HEIG-VD — POO

Laboratoire 8 (Échecs) – Compte-rendu

[redacted]

20 octobre 2023

Ce document décrit nos choix et hypothèses de travail effectués pour réaliser le laboratoire n° 8 dans lequel nous avons dû créer une représentation d'un jeu d'échecs.

1 Modélisation

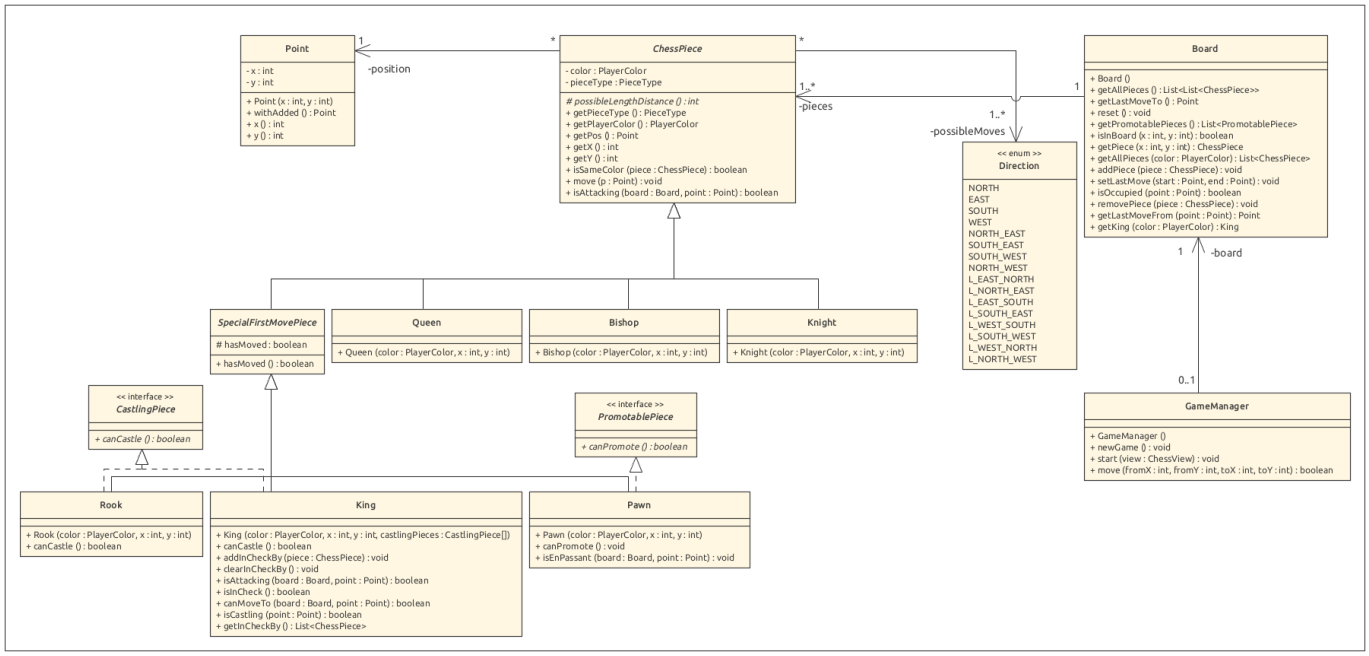

Figure 1 – Diagramme UML représentant les classes implémentées

2 Choix et hypothèses de travail

2.1 Paquetage `piece`

Piece est un paquetage contenant l'intégralité des classes représentant les pièces disponibles dans notre partie. On trouve également des interfaces ayant comme but de modéliser des pièces ayant un comportement spécifique pour certaines situations.

2.1.1 ChessPiece

'ChessPiece' est la classe mère de toutes les pièces, cette classe contient les attributs et méthodes communes à toutes les pièces pour un fonctionnement identique.
Une ChessPiece est composé de son type de pièce, sa couleur, sa position et les mouvements possibles qu'elle peut effectuer.
Chaque pièce devra redéfinir la méthode 'possibleLengthDistance()' dans le but d'avoir la distance maximale d'un déplacement possible pour la pièce.
Les méthodes publiques 'canMoveTo()' et 'move()' sont à utiliser pour déplacer une pièce sur le plateau. La méthode 'canMoveTo()' permet de vérifier si le déplacement est possible et 'move()' sert seulement à mettre à jour la piece en question. 'canMoveTo()' contient des conditions générales pour toutes les pièces, les classes filles devront redéfinir cette méthode pour ajouter des conditions spécifiques si nécessaire.
La méthode 'isAttacking()' permet de vérifier si une pièce peut attaquer une autre pièce à une position donnée.

De manière générale, cette méthode ne fait rien de plus que de vérifier si la pièce peut atteindre la position donnée avec canReach(), or les pièces ayant un mode de capture différent de celui de déplacement devront redéfinir isAttacking() pour ajouter des conditions supplémentaires et/ou spécifiques.
La méthode 'isSameColor()' permet de vérifier qu'une pièce est de la même couleur.
Les getters 'getDirections()', 'getPieceType()', 'getPlayerColor()', 'getPos()', ainsi que 'getX()' et 'getY()' retournent simplement des attributs utilisés en dehors la classe.
'ChessPiece' implémente l'interface fournie 'ChessView.UserChoice' dans le but de permettre à l'utilisateur de choisir une pièce lorsqu'une 'PromotablePiece' est promue.

2.1.2 SpecialFirstMovePiece

Une classe 'SpecialFirstMovePiece' a été créée pour les pièces ayant un attribut spécifique indiquant si la pièce s'est déjà déplacée à l'aide de 'move()'. Ainsi que deux interfaces 'CastlingPiece' et 'PromotablePiece' pour les pièces pouvant intervenir dans un roque ou une promotion.
'Pawn' et 'King' sont les classes ayant des comportements spécifiques et doivent à leur tour redéfinir certaines méthodes.

2.1.3 Pawn

'Pawn' récrit 'canReach()' pour prendre en compte le fait qu'il peut 'avancer' en diagonale seulement s'il y a une pièce (adversaire ou pas, 'canMoveTo()' fait les autres vérifications).
Il peut également redéfinir 'isAttacking()', pour considérer le fait qu'il peut attaquer uniquement en diagonale ou s'il s'agit d'un mouvement spécial comme la prise en passant.
Implémentant l'interface 'PromotablePiece', 'Pawn' doit définir la méthode 'canPromote()', permettant de vérifier s'il peut être promu.

2.1.4 King

Quant au 'King', il a des attributs supplémentaires permettant de stocker les pièces mettant en échec le roi et les 'CastlingPiece' pouvant intervenir dans un roque avec le roi.
Il doit redéfinir 'canReach()' pour prendre en compte le fait qu'il ne peut pas se déplacer sur un emplacement mettant en échec le roi.
Il doit aussi redéfinir 'isAttacking()' pour prendre en compte le fait qu'il peut attaquer un emplacement même si l'emplacement est en échec.
Il doit aussi réécrire 'canMoveTo()' pour prendre en compte le fait qu'il peut effectuer un roque si les conditions sont remplies.

2.2 Paquetage `util`

Ce paquetage contient des classes utilitaires nécessaires pour le bon fonctionnement du jeu d'échecs.

2.2.1 Direction

La classe 'Direction' est un type énuméré constitué de deux entiers x et y. Ces deux entiers représentent les coordonnées d'un vecteur directeur.
Nous avons défini les directions possibles pour une pièce dans cette classe.
Il existe 8 directions unitaires représentant les mouvements linéaires possibles pour une pièce.
En ce qui concerne le mouvement en L, nous avons également créé les 8 directions correspondantes à ce mouvement.
L'implémentation permet une utilisation simple et efficace de ces directions or ceci a des limitations si l'on souhaite faire des mouvements plus complexes. En effet, il est possible de définir des mouvements plus complexes en combinant les directions unitaires, or cela rend la lecture du code plus difficile pour ce qui est demandé dans le cadre du projet. Or, cette implémentation permet la réutilisation de ces directions dans toutes les pièces sans avoir besoin d'en créer davantage des nouvelles instances pour chaque pièce.
Certaines fonctions comme 'offSetOf()' permettent d'obtenir la direction représentée par deux coordonnées.
La méthode 'moveDistance()' permet de réduire les vecteurs en forme irréductible et retourner le lambda correspondant grâce à l'algorithme d'Euclide se trouvant dans la classe 'ChessMath'.

D'autres méthodes comme 'opposite()' permettant d'obtenir la direction opposée à l'actuelle, 'isDiagonal()' permettant de vérifier si la direction est diagonale, 'exists()' permettant de vérifier si une direction existe sont mis à disposition pour rendre le code plus lisible et facile à utiliser.
Des getters pour les offsets x et y sont également disponibles.

2.2.2 Point

'Point' n'est qu'un record contenant deux entiers x et y. Il est utilisé pour représenter une position. Par default, pour représenter les positions des pièces sur le plateau. Il contient une méthode faisant 'avancer' un point dans une direction donnée et rendant le nouveau point.

2.2.3 ChessString

'ChessString' est une classe finale contenant seulement des attributs et méthodes statiques permettant de manipuler des chaînes les chaînes de caractères que l'on affichera dans notre jeu d'échecs. Cela permet de centraliser les chaînes de caractères utilisées dans le jeu et de les modifier facilement si besoin.

2.2.4 Assertions

'Assertions' est une classe finale contenant uniquement des méthodes statiques comme 'assertTrue()' permettant de vérifier si une un ou plusieurs arguments remplissent une certaine condition.
Si la condition n'est pas remplie, un 'IllegalArgumentException' est lancée avec le message donné en paramètre.
La méthode 'assertNotNull()' permet de vérifier si un argument est null. Si c'est le cas, la même exception est lancée.

2.2.5 Board

'Board' est une classe finale stockant les informations importantes de notre jeu d'échecs.
L'attribut 'pieces' est un tableau à deux dimensions de 'ChessPiece' de taille 8x8 représentant notre tableau de jeu. Il a une liste de 'PromotablePiece' permettant de stocker toutes les pièces pouvant être promues.
Cette solution permet de ne pas avoir à parcourir le tableau de jeu à chaque fois qu'une pièce doit être promue et d'éviter les 'instance of' qui sont plus coûteux en temps d'exécution. Il contient une autre liste contentant les deux rois nous permettant de d'obtenir facilement le roi de chaque couleur.
L'attribut 'lastMove' est un tableau de deux 'Point' permettant de stocker la position de la pièce ayant effectué le dernier mouvement et la position d'arrivée de cette pièce. Aucune logique n'est implémentée dans cette classe. Elle contient seulement des attributs et des méthodes permettant de manipuler ces attributs.
Il contient aussi d'autres méthodes nous donnant accès à différentes informations sur le plateau de jeu comme : 'addPiece()', 'removePiece()', 'getPiece()', 'isOccupied()', 'isInBoard()'.
Par défaut, 'Board' est initialisé avec les pièces placées aux positions initiales du jeu d'échecs grâce à fillBoard(). Lorsqu'on utilise reset(), la partie est remise à zéro et les pièces sont remises à leurs positions initiales.

2.2.6 GameManager

'GameManager' est le centre de notre jeu d'échecs.
Elle doit tout d'abord implementer l'interface fournie 'ChessController', laquelle nous demande de définir les méthodes 'newGame()', 'start()' et move().
La première nous permettant de préparer une nouvelle partie, la deuxième nous permettant d'initialiser le jeu à l'aide d'un ChessView et la dernière nous permettant d'effectuer les mouvements dans notre jeu.
Elle contient un attribut 'board' de type 'Board' permettant de stocker les informations du jeu. Elle contient également un attribut 'view' de type 'ChessView' permettant de stocker la vue du jeu ayant été initialisée par 'start()'. Elle contient également l'attribut 'turn', un entier représentant le tour actuel de la partie et 'mandatoryAdversaryMoves', une HashMap contenant les mouvements obligatoires de l'adversaire. Cette HashMap est utilisée pour vérifier si un mouvement fait partie des mouvements obligatoires de l'adversaire pour sortir d'un échec.
Cette HashMap est récréée lorsque le joueur adverse est mis en échec et vidée quand le joueur adverse effectue un mouvement, ce qui voudrait dire qu'il n'est plus en échec. Il existe trois méthodes manipulant cette HashMap : 'displacementMoves()', 'blockingMoves()' et 'capturingMoves()' permettent de trouver les

manières possibles pour échapper à en bougeant de la position actuelle, en bloquant la direction de l'attaque ou en capturant la pièce attaquante.
Lorsque la méthode 'move()' est appelée, la première méthode indispensable au bon fonctionnement est 'movePreconditions()', elle vérifie si le mouvement est valide et si le joueur peut effectuer ce mouvement. Si le mouvement est valide, 'executeSpecialMoves()' réalisera les mouvements spéciaux dans lequel une deuxième pièce est impliquée (comme la prise en passant ou le roque).
Par la suite, 'movePiece()' effectuera le mouvement de la pièce en mettant à jour le board et la view. La méthode 'promoteIfNeeded()' regardera si une promotion est possible par l'une des 'PromotablePiece' et la demandera à l'utilisateur l'une des options disponibles.
Finalement, 'postMoveActions()' effectuera les actions nécessaires après le mouvement : regarder si le roi est en échec par un échec positionnel, échec à découvert ou les deux. Notre engine regardera par la suite si on se retrouve dans une fin de jeu ('pat', 'impossibilité de mat par manque de matériel' ou 'échec et mat').
Le message correspondant s'affichera et l'utilisateur sera invité à recommencer une partie grâce à la méthode 'postGameActions()' et à une classe anonyme dérivant de 'ChessView.UserChoice'. Une fonction clearView() personnelle a dû être implémentée pour nettoyer la vue du jeu après une partie, car celle qui est disponible dans la vue n'est pas publique.

2.3 Conclusion

Nous avons réalisé un jeu d'échecs en utilisant au mieux les concepts de la programmation orientée objet. L'utilisation des interfaces pour définir des comportements très spécifique à un groupe de pièces ainsi que des classes abstraites pour définir leurs comportements communs ont été au cœur de notre conception.
La redéfinition de méthodes nous a permis de simplifier de manière importante le code réalisé. Nous avons analysé et optimisé les règles du jeu dans le but d'obtenir un code, non seulement fonctionnel, mais aussi efficace avec les différents événements qui peuvent se produire dans une partie d'échecs.
Des améliorations peuvent encore être effectuées, mais les changements s'implémentent facilement grâce à la façon dont nous avons conçu notre code.

3 Tests des fonctionalités

1. Classe `AutomaticTest`.
   Permet de jouer automatiquement une partie dont on donne l'ensemble des coups. Lorsque AutomaticTests est lancé, le testeur doit sélectionner "Yes" pour lancer la partie suivante.
   - Les blancs gagnent par echec et mat
   - Les blancs gagnent par echec et mat, avec utilisation de la prise en passant
   - Les blancs gagnent par echec et mat, avec utilisation du roque
   - Les noirs gagnent par echec et mat
   - Egalité par pat (2 versions)
   - Egalité par matériel insuffisant

   Résultats : les parties se déroulent correctement
2. Classe `ChessPieceTest`
   Permet de tester les différentes sous classes de ChessPiece, en utilisant la classe TestFramework.
   - Tests des mouvements de la classe Pawn
   - Test de la prise en passant
   - Test pour la promotion
   - Test des mouvements de la classe Rook
   - Test des mouvements de la classe Bishop
   - Test des mouvements de la classe Knight
   - Test des mouvements de la classe Queen
   - Test des mouvements de la classe King
   - Test du roque (castling)

   Résultats : les mouvements sont correctement testés