Mitzi - Exercise 1

Christoph Hofer 0955139

Stefan Lew 0856722

20. November 2013

1 Implementation of chess pieces

The different chess pieces and the two are represented as an `enum` in `Piece.java` and `Side.java`. The class `Side` additionally provides methods for getting the opposite side and the sign of the color (+ for white and - for black), which is particularly important board evaluation and negamax algorithm.

2 Implementation of a simple chess board

The chess board is implemented via the class `GameState`. The class stores the position of the pieces, counts the `full_move_clock`, `halfe_move_clock` and saves the history of all played moves. A move can be performed via the method `doMove()` which additionally checks if the move is valid or not.

2.1 Implementation of the position class

The class `Position` contains the main information of the chess board. The class contains among others:

- `side_board`: An array of 65 `Sides`, representing the color (side) if the several pieces.
- `piece_board`: An array of 65 `Pieces`, representing the piece on a the different squares.
- `castling`: An array, which contains the square where the king can castle. It contains -1, if it is not possible.
- `en_passant_target`: The square, where the en-passant target is positioned.
- `active_color`: The side, which has to move.
- `analysis_result`: This class stores information of the value of the position and is of no interest for this exercise.

The arrays contains `null` if at a square is no pieces. The additional entry is reserved for illegal squares and is always set to `null`. Furthermore the class stores data, which is computed once and reuses it.

The class is able to:

- read and set Pieces on the board.
- reset the board to initial state.
- compute an copy of the board, where only the necessary members are copied.

- compute all valid moves for the active side and for each square.
- perform a given move
- check if a move is valid
- check if a move is a hit
- check if castling for a side is possible
- check if the current position is a check, mate or stale mate position.
- return a string representation of the position (FEN notation, seehttp://en.wikipedia.org/wiki/Forsyth-Edwards_Notation)

2.2 Representation of squares

We represent the squares as integer, however we do not use the usual notation 1,2,3, . . . , but we use the so called ICCF numeric notation (see http://en.wikipedia.org/wiki/ICCF_numeric_notation). The class `SquareHelper` provides methods to work with the notation:

- conversion: `int` ↔ `[row][column]`
- check if a square is black or white.
- check if a square is valid.
- conversation to string representation.
- receiving squares in a given `Direction`.

2.3 The enum Direction

The enum `Direction` contains the offset for the integer value of the square for each direction. Since the knight does not use the usual directions, it needs a separate offset. To simplify the code, an additional function was generated to return the capturing direction for a pawn for a certain side.

3 Implementation of chess moves

The class `Move` implements a move in a chess game. A move consists of

- the source square
- the destination square

- a `Piece` representing the promotion of the pawn.

4 The random chess player

The random chess player implemented in `RandyBrain` uses the function `search` to choose randomly a possible move.

5 The Code

ChessGame.java

```
package mitzi;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.List;

import mitzi.IMove;
import mitzi.RandyBrain;

/**
 * The environment for playing chess
 *
 */
public class ChessGame {

  private static GameState game_state;

  public static void main(String[] args) {

    System.out.println("Lets play chess!");

    IMove move;

    game_state = new GameState();

    RandyBrain randy = new RandyBrain();
    HumanBrain human = new HumanBrain();
    while (true) {
      //Humans turn
      human.set(game_state);
      move = human.search(0, 0, 0, false, null);
```

```
      game_state.doMove(move);
      if (game_state.getPosition().isMatePosition()) {
        System.out.println("You won!");
        break;
      }
      if (game_state.getPosition().isStaleMatePosition()) {
        System.out.println("Draw!");
        break;
      }
      System.out.println(game_state.getPosition());

      //Randys turn
      randy.set(game_state);
      move = randy.search(0, 0, 0, false, null);
      System.out.println("Randy plays:" + move);
      game_state.doMove(move);
      if (game_state.getPosition().isMatePosition()) {
        System.out.println("You lost!");
        break;
      }
      if (game_state.getPosition().isStaleMatePosition()) {
        System.out.println("Draw!");
        break;
      }
      System.out.println(game_state.getPosition());

    }

  }

}
```

Direction.java

```
package mitzi;

import java.util.EnumSet;

/**
 * This class represents stores the information about the offset for moving a
 * piece from a square in a specific direction. The offset for a knight is
 * different for the other figures.
 */
public enum Direction {
```

```
EAST(10, 21), NORTHEAST(11, 12), NORTH(1, -8), NORTHWEST(-9, -19), WEST(
    -10, -21), SOUTHWEST(-11, -12), SOUTH(-1, 8), SOUTHEAST(9, 19);

/**
 * Add to a square value to go one step in the specified direction.
 *
 * White is South, Black is North.
 */
public final int offset;

/**
 * Add to a square value to go one knight-step in the specified direction.
 *
 * One up and two right is East. Two up one right is Northeast. Basically,
 * the orientation is shifted a bit counterclockwise.
 */
public final int knight_offset;

Direction(int offset, int knight_offset) {
  this.offset = offset;
  this.knight_offset = knight_offset;
}

/**
 * Returns the direction in which a pawn of the specified color can move
 * (without capturing).
 *
 * @param color
 *          the color of the piece
 * @return NORTH for white and SOUTH for black
 */
public static Direction pawnDirection(Side color) {
  if (color == Side.WHITE) {
    return NORTH;
  } else {
    return SOUTH;
  }
}

/**
 * Returns a set of directions in which a pawn of the specified color can
 * capture other pieces.
 *
 * @param color
 *          the color of the piece
 * @return the set of directions allowed
```

```
   */
  public static EnumSet<Direction> pawnCapturingDirections(Side color) {
    if (color == Side.WHITE) {
      return EnumSet.of(NORTHEAST, NORTHWEST);
    } else {
      return EnumSet.of(SOUTHEAST, SOUTHWEST);
    }
  }

}
```

Piece.java

```
package mitzi;

/**
 * An enum containing the different Pieces
 */
public enum Piece {
  PAWN, ROOK, BISHOP, KNIGHT, QUEEN, KING;
}
```

Side.java

```
package mitzi;

/**
 * An enum containing the two different sides.
 *
 */
public enum Side {
  BLACK, WHITE;

  /**
   * returns the opposite side of the given side
   * @param side the given side
   * @return the opposite side
   */
  public static Side getOppositeSide(Side side) {
    switch (side) {
    case BLACK:
      return WHITE;
```

```
    default:
      return BLACK;
    }
  }

  /**
   * returns the side sign of the given side
   * @param side the given side
   * @return -1 if side == black, 1 otherwise.
   */
  public static int getSideSign(Side side) {
    switch (side) {
    case BLACK:
      return -1;
    default:
      return +1;
    }
  }
}
```

IBrain.java

```
package mitzi;

import java.util.List;

public interface IBrain {

  /**
   * Before the engine is asked to search on a game state, there will always
      be
   * this command to tell the engine about the current game state.
   *
   * @param game_state
   *           the current game state
   */
  public void set(GameState game_state);

  /**
   * Start calculating on the current position.
   *
   * @param movetime
   *           search for exactly this time in milliseconds
   * @param maxMoveTime
```

```
   *           search for at most this time in milliseconds
   * @param searchDepth
   *           the maximum search depth in plys
   * @param infinite
   *           If set to true, search until the "stop" command. Do not exit
   *           the search without being told so in this mode!
   * @param searchMoves
   *           Restrict search to this moves only. If null, the engine may
   *           search any moves.
   * @return the hopefully best move
   */
  public IMove search(int movetime, int maxMoveTime, int searchDepth,
      boolean infinite, List<IMove> searchMoves);

  /**
   * Stop calculating immediately and return the best move.
   *
   * @return the currently best move
   */
  public IMove stop();

}
```

IMove.java

```
package mitzi;

public interface IMove {

  /**
   *
   * @return the source of the move
   */
  public int getFromSquare();

  /**
   *
   * @return the destination of the move
   */
  public int getToSquare();

  /**
   *
   * @return the promotion of the pawn. EMPTY if no promotion.
```

```
   */
  public Piece getPromotion();

  /**
   *
   * @return the string representation of the move
   */
  public String toString();

}
```

IPosition.java

```
package mitzi;

import java.util.List;
import java.util.Set;

/**
 * This class provides an interface for a generic chess for the positions on a
 * chess board.
 *
 */
public interface IPosition {

  /**
   * Sets the board to the initial position at the start of a game.
   */
  public void setToInitial();

  /**
   * Sets the board to a position given in Forsyth-Edwards Notation (FEN).
   *
   * @see <a
   *
      href="https://en.wikipedia.org/wiki/Forsyth-Edwards_Notation">Wikipedia
   *      - Forsyth-Edwards Notation</a>
   */
  public void setToFEN(String fen);

  /**
   *
   * This class represents the result of the doMove function and, recognizes
   * if the half_move_clock should be set.
```

```
 *
 */
public class MoveApplication {
  IPosition new_position;
  boolean resets_half_move_clock = false;
}

/**
 * Performs the given move and returns a new position. There is no check,
 * that the performed move is legal!
 *
 * @param move
 *           the move, which should be performed. Please note, that the
 *           move must be valid, no checking is done.
 * @return the new board and a boolean, if the half_move_clock should be
 *         reseted.
 */
public MoveApplication doMove(IMove move);

/**
 * Returns, which side has to move.
 *
 * @return the active Side of the actual position
 */
public Side getActiveColor();

/**
 * En passant target square. If there's no en passant target square, this is
 * -1. If a pawn has just made a two-square move, this is the position
 * "behind" the pawn. This is recorded regardless of whether there is a pawn
 * in position to make an en passant capture.
 *
 * @return the square "behind" the pawn which can be take en passant
 */
public int getEnPassant();

/**
 * Check if the king can use castling to get to a specified square.
 *
 * @param king_to
 *           the square to be checked
 *
 * @return true if the king is allowed to move to the square by castling
 *
 * @see <a href="http://www.fide.com/fide/handbook?id=124&view=article">FIDE
 *      Rule 3.8</a>
```

```
 */
public boolean canCastle(int king_to);

/**
 * The position stores also an eventual analysis result from board
 * evaluation.
 *
 * @return the analysis result of the board.
 */
public AnalysisResult getAnalysisResult();

/**
 * Sets/update the actual analysis result.
 *
 * @param new_result
 *           the new analysis result.
 */
public void updateAnalysisResult(AnalysisResult new_result);

/**
 * Checks if a given side, can still castle.
 *
 * @param color
 *           the given side
 * @return true, if the given side can castle, false else.
 */
public Boolean colorCanCastle(Side color);

/**
 * Returns all squares, occupied by a given side.
 *
 * @param color
 *           the given side
 * @return a set of integers, containing all squares, where a piece of this
 *         side is placed.
 */
public Set<Integer> getOccupiedSquaresByColor(Side color);

/**
 * Returns all squares, occupied by a given piece.
 *
 * @param type
 *           the given piece
 * @return a set of integers, containing all squares, where this piece is
 *         placed.
 */
```

```
public Set<Integer> getOccupiedSquaresByType(Piece type);

/**
 * Returns all squares, occupied by a given piece and side.
 *
 * @param color
 *           the given side
 * @param type
 *           the given piece
 * @return a set of integers, containing all squares, where the piece of
 *         this side is placed.
 */
public Set<Integer> getOccupiedSquaresByColorAndType(Side color, Piece
    type);

/**
 * Returns the number of occupied squares by a given side.
 *
 * @param color
 *           the given side
 * @return the number of squares, where a piece of the given side is placed.
 */
public int getNumberOfPiecesByColor(Side color);

/**
 * Returns the number of occupied squares by a given piece.
 *
 * @param type
 *           the given piece
 * @return the number of squares, where the piece is placed.
 */
public int getNumberOfPiecesByType(Piece type);

/**
 * Returns the number of occupied squares by a given piece and side.
 *
 * @param color
 *           the given side
 * @param type
 *           the given piece
 * @return the number of squares, where the piece of this side is placed.
 */
public int getNumberOfPiecesByColorAndType(Side color, Piece type);

/**
 * Computes all possible moves for the active side. Moves, where the active
```

```
     * color is check, are invalid and got deleted.
     *
     * @return a set of all valid and possible moves.
     */
    public List<IMove> getPossibleMoves();

    /**
     * Computes all possible moves for the active side from a specific square.
     * Moves, where the active color is check, are invalid and got deleted.
     *
     * @param square
     *          the given square
     * @return a set of all valid and possible moves from the given square.
     */
    public List<IMove> getPossibleMovesFrom(int square);

    /**
     * Computes all possible moves for the active side to a specific square.
     * Moves, where the active color is check, are invalid and got deleted.
     * Please note, that this functions calls getPossibleMoves() and extracts
     * the desired ones.
     *
     * @param square
     *          the given square
     * @return a set of all valid and possible moves to the given square.
     */
    public List<IMove> getPossibleMovesTo(int square);

    /**
     * returns the side of the piece on a given square
     *
     * @param square
     *          the given square
     * @return the side, if this square is occupied by a side and null if it is
     *        empty.
     */
    public Side getSideFromBoard(int square);

    /**
     * returns the piece on a given square
     *
     * @param square
     *          the given square
     * @return the piece, if this square is occupied and null if it is empty.
     */
    public Piece getPieceFromBoard(int square);
```

```
/**
 * checks if the actual position is a check position.
 *
 * @return true if the position is a check position
 */
public boolean isCheckPosition();

/**
 * checks if the actual position is a mate position.
 *
 * @return true if the position is a mate position
 */
public boolean isMatePosition();

/**
 * checks if the actual position is a stalemate position.
 *
 * @return true if the position is a stalemate position
 */
public boolean isStaleMatePosition();

/**
 * checks if a given move is a valid move. Note, that this function calls
 * first getPossibleMoves() and then searches the given move in all possible
 * moves
 *
 * @param move
 *           the move to be checked
 * @return true, if the move is possible
 */
public boolean isPossibleMove(IMove move);

/**
 * converts the given position in fen notation
 *
 * @return a string of the actual position in fen notation
 */
public String toFEN();

/**
 * searches all moves, which are a capture and promotions
 *
 * @return the desired set of moves of all captures and promotions.
 */
public List<IMove> generateCaptures();
```

```
  /**
   * Since AnalysisResults are stored in the Transposition Tables
   * (ResultCache), it is important to ensure that the AnalysisResult
   * corresponding to the actual position should be used, if there are
   * collisions with hashvalues. Therefore a second one (this one) is created
   * to identify the position and these problems unlikely.
   *
   * @return a different hashvalue
   */
  public long hashCode2();
}
```

RandyBrain.java

```
package mitzi;

import java.util.List;
import java.util.Random;

/**
 * This class implements the most basic search engine, the random move
 * selection. All possible moves of the actual game state are computed and one
 * of them is randomly selected.
 *
 */
public class RandyBrain implements IBrain {

  /**
   * The current game state
   */
  private GameState game_state;

  @Override
  public void set(GameState game_state) {
    this.game_state = game_state;
  }

  @Override
  public IMove search(int movetime, int maxMoveTime, int searchDepth,
      boolean infinite, List<IMove> searchMoves) {

    List<IMove> moves = game_state.getPosition().getPossibleMoves();
```

```
    int randy = new Random().nextInt(moves.size());
    int i = 0;
    for (IMove move : moves) {
      if (i == randy)
        return move;
      i = i + 1;
    }

    return null; // cannot not happen anyway
  }

  @Override
  public IMove stop() {
    // no need to implement the stop function, since RandyBrain is fast
    // enough.
    return null;
  }

}
```

HumanBrain.java

```
package mitzi;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

import mitzi.GameState;

public class HumanBrain implements IBrain {

  /**
   * The current game state
   */
  private GameState game_state;

  @Override
  public void set(GameState game_state) {
    this.game_state = game_state;
  }
```

```
  @Override
  public IMove search(int movetime, int maxMoveTime, int searchDepth,
      boolean infinite, List<IMove> searchMoves) {

    //Read in the move as string
    BufferedReader reader = new BufferedReader(new
        InputStreamReader(System.in));
    String string_move = null;
    try {
      string_move = reader.readLine();
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }

    //convert it to an object move.
    IMove move = new Move(string_move);
    //if the move was illegal, the player has to choose another one.
    while(!game_state.getPosition().isPossibleMove(move)){
      System.out.println("Illegal move, choose another one!");
      try {
        string_move = reader.readLine();
      } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
      move = new Move(string_move);
    }
    //return the choosen move.
    return move;
  }

  @Override
  public IMove stop() {
    return null;
  }

}
```

Move.java

```
package mitzi;

import java.util.Locale;
```

```
import java.util.Set;

public final class Move implements IMove {

  /**
   * the source square of the move
   */
  private final short src;

  /**
   * the destination square of the move
   */
  private final short dest;

  /**
   * the piece, resulting from promotion. null if no promotion
   */
  private final Piece promotion;

  /**
   * Move constructor
   *
   * @param src
   *           Source
   * @param dest
   *           Destination
   * @param promotion
   *           Promotion (if no, then omit)
   */
  public Move(int src, int dest, Piece promotion) {
    this.src = (short) src;
    this.dest = (short) dest;
    this.promotion = promotion;
  }

  /**
   * Move constructor (no promotion)
   *
   * @param src
   *           Source square
   * @param dest
   *           Destination square
   */
  public Move(int src, int dest) {
    this(src, dest, null);
  }
```

```
/**
 * Move constructor from string notation
 *
 * @param notation
 *           the string representation of the move
 */
public Move(String notation) {
  String[] squares = new String[2];

  squares[0] = notation.substring(0, 2);
  squares[1] = notation.substring(2, 4);

  src = (short) SquareHelper.fromString(squares[0]);
  dest = (short) SquareHelper.fromString(squares[1]);

  if (notation.length() > 4) {
    String promo_string = notation.substring(4, 5).toLowerCase(
        Locale.ENGLISH);
    if (promo_string.equals("q")) {
      promotion = Piece.QUEEN;
    } else if (promo_string.equals("r")) {
      promotion = Piece.ROOK;
    } else if (promo_string.equals("n")) {
      promotion = Piece.KNIGHT;
    } else if (promo_string.equals("b")) {
      promotion = Piece.BISHOP;
    } else {
      promotion = null;
    }
  } else {
    promotion = null;
  }
}

/**
 *
 * Checks if a move is in a given List of moves
 *
 * @param moves
 *           List of moves
 * @param move
 *           the move to be searched
 * @return true if move is in moves, else false
 */
public static boolean MovesListIncludesMove(Set<Move> moves, Move move) {
```

```
    return moves.contains(move);

  }

  @Override
  public int getFromSquare() {
    return src;
  }

  @Override
  public int getToSquare() {
    return dest;
  }

  @Override
  public Piece getPromotion() {
    return promotion;
  }

  @Override
  public String toString() {
    String promote_to;
    if (getPromotion() != null) {
      promote_to = PieceHelper.toString(Side.WHITE, getPromotion());
    } else {
      promote_to = "";
    }
    return SquareHelper.toString(getFromSquare())
        + SquareHelper.toString(getToSquare()) + promote_to;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + dest;
    result = prime * result
        + ((promotion == null) ? 0 : promotion.hashCode());
    result = prime * result + src;
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
```

```
    }
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    Move other = (Move) obj;
    if (dest != other.dest || promotion != other.promotion
        || src != other.src) {
      return false;
    }
    return true;
  }
}
```

Position.java

```
package mitzi;

import java.lang.ref.SoftReference;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;

/**
 * The class implements the position of the figures on a chess board. The
    board
 * is represented as two 8*8 +1 arrays - one for the sides, one for the
    pieces.
 * All accesses to a square outside the chessboard are mapped to the 65th
    entry
 * of the board, which is always null. This map from square to array index is
 * performed by the function <code>squareToArrayIndex(square) </code>, which
 * looks up in the <code>square_to_array_index array</code>. For informations
 * about the <code>int</code> value of a square, see
 * <code>SqaureHelper.java</code>.
 *
 */
```

```
public class Position implements IPosition {

  /**
   * the initial position of the sides
   */
  protected static Side[] initial_side_board = { Side.BLACK, Side.BLACK,
      Side.BLACK, Side.BLACK, Side.BLACK, Side.BLACK, Side.BLACK,
      Side.BLACK, Side.BLACK, Side.BLACK, Side.BLACK, Side.BLACK,
      Side.BLACK, Side.BLACK, Side.BLACK, Side.BLACK, null, null, null,
      null, null, null, null, null, null, null, null, null, null, null,
      null, null, null, null, null, null, null, null, null, null, null,
      null, null, null, null, null, null, null, Side.WHITE, Side.WHITE,
      Side.WHITE, Side.WHITE, Side.WHITE, Side.WHITE, Side.WHITE,
      Side.WHITE, Side.WHITE, Side.WHITE, Side.WHITE, Side.WHITE,
      Side.WHITE, Side.WHITE, Side.WHITE, Side.WHITE, null };

  /**
   * the initial position of the pieces
   */
  protected static Piece[] initial_piece_board = { Piece.ROOK, Piece.KNIGHT,
      Piece.BISHOP, Piece.QUEEN, Piece.KING, Piece.BISHOP, Piece.KNIGHT,
      Piece.ROOK, Piece.PAWN, Piece.PAWN, Piece.PAWN, Piece.PAWN,
      Piece.PAWN, Piece.PAWN, Piece.PAWN, Piece.PAWN, null, null, null,
      null, null, null, null, null, null, null, null, null, null, null,
      null, null, null, null, null, null, null, null, null, null, null,
      null, null, null, null, null, null, null, Piece.PAWN, Piece.PAWN,
      Piece.PAWN, Piece.PAWN, Piece.PAWN, Piece.PAWN, Piece.PAWN,
      Piece.PAWN, Piece.ROOK, Piece.KNIGHT, Piece.BISHOP, Piece.QUEEN,
      Piece.KING, Piece.BISHOP, Piece.KNIGHT, Piece.ROOK, null };

  /**
   * this array maps the integer value of an square to the array index of
   * array representation of the board in this class
   */
  protected static int[] square_to_array_index = { 64, 64, 64, 64, 64, 64,
      64, 64, 64, 64, 64, 56, 48, 40, 32, 24, 16, 8, 0, 64, 64, 57, 49,
      41, 33, 25, 17, 9, 1, 64, 64, 58, 50, 42, 34, 26, 18, 10, 2, 64,
      64, 59, 51, 43, 35, 27, 19, 11, 3, 64, 64, 60, 52, 44, 36, 28, 20,
      12, 4, 64, 64, 61, 53, 45, 37, 29, 21, 13, 5, 64, 64, 62, 54, 46,
      38, 30, 22, 14, 6, 64, 64, 63, 55, 47, 39, 31, 23, 15, 7, 64, 64,
      64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
      64, 64, 64 };

  /**
   * the array of Sides, containing the information about the position of the
   * sides of the pieces
```

```
 */
private Side[] side_board = new Side[65];

/**
 * the array of Pieces, containing the information about the position of the
 * pieces
 */
private Piece[] piece_board = new Piece[65];

/**
 * squares c1, g1, c8 and g8 in ICCF numeric notation. do not change the
 * squares' order or bad things will happen! set to -1 if castling not
 * allowed.
 */
private int[] castling = { -1, -1, -1, -1 };

/**
 * the square of the en_passant_target, -1 if none.
 */
private int en_passant_target = -1;

/**
 * the side, which has to move
 */
private Side active_color;

/**
 * contains the information about the value of the position.
 */
private AnalysisResult analysis_result = null;

// The following class members are used to prevent multiple computations
/**
 * caching of the possible moves
 */
private SoftReference<List<IMove>> possible_moves;

/**
 * caching if the current position is check.
 */
private Boolean is_check;

/**
 * caching if the current position is mate.
 */
private Boolean is_mate;
```

```
/**
 * caching if the current position is stalemate.
 */
private Boolean is_stale_mate;

// the following maps takes and Integer, representing the color, type or
// PieceValue and returns the set of squares or the number of squares!
/**
 * this map maps the PieceValue, i.e. 10*side.ordinal + piece.ordinal, to
 * the set of squares where the pieces of the side are positioned.
 */
private Map<Integer, Set<Integer>> occupied_squares_by_color_and_type = new
    HashMap<Integer, Set<Integer>>();

/**
 * this map maps the side, i.e. side.ordinal, to the set of squares where
 * the side has pieces.
 */
private Map<Side, Set<Integer>> occupied_squares_by_color = new
    HashMap<Side, Set<Integer>>();

/**
 * this map maps the piece, i.e. piece.ordinal, to the set of squares where
 * the pieces are positioned.
 */
private Map<Piece, Set<Integer>> occupied_squares_by_type = new
    HashMap<Piece, Set<Integer>>();

/**
 * caching the number of occupied squares for each side of an piece in an
 * small array.
 */
private byte[] num_occupied_squares_by_color_and_type = new byte[16];

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

/**
 * Resets and clears the stored class members.
 */
private void resetCache() {
  possible_moves = null;
  is_check = null;
  is_mate = null;
  is_stale_mate = null;
```

```
    analysis_result = null;
    occupied_squares_by_color_and_type.clear();
    occupied_squares_by_type.clear();
    occupied_squares_by_color.clear();
  }

  /**
   * computes the index for the internal array representation of an square
   *
   * @param square
   *           the given square
   * @return the index
   */
  private int squareToArrayIndex(int square) {
    if (square < 0)
      return 64;
    return square_to_array_index[square];
  }

  /**
   * computes a copy of the actual board, only the necessary informations are
   * copied, plus <code>num_occupied_squares_by_color_and_type</code>
   *
   * @return a incomplete copy of the board.
   */
  private Position returnCopy() {
    Position newBoard = new Position();

    newBoard.active_color = active_color;
    newBoard.en_passant_target = en_passant_target;
    System.arraycopy(castling, 0, newBoard.castling, 0, 4);

    System.arraycopy(side_board, 0, newBoard.side_board, 0, 65);
    System.arraycopy(piece_board, 0, newBoard.piece_board, 0, 65);

    System.arraycopy(num_occupied_squares_by_color_and_type, 0,
        newBoard.num_occupied_squares_by_color_and_type, 0, 16);

    return newBoard;
  }

  /**
   * returns the Side, which occupies a given square
   *
   * @return the side of the piece which is on the square
   */
```

```
public Side getSideFromBoard(int square) {
  int i = squareToArrayIndex(square);
  return side_board[i];
}

/**
 * returns the piece, which occupies a given square
 *
 * @return the piece which is on the square
 */
public Piece getPieceFromBoard(int square) {
  int i = squareToArrayIndex(square);
  return piece_board[i];
}

/**
 * sets a piece on the board.
 *
 * @param square
 *          the square, were the piece should be set
 * @param side
 *          the given side
 * @param piece
 *          the given piece
 */
private void setOnBoard(int square, Side side, Piece piece) {
  int i = squareToArrayIndex(square);
  side_board[i] = side;
  piece_board[i] = piece;
}

/**
 * returns the opponents side of the actual board
 *
 * @return the side of the opponent
 */
public Side getOpponentsColor() {
  if (active_color == Side.BLACK)
    return Side.WHITE;
  else
    return Side.BLACK;
}

/**
 * returns the eventual result of the position evaluation
 */
```

```
public AnalysisResult getAnalysisResult() {
  return analysis_result;
}

/**
 * updates the result of the board. (only if it more valuable, i.e.
 * comparison of the depth)
 *
 * @param analysis_result
 *           the new analysis result
 */
public void updateAnalysisResult(AnalysisResult analysis_result) {
  if (analysis_result == null)
    throw new NullPointerException();

  if (this.analysis_result == null ||
      this.analysis_result.compareQualityTo(analysis_result) <= 0) {
    this.analysis_result = analysis_result;
  }
}

/**
 * checks is a move is a hit. there is no check, that the move is legal!.
 *
 * @param move
 *           the move to be checked
 * @return true, if it is a hit, false otherwise
 */
public boolean isHit(IMove move) {
  int dest = move.getToSquare();
  int src = move.getFromSquare();

  // a hit happens iff the dest is an enemy or its en passant
  if (getSideFromBoard(dest) == Side.getOppositeSide(active_color)
      || (getPieceFromBoard(src) == Piece.PAWN && dest == this
          .getEnPassant()))
    return true;
  return false;
}

@Override
public void setToInitial() {
  System.arraycopy(initial_side_board, 0, side_board, 0, 65);
  System.arraycopy(initial_piece_board, 0, piece_board, 0, 65);

  castling[0] = 31;
```

```
    castling[1] = 71;
    castling[2] = 38;
    castling[3] = 78;

    en_passant_target = -1;
    active_color = Side.WHITE;

    num_occupied_squares_by_color_and_type[Side.WHITE.ordinal() * 10
        + Piece.KING.ordinal()] = 1;
    num_occupied_squares_by_color_and_type[Side.WHITE.ordinal() * 10
        + Piece.QUEEN.ordinal()] = 1;
    num_occupied_squares_by_color_and_type[Side.WHITE.ordinal() * 10
        + Piece.ROOK.ordinal()] = 2;
    num_occupied_squares_by_color_and_type[Side.WHITE.ordinal() * 10
        + Piece.BISHOP.ordinal()] = 2;
    num_occupied_squares_by_color_and_type[Side.WHITE.ordinal() * 10
        + Piece.KNIGHT.ordinal()] = 2;
    num_occupied_squares_by_color_and_type[Side.WHITE.ordinal() * 10
        + Piece.PAWN.ordinal()] = 8;
    num_occupied_squares_by_color_and_type[Side.BLACK.ordinal() * 10
        + Piece.KING.ordinal()] = 1;
    num_occupied_squares_by_color_and_type[Side.BLACK.ordinal() * 10
        + Piece.QUEEN.ordinal()] = 1;
    num_occupied_squares_by_color_and_type[Side.BLACK.ordinal() * 10
        + Piece.ROOK.ordinal()] = 2;
    num_occupied_squares_by_color_and_type[Side.BLACK.ordinal() * 10
        + Piece.BISHOP.ordinal()] = 2;
    num_occupied_squares_by_color_and_type[Side.BLACK.ordinal() * 10
        + Piece.KNIGHT.ordinal()] = 2;
    num_occupied_squares_by_color_and_type[Side.BLACK.ordinal() * 10
        + Piece.PAWN.ordinal()] = 8;

    resetCache();
  }

  @Override
  public void setToFEN(String fen) {
    side_board = new Side[65];
    piece_board = new Piece[65];

    castling[0] = -1;
    castling[1] = -1;
    castling[2] = -1;
    castling[3] = -1;
    en_passant_target = -1;
```

```
resetCache();

String[] fen_parts = fen.split(" ");

// populate the squares
String[] fen_rows = fen_parts[0].split("/");
char[] pieces;
for (int row = 1; row <= 8; row++) {
  int offset = 0;
  for (int column = 1; column + offset <= 8; column++) {
    pieces = fen_rows[8 - row].toCharArray();
    int square = (column + offset) * 10 + row;
    switch (pieces[column - 1]) {
    case 'P':
      setOnBoard(square, Side.WHITE, Piece.PAWN);
      num_occupied_squares_by_color_and_type[Side.WHITE.ordinal()
          * 10 + Piece.PAWN.ordinal()]++;
      break;
    case 'R':
      setOnBoard(square, Side.WHITE, Piece.ROOK);
      num_occupied_squares_by_color_and_type[Side.WHITE.ordinal()
          * 10 + Piece.ROOK.ordinal()]++;
      break;
    case 'N':
      setOnBoard(square, Side.WHITE, Piece.KNIGHT);
      num_occupied_squares_by_color_and_type[Side.WHITE.ordinal()
          * 10 + Piece.KNIGHT.ordinal()]++;
      break;
    case 'B':
      setOnBoard(square, Side.WHITE, Piece.BISHOP);
      num_occupied_squares_by_color_and_type[Side.WHITE.ordinal()
          * 10 + Piece.BISHOP.ordinal()]++;
      break;
    case 'Q':
      setOnBoard(square, Side.WHITE, Piece.QUEEN);
      num_occupied_squares_by_color_and_type[Side.WHITE.ordinal()
          * 10 + Piece.QUEEN.ordinal()]++;
      break;
    case 'K':
      setOnBoard(square, Side.WHITE, Piece.KING);
      num_occupied_squares_by_color_and_type[Side.WHITE.ordinal()
          * 10 + Piece.KING.ordinal()]++;
      break;
    case 'p':
      setOnBoard(square, Side.BLACK, Piece.PAWN);
      num_occupied_squares_by_color_and_type[Side.BLACK.ordinal()
```

```
            * 10 + Piece.PAWN.ordinal()]++;
          break;
        case 'r':
          setOnBoard(square, Side.BLACK, Piece.ROOK);
          num_occupied_squares_by_color_and_type[Side.BLACK.ordinal()
              * 10 + Piece.ROOK.ordinal()]++;
          break;
        case 'n':
          setOnBoard(square, Side.BLACK, Piece.KNIGHT);
          num_occupied_squares_by_color_and_type[Side.BLACK.ordinal()
              * 10 + Piece.KNIGHT.ordinal()]++;
          break;
        case 'b':
          setOnBoard(square, Side.BLACK, Piece.BISHOP);
          num_occupied_squares_by_color_and_type[Side.BLACK.ordinal()
              * 10 + Piece.BISHOP.ordinal()]++;
          break;
        case 'q':
          setOnBoard(square, Side.BLACK, Piece.QUEEN);
          num_occupied_squares_by_color_and_type[Side.BLACK.ordinal()
              * 10 + Piece.QUEEN.ordinal()]++;
          break;
        case 'k':
          setOnBoard(square, Side.BLACK, Piece.KING);
          num_occupied_squares_by_color_and_type[Side.BLACK.ordinal()
              * 10 + Piece.KING.ordinal()]++;
          break;
        default:
          offset += Character.getNumericValue(pieces[column - 1]) - 1;
          break;
        }
      }
    }

    // set active color
    switch (fen_parts[1]) {
    case "b":
      active_color = Side.BLACK;
      break;
    case "w":
      active_color = Side.WHITE;
      break;
    }

    // set possible castling moves
    if (!fen_parts[2].equals("-")) {
```

```
    char[] castlings = fen_parts[2].toCharArray();
    for (int i = 0; i < castlings.length; i++) {
      switch (castlings[i]) {
      case 'K':
        castling[1] = 71;
        break;
      case 'Q':
        castling[0] = 31;
        break;
      case 'k':
        castling[3] = 78;
        break;
      case 'q':
        castling[2] = 38;
        break;
      }
    }
  }

  // set en passant square
  if (!fen_parts[3].equals("-")) {
    en_passant_target = SquareHelper.fromString(fen_parts[3]);
  }
}

@Override
public MoveApplication doMove(IMove move) {
  MoveApplication mova = new MoveApplication();
  Position newBoard = this.returnCopy();

  int src = move.getFromSquare();
  int dest = move.getToSquare();

  Piece piece = getPieceFromBoard(src);
  Piece capture = getPieceFromBoard(dest);

  // if promotion
  if (move.getPromotion() != null) {
    newBoard.setOnBoard(src, null, null);
    newBoard.setOnBoard(dest, active_color, move.getPromotion());
    mova.resets_half_move_clock = true;
    newBoard.num_occupied_squares_by_color_and_type[active_color
        .ordinal() * 10 + Piece.PAWN.ordinal()]--;
    newBoard.num_occupied_squares_by_color_and_type[active_color
        .ordinal() * 10 + move.getPromotion().ordinal()]++;
  }
```

```
// If castling
else if (piece == Piece.KING && Math.abs((src - dest)) == 20) {
  newBoard.setOnBoard(dest, active_color, Piece.KING);
  newBoard.setOnBoard(src, null, null);
  newBoard.setOnBoard((src + dest) / 2, active_color, Piece.ROOK);
  if (SquareHelper.getColumn(dest) == 3)
    newBoard.setOnBoard(src - 40, null, null);
  else
    newBoard.setOnBoard(src + 30, null, null);

}
// If en passant
else if (piece == Piece.PAWN && dest == this.getEnPassant()) {
  newBoard.setOnBoard(dest, active_color, Piece.PAWN);
  newBoard.setOnBoard(src, null, null);
  if (active_color == Side.WHITE) {
    capture = getPieceFromBoard(dest - 1);
    newBoard.setOnBoard(dest - 1, null, null);
  } else {
    capture = getPieceFromBoard(dest + 1);
    newBoard.setOnBoard(dest + 1, null, null);
  }
  mova.resets_half_move_clock = true;
}
// Usual move
else {
  Side side = getSideFromBoard(src);
  newBoard.setOnBoard(dest, side, piece);
  newBoard.setOnBoard(src, null, null);
  if (this.getSideFromBoard(dest) != null || piece == Piece.PAWN)
    mova.resets_half_move_clock = true;
}

// update counters
if (capture != null) {
  newBoard.num_occupied_squares_by_color_and_type[Side
      .getOppositeSide(active_color).ordinal()
      * 10
      + capture.ordinal()]--;
}

// Change active_color after move
newBoard.active_color = Side.getOppositeSide(active_color);
if (active_color == Side.BLACK)

  // Update en_passant
```

```
    if (piece == Piece.PAWN && Math.abs(dest - src) == 2)
      newBoard.en_passant_target = (dest + src) / 2;
    else
      newBoard.en_passant_target = -1;

  // Update castling
  if (piece == Piece.KING) {
    if (active_color == Side.WHITE && src == 51) {
      newBoard.castling[0] = -1;
      newBoard.castling[1] = -1;
    } else if (active_color == Side.BLACK && src == 58) {
      newBoard.castling[2] = -1;
      newBoard.castling[3] = -1;
    }
  } else if (piece == Piece.ROOK) {
    if (active_color == Side.WHITE) {
      if (src == 81)
        newBoard.castling[1] = -1;
      else if (src == 11)
        newBoard.castling[0] = -1;
    } else {
      if (src == 88)
        newBoard.castling[3] = -1;
      else if (src == 18)
        newBoard.castling[2] = -1;
    }
  }

  mova.new_position = newBoard;

  return mova;
}

@Override
public int getEnPassant() {
  return en_passant_target;
}

@Override
public boolean canCastle(int king_to) {
  if ((king_to == 31 && castling[0] != -1)
      || (king_to == 71 && castling[1] != -1)
      || (king_to == 38 && castling[2] != -1)
      || (king_to == 78 && castling[3] != -1)) {
    return true;
  } else {
```

```
    return false;
  }
}

@Override
public Boolean colorCanCastle(Side color) {

  // Set the right color
  if (active_color != color)
    ;
  active_color = getOpponentsColor();

  // check for castling
  if (!isCheckPosition()) {
    Move move;
    int off = 0;
    int square = 51;

    if (color == Side.BLACK) {
      off = 2;
      square = 58;
    }

    for (int i = 0; i < 2; i++) {
      int castle_flag = 0;
      Integer new_square = castling[i + off];
      // castling must still be possible to this side
      if (new_square != -1) {

        Direction dir;
        if (i == 0)
          dir = Direction.WEST;
        else
          dir = Direction.EAST;

        List<Integer> line = SquareHelper.getAllSquaresInDirection(
            square, dir);

        // Check each square if it is empty
        for (Integer squ : line) {
          if (getSideFromBoard(squ) != null) {
            castle_flag = 1;
            break;
          }
          if (squ == new_square)
            break;
```

```
        }
        if (castle_flag == 1)
          continue;

        // Check each square if the king on it would be check
        for (Integer squ : line) {
          move = new Move(square, squ);
          Position board = (Position) doMove(move).new_position;
          board.active_color = active_color;

          if (board.isCheckPosition())
            break;
          if (squ == new_square) {
            // If the end is reached, then stop checking.

            // undoing change of color
            if (active_color == color)
              active_color = getOpponentsColor();

            return true;
          }
        }
      }
    }
  }

  // undoing change of color
  if (active_color == color)
    active_color = getOpponentsColor();

  return false;
}

@Override
public Set<Integer> getOccupiedSquaresByColor(Side color) {

  if (occupied_squares_by_color.containsKey(color) == false) {
    int square;
    Set<Integer> set = new HashSet<Integer>();

    for (int i = 1; i < 9; i++)
      for (int j = 1; j < 9; j++) {
        square = SquareHelper.getSquare(i, j);
        if (getSideFromBoard(square) == color)
          set.add(square);
```

```
      }

    occupied_squares_by_color.put(color, set);
    return set;
  }
  return occupied_squares_by_color.get(color);
}

@Override
public Set<Integer> getOccupiedSquaresByType(Piece type) {

  if (occupied_squares_by_type.containsKey(type) == false) {
    int square;
    Set<Integer> set = new HashSet<Integer>();

    for (int i = 1; i < 9; i++)
      for (int j = 1; j < 9; j++) {
        square = SquareHelper.getSquare(i, j);
        if (getPieceFromBoard(square) == type)
          set.add(square);
      }

    occupied_squares_by_type.put(type, set);
    return set;
  }
  return occupied_squares_by_type.get(type);

}

@Override
public Set<Integer> getOccupiedSquaresByColorAndType(Side color, Piece
    type) {

  int value = color.ordinal() * 10 + type.ordinal();

  if (occupied_squares_by_color_and_type.containsKey(value) == false) {
    int square;
    Set<Integer> set = new HashSet<Integer>();

    for (int i = 1; i < 9; i++)
      for (int j = 1; j < 9; j++) {
        square = SquareHelper.getSquare(i, j);
        if (color == getSideFromBoard(square)
            && type == getPieceFromBoard(square))
          set.add(square);
      }
```

```
    occupied_squares_by_color_and_type.put(value, set);
    return set;
  }
  return occupied_squares_by_color_and_type.get(value);
}

@Override
public int getNumberOfPiecesByColor(Side side) {
  int result = 0;
  for (Piece piece : Piece.values()) {
    result += num_occupied_squares_by_color_and_type[side.ordinal()
        * 10 + piece.ordinal()];
  }
  return result;
}

@Override
public int getNumberOfPiecesByType(Piece piece) {
  int result = 0;
  for (Side side : Side.values()) {
    result += num_occupied_squares_by_color_and_type[side.ordinal()
        * 10 + piece.ordinal()];
  }
  return result;
}

@Override
public int getNumberOfPiecesByColorAndType(Side color, Piece type) {
  int value = color.ordinal() * 10 + type.ordinal();
  return num_occupied_squares_by_color_and_type[value];
}

@Override
public List<IMove> getPossibleMoves() {
  if (possible_moves == null) {
    List<IMove> total_list = new ArrayList<IMove>(40);

    // loop over all squares
    for (int square : getOccupiedSquaresByColor(active_color)) {
      total_list.addAll(getPossibleMovesFrom(square));
    }

    // cache it
    possible_moves = new SoftReference<List<IMove>>(total_list);
    return total_list;
  } else {
```

```
    // return from cache
    return possible_moves.get();
  }
}

@Override
public List<IMove> getPossibleMovesFrom(int square) {
  // The case, that the destination is the opponents king cannot happen.

  Piece type = getPieceFromBoard(square);
  Side opp_color = getOpponentsColor();
  List<Integer> squares;
  List<IMove> moves = new ArrayList<IMove>();
  Move move;

  // Types BISHOP, QUEEN, ROOK
  if (type == Piece.BISHOP || type == Piece.QUEEN || type == Piece.ROOK) {

    // Loop over all directions and skip not appropriate ones
    for (Direction direction : Direction.values()) {

      // Skip N,W,E,W with BISHOP and skip NE,NW,SE,SW with ROOK
      if (((direction == Direction.NORTH
          || direction == Direction.EAST
          || direction == Direction.SOUTH || direction == Direction.WEST) &&
             type == Piece.BISHOP)
          || ((direction == Direction.NORTHWEST
              || direction == Direction.NORTHEAST
              || direction == Direction.SOUTHEAST || direction ==
                 Direction.SOUTHWEST) && type == Piece.ROOK)) {

        continue;
      } else {
        // do stuff
        squares = SquareHelper.getAllSquaresInDirection(square,
            direction);

        for (Integer new_square : squares) {
          Piece piece = getPieceFromBoard(new_square);
          Side color = getSideFromBoard(new_square);
          if (piece == null || color == opp_color) {

            move = new Move(square, new_square);
            moves.add(move);
            if (piece != null && color == opp_color)
              // not possible to go further
```

```
            break;
          } else
            break;
        }
      }

    }

  }

  if (type == Piece.PAWN) {
    // If Pawn has not moved yet (steps possible)
    if ((SquareHelper.getRow(square) == 2 && active_color == Side.WHITE)
        || (SquareHelper.getRow(square) == 7 && active_color ==
           Side.BLACK)) {

      if (getSideFromBoard(square
          + Direction.pawnDirection(active_color).offset) == null) {
        move = new Move(square, square
            + Direction.pawnDirection(active_color).offset);
        moves.add(move);
        if (getSideFromBoard(square + 2
            * Direction.pawnDirection(active_color).offset) == null) {
          move = new Move(square, square + 2
              * Direction.pawnDirection(active_color).offset);
          moves.add(move);
        }
      }

      Set<Direction> pawn_capturing_directions = Direction
          .pawnCapturingDirections(active_color);
      for (Direction direction : pawn_capturing_directions) {
        if (getSideFromBoard(square + direction.offset) ==
            getOpponentsColor()) {
          move = new Move(square, square + direction.offset);
          moves.add(move);
        }
      }

    }
    // if Promotion will happen
    else if ((SquareHelper.getRow(square) == 7 && active_color ==
        Side.WHITE)
        || (SquareHelper.getRow(square) == 2 && active_color ==
           Side.BLACK)) {
      if (getSideFromBoard(square
```

```
        + Direction.pawnDirection(active_color).offset) == null) {
      move = new Move(square, square
          + Direction.pawnDirection(active_color).offset,
          Piece.QUEEN);
      moves.add(move);
      move = new Move(square, square
          + Direction.pawnDirection(active_color).offset,
          Piece.KNIGHT);
      moves.add(move);
      /*
       * A Queen is always better then a rook or a bishop move =
       * new Move(square, square +
       * Direction.pawnDirection(active_color).offset,
       * Piece.ROOK); moves.add(move); move = new Move(square,
       * square + Direction.pawnDirection(active_color).offset,
       * Piece.BISHOP); moves.add(move);
       */
    }
    Set<Direction> pawn_capturing_directions = Direction
        .pawnCapturingDirections(active_color);
    for (Direction direction : pawn_capturing_directions) {
      if (getSideFromBoard(square + direction.offset) ==
          getOpponentsColor()) {
        move = new Move(square, square + direction.offset,
            Piece.QUEEN);
        moves.add(move);
        move = new Move(square, square + direction.offset,
            Piece.KNIGHT);
        moves.add(move);
      }
    }

  }
  // Usual turn and en passant is possible, no promotion
  else {
    if (getSideFromBoard(square
        + Direction.pawnDirection(active_color).offset) == null) {
      move = new Move(square, square
          + Direction.pawnDirection(active_color).offset);
      moves.add(move);
    }
    Set<Direction> pawn_capturing_directions = Direction
        .pawnCapturingDirections(active_color);
    for (Direction direction : pawn_capturing_directions) {
      if ((getSideFromBoard(square + direction.offset) ==
          getOpponentsColor())
```

```
          || square + direction.offset == getEnPassant()) {
        move = new Move(square, square + direction.offset);
        moves.add(move);
      }
    }
  }

}
if (type == Piece.KING) {
  for (Direction direction : Direction.values()) {
    Integer new_square = square + direction.offset;

    if (SquareHelper.isValidSquare(new_square)) {
      move = new Move(square, new_square);
      Side side = getSideFromBoard(new_square);
      // if the new square is empty or occupied by the opponent
      if (side != active_color)
        moves.add(move);
    }
  }

  // Castle Moves
  // If the King is not check now, try castle moves
  if (!isCheckPosition()) {
    int off = 0;
    if (active_color == Side.BLACK)
      off = 2;

    for (int i = 0; i < 2; i++) {
      int castle_flag = 0;
      Integer new_square = castling[i + off];
      // castling must still be possible to this side
      if (new_square != -1) {

        Direction dir;
        if (i == 0)
          dir = Direction.WEST;
        else
          dir = Direction.EAST;

        List<Integer> line = SquareHelper
            .getAllSquaresInDirection(square, dir);

        // Check each square if it is empty
        for (Integer squ : line) {
          if (getSideFromBoard(squ) != null) {
```

```
            castle_flag = 1;
            break;
          }
          if (squ == new_square)
            break;
        }
        if (castle_flag == 1)
          continue;

        // Check each square if the king on it would be check
        for (Integer squ : line) {
          move = new Move(square, squ);
          Position board = (Position) doMove(move).new_position;
          board.active_color = active_color;
          if (board.isCheckPosition())
            break;
          if (squ == new_square) {
            // if everything is right, then add the move
            moves.add(move);
            break;
          }
        }
      }
    }
  }
}
if (type == Piece.KNIGHT) {
  squares = SquareHelper.getAllSquaresByKnightStep(square);
  for (Integer new_square : squares) {
    Side side = getSideFromBoard(new_square);
    if (side != active_color) {
      move = new Move(square, new_square);
      moves.add(move);
    }
  }
}

// remove invalid positions
// TODO do this in a more efficient way
Iterator<IMove> iter = moves.iterator();
while (iter.hasNext()) {
  Position temp_board = (Position) this.doMove(iter.next()).new_position;
  temp_board.active_color = active_color;
  if (temp_board.isCheckPosition()) {
    iter.remove();
```

```
    }
  }

  return moves;
}

@Override
public List<IMove> getPossibleMovesTo(int square) {
  List<IMove> possible_moves = getPossibleMoves();
  List<IMove> result = new ArrayList<IMove>(possible_moves.size());

  for (IMove move : possible_moves) {
    if (move.getToSquare() == square)
      result.add(move);
  }

  return result;
}

@Override
public boolean isCheckPosition() {
  if (is_check == null) {
    is_check = true;
    Set<Integer> temp_king_pos = getOccupiedSquaresByColorAndType(
        active_color, Piece.KING);
    int king_pos = temp_king_pos.iterator().next();

    // go in each direction
    for (Direction direction : Direction.values()) {
      List<Integer> line = SquareHelper.getAllSquaresInDirection(
          king_pos, direction);
      // go until
      int iter = 0;
      for (int square : line) {
        iter++;
        // some piece is found
        Piece piece = getPieceFromBoard(square);
        if (piece != null) {
          Side side = getSideFromBoard(square);
          if (side == active_color) {
            break;
          } else {
            if (piece == Piece.PAWN && iter == 1) {
              if (((direction == Direction.NORTHEAST || direction ==
                  Direction.NORTHWEST) && active_color == Side.WHITE)
```

```
                || ((direction == Direction.SOUTHEAST || direction ==
                    Direction.SOUTHWEST) && active_color == Side.BLACK)) {
              return true;
            }
          } else if (piece == Piece.ROOK) {
            if (direction == Direction.EAST
                || direction == Direction.WEST
                || direction == Direction.NORTH
                || direction == Direction.SOUTH) {
              return true;
            }
          } else if (piece == Piece.BISHOP) {
            if (direction == Direction.NORTHEAST
                || direction == Direction.NORTHWEST
                || direction == Direction.SOUTHEAST
                || direction == Direction.SOUTHWEST) {
              return true;
            }
          } else if (piece == Piece.QUEEN) {
            return true;
          } else if (piece == Piece.KING && iter == 1) {
            return true;
          }
          break;
        }
      }
    }
  }

  // check for knight attacks
  List<Integer> knight_squares = SquareHelper
      .getAllSquaresByKnightStep(king_pos);
  for (int square : knight_squares) {
    Piece piece = getPieceFromBoard(square);
    if (piece != null) {
      Side side = getSideFromBoard(square);
      if (side != active_color && piece == Piece.KNIGHT) {
        return true;
      }
    }
  }
  is_check = false;
 }
 return is_check.booleanValue();

}
```

```
@Override
public boolean isMatePosition() {
  if (is_mate == null) {
    is_mate = true;
    List<IMove> moves = getPossibleMoves();
    if (moves.isEmpty() && isCheckPosition())
      return true;
    is_mate = false;
  }
  return is_mate.booleanValue();
}

@Override
public boolean isStaleMatePosition() {
  if (is_stale_mate == null) {
    is_stale_mate = true;
    List<IMove> moves = getPossibleMoves();
    if (moves.isEmpty())
      return true;
    is_stale_mate = false;
  }
  return is_stale_mate.booleanValue();
}

@Override
public boolean isPossibleMove(IMove move) {

  List<IMove> possible_moves = getPossibleMoves();

  return possible_moves.contains(move);
}

public String toString() {
  return toFEN();
}

@Override
public String toFEN() {
  StringBuilder fen = new StringBuilder();

  // piece placement
  for (int row = 0; row < 8; row++) {

    int counter = 0;
```

```
    for (int column = 0; column < 8; column++) {

      if (side_board[row * 8 + column] == null) {
        counter++;
      } else {
        if (counter != 0) {
          fen.append(counter);
          counter = 0;
        }
        fen.append(PieceHelper.toString(
            side_board[row * 8 + column], piece_board[row * 8
                + column]));
      }
      if (column == 7 && counter != 0) {
        fen.append(counter);
      }
    }

    if (row != 7) {
      fen.append("/");
    }
  }
  fen.append(" ");

  // active color
  if (active_color == Side.WHITE) {
    fen.append("w");
  } else {
    fen.append("b");
  }
  fen.append(" ");

  // castling availability
  boolean castle_flag = false;
  if (castling[1] != -1) {
    fen.append("K");
    castle_flag = true;
  }
  if (castling[0] != -1) {
    fen.append("Q");
    castle_flag = true;
  }
  if (castling[3] != -1) {
    fen.append("k");
    castle_flag = true;
  }
```

```
    if (castling[2] != -1) {
      fen.append("q");
      castle_flag = true;
    }
    if (!castle_flag) {
      fen.append("-");
    }
    fen.append(" ");

    // en passant target square
    if (en_passant_target == -1) {
      fen.append("-");
    } else {
      fen.append(SquareHelper.toString(en_passant_target));
    }

    return fen.toString();
  }

  @Override
  public Side getActiveColor() {
    return active_color;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;

    for (Side element : side_board)
      result = prime * result
          + (element == null ? 0 : element.ordinal() + 1);

    for (Piece element : piece_board)
      result = prime * result
          + (element == null ? 0 : element.ordinal() + 1);

    for (int element : castling)
      result = prime * result + element;

    result = prime * result + active_color.ordinal();

    result = prime * result + en_passant_target;

    return result;
  }
```

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }
  if (getClass() != obj.getClass()) {
    return false;
  }
  Position other = (Position) obj;
  if (!Arrays.equals(side_board, other.side_board)
      || !Arrays.equals(piece_board, other.piece_board)
      || !Arrays.equals(castling, other.castling)
      || en_passant_target != other.en_passant_target
      || active_color != other.active_color) {
    return false;
  }
  return true;
}

@Override
public List<IMove> generateCaptures() {
  List<IMove> poss_moves = getPossibleMoves();
  List<IMove> result = new ArrayList<IMove>(poss_moves.size());

  for (IMove move : poss_moves)
    if (isHit(move) || move.getPromotion() != null)
      result.add(move);
  return result;
}

@Override
public long hashCode2() {
  final int prime = 23;
  long result = 1;

  for (Side element : side_board)
    result = prime * result
        + (element == null ? 0 : element.ordinal() + 1);

  for (Piece element : piece_board)
    result = prime * result
        + (element == null ? 0 : element.ordinal() + 1);
```

```
    for (int element : castling)
      result = prime * result + element;

    result = prime * result + active_color.ordinal();

    result = prime * result + en_passant_target;

    return result;
  }
}
```

SquareHelper.java

```
package mitzi;

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

/**
 * In brief, each square of the chessboard has a two-digit designation. The
 * first digit is the number of the column, from left to right from White's
 * point of view. The second digit is the row from the edge near White to the
 * other edge.
 *
 * @see <a href="https://en.wikipedia.org/wiki/ICCF_numeric_notation">ICCF
 *      numeric notation</a>
 */
public final class SquareHelper {

  /**
   * the letters of the columns of the chessboard
   */
  private static final String[] letters = { "a", "b", "c", "d", "e", "f",
      "g", "h" };

  private SquareHelper() {
  };

  /**
   * Returns the integer value of the square's column. Starting with 1 at
   * column a and ending with 8 at column h.
   *
   * @return the integer value of the square's column.
```

```
 */
public static int getColumn(int square) {
  return square / 10;
}

/**
 * Returns the integer value of the square's row. Where row 1 is row 1 and
 * so forth, obviously.
 *
 * @return the integer value of the square's row.
 */
public static int getRow(int square) {
  return square % 10;
}

/**
 * Returns the square-number for a given row and column. Row 1 and column 2
 * results in 12.
 *
 * @return the integer value of the square
 */
public static int getSquare(int row, int column) {
  return 10 * column + row;
}

/**
 * Check if the square is white on a traditional chess board.
 *
 * @param square
 *           the integer code of the square
 *
 * @return true if the square is white and false otherwise
 */
public static boolean isWhite(int square) {
  return (square / 10 + square % 10) % 2 != 0;
}

/**
 * Check if the square is black on a traditional chess board.
 *
 * @param square
 *           the integer code of the square
 *
 * @return true if the square is black and false otherwise
 */
public static boolean isBlack(int square) {
```

```
  return !isWhite(square);
}

/**
 * Gives an ordered List of squares going in a straight line from the source
 * square.
 *
 * @param source_square
 *           the square from where to start
 * @param direction
 *           one of the values SquareHelper.EAST, SquareHelper.NORTHEAST,
 *           SquareHelper.NORTH,
 * @return the list of squares ordered from the source_square to the boards
 *         edge
 */
public static List<Integer> getAllSquaresInDirection(int source_square,
    Direction direction) {

  ArrayList<Integer> square_list = new ArrayList<Integer>();

  int square = source_square += direction.offset;
  while (isValidSquare(square)) {
    square_list.add(square);
    square += direction.offset;
  }

  return square_list;
}

/**
 * Gives a List of squares reached by a knight from the source square (in no
 * specific order).
 *
 * @param source_square
 *           the square from where to start
 * @return the list of squares a knight can reach
 */
public static List<Integer> getAllSquaresByKnightStep(int source_square) {

  ArrayList<Integer> square_list = new ArrayList<Integer>();

  for (Direction direction : Direction.values()) {
    int square = source_square + direction.knight_offset;
    if (isValidSquare(square)) {
      square_list.add(square);
    }
```

```
    }

    return square_list;
  }

  /**
   * Checks if the integer value of the square is inside the board's borders.
   *
   * @param square
   *           the square to be checked
   * @return true if the square is on the board
   */
  public static boolean isValidSquare(int square) {
    int row = getRow(square);
    int column = getColumn(square);
    return (row >= 1 && row <= 8 && column >= 1 && column <= 8);
  }

  /**
   * Returns a string representation of the square in algebraic notation.
   *
   * Each square is traditionally identified by a unique coordinate pair
   * consisting of a letter and a number. The vertical columns from White's
   * left (the queenside) to his right (the kingside) are labeled a through h.
   * The horizontal rows are numbered 1 to 8 starting from White's side of the
   * board. Thus, each square has a unique identification of a letter followed
   * by a number. For example, the white king starts the game on square e1,
   * while the black knight on b8 can move to open squares a6 or c6.
   *
   * @return a string representation of the square in algebraic notation.
   */
  public static String toString(int square) {
    return letters[getColumn(square) - 1]
        + Integer.toString(getRow(square));
  }

  /**
   * converts the string representation of a square into a the ICCF notation.
   * @param notation the given square in string notation
   * @return the square in integer representation.
   */
  public static int fromString(String notation) {
    int i = 0;
    while (letters[i].charAt(0) != notation.charAt(0)) {
      i++;
    }
```

```
    return (i + 1) * 10 + Character.getNumericValue(notation.charAt(1));
  }
}
```