HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

INTRODUCTION

Dans le cadre du cours de programmation orientée objet 1 (POO1) de l'HEIG-VD, nous devons effectuer des laboratoires afin de pouvoir mettre en pratique ce qui est vu durant les périodes de théorie du cours.

Ce neuvième laboratoire est effectué par un groupe composé de 2 personnes, durant 12 périodes de laboratoire.

Les objectifs pour ce laboratoire sont de réaliser un jeu d'échec. Pour ceci, une GUI et une interface console nous est donnée. Nous avons également reçu un schéma UML que l'on doit compléter. Ce laboratoire est assez libre et a pour but de voir notre conception d'un problème plus complexe en programmation orientée objet.

Toutes les sources Java représentant les classes de l'UML de base qui a été donné ont également été distribuées aux élèves.

Nous avons prévu de travailler les deux en même temps sur la conceptualisation de ce laboratoire pour que chacun puisse directement apporter ses idées et également pour gagner en productivité. Ensuite, nous nous répartirons l'implémentation à réaliser en fonction de l'UML créé à deux.

Avant de commencer ce laboratoire, nous avons hâte de pouvoir mettre en pratique les éléments appris afin de mieux comprendre leur utilisation. Nous nous réjouissons également de pouvoir utiliser la GUI ce qui rend toujours l'implémentation de programmes plus intéressante qu'en mode console.

DIAGRAMME DE CLASSES UML

En annexe.

NOTES SUR LE DIAGRAMME

Nous avons gardé les références des rois dans la board afin de plus facilement détecter s'il y a un cas d'échec. Cela nous permet de ne pas parcourir toutes les cases et ainsi de réduire la complexité de l'algorithme.

L'attribut isCheckingCatling de la classe King a été ajouté car il permet d'éviter un

appel en boucle infini lors de l'algorithme de détection d'échec. En effet, pour savoir si un roi est échec on doit analyser tous les mouvements possibles des pièces adverses ainsi en analysant le roi de couleur adverse cela déclenchait l'appel récursif. Ainsi, dans la méthode nous passons ce booléen à vrai pour ne pas rappeler cette méthode lors de l'analyse des mouvements du roi adverse.

Les attributs hasMoved de King et Rook sont utilisés pour analyser si le mouvement de roque est disponible ou non.

Dans la classe "Move", nous stockons le pion pris en passant. En effet, cela nous permet de pouvoir annuler le mouvement dans le cas où une prise en passant met en échec son propre roi.

Dans la classe Board, nous stockons un attribut de l'énumération playerColor pour savoir à qui est le tour.

Nous avons choisi de réutiliser les énumérations PieceType et PlayerColor dans nos classes. Nous sommes conscients que ces classes sont utilisées dans la vue. Ainsi, cela nous rend fortement dépendant de celle-ci pour que le jeu d'échec fonctionne. Si on avait voulu ne pas se rendre dépendant d'elle, nous aurions dû recréer ces énumérations qui ne seraient pas utilisées dans la vue mais seulement dans le contrôleur (la board).

• FONCTIONNEMENT DU PROGRAMME

Note : tout le code source se trouve en annexe du rapport

Lors d'une nouvelle partie, la méthode newGame de la classe Board instancie un nouveau tableau de 8x8 cases ainsi qu'une nouvelle liste de pièces.

Les pièces sont ensuite instanciées, positionnées sur la board, affichées et stockées dans la liste de pièce à l'aide de la méthode addPieceDefaultPosition.

• VÉRIFIER SI UN MOUVEMENT EFFECTUÉ EST VALIDE

Lorsqu'un mouvement est joué sur la vue, la méthode move de la board est appelée. Celle-ci recherche la pièce concernée par le déplacement. Si une pièce est concernée, alors la méthode demande la liste des mouvements que peut effectuer la pièce. Si le mouvement joué par le joueur correspond à mouvement disponible

pour la pièce, alors le déplacement de la pièce est effectué.

Lorsqu'un joueur est échec, la méthode valide également que le mouvement joué par le joueur enlève l'échec. Si ce n'est pas le cas, le mouvement est annulé. Il en va de même lorsqu'un joueur se met en échec en bougeant un de ses pions. Ce mouvement n'est au même titre pas valide et est annulé par la méthode.

- JOUER CHACUN SON TOUR

Pour jouer chacun son tour, en commençant par le joueur blanc, nous avons ajouté un attribut dans la Board qui vérifie que lorsqu'un mouvement est effectué, le pion joué correspond au joueur qui a le tour. Nous avons utilisé le message de la vue pour afficher quel joueur doit jouer à l'aide de la méthode suivante :

```
private void setState() {
    StringBuilder sb = new StringBuilder();
    sb.append("The next player is : ");
    sb.append(currentPlayer);
    if (isCheck)
        sb.append(" / Check !");
    view.displayMessage(sb.toString());
}
```

De plus, celle-ci affiche si un joueur est échec à l'aide de l'attribut isCheck stocké dans la classe.

- OBTENIR LA LISTE DES MOUVEMENTS DE CHAQUE PIÈCE

Pour toutes les pièces sauf les pions, nous avons seulement deux méthodes disponibles dans la classe Piece. Il suffit de varier les paramètres de celle-ci pour obtenir la liste des mouvements. Les méthodes sont les suivantes :

```
protected void addSimpleMoveWithChoosedVariation(int valX, int valY,
List<Move> moves) {
    Tile currentTile = tile;
    int x = tile.getPosX();
    int y = tile.getPosY();

    if (!valid(x + valX, y + valY))
        return;

    Tile newTile = board.getTile(x + valX, y + valY);
    if (!newTile.isOccupied() || newTile.getPiece().playerColor != p
        moves.add(new Move(currentTile, newTile, this));
}
```

Cette méthode permet de déplacer un pion en fonction de variations en x et en y données et d'ajouter les mouvements à la liste des mouvements disponibles.

```
protected void addMovesWithChoosedVariations(int valX, int valY, Boa
List<Move> moves) {
    Tile currentTile = tile;
    int x = tile.getPosX();
    int y = tile.getPosY();

    if (!valid(x + valX, y + valY))
        return;

    Tile newTile;
    boolean stop = false;
    for (int i = x + valX, j = y + valY; i >= 0 && j >= 0 && i <= 7
!stop; i += valX, j += valY) {
        newTile = board.getTile(i, j);

        if (!newTile.isOccupied())
            moves.add(new Move(currentTile, newTile, this));
        else if (newTile.getPiece().playerColor != playerColor) { //
par une pièce adverse
            moves.add(new Move(currentTile, newTile, this));
            stop = true;
        } else // Case occupée par une pièce du joueur
            stop = true;
    }
}
```

Cette seconde méthode permet de trouver les positions disponibles et les ajouter à la liste des mouvements. Cependant, contrairement à la première méthode, celle-ci effectue une boucle faisant varier les pions jusqu'à ce qu'une position soit occupée ou ne soit pas valide.

Concernant les pions, les algorithmes ne sont pas identiques et nous n'avons donc pas pu utiliser les méthodes présentées ci-dessus. Par exemple, un pion ne peut se déplacer en diagonale si et seulement s'il mange un pion adverse. Ainsi, nous avons créé des méthodes dans la classe Pawn qui prennent en paramètre les variations de manière identique aux méthode présentées ci-dessus mais avec l'algorithme adapté aux pions.

- VÉRIFIER L'ÉCHEC

Pour savoir si le joueur a été mis échec par le joueur ayant effectué le dernier déplacement, nous avons stocké un booléen isCheck dans la classe Board.

A chaque tour nous pouvons ainsi vérifer si un mouvement est valide ou non en fonction de si le joueur était échec ou non.

Ensuite, nous avons une méthode canBeKilled qui est la suivante :

```
public boolean canBeKilled(PlayerColor playerColor, Tile tile) {
    // Ajout de tous les mouvements futurs possibles de l'adversaire
    ArrayList<Move> moves = new ArrayList<>();
    for (Piece piece : piecesOnBoard)
        if (piece.getPlayerColor() != playerColor)
            moves.addAll(piece.getMoves(this));

    // On regarde si un mouvement peut tuer la case donnée
    for (Move move : moves)
        if (move.getTo().equals(tile))
            return true;

    return false;
}
```

Ainsi, pour vérifier l'échec nous appelons celle-ci ainsi :

```
private boolean isCheck(PlayerColor playerColor){
    King kingToCheck = playerColor == PlayerColor.WHITE ? whiteKing
    return canBeKilled(kingToCheck.getPlayerColor(), kingToCheck.get
}
```

Nous avions au départ pensé à un algorithme qui regarde dans toutes les directions depuis le roi pour voir si un pion peut le mettre échec. Cependant, cela ne nous permettait pas d'empêcher le joueur de se mettre soi-même échec. C'est pour cette raison que nous avons utiliser l'algorithme qui consiste à parcourir toutes les pièces adverses et lister sa liste de mouvement afin de vérifier si un d'eux peut tueur le roi.

Note : la méthode canBeKilled est également utilisée pour vérifier le grand/petit roque.

- PRISE EN PASSANT

Pour la prise en passant, c'est au moment de l'obtention de la liste des mouvements que si c'est un pion qui à bouger, nous ajoutons ci-disponible le coup de la prise en passant :

```
// Ajout du mouvement en passant si disponible
if (piece.getPieceType() == PieceType.PAWN) {
    Move enPassantMove = ((Pawn) piece).enPassant(lastMove, this);
    if (enPassantMove != null) {
        availableMoves.add(enPassantMove);
        if (move.equals(enPassantMove))
            move = enPassantMove;
    }
}
```

Pour ceci, nous appelons la méthode enPassant du pion. Celle-ci prend en paramètre le dernier mouvement joué qui permet de savoir s'il s'agit d'un mouvement étant susceptible d'être pris en passant. Ce dernier mouvement joué est un attribut privé qui est stocké dans la classe et mis à jour lors de tous les mouvements.

```
public Move enPassant(Move lastMove, Board board) {
    if (lastMove == null || lastMove.getPiece().getPieceType() != Pi
        return null;

    // Vérification que les conditions de la prise en passant sont r
    if (lastMove.nbTileMovedOnY() == 2) {
        if (playerColor == PlayerColor.BLACK && tile.getPosY() == 3
Math.abs(lastMove.getTo().getPosX() - tile.getPosX()) == 1)
            return new Move(tile, board.getTile(lastMove.getTo().get
this, lastMove.getPiece());
        else if (tile.getPosY() == 4 && Math.abs(lastMove.getTo().ge
tile.getPosX()) == 1)
            return new Move(tile, board.getTile(lastMove.getTo().get
this, lastMove.getPiece());
    }

    return null;
}
```

Cette méthode disponible dans la classe Pawn vérifie les conditions de la prise en

passant et si celles-ci sont respectées, elle retourne le mouvement de prise.

- PROMOTION DE PION

Un pion est promu s'il arrive à traverser l'échiquier. Ainsi, pour vérifier si un pion est arrivé de l'autre côté du plateau nous avons implémenter la méthode suivante dans la classe Pawn :

```
public boolean canBePromoted() {
    int y = super.tile.getPosY();
    return y == (playerColor.equals(PlayerColor.BLACK) ? 0 : 7);
}
```

Cette méthode est appelée lors d'un mouvement et appelle pawnPromotion :

```
// Promotion de pion
if (piece.getClass() == Pawn.class && ((Pawn) piece).canBePromoted()
    piece = pawnPromotion((Pawn) piece, getTile(toX, toY));
```

```
private Piece pawnPromotion(Pawn pawn, Tile tile) {
    ChessView.UserChoice choice =
            view.askUser("Pawn promotion", "In which type of piece d
your pawn to be promoted ? ",
                    new ChessView.UserChoice() {
                        @Override
                        public String textValue() {
                            return "Bishop";
                        }

                        @Override
                        public String toString() {
                            return "Bishop";
                        }
                    },
                    new ChessView.UserChoice() {…},
                    new ChessView.UserChoice() {…},
                    new ChessView.UserChoice() {…});

    Piece newPiece = new Bishop(pawn.getPlayerColor());
    switch (choice.toString()) {
        case "Knight":
            newPiece = new Knight(pawn.getPlayerColor());
            break;
        case "Queen":
            …
        case "Rook":
            …
    }

    pawn.wipeOff(view);
    piecesOnBoard.remove(tile.removePiece());
    piecesOnBoard.add(newPiece);
    tile.setPiece(newPiece);
    newPiece.display(view);
    return newPiece;
}
```

La méthode donne une liste de choix à l'aide de classe internes ChessView.UserChoice. En fonction du retour du choix de l'utilisateur, le pion est remplacé par le choix de l'utilisateur. Pour ceci, la nouvelle pièce est affichée et ajouter à la liste de pions contrairement au pion qui est supprimé de la liste des pions et enlever de l'affichage.

- GRAND ET PETIT ROQUE

Tout d'abord, pour le grand et le petit roque nous avons utilisé des attributs booléens hasMoved dans les rois et les tours qui sont mis à jour à vrai dès lors que

la pièce bouge.

Ensuite, pour savoir si le mouvement de roque est disponible nous avons la méthode suivante dans la classe King :

```
private void addCastlingMove(boolean isSmallCastling, Board board, I
moves) {
    isCheckingCastling = true; // Utilisée pour éviter un appel infi
validation du roque si les deux joueurs peuvent roquer en même temps

    // Instanciation des positions d'arrivée du roi
    int posX = 2;
    if (isSmallCastling)
        posX = 6;

    int posY = 0;
    if (playerColor == PlayerColor.BLACK)
        posY = 7;

    // Choix de la tour en fonction de la destination du roi choisie
    Piece rook = isSmallCastling ? board.getTile(7, posY).getPiece()
board.getTile(0, posY).getPiece();

    // Vérification si les bons pions sont aux bonnes cases et qu'il
bougé
    if (hasMoved || rook == null || rook.getPieceType() != PieceType
((Rook) rook).hasMoved() || board.isCheck())
        return;

    // Vérification que les cases concernées par le roque ne sont pa
    int[] posToValidate = isSmallCastling ? new int[]{5, 6} : new in
    for (int pos : posToValidate)
        if (board.getTile(pos, posY).isOccupied())
            return;

    // Vérification que les cases par lesquelles le roi passe par le
peuvent pas être prises
    // deuxième boucle pour réduire la complexité car on vérifie que
vides avant de générer tous les coups possibles de l'adversaire
    posToValidate = isSmallCastling ? new int[]{5, 6} : new int[]{2,
    if (!isCheckingCastling)
        for (int pos : posToValidate)
            if (board.canBeKilled(playerColor, board.getTile(pos, po
                return;

    moves.add(new Move(tile, board.getTile(posX, posY), this, isSmal
!isSmallCastling));
    isCheckingCastling = false;
}
```

C’est ensuite la board qui regarde si le mouvement de roque est joué et si c’est la cas, la tour est déplacée en premier :

```
// Si petit roque, il faut déplacer la tour en premier
if(move.isSmallCastling())
    makeMovement(new Move(getTile(7, fromY), getTile(5, fromY), getT
fromY).getPiece()));

// Si grand roque, il faut déplacer la tour en premier
if(move.isBigCastling())
    makeMovement(new Move(getTile(0, fromY), getTile(3, fromY), getT
fromY).getPiece()));
```

Les méthodes isSmallCastling et isBigCastling de la classe Move retournent si les attributs booléens privé de celle-ci sont à vrai. Ceux-ci sont définis à vrai lors de l'ajout du mouvement de roque lors de l'appel d'un constructeur adapté.

- CHOIX D'IMPLÉMENTATION

Nous avons choisi de réutiliser les énumérations PieceTypes et PlayerColors dans nos classes. Nous sommes conscients que ces classes sont utilisées dans la vue. Ainsi, cela nous rend fortement dépendant de la vue pour que le jeu d'échec fonctionne. Si on avait voulu ne pas se rendre dépendant de celle-ci, nous aurions dû recréer ces énumérations qui ne seraient pas utilisées dans la vue mais seulement dans le contrôleur.

- TESTS RÉALISÉS
  - VÉRIFICATION DU RESPECT DES RÈGLES DU JEU

L'ensemble des fonctionnalités propres au jeu énoncées ci-dessous ont été testées dans la GUI et dans la console.

PRISE EN PASSANT :

Si et seulement si on prend un pion en passant qui a avancé de deux cases lors du dernier mouvement	Un attribut est stocké dans la classe Board pour conserver le dernier mouvement joué. Une méthode dans Pion permet d'ajouter un mouvement dans la liste de ceux disponible en cas de prise en passant.

PROMOTION DE PION :

En reine, en tour, en chevalier, en fou	Une méthode pawnPromotion permet de remplacer un pion sur la Board par la pièce choisie par l'utilisateur. La vérification de l'échec s'effectue après la promotion.

PETIT / GRAND ROQUE :

La tour et le roi ne doivent pas avoir bougé avant	Un attribut dans la classe Rook et King est mis à vrai dès lors qu'un mouvement est effectué par la pièce.

Les cases sont toutes libres aux positions concernées par le roque	L'algorithme commence par vérifier que les cases sont libres en les parcourant et en appelant la méthode « isOccupied » sur celles-ci.
Le roi ne peut pas passer par une case qui est échec lors de son déplacement	Pour toutes les cases par lesquelle le roi passe, on appelle la méthode « canBeKilled ».
MOUVEMENT DE LA TOUR :	
Ne peut pas sauter par-dessus un autre pion	L'algorithme de recherche des mouvements disponible s'arrête de rechercher dans une direction dès qu'il rencontre une pièce. Si c'est une pièce adverse, on peut la manger ; sinon, on ne peut pas aller sur cette case.
MOUVEMENT DU CHEVALIER :	
Mouvements en « L »	L'algorithme « addSimpleMoveWithChoosedVariation » permet de se rendre aux cases en « L » en vérifiant que ces cases existent.
Peut sauter par-dessus un pion	Oui, car l'algorithme ne se soucie pas des cases par lesquelles la variation est effectuée.
MOUVEMENT DU FOU :	
Mouvement en diagonale sur une case qui n'est pas occupée	Méthode « addMovesWithChoosedVariation » pour les directions autorisées. Si c'est une pièce adverse, on peut la manger ; sinon, on ne peut pas aller sur cette case.
Ne peut pas sauter par-dessus un pion	L'algorithme de recherche des mouvements disponible s'arrête de rechercher dans une direction dès qu'il rencontre une pièce. Si c'est une pièce adverse, on peut la manger ; sinon, on ne peut pas aller sur cette case.
MOUVEMENT DU ROI :	
Sur toutes les cases adjacentes qui ne sont pas occupées	Méthode « AddSimpleMoveWithChoosedVariation ». Si c'est une pièce adverse, on peut la manger ; sinon, on ne peut pas aller sur cette case.
MOUVEMENT DE LA REINE :	
Mouvement en diagonale sur une case qui n'est pas occupée	Méthode « addMovesWithChoosedVariation » pour les directions autorisées. Si c'est une pièce adverse, on peut la manger ; sinon, on ne peut pas aller sur cette case.
Mouvement horizontal et vertical sur une case qui n'est pas occupée	Méthode « addMovesWithChoosedVariation » pour les directions autorisées. Si c'est une pièce adverse, on peut la manger ; sinon, on ne peut pas aller sur cette case.

Ne peut pas sauter par-dessus un pion	L'algorithme de recherche des mouvements disponible s'arrête de rechercher dans une direction dès qu'il rencontre une pièce. Si c'est une pièce adverse, on peut la manger ; sinon, on ne peut pas aller sur cette case.
MOUVEMENT DU PION :	
Avancer d'une case sur une case qui n'est pas occupée	Méthode dédiée qui vérifie que la case n'est pas occupée.
Manger en diagonale	Méthode dédiée qui vérifie que la case de destination mange absolument une pièce adverse.
Avancer de deux cases la première fois si elles ne sont pas occupées	Méthode dédiée qui vérifie que la case par laquelle on passe est libre.
Ne peut pas reculer	Les méthodes font en sorte de ne pas ajouter un mouvement du reculons
AUTRES RÈGLES :	
Un joueur met en échec l'adversaire	Voir chapitre 3.43.4
Le joueur se met en échec lui même	On effectue la vérification de l'échec après le mouvement. On annule le mouvement s'il se met échec.
Un joueur ne peut pas manger un de ses pions	Les méthodes « getMoves » des différentes pièces n'autorisent pas un mouvement sur une case occupée par une pièce de même couleur.
Manger un pion adverse	Les méthodes « getMoves » des différentes pièces autorisent un mouvement sur une case occupée par une pièce de couleur adverse.

- VALIDITÉ DES PARAMÈTRES

Dans toutes les méthodes publiques de toutes les classes de notre programme, nous avons validé les paramètres en pris en entrée. Voici un exemple pour le constructeur de mouvement :

```
public Move(Tile from, Tile to, Piece piece, Piece enPassantPiece) {
    if(from == null || to == null || piece == null)
        throw new RuntimeException("Mouvement invalide");

    this.from = from;
    this.to = to;
    this.piece = piece;
    this.enPassantPiece = enPassantPiece;
}
```

Ici, seul le paramètre enPassantPiece peut être nul ; tous les autres paramètres

sont validés et une exception est levée en cas de non-respect de la cohérence des paramètres.

Concernant les méthodes privées, nous sommes partis du postulat que nous nous faisons confiance comme quoi les paramètres que l'on passe sont correctes. En effet, nous seuls les seuls responsables de notre classe et les méthodes ne peuvent pas être accédées en dehors d'elle.

- PROGRAMME DE TESTS

Un programme de tests complet réalisé en Java est disponible en annexe.

- CONCLUSION

Pour ce laboratoire, nous avons commencé par réaliser le schéma UML de notre jeu d'échec pour ne pas commencer d'implémenter le code et de partir dans une mauvaise direction. Une fois l'UML réalisé et validé par l'assistant du cours, nous sommes partis sur l'implémentation.

Nous avons commencé par réaliser les différentes classes que l'on avait prévu dans l'UML avec leurs attributs. Nous avons ensuite créé les différentes méthodes des classes. Afin de pouvoir tester l'implémentation des méthodes au fur et à mesure de l'avance de notre programme, nous avons mis en place l'utilisation de la GUI donnée. A ce moment-là, nous avons rencontré une erreur car la GUI ne parvenait pas à trouver les images dans les sources compilées du programme. L'erreur provenait du fait que nous n'avions pas inclus les images en tant que ressource du projet Maven.

Nous avons passé beaucoup de temps sur la recherche d'un bon algorithme pour chaque pièce afin d'identifier les positions disponibles auxquelles chaque pièce du jeu peut se rendre. Nous avons finalement créé le même algorithme avec des paramètres pour traiter le déplacement d'un maximum de pièces à l'aide du même code.

Une autre grande partie du temps a été passée sur la réalisation des coups spéciaux tels que la prise en passant ou le grand et le petit roque.

Finalement, un fois l'implémentation terminée et les tests réalisés en profondeur, nous avons mis à jour l'UML. Nous étions surpris en bien de notre réflexion par rapport à celle réalisée en code. En effet, nous n'avons pas eu à ajouter d'autres classes. Cependant, nous n'avions pas identifier tous les attributs et les méthodes dont nous aurions besoin.

Le bilan de ce projet est très positif. En effet, nous avons énormément appris de la réalisation de ce jeu d'échec. Nous pensons qu'il s'agit d'un des laboratoires les plus intéressant que nous avons réalisé jusqu'à présent et nous espérons que les élèves des autres années pourront aussi réaliser ce travail intéressant avec une application

concrète.

- ANNEXES

ANNEXE 1 – SCHÉMA UML

ANNEXE 2 – PROGRAMME DE TESTS

ANNEXE 3 - CODE SOURCE DU PROGRAMME