Alquerque

Danny Nicolai Larsen, Mikkel Brix Nielsen & Steffen Bach

December 9, 2021

Contents

CONTENTS

PUT MAIN TEXT HERE

1 Appendix

1.1 Move class

```
public class Move {
    private int from;
    private int to;
    /**
     * Creates a new move with given origin and destination.
     * @param from the place to move the piece from.
     * @param to the place to move the peiece to.
     */
    public Move(int from, int to) {
        this.from = from;
        this.to = to;
    }

    /**
     * Returns the origin of this move.
     * @return the origin of this move.
     */
    public int from() {
        return from;
    }

    /**
     * Returns the destination of this move.
     * @return the destination of this move
     */
    public int to() {
        return to;
    }
}
```

1.2 Board class

```
import java.util.ArrayList;

public class Board {
    private char[] board;
    private int turn;
    private boolean isWhite;
    private boolean isGameDone;
    private static int finishedGames = 0;
    private static final char EMPTY = ' ';

    /**
     * Creates a new Alquerque board in the starting state:
     * each player has twelve pieces in their original position, and it is white's turn.
     */
    public Board() {
```

```
16          turn = 1;
17          board = new char[26];
18          for (int i = 1; i < 26; i++) {
19              if (i < 13)
20                  board[i] = 'B';
21              else if (i == 13)
22                  board[i] = EMPTY;
23              else
24                  board[i] = 'W';
25          }
26          isWhite = (turn % 2 == 1);
27          isGameDone = false;
28      }
29
30      /**
31       * Returns the positions of all black pieces on the board.
32       *
33       * @return the positions of all black pieces on the board.
34       */
35      public int[] black() {
36          ArrayList<Integer> blackPieces = new ArrayList<Integer>();
37          for (int i = 1; i <= 25; i++)
38              if (this.board[i] == 'B')
39                  blackPieces.add(i);
40          int[] black = new int[blackPieces.size()];
41          for (int i = 0; i < blackPieces.size(); i++)
42              black[i] = blackPieces.get(i);
43          return black;
44      }
45
46      /**
47       * Returns the positions of all white pieces on the board.
48       * @return the positions of all white pieces on the board.
49       */
50      public int[] white() {
51          ArrayList<Integer> whitePieces = new ArrayList<Integer>();
52          for (int i = 1; i <= 25; i++)
53              if (this.board[i] == 'W')
54                  whitePieces.add(i);
55          int[] white = new int[whitePieces.size()];
56          for (int i = 0; i < whitePieces.size(); i++)
57              white[i] = whitePieces.get(i);
58          return white;
59      }
60
61      /**
62       * Moves a piece and updates the board correspondingly.
63       * Precondition: move must be a legal between 1 and 25
64       * @param move the move to simulate.
65       */
66      public void move(Move move) {
67          board[move.to()] = board[move.from()];
68          board[move.from()] = EMPTY;
69          if (isTakeMove(move))    //if the move is a take, the taken piece ↩
                is removed
70              board[(move.to() + move.from()) / 2] = EMPTY; //calculates ↩
                    average position value and removes piece
71          // Updates who's turn it is
72          this.turn++;
73          isWhite = (turn % 2 == 1);
74          // updates finishedGames after eachmove
75          isGameOver();
```

```
76      }
77
78      /**
79       * Checks whether a move is legal.
80       * Precondition: move must be an int from 1 through 25
81       * @param move move input to evaluate.
82       */
83      public boolean isLegal(Move move) {
84          if (board[move.to()] != EMPTY)  // Checks whether the player tries to move from an empty cell
85              return false;
86          else if ((isWhite && board[move.from()] != 'W') || (!isWhite && board[move.from()] != 'B'))
87              // Checks if the player tries to move the opponents piece
88              return false;
89          else if (fileDiff(move) > 2)
90              // Checks if the player tries to move to a column that is too far away, which prevents moves rolling over from one row to the next
91              return false;
92          else if (!isTakeMove(move)){ // Logic for regular moves
93              if ((isWhite && (pieceDiff(move) < -6 || pieceDiff(move) > -4)) ||
94                      (!isWhite && (pieceDiff(move) < 4 || pieceDiff(move) > 6)))
95                  // Checks if direction is correct and if it is within the range of allowed cells to move to
96                  return false;
97              else if (move.from() % 2 == 0 && move.to() % 2 == 0)
98                  // Check for moves on even cells (To confirm that it follows the lines on the board)
99                  return false;
100         } else if (isTakeMove(move)) { // Logic for moves that take another piece
101             if (Math.abs(pieceDiff(move)) != 2 && Math.abs(pieceDiff(move)) != 8 &&
102                     Math.abs(pieceDiff(move)) != 10 && Math.abs(pieceDiff(move)) != 12)
103                 // Checks if the move is to the specified allowed cells for a take move
104                 return false;
105             /*
106             else if (move.from() % 2 == 0 && move.to() % 2 == 1)
107                 // Checks for moves on even cells (to confirm it follows the lines on the board)
108                 return false;
109              */
110             else if (move.from() % 2 == 0 && Math.abs(pieceDiff(move)) != 10 && Math.abs(pieceDiff(move)) != 2)
111                 // Checks for moves on even cells (to confirm it follows the lines on the board)
112                 return false;
113         }
114         return true;
115     }
116
117     /**
118      * Returns an array of all legal moves for this board
119      * @return an array of all legal moves for this board
120      */
121     public Move[] legalMoves() {
122         ArrayList<Move> legalList = new ArrayList<Move>();
```

```
123         for (int i = 1; i < board.length; i++)
124             if (board[i] != EMPTY)
125                 for (int j = 1; j < board.length; j++)
126                     if (isLegal(new Move(i,j)))
127                         legalList.add(new Move(i,j));
128         Move[] legalMoves = new Move[legalList.size()];
129         for (int i = 0; i < legalList.size(); i++)
130             legalMoves[i] = legalList.get(i);
131         return legalMoves;
132     }
133 
134     /**
135      * Returns if the game is over
136      * @return if the game is over
137      */
138     public boolean isGameOver() {
139         if (white().length == 0 || black().length == 0 || legalMoves().↩
              length == 0) {
140             if (!isGameDone) {
141                 finishedGames++;
142                 isGameDone = true;
143             }
144             return true;
145         } else
146             return false;
147     }
148 
149 
150     /**
151      * Returns how many objects of type Board that represents games, that ↩
           are finished games.
152      * @return how many objects of type Board that represents games, that ↩
           are finished games.
153      */
154     public static int finishedGames() {
155         return finishedGames;
156     }
157 
158     /**
159      * Returns a copy of this board
160      * @return a copy of this board
161      */
162     public Board copy() {
163         Board newBoard = new Board();
164         for (int i = 0; i < this.board.length; i++)
165             newBoard.board[i] = this.board[i];
166         newBoard.turn = this.turn;
167         newBoard.isWhite = this.isWhite;
168         newBoard.isGameDone = this.isGameDone;
169         return newBoard;
170     }
171 
172     /**
173      * Checks whether this Board is equal to other Object
174      * @param other Object to check against this board
175      * @return whether this Board is equal to other Object
176      */
177     public boolean equals(Object other){
178         if (other == null) return false;
179         else if (this == other) return true;
180         else if (!(other instanceof Board)) return false;
181         Board otherBoard = (Board) other;
```

```
182         int i = 0;
183         while(i < this.board.length && this.board[i] == otherBoard.board[i↩
                ])
184             i++;
185         return (i == this.board.length && this.turn == otherBoard.turn && ↩
                this.isGameDone == otherBoard.isGameDone);
186     }
187 
188 
189     /**
190      * Returns a hashCode compised of this boards attributes
191      * @return a hashCode comprised of this boards attributes
192      */
193     public int hashCode() {
194         return (this.board.hashCode() + this.turn*31);
195     }
196 
197     /*
198      * Auxillerary methods to check how far there are between the columns ↩
              in the move
199      */
200     private int fileDiff(Move move){
201         return Math.abs(((move.from() - 1) % 5 + 1) - ((move.to() - 1) % 5↩
                + 1));
202     }
203 
204     /*
205      * Auxillerary method to check how far there is between two pieces
206      */
207     private int pieceDiff(Move move) {
208         return (move.to() - move.from());
209     }
210 
211     /*
212      * checks whether the move is a take move
213      */
214     private boolean isTakeMove(Move move) {
215         return ((Math.abs(pieceDiff(move)) > 6 || Math.abs(pieceDiff(move)↩
                ) < 4) &&
216                 ((isWhite && board[(move.to() + move.from()) / 2] == 'B') ↩
                    || //checks if opponent piece is taken
217                         (!isWhite && board[(move.to() + move.from()) / 2] ↩
                            == 'W'))); //checks if opponent piece is taken
218     }
219 }
```