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 Reversi на языке Kotlin

Выполнил студент гр. 3530901/80003 Тарасенко Н.С.

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 Reversi, которая представляет из себя настольную игру для двух человек на доске 8 на 8 клеток.

В игре используется квадратная доска размером 8 × 8 клеток (все клетки могут быть одного цвета) и 64 специальные фишки, окрашенные с разных сторон в контрастные цвета, например, в белый и чёрный. Один из игроков играет белыми, другой — чёрными. Делая ход, игрок ставит фишку на клетку доски «своим» цветом вверх.

В начале игры в центр доски выставляются 4 фишки: чёрные на *d5* и *e4,* белые на *d4* и *e5.*

GitHub репозиторий: <https://github.com/GetRhymes/Reversi>

1. **Метод решения**

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

Весь код разбит на 2 пакета: GUI и Logic

Пакет GUI отвечает за визуальное представление приложения. Он содержит функцию main, задание параметров окна и расположение внутри него вс ех элементов графического интерфейс а, а такжа события на взаимодействие с этими элементами. Так же содержит классы Finish (отвечает за окно всплывающее по окончанию игры), Help (правила игры), MainMenu (главное меню), PlayingField (окно на котором происходят действия игры), Records (статистика матчей)

Пакет Logic отвечает за логическую составляющую приложения. Он содержит внутри себя 2 класса: Board (прописаны правила и возможности игры), Opponent (отвечает за действия бота и возможности подсказок)

Когда начинается игра, первым делает ход игрок, который играет за белых, ему высвечиваются все возможные ходы и, опираясь на личный опыт и подсказки, совершает первый ход, после алгоритм высчитывает наиболее выгодный ход в данный момент, просчитывая наибольшее кол-во фишек, которые он сможет перевернуть в течение данного хода и ставит свою фишку 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, вниз, влево, вправо, вверх-влево, вверх-вправо, вниз-вправо, вниз-влево.

Например, разберем одно из направлений: «вверх»

1. Идет проверка на какой позиции стоит фишка соперника, в случае если она стоит на грани, то ход не возможен.
2. Если между фишкой соперника и гранью поля есть место, то проверяется след позиция на поле после фишки соперника, если там фишка соперника так происходит пока там не будет либо грани либо пустого места. В случае пустого пространства на его место ставится мнимая фишка и считается кол-во фишек которые перевернуться между мнимой и изначальной. Эта информация записывается в коллекции вместе с координатами мнимой точки.
3. В коллекции ищется максимальное значение и метод выводит направления и координаты, благодаря которым изменяется GUI поле
4. **Листинг программы**

**Пакет GUI**

* **Finish.kt**

1. import javafx.geometry.Pos  
   import javafx.scene.text.FontWeight  
   import tornadofx.\*  
   import java.io.File  
     
   class Finish : Fragment("Finish") {  
    override val root = *anchorpane* **{** setPrefSize(800.0, 600.0)  
    setMaxSize(800.0, 600.0)  
    setMinSize(800.0, 600.0)  
    *vbox* **{** *label*("GAME OVER") **{** setPrefSize(400.0, 200.0)  
    setMaxSize(400.0, 200.0)  
    setMinSize(400.0, 200.0)  
    *alignment* = Pos.CENTER  
    *translateX* = 200.0  
    *translateY* = 50.0  
    *style* **{** fontSize = 40.*px* fontWeight = FontWeight.EXTRA\_BOLD  
    **}  
    }** *hbox* **{** *vbox* **{** *translateX* = 200.0  
    *label*(*name*) **{** setPrefSize(150.0, 75.0)  
    setMaxSize(150.0, 75.0)  
    setMinSize(150.0, 75.0)  
    *alignment* = Pos.CENTER  
    *style* **{** fontSize = 36.*px* fontWeight = FontWeight.EXTRA\_BOLD  
    **}  
    }** *label*(*playerFP*) **{** setPrefSize(150.0, 75.0)  
    setMaxSize(150.0, 75.0)  
    setMinSize(150.0, 75.0)  
    *alignment* = Pos.TOP\_CENTER  
    *style* **{** fontSize = 36.*px* fontWeight = FontWeight.EXTRA\_BOLD  
    **}  
    }  
    }** *vbox* **{** *translateX* = 300.0  
    *label*("Player2") **{** setPrefSize(150.0, 75.0)  
    setMaxSize(150.0, 75.0)  
    setMinSize(150.0, 75.0)  
    *alignment* = Pos.CENTER  
    *style* **{** fontSize = 36.*px* fontWeight = FontWeight.EXTRA\_BOLD  
    **}  
    }** *label*(*playerSP*) **{** setPrefSize(150.0, 75.0)  
    setMaxSize(150.0, 75.0)  
    setMinSize(150.0, 75.0)  
    *alignment* = Pos.TOP\_CENTER  
    *style* **{** fontSize = 36.*px* fontWeight = FontWeight.EXTRA\_BOLD  
    **}  
    }  
    }  
    }** *hbox* **{** *translateY* = 65.0  
    *translateX* = 450.0  
    *button*("Главное меню") **{** setPrefSize(150.0, 25.0)  
    setMaxSize(150.0, 25.0)  
    setMinSize(150.0, 25.0)  
    *action* **{** writeFile()  
    close()  
    find<MainMenu>().openWindow()  
    **}  
     
    }** *button*("Выход") **{** *translateX* = 25.0  
    setPrefSize(150.0, 25.0)  
    setMaxSize(150.0, 25.0)  
    setMinSize(150.0, 25.0)  
    *action* **{** writeFile()  
    close()  
    **}  
    }  
    }  
    }  
    }** private fun writeFile() {  
    val listOfRecords = *mutableListOf*<Pair<String, Int>>()  
    var input = *listOf*<String>()  
    if (File("records").exists()) {  
    input = File("records").*readLines*()  
    } else File("records").createNewFile()  
    //println(input)  
    for (i in input) {  
    val parts = i.*split*(" ")  
    listOfRecords.add(Pair(parts[0], parts[1].*toInt*()))  
    }  
    listOfRecords.add(Pair(*name*, *playerFP*.*toInt*()))  
    val out = File("records").*bufferedWriter*()  
    for (i in listOfRecords.*sortedBy* **{ it**.second **}**) {  
    out.write("${i.first} ${i.second}")  
    out.newLine()  
    }  
    out.close()  
    }  
   }

* **Help.kt**
* class Help : View("My View") {  
   override val root = *anchorpane* **{** setPrefSize(800.0, 700.0)  
   setMaxSize(800.0, 700.0)  
   setMinSize(800.0, 700.0)  
   *vbox* **{** *label*("Правила игры") **{** *translateX* = 300.0  
   setPrefSize(200.0, 50.0)  
   setMaxSize(200.0, 50.0)  
   setMinSize(200.0, 50.0)  
   *alignment* = Pos.CENTER  
   *style* **{** fontSize = 24.*px* fontWeight = FontWeight.BOLD  
   **}  
   }** *textarea*("В игре используется квадратная доска размером 8х8 (все клетки могут быть одного цвета) и 64\n" +  
   "специальные фишки, окрашенные с разных сторон в контрастные цвета,\n " +  
   "например, в белый и чёрный. Клетки доски нумеруются от верхнего левого угла: вертикали - латинскими буквами,\n" +  
   " горизонтали - цифрами. Один из игроков играет белыми, другой - чёрными.\n" +  
   " Делая ход, игрок ставит фишку на клетку доски \"своим\" цветом вверх.\n" +  
   "В начале игры в центр доски выставляются 4 фишки: чёрные на d5 и e4, белые \n" +  
   "на d4 и e5. Первый ход делают чёрные. Далее игроки ходят по очереди.\n" +  
   "Делая ход, игрок должен поставить свою фишку на одну из клеток доски таким образом,\n" +  
   " чтобы между этой поставленной фишкой и одной из имеющихся уже на доске фишек его \n " +  
   "цвета находился непрерывный ряд фишек соперника, горизонтальный, вертикальный или \n" +  
   "диагональный (другими словами, чтобы непрерывный ряд фишек соперника оказался \"закрыт\"\n" +  
   " фишками игрока с двух сторон). Все фишки соперника, входящие в \"закрытый\" на этом ходу ряд,\n" +  
   " переворачиваются на другую сторону (меняют цвет) и переходят к ходившему игроку.\n" +  
   "Если в результате одного хода закрывается одновременно более одного ряда фишек противника,\n" +  
   " то переворачиваются все фишки, оказавшиеся на всех \"закрытых\" рядах.\n" +  
   "Игрок вправе выбирать любой из возможных для него ходов. Если игрок имеет возможные ходы,\n" +  
   " он не может отказаться от хода. Если игрок не имеет допустимых ходов, то ход передаётся сопернику.\n" +  
   "Игра прекращается, когда на доску выставлены все фишки или когда ни один из игроков не может сделать \n" +  
   "хода. По окончании игры проводится подсчёт фишек каждого цвета, и игрок, чьих фишек на доске выставлено\n" +  
   " больше, объявляется победителем. В случае равенства количества фишек засчитывается ничья.")  
   **{** *translateX* = 100.0  
   setPrefSize(600.0, 575.0)  
   setMaxSize(600.0, 575.0)  
   setMinSize(600.0, 575.0)  
   *isWrapText* = true  
   *isDisable* = false  
   *isEditable* = false  
   **}** *button*("Назад") **{** setPrefSize(200.0, 30.0)  
   setMaxSize(200.0, 30.0)  
   setMinSize(200.0, 30.0)  
   *translateX* = 50.0  
   *translateY* = 25.0  
   *action* **{** close()  
   val window = find<MainMenu>()  
   window.openWindow()  
   window.currentStage!!.*isResizable* = false  
   **}  
   }  
   }  
   }**}
* **Main.kt**
* import javafx.stage.Stage  
  import tornadofx.App  
    
   class Main: App(MainMenu::class) {  
   override val primaryView = MainMenu::class  
    
    
   override fun start(stage: Stage) {  
   stage.*isResizable* = false  
   super.start(stage)  
   }  
    
   }
* **MainMenu.kt**
* import javafx.beans.property.SimpleStringProperty  
  import javafx.geometry.Pos  
  import javafx.scene.control.Button  
  import javafx.scene.input.KeyEvent  
  import javafx.scene.text.FontWeight  
  import tornadofx.\*  
  import java.net.URI  
    
    
  var *name* = ""  
  class MainMenu : Fragment("Reversi") {  
   override val root = *borderpane* **{** primaryStage.*isResizable* = false  
   val nameS = SimpleStringProperty()  
   *style* **{** backgroundImage = *multi*(URI("/Pictures/reversiV2.jpg"))  
   **}** setPrefSize(800.0, 600.0)  
   setMaxSize(800.0, 600.0)  
   setMinSize(800.0, 600.0)  
   var buttonStart: Button by *singleAssign*()  
   *center* **{** *vbox* **{** *translateY* = 122.5 // отступ слева направо внутреннего контейнера, который содержит кнопки меню  
   setPrefSize(200.0, *minHeight*)  
   setMaxSize(200.0, *minHeight*)  
   setMinSize(200.0, *minHeight*)  
    
   *label*("Главное меню") **{** *translateX* = -100.0  
   *translateY* = -50.0 // смещение для лейбла "Главное меню"  
   setPrefSize(400.0, 50.0)  
   setMaxSize(400.0, 50.0)  
   setMinSize(400.0, 50.0)  
   *alignment* = Pos.CENTER  
   *style* **{** fontSize = 46.*px* fontWeight = FontWeight.BOLD  
   **}  
   }** *vbox* **{** *label*("Имя Игрока") **{** *translateY* = -5.0  
   *style* **{** fontWeight = FontWeight.BOLD  
   **}  
   }** *textfield*(nameS) **{** setPrefSize(200.0, 25.0)  
   setMaxSize(200.0, 25.0)  
   setMinSize(200.0, 25.0)  
   *promptText* = "Введите никнейм"  
   addEventFilter(KeyEvent.*ANY*) **{** *println*(nameS.*value*)  
   buttonStart.*isDisable* = !(nameS.*value* != "" && nameS.*value* != null)  
   **}  
   }** buttonStart = *button*("Играть") **{** *translateY* = 15.0  
   *isDisable* = true  
   setPrefSize(200.0, 50.0)  
   setMaxSize(200.0, 50.0)  
   setMinSize(200.0, 50.0)  
   *action* **{** *name* = nameS.*value* close()  
   val window = find<PlayingField>()  
   window.openWindow()  
   window.currentStage!!.*isResizable* = false  
   **}  
   }  
   }** *button*("Статистика") **{** *translateY* = 30.0  
   setPrefSize(200.0, 50.0)  
   setMaxSize(200.0, 50.0)  
   setMinSize(200.0, 50.0)  
   *action* **{** close()  
   val window = find<Records>()  
   window.openWindow()  
   window.currentStage!!.*isResizable* = false  
   **}  
   }** *button*("Справка") **{** *translateY* = 45.0  
   setPrefSize(200.0, 50.0)  
   setMaxSize(200.0, 50.0)  
   setMinSize(200.0, 50.0)  
   *action* **{** close()  
   val window = find<Help>()  
   window.openWindow()  
   window.currentStage!!.*isResizable* = false  
   **}  
   }  
   }  
   }  
   }**}
* **PlayingField.kt**

import Event.OpponentEvent  
import Logic.Board  
import Logic.Opponent  
import Logic.*conditional*import javafx.geometry.Pos  
import javafx.scene.control.Button  
import javafx.scene.control.Label  
import javafx.scene.paint.Paint  
import tornadofx.\*  
import java.net.URI  
import kotlin.random.Random  
  
var *playerFP* = "" // очки игрока  
var *playerSP* = "" // очки бота  
class PlayingField : Fragment("Reversi") {  
 private val opponentMoves = Opponent()  
  
  
 override val root = *anchorpane* **{** setPrefSize(800.0, 600.0)  
 setMaxSize(800.0, 600.0)  
 setMinSize(800.0, 600.0)  
 *vbox* **{** setPrefSize(800.0, 600.0)  
 setMaxSize(800.0, 600.0)  
 setMinSize(800.0, 600.0)  
 var player1: Label by *singleAssign*()  
 var player2: Label by *singleAssign*()  
 var player1Button: Button by *singleAssign*()  
 var player2Button: Button by *singleAssign*()  
 *anchorpane* **{** *hbox* **{** *vbox* **{** *translateX* = 180.0  
 *style* **{** backgroundColor = *multi*(Paint.valueOf("000000")) // черный цвет  
 **}** *hbox* **{** player1 = *label*("0") **{** *translateX* = -10.0  
 *style* **{** textFill = Paint.valueOf("FFFFFF") // белый цвет  
 fontSize = 24.*px* **}** setPrefSize(150.0, 50.0)  
 setMaxSize(150.0, 50.0)  
 setMinSize(150.0, 50.0)  
 *alignment* = Pos.CENTER\_RIGHT  
 **}** *anchorpane* **{** setPrefSize(50.0, 50.0)  
 setMaxSize(50.0, 50.0)  
 setMinSize(50.0, 50.0)  
 *style* **{** backgroundImage = *multi*(URI("/Pictures/whiteV2.png"))  
 **}  
 }  
 }** *hbox* **{** *label*(*name*) **{** *style* **{** textFill = Paint.valueOf("FFFFFF")  
 fontSize = 24.*px* **}** setPrefSize(150.0, 50.0)  
 setMaxSize(150.0, 50.0)  
 setMinSize(150.0, 50.0)  
 *alignment* = Pos.CENTER  
 **}** *anchorpane* **{** setPrefSize(50.0, 50.0)  
 setMaxSize(50.0, 50.0)  
 setMinSize(50.0, 50.0)  
  
 player1Button = *button* **{** *translateX* = 10.0  
 *translateY* = 10.0  
 setPrefSize(30.0, 30.0)  
 setMaxSize(30.0, 30.0)  
 setMinSize(30.0, 30.0)  
 *style* **{** backgroundColor = *multi*(Paint.valueOf("#00e676")) // светофор (чей ход) зеленый  
 shape = "M 200 300 A 50 50 0 1 1 350 300 A 50 50 0 1 1 200 300 " // изменение стандартной кнопки в круглую форму  
 **}  
 }  
 }  
 }  
 }** *vbox* **{** *style* **{** backgroundColor = *multi*(Paint.valueOf("000000"))  
 **}** *translateX* = 220.0  
 *hbox* **{** *anchorpane* **{** setPrefSize(50.0, 50.0)  
 setMaxSize(50.0, 50.0)  
 setMinSize(50.0, 50.0)  
 *style* **{** backgroundImage = *multi*(URI("/Pictures/blackV2.png"))  
 **}  
 }** player2 = *label*("0") **{** *translateX* = 10.0  
 *style* **{** textFill = Paint.valueOf("FFFFFF")  
 fontSize = 24.*px* **}** setPrefSize(150.0, 50.0)  
 setMaxSize(150.0, 50.0)  
 setMinSize(150.0, 50.0)  
 *alignment* = Pos.CENTER\_LEFT  
 **}  
 }** *hbox* **{** *anchorpane* **{** setPrefSize(50.0, 50.0)  
 setMaxSize(50.0, 50.0)  
 setMinSize(50.0, 50.0)  
  
 player2Button = *button* **{** *translateX* = 10.0  
 *translateY* = 10.0  
 setPrefSize(30.0, 30.0)  
 setMaxSize(30.0, 30.0)  
 setMinSize(30.0, 30.0)  
 *style* **{** backgroundColor = *multi*(Paint.valueOf("#f44336")) // красный  
 shape = "M 200 300 A 50 50 0 1 1 350 300 A 50 50 0 1 1 200 300 "  
 **}  
 }  
 }** *label*("Player2") **{** *style* **{** textFill = Paint.valueOf("FFFFFF")  
 fontSize = 24.*px* **}** setPrefSize(150.0, 50.0)  
 setMaxSize(150.0, 50.0)  
 setMinSize(150.0, 50.0)  
 *alignment* = Pos.CENTER  
 **}  
 }  
 }  
 }  
 }** *vbox* **{** var check = true  
 *translateY* = 25.0  
 *translateX* = 200.0  
  
 val arrayOfButton = *arrayOf*(  
 *arrayListOf*(Button(), Button(), Button(), Button(), Button(), Button() , Button(), Button()),  
 *arrayListOf*(Button(), Button(), Button(), Button(), Button(), Button() , Button(), Button()),  
 *arrayListOf*(Button(), Button(), Button(), Button(), Button(), Button() , Button(), Button()),  
 *arrayListOf*(Button(), Button(), Button(), Button(), Button(), Button() , Button(), Button()),  
 *arrayListOf*(Button(), Button(), Button(), Button(), Button(), Button() , Button(), Button()),  
 *arrayListOf*(Button(), Button(), Button(), Button(), Button(), Button() , Button(), Button()),  
 *arrayListOf*(Button(), Button(), Button(), Button(), Button(), Button() , Button(), Button()),  
 *arrayListOf*(Button(), Button(), Button(), Button(), Button(), Button() , Button(), Button())  
  
 )  
  
 fun parse(): Pair<Int, Int> {  
 var white = 0  
 var black = 0  
 for (line in arrayOfButton) {  
 for (button in line) {  
 if(button.*style*.length > 90) {  
 val part = button.*style*.toString().*split*(Regex("""/Pictures/"""))  
 val playerColor = part[1]  
 if(playerColor == "blackV2.png\");") black += 1  
 if(playerColor == "whiteV2.png\");") white += 1  
 }  
 }  
 }  
 return Pair(white, black)  
 }  
  
 val boardField = Board(*arrayOf*())  
 boardField.boardFillUp()  
 val playerBoard = Opponent()  
  
  
 fun reMakeTable() {  
 for (line in 0 *until* 8) {  
 for (elements in 0 *until* 8) {  
 if(boardField.board[line][elements] == 'W') {  
 arrayOfButton[line][elements].*style* **{** backgroundColor = if (line % 2 == 0 && elements % 2 == 0 || line % 2 != 0 && elements % 2 != 0)  
 *multi*(Paint.valueOf("#feffd9"))  
 else *multi*(Paint.valueOf("#5e1919"))  
 backgroundImage = *multi*(URI("/Pictures/whiteV2.png"))  
 //arrayOfButton[line][elements].action { println("GOOD") }  
 **}** }  
 if(boardField.board[line][elements] == 'B') {  
 arrayOfButton[line][elements].*style* **{** if (line % 2 == 0 && elements % 2 == 0 || line % 2 != 0 && elements % 2 != 0)  
 backgroundColor = *multi*(Paint.valueOf("#feffd9"))  
 else backgroundColor = *multi*(Paint.valueOf("#5e1919"))  
 backgroundImage = *multi*(URI("/Pictures/blackV2.png"))  
 **}** }  
 }  
 }  
 }  
 //board = arrayOf(  
 // charArrayOf('\_','\_','\_','\_','\_','\_','\_','\_'),  
 // charArrayOf('\_','\_','\_','\_','\_','\_','\_','\_'),  
 // charArrayOf('\_','\_','\_','\_','\_','\_','\_','\_'),  
 // charArrayOf('\_','\_','\_','B','W','\_','\_','\_'),  
 // charArrayOf('\_','\_','\_','W','B','\_','\_','\_'),  
 // charArrayOf('\_','\_','\_','\_','\_','\_','\_','\_'),  
 // charArrayOf('\_','\_','\_','\_','\_','\_','\_','\_'),  
 // charArrayOf('\_','\_','\_','\_','\_','\_','\_','\_')  
 //)  
  
 subscribe<OpponentEvent> **{** opponentMoves.move2(**it**.board, **it**.opponent, **it**.player)  
 if (*conditional*) {  
 reMakeTable()  
 check = true  
 player1Button.*style* **{** backgroundColor = *multi*(Paint.valueOf("#00e676"))  
 shape = "M 200 300 A 50 50 0 1 1 350 300 A 50 50 0 1 1 200 300 "  
 **}** player2Button.*style* **{** backgroundColor = *multi*(Paint.valueOf("#f44336"))  
 shape = "M 200 300 A 50 50 0 1 1 350 300 A 50 50 0 1 1 200 300 "  
 **}** val helpBoard = playerBoard.move(**it**.board, **it**.player, **it**.opponent)  
 for (line in arrayOfButton) {  
 for (button in line) {  
 if(button.*style*.length < 90) {  
 button.*isDisable* = true  
 }  
 }  
 }  
 for (position in helpBoard) {  
 if(arrayOfButton[position.key.first][position.key.second].*style*.length < 90) {  
 arrayOfButton[position.key.first][position.key.second].*isDisable* = position.value.isEmpty()  
 }  
 }  
 for (line in arrayOfButton) {  
 for (button in line) {  
 if (button.*style*.length > 90) {  
 button.*isDisable* = false  
 }  
 }  
 }  
 player1.*text* = (parse().first).toString()  
 player2.*text* = (parse().second).toString()  
  
 if (parse().first + parse().second == 64 || parse().first == 0 || parse().second == 0) {  
 *playerFP* = parse().first.toString()  
 *playerSP* = parse().second.toString()  
 close()  
 find<Finish>().openWindow()  
 }  
 } else {  
 player1Button.*style* **{** backgroundColor = *multi*(Paint.valueOf("#00e676"))  
 shape = "M 200 300 A 50 50 0 1 1 350 300 A 50 50 0 1 1 200 300"  
 **}** player2Button.*style* **{** backgroundColor = *multi*(Paint.valueOf("#f44336"))  
 shape = "M 200 300 A 50 50 0 1 1 350 300 A 50 50 0 1 1 200 300"  
 **}** val helpBoard = playerBoard.move(**it**.board, **it**.player, **it**.opponent)  
 for (line in arrayOfButton) {  
 for (button in line) {  
 if(button.*style*.length < 90) {  
 button.*isDisable* = true  
 }  
 }  
 }  
 for (position in helpBoard) {  
 if(arrayOfButton[position.key.first][position.key.second].*style*.length < 90) {  
 arrayOfButton[position.key.first][position.key.second].*isDisable* = position.value.isEmpty()  
 }  
 }  
 for (line in arrayOfButton) {  
 for (button in line) {  
 if (button.*style*.length > 90) {  
 button.*isDisable* = false  
 }  
 }  
 }  
 check = true  
 }  
 **}** for (line in 0 *until* boardField.board.size) {  
 *hbox* **{** for (element in 0 *until* boardField.board[line].size) {  
 val buttonBoard = *button* **{** *isDisable* = true  
 // #feffd9 беж  
 // #5e1919 борд  
 *style* **{** if (line % 2 == 0 && element % 2 == 0 || line % 2 != 0 && element % 2 != 0)  
 backgroundColor = *multi*(Paint.valueOf("#feffd9"))  
 else backgroundColor = *multi*(Paint.valueOf("#5e1919"))  
 **}** setPrefSize(50.0, 50.0)  
 setMinSize(50.0, 50.0)  
 setMaxSize(50.0, 50.0)  
  
 if (line == 3 && element == 3 || line == 4 && element == 4) { // начлаьные позиции фишек черных  
 *isDisable* = false  
 *style* **{** backgroundColor = *multi*(Paint.valueOf("#feffd9"))  
 backgroundImage = *multi*(URI("/Pictures/blackV2.png"))  
 **}** }  
 if (line == 4 && element == 3 || line == 3 && element == 4) { // начальные позиции фишек белых  
 *isDisable* = false  
 *style* **{** backgroundColor = *multi*(Paint.valueOf("#5e1919"))  
 backgroundImage = *multi*(URI("/Pictures/whiteV2.png"))  
 **}** }  
 *action* **{** if (*style*.length < 90) { // если < 90 то клетка пустая если > 90 то с фишкой  
 if (check) { // true - white // false - black  
 *style* **{** backgroundColor = if (line % 2 == 0 && element % 2 == 0 || line % 2 != 0 && element % 2 != 0)  
 *multi*(Paint.valueOf("#feffd9"))  
 else *multi*(Paint.valueOf("#5e1919"))  
 backgroundImage = *multi*(URI("/Pictures/whiteV2.png"))  
 **}** check = false  
 boardField.checkPoint(line, element, 'B', 'W')  
 reMakeTable()  
 player1Button.*style* **{** backgroundColor = *multi*(Paint.valueOf("#f44336"))  
 shape = "M 200 300 A 50 50 0 1 1 350 300 A 50 50 0 1 1 200 300 "  
 **}** player2Button.*style* **{** backgroundColor = *multi*(Paint.valueOf("#00e676"))  
 shape = "M 200 300 A 50 50 0 1 1 350 300 A 50 50 0 1 1 200 300 "  
 **}** *runLater*(Random.nextDouble(0.3, 1.0).*seconds*) **{** fire(OpponentEvent('W', boardField.board, 'B'))  
 **}** }  
 }  
 player1.*text* = (parse().first).toString()  
 player2.*text* = (parse().second).toString()  
 if (parse().first + parse().second == 64 || parse().first == 0 || parse().second == 0) {  
 *playerFP* = parse().first.toString()  
 *playerSP* = parse().second.toString()  
 close()  
 find<Finish>().openWindow()  
 }  
 **}  
 }** arrayOfButton[line][element] = buttonBoard  
 }  
 **}** }  
 val playerHelp = playerBoard.move(boardField.board, 'B', 'W' )  
 for (position in playerHelp) {  
 if(arrayOfButton[position.key.first][position.key.second].*style*.length < 90) {  
 arrayOfButton[position.key.first][position.key.second].*isDisable* = position.value.isEmpty()  
 }  
 }  
 **}** *hbox* **{** *translateY* = 65.0  
 *translateX* = 450.0  
 *button*("Главное меню") **{** setPrefSize(150.0, 25.0)  
 setMaxSize(150.0, 25.0)  
 setMinSize(150.0, 25.0)  
 *action* **{** close()  
 find<MainMenu>().openWindow()  
 **}  
  
 }** *button*("Выход") **{** *translateX* = 25.0  
 setPrefSize(150.0, 25.0)  
 setMaxSize(150.0, 25.0)  
 setMinSize(150.0, 25.0)  
 *action* **{** close()  
 **}  
 }  
 }  
 }  
 }**}

* **Records.kt**
* import javafx.geometry.Pos  
  import javafx.scene.text.FontWeight  
  import tornadofx.\*  
  import java.io.File  
    
  class Records : Fragment("Статистика") {  
   override val root = *gridpane* **{** setPrefSize(600.0,600.0)  
   setMaxSize(600.0,600.0)  
   setMinSize(600.0,600.0)  
   var rec = *listOf*<String>()  
   if (File("records").exists()) {  
   rec = File("records").*readLines*()  
   } else File("records").createNewFile()  
    
   *anchorpane* **{** *gridpaneConstraints* **{** columnRowIndex(0,0)  
   **}** setPrefSize(200.0,50.0)  
   setMaxSize(200.0,50.0)  
   setMinSize(200.0,50.0)  
    
   *label*("Номер") **{** setPrefSize(200.0,50.0)  
   setMaxSize(200.0,50.0)  
   setMinSize(200.0,50.0)  
   *style* **{** fontWeight = FontWeight.BOLD  
   fontSize = 24.*px* **}** *alignment* = Pos.CENTER  
   **}  
   }** *anchorpane* **{** *gridpaneConstraints* **{** columnRowIndex(1,0)  
   **}** setPrefSize(200.0,50.0)  
   setMaxSize(200.0,50.0)  
   setMinSize(200.0,50.0)  
    
   *label*("Участники") **{** setPrefSize(200.0,50.0)  
   setMaxSize(200.0,50.0)  
   setMinSize(200.0,50.0)  
   *style* **{** fontWeight = FontWeight.BOLD  
   fontSize = 24.*px* **}** *alignment* = Pos.CENTER  
   **}  
   }** *anchorpane* **{** *gridpaneConstraints* **{** columnRowIndex(2,0)  
   **}** setPrefSize(200.0,50.0)  
   setMaxSize(200.0,50.0)  
   setMinSize(200.0,50.0)  
    
   *label*("Счет") **{** setPrefSize(200.0,50.0)  
   setMaxSize(200.0,50.0)  
   setMinSize(200.0,50.0)  
   *style* **{** fontWeight = FontWeight.BOLD  
   fontSize = 24.*px* **}** *alignment* = Pos.CENTER  
   **}  
   }** *anchorpane* **{** *gridpaneConstraints* **{** columnRowIndex(0,1)  
   **}** setPrefSize(200.0, *minHeight*)  
   setMaxSize(200.0, *minHeight*)  
   setMinSize(200.0, *minHeight*)  
   *vbox* **{** for (i in 0 *until* rec.size) {  
   *label*("${i + 1}") **{** *translateX* = 100.0  
   *style* **{** fontWeight = FontWeight.BOLD  
   fontSize = 24.*px* **}** *alignment* = Pos.CENTER  
   **}** }  
   **}  
   }** *anchorpane* **{** *gridpaneConstraints* **{** columnRowIndex(1,1)  
   **}** setPrefSize(200.0, *minHeight*)  
   setMaxSize(200.0, *minHeight*)  
   setMinSize(200.0, *minHeight*)  
    
   *vbox* **{** for (i in rec.size - 1 *downTo* 0) {  
   val me = rec[i].*split*(" ")  
   *label*(me[0]) **{** *translateX* = 85.0  
   *style* **{** fontWeight = FontWeight.BOLD  
   fontSize = 24.*px* **}** *alignment* = Pos.CENTER  
   **}** }  
   **}  
   }** *anchorpane* **{** *gridpaneConstraints* **{** columnRowIndex(2,1)  
   **}** setPrefSize(200.0, *minHeight*)  
   setMaxSize(200.0, *minHeight*)  
   setMinSize(200.0, *minHeight*)  
    
   *vbox* **{** for (i in rec.size - 1 *downTo* 0) {  
   val me = rec[i].*split*(" ")  
   *label*(me[1]) **{** *translateX* = 85.0  
   *style* **{** fontWeight = FontWeight.BOLD  
   fontSize = 24.*px* **}** *alignment* = Pos.CENTER  
   **}** }  
   **}  
   }** *anchorpane* **{** *gridpaneConstraints* **{** columnRowIndex(0,2)  
   **}** setPrefSize(200.0, *minHeight*)  
   setMaxSize(200.0, *minHeight*)  
   setMinSize(200.0, *minHeight*)  
    
   *button*("Назад") **{** setPrefSize(200.0, 30.0)  
   setMaxSize(200.0, 30.0)  
   setMinSize(200.0, 30.0)  
   *translateX* = 50.0  
   *translateY* = 25.0  
   *action* **{** close()  
   val window = find<MainMenu>()  
   window.openWindow()  
   window.currentStage!!.*isResizable* = false  
   **}  
   }  
    
   }** *anchorpane* **{** *gridpaneConstraints* **{** columnRowIndex(1,2)  
   **}** setPrefSize(200.0, *minHeight*)  
   setMaxSize(200.0, *minHeight*)  
   setMinSize(200.0, *minHeight*)  
    
   **}** *anchorpane* **{** *gridpaneConstraints* **{** columnRowIndex(2,2)  
   **}** setPrefSize(200.0, *minHeight*)  
   setMaxSize(200.0, *minHeight*)  
   setMinSize(200.0, *minHeight*)  
    
   **}  
    
   }**}

**Пакет Logic**

* **Board.kt**

class Board(var board: Array<CharArray>) {  
  
 fun boardFillUp() {  
 board = *arrayOf*(  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', 'B', 'W', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', 'W', 'B', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_')  
 )  
 }  
 fun checkPoint(rowIndex: Int, columnIndex: Int, opponent: Char, player: Char) {  
 board[rowIndex][columnIndex] = player  
 if(rowIndex < 7 && columnIndex < 7 && board[rowIndex + 1][columnIndex + 1] == opponent) {  
 var rowIndexOp = rowIndex + 1  
 var columnIndexOp = columnIndex + 1  
 while(rowIndexOp < 8 || columnIndexOp < 8) {  
 while(board[rowIndexOp][columnIndexOp] == opponent) {  
 if(rowIndexOp != 7 && columnIndexOp != 7) {  
 rowIndexOp += 1  
 columnIndexOp += 1  
 } else break  
 }  
 val countRow = rowIndexOp - rowIndex  
 if (board[rowIndexOp][columnIndexOp] != '\_' && board[rowIndexOp][columnIndexOp] != opponent) {  
 for (i in 1..countRow) {  
 board[rowIndex + i][columnIndex + i] = player  
 }  
 }  
 break  
 }  
 }  
 if(rowIndex < 7 && board[rowIndex + 1][columnIndex] == opponent) {  
 var rowIndexOp = rowIndex + 1  
 while(rowIndexOp < 8) {  
 while(board[rowIndexOp][columnIndex] == opponent) {  
 if(rowIndexOp != 7) {  
 rowIndexOp += 1  
 } else break  
 }  
 val countRow = rowIndexOp - rowIndex  
 if (board[rowIndexOp][columnIndex] != '\_' && board[rowIndexOp][columnIndex] != opponent) {  
 for (i in 1..countRow) {  
 board[rowIndex + i][columnIndex] = player  
 }  
 }  
 break  
 }  
 }  
 if(columnIndex < 7 && board[rowIndex][columnIndex + 1] == opponent) {  
 var columnIndexOp = columnIndex + 1  
 while(columnIndexOp < 8) {  
 while(board[rowIndex][columnIndexOp] == opponent) {  
 if (columnIndexOp != 7) {  
 columnIndexOp += 1  
 } else break  
 }  
 val countColumn = columnIndexOp - columnIndex  
 if (board[rowIndex][columnIndexOp] != '\_' && board[rowIndex][columnIndexOp] != opponent) {  
 for (i in 1..countColumn) {  
 board[rowIndex][columnIndex + i] = player  
 }  
 }  
 break  
 }  
 }  
 if(rowIndex > 1 && board[rowIndex - 1][columnIndex] == opponent) {  
 var rowIndexOp = rowIndex - 1  
 while(rowIndexOp > 0) {  
 while(board[rowIndexOp][columnIndex] == opponent) {  
 if(rowIndexOp != 0) {  
 rowIndexOp -= 1  
 } else break  
 }  
 val countRow = rowIndex - rowIndexOp  
 if (board[rowIndexOp][columnIndex] != '\_' && board[rowIndexOp][columnIndex] != opponent) {  
 for (i in 1..countRow) {  
 board[rowIndex - i][columnIndex] = player  
 }  
 }  
 break  
 }  
 }  
 if(columnIndex > 1 && board[rowIndex][columnIndex - 1] == opponent) {  
 var columnIndexOp = columnIndex - 1  
 while(columnIndexOp > 0) {  
 while(board[rowIndex][columnIndexOp] == opponent) {  
 if(columnIndexOp != 0) {  
 columnIndexOp -= 1  
 } else break  
 }  
 val countColumn = columnIndex - columnIndexOp  
 if (board[rowIndex][columnIndexOp] != '\_' && board[rowIndex][columnIndexOp] != opponent) {  
 for (i in 1..countColumn) {  
 board[rowIndex][columnIndex - i] = player  
 }  
 }  
 break  
 }  
 }  
 if(rowIndex > 1 && columnIndex > 1 && board[rowIndex - 1][columnIndex - 1] == opponent) {  
 var rowIndexOp = rowIndex - 1  
 var columnIndexOp = columnIndex - 1  
 while(rowIndexOp >= 1 || columnIndexOp >= 1) {  
 while(board[rowIndexOp][columnIndexOp] == opponent) {  
 if (rowIndexOp != 0 && columnIndexOp != 0) {  
 rowIndexOp -= 1  
 columnIndexOp -= 1  
 } else break  
 }  
 val countRow = rowIndex - rowIndexOp  
 if (board[rowIndexOp][columnIndexOp] != '\_' && board[rowIndexOp][columnIndexOp] != opponent) {  
 for (i in 1..countRow) {  
 board[rowIndex - i][columnIndex - i] = player  
 }  
 }  
 break  
 }  
 }  
 if(rowIndex > 1 && columnIndex < 7 && board[rowIndex - 1][columnIndex + 1] == opponent) {  
 var rowIndexOp = rowIndex - 1  
 var columnIndexOp = columnIndex + 1  
 while(rowIndexOp > 0 || columnIndexOp < 8) {  
 while(board[rowIndexOp][columnIndexOp] == opponent) {  
 if(rowIndexOp != 0 && columnIndexOp != 7) {  
 rowIndexOp -= 1  
 columnIndexOp += 1  
 } else break  
 }  
 val countRow = rowIndex - rowIndexOp  
 if (board[rowIndexOp][columnIndexOp] != '\_' && board[rowIndexOp][columnIndexOp] != opponent) {  
 for (i in 1..countRow) {  
 board[rowIndex - i][columnIndex + i] = player  
 }  
 }  
 break  
 }  
 }  
 if(rowIndex < 7 && columnIndex > 1 && board[rowIndex + 1][columnIndex - 1] == opponent) {  
 var rowIndexOp = rowIndex + 1  
 var columnIndexOp = columnIndex - 1  
 while(rowIndexOp < 8 || columnIndexOp > 0) {  
 while(board[rowIndexOp][columnIndexOp] == opponent) {  
 if(rowIndexOp != 7 && columnIndexOp != 0) {  
 rowIndexOp += 1  
 columnIndexOp -= 1  
 } else break  
 }  
 val countRow = rowIndexOp - rowIndex  
 if (board[rowIndexOp][columnIndexOp] != '\_' && board[rowIndexOp][columnIndexOp] != opponent) {  
 for (i in 1..countRow) {  
 board[rowIndex + i][columnIndex - i] = player  
 }  
 }  
 break  
 }  
 }  
 for (i in board) {  
 *println*(i)  
 }  
 }  
}

* **Opponent.kt**

class Opponent {  
 enum class Direction {  
 DOWN\_RIGHT, DOWN, RIGHT, UP, LEFT, UP\_RIGHT, UP\_LEFT, DOWN\_LEFT  
 }  
 fun move(board: Array<CharArray>, opponent: Char, player: Char): MutableMap<Pair<Int, Int>, MutableList<Pair<Int, Direction>>> {  
 val listOfPositionPlayer = *mutableListOf*<Pair<Int, Int>>()  
 val listOfPositionOpponent = *mutableListOf*<Pair<Int, Int>>()  
 val moveOption = *mutableMapOf*<Pair<Int, Int>, MutableList<Pair<Int, Direction>>>()  
  
 for (line in 0 *until* board.size) {  
 for (column in 0 *until* board[line].size) {  
 if (board[line][column] == opponent) listOfPositionOpponent.add(Pair(line, column))  
 }  
 }  
  
 for (realPosition in listOfPositionOpponent) {  
 if(realPosition.first + 1 < 8 && realPosition.second + 1 < 8 && board[realPosition.first + 1][realPosition.second + 1] == '\_') {  
 listOfPositionPlayer.add(Pair(realPosition.first + 1, realPosition.second + 1))  
 }  
 if(realPosition.first + 1 < 8 && board[realPosition.first + 1][realPosition.second] == '\_') {  
 listOfPositionPlayer.add(Pair(realPosition.first + 1, realPosition.second))  
 }  
 if(realPosition.second + 1 < 8 && board[realPosition.first][realPosition.second + 1] == '\_') {  
 listOfPositionPlayer.add(Pair(realPosition.first, realPosition.second + 1))  
 }  
 if(realPosition.first - 1 > -1 && board[realPosition.first - 1][realPosition.second] == '\_') {  
 listOfPositionPlayer.add(Pair(realPosition.first - 1, realPosition.second))  
 }  
 if(realPosition.second - 1 > -1 && board[realPosition.first][realPosition.second - 1] == '\_') {  
 listOfPositionPlayer.add(Pair(realPosition.first, realPosition.second - 1))  
 }  
 if(realPosition.first - 1 > -1 && realPosition.second - 1 > -1 && board[realPosition.first - 1][realPosition.second - 1] == '\_') {  
 listOfPositionPlayer.add(Pair(realPosition.first - 1, realPosition.second - 1))  
 }  
 if(realPosition.first + 1 < 8 && realPosition.second - 1 > -1 && board[realPosition.first + 1][realPosition.second - 1] == '\_') {  
 listOfPositionPlayer.add(Pair(realPosition.first + 1, realPosition.second - 1))  
 }  
 if(realPosition.first - 1 > -1 && realPosition.second + 1 < 8 &&board[realPosition.first - 1][realPosition.second + 1] == '\_') {  
 listOfPositionPlayer.add(Pair(realPosition.first - 1, realPosition.second + 1))  
 }  
 }  
  
 for (checker in listOfPositionPlayer) {  
 moveOption[checker] = *mutableListOf*()  
 if(checker.first < 7 && checker.second < 7 && board[checker.first + 1][checker.second + 1] == opponent) {  
 var rowIndex = checker.first + 1  
 var columnIndex = checker.second + 1  
 while(rowIndex < 8 || columnIndex < 8) {  
 while(board[rowIndex][columnIndex] == opponent) {  
 if(rowIndex != 7 && columnIndex != 7) {  
 rowIndex += 1  
 columnIndex += 1  
 } else break  
 }  
 val numberOfCoups = rowIndex - checker.first  
 if (board[rowIndex][columnIndex] != '\_' && board[rowIndex][columnIndex] != opponent) {  
 moveOption[checker]!!.add(Pair(numberOfCoups, Direction.DOWN\_RIGHT))  
 }  
 break  
 }  
 }  
 if(checker.first < 7 && board[checker.first + 1][checker.second] == opponent) {  
 var rowIndex = checker.first+ 1  
 while(rowIndex < 8) {  
 while(board[rowIndex][checker.second] == opponent) {  
 if(rowIndex != 7) {  
 rowIndex += 1  
 } else break  
 }  
 val numberOfCoups = rowIndex - checker.first  
 if (board[rowIndex][checker.second] != '\_' && board[rowIndex][checker.second] != opponent) {  
 moveOption[checker]!!.add(Pair(numberOfCoups, Direction.DOWN))  
 }  
 break  
 }  
 }  
 if(checker.second < 7 && board[checker.first][checker.second + 1] == opponent) {  
 var columnIndex = checker.second + 1  
 while(columnIndex < 8) {  
 while(board[checker.first][columnIndex] == opponent) {  
 if (columnIndex != 7) {  
 columnIndex += 1  
 } else break  
 }  
 val numberOfCoups = columnIndex - checker.second  
 if (board[checker.first][columnIndex] != '\_' && board[checker.first][columnIndex] != opponent) {  
 moveOption[checker]!!.add(Pair(numberOfCoups, Direction.RIGHT))  
 }  
 break  
 }  
 }  
 if(checker.first > 1 && board[checker.first - 1][checker.second] == opponent) {  
 var rowIndex = checker.first - 1  
 while(rowIndex > 0) {  
 while(board[rowIndex][checker.second] == opponent) {  
 if(rowIndex != 0) {  
 rowIndex -= 1  
 } else break  
 }  
 val numberOfCoups = checker.first - rowIndex  
 if (board[rowIndex][checker.second] != '\_' && board[rowIndex][checker.second] != opponent) {  
 moveOption[checker]!!.add(Pair(numberOfCoups, Direction.UP))  
 }  
 break  
 }  
 }  
 if(checker.second > 1 && board[checker.first][checker.second - 1] == opponent) {  
 var columnIndex = checker.second - 1  
 while(columnIndex > 0) {  
 while(board[checker.first][columnIndex] == opponent) {  
 if(columnIndex != 0) {  
 columnIndex -= 1  
 } else break  
 }  
 val numberOfCoups = checker.second - columnIndex  
 if (board[checker.first][columnIndex] != '\_' && board[checker.first][columnIndex] != opponent) {  
 moveOption[checker]!!.add(Pair(numberOfCoups, Direction.LEFT))  
 }  
 break  
 }  
 }  
 if(checker.first > 1 && checker.second > 1 && board[checker.first - 1][checker.second - 1] == opponent) {  
 var rowIndex = checker.first - 1  
 var columnIndex = checker.second - 1  
 while(rowIndex >= 1 || columnIndex >= 1) {  
 while(board[rowIndex][columnIndex] == opponent) {  
 if (rowIndex != 0 && columnIndex != 0) {  
 rowIndex -= 1  
 columnIndex -= 1  
 } else break  
 }  
 val numberOfCoups = checker.first - rowIndex  
 if (board[rowIndex][columnIndex] != '\_' && board[rowIndex][columnIndex] != opponent) {  
 moveOption[checker]!!.add(Pair(numberOfCoups, Direction.UP\_LEFT))  
 }  
 break  
 }  
 }  
 if(checker.first > 1 && checker.second < 7 && board[checker.first - 1][checker.second + 1] == opponent) {  
 var rowIndex = checker.first - 1  
 var columnIndex = checker.second + 1  
 while(rowIndex > 0 || columnIndex < 8) {  
 while(board[rowIndex][columnIndex] == opponent) {  
 if(rowIndex != 0 && columnIndex != 7) {  
 rowIndex -= 1  
 columnIndex += 1  
 } else break  
 }  
 val numberOfCoups = checker.first - rowIndex  
 if (board[rowIndex][columnIndex] != '\_' && board[rowIndex][columnIndex] != opponent) {  
 moveOption[checker]!!.add(Pair(numberOfCoups, Direction.UP\_RIGHT))  
 }  
 break  
 }  
 }  
 if(checker.first < 7 && checker.second > 1 && board[checker.first + 1][checker.second - 1] == opponent) {  
 var rowIndex = checker.first + 1  
 var columnIndex = checker.second - 1  
 while(rowIndex < 8 || columnIndex > 0) {  
 while(board[rowIndex][columnIndex] == opponent) {  
 if(rowIndex != 7 && columnIndex != 0) {  
 rowIndex += 1  
 columnIndex -= 1  
 } else break  
 }  
 val numberOfCoups = rowIndex - checker.first  
 if (board[rowIndex][columnIndex] != '\_' && board[rowIndex][columnIndex] != opponent) {  
 moveOption[checker]!!.add(Pair(numberOfCoups, Direction.DOWN\_LEFT))  
 }  
 break  
 }  
 }  
 }  
 return moveOption  
 }  
  
 fun move2(board: Array<CharArray>, opponent: Char, player: Char) {  
 val moveOption = move(board, opponent, player)  
 val listMax = *mutableMapOf*<Pair<Int, Int>, Int>()  
 for (i in moveOption) {  
 var count = 0  
 for (el in i.value) {  
 count += el.first  
 }  
 listMax[i.key] = count  
 }  
  
 *println*(moveOption)  
 if(!moveOption.*isNullOrEmpty*()) {  
 val maxValue = listMax.*maxBy* **{ it**.value **}** *println*(maxValue)  
 if (maxValue!!.value != 0 ) {  
 board[maxValue.key.first][maxValue.key.second] = player  
 for (move in moveOption[maxValue.key]!!) {  
 if(move.second == Direction.DOWN\_RIGHT) {  
 for (i in 1..move.first) {  
 board[maxValue.key.first + i][maxValue.key.second + i] = player  
 }  
 }  
 if(move.second == Direction.DOWN) {  
 for (i in 1..move.first) {  
 board[maxValue.key.first + i][maxValue.key.second] = player  
 }  
 }  
 if(move.second == Direction.RIGHT) {  
 for (i in 1..move.first) {  
 board[maxValue.key.first][maxValue.key.second + i] = player  
 }  
 }  
 if(move.second == Direction.UP) {  
 for (i in 1..move.first) {  
 board[maxValue.key.first - i][maxValue.key.second] = player  
 }  
 }  
 if(move.second == Direction.LEFT) {  
 for (i in 1..move.first) {  
 board[maxValue.key.first][maxValue.key.second - i] = player  
 }  
 }  
 if(move.second == Direction.UP\_RIGHT) {  
 for (i in 1..move.first) {  
 board[maxValue.key.first - i][maxValue.key.second + i] = player  
 }  
 }  
 if(move.second == Direction.UP\_LEFT) {  
 for (i in 1..move.first) {  
 board[maxValue.key.first - i][maxValue.key.second - i] = player  
 }  
 }  
 if(move.second == Direction.DOWN\_LEFT) {  
 for (i in 1..move.first) {  
 board[maxValue.key.first + i][maxValue.key.second - i] = player  
 }  
 }  
 }  
 *conditional* = true  
 } else *conditional* = false  
 } else *conditional* = false  
 }

1. **Тесты**

Все функции в пакете kotlin.test проверяет работоспособность метода move2

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

import org.junit.jupiter.api.Tag  
import org.junit.jupiter.api.Test  
import Logic.Opponent  
import kotlin.test.assertEquals  
  
class Test {  
  
 private val boardTest1 = *arrayOf*(  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', 'W', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', 'W', 'W', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', 'W', 'W', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', 'W', 'B', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_')  
 )  
  
 private val boardTest2 = *arrayOf*(  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', 'W', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', 'W', 'W', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', 'W', 'B', 'W', 'W', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_')  
 )  
  
 private val boardTest3 = *arrayOf*(  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', 'W', '\_'),  
 *charArrayOf*('\_', '\_', '\_', 'W', '\_', '\_', 'W', '\_'),  
 *charArrayOf*('\_', '\_', '\_', 'W', 'W', '\_', 'W', '\_'),  
 *charArrayOf*('\_', 'W', 'W', 'W', 'B', 'W', 'W', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', 'B', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_'),  
 *charArrayOf*('\_', '\_', '\_', '\_', '\_', '\_', '\_', '\_')  
 )  
  
 private val mop = Opponent()  
 private fun move2(board: Array<CharArray>): Int {  
  
 val listMax = *mutableMapOf*<Pair<Int, Int>, Int>()  
 val moveOption = mop.move(board, 'W', 'B')  
 for (i in moveOption) {  
 var count = 0  
 for (el in i.value) {  
 count += el.first  
 }  
 listMax[i.key] = count  
 }  
 *println*(mop.move(board, 'B', 'W'))  
 *println*(listMax)  
 *println*(listMax.*maxBy* **{ it**.value **}**)  
 val result = listMax.*maxBy* **{ it**.value **}** return result!!.value  
 }  
 @Test  
 @Tag("Easy")  
 fun testMove1() {  
 *assertEquals*(4, move2(boardTest1))  
 }  
  
 @Test  
 @Tag("Easy")  
 fun testMove2() {  
 *assertEquals*(3, move2(boardTest2))  
 }  
  
 @Test  
 @Tag("Easy")  
 fun testMove3() {  
 *assertEquals*(5, move2(boardTest3))  
 }  
}

1. **Скриншоты программы**

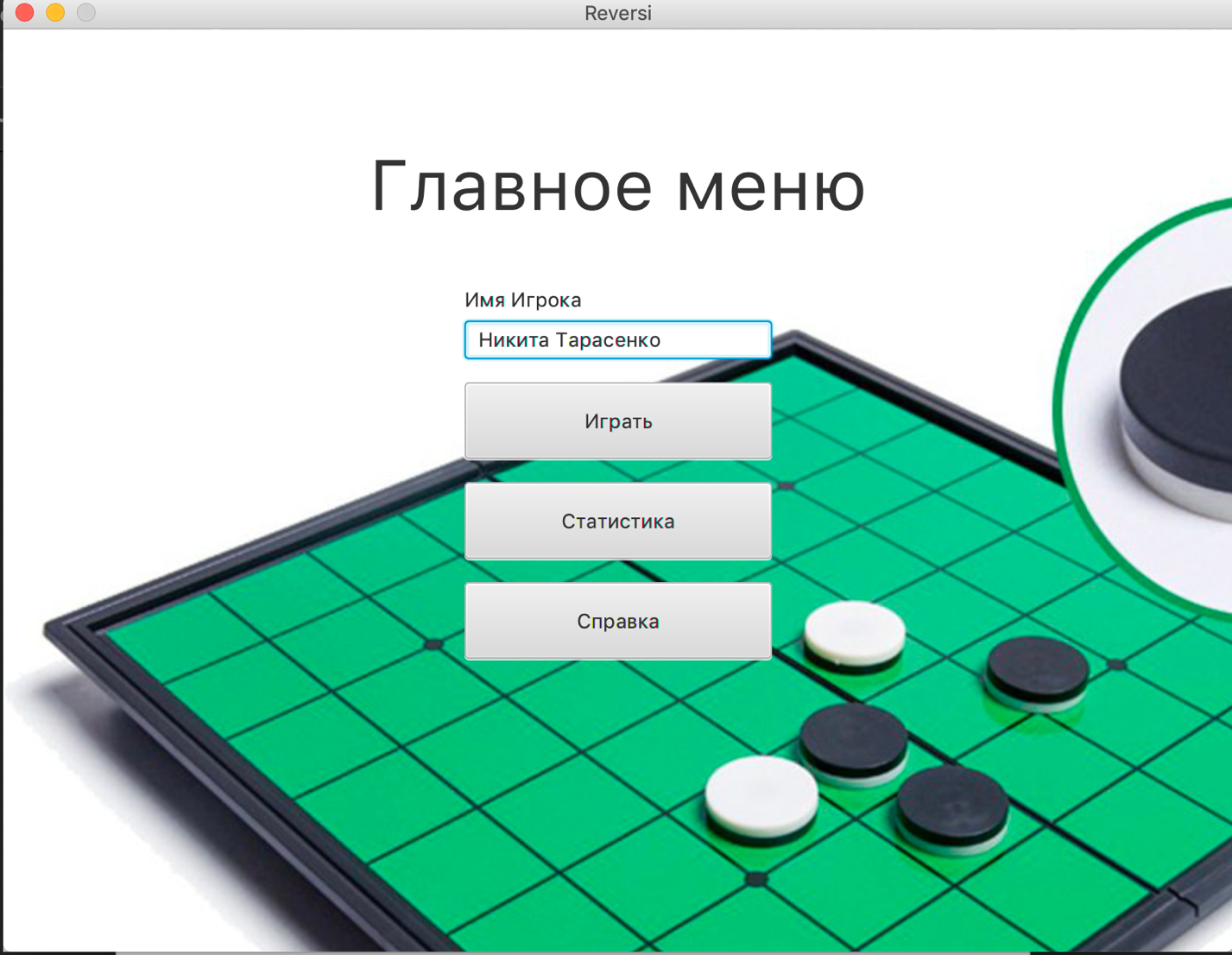
****

Рисунок 5.1 Главное меню

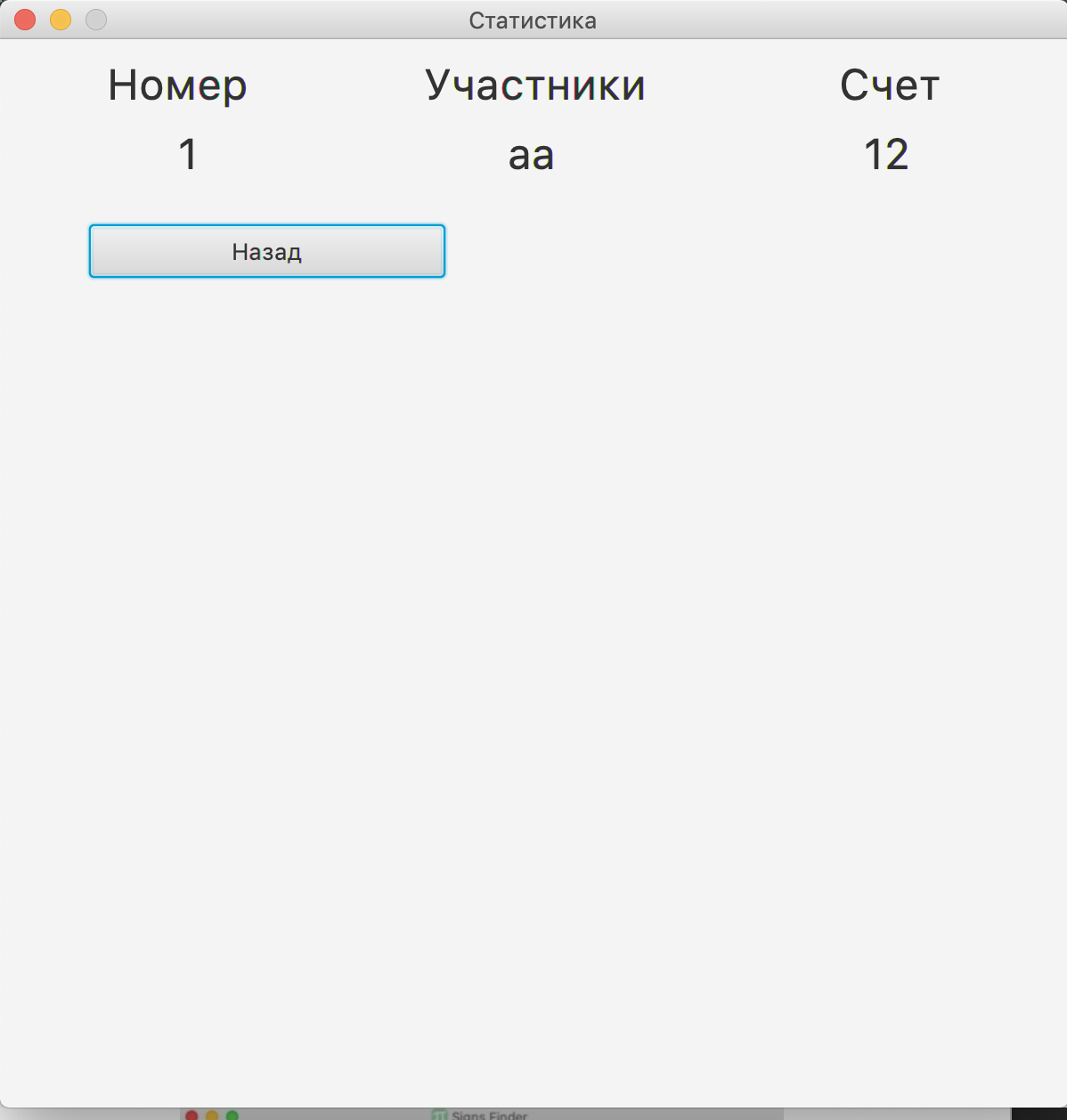


Рисунок 5.2 Статистика игр

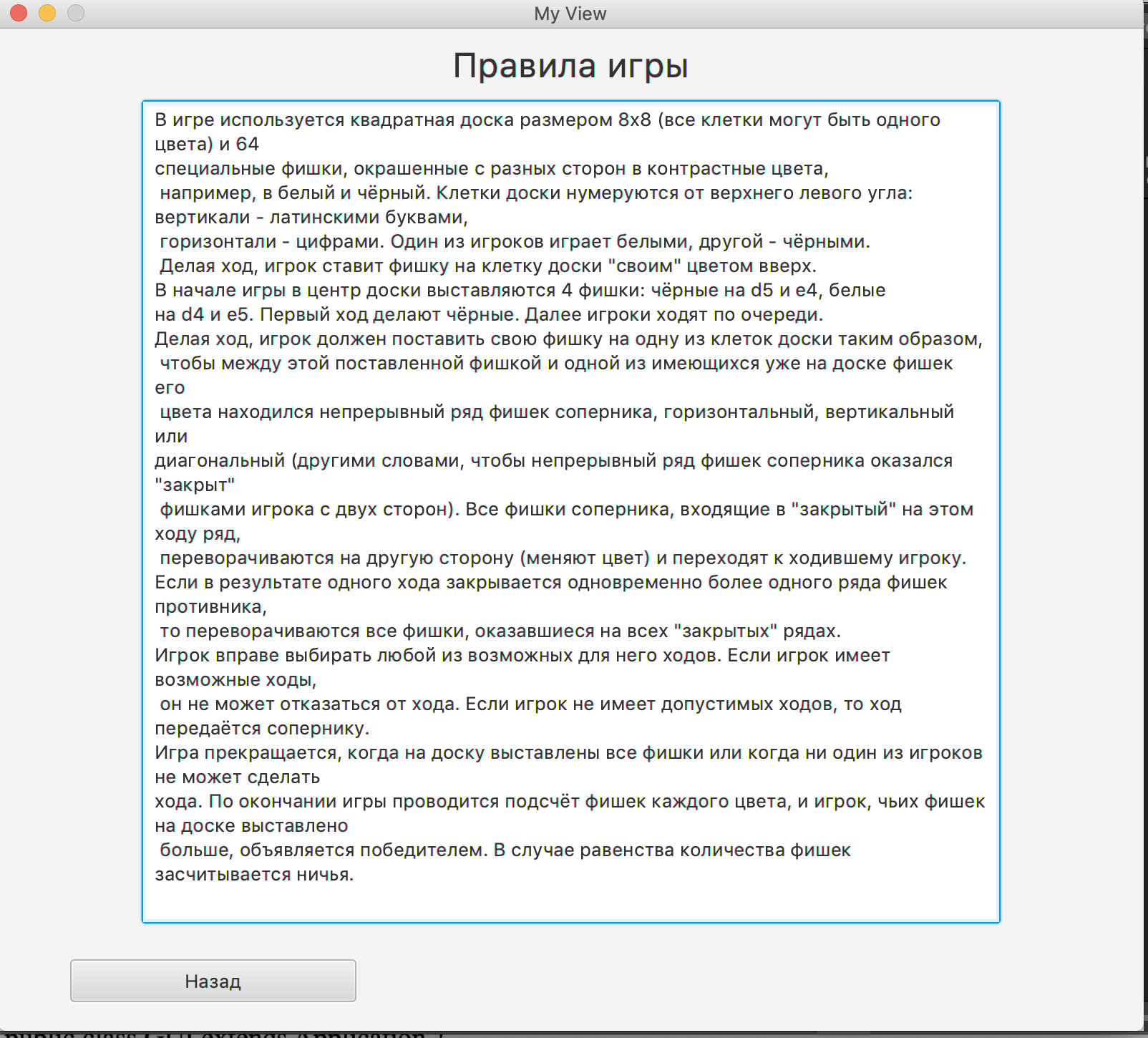


Рисунок 5.3 Справка (правила игры)

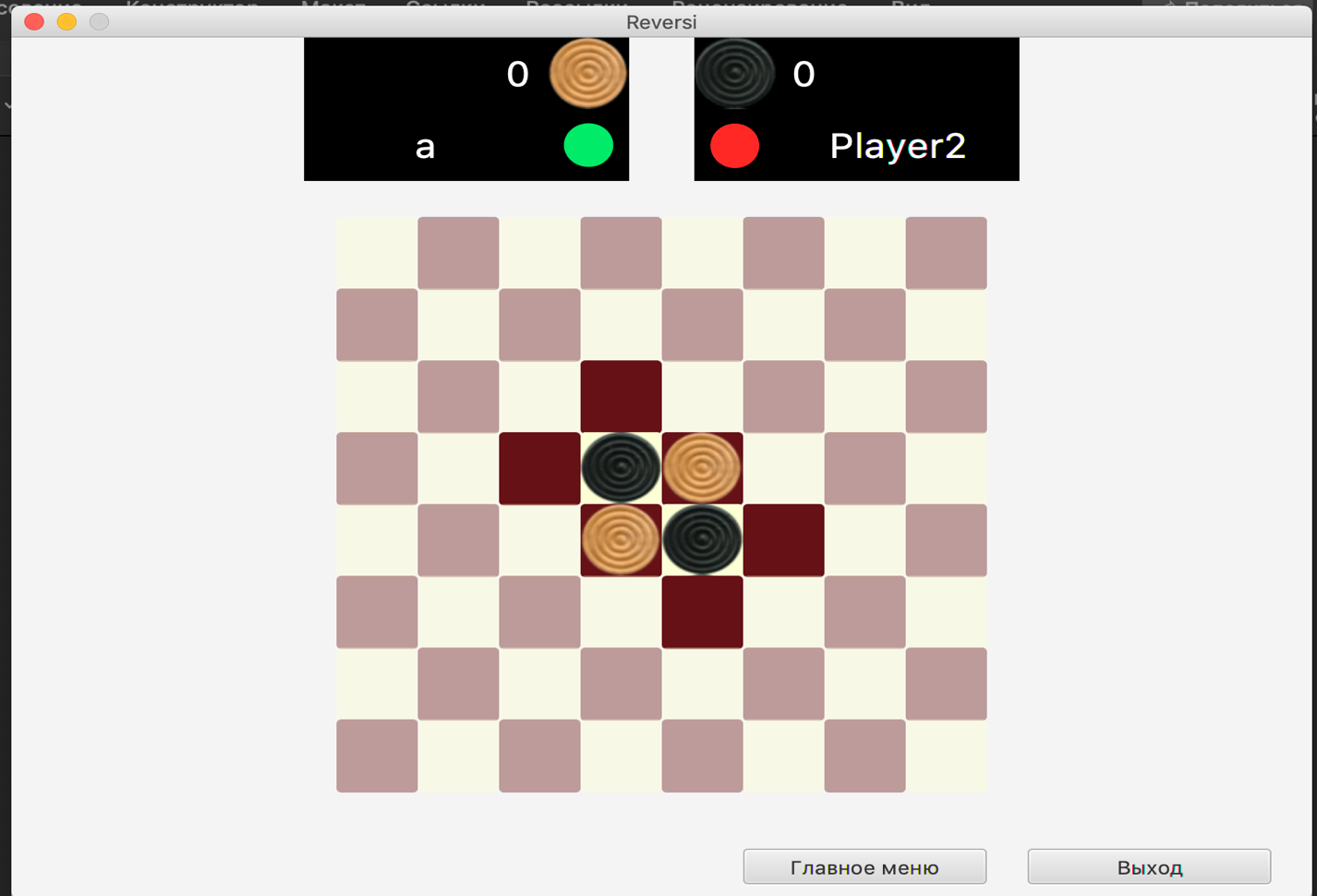


Рисунок 5.4 Старт игры

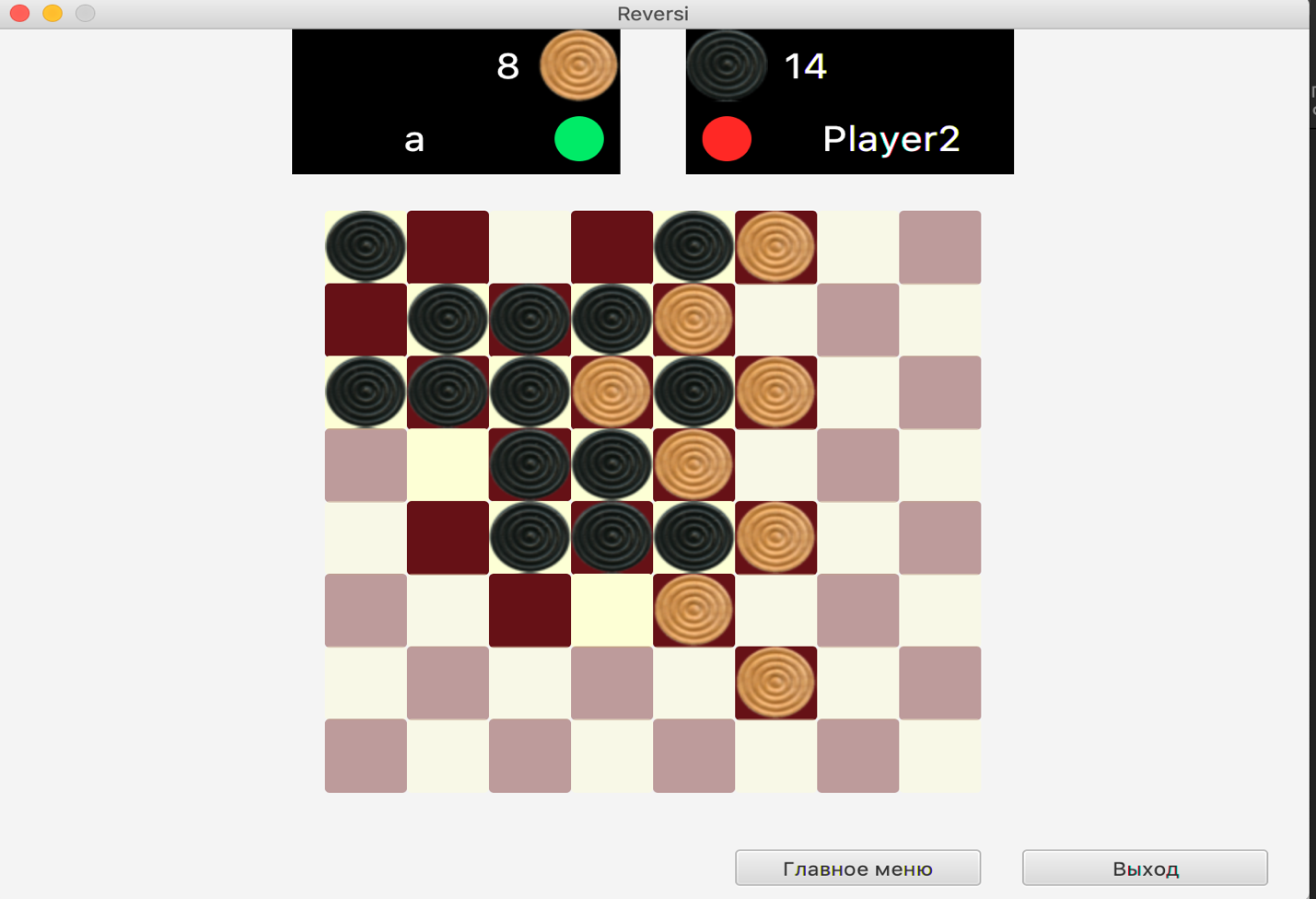


Рисунок 5.5 – Середина игры



Рисунок 5.6 – Конец игры