Mitzi - Exercise 2

Christoph Hofer 0955139

Stefan Lew 0856722

10. Dezember 2013

Inhaltsverzeichnis

1 Implementation of artificial chess players

As requested, we have implemented three different agents for playing. The first type is a random player, who choses randomly among all possible moves for a situation, see `RandyBrain.java` in section 3 on page 23. The second type is an interface for human players, who can enter their moves in long algebraic notation (e. g. d2d4), see `HumanBrain.java` in section 3 on page 24.

1.1 MitziBrain

The heart of our program is our (somewhat) intelligent player `MitziBrain.java` in section 3 on page 103.

The basic evaluation algorithm is a standard implementation of Negamax with Alpha-Beta-Pruning. On top of this we are using techniques like iterative deepening, aspiration windows, transposition tables, different approaches to move ordering, and a quiescence search.

The correct procedure is to create the current `GameState`, e. g. by applying a sequence of moves to the initial game state or by entering the current position as a FEN string. This game state is then handed over to `MitziBrain` and the command `search` starts the evaluation procedure. At the moment, search can be limited by depth and by evaluation time. The method returns one move, it considers to be best.

For debugging reasons, Mitzi keeps printing some status updates, like the number of nodes searched per second (nps), the current search depth, the current principal variation (pv) and the percentage of maximum cache size currently in use.

2 Implementation of a chess game

An example for using our engine is given in `ChessGame.java`, see section 3 on the next page. After starting this example program, you can enter your move and, in this case, our random player will answer. As you will see: you cannot expect high quality games against this opponent.

```
Lets play chess!
e2e4
rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3
Randy plays:e7e6
rnbqkbnr/pppp1ppp/4p3/8/4P3/8/PPPP1PPP/RNBQKBNR w KQkq -
d2d4
rnbqkbnr/pppp1ppp/4p3/8/3PP3/8/PPP2PPP/RNBQKBNR b KQkq d3
Randy plays:a7a6
```

```
rnbqkbnr/1ppp1ppp/p3p3/8/3PP3/8/PPP2PPP/RNBQKBNR w KQkq -
g1f3
rnbqkbnr/1ppp1ppp/p3p3/8/3PP3/5N2/PPP2PPP/RNBQKB1R b KQkq -
Randy plays:d8g5
rnb1kbnr/1ppp1ppp/p3p3/6q1/3PP3/5N2/PPP2PPP/RNBQKB1R w KQkq -
```

The game played was: 1. e4 e6 2. d4 a6 3. Nf3 Qg5??. A more than dubious opening. ;)

2.1 UCI Protocol

For communication with common chess GUIs we are supporting (parts of) an open protocol called UCI. You can find the specification here: http://wbec-ridderkerk.nl/html/UCIProtocol.html.

2.2 ForsythEdwards Notation (FEN)

Representing a game situation is done in FEN: https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation.

3 The Code

ChessGame.java

```java
package mitzi;

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());

    }

  }

}
```

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;
    }
  }
}
```

PieceHelper.java

```
package mitzi;

import java.util.Locale;

public final class PieceHelper {

  /**
   * A String for the algebraic names of the pieces. P... Pawn, R... Rook,
   * etc.
   */
  public static final String[] ALGEBRAIC_NAMES = { "P", "R", "N", "B", "Q",
      "K" };

  private PieceHelper() {
  };

  /**
   * Converts a Piece of a given Side into string. Capital letters are white,
   * lower case letters are black.
   *
   * @param side
   *          the gives side
   * @param piece
   *          the given piece
   * @return the string representation of the piece.
   */
  public static String toString(final Side side, final Piece piece) {
    return toString(side, piece, false);
  }

  /**
   * Converts a Piece of a given Side into string. Capital letters are white,
   * lower case letters are black. Additionally, you have the choice to omit
   * writing a P for pawn.
   *
   * @param side
   *          the gives side
   * @param piece
   *           the given piece
   * @param omitPawnLetter
   *           if the pawnletter should be omitted or not.
   * @return the string representation of the piece.
   */
  public static String toString(final Side side, final Piece piece,
      final boolean omitPawnLetter) {

    if (omitPawnLetter && piece == Piece.PAWN) {
      return "";
    } else if (side == Side.BLACK) {
      return pieceToString(piece).toLowerCase(Locale.ENGLISH);
    } else {
      return pieceToString(piece);
    }
  }

  /**
   * converts a given piece into a string, no distinction which side.
   *
   * @param piece
   *           the given piece
   * @return the string representation.
   */
  private static String pieceToString(final Piece piece) {
    switch (piece) {
    case PAWN:
      return "P";
    case ROOK:
      return "R";
    case KNIGHT:
      return "N";
    case BISHOP:
      return "B";
    case QUEEN:
      return "Q";
    default:
      return "K";
    }

  }

}
```

SquareHelper.java

```java
package mitzi;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;

/**
 * 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" };

  /**
   * stores all squares in a certain direction for a certain source square
   */
  private static ArrayList<ArrayList<List<Integer>>> squares_direction = new
     ArrayList<ArrayList<List<Integer>>>();

  /**
   * stores all squares reachable for the knight from a certain source square
   */
  private static ArrayList<List<Integer>> squares_direction_knight = new
     ArrayList<List<Integer>>();

  public static LinkedList<Integer> all_squares = new LinkedList<Integer>();

  private SquareHelper() {
  };

  static {
    //initialize with null
  for (int i = 0; i < 89; i++) {
    squares_direction.add(null);
    squares_direction_knight.add(null);
  }

  for (int i = 1; i < 9; i++)
    for (int j = 1; j < 9; j++) {
      int source_square = getSquare(i, j);
      all_squares.add(source_square);
      ArrayList<List<Integer>> dir_list = new ArrayList<List<Integer>>();
      for (int k = 0; k < 9; k++)
        dir_list.add(null);
      ArrayList<Integer> dir_list_knight = new ArrayList<Integer>();

      // compute squares for pieces except the knight
      for (Direction dir : Direction.values()) {
        ArrayList<Integer> square_list = new ArrayList<Integer>();

        int square = source_square + dir.offset;
        while (isValidSquare(square)) {
          square_list.add(square);
          square += dir.offset;
        }
        dir_list.set(dir.ordinal(), square_list);
      }

      // squares for Knight
      for (Direction dir : Direction.values()) {

        int square = source_square + dir.knight_offset;
        if (isValidSquare(square)) {
          dir_list_knight.add(square);
        }
      }
      squares_direction.set(source_square, dir_list);
      squares_direction_knight.set(source_square, dir_list_knight);
    }

}

/**
 * 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);
```

```
}

/**
 * returns all squares in all possible directions for a given source square
 *
 * @param square
 *          the source square
 * @return an ArrayList, indexed by the ordinal of the direction, containing
 *        a List of squares in the desired direction
 */
public static ArrayList<List<Integer>> getSquaresAllDirections(int square) {
  return squares_direction.get(square);
}

public static List<Integer> getAllSquaresInDirection(
    ArrayList<List<Integer>> squares, Direction direction) {
  return squares.get(direction.ordinal());
}

/**
 * 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) {

  return squares_direction.get(source_square).get(direction.ordinal());
}

/**
 * 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) {
  return squares_direction_knight.get(source_square);
}

/**
 * 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));
}

/**
 * returns the number for the i_th row seen from a given side. i.e. the last
 * row for black is 1, the 3rd row for white is 3, the 3rd row for black is
 * 6;
 *
 * @param side
 *           the given side
 * @param i_th
 *           the i_th row, where the (global) row number is wanted.
 * @return the (global) row number.
 */
public static int getRowForSide(Side side, int i_th) {
  if (side == Side.BLACK)
    return 9 - i_th;
  else
    return i_th;
}

}
```

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);
```

```
/**
 * 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 IPosition doMove_copy(IMove move);

/**
 * Performs the given move on the actual board. 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.
 */
public void doMove(IMove move);

/**
 * Reverts the given move. In addition a stack is used to recover the whole
 * information. 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.
 */
public void undoMove(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 square, where the king for a side is positioned.
 *
 * @param side
 *            the given side
 * @return the square where the king is
 */
public int getKingPos(Side side);

/**
 * 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);
```

```java
/**
 * 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();

  /**
   * computes all information and stores them, which is needed for fast board
   * evaluation.
   */
  public void cacheOccupiedSquares();

  public void setHalfMoveClock(int parseInt);

  public int getHalfMoveClock();
}
```

IPositionAnalyzer.java

```java
package mitzi;

public interface IPositionAnalyzer {

  /**
   * Evaluates the given board and returns a value in centipawns, this
   * function should not include further increase of search depth.
   *
   * @param board
   *           the board to be analyzed
   * @return a analysisResult, containing the value in centipawns
   */
  public AnalysisResult eval0(IPosition board);

  /**
   * Evaluates the given board and returns a value in centipawns, this
   * function should/can include further increase of search depth.
   *
   * @param board
   *           the board to be analyzed
   * @param alpha
   *           the alpha value of the alpha-beta algorithm
   * @param beta
   *           the beta value of the alpha-beta algorithm
   * @return a analysisResult, containing the value in centipawns and the
   *        selective depth
   * @throws InterruptedException
   */
  public AnalysisResult evalBoard(IPosition board, int alpha, int beta)
      throws InterruptedException;

}
```

RandyBrain.java

```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;
  }
}
```

Direction.java

```
package mitzi;

import java.util.EnumSet;
```

```java
/**
 * 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);
    }
  }

}
```

Position.java

```
package mitzi;

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;

import mitzi.IrreversibleMoveStack.MoveInfo;

/**
 * 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;

  /**
   * This is the number of halfmoves since the last pawn advance or capture.
   * This is used to determine if a draw can be claimed under the fifty-move
   * rule.
   */
  public int half_move_clock;

  // The following class members are used to prevent multiple computations
  /**
   * caching of the possible moves
   */
  private List<IMove> possible_moves = new ArrayList<IMove>(50);
```

```
/**
 * true if, the possible moves were not computed for this position.
 */
private boolean possible_moves_is_null = true;

/**
 * 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 int[] num_occupied_squares_by_color_and_type = new int[16];

/**
 * caching the positions of the kings. (indexed by the ordinal of the side)
 */
private int[] king_pos = new int[2];

/**
 * saves the side, which got captured by the last tinyDoMove
 */
private Side side_capture;

/**
 * saves the piece, which got captured by the last tinyDoMove
 */
private Piece piece_capture;

/**
 *  saves if the old position after tinyDoMove was check or not
 */
Boolean old_check;

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

/**
 * Resets and clears the stored class members.
 */
private void resetCache() {
  possible_moves.clear();
  possible_moves_is_null = true;
  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);

    System.arraycopy(king_pos, 0, newBoard.king_pos, 0, 2);
    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;

  half_move_clock = 0;
  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;

  king_pos[Side.WHITE.ordinal()] = 51;
  king_pos[Side.BLACK.ordinal()] = 58;
  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);
      king_pos[Side.WHITE.ordinal()] = (byte) square;
      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);
        king_pos[Side.BLACK.ordinal()] = (byte) square;
        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 IPosition doMove_copy(IMove move) {
    Position newBoard = this.returnCopy();

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

    Piece piece = getPieceFromBoard(src);
    Piece capture = getPieceFromBoard(dest);
    boolean resets_half_move_clock = false;
    // if promotion
    if (move.getPromotion() != null) {
      newBoard.setOnBoard(src, null, null);
      newBoard.setOnBoard(dest, active_color, move.getPromotion());
      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);
      }
      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)
        resets_half_move_clock = true;
    }

    if (resets_half_move_clock)
      newBoard.half_move_clock = 0;

    // 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);

    // 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) {
      newBoard.king_pos[active_color.ordinal()] = (byte) dest;
      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;
      }
    }
    if (capture == Piece.ROOK) {
      if (active_color == Side.BLACK) {
        if (dest == 81)
          newBoard.castling[1] = -1;
        else if (dest == 11)
          newBoard.castling[0] = -1;
      } else {
        if (dest == 88)
          newBoard.castling[3] = -1;
        else if (dest == 18)
          newBoard.castling[2] = -1;
      }
    }

    return newBoard;
  }

  @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()) {
    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) {
          setOnBoard(squ, active_color, Piece.KING);
          setOnBoard(square, null, null);

          if (isCheckPosition()) {
            setOnBoard(square, active_color, Piece.KING);
            setOnBoard(squ, null, null);
            break;
          }
          setOnBoard(square, active_color, Piece.KING);
          setOnBoard(squ, null, null);
          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) {
    Set<Integer> set = new HashSet<Integer>();

    for (int square : SquareHelper.all_squares)
      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) {
    Set<Integer> set = new HashSet<Integer>();

    for (int square : SquareHelper.all_squares)
      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) {
    Set<Integer> set = new HashSet<Integer>();
    if (type == Piece.KING)
      set.add((int) king_pos[color.ordinal()]);
    else {
      for (int square : SquareHelper.all_squares)
        if (type == getPieceFromBoard(square)
            && color == getSideFromBoard(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_is_null == true) {

    // loop over all squares
    for (int square : SquareHelper.all_squares) {
      if (getSideFromBoard(square) == active_color)
        possible_moves.addAll(getPossibleMovesFrom(square));
    }
    possible_moves_is_null = false;
  }

  return possible_moves;
}
```

```
@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();

  ArrayList<List<Integer>> all_squares = SquareHelper
      .getSquaresAllDirections(square);
  List<Integer> squares;
  List<IMove> moves = new ArrayList<IMove>(35);
  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(
            all_squares, 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);

          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);

            move = new Move(square, square + direction.offset,
                Piece.ROOK);
            moves.add(move);
            move = new Move(square, square + direction.offset,
                Piece.BISHOP);
            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
          int last_squ = line.get(line.size() - 1);
          for (Integer squ : line) {
            if (squ == last_squ)
              break;
            if (getSideFromBoard(squ) != null) {
              castle_flag = 1;
              break;
            }

          }
          if (castle_flag == 1)
            continue;

          // Check each square if the king on it would be check
          for (Integer squ : line) {

            setOnBoard(squ, active_color, Piece.KING);
            setOnBoard(square, null, null);

            if (isCheckPosition()) {
              setOnBoard(square, active_color, Piece.KING);
              setOnBoard(squ, null, null);
              break;
            }
            setOnBoard(square, active_color, Piece.KING);
            setOnBoard(squ, null, null);
            if (squ == new_square) {
              // if everything is right, then add the move
              move = new Move(square, squ);
              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
    Iterator<IMove> iter = moves.iterator();
    IMove mv;
    while (iter.hasNext()) {
      mv = iter.next();
      tinyDoMove(mv);
      active_color = Side.getOppositeSide(active_color);
      if (isCheckPosition()) {
        iter.remove();
      }
      active_color = Side.getOppositeSide(active_color);
      tinyUndoMove(mv);
    }

    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;
      int king_pos = getKingPos(active_color);
      ArrayList<List<Integer>> all_squares = SquareHelper
          .getSquaresAllDirections(king_pos);
```

```
// go in each direction
for (Direction direction : Direction.values()) {
  List<Integer> line = SquareHelper.getAllSquaresInDirection(
      all_squares, 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;
}

@Override
public int getKingPos(Side side) {
  return king_pos[side.ordinal()];
}

@Override
public void cacheOccupiedSquares() {

  Side s;
  Piece p;
  Set<Integer> w_pawn = new HashSet<Integer>();
  Set<Integer> w_rook = new HashSet<Integer>();
  Set<Integer> w_bishop = new HashSet<Integer>();
  Set<Integer> w_knight = new HashSet<Integer>();
  Set<Integer> w_queen = new HashSet<Integer>();

  Set<Integer> b_pawn = new HashSet<Integer>();
```

```
Set<Integer> b_rook = new HashSet<Integer>();
Set<Integer> b_bishop = new HashSet<Integer>();
Set<Integer> b_knight = new HashSet<Integer>();
Set<Integer> b_queen = new HashSet<Integer>();

for (int square : SquareHelper.all_squares) {

  s = getSideFromBoard(square);
  if (s == null)
    continue;
  p = getPieceFromBoard(square);
  switch (s) {
  case WHITE:
    switch (p) {
    case PAWN:
      w_pawn.add(square);
      break;
    case ROOK:
      w_rook.add(square);
      break;
    case BISHOP:
      w_bishop.add(square);
      break;
    case KNIGHT:
      w_knight.add(square);
      break;
    case QUEEN:
      w_queen.add(square);
      break;
    default:
      break;
    }
    break;
  case BLACK:
    switch (p) {
    case PAWN:
      b_pawn.add(square);
      break;
    case ROOK:
      b_rook.add(square);
      break;
    case BISHOP:
      b_bishop.add(square);
      break;
    case KNIGHT:
      b_knight.add(square);
```

```
        break;
      case QUEEN:
        b_queen.add(square);
        break;
      default:
        break;
      }
      break;
    }

  }

  occupied_squares_by_color_and_type.put(Side.WHITE.ordinal() * 10
      + Piece.PAWN.ordinal(), w_pawn);
  occupied_squares_by_color_and_type.put(Side.WHITE.ordinal() * 10
      + Piece.ROOK.ordinal(), w_rook);
  occupied_squares_by_color_and_type.put(Side.WHITE.ordinal() * 10
      + Piece.BISHOP.ordinal(), w_bishop);
  occupied_squares_by_color_and_type.put(Side.WHITE.ordinal() * 10
      + Piece.KNIGHT.ordinal(), w_knight);
  occupied_squares_by_color_and_type.put(Side.WHITE.ordinal() * 10
      + Piece.QUEEN.ordinal(), w_queen);

  occupied_squares_by_color_and_type.put(Side.BLACK.ordinal() * 10
      + Piece.PAWN.ordinal(), b_pawn);
  occupied_squares_by_color_and_type.put(Side.BLACK.ordinal() * 10
      + Piece.ROOK.ordinal(), b_rook);
  occupied_squares_by_color_and_type.put(Side.BLACK.ordinal() * 10
      + Piece.BISHOP.ordinal(), b_bishop);
  occupied_squares_by_color_and_type.put(Side.BLACK.ordinal() * 10
      + Piece.KNIGHT.ordinal(), b_knight);
  occupied_squares_by_color_and_type.put(Side.BLACK.ordinal() * 10
      + Piece.QUEEN.ordinal(), b_queen);

}

@Override
public void doMove(IMove move) {

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

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

  setOnBoard(dest, active_color, piece);
  setOnBoard(src, null, null);

  boolean resets_half_move_clock = false;

  // if promotion
  if (move.getPromotion() != null) {
    setOnBoard(dest, active_color, move.getPromotion());
    resets_half_move_clock = true;
    num_occupied_squares_by_color_and_type[active_color.ordinal() * 10
        + Piece.PAWN.ordinal()]--;
    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) {
    setOnBoard((src + dest) / 2, active_color, Piece.ROOK);
    if (SquareHelper.getColumn(dest) == 3)
      setOnBoard(src - 40, null, null);
    else
      setOnBoard(src + 30, null, null);
  }
  // If en passant
  else if (piece == Piece.PAWN && dest == en_passant_target) {
    if (active_color == Side.WHITE) {
      setOnBoard(dest - 1, null, null);
    } else {
      setOnBoard(dest + 1, null, null);
    }
    num_occupied_squares_by_color_and_type[Side.getOppositeSide(
        active_color).ordinal()
        * 10 + Piece.PAWN.ordinal()]--;
    resets_half_move_clock = true;
  }
  // Usual move
  else {
    if (capture != null || piece == Piece.PAWN)
      resets_half_move_clock = true;
  }

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

```
IrreversibleMoveStack.addInfo(half_move_clock, castling,
    en_passant_target, capture, is_check);

// reset half move clock
if (resets_half_move_clock)
  half_move_clock = 0;

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

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

```
        castling[2] = -1;
      }
    }

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

    resetCache();

  }

  @Override
  public void undoMove(IMove move) {

    resetCache();

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

    Piece piece = getPieceFromBoard(dest);

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

    // get the missing information
    MoveInfo inf = IrreversibleMoveStack.irr_move_info.removeLast();

    en_passant_target = inf.en_passant_square;
    Piece capture = inf.capture;
    half_move_clock = inf.half_move_clock;
    System.arraycopy(inf.castling, 0, castling, 0, 4);
    is_check = inf.is_check;

    setOnBoard(src, active_color, piece);
    if (capture != null)
      setOnBoard(dest, Side.getOppositeSide(active_color), capture);
    else
      setOnBoard(dest, null, null);

    // if promotion
    if (move.getPromotion() != null) {
      setOnBoard(src, active_color, Piece.PAWN);
      num_occupied_squares_by_color_and_type[active_color.ordinal() * 10
          + Piece.PAWN.ordinal()]++;
      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) {
      setOnBoard((src + dest) / 2, null, null);
      if (SquareHelper.getColumn(dest) == 3)
        setOnBoard(src - 40, active_color, Piece.ROOK);
      else
        setOnBoard(src + 30, active_color, Piece.ROOK);

    }
    // If en passant
    else if (piece == Piece.PAWN && dest == en_passant_target) {
      if (active_color == Side.WHITE) {
        setOnBoard(dest - 1, Side.getOppositeSide(active_color),
            Piece.PAWN);
      } else {
        setOnBoard(dest + 1, Side.getOppositeSide(active_color),
            Piece.PAWN);
      }
      num_occupied_squares_by_color_and_type[Side.getOppositeSide(
          active_color).ordinal()
          * 10 + Piece.PAWN.ordinal()]++;
    }

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

    if (piece == Piece.KING) {
      king_pos[active_color.ordinal()] = (byte) src;
    }

    is_mate = false;
    is_stale_mate = false;

  }

  /**
   * Performs a incomplete version of doMove. This function only sets the new
   * figure, deletes the captures ones (are saved in side_capture and
   * piece_capture) and changes the active color. Note that it is not possible
   * to perform tinyDoMove twice, because the captured figure of the first
   * application will be lost.
   *
   * @param move
   *           the move to be performed, must be a legal move
   */
  private void tinyDoMove(IMove move) {

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

    Piece piece = getPieceFromBoard(src);
    piece_capture = getPieceFromBoard(dest);
    side_capture = getSideFromBoard(dest);

    setOnBoard(dest, active_color, piece);
    setOnBoard(src, null, null);

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

    }
    // If en passant
    else if (piece == Piece.PAWN && dest == en_passant_target) {
      if (active_color == Side.WHITE)
        setOnBoard(dest - 1, null, null);
      else
        setOnBoard(dest + 1, null, null);
    }

    // Update castling
    if (piece == Piece.KING)
      king_pos[active_color.ordinal()] = (byte) dest;

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

    old_check = is_check;
    is_check = null;
    is_mate = null;
    is_stale_mate = null;

  }

  /**
   * inverts the function tinyDoMove(), note that only one application can be
   * inverted!
   *
   * @param move
   *          the move to be inverted.
   */
  private void tinyUndoMove(IMove move) {

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

    Piece piece = getPieceFromBoard(dest);

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

    setOnBoard(dest, side_capture, piece_capture);
    setOnBoard(src, active_color, piece);
    // if promotion
    if (move.getPromotion() != null) {
      setOnBoard(src, active_color, Piece.PAWN);
    }
    // If castling
    else if (piece == Piece.KING && Math.abs((src - dest)) == 20) {
      setOnBoard((src + dest) / 2, null, null);
      if (SquareHelper.getColumn(dest) == 3)
        setOnBoard(src - 40, active_color, Piece.ROOK);
      else
        setOnBoard(src + 30, active_color, Piece.ROOK);

    }
    // If en passant
    else if (piece == Piece.PAWN && dest == en_passant_target) {
      if (active_color == Side.WHITE)
        setOnBoard(dest - 1, Side.getOppositeSide(active_color),
            Piece.PAWN);
      else
        setOnBoard(dest + 1, Side.getOppositeSide(active_color),
            Piece.PAWN);
    }

    // Update king position
    if (piece == Piece.KING)
      king_pos[active_color.ordinal()] = (byte) src;

    is_check = old_check;
    is_mate = false;
    is_stale_mate = false;

  }

  @Override
  public void setHalfMoveClock(int parseInt) {
    half_move_clock = parseInt;

  }

  @Override
  public int getHalfMoveClock() {
    return half_move_clock;
  }
}
```

AnalysisResult.java

```
package mitzi;

import java.util.LinkedList;

/*
 * Size of the class:
 * Header:    8 bytes
 * short :    2 bytes
 * Boolean:   16 bytes (8 header + 1 boolean + 7 round up to multiple of 8)
 * boolean:   1 byte
 * 2*byte:    2 bytes
 * Flag:    4 bytes (like int? )
 * IMove:     16 bytes ( 8 header + 2 short+ 2 short + 4 Piece (like int) + 0
    round up to multiple of 8)
 * long:      8 bytes
 * linkedList: 24 + (8 + 24 + 16)k = 24 + 48k bytes
 *         (list: 8 header + 8 reference to first elem + 4 size + 8k reference
    to each node + 4 round up 24+ 8k)
 *        (node: 8 header + 8 reference to next node + 8 reference to class =
    24)
 *        (class: 16 byte)
 *
 * round up:  7 byte
 *  total size: 88 + 48k bytes.
 *
 * k = 3 : 232 bytes
 * k = 5 : 328 bytes
 * k =10 : 568 bytes
 *
 * lets assume that a single analysis needs about 500 bytes.
 * Then a 500 MB TranspositionTable can hold about 1 Mio entries.
 *
 * http://www.javamex.com/tutorials/memory/object_memory_usage.shtml
 * http://www.sandeshshrestha.com/blog/memory-used-by-java-data-types/
 */

public final class AnalysisResult {

  /**
   * The boards score in centipawns.
   */
  public short score;

  /**
   * If true, the board is a stalemate position. I. e. no moves are possible
   * but there is no check. If null, then it has not been analyzed.
   */
  public Boolean is_stalemate;

  /**
   * If the evaluation method considers this board to be in an unstable state
   * and recommends a deeper evalutation or is simply not sure, this is set to
   * true.
   */
  public boolean needs_deeper;
  /**
   * The distance to (complete) search depth at which this result was
   * obtained.
   */
  public byte plys_to_eval0 = 0;

  /**
   * The distance to selective search depth at which this result was obtained.
   */
```

```
public byte plys_to_seldepth = 0;

/**
 * The state of the result in alpha-beta search: exact, fail-high or
 * fail-low
 */
public Flag flag;

/**
 * The best move from current board.
 */
public IMove best_move;

/**
 * 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.
 **/
public long hashvalue;

/**
 * A sorted list of the better moves in reverse order, i.e. the last moves
 * are better, then the first ones.
 */
public LinkedList<IMove> best_moves = new LinkedList<IMove>();

AnalysisResult(int score, Boolean is_stalemate, boolean needs_deeper,
    int plys_to_eval0, int plys_to_seldepth, Flag flag) {
  this.score = (short) score;
  this.is_stalemate = is_stalemate;
  this.needs_deeper = needs_deeper;
  this.plys_to_eval0 = (byte) plys_to_eval0;
  this.plys_to_seldepth = (byte) plys_to_seldepth;
  this.flag = flag;
}

AnalysisResult(int score, Boolean is_stalemate, boolean needs_deeper,
    int plys_to_eval0, int plys_to_seldepth, Flag flag,
    IMove best_move, long hashvalue) {
  this.score = (short) score;
  this.is_stalemate = is_stalemate;
  this.needs_deeper = needs_deeper;
  this.plys_to_eval0 = (byte) plys_to_eval0;
  this.plys_to_seldepth = (byte) plys_to_seldepth;
```

```
    this.flag = flag;
    this.best_move = best_move;
    this.hashvalue = hashvalue;
  }

  /**
   * computes a copy of the analysis result without the list of good moves and
   * the hashvalue.
   *
   * @return a copy without some elements.
   */
  public AnalysisResult tinyCopy() {
    return new AnalysisResult(score, is_stalemate, needs_deeper,
        plys_to_eval0, plys_to_seldepth, null, best_move, 0);
  }

  /**
   * sets all values of analysis result except the list of good moves and the
   * hashvalue.
   *
   * @param score
   *          the new score
   * @param is_stalemate
   *          the new status of is_stalemate
   * @param needs_deeper
   *          the new status of needs_deeper
   * @param plys_to_eval0
   *          the new number of plys to base case.
   * @param plys_to_seldepth
   *          the new number of plys to base case of selective depth.
   * @param flag
   *          the new flag
   * @param best_move
   *          the new best move.
   */
  public void tinySet(int score, boolean is_stalemate, boolean needs_deeper,
      int plys_to_eval0, int plys_to_seldepth, Flag flag, IMove best_move) {
    this.score = (short) score;
    this.is_stalemate = is_stalemate;
    this.needs_deeper = needs_deeper;
    this.plys_to_eval0 = (byte) plys_to_eval0;
    this.plys_to_seldepth = (byte) plys_to_seldepth;
    this.flag = flag;
    this.best_move = best_move;
  }
```

```
/**
 * sets all values of analysis result except the list of good moves and the
 * hashvalue.
 *
 * @param ar
 *          the new analysis result
 */
public void tinySet(AnalysisResult ar) {
  tinySet(ar.score, ar.is_stalemate, ar.needs_deeper, ar.plys_to_eval0,
      ar.plys_to_seldepth, ar.flag, ar.best_move);
}

/**
 * enables a comparison of two results.
 *
 * @param o
 *          the other result.
 * @return 0 if there are the same or have the same value, 1 if the actual
 *         one is more valuable then the other one, -1 else.
 */
public int compareQualityTo(AnalysisResult o) {
  if (o == null)
    throw new NullPointerException();

  if (this == o)
    return 0;

  // (deeper results)
  if (this.plys_to_eval0 > o.plys_to_eval0)
    return 1;

  if (this.plys_to_eval0 == o.plys_to_eval0
      && this.plys_to_seldepth == o.plys_to_seldepth)
    return 0;

  return -1;
}

/**
 * returns the Principal Variation, i.e. a sequence of best moves moves. It
 * may be cut off, if entries in the ResultCache are overridden.
 *
 * @param pos
 *          the position, where the PV starts
 * @return a linked list with the PV
 */
```

```
  public LinkedList<IMove> getPV(IPosition pos, int counter) {
    LinkedList<IMove> pv = new LinkedList<IMove>();
    IPosition best_child;
    AnalysisResult ar;
    if (best_move != null && counter >=0) {
      pv.add(best_move);
      best_child = pos.doMove_copy(best_move);
      ar = ResultCache.getResult(best_child);
      counter--;
      if (ar != null)
        pv.addAll(ar.getPV(best_child, counter));
    }
    return pv;
  }

  @Override
  public String toString() {
    return "cp: " + score + " depth: " + plys_to_eval0
        + (flag != null ? " flag: " + flag : "");
  }
}
```

BasicBoardAnalyzer.java

```
package mitzi;

public class BasicBoardAnalyzer implements IPositionAnalyzer {

  @Override
  public AnalysisResult eval0(IPosition board) {

    int score = 0;

    int[] piece_values = { 100, 500, 325, 325, 975, 000 };
    int bishop_pair_value = 50;

    // basic evaluation
    for (Side side : Side.values()) {
      int side_sign = Side.getSideSign(side);

      // piece values
      for (Piece piece : Piece.values()) {
        score += board.getNumberOfPiecesByColorAndType(side, piece)
            * piece_values[piece.ordinal()] * side_sign;
      }

      // bishop pair gives bonus
      if (board.getNumberOfPiecesByColorAndType(side, Piece.BISHOP) == 2) {
        score += bishop_pair_value * side_sign;
      }
    }
    AnalysisResult result = new AnalysisResult(score, null, true, 0, 0, null);
    return result;
  }

  @Override
  public AnalysisResult evalBoard(IPosition board,int alpha, int beta){

    return eval0(board);
  }
}
```

BasicMoveComparator.java

```
package mitzi;

import java.util.Comparator;
import java.util.HashMap;
import java.util.Map;

public class BasicMoveComparator implements Comparator<IMove> {

  /**
   * saves the actual board, where the moves should be compared
   */
  private IPosition board;

  /**
   * map, which maps a move to its value. Initial size set to 35 to prevent
   */
  private Map<IMove, Integer> move_values = new HashMap<IMove, Integer>(35,
      1);

  /**
   * contains values for move comparison
   */
  private static final int[] piece_values = { 100, 500, 325, 325, 975, 000 };
```

```
/**
 * value of a square where a piece moves to or from.
 */
private static final int[] center_values = { -1, -1, -1, -1, -1, -1, -1,
    -1, -1, -1, -1, 0, 2, 4, 7, 7, 4, 3, 0, -1, -1, 1, 5, 8, 12, 12, 8,
    5, 1, -1, -1, 3, 8, 12, 17, 17, 12, 8, 3, -1, -1, 6, 10, 15, 20,
    20, 15, 10, 6, -1, -1, 6, 10, 15, 20, 20, 15, 10, 6, -1, -1, 3, 8,
    12, 17, 17, 12, 8, 3, -1, -1, 1, 5, 8, 12, 12, 8, 5, 1, -1, -1, 0,
    2, 4, 7, 7, 4, 2, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 };


public BasicMoveComparator(IPosition board) {
  this.board = board;
}

/**
 * Grades an IMove by some heuristics.
 *
 * Ignoring special situations like en passant and castling.
 *
 * @param move the current move
 */
private void computeValue(IMove move) {
  int value = 0;

  // moved figure
  Piece src_piece = board.getPieceFromBoard(move.getFromSquare());

  // captured figure
  Piece dest_piece = board.getPieceFromBoard(move.getToSquare());

  if (dest_piece != null) {
    // try to get advantage in exchange
    value += (piece_values[dest_piece.ordinal()]
        - piece_values[src_piece.ordinal()] + 1) * 16;
  }
  // move with more powerful pieces
  value += piece_values[src_piece.ordinal()];

  // move to the center (but away with the king)
  value += (center_values[move.getToSquare()] - center_values[move
      .getFromSquare()]) * (src_piece == Piece.KING ? -1 : 1);

  move_values.put(move, value);
}
```

```
  /**
   * compares two moves by there value.
   */
  @Override
  public int compare(IMove m1, IMove m2) {
    if (!move_values.containsKey(m1))
      computeValue(m1);
    if (!move_values.containsKey(m2))
      computeValue(m2);

    return Integer.compare(move_values.get(m1), move_values.get(m2));
  }

}
```

BoardAnalyzer.java

```
package mitzi;

import static mitzi.MateScores.NEG_INF;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Set;

/**
 *
 * This class computes the value of a board in a proper way, see
 * http://philemon.cycovery.com/site/part2.html for more details.
 *
 */
public class BoardAnalyzer implements IPositionAnalyzer {

  /**
   * the square to array index from Position.java
   */
  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 material value of a piece.
   */
  static private int[] piece_values = { 100, 500, 325, 325, 975, 000 };

  // The following arrays contains the value of a piece on a specific square,
  // always in favor of white. Since the arrays are symmetric w.r.t. the
  // columns, BLACK uses 63-i entry with opposite sign.
  /**
   * value of squares for bishop and knight, in favor of white
   */
  static private int[] piece_activity_b_k = { -16, -16, -8, -8, -8, -8, -16,
      -16, -16, -16, -4, -4, -4, -4, -16, -16, -8, 2, 6, 6, 6, 6, 2, -8,
      -8, 2, 6, 6, 6, 6, 2, -8, -8, 2, 4, 4, 4, 4, 2, -8, -8, 2, 2, 2, 2,
      2, 2, -8, -8, -8, 0, 0, 0, 0, -8, -8, -16, -8, -8, -8, -8, -8, -8,
      -16 };
  /**
   * value of squares for rook, in favor of white
   */
  static private int[] piece_activity_r = { 0, 0, 4, 6, 6, 4, 0, 0, 0, 0, 4,
      6, 6, 4, 0, 0, 0, 0, 4, 6, 6, 4, 0, 0, 0, 0, 4, 6, 6, 4, 0, 0, 0,
      0, 4, 6, 6, 4, 0, 0, 0, 0, 4, 6, 6, 4, 0, 0, 0, 0, 4, 6, 6, 4, 0,
      0, 0, 0, 4, 6, 6, 4, 0, 0, };

  /**
   * value of squares for queen, in favor of white
   */
  static private int[] piece_activity_q = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4,
      5, 5, 4, 0, 0, 0, 2, 4, 10, 10, 4, 2, 0, 0, 2, 10, 12, 12, 10, 2,
      0, -10, 2, 10, 12, 12, 10, 2, -10, -10, -10, 4, 10, 10, 4, -10,
      -10, -10, 2, 8, 8, 8, 8, 2, -10, -10, -8, 0, 0, 0, 0, -8, -10, };

  /**
   * value of squares, which are weak/strong squares for bishop and knight
   */
  static private int[] weak_positions = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 8, 12, 12, 8, 0, 0, 0, 2, 12, 16, 16, 12, 2, 0,
      0, 2, 12, 20, 20, 12, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, };

  /**
   * value of squares for white pawns. (not symmetric)
   */
```

```
static private int[] pawn_positions_w = { 0, 0, 0, 0, 0, 0, 0, 0, 28, 28,
   35, 42, 45, 35, 28, 28, -9, -3, 7, 12, 15, 7, -3, -9, -10, -10, 6,
   9, 10, 6, -11, -10, -11, -11, 4, 5, 6, 2, -11, -11, -11, -11, 0, 0,
   1, 0, -11, -11, -6, -6, 4, 5, 5, 4, -6, -6, 0, 0, 0, 0, 0, 0, 0, 0 };

/**
 * value of squares for black pawns. (not symmetric)
 */
static private int[] pawn_positions_b = { 0, 0, 0, 0, 0, 0, 0, 0, -6, -6,
   4, 5, 5, 4, -6, -6, -11, -11, 0, 0, 1, 0, -11, -11, -11, -11, 4, 5,
   6, 2, -11, -11, -10, -10, 6, 9, 10, 6, -11, -10, -9, -3, 7, 12, 15,
   7, -3, -9, 28, 28, 35, 42, 45, 35, 28, 28, 0, 0, 0, 0, 0, 0, 0, 0 };

/**
 * value of squares for white king, not valid in endgame. (not symmetric)
 */
static private int[] king_positions_w = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -10, -15,
   -10, 0, 0, 5, 10, 18, -8, -3, -8, 23, 10 };

/**
 * value of squares for black king, not valid in endgame. (not symmetric)
 */
static private int[] king_positions_b = { 5, 10, 18, -8, -3, -8, 23, 10, 0,
   0, 0, -10, -15, -10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

/**
 * value for twin pawns for different rows.
 */
static private int[] twin_pawns = { 0, 0, 1, 2, 3, 4, 7, 0 };

/**
 * value for covered pawns for different rows.
 */
static private int[] covered_pawns = { 0, 0, 4, 6, 8, 12, 16, 0 };

/**
 * value for passed pawns for different rows.
 */
static private int[] passed_pawn = { 0, 2, 10, 20, 40, 60, 70, 0 };

/**
 * value for passed pawns, where in front of the pawn is a king for
 * different rows.
 */
static private int[] passed_pawn_with_king = { 0, 0, 0, 0, 10, 50, 80, 0 };

/**
 * value for blocked passed pawns
 */
static private int[] blocked_passed_pawn = { 0, 0, -8, -16, -32, -45, -58,
    0 };

/**
 * if not all Bishop and Knight has moved, moving the queen results in
 * negative score
 */
static private int PREMATURE_QUEEN = -17;

/**
 * bonus, if the rook is on an open line (no other pawns)
 */
static private int ROOK_OPEN_LINE = 20;

/**
 * bonus if the rook is on an halfopen line (only opponents pawns)
 */
static private int ROOK_HALFOPEN_LINE = 5;

/**
 * bonus if the rook is in the 7th row and opponents king is in the 8th or
 * pawn in the 7th
 */
static private int ROOK_7TH_2ND = 25;

/**
 * bonus if the previous bonus holds and the 7th row is empty.
 */
static private int ROOK_7TH_2ND_ABSOLUTE = 15;

/**
 * bonus if a rook covers the other rook, this replaces the ROOK_7TH_2ND and
 * counts for each rook (both on the 7th row)
 */
static private int REINFORCED_ROOK_7TH_2ND = 40;

/**
 * bonus if a rook is behind a passed pawn
 */
```

```
static private int PASSED_ROOK_SUPPORT = 10;

/**
 * gives a bonus if both bishops are still available in the endgame
 */
static private int ENDGAME_BISHOP_BONUS = 10; // not yet implemented

/**
 * bonus/malus if the bishop is caged on he baseline (the pawn in front of
 * the bishop has moved and the two pawns left and right of the bisop have
 * not moved )
 */
static private int BISHOP_BASELINE_CAGED = -12;

/**
 * bonus if a queen is covered on the 7th row by a rook
 */
static private int REINFORCING_QUEEN_7TH_2ND = 20;

/**
 * The player receives a bonus if the 2 bishops are alive.
 */
static private int bishop_pair_value = 25;

/**
 * multiple pawns in a columns get a malus.
 */
static private int MULTI_PAWN = -10;

/**
 * a isolated pawns (no pawn on the neighboring colums) get a malus
 */
static private int ISOLATED_PAWN = -20;

/**
 * bonus for coverd passed pawns on the 7th row
 */
static private int COVERED_PASSED_7TH_PAWN = 90;

/**
 * malus if castling is loss (needs to be fixed and optimized)
 */
static private int CASTLING_LOSS = -40;

/**
 * the number of pieces when the endgame starts (a first draft, needs to be
 * optimized)
 */
static public int ENDGAME_THRESHOLD = 8;

/**
 * counts the number of board evaluations in quiesce().
 */
static public long eval_counter_seldepth = 0;

/**
 * counts the number of found positions in Transposition Table
 */
static public long table_counter = 0;

@Override
public AnalysisResult eval0(IPosition board) {
  int score = 0;
  // compute all the information needed by the evaluation function once.
  board.cacheOccupiedSquares();

  // Evaluate the pieces
  score += evalPieces(board);

  // Evaluate Pawn Stucture
  score += evalPawns(board);

  // Evaluate Diagonals and lines
  score += evalLinesAndDiagonals(board);

  // Evaluate position - activity
  score += evalPieceActivity(board);

  // Evaluate weak/strong position
  score += evalWeakPosition(board);

  // Evaluate the King's position (not in endgame)
  score += evalKingPos(board);

  AnalysisResult result = new AnalysisResult(score, false, false, 0, 0,
      Flag.EXACT);
  return result;
}

@Override
public AnalysisResult evalBoard(IPosition position, int alpha, int beta)
    throws InterruptedException {
```

```
    AnalysisResult result = quiesce(position, alpha, beta);

    // The analysis result should always contain the pure value (not
    // perturbed via side_sign)
    return result;
  }

  /**
   * Implements Quiescence search to avoid the horizon effect. The function
   * increase the search depth until no capture is possible, where only
   * captures are analyzed. The optimal value is found using the negamax
   * algorithm.
   *
   * @see <a
   *
       href="http://chessprogramming.wikispaces.com/Quiescence+Search">http://chessprogrammi
   *
   * @param position
   *          the position to be analyzed
   * @param alpha
   *          the alpha value of alpha-beta search
   * @param beta
   *          the beta value of alpha-beta search
   * @return the value of the board ( in favor of white)
   *
   * @throws InterruptedException
   */
  private AnalysisResult quiesce(IPosition position, int alpha, int beta)
      throws InterruptedException {

    if (Thread.interrupted()) {
      throw new InterruptedException();
    }

    int side_sign = Side.getSideSign(position.getActiveColor());

    // Cache lookup
    AnalysisResult entry = ResultCache.getResult(position);
    if (entry != null) {
      table_counter++;
      if (entry.flag == Flag.EXACT) {
        AnalysisResult new_entry = entry.tinyCopy();
        return new_entry;
      } else if (entry.flag == Flag.LOWERBOUND)
        alpha = Math.max(alpha, entry.score * side_sign);
      else if (entry.flag == Flag.UPPERBOUND)
        beta = Math.min(beta, entry.score * side_sign);

      if (alpha >= beta) {
        AnalysisResult new_entry = entry.tinyCopy();
        return new_entry;
      }
    }

    // generate moves
    List<IMove> moves = position.getPossibleMoves();

    // check for mate and stalemate
    if (moves.isEmpty()) {
      eval_counter_seldepth++;
      if (position.isCheckPosition()) {
        return new AnalysisResult(NEG_INF * side_sign, false, false, 0,
            0, Flag.EXACT);
      } else {
        return new AnalysisResult(0, true, false, 0, 0, Flag.EXACT);
      }
    }

    // evaluation of the current board.
    AnalysisResult standing_pat = eval0(position);
    eval_counter_seldepth++;

    int negaval = standing_pat.score * side_sign;

    // alpha beta cutoff
    if (negaval >= beta)
      return standing_pat;
    alpha = Math.max(alpha, negaval);

    // Generate possible Captures
    List<IMove> caputures = position.generateCaptures();

    // Generate MoveComperator
    BasicMoveComparator move_comparator = new BasicMoveComparator(position);

    // no previous computation given, use basic heuristic
    ArrayList<IMove> ordered_captures = new ArrayList<IMove>(caputures);
    Collections.sort(ordered_captures,
        Collections.reverseOrder(move_comparator));

    AnalysisResult result = null;
    int best_value = NEG_INF;
```

```
    for (IMove move : ordered_captures) {

      position.doMove(move);
      AnalysisResult result_temp = quiesce(position, -beta, -alpha);
      position.undoMove(move);

      negaval = result_temp.score * side_sign;

      // find the best result
      if (negaval > best_value) {
        best_value = negaval;
        result = result_temp;
      }

      // cut-off
      if (negaval >= beta) {
        result.plys_to_seldepth++;
        return result;
      }
      alpha = Math.max(alpha, negaval);
    }

    // the standing_pat was computed in this depth
    if (result == null)
      return standing_pat;

    // the result comes from a depth below
    result.plys_to_seldepth++;
    return result;

  }

  /**
   * Evaluates only the material value of the board.
   *
   * @param board
   *           the actual board
   * @return the material value ( in favor of white)
   */
  private int evalPieces(IPosition board) {
    int score = 0;

    // basic evaluation
    for (Side side : Side.values()) {
      int side_sign = Side.getSideSign(side);
      // piece values
      for (Piece piece : Piece.values()) {
        score += board.getNumberOfPiecesByColorAndType(side, piece)
            * piece_values[piece.ordinal()] * side_sign;
      }

      // bishop pair gives bonus
      if (board.getNumberOfPiecesByColorAndType(side, Piece.BISHOP) == 2) {
        score += bishop_pair_value * side_sign;
      }
    }

    return score;
  }

  /**
   * Computes the value of the possible activity of the pieces, e.g.
   * centralization,...
   *
   * @param board
   *          the board to be analyzed
   * @return the score for the activity of Rook, Bishop, Knight, Queen ( in
   *         favor of white)
   */
  private int evalPieceActivity(IPosition board) {
    int score = 0;
    Set<Integer> squares;
    boolean queen_moved_last, queen_startpos;

    for (Side side : Side.values()) {
      int side_sign = Side.getSideSign(side);
      queen_moved_last = true;
      queen_startpos = false;

      // Queen
      squares = board.getOccupiedSquaresByColorAndType(side, Piece.QUEEN);
      if (side == Side.WHITE)
        for (int squ : squares)
          score += piece_activity_q[square_to_array_index[squ]];
      else
        for (int squ : squares)
          score -= piece_activity_q[63 - square_to_array_index[squ]];

      if ((squares.contains(SquareHelper.getSquare(
          SquareHelper.getRowForSide(side, 1), 4)))
        queen_startpos = true;

      // Bishop
      squares = board
          .getOccupiedSquaresByColorAndType(side, Piece.BISHOP);
      if (side == Side.WHITE)
        for (int squ : squares)
          score += piece_activity_b_k[square_to_array_index[squ]];
      else
        for (int squ : squares)
          score -= piece_activity_b_k[63 - square_to_array_index[squ]];

      if (!queen_startpos
          && (squares.contains(SquareHelper.getSquare(
              SquareHelper.getRowForSide(side, 1), 3)) || squares
              .contains(SquareHelper.getSquare(
                  SquareHelper.getRowForSide(side, 1), 3))))
        queen_moved_last = false;

      // Knight
      squares = board
          .getOccupiedSquaresByColorAndType(side, Piece.KNIGHT);
      if (side == Side.WHITE)
        for (int squ : squares)
          score += piece_activity_b_k[square_to_array_index[squ]];
      else
        for (int squ : squares)
          score -= piece_activity_b_k[63 - square_to_array_index[squ]];

      if (!queen_startpos
          && (squares.contains(SquareHelper.getSquare(
              SquareHelper.getRowForSide(side, 1), 2)) || squares
              .contains(SquareHelper.getSquare(
                  SquareHelper.getRowForSide(side, 1), 2))))
        queen_moved_last = false;

      // Rook
      squares = board.getOccupiedSquaresByColorAndType(side, Piece.ROOK);
      if (side == Side.WHITE)
        for (int squ : squares)
          score += piece_activity_r[square_to_array_index[squ]];
      else
        for (int squ : squares)
          score -= piece_activity_r[63 - square_to_array_index[squ]];

      if (!queen_startpos && !queen_moved_last)
```

```
        score += side_sign * PREMATURE_QUEEN;

    }
    return score;
  }

  /**
   * this function evaluates the weak position of an outpost, however only for
   * bishop and knight. If a knight is covered by pawn, the value increases.
   *
   * @param board
   *           the board to be analyzed
   * @return the score w.r.t. weak/ strong positions ( in favor of white)
   */
  private int evalWeakPosition(IPosition board) {

    int score = 0;
    Set<Integer> squares;

    for (Side side : Side.values()) {

      // Bishop
      squares = board
          .getOccupiedSquaresByColorAndType(side, Piece.BISHOP);
      if (side == Side.WHITE)
        for (int squ : squares)
          score += weak_positions[square_to_array_index[squ]];
      else
        for (int squ : squares)
          score -= weak_positions[63 - square_to_array_index[squ]];

      // Knight (value get multiplied times the number of pawn covering
      // the knight, if no cover no bonus is added)
      squares = board
          .getOccupiedSquaresByColorAndType(side, Piece.BISHOP);
      int count = 0;
      if (side == Side.WHITE) {
        for (int squ : squares) {
          for (Direction dir : Direction
              .pawnCapturingDirections(Side.BLACK))
            if (board.getPieceFromBoard(squ + dir.offset) == Piece.PAWN)
              count++;
          score += count * weak_positions[square_to_array_index[squ]];
        }
      } else {
        for (int squ : squares) {
```

```
        for (Direction dir : Direction
            .pawnCapturingDirections(Side.WHITE))
          if (board.getPieceFromBoard(squ + dir.offset) == Piece.PAWN)
            count++;
        score -= count
            * weak_positions[63 - square_to_array_index[squ]];
      }
    }

  }

  return score;
}

/**
 * Evaluates if rooks occupies open/halfopen lines, if they occupies the
 * 7-th row or are covered there and if the bishop is caged on the baseline
 * (the pawn in front of him has moved the the neighboring ones are here)
 *
 * @param board
 *          the board to be evaluated
 * @return the score ( in favor of white)
 */
private int evalLinesAndDiagonals(IPosition board) {
  int score = 0;

  Set<Integer> squares_rook, squares_bishop;
  Piece p;
  Side s;

  for (Side side : Side.values()) {
    int side_sign = Side.getSideSign(side);
    Side opp_side = Side.getOppositeSide(side);

    squares_rook = board.getOccupiedSquaresByColorAndType(side,
        Piece.ROOK);

    // Open line and halfopen line bonus
    for (int square : squares_rook) {

      boolean half_open = true;
      boolean open = true;
      List<Integer> squares = new ArrayList<Integer>(
          SquareHelper.getAllSquaresInDirection(square,
              Direction.NORTH));
      squares.addAll(SquareHelper.getAllSquaresInDirection(square,
          Direction.SOUTH));

      for (int squ : squares) {
        if (board.getPieceFromBoard(squ) == Piece.PAWN) {
          if (board.getSideFromBoard(squ) == board
              .getActiveColor()) {
            half_open = false;
            open = false;
            break;
          } else
            open = false;
        }
      }
      if (half_open && open)
        score += side_sign * ROOK_OPEN_LINE;
      else if (half_open)
        score += side_sign * ROOK_HALFOPEN_LINE;

      // 7th || 2nd line bonus
      if (SquareHelper.getRow(square) == SquareHelper.getRowForSide(
          side, 7)) {

        boolean rook_7 = true;
        boolean rook_7_abs = true;
        boolean rook_7_cover_r = false;
        boolean rook_7_cover_q = false;
        boolean emty_direction = true;

        // Check all squares at the west side of the rook
        for (int squ : SquareHelper.getAllSquaresInDirection(
            square, Direction.WEST)) {
          p = board.getPieceFromBoard(squ);
          s = board.getSideFromBoard(squ);
          if (p == Piece.PAWN && s == opp_side) {
            rook_7 = false;
            rook_7_abs = false;
            break;
          } else if (emty_direction && p != null) {
            emty_direction = false;
            rook_7_abs = false;
            if (p == Piece.ROOK && s == side)
              rook_7_cover_r = true;
            else if (p == Piece.QUEEN && s == side)
              rook_7_cover_q = true;
          }
```

```
    }

    // Check all squares at the west side of the rook
    emty_direction = true;
    for (int squ : SquareHelper.getAllSquaresInDirection(
        square, Direction.EAST)) {
      p = board.getPieceFromBoard(squ);
      s = board.getSideFromBoard(squ);
      if (rook_7 && p == Piece.PAWN && s == opp_side) {
        rook_7 = false;
        rook_7_abs = false;
        break;
      } else if (emty_direction && p != null) {
        rook_7_abs = false;
        emty_direction = false;
        if (p == Piece.ROOK && s == side)
          rook_7_cover_r = true;
        else if (p == Piece.QUEEN && s == side)
          rook_7_cover_q = true;
      }

    }

    int king_pos = board.getKingPos(opp_side);
    if (SquareHelper.getRow(king_pos) == SquareHelper
        .getRowForSide(side, 8))
      rook_7 = true;

    if (rook_7)
      score += side_sign * ROOK_7TH_2ND;
    if (rook_7_abs)
      score += side_sign * ROOK_7TH_2ND_ABSOLUTE;
    if (rook_7_cover_r)
      score += side_sign * REINFORCED_ROOK_7TH_2ND;
    if (rook_7_cover_q)
      score += side_sign * REINFORCING_QUEEN_7TH_2ND;
  }

}
squares_bishop = board.getOccupiedSquaresByColorAndType(side,
    Piece.BISHOP);
int row_s = SquareHelper.getRowForSide(side, 1);
boolean bishop_caged = false;
for (int square : squares_bishop)
  if ((square == SquareHelper.getSquare(row_s, 3) || square ==
      SquareHelper
```

```
          .getSquare(row_s, 6))
          && (board.getPieceFromBoard(square
              + Direction.pawnDirection(side).offset) == Piece.PAWN && board
              .getSideFromBoard(square
                  + Direction.pawnDirection(side).offset) == side)) {
        bishop_caged = true;
        for (Direction dir : Direction
            .pawnCapturingDirections(side)) {
          if (board.getPieceFromBoard(square + dir.offset) != Piece.PAWN
              || board.getSideFromBoard(square + dir.offset) != side)
            bishop_caged = false;
        }
      }
    if (bishop_caged == true)
      score += side_sign * BISHOP_BASELINE_CAGED;
  }
  return score;
}

/**
 * evaluates the pawn structure. Checks for covered pawns, passed pawns,
 * isolated pawns, twin pawns... value dependent of the row
 *
 * @param position
 *            the current position
 * @return the value of the pawn structure in favor of white
 */
private int evalPawns(IPosition position) {

  int score = 0;
  int row, col, col_2, row_side;
  boolean isolated, covered, passed;
  for (Side side : Side.values()) {
    int side_sign = Side.getSideSign(side);
    Side opp_side = Side.getOppositeSide(side);
    Set<Integer> squares_pawn = position
        .getOccupiedSquaresByColorAndType(side, Piece.PAWN);
    Set<Integer> squares_pawn_opp = position
        .getOccupiedSquaresByColorAndType(opp_side, Piece.PAWN);

    if (side == Side.WHITE)
      for (int squ : squares_pawn)
        score += pawn_positions_w[square_to_array_index[squ]];
    else
      for (int squ : squares_pawn)
        score -= pawn_positions_b[square_to_array_index[squ]];
```

```
for (int squ_1 : squares_pawn) {
  row = SquareHelper.getRow(squ_1);
  col = SquareHelper.getColumn(squ_1);

  row_side = SquareHelper.getRowForSide(side, row);

  isolated = true;
  covered = false;

  for (int squ_2 : squares_pawn) {
    // dont check the pawn with himself
    if (squ_2 == squ_1)
      continue;

    col_2 = SquareHelper.getColumn(squ_2);
    if (col == col_2)
      // add malus for multiple pawns in the same line.
      // TODO: maybe dont increase malus for triple,.. pawns
      score += side_sign * MULTI_PAWN;
    else if (col == col_2 + 1 || col == col_2 - 1) {
      isolated = false;

      if (row == SquareHelper.getRow(squ_2))
        // add bonus for twinpawns
        score += side_sign * twin_pawns[row_side];
      else if (row == SquareHelper.getRow(squ_2
          - Direction.pawnDirection(side).offset)) {
        // add bonus for covered pawns
        // TODO: maybe dont increase bonus for pawns covered
        // by 2 pawns
        covered = true;
        score += side_sign * covered_pawns[row_side];
      }
    }

  }

  if (isolated == true)
    score += side_sign * ISOLATED_PAWN;

  // check if a pawn is passed
  passed = true;
  for (int squ_2 : squares_pawn_opp) {
    col_2 = SquareHelper.getColumn(squ_2);
```

```
        if (col == col_2 || col == col_2 + 1 || col == col_2 - 1) {
          passed = false;
          break;
        }

      }
      if (passed == true) {
        // check if a passed pawn is blocked
        for (int squ_2 : squares_pawn_opp) {
          if (squ_1 + Direction.pawnDirection(side).offset == squ_2) {
            score += side_sign * blocked_passed_pawn[row_side];
            break;
          }
        }

        // check if a passed pawn is covered by a king (the king
        // should be in front of the pawn)
        for (Direction dir : Direction
            .pawnCapturingDirections(side))
          if (squ_1 + dir.offset == position.getKingPos(side))
            score += side_sign
                * passed_pawn_with_king[row_side];

        // add the bonus for a passed pawn
        score += side_sign * passed_pawn[row_side];

        // additional bonus for covered passed pawn
        if (covered == true
            && row == SquareHelper.getRowForSide(side, 7))
          score += side_sign * COVERED_PASSED_7TH_PAWN;

        // if a rook is behind a passed pawn
        // TODO: check if it better do add the bonus is a rook is on
        // the same line (behind the pawn)
        if (position.getPieceFromBoard(squ_1
            - Direction.pawnDirection(side).offset) == Piece.ROOK
            && position.getSideFromBoard(squ_1
                - Direction.pawnDirection(side).offset) == side)
          score += side_sign * PASSED_ROOK_SUPPORT;
      }

    }

  }
  return score;
}
```

```
  /**
   * draft of king's position evaluation function.
   *
   * @param position
   *           the current position
   * @return the score
   */
  private int evalKingPos(IPosition position) {
    int score = 0;
    int count_fig = position.getNumberOfPiecesByColor(Side.WHITE)
        + position.getNumberOfPiecesByColor(Side.BLACK);
    if (count_fig > ENDGAME_THRESHOLD)
      for (Side side : Side.values()) {
        int side_sign = Side.getSideSign(side);
        int row_1 = SquareHelper.getRowForSide(side, 1);
        if (side == Side.WHITE)
          score += side_sign
              * king_positions_w[square_to_array_index[position
                  .getKingPos(side)]];
        else
          score += side_sign
              * king_positions_b[square_to_array_index[position
                  .getKingPos(side)]];

        // TODO: this dont work as it should... needs to be fixed (only
        // should
        // give penalty if during search castling is lost)
        if (!position.canCastle(SquareHelper.getSquare(row_1, 3))
            || !position
                .canCastle(SquareHelper.getSquare(row_1, 7)))
          score += side_sign * CASTLING_LOSS;

      }

    return score;
  }
}
```

Flag.java

```
package mitzi;

/**
```

```
 * The flags for the entries in the Transposition Table (PositionCache).
 *
 */
public enum Flag {
  EXACT, LOWERBOUND, UPPERBOUND
}
```

GameState.java

```
package mitzi;

import java.util.ArrayList;

public class GameState {

  /**
   * the actual position of the current game state
   */
  private IPosition position;

  /**
   * the history of played moves
   */
  private ArrayList<IMove> history = new ArrayList<IMove>();


  /**
   * The number of the full move. It starts at 1, and is incremented after
   * Black's move.
   */
  private int full_move_clock;

  private class GameClock {
    // TODO study UCI time management
  }

  /**
   * creates a new Game with initial position.
   */
  public GameState() {
    position = new Position();
    setToInitial();
  }
```

```
/**
 * sets the current game to the initial state.
 */
public void setToInitial() {
  position.setToInitial();
  full_move_clock = 1;
}

/**
 * sets the current game to the position of the given fen string
 *
 * @param fen
 *          the position in fen notation
 */
public void setToFEN(String fen) {
  position = new Position();
  position.setToFEN(fen);

  String[] fen_parts = fen.split(" ");
  // set half move clock
  position.setHalfMoveClock(Integer.parseInt(fen_parts[4]));
  // set full move clock
  full_move_clock = Integer.parseInt(fen_parts[5]);
}

/**
 * Do the given move and update half_move_clock, full_move_clock and
 * history. It is checked, if the move is valid or not.
 *
 * @param move
 *          the given move
 */
public void doMove(IMove move) {
  if (position.isPossibleMove(move)) {
    position = position.doMove_copy(move);
    /*if (mova.resets_half_move_clock) {
      half_move_clock = 0;
    }*/
    if (position.getActiveColor() == Side.BLACK) {
      full_move_clock++;
    }
    history.add(move);

  } else {
    throw new IllegalArgumentException("INVALID MOVE");
  }
}

/**
 * @return the actual position of the game
 */
public IPosition getPosition() {
  return position;
}

/**
 * creates the fen string of the actual board.
 */
@Override
public String toString() {
  StringBuilder fen = new StringBuilder();

  fen.append(position.toString());
  fen.append(" ");

  // halfmove clock
  fen.append(position.getHalfMoveClock());
  fen.append(" ");

  // fullmove clock
  fen.append(full_move_clock);

  return fen.toString();
}

/**
 * This is the number of halfmoves since the last pawn advance or capture.
 * This is used to determine if a draw can be claimed under the fifty-move
 * rule.
 *
 * @return number of halfmoves since the last pawn advance or capture
 */
public int getHalfMoveClock() {
  return position.getHalfMoveClock();
}

/**
 * The number of the full move. It starts at 1, and is incremented after
 * Black's move.
 *
 * @return number of the full move
 */
```

```
  public int getFullMoveClock() {
    return full_move_clock;
  }

  /**
   * return all previous played moves.
   *
   * @return returns a list of all played moves.
   */
  public ArrayList<IMove> getHistory() {
    return history;
  }
}
```

IrreversibleMoveStack.java

```
package mitzi;

import java.util.LinkedList;

/**
 * This class represents a stack, storing the information, which cannot be
 * reverted only with a given move. It is implemented as a LinkedList
    containing
 * a class which stores the half move clock, the castling, the en passant
    target
 * and the captured piece. (en passant captures does not count as capture).
    The
 * elements should be accessed via irr_move_info.removeLast();
 */
public class IrreversibleMoveStack {

  static public class MoveInfo {

    int half_move_clock;
    int[] castling = new int[4];
    int en_passant_square;
    Piece capture;
    Boolean is_check;

  }

  /**
   * the stack containing the information
   */
  static public LinkedList<MoveInfo> irr_move_info = new
     LinkedList<MoveInfo>();

  private IrreversibleMoveStack() {
  }

  /**
   * add a new entry.
   *
   * @param half_move_clock
   *           the old half move clock
   * @param castling
   *           the castling array
   * @param en_passant_square
   *           the en passant target square
   * @param capture
   *           the piece, which got captured (null if no capture)
   */
  static public void addInfo(int half_move_clock, int[] castling,
      int en_passant_square, Piece capture, Boolean is_check) {

    MoveInfo inf = new MoveInfo();
    System.arraycopy(castling, 0, inf.castling, 0, 4);
    inf.en_passant_square = en_passant_square;
    inf.half_move_clock = half_move_clock;
    inf.capture = capture;
    inf.is_check = is_check;

    irr_move_info.addLast(inf);
  }

}
```

KillerMoves.java

```
package mitzi;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

/**
```

```
 * this class saves for each ply a certain number (e.g. 2) of moves
 * (killermoves), which causes an alpha-beta cutoff. If more moves are saved,
 * that allowed, then they get deleted in the order they are saved (like
 * FIFO).This should improve the move ordering.
 *
 */
public class KillerMoves {

  /**
   * a map from a ply to the killermoves.
   */
  private static Map<Integer, LinkedList<IMove>> killer_moves = new
      HashMap<Integer, LinkedList<IMove>>(
      35);

  /**
   * number of killermoves saved
   */
  private static int MAX_SIZE = 2;

  KillerMoves() {
  };

  /**
   * returns for a given ply the killer moves, note that it should be checked
   * if the move is legal.
   *
   * @param ply
   *          the plys from root node
   * @return a list of killer moves.
   */
  static LinkedList<IMove> getKillerMoves(int ply) {
    LinkedList<IMove> k_m = killer_moves.get(ply);
    if (k_m == null)
      k_m = new LinkedList<IMove>();
    return k_m;
  }

  /**
   * add a new killermove, if more moves are saved than MAX_SIZE, the first
   * killermove got removed.
   *
   * @param ply
   *          depth in the search tree
   * @param move
   *          the move to be added
   */
  static void addKillerMove(int ply, IMove move) {
    LinkedList<IMove> k_m = killer_moves.get(ply);
    if (k_m == null)
      k_m = new LinkedList<IMove>();
    if (k_m.size() == MAX_SIZE)
      k_m.iterator().remove();

    k_m.add(move);
  }

  /**
   * add a new killermove, if more moves are saved than MAX_SIZE, the first
   * killermove got removed.
   *
   * @param ply
   *           depth in the search tree
   * @param move
   *           the move to be added
   * @param entry
   *           if available the old entry can be used for faster update. This
   *           should be a reference to the old element.
   */
  static void addKillerMove(int ply, IMove move, List<IMove> entry) {
    if (entry.size() == MAX_SIZE)
      entry.iterator().remove();
    entry.add(move);
  }

  /**
   * updates the killermoves after the best move was found, i.e. all moves are
   * shifted from depth -> depth -2
   */
  static void updateKillerMove() {
    for (int i = 2; killer_moves.containsKey(i); i++)
      killer_moves.put(i - 2, killer_moves.get(i));

  }
}
```

MateScores.java

```
package mitzi;
```

```
/**
 * contains the scores for mate positions
 */
public final class MateScores {

  private MateScores() {
  }

  public static final int POS_INF = +30767;
  public static final int NEG_INF = -30767;

}
```

MitziBrain.java

```
package mitzi;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import java.util.Timer;
import java.util.TimerTask;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

import static mitzi.MateScores.*;
import mitzi.UCIReporter.InfoType;

/**
 * This class implements the AI of Mitzi. The best move is found using the
 * negamax algorithms with Transposition tables. The class regularly sends
 * information about the current search, including nodes per second ("nps"),
    the
 * filling of the Transposition Table ("hashfull") and the current searched
    move
 * on top-level. The board evaluation is moved to a separate class
 * BoardAnalyzer.
 *
 */
public class MitziBrain implements IBrain {

  /**
 * maximal number of threads
 */
private static final int THREAD_POOL_SIZE = 1;

/**
 * unit for time management
 */
private static final TimeUnit THREAD_TIMEOUT_UNIT = TimeUnit.MILLISECONDS;

/**
 * timeout for thread shutdown
 */
private static final int THREAD_TIMEOUT = 1000;

/**
 * upper limit for evaluation time
 */
private int maxEvalTime;

/**
 * the currently best result
 */
private AnalysisResult result;

/**
 * the executor for the tasks
 */
private ExecutorService exe;

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

private class PositionEvaluator implements Runnable {

  private final IPosition position;
  private final int searchDepth;

  public PositionEvaluator(final IPosition position, final int depth) {
    this.position = position;
    this.searchDepth = depth;
  }

  @Override
  public void run() {
```

```
try {
  // Parameters for aspiration windows
  int alpha = NEG_INF; // initial value
  int beta = POS_INF; // initial value
  int asp_window = 25; // often 50 or 25 is used
  int factor = 2; // factor for increasing if out of bounds

  // iterative deepening
  for (int current_depth = 1; current_depth <= searchDepth;
      current_depth++) {
    table_counter = 0;
    BoardAnalyzer.table_counter = 0;

    result = negaMax(position, current_depth, current_depth,
        alpha, beta);
    position.updateAnalysisResult(result);

    if (result.score == POS_INF || result.score == NEG_INF) {
      break;
    }

    // If Value is out of bounds, redo search with larger
    // bounds, but with the same variation tree
    if (result.score <= alpha) {
      alpha -= factor * asp_window;
      current_depth--;
      UCIReporter
          .sendInfoString("Boards found: "
              + (table_counter + BoardAnalyzer.table_counter));
      continue;
    } else if (result.score >= beta) {
      beta += factor * asp_window;
      current_depth--;
      UCIReporter
          .sendInfoString("Boards found: "
              + (table_counter + BoardAnalyzer.table_counter));
      continue;
    }

    alpha = result.score - asp_window;
    beta = result.score + asp_window;

    UCIReporter.sendInfoString("Boards found: "
        + (table_counter + BoardAnalyzer.table_counter));
  }
```

```
      } catch (InterruptedException e) {
      }

    }

    @Override
    public String toString() {
      return position.toString();
    }

  }

  /**
   * counts the number of evaluated board
   */
  private long eval_counter;

  /**
   * counts the number of found boards in the transposition table.
   */
  private long table_counter;

  /**
   * the board analyzer for board evaluation
   */
  private IPositionAnalyzer board_analyzer = new BoardAnalyzer();

  /**
   * the current time.
   */
  private long start_mtime = System.currentTimeMillis();

  private Timer timer;

  @Override
  public void set(GameState game_state) {
    this.game_state = game_state;
    this.eval_counter = 0;
    this.table_counter = 0;
  }

  /**
   * @return the time, which passes since start_mtime
   */
  private long runTime() {
    return System.currentTimeMillis() - start_mtime;
```

```
}

/**
 * Sends updates about evaluation status to UCI GUI, namely the number of
 * searched board per second and the size of the Transposition Table in
 * permill of the maximal size.
 *
 */
class UCIUpdater extends TimerTask {
  private long old_mtime;
  private long old_eval_counter;
  private long old_eval_counter_seldepth;

  @Override
  public void run() {
    long mtime = System.currentTimeMillis();

    long eval_span_0 = eval_counter - old_eval_counter;
    long eval_span_sel = BoardAnalyzer.eval_counter_seldepth
        - old_eval_counter_seldepth;
    long eval_span = eval_span_0 + eval_span_sel;

    if (old_mtime != 0) {
      long time_span = mtime - old_mtime;
      UCIReporter.sendInfoNum(InfoType.NPS, eval_span * 1000
          / time_span);

      UCIReporter.sendInfoNum(InfoType.HASHFULL,
          ResultCache.getHashfull());
    }

    old_mtime = mtime;
    old_eval_counter += eval_span_0;
    old_eval_counter_seldepth += eval_span_sel;

  }
}

/**
 * NegaMax with Alpha Beta Pruning and Transposition Tables
 *
 * @see <a
 *
    href="http://en.wikipedia.org/wiki/Negamax#NegaMax_with_Alpha_Beta_Pruning_and_Transp
 *     with Alpha Beta Pruning and Transposition Tables</a>
 * @param position
 *          the position to evaluate
 * @param total_depth
 *          the total depth to search
 * @param depth
 *          the remaining depth to search
 * @param alpha
 *          the alpha value
 * @param beta
 *          the beta value
 * @return returns the result of the evaluation, stored in the class
 *        AnalysisResult
 *
 * @throws InterruptedException
 */
private AnalysisResult negaMax(IPosition position, int total_depth,
    int depth, int alpha, int beta) throws InterruptedException {

  if (Thread.interrupted()) {
    throw new InterruptedException();
  }
  //
    ---------------------------------------------------------------------------------------
  // whose move is it?
  Side side = position.getActiveColor();
  int side_sign = Side.getSideSign(side);

  //
    ---------------------------------------------------------------------------------------
  int alpha_old = alpha;

  // Cache lookup (Transposition Table)
  AnalysisResult entry = ResultCache.getResult(position);
  if (entry != null && entry.plys_to_eval0 >= depth) {
    table_counter++;
    if (entry.flag == Flag.EXACT)
      return entry.tinyCopy();
    else if (entry.flag == Flag.LOWERBOUND)
      alpha = Math.max(alpha, entry.score * side_sign);
    else if (entry.flag == Flag.UPPERBOUND)
      beta = Math.min(beta, entry.score * side_sign);

    if (alpha >= beta)
      return entry.tinyCopy();
  }
```

```
//
    -----------------------------------------------------------------------------
// base of complete tree search
if (depth == 0) {
  // position is a leaf node
  return board_analyzer.evalBoard(position, alpha, beta);
}

//
    -----------------------------------------------------------------------------
// generate moves
List<IMove> moves = position.getPossibleMoves();

// check for mate and stalemate
if (moves.isEmpty()) {
  eval_counter++;
  if (position.isCheckPosition()) {
    return new AnalysisResult(NEG_INF * side_sign, false, false, 0,
        0, Flag.EXACT);
  } else {
    return new AnalysisResult(0, true, false, 0, 0, Flag.EXACT);
  }
}
//
    -----------------------------------------------------------------------------
// Sort the moves:
ArrayList<IMove> ordered_moves = new ArrayList<IMove>(40);
ArrayList<IMove> remaining_moves = new ArrayList<IMove>(40);
BasicMoveComparator move_comparator = new BasicMoveComparator(position);

// Get Killer Moves:
List<IMove> killer_moves = KillerMoves.getKillerMoves(total_depth
    - depth);

// if possible use the moves from Position cache as the moves with
// highest priority
if (entry != null) {
  ordered_moves.addAll(entry.best_moves);

  for (IMove k_move : killer_moves)
    if (position.isPossibleMove(k_move)
        && !ordered_moves.contains(k_move))
      ordered_moves.add(k_move);

} else {
  // Killer_moves have highest priority
```

```
    for (IMove k_move : killer_moves)
      if (position.isPossibleMove(k_move))
        ordered_moves.add(k_move);
  }
  // add the remaining moves and sort them using a basic heuristic
  for (IMove move : moves)
    if (!ordered_moves.contains(move))
      remaining_moves.add(move);

  Collections.sort(remaining_moves,
      Collections.reverseOrder(move_comparator));
  ordered_moves.addAll(remaining_moves);
  //
      ----------------------------------------------------------------------------------------------------

  if (entry != null && entry.plys_to_eval0 < depth)
    entry.best_moves.clear();

  // create new AnalysisResult and parent
  AnalysisResult new_entry = null, parent = null;
  if (entry == null)
    new_entry = new AnalysisResult(0, null, false, 0, 0, null);

  int best_value = NEG_INF; // this starts always at negative!

  int i = 0;
  // alpha beta search
  for (IMove move : ordered_moves) {

    // output currently searched move to UCI
    if (depth == total_depth && total_depth >= 6)
      UCIReporter.sendInfoCurrMove(move, i + 1);

    position.doMove(move);
    AnalysisResult result = negaMax(position, total_depth, depth - 1,
        -beta, -alpha);
    position.undoMove(move);

    int negaval = result.score * side_sign;

    // better variation found
    if (negaval > best_value || parent == null) {

      best_value = negaval;

      // update cache entry
      if (entry != null && entry.plys_to_eval0 < depth)
       entry.best_moves.add(move);
      if (entry == null)
       new_entry.best_moves.add(move);

      // update AnalysisResult
      byte old_seldepth = (parent == null ? 0
        : parent.plys_to_seldepth);
      parent = result; // change reference
      parent.best_move = move;
      parent.plys_to_eval0 = (byte) depth;
      if (best_value != POS_INF) {
       parent.plys_to_seldepth = (byte) Math.max(old_seldepth,
           parent.plys_to_seldepth);
      }

      // output to UCI
      // boolean truly_better = negaval > best_value;
      if (depth == total_depth) { // && truly_better) {
       position.updateAnalysisResult(parent);
       UCIReporter.sendInfoPV(game_state.getPosition(), runTime());
      }
    }

    // alpha beta cutoff
    alpha = Math.max(alpha, negaval);
    if (alpha >= beta) {
      // set also KillerMove:
      if (!killer_moves.contains(move))
       KillerMoves.addKillerMove(total_depth - depth, move,
           killer_moves);
      break;
    }

    i++;
  }

  //
      --------------------------------------------------------------------------
  // Transposition Table Store;
  if (best_value <= alpha_old)
    parent.flag = Flag.UPPERBOUND;
  else if (best_value >= beta)
    parent.flag = Flag.LOWERBOUND;
  else
    parent.flag = Flag.EXACT;
```

```
    if (entry != null && entry.plys_to_eval0 < depth) {
      entry.tinySet(parent);
      Collections.reverse(entry.best_moves);
    }

    if (entry == null) {
      new_entry.tinySet(parent);
      Collections.reverse(new_entry.best_moves);
      ResultCache.setResult(position, new_entry);
    }

    return parent;

  }

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

    // note, the variable seachMoves is currently unused, this feature is
    // not yet implemented!

    // set up threading
    timer = new Timer();
    exe = Executors.newFixedThreadPool(THREAD_POOL_SIZE);

    // store the actual position
    IPosition position = game_state.getPosition();

    int max_depth;

    // set parameters for searchtime and searchdepth
    if (movetime == 0 && maxMoveTime == 0) {
      maxEvalTime = 60 * 60 * 1000; // 1h
      max_depth = searchDepth;
    } else if (movetime == 0 && infinite == false) {
      maxEvalTime = maxMoveTime;
      max_depth = searchDepth;
    } else if (movetime == 0 && infinite == true) {
      maxEvalTime = maxMoveTime;
      max_depth = 200;
    } else if (maxMoveTime == 0) {
      maxEvalTime = movetime;
      max_depth = 200; // this can never be reached :)
    } else if (infinite == true) {
      maxEvalTime = maxMoveTime;
      max_depth = 200; // this can never be reached :)
    } else {
      maxEvalTime = Math.min(movetime, maxMoveTime);
      max_depth = searchDepth;
    }

    timer.scheduleAtFixedRate(new UCIUpdater(), 1000, 5000);
    start_mtime = System.currentTimeMillis();

    // reset the result
    result = null;

    // create a new task
    PositionEvaluator evaluator = new PositionEvaluator(position, max_depth);

    // execute the task
    exe.execute(evaluator);

    return wait_until();
  }

  /**
   * stops all active threads if mitzi is running out of time
   *
   * @return the best move
   */
  public IMove wait_until() {

    exe.shutdown();

    // wait for termination of execution
    try {
      if (exe.awaitTermination(maxEvalTime, THREAD_TIMEOUT_UNIT)) {
        UCIReporter.sendInfoString("task completed");
      } else {
        UCIReporter.sendInfoString("forcing task shutdown");
        exe.shutdownNow();
        exe.awaitTermination(THREAD_TIMEOUT, TimeUnit.SECONDS);
      }
    } catch (InterruptedException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }

    // shut down timers and update killer moves
    timer.cancel();
    UCIReporter.sendInfoPV(game_state.getPosition(), runTime());
    KillerMoves.updateKillerMove();

    // if no best_move has been found yet, choose any
    if (result == null) {
      List<IMove> possibleMoves = game_state.getPosition().getPossibleMoves();
      int randy = new Random().nextInt(possibleMoves.size());
      return possibleMoves.get(randy);
    }

    // return the best move of the last completely searched tree
    return result.best_move;
  }

  @Override
  public IMove stop() {
    // shut down immediately
    exe.shutdownNow();

    // shut down timers and update killer moves
    timer.cancel();
    UCIReporter.sendInfoPV(game_state.getPosition(), runTime());
    KillerMoves.updateKillerMove();

    // return the best move of the last completely searched tree
    if (result == null)
      return null; // this should never happen

    return result.best_move;
  }
}
```

MitziEngine.java

```
package mitzi;

import java.io.*;

public class MitziEngine {

  private static BufferedReader reader;
  private static String cmd;
  private static GameState game_state;
```

```
/** time management variables */
public static final int DEFAULT_WTIME = 1000;
public static final int DEFAULT_BTIME = 1000;
public static final int DEFAULT_WINC = 0;
public static final int DEFAULT_BINC = 0;
public static final int DEFAULT_TOGO = 40;

public static void main(String[] args) {

  reader = new BufferedReader(new InputStreamReader(System.in));
  printGreeting();
  try {
    getCmd();
  } catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
  }

}

public static void printGreeting() {
  System.out.println("============= MITZI 0.1 =============");
  System.out.println("==== please enter \"uci\" to start ====");
}

public static void uci() throws IOException {
  int movetime;
  int maxMoveTime;
  int searchDepth;
  int wtime = 0;
  int btime = 0;
  int winc = 0;
  int binc = 0;
  int togo = 0;
  boolean infinite = false;
  IBrain brain = new MitziBrain();
  game_state = new GameState();

  System.out.println("id name Mitzi 0.1");
  System.out.println("id author Christoph Hofer, Stefan Lew");
  // System.out
  // .println("option name Ponder type check default false");
  // System.out
  // .println("option name Evaluation Table type spin default 8 min 1 max
     64");
```

```
// System.out
// .println("option name Pawn Table type spin default 8 min 1 max 64");
//System.out
// .println("option name Hash type spin default 64 min 1 max 4096");
System.out.println("uciok");

while (true) {
  cmd = reader.readLine();
  if (cmd.startsWith("quit")) {
    System.exit(0);
    // } else if (cmd.equals("eval_dump_white")) {
    // Evaluation2.getEval(Global.COLOUR_WHITE, -Global.MATE_SCORE,
    // Global.MATE_SCORE, 0);
    // Evaluation2.printEvalTerms();
    // } else if (cmd.equals("eval_dump_black")) {
    // Evaluation2.getEval(Global.COLOUR_BLACK, -Global.MATE_SCORE,
    // Global.MATE_SCORE, 0);
    // Evaluation2.printEvalTerms();
  } else if ("isready".equals(cmd)) {
    System.out.println("readyok");
    // } else if (cmd.startsWith("perft")) {
    // cmd = cmd.substring(5);
    // cmd = cmd.trim();
    // int depth = Integer.parseInt(cmd.substring(0));
    // theSearch.PerftTest(depth);
    // } else if (cmd.startsWith("divide")) {
    // cmd = cmd.substring(6);
    // cmd = cmd.trim();
    // int depth = Integer.parseInt(cmd.substring(0));
    // theSearch.Divide(depth);
  }
  if (cmd.startsWith("position")) {
    if (cmd.indexOf(("startpos")) != -1) {
      game_state.setToInitial();
      int mstart = cmd.indexOf("moves");
      if (mstart > -1) {
        String[] moves = cmd.substring(mstart + 6).split(" ");
        for (int i = 0; i < moves.length; i++) {
          Move mv = new Move(moves[i]);
          game_state.doMove(mv);
        }
      }
    } else {
      int mstart = cmd.indexOf("moves");
      if (mstart > -1) {
        String fen = cmd.substring(cmd.indexOf("fen") + 4,
            mstart - 1);
        game_state.setToFEN(fen);
        String[] moves = cmd.substring(mstart + 6).split(" ");
        for (int i = 0; i < moves.length; i++) {
          Move mv = new Move(moves[i]);
          game_state.doMove(mv);
        }
      } else {
        String fen = cmd.substring(cmd.indexOf("fen") + 4,
            cmd.length());
        game_state.setToFEN(fen);
      }
    }
  /*} else if (cmd.startsWith("setoption")) {
    int index = cmd.indexOf("Hash");
    if (index != -1) {
      index = cmd.indexOf("value");
      cmd = cmd.substring(index + 5);
      cmd = cmd.trim();
      int hashSize = Integer.parseInt(cmd.substring(0));
      ResultCache.setSize(hashSize);
      // } else if (cmd.indexOf("Evaluation Table") != -1) {
      // index = cmd.indexOf("value");
      // cmd = cmd.substring(index + 5);
      // cmd = cmd.trim();
      // int evalSize = Integer.parseInt(cmd.substring(0));
      // Global.EvalHASHSIZE = evalSize * 131072;
      // Evaluation2.reSizeEvalHash();
      // System.out.println("info string evalHash is " +
      // evalSize);
      // } else if (cmd.indexOf("Pawn Table") != -1) {
      // index = cmd.indexOf("value");
      // cmd = cmd.substring(index + 5);
      // cmd = cmd.trim();
      // int evalSize = Integer.parseInt(cmd.substring(0));
      // Global.PawnHASHSIZE = evalSize * 43960;
      // Evaluation2.reSizePawnHash();
      // System.out.println("info string pawnHash is " +
      // evalSize);
    } else {
      System.out.println("info string command not recognized");
    }*/
  } else if (cmd.startsWith("go")) {
    movetime = 0;
    maxMoveTime = 0;
    searchDepth = 0;
```

```
infinite = false;
if (cmd.indexOf("depth") != -1) {
  int index = cmd.indexOf("depth");
  cmd = cmd.substring(index + 5);
  cmd = cmd.trim();
  searchDepth = Integer.parseInt(cmd.substring(0));
  movetime = 9999999;
  maxMoveTime = movetime;
} else if (cmd.indexOf("movetime") != -1) {
  int index = cmd.indexOf("movetime");
  cmd = cmd.substring(index + 8);
  cmd = cmd.trim();
  movetime = Integer.parseInt(cmd.substring(0));
  maxMoveTime = movetime;
  searchDepth = 40;
} else if (cmd.indexOf("infinite") != -1) {
  infinite = true;
  searchDepth = 40;
  movetime = 1000;
  maxMoveTime = movetime;
} else {
  searchDepth = 40;
  String temp;
  int index = cmd.indexOf("wtime");
  if (index == -1) {
    wtime = DEFAULT_WTIME;
  } else {
    temp = cmd.substring(index + 5).trim();
    wtime = Integer.parseInt(temp.substring(0,
        temp.indexOf(" ")));
  }
  index = cmd.indexOf("btime");
  if (index == -1) {
    btime = DEFAULT_BTIME;
  } else {
    temp = cmd.substring(index + 5).trim();
    if (temp.indexOf(" ") != -1) {
      btime = Integer.parseInt(temp.substring(0,
          temp.indexOf(" ")));
    } else {
      btime = Integer.parseInt(temp);
    }
    index = cmd.indexOf("winc");
    if (index == -1) {
      winc = DEFAULT_WINC;
    } else {
```

```
    temp = cmd.substring(index + 4).trim();
    if (temp.indexOf(" ") != -1) {
      winc = Integer.parseInt(temp.substring(0,
          temp.indexOf(" ")));
    } else {
      winc = Integer.parseInt(temp);
    }
  }
  index = cmd.indexOf("binc");
  if (index == -1) {
    binc = DEFAULT_BINC;
  } else {
    temp = cmd.substring(index + 4);
    temp = temp.trim();
    if (temp.indexOf(" ") != -1) {
      binc = Integer.parseInt(temp.substring(0,
          temp.indexOf(" ")));
    } else {
      binc = Integer.parseInt(temp);
    }
  }
  index = cmd.indexOf("movestogo");
  if (index == -1) {
    togo = DEFAULT_TOGO;
  } else {
    temp = cmd.substring(index + 9).trim();
    togo = Integer.parseInt(temp);
  }
  if (game_state.getPosition().getActiveColor() == Side.BLACK) {
    movetime = Math.max(0, (btime / togo + binc));
    // reduce the move time a little, as most of the
    // time we will be extending this time to find the
    // first move of the last iteration
    movetime = (int) (((double) movetime) * 0.85);
    int maxTimeLimit = (int) (((double) btime + (double) binc) *
        0.40);
    maxMoveTime = Math.min(movetime * 3, maxTimeLimit);
  } else {
    movetime = Math.max(0, (wtime / togo + winc));
    // reduce the move time a little, as most of the
    // time we will be extending this time to find the
    // first move of the last iteration
    movetime = (int) (((double) movetime) * 0.85);
    int maxTimeLimit = (int) (((double) wtime + (double) winc) *
        0.40);
    maxMoveTime = Math.min(movetime * 3, maxTimeLimit);
```

```
          }
          // on the last move before the time is increased, the
          // move time will be higher than the maxMoveTime,
          // so we adjust the maxMoveTime to be equal to the
          // movetime
          if (movetime > maxMoveTime) {
            maxMoveTime = movetime;
          }
        }
      }
      brain.set(game_state);
      IMove bestmove;
      bestmove = brain.search(movetime, maxMoveTime, searchDepth,
          infinite, null);
      System.out.println("bestmove " + bestmove);

    } else if (cmd.equals("ucinewgame")) {
      brain = new MitziBrain();
      game_state = new GameState();
    }
  }
}

public static void getCmd() throws IOException {
  while (true) {
    cmd = reader.readLine();
    if (cmd.equals("uci")) {
      uci();
      break;
    } else if (cmd.startsWith("quit")) {
      System.exit(0);
    }
  }
}

}
```

ResultCache.java

```
package mitzi;

import java.util.LinkedHashMap;
import java.util.Map;
```

```
/**
 * After creating a new <code>AnalysisResult</code> instance, use this class
    to
 * cache it for later lookup of moves, score, etc. The AnalysisResults are
 * indexed by the HashCode of the corresponding position, therefore it can
 * happen, that different AnalysisResults have the same HashCode. In such a
 * case, the old values get overridden. The AnalysisResults store a different
 * hashvalue to reduce the probability of using a wrong AnalysisResult, if two
 * positions have the same HashCode.
 *
 */
public class ResultCache {

  private static final int MAX_ENTRIES = 600000;

  /**
   * A map from the Position's <code>hashCode</code> to the AnalysisResult.
   * The size of the table is limited with <code>MAX_ENTRIES</code>
   */
  private static LinkedHashMap<Integer, AnalysisResult> position_cache = new
      LinkedHashMap<Integer, AnalysisResult>(
      MAX_ENTRIES + 1, 1) {

    private static final long serialVersionUID = 4582735742585308092L;

    protected boolean removeEldestEntry(
        Map.Entry<Integer, AnalysisResult> eldest) {
      return size() > MAX_ENTRIES;
    }
  };

  /**
   * Cannot be instantiated. For access to the static cache use
   * <code>ResultCache.getPosition(p)</code>.
   */
  private ResultCache() {
  }

  /**
   * Looks up a <code>Position</code> in the cache and returns the saved value
   * if found and with coinciding second hashvalue. otherwise null.
   *
   * @param lookup
   *          the <code>Position</code> to look up in the cache
   * @return a previously cached <code>AnalysisResult</code> if available,
   *        null otherwise.
   */
  public static AnalysisResult getResult(IPosition lookup) {
    int hash = lookup.hashCode();
    AnalysisResult ce = position_cache.get(hash);
    if (ce == null || lookup.hashCode2() != ce.hashvalue)
      return null;
    else
      return ce;
  }

  /**
   * stores a AnalysisResult corresponding to a Position. The second
       hashvalue is automatically set here.
   * @param pos the position corresponding to the AnalysisResult
   * @param ce the AnalysisResult
   */
  public static void setResult(IPosition pos, AnalysisResult ce) {
    ce.hashvalue = pos.hashCode2();
    int hash = pos.hashCode();
    position_cache.put(hash, ce);

  }

  /**
   *
   * @return the number of stored results in this cache
   */
  public static int size() {
    return position_cache.size();
  }

  /**
   * @return the hash is x permill full
   */
  public static int getHashfull() {
    return (int) ((double) position_cache.size() / MAX_ENTRIES * 1000);
  }
}
```

UCIReporter.java

```
package mitzi;

import static mitzi.MateScores.*;
```

```
public final class UCIReporter {

  /**
   * the engine should send these infos regularly
   *
   * DEPTH: search depth in plies
   *
   * NODES: number of nodes searched
   *
   * NPS: number of nodes per second searched
   *
   * HASHFULL: the hash is x permill full
   */
  public static enum InfoType {
    DEPTH("depth"), NODES("nodes"), NPS("nps"), HASHFULL("hashfull");

    public String string;

    InfoType(String string) {
      this.string = string;
    }
  }

  private static String last_pv = "";

  private UCIReporter() {
  };

  /**
   * Send debugging messages to the GUI.
   *
   * @param string
   *           the message to be displayed
   */
  public static void sendInfoString(String string) {
    System.out.println("info string " + string);
  }

  /**
   * Send information about search depth, number of nodes searched and number
   * of nodes searched per second to the GUI.
   *
   * @param type
   *           one of UCIReporter.InfoType
   * @param eval_counter
```

```
 *           the integer value to be sent
 */
public static void sendInfoNum(InfoType type, long eval_counter) {
  System.out.println("info " + type.string + " " + eval_counter);
}

/**
 * Send information about the currently searched move to the GUI.
 *
 * @param move
 *           currently searching this IMove
 * @param move_number
 *           currently searching move number n, for the first move n should
 *           be 1 not 0.
 */
public static void sendInfoCurrMove(IMove move, int move_number) {
  System.out.println("info currmove " + move + " currmovenumber "
      + move_number);
}



/**
 * The Principal variation (PV) is a sequence of moves that programs
 * consider best and therefore expect to be played. Also all infos belonging
 * to the PV should be sent together.
 *
 * @param position
 *           a Position with an AnalysisResult
 * @param time
 *           the time searched in ms
 */
public static void sendInfoPV(IPosition position, long time) {
  AnalysisResult result = position.getAnalysisResult();
  if (result == null)
    return;

  StringBuilder pv = new StringBuilder();

  if (result.score == NEG_INF && position.getActiveColor() == Side.WHITE
      || result.score == POS_INF
      && position.getActiveColor() == Side.BLACK) {
    pv.append("info score mate -"
        + ((result.plys_to_eval0 + result.plys_to_seldepth + 1) / 2)
        + " depth " + result.plys_to_eval0 + " seldepth "
        + (result.plys_to_seldepth + result.plys_to_eval0) + " pv");
    } else if (result.score == NEG_INF
        && position.getActiveColor() == Side.BLACK
        || result.score == POS_INF
        && position.getActiveColor() == Side.WHITE) {
      pv.append("info score mate "
          + ((result.plys_to_eval0 + result.plys_to_seldepth + 1) / 2)
          + " depth " + result.plys_to_eval0 + " seldepth "
          + (result.plys_to_seldepth + result.plys_to_eval0) + " pv");
    } else {
      pv.append("info score cp " + result.score + " depth "
          + result.plys_to_eval0 + " seldepth "
          + (result.plys_to_seldepth + result.plys_to_eval0) + " pv");
    }

    for (IMove move : result.getPV(position, result.plys_to_eval0)) {
      pv.append(" " + move);
    }

    String new_pv = pv.toString();
    if (!last_pv.equals(new_pv)) {
      System.out.print(new_pv);
      System.out.println(" time " + time);
      last_pv = new_pv;
    }
  }
}
```