```java
package engine.game.displayChess;

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

import java.util.Objects;

/**
 * Controller between engine and GUI
 * Entrypoint of the program
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class Controller implements ChessController {
    private final DisplayChess chess;
    private ChessView view;

    /**
     * Controller constructor
     */
    public Controller() {
        chess = new DisplayChess(this);
    }

    /**
     * Get the view type
     * @return View type
     */
    public ChessView getView() {
        return view;
    }

    /**
     * Start a new game
     * @param view View in which to start
     */
    public void start(ChessView view) {
        this.view = Objects.requireNonNull(view, "Chessview must be non null");
        view.startView();
    }

    /**
     * Move a piece
     * @param fromX Start X value
     * @param fromY  Start Y value
     * @param toX Destination X value
     * @param toY Destination Y value
     * @return Either the piece can move or not
     */
    public boolean move(int fromX, int fromY, int toX, int toY) {
        // Déplacement à la même position impossible
        if(fromX != toX || fromY != toY){
            return chess.move(fromX, fromY, toX, toY);
        }
        return false;
    }

    /**
     * Start a new game
     */
    @Override
    public void newGame() {
        chess.startGame();
    }

    /**
     * Main programm
     * @param args Programm arguments
     */
    public static void main(String[] args) {
        if(args.length == 1){
            ChessController c = new Controller();
            switch (args[0]) {
                case "0":
                    c.start(new ConsoleView(c));
```

```
                    break;
                case "1":
                    c.start(new GUIView(c));
                    c.newGame();
                    break;
                default:
                    System.out.println("Invalid Gamemode : 0 -> Console | 1 -> Graphics");
                    System.exit(-1);
            }
        } else{
            System.out.println("Invalid Number of args");
            System.exit(-1);
        }
    }
}
```

```java
package engine.game.displayChess;

import chess.PieceType;
import chess.PlayerColor;
import engine.game.board.Move;
import engine.game.board.Piece;
import engine.game.board.Vector;
import engine.game.chess.Chess;
import engine.game.chess.ChessColor;

import java.util.Objects;

/**
 * Synchronize GUI with the engine
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class DisplayChess extends Chess {
    private final Controller controller;

    boolean areGUIPromptsDisable() {
        return guiPromptsDisabled;
    }

    private boolean guiPromptsDisabled;

    /**
     * DisplayChess constructor
     * @param controller Concerned controller
     */
    public DisplayChess(Controller controller) {
        super();
        this.controller = Objects.requireNonNull(controller, "controller must be non null");
        guiPromptsDisabled = false;
    }

    /**
     * Init rules
     */
    @Override
    protected void initRules() {
        super.initRules();
    }

    /**
     * Move a piece
     * @param fromX Start X value
     * @param fromY  Start Y value
     * @param toX Destination X value
     * @param toY Destination Y value
     * @return Either the piece can move or not
     */
    public boolean move(int fromX, int fromY, int toX, int toY) {
        return move(new Vector(fromX, fromY), new Vector(toX, toY));
    }

    /**
     * Check if the King is in check
     * @param defendingColor Concerned color
     * @return Either the King is in check or not
     */
    @Override
    public boolean check(ChessColor defendingColor) {
        if(super.check(defendingColor)){
            displayCheck();
            return true;
        }
        return false;
    }

    /**
     * Display winner
     * @param winner Winner color
     */
    @Override
    protected void endGame(ChessColor winner) {
        super.endGame(winner);
```

```
        displayWinner(winner);
    }

    /**
     * Set piece to a given position
     * @param piece Piece to set at given position
     * @param position Position to set the piece
     * @return The moved piece
     */
    @Override
    public ChessPiece setPieceAtPosition(Piece<Chess> piece, Vector position) {
        super.setPieceAtPosition(piece, position);
        if(piece != null)
            controller.getView().putPiece(getPieceType(((ChessPiece)piece).getPieceType()),
            getPlayerColor(((ChessPiece)piece).getColor()), position.getI(), position.getJ());
        return (ChessPiece)piece;
    }

    /**
     * Remove piece at a given position
     * @param position Position to remove the piece
     * @return The removed piece
     */
    @Override
    public ChessPiece removePieceAtPosition(Vector position) {
        Objects.requireNonNull(position, "position must be non null");
        ChessPiece piece = super.removePieceAtPosition(position);
        controller.getView().removePiece(position.getI(), position.getJ());
        return piece;
    }

    @Override
    public boolean doesMoveCheck(ChessPiece piece, Vector start, Vector destination, Move<Chess>
    moveType) {
        guiPromptsDisabled = true;
        boolean status = super.doesMoveCheck(piece, start, destination, moveType);
        guiPromptsDisabled = false;
        return status;
    }

    /**
     * Get promoted piece
     * @return Engine promoted piece
     */
    public Chess.ChessPiece getPromotedPiece() {
        if(!areGUIPromptsDisable())
            return askUserPromotion();
        return super.getPromotedPiece();
    }

    /**
     * Ask user to which Piece does he want to promote his pawn
     * @return Promoted piece
     */
    private ChessPiece askUserPromotion() {
        Piece[] possibilities = {
                new Queen(getTurn(), this),
                new Knight(getTurn(), this),
                new Rook(getTurn(), this),
                new Bishop(getTurn(), this)
        };

        return (ChessPiece) controller.getView().askUser("You can promote your piece !", "In what will
        your piece promote to ?", possibilities);
    }

    /**
     * Display Check message
     */
    public void displayCheck(){
        controller.getView().displayMessage("Check");
    }

    /**
     * Display winner when checkmate
     */
    protected void displayWinner(ChessColor winner) {
```

```
        controller.getView().displayMessage("Checkmate ! " + Objects.requireNonNull(winner, "winner
        must be non null") + " won the game !");
    }

    /**
     * Get the player color
     * @param color Chess color
     * @return Player color
     */
    private PlayerColor getPlayerColor(ChessColor color){
        switch (Objects.requireNonNull(color, "color must be non null")){
            case WHITE:
                return PlayerColor.WHITE;
            case BLACK:
                return PlayerColor.BLACK;
        }
        throw new IllegalArgumentException(color + " is not handled by GUI");
    }

    /**
     * Get enum Piece type
     * @param type Engine piece type
     * @return Enum Piece type
     */
    private PieceType getPieceType(ChessPieceType type){
        switch (Objects.requireNonNull(type, "type must be non null")) {
            case PAWN:
                return PieceType.PAWN;
            case ROOK:
                return PieceType.ROOK;
            case KNIGHT:
                return PieceType.KNIGHT;
            case BISHOP:
                return PieceType.BISHOP;
            case QUEEN:
                return PieceType.QUEEN;
            case KING:
                return PieceType.KING;
        }
        throw new IllegalArgumentException(type + " is not handled by GUI");
    }
}
```

```java
package engine.game.board;

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

/**
 * Board modelise a board of Squares in which different action can be made
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 * @param <T> Type of Board
 */
public abstract class Board<T extends Board<T>> {
    private final int LENGTH;
    private final int HEIGHT;
    private final ArrayList<Square<T>> boardArray;

    protected final Historic<T> historicMoves;

    /**
     * Board Constructor
     * @param length Boards length
     * @param height Boards height
     */
    protected Board(int length, int height){
        if(length <= 0|| height <= 0)
            throw new IllegalArgumentException("Size must be above 0");
        LENGTH = length;
        HEIGHT = height;
        boardArray = new ArrayList<>();
        historicMoves = new Historic<>();
        // populate board
        for (int j = 0; j < HEIGHT; j++) {
            for (int i = 0; i < LENGTH; i++) {
                boardArray.add(new Square<>(i, j));
            }
        }
    }

    /**
     * Start a new game
     */
    protected void startGame(){
        initPieces();
    }

    /**
     * Init pieces on the board
     */
    protected void initPieces(){
        emptyBoard();
    }

    /**
     * Move request
     * @param from Vector from which the request move is made
     * @param to Vector to which the request move is made
     * @return Either the move can be made or not
     */
    public boolean move(Vector from, Vector to) {
        Objects.requireNonNull(from, "from vector must be non null");
        Objects.requireNonNull(from, "to vector must be non null");

        Square<T> fromSquare = getSquareAtPosition(from);

        // Si y a bien une pièce
        Piece<T> selectedPiece = fromSquare.getPiece();
        if(selectedPiece != null){
            return selectedPiece.move(from, to, true);
        }
        return false;
    }

    /**
     * Get the length of the board
     * @return The length of the board
     */
    public int getLENGTH() {
```

```
        return LENGTH;
    }

    /**
     * Get the height of the board
     * @return The height of the board
     */
    public int getHEIGHT() {
        return HEIGHT;
    }

    /**
     * Set piece at a given position
     * @param piece Piece to set at given position
     * @param position Position to set the piece on
     * @return The piece on the given position
     */
    public Piece<T> setPieceAtPosition(Piece<T> piece, Vector position){
        Objects.requireNonNull(piece, "piece must be non null");
        Objects.requireNonNull(position, "position must be non null");
        if (position.getI() < 0 || position.getJ() < 0 || position.getI() >= HEIGHT || position.getJ()
        >= LENGTH) {
            throw new IllegalArgumentException("Position is out of bounds");
        }
        else {
            Square<T> c = getSquareAtPosition(position);
            c.setPiece(piece);
            return piece;
        }
    }

    /**
     * Set piece at a given position by indexes
     * @param piece Piece to set at given indexes
     * @param i Index i to set the piece on
     * @param j Index j to set the piece on
     * @return The piece on the given position
     */
    public Piece<T> setPieceAtPosition(Piece<T> piece, int i, int j) {
        return setPieceAtPosition(piece, new Vector(i, j));
    }

    /**
     * Get the piece at a given position
     * @param position The position to get the piece
     * @return The piece at the given position
     */
    public Piece<T> getPieceAtPosition(Vector position){
        return getSquareAtPosition(Objects.requireNonNull(position, "position must be non
        null")).getPiece();
    }

    /**
     * Remove piece at a given position
     * @param position Position to remove the piece
     * @return The removed piece
     */
    public Piece<T> removePieceAtPosition(Vector position){
        return getSquareAtPosition(Objects.requireNonNull(position, "position must be non
        null")).removePiece();
    }

    /**
     * Move the position at a given position
     * @param from Vector from where to move the piece
     * @param to Vector to where to move the piece
     * @return The piece at the new position
     */
    public Piece<T> movePieceAtPosition(Vector from, Vector to){
        return setPieceAtPosition(removePieceAtPosition(Objects.requireNonNull(from, "from vector must
        be non null")), Objects.requireNonNull(to, "to vector must be non null"));
    }

    /**
     * Get square at a given position
     * @param position Position to get the square
     * @return The square at the given position
```

```
     */
    private Square<T> getSquareAtPosition(Vector position){
        Objects.requireNonNull(position);
        if(position.getI() >= 0 && position.getJ() >= 0 && position.getI() < HEIGHT && position.getJ()
        < LENGTH) {
            return boardArray.get(position.getI() + position.getJ() * LENGTH);
        }
        throw new IllegalArgumentException("Position is out of bounds");
    }

    /**
     * Remove all pieces from the board
     */
    public void emptyBoard(){
        for (Square<T> c : boardArray) {
            c.removePiece();
        }
    }

    /**
     * Search for a piece in the board
     * @param pieceToSearch Piece to be searched
     * @return The list off all pieces found
     */
    public ArrayList<Vector> searchPieces(Piece<T> pieceToSearch){
        ArrayList<Vector> foundPieces = new ArrayList<>();
        if(pieceToSearch == null){
            return foundPieces;
        }
        for (Square<T> c : boardArray) {
            Piece<T> pieceOnPosition = c.getPiece();
            if(pieceToSearch.equals(pieceOnPosition)){
                foundPieces.add(c.getPosition());
            }
        }
        return foundPieces;
    }

    /**
     * Intern square class
     * @param <T> Type of Square
     */
    private static class Square<T extends Board<T>> {
        private Piece<T> piece;
        private final int X;
        private final int Y;

        /**
         * Square constructor
         * @param x Position x of the square
         * @param y Position y of the square
         */
        private Square(int x, int y) {
            if(x < 0 || y < 0)
                throw new IllegalArgumentException("Square position can't be under 0");
            X = x;
            Y = y;
        }

        /**
         * Get the piece on the square
         * @return
         */
        private Piece<T> getPiece() {
            return piece;
        }

        /**
         * Set a piece on the square
         * @param piece
         */
        private void setPiece(Piece<T> piece) {
            Objects.requireNonNull(piece);
            this.piece = piece;
        }

        /**
```

```
         * Get the position of a square
         * @return
         */
        private Vector getPosition(){
            return new Vector(X, Y);
        }

        /**
         * Remove piece from the board
         * @return The removed piece
         */
        private Piece<T> removePiece(){
            Piece<T> returnPiece = piece;
            piece = null;
            return returnPiece;
        }
    }

    /**
     * Check if the piece has moved on the board
     * @param piece Piece to be checked
     * @return Either the piece has moved or not
     */
    public boolean hasMoved(Piece<T> piece){
        return historicMoves.isPieceContained(piece);
    }

    /**
     * Check if the move a move is the last one executed
     * @param move move we are looking for
     * @return true if the move given is the last one executed
     */
    public boolean isLastAction(Move<T> move){
        return historicMoves.isLastAction(move);
    }

    /**
     * Returns the last piece moved in the historic
     * @return
     */
    public Piece<T> lastPieceMoved(){
        return historicMoves.lastPieceMoved();
    }

    /**
     * Return itself from the most specific viewpoint
     * Exemple : A board of chess would return itself as a Chess class
     * @return
     */
    public abstract T self();
}
```

```java
package engine.game.board;

/**
 * Interface for game actions
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 * @param <T> Type of Board
 */
public interface GameAction<T extends Board<T>> {
    /**
     * Perform an action
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param t Concerned Board on which the action will be performed
     * @return The piece on the destination
     */
    Piece<T> doAction(Vector start, Vector destination, Board<T> t);

    /**
     * Revert an action
     * @param start Vector from where the action started
     * @param destination Vector to where the action ended
     * @param affectedPiece Affected pieces by the action
     * @param t Concerned Board on which the action was perform
     */
    void revertAction(Vector start, Vector destination, Piece<T> affectedPiece, Board<T> t);
}
```

```java
package engine.game.board;

/**
 * Interface for game conditions
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 * @param <T> Type of Board
 */
public interface GameCondition<T extends Board<T>> {
    /**
     * Check a condition
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param board Concerned Board on which the action will be performed
     * @return Either the condition is accepted or not
     */
    boolean checkCondition(Vector start, Vector destination, Board<T> board);
}
```

```java
package engine.game.board;

import java.util.List;
import java.util.Objects;
import java.util.Stack;

/**
 * Historic of all moves
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 * @param <T> Type of Board
 */
public class Historic<T extends Board<T>> {
    private final Stack<Action<T>> historicMoves;

    /**
     * Historic constructor
     */
    Historic(){
        this.historicMoves = new Stack<>();
    }

    /**
     * Add an action to the stack of historic moves
     * @param piece Concerned piece
     * @param move Move made by the piece
     * @param depart Begin vector
     * @param arrivee Destination vector
     * @param affectedPieces List of affected pieces
     */
    void add(Piece<T> piece, Move<T> move, Vector depart, Vector arrivee, List<Piece<T>> affectedPieces){
        historicMoves.push(new Action<>(piece, move, depart, arrivee, affectedPieces));
    }

    /**
     * Check if the piece is contained in stack of historic moves
     * @param piece The piece to be checked
     * @return Either the piece is contained in the historic moves or not
     */
    public boolean isPieceContained(Piece<T> piece){
        if(piece == null)
            return false;
        for(Action<T> action : historicMoves){
            if(piece == action.piece)
                return true;
        }
        return false;
    }

    /**
     * Cancel last move
     */
    public void revertLastMove(){
        if(historicMoves.empty())
            throw new RuntimeException("No move have been done. Can't revert");
        historicMoves.pop().revert();
    }

    /**
     * Check if the move a move is the last one executed
     * @param move move we are looking for
     * @return true if the move given is the last one executed
     */
    boolean isLastAction(Move<T> move){
        return getLastAction().isMove(move);
    }

    /**
     * Returns the last piece moved in the historic
     * @return
     */
    Piece<T> lastPieceMoved(){
        return getLastAction().piece;
    }


    /**
```

```
     * Get the last action recorded
     * @return Last added action in historic list
     */
    private Action<T> getLastAction(){
        if(historicMoves.empty())
            throw new RuntimeException("No move have been done. Can't get last action");
        return historicMoves.peek();
    }

    /**
     * Action tracks the move of a Piece
     * @author Alen Bijelic
     * @author Nelson Jeanrenaud
     * @param <T> Type of Board
     */
    private static class Action<T extends Board<T>> {
        private final Piece<T> piece;
        private final Move<T> move;
        private final Vector depart;
        private final Vector arrivee;

        // List of affected pieces i.e. pieces eaten by the piece
        private final List<Piece<T>> affectedPieces;

        /**
         * Action constructor
         * @param piece Piece that made the action
         * @param move The move made by the piece
         * @param depart Vector from where the piece is moved
         * @param arrivee Vector to where the piece is moved
         * @param affectedPieces Affected pieces by the action
         */
        private Action(Piece<T> piece, Move<T> move, Vector depart, Vector arrivee, List<Piece<T>>
        affectedPieces){
            Objects.requireNonNull(piece);
            Objects.requireNonNull(move);
            Objects.requireNonNull(depart);
            Objects.requireNonNull(arrivee);
            this.piece = piece;
            this.move = move;
            this.depart = depart;
            this.arrivee = arrivee;
            this.affectedPieces = affectedPieces;
        }

        /**
         * is the move the one done at this action
         * @param move move we are comparing it to
         * @return true if it's the case
         */
        private boolean isMove(Move<T> move){
            return move.equals(this.move);
        }

        /**
         * Brings back to the original state
         */
        private void revert(){
            move.revertMove(depart, arrivee, affectedPieces, piece.getBoard());
        }

    }

}
```

```java
package engine.game.board;

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

/**
 * Manage all moves that can be made in a board
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 * @param <T> Type of Board
 */
public class Move<T extends Board<T>> {
    private final Vector vector;
    private final boolean isMirroredX;
    private final boolean isMirroredY;
    private final List<GameAction<T>>actions;
    private final List<GameCondition<T>> conditions;

    /**
     * Move constructor
     * @param vector Move vector made by the piece
     * @param isMirroredX Either the move is vertically mirrored or not
     * @param isMirroredY Either the move is horizontally mirrored or not
     * @param conditions List all conditions that applies to the move
     * @param actions List all actions that applied to the move
     */
    public Move(Vector vector, boolean isMirroredX, boolean isMirroredY, List<GameCondition<T>>
    conditions, List<GameAction<T>> actions) {
        this.vector = Objects.requireNonNull(vector, "movement vector must be non null");
        this.isMirroredX = isMirroredX;
        this.isMirroredY = isMirroredY;
        this.actions = actions;
        this.conditions = conditions;
    }

    /**
     * Move constructor whitout conditions and actions
     * @param vector  Move vector made by the piece
     * @param isMirroredX Either the move is vertically mirrored or not
     * @param isMirroredY Either the move is horizontally mirrored or not
     */
    public Move(Vector vector, boolean isMirroredX, boolean isMirroredY) {
        this(vector, isMirroredX, isMirroredY, null, null);
    }

    /**
     * Check conditions for moves
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param t Concerned Board on which the action will be performed
     * @return Either the move is legit or not
     */
    protected boolean checkConditions(Vector start, Vector destination, Board<T> t) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(t, "board must be non null");

        if (conditions != null) {
            for (GameCondition<T> condition : conditions) {
                if (!condition.checkCondition(start, destination, t)) {
                    return false;
                }
            }
        }
        return true;
    }

    /**
     * Check if a move can be made from start to destination
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param t Concerned Board on which the action will be performed
     * @return Either the move is legit or not
     */
    boolean canMove(Vector start, Vector destination, Board<T> t){
        Objects.requireNonNull(start, "start vector must be non null");
```

```
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(t, "board must be non null");

        Vector movementVector = new Vector(destination.getI() - start.getI(), destination.getJ() -
        start.getJ());

        if(vector.norm() < movementVector.norm()){
            return false;
        }

        // Check si dans la bonne direction
        return ((movementVector.areCollinearAndSameDirection(vector) ||
                (isMirroredX && movementVector.areCollinearAndSameDirection(vector.getMirrorXVector()))
                ||
                (isMirroredY && movementVector.areCollinearAndSameDirection(vector.getMirrorYVector()))
                ||
                (isMirroredX && isMirroredY &&
                movementVector.areCollinearAndSameDirection(vector.getOpposedVector())))
                && checkConditions(start, destination, t));
    }

    /**
     * Perform a move
     * @param start Vector from where the move starts
     * @param destination Vector to where the move ends
     * @param t Concerned Board on which the move will be performed
     * @return List of affected pieces
     */
    ArrayList<Piece<T>> doMove(Vector start, Vector destination, T t){
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(t, "board must be non null");

        ArrayList<Piece<T>> affectedPieces = new ArrayList<>();
        if(actions != null) {
            for (GameAction<T> actionToDo : actions) {
                    affectedPieces.add(actionToDo.doAction(start, destination, t));
            }
        }
        return affectedPieces;
    }

    /**
     * Revert a move
     * @param start Vector from where the move started
     * @param destination Vector to where the move ended
     * @param affectedPieces List of affected pieces
     * @param t Concerned Board on which the move will be performed
     */
    void revertMove(Vector start, Vector destination, List<Piece<T>> affectedPieces, Board<T> t){
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(t, "board must be non null");

        if(actions != null) {
            for (int i = actions.size() - 1; i >= 0; i--) {
                actions.get(i).revertAction(start, destination, affectedPieces.get(i), t);
            }
        }
    }
}
```

```java
package engine.game.board;

import chess.ChessView;

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

/**
 * Manage pieces move
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 * @param <T> Type of Board
 */
public abstract class Piece<T extends Board<T>> implements ChessView.UserChoice {
    private final T board;
    private final List<Move<T>> movements;

    /**
     * Piece constructor
     * @param board The board type
     * @param moves List of legit moves
     */
    public Piece(T board, List<Move<T>> moves) {
        this.board = Objects.requireNonNull(board, "board must be non null");
        this.movements = moves;
    }

    /**
     * Get the board type
     * @return the board type
     */
    public T getBoard() {
        return board;
    }

    /**
     * Move a piece
     * @param fromX Start X value
     * @param fromY  Start Y value
     * @param toX Destination X value
     * @param toY Destination Y value
     * @param doMove Perform or simulate the move
     * @return Either the piece can move or not
     */
    public boolean move(int fromX, int fromY, int toX, int toY, boolean doMove){
        for (Move<T> moveType: movements) {
            Vector start = new Vector(fromX, fromY), destination = new Vector(toX, toY);
            if(canMove(start, destination, moveType)){
                if(doMove) {
                    doMove(start, destination, moveType);
                }
                return true;
            }
        }
        return false;
    }

    /**
     * Check if a move can be made from start to destination
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param moveType Move type
     * @return Either the move is legit or not
     */
    protected boolean canMove(Vector start, Vector destination, Move<T> moveType){
        Objects.requireNonNull(moveType, "the move type must be non null");
        return moveType.canMove(Objects.requireNonNull(start, "start vector must be non null"),
                Objects.requireNonNull(destination, "destination vector must be non null"), getBoard());
    }

    /**
     * Perform a move
     * @param start Vector from where the move starts
     * @param destination Vector to where the move ends
     * @param moveType Move type
     */
```

```
    protected void doMove(Vector start, Vector destination, Move<T> moveType){
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(moveType, "moveType must be non null");
        getBoard().historicMoves.add(this, moveType, start, destination, moveType.doMove(start,
        destination, getBoard()));
    }

    /**
     * Move a piece
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param doMove Perform or simulate the move
     * @return Either the piece can move or not
     */
    public boolean move(Vector start, Vector destination, boolean doMove){
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        return move(start.getI(), start.getJ(), destination.getI(), destination.getJ(), doMove);
    }

    /**
     * Lists all possible moves of a piece
     * @param start Start position of the piece
     * @return List of all possible moves
     */
    public List<Vector> possibleMoves(Vector start){
        Objects.requireNonNull(start, "start vector must be non null");
        List<Vector> possibleMoves = new ArrayList<>();
        for (Move<T> moveType: movements) {
            for (int i = 0; i < getBoard().getLENGTH(); i++) {
                for (int j = 0; j < getBoard().getHEIGHT() ; j++) {
                    Vector destination = new Vector(i, j);
                    if(canMove(start, destination, moveType)){
                        possibleMoves.add(destination);
                    }
                }
            }
        }
        return possibleMoves;
    }


    /**
     * Get Piece to a string formatted value
     * @return String formatted value of the class
     */
    @Override
    public String toString() {
        return textValue();
    }

    /**
     * Get the class name
     * @return Class name
     */
    public String textValue(){
        return getClass().getSimpleName();
    }
}
```

```java
package engine.game.board;

import java.lang.Math;
import java.util.ArrayList;
import java.util.Objects;

/**
 * Vector modelisation
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class Vector {
    private int i;
    private int j;

    /**
     * Vector constructor
     * @param i Index i of the vector
     * @param j Index j of the vector
     */
    public Vector(int i, int j) {
        this.i = i;
        this.j = j;
    }

    /**
     * Get the i index
     * @return i index
     */
    public int getI() {
        return i;
    }

    /**
     * Get the j index
     * @return j index
     */
    public int getJ() {
        return j;
    }

    /**
     * Set the i index
     * @param i Index to be set
     */
    public void setI(int i) {
        this.i = i;
    }

    /**
     * Set the j index
     * @param j Index to be set
     */
    public void setJ(int j) {
        this.j = j;
    }

    /**
     * Perform addition with another Vector
     * @param other Vector to be added to this
     * @return The new obtained vector
     */
    public Vector add(Vector other){
        Objects.requireNonNull(other, "other vector must be non null");
        return new Vector(i + other.i, j + other.j);
    }

    /**
     * Perform substraction with another Vector
     * @param other Vector to be substracted to this
     * @return The new obtained vector
     */
    public Vector sub(Vector other){
        Objects.requireNonNull(other, "other vector must be non null");
        return new Vector(i - other.i, j - other.j);
    }
```

```java
    /**
     * Perform multiplication with a factor
     * @param factor The factor to apply to the vector
     * @return The new obtained vector
     */
    public Vector multiply(int factor){
        return new Vector(factor * i, factor * j);
    }

    /**
     * Get the vector norm
     * @return Vector norm
     */
    public double norm(){
        return Math.sqrt(Math.pow(i, 2) + Math.pow(j, 2));
    }

    /**
     * Get the smallest collinear Vector
     * @return Smallest collinear Vector
     */
    public Vector getSmallestCollinearVector(){
        int gcd = gcd(i, j);
        if(gcd != 0)
            return new Vector(i / gcd, j / gcd);
        return new Vector(i, j);
    }

    /**
     * Lists all included vectors
     * @return All included vectors
     */
    public ArrayList<Vector> includedVectors(){
        ArrayList<Vector> includedVectors = new ArrayList<>();
        Vector base = getSmallestCollinearVector();
        int nb = Math.abs(gcd(i, j));
        for (int factor = 1; factor < nb; factor++) {
            includedVectors.add(base.multiply(factor));
        }
        return includedVectors;
    }

    /**
     * Perform a cross product
     * @param other Vector to perform cross product
     * @return Cross product between both vectors
     */
    public int crossProduct(Vector other){
        Objects.requireNonNull(other, "other vector must be non null");
        return i * other.j - j * other.i;
    }

    /**
     * Check if vectors are collinear
     * @param other Vector to check with
     * @return Either the vetors are collinear or not
     */
    public boolean areCollinear(Vector other){
        return crossProduct(Objects.requireNonNull(other, "other vector must be non null")) == 0;
    }

    /**
     * Check if vectors are collinear and also in the same direction
     * @param other Vector to check with
     * @return Either the vetors are collinear and in the same direction or not
     */
    public boolean areCollinearAndSameDirection(Vector other) {
        return getSmallestCollinearVector().equals(other.getSmallestCollinearVector());
    }

    /**
     * Get vertically mirrored vector
     * @return Vertically mirrored vector
     */
    public Vector getMirrorYVector(){
        return new Vector(-i, j);
    }
```

```
    /**
     * Get horizontally mirrored vector
     * @return Horizontally mirrored vector
     */
    public Vector getMirrorXVector(){
        return new Vector(i, -j);
    }

    /**
     * Get opposed vector
     * @return Opposed vector
     */
    public Vector getOpposedVector(){
        return new Vector(-i, -j);
    }

    /**
     * Get the greatest common divisor between two values
     * @param a Value A
     * @param b Value B
     * @return The greatest common divisor
     */
    private static int gcd(int a, int b) {
        a = Math.abs(a);
        b = Math.abs(b);
        if (a == 0) return b;
        if (b == 0) return a;
        if (a > b) return gcd(b, a);
        return gcd(b%a, a);
    }

    /**
     * Transform Vector to a String representation
     * @return String of a Vector
     */
    @Override
    public String toString() {
        return "Vector{" +
                "i=" + i +
                ", j=" + j +
                '}';
    }

    /**
     * Check if the vector is equal to an object
     * @param o Object to check equality
     * @return Either the vector and the object are equal or not
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Vector vector = (Vector) o;
        return i == vector.i && j == vector.j;
    }

    /**
     * Get the hash code of the vector
     * @return Hash code of the vector
     */
    @Override
    public int hashCode() {
        return Objects.hash(i, j);
    }
}
```

```java
package engine.game.chess;

import engine.game.board.Board;
import engine.game.board.GameCondition;
import engine.game.board.Vector;

import java.util.Objects;

/**
 * Condition to not eat
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class CanNotEat implements GameCondition<Chess> {

    /**
     * Check if there is no pieces at destination
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param board Concerned Board on which the action will be performed
     * @return If there is no pieces at destination
     */
    @Override
    public boolean checkCondition(Vector start, Vector destination, Board<Chess> board) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(board, "chess board must be non null");
        return board.getPieceAtPosition(destination) == null;
    }
}
```

```java
package engine.game.chess;

import engine.game.board.Board;
import engine.game.board.Move;
import engine.game.board.Piece;
import engine.game.board.Vector;

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

/**
 * Chessboard
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class Chess extends Board<Chess> {

    /**
     * All types of chess pieces
     */
    public enum ChessPieceType {
        PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING
    }

    /**
     * All directions available
     */
    public enum Direction {
        DOWN{
            /**
             * Starting at edge
             * @return Edge index
             */
            @Override
            protected int startingEdge() {
                return SIZE - 1;
            }

            /**
             * Starting at edge
             * @param value Value from edge
             * @return Edge without value index
             */
            @Override
            protected int startingEdge(int value) {
                return startingEdge() - value;
            }

            /**
             * Get oppposite Direction
             * @return Opposite Direction
             */
            @Override
            protected Direction opposite() {
                return UP;
            }

            /**
             * Get adjacent Directions
             * @return Array of adjacent Direction
             */
            @Override
            protected Direction[] adjacent() {
                return new Direction[] {LEFT, RIGHT};
            }
        }, LEFT{
            /**
             * Starting at edge
             * @return Edge index
             */
            @Override
            protected int startingEdge() {
                return 0;
            }

            /**
```

```
             * Starting at edge
             * @param value Value from edge
             * @return Edge with value index
             */
            @Override
            protected int startingEdge(int value) {
                return startingEdge() + value;
            }

            /**
             * Get the opposite Direction
             * @return Opposite direction
             */
            @Override
            protected Direction opposite() {
                return RIGHT;
            }

            /**
             * Get adjacent directions
             * @return Array of adjacent directions
             */
            @Override
            protected Direction[] adjacent() {
                return new Direction[]{UP, DOWN};
            }
        }, UP{
            /**
             * Starting at edge
             * @return Edge index
             */
            @Override
            protected int startingEdge() {
                return 0;
            }

            /**
             * Starting at edge
             * @param value Value from edge
             * @return Edge with value index
             */
            @Override
            protected int startingEdge(int value) {
                return startingEdge() + value;
            }

            /**
             * Get the opposite Direction
             * @return Opposite direction
             */
            @Override
            protected Direction opposite() {
                return DOWN;
            }

            @Override
            protected Direction[] adjacent() {
                return new Direction[] {LEFT, RIGHT};
            }
        }, RIGHT{
            /**
             * Starting at edge
             * @return Edge index
             */
            @Override
            protected int startingEdge() {
                return SIZE - 1;
            }

            /**
             * Starting at edge
             * @param value Value from edge
             * @return Edge without value index
             */
            @Override
            protected int startingEdge(int value) {
                return startingEdge() - value;
```

```
        }

        /**
         * Get the opposite Direction
         * @return Opposite direction
         */
        @Override
        protected Direction opposite() {
            return LEFT;
        }

        /**
         * Get adjacent directions
         * @return Array of adjacent directions
         */
        @Override
        protected Direction[] adjacent() {
            return new Direction[] {UP, DOWN};
        }
    };

    /**
     * Get index of the starting edge
     * @return Index of the starting edge
     */
    protected abstract int startingEdge();

    /**
     * Get index of the starting edge including value
     * @param value Value from edge
     * @return Edge with value index
     */
    protected abstract int startingEdge(int value);

    /**
     * Get the opposite Direction
     * @return Opposite direction
     */
    protected abstract Direction opposite();

    /**
     * Get adjacent directions
     * @return Array of adjacent directions
     */
    protected abstract Direction[] adjacent();
}

private static final int SIZE = 8;
private final ChessColor FIRST_COLOR = ChessColor.WHITE;

private boolean isStarted;
private ChessColor turn;

Move<Chess> getPawnStraight2Up() {
    return pawnStraight2Up;
}
Move<Chess> getPawnStraight2Down() {
    return pawnStraight2Down;
}

// Rules...
private Promote promote;
private Roque roque;
private EnPassant pawnEnPassant;
private EatPiece eatAction;
private MoveChessPiece moveChessPiece;
private MustEat mustEat;
private CanNotEat cannotEat;
private OnlyFirstMove onlyFirstMove;
private MustNotCollide noCollision;

/**
 * Get turn color
 * @return Turn color
 */
public ChessColor getTurn() {
    return turn;
```

```
}


// Moves...
private Move<Chess> kingHorizontalStraights;
private Move<Chess> kingVerticalStraights;
private Move<Chess> kingDiagonals;
private Move<Chess> kingGrandRoque;
private Move<Chess> kingPetitRoque;

private Move<Chess> pawnStraight1Up;
private Move<Chess> pawnStraight2Up;
private Move<Chess> pawnEat1Up;
private Move<Chess> pawnEnPassantUp;
private Move<Chess> pawnStraight1Down;
private Move<Chess> pawnStraight2Down;
private Move<Chess> pawnEat1Down;
private Move<Chess> pawnEnPassantDown;

private Move<Chess> knightL;
private Move<Chess> knightL2;

private Move<Chess> horizontalStraights;
private Move<Chess> verticalStraights;
private Move<Chess> diagonals;

/**
 * Chess constructor
 */
public Chess(){
    super(SIZE, SIZE);
    isStarted = false;
    initRules();
    initMoves();
}

/**
 * Start a game
 */
@Override
public void startGame() {
    turn = FIRST_COLOR;
    isStarted = true;
    super.startGame();
}

/**
 * Init legit moves in chess
 */
protected void initMoves() {
    horizontalStraights = new Move<>(new Vector(super.getLENGTH(), 0), false, true,
    List.of(noCollision), List.of(eatAction));
    verticalStraights = new Move<>(new Vector(0, super.getHEIGHT()), true, false,
    List.of(noCollision), List.of(eatAction));
    diagonals = new Move<>(new Vector(super.getLENGTH(), super.getHEIGHT()), true, true,
    List.of(noCollision), List.of(eatAction));

    kingGrandRoque = new Move<>(new Vector(-4, 0), false, false, List.of(onlyFirstMove,
    noCollision, roque), List.of(roque));
    kingPetitRoque = new Move<>(new Vector(3, 0), false, false, List.of(onlyFirstMove,
    noCollision, roque), List.of(roque));
    kingHorizontalStraights = new Move<>(new Vector(1, 0), true, true, List.of(noCollision),
    List.of(eatAction));
    kingVerticalStraights = new Move<>(new Vector(0, 1), true, false, List.of(noCollision),
    List.of(eatAction));
    kingDiagonals = new Move<>(new Vector(1, 1), true, true, List.of(noCollision),
    List.of(eatAction));

    pawnStraight1Up = new Move<>(new Vector(0, 1), false, false, List.of(noCollision, cannotEat),
    List.of(moveChessPiece, promote));
    pawnEat1Up = new Move<>(new Vector(1, 1), false, true, List.of(noCollision, mustEat),
    List.of(eatAction, promote));
    pawnStraight2Up = new Move<>(new Vector(0,2), false, false, List.of(noCollision, cannotEat,
    onlyFirstMove), List.of(moveChessPiece));
    pawnEnPassantUp = new Move<>(new Vector(1, 1), false, true, List.of(noCollision,
    pawnEnPassant), List.of(pawnEnPassant, moveChessPiece));
```

```
        pawnStraight1Down = new Move<>(new Vector(0, -1), false, false, List.of(noCollision,
        cannotEat), List.of(moveChessPiece, promote));
        pawnEat1Down = new Move<>(new Vector(-1, -1), false, true, List.of(noCollision, mustEat),
        List.of(eatAction, promote));
        pawnStraight2Down = new Move<>(new Vector(0,-2), false, false, List.of(noCollision, cannotEat,
        onlyFirstMove), List.of(moveChessPiece));
        pawnEnPassantDown = new Move<>(new Vector(-1, -1), false, true, List.of(noCollision,
        pawnEnPassant), List.of(pawnEnPassant, moveChessPiece));

        knightL = new Move<>(new Vector(2, 1), true, true, null, List.of(eatAction));
        knightL2 = new Move<>(new Vector(1, 2), true, true, null, List.of(eatAction));
    }

    /**
     * Init all chess rules
     */
    protected void initRules(){
        promote = new Promote();
        roque = new Roque();
        pawnEnPassant = new EnPassant();

        eatAction = new EatPiece();
        moveChessPiece = new MoveChessPiece();
        mustEat = new MustEat();
        cannotEat = new CanNotEat();
        onlyFirstMove = new OnlyFirstMove();
        noCollision = new MustNotCollide();
    }

    /**
     * Init all pieces available in chess
     */
    @Override
    protected void initPieces() {
        super.initPieces();
        for (ChessColor color: ChessColor.values()) {
            // Pawns
            for (int i = 0; i < getLENGTH(); i++) {
                setPieceAtPosition(new Pawn(color, this),
                color.getDirection().adjacent()[0].startingEdge(i),
                        color.getDirection().startingEdge(Pawn.STARTING_ROW_FROM_EDGE));
            }
            setPieceAtPosition(new Queen(color, this),
            color.getDirection().adjacent()[0].startingEdge(Queen.STARTING_COLUMN_FROM_EDGE),
                    color.getDirection().startingEdge(Queen.STARTING_ROW_FROM_EDGE));
            setPieceAtPosition(new King(color, this),
            color.getDirection().adjacent()[0].startingEdge(King.STARTING_COLUMN_FROM_EDGE),
                    color.getDirection().startingEdge(King.STARTING_ROW_FROM_EDGE));

            // for symetric pairs
            for (Direction d: color.getDirection().adjacent()) {
                setPieceAtPosition(new Rook(color, this), d.startingEdge(Rook.STARTING_COLUMN_FROM_EDGE),
                        color.getDirection().startingEdge(Rook.STARTING_ROW_FROM_EDGE));

                setPieceAtPosition(new Knight(color, this),
                d.startingEdge(Knight.STARTING_COLUMN_FROM_EDGE),
                        color.getDirection().startingEdge(Knight.STARTING_ROW_FROM_EDGE));

                setPieceAtPosition(new Bishop(color, this),
                d.startingEdge(Bishop.STARTING_COLUMN_FROM_EDGE),
                        color.getDirection().startingEdge(Bishop.STARTING_ROW_FROM_EDGE));
            }

        }
    }

    /**
     * Move a piece
     * @param from Vector from which the request move is made
     * @param to Vector to which the request move is made
     * @return Either the move is made or not
     */
    public boolean move(Vector from, Vector to){
        Objects.requireNonNull(from, "from vector must be non null");
        Objects.requireNonNull(to, "to vector must be non null");
        if(!isStarted)
            return false;
```

```
        boolean status = false;
        ChessPiece movedPiece = getPieceAtPosition(from);
        if(movedPiece != null
                && movedPiece.getColor() == turn
                && super.move(from, to)){

            turn = turn.next();
            if(checkmate(turn)) {
                endGame(turn.next());
            }
            status = true;

        }

        return status;
    }

    /**
     * End the current game
     * @param winner Winner color
     */
    protected void endGame(ChessColor winner){
        Objects.requireNonNull(winner, "winner must be non null");
        isStarted = false;
    }

    /**
     * Get the current chess
     * @return Current chess
     */
    @Override
    public Chess self() {
        return this;
    }

    /**
     * Check for King in check
     * @param defendingColor Defending color
     * @return Either the King is in check or not
     */
    public boolean check(ChessColor defendingColor){
        Objects.requireNonNull(defendingColor, "defending color must be non null");
        // Search for the King
        for (Vector positionKing : searchPieces(new King(defendingColor, this))) {
            if(isAttacked(defendingColor, positionKing))
                return true;
        }
        return false;
    }

    /**
     * Check for checkmate
     * @param defendingColor Defending color
     * @return Either the King is checkmated or not
     */
    public boolean checkmate(ChessColor defendingColor){
        Objects.requireNonNull(defendingColor, "defending color must be non null");
        if(!check(defendingColor)){
            return false;
        }
        // Search for the King
        for (Vector positionKing : searchPieces(new King(defendingColor, this))) {
            // for each pieces that player controls
            for (Vector positionPiece : searchPieces(defendingColor)) {
                if(getPieceAtPosition(positionPiece).possibleMoves(positionPiece).size() > 0){
                    return false;
                }
            }
        }
        return true;
    }

    /**
     * Get piece at a given position
     * @param position The position to get the piece
     * @return Piece on given position
```

```
     */
    @Override
    public ChessPiece getPieceAtPosition(Vector position) {
        return (ChessPiece) super.getPieceAtPosition(position);
    }

    /**
     * Remove piece at a given position
     * @param position Position to remove the given piece
     * @return Removed piece
     */
    @Override
    public ChessPiece removePieceAtPosition(Vector position) {
        return (ChessPiece) super.removePieceAtPosition(position);
    }

    /**
     * Set piece at given position
     * @param piece Piece to set at given indexes
     * @param i Index i to set the piece on
     * @param j Index j to set the piece on
     * @return Set piece
     */
    @Override
    public ChessPiece setPieceAtPosition(Piece<Chess> piece, int i, int j) {
        return (ChessPiece) super.setPieceAtPosition(piece, i, j);
    }

    /**
     * Move a piece to a given position
     * @param from Vector from where to move the piece
     * @param to Vector to where to move the piece
     * @return Moved piece
     */
    @Override
    public ChessPiece movePieceAtPosition(Vector from, Vector to) {
        return (ChessPiece) super.movePieceAtPosition(from, to);
    }

    /**
     * Lists all pieces of the same color
     * @param colorToSearch Piece color to search
     * @return List of all position of pieces of the same color
     */
    public ArrayList<Vector> searchPieces(ChessColor colorToSearch){
        Objects.requireNonNull(colorToSearch, "color to search must be non null");
        ArrayList<Vector> foundPieces = new ArrayList<>();
        for (int i = 0; i < getLENGTH(); i++) {
            for (int j = 0; j < getHEIGHT(); j++) {
                Vector pos = new Vector(i, j);
                ChessPiece pieceOnPosition = getPieceAtPosition(pos);
                if (pieceOnPosition != null && pieceOnPosition.getColor() == colorToSearch) {
                    foundPieces.add(pos);
                }
            }
        }
        return foundPieces;
    }

    /**
     * Check if the piece is attacked
     * @param defendingColor Defending piece color
     * @param position Position to check if attacked
     * @return Either the pice at position is attacked or not
     */
    public boolean isAttacked(ChessColor defendingColor, Vector position) {
        Objects.requireNonNull(defendingColor, "defending color must be non null");
        Objects.requireNonNull(position, "position vector must be non null");

        for (int i = 0; i < getLENGTH(); i++) {
            for (int j = 0; j < getHEIGHT(); j++) {
                ChessPiece pieceOnPosition = getPieceAtPosition(new Vector(i, j));
                if (pieceOnPosition != null && !pieceOnPosition.getColor().equals(defendingColor)) {
                    if (pieceOnPosition.move(i, j, position.getI(), position.getJ(), false)) {
                        return true;
                    }
                }
```

```
            }
        }
        return false;
    }

    /**
     * Check if a move is  produce a check
     * @param piece Piece to check on
     * @param start Starting position
     * @param destination Ending position
     * @param moveType Move type
     * @return Either the move is producing a check or not
     */
    public boolean doesMoveCheck(ChessPiece piece, Vector start, Vector destination, Move<Chess>
    moveType){
        Objects.requireNonNull(piece, "piece must be non null");
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(moveType, "moveType must be non null");
        piece.doMove(start, destination, moveType);
        boolean isCheck = check(turn);
        historicMoves.revertLastMove();
        return isCheck;
    }

    /**
     * Get promoted piece
     * @return Affected piece
     */
    public ChessPiece getPromotedPiece(){
        return new Queen(getTurn(), this);
    }

    /**
     * Available chess pieces
     */
    public abstract class ChessPiece extends  Piece<Chess>{
        private final ChessColor color;

        /**
         * ChessPiece constructor
         * @param color Piece color
         * @param board Concerned Board
         * @param moves List of all moves
         */
        public ChessPiece(ChessColor color, Chess board, List<Move<Chess>> moves) {
            super(Objects.requireNonNull(board, "board must be non null"), moves);
            this.color = Objects.requireNonNull(color, "color must be non null");
        }

        /**
         * Get the piece color
         * @return Piece color
         */
        public ChessColor getColor() {
            return color;
        }

        /**
         * Check if a move is legit
         * @param start Vector from where the action starts
         * @param destination Vector to where the action ends
         * @param moveType Move type
         * @return Either the move is legit or not
         */
        @Override
        protected boolean canMove(Vector start, Vector destination, Move<Chess> moveType) {
            Objects.requireNonNull(start, "start vector must be non null");
            Objects.requireNonNull(destination, "destination vector must be non null");
            Objects.requireNonNull(moveType, "moveType must be non null");
            ChessPiece pieceEaten = getPieceAtPosition(destination);
            if(pieceEaten != null && pieceEaten.getColor() == getColor()){
                return false;
            }
            if(super.canMove(start, destination, moveType)){
                return !doesMoveCheck(this, start, destination, moveType);
            }
```

```
            return false;
        }

        /**
         * Perform a move
         * @param start Vector from where the move starts
         * @param destination Vector to where the move ends
         * @param moveType Move type
         */
        @Override
        protected void doMove(Vector start, Vector destination, Move<Chess> moveType) {
            Objects.requireNonNull(start, "start vector must be non null");
            Objects.requireNonNull(destination, "destination vector must be non null");
            Objects.requireNonNull(moveType, "moveType must be non null");
            super.doMove(start, destination, moveType);
        }

        /**
         * Get the piece type
         * @return Piece type
         */
        public abstract ChessPieceType getPieceType();

        /**
         * Genereate Chess hash code
         * @return
         */
        @Override
        public int hashCode() {
            return Objects.hash(getPieceType().hashCode(), getColor());
        }

        /**
         * Check if the Chess is equal to an object
         * @param o Object to check with
         * @return Either the Chess is equal to the object or not
         */
        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;
            ChessPiece that = (ChessPiece) o;
            return color == that.color;
        }
    }

    /**
     * Bishop piece for Chess
     */
    public class Bishop extends ChessPiece{
        private static final int STARTING_COLUMN_FROM_EDGE = 2;
        private static final int STARTING_ROW_FROM_EDGE = 0;

        /**
         * Bishop constructor
         * @param color Color of the Bishop
         * @param chess Concerned chess
         */
        public Bishop(ChessColor color, Chess chess) {
            super(color, chess, List.of(diagonals));
        }

        /**
         * Get Bishop type
         * @return Bishop type
         */
        @Override
        public ChessPieceType getPieceType() {
            return ChessPieceType.BISHOP;
        }

    }

    /**
     * Rook piece for Chess
     */
    public class Rook extends ChessPiece{
```

```
        private static final int STARTING_COLUMN_FROM_EDGE = 0;
        private static final int STARTING_ROW_FROM_EDGE = 0;

        /**
         * Rook constructor
         * @param color Color of the rook
         * @param chess Concerned color
         */
        public Rook(ChessColor color, Chess chess) {
            super(color, chess, List.of(verticalStraights, horizontalStraights));
        }

        /**
         * Get Rook type
         * @return Rook type
         */
        @Override
        public ChessPieceType getPieceType() {
            return ChessPieceType.ROOK;
        }
    }

    /**
     * Queen piece for Chess
     */
    public class Queen extends ChessPiece{
        private static final int STARTING_COLUMN_FROM_EDGE = 3;
        private static final int STARTING_ROW_FROM_EDGE = 0;

        /**
         * Queen constructor with King
         * @param color Queen color
         * @param chess Concerned chess
         */
        public Queen(ChessColor color, Chess chess){
            super(color, chess, List.of(verticalStraights, horizontalStraights, diagonals));
        }

        /**
         * Get Queen type
         * @return Queen type
         */
        @Override
        public ChessPieceType getPieceType() {
            return ChessPieceType.QUEEN;
        }
    }

    /**
     * King piece for Chess
     */
    public class King extends ChessPiece{
        private static final int STARTING_COLUMN_FROM_EDGE = 4;
        private static final int STARTING_ROW_FROM_EDGE = 0;

        /**
         * King constructor
         * @param color Color of the King
         * @param chess Concerned chess
         */
        public King(ChessColor color, Chess chess){
            super(color, chess, List.of(kingVerticalStraights, kingHorizontalStraights, kingDiagonals,
            kingGrandRoque, kingPetitRoque));
        }

        /**
         * Get King type
         * @return King type
         */
        @Override
        public ChessPieceType getPieceType() {
            return ChessPieceType.KING;
        }
    }

    /**
     * Knight piece for Chess
```

```
     */
    public class Knight extends ChessPiece{
        private static final int STARTING_COLUMN_FROM_EDGE = 1;
        private static final int STARTING_ROW_FROM_EDGE = 0;

        /**
         * Knight constructor
         * @param color Color of the knight
         * @param chess Concerned chess
         */
        public Knight(ChessColor color, Chess chess){
            super(color, chess, List.of(knightL, knightL2));
        }

        /**
         * Get King type
         * @return King type
         */
        @Override
        public ChessPieceType getPieceType() {
            return ChessPieceType.KNIGHT;
        }
    }

    /**
     * Pawn piece for Chess
     */
    public class Pawn extends ChessPiece{
        private static final int STARTING_ROW_FROM_EDGE = 1;

        /**
         * Pawn constructor
         * @param color
         * @param chess
         */
        public Pawn(ChessColor color, Chess chess){
            super(color,
                    chess, color.getDirection() == Direction.UP ?
                            List.of(pawnStraight1Up, pawnStraight2Up, pawnEat1Up, pawnEnPassantUp)
                            : List.of(pawnStraight1Down, pawnStraight2Down, pawnEat1Down,
                            pawnEnPassantDown));
        }

        /**
         * Get Pawn type
         * @return Pawn type
         */
        @Override
        public ChessPieceType getPieceType() {
            return ChessPieceType.PAWN;
        }

    }

}
```

```java
package engine.game.chess;

/**
 * Chess player color
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public enum ChessColor {
    WHITE("White"){
        /**
         * Get White Direction
         * @return
         */
        @Override
        Chess.Direction getDirection() {
            return Chess.Direction.UP;
        }

        /**
         * Get next turn Color
         * @return
         */
        @Override
        ChessColor next() {
            return BLACK;
        }
    },
    BLACK("Black"){
        /**
         * Get Black Direction
         * @return
         */
        @Override
        Chess.Direction getDirection() {
            return Chess.Direction.DOWN;
        }

        /**
         * Get next turn Color
         * @return
         */
        @Override
        ChessColor next() {
            return WHITE;
        }
    };

    @Override
    public String toString() {
        return text;
    }

    private final String text;

    ChessColor(String text) {
        this.text = text;
    }

    /**
     * Get Direction
     * @return
     */
    abstract Chess.Direction getDirection();

    /**
     * Get next turn Color
     * @return
     */
    abstract ChessColor next();

    /**
     * Get promotion row
     * @return Promotion row index
     */
    int getPromotionRow(){
        return getDirection().opposite().startingEdge();
    }
```

```
}
```

```java
package engine.game.chess;

import engine.game.board.Board;
import engine.game.board.Piece;
import engine.game.board.Vector;
import java.util.Objects;

/**
 * Action when eating a piece
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class EatPiece extends MoveChessPiece {

    /**
     * Perform action when eating a piece
     * @param start Start position of the action
     * @param destination Desination of the action
     * @param board Concerned board
     * @return Removed piece
     */
    @Override
    public Piece<Chess> doAction(Vector start, Vector destination, Board<Chess> board) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(board, "chess board must be non null");
        Piece<Chess> removedPiece = board.removePieceAtPosition(destination);
        super.doAction(start, destination, board);
        return removedPiece;
    }

    /**
     * Revert action when a piece is eaten
     * @param start Start position of the action
     * @param destination Desination of the action
     * @param board Concerned board
     */
    @Override
    public void revertAction(Vector start, Vector destination, Piece<Chess> affectedPiece, Board<Chess>
    board) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(board, "chess board must be non null");
        super.revertAction(start, destination, affectedPiece, board);
        if(affectedPiece != null)
             board.setPieceAtPosition(affectedPiece, destination);
    }
}
```

```java
package engine.game.chess;

import engine.game.board.*;
import java.util.Objects;

/**
 * Action and condition for En passant special move
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class EnPassant implements GameAction<Chess>, GameCondition<Chess> {

    /**
     * Check condition for En passant
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param chess Concerned Chessboard
     * @return Either the condition for En passant is accepted
     */
    @Override
    public boolean checkCondition(Vector start, Vector destination, Board<Chess> chess) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(chess, "chess board must be non null");
        Chess.ChessPiece pieceToEat = chess.self().getPieceAtPosition(eatPosition(start, destination));
        return (pieceToEat instanceof Chess.Pawn
            && chess.lastPieceMoved().equals(pieceToEat)
            && (chess.isLastAction(chess.self().getPawnStraight2Up()) ||
            (chess.isLastAction(chess.self().getPawnStraight2Down()))));
    }

    /**
     * Get the victim of the En passant
     * @param start Start position
     * @param destination Destination position
     * @return Position of the victime piece
     */
    protected Vector eatPosition(Vector start, Vector destination){
        return new Vector(destination.getI(), start.getJ());
    }

    /**
     * Perform En passant action
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param chess Concerned ChessBoard
     * @return Eaten piece
     */
    @Override
    public Piece<Chess> doAction(Vector start, Vector destination, Board<Chess> chess) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(chess, "chess board must be non null");
        return chess.removePieceAtPosition(eatPosition(start, destination));
    }

    /**
     * Revert En passant action
     * @param start Vector from where the action started
     * @param destination Vector to where the action ended
     * @param affectedPiece Affected pieces by the action
     * @param chess Concerned ChessBoard
     */
    @Override
    public void revertAction(Vector start, Vector destination, Piece<Chess> affectedPiece, Board<Chess>
    chess) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(chess, "chess board must be non null");
        if(affectedPiece != null)
            chess.setPieceAtPosition(affectedPiece, eatPosition(start, destination));
    }
}
```

```java
package engine.game.chess;

import engine.game.board.*;
import java.util.Objects;

/**
 * Action and condition for ChessPiece move
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class MoveChessPiece implements GameCondition<Chess>, GameAction<Chess> {

    /**
     * Perform a ChessPiece move
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param board Concerned ChessBoard
     * @return Moved piece
     */
    @Override
    public Piece<Chess> doAction(Vector start, Vector destination, Board<Chess> board) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(board, "chess board must be non null");
        board.movePieceAtPosition(start, destination);
        return null;
    }

    /**
     * Revert ChessPiece move
     * @param start Vector from where the action started
     * @param destination Vector to where the action ended
     * @param affectedPiece Affected pieces by the action
     * @param board Concerned ChessBoard
     */
    @Override
    public void revertAction(Vector start, Vector destination, Piece<Chess> affectedPiece, Board<Chess>
    board) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(board, "chess board must be non null");
        board.movePieceAtPosition(destination, start);
    }

    /**
     * Check if a ChessPiece move is valid
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param board Concerned ChessBoard
     * @return Either the move is valid or not
     */
    @Override
    public boolean checkCondition(Vector start, Vector destination, Board<Chess> board) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(board, "chess board must be non null");
        Chess.ChessPiece piece = board.self().getPieceAtPosition(destination);
        if(piece != null)
            return piece.getColor() != board.self().getPieceAtPosition(start).getColor();
        return true;
    }
}
```

```java
package engine.game.chess;

import engine.game.board.Board;
import engine.game.board.GameCondition;
import engine.game.board.Vector;

import java.util.Objects;

/**
 * Condition when a piece should eat
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class MustEat implements GameCondition<Chess> {

    /**
     * Check if destination contains a piece to be eaten
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param board Concerned Board on which the action will be performed
     * @return Either the condition is valid or not
     */
    @Override
    public boolean checkCondition(Vector start, Vector destination, Board<Chess> board) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(board, "chess board must be non null");
        return board.getPieceAtPosition(destination) != null;
    }
}
```

```java
package engine.game.chess;

import engine.game.board.Board;
import engine.game.board.GameCondition;
import engine.game.board.Vector;

import java.util.Objects;

/**
 * Condition to check if there is no check
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class MustNotCheck implements GameCondition<Chess> {
    /**
     * Check if during a movement there is a check
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param chess Concerned ChessBoard
     * @return Either the condition is valid or not
     */
    @Override
    public boolean checkCondition(Vector start, Vector destination, Board<Chess> chess) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(chess, "chess board must be non null");
        Vector movementVector = new Vector(destination.getI() - start.getI(), destination.getJ() -
        start.getJ());
        for (Vector squareMovedThrough: movementVector.includedVectors()) {
            if(chess.self().isAttacked(chess.self().getPieceAtPosition(start).getColor(),
            start.add(squareMovedThrough))){
                return false;
            }
        }
        return !chess.self().isAttacked(chess.self().getPieceAtPosition(start).getColor(), destination);
    }
}
```

```java
package engine.game.chess;

import engine.game.board.Board;
import engine.game.board.GameCondition;
import engine.game.board.Vector;
import java.util.Objects;

/**
 * Condition to check if there is no collision between pieces
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class MustNotCollide implements GameCondition<Chess> {

    /**
     * Check if there is no collision between pieces
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param board Concerned ChessBoard
     * @return  Either the condition is valid or not
     */
    @Override
    public boolean checkCondition(Vector start, Vector destination, Board<Chess> board) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(board, "chess board must be non null");
        for (Vector positionOffset: destination.sub(start).includedVectors()) {
            Vector coordinates = start.add(positionOffset);
            if(board.getPieceAtPosition(coordinates) != null){
                return false;
            }
        }
        return true;
    }
}
```

```java
package engine.game.chess;

import engine.game.board.Board;
import engine.game.board.GameCondition;
import engine.game.board.Vector;

import java.util.Objects;

/**
 * Condition for one first moves
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class OnlyFirstMove implements GameCondition<Chess> {
    /**
     * Check if the piece has not moved before
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param board Concerned Board on which the action will be performed
     * @return Either the condition is valid or not
     */
    @Override
    public boolean checkCondition(Vector start, Vector destination, Board<Chess> board) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(board, "chess board must be non null");
        return !board.hasMoved(board.getPieceAtPosition(start));
    }
}
```

```java
package engine.game.chess;

import engine.game.board.Board;
import engine.game.board.GameAction;
import engine.game.board.Piece;
import engine.game.board.Vector;

import java.util.Objects;

/**
 * Action for promotion
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class Promote implements GameAction<Chess> {

    /**
     * Perform promotion
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param chess Concerned Board on which the action will be performed
     * @return Affected piece
     */
    @Override
    public Piece<Chess> doAction(Vector start, Vector destination, Board<Chess> chess) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(chess, "chess board must be non null");

        Chess.ChessPiece candidatePiece = chess.self().getPieceAtPosition(destination);
        Piece<Chess> affectedPiece = null;
        if(destination.getJ() == candidatePiece.getColor().getPromotionRow()) {
            Chess.ChessPiece piece = chess.self().getPromotedPiece();
            if(piece != null){
                affectedPiece = chess.self().removePieceAtPosition(destination);
                chess.setPieceAtPosition(piece, destination);
            }
        }
        return affectedPiece;
    }

    /**
     * Revert promotion
     * @param start Vector from where the action started
     * @param destination Vector to where the action ended
     * @param affectedPiece Affected pieces by the action
     * @param chess Concerned Board on which the action will be performed
     */
    @Override
    public void revertAction(Vector start, Vector destination, Piece<Chess> affectedPiece, Board<Chess>
    chess) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(chess, "chess board must be non null");

        if(affectedPiece != null){
            chess.setPieceAtPosition(affectedPiece, destination);
        }
    }
}
```

```java
package engine.game.chess;

import engine.game.board.*;
import engine.game.board.Vector;

import java.util.*;

/**
 * Action and condition for roque
 * @author Alen Bijelic
 * @author Nelson Jeanrenaud
 */
public class Roque extends MustNotCheck implements GameAction<Chess>, GameCondition<Chess> {

    private static final Set<Chess.ChessPieceType> CAN_ROQUE_WITH = EnumSet.of(
            Chess.ChessPieceType.ROOK);

    /**
     * Perform roque
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param chess Concerned Board on which the action will be performed
     * @return Affected Pieces
     */
    @Override
    public Piece<Chess> doAction(Vector start, Vector destination, Board<Chess> chess) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(chess, "chess board must be non null");
        movePiece(start, getPieceRoqueDestination(start, destination), chess);
        movePiece(getPieceRoqueWithPosition(start, destination, chess),
        getPieceRoqueWithDestination(start, destination, chess), chess);
        return null;
    }

    /**
     * Revert roque
     * @param start Vector from where the action started
     * @param destination Vector to where the action ended
     * @param affectedPiece Affected pieces by the action
     * @param chess Concerned Board on which the action will be performed
     */
    @Override
    public void revertAction(Vector start, Vector destination, Piece<Chess> affectedPiece, Board<Chess>
    chess) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(chess, "chess board must be non null");
        movePiece(getPieceRoqueDestination(start, destination), start, chess);
        movePiece(getPieceRoqueWithDestination(start, destination, chess),
        getPieceRoqueWithPosition(start, destination, chess), chess);
    }

    /**
     * Get the Rook with which roque
     * @param start Vector from where the action started
     * @param destination Vector to where the action ended
     * @param chess Concerned Board on which the action will be performed
     * @return Roquing with Rook position
     */
    private Vector getPieceRoqueWithPosition(Vector start, Vector destination, Board<Chess> chess){
        return (destination.getI() < start.getI())
                ? new Vector(0, start.getJ())
                : new Vector(chess.self().getLENGTH() - 1, start.getJ());
    }

    /**
     * Get Rook destination
     * @return Rook destination position
     */
    private Vector getPieceRoqueDestination(Vector start, Vector destination) {
        return start.add(new Vector(destination.getI() - start.getI(), destination.getJ() -
        start.getJ()));
    }

    /**
     * Get Rook destination
```

```
     * @param start Vector from where the action started
     * @param destination Vector to where the action ended
     * @param chess Concerned Board on which the action will be performed
     * @return Rook destination position
     */
    private Vector getPieceRoqueWithDestination(Vector start, Vector destination, Board<Chess> chess) {
        Vector pieceRoqueVector = getPieceRoqueDestination(start, destination);
        Vector pieceRoqueWithVector = getPieceRoqueWithPosition(start, destination, chess);

        return  pieceRoqueWithVector.getI() < pieceRoqueVector.getI()
                ? new Vector(pieceRoqueVector.getI() + 1, pieceRoqueVector.getJ())
                : new Vector(pieceRoqueVector.getI() - 1, pieceRoqueVector.getJ());
    }

    /**
     * Move pieces concerned by Roque
     * @param start Vector from where the action started
     * @param destination Vector to where the action ended
     * @param chess Concerned Board on which the action will be performed
     */
    protected void movePiece(Vector start, Vector destination, Board<Chess> chess) {
        Piece<Chess> piece = chess.self().removePieceAtPosition(start);
        chess.self().setPieceAtPosition(piece, destination);
    }

    /**
     * Check if all conditions for a Roque are valid
     * @param start Vector from where the action starts
     * @param destination Vector to where the action ends
     * @param chess Concerned ChessBoard
     * @return Either the condition is valid or not
     */
    @Override
    public boolean checkCondition(Vector start, Vector destination, Board<Chess> chess) {
        Objects.requireNonNull(start, "start vector must be non null");
        Objects.requireNonNull(destination, "destination vector must be non null");
        Objects.requireNonNull(chess, "chess board must be non null");

        Chess.ChessPiece pieceOnStart = chess.self().getPieceAtPosition(start);
        if(pieceOnStart == null)
            return false;

        Chess.ChessPiece pieceOnDestination =
        chess.self().getPieceAtPosition(getPieceRoqueWithPosition(start, destination, chess));

        Vector legitVector = new Vector(2, 0);
        Vector movementVector = new Vector(destination.getI() - start.getI(), destination.getJ() -
        start.getJ());

        int begin = movementVector.getI() < 0 ? 1 : start.getI() + 1;
        int end = movementVector.getI() < 0 ? start.getI() - 1 : chess.self().getLENGTH() - 1;

        // Check for Piece between the piece who Roque and the piece we're roquing with
        for(int i = begin; i < end; ++i) {
            if(chess.self().getPieceAtPosition(new Vector(i, start.getJ())) != null){
                return false;
            }
        }

        return  pieceOnDestination != null
                && CAN_ROQUE_WITH.contains(pieceOnDestination.getPieceType())
                && legitVector.areCollinear(movementVector)
                && legitVector.norm() == movementVector.norm()
                && !chess.self().hasMoved(pieceOnStart)
                && !chess.self().hasMoved(pieceOnDestination)
                && super.checkCondition(start, destination, chess);
    }
}
```