HEIG VD

Laboratoire 08

Jeu d'échec

Alen Bijelic
Nelson Jeanrenaud
20/01/2021

Table des matières

Introduction

L'objectif de ce laboratoire est de mettre en place la partie logique de la gestion d'un jeu d'échecs. L'interface utilisateur nous est déjà fourni et nous permet de lancer une partie en mode console ou graphique.

Il faut donc implémenter un échiquier, les pièces, leur mouvement et les coups spéciaux en respectant toutes les règles spécifiées dans la consigne. Des points bonus consistent à implémenter l'échec et mat et les matchs nuls par pat ou impossibilité de mater.

Choix d'implémentation

Structure générale

Les classes ont été regroupées dans différents packages de telle manière à avoir un programme séparé de manière logique.

Le package **board** contient toutes les classes concernant la gestion d'un jeu de plâteau. notamment les mouvements, les pièces, l'historique des mouvements. Cela inclut aussi les interfaces permettant de gérer les actions et les conditions du jeu.

Le package **chess** contient les classes s'applique entièrement au jeu d'échecs. La classe Chess permet la gestion du jeu d'échecs et les autres classes implémentent les interfaces d'actions et de conditions pour la gestion des mouvements autorisés des pièces.

Le package **displayChess** permet la gestion entre l'engine et l'interaction utililisateur, notamment pour les mouvements des pièces et la promotion, qui nécessitent une action utilisateur.

L'implémentation de notre solution permet une flexibilité d'amélioration car la gestion du plâteau et le jeu d'échecs sont distincts. Ceci nous permet d'ajouter d'autres jeux de plâteau comme le jeu de dames par exemple. Ou encore d'utiliser notre classe chess avec une autre interface graphique que celle fournie pour le laboratoire.

Un utilisateur de nos packages pourrait aussi créer sa propre variante d'échecs en partant de notre base et rajouter des règles et des pièces uniques.

Généricité

Les classes du packages board sont génériques en <T extends Board>. Un utilisateur du package qui voudrait créer son propre jeu sur plateau aura donc une classe héritant de Board tel que : class Dame extends Board<Dame>

Ce choix d'implémentation nous permet de garantir que pour une instance, les pièces, mouvements et plateau appartiennent au même type de jeu. Et que ce dernier implémente au moins les fonctionnalitées nécessaires en héritant de Board.

Controller

Le fichier Controller fait l'intermédiaire entre l'engine du jeu et l'interface graphique (GUI ou console). Ce fichier implémente ChessController et dispose des fonctions permettant de démarrer une nouvelle partie, d'effectuer une requête pour initier un mouvement d'une case de départ à une case d'arrivée.

Pièce

La classe abstraite Piece fournit les méthodes nécessaires pour créer des pièces et les faire bouger sur un plateau de jeu. Elle contient comme base le plateau de jeu en question et la liste des mouvements que cette pièce peut effectuer.

Dans la classe Chess, il y a la classe interne abstraite ChessPiece qui hérite de Piece. Puis, pour chaque pièce a été créé une classe héritant de ChessPiece permettant de modéliser les mouvements que peut faire chacune des pièces du jeu d'échec.

Mouvements et prises

Le mouvement d'une pièce est initié (en ce qui concerne la partie engine et non pas GUI) dans le controller grâce à la fonction move. Cette fonction prend en paramètre les index X et Y de départ et d'arrivée. La première vérification d'un mouvement est faite à cet instant et permet de vérifier si la position d'arrivée et identique à la position de départ. Si c'est le cas, on retourn false, sinon on passe au move de DisplayChess.

Ce move va simplement créer des Vector à l'aide des indexes et appeler le move de Chess.

Move de Chess permet de faire des vérifications supplémentaires. La première consiste à vérifier si une partie a débuté ou non. Puis, on récupère la pièce censée être sur la case from et on vérifie qu'elle est bien là. Si elle est bien là, on vérifie que c'est au tour de la couleur de la pièce, puis on passe au move de Board, qui va appeler le move de Piece, où l'on va vérifier que le vecteur résultant du mouvement demandé correspond bien à un des vecteurs faisant partie de la liste des mouvements autorisé pour la pièce concerné. Cette vérification se fait

dans canMove, ainsi que d'autres vérifications pour les mouvements spéciaux nécessitant certaines conditions bien précises. Passé cette étape, si le movement peut être réalisé, on vérifie la valeur du booléean doMove, permettant de simuler si un mouvement est possible. Ce booléean à false est surtout utilisé pour vérifier si un emplacement du plateau est attaqué par les pièces adverses.

Si ce booléean vaut true, on appel la fonction doMove de Piece qui va ajouter l'action dans l'historique et appeler doMove de Move, qui lui vérifie les actions dans la liste des actions autorisé comme manger une autre pièce, la promotion pour le pion, etc.. et les appliquer.

Après toutes ces vérifications, on revient au Move de Chess, on change de tour et on vérifie si le joueur adverse est en échec et mat ou pas. Si oui, on termine le jeu.

Une prise consiste à manger une pièce par un mouvement légal d'une pièce adverse. Pour les coups simples, c'est-à-dire un mouvement de base d'une pièce, c'est EatPiece qui s'occupe de gérer cette action. La méthode doAction va retirer la pièce sur la position de destination et bouger la pièce de la position de départ à la position d'arrivée. La méthode RevertAction permet d'annuler cette action et revenir à l'état d'avant l'action. Les pièces mangées lors de coups spéciaux sont décrites ci-dessous.

Coups spéciaux

Lorsqu'un coup spécial va être effectué, il va suivre le même cheminement qu'un mouvement jusqu'à arriver au doMove de Piece, où l'appel à la méthode checkConditions vérifie les différentes actions possibles pour la pièce concernée.

Chaque action modélisée, dispose d'une méthode de conditions (checkConditions), d'une méthode d'action (doAction) et d'une méthode de rétroaction (revertAction).

Ces méthodes sont décrites ci-dessous pour chaque coup spécial.

Petit et grand roque

Conditions

La prise en passant permet à un pion de manger un autre pion en biai se trouvant au même niveau que lui et dont le dernier mouvement était d'avancer de 2 cases.

La première vérification est donc de vérifier que la pièce que le pion essaye de manger est bien un pion. Puis on vérifie que le dernier mouvement a bien été d'avancer de 2 cases (vers le haut pour les blancs et vers le bas pour les noirs).

Actions

L'action consiste simplement à supprimer la pièce victime du En passant.

Promotion

Actions

Une promotion se déclenche lorsqu'un pion avance à l'une des dernières cases. La méthode doAction de Promote fait appel à une méthode getPromotedPiece qui va demander à l'utilisateur, à travers d'un popup pour le mode graphique, en quelle pièce celui-ci veut transformer son pion. Finalement, le pion est remplacé par la pièce choisie par l'utilisateur. La reine est la transformation par défaut.

Vérifications du jeu

Mise en échec

La vérification de la mise en échec se fait depuis Chess grâce à la méthode check qui va vérifier si les deux rois sont en échec en faisant appel à la méthode isAttacked. Cette méthode vérifie tous les angles d'attaque qui pourrait mettre le roi en échec.

La fonction check est notamment redéfinie par la classe DisplayChess pour afficher le message dans l'interface utilisateur.

Echec et mat

Echec et mat correspond à une implémentation bonus. La méthode checkMate de la classe Chess vérifie si le roi est mis à l'échec, ainsi que les mouvements possibles de chaque pièce du joueur sont égaux à 0. Cela signifie qu'aucun mouvement ne peut sauver le roi, ainsi que les autres pièces en bloquant la pièce qui attaque ou en la mangeant.

Historique

La classe Historic est intéressante car elle permet de stocker, sur une stack, l'historique des actions qui se sont déroulés dans la partie. Cette classe est utilisée notamment pour faire de retour en arrière de certaines actions, mais peut être sujette à évoluer et permettre la notation algébrique des mouvements de pièces.

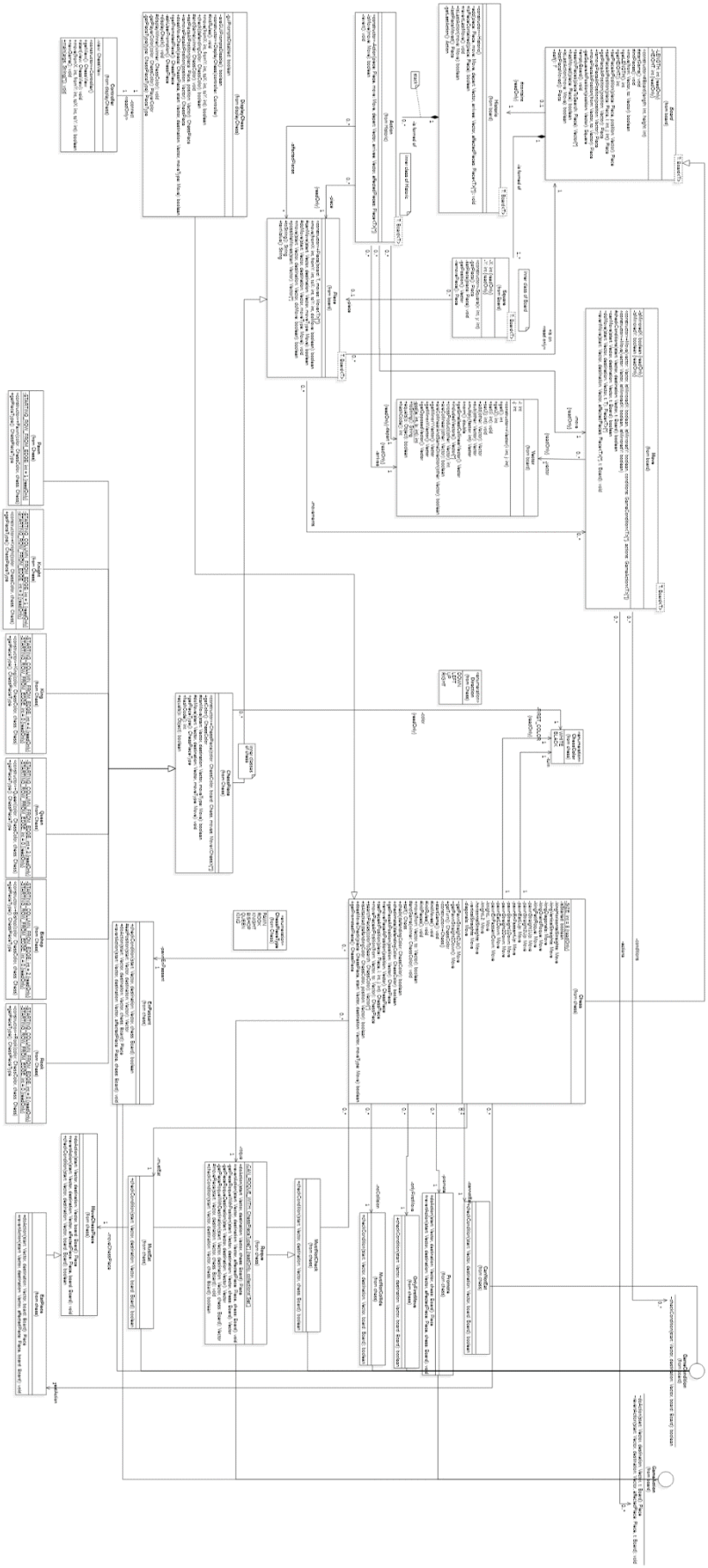
Cette classe implémente une classe interne nommée Action qui contient, une pièce, un mouvement, une position de départ et d'arrivée et une liste de pièces affectées par l'action. Comme toutes les actions sont stockées dans une stack, on peut facilement obtenir la dernière Action ajoutée, celle que l'on modifie le plus dans notre programme.

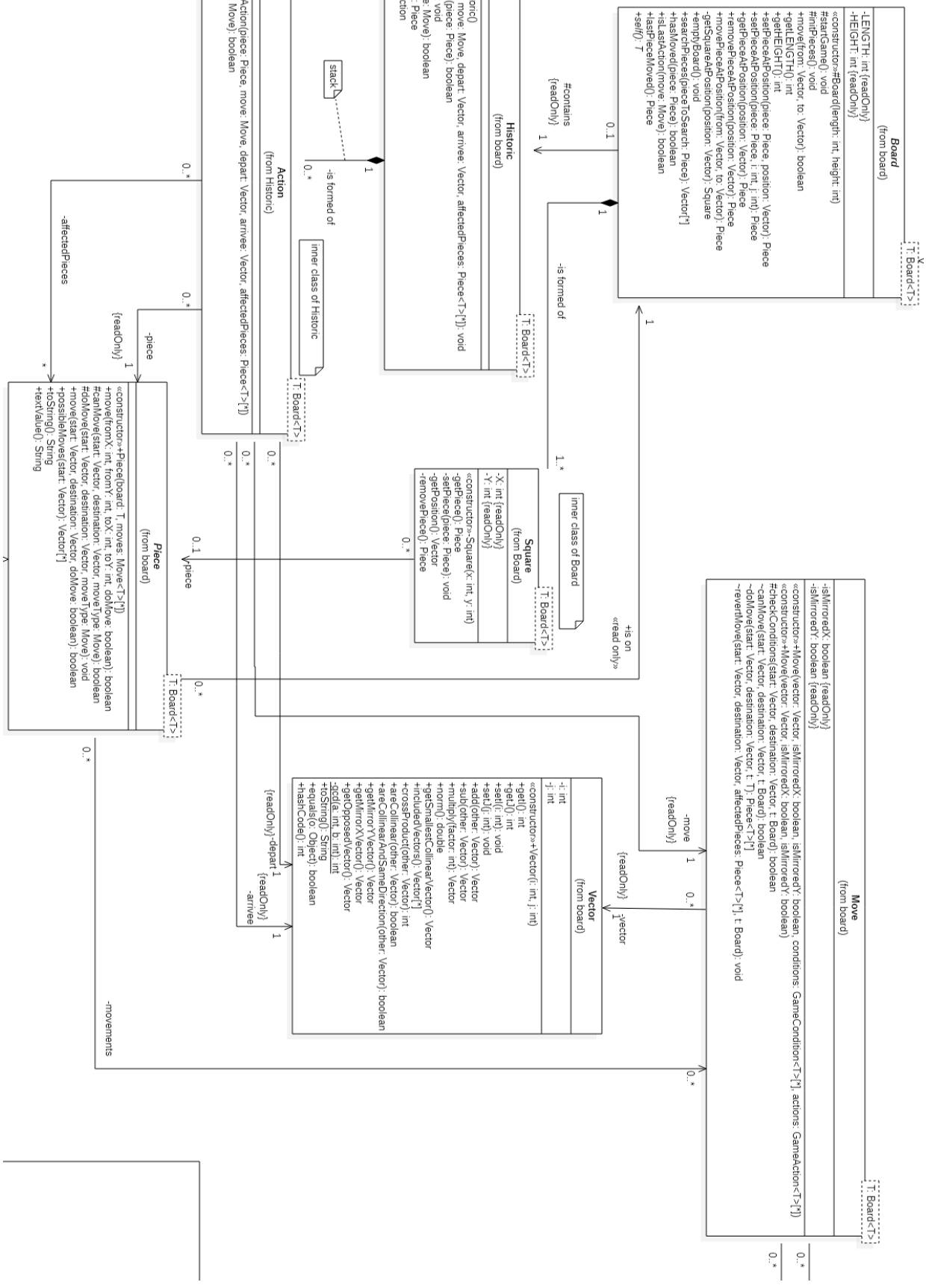
Vecteur

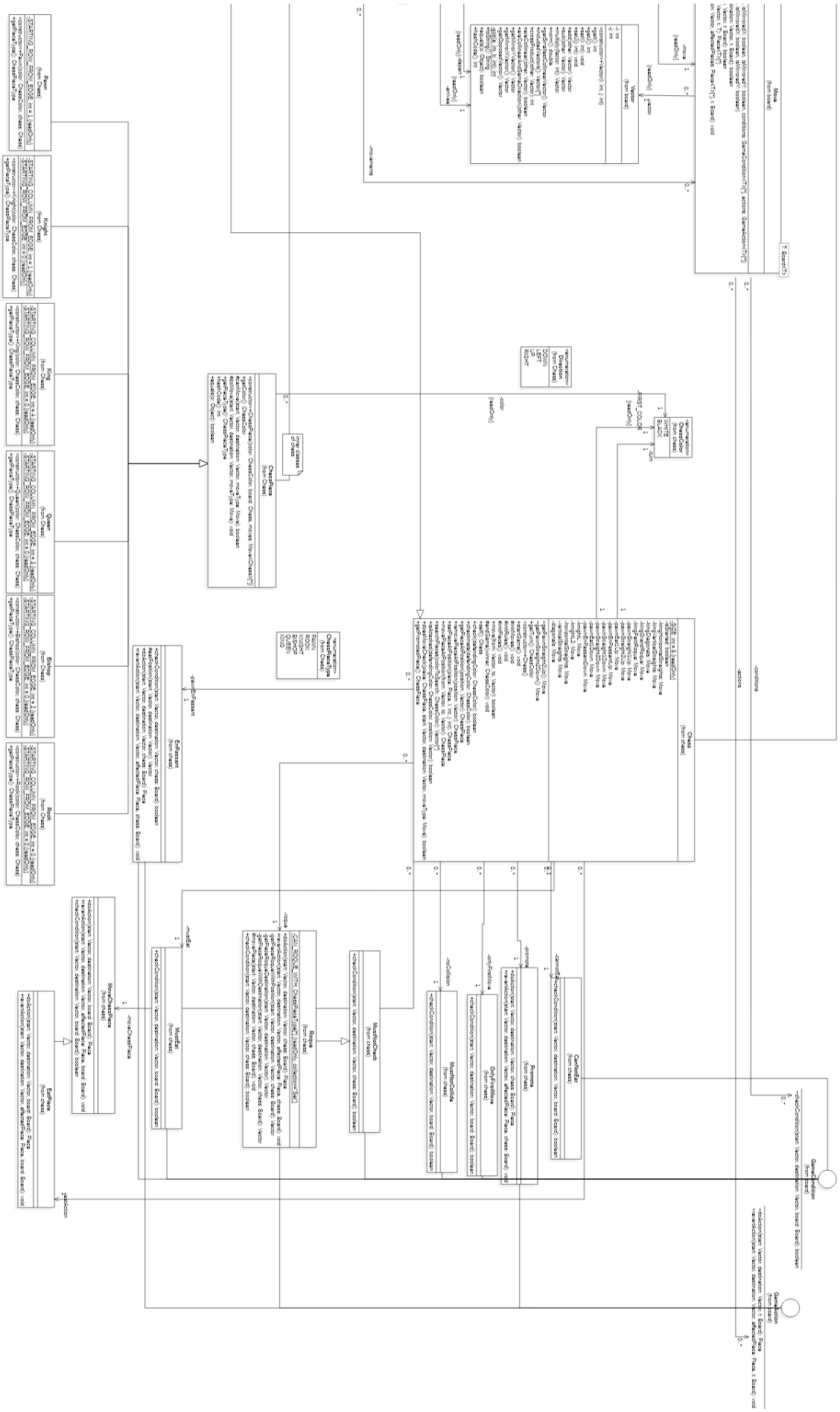
La classe Vector modellise un vecteur qui contient deux index entiers, i et j. Les méthodes disponibles dans cette classe permettent la gestion des positions des pièces ainsi que de leur mouvement. Il est possible de faire presque toutes les manipulations sur les vecteurs au sens mathématique.

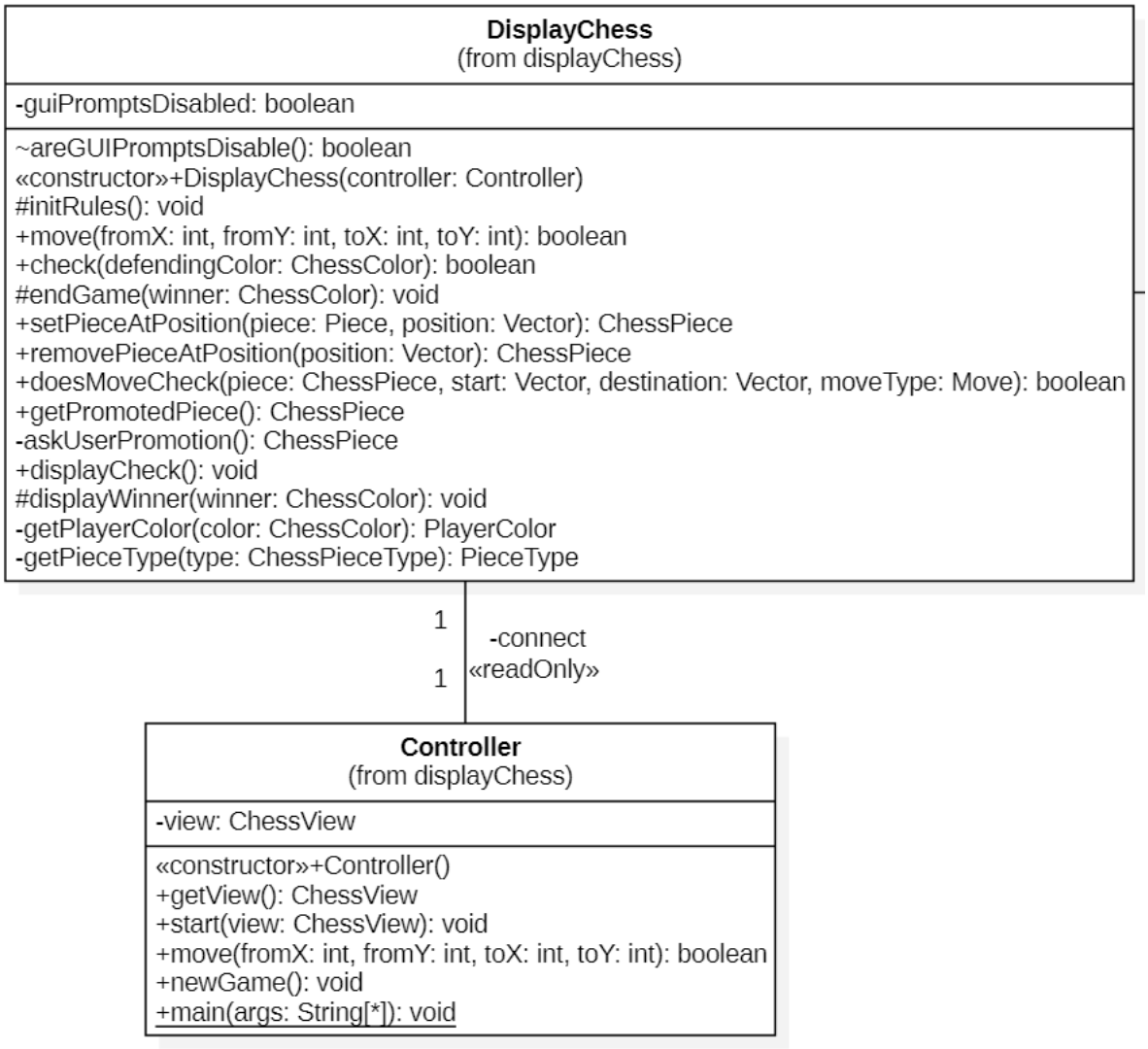
Entre autres, les méthodes add et sub qui permettent d'obtenir un nouveau vecteur en l'ajoutant ou en l'enlevant du vecteur. Il est possible de multiplier un vecteur par un facteur, d'obtenir la norme, de calculer le plus petit vecteur collinéaire, lister tous les vecteurs inclus dans celui-ci, etc…

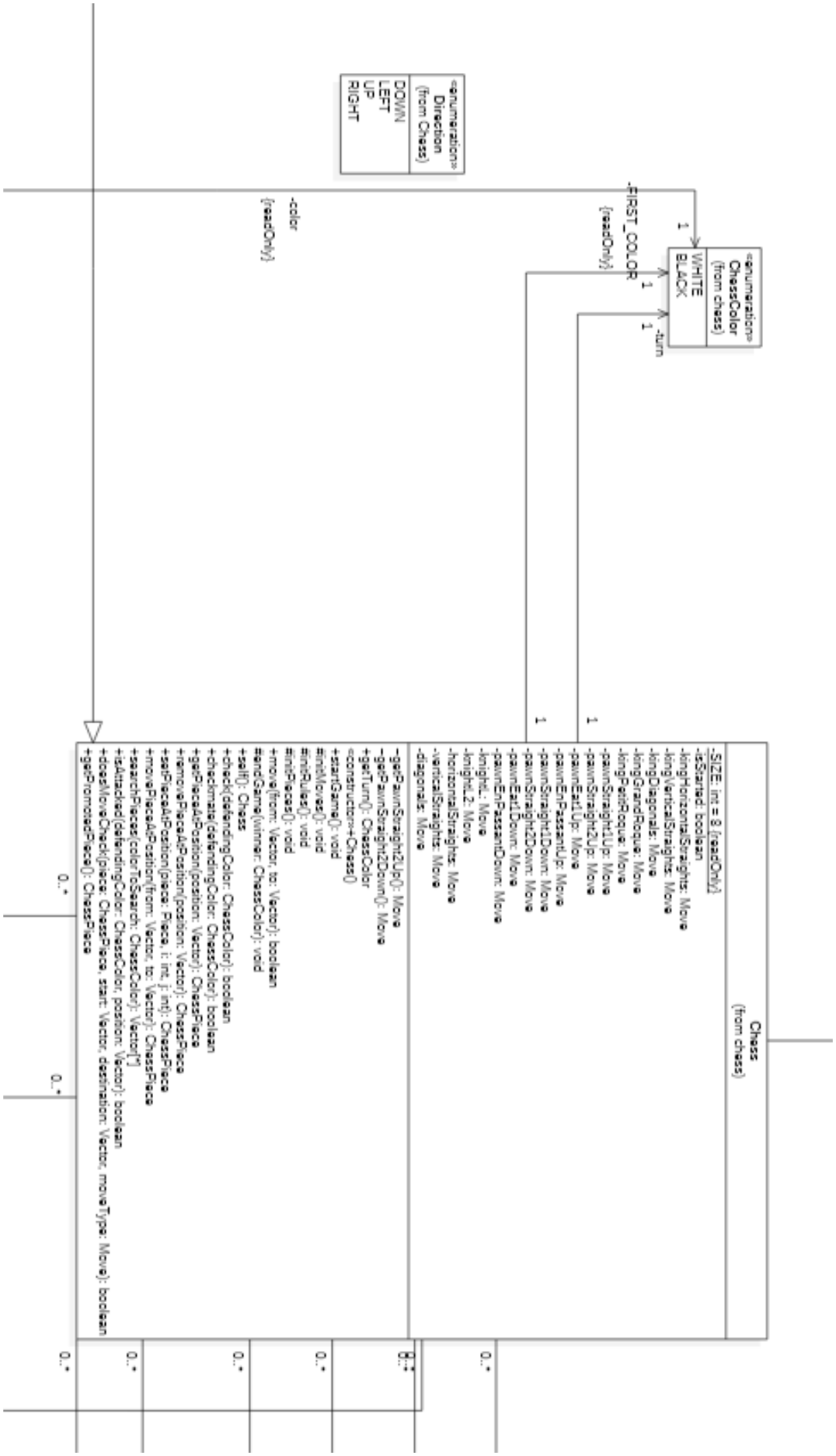
Diagramme UML

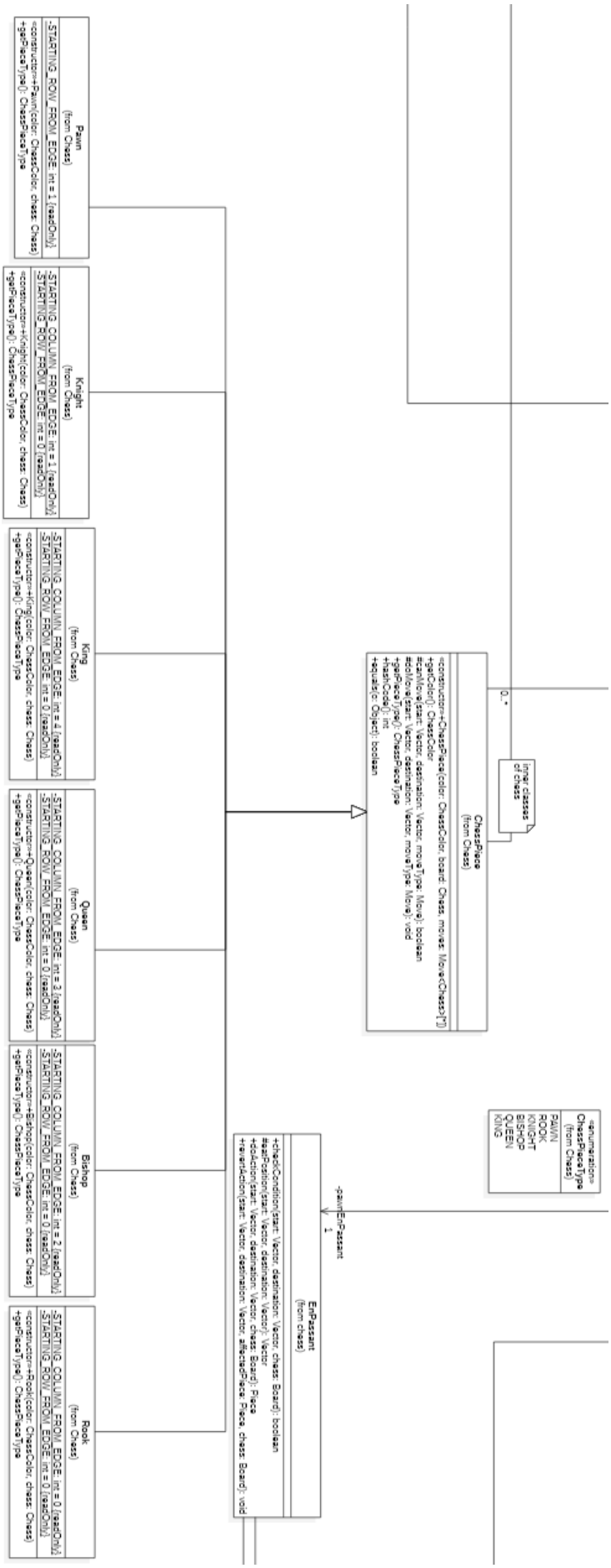


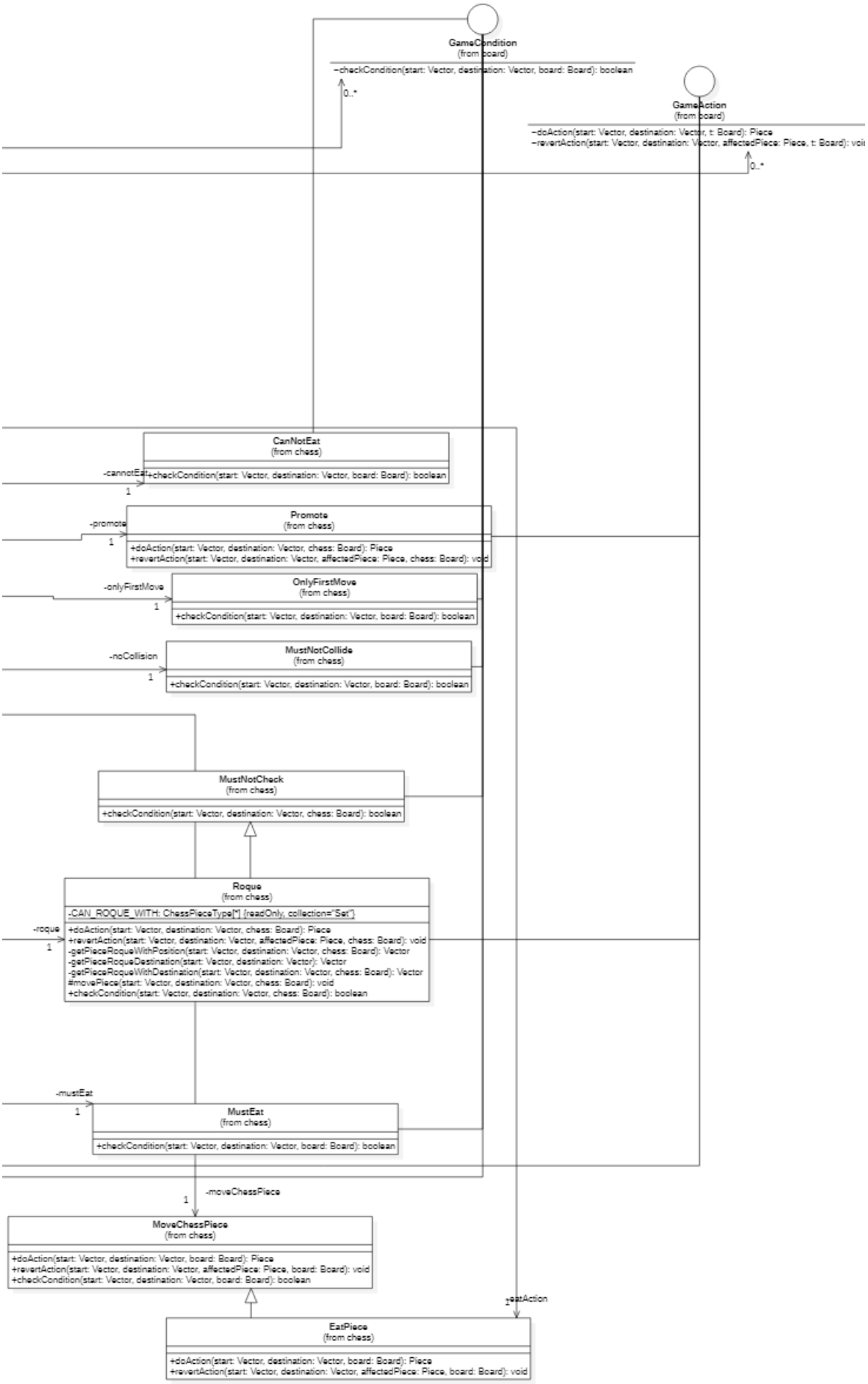

Board
(from board)
-LENGTH: int {readOnly}
-HEIGHT: int {readOnly}
«constructor»#Board(length: int, height: int)
#startGame(): void
#initPieces(): void
+move(from: Vector, to: Vector): boolean
+getLENGTH(): int
+getHEIGHT(): int
+setPieceAtPosition(piece: Piece, position: Vector): Piece
+setPieceAtPosition(piece: Piece, i: int, j: int): Piece
+getPieceAtPosition(position: Vector): Piece
+removePieceAtPosition(position: Vector): Piece
+movePieceAtPosition(from: Vector, to: Vector): Piece
-getSquareAtPosition(position: Vector): Square
+emptyBoard(): void
+searchPieces(pieceToSearch: Piece): Vector[*]
+hasMoved(piece: Piece): boolean
+isLastAction(move: Move): boolean
+lastPieceMoved(): Piece
+set(i): T
T: Board<T>
#contains
{readOnly}
-is formed of
Historic
(from board)
«constructor»~Historic()
~add(piece: Piece, move: Move, depart: Vector, arrivee: Vector, affectedPieces: Piece<T>[*]): void
+isPieceContained(piece: Piece): boolean
+revertLastMove(): void
~isLastAction(move: Move): boolean
~lastPieceMoved(): Piece
-getLastAction(): Action
stack
-is formed of
inner class of Historic
Square
(from Board)
-X: int {readOnly}
-Y: int {readOnly}
«constructor»~Square(x: int, y: int)
-getPiece(): Piece
-setPiece(piece: Piece): void
-getPosition(): Vector
-removePiece(): Piece
inner class of Board
+is on
«read only»
Move
(from board)
-isMirroredX: boolean {readOnly}
-isMirroredY: boolean {readOnly}
«constructor»+Move(vector: Vector, isMirroredX: boolean, isMirroredY: boolean, conditions: GameCondition<T>[*], actions: GameAction<T>[*])
#checkConditions(start: Vector, destination: Vector, t: Board): boolean
~canMove(start: Vector, destination: Vector, t: Board): boolean
~doMove(start: Vector, destination: Vector, t: T): Piece<T>[*]
~revertMove(start: Vector, destination: Vector, affectedPieces: Piece<T>[*], t: Board): void
-move
{readOnly}
-vector
{readOnly}
Vector
(from board)
-i: int
-j: int
«constructor»+Vector(i: int, j: int)
+get(): int
+getJ(): int
+set(i: int): void
+setJ(j: int): void
+add(other: Vector): Vector
+sub(other: Vector): Vector
+multiply(factor: int): Vector
+norm(): double
+getSmallestCollinearVector(): Vector
+includedVectors(): Vector[*]
+crossProduct(other: Vector): int
+areCollinear(other: Vector): boolean
+areCollinearAndSameDirection(other: Vector): boolean
+getMirrorYVector(): Vector
+getMirrorXVector(): Vector
+getOpposedVector(): Vector
-gcd(a: int, b: int): int
+toString(): String
+equals(o: Object): boolean
+hashCode(): int
-movements
Action
(from Historic)
«constructor»+Action(piece: Piece, move: Move, depart: Vector, arrivee: Vector, affectedPieces: Piece<T>[*])
-isMove(move: Move): boolean
-revert(): void
-affectedPieces
-piece
{readOnly}
Piece
(from board)
«constructor»+Piece(board: T, moves: Move<T>[*])
+move(fromX: int, fromY: int, toX: int, toY: int, doMove: boolean): boolean
#canMove(start: Vector, destination: Vector, moveType: Move): boolean
#doMove(start: Vector, destination: Vector, moveType: Move): void
+move(start: Vector, destination: Vector, doMove: boolean): boolean
+possibleMoves(start: Vector): Vector[*]
+toString(): String
+textValue(): String
-depart
{readOnly}
-arrivee
{readOnly}
-piece
T: Board<T>




DisplayChess
(from displayChess)
-guiPromptsDisabled: boolean
~areGUIPromptsDisable(): boolean
«constructor»+DisplayChess(controller: Controller)
#initRules(): void
+move(fromX: int, fromY: int, toX: int, toY: int): boolean
+check(defendingColor: ChessColor): boolean
#endGame(winner: ChessColor): void
+setPieceAtPosition(piece: Piece, position: Vector): ChessPiece
+removePieceAtPosition(position: Vector): ChessPiece
+doesMoveCheck(piece: ChessPiece, start: Vector, destination: Vector, moveType: Move): boolean
+getPromotedPiece(): ChessPiece
-askUserPromotion(): ChessPiece
+displayCheck(): void
#displayWinner(winner: ChessColor): void
-getPlayerColor(color: ChessColor): PlayerColor
-getPieceType(type: ChessPieceType): PieceType
1
1
-connect
«readOnly»
Controller
(from displayChess)
-view: ChessView
«constructor»+Controller()
+getView(): ChessView
+start(view: ChessView): void
+move(fromX: int, fromY: int, toX: int, toY: int): boolean
+newGame(): void
+main(args: String[*]): void


«enumeration»
Direction
(from Chess)
DOWN
LEFT
UP
RIGHT
«enumeration»
ChessColor
(from chess)
WHITE
BLACK
-FIRST_COLOR
{readOnly}
-color
{readOnly}
-turn
1
Chess
(from chess)
-SIZE: int = 8 {readOnly}
-isStarted: boolean
-kingHorizontalStraights: Move
-kingVerticalStraights: Move
-kingDiagonals: Move
-kingGrandRoque: Move
-kingPetitRoque: Move
+pawnStraight1Up: Move
+pawnStraight2Up: Move
+pawnEat1Up: Move
+pawnEnPassantUp: Move
+pawnStraight1Down: Move
+pawnStraight2Down: Move
+pawnEat1Down: Move
+pawnEnPassantDown: Move
-knight1: Move
-knight2: Move
-horizontalStraights: Move
-verticalStraights: Move
-diagonals: Move
-getPawnStraight2Up(): Move
-getPawnStraight2Down(): Move
+getTurn(): ChessColor
«constructor»+Chess()
+startGame(): void
#initMoves(): void
#initRules(): void
#initPieces(): void
+move(from: Vector, to: Vector): boolean
#endGame(winner: ChessColor): void
+self(): Chess
+check(defendingColor: ChessColor): boolean
+checkmate(defendingColor: ChessColor): boolean
+getPieceAtPosition(position: Vector): ChessPiece
+removePieceAtPosition(position: Vector): ChessPiece
+setPieceAtPosition(piece: Piece, i: int, j: int): ChessPiece
+movePieceAtPosition(from: Vector, to: Vector): ChessPiece
+searchPieces(colorToSearch: ChessColor): Vector[*]
+isAttacked(defendingColor: ChessColor, position: Vector): boolean
+doesMoveCheck(piece: ChessPiece, start: Vector, destination: Vector, moveType: Move): boolean
+getPromotedPiece(): ChessPiece
0..*
0..*
0..*
0..*
0..*
0..*


Pawn
(from Chess)
-STARTING_ROW_FROM_EDGE: int = 1 {readOnly}
+<<constructor>>+Pawn(color: ChessColor, chess: Chess)
+getPieceType(): ChessPieceType
Knight
(from Chess)
-STARTING_COLUMN_FROM_EDGE: int = 1 {readOnly}
-STARTING_ROW_FROM_EDGE: int = 0 {readOnly}
+<<constructor>>+Knight(color: ChessColor, chess: Chess)
+getPieceType(): ChessPieceType
King
(from Chess)
-STARTING_COLUMN_FROM_EDGE: int = 4 {readOnly}
-STARTING_ROW_FROM_EDGE: int = 0 {readOnly}
+<<constructor>>+King(color: ChessColor, chess: Chess)
+getPieceType(): ChessPieceType
Queen
(from Chess)
-STARTING_COLUMN_FROM_EDGE: int = 3 {readOnly}
-STARTING_ROW_FROM_EDGE: int = 0 {readOnly}
+<<constructor>>+Queen(color: ChessColor, chess: Chess)
+getPieceType(): ChessPieceType
Bishop
(from Chess)
-STARTING_COLUMN_FROM_EDGE: int = 2 {readOnly}
-STARTING_ROW_FROM_EDGE: int = 0 {readOnly}
+<<constructor>>+Bishop(color: ChessColor, chess: Chess)
+getPieceType(): ChessPieceType
Rook
(from Chess)
-STARTING_COLUMN_FROM_EDGE: int = 0 {readOnly}
-STARTING_ROW_FROM_EDGE: int = 0 {readOnly}
+<<constructor>>+Rook(color: ChessColor, chess: Chess)
+getPieceType(): ChessPieceType
ChessPiece
(from Chess)
+<<constructor>>+ChessPiece(color: ChessColor, board: Chess, moves: MovesChess[*])
+getColor(): ChessColor
#canMove(start: Vector, destination: Vector, moveType: Move): boolean
#doMove(start: Vector, destination: Vector, moveType: Move): void
+getPieceType(): ChessPieceType
+hashCode(): int
+equals(o: Object): boolean
0..*
inner classes of chess
<<enumeration>>
ChessPieceType
(from Chess)
PAWN
ROOK
KNIGHT
BISHOP
QUEEN
KING
EnPassant
(from chess)
+checkCondition(start: Vector, destination: Vector, chess: Board): boolean
#eatPosition(start: Vector, destination: Vector): Vector
+doAction(start: Vector, destination: Vector, chess: Board): Piece
+reverseAction(start: Vector, destination: Vector, attackedPiece: Piece, chess: Board): void
-pawnEnPassant
1


GameCondition
(from board)
-checkCondition(start: Vector, destination: Vector, board: Board): boolean
0..*
GameAction
(from board)
-doAction(start: Vector, destination: Vector, t: Board): Piece
-revertAction(start: Vector, destination: Vector, affectedPiece: Piece, t: Board): voi
0..*
CanNotEat
(from chess)
-cannotEat
1
+checkCondition(start: Vector, destination: Vector, board: Board): boolean
Promote
(from chess)
-promote
1
+doAction(start: Vector, destination: Vector, chess: Board): Piece
+revertAction(start: Vector, destination: Vector, affectedPiece: Piece, chess: Board): void
OnlyFirstMove
(from chess)
-onlyFirstMove
1
+checkCondition(start: Vector, destination: Vector, board: Board): boolean
MustNotCollide
(from chess)
-noCollision
1
+checkCondition(start: Vector, destination: Vector, board: Board): boolean
MustNotCheck
(from chess)
+checkCondition(start: Vector, destination: Vector, chess: Board): boolean
Roque
(from chess)
-CAN_ROQUE_WITH: ChessPieceType[*] {readOnly, collection='Set'}
-roque
1
+doAction(start: Vector, destination: Vector, chess: Board): Piece
+revertAction(start: Vector, destination: Vector, affectedPiece: Piece, chess: Board): void
-getPieceRoqueWithPosition(start: Vector, destination: Vector, chess: Board): Vector
-getPieceRoqueDestination(start: Vector, destination: Vector): Vector
-getPieceRoqueWithDestination(start: Vector, destination: Vector, chess: Board): Vector
#movePiece(start: Vector, destination: Vector, chess: Board): void
+checkCondition(start: Vector, destination: Vector, chess: Board): boolean
-mustEat
1
MustEat
(from chess)
+checkCondition(start: Vector, destination: Vector, board: Board): boolean
1
-moveChessPiece
MoveChessPiece
(from chess)
+doAction(start: Vector, destination: Vector, board: Board): Piece
+revertAction(start: Vector, destination: Vector, affectedPiece: Piece, board: Board): void
+checkCondition(start: Vector, destination: Vector, board: Board): boolean
1
eatAction
EatPiece
(from chess)
+doAction(start: Vector, destination: Vector, board: Board): Piece
+revertAction(start: Vector, destination: Vector, affectedPiece: Piece, board: Board): void