Programmation d'un jeu d'échecs

Projet Final (S3 Programmation 2)

1 Objectif

L'objectif de ce projet est de programmer un jeu d'échec. Ce jeu permet à deux joueurs humains de jouer ensemble ou bien à un joueur humain de jouer contre une version très simple d'un joueur artificiel, ou à deux joueurs artificiels de jouer ensemble !

Si vous ne connaissez pas les règles des échecs, vous pouvez les trouver sur la page wikipédia française du jeu d'échec.

2 Les classes principales

Le projet est constitué de quatorze classes. Certaines, telles que les classes implémentant l'interface graphique, sont complètement programmées tandis que d'autres ne le sont pas du tout. Vous trouverez dans les sections ci-dessous la description des classes et de certaines de leurs principales méthodes. Lorsque le nom des méthodes est assez explicite, nous ne décrirons pas leur comportement.

2.1 Case et coordonnées

Une case de l'échiquier peut être identifiée grâce à ses coordonnées (une ligne et une colonne), représentées par un objet de la classe `Coordinates`, qui définit les méthodes suivantes :

- `public String toString()` retourne une chaîne de caractère représentant des coordonnées.
- `public boolean equals(Object o)` retourne vrai si le paramètre `o` est un objet de la classe `Cooordonnées` et si les composantes de `o` sont égales à celles de `this`.

Un déplacement est représenté par un objet de la classe `FromTo` qui définit deux membres : les coordonnées de l'origine : `from` et les coordonnées de la destination : `to`.

2.2 Les pièces

Une pièce est représentée par la classe abstraite `Piece`. Cette dernière définit les attributs `position` (la position actuelle de la pièce sur l'échiquier) et `owner` (de la classe `Player`) qui est le joueur auquel appartient la pièce.

Six classes étendent la classe `Piece` : `Bishop` (fou), `Queen` (reine), `King` (roi), `Pawn` (pion), `Rook` (tour) et `Knight` (cavalier).

La classe `Piece` déclare ou définit les méthodes suivantes :

- `public void setPosition(Coordinates position)`
- `public List<Move> getAllMoves(Board board)` Renvoie tous les mouvements (voir la classe `Move`) possibles de la pièce. Cette méthode doit être définie dans les classes qui étendent `Piece`.
- `public boolean sameColor(Piece anotherPiece)` retourne `true` si la pièce est de la même couleur que `anotherPiece`.
- `public abstract boolean isMoveAuthorized(Board b, Coordinates dest)` retourne `true` si la pièce peut se déplacer sur la case de coordonnées `dest`, selon les règles de déplacement de cette pièce. Dans ce cas, on dit que ce déplacement est légal. Cette méthode doit être définie dans les classes qui étendent `Piece`.
- `public abstract Type getType()` retourne le `Type` (tel que défini dans la classe `Piece`) de la pièce.
- `public abstract int getValue()` retourne la valeur de la pièce [1].

1. Les valeurs des pièces sont les suivantes : un pion vaut un point, un fou ou un cavalier vaut 3 point, une tour vaut 5 points et une dame vaut 9 points.

2.3 L'échiquier

L'échiquier est représenté par la classe `Board`. Cette dernière définit un tableau bidimensionnel 8×8 de `Piece`. La classe `Board` définit les méthodes suivantes :

- `public Board(String fileName, Player white, Player black)` C'est un constructeur de la classe `Board`, il prend en paramètre un nom de fichier et deux joueurs. Le fichier décrit la configuration de l'échiquier. Chaque ligne de ce fichier correspond à une pièce et à sa position. La pièce est représentée par une lettre minuscule s'il s'agit d'une pièce blanche et majuscule s'il s'agit d'une pièce noire. La lettre indique la nature de la pièce (`k` pour roi, `q` pour reine, `n` pour cavalier, `b` pour fou, `r` pour tour et `p` pour pion. La suite de caractères `r00` désigne la tour blanche située sur la case de coordonnées $(0, 0)$. Vous trouverez dans le répertoire `boardConfigurationFiles` des fichiers de configuration, dont le fichier `FullBoard.txt` qui correspond à la configuration de début de partie.
- `public List<Piece> getPieces(Player p)` renvoie une liste de toutes les pièces appartenant au joueur `p`.
- `public List<Piece> getPieces()` renvoie une liste de toutes les pièces se trouvant sur l'échiquier.
- `public void addPiece(Piece p)` ajoute la pièce `p` sur l'échiquier.
- `public Piece getPiece(Coordinates pos)` retourne la pièce se trouvant à la position `pos`.
- `public void emptyCell(Coordinates pos)` enlève la pièce se trouvant à la position `pos`, s'il y en a une.
- `public boolean isEmptyCell(Coordinates pos)` retourne `true` si la position `pos` est vide.
- `public List<Coordinates> getAllCoordinates()` retourne une liste des coordonnées de toutes les cases de l'échiquier.
- `public boolean sameColumnNothingBetween(Coordinates o, Coordinates d)` retourne `true` si `o` et `d` sont sur la même colonne et qu'il n'y a pas de pièces situées entre `o` et `d`.
- `public boolean sameRowNothingBetween(Coordinates o, Coordinates d)` même chose pour les lignes.
- `public boolean sameDiagonalNothingBetween(Coordinates o, Coordinates d)` même chose pour les diagonales.

2.4 Les Coups

Un coup est représenté par la classe `Move` qui définit une position de départ, une position d'arrivée, une pièce de départ (la pièce se trouvant à la position de départ avant le jeu du coup) et une pièce d'arrivée (la pièce se trouvant à la position d'arrivée avant le jeu du coup).

2.5 Les joueurs

Un joueur est représenté par la classe abstraite `Player`, qui possède comme attributs une couleur, un score, un roi (qui permet d'accéder directement au roi d'un joueur) et deux attributs booléens qui permettent de savoir sur le joueur est en échec ou en échec et mat. La classe `Player` définit les méthodes suivantes :

- `public abstract Move getMove()` retourne un coup à jouer par le joueur. Cette méthode est abstraite, elle doit être implémentée par chaque classe qui étend `Player`.
- `public List<Move> getAllMoves(Board board)` retourne la liste de tous les mouvements légaux du joueur.

Il existe déjà une classe qui étend la classe `Player`, il s'agit de la classe `Human` qui représente un joueur humain. La méthode `getMove()` pour cette classe attend simplement que l'utilisateur déplace la souris pour indiquer un déplacement (Voir `waitForPlayerMove()` dans la classe `ChessUI`).

2.6 La partie

Une partie est représentée par la classe `GameUI`, qui possède pour membres un échiquier, deux joueurs (blanc et noir), le joueur qui a le trait (celui qui doit jouer) et l'histoire de tous les coups qui ont été joués depuis le début de la partie.

- `public boolean isMovePlayable(Move move)` retourne `true` si `move` peut être joué, c'est à dire si il est légal et si les autres conditions sont vérifiées (c'est le bon joueur qui joue, il joue bien une de ses pièces, la prise n'est pas un roi ...).
- `public void applyMove(Move move)` joue le coup `move`
- `public boolean undo()` défait le dernier coup joué.
- `public void switchPlayers()` change le joueur qui a le trait (si c'est blanc qui jouait, c'est maintenant noir et vice-versa).
- `public boolean isPrey(Piece prey)` retourne `true` si la pièce `prey` peut être prise.
- `public boolean isCheck(Player player)` retourne `true` si le joueur qui a le trait est en échec.
- `public boolean isCheckMate(Player player)` retourne `true` si le joueur qui a le trait est en échec et mat.

— `public Player determineWinner()` retourne le joueur qui a remporté la partie, `null` si les deux joueurs sont *ex aequo*.
— `public void play()` joue une partie. Le jeu consiste à donner le trait successivement aux deux joueurs et de déterminer si un joueur est échec et mat. Si au bout de 50 coups, aucun des deux joueurs est échec et mat, la partie s'arrête.

2.7 L'interface graphique

L'interface graphique est implémentée par la classe `ChessUI`. Cette Classe est entièrement implémentée et vous n'avez pas besoin de la modifier. Elle définit, entre autre, la méthode suivante :
— `public FromTo waitForPlayerMove()` retourne le déplacement réalisé par le joueur humain à l'aide de la souris.

2.8 Les tests

Les tests sont réalisés à l'aide de la classe `TestChess`, qui définit les méthodes suivantes :
— `public static boolean testAuthorizedMove(String configurationFileName, Coordinates o, Coordinates d)` crée une partie à partir d'un fichier de configuration et retourne `true` si le coup ayant pour origine `o` et pour destination `d` est légal.
— `public static boolean testPlayableMove(String configurationFileName, Coordinates o, Coordinates d)` crée une partie à partir d'un fichier de configuration et retourne `true` si le coup ayant pour origine `o` et pour destination `d` est jouable.
— `public static boolean testIsCheck(String configurationFileName, Player p)` crée une partie à partir d'un fichier de configuration et retourne `true` si le joueur `p` est en échec.
— `public static boolean testIsCheckMate(String configurationFileName, Player p)`crée une partie à partir d'un fichier de configuration et retourne `true` si le joueur `p` est en échec et mat.

3 Les tâches

3.1 Tâche 1 : les différents types de pièce

Définir les classes `Bishop` (fou), `Queen` (reine), `Pawn` (pion), `Rook` (tour) et `Knight` (cavalier), qui étendent toutes la classe `Piece`. La classe `King` (roi) est déjà partiellement implémentée, vous pouvez vous en inspirer pour construire les classes pour les autres pièces.

Chacune de ces classes doit implémenter la méthode `isMoveAuthorized`, qui définit les déplacements légaux pour chaque type de pièce. On trouvera une description de règles dans la page wikipedia française du jeu d'échecs, section *Déplacements*[2]. On autorisera au niveau de chaque type de pièce les déplacements provoquant la capture du roi. C'est la méthode `isMovePlayable` de la classe `GameUI` qui doit vérifier qu'un mouvement ne provoque pas la capture d'un roi.

Test des déplacements Légaux

Les règles de déplacement légaux étant relativement complexes, il est nécessaire de les tester rigoureusement à l'aide de la méthode `testAuthorizedMove` de la classe `TestChess`. Pour chaque type de pièce, vous devez écrire des tests permettant de tester **toutes** les règles de déplacement légaux.

3.2 Tâche 2 : Partie

Implémenter une première version de la méthode `play` de la classe `GameUI`. qui réalise une partie de jeu entre deux joueurs `black` et `white` Pour verifier chaque mouvements choisi par les joueurs, implementer la méthode `isMovePlayable`. Vous ne vous soucierez pas à ce stade de savoir si un joueur est échec ou mat. Une partie prend fin si le nombre de coups joués atteint 50 ou si un des joueurs est mat. Le gagnant est celui qui a totalisé le plus de points. Les points dépendent des pièces capturées à l'adversaire.

2. On ignorera dans ce projets les déplacements suivants : le roque, la prise en passant et la promotion.

Test des déplacements Jouables

Utilisez la méthode `testPlayableMove` de la classe `TestChess` pour vérifier que toutes les règles de déplacement jouables ont été implémentées.

3.3 Tâche 3 : la mise en échec

Un joueur est en échec si son roi peut être capturé par son adversaire. Implémentez la méthode `isCheck` de la classe `GameUI` qui vérifie si un joueur est en échec ou pas. Vous modifierez ensuite la méthode `play` pour vérifier qu'un mouvement d'un joueur ne le mets pas en échec.

Tests de la mise en échec

Utilisez la méthode `testIsCheck` de la classe `TestChess` pour vérifier toutes les cas de mise en échec.

3.4 Tâche 4 : Echec et mat

Un joueur est échec et mat s'il est en échec et, quoi qu'il fasse, il reste en échec. Ecrire la méthode `isCheckMate(Board board)` de la classe `GameUI` qui vérifie si un joueur est échec et mat. Vous modifierez ensuite la méthode `play` pour vérifier qu'un joueur n'est pas échec et mat avant de jouer.

Tests d' Echec et mat

Utilisez la méthode `testIsCheckMate` de la classe `TestChess` pour vérifier toutes les cas de mise en échec et mat.

3.5 Tâche 5 : `ChessBot`

Implémenter la classe `ChessBot` qui étend la classe `Player`. Cette classe représente un joueur artificiel. Ce joueur choisit, parmi tous les coups possibles, un de ceux qui lui rapporte le plus de points.

Tests pour la classe ChessBot

Créer une partie entre un joueur `ChessBot` et une joueur humain pour vérifier le bon comportement de cette classe.