Design chess

Note:

① Valid moves from (x_1, y_1) to (x_2, y_2)

- King: $|x_2 - x_1| \leq 1$ and $|y_2 - y_1| \leq 1$
- Bishop: $|x_2 - x_1| = |y_2 - y_1|$
- Rook: $x_2 = x_1$ or $y_2 = y_1$
- Queen: valid if it is valid move for bishop or rook
- Knight: $|x_2 - x_1| * |y_2 - y_1| = 2$
- Pawn: $x_2 = x_1$ and $y_2 - y_1 = 1$.

② Classes

- Board
- Spot
- Piece (abstract)
  - King
  - Bishop
  - Rook
  - Queen
  - Knight
  - Pawn
- Game
- Player — [Human, computer]
- Move

③ Castling rules

- Your king (or rook) has not moved
- _____ is not in check
- _____ does not pass through a check
- No pieces between the king and the rook.

```
public abstract class Piece {
    private boolean killed;
    private boolean white;

    public Piece (boolean white) {
        this.white = white;
    }

    // getters & setters.

    public abstract canMove (Board
        board, Spot start, Spot end);
}
```

```
public class Spot {
    private Piece piece;
    private int x;
    private int y;

    public Spot (int x, int y, Piece piece)
    {  this.x = x;
       this.y = y;
       this.piece = piece;
    }
    // getters & setters
}
```

```
public class Board {
    private Spot [ ][ ] boxes;

    public Board ( ) {
        this.resetBoard ( );
    }

    public Spot getBox (int x, int y) {
        if (x<0 || y<0 || x>7 || y>7) {
            throw new ChessException ("Invalid input");
        }
        return boxes [x] [y];
    }

    public void resetBoard ( ) {
        boxes [0] [0] = new Spot (0, 0, new Rook (true));
        boxes [0] [1] = ———— 0, 1, new Knight (true));
        boxes [1] [0] = ———— 1, 0, new Pawn (true));
        ==
    }
}.
```

```
public class Knight extends Piece {
    public Knight(boolean white) {
        super(white);
    }

    public boolean canMove(Board board, Spot start, Spot end) {
        if (end.piece.isWhite() == this.isWhite()) {
            return false;
        }

        int x = Math.abs(start.getX() - end.getX());
        int y = Math.abs(start.getY() - end.getY());
        return x * y == 2;
    }
}
```

Same color →

```
public class King extends Piece {
    boolean castlingDone = false;
    boolean hasMoved = false;

    public boolean canMove(Board board, Spot start, Spot end) {
        if (end.piece.isWhite() == this.isWhite()) {
            return false;
        }
        int x = Math.abs(start.getX() - end.getX());
        int y = Math.abs(start.getY() - end.getY());
        if (x <= 1 && y <= 1) {
            return true;
        }
        return this.isValidCastling(board, start, end);
    }

    public boolean isValidCastling(Board ____) {
        if (isCastlingDone()) {
            return false;
        }
        return validateCastling();
    }

    public boolean validateCastlingMove() {
```

```
public abstract class Player {
    boolean whiteSide;
    boolean human;
    ==
}
```

```
Public class HumanPlayer extends Player {
    public HumanPlayer(boolean whiteSide){
        this.whiteSide = whiteSide;
        this.human = true;
    }
}
public class ComputerPlayer extends Player {
    ——
}
```

```
public class Move {
    private Player player;
    —— Spot start;
    —— Spot end;
    —— Piece pieceMoved;
    —— Piece pieceKilled;
    —— boolean castlingMove = false;

    // getters and setters
}
```

```
public enum GameStatus{
    ACTIVE,
    BLACK_WIN,
    WHITE_WIN,
    FORFEIT,
    STALEMATE,
    RESIGNATION
}
```

```
public class Game {
    private Player white;
    private Player black;
    —— Board board,
       Player currentTurn;
       GameStatus status;
       List<Move> movesPlayed;
    public Game(Player white, Player black){
        this.initialize(white, black);
    }
    private void initialize(Player white, Player black){
        this.white = white;
        this.black = black;
        this.board = new Board();
        this.currentTurn = white;
        movesplayed = new LinkedList<>();
    }
```

```
public GameStatus getStatus() {
    return this.status;
}
```

```
private void setStatus(GameStatus status) {
    this.status = status;
}
```

```
public boolean playerMove(Player player, int startX, int startY,
                          int endX, int endY) {
    Spot start = board.getSpot(startX, startY);
    Spot end = board.getSpot(endX, endY);
    Move move = new Move(player, start, end);
    return this.makeMove(move);
}
```

```
public boolean makeMove(Move move) {
    Piece sourcePiece = move.getStart().piece;
    if (sourcePiece == null) {
        return false;
    }
```
Soure

```
    Player p1 = move.getPlayer();
    if (p1 != currentTurn) {
        return false;
    }
```
correct turn

```
    if (sourcePiece.isWhite() != p1.isWhiteSide()) {
        return false;
    }
```
correct color

```
    if (!sourcePiece.canMove(board, move.start, move.end)) {
        return false;
    }
```
valid move

```
    Piece destPiece = move.end.getPiece();
    if (destPiece != null) {
        destPiece.setKilled(true);
        move.setPieceKilled(destPiece);
    }
```
Killed

castling

```
if ( sourcePiece != null && sourcePiece instanceof King
        && sourcePiece.isCastlingMove() ) {
    move.setCastlingMove(true);
}

movesPlayed.add(move);
```

move done

```
move.getEnd().setPiece(move.getStart().getPiece());
move.getStart().setPiece(null);
```

is victory

```
if (destPiece != null && destPiece instanceof King) {
    this.setStatus(pl.isWhiteSide() ? GameStatus.WHITE_WIN
                                    : GameStatus.BLACK_WIN);
}

if (this.currentTurn == this.white) {
    this.currentTurn = this.black;
} else {
    this.currentTurn = this.white;
}
return true;
}
```