~/Documents/HEIG/24-25/POO/Lab08/all.java

```java
package chess;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.pieces.King;
import chess.pieces.Piece;

import java.util.ArrayList;
import java.util.List;

/**
 * Représente l'échiquier du jeu d'échecs.
 * Contient les cases, les pièces, et les fonctionnalités permettant de manipuler les pièces et
 * de vérifier des états spécifiques comme les cases sous attaque.
 */
public class Board {


    private static final int NB_SQUARES_PER_LINE = 8;
    private final Square[][] board = new Square[NB_SQUARES_PER_LINE][NB_SQUARES_PER_LINE];
    private GameController gameController; // Référence au contrôleur

    /**
     * Constructeur de la classe Board.
     * Initialise l'échiquier avec des cases vides.
     */
    public Board() {
        reset();
    }

    /**
     * Définit le contrôleur du jeu pour ce plateau.
     *
     * @param gameController l'instance du contrôleur de jeu.
     */
    public void setGameController(GameController gameController) {
        this.gameController = gameController;
    }

    /**
     * Obtient le contrôleur de jeu associé au plateau.
     *
     * @return l'instance du contrôleur de jeu.
     */
    public GameController getGameController() {
        return gameController;
    }

    /**
     * Récupère une pièce à une position donnée sur l'échiquier.
     *
     * @param x la position horizontale (0-7).
     * @param y la position verticale (0-7).
     * @return la pièce à la position donnée, ou null si la case est vide.
     * @throws IllegalArgumentException si les coordonnées sont hors limites.
     */
    public Piece getPiece(int x, int y) {
        if (x < 0 || x >= NB_SQUARES_PER_LINE || y < 0 || y >= NB_SQUARES_PER_LINE) {
            throw new IllegalArgumentException("Les coordonnées sont hors du plateau.");
```

```java
    }
    return board[x][y].getPiece();
}

/**
 * Déplace une pièce sur l'échiquier vers une destination donnée.
 *
 * @param piece la pièce à déplacer.
 * @param destination la case cible.
 */
public void movePiece(Piece piece, Square destination) {
    Square currentSquare = piece.getSquare();
    currentSquare.setPiece(null);
    destination.setPiece(piece);
}

/**
 * Supprime une pièce capturée de la case donnée.
 *
 * @param destination la case où une pièce est capturée.
 */
public void removeCapturedPiece(Square destination) {
    if (destination.isOccupied()) {
        destination.setPiece(null);
    }
}

/**
 * Réinitialise l'échiquier en vidant toutes les cases.
 */
public void reset() {
    for (int i = 0; i < NB_SQUARES_PER_LINE; ++i) {
        for (int j = 0; j < NB_SQUARES_PER_LINE; ++j) {
            board[i][j] = new Square(i, j);
        }
    }
}

/**
 * Récupère une case spécifique à une position donnée.
 *
 * @param x la position horizontale (0-7).
 * @param y la position verticale (0-7).
 * @return la case à la position donnée.
 * @throws IllegalArgumentException si les coordonnées sont hors limites.
 */
public Square getSquare(int x, int y) {
    if (x < 0 || x >= NB_SQUARES_PER_LINE || y < 0 || y >= NB_SQUARES_PER_LINE) {
        throw new IllegalArgumentException("Les coordonnées sont hors du plateau.");
    }
    return board[x][y];
}

/**
 * Vérifie si une case est sous attaque par une pièce ennemie.
 *
 * @param square la case à vérifier.
 * @param color la couleur du joueur qui défend la case.
 * @return true si la case est attaquée, false sinon.
 */
public boolean isSquareUnderAttack(Square square, PlayerColor color) {
    List<Piece> enemyPieces = getAllPiecesOfColor(color.opposite());

    for (Piece piece : enemyPieces) {
```

```
            if (piece.canMove(this, piece.getSquare(), square)) {
                return true;
            }
        }
        return false;
    }

    /**
     * Récupère toutes les pièces d'une couleur donnée, sauf les rois.
     *
     * @param color la couleur des pièces à récupérer.
     * @return une liste contenant toutes les pièces de la couleur donnée.
     */
    private List<Piece> getAllPiecesOfColor(PlayerColor color) {
        List<Piece> pieces = new ArrayList<>();
        for (int i = 0; i < NB_SQUARES_PER_LINE; ++i) {
            for (int j = 0; j < NB_SQUARES_PER_LINE; ++j) {
                Piece piece = board[i][j].getPiece();
                if (piece != null && piece.getColor() == color && !(piece instanceof King)) {
                    pieces.add(piece);
                }
            }
        }
        return pieces;
    }
}
```

```
package chess;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.pieces.*;

/**
 * Contrôleur principal pour le jeu d'échecs.
 * Gère les interactions entre le modèle (échiquier et pièces), la vue (interface utilisateur)
 * et la logique de jeu (mouvements, vérification des règles, etc.).
 */
public class GameController implements ChessController {

    private Board board;
    private ChessView view;
    private Move lastMove; // Attribut pour garder en mémoire le dernier coup joué

    /**
     * Constructeur de la classe GameController.
     * Initialise le plateau de jeu et le dernier coup joué.
     */
    public GameController() {
        board = new Board();
        lastMove = null;
        board.setGameController(this);// Initialisation du dernier coup joué
    }

    /**
     * Démarre une nouvelle partie et initialise la vue.
     *
     * @param view l'interface utilisateur pour le jeu.
     */
    @Override
```

```java
    public void start(ChessView view) {
        this.view = view;
        view.startView();
        this.newGame();
    }

    /**
     * Vérifie si le roi d'une couleur donnée est en échec.
     *
     * @param color la couleur du roi à vérifier.
     * @return true si le roi est en échec, false sinon.
     */
    public boolean isKingInCheck(PlayerColor color) {
        Square kingSquare = findKing(color);
        return board.isSquareUnderAttack(kingSquare, color);
    }

    /**
     * Recherche la position du roi d'une couleur donnée.
     *
     * @param color la couleur du roi à localiser.
     * @return la case où se trouve le roi.
     * @throws IllegalStateException si le roi n'est pas trouvé sur l'échiquier.
     */
    private Square findKing(PlayerColor color) {
        for (int x = 0; x < 8; x++) {
            for (int y = 0; y < 8; y++) {
                Piece piece = board.getSquare(x, y).getPiece();
                if (piece != null && piece.getType() == PieceType.KING && piece.getColor() == color) {
                    return board.getSquare(x, y);
                }
            }
        }
        throw new IllegalStateException("King not found on the board");
    }

    /**
     * Tente de déplacer une pièce d'une case à une autre.
     *
     * @param fromX la coordonnée x de la case de départ.
     * @param fromY la coordonnée y de la case de départ.
     * @param toX la coordonnée x de la case de destination.
     * @param toY la coordonnée y de la case de destination.
     * @return true si le mouvement est valide et exécuté, false sinon.
     */
    @Override
    public boolean move(int fromX, int fromY, int toX, int toY) {
        Square fromSquare = board.getSquare(fromX, fromY);
        Square toSquare = board.getSquare(toX, toY);

        if (fromSquare.getPiece() == null || (lastMove == null && fromSquare.getPiece().getColor() !=
PlayerColor.WHITE) || lastMove != null && !lastMove.getPiece().isNotSameColor(fromSquare.getPiece())) {
            return false;
        }

        // Vérification des conditions de déplacement
        if (fromSquare.isOccupied()) {
            Piece piece = fromSquare.getPiece();
            if (piece.canMove(board, fromSquare, toSquare)) {

                //Simuler le mouvement
                Piece capturedPiece = toSquare.getPiece();
                board.movePiece(piece, toSquare);
                boolean isInCheck = isKingInCheck(piece.getColor());
```

```
            //Annuler le mouvement
            board.movePiece(piece, fromSquare);
            toSquare.setPiece(capturedPiece);

            if (!isInCheck) {
                piece.executeMove(board, fromSquare, toSquare);

                // Mise à jour du dernier coup joué
                lastMove = new Move(fromSquare, toSquare, piece);

                view.removePiece(fromX, fromY);
                view.putPiece(toSquare.getPiece().getType(), toSquare.getPiece().getColor(), toX, toY);

                return true;
            }
        }
    }
    return false;
}

/**
 * Retire une pièce de la vue à une position donnée.
 *
 * @param x la coordonnée x de la pièce à retirer.
 * @param y la coordonnée y de la pièce à retirer.
 */
public void removePiece(int x, int y) {
    view.removePiece(x, y);
}

/**
 * Place une pièce dans la vue à une position donnée.
 *
 * @param x la coordonnée x de la pièce à placer.
 * @param y la coordonnée y de la pièce à placer.
 */
public void setPiece(int x, int y) {
    view.putPiece(board.getPiece(x, y).getType(), board.getPiece(x, y).getColor(), x, y);
}

/**
 * Démarre une nouvelle partie en réinitialisant le plateau et les pièces.
 */
@Override
public void newGame() {
    board.reset();// Réinitialisation de l'échiquier
    setLastMoveAtNull();

    // Placement des pièces blanches
    for (int i = 0; i < 8; i++) {
        board.getSquare(i, 1).setPiece(new Pawn(PlayerColor.WHITE));
    }
    board.getSquare(0, 0).setPiece(new Rook(PlayerColor.WHITE));
    board.getSquare(7, 0).setPiece(new Rook(PlayerColor.WHITE));
    board.getSquare(1, 0).setPiece(new Knight(PlayerColor.WHITE));
    board.getSquare(6, 0).setPiece(new Knight(PlayerColor.WHITE));
    board.getSquare(2, 0).setPiece(new Bishop(PlayerColor.WHITE));
    board.getSquare(5, 0).setPiece(new Bishop(PlayerColor.WHITE));
    board.getSquare(3, 0).setPiece(new Queen(PlayerColor.WHITE));
    board.getSquare(4, 0).setPiece(new King(PlayerColor.WHITE));

    // Placement des pièces noires
    for (int i = 0; i < 8; i++) {
```

```
            board.getSquare(i, 6).setPiece(new Pawn(PlayerColor.BLACK));
        }
        board.getSquare(0, 7).setPiece(new Rook(PlayerColor.BLACK));
        board.getSquare(7, 7).setPiece(new Rook(PlayerColor.BLACK));
        board.getSquare(1, 7).setPiece(new Knight(PlayerColor.BLACK));
        board.getSquare(6, 7).setPiece(new Knight(PlayerColor.BLACK));
        board.getSquare(2, 7).setPiece(new Bishop(PlayerColor.BLACK));
        board.getSquare(5, 7).setPiece(new Bishop(PlayerColor.BLACK));
        board.getSquare(3, 7).setPiece(new Queen(PlayerColor.BLACK));
        board.getSquare(4, 7).setPiece(new King(PlayerColor.BLACK));

        // Mise à jour de la vue
        for (int x = 0; x < 8; x++) {
            for (int y = 0; y < 8; y++) {
                Piece piece = board.getSquare(x, y).getPiece();
                if (piece != null) {
                    view.putPiece(piece.getType(), piece.getColor(), x, y);
                }
            }
        }
    }

    /**
     * Gère la promotion d'un pion.
     * Demande à l'utilisateur de choisir une pièce pour remplacer le pion.
     *
     * @param square la case où le pion est promu.
     */
    public void promotePawn(Square square) {
        PromotablePiece[] promotionChoices = {new PromotablePiece(PieceType.QUEEN), new
PromotablePiece(PieceType.ROOK), new PromotablePiece(PieceType.BISHOP), new PromotablePiece(PieceType.KNIGHT)}
;
        PromotablePiece choice = this.view.askUser("Promotion", "Choisissez une pièce pour la promotion :",
promotionChoices);
        switch (choice.getPieceType()) {
            case ROOK:
                square.setPiece(new Rook(square.getPiece().getColor()));
                break;
            case BISHOP:
                square.setPiece(new Bishop(square.getPiece().getColor()));
                break;
            case KNIGHT:
                square.setPiece(new Knight(square.getPiece().getColor()));
                break;
            default:
                square.setPiece(new Queen(square.getPiece().getColor()));
                break;
        }
    }

    /**
     * Classe interne représentant un mouvement.
     */
    public static class Move {
        private final Square from;
        private final Square to;
        private final Piece piece;

        public Move(Square from, Square to, Piece piece) {
            this.from = from;
            this.to = to;
            this.piece = piece;
        }

        public Square getFrom() {
```

```
            return from;
        }

        public Square getTo() {
            return to;
        }

        public Piece getPiece() {
            return piece;
        }
    }

    /**
     * Obtient le dernier coup joué.
     *
     * @return le dernier mouvement.
     */
    public Move getLastMove() {
        return lastMove;
    }

    /**
     * Réinitialise le dernier mouvement à null.
     */
    public void setLastMoveAtNull() {
        lastMove = null;
    }
}
```

```
package chess;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */


/**
 * Enumération représentant les types de pièces d'échecs.
 */
 public enum PieceType {
  PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING
}
```

```
package chess;

/**
 * @modified by Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

public enum PlayerColor {
    WHITE, BLACK;

    /**
     * Retourne la couleur opposée.
     *
     * @return PlayerColor opposé (WHITE devient BLACK et vice-versa).
     */
    public PlayerColor opposite() {
        return this == WHITE ? BLACK : WHITE;
    }
```

```
}
```

```
package chess;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.pieces.Piece;

/**
 * Représente une case sur l'échiquier.
 * Chaque case est définie par ses coordonnées (x, y) et peut contenir une pièce.
 */
public class Square {
    private final int x;
    private final int y;
    private Piece piece;

    /**
     * Constructeur d'une case à des coordonnées spécifiques.
     *
     * @param x Coordonnée X de la case
     * @param y Coordonnée Y de la case
     */
    public Square(int x, int y) {
        this.x = x;
        this.y = y;
        this.piece = null;
    }

    /**
     * Vérifie si la case est occupée par une pièce.
     *
     * @return true si une pièce occupe la case, false sinon.
     */
    public boolean isOccupied() {
        return piece != null;
    }

    /**
     * Obtient la pièce sur la case.
     *
     * @return La pièce présente ou null si la case est vide.
     */
    public Piece getPiece() {
        return piece;
    }

    /**
     * Place une pièce sur la case.
     * Met également à jour la case associée à la pièce.
     *
     * @param piece La pièce à placer (null pour vider la case).
     */
    public void setPiece(Piece piece) {
        this.piece = piece;
        if (piece != null) {
            piece.setSquare(this);
        }
    }
```

```java
    /**
     * Obtient la coordonnée X de la case.
     *
     * @return La coordonnée X.
     */
    public int getX() {
        return x;
    }

    /**
     * Obtient la coordonnée Y de la case.
     *
     * @return La coordonnée Y.
     */
    public int getY() {
        return y;
    }
}
```

```java
package chess;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.views.console.ConsoleView;
import chess.views.gui.GUIView;

public class Main {
    public static void main(String[] args) {
        ChessController controller = new GameController();
        ChessView view = new GUIView(controller);
        //ChessView view = new ConsoleView(controller); //mode console
        controller.start(view);
    }
}
```

```java
package chess;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import chess.views.console.ConsoleView;

class ChessTests {

    private GameController controller;
    private ChessView view;

    @BeforeEach
    void setUp() {
        controller = new GameController();
        view = new ConsoleView(controller); // Utilisation de ConsoleView comme implémentation concrète
        controller.start(view);
    }

    @Test
    void testStartGame() {
        assertDoesNotThrow(() -> controller.start(view));
    }

    @Test
```

```java
void testMoveValid() {
    assertTrue(controller.move(0, 1, 0, 2));
}

@Test
void testMoveInvalid() {
    assertFalse(controller.move(0, 1, 0, 4));
}

@Test
void testMoveOutOfBounds() {
    assertFalse(controller.move(0, 1, 0, 8));
}

@Test
void testCapturePiece() {
    controller.move(3, 1, 3, 3); // Déplacement d'un pion
    controller.move(4, 6, 4, 4); // Déplacement d'un pion adverse
    assertTrue(controller.move(3, 3, 4, 4)); // Capture
}

@Test
void testNewGame() {
    assertDoesNotThrow(() -> controller.newGame());
}

@Test
void testPawnPromotion() {
    controller.move(6, 1, 6, 3);
    controller.move(7, 6, 7, 4);
    controller.move(6, 3, 6, 4);
    controller.move(7, 4, 7, 3);
    controller.move(6, 4, 6, 5);
    controller.move(7, 3, 7, 2);
    controller.move(6, 5, 6, 6);
    controller.move(7, 2, 7, 1);
    controller.move(6, 6, 6, 7);
    // Promotion simulée
    assertDoesNotThrow(() -> controller.promotePawn(controller.getBoard().getSquare(6, 7)));
}

@Test
void testCastling() {
    controller.move(4, 1, 4, 3);
    controller.move(4, 6, 4, 4);
    controller.move(5, 0, 4, 1);
    controller.move(5, 7, 4, 6);
    controller.move(6, 0, 5, 2);
    controller.move(6, 7, 5, 5);
    assertTrue(controller.move(4, 0, 6, 0)); // Petit roque
    controller.newGame(); // Réinitialisation
    assertTrue(controller.move(4, 0, 2, 0)); // Grand roque
}

@Test
void testKingInCheck() {
    controller.move(4, 1, 4, 3);
    controller.move(3, 6, 3, 4);
    controller.move(5, 0, 2, 3);
    assertTrue(controller.isKingInCheck(PlayerColor.BLACK));
}

@Test
void testCheckmate() {
```

```
        controller.move(5, 1, 5, 3);
        controller.move(4, 6, 4, 4);
        controller.move(6, 1, 6, 3);
        controller.move(3, 7, 7, 3);
        assertTrue(controller.isKingInCheck(PlayerColor.WHITE));
    }
}
```