Analyse détaillée des diagrammes

de classe et de séquence du jeu « Tetris »

Ce document propose une analyse détaillée des diagrammes de classe et de séquence (démontrant les étapes lors du déplacement d'une brique vers le bas), spécialement conçus pour représenter la structure et le fonctionnement du jeu de « Tetris » actuellement en développement, qui sera implémenté en langage C++.

Pour commencer, nous avons opté pour l'utilisation d'une classe 'Square' afin de représenter chaque case individuelle sur le plateau de jeu, ainsi qu'une classe 'GameBoard' pour encapsuler l'ensemble du plateau. Cette approche nous permet de bénéficier d'un code plus modulaire et organisé, facilitant ainsi sa lecture et sa compréhension.

La classe 'Square' agit comme un conteneur, dissimulant les détails internes de chaque case du plateau tels que ses coordonnées et son état d'occupation. Afin de déterminer celui-ci, nous avons opté pour un pointeur de pièce : 'Piece* piece', qui sera initialisé à 'nullptr' et lorsqu'une pièce se situera sur un 'Square', on assignera simplement le pointeur de 'Piece' de ce 'Square' à celle-ci. Cela nous semblait plus efficace que d'utiliser, par exemple, un booléen 'isOccupied' car cela nous donne un accès direct à la pièce qui se situe sur le 'Square' nous facilitant ainsi la tâche lors de la mise à jour de ceux-ci et nous donnant plus d'informations sur la 'Piece' occupant le 'Square'. De manière similaire, la classe 'GameBoard' offre une encapsulation des détails internes de la structure du plateau de jeu, tels que sa taille et son contenu. Cette abstraction permet de manipuler le plateau de jeu à un niveau plus abstrait, en utilisant des méthodes spécifiques pour interagir avec celui-ci, plutôt que de manipuler directement les données sous-jacentes. Vous l'aurez compris, 'Square' sera utilisé comme attribut direct de notre 'GameBoard', c'est d'ailleurs pour cela que nous l'avons relier par un lien de composition à la classe 'GameBoard', car, sans ce contexte la, 'Square' n'aurait pas de raisons d' « exister ».

Nous avons choisi d'utiliser un attribut 'coord' de type `std::pair<int, int>` pour représenter les coordonnées d'une case ('Square') sur le plateau de jeu, plutôt que deux attributs distincts `int x` et `int y`. Cette décision a été prise afin de regrouper les coordonnées x et y en une seule entité, simplifiant ainsi le code et le rendant plus lisible.

Ensuite, nous avons opté pour l'utilisation d'un vecteur de vecteurs de 'Square'(vector<vector<Square>>), pour représenter le plateau de jeu, une décision motivée par plusieurs raisons. Tout d'abord, cette structure de données offre une flexibilité considérable en termes de taille et de forme du plateau. Grâce à cette approche, il est possible de modifier facilement la taille du plateau en ajoutant ou en supprimant des lignes ou des colonnes, ou même en ajustant simplement les dimensions du vecteur externe.

De plus, l'utilisation d'un vecteur de vecteurs simplifie grandement la gestion dynamique de la mémoire. Contrairement à un tableau 2D, où la taille doit être spécifiée à la compilation, un vecteur de vecteurs permet une allocation dynamique de mémoire, ce qui signifie que la taille du plateau peut être ajustée à tout moment pendant l'exécution du programme, offrant ainsi une plus grande adaptabilité.

La classe `Piece` a été conçue pour représenter les pièces de notre jeu, offrant ainsi une abstraction générale réutilisable pour différentes formes de pièces.

L'utilisation d'une classe abstraite pour représenter les pièces de notre jeu offre plusieurs avantages. Tout d'abord, elle permet une encapsulation efficace des fonctionnalités et des comportements communs à toutes les pièces du jeu. Cela rend le code plus modulaire et facilite l'ajout de nouvelles fonctionnalités ou de nouvelles formes de pièces sans perturber la structure existante. De plus, l'utilisation d'une classe abstraite favorise la réutilisation du code. Les fonctionnalités communes à toutes les pièces, telles que les méthodes de déplacement ou de rotation, peuvent être implémentées une seule fois dans la classe abstraite, puis héritées et réutilisées par les sous-classes représentant des formes spécifiques de pièces.

L'attribut 'piece_pos' regroupe les positions des cases formant chaque pièce, ce qui simplifie leur manipulation. En consolidant ces informations en un seul attribut, la classe simplifie la gestion interne de chaque pièce, facilitant ainsi leur utilisation dans le jeu.

La classe 'PiecesBag' génère de manière aléatoire la pièce suivante dans le jeu. Son attribut 'bag' de type 'vector<Piece>' gère les différentes pièces disponibles, facilitant leur accès et leur manipulation. Elle se compose de 2 méthodes principales : 'getNextPiece()', permettant l'accès à la pièce suivante du 'bag' et 'refill()' qui, une fois le 'bag' vide, le remplit et mélange les pièces afin de ne pas avoir de séquences de pièce trop « répétitives ».

Nous avons pris la décision d'encapsuler toutes les informations relatives à l'état actuel (score, pièce courante, niveau courant,...) d'une partie au sein d'une classe nommée 'GameState'. Cette approche contribue à rendre le code plus modulaire et organisé, car toutes les informations pertinentes sur l'état d'une partie sont regroupées en une seule classe, rendant ainsi le code plus facile à comprendre. Cette encapsulation offre une certaine flexibilité, car elle permet d'ajouter facilement de nouvelles informations pertinentes à l'état du jeu. Par exemple, l'ajout d'attributs pour suivre le temps écoulé depuis le début d'une partie ou pour enregistrer le nombre de lignes supprimées pourrait être facilement réalisé avec cette classe.

Le `TetrisModel` représente le « cœur » de notre application, contenant trois attributs essentiels : `board`, `bag`, et `state`. À travers ses différentes méthodes, il gère le déroulement du jeu. La méthode `movePiece(direction :char)` est fondamentale pour déplacer la pièce actuelle. En son sein, plusieurs vérifications sont effectuées :

- La validité du déplacement (via `isValidMove(Piece piece, int row, int col)`).

- La présence de collision (par le biais de `checkCollision(Piece piece)`).

Si ces conditions sont respectées, cette méthode fait appel à `setPiece(Piece piece, int row, int col)` de la classe `GameBoard` pour modifier la position de la pièce courante sur le plateau. D'autres méthodes importantes incluent `rotatePiece(direction : char)`, qui permet de faire tourner la pièce actuelle en utilisant une formule adaptée à chaque forme, nous avons ici opté pour une formule de rotation générale à toutes les pièces afin de réduire les coût en mémoire de cette méthode ; `instantDrop()`, pour placer instantanément la pièce à sa position finale, et `clearLines(vector<int> rows)`, qui supprime les lignes complètes du plateau.

La classe ‘TetrisGame’ agit comme une façade pour ‘TetrisModel’, simplifiant la logique du jeu pour une réutilisabilité et une maintenance aisée. Son encapsulation favorise une conception modulaire, permettant des tests unitaires efficaces pour assurer un fonctionnement fiable du jeu.

Concernant les différents liens de dépendances, associations et héritages, mis à part le lien de composition expliqué plus haut entre ‘Square’ et ‘GameBoard’, nous trouvions ceux-là assez explicites dans le diagramme UML que pour devoir les justifier dans ce PDF étant donné qu’il était déjà assez « volumineux » comme ça.

Diagramme de séquence :

Le déplacement d’une brique vers le bas commence par le déclenchement de la méthode ‘movePiece(down)’ provenant de notre façade ‘TetrisGame’ qui va lui-même déclencher ‘movePiece(down)’ qui se trouve dans la classe ‘TetrisModel’ et la plus grande partie du déroulement se passera dans cette méthode. Cette méthode va d’abord lancer la méthode ‘isValidMove()’ qui a comme rôle de vérifier si le mouvement voulu est autorisé et pour ça il a 2 conditions à vérifier si le mouvement est compris dans le bord du jeu ce qui est vérifié par ‘isInsideBoard()’ qui va retourner un booléen et si il retourne ‘true’ la dernière vérification est effectué en vérifiant si il n’y a pas de brique à l’endroit du déplacement ou à ses alentours qui peut faire une collision avec la pièce actuel cette vérification est réalisé par ‘checkColission()’ qui va aussi retourner un booléen.

En résumé la méthode ‘isValidMove()’ va déclencher 2 méthodes privées qui se trouve dans la même classe pour vérifier si le mouvement est possible pour la pièce courrante (donnée par l’attribut state de type GameState) lorsque ces méthodes vont retourner ‘true’, elle aussi va retourner ‘true’ à la méthode ‘movePiece(down)’ qui permettra à cette méthode de continuer ses actions. MovePiece(down), après avoir reçu l’autorisation du mouvement, va faire appel à la méthode ‘setPiece(piece,row,col)’ se trouvant dans la classe ‘GameBoard’ pour placer la pièce actuelle à l’endroit voulu par l’utilisateur, après avoir placer la pièce, la méthode va vérifier si la ligne est complète et pour cela, elle fait appel à la méthode ‘isLineCompleted()’ qui va retourner un booléen, si elle retourne ‘true’ comme dans notre cas la méthode ‘movePiece(down)’ va faire, encore une fois, appel à une méthode privé se trouvant dans la même classe qui s’appelle ‘clearLines(rows)’ qui a comme but de supprimer la ou les ligne(s) complète(s) et si une ou des ligne(s) étai(en)t supprimée(s), elle retournerait un vecteur de ‘int’ qui sera composé de la ou les lignes supprimée(s).

Nous arrivons à la fin de notre méthode movePiece(down) qui, après avoir potentiellement supprimé une ou des ligne(s), va faire appel à ‘computeScore(clearedLines)’ pour calculer le score gagné après ce mouvement en donnant la ou ligne(s) supprimée(s) en paramètre et va ensuite faire appel à ‘setScore(score)’ qui se trouve dans la classe GameState pour modifier le score du jeu actuel. Après avoir modifier l’état du jeu, on fait appel à ‘getNextPiece()‘ se trouvant dans ‘PiecesBag’ pour recevoir la prochaine pièce du jeu et donc cette méthode va retourner la ‘nextPiece’ qui est de type ‘Piece’. Finalement, cette méthode va modifier une dernière fois l’état du jeu en appelant ‘setCurrentPiece(nextPiece)’ pour remplacer la pièce courante avec la pièce suivante.