Санкт-Петербургский политехнический университет Петра Великого Институт компьютерных наук и технологи  
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**КУРСОВОЙ ПРОЕКТ**

**Дисциплина**: Алгоритмы и структуры данных

**Тема**: разработка GUI приложения Checkers на языке Java

Выполнил студент гр. 3530901/80003 Ступин А. Г.

Преподаватель Глухих М. И.

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2019г.

Санкт-Петербург

2019

Оглавление

[1. Техническое задание 3](#_Toc25152468)

[2. Метод решения 4](#_Toc25152469)

[3. Листинг программы 5](#_Toc25152470)

[ Finish.kt 5](#_Toc25152471)

[ Help.kt 7](#_Toc25152472)

[ Main.kt 8](#_Toc25152473)

[ MainMenu.kt 9](#_Toc25152474)

[ PlayingField.kt 10](#_Toc25152475)

[ Records.kt 18](#_Toc25152476)

[ Board.kt 22](#_Toc25152477)

[ Opponent.kt 25](#_Toc25152478)

[4. Тесты 31](#_Toc25152479)

[5. Скриншоты программы 33](#_Toc25152480)

# **Техническое задание**

Создать игру Checkers (Шашки), которая представляет из себя настольную игру для двух человек на доске 8 на 8 клеток.

**Ша́шки** — логическая настольная игра для двух игроков, заключающаяся в передвижении определённым образом фишек-шашек по клеткам [шашечной доски](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D1%88%D0%B5%D1%87%D0%BD%D0%B0%D1%8F_%D0%B4%D0%BE%D1%81%D0%BA%D0%B0). Во время партии каждому игроку принадлежат шашки одного цвета: чёрного или белого (иногда других цветов, один из которых, считается тёмным, а другой — светлым). Цель игры — взять все шашки соперника или лишить их возможности хода (запереть). Вариант игры шашек: Английские (Checkers).

Популярны в Великобритании, США, Австралии, Ирландии, Индии, некоторых бывших колониях Великобритании, всего в 30 странах мира. Также играют в чекерс шашисты Италии. В начальной позиции у каждого игрока по 12 шашек, расположенных в первых трёх рядах на чёрных клетках. При нескольких вариантах взятия игрок выбирает вариант по своему усмотрению, в выбранном варианте необходимо бить все доступные для взятия шашки.

##### Отличие правил от русских шашек

* Первый ход делают чёрные.
* Простые шашки могут бить только вперёд.
* Дамка может ходить на одно поле по диагонали вперёд или назад, при взятии ходит только через одно поле в любую сторону, а не на любое поле диагонали, как в русских или международных шашках.

GitHub репозиторий: <https://github.com/Gaussboy/CheckersNew>

1. **Метод решения**

В п рограмме использована концепция MVC (Model-View-Controller) для отделения логики от визуализации.

Весь код разбит на 3 пакета: UI, Logic, Model.

Пакет UI отвечает за визуальное представление приложения. Он содержит функцию main, задание параметров окна и расположение внутри него вс ех элементов графического интерфейс а, а также события на взаимодействие с этими элементами.

Пакет Logic отвечает за логическую составляющую приложения, а также за взаимодействие моделей и визуальной части приложения. Он содержит внутри себя 2 класса: MoveLogic (прописаны правила и возможности игры), MoveGenerator (содержит проверки на возможности шашки ходящего игрока).

Пакет Model за модели шашек, доски, за игру и за игроков. Содержит в себе 6 классов: Board (этот класс реализует доску 8x8. По правилам, шашка может двигаться только на черных плитках, то есть всего 32 доступные плитки.), ComputerPlayer (этот класс представляет компьютерного игрока), Game (этот класс представляет собой игру в шашки. Он предоставляет метод для обновления состояния игры и отслеживания того, чья очередь это), HumanPlayer (этот класс представляет собой игрока), Move (этот класс представляет собой все движение), Player (этот класс содержит методы для игрока (компьютерного или человеческого)).

Когда начинается игра, первым делает ход игрок, который играет за красных, нажимая на шашку игрок видит, возможно ей сходить или нет, и опираясь на личный опыт и вкладку помощь (где расписаны правила игры), совершает первый ход. Если играет бот, алгоритм высчитывает наиболее выгодный ход в данный момент, просчитывая все ходы, которые он сможет совершить в течение данного хода и ставит свою шашку в выгодное положение, дальше игра проходит по аналогии выше написанному.

Принцип работы алгоритма:

Для начала программа ищет все возможные ходы, проверяя все возможные направления, куда может сходить: вверх-влево, вверх-вправо, вниз-вправо(королевы), вниз-влево(королевы), или кого может съесть (через клетку по тем же направлениям).

1. **Листинг программы**

**Пакет UI**

* **CheckerBoard.java**
* package ui;  
  import java.awt.Color;  
  import java.awt.Font;  
  import java.awt.Graphics;  
  import java.awt.Graphics2D;  
  import java.awt.Point;  
  import java.awt.RenderingHints;  
  import java.awt.event.ActionEvent;  
  import java.awt.event.ActionListener;  
  import java.util.List;  
  import javax.swing.JButton;  
  import javax.swing.Timer;  
  import logic.MoveGenerator;  
  import model.Board;  
  import model.Game;  
  import model.HumanPlayer;  
  import model.Player;  
    
    
    
  public class CheckerBoard extends JButton {  
    
   private static final int *TIMER\_DELAY* = 1000;  
    
   private static final int *PADDING* = 16;  
    
   private Game game;  
    
   private Player player1;  
    
   private Player player2;  
    
   private Point selected;  
    
   private boolean selectionValid;  
    
   private Color lightTile;  
    
   private Color darkTile;  
    
   private boolean isGameOver;  
    
   private Timer timer;  
     
   CheckerBoard(CheckersWindow window) {  
   this(window, new Game(), null, null);  
   }  
     
   private CheckerBoard(CheckersWindow window, Game game,  
   Player player1, Player player2) {  
    
   super.setBorderPainted(false);  
   super.setFocusPainted(false);  
   super.setContentAreaFilled(false);  
   super.setBackground(Color.*LIGHT\_GRAY*);  
   this.addActionListener(new ClickListener());  
    
   this.game = (game == null)? new Game() : game;  
   this.lightTile = Color.*WHITE*;  
   this.darkTile = Color.*BLACK*;  
   setPlayer1(player1);  
   setPlayer2(player2);  
   }  
     
    
   void update() {  
   runPlayer();  
   this.isGameOver = game.isGameOver();  
   repaint();  
   }  
     
   private void runPlayer() {  
   Player player = getCurrentPlayer();  
   if (player == null || player.isHuman() ) {  
   return;  
   }  
   this.timer = new Timer(*TIMER\_DELAY*, e -> {  
   getCurrentPlayer().updateGame(game);  
   timer.stop();  
   update();  
   });  
   this.timer.start();  
   }  
    
   private synchronized boolean setGameState(String newState, String expected) {  
    
   if (!game.getGameState().equals(expected)) {  
   return false;  
   }  
    
   this.game.setGameState(newState);  
   repaint();  
    
   return true;  
   }  
    
    
   @Override  
   public void paint(Graphics g) {  
   super.paint(g);  
     
   Graphics2D g2d = (Graphics2D) g;  
   g2d.setRenderingHint(RenderingHints.*KEY\_ANTIALIASING*,  
   RenderingHints.*VALUE\_ANTIALIAS\_ON*);  
   Game game = this.game.copy();  
    
   final int BOX\_PADDING = 4;  
   final int W = getWidth(), H = getHeight();  
   final int DIM = Math.*min*(W, H), BOX\_SIZE = (DIM - 2 \* *PADDING*) / 8;  
   final int OFFSET\_X = (W - BOX\_SIZE \* 8) / 2;  
   final int OFFSET\_Y = (H - BOX\_SIZE \* 8) / 2;  
   final int CHECKER\_SIZE = Math.*max*(0, BOX\_SIZE - 2 \* BOX\_PADDING);  
    
   g.setColor(Color.*BLACK*);  
   g.drawRect(OFFSET\_X - 1, OFFSET\_Y - 1, BOX\_SIZE \* 8 + 1, BOX\_SIZE \* 8 + 1);  
   g.setColor(lightTile);  
   g.fillRect(OFFSET\_X, OFFSET\_Y, BOX\_SIZE \* 8, BOX\_SIZE \* 8);  
   g.setColor(darkTile);  
   for (int y = 0; y < 8; y ++) {  
   for (int x = (y + 1) % 2; x < 8; x += 2) {  
   g.fillRect(OFFSET\_X + x \* BOX\_SIZE, OFFSET\_Y + y \* BOX\_SIZE,  
   BOX\_SIZE, BOX\_SIZE);  
   }  
   }  
    
   if (Board.*isValidPoint*(selected)) {  
   g.setColor(selectionValid? Color.*GREEN* : Color.*RED*);  
   g.fillRect(OFFSET\_X + selected.x \* BOX\_SIZE,  
   OFFSET\_Y + selected.y \* BOX\_SIZE,  
   BOX\_SIZE, BOX\_SIZE);  
   }  
    
   Board b = game.getBoard();  
   for (int y = 0; y < 8; y ++) {  
   int cy = OFFSET\_Y + y \* BOX\_SIZE + BOX\_PADDING;  
   for (int x = (y + 1) % 2; x < 8; x += 2) {  
   int id = b.get(x, y);  
    
   if (id == Board.*EMPTY*) {  
   continue;  
   }  
     
   int cx = OFFSET\_X + x \* BOX\_SIZE + BOX\_PADDING;  
    
   if (id == Board.*BLACK\_CHECKER*) {  
   g.setColor(Color.*DARK\_GRAY*);  
   g.fillOval(cx + 1, cy + 2, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*LIGHT\_GRAY*);  
   g.drawOval(cx + 1, cy + 2, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*RED*);  
   g.fillOval(cx, cy, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*LIGHT\_GRAY*);  
   g.drawOval(cx, cy, CHECKER\_SIZE, CHECKER\_SIZE);  
   }  
    
    
   else if (id == Board.*BLACK\_KING*) {  
   g.setColor(Color.*DARK\_GRAY*);  
   g.fillOval(cx + 1, cy + 2, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*LIGHT\_GRAY*);  
   g.drawOval(cx + 1, cy + 2, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*DARK\_GRAY*);  
   g.fillOval(cx, cy, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*LIGHT\_GRAY*);  
   g.drawOval(cx, cy, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*RED*);  
   g.fillOval(cx - 1, cy - 2, CHECKER\_SIZE, CHECKER\_SIZE);  
   }  
    
    
   else if (id == Board.*WHITE\_CHECKER*) {  
   g.setColor(Color.*LIGHT\_GRAY*);  
   g.fillOval(cx + 1, cy + 2, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*DARK\_GRAY*);  
   g.drawOval(cx + 1, cy + 2, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*BLUE*);  
   g.fillOval(cx, cy, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*DARK\_GRAY*);  
   g.drawOval(cx, cy, CHECKER\_SIZE, CHECKER\_SIZE);  
   }  
    
    
   else if (id == Board.*WHITE\_KING*) {  
   g.setColor(Color.*LIGHT\_GRAY*);  
   g.fillOval(cx + 1, cy + 2, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*DARK\_GRAY*);  
   g.drawOval(cx + 1, cy + 2, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*LIGHT\_GRAY*);  
   g.fillOval(cx, cy, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*DARK\_GRAY*);  
   g.drawOval(cx, cy, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.setColor(Color.*BLUE*);  
   g.fillOval(cx - 1, cy - 2, CHECKER\_SIZE, CHECKER\_SIZE);  
   }  
    
   if (id == Board.*BLACK\_KING* || id == Board.*WHITE\_KING*) {  
   g.setColor(new Color(0, 255,0));  
   g.drawOval(cx - 1, cy - 2, CHECKER\_SIZE, CHECKER\_SIZE);  
   g.drawOval(cx + 1, cy, CHECKER\_SIZE - 4, CHECKER\_SIZE - 4);  
   }  
   }  
   }  
    
    
   String msg = game.isP1Turn()? "Player 1's turn" : "Player 2's turn";  
   int width = g.getFontMetrics().stringWidth(msg);  
   Color back = game.isP1Turn()? Color.*BLACK* : Color.*WHITE*;  
   Color front = game.isP1Turn()? Color.*WHITE* : Color.*BLACK*;  
   g.setColor(back);  
   g.fillRect(W / 2 - width / 2 - 5, OFFSET\_Y + 8 \* BOX\_SIZE + 2,  
   width + 10, 15);  
   g.setColor(front);  
   g.drawString(msg, W / 2 - width / 2, OFFSET\_Y + 8 \* BOX\_SIZE + 2 + 11);  
    
   if (isGameOver) {  
   g.setFont(new Font("Arial", Font.*BOLD*, 20));  
   msg = "Game Over, try again.";  
   width = g.getFontMetrics().stringWidth(msg);  
   g.setColor(new Color(240, 240, 255));  
   g.fillRoundRect(W / 2 - width / 2 - 5,  
   OFFSET\_Y + BOX\_SIZE \* 4 - 16,  
   width + 10, 30, 10, 10);  
   g.setColor(Color.*RED*);  
   g.drawString(msg, W / 2 - width / 2, OFFSET\_Y + BOX\_SIZE \* 4 + 7);  
   }  
   }  
     
   Game getGame() {  
   return game;  
   }  
    
   void setPlayer1(Player player1) {  
   this.player1 = (player1 == null)? new HumanPlayer() : player1;  
   if (game.isP1Turn() && !this.player1.isHuman()) {  
   this.selected = null;  
   }  
   }  
    
   void setPlayer2(Player player2) {  
   this.player2 = (player2 == null)? new HumanPlayer() : player2;  
   if (!game.isP1Turn() && !this.player2.isHuman()) {  
   this.selected = null;  
   }  
   }  
     
   private Player getCurrentPlayer() {  
   return game.isP1Turn()? player1 : player2;  
   }  
    
    
   private void handleClick(int x, int y) {  
    
   if (isGameOver || !getCurrentPlayer().isHuman()) {  
   return;  
   }  
     
   Game copy = game.copy();  
    
   final int W = getWidth(), H = getHeight();  
   final int DIM = Math.*min*(W, H), BOX\_SIZE = (DIM - 2 \* *PADDING*) / 8;  
   final int OFFSET\_X = (W - BOX\_SIZE \* 8) / 2;  
   final int OFFSET\_Y = (H - BOX\_SIZE \* 8) / 2;  
   x = (x - OFFSET\_X) / BOX\_SIZE;  
   y = (y - OFFSET\_Y) / BOX\_SIZE;  
   Point sel = new Point(x, y);  
    
   if (Board.*isValidPoint*(sel) && Board.*isValidPoint*(selected)) {  
   boolean change = copy.isP1Turn();  
   String expected = copy.getGameState();  
   boolean move = copy.move(selected, sel);  
   boolean updated = (move && setGameState(copy.getGameState(), expected));  
   change = (copy.isP1Turn() != change);  
   this.selected = change? null : sel;  
   } else {  
   this.selected = sel;  
   }  
    
   this.selectionValid = isValidSelection(  
   copy.getBoard(), copy.isP1Turn(), selected);  
     
   update();  
   }  
    
   private boolean isValidSelection(Board b, boolean isP1Turn, Point selected) {  
    
   int i = Board.*toIndex*(selected), id = b.get(i);  
   if (id == Board.*EMPTY* || id == Board.*INVALID*) {  
   return false;  
   } else if(isP1Turn ^ (id == Board.*BLACK\_CHECKER* ||  
   id == Board.*BLACK\_KING*)) {  
   return false;  
   } else if (!MoveGenerator.*getSkips*(b, i).isEmpty()) {  
   return true;  
   } else if (MoveGenerator.*getMoves*(b, i).isEmpty()) {  
   return false;  
   }  
    
   List<Point> points = b.find(  
   isP1Turn? Board.*BLACK\_CHECKER* : Board.*WHITE\_CHECKER*);  
   points.addAll(b.find(  
   isP1Turn? Board.*BLACK\_KING* : Board.*WHITE\_KING*));  
   for (Point p : points) {  
   int checker = Board.*toIndex*(p);  
   if (checker == i) {  
   continue;  
   }  
   if (!MoveGenerator.*getSkips*(b, checker).isEmpty()) {  
   return false;  
   }  
   }  
    
   return true;  
   }  
    
   private class ClickListener implements ActionListener {  
    
   @Override  
   public void actionPerformed(ActionEvent e) {  
   Point m = CheckerBoard.this.getMousePosition();  
   if (m != null) {  
   handleClick(m.x, m.y);  
   }  
   }  
   }  
  }
* **CheckersWindow.java**

package ui;  
  
import java.awt.\*;  
  
import javax.swing.JFrame;  
import javax.swing.JPanel;  
  
import model.Player;  
  
  
  
class CheckersWindow extends JFrame {  
  
 private static final int *DEFAULT\_WIDTH* = 500;  
  
 private static final int *DEFAULT\_HEIGHT* = 600;  
  
 private static final String *DEFAULT\_TITLE* = "MY CHECKERS";  
  
 private CheckerBoard board;  
  
  
 CheckersWindow() {  
 this(*DEFAULT\_WIDTH*, *DEFAULT\_HEIGHT*, *DEFAULT\_TITLE*);  
  
 }  
  
 private CheckersWindow(int width, int height, String title) {  
  
 super(title);  
 super.setSize(width, height);  
 super.setLocationByPlatform(true);  
  
 JPanel layout = new JPanel(new BorderLayout());  
 this.board = new CheckerBoard(this);  
 OptionPanel opts = new OptionPanel(this);  
 layout.add(board, BorderLayout.*CENTER*);  
 layout.add(opts, BorderLayout.*SOUTH*);  
 this.add(layout);  
 }  
   
 void getBoard() {  
 }  
  
 void setPlayer1(Player player1) {  
 this.board.setPlayer1(player1);  
 this.board.update();  
 }  
  
 void setPlayer2(Player player2) {  
 this.board.setPlayer2(player2);  
 this.board.update();  
 }  
   
 void restart() {  
 this.board.getGame().restart();  
 this.board.update();  
 }  
}

* **Main.java**

package ui;  
  
import javax.swing.UIManager;  
  
public class Main {  
  
 public static void main(String[] args) {  
 try {  
 UIManager.*setLookAndFeel*(  
 UIManager.*getSystemLookAndFeelClassName*());  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 CheckersWindow window = new CheckersWindow();  
 window.setDefaultCloseOperation(CheckersWindow.*EXIT\_ON\_CLOSE*);  
 window.setVisible(true);  
 }  
}

* **OptionPanel.java**

package ui;  
  
import java.awt.\*;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
import javax.swing.\*;  
  
import model.ComputerPlayer;  
import model.HumanPlayer;  
import model.Player;  
  
  
class OptionPanel extends JPanel {  
  
 private static final long *serialVersionUID* = -4763875452164030755L;  
  
 private CheckersWindow window;  
  
 private JButton helpBtn;  
  
 private JButton restartBtn;  
  
 private JComboBox<String> player1Opts;  
  
 private JComboBox<String> player2Opts;  
  
  
 OptionPanel(CheckersWindow window) {  
 super(new GridLayout(0, 1));  
   
 this.window = window;  
  
 OptionListener ol = new OptionListener();  
 final String[] playerTypeOpts = {"Human", "Computer"};  
 this.helpBtn = new JButton("Help");  
 this.restartBtn = new JButton("Restart");  
 this.player1Opts = new JComboBox<>(playerTypeOpts);  
 this.player2Opts = new JComboBox<>(playerTypeOpts);  
 this.restartBtn.addActionListener(ol);  
 this.helpBtn.addActionListener(ol);  
 this.player1Opts.addActionListener(ol);  
 this.player2Opts.addActionListener(ol);  
 JPanel top = new JPanel(new FlowLayout(FlowLayout.*CENTER*));  
 JPanel middle = new JPanel(new FlowLayout(FlowLayout.*LEFT*));  
 JPanel bottom = new JPanel(new FlowLayout(FlowLayout.*LEFT*));  
 JButton player1Btn = new JButton();  
 player1Btn.addActionListener(ol);  
 player1Btn.setVisible(false);  
 JButton player2Btn = new JButton();  
 player2Btn.addActionListener(ol);  
 player2Btn.setVisible(false);  
 top.add(helpBtn);  
 top.add(restartBtn);  
 middle.add(new JLabel("(black) Player 1: "));  
 middle.add(player1Opts);  
 middle.add(player1Btn);  
 bottom.add(new JLabel("(white) Player 2: "));  
 bottom.add(player2Opts);  
 bottom.add(player2Btn);  
 this.add(top);  
 this.add(middle);  
 this.add(bottom);  
 }  
  
 private static Player getPlayer(JComboBox<String> playerOpts) {  
  
 Player player = new HumanPlayer();  
 if (playerOpts == null) {  
 return player;  
 }  
  
 String type = "" + playerOpts.getSelectedItem();  
 if (type.equals("Computer")) {  
 player = new ComputerPlayer();  
 }  
  
 return player;  
 }  
  
  
 private class OptionListener implements ActionListener {  
  
 @Override  
 public void actionPerformed(ActionEvent e) {  
  
 if (window == null) {  
 return;  
 }  
  
 Object src = e.getSource();  
  
 if (src == restartBtn) {  
 window.restart();  
 window.getBoard();  
 } else if (src == player1Opts) {  
 Player player = *getPlayer*(player1Opts);  
 window.setPlayer1(player);  
 } else if (src == player2Opts) {  
 Player player = *getPlayer*(player2Opts);  
 window.setPlayer2(player);  
 } else if (src == helpBtn) {  
 JOptionPane.*showMessageDialog*(null, "1.\tCheckers can only move diagonally, on dark tiles.\n" +  
 " \n" +  
 " 2.\tNormal checkers can only move forward diagonally (for black checkers,\n" +  
 " \t\tthis is down and for white checkers, this is up).\n" +  
 " \n" +  
 " 3.\tA checker becomes a king when it reaches the opponents end and cannot\n" +  
 " \t\tmove forward anymore.\n" +  
 " \n" +  
 " 4.\tOnce a checker becomes a king, the player's turn is over.\n" +  
 " \n" +  
 " 5.\tAfter a checker/king moves one space diagonally, the player's turn is\n" +  
 " \t\tover.\n" +  
 " \n" +  
 " 6.\tIf an opponent's checker/king can be skipped, it must be skipped.\n" +  
 " \n" +  
 " 7.\tIf after a skip, the same checker can skip again, it must. Otherwise,\n" +  
 " \t\tthe turn is over.\n" +  
 " \n" +  
 " 8.\tThe game is over if a player either has no more checkers or cannot make\n" +  
 " \t\ta move on their turn.\n" +  
 " \n" +  
 " 9.\tThe player with the black checkers moves first.");  
 }  
 }  
 }  
}

**Пакет Logic**

* **MoveGenerator.java**

package logic;  
  
import java.awt.Point;  
import java.util.ArrayList;  
import java.util.List;  
  
import model.Board;  
  
public class MoveGenerator {  
  
 public static List<Point> getMoves(Board board, int startIndex) {  
   
  
 List<Point> endPoints = new ArrayList<>();  
 if (board == null || !Board.*isValidIndex*(startIndex)) {  
 return endPoints;  
 }  
  
 int id = board.get(startIndex);  
 Point p = Board.*toPoint*(startIndex);  
 *addPoints*(endPoints, p, id, 1);  
  
 for (int i = 0; i < endPoints.size(); i ++) {  
 Point end = endPoints.get(i);  
 if (board.get(end.x, end.y) != Board.*EMPTY*) {  
 endPoints.remove(i --);  
 }  
 }  
   
 return endPoints;  
 }  
  
 public static List<Point> getSkips(Board board, int startIndex) {  
   
  
 List<Point> endPoints = new ArrayList<>();  
 if (board == null || !Board.*isValidIndex*(startIndex)) {  
 return endPoints;  
 }  
  
 int id = board.get(startIndex);  
 Point p = Board.*toPoint*(startIndex);  
 *addPoints*(endPoints, p, id, 2);  
  
 for (int i = 0; i < endPoints.size(); i ++) {  
 Point end = endPoints.get(i);  
 if (!*isValidSkip*(board, startIndex, Board.*toIndex*(end))) {  
 endPoints.remove(i --);  
 }  
 }  
 return endPoints;  
 }  
  
 static boolean isValidSkip(Board board,  
 int startIndex, int endIndex) {  
 if (board == null) {  
 return false;  
 }  
  
 if (board.get(endIndex) != Board.*EMPTY*) {  
 return false;  
 }  
  
 int id = board.get(startIndex);  
 int midID = board.get(Board.*toIndex*(Board.*middle*(startIndex, endIndex)));  
 if (id == Board.*INVALID* || id == Board.*EMPTY*) {  
 return false;  
 } else if (midID == Board.*INVALID* || midID == Board.*EMPTY*) {  
 return false;  
 } else return (midID == Board.*BLACK\_CHECKER* || midID == Board.*BLACK\_KING*) == (id == Board.*WHITE\_CHECKER* || id == Board.*WHITE\_KING*);  
 }  
  
 static void addPoints(List<Point> points, Point p, int id, int delta) {  
  
 boolean isKing = (id == Board.*BLACK\_KING* || id == Board.*WHITE\_KING*);  
 if (isKing || id == Board.*BLACK\_CHECKER*) {  
 points.add(new Point(p.x + delta, p.y + delta));  
 points.add(new Point(p.x - delta, p.y + delta));  
 }  
  
 if (isKing || id == Board.*WHITE\_CHECKER*) {  
 points.add(new Point(p.x + delta, p.y - delta));  
 points.add(new Point(p.x - delta, p.y - delta));  
 }  
 }  
}

* **MoveLogic**

package logic;  
  
import java.awt.Point;  
import java.util.ArrayList;  
import java.util.List;  
import model.Board;  
import model.Game;  
  
public class MoveLogic {  
  
 public static boolean isValidMove(Game game,  
 int startIndex, int endIndex) {  
 return game != null && *isValidMove*(game.getBoard(),  
 game.isP1Turn(), startIndex, endIndex, game.getSkipIndex());  
 }  
  
  
 private static boolean isValidMove(Board board, boolean isP1Turn,  
 int startIndex, int endIndex, int skipIndex) {  
  
  
 if (board == null || !Board.*isValidIndex*(startIndex) ||  
 !Board.*isValidIndex*(endIndex)) {  
 return false;  
 } else if (startIndex == endIndex) {  
 return false;  
 } else if (Board.*isValidIndex*(skipIndex) && skipIndex != startIndex) {  
 return false;  
 }  
  
  
 if (!*validateIDs*(board, isP1Turn, startIndex, endIndex)) {  
 return false;  
 } else return *validateDistance*(board, isP1Turn, startIndex, endIndex);  
 }  
  
 private static boolean validateIDs(Board board, boolean isP1Turn,  
 int startIndex, int endIndex) {  
  
 if (board.get(endIndex) != Board.*EMPTY*) {  
 return false;  
 }  
  
 int id = board.get(startIndex);  
 if ((isP1Turn && id != Board.*BLACK\_CHECKER* && id != Board.*BLACK\_KING*)  
 || (!isP1Turn && id != Board.*WHITE\_CHECKER* && id != Board.*WHITE\_KING*)) {  
 return false;  
 }  
  
 Point middle = Board.*middle*(startIndex, endIndex);  
 int midID = board.get(Board.*toIndex*(middle));  
 return midID == Board.*INVALID* || ((isP1Turn ||  
 midID == Board.*BLACK\_CHECKER* || midID == Board.*BLACK\_KING*) &&  
 (!isP1Turn || midID == Board.*WHITE\_CHECKER* ||  
 midID == Board.*WHITE\_KING*));  
 }  
  
 private static boolean validateDistance(Board board, boolean isP1Turn,  
 int startIndex, int endIndex) {  
  
 Point start = Board.*toPoint*(startIndex);  
 Point end = Board.*toPoint*(endIndex);  
 int dx = end.x - start.x;  
 int dy = end.y - start.y;  
 if (Math.*abs*(dx) != Math.*abs*(dy) || Math.*abs*(dx) > 2 || dx == 0) {  
 return false;  
 }  
  
 int id = board.get(startIndex);  
 if ((id == Board.*WHITE\_CHECKER* && dy > 0) ||  
 (id == Board.*BLACK\_CHECKER* && dy < 0)) {  
 return false;  
 }  
  
 Point middle = Board.*middle*(startIndex, endIndex);  
 int midID = board.get(Board.*toIndex*(middle));  
 if (midID < 0) {  
  
 List<Point> checkers;  
 if (isP1Turn) {  
 checkers = board.find(Board.*BLACK\_CHECKER*);  
 checkers.addAll(board.find(Board.*BLACK\_KING*));  
 } else {  
 checkers = board.find(Board.*WHITE\_CHECKER*);  
 checkers.addAll(board.find(Board.*WHITE\_KING*));  
 }  
  
  
 for (Point p : checkers) {  
 int index = Board.*toIndex*(p);  
 if (!MoveGenerator.*getSkips*(board, index).isEmpty()) {  
 return false;  
 }  
 }  
 }  
 return true;  
 }  
  
 public static boolean isSafe(Board board, Point checker) {  
  
 if (board == null || checker == null) {  
 return true;  
 }  
 int index = Board.*toIndex*(checker);  
 if (index < 0) {  
 return true;  
 }  
 int id = board.get(index);  
 if (id == Board.*EMPTY*) {  
 return true;  
 }  
  
 boolean isBlack = (id == Board.*BLACK\_CHECKER* || id == Board.*BLACK\_KING*);  
 List<Point> check = new ArrayList<>();  
 MoveGenerator.*addPoints*(check, checker, Board.*BLACK\_KING*, 1);  
 for (Point p : check) {  
 int start = Board.*toIndex*(p);  
 int tid = board.get(start);  
 if (tid == Board.*EMPTY* || tid == Board.*INVALID*) {  
 continue;  
 }  
  
 boolean isWhite = (tid == Board.*WHITE\_CHECKER* ||  
 tid == Board.*WHITE\_KING*);  
 if (isBlack && !isWhite) {  
 continue;  
 }  
 boolean isKing = (tid == Board.*BLACK\_KING*);  
  
  
 int dx = (checker.x - p.x) \* 2;  
 int dy = (checker.y - p.y) \* 2;  
 if (!isKing && (isWhite ^ (dy < 0))) {  
 continue;  
 }  
 int endIndex = Board.*toIndex*(new Point(p.x + dx, p.y + dy));  
 if (MoveGenerator.*isValidSkip*(board, start, endIndex)) {  
 return false;  
 }  
 }  
 return true;  
 }  
}

**Пакет model**

* **Board.java**
* package model;  
    
  import java.awt.Point;  
  import java.util.ArrayList;  
  import java.util.List;  
    
    
  public class Board {  
    
   public static final int *INVALID* = -1;  
    
   public static final int *EMPTY* = 0;  
    
   public static final int *BLACK\_CHECKER* = 4 + 2;  
    
   public static final int *WHITE\_CHECKER* = 4;  
    
   public static final int *BLACK\_KING* = 4 + 2 + 1;  
    
   public static final int *WHITE\_KING* = 4 + 1;  
    
   private int[] state;  
    
    
   Board() {  
   reset();  
   }  
     
    
   Board copy() {  
   Board copy = new Board();  
   copy.state = state.clone();  
   return copy;  
   }  
    
    
   private void reset() {  
    
   this.state = new int[3];  
   for (int i = 0; i < 12; i ++) {  
   set(i, *BLACK\_CHECKER*);  
   set(31 - i, *WHITE\_CHECKER*);  
   }  
   }  
    
    
   public List<Point> find(int id) {  
   List<Point> points = new ArrayList<>();  
   for (int i = 0; i < 32; i ++) {  
   if (get(i) == id) {  
   points.add(*toPoint*(i));  
   }  
   }  
     
   return points;  
   }  
    
    
   void set(int index, int id) {  
    
   if (!*isValidIndex*(index)) {  
   return;  
   }  
    
   if (id < 0) {  
   id = *EMPTY*;  
   }  
    
   for (int i = 0; i < state.length; i ++) {  
   boolean set = ((1 << (state.length - i - 1)) & id) != 0;  
   this.state[i] = *setBit*(state[i], index, set);  
   }  
   }  
    
    
   public int get(int x, int y) {  
   return get(*toIndex*(x, y));  
   }  
    
   public int get(int index) {  
   if (!*isValidIndex*(index)) {  
   return *INVALID*;  
   }  
   return *getBit*(state[0], index) \* 4 + *getBit*(state[1], index) \* 2  
   + *getBit*(state[2], index);  
   }  
    
    
   public static Point toPoint(int index) {  
   int y = index / 4;  
   int x = 2 \* (index % 4) + (y + 1) % 2;  
   return !*isValidIndex*(index)? new Point(-1, -1) : new Point(x, y);  
   }  
     
    
   private static int toIndex(int x, int y) {  
    
   if (!*isValidPoint*(new Point(x, y))) {  
   return -1;  
   }  
     
   return y \* 4 + x / 2;  
   }  
    
    
   public static int toIndex(Point p) {  
   return (p == null)? -1 : *toIndex*(p.x, p.y);  
   }  
     
    
   private static int setBit(int target, int bit, boolean set) {  
    
   if (bit < 0 || bit > 31) {  
   return target;  
   }  
    
    
   if (set) {  
   target |= (1 << bit);  
   }  
    
   else {  
   target &= (~(1 << bit));  
   }  
     
   return target;  
   }  
     
    
   private static int getBit(int target, int bit) {  
    
   if (bit < 0 || bit > 31) {  
   return 0;  
   }  
     
   return (target & (1 << bit)) != 0? 1 : 0;  
   }  
     
    
   private static Point middle(Point p1, Point p2) {  
    
   if (p1 == null || p2 == null) {  
   return new Point(-1, -1);  
   }  
     
   return *middle*(p1.x, p1.y, p2.x, p2.y);  
   }  
     
    
   public static Point middle(int index1, int index2) {  
   return *middle*(*toPoint*(index1), *toPoint*(index2));  
   }  
     
    
   private static Point middle(int x1, int y1, int x2, int y2) {  
    
   int dx = x2 - x1, dy = y2 - y1;  
   if (x1 < 0 || y1 < 0 || x2 < 0 || y2 < 0 ||  
   x1 > 7 || y1 > 7 || x2 > 7 || y2 > 7) {  
   return new Point(-1, -1);  
   } else if (x1 % 2 == y1 % 2 || x2 % 2 == y2 % 2) {  
   return new Point(-1, -1);  
   } else if (Math.*abs*(dx) != Math.*abs*(dy) || Math.*abs*(dx) != 2) {  
   return new Point(-1, -1);  
   }  
     
   return new Point(x1 + dx / 2, y1 + dy / 2);  
   }  
     
    
   public static boolean isValidIndex(int testIndex) {  
   return testIndex >= 0 && testIndex < 32;  
   }  
     
    
   public static boolean isValidPoint(Point testPoint) {  
     
   if (testPoint == null) {  
   return false;  
   }  
     
    
   final int x = testPoint.x, y = testPoint.y;  
   if (x < 0 || x > 7 || y < 0 || y > 7) {  
   return false;  
   }  
    
   return x % 2 != y % 2;  
   }  
     
   @Override  
   public String toString() {  
   StringBuilder obj = new StringBuilder(getClass().getName() + "[");  
   for (int i = 0; i < 31; i ++) {  
   obj.append(get(i)).append(", ");  
   }  
   obj.append(get(31));  
   return obj + "]";  
   }  
  }
* **ComputerPlayer.java**
* package model;  
    
  import java.awt.Point;  
  import java.util.ArrayList;  
  import java.util.List;  
  import logic.MoveGenerator;  
  import logic.MoveLogic;  
    
    
  public class ComputerPlayer extends Player {  
     
    
   private static final double *WEIGHT\_SKIP* = 25;  
    
   private static final double *SKIP\_ON\_NEXT* = 20;  
    
   private static final double *SAFE\_SAFE* = 5;  
    
   private static final double *SAFE\_UNSAFE* = -40;  
    
   private static final double *UNSAFE\_SAFE* = 40;  
    
   private static final double *UNSAFE\_UNSAFE* = -40;  
    
   private static final double *SAFE* = 3;  
    
   private static final double *UNSAFE* = -5;  
    
   private static final double *KING\_FACTOR* = 2;  
    
    
   @Override  
   public boolean isHuman() {  
   return false;  
   }  
    
   @Override  
   public void updateGame(Game game) {  
    
   if (game == null || game.isGameOver()) {  
   return;  
   }  
    
   Game copy = game.copy();  
   List<Move> moves = getMoves(copy);  
    
   int count = 1;  
   double bestWeight = Move.*WEIGHT\_INVALID*;  
   for (Move m : moves) {  
   getMoveWeight(copy.copy(), m);  
   if (m.getWeight() > bestWeight) {  
   count = 1;  
   bestWeight = m.getWeight();  
   } else if (m.getWeight() == bestWeight) {  
   count++;  
   }  
   }  
    
    
   int move = ((int) (Math.*random*() \* count)) % count;  
   for (Move m : moves) {  
   if (bestWeight == m.getWeight()) {  
   if (move == 0) {  
   game.move(m.getStartIndex(), m.getEndIndex());  
   } else {  
   move--;  
   }  
   }  
   }  
   }  
    
   private List<Move> getMoves(Game game) {  
    
   if (game.getSkipIndex() >= 0) {  
     
   List<Move> moves = new ArrayList<>();  
   List<Point> skips = MoveGenerator.*getSkips*(game.getBoard(),  
   game.getSkipIndex());  
   for (Point end : skips) {  
   moves.add(new Move(game.getSkipIndex(), Board.*toIndex*(end)));  
   }  
   return moves;  
   }  
    
   List<Point> checkers = new ArrayList<>();  
   Board b = game.getBoard();  
   if (game.isP1Turn()) {  
   checkers.addAll(b.find(Board.*BLACK\_CHECKER*));  
   checkers.addAll(b.find(Board.*BLACK\_KING*));  
   } else {  
   checkers.addAll(b.find(Board.*WHITE\_CHECKER*));  
   checkers.addAll(b.find(Board.*WHITE\_KING*));  
   }  
    
   List<Move> moves = new ArrayList<>();  
   for (Point checker : checkers) {  
   int index = Board.*toIndex*(checker);  
   List<Point> skips = MoveGenerator.*getSkips*(b, index);  
   for (Point end : skips) {  
   Move m = new Move(index, Board.*toIndex*(end));  
   m.changeWeight(*WEIGHT\_SKIP*);  
   moves.add(m);  
   }  
   }  
    
   if (moves.isEmpty()) {  
   for (Point checker : checkers) {  
   int index = Board.*toIndex*(checker);  
   List<Point> movesEnds = MoveGenerator.*getMoves*(b, index);  
   for (Point end : movesEnds) {  
   moves.add(new Move(index, Board.*toIndex*(end)));  
   }  
   }  
   }  
   return moves;  
   }  
     
    
   private int getSkipDepth(Game game, int startIndex, boolean isP1Turn) {  
   if (isP1Turn != game.isP1Turn()) {  
   return 0;  
   }  
   List<Point> skips = MoveGenerator.*getSkips*(game.getBoard(), startIndex);  
   int depth = 0;  
   for (Point end : skips) {  
   int endIndex = Board.*toIndex*(end);  
   game.move(startIndex, endIndex);  
   int testDepth = getSkipDepth(game, endIndex, isP1Turn);  
   if (testDepth > depth) {  
   depth = testDepth;  
   }  
   }  
   return depth + (skips.isEmpty()? 0 : 1);  
   }  
   private void getMoveWeight(Game game, Move m) {  
   Point start = m.getStart(), end = m.getEnd();  
   int startIndex = Board.*toIndex*(start), endIndex = Board.*toIndex*(end);  
   Board b = game.getBoard();  
   boolean changed = game.isP1Turn();  
   boolean safeBefore = MoveLogic.*isSafe*(b, start);  
   int id;  
   boolean isKing;  
   m.changeWeight(getSafetyWeight(b, game.isP1Turn()));  
   if (!game.move(m.getStartIndex(), m.getEndIndex())) {  
   m.setWeight(Move.*WEIGHT\_INVALID*);  
   return;  
   }  
   b = game.getBoard();  
   changed = (changed != game.isP1Turn());  
   id = b.get(endIndex);  
   isKing = (id == Board.*BLACK\_KING* || id == Board.*WHITE\_KING*);  
   boolean safeAfter = true;  
    
   if (changed) {  
   safeAfter = MoveLogic.*isSafe*(b, end);  
   int depth = getSkipDepth(game, endIndex, !game.isP1Turn());  
   if (safeAfter) {  
   m.changeWeight(*SKIP\_ON\_NEXT* \* depth \* depth);  
   } else {  
   m.changeWeight(*SKIP\_ON\_NEXT*);  
   }  
   }  
   else {  
   int depth = getSkipDepth(game, startIndex, game.isP1Turn());  
   m.changeWeight(*WEIGHT\_SKIP* \* depth \* depth);  
   }  
   if (safeBefore && safeAfter) {  
   m.changeWeight(*SAFE\_SAFE*);  
   } else if (!safeBefore && safeAfter) {  
   m.changeWeight(*UNSAFE\_SAFE*);  
   } else if (safeBefore) {  
   m.changeWeight(*SAFE\_UNSAFE* \* (isKing? *KING\_FACTOR* : 1));  
   } else {  
   m.changeWeight(*UNSAFE\_UNSAFE*);  
   }  
   m.changeWeight(getSafetyWeight(b,  
   changed != game.isP1Turn()));  
   }  
    
   private double getSafetyWeight(Board b, boolean isBlack) {  
   double weight = 0;  
   List<Point> checkers = new ArrayList<>();  
   if (isBlack) {  
   checkers.addAll(b.find(Board.*BLACK\_CHECKER*));  
   checkers.addAll(b.find(Board.*BLACK\_KING*));  
   } else {  
   checkers.addAll(b.find(Board.*WHITE\_CHECKER*));  
   checkers.addAll(b.find(Board.*WHITE\_KING*));  
   }  
     
    
   for (Point checker : checkers) {  
   int index = Board.*toIndex*(checker);  
   int id = b.get(index);  
   boolean isKing = (id == Board.*BLACK\_KING* || id == Board.*WHITE\_KING*);  
   if (MoveLogic.*isSafe*(b, checker)) {  
   weight += *SAFE*;  
   } else {  
   weight += *UNSAFE* \* (isKing? *KING\_FACTOR* : 1);  
   }  
   }  
   return weight;  
   }  
  }
* **Game.java**
* package model;  
    
  import java.awt.Point;  
  import java.util.List;  
    
  import logic.MoveGenerator;  
  import logic.MoveLogic;  
    
    
  public class Game {  
    
   private Board board;  
    
   private boolean isP1Turn;  
    
   private int skipIndex;  
     
   public Game() {  
   restart();  
   }  
    
   public Game copy() {  
   Game g = new Game();  
   g.board = board.copy();  
   g.isP1Turn = isP1Turn;  
   g.skipIndex = skipIndex;  
   return g;  
   }  
    
    
   public void restart() {  
   this.board = new Board();  
   this.isP1Turn = true;  
   this.skipIndex = -1;  
   }  
    
   public boolean move(Point start, Point end) {  
   if (start == null || end == null) {  
   return false;  
   }  
   return move(Board.*toIndex*(start), Board.*toIndex*(end));  
   }  
     
    
   boolean move(int startIndex, int endIndex) {  
    
   if (!MoveLogic.*isValidMove*(this, startIndex, endIndex)) {  
   return false;  
   }  
    
   Point middle = Board.*middle*(startIndex, endIndex);  
   int midIndex = Board.*toIndex*(middle);  
   this.board.set(endIndex, board.get(startIndex));  
   this.board.set(midIndex, Board.*EMPTY*);  
   this.board.set(startIndex, Board.*EMPTY*);  
   Point end = Board.*toPoint*(endIndex);  
   int id = board.get(endIndex);  
   boolean switchTurn = false;  
   if (end.y == 0 && id == Board.*WHITE\_CHECKER*) {  
   this.board.set(endIndex, Board.*WHITE\_KING*);  
   switchTurn = true;  
   } else if (end.y == 7 && id == Board.*BLACK\_CHECKER*) {  
   this.board.set(endIndex, Board.*BLACK\_KING*);  
   switchTurn = true;  
   }  
    
   boolean midValid = Board.*isValidIndex*(midIndex);  
   if (midValid) {  
   this.skipIndex = endIndex;  
   }  
   if (!midValid || MoveGenerator.*getSkips*(  
   board.copy(), endIndex).isEmpty()) {  
   switchTurn = true;  
   }  
   if (switchTurn) {  
   this.isP1Turn = !isP1Turn;  
   this.skipIndex = -1;  
   }  
     
   return true;  
   }  
    
   public Board getBoard() {  
   return board.copy();  
   }  
     
    
   public boolean isGameOver() {  
    
   List<Point> black = board.find(Board.*BLACK\_CHECKER*);  
   black.addAll(board.find(Board.*BLACK\_KING*));  
   if (black.isEmpty()) {  
   return true;  
   }  
   List<Point> white = board.find(Board.*WHITE\_CHECKER*);  
   white.addAll(board.find(Board.*WHITE\_KING*));  
   if (white.isEmpty()) {  
   return true;  
   }  
    
   List<Point> test = isP1Turn? black : white;  
   for (Point p : test) {  
   int i = Board.*toIndex*(p);  
   if (!MoveGenerator.*getMoves*(board, i).isEmpty() ||  
   !MoveGenerator.*getSkips*(board, i).isEmpty()) {  
   return false;  
   }  
   }  
    
   return true;  
   }  
     
   public boolean isP1Turn() {  
   return isP1Turn;  
   }  
    
   public int getSkipIndex() {  
   return skipIndex;  
   }  
     
    
   public String getGameState() {  
    
   StringBuilder state = new StringBuilder();  
   for (int i = 0; i < 32; i ++) {  
   state.append(board.get(i));  
   }  
    
   state.append(isP1Turn ? "1" : "0");  
   state.append(skipIndex);  
     
   return state.toString();  
   }  
    
   public void setGameState(String state) {  
   restart();  
   if (state == null || state.isEmpty()) {  
   return;  
   }  
   int n = state.length();  
   for (int i = 0; i < 32 && i < n; i ++) {  
   try {  
   int id = Integer.*parseInt*("" + state.charAt(i));  
   this.board.set(i, id);  
   } catch (NumberFormatException ignored) {}  
   }  
    
   if (n > 32) {  
   this.isP1Turn = (state.charAt(32) == '1');  
   }  
   if (n > 33) {  
   try {  
   this.skipIndex = Integer.*parseInt*(state.substring(33));  
   } catch (NumberFormatException e) {  
   this.skipIndex = -1;  
   }  
   }  
   }  
  }
* **HumanPlayer.java**
* package model;  
    
    
  public class HumanPlayer extends Player {  
    
   @Override  
   public boolean isHuman() {  
   return true;  
   }  
    
   @Override  
   public void updateGame(Game game) {}  
  }
* **Move.java**
* package model;  
    
  import java.awt.Point;  
    
  public class Move {  
     
    
   static final double *WEIGHT\_INVALID* = Double.*NEGATIVE\_INFINITY*;  
    
   private byte startIndex;  
    
   private byte endIndex;  
    
   private double weight;  
     
   Move(int startIndex, int endIndex) {  
   setStartIndex(startIndex);  
   setEndIndex(endIndex);  
   }  
    
   int getStartIndex() {  
   return startIndex;  
   }  
     
   private void setStartIndex(int startIndex) {  
   this.startIndex = (byte) startIndex;  
   }  
     
   int getEndIndex() {  
   return endIndex;  
   }  
     
   private void setEndIndex(int endIndex) {  
   this.endIndex = (byte) endIndex;  
   }  
     
   Point getStart() {  
   return Board.*toPoint*(startIndex);  
   }  
    
   Point getEnd() {  
   return Board.*toPoint*(endIndex);  
   }  
    
   double getWeight() {  
   return weight;  
   }  
    
   void setWeight(double weight) {  
   this.weight = weight;  
   }  
     
   void changeWeight(double delta) {  
   this.weight += delta;  
   }  
     
   @Override  
   public String toString() {  
   return getClass().getSimpleName() + "[startIndex=" + startIndex + ", "  
   + "endIndex=" + endIndex + ", weight=" + weight + "]";  
   }  
  }

**Player.java**

package model;  
  
  
public abstract class Player {  
  
  
 public abstract boolean isHuman();  
  
 public abstract void updateGame(Game game);  
   
 @Override  
 public String toString() {  
 return getClass().getSimpleName() + "[isHuman=" + isHuman() + "]";  
 }  
}

1. **Тесты**

Суть тестов заключается в том, что задается игровое поле и алгоритм должен совершить правильный ход, неправильный ход, и рестарт игры.

package model;  
  
public class Test {  
  
 @org.junit.Test  
 public void testToFailMove() {  
 Game game = new Game();  
 game.move(11,28);  
 System.*out*.println(game.getBoard().toString());  
  
 }  
 @org.junit.Test  
 public void testToTrueMove() {  
 Game game = new Game();  
 game.move(11,15);  
 System.*out*.println(game.getBoard().toString());  
 }  
 @org.junit.Test  
 public void testToRestart() {  
 Game game = new Game();  
 game.move(11,15);  
 game.restart();  
 System.*out*.println(game.getBoard().toString());  
 }  
}

1. **Скриншоты программы**

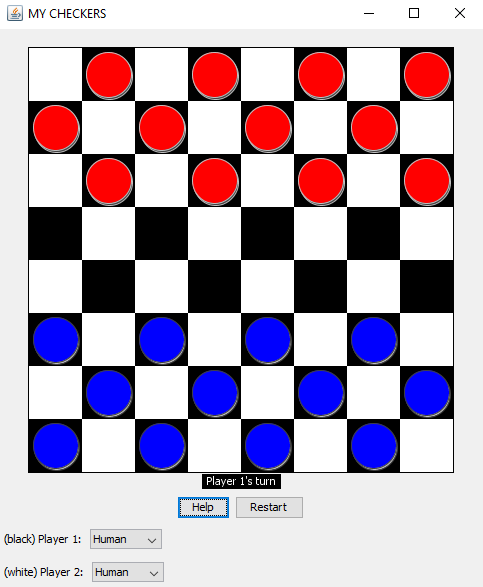
****

Рисунок 5.1 Начало игры

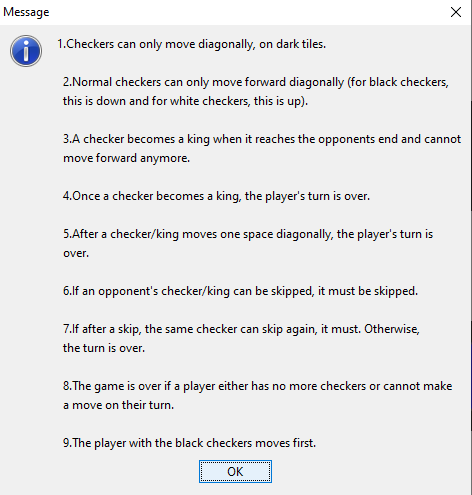


Рисунок 5.2 Помощь

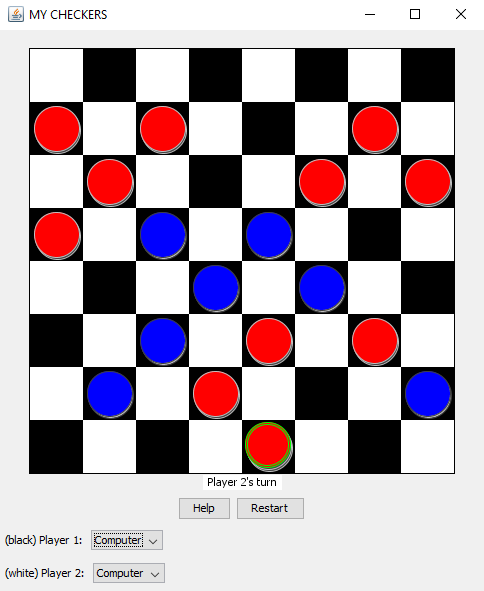


Рисунок 5.5 – Середина игры

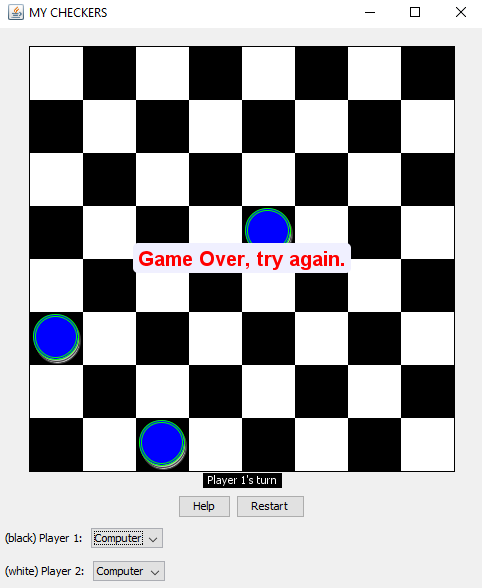


Рисунок 5.6 – Конец игры