# Modélisation des classes métiers

Dans notre ambitieux projet de développement du jeu Tetris, la modélisation occupe une place cruciale pour garantir une expérience de jeu à la fois fluide et captivante. Cette section est dédiée à l'exploration de notre diagramme de classe Tetris, mettant en évidence nos choix de conception et offrant une analyse détaillée du diagramme de séquence qui révèle la dynamique interne lors des déplacements des briques vers le bas.

## La classe " Brick "

La classe Brick occupe une place centrale dans notre conception, étant soigneusement élaborée pour représenter non seulement la forme d'une brique Tetris, mais également son impact concret au sein du plateau de jeu. Cette classe est fondamentale et encapsule des attributs essentiels tels que " shape " de type énumération " TypeShape ", qui offre une abstraction pour gérer les différentes formes de briques, " rotation " de type énumération " Orientation ", facilitant la manipulation et la visualisation de la rotation des briques, et surtout, " position " de type " Position ", déterminant la place précise dans le plateau de jeu.

Une autre alternative serait de représenter chaque type de brique par une classe distincte en utilisant le polymorphisme et l'héritage. Cependant, la hiérarchie des classes pourrait devenir complexe avec l'ajout de nouveaux types de briques, et des modifications dans cette hiérarchie pourraient entraîner des répercussions sur l'ensemble du système. C'est pourquoi nous n’avons pas choisi cette approche.

Notre modélisation offre plusieurs avantages : l'introduction facile de nouvelles configurations de briques en ajoutant simplement des valeurs à l'énumération " TypeShape ", sans altérer la complexité du code existant. De plus, la manipulation d'une position centrale, servant de pivot, plutôt qu'un vecteur de positions pour représenter une brique, présente des avantages significatifs :

- Simplicité du code : L'utilisation d'une position centrale simplifie la logique de manipulation de la brique, rendant le code plus concis et plus facile à comprendre.

- Facilité d'extension : La représentation par un pivot offre une flexibilité d'extension pour intégrer de nouvelles formes de briques sans altérer la structure de base du code.

- Clarté des Opérations : Les opérations de déplacement et de rotation sont simplifiées, car elles sont effectuées par rapport à un point central fixe, améliorant ainsi la lisibilité du code.

- Réduction du coût en ressources : En utilisant une seule position centrale pour représenter la brique, la gestion mémoire est optimisée, réduisant ainsi les coûts en ressources. Cette simplification élimine également le besoin redondant de calculer individuellement les coordonnées de chaque case lors des opérations de déplacement et de rotation, améliorant ainsi l'efficacité globale du système.

Remarque importante sur l'énumération " Orientation " : Bien qu'elle inclue les valeurs UP, RIGHT, DOWN, LEFT, cette énumération ne se limite pas à la simple représentation des rotations des briques. Elle joue également le rôle de représentation d’une direction.

La méthode qui déplace une brique utilise cette énumération en paramètre pour éviter la création d'une classe distincte " Direction " avec les valeurs RIGHT, DOWN, LEFT. De même, pour la méthode de rotation d'une brique, l'énumération Orientation (RIGHT, LEFT) est utilisée plutôt qu'une classe énumérée " Rotation " (CLOCKWISE, COUNTERCLOCKWISE). Si nécessaire lors de l'implémentation, la création d'une énumération " Direction " et " Rotation " sera envisagée.

## La classe " ShapesRotation "

L'intégration de la classe " ShapesRotation " simplifie considérablement la gestion des rotations et des positions des briques. En utilisant le patron de conception " Singleton ", la classe centralise les données complexes liées aux différentes configurations de rotations des briques, en associant chaque forme de brique à une carte détaillant les positions pour chaque orientation de brique. Ainsi, grâce à la méthode " getVectorPositions ", on obtient rapidement les positions de la brique sous forme de vecteur.

Notre choix d'une gestion des rotations " hardcodée " est motivé par la volonté de maintenir la simplicité et la clarté du code. Une gestion dynamique des rotations, basée sur des calculs ou des mécanismes plus complexes, aurait pu introduire une complexité supplémentaire dans la logique du jeu.

En optant pour une approche statique, les configurations de rotations des briques sont directement définies dans le code source, ce qui rend le système plus lisible et facile à comprendre. Ce choix permet également d'éviter des mécanismes de calculs dynamiques qui pourraient rendre la gestion des rotations plus délicate et potentiellement sujette à des erreurs difficiles à diagnostiquer. Ainsi, en privilégiant la simplicité, notre approche vise à faciliter la maintenance du code et à minimiser les risques d'erreurs.

## La classe " Board "

La classe " Board " occupe un rôle central dans notre modèle Tetris, servant de toile de fond où prennent vie les briques. Ses attributs incluent " currentBrick ", représentant la pièce en cours avec une instance de la classe " Brick ", " shapesRotation ", notre instance unique du singleton " ShapesRotation " contenant les configurations de rotations, " boardArea ", un vecteur bidimensionnel représentant la grille de jeu, et " startPosition " déterminant l'emplacement initial de la brique sur le " boardArea. "

Parmi ses méthodes, les interactions principales de l'utilisateur avec le jeu, telles que la translation, la rotation et le " drop " de la brique courante, sont implémentées.
La méthode " isCollision " effectue une vérification pour déterminer s'il y a des collisions entre la brique reçue en paramètre et celles existantes dans la grille du jeu. Pour cela, elle utilise le vecteur des positions de l'instance unique " shapesRotation ", récupéré à l'aide d'une méthode privée.
La méthode " isCurrentBrickFallen " détermine si la brique est immobilisée, et " deletePossibleLines " supprime les lignes réalisées si elles existent.
Enfin, des méthodes privées comme " updateArea " et " removeCurrentBrickOnArea " assurent la cohérence de l'attribut " boardArea " lors des différentes manipulations de la brique courante.

Cette conception favorise une structure modulaire et organisée, simplifiant la gestion des briques et du plateau de jeu et présente plusieurs avantages :

- Placer la brique courante dans la classe " Board " permet une gestion cohérente et centralisée de son état et de ses interactions avec la grille de jeu. Cette approche facilite la coordination des actions de la brique avec les opérations du plateau, simplifiant ainsi la logique de jeu. En ayant la brique courante directement liée à la classe " Board ", les opérations telles que le déplacement, la rotation et la vérification des collisions sont traitées de manière homogène et intégrée, contribuant à une conception plus fluide et compréhensible.

- Le choix du vecteur bidimensionnel pour " boardArea " offre une représentation claire et efficiente de la grille de jeu. Cela simplifie les opérations de mise à jour et de vérification, contribuant ainsi à une manipulation aisée des briques Tetris.

- En intégrant " shapesRotation " directement dans " Board ", les opérations liées aux rotations, comme la vérification des collisions lors d'une rotation, peuvent être effectuées de manière homogène avec d'autres opérations de la grille. Cela assure une cohérence d'action, simplifiant ainsi le flux de contrôle et la compréhension du code.

## La classe " Game "

La classe Game, en tant que modèle central du MVC, agit comme une façade et joue le rôle de sujet observé en utilisant le patron de conception " Observateur/Observé ". Elle est dotée de plusieurs attributs essentiels pour représenter l'état du jeu à un moment donné, notamment l'état actuel du jeu, le sac de briques, le plateau de jeu, le score, le niveau, ainsi que la distance de " drop " nécessaire au calcul du score. Des accesseurs sont également disponibles pour permettre l'accès à ces différents attributs.

Parmi les méthodes de la classe, on retrouve les trois interactions principales de l'utilisateur avec le jeu, implémentées dans la classe " Board ". Mais également, pour assurer la cohérence de l'état du jeu, la classe Game intègre une méthode spécifique " updateGame " qui effectue une vérification après chaque déplacement de la brique courante. Cette méthode détermine si la brique est immobilisée ou si un " drop " a été effectué, et si tel est le cas, elle met à jour le score et le niveau en conséquence, en fonction de l'évolution de la grille de jeu.

En synthèse, notre modélisation Tetris repose sur des choix structurés visant à garantir une expérience de jeu fluide tout en simplifiant le développement et la maintenance du code. L'adoption d'une approche statique pour la gestion des rotations et l'intégration cohérente des éléments centraux tels que les briques, le plateau de jeu et les mécanismes de jeu ont abouti à la création d'un modèle robuste et évolutif. Ces décisions contribuent à la clarté du code, réduisent la complexité et facilitent l'extension du jeu Tetris avec une approche résolument axée sur la simplicité et l'efficacité.

## Diagramme de séquence

Pour illustrer la méthode " moveCurrentBrick(Orientation::DOWN) ", voici le déroulement des instructions, repris du diagramme de séquence :

1. L'utilisateur effectue une action pour déplacer la brique courante vers le bas à travers l'interface utilisateur (vue).

2. La vue transmet cette action au contrôleur.

3. Le contrôleur, informé de l'action de l'utilisateur, appelle la méthode " moveCurrentBrick(Orientation::DOWN) " de la classe Game.

4. À l'intérieur de cette méthode, la classe Game appelle à son tour la méthode " moveCurrentBrick(Orientation::DOWN) " de son attribut " board ".

5. À l'intérieur de la méthode " moveCurrentBrick(Orientation::DOWN) " du " board ", une nouvelle position est créée pour stocker la nouvelle position de la brique courante déplacée d'une case vers le bas du plateau.

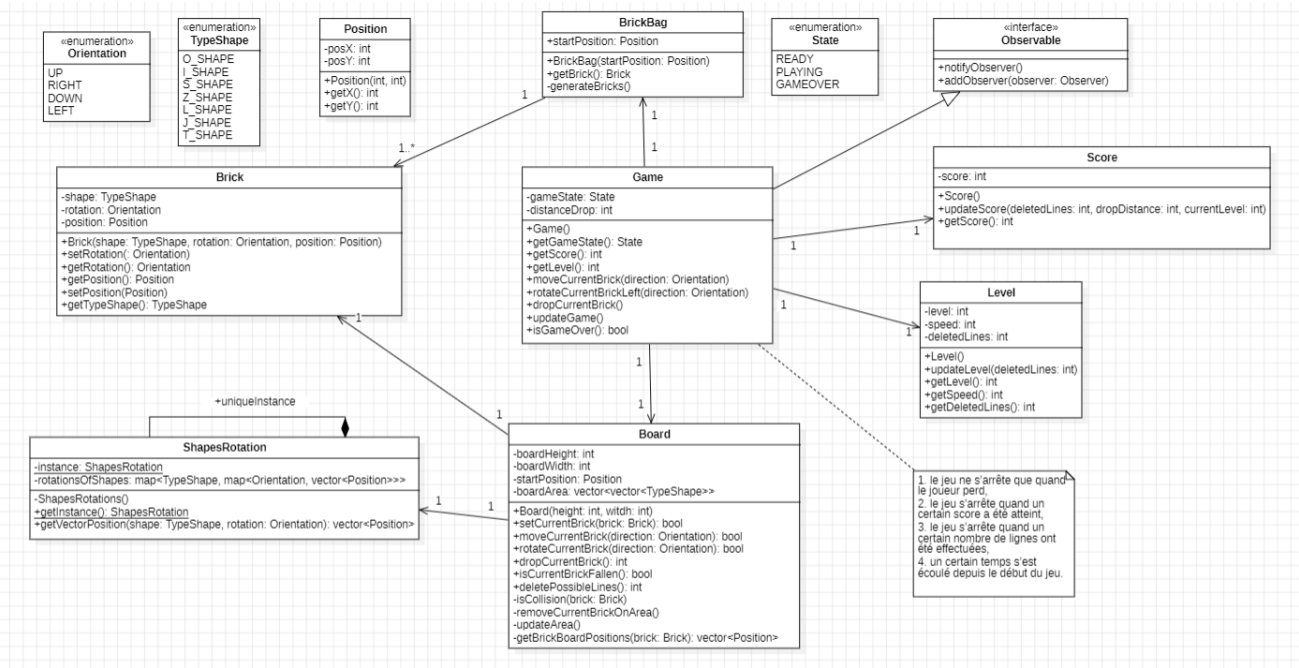
6. Ensuite, une nouvelle brique est créée en attribuant les mêmes caractéristiques que la brique courante, à l'exception de la nouvelle position qui lui est affectée.

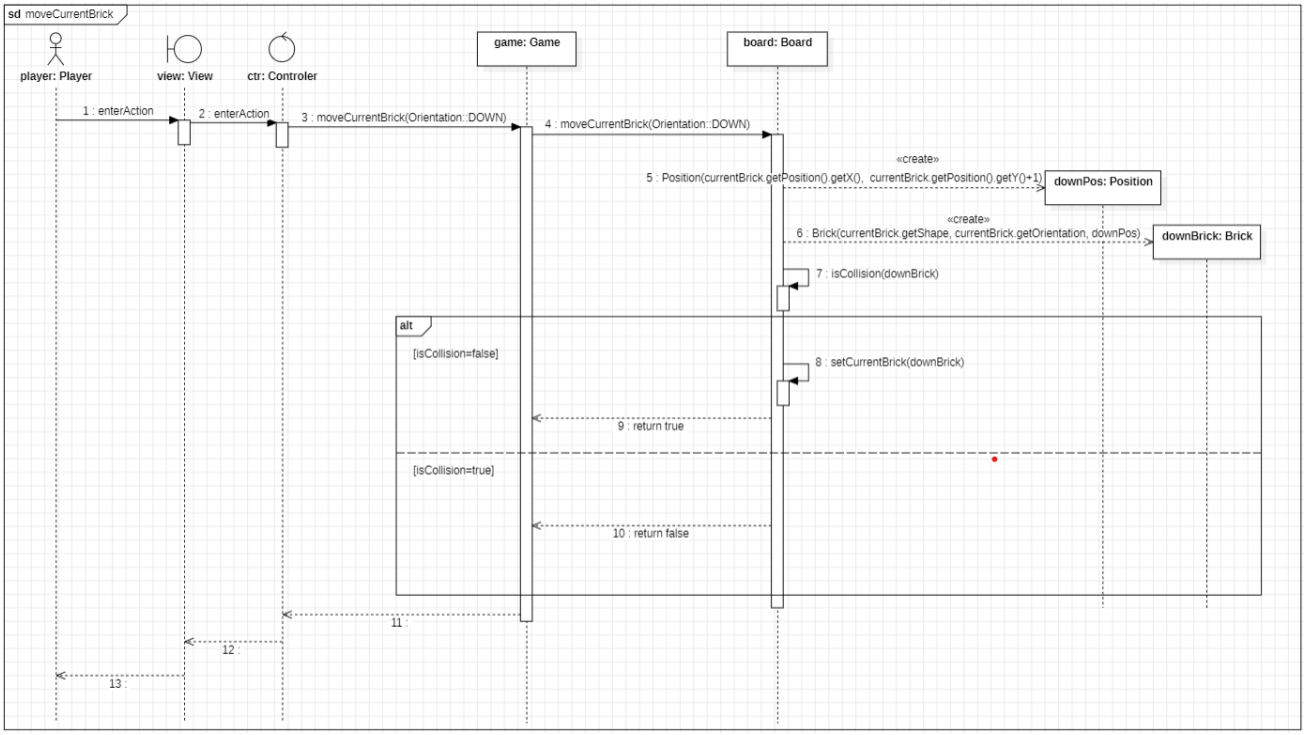
7. La brique déplacée est passée en paramètre à la méthode " isCollision ", qui effectue une vérification pour déterminer s'il y a des collisions entre la brique reçue en paramètre et celles existantes dans le plateau.

8. Si aucune collision n'est détectée, la méthode " isCollision " renvoie faux, et la méthode " setCurrentBrick " est appelée avec la nouvelle brique en paramètre pour mettre à jour la brique courante dans le plateau.

9. La méthode " moveCurrentBrick " de la classe " board " renvoie vrai, indiquant que le déplacement de la brique courante vers le bas a été réussi.

10. Si une collision est détectée, la méthode " isCollision " retourne vrai, et la méthode " moveCurrentBrick " de la classe " board " renvoie faux, indiquant que la brique courante ne peut pas être déplacée vers le bas.


«enumeration»
Orientation
UP
RIGHT
DOWN
LEFT
«enumeration»
TypeShape
O_SHAPE
I_SHAPE
S_SHAPE
Z_SHAPE
L_SHAPE
J_SHAPE
T_SHAPE
Position
-posX: int
-posY: int
+Position(int, int)
+getX(): int
+getY(): int
BrickBag
+startPosition: Position
+BrickBag(startPosition: Position)
+getBrick(): Brick
-generateBricks()
«enumeration»
State
READY
PLAYING
GAMEOVER
«interface»
Observable
+notifyObserver()
+addObserver(observer: Observer)
Brick
-shape: TypeShape
-rotation: Orientation
-position: Position
+Brick(shape: TypeShape, rotation: Orientation, position: Position)
+setRotation(: Orientation)
+getRotation(): Orientation
+getPosition(): Position
+setPosition(Position)
+getTypeShape(): TypeShape
Game
-gameState: State
-distanceDrop: int
+Game()
+getGameState(): State
+getScore(): int
+getLevel(): int
+moveCurrentBrick(direction: Orientation)
+rotateCurrentBrickLeft(direction: Orientation)
+dropCurrentBrick()
+updateGame()
+isGameOver(): bool
Score
-score: int
+Score()
+updateScore(deletedLines: int, dropDistance: int, currentLevel: int)
+getScore(): int
Level
-level: int
-speed: int
-deletedLines: int
+Level()
+updateLevel(deletedLines: int)
+getLevel(): int
+getSpeed(): int
+getDeletedLines(): int
+uniqueInstance
ShapesRotation
-instance: ShapesRotation
-rotationsOfShapes: map<TypeShape, map<Orientation, vector<Position>>>
-ShapesRotations()
+getInstance(): ShapesRotation
+getVectorPosition(shape: TypeShape, rotation: Orientation): vector<Position>
Board
-boardHeight: int
-boardWidth: int
-startPosition: Position
-boardArea: vector<vector<TypeShape>>
+Board(height: int, witdh: int)
+setCurrentBrick(brick: Brick): bool
+moveCurrentBrick(direction: Orientation): bool
+rotateCurrentBrick(direction: Orientation): bool
+dropCurrentBrick(): int
+isCurrentBrickFallen(): bool
+deletePossibleLines(): int
-isCollision(brick: Brick)
-removeCurrentBrickOnArea()
-updateArea()
-getBrickBoardPositions(brick: Brick): vector<Position>
1. le jeu ne s'arrête que quand le joueur perd,
2. le jeu s'arrête quand un certain score a été atteint,
3. le jeu s'arrête quand un certain nombre de lignes ont été effectuées,
4. un certain temps s'est écoulé depuis le début du jeu.


sd moveCurrentBrick
player: Player
view: View
ctr: Controler
game: Game
board: Board
1 : enterAction
2 : enterAction
3 : moveCurrentBrick(Orientation::DOWN)
4 : moveCurrentBrick(Orientation::DOWN)
«create»
5 : Position(currentBrick.getPosition().getX(), currentBrick.getPosition().getY()+1)
downPos: Position
«create»
6 : Brick(currentBrick.getShape, currentBrick.getOrientation, downPos)
downBrick: Brick
7 : isCollision(downBrick)
alt
[isCollision=false]
8 : setCurrentBrick(downBrick)
9 : return true
[isCollision=true]
10 : return false
11 :
12 :
13 :