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 Русские шашки на языке Java

Выполнил студент гр. 3530901/80003 Джужуев. Э.Х.

Преподаватель Глухих М.И.

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

Санкт-Петербург

2019

Оглавление

1. Техническое задание ...............................................................................3

2. Метод решения ..........................................................................................4

3. Листинг программы ................................................................................5

ChessBoard.java ................................................................................5

MainWindow.java .............................................................................6

UpdateGuiEvent.java ......................................................................11

PlayerErrorException.java ..........................................................11

ObservePlayerQueue.java ..............................................................12

NoSuchDirectionException.java ....................................................12

Logic.java .........................................................................................12

DataListener.java ............................................................................29

ChessBoardData.java ....................................................................29

Cell.java ...........................................................................................30

ActionChessBoard.java ..................................................................33

4. Тесты ..........................................................................................................38

5. Скриншоты программы ........................................................................40

1. **Техническое задание**

Создать игру Русские шашки, которая представляет из себя настольную игру для двух человек на доске 8 на 8 клеток.

В игре используется квадратная доска размером 8 × 8 клеток и по 12 шашек белого и красного цвета. Один из игроков играет белыми, другой — чёрными. Ходы возможно лишь диагонально, обычная шашка может ходить только вперед, но есть вражескую может и назад, простой ход возможен лишь на одну клетку, при взятие вражеской шашки дружеская через нее переступает, тем самым ходя на 2 клетки. Дама может ходить на неограниченное количество клеток по вертикали как вперед так и назад.

GitHub репозиторий: <https://github.com/Shadow-Haund/Shashki>

**Метод решения**

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

Весь код разбит на 3 пакета: view, controller и model

Пакет view отвечает за визуальное представление приложения. Он содержит функцию MainWindow, задание параметров окна и расположение внутри него всех элементов графического интерфейса. Так же содержит класс ChessBoard (отвечает отрисовку доски)

Пакет model отвечает за логическую составляющую приложения. Он содержит внутри себя 8 классов: Cell (содержит все параметры клетки доски, как индекс, координаты.), ChessBoardData (содержив все основные переменные программы, содержит перечисления статуса, выбора игрока и текущего состояния игры и тд.), DataListener (класс расширение для EventListener), Logic (содержит всю логику поведения бота), NoSuchDirectionException (исключение неверного направления), PlayerErrorException (исключение кидается в случае проблем с выбором игрока), UpdateGuiEvent (служит для обновления интерфейса), ObservePlayerQueue (определяет кому сейчас ходить)

Пакет controller отвечает за взаимодействие игрока с игрой. Содержит 1 класс – ActionChessBoard (взаимодействие игрока с игрой)

Когда начинается игра, первым делает ход игрок, который играет за белых, после алгоритм высчитывает наиболее выгодный ход в данный момент, возможную защиту своей фишки, возможность взять фишку соперника, возможность создать Даму, возможность взять фишку соперника находящегося в центре поля.

Принцип работы алгоритма:

Алгоритм работает по принциы высчитывания веса хода

1. **Листинг программы**

**Пакет view**

* **ChessBoard.java**
* import model.Cell;  
  import model.ChessBoardData;  
    
  import javax.swing.\*;  
  import java.awt.\*;  
  class ChessBoard extends JPanel {  
    
   ChessBoardData data;  
    
   ChessBoard(ChessBoardData data) {  
   this.data = data;  
   this.setMinimumSize(data.PREFERRED\_SIZE);  
   this.setPreferredSize(data.PREFERRED\_SIZE);  
   }// конструктор  
    
   @Override  
   public void paintComponent(Graphics g) {  
   super.paintComponent(g);  
   Graphics2D g2d = (Graphics2D) g;  
   g2d.setRenderingHint(RenderingHints.*KEY\_ANTIALIASING*, RenderingHints.*VALUE\_ANTIALIAS\_ON*);  
   g2d.setRenderingHint(RenderingHints.*KEY\_TEXT\_ANTIALIASING*, RenderingHints.*VALUE\_TEXT\_ANTIALIAS\_ON*);  
   Font font = new Font("Dialog", Font.*PLAIN*, 14);  
   g2d.setFont(font);  
    
   // Отрисовка букв и цифр у доски  
   for (int i = 1; i < (data.CELL\_SIDE\_NUM + 1); i++) {  
   // цифры  
   g2d.drawString(Integer.*toString*(data.REVERS\_NUMBERS[i]), data.OFFSET\_LEFT\_BOUND + 40, data.OFFSET\_TOP\_BOUND + (i \* data.CELL\_SIZE) + 30);  
   // буквы  
   g2d.drawString(data.LITERALS[i], data.OFFSET\_LEFT\_BOUND + (i \* data.CELL\_SIZE) + 20, data.OFFSET\_TOP\_BOUND + 50);  
   }  
    
   for (int cCount = 0; cCount < data.CELL\_NUM; cCount++) {  
   Cell cell = data.cells[cCount];  
   data.checkerX = cell.cX + data.CELL\_SIZE / 2 - data.CHECKER\_DIAMETER / 2; //без них шашки съезжают в левый край  
   data.checkerY = cell.cY + data.CELL\_SIZE / 2 - data.CHECKER\_DIAMETER / 2; //верхний край  
    
   switch (cell.getStatus()) { // отрисовка согласно статусу  
   case *WC*:  
   paintCell(Color.*WHITE*, g2d, cell);  
   break;  
   case *BC*:  
   paintCell(Color.*GRAY*, g2d, cell);  
   break;  
   case *WHITE\_CH*:  
   paintChecker(Color.*WHITE*, g2d, cell);  
   break;  
   case *BLACK\_CH*:  
   paintChecker(Color.*BLACK*, g2d, cell);  
   break;  
   case *WHITE\_ACH*:  
   paintChecker(Color.*RED*, g2d, cell);  
   break;  
   case *BLACK\_ACH*:  
   paintChecker(Color.*BLUE*, g2d, cell);  
   break;  
   case *WHITE\_Q*:  
   paintQueen(Color.*WHITE*, Color.*LIGHT\_GRAY*, g2d, cell);  
   break;  
   case *BLACK\_Q*:  
   paintQueen(Color.*BLACK*, Color.*LIGHT\_GRAY*, g2d, cell);  
   break;  
   case *WHITE\_AQ*:  
   paintQueen(Color.*WHITE*, Color.*RED*, g2d, cell);  
   break;  
   case *BLACK\_AQ*:  
   paintQueen(Color.*BLACK*, Color.*RED*, g2d, cell);  
   break;  
   case *TBCH*:  
   paintChecker(Color.*GREEN*, g2d, cell);  
   break;  
   }  
    
   }// конец цикла  
   this.setPreferredSize(data.PREFERRED\_SIZE);  
   repaint();  
   }  
    
   private void paintCell(Color color, Graphics2D g2d, Cell cell) {  
   g2d.setPaint(color);  
   g2d.fillRect(cell.cX, cell.cY, data.CELL\_SIZE, data.CELL\_SIZE);  
   }  
    
   private void paintChecker(Color color, Graphics2D g2d, Cell cell) {  
   g2d.setPaint(Color.*GRAY*);  
   g2d.fillRect(cell.cX, cell.cY, data.CELL\_SIZE, data.CELL\_SIZE);  
   g2d.setPaint(color);  
   g2d.fillOval(data.checkerX, data.checkerY, data.CHECKER\_DIAMETER, data.CHECKER\_DIAMETER);  
   }  
    
   private void paintQueen(Color color, Color colorInner, Graphics2D g2d, Cell cell) {  
   g2d.setPaint(Color.*GRAY*);  
   g2d.fillRect(cell.cX, cell.cY, data.CELL\_SIZE, data.CELL\_SIZE);  
   g2d.setPaint(color);  
   g2d.fillOval(data.checkerX, data.checkerY, data.CHECKER\_DIAMETER, data.CHECKER\_DIAMETER);  
   g2d.setPaint(colorInner);  
   g2d.fillOval(data.checkerX + data.QUEEN\_INNER\_OFFSET, data.checkerY + data.QUEEN\_INNER\_OFFSET, data.QUEEN\_INNER\_DIAMETER, data.QUEEN\_INNER\_DIAMETER);  
   }  
  }
* **MainWindow.java**
* package view;  
    
  import controller.ActionChessBoard;  
  import model.ChessBoardData;  
  import model.Logic;  
    
  import javax.swing.\*;  
  import java.awt.\*;  
  public class MainWindow {  
    
   private ChessBoardData data;  
   private final ChessBoard cBoard;  
   private JTextArea tArea = new JTextArea(26, 12);  
   private JFrame frame = new JFrame();  
   private JMenuBar menuBar = new JMenuBar();  
   private JMenu menuGame = new JMenu();  
   private JMenu menuSettings = new JMenu();  
   private JMenu itemLanguage = new JMenu();  
   private ButtonGroup langButtonGroup = new ButtonGroup();  
   private JRadioButtonMenuItem rbRusLang = new JRadioButtonMenuItem("Русский");  
   private JRadioButtonMenuItem rbEngLang = new JRadioButtonMenuItem("English", true);  
   private JMenu menuHelp = new JMenu();  
   private JMenuItem itemNewGame = new JMenuItem();  
   private JMenuItem itemExit = new JMenuItem();  
   private JMenuItem itemRules = new JMenuItem();  
   private JMenu itemGameActors = new JMenu();  
   private ButtonGroup actorButtonGroup = new ButtonGroup();  
   private JRadioButtonMenuItem rbCompVSuser = new JRadioButtonMenuItem();  
   private JRadioButtonMenuItem rbUserVScomp = new JRadioButtonMenuItem();  
   private JRadioButtonMenuItem rbUserVSuser = new JRadioButtonMenuItem();  
   private JRadioButtonMenuItem rbCompVScomp = new JRadioButtonMenuItem();  
   private JLabel labelComp = new JLabel();  
   private JLabel labelUser = new JLabel();  
   private JScrollPane scrollPane = new JScrollPane(tArea);  
   private JPanel resultPanel = new JPanel();  
   private BoxLayout boxL = new BoxLayout(resultPanel, BoxLayout.*Y\_AXIS*);  
   private JPanel mainPanel = new JPanel(new FlowLayout());  
   //  
   private JFrame frameModal;  
   private JLabel labelModal;  
   private JButton buttonModalExit = new JButton();  
   private JButton buttonModalNewGame = new JButton();  
    
   public void updateGui() {  
   labelUser.setText(data.labelWhiteTitle + data.whiteCheckers);  
   labelComp.setText(data.labelBlackTitle + data.blackCheckers);  
   tArea.append(data.resultBuf);  
   tArea.setCaretPosition(tArea.getDocument().getLength());  
   }  
    
   public void updateTextGuiLanguageInfo() {  
   setGuiText();  
   }  
    
   public void updateTextGuiLabels() {  
   labelUser.setText(data.labelWhiteTitle + data.whiteCheckers);  
   labelComp.setText(data.labelBlackTitle + data.blackCheckers);  
   }  
    
   public void clearTextArea() {  
   tArea.setText("");  
   }  
    
   private void setGuiText() {  
   labelUser.setText(data.labelWhiteTitle + data.whiteCheckers);  
   labelComp.setText(data.labelBlackTitle + data.blackCheckers);  
   frame.setTitle(data.frameTitle);  
   menuGame.setText(data.gameTitle);  
   menuSettings.setText(data.settingsTitle);  
   itemLanguage.setText(data.languageTitle);  
   itemGameActors.setText(data.gameActorsTitle);  
   rbUserVScomp.setText(data.userVScompTitle);  
   rbCompVSuser.setText(data.compVSuserTitle);  
   rbUserVSuser.setText(data.userVSuserTitle);  
   rbCompVScomp.setText(data.compVScompTitle);  
   menuHelp.setText(data.helpTitle);  
   itemNewGame.setText(data.newGameTitle);  
   itemExit.setText(data.exitTitle);  
   itemRules.setText(data.rulesTitle);  
   labelComp.setText(data.labelBlackTitle + data.blackCheckers);  
   labelUser.setText(data.labelWhiteTitle + data.whiteCheckers);  
   }  
    
   public void createShowModalFrame(String frameModalTitle, String labelText, String bNewGame, String bExit) {  
   frameModal = new JFrame(frameModalTitle);  
   frameModal.setLayout(new BorderLayout());  
   buttonModalNewGame.setText(bNewGame);  
   buttonModalExit.setText(bExit);  
   labelModal = new JLabel(labelText);  
    
   JPanel frameModalPanel = new JPanel(new FlowLayout());  
   JPanel buttonPanel = new JPanel(new FlowLayout());  
   JPanel labelPanel = new JPanel(new BorderLayout());  
    
   labelPanel.add(labelModal);  
   frameModalPanel.add(BorderLayout.*NORTH*, labelPanel);  
   frameModalPanel.add(BorderLayout.*SOUTH*, buttonPanel);  
    
   buttonPanel.add(buttonModalNewGame);  
   buttonPanel.add(buttonModalExit);  
    
   frameModal.getContentPane().add(frameModalPanel);  
   frameModal.setResizable(false);  
   frameModal.setPreferredSize(new Dimension(300, 100));  
   frameModal.setFocusable(true);  
   frameModal.getRootPane().setOpaque(true);  
   frameModal.setDefaultCloseOperation(JFrame.*DO\_NOTHING\_ON\_CLOSE*);  
   frameModal.pack();  
   frameModal.setLocationRelativeTo(null);  
   frameModal.setVisible(true);  
   frame.setEnabled(false);  
   }  
    
   public void noBlackCkeckersLeft() {  
   createShowModalFrame(data.whiteWon, data.noBlackCheckersText, data.newGameTitle, data.exitTitle);  
   }  
    
   public void noWhiteCkeckersLeft() {  
   createShowModalFrame(data.blackWon, data.noWhiteCheckersText, data.newGameTitle, data.exitTitle);  
   }  
    
   public void blackIsBlocked() {  
   createShowModalFrame(data.whiteWon, data.blackIsBlockedText, data.newGameTitle, data.exitTitle);  
   }  
    
   public void whiteIsBlocked() {  
   createShowModalFrame(data.blackWon, data.whiteIsBlockedText, data.newGameTitle, data.exitTitle);  
   }  
    
   public void modalFrameDispose() {  
   if (frameModal != null) {  
   frameModal.dispose();  
   frameModal = null;  
   }  
   frame.setEnabled(true);  
   frame.requestFocus();  
   }  
    
   public void showAbout() {  
   JOptionPane.*showMessageDialog*(null,null, data.aboutTitle, JOptionPane.*INFORMATION\_MESSAGE*);  
   }  
    
   MainWindow(ChessBoardData data) {  
   this.data = data;  
   cBoard = new ChessBoard(data);  
   setGuiText();  
    
   rbUserVScomp.setSelected(true);  
    
   menuGame.add(itemNewGame);  
   menuGame.add(itemExit);  
    
   langButtonGroup.add(rbRusLang);  
   langButtonGroup.add(rbEngLang);  
    
   itemLanguage.add(rbRusLang);  
   itemLanguage.add(rbEngLang);  
    
   actorButtonGroup.add(rbUserVScomp);  
   actorButtonGroup.add(rbCompVSuser);  
   actorButtonGroup.add(rbUserVSuser);  
   actorButtonGroup.add(rbCompVScomp);  
    
   itemGameActors.add(rbUserVScomp);  
   itemGameActors.add(rbCompVSuser);  
   itemGameActors.add(rbUserVSuser);  
   itemGameActors.add(rbCompVScomp);  
    
   menuSettings.add(itemGameActors);  
   menuSettings.add(itemLanguage);  
    
   menuHelp.add(itemRules);  
    
   menuBar.add(menuGame);  
   menuBar.add(menuSettings);  
   menuBar.add(menuHelp);  
    
   tArea.setFont(new Font("Dialog", Font.*PLAIN*, 12));  
   tArea.setLineWrap(true);  
   tArea.setWrapStyleWord(true);  
   tArea.setEditable(false);  
    
   scrollPane.setVerticalScrollBarPolicy(JScrollPane.*VERTICAL\_SCROLLBAR\_ALWAYS*);  
   scrollPane.setAlignmentX(JScrollPane.*LEFT\_ALIGNMENT*);  
    
   labelUser.setAlignmentX(JLabel.*LEFT\_ALIGNMENT*);  
   labelComp.setAlignmentX(JLabel.*LEFT\_ALIGNMENT*);  
    
   resultPanel.setLayout(boxL);  
   resultPanel.add(labelUser);  
   resultPanel.add(labelComp);  
   resultPanel.add(Box.*createVerticalStrut*(10));  
   resultPanel.add(scrollPane);  
   resultPanel.add(Box.*createVerticalStrut*(20));  
    
   mainPanel.add(cBoard);  
   mainPanel.add(resultPanel);  
    
   frame.setIconImage(new ImageIcon("./logo.png").getImage());  
   frame.setFocusable(true);  
   frame.getRootPane().setOpaque(true);  
   frame.getContentPane().setLayout(new BorderLayout());  
   frame.getContentPane().add(menuBar, BorderLayout.*NORTH*);  
   frame.getContentPane().add(mainPanel, BorderLayout.*CENTER*);  
   frame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
   frame.setMinimumSize(new Dimension(700, 500));  
   frame.pack();  
   frame.setLocationRelativeTo(null);  
   frame.setVisible(true);  
   }  
    
   public static void main(String[] args) {  
   try {  
   UIManager.*setLookAndFeel*(UIManager.*getSystemLookAndFeelClassName*());  
    
   Logic logic = new Logic();  
   logic.data.setEnglishLang();  
   logic.data.setUSERvsCOMP();  
    
   MainWindow mainW = new MainWindow(logic.data);  
   ActionChessBoard act = new ActionChessBoard(logic.data, logic, mainW);  
   logic.data.addDataListener(act);  
    
   mainW.cBoard.addMouseListener(act);  
   mainW.frame.addKeyListener(act);  
    
   mainW.itemExit.addActionListener(act);  
   mainW.itemRules.addActionListener(act);  
   mainW.itemNewGame.addActionListener(act);  
    
   mainW.rbRusLang.addActionListener(act);  
   mainW.rbEngLang.addActionListener(act);  
    
   mainW.rbUserVScomp.addActionListener(act);  
   mainW.rbCompVSuser.addActionListener(act);  
   mainW.rbUserVSuser.addActionListener(act);  
   mainW.rbCompVScomp.addActionListener(act);  
    
   mainW.buttonModalNewGame.addActionListener(act);  
   mainW.buttonModalExit.addActionListener(act);  
    
   } catch (Exception ex) {  
   ex.printStackTrace();  
   }  
   }  
  }

**Пакет model**

* **UpdateGuiEvent.java**

package model;  
  
import java.util.EventObject;  
public class UpdateGuiEvent extends EventObject { // обновление итерфейса  
  
 public UpdateGuiEvent(Object source) {super(source);}  
}

* **PlayerErrorException.java**

package model;  
  
public class PlayerErrorException extends Exception {  
  
 PlayerErrorException() {  
 super();  
 }  
}

* **ObservePlayerQueue.java**

package model;  
public class ObservePlayerQueue implements Runnable { // определение хода комьютера  
  
 private Logic logic;  
  
 public void run() {  
 try {  
 while (!logic.data.gameExit) {  
 Thread.*sleep*(1000);  
 if ((!logic.data.whiteIsHuman && logic.data.whiteSessionContinue && !logic.data.gameOver)  
 || (!logic.data.blackIsHuman && logic.data.blackSessionContinue && !logic.data.gameOver)) {  
 logic.compStep(logic.data);  
 }  
 }  
 } catch (InterruptedException ex) {  
 ex.printStackTrace();  
 }  
  
 }  
  
 public ObservePlayerQueue(Logic logic) {  
 this.logic = logic;  
 }  
}

* **NoSuchDirectionException.java**

package model;  
public class NoSuchDirectionException extends Exception {  
  
 NoSuchDirectionException(String message) {  
 super(message);  
 }  
  
 NoSuchDirectionException() {  
 super();  
 }  
}

* **Logic.java**

package model;  
  
import java.util.ArrayList;  
import java.util.Random;  
  
import static model.Logic.Action.*FIGHT*;  
import static model.Logic.Action.*MOVE*;  
import static model.Logic.Direction.\*;  
import static model.Logic.Player.*BLACK*;  
import static model.Logic.Player.*WHITE*;  
public class Logic {  
  
 public ChessBoardData data;  
 boolean steelFighterFlag = false;  
 private String userResultCheckersNum;  
 private ArrayList<Cell> turkishArr = new ArrayList<Cell>();  
  
 enum Direction {  
  
 *RU*(1, -1), // направления движения для белых  
 *RB*(1, 1), // черных  
 *LB*(-1, 1), // черных  
 *LU*(-1, -1); // белых  
 private final int kX;  
 private final int kY;  
  
 boolean isWhiteD() {  
 return (this == *RU* || this == *LU*);  
 }  
  
 boolean isBlackD() {  
 return (this == *RB* || this == *LB*);  
 }  
  
 boolean isExist() {  
 return (this == *RU* || this == *LU* || this == *RB* || this == *LB*);  
 }  
  
 Direction(int kX, int kY) {  
 this.kX = kX;  
 this.kY = kY;  
 }  
 }  
  
 enum Action {  
  
 *MOVE*, *FIGHT*;  
  
 boolean isFight() {  
 return this == *FIGHT*;  
 }  
  
 boolean isMove() {  
 return this == *MOVE*;  
 }  
 }  
  
 enum Player {  
  
 *WHITE*, *BLACK*;  
  
 boolean isWhite() {  
 return this == *WHITE*;  
 }  
  
 boolean isBlack() {  
 return this == *BLACK*;  
 }  
 }  
  
 public void compStep(ChessBoardData data) {  
 try {  
 Player computer;  
 if ((!data.whiteIsHuman) && data.whiteSessionContinue) { // определяет когда ходить компьютеру  
 computer = *WHITE*;  
 } else if ((!data.blackIsHuman) && data.blackSessionContinue) {  
 computer = *BLACK*;  
 } else {  
 throw new PlayerErrorException();  
 }  
 if (data.whiteSessionContinue) { // сообщения действий компьютера  
 data.resultBuf = data.stepWhiteText + "\n";  
 } else {  
 data.resultBuf = data.stepBlackText + "\n";  
 }  
  
 Cell activeCell;  
 Cell targetCell;  
 Cell victimCell;  
 // Сначала битва  
 if (getSome(this.data, *FIGHT*, computer).isExist()) {  
 do {  
 Thread.*currentThread*().*sleep*(500);  
 Cell actCells[] = getBestEnemy(computer); // получение лучшей цели  
 activeCell = actCells[0];  
 targetCell = actCells[1];  
 victimCell = getVictim(this.data, activeCell, targetCell, getDbyTarget(this.data, activeCell, targetCell));  
 turkishArr.add(victimCell); // турейцкий ход, взятие нескольких шашек если такая возможность есть  
 victimCell.setTurkichChecker();  
 activeCell.resetActive();  
 targetCell.setStatus(activeCell.getStatus());  
 activeCell.setBlackCell();  
 checkSetQeen(targetCell);  
 activeCell = targetCell;  
 if (!steelFighterFlag) {  
 data.resultBuf += actCells[0].index + ":" + actCells[1].index; // вывод данных на конец битвы  
 } else {  
 data.resultBuf += ":" + activeCell.index; // продолжения битвы  
 }  
 if (isSome(this.data, activeCell, *FIGHT*)) { // бой еще продолжается  
 activeCell.setActive();  
 steelFighterFlag = true;  
 } else {  
 activeCell.resetActive();  
 steelFighterFlag = false;  
 resetTurkishArr();  
 changeSession();  
 data.resultBuf += "\n";  
 customResult();  
 return;  
 }  
 } while (steelFighterFlag);  
 } else {  
 // если нет возможности битвы то просто ходьба  
 Cell actCells[] = getBestEnemy(computer); // получение возможного врага  
 activeCell = actCells[0];  
 targetCell = actCells[1];  
 Thread.*currentThread*().*sleep*(500);  
 targetCell.setStatus(activeCell.getStatus());  
 targetCell.resetActive();  
 checkSetQeen(targetCell);  
 activeCell.setBlackCell();  
 data.resultBuf += activeCell.index + ":" + targetCell.index + "\n";  
 customResult();  
 changeSession();  
 return;  
 }  
 } catch (PlayerErrorException e) {  
 e.printStackTrace();  
 return;  
 } catch (NoSuchDirectionException e) {  
 e.printStackTrace();  
 return;  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public void userStep(Cell activeCell, Cell targetCell) { // определяет ходы игрока человека  
 try {  
 Player player;  
 if (activeCell.isWhite()) {  
 player = *WHITE*;  
 if (!steelFighterFlag) {  
 data.resultBuf = data.stepWhiteText + "\n";  
 }  
 } else if (activeCell.isBlack()) {  
 player = *BLACK*;  
 if (!steelFighterFlag) {  
 data.resultBuf = data.stepBlackText + "\n";  
 }  
 } else {  
 throw new PlayerErrorException();  
 }  
  
 Direction d = getDbyTarget(this.data, activeCell, targetCell); // направление с целью атаковать  
  
 if (getSome(this.data, *FIGHT*, player).isExist() && !isSome(this.data, activeCell, *FIGHT*)) {  
 data.resultBuf += data.userHasFighterText + " " + getSome(this.data, *FIGHT*, player).index + "\n";  
 customResult();  
 return;  
 }  
 // сначало битва  
 if (isRightActByD(this.data, activeCell, targetCell, d, *FIGHT*)) {  
 Cell victimCell = getVictim(this.data, activeCell, targetCell, d);  
 turkishArr.add(victimCell);  
 victimCell.setTurkichChecker();  
 activeCell.resetActive();  
 targetCell.setStatus(activeCell.getStatus());  
 activeCell.setBlackCell();  
 checkSetQeen(targetCell);  
 if (!steelFighterFlag) {  
 userResultCheckersNum = activeCell.index;  
 }  
 activeCell = targetCell;  
 if (isSome(this.data, activeCell, *FIGHT*)) {  
 steelFighterFlag = true;  
 activeCell.setActive();  
 userResultCheckersNum += ":" + targetCell.index;  
 return;  
 } else {  
 activeCell.resetActive();  
 resetTurkishArr();  
 data.resultBuf += userResultCheckersNum + ":" + targetCell.index + "\n";  
 userResultCheckersNum = "";  
 customResult();  
 steelFighterFlag = false;  
 changeSession();  
 return;  
 }  
 }  
  
 // кого надо убить  
 if (isSome(this.data, activeCell, *FIGHT*)) {  
 data.resultBuf += data.userMustFightText + "\n";  
 return;  
 }  
  
 // если убивать уже некого, то можно ходить  
 if (isRightActByD(this.data, activeCell, targetCell, d, *MOVE*)) {  
 targetCell.setStatus(activeCell.getStatus());  
 targetCell.resetActive();  
 checkSetQeen(targetCell);  
 activeCell.setBlackCell();  
 data.resultBuf += activeCell.index + ":" + targetCell.index + "\n";  
 customResult();  
 changeSession();  
 return;  
 }  
  
 } catch (PlayerErrorException e) {  
 e.printStackTrace();  
 return;  
 } catch (NoSuchDirectionException e) {  
 e.printStackTrace();  
 return;  
 }  
 }  
  
 private void changeSession() {  
 data.whiteSessionContinue = !data.whiteSessionContinue;  
 data.blackSessionContinue = !data.blackSessionContinue;  
 }  
  
 private Cell getVictim(ChessBoardData xData, Cell activeCell, Cell targetCell, Direction d) {  
 // функция можно ли шашку впринципе убить  
 if (activeCell.isChecker()) {  
 if (targetCell.equals(getCellByD(xData, activeCell, d, 2))) {  
 return getCellByD(xData, activeCell, d, 1);  
 }  
 }  
 if (activeCell.isQueen()) {  
 int deep = 1;  
 while (getCellByD(xData, activeCell, d, deep).isBlackCell()) {  
 deep++;  
 }  
 if (getCellByD(xData, activeCell, d, deep).isOpposite(activeCell)) {  
 if (getCellByD(xData, activeCell, d, deep + 1).isBlackCell()) {  
 return getCellByD(xData, activeCell, d, deep);  
 }  
 }  
 }  
  
 return new Cell();  
 }  
  
 private Cell[] getBestEnemy(Player computer) throws NoSuchDirectionException {  
 Action act;  
 ArrayList<Cell> variants = new ArrayList<Cell>();  
 Cell activeCell = new Cell();  
 if (computer.isWhite()) {  
 for (int i = 0; i < data.cells.length; i++) {  
 if (data.cells[i].isWhiteActiveChecker() || data.cells[i].isWhiteActiveQueen()) {  
 activeCell = data.cells[i];  
 }  
 }  
 } else if (computer.isBlack()) {  
 for (int i = 0; i < data.cells.length; i++) {  
 if (data.cells[i].isBlackActiveChecker() || data.cells[i].isBlackActiveQueen()) {  
 activeCell = data.cells[i];  
 }  
 }  
 }  
 if (activeCell.isExist()) {  
 act = *FIGHT*;  
 } else {  
 if (getSome(this.data, *FIGHT*, computer).isExist()) {  
 act = *FIGHT*;  
 } else {  
 act = *MOVE*;  
 }  
 }  
  
 if (activeCell.isExist()) {  
 variants.add(activeCell);  
 } else {  
 if (computer.isWhite()) {  
 for (int i = 0; i < data.cells.length; i++) {  
 if (isSome(this.data, data.cells[i], act) && data.cells[i].isWhite()) {  
 variants.add(data.cells[i]);  
 }  
 }  
 } else {  
 for (int i = 0; i < data.cells.length; i++) {  
 if (isSome(this.data, data.cells[i], act) && data.cells[i].isBlack()) {  
 variants.add(data.cells[i]);  
 }  
 }  
 }  
 }  
  
 Cell targets[][] = new Cell[variants.size()][];  
 int points[][] = new int[variants.size()][];  
  
  
 for (int v = 0; v < variants.size(); v++) {  
 ArrayList<Cell> tmpTargets = new ArrayList<Cell>();  
 for (Direction d : Direction.*values*()) {  
 int deep = 1;  
 while (getCellByD(this.data, variants.get(v), d, deep).isExist()) {  
 Cell targetCell = getCellByD(this.data, variants.get(v), d, deep);  
 if (isRightActByD(this.data, variants.get(v), targetCell, d, act)) {  
 tmpTargets.add(targetCell);  
 }  
 deep++;  
 }  
 }  
 targets[v] = tmpTargets.toArray(new Cell[tmpTargets.size()]);  
 points[v] = new int[tmpTargets.size()];  
 }  
  
 int minIndex = -1;  
 int maxIndex = 9;  
 for (int v = 0; v < variants.size(); v++) {  
 if (variants.get(v).isWhiteChecker() && Integer.*parseInt*(variants.get(v).index.substring(1)) > minIndex) {  
 minIndex = Integer.*parseInt*(variants.get(v).index.substring(1));  
 }  
 if (variants.get(v).isBlackChecker() && Integer.*parseInt*(variants.get(v).index.substring(1)) < maxIndex) {  
 maxIndex = Integer.*parseInt*(variants.get(v).index.substring(1));  
 }  
 }  
  
  
 System.*out*.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");  
 for (int v = 0; v < targets.length; v++) {  
 System.*out*.println();  
 System.*out*.println("variant: " + variants.get(v).index);  
 for (int t = 0; t < targets[v].length; t++) {  
 System.*out*.println(" target: " + targets[v][t].index);  
 if (willBeLessUnderAttack(variants.get(v), targets[v][t])) {  
 points[v][t] += 4;  
 System.*out*.println(" willBeLessUnderAttack");  
 }  
 if (willBeMoreUnderAttack(variants.get(v), targets[v][t])) {  
 points[v][t] += -4;  
 System.*out*.println(" willBeMoreUnderAttack");  
 }  
 if (willBeUnderAtackAfterStep(variants.get(v), targets[v][t])) {  
 System.*out*.println(" willBeUnderAtackAfterStep");  
 if (act.isMove()) {  
 points[v][t] += -1;  
 }  
 if (act.isFight() && willBeFighterAfter(variants.get(v), targets[v][t])) {  
 points[v][t] += 1;  
 System.*out*.println(" willBeFighterAfter AND FIGHT");  
 points[v][t] += commonCheckPoints(variants.get(v), targets[v][t], maxIndex, minIndex);  
 }  
 } else {  
 if (willBeFighterAfter(variants.get(v), targets[v][t])) {  
 points[v][t] += 3;  
 System.*out*.println(" willBeFighterAfter");  
 points[v][t] += commonCheckPoints(variants.get(v), targets[v][t], maxIndex, minIndex);  
  
 } else {  
 points[v][t] += 1;  
 System.*out*.println(" NOT willBeFighterAfter");  
 points[v][t] += commonCheckPoints(variants.get(v), targets[v][t], maxIndex, minIndex);  
 }  
 }  
 System.*out*.println(" POINTS: " + points[v][t]);  
 }  
 }  
  
  
  
 int min = -100;  
 for (int v = 0; v < points.length; v++) {  
 for (int t = 0; t < points[v].length; t++) {  
 if (points[v][t] > min) {  
 min = points[v][t];  
 }  
 }  
 }  
  
 int maxValueCount = 0;  
 for (int v = 0; v < points.length; v++) {  
 for (int t = 0; t < points[v].length; t++) {  
 if (points[v][t] == min) {  
 maxValueCount++;  
 }  
 }  
 }  
  
 int i = 0;  
 Cell arrMaxVal[][] = new Cell[maxValueCount][];  
 for (int v = 0; v < points.length; v++) {  
 for (int t = 0; t < points[v].length; t++) {  
 if (points[v][t] == min) {  
 arrMaxVal[i] = new Cell[]{variants.get(v), targets[v][t]};  
 i++;  
 }  
 }  
 }  
  
 if (maxValueCount > 0) {  
 Random rand = new Random();  
 return arrMaxVal[rand.nextInt(maxValueCount)];  
 } else {  
 return new Cell[]{new Cell(), new Cell()};  
 }  
 }  
  
 private int commonCheckPoints(Cell variant, Cell target, int maxIndex, int minIndex) { // определение веса хода  
 int pointSummary = 0;  
 if (isQueenIndex(target)) {  
 pointSummary += 2;  
 System.*out*.println(" isQueenIndex");  
 }  
 if (isCentralField(target)) {  
 pointSummary += 1;  
 System.*out*.println(" isCentralField");  
 }  
 if (variant.isWhiteChecker() && Integer.*parseInt*(target.index.substring(1)) == minIndex) {  
 pointSummary += 1;  
 System.*out*.println("min index: " + target.index + " " + minIndex);  
 }  
 if (variant.isBlackChecker() && Integer.*parseInt*(target.index.substring(1)) == maxIndex) {  
 pointSummary += 1;  
 System.*out*.println("max index: " + target.index + " " + maxIndex);  
 }  
 System.*out*.println(" Points in common step: " + pointSummary);  
 return pointSummary;  
 }  
  
 private boolean willBeFighterAfter(Cell activeCell, Cell targetCell) throws NoSuchDirectionException {  
 // прогноз того, какая шашка станет бойцом после шага  
 ChessBoardData xData = new ChessBoardData();  
 Cell activeCellTmp = new Cell();  
 Cell targetCellTmp = new Cell();  
 for (int i = 0; i < xData.cells.length; i++) {  
 xData.cells[i].setStatus(data.cells[i].getStatus());  
 if (activeCell.index.equals(xData.cells[i].index)) {  
 activeCellTmp = xData.cells[i];  
 }  
 if (targetCell.index.equals(xData.cells[i].index)) {  
 targetCellTmp = xData.cells[i];  
 }  
 }  
  
 Direction d = getDbyTarget(xData, activeCellTmp, targetCellTmp);  
 Cell victimCell = getVictim(xData, activeCellTmp, targetCellTmp, d);  
 targetCellTmp.setStatus(activeCellTmp.getStatus());  
 activeCellTmp.setBlackCell();  
 if (victimCell.isExist()) {  
 victimCell.setTurkichChecker();  
 }  
  
 return isSome(xData, targetCellTmp, *FIGHT*);  
 }  
// я тут  
 private boolean willBeLessUnderAttack(Cell activeCell, Cell targetCell) throws NoSuchDirectionException {  
 // прогноз вероятности быть под угрозой  
 ChessBoardData xData = new ChessBoardData();  
 Cell activeCellTmp = new Cell();  
 Cell targetCellTmp = new Cell();  
 for (int i = 0; i < xData.cells.length; i++) {  
 //  
 xData.cells[i].setStatus(data.cells[i].getStatus());  
 if (activeCell.index.equals(xData.cells[i].index)) {  
 activeCellTmp = xData.cells[i];  
 }  
 if (targetCell.index.equals(xData.cells[i].index)) {  
 targetCellTmp = xData.cells[i];  
 }  
 }  
 Player computer = activeCellTmp.isWhite() ? *WHITE* : *BLACK*;  
 int ownUnderAttackBefore = 0;  
 int ownUnderAttackAfter = 0;  
 for (int i = 0; i < xData.cells.length; i++) {  
 if (xData.cells[i].isOwn(activeCellTmp)) {  
 if (isCellUnderAtack(xData, xData.cells[i], computer)) {  
 ownUnderAttackBefore++;  
 }  
 }  
 }  
  
 targetCellTmp.setStatus(activeCellTmp.getStatus());  
 activeCellTmp.setBlackCell();  
 for (int i = 0; i < xData.cells.length; i++) {  
 if (xData.cells[i].isOwn(targetCellTmp)) {  
 if (isCellUnderAtack(xData, xData.cells[i], computer)) {  
 ownUnderAttackAfter++;  
 }  
 }  
 }  
 return (ownUnderAttackAfter < ownUnderAttackBefore);  
 }  
  
 private boolean willBeMoreUnderAttack(Cell activeCell, Cell targetCell) throws NoSuchDirectionException {  
 ChessBoardData xData = new ChessBoardData();  
 Cell activeCellTmp = new Cell();  
 Cell targetCellTmp = new Cell();  
 for (int i = 0; i < xData.cells.length; i++) {  
 //  
 xData.cells[i].setStatus(data.cells[i].getStatus());  
 if (activeCell.index.equals(xData.cells[i].index)) {  
 activeCellTmp = xData.cells[i];  
 }  
 if (targetCell.index.equals(xData.cells[i].index)) {  
 targetCellTmp = xData.cells[i];  
 }  
 }  
 Player computer = activeCellTmp.isWhite() ? *WHITE* : *BLACK*;  
 int ownUnderAttackBefore = 0;  
 int ownUnderAttackAfter = 0;  
 for (int i = 0; i < xData.cells.length; i++) {  
 if (xData.cells[i].isOwn(activeCellTmp)) {  
 if (isCellUnderAtack(xData, xData.cells[i], computer)) {  
 ownUnderAttackBefore++;  
 }  
 }  
 }  
  
 targetCellTmp.setStatus(activeCellTmp.getStatus());  
 activeCellTmp.setBlackCell();  
 for (int i = 0; i < xData.cells.length; i++) {  
 if (xData.cells[i].isOwn(targetCellTmp)) {  
 if (isCellUnderAtack(xData, xData.cells[i], computer)) {  
 ownUnderAttackAfter++;  
 }  
 }  
 }  
 return (ownUnderAttackAfter > ownUnderAttackBefore);  
 }  
  
 private boolean willBeUnderAtackAfterStep(Cell activeCell, Cell targetCell) throws NoSuchDirectionException {  
// |o| | | Будет ли шашка под атакой ПОСЛЕ ХОДА на ближней дистанции(одна клетка) Рассматриваем вариант по диагонали:  
// наша шашка, пустая клетка(куда собираемся походить), вражеская шашка.  
// | |x| | После хода клетка, занятая нашей шашкой освободиться, и наша шашка станет под атаку. Чтобы этого избежать  
// проверяем эту ситуацию.  
// | | |\*| Для обычных шашек проверяются ихние направления - черные - вниз, белые - вверх. Дамки проверяются по всем  
// направлениям.  
 for (Direction d : Direction.*values*()) {  
 if (getCellByD(this.data, targetCell, d, 1).isOpposite(activeCell) && getCellByD(this.data, targetCell, getOppositeD(d), 1).equals(activeCell)) {  
 return true;  
 }  
 }  
// Проверяем накрест по диагонали наличие шашки противника, нашей шашки, пустой клетки.  
 for (Direction d : Direction.*values*()) {  
 if (getCellByD(this.data, targetCell, d, 1).isOpposite(activeCell) && getCellByD(this.data, targetCell, getOppositeD(d), 1).isBlackCell()) {  
 return true;  
 }  
 }  
// Ишем дамку во всех направлениях до конца диагонали или наличия шашки на диагонали, если находим, проверяем будет ли  
// позади пустая клетка, или та с которой мы будем ходить (соответсвенно она будет тоже пустая после хода)  
 for (Direction d : Direction.*values*()) {  
 int i = 1;  
 while (getCellByD(this.data, targetCell, d, i).isBlackCell() || (getCellByD(this.data, targetCell, d, i).isOwn(activeCell) && getCellByD(this.data, targetCell, d, i).equals(activeCell))) {  
 i++;  
 }  
 if (getCellByD(this.data, targetCell, d, i).isOpposite(activeCell) && getCellByD(this.data, targetCell, d, i).isQueen()) {  
 if (getCellByD(this.data, targetCell, getOppositeD(d), 1).isBlackCell() || getCellByD(this.data, targetCell, getOppositeD(d), 1).equals(activeCell)) {  
 return true;  
 }  
 }  
  
 }  
 return false;  
 }  
  
 // |\*| |\*| |x| |x| Проверяем по диагоналям под атакой ли любая шашка(клетка) НА ДАННЫЙ МОМЕНТ.  
// | |o| | | |o| |  
// |x| |x| |\*| |\*|  
 private boolean isCellUnderAtack(ChessBoardData xData, Cell cell, Player player) throws NoSuchDirectionException {  
 Cell tmpCell = new Cell();  
 if (player.isWhite()) {  
 tmpCell.setWhiteChecker();  
 } else {  
 tmpCell.setBlackChecker();  
 }  
// ближайшая вражеская шашка  
 for (Direction d : Direction.*values*()) {  
 if (getCellByD(xData, cell, d, 1).isOpposite(tmpCell) && getCellByD(xData, cell, getOppositeD(d), 1).isBlackCell()) {  
 return true;  
 }  
 }  
// поиск вражеской королевы  
 for (Direction d : Direction.*values*()) {  
 int i = 1;  
 while (getCellByD(xData, cell, d, i).isBlackCell()) {  
 i++;  
 }  
 if (getCellByD(xData, cell, d, i).isOpposite(tmpCell) && getCellByD(xData, cell, d, i).isQueen()) {  
 if (getCellByD(xData, cell, getOppositeD(d), 1).isBlackCell()) {  
 return true;  
 }  
 }  
 }  
 return false;  
 }  
  
 private Direction getOppositeD(Direction d) throws NoSuchDirectionException {  
 if (d == *LU*) {  
 return *RB*;  
 }  
 if (d == *RU*) {  
 return *LB*;  
 }  
 if (d == *RB*) {  
 return *LU*;  
 }  
 if (d == *LB*) {  
 return *RU*;  
 }  
 throw new NoSuchDirectionException();  
 }  
  
 private void resetTurkishArr() {  
 for (int i = 0; i < turkishArr.size(); i++) {  
 turkishArr.get(i).setBlackCell();  
 }  
 turkishArr.clear();  
 }  
  
 private boolean isSomeByD(ChessBoardData xData, Cell cell, Action act, Direction d) {  
 int i = 1;  
 while (getCellByD(xData, cell, d, i).isExist()) {  
 if (isRightActByD(xData, cell, getCellByD(xData, cell, d, i), d, act)) {  
 return true;  
 }  
 i++;  
 }  
 return false;  
 }  
  
 private boolean isSome(ChessBoardData xData, Cell cell, Action act) {  
 for (Direction d : Direction.*values*()) {  
 if (isSomeByD(xData, cell, act, d)) {  
 return true;  
 }  
  
 }  
 return false;  
 }  
  
 Cell getSome(ChessBoardData xData, Action act, Player pl) {  
 for (int i = 0; i < xData.cells.length; i++) {  
 if (xData.cells[i].isWhite() && isSome(xData, xData.cells[i], act) && pl.isWhite()) {  
 return xData.cells[i];  
 }  
 if (xData.cells[i].isBlack() && isSome(xData, xData.cells[i], act) && pl.isBlack()) {  
 return xData.cells[i];  
 }  
 }  
 return new Cell();  
 }  
  
 // return:  
// двидение - [targetCell, new Cell()]  
// битва - [targetCell, victimCell]  
// ничего - [new Cell(), new Cell()]  
 private Cell[] checkCells(ChessBoardData xData, Cell activeCell, Cell targetCell, Direction d, Action act) {  
 // проверка того, какое будет движение в следующую точку  
 Cell victimCell;  
 //движение  
 if (act.isMove()) {  
 //проверка правельного направления шашки  
 if (((activeCell.isBlackChecker() || activeCell.isBlackActiveChecker()) && !d.isBlackD())  
 || ((activeCell.isWhiteChecker() || activeCell.isWhiteActiveChecker()) && !d.isWhiteD())) {  
 return new Cell[]{new Cell(), new Cell()};  
 }  
  
 // шаг шашки  
 if (activeCell.isChecker()) {  
 if (getCellByD(xData, activeCell, d, 1).isBlackCell()) {  
 if (getCellByD(xData, activeCell, d, 1).equals(targetCell)) {  
 return new Cell[]{targetCell, new Cell()};  
 }  
 }  
 }  
 // шаг королевы  
 if (activeCell.isQueen()) {  
 int deep = 1;  
 while (getCellByD(xData, activeCell, d, deep).isBlackCell()) {  
 if (getCellByD(xData, activeCell, d, deep).equals(targetCell)) {  
 return new Cell[]{targetCell, new Cell()};  
 }  
 deep++;  
 }  
 }  
 }  
 //битва  
 if (act.isFight()) {  
 //шашка  
 if (activeCell.isChecker()) {  
 if (getCellByD(xData, activeCell, d, 1).isOpposite(activeCell)) {  
 if (getCellByD(xData, activeCell, d, 2).isBlackCell() && getCellByD(xData, activeCell, d, 2).equals(targetCell)) {  
 return new Cell[]{targetCell, getCellByD(xData, activeCell, d, 1)};  
 }  
 }  
 }  
 //королева  
 if (activeCell.isQueen()) {  
 int deep = 1;  
 while (getCellByD(xData, activeCell, d, deep).isBlackCell()) {  
 deep++;  
 }  
 victimCell = getCellByD(xData, activeCell, d, deep);  
 if (victimCell.isOpposite(activeCell)) {  
 deep = 1;  
 while (getCellByD(xData, victimCell, d, deep).isBlackCell()) {  
 if (getCellByD(xData, victimCell, d, deep).equals(targetCell)) {  
 return new Cell[]{targetCell, victimCell};  
 }  
 deep++;  
 }  
 }  
 }  
 }  
 return new Cell[]{new Cell(), new Cell()};  
 }  
  
 private boolean isRightActByD(ChessBoardData xData, Cell activeCell, Cell targetCell, Direction d, Action act) {  
 // выбор правельное направление  
 Cell actCells[] = checkCells(xData, activeCell, targetCell, d, act);  
 // движение  
 if (act.isMove() && actCells[0].isExist() && (!actCells[1].isExist())) {  
 return true;  
 }  
 // битва  
 if (act.isFight() && actCells[0].isExist() && actCells[1].isExist()) {  
 return true;  
 }  
 return false;  
 }  
  
 public Direction getDbyTarget(ChessBoardData xData, Cell activeCell, Cell targetCell) throws NoSuchDirectionException { // задает направление с целью атаковать  
 for (Direction d : Direction.*values*()) {  
 int i = 1;  
 while (getCellByD(xData, activeCell, d, i).isExist()) {  
 if (targetCell.equals(getCellByD(xData, activeCell, d, i))) {  
 return d;  
 }  
 i++;  
 }  
// System.out.println("active index, cx, cy" + "\n" + activeCell.index + " " + activeCell.cX + " " + activeCell.cY);  
// System.out.println("target index, cx, cy" + "\n" + targetCell.index + " " + targetCell.cX + " " + targetCell.cY);  
 }  
 throw new NoSuchDirectionException();  
 }  
  
 */\*\*  
 \*  
 \*/* Cell getCellByXY(ChessBoardData xData, int clickedX, int clickedY) {  
 for (Cell cell : xData.cells) {  
 if ((clickedX >= (cell.cX))  
 && (clickedX < (cell.cX + xData.CELL\_SIZE))  
 && (clickedY >= (cell.cY))  
 && (clickedY < (cell.cY + xData.CELL\_SIZE))) {  
 return cell;  
 }  
 }  
 return new Cell();  
 }  
  
 */\*\*  
 \*  
 \*/* public Cell getCellByXY(int clickedX, int clickedY) {  
 for (Cell cell : data.cells) {  
 if ((clickedX >= (cell.cX))  
 && (clickedX < (cell.cX + data.CELL\_SIZE))  
 && (clickedY >= (cell.cY))  
 && (clickedY < (cell.cY + data.CELL\_SIZE))) {  
 return cell;  
 }  
 }  
 return new Cell();  
 }  
  
 private Cell getCellByD(ChessBoardData xData, Cell cell, Direction d, int deep) {  
 return getCellByXY(xData, cell.cX + xData.CELL\_SIZE \* d.kX \* deep, cell.cY + xData.CELL\_SIZE \* d.kY \* deep);  
 }  
  
 private boolean checkSetQeen(Cell cell) { // устанавливает шашку как королеву  
 String userIndexQ[] = {"a8", "b8", "c8", "d8", "e8", "f8", "g8", "h8"};  
 String compIndexQ[] = {"a1", "b1", "c1", "d1", "e1", "f1", "g1", "h1"};  
 if (cell.isWhiteChecker()) {  
 for (String uIndex : userIndexQ) {  
 if (uIndex.equals(cell.index)) {  
 cell.setWhiteQueen();  
 return true;  
 }  
 }  
 }  
 if (cell.isBlackChecker()) {  
 for (String cIndex : compIndexQ) {  
 if (cIndex.equals(cell.index)) {  
 cell.setBlackQueen();  
 return true;  
 }  
 }  
 }  
 return false;  
 }  
  
 private boolean isQueenIndex(Cell cell) { // проверка находиться ли шашка на клетке превращающей в королеву  
 String whiteIndexQueen[] = {"a8", "b8", "c8", "d8", "e8", "f8", "g8", "h8"};  
 String blackIndexQueen[] = {"a1", "b1", "c1", "d1", "e1", "f1", "g1", "h1"};  
 if (cell.isBlackChecker()) {  
 for (String bIndex : blackIndexQueen) {  
 if (bIndex.equals(cell.index)) {  
 return true;  
 }  
 }  
 }  
 if (cell.isWhiteChecker()) {  
 for (String wIndex : whiteIndexQueen) {  
 if (wIndex.equals(cell.index)) {  
 return true;  
 }  
 }  
 }  
 return false;  
 }  
  
 private boolean isCentralField(Cell cell) { // положение шашки в центре  
 String centralFields[] = {"b2", "b4", "b6", "c3", "c5", "c7", "d2", "d4", "d6", "e3", "e3", "e7", "f2", "f4", "f6", "g3", "g5", "g7"};  
  
 for (String cetralField : centralFields) {  
 if (cetralField.equals(cell.index)) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 void customResult() { //тут определяются варианты окончания игры, убиты или блокированы  
 data.setCheckersNum();  
// data.notifyUpdateGUI();  
  
 if (data.blackCheckers == 0) {  
 data.gameOver = true;  
 data.notifyNoBlackCkeckersLeft();  
 return;  
 }  
 if (data.whiteCheckers == 0) {  
 data.gameOver = true;  
 data.notifyNoWhiteCkeckersLeft();  
 return;  
 }  
  
 if (!getSome(this.data, Action.*FIGHT*, Player.*BLACK*).isExist() && !getSome(this.data, Action.*MOVE*, Player.*BLACK*).isExist() && data.blackCheckers != 0) {  
 data.gameOver = true;  
 data.notifyBlackIsBlocked();  
 return;  
 }  
 if (!getSome(this.data, Action.*FIGHT*, Player.*WHITE*).isExist() && !getSome(this.data, Action.*MOVE*, Player.*WHITE*).isExist() && data.whiteCheckers != 0) {  
 data.gameOver = true;  
 data.notifyWhiteIsBlocked();  
 return;  
 }  
 data.resultBuf = "";  
 }  
  
 Cell getCellByIndex(String index) {  
 for (int i = 0; i < data.cells.length; i++) {  
 if (index.equals(data.cells[i].index)) {  
 return data.cells[i];  
 }  
 }  
 return new Cell();  
 }  
  
 public Logic() {  
 this.data = new ChessBoardData();  
 (new Thread(new ObservePlayerQueue(this))).start();  
 }  
}

* **DataListener.java**

package model;  
  
import java.util.EventListener;  
  
public interface DataListener extends EventListener {  
  
 void updateTextGuiLanguageInfo(UpdateGuiEvent e);  
  
 void updateGUI(UpdateGuiEvent e);  
  
 void noBlackCkeckersLeft(UpdateGuiEvent e);  
  
 void noWhiteCkeckersLeft(UpdateGuiEvent e);  
  
 void blackIsBlocked(UpdateGuiEvent e);  
  
 void whiteIsBlocked(UpdateGuiEvent e);  
}

* **ChessBoardData.java**
* package model;  
    
  public class ObservePlayerQueue implements Runnable { // определение хода комьютера  
    
   private Logic logic;  
    
   public void run() {  
   try {  
   while (!logic.data.gameExit) {  
   Thread.*sleep*(1000);  
   if ((!logic.data.whiteIsHuman && logic.data.whiteSessionContinue && !logic.data.gameOver)  
   || (!logic.data.blackIsHuman && logic.data.blackSessionContinue && !logic.data.gameOver)) {  
   logic.compStep(logic.data);  
   }  
   }  
   } catch (InterruptedException ex) {  
   ex.printStackTrace();  
   }  
    
   }  
    
   public ObservePlayerQueue(Logic logic) {  
   this.logic = logic;  
   }  
  }
* **Cell.java**
* package model;  
    
  import model.ChessBoardData.Status;  
    
  import static model.ChessBoardData.Status.\*;  
    
  public class Cell {  
    
   public String index;  
   public int cX;  
   public int cY;  
   private ChessBoardData.Status status;  
    
   public Status getStatus() {  
   return this.status;  
   }  
    
   public void setStatus(Status status) {  
   this.status = status;  
   }  
    
   public boolean isExist() {  
   return (status != *NILL*);  
   }  
    
   public boolean isWhite() {  
   return (status == *WHITE\_CH* || status == *WHITE\_ACH* || status == *WHITE\_Q* || status == *WHITE\_AQ*);  
   }  
    
   public boolean isBlack() {  
   return (status == *BLACK\_CH* || status == *BLACK\_ACH* || status == *BLACK\_Q* || status == *BLACK\_AQ*);  
   }  
    
   public void resetActive() {  
   if (status == *WHITE\_ACH*) {  
   status = *WHITE\_CH*;  
   return;  
   }  
   if (status == *WHITE\_AQ*) {  
   status = *WHITE\_Q*;  
   return;  
   }  
   if (status == *BLACK\_ACH*) {  
   status = *BLACK\_CH*;  
   return;  
   }  
   if (status == *BLACK\_AQ*) {  
   status = *BLACK\_Q*;  
   return;  
   }  
   }  
    
   public void setActive() {  
   if (status == *WHITE\_CH*) {  
   status = *WHITE\_ACH*;  
   return;  
   }  
   if (status == *WHITE\_Q*) {  
   status = *WHITE\_AQ*;  
   return;  
   }  
   if (status == *BLACK\_CH*) {  
   status = *BLACK\_ACH*;  
   return;  
   }  
   if (status == *BLACK\_Q*) {  
   status = *BLACK\_AQ*;  
   return;  
   }  
   }  
    
   public boolean isBlackCell() {  
   return (status == *BC*);  
   }  
    
   boolean isBlackQueen() {  
   return (status == *BLACK\_Q*);  
   }  
    
   boolean isBlackChecker() {  
   return (status == *BLACK\_CH*);  
   }  
    
   public boolean isBlackActiveQueen() {  
   return (status == *BLACK\_AQ*);  
   }  
    
   public boolean isBlackActiveChecker() {  
   return (status == *BLACK\_ACH*);  
   }  
    
   boolean isWhiteQueen() {  
   return (status == *WHITE\_Q*);  
   }  
    
   public boolean isWhiteActiveQueen() {  
   return (status == *WHITE\_AQ*);  
   }  
    
   boolean isWhiteChecker() {  
   return (status == *WHITE\_CH*);  
   }  
    
   public boolean isWhiteActiveChecker() {  
   return (status == *WHITE\_ACH*);  
   }  
    
   boolean isActive() {  
   return (status == *WHITE\_ACH* || status == *WHITE\_AQ* || status == *BLACK\_ACH* || status == *BLACK\_AQ*);  
   }  
    
   public boolean isChecker() {  
   return (status == *WHITE\_CH* || status == *WHITE\_ACH* || status == *BLACK\_CH* || status == *BLACK\_ACH*);  
   }  
    
   public boolean isQueen() {  
   return (status == *WHITE\_Q* || status == *WHITE\_AQ* || status == *BLACK\_Q* || status == *BLACK\_AQ*);  
   }  
    
   boolean isTurkichChecker() {  
   return status == *TBCH*;  
   }  
    
   void setBlackCell() {  
   status = *BC*;  
   }  
    
   void setBlackQueen() {  
   status = *BLACK\_Q*;  
   }  
    
   void setBlackChecker() {  
   status = *BLACK\_CH*;  
   }  
    
   void setWhiteQueen() {  
   status = *WHITE\_Q*;  
   }  
    
   void setWhiteChecker() {  
   status = *WHITE\_CH*;  
   }  
    
   void setTurkichChecker() {  
   status = *TBCH*;  
   }  
    
   boolean isOpposite(Cell cell) {  
   if (this.isWhite() && cell.isBlack()) {  
   return true;  
   }  
   if (this.isBlack() && cell.isWhite()) {  
   return true;  
   }  
   return false;  
   }  
    
   boolean isOwn(Cell cell) {  
   if (this.isWhite() && cell.isWhite()) {  
   return true;  
   }  
   if (this.isBlack() && cell.isBlack()) {  
   return true;  
   }  
   return false;  
   }  
    
   Cell(String index, int cX, int cY, ChessBoardData.Status status) {  
   this.index = index;  
   this.cX = cX;  
   this.cY = cY;  
   this.status = status;  
   }  
    
   public Cell(ChessBoardData.Status status) {  
   this.status = status;  
   }  
    
   public Cell() {  
   this.status = *NILL*;  
   }  
  }

**Пакет controller**

* **ActionChessBoard.java**

package controller;  
  
import model.\*;  
import view.MainWindow;  
  
import java.awt.\*;  
import java.awt.event.\*;  
import java.net.URI;  
  
public class ActionChessBoard implements MouseListener, KeyListener, ActionListener, DataListener {  
  
 private Logic logic;  
 private ChessBoardData data;  
 private MainWindow mainW;  
  
 public ActionChessBoard(ChessBoardData data, Logic logic, MainWindow mainW) {  
 this.data = data;  
 this.logic = logic;  
 this.mainW = mainW;  
 }  
   
 public void mousePressed(MouseEvent e) {  
 if ((!data.whiteIsHuman) && (!data.blackIsHuman)) {  
 return;  
 }  
 if (e.getButton() == 1) {  
 Cell clickedCell = logic.getCellByXY(e.getX(), e.getY());  
  
 // Во время произведения сессии белых клик по черным ни к чему не приведет  
 if (logic.data.whiteSessionContinue) {  
 if (data.whiteIsHuman) {  
 if (!(clickedCell.isWhite() || clickedCell.isBlackCell())) {  
 return;  
 }  
 } else {  
 return;  
 }  
 }  
 if (logic.data.blackSessionContinue) {  
 if (data.blackIsHuman) {  
 if (!(clickedCell.isBlack() || clickedCell.isBlackCell())) {  
 return;  
 }  
 } else {  
 return;  
 }  
 }  
  
 // выбранная шашка становиться активной  
 if (!existActiveChecker()) {  
 clickedCell.setActive();  
 return;  
 }  
  
 // выбранная активная шашка становиться неактивной  
 if (existActiveChecker()) {  
 if (getActiveChecker().equals(clickedCell)) {  
 clickedCell.resetActive();  
 return;  
 }  
 }  
  
  
 // если нажать на неактивную и какая-то шашка уже активна, она становиться неактивной а выбранная становиться активной  
 if (existActiveChecker() && (clickedCell.isChecker() || clickedCell.isQueen())) {  
 getActiveChecker().resetActive();  
 clickedCell.setActive();  
 return;  
 }  
  
 // выберает для активной шашки целевую клетку  
 if (existActiveChecker()) {  
 Cell activeCell = getActiveChecker();  
 Cell targetCell = clickedCell;  
 logic.userStep(activeCell, targetCell);  
 return;  
 }  
 }  
  
 // правая кнока мыши делает все неактивным  
 if (e.getButton() == 3) {  
 getActiveChecker().resetActive();  
 return;  
 }  
  
  
 }  
  
 public void actionPerformed(ActionEvent ev) {  
 try {  
 if (ev.getActionCommand().equals(data.exitTitle)) { // закончить игру  
 data.setGameExit();  
 System.*exit*(0);  
 }  
  
 if (ev.getActionCommand().equals(data.aboutTitle)) { // About  
 mainW.showAbout();  
 }  
  
 if (ev.getActionCommand().equals(data.rulesTitle)) { // правила  
 Desktop desktop = Desktop.*getDesktop*();  
 desktop.browse(URI.*create*(data.rulesLink));  
 }  
  
 if (ev.getActionCommand().equals(data.newGameTitle)) { // новая игра, все откатывается к начальному виду  
 data.restartGame();  
 mainW.modalFrameDispose();  
 mainW.clearTextArea();  
 mainW.updateTextGuiLabels();  
 }  
  
 if (ev.getActionCommand().equals("Русский")) {  
 data.setRussianLang();  
 }  
  
 if (ev.getActionCommand().equals("English")) {  
 data.setEnglishLang();  
 }  
  
 if (ev.getActionCommand().equals(data.userVScompTitle)) {  
 data.setUSERvsCOMP();  
 }  
 if (ev.getActionCommand().equals(data.compVSuserTitle)) {  
 data.setCOMPvsUSER();  
 }  
 if (ev.getActionCommand().equals(data.compVScompTitle)) {  
 data.setCOMPvsCOMP();  
 }  
 if (ev.getActionCommand().equals(data.userVSuserTitle)) {  
 data.setUSERvsUSER();  
 }  
 } catch (Exception ex) {  
 ex.printStackTrace();  
 }  
  
 }  
  
 public void noBlackCkeckersLeft(UpdateGuiEvent e) {  
 mainW.noBlackCkeckersLeft();  
 }  
  
 public void noWhiteCkeckersLeft(UpdateGuiEvent e) {  
 mainW.noWhiteCkeckersLeft();  
 }  
  
 public void blackIsBlocked(UpdateGuiEvent e) {  
 mainW.blackIsBlocked();  
 }  
  
 public void whiteIsBlocked(UpdateGuiEvent e) {  
 mainW.whiteIsBlocked();  
 }  
  
 public void updateGUI(UpdateGuiEvent e) {  
 mainW.updateGui();  
 }  
  
 public void updateTextGuiLanguageInfo(UpdateGuiEvent e) {  
 mainW.updateTextGuiLanguageInfo();  
 }  
  
 private Cell getActiveChecker() {  
 for (Cell cell : data.cells) {  
 if (data.whiteIsHuman) {  
 if (cell.isWhiteActiveChecker() || cell.isWhiteActiveQueen()) {  
 return cell;  
 }  
 }  
 if (data.blackIsHuman) {  
 if (cell.isBlackActiveChecker() || cell.isBlackActiveQueen()) {  
 return cell;  
 }  
 }  
 }  
 return new Cell();  
 }  
  
 private boolean existActiveChecker() {  
 if (getActiveChecker().isExist()) {  
 return true;  
 }  
 return false;  
 }  
  
 public void keyPressed(KeyEvent e) {  
 if (e.getKeyCode() == KeyEvent.*VK\_ESCAPE*) {  
 System.*exit*(0);  
 }  
 }  
  
 public void keyTyped(KeyEvent e) {  
 }  
  
 public void keyReleased(KeyEvent e) {  
 }  
  
 public void mouseClicked(MouseEvent e) {  
 }  
  
 public void mouseReleased(MouseEvent e) {  
 }  
  
 public void mouseEntered(MouseEvent e) {  
 }  
  
 public void mouseExited(MouseEvent e) {  
 }  
}

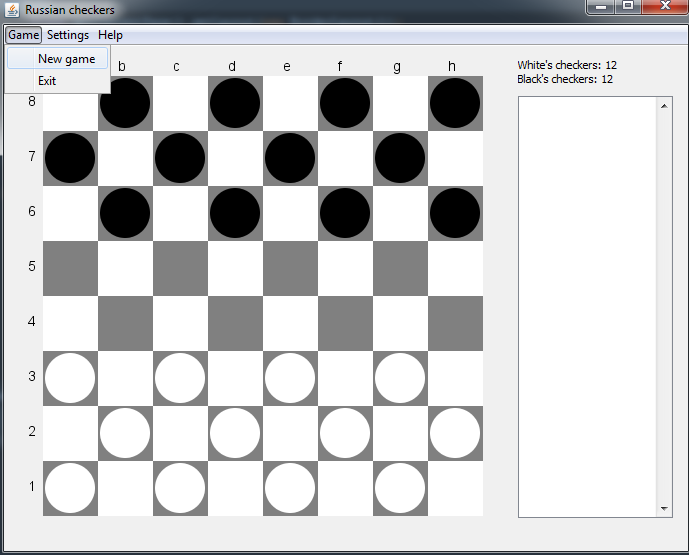
1. **Тесты**

Функции forException1 и forException2 в пакете java.testPackage проверяет правельность направления движения, forKill проверяет снятие вражеской шашки, noMoveWhileAttack проверяет необходимость перед движением снять возможную шашку, turkish проверяет снятие нескольких шашек за 1 ход.

Суть тестов заключается в том, что предлагается ситуация на игровом поле и алгоритм должен совершить правильный ход.

package testPackage  
import model.Cell  
import model.ChessBoardData  
import model.Logic  
import model.NoSuchDirectionException  
import org.junit.jupiter.api.Assertions.assertThrows  
import org.junit.jupiter.api.Test  
  
  
class Test {  
 private val logic = Logic()  
 private val data = ChessBoardData()  
  
 @Test  
 fun forException1() { //направление движения  
 var act = Cell()  
 var targ = Cell()  
 val map = *listOf*(Pair(42, 25)) // 42-откуда, 25-куда  
 act = logic.data.cells[map[0].first]  
 targ = logic.data.cells[map[0].second]  
 assertThrows(NoSuchDirectionException::class.*java*) **{** logic.getDbyTarget(logic.data, act, targ) **}** }  
  
 @Test  
 fun forException2() { //направление движения  
 var act = Cell()  
 var targ = Cell()  
 val map = *listOf*(Pair(42, 32))  
 act = logic.data.cells[map[0].first]  
 targ = logic.data.cells[map[0].second]  
 assertThrows(NoSuchDirectionException::class.*java*) **{** logic.getDbyTarget(logic.data, act, targ) **}** }  
  
 enum class Status {  
 BC, // черная клетка  
 }  
  
 @Suppress("INACCESSIBLE\_TYPE")  
 @Test  
 fun forKill() { //убийство шашки  
 // e3 = 44 f4 = 37, f6 = 21 g5 = 30, g3 = 46 h4 = 39, g5 = 30 e3 = 44  
 val map = *listOf*(Pair(44, 37), Pair(21, 30), Pair(46, 39), Pair(30, 44))  
 var act = Cell()  
 var targ = Cell()  
 for (i in map.*indices*) {  
 act = logic.data.cells[map[i].first]  
 targ = logic.data.cells[map[i].second]  
 logic.getDbyTarget(logic.data, act, targ)  
 }  
 if (Status.BC.toString() == logic.data.cells[37].*status*.name) {  
 *println*("Ok")  
 }  
 }  
  
 @Suppress("INACCESSIBLE\_TYPE")  
 @Test  
 fun turkish() { // есть две шашки за 1 ход  
 // w b w b w b  
 // c3=42:d4=35 b6=17:a5=24 b2=49:c3=42 h6=23:g5=30 d4=35:e5=28 f6=21:d4=35:b2=49  
 val map = *listOf*(Pair(42, 35), Pair(17, 24), Pair(49, 42), Pair(23, 30), Pair(35, 28), Pair(21, 35), Pair(35, 49))  
 var act = Cell()  
 var targ = Cell()  
 for (i in map.*indices*) {  
 act = logic.data.cells[map[i].first]  
 targ = logic.data.cells[map[i].second]  
 logic.userStep(act, targ)  
 }  
 if (Status.BC.toString() == logic.data.cells[28].*status*.name && Status.BC.toString() == logic.data.cells[42].*status*.name) {  
 *println*("Ok")  
 }  
 }  
  
 @Suppress("INACCESSIBLE\_TYPE")  
 @Test  
 fun noMoveWhileAttack() { //блокировать ход если есть возможность атаки  
 // e3 = 44 f4 = 37, f6 = 21 g5 = 30, g3 = 46 h4 = 39, g5 = 30 e3 = 44  
 val map = *listOf*(Pair(44, 37), Pair(21, 30), Pair(46, 39), Pair(19, 28))  
 var act = Cell()  
 var targ = Cell()  
 for (i in map.*indices*) {  
//  
 act = logic.data.cells[map[i].first]  
 targ = logic.data.cells[map[i].second]  
 logic.getDbyTarget(logic.data, act, targ)  
 }  
 if (Status.BC.toString() == logic.data.cells[28].*status*.name) {  
 *println*("Ok")  
 }  
 }  
}

1. **Скриншоты программы**

****

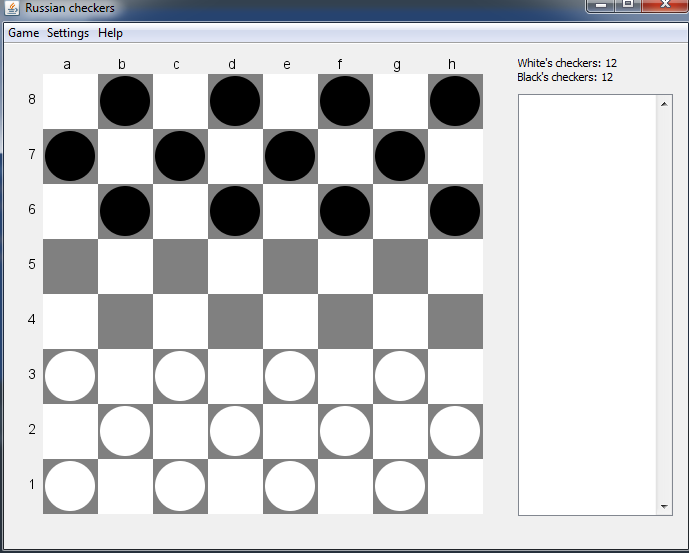
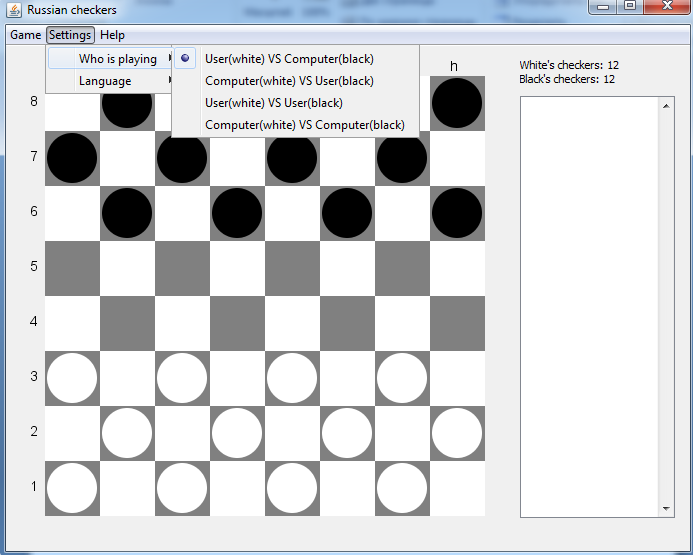
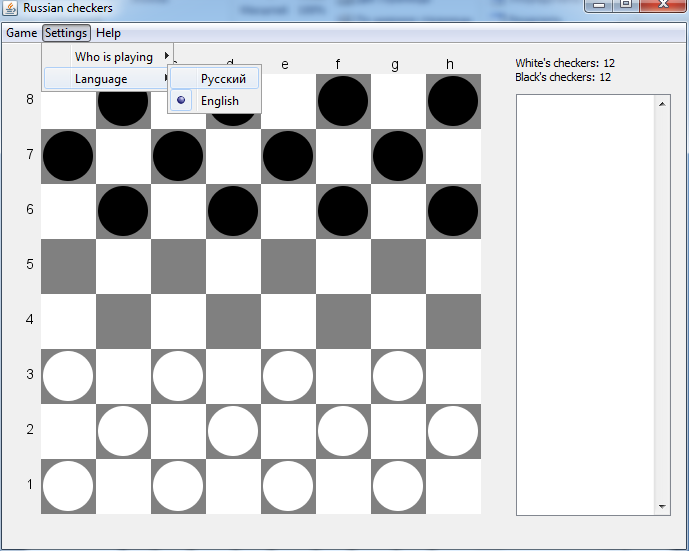
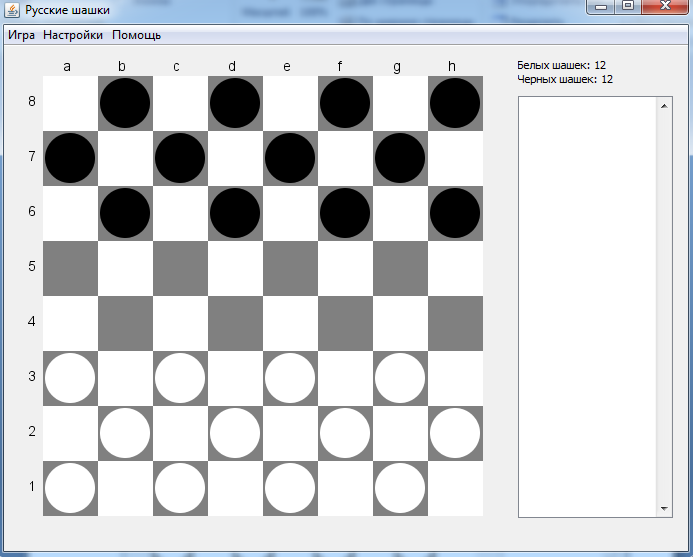
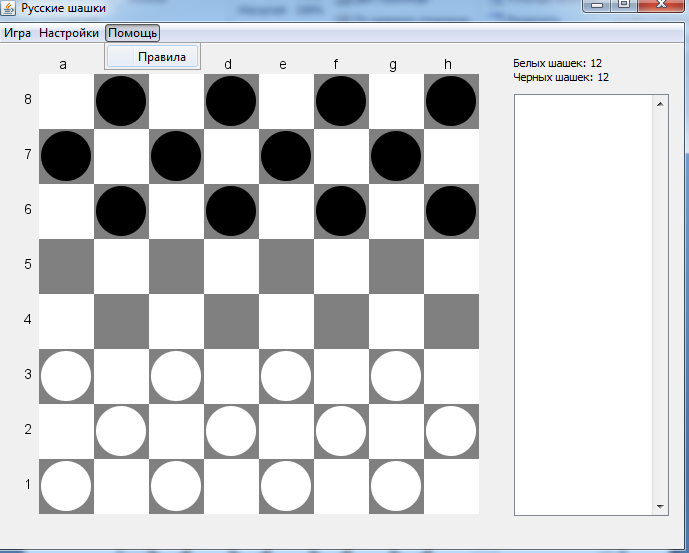


Рисунок 5.1 Старт игры









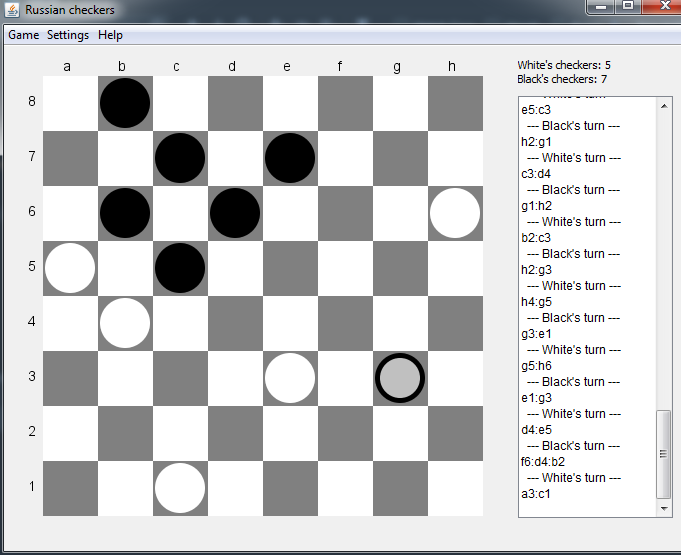


Рисунок 5.2 – Середина игры

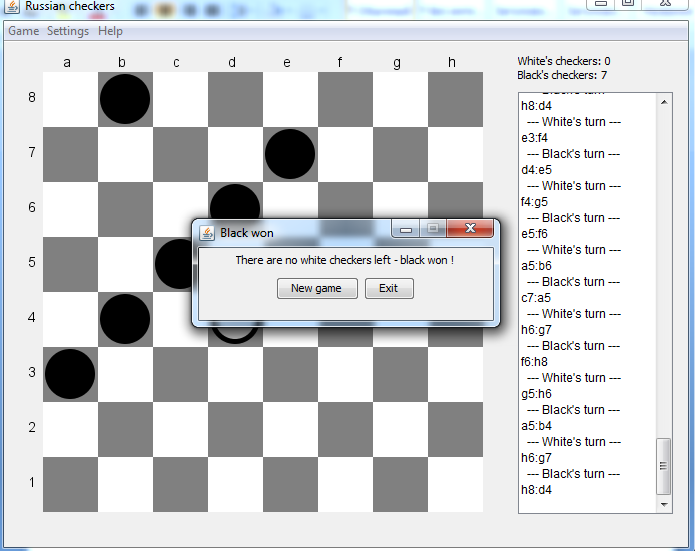


Рисунок 5.3 – Конец игры