et le meilleurs score et le meilleurs plateau généré est enregistré . Cela permet donc d'avoir plusieurs possibilités aléatoires, et d'essayer de trouver la meilleurs composition possible. Cela évite donc de n'avoir qu'une possibilité qui regarde jusqu'à une certaine profondeur.

La fourmis appel la méthode *jeu()*, qui est une commune entre l'ia et le joueur console. Cette méthode permet de choisir l'action que le joueur va effectuer. Il y a un choix a faire, alors cette méthode appel l'attribut *joueurActuel*, permetant de faire un choix random pour l'ia. Pour éviter d'avoir un simple choix aléatoire où toutes les actions ont le même niveau de piorité, une liste d'actions a été créer avec plusieurs même actions à l'interieur. Car, une action pour "avancer" (placer et déplacer) dans le jeu est plus important que de "rester sur place" (rotation dans le plateau et rotation des pièces a jouer) et est plus important qu'une action qui "recule" l'avancement du jeu (supprimer une pièce). La liste peut donc avoir la même actions plusieurs fois, en fonction de son importance dans l'avancement du jeu. Une fois l'action choisis, d'autres méthodes de choix sont appelées en fonction du choix de l'action.

Pour les actions pour "avancer" dans le jeu, les choix ne sont pas de simple random. En effet, si la fourmis a choisi l'action de PLACER ou DEPLACER une pièce, la méthode *choixDeplacement()* ou *choixAjout()* est appelée. Ces deux méthodes prenent en compte la largeur, la longueur et le plateau. Pour *choixDeplacement()*, l'ia choisit d'abord une pièce au hasard sur le plateau, grâce à la méthode *selectPiece()* (méthode commune de l'**InterfacePLay**) qui sélectionne une pièce dans la liste des pièces placer. Les coordonnées récupéré sont les coordonnées de la case de référence de la pièce (cf : voir **??**). Une boucle while, presque similaire à ici1, est donc nécessaire pour avoir les coordonnées qui aurait pu être cliquées ou indiquées par un véritable joueur. Une fois les coordonnées obtenus, la pièce est supprimé afin d'exercer une simulation de possibilités sans avoir la piècedans le plateau. A partir d'ici, *choixDeplacement()* exerce comme *choixAjout()*. Elle créer une copie du plateau actuel, puis parcours tout le plateau à l'aide des coordonnées placé en clé du HashMap. Si la case est vide et au moins une pièce est situé à 1 ou 2 cases autour de celle ci, alors un test de placement est fait (grâce à la méthode *validePlacement()*B). Si ce test est réussi, un deuxième test est necessaire : faire un ajout de la pièce aux coordonnées et calculer le score grâce à la méthode dans la classe **PlateauPuzzle**). Si le score est supérieur au score du test précédent, une sauvegarde ce celui-ci et des coordonnées est réalisé. Sinon, la boule continue pour les coordonnées suivantes. Le *validePlacement()* est déjà appellé dans la méthode d'ajout

ou de déplacement de la classe **PlateauPuzzle**, mais cela diminue de temps d'éxecution en vérifiant le placement avant d'appeler la méthode.

Pour les autres actions (Rotation, Supprimer), le choix est défini en aléatoires.

Une fois que toutes les fourmis ont finit leurs parties, *afficheIa()* affiche le plateau ayant obtenu le meilleurs score. L'affichage déplace d'abord les pièces déjà placer sur le plateau initiale(si c'est une partie charger), puis place ensuite les pièces a jouer en fonction du plateau obtenu par les fourmis. Pour cela, elle utilise les méthodes communes aux différents controleur, pour effectuer des actions.

Controleur : Joueur Graphique

Si la vue graphique est choisie, les instances de **InterfaceGraphique**, **MouseClicker** et **ActionGraphique** sont créés a partir du constructeur de la classe **Play**, permettant ainsi d'afficher la "Vue". Ensuite, la méthode *menuGraph()* est appeler, permettant d'appeler *jeuIa()* ou *jeuVue* en fonction de ce que choisis l'utilisateur. Le choix se fait grâce aux différents boutons. Le joueur peut donc sélectionner la taille de son plateau (via le bouton ligne et colonne qui vont de 5 à 20), charger une partie, afficher le tableau des scores ou encore les règles du jeu. Le fonctionnement des boutons sont assez simple. Le controleur attend que **ActionGraphique** le notifie, permettant ainsi de récuperer le choix du joueur (via la méthode *getChoix()*) et ordonne à la vue ce qu'elle doit faire (afficher la grille, les scores, demander au joueur de selectionné une pièce,...).

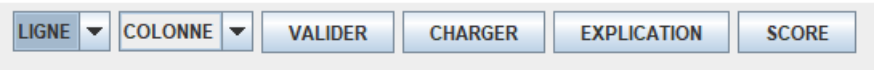


FIGURE 3.4 – Menu principal

D Vue

La vue est composé de 3 classes :

classe *InterfaceGraphique* : Construit et affiche la vue graphique

classe *ActionGraphique* : Gère les interactions entre le joueur et la vue

classe *MouseClicker* : Permet de récupéré les cliques souris du joueur sur la vue

i InterfaceGraphique

L'interface Graphique est composé de nombreuses méthodes pour afficher la vue. La méthode principale est *buildContentPane()*.Elle permet à la fenêtre, dans un premier temps, d'afficher le menu principale lorsqu'elle n'a pas de modèle. Ce menu est composé de divers bouton (cf : voir **??**). Et quand nous avons un modèle, on affiche la grille en parcourant le plateau et en vérifiant la présence de pièce. Cette grille est une grande fenêtre de couleur noir composé de plusieurs petites fenêtre grise espacé entre-elles, donnant cette impression de grille. On oublie pas d'ajouter à cela les pièces à placer, les boutons de jeu et les rotations des pièces, nous obtenons ceci**??**. Cette classe permet aussi l'affichage des scores, des explications de jeu, des aides, du formulaire pour rentrer son pseudo et des fenêtre de choix.

FIGURE 3.5 – Tableau des scores

ii ActionGraphique et MouseClicke

Ces 2 méthodes sont liés car **MouseClicker** est utilisé quand les méthodes d'interaction d'**ActionGraphique** sont appelés. La tache d'**ActionGraphique** est d'attribuer un rôle à chaque bouton et de gérer l'ordre des évènements quand un joueur veut réaliser une action. Par exemple, le bouton "PLACER" va appeler la méthode *actionBouton()* pour vérifier si aucune autre action est en cours. Si oui, on appelle la méthode *placementPieceVue()* qui va demander au joueur de sélectionner une pièce. Cette méthode va attendre que **MouseClicker** lui envoie la pièce sélectionnée pour ensuite demander au joueur de choisir sa rotation et son emplacement. La méthode va de nouveau attendre que **MouseClicker** lui envoie la case sélectionné. Une fois cette dernière selectionné, les choix sont envoyés au contrôleur dans les méthodes communes d'actions. Pendant les 2 périodes où **ActionGraphique** attend, une vérification est réalisé afin de contrôler que l'utilisateur n'a pas décidé d'annuler son action. Dans le cas contraire, l'attente est arrété grâce au "break" et la vue est réafficher à son état initiale. Ce processus est semblable pour les méthodes *deplacementPieceVue()* et *supprimerPieceVue()*. Cette class gère aussi les **Patern** de listener des pièces et des case de la grille, pour que le joueur ne fasse pas de mauvaise sélection.

4 Conclusion

Ce projet était interessant dans l'ensemble. Nous en somme satisfait, même si l'application peut encore être amélioré, nous avons une vue console et graphique fonctionnelle ainsi qu'un joueur IA un peu plus performant que de simple choix aléatoire. Nous avons respecter, dans l'ensemble, ce qui nous avez été demandé, même si nous n'avons pas implementer de patern observeur/observable, nous avons un MVC correct et respecter.