```java
package engine;

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

/**
 * Classe lançant le programme du jeu d'échecs.
 * Il est possible de jouer via un GUI ou en Console selon les envies.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class Main {
    public static void main(String[] args) {
        ChessController controller = new GameManager();

        // Choix de la vue : mode GUI ou mode Console
        ChessView view = new GUIView(controller);
        // ChessView view = new ConsoleView(controller);

        controller.start(view);
    }
}

// ------------------------------------------------------------------------------------------------

package engine;

import chess.ChessController;
import chess.ChessView;
import chess.views.gui.GUIView;

/**
 * Classe lançant le programme de test du jeu d'échecs.
 * Il est possible de définir quelle position initiale choisir pour les pièces
 * afin de tester une pièce ou un mouvement spécifique.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class MainTest {
    public static void main(String[] args) {
        // Choix du test à lancer
        GameManagerTest.Type type = GameManagerTest.Type.QUEEN;
        ChessController controller = new GameManagerTest(type);

        ChessView view = new GUIView(controller);

        controller.start(view);
    }
}

// ------------------------------------------------------------------------------------------------

package engine;

import chess.ChessController;
import chess.ChessView;
import chess.PlayerColor;
import engine.pieces.*;
import engine.utils.Cell;

import java.util.Objects;
```

```java
/**
 * Classe principale de la gestion du jeu d'échecs.
 * Elle s'occupe de démarrer le jeu ainsi qu'écouter et répondre aux événements de
 * la view.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class GameManager implements ChessController {
    private ChessView view;
    private Board board;

    /**
     * @return le plateau de jeu
     */
    protected Board getBoard() {
        return board;
    }

    /**
     * Met à jour le message de la vue
     */
    private void updateDisplayMessage() {
        if (view == null || board == null)
            return;

        String color = board.currentPlayer() == PlayerColor.WHITE ? "blancs" : "noirs";
        StringBuilder msg = new StringBuilder("Aux " + color);

        if (board.isCheck(board.currentPlayer())) {
            if (board.isCheckMate(board.currentPlayer())) {
                msg.setLength(0);
                String winner = board.currentPlayer() == PlayerColor.WHITE ?
                        "noirs" : "blancs";
                msg.append("CHECKMATE! Les ").append(winner).append(" ont gagnés!");
            } else {
                msg.append(" CHECK!");
            }
        }

        view.displayMessage(msg.toString());
    }
```

```java
    /**
     * Initialise le plateau, écoute les différents événements
     */
    private void initBoard() {
        board = new Board();

        // Events listeners
        board.setAddPieceListener((piece, cell) -> {
            if (view != null)
                view.putPiece(piece.getType(), piece.getColor(), cell.getX(), cell.getY());
        });

        board.setRemovePieceListener((piece, cell) -> {
            if (view != null)
                view.removePiece(cell.getX(), cell.getY());
        });

        board.setPromotionListener((piece) -> {
            Cell cell = piece.getCell();
            PlayerColor color = piece.getColor();
            Piece[] choices = {
                    new Queen(board, cell, color),
                    new Knight(board, cell, color),
                    new Rook(board, cell, color),
                    new Bishop(board, cell, color)
            };

            Piece userChoice;
            while ((userChoice = view.askUser("Promotion",
                    "Choisir une pièce pour la promotion", choices)) == null) {
            }
            board.removePiece(cell);
            board.setPiece(userChoice, cell);
        });
    }

    @Override
    public void start(ChessView view) {
        Objects.requireNonNull(view, "View invalide");
        this.view = view;
        view.startView();
        initBoard();
        board.fillBoard();
        updateDisplayMessage();
    }

    @Override
    public boolean move(int fromX, int fromY, int toX, int toY) {
        if (board == null)
            return false;

        Cell from = new Cell(fromX, fromY);
        Cell to = new Cell(toX, toY);

        boolean canMove = board.move(from, to);
        updateDisplayMessage();

        return canMove;
    }

    @Override
    public void newGame() {
        board.resetBoard();
        board.fillBoard();
        updateDisplayMessage();
    }
}
```

```java
// ----------------------------------------------------------------------------------------------

package engine;

import chess.ChessView;
import chess.PlayerColor;
import engine.pieces.*;
import engine.utils.Cell;

/**
 * Classe permettant de tester rapidement diverses situations initiales.
 * Il est possible de spécifier quelle pièce ou mouvement nous souhaitons tester
 * grâce au constructeur.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class GameManagerTest extends GameManager {
    enum Type {
        CHECK, CHECKMATE, CASTLE, ROOK, BISHOP, KING, KNIGHT, PAWN, QUEEN
    }

    private final Type type;

    /**
     * Crée une instance de test
     *
     * @param type : pièce ou mouvement à tester
     */
    public GameManagerTest(Type type) {
        this.type = type;
    }

    @Override
    public void start(ChessView view) {
        super.start(view);
        newGame();
    }

    @Override
    public void newGame() {
        getBoard().resetBoard();
        // Applique la situation initiale de test
        fillBoard(type);
    }
```

```java
    /**
     * Remplit le plateau selon le test choisi
     *
     * @param type : pièce ou mouvement à tester
     */
    private void fillBoard(Type type) {
        switch (type) {
            case CHECK:
                testCheck();
                break;
            case CHECKMATE:
                testCheckMate();
                break;
            case CASTLE:
                testCastle();
                break;
            case ROOK:
                testRook();
                break;
            case BISHOP:
                testBishop();
                break;
            case KING:
                testKing();
                break;
            case KNIGHT:
                testKnight();
                break;
            case PAWN:
                testPawn();
                break;
            case QUEEN:
                testQueen();
                break;
            default:
                break;
        }
    }

    /**
     * Ajoute des pions de la couleur souhaitée autour de la position spécifiée
     *
     * @param pos   position que l'on souhaite entourer
     * @param color couleur de pions
     */
    private void pawnAroundPos(Cell pos, PlayerColor color) {
        for (int i = pos.getX() - 1; i < pos.getX() + 2; i++) {
            for (int j = pos.getY() - 1; j < pos.getY() + 2; j++) {
                if (i == pos.getX() && j == pos.getY()) {
                    continue;
                }
                getBoard().addPiece(new Pawn(getBoard(), new Cell(i, j), color));
            }
        }
    }
```

```java
    /**
     * Permet de rapidement position des pièces afin de tester le fonctionnement des
     * Cavaliers
     */
    private void testKnight() {
        Cell knight1 = new Cell(5, 5);
        Cell knight2 = new Cell(2, 5);
        Cell knight3 = new Cell(5, 2);
        Cell knight4 = new Cell(2, 2);
        getBoard().addPiece(new Knight(getBoard(), knight1, PlayerColor.WHITE));
        pawnAroundPos(knight1, PlayerColor.BLACK);
        getBoard().addPiece(new Knight(getBoard(), knight2, PlayerColor.WHITE));
        pawnAroundPos(knight2, PlayerColor.WHITE);
        getBoard().addPiece(new Knight(getBoard(), knight3, PlayerColor.BLACK));
        pawnAroundPos(knight3, PlayerColor.WHITE);
        getBoard().addPiece(new Knight(getBoard(), knight4, PlayerColor.BLACK));
        pawnAroundPos(knight4, PlayerColor.BLACK);
    }

    /**
     * Permet de rapidement position des pièces afin de tester le fonctionnement des
     * Tours.
     */
    private void testRook() {
        Cell rookPos1 = new Cell(5, 4);
        Cell rookPos2 = new Cell(2, 4);
        getBoard().addPiece(new Rook(getBoard(), new Cell(0, 0), PlayerColor.WHITE));
        getBoard().addPiece(new Rook(getBoard(), new Cell(7, 7), PlayerColor.BLACK));
        getBoard().addPiece(new Rook(getBoard(), rookPos1, PlayerColor.WHITE));
        getBoard().addPiece(new Rook(getBoard(), rookPos2, PlayerColor.WHITE));
        pawnAroundPos(rookPos1, PlayerColor.WHITE);
        pawnAroundPos(rookPos2, PlayerColor.BLACK);
    }

    /**
     * Permet de rapidement position des pièces afin de tester le fonctionnement des
     * Fous.
     */
    private void testBishop() {
        Cell bishopPos1 = new Cell(5, 4);
        Cell bishopPos2 = new Cell(2, 4);
        getBoard().addPiece(new Bishop(getBoard(), bishopPos1, PlayerColor.WHITE));
        getBoard().addPiece(new Bishop(getBoard(), bishopPos2, PlayerColor.WHITE));
        pawnAroundPos(bishopPos1, PlayerColor.WHITE);
        pawnAroundPos(bishopPos2, PlayerColor.BLACK);
    }

    /**
     * Permet de rapidement position des pièces afin de tester le fonctionnement des
     * Pions.
     */
    private void testPawn() {
        for (int i = 0; i < 7; i++) {
            getBoard().addPiece(new Pawn(getBoard(), new Cell(i, 1),
                    PlayerColor.WHITE));
            getBoard().addPiece(new Knight(getBoard(), new Cell(i, 2), (i % 2 == 0) ?
                    PlayerColor.WHITE : PlayerColor.BLACK));
        }
        getBoard().addPiece(new Pawn(getBoard(), new Cell(7, 1), PlayerColor.BLACK));
        getBoard().addPiece(new Pawn(getBoard(), new Cell(4, 6), PlayerColor.BLACK));
        getBoard().addPiece(new Pawn(getBoard(), new Cell(3, 4), PlayerColor.WHITE));
    }
```

```java
    /**
     * Permet de rapidement position des pièces afin de tester le fonctionnement des
     * Reines.
     */
    private void testQueen() {
        Cell queenPos = new Cell(5, 4);
        Cell queenPos2 = new Cell(2, 4);
        getBoard().addPiece(new Queen(getBoard(), queenPos, PlayerColor.WHITE));
        getBoard().addPiece(new Queen(getBoard(), queenPos2, PlayerColor.WHITE));
        pawnAroundPos(queenPos, PlayerColor.BLACK);
        pawnAroundPos(queenPos2, PlayerColor.WHITE);
    }

    /**
     * Permet de rapidement position des pièces afin de tester le fonctionnement des
     * Rois.
     */
    private void testKing() {
        Cell kingPos = new Cell(4, 4);
        getBoard().addPiece(new King(getBoard(), kingPos, PlayerColor.WHITE));
        pawnAroundPos(kingPos, PlayerColor.BLACK);
    }

    /**
     * Permet de rapidement position des pièces afin de tester le fonctionnement
     * du roque.
     */
    private void testCastle() {
        getBoard().addPiece(new King(getBoard(), new Cell(4, 0), PlayerColor.WHITE));
        getBoard().addPiece(new Rook(getBoard(), new Cell(7, 0), PlayerColor.WHITE));
        getBoard().addPiece(new Rook(getBoard(), new Cell(0, 0), PlayerColor.WHITE));
        getBoard().addPiece(new Queen(getBoard(), new Cell(3, 7), PlayerColor.BLACK));
    }

    /**
     * Permet de rapidement position des pièces afin de tester le fonctionnement
     * de la mise en échec.
     */
    private void testCheck() {
        getBoard().addPiece(new King(getBoard(), new Cell(3, 5), PlayerColor.BLACK));
        getBoard().addPiece(new Rook(getBoard(), new Cell(4, 3), PlayerColor.WHITE));
        getBoard().addPiece(new Queen(getBoard(), new Cell(5, 3), PlayerColor.WHITE));
        getBoard().addPiece(new Bishop(getBoard(), new Cell(6, 3),
                PlayerColor.WHITE));
        getBoard().addPiece(new Knight(getBoard(), new Cell(7, 3),
                PlayerColor.WHITE));
        getBoard().addPiece(new King(getBoard(), new Cell(1, 3), PlayerColor.WHITE));
        getBoard().addPiece(new Pawn(getBoard(), new Cell(2, 3), PlayerColor.WHITE));
    }
}

    /**
     * Permet de rapidement position des pièces afin de tester le fonctionnement
     * de l'echec et mat.
     */
    private void testCheckMate(){
        getBoard().addPiece(new King(getBoard(), new Cell(0, 3), PlayerColor.BLACK));
        getBoard().addPiece(new King(getBoard(), new Cell(2, 3), PlayerColor.WHITE));
        getBoard().addPiece(new Queen(getBoard(), new Cell(3, 5), PlayerColor.WHITE));

    }

// ----------------------------------------------------------------------------------------
```

```java
package engine;

import chess.PieceType;
import chess.PlayerColor;
import engine.pieces.*;
import engine.utils.Cell;

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

/**
 * Classe modélisant un plateau virtuel du jeu d'échecs.
 * Elle s'occupe notamment de stocker et modifier les positions des différentes
 * pièces.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class Board {
    public interface PieceListener {
        void action(Piece piece, Cell cell);
    }

    public interface PromotionListener {
        void action(Piece piece);
    }

    public static final int BOARD_SIZE = 8;
    private int turn;
    private final Piece[][] pieces;
    private final List<King> kings;
    private Piece lastPiecePlayed;

    private PieceListener onAddPiece;
    private PieceListener onRemovePiece;
    private PromotionListener onPromotion;

    /**
     * Constructeur de base initialisant les différentes structures
     */
    public Board() {
        pieces = new Piece[BOARD_SIZE][BOARD_SIZE];
        kings = new ArrayList<>();
    }

    /**
     * Remet le plateau à son état initial
     */
    public void resetBoard() {
        // On vide le plateau pour éviter de recréer un tableau
        for (int i = 0; i < BOARD_SIZE; i++)
            for (int j = 0; j < BOARD_SIZE; j++)
                removePiece(new Cell(i, j));

        kings.clear();
        lastPiecePlayed = null;
        turn = 0;
    }
```

```java
    /**
     * Remplit le tableau avec la position habituelle des différentes pièces
     * Commence par les pièces blanches puis les noires
     */
    public void fillBoard() {
        PlayerColor color = PlayerColor.WHITE;
        int line = 0, pawnLine = 1;
        for (int i = 0; i < 2; i++) {
            addPiece(new Rook(this, new Cell(0, line), color));
            addPiece(new Knight(this, new Cell(1, line), color));
            addPiece(new Bishop(this, new Cell(2, line), color));
            addPiece(new Queen(this, new Cell(3, line), color));
            addPiece(new King(this, new Cell(4, line), color));
            addPiece(new Bishop(this, new Cell(5, line), color));
            addPiece(new Knight(this, new Cell(6, line), color));
            addPiece(new Rook(this, new Cell(7, line), color));

            // Pions
            for (int xPawn = 0; xPawn < BOARD_SIZE; xPawn++)
                addPiece(new Pawn(this, new Cell(xPawn, pawnLine), color));

            color = PlayerColor.BLACK;
            line = 7;
            pawnLine = 6;
        }
    }

    /**
     * Vérifie que les coordonnées de la case sont valides
     *
     * @param cell : case à vérifier
     * @throws RuntimeException si la case est invalide
     */
    private void checkCoordsOnBoard(Cell cell) {
        if (cell == null || cell.getX() >= BOARD_SIZE || cell.getX() < 0 ||
                cell.getY() >= BOARD_SIZE || cell.getY() < 0)
            throw new RuntimeException("Coordonnées de la pièce invalides.");
    }

    /**
     * @return le tour actuel
     */
    public int getTurn() {
        return turn;
    }

    /**
     * @return la dernière pièce jouée
     */
    public Piece getLastPiecePlayed() {
        return lastPiecePlayed;
    }

    /**
     * @param cell : case souhaitée
     * @return la pièce à la case souhaitée ou null
     * @throws RuntimeException si la case est invalide
     */
    public Piece getPiece(Cell cell) {
        checkCoordsOnBoard(cell);
        return pieces[cell.getX()][cell.getY()];
    }
```

```java
    /**
     * Ajoute la pièce à la case souhaitée
     *
     * @param piece : pièce à ajouter
     * @param cell  : case souhaitée
     * @throws RuntimeException si la case est invalide
     */
    public void setPiece(Piece piece, Cell cell) {
        Objects.requireNonNull(piece, "Pièce invalide");
        checkCoordsOnBoard(cell);

        pieces[cell.getX()][cell.getY()] = piece;
        piece.setCell(cell);

        if (piece.getType() == PieceType.KING)
            kings.add((King) piece);

        if (onAddPiece != null)
            onAddPiece.action(piece, cell);
    }

    /**
     * Petite fonction helper permettant d'ajouter une pièce à sa case actuelle
     *
     * @param piece : pièce à ajouter
     * @throws RuntimeException si la case est invalide
     */
    public void addPiece(Piece piece) {
        setPiece(piece, piece.getCell());
    }

    /**
     * Supprime une pièce du tableau
     *
     * @param cell : case de la pièce à supprimer
     * @throws RuntimeException si la case est invalide
     */
    public void removePiece(Cell cell) {
        Piece piece = getPiece(cell);
        pieces[cell.getX()][cell.getY()] = null;

        if (piece != null) {
            if (piece.getType() == PieceType.KING)
                kings.remove((King) piece);

            if (onRemovePiece != null)
                onRemovePiece.action(piece, cell);
        }
    }

    /**
     * @return le joueur à qui c'est le tour de jouer
     */
    public PlayerColor currentPlayer() {
        return turn % 2 == 0 ? PlayerColor.WHITE : PlayerColor.BLACK;
    }
```

```java
    /**
     * Vérifie que le déplacement d'une pièce peut se faire. Si c'est le cas,
     * il est réalisé.
     *
     * @param from : case de départ
     * @param to   : case d'arrivée
     * @return true si le mouvement a pu être fait, false sinon
     */
    public boolean move(Cell from, Cell to) {
        Piece p;
        try {
            p = getPiece(from);
            checkCoordsOnBoard(to);
        } catch (RuntimeException e) {
            return false;
        }

        if (p == null || p.getColor() != currentPlayer())
            return false;

        if (p.checkMove(to) && p.applyMove(to)) {
            postUpdate(p);
            return true;
        }

        return false;
    }

    /**
     * Applique le mouvement d'une pièce à une destination
     *
     * @param piece : pièce à déplacer
     * @param to    : case d'arrivée
     */
    public void applyMove(Piece piece, Cell to) {
        Objects.requireNonNull(piece, "Pièce invalide");
        removePiece(piece.getCell());
        removePiece(to);
        setPiece(piece, to);
    }

    /**
     * Vérifie si une pièce est actuellement menacée/attaquée
     *
     * @param color : couleur de la pièce à vérifier
     * @param cell  : case de la pièce à vérifier
     * @return true si la pièce est attaquée, false sinon
     */
    public boolean isAttacked(PlayerColor color, Cell cell) {
        Objects.requireNonNull(color, "Couleur invalide");
        Objects.requireNonNull(cell, "Case invalide");

        for (Piece[] row : pieces)
            for (Piece piece : row)
                if (piece != null && piece.getColor() != color && piece.checkMove(cell))
                    return true;

        return false;
    }
```

```java
    /**
     * Vérifie si un joueur est actuellement en échec
     *
     * @param color : la couleur du joueur
     * @return true si le joueur est en échec, false sinon
     */
    public boolean isCheck(PlayerColor color) {
        Objects.requireNonNull(color, "Couleur invalide");

        King king = kings.stream()
                .filter(k -> k.getColor() == color)
                .findAny()
                .orElse(null);

        if (king == null)
            return false;

        return isAttacked(color, king.getCell());
    }

    /**
     * Vérifie si un roi déjà en échec et échec et mat ou non
     *
     * @param color : couleur du roi en échec
     * @return true si le roi est échec et mat, false sinon
     */
    public boolean isCheckMate(PlayerColor color) {
        // Boucle sur toutes les cases
        for (int i = 0; i < BOARD_SIZE; i++) {
            for (int j = 0; j < BOARD_SIZE; j++) {
                Cell destination = new Cell(i, j);
                Piece eaten = getPiece(destination);
                // Vérifie qu'une pièce alliée peut bouger
                for (Piece[] row : pieces) {
                    for (Piece piece : row) {
                        if (piece == null || piece.getColor() != color)
                            continue;
                        Cell oldPos = piece.getCell();
                        if (piece.checkMove(destination) && piece.applyMove(destination)) {
                            // On annule le déplacement qui a bien été effectué
                            applyMove(piece, oldPos);
                            if (eaten != null)
                                applyMove(eaten, destination);
                            return false;
                        }
                    }
                }
            }
        }
        return true;
    }

    /**
     * Applique les changements nécessaires à la fin d'un tour
     *
     * @param piece : pièce jouée
     */
    private void postUpdate(Piece piece) {
        Objects.requireNonNull(piece, "Pièce invalide");
        lastPiecePlayed = piece;
        piece.postUpdate();
        turn++;
    }
```

```
829       /**
830        * Définit le listener appelé lors de l'ajout d'une pièce
831        *
832        * @param onAddPiece : listener à exécuter
833        */
834       public void setAddPieceListener(PieceListener onAddPiece) {
835           Objects.requireNonNull(onAddPiece, "Listener invalide");
836           this.onAddPiece = onAddPiece;
837       }
838
839       /**
840        * Définit le listener appelé lors de la suppression d'une pièce
841        *
842        * @param onRemovePiece : listener à exécuter
843        */
844       public void setRemovePieceListener(PieceListener onRemovePiece) {
845           Objects.requireNonNull(onRemovePiece, "Listener invalide");
846           this.onRemovePiece = onRemovePiece;
847       }
848
849       /**
850        * Définit le listener appelé lors de la promotion d'une pièce
851        *
852        * @param onPromotion : listener à exécuter
853        */
854       public void setPromotionListener(PromotionListener onPromotion) {
855           Objects.requireNonNull(onPromotion, "Listener invalide");
856           this.onPromotion = onPromotion;
857       }
858
859       /**
860        * @return le listener appelé lors d'une promotion
861        */
862       public PromotionListener getOnPromotion() {
863           return onPromotion;
864       }
865   }
866
867   // ------------------------------------------------------------------------------------------
868
869   package engine.utils;
870
871   import java.util.Objects;
872
873   /**
874    * Classe représentant une case de l'échiquier
875    *
876    * @author Jonathan Friedli
877    * @author Valentin Kaelin
878    */
879   public class Cell {
880       private final int x;
881       private final int y;
882
883       /**
884        * @param x : coordonnée x de la case
885        * @param y : coordonnée y de la case
886        */
887       public Cell(int x, int y) {
888           this.x = x;
889           this.y = y;
890       }
891
892       /**
893        * @return la coordonnée X de la case
894        */
895       public int getX() {
896           return x;
897       }
```

```java

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

    /**
     * Additionne une seconde case
     *
     * @param cell : la case à ajouter
     * @return le résultat de l'addition via une nouvelle case
     * @throws RuntimeException si la case à additionner est invalide
     */
    public Cell add(Cell cell) {
        if (cell == null)
            throw new RuntimeException("Addition d'une case invalide");

        return new Cell(x + cell.x, y + cell.y);
    }

    /**
     * Soustrait une seconde case
     *
     * @param cell : la case à soustraire
     * @return le résultat de la soustraction via une nouvelle case
     * @throws RuntimeException si la case à soustraire est invalide
     */
    public Cell subtract(Cell cell) {
        if (cell == null)
            throw new RuntimeException("Soustraction d'une case invalide");

        return new Cell(x - cell.x, y - cell.y);
    }


    /**
     * Multiplie la case par un scalaire
     *
     * @param n : scalaire
     * @return le résultat de la multiplication via une nouvelle case
     */
    public Cell multiply(int n) {
        return new Cell(n * x, n * y);
    }

    /**
     * Vérifie qu'une case peut être atteinte depuis une autre
     *
     * @param cell : case de potentielle arrivée
     * @return true si la case est atteignable, false sinon
     */
    public boolean reachable(Cell cell) {
        return cell != null && x * cell.y == y * cell.x;
    }

    /**
     * Vérifie que deux cases ont les mêmes signes sur leurs deux coordonnées
     *
     * @param cell : la seconde case
     * @return true si les signes sont les mêmes, false sinon
     */
    public boolean sameDirection(Cell cell) {
        return cell != null && (x < 0 == cell.getX() < 0) && (y < 0 == cell.getY() < 0);
    }


```

```java
    /**
     * Retourne la distance jusqu'à une case.
     * Ne vérifie pas si la case est accessible.
     *
     * @param to : case d'arrivée
     * @return la distance entre les deux cases
     * @throws RuntimeException si la case d'arrivée est invalide
     */
    public int getDistance(Cell to) {
        Objects.requireNonNull(to, "Case invalide");
        Cell fromTo = to.subtract(this);
        return Math.max(Math.abs(fromTo.getX()), Math.abs(fromTo.getY()));
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Cell cell = (Cell) o;
        return x == cell.x && y == cell.y;
    }
}

// ------------------------------------------------------------------------------------------------


package engine.utils;

/**
 * Énumération permettant de modéliser des déplacements dans une certaine direction
 * Les directions gauches et droites ne sont pas utilisées, mais sont implémentée
 * dans un souci d'harmonisation.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public enum Direction {
    UP(0, 1), DOWN(0, -1), LEFT(-1, 0), RIGHT(1, 0);
    private final Cell value;

    private Direction(int x, int y) {
        this.value = new Cell(x, y);
    }

    /**
     * @return la valeur de la direction sous forme d'une case
     */
    public Cell getValue() {
        return value;
    }

    /**
     * @return la valeur de la direction sous forme d'un nombre
     */
    public int intValue() {
        return value.getX() == 0 ? value.getY() : value.getX();
    }
}

// ------------------------------------------------------------------------------------------------
```

```java
package engine.pieces;

import chess.ChessView;
import chess.PieceType;
import chess.PlayerColor;
import engine.Board;
import engine.moves.Move;
import engine.utils.Cell;

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

/**
 * Classe abstraite permettant de définir la base de toutes les pièces du jeu
 * d'échecs.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public abstract class Piece implements ChessView.UserChoice {
    private final Board board;
    private final PlayerColor color;
    private Cell cell;
    protected List<Move> moves;

    /**
     * Crée une nouvelle pièce
     *
     * @param board : plateau de la pièce
     * @param cell  : case de la pièce
     * @param color : couleur de la pièce
     * @throws RuntimeException s'il manque un paramètre
     */
    public Piece(Board board, Cell cell, PlayerColor color) {
        if (board == null || cell == null || color == null)
            throw new RuntimeException("Construction de la pièce invalide");

        this.board = board;
        this.cell = cell;
        this.color = color;
        moves = new ArrayList<>();
    }

    /**
     * @return le type de la pièce
     */
    public abstract PieceType getType();

    /**
     * @return le texte en français représentant la pièce
     */
    public abstract String textValue();

    /**
     * @return le plateau de la pièce
     */
    public Board getBoard() {
        return board;
    }

    /**
     * @return la couleur de la pièce
     */
    public PlayerColor getColor() {
        return color;
    }
```

```java
    @Override
    public String toString() {
        return textValue();
    }

    /**
     * @return la case de la pièce
     */
    public Cell getCell() {
        return cell;
    }

    /**
     * Change la case de la pièce
     *
     * @param cell : nouvelle case
     * @throws RuntimeException si le case est inexistante
     */
    public void setCell(Cell cell) {
        if (cell == null)
            throw new RuntimeException("Case de la pièce invalide.");

        this.cell = cell;
    }



    /**
     * Vérifie qu'un mouvement peut-être réalisé par la pièce
     *
     * @param to : case de destination souhaitée
     * @return true si le mouvement peut être fait, false sinon
     */
    public boolean checkMove(Cell to) {
        // Si la case de destination est occupée par une pièce de même couleur
        if (to == null || (board.getPiece(to) != null &&
                board.getPiece(to).getColor() == color))
            return false;

        for (Move move : moves) {
            if (move.canMove(cell, to))
                return true;
        }

        return false;
    }

    /**
     * Vérifie qu'un mouvement (légal) peut être appliqué
     *
     * @param to : case de destination souhaitée
     * @return true s'il peut être appliqué, false s'il met le roi du joueur en échec
     */
    public boolean applyMove(Cell to) {
        Cell oldCell = getCell();
        Piece eaten = board.getPiece(to);

        board.applyMove(this, to);

        // En échec : on annule le move
        if (board.isCheck(color)) {
            board.applyMove(this, oldCell);
            if (eaten != null)
                board.setPiece(eaten, to);
            return false;
        }

        return true;
    }
```

```java
    /**
     * Méthode à implémenter dans les pièces devant réaliser des actions après un
     * tour.
     */
    public void postUpdate() {
    }
}

// ----------------------------------------------------------------------------------------------------

package engine.pieces;

import chess.PlayerColor;
import engine.Board;
import engine.utils.Cell;

/**
 * Classe abstraite permettant d'ajouter la gestion de premier coup spécifique à
 * certaines pièces.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public abstract class FirstMoveSpecificPiece extends Piece {
    private boolean hasMoved;

    public FirstMoveSpecificPiece(Board board, Cell cell, PlayerColor color) {
        super(board, cell, color);
        hasMoved = false;
    }

    /**
     * @return true si la pièce a déjà bougé, false sinon
     */
    public boolean hasMoved() {
        return hasMoved;
    }

    /**
     * Indique à la fin du tour que la pièce a déjà bougé
     */
    public void postUpdate() {
        hasMoved = true;
    }
}

// ----------------------------------------------------------------------------------------------------

package engine.pieces;

import chess.PieceType;
import chess.PlayerColor;
import engine.Board;
import engine.moves.LinearMove;
import engine.utils.Cell;

/**
 * Classe représentant un fou
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class Bishop extends Piece {
    public Bishop(Board board, Cell cell, PlayerColor color) {
        super(board, cell, color);
        moves.add(new LinearMove(this, new Cell(1, 1)));
        moves.add(new LinearMove(this, new Cell(1, -1)));
    }
```

```
1243
1244        @Override
1245        public PieceType getType() {
1246            return PieceType.BISHOP;
1247        }
1248
1249        @Override
1250        public String textValue() {
1251            return "Fou";
1252        }
1253    }
1254
1255    // ---------------------------------------------------------------------------------------------
1256
1257    package engine.pieces;
1258
1259    import chess.PieceType;
1260    import chess.PlayerColor;
1261    import engine.Board;
1262    import engine.moves.LinearMove;
1263    import engine.utils.Cell;
1264
1265    /**
1266     * Classe représentant un roi
1267     *
1268     * @author Jonathan Friedli
1269     * @author Valentin Kaelin
1270     */
1271    public class King extends FirstMoveSpecificPiece {
1272        private static final int CASTLE_DISTANCE = 2;
1273
1274        public King(Board board, Cell cell, PlayerColor color) {
1275            super(board, cell, color);
1276            moves.add(new LinearMove(this, new Cell(0, 1), 1));
1277            moves.add(new LinearMove(this, new Cell(1, 0), 1));
1278            moves.add(new LinearMove(this, new Cell(1, 1), 1));
1279            moves.add(new LinearMove(this, new Cell(1, -1), 1));
1280        }
1281
1282        @Override
1283        public PieceType getType() {
1284            return PieceType.KING;
1285        }
1286
1287        @Override
1288        public String textValue() {
1289            return "Roi";
1290        }
1291
1292        @Override
1293        public boolean checkMove(Cell to) {
1294            return super.checkMove(to) || castle(to);
1295        }
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
```

```java
    /**
     * Vérifie si le déplacement est un roque légal
     *
     * @param to : case de destination
     * @return true si le roque a bien été effectué, false sinon
     */
    private boolean castle(Cell to) {
        if (to == null)
            return false;
        int deltaY = to.getY() - getCell().getY();
        int deltaX = to.getX() - getCell().getX();
        if (hasMoved() || Math.abs(deltaX) != CASTLE_DISTANCE || deltaY != 0)
            return false;

        boolean leftSide = deltaX < 0;
        Cell direction = new Cell(leftSide ? -1 : 1, 0);
        Cell rookCell = new Cell(leftSide ? 0 : Board.BOARD_SIZE - 1, getCell().getY());
        Piece rook = getBoard().getPiece(rookCell);
        Cell rookDestination = getCell().add(direction);

        // Vérification de la tour et que le chemin est libre
        if (rook == null || rook.getType() != PieceType.ROOK ||
                ((Rook) rook).hasMoved() || !rook.checkMove(rookDestination))
            return false;

        // Vérification que le chemin ne met pas le roi en échec
        Cell initialPosition = getCell();
        for (int i = 0; i <= CASTLE_DISTANCE; i++) {
            Cell position = initialPosition.add(direction.multiply(i));
            getBoard().setPiece(this, position);

            boolean isAttacked = getBoard().isAttacked(getColor(), position);
            getBoard().removePiece(position);
            if (isAttacked) {
                getBoard().setPiece(this, initialPosition);
                return false;
            }
        }

        // Roque appliqué
        getBoard().applyMove(rook, rookDestination);
        rook.postUpdate();

        return true;
    }
}


// ------------------------------------------------------------------------------------------------
```

```java
package engine.pieces;

import chess.PieceType;
import chess.PlayerColor;
import engine.Board;
import engine.moves.LinearMove;
import engine.utils.Cell;

/**
 * Classe représentant un cavalier
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class Knight extends Piece {
    public Knight(Board board, Cell cell, PlayerColor color) {
        super(board, cell, color);
        moves.add(new LinearMove(this, new Cell(1, 2), 2, true));
        moves.add(new LinearMove(this, new Cell(1, -2), 2, true));
        moves.add(new LinearMove(this, new Cell(2, 1), 2, true));
        moves.add(new LinearMove(this, new Cell(2, -1), 2, true));
    }

    @Override
    public PieceType getType() {
        return PieceType.KNIGHT;
    }

    @Override
    public String textValue() {
        return "Cavalier";
    }
}

// ----------------------------------------------------------------------------------------------

package engine.pieces;

import chess.PieceType;
import chess.PlayerColor;
import engine.Board;
import engine.utils.Direction;
import engine.moves.OneDirectionMove;
import engine.utils.Cell;

/**
 * Classe représentant un pion
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class Pawn extends FirstMoveSpecificPiece {
    private final Direction direction;
    private int doubleMoveTurn;

    public Pawn(Board board, Cell cell, PlayerColor color) {
        super(board, cell, color);
        direction = color == PlayerColor.WHITE ? Direction.UP : Direction.DOWN;
        moves.add(new OneDirectionMove(this, 1, direction));
        moves.add(new OneDirectionMove(this, 2, direction, true));
    }

    @Override
    public PieceType getType() {
        return PieceType.PAWN;
    }
```

```java
    @Override
    public String textValue() {
        return "Pion";
    }

    @Override
    public boolean checkMove(Cell to) {
        if (super.checkMove(to)) {
            // On stocke le tour actuel si le pion s'est déplacé de deux cases
            if (Math.abs(to.getY() - getCell().getY()) == 2)
                doubleMoveTurn = getBoard().getTurn();
            return true;
        }

        if (to == null)
            return false;

        int deltaX = to.getX() - getCell().getX();
        int deltaY = to.getY() - getCell().getY();

        // Manger en diagonale
        if (Math.abs(deltaX) == 1 && deltaY == direction.intValue()) {
            if (getBoard().getPiece(to) != null)
                return true;
        }
        // En passant
        return enPassant(new Cell(to.getX(), getCell().getY()));
    }

    @Override
    public boolean applyMove(Cell to) {
        if (to == null)
            return false;

        Cell oldCell = getCell();
        Piece piece = getBoard().getLastPiecePlayed();

        // Vérification de la mise en échec du en-passant
        if (enPassant(new Cell(to.getX(), oldCell.getY()))) {
            getBoard().applyMove(this, to);
            getBoard().removePiece(piece.getCell());

            // En échec : on annule les moves
            if (getBoard().isCheck(getColor())) {
                getBoard().applyMove(this, oldCell);
                getBoard().setPiece(piece, piece.getCell());
                return false;
            }
            return true;
        }

        return super.applyMove(to);
    }

    @Override
    public void postUpdate() {
        super.postUpdate();

        // Gestion de la promotion
        if (canBePromoted() && getBoard().getOnPromotion() != null)
            getBoard().getOnPromotion().action(this);
    }
```

```java
    /**
     * @return true si le pion peut être promu, false sinon
     */
    public boolean canBePromoted() {
        return direction == Direction.UP ?
                getCell().getY() == Board.BOARD_SIZE - 1 :
                getCell().getY() == 0;
    }

    /**
     * Vérifie si le move en-passant peut être réalisé
     *
     * @param cell : case de destination
     * @return true si le move est légal, false sinon
     */
    public boolean enPassant(Cell cell) {
        Piece piece = getBoard().getLastPiecePlayed();
        int lastTurn = getBoard().getTurn() - 1;
        return piece != null && piece != this && piece.getColor() != getColor() &&
               piece.getType() == PieceType.PAWN &&
                ((Pawn) piece).doubleMoveTurn == lastTurn &&
                piece.getCell().equals(cell);
    }
}

// ------------------------------------------------------------------------------------------------

package engine.pieces;

import chess.PieceType;
import chess.PlayerColor;
import engine.Board;
import engine.moves.LinearMove;
import engine.utils.Cell;

/**
 * Classe représentant une reine
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class Queen extends Piece {
    public Queen(Board board, Cell cell, PlayerColor color) {
        super(board, cell, color);
        moves.add(new LinearMove(this, new Cell(0, 1)));
        moves.add(new LinearMove(this, new Cell(1, 0)));
        moves.add(new LinearMove(this, new Cell(1, 1)));
        moves.add(new LinearMove(this, new Cell(1, -1)));
    }

    @Override
    public PieceType getType() {
        return PieceType.QUEEN;
    }

    @Override
    public String textValue() {
        return "Reine";
    }
}

// ------------------------------------------------------------------------------------------------
```

```java
package engine.pieces;

import chess.PieceType;
import chess.PlayerColor;
import engine.Board;
import engine.moves.LinearMove;
import engine.utils.Cell;

/**
 * Classe représentant une tour
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class Rook extends FirstMoveSpecificPiece {
    public Rook(Board board, Cell cell, PlayerColor color) {
        super(board, cell, color);
        moves.add(new LinearMove(this, new Cell(0, 1)));
        moves.add(new LinearMove(this, new Cell(1, 0)));
    }

    @Override
    public PieceType getType() {
        return PieceType.ROOK;
    }

    @Override
    public String textValue() {
        return "Tour";
    }
}

// ------------------------------------------------------------------------------------------------

package engine.moves;

import engine.Board;
import engine.pieces.Piece;
import engine.utils.Cell;

/**
 * Classe abstraite modélisant la base des divers déplacements.
 * Le mouvement peut être limité à un nombre de cases.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public abstract class Move {
    private final Piece piece;
    private final int maxDistance;

    /**
     * Crée un déplacement
     *
     * @param piece       : pièce concernée
     * @param maxDistance : potentielle distance maximale
     * @throws RuntimeException si les arguments sont invalides
     */
    public Move(Piece piece, int maxDistance) {
        if (piece == null || maxDistance < 0)
            throw new RuntimeException("Création du Move invalide");

        this.piece = piece;
        this.maxDistance = maxDistance;
    }
```

```java
    /**
     * Vérifie qu'une case peut être atteinte grâce au déplacement
     *
     * @param from : case de départ
     * @param to   : case d'arrivée
     * @return true si la case est atteignable, false sinon
     */
    public abstract boolean canMove(Cell from, Cell to);

    /**
     * @return la pièce du déplacement
     */
    public Piece getPiece() {
        return piece;
    }

    /**
     * Helper permettant de récupérer plus facilement le plateau du déplacement
     *
     * @return le plateau de la pièce du déplacement
     */
    public Board getBoard() {
        return piece.getBoard();
    }

    /**
     * @return la distance maximale du déplacement
     */
    public int getMaxDistance() {
        return maxDistance;
    }
}

// ------------------------------------------------------------------------------------------

package engine.moves;

import engine.pieces.Piece;
import engine.utils.Cell;

/**
 * Classe représentant un déplacement linéaire dans un plan 2D.
 * La gestion des collisions est potentiellement gérée.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class LinearMove extends Move {
    protected final Cell direction;
    private final boolean flyOver;

    /**
     * Crée un déplacement linéaire
     *
     * @param piece       : pièce concernée
     * @param direction   : direction du déplacement
     * @param maxDistance : potentielle distance maximale
     * @param flyOver     : indique si la pièce prend en compte les collisions ou pas
     * @throws RuntimeException si les arguments sont invalides
     */
    public LinearMove(Piece piece, Cell direction, int maxDistance, boolean flyOver) {
        super(piece, maxDistance);

        if (direction == null)
            throw new RuntimeException("Création du LinearMove invalide");

        this.direction = direction;
        this.flyOver = flyOver;
    }
```

```java

    /**
     * Crée un déplacement linéaire
     *
     * @param piece       : pièce concernée
     * @param direction   : direction du déplacement
     * @param maxDistance : potentielle distance maximale
     */
    public LinearMove(Piece piece, Cell direction, int maxDistance) {
        this(piece, direction, maxDistance, false);
    }

    /**
     * Crée un déplacement linéaire
     *
     * @param piece     : pièce concernée
     * @param direction : direction du déplacement
     */
    public LinearMove(Piece piece, Cell direction) {
        this(piece, direction, Integer.MAX_VALUE, false);
    }


    @Override
    public boolean canMove(Cell from, Cell to) {
        if (from == null || to == null)
            return false;

        Cell fromTo = to.subtract(from);
        int distance = direction.reachable(fromTo) ? from.getDistance(to) : 0;
        int sign = direction.sameDirection(fromTo) ? 1 : -1;

        if (distance == 0 || distance > getMaxDistance())
            return false;

        // Gestion des collisions
        if (!flyOver) {
            for (int i = 1; i < distance; ++i) {
                Cell position = from.add(direction.multiply(i * sign));
                // Si une case sur le chemin est occupée
                if (getBoard().getPiece(position) != null)
                    return false;
            }
        }

        return true;
    }
}

// ------------------------------------------------------------------------------------------------
```

```java
package engine.moves;

import engine.pieces.FirstMoveSpecificPiece;
import engine.pieces.Piece;
import engine.utils.Cell;
import engine.utils.Direction;

/**
 * Classe représentant un déplacement réduit à une seule direction.
 * Le déplacement peut potentiellement être à usage unique.
 * La gestion des collisions est également gérée.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class OneDirectionMove extends Move {
    private final Direction boundToDirection;
    private final boolean oneTimeMove;

    /**
     * Crée un déplacement à une direction
     *
     * @param piece            : pièce concernée
     * @param maxDistance      : potentielle distance maximale
     * @param boundToDirection : unique direction possible
     * @param oneTimeMove      : true si le déplacement est à usage unique
     */
    public OneDirectionMove(Piece piece, int maxDistance, Direction boundToDirection,
                            boolean oneTimeMove) {
        super(piece, maxDistance);

        if (boundToDirection == null)
            throw new RuntimeException("Création du OneDirectionMove invalide");

        this.boundToDirection = boundToDirection;
        this.oneTimeMove = oneTimeMove;
    }

    /**
     * Crée un déplacement à une direction
     *
     * @param piece            : pièce concernée
     * @param maxDistance      : potentielle distance maximale
     * @param boundToDirection : unique direction possible
     */
    public OneDirectionMove(Piece piece, int maxDistance, Direction boundToDirection) {
        this(piece, maxDistance, boundToDirection, false);
    }
```

```java
    @Override
    public boolean canMove(Cell from, Cell to) {
        if (from == null || to == null)
            return false;

        // Vérification si le déplacement est à usage unique
        if (oneTimeMove && (!(getPiece() instanceof FirstMoveSpecificPiece) ||
                ((FirstMoveSpecificPiece) getPiece()).hasMoved()))
            return false;

        Cell calculatedTo = from.add(
                boundToDirection.getValue().multiply(getMaxDistance())
        );

        for (int i = 1; i < getMaxDistance(); ++i) {
            Cell position = from.add(boundToDirection.getValue().multiply(i));
            // Si une case sur le chemin est occupée
            if (getBoard().getPiece(position) != null)
                return false;
        }

        return getBoard().getPiece(to) == null && to.equals(calculatedTo);
    }
}
```