Министерство образования Российской федерации  
Пензенский государственный университет  
Кафедра «Вычислительная техника»

**Пояснительная записка**

к курсовой работа

по курсу «Программирование»

на тему «Игра Шахматы»

Выполнил:

студент группы 24ВВВ1

Будников А.С.

Приняла:

к.т.н., доцент Генералова А.А.

**Оглавление**

**Введение**

«Шахматы» – это одна из самых древних и популярных интеллектуальных игр, сочетающей в себе элементы науки и спорта. С развитием компьютерных технологий шахматы стали важно областью исследования в программировании за счет возможности изучения разработки алгоритмов анализа игровых стратегий и создания пользовательских интерфейсов.

В данной курсовой работе рассматривается разработка шахматной программы на языке программирования Golang, включая реализацию игровой логики и применения алгоритма поиска лучших ходов. Данная работа является актуальной и важной в контексте создания современных интеллектуальных компьютерных игр.

1. **Постановка задачи**

Необходимо разработать программу — игру «Шахматы» с возможностью игры против компьютера.  
Программа должна быть интуитивно понятной, с реализованными основными правилами шахматной игры, включая начальную расстановку фигур, правила движения, взятия фигур, шаха, мата и патовой ситуации.

Многомодульность программы**.** Необходимо поделить программу на логические модули с четким разделением логики выполнения. Такой подход значительно упростит отладку и тестирование программы, а также повысит расширяемость проекта.

Интерфейс и пользовательское взаимодействие. Программа должна работать в текстовом или графическом режиме. Необходимо определить тип пользовательского интерфейса и разработать его структуру. В случае графического режима – предусмотреть визуальные элементы управления с возможностью управления при помощи компьютерной мыши.Программа должна поддерживать ввод данных с клавиатуры (или мыши), анализировать действия игрока (выбор и перемещение фигур), отображать ходы и информировать о состоянии партии.

Устройство ввода информации **–** клавиатура и мышь. Необходимо реализовать обработку пользовательского ввода таким образом, чтобы все действия, производимые пользователем, были однозначно интерпретируемы.

Необходимо реализовать простой алгоритм игры компьютера, возможным вариантом может стать алгоритм оценки позиции и выбора лучшего хода.

**2. Выбор решения**

При запуске программы на экран пользователя выводится приветственный экран с заставкой и меню, состоящее из 3 пунктов:

1. Запуск игры. На экран выводится пользовательский интерфейс с возможностью начать игру.
2. Настройка. Пользователь изменяет необходимые параметры программы по средством ввода предложенных ему команд.
3. Выход. Программа завершается без сохранения данных.

Разработанная программа состоит из 11 модулей:

1. main.go
2. ui.go
3. custom\_button.go
4. evaluation.go
5. board.go
6. piece.go
7. generator.go
8. move.go
9. minimax.go
10. popcount.go
11. popcount.s

Файл main.go – главный файл программы, служащий для инициализации приложения, вывода главного меню и создания лог-файла[[1]](#footnote-1).

Файл ui.go нужен для отображения пользовательского интерфейса и обработки событий при взаимодействии пользователя с ним.

Файл custom\_button.go нужен для инициализации кнопки, использующейся в качестве клетки на шахматной доске.

Файл evaluation.go нужен для реализации функций оценки позиции.

Файл board.go нужен для инициализации шахматной доски.

Файл piece.go нужен для инициализации переменных шахматных фигур.

Файл generator.go содержит описание функций, использующихся для генерации возможных ходов фигур.

Файл move.go содержит описание функций, использующихся для перемещения фигур по шахматной доске.

Файл minimax.go содержит описание функций выбора лучшего хода.

Файл popcount.go содержит объявление функции, реализованной на языке Ассемблера.

Файл popcount.s содержит описание функции, объявленной в файле popcount.go.

**3.Описание разработки программы**

В рамках разработки данной программы в качестве основного языка программирования был выбран Go (Golang). Этот язык представляет собой современное и востребованное средство разработки, отличающееся простым и лаконичным синтаксисом, высокой производительностью, а также возможностью взаимодействия с языком Ассемблера. Go ориентирован на эффективную работу с многопоточностью, что делает его особенно удобным для создания масштабируемых прикладных решений. Язык обеспечивает высокоуровневые абстракции при сохранении контроля над управлением памятью и производительностью, предоставляет встроенные средства параллельного выполнения, сборки мусора и строгую типизацию. Кроме того, наличие системы пакетов позволяет структурировать проект на модули, обеспечивая тем самым упорядоченность архитектуры и удобство сопровождения.

Для разработки были использованы среды Visual Studio Code и NeoVim, обладающие широкими возможностями для поддержки Go за счёт расширений и LSP-серверов. Оба инструмента обеспечивают функции статического анализа, отладки и интеллектуальных подсказок, что значительно упрощает и ускоряет процесс программирования.

После запуска программы пользователю отображается заставка и главное меню. Управление осуществляется путём ввода соответствующего номера пункта меню. Начать игру можно, выбрав пункт 1 – «Начать игру», переход к настройкам осуществляется через пункт 2, в котором предоставляется возможность изменения параметров шахматного движка и других опций программы. Завершение работы доступно через пункт 3 – «Выход».

Начальный этап разработки включал реализацию приветственного экрана и главного меню, функционирующего на основе бесконечного цикла и встроенной функции bufio.Reader.Read() для считывания пользовательского ввода из терминала. На основе данной логики позднее была реализована консоль ввода команд, доступная в процессе игры.

Ключевым этапом разработки шахматного движка стало создание двух основных структур: шахматных фигур и игровой доски. Для описания фигур был реализован отдельный модуль, содержащий константы для обозначения типов фигур и их цвета, а также структуру клетки доски с соответствующими полями. Игровая доска была реализована в виде двумерного массива обобщённого типа Board[8][8]Square. Дополнительно были разработаны функции для инициализации доски, получения и перемещения фигур, проверки состояния клетки, а также создания «глубокой копии»[[2]](#footnote-2) доски, необходимой для алгоритма оценки позиции.

Затем был разработан модуль evaluation, содержащий оценочные функции на основе стоимости фигур и оценки позиционного преимущества. В модуле реализована карта соответствия фигура-ценность, массив бонусных значений для центральных клеток и функция оценки безопасности короля. Одна из функций, реализованных в этом модуле, написана на языке Ассемблера в файле popcount.s и предназначена для подсчёта количества фигур заданного цвета.

Для корректной работы движка были реализованы алгоритмы генерации и валидации ходов, а также перемещения фигур. Для этого созданы файлы move.go и generator.go. В последнем определены ключевые функции генерации возможных ходов для каждой фигуры, оптимизированные за счёт использования универсальных функций для прямолинейных и диагональных перемещений. В move.go описаны следующие функции:

1. abs(x int) int – вспомогательная функция для вычисления абсолютного значения числа;
2. MakeMove(b \*board.Board, m Move) error – функция выполнения хода с учётом правил и проверки безопасности короля;
3. IsKingInCheck(b board.Board, color board.Color) bool – функция, определяющая, находится ли король под шахом.

Следующим этапом стала реализация алгоритмов поиска, сортировки и выбора оптимального хода. В качестве основного алгоритма был использован Minimax с альфа-бета-отсечением, сочетающий простоту реализации с высокой эффективностью. Данный алгоритм был дополнен следующими компонентами: транспозиционной таблицей для хранения оценок, алгоритмом поиска покоя (для устранения «горизонтального эффекта»[[3]](#footnote-3)), функциями сортировки и эвристиками, учитывающими стратегическую важность центральных клеток.

Заключительным этапом стало создание графического интерфейса, обеспечивающего взаимодействие пользователя с программой. Для этого были реализованы файлы custom\_button.go и ui.go в соответствующем пакете ui. Первый отвечает за обработку событий мыши и выбор клеток на доске, второй — за построение и отрисовку графического интерфейса. В качестве библиотеки был выбран популярный фреймворк[[4]](#footnote-4) Fyne, обеспечивающий удобные средства для создания GUI [[5]](#footnote-5)в Go. Реализованы функции запуска интерфейса, обработки команд, воспроизведения звука при перемещении фигуры и обновления отображения доски.

**4.Отладка и тестирование**

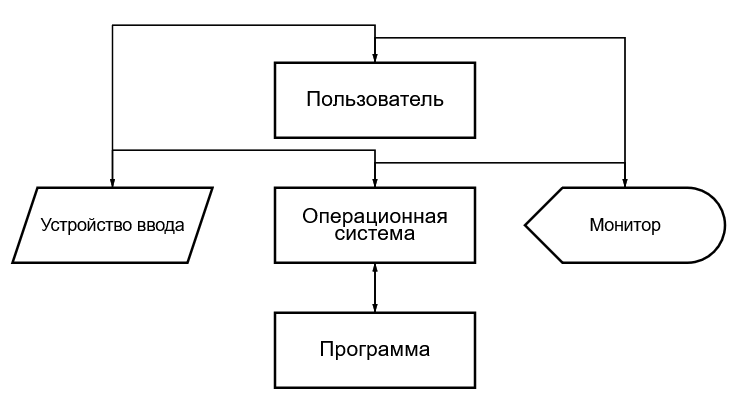
В качестве основных инструментов разработки использовались Visual Studio Code и NeoVim. Оба редактора обеспечивают необходимые средства для эффективной работы с языком Go, включая поддержку LSP (Language Server Protocol)[[6]](#footnote-6), статического анализа кода, автодополнения, навигации по проекту и встроенных терминалов. Visual Studio Code был использован как основная среда для структурированной разработки и отладки, а NeoVim — в качестве лёгкого и настраиваемого инструмента для быстрого редактирования кода и работы в терминале.

Отладка производилась с использованием расширений, поддерживающих пошаговое выполнение кода, установку точек останова, анализ значений переменных и состояния памяти во время выполнения. Это позволяло эффективно выявлять и устранять ошибки в логике работы шахматного движка, особенно на этапах взаимодействия с Ассемблером и реализации алгоритмов поиска.

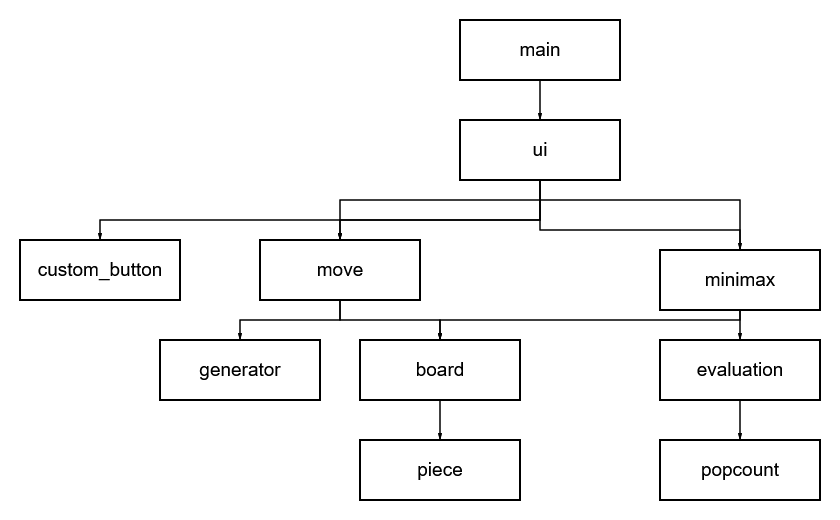
Тестирование осуществлялось поэтапно, в процессе разработки каждого модуля, а также после завершения реализации основных функций программы. В результате тестирования были обнаружены и устранены многочисленные ошибки, связанные с обработкой игровых структур, системой оценки позиций и корректностью исполнения ассемблерных подпрограмм.

**5. Описание программы**

**5.1. Схема данных**

****

**5.2. Иерархическая структура приложения**

****

**5.3. Описание модулей**

1) main.go – основной файл программы.

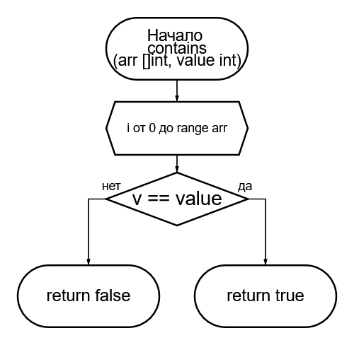
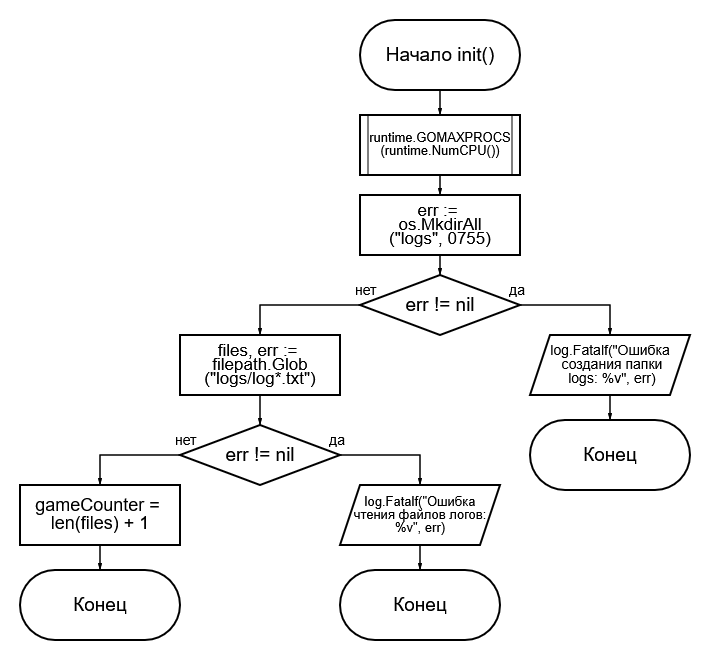
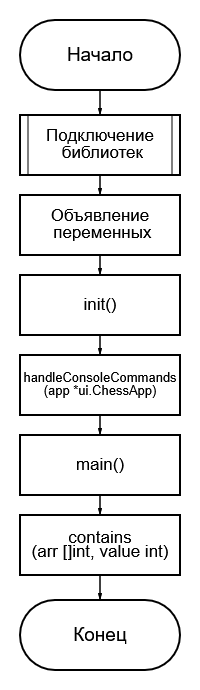
1.1) init() – функция, которая устанавливает максимальное количество операционных системных потоков, которые могут одновременно выполняться, проверяет наличие папки с лог-файлами и определяет название нового лог-файла.

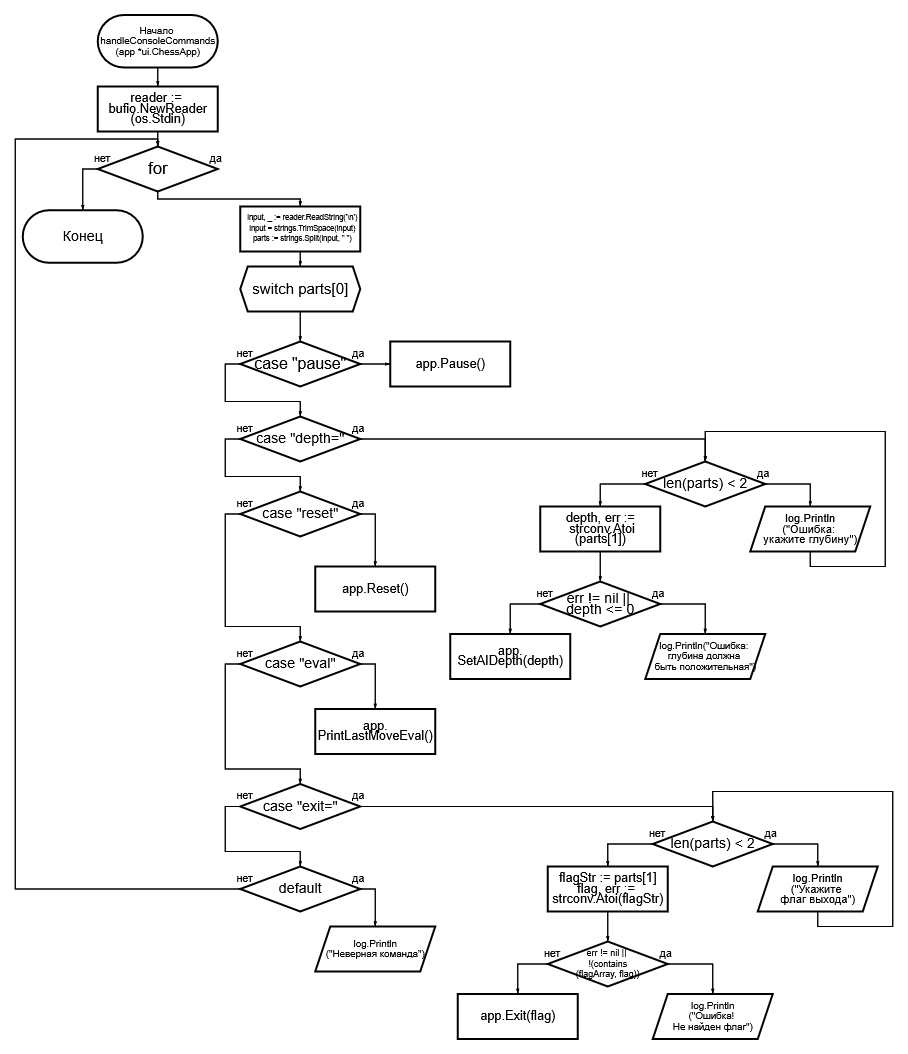
1.2) handleConsoleCommands(app \*ui.ChessApp) – функция, в которой реализуются консольные команды.

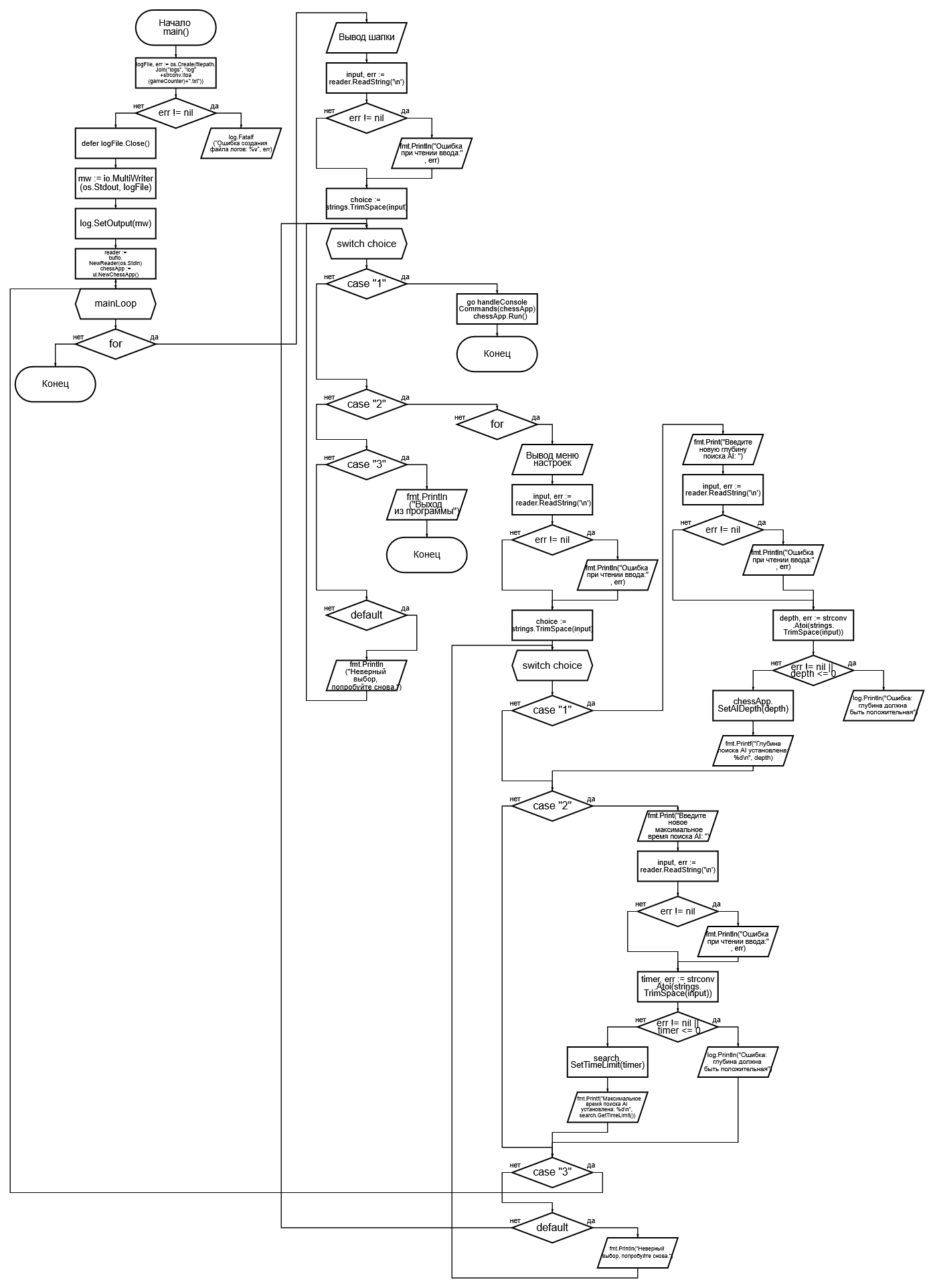
1.3) main() – основная функция программы, в которой реализовано главное меню программы, создание нового лог-файла, установка вывода программы в файл и запуск графического интерфейса.

1.4) contains(arr []int, value int) – вспомогательная функция, в которой реализован алгоритм определения факта нахождения элемента в массиве.

Блок схема файла main.go представлена на рисунках.







2) ui.go

**6.Руководство пользователя**

**Заключение**

При выполнении данной курсовой работы были получены навыки разработки многомодульных программ. Были освоены приемы создания пользовательского интерфейса, изучены функции работы с консолью, способы обработки событий с клавиатуры. Получены базовые навыки программирования на языках программирования Golang и Assembler. Изучены основные возможности редакторов кода Visual Studio Code и NeoVim. Получены навыки отладки и тестирования программ.

В рамках выполнения курсовой работы была написана игра «Шахматы». Программа предоставляет достаточный список возможностей, включая игру с компьютером.

В дальнейшем программу можно улучшить, обновив интерфейс и внедрив его настройку, добавив поддержку UCI-протокола и усовершенствовав искусственный интеллект.

**Библиографический список**

1. <https://www.chessprogramming.org>
2. Корнилов Е.Н. «Программирование шахмат и других логических игр.» — СПб.: БХВ-Петербург, 2005.
3. Михалис Цукалос. «Golang для профи», 2020.
4. <https://talkchess.com>
5. <https://habr.com/ru/articles/682122/>
6. <https://chess.fandom.com>
7. Claude E. Shannon, David Levy «Computer Chess Compendium», 1988.

**Листинг программы**

**Файл main.go**

package main

import (

    "bufio"

    "chess-engine/search"

    "chess-engine/ui"

    "fmt"

    "io"

    "log"

    "os"

    "path/filepath"

    "runtime"

    "strconv"

    "strings"

)

var gameCounter int

var flagArray []int = []int{0, 1}

func init() {

    runtime.GOMAXPROCS(runtime.NumCPU())

    err := os.MkdirAll("logs", 0755)

    if err != nil {

        log.Fatalf("Ошибка создания папки logs: %v", err)

    }

    files, err := filepath.Glob("logs/log\*.txt")

    if err != nil {

        log.Fatalf("Ошибка чтения файлов логов: %v", err)

    }

    gameCounter = len(files) + 1

}

func handleConsoleCommands(app \*ui.ChessApp) {

    reader := bufio.NewReader(os.Stdin)

    for {

        input, \_ := reader.ReadString('\n')

        input = strings.TrimSpace(input)

        parts := strings.Split(input, " ")

        switch parts[0] {

        case "pause":

            app.Pause()

        case "depth=":

            if len(parts) < 2 {

                log.Println("Ошибка: укажите глубину")

            } else {

                depth, err := strconv.Atoi(parts[1])

                if err != nil || depth <= 0 {

                    log.Println("Ошибка: глубина должна быть положительная")

                } else {

                    app.SetAIDepth(depth)

                }

            }

        case "reset":

            app.Reset()

        case "eval":

            app.PrintLastMoveEval()

        case "help":

            log.Println("pause, help, depth= <value>, reset, eval, exit= <flag>")

        case "exit=":

            if len(parts) < 2 {

                log.Println("Укажите флаг выхода")

            } else {

                flagStr := parts[1]

                flag, err := strconv.Atoi(flagStr)

                if err != nil || !(contains(flagArray, flag)) {

                    log.Println("Ошибка! Не найден флаг")

                } else {

                    app.Exit(flag)

                }

            }

        default:

            log.Println("Неверная команда")

        }

    }

}

func main() {

    logFile, err := os.Create(filepath.Join("logs", "log"+strconv.Itoa(gameCounter)+".txt"))

    if err != nil {

        log.Fatalf("Ошибка создания файла логов: %v", err)

    }

    defer logFile.Close()

    mw := io.MultiWriter(os.Stdout, logFile)

    log.SetOutput(mw)

    reader := bufio.NewReader(os.Stdin)

    chessApp := ui.NewChessApp()

mainLoop:

    for {

        fmt.Println("\nКурсовая работа на тему: игра Шахматы\nВыполнил: студент группы 24ВВВ1 Будников А.С.\nПриняла: к.т.н. доцент Генералова А.А.\n")

        fmt.Println("Выберите один из пунктов меню\n1. Начать игру\n2. Настройки\n3. Выход")

        fmt.Print("\nМой выбор: ")

        input, err := reader.ReadString('\n')

        if err != nil {

            fmt.Println("Ошибка при чтении ввода:", err)

            continue

        }

        choice := strings.TrimSpace(input)

        switch choice {

        case "1":

            go handleConsoleCommands(chessApp)

            chessApp.Run()

            return

        case "2":

            for {

                fmt.Println("\n=== МЕНЮ НАСТРОЕК ===")

                fmt.Printf("1. Глубина поиска AI (текущая: %d)\n", chessApp.GetAiDepth())

                fmt.Printf("2. Максимальное время поиска AI (текущая: %d)\n", search.GetTimeLimit())

                fmt.Println("3. Вернуться в главное меню")

                fmt.Print("\nМой выбор: ")

                input, err := reader.ReadString('\n')

                if err != nil {

                    fmt.Println("Ошибка при чтении ввода:", err)

                    continue

                }

                choice := strings.TrimSpace(input)

                switch choice {

                case "1":

                    fmt.Print("Введите новую глубину поиска AI: ")

                    input, err := reader.ReadString('\n')

                    if err != nil {

                        fmt.Println("Ошибка при чтении ввода:", err)

                        continue

                    }

                    depth, err := strconv.Atoi(strings.TrimSpace(input))

                    if err != nil || depth <= 0 {

                        fmt.Println("Ошибка: глубина должна быть положительным числом")

                        continue

                    } else {

                        chessApp.SetAIDepth(depth)

                        fmt.Printf("Глубина поиска AI установлена: %d\n", depth)

                        continue

                    }

                case "2":

                    fmt.Print("Введите новое максимальное время поиска AI: ")

                    input, err := reader.ReadString('\n')

                    if err != nil {

                        fmt.Println("Ошибка при чтении ввода:", err)

                        continue

                    }

                    timer, err := strconv.Atoi(strings.TrimSpace(input))

                    if err != nil || timer <= 0 {

                        fmt.Println("Ошибка: время должно быть положительным числом")

                        continue

                    } else {

                        search.SetTimeLimit(timer)

                        fmt.Printf("Максимальное время поиска AI установлена: %d\n", search.GetTimeLimit())

                        continue

                    }

                case "3":

                    continue mainLoop

                default:

                    fmt.Println("Неверный выбор, попробуйте снова.")

                }

            }

        case "3":

            fmt.Println("Выход из программы")

            return

        default:

            fmt.Println("Неверный выбор, попробуйте снова.")

            continue

        }

    }

}

func contains(arr []int, value int) bool {

    for \_, v := range arr {

        if v == value {

            return true

        }

    }

    return false

}

**Файл board.go**

package board

import (

    "errors"

)

type Board [8][8]Square

func NewBoard() Board {

    var b Board

    //Расстановка белых фигур

    b[0] = [8]Square{

        {Rook, White}, {Knight, White}, {Bishop, White}, {Queen, White},

        {King, White}, {Bishop, White}, {Knight, White}, {Rook, White},

    }

    for i := 0; i < 8; i++ {

        b[1][i] = Square{Pawn, White}

    }

    //Расстановка черных фигур

    b[7] = [8]Square{

        {Rook, Black}, {Knight, Black}, {Bishop, Black}, {Queen, Black},

        {King, Black}, {Bishop, Black}, {Knight, Black}, {Rook, Black},

    }

    for i := 0; i < 8; i++ {

        b[6][i] = Square{Pawn, Black}

    }

    //Остальные клетки пустые

    for i := 2; i < 6; i++ {

        for j := 0; j < 8; j++ {

            b[i][j] = Square{Empty, White}

        }

    }

    return b

}

func (b Board) GetPiece(x, y int) (Piece, Color, error) {

    if x < 0 || x >= 8 || y < 0 || y >= 8 {

        return Empty, White, errors.New("координаты за пределами доски")

    }

    return b[x][y].Piece, b[x][y].Color, nil

}

func (b \*Board) SetPiece(x, y int, piece Piece, color Color) error {

    if x < 0 || x >= 8 || y < 0 || y >= 8 {

        return errors.New("координаты за пределами доски")

    }

    b[x][y] = Square{piece, color}

    return nil

}

func (b Board) IsEmpty(x, y int) bool {

    if x < 0 || x >= 8 || y < 0 || y >= 8 {

        return false

    }

    return b[x][y].Piece == Empty

}

// Cоздание глубокой копии доски

func (b Board) Copy() Board {

    var newBoard Board

    for i := 0; i < 8; i++ {

        for j := 0; j < 8; j++ {

            newBoard[i][j] = b[i][j]

        }

    }

    return newBoard

}

**Файл piece.go**

package board

type Piece int

const (

    Empty Piece = iota

    Pawn

    Knight

    Bishop

    Rook

    Queen

    King

)

type Color int

const (

    White Color = iota

    Black

)

type Square struct {

    Piece Piece

    Color Color

}

**Файл evaluation.go**

package evaluation

import (

    "chess-engine/board"

    "chess-engine/move"

    "chess-engine/util"

    "math"

)

var PieceValues = map[board.Piece]int{

    board.Pawn:   100,

    board.Knight: 320,

    board.Bishop: 330,

    board.Rook:   500,

    board.Queen:  900,

    board.King:   20000,

}

// Бонусы за контроль центра для пешек и легких фигур

var centerBonus = [8][8]int{

    {0, 0, 0, 0, 0, 0, 0, 0},

    {0, 0, 0, 0, 0, 0, 0, 0},

    {0, 0, 5, 10, 10, 5, 0, 0},

    {0, 5, 10, 20, 20, 10, 5, 0},

    {0, 5, 10, 20, 20, 10, 5, 0},

    {0, 0, 5, 10, 10, 5, 0, 0},

    {0, 0, 0, 0, 0, 0, 0, 0},

    {0, 0, 0, 0, 0, 0, 0, 0},

}

func Evaluate(b board.Board) int {

    score := 0

    //Битовые маски для подсчета фигур

    var whitePieces, blackPieces uint64

    for i := 0; i < 8; i++ {

        for j := 0; j < 8; j++ {

            piece, color, \_ := b.GetPiece(i, j)

            if piece == board.Empty {

                continue

            }

            //Индекс клетки в 64-битной маске

            bitPos := uint(i\*8 + j)

            if color == board.White {

                whitePieces |= (1 << bitPos)

                score += PieceValues[piece]

                //Бонус за контроль центра для пешек и легких фигур

                if piece == board.Pawn || piece == board.Knight || piece == board.Bishop {

                    score += centerBonus[i][j]

                }

            } else {

                blackPieces |= (1 << bitPos)

                score -= PieceValues[piece]

                //Штраф за контроль центра для черных (отзеркаливаем доску)

                if piece == board.Pawn || piece == board.Knight || piece == board.Bishop {

                    score -= centerBonus[7-i][j]

                }

            }

        }

    }

    //Подсчет активных фигур

    whiteCount := util.PopCount(whitePieces)

    blackCount := util.PopCount(blackPieces)

    //Бонус за мобильность на основе количества фигур

    score += (whiteCount - blackCount) \* 10

    //Штраф за короля под шахом

    if move.IsKingInCheck(b, board.White) {

        score -= 50

    }

    if move.IsKingInCheck(b, board.Black) {

        score += 50

    }

    //Бонус за безопасность короля

    score += kingSafety(b, board.White)

    score -= kingSafety(b, board.Black)

    return score

}

// Оценка безопасности короля

func kingSafety(b board.Board, color board.Color) int {

    safetyScore := 0

    //Находим позицию короля

    var kingX, kingY int

    for x := 0; x < 8; x++ {

        for y := 0; y < 8; y++ {

            piece, pieceColor, \_ := b.GetPiece(x, y)

            if piece == board.King && pieceColor == color {

                kingX, kingY = x, y

                break

            }

        }

    }

    //Проверяем близость фигур противника

    opponentColor := board.Black

    if color == board.Black {

        opponentColor = board.White

    }

    for x := 0; x < 8; x++ {

        for y := 0; y < 8; y++ {

            piece, pieceColor, \_ := b.GetPiece(x, y)

            if piece != board.Empty && pieceColor == opponentColor {

                //Вычисляем расстояние до короля, учитывая только близкие фигуры

                distance := int(math.Sqrt(float64((x-kingX)\*(x-kingX) + (y-kingY)\*(y-kingY))))

                if distance > 0 && distance <= 3 {

                    switch piece {

                    case board.Pawn:

                        safetyScore -= 5 / distance

                    case board.Knight:

                        safetyScore -= 10 / distance

                    case board.Bishop:

                        safetyScore -= 15 / distance

                    case board.Rook:

                        safetyScore -= 20 / distance

                    case board.Queen:

                        safetyScore -= 30 / distance

                    }

                }

            }

        }

    }

    //Бонус за защиту короля пешками

    for dx := -1; dx <= 1; dx++ {

        for dy := -1; dy <= 1; dy++ {

            nx, ny := kingX+dx, kingY+dy

            if nx >= 0 && nx < 8 && ny >= 0 && ny < 8 {

                piece, pieceColor, \_ := b.GetPiece(nx, ny)

                if piece == board.Pawn && pieceColor == color {

                    safetyScore += 10

                }

            }

        }

    }

    return safetyScore

}

**Файл generator.go**

package move

import "chess-engine/board"

//Генератор всех возможных ходов для указанного цвета

func GenerateMoves(b board.Board, color board.Color) []Move {

    var moves []Move

    for i := 0; i < 8; i++ {

        for j := 0; j < 8; j++ {

            piece, pieceColor, \_ := b.GetPiece(i, j)

            if pieceColor != color || piece == board.Empty {

                continue

            }

            switch piece {

            case board.Pawn:

                moves = append(moves, generatePawnMoves(b, i, j, color)...)

            case board.Knight:

                moves = append(moves, generateKnightMoves(b, i, j, color)...)

            case board.Bishop:

                moves = append(moves, generateBishopMoves(b, i, j, color)...)

            case board.Rook:

                moves = append(moves, generateRookMoves(b, i, j, color)...)

            case board.Queen:

                moves = append(moves, generateQueenMoves(b, i, j, color)...)

            case board.King:

                moves = append(moves, generateKingMoves(b, i, j, color)...)

                moves = append(moves, generateCastlingMoves(b, i, j, color)...)

            }

        }

    }

    //Фильтр ходов, чтобы оставить только те, которые не подвергают короля шаху

    var validMoves []Move

    for \_, m := range moves {

        newBoard := b

        if err := MakeMove(&newBoard, m); err == nil {

            if !IsKingInCheck(newBoard, color) {

                validMoves = append(validMoves, m)

            }

        }

    }

    return validMoves

}

//Генерирует ходы для рокировки

func generateCastlingMoves(b board.Board, x, y int, color board.Color) []Move {

    var moves []Move

    //Проверяем, может ли король рокироваться

    if x == 0 && y == 4 && color == board.White || x == 7 && y == 4 && color == board.Black {

        //Короткая рокировка

        if b.IsEmpty(x, y+1) && b.IsEmpty(x, y+2) {

            rookPiece, rookColor, \_ := b.GetPiece(x, y+3)

            if rookPiece == board.Rook && rookColor == color {

                moves = append(moves, Move{FromX: x, FromY: y, ToX: x, ToY: y + 2})

            }

        }

        //Длинная рокировка

        if b.IsEmpty(x, y-1) && b.IsEmpty(x, y-2) && b.IsEmpty(x, y-3) {

            rookPiece, rookColor, \_ := b.GetPiece(x, y-4)

            if rookPiece == board.Rook && rookColor == color {

                moves = append(moves, Move{FromX: x, FromY: y, ToX: x, ToY: y - 2})

            }

        }

    }

    return moves

}

//Генерирует ходы для пешки

func generatePawnMoves(b board.Board, x, y int, color board.Color) []Move {

    var moves []Move

    direction := 1 //Направление движения пешки (1 для белых, -1 для черных)

    if color == board.Black {

        direction = -1

    }

    //Ход на одну клетку вперед

    if b.IsEmpty(x+direction, y) {

        moves = append(moves, Move{FromX: x, FromY: y, ToX: x + direction, ToY: y})

    }

    //Ход на две клетки вперед (только из начальной позиции)

    if (color == board.White && x == 1) || (color == board.Black && x == 6) {

        if b.IsEmpty(x+direction, y) && b.IsEmpty(x+2\*direction, y) {

            moves = append(moves, Move{FromX: x, FromY: y, ToX: x + 2\*direction, ToY: y})

        }

    }

    //Взятие фигур по диагонали

    for \_, dy := range []int{-1, 1} {

        if !b.IsEmpty(x+direction, y+dy) {

            \_, targetColor, \_ := b.GetPiece(x+direction, y+dy)

            if targetColor != color {

                moves = append(moves, Move{FromX: x, FromY: y, ToX: x + direction, ToY: y + dy})

            }

        }

    }

    return moves

}

//Генерирует ходы для коня

func generateKnightMoves(b board.Board, x, y int, color board.Color) []Move {

    var moves []Move

    //Все возможные ходы коня

    knightMoves := [][]int{

        {x + 2, y + 1}, {x + 2, y - 1},

        {x - 2, y + 1}, {x - 2, y - 1},

        {x + 1, y + 2}, {x + 1, y - 2},

        {x - 1, y + 2}, {x - 1, y - 2},

    }

    for \_, move := range knightMoves {

        nx, ny := move[0], move[1]

        if nx >= 0 && nx < 8 && ny >= 0 && ny < 8 {

            if b.IsEmpty(nx, ny) {

                moves = append(moves, Move{FromX: x, FromY: y, ToX: nx, ToY: ny})

            } else {

                \_, targetColor, \_ := b.GetPiece(nx, ny)

                if targetColor != color {

                    moves = append(moves, Move{FromX: x, FromY: y, ToX: nx, ToY: ny})

                }

            }

        }

    }

    return moves

}

//Генерирует ходы для слона

func generateBishopMoves(b board.Board, x, y int, color board.Color) []Move {

    return generateDiagonalMoves(b, x, y, color)

}

//Генерирует ходы для ладьи

func generateRookMoves(b board.Board, x, y int, color board.Color) []Move {

    return generateStraightMoves(b, x, y, color)

}

//Генерирует ходы для ферзя

func generateQueenMoves(b board.Board, x, y int, color board.Color) []Move {

    // Ферзь сочетает возможности ладьи и слона

    moves := generateStraightMoves(b, x, y, color)

    moves = append(moves, generateDiagonalMoves(b, x, y, color)...)

    return moves

}

//Генерирует ходы для короля

func generateKingMoves(b board.Board, x, y int, color board.Color) []Move {

    var moves []Move

    //Все возможные ходы короля

    kingMoves := [][]int{

        {x + 1, y}, {x - 1, y},

        {x, y + 1}, {x, y - 1},

        {x + 1, y + 1}, {x + 1, y - 1},

        {x - 1, y + 1}, {x - 1, y - 1},

    }

    for \_, move := range kingMoves {

        nx, ny := move[0], move[1]

        if nx >= 0 && nx < 8 && ny >= 0 && ny < 8 {

            if b.IsEmpty(nx, ny) {

                moves = append(moves, Move{FromX: x, FromY: y, ToX: nx, ToY: ny})

            } else {

                \_, targetColor, \_ := b.GetPiece(nx, ny)

                if targetColor != color {

                    moves = append(moves, Move{FromX: x, FromY: y, ToX: nx, ToY: ny})

                }

            }

        }

    }

    return moves

}

//Генерирует ходы по диагонали (для слона и ферзя)

func generateDiagonalMoves(b board.Board, x, y int, color board.Color) []Move {

    var moves []Move

    directions := [][]int{

        {1, 1}, {1, -1}, {-1, 1}, {-1, -1},

    }

    for \_, dir := range directions {

        dx, dy := dir[0], dir[1]

        nx, ny := x+dx, y+dy

        for nx >= 0 && nx < 8 && ny >= 0 && ny < 8 {

            if b.IsEmpty(nx, ny) {

                moves = append(moves, Move{FromX: x, FromY: y, ToX: nx, ToY: ny})

            } else {

                \_, targetColor, \_ := b.GetPiece(nx, ny)

                if targetColor != color {

                    moves = append(moves, Move{FromX: x, FromY: y, ToX: nx, ToY: ny})

                }

                break //Прерываем цикл, если нашли фигуру

            }

            nx += dx

            ny += dy

        }

    }

    return moves

}

//Генерирует ходы по прямой (для ладьи и ферзя)

func generateStraightMoves(b board.Board, x, y int, color board.Color) []Move {

    var moves []Move

    directions := [][]int{

        {1, 0}, {-1, 0}, {0, 1}, {0, -1},

    }

    for \_, dir := range directions {

        dx, dy := dir[0], dir[1]

        nx, ny := x+dx, y+dy

        for nx >= 0 && nx < 8 && ny >= 0 && ny < 8 {

            if b.IsEmpty(nx, ny) {

                moves = append(moves, Move{FromX: x, FromY: y, ToX: nx, ToY: ny})

            } else {

                \_, targetColor, \_ := b.GetPiece(nx, ny)

                if targetColor != color {

                    moves = append(moves, Move{FromX: x, FromY: y, ToX: nx, ToY: ny})

                }

                break //Прерываем цикл, если нашли фигуру

            }

            nx += dx

            ny += dy

        }

    }

    return moves

}

//Генерирует ходы для конкретной фигуры (экспортируемая функция)

func GenerateMovesForPiece(b board.Board, x, y int, color board.Color, piece board.Piece) []Move {

    switch piece {

    case board.Pawn:

        return generatePawnMoves(b, x, y, color)

    case board.Knight:

        return generateKnightMoves(b, x, y, color)

    case board.Bishop:

        return generateBishopMoves(b, x, y, color)

    case board.Rook:

        return generateRookMoves(b, x, y, color)

    case board.Queen:

        return generateQueenMoves(b, x, y, color)

    case board.King:

        return generateKingMoves(b, x, y, color)

    }

    return nil

}

**Файл move.go**

package move

import (

    "chess-engine/board"

    "errors"

)

type Move struct {

    FromX, FromY int

    ToX, ToY     int

    PromoteTo    board.Piece //Фигура, в которую превращается пешка (0 если нет превращения)

}

// Выполняет ход на доске

func MakeMove(b \*board.Board, m Move) error {

    if m.FromX < 0 || m.FromX >= 8 || m.FromY < 0 || m.FromY >= 8 ||

        m.ToX < 0 || m.ToX >= 8 || m.ToY < 0 || m.ToY >= 8 {

        return errors.New("некорректные координаты хода")

    }

    piece, color, \_ := b.GetPiece(m.FromX, m.FromY)

    if piece == board.Empty {

        return errors.New("на начальной клетке нет фигуры")

    }

    //Создаем копию доски и проверяем, не приводит ли ход к шаху

    newBoard := \*b

    newBoard.SetPiece(m.ToX, m.ToY, piece, color)

    newBoard.SetPiece(m.FromX, m.FromY, board.Empty, color)

    //Превращение пешки в ферзя

    if piece == board.Pawn {

        if color == board.White && m.ToX == 7 { // Белая пешка на 8-й горизонтали

            if m.PromoteTo != 0 {

                newBoard.SetPiece(m.ToX, m.ToY, m.PromoteTo, color)

            } else {

                newBoard.SetPiece(m.ToX, m.ToY, board.Queen, color) // По умолчанию ферзь

            }

        } else if color == board.Black && m.ToX == 0 { // Чёрная пешка на 1-й горизонтали

            if m.PromoteTo != 0 {

                newBoard.SetPiece(m.ToX, m.ToY, m.PromoteTo, color)

            } else {

                newBoard.SetPiece(m.ToX, m.ToY, board.Queen, color) // По умолчанию ферзь

            }

        }

    }

    //Если это рокировка, перемещаем ладью

    if piece == board.King && abs(m.FromY-m.ToY) == 2 {

        if m.ToY > m.FromY {

            //Короткая рокировка

            newBoard.SetPiece(m.FromX, m.FromY+1, board.Rook, color)

            newBoard.SetPiece(m.FromX, m.FromY+3, board.Empty, color)

        } else {

            //Длинная рокировка

            newBoard.SetPiece(m.FromX, m.FromY-1, board.Rook, color)

            newBoard.SetPiece(m.FromX, m.FromY-4, board.Empty, color)

        }

    }

    //Проверяем, не приводит ли ход к шаху

    if IsKingInCheck(newBoard, color) {

        return errors.New("ход подвергает короля шаху")

    }

    //Если все в порядке, применяем ход

    \*b = newBoard

    return nil

}

// Проверяет, находится ли король под шахом

func IsKingInCheck(b board.Board, color board.Color) bool {

    //Находим позицию короля

    var kingX, kingY int

    for i := 0; i < 8; i++ {

        for j := 0; j < 8; j++ {

            piece, pieceColor, \_ := b.GetPiece(i, j)

            if piece == board.King && pieceColor == color {

                kingX, kingY = i, j

                break

            }

        }

    }

    //Определяем цвет противника

    opponentColor := board.White

    if color == board.White {

        opponentColor = board.Black

    }

    //Проверяем все фигуры противника

    for i := 0; i < 8; i++ {

        for j := 0; j < 8; j++ {

            piece, pieceColor, \_ := b.GetPiece(i, j)

            if pieceColor != opponentColor || piece == board.Empty {

                continue

            }

            //Генерируем возможные ходы фигуры противника

            moves := GenerateMovesForPiece(b, i, j, opponentColor, piece)

            for \_, m := range moves {

                if m.ToX == kingX && m.ToY == kingY {

                    return true

                }

            }

        }

    }

    return false

}

// Вспомогательная функция для вычисления абсолютного значения

func abs(x int) int {

    if x < 0 {

        return -x

    }

    return x

}

**Файл minimax.go**

package search

import (

    "chess-engine/board"

    "chess-engine/evaluation"

    "chess-engine/move"

    "encoding/json"

    "fmt"

    "math"

    "math/rand"

    "os"

    "sort"

    "sync"

    "time"

)

type transpositionTableStruct struct {

    sync.Mutex

    data map[string]SearchResult

}

var transpositionTable = transpositionTableStruct{

    data: make(map[string]SearchResult),

}

var killerMoves [32][2]move.Move

var history [12][64]int

type SearchResult struct {

    BestMoves []move.Move `json:"best\_moves"`

    Score     int         `json:"score"`

}

type SearchStats struct {

    NodesEvaluated int

    SearchTime     time.Duration

}

func LoadData() {

    if \_, err := os.Stat("transpositions.json"); err == nil {

        if data, err := os.ReadFile("transpositions.json"); err == nil && len(data) > 0 {

            if err := json.Unmarshal(data, &transpositionTable.data); err != nil {

                fmt.Printf("Ошибка загрузки транспозиционной таблицы: %v\n", err)

            } else {

                fmt.Printf("Загружено %d позиций из транспозиционной таблицы\n", len(transpositionTable.data))

            }

        }

    }

    if \_, err := os.Stat("killers.json"); err == nil {

        if data, err := os.ReadFile("killers.json"); err == nil && len(data) > 0 {

            if err := json.Unmarshal(data, &killerMoves); err != nil {

                fmt.Printf("Ошибка загрузки killer moves: %v\n", err)

            } else {

                fmt.Println("Загружены killer moves")

            }

        }

    }

}

func SaveData() {

    transpositionTable.Lock()

    defer transpositionTable.Unlock()

    if data, err := json.MarshalIndent(transpositionTable.data, "", "  "); err == nil {

        if err := os.WriteFile("transpositions.json", data, 0644); err != nil {

            fmt.Printf("Ошибка сохранения транспозиционной таблицы: %v\n", err)

        } else {

            fmt.Printf("Сохранено %d позиций в транспозиционную таблицу\n", len(transpositionTable.data))

        }

    }

    if data, err := json.MarshalIndent(killerMoves, "", "  "); err == nil {

        if err := os.WriteFile("killers.json", data, 0644); err != nil {

            fmt.Printf("Ошибка сохранения killer moves: %v\n", err)

        } else {

            fmt.Println("Сохранены killer moves")

        }

    }

}

func Minimax(b board.Board, depth int, alpha int, beta int, maximizingPlayer bool, deadline time.Time, stats \*SearchStats) SearchResult {

    if time.Now().After(deadline) {

        stats.NodesEvaluated++

        return SearchResult{Score: evaluation.Evaluate(b)}

    }

    hash := boardToString(b)

    transpositionTable.Lock()

    if result, ok := transpositionTable.data[hash]; ok && depth <= result.Score {

        transpositionTable.Unlock()

        return result

    }

    transpositionTable.Unlock()

    if depth == 0 {

        return SearchResult{Score: QuiescenceSearch(b, alpha, beta, maximizingPlayer, 4, deadline, stats)}

    }

    color := board.Black

    if maximizingPlayer {

        color = board.White

    }

    moves := move.GenerateMoves(b, color)

    if len(moves) == 0 {

        if maximizingPlayer && move.IsKingInCheck(b, board.White) {

            return SearchResult{Score: -1000000}

        } else if !maximizingPlayer && move.IsKingInCheck(b, board.Black) {

            return SearchResult{Score: 1000000}

        }

        fmt.Println("Пат или нет ходов для", color)

        stats.NodesEvaluated++

        return SearchResult{Score: evaluation.Evaluate(b)}

    }

    sortMoves(moves, b, depth)

    var bestMoves []move.Move

    var bestScore int

    if maximizingPlayer {

        bestScore = math.MinInt

    } else {

        bestScore = math.MaxInt

    }

    for \_, m := range moves {

        newBoard := b.Copy()

        if err := move.MakeMove(&newBoard, m); err != nil {

            fmt.Printf("Ошибка в MakeMove для хода %v: %v\n", m, err)

            continue

        }

        res := Minimax(newBoard, depth-1, alpha, beta, !maximizingPlayer, deadline, stats)

        stats.NodesEvaluated++

        if maximizingPlayer {

            if res.Score > bestScore {

                bestScore = res.Score

                bestMoves = []move.Move{m}

            } else if res.Score == bestScore {

                bestMoves = append(bestMoves, m)

            }

            alpha = max(alpha, bestScore)

            if beta <= alpha {

                updateKillerAndHistory(b, m, depth, color)

                break

            }

        } else {

            if res.Score < bestScore {

                bestScore = res.Score

                bestMoves = []move.Move{m}

            } else if res.Score == bestScore {

                bestMoves = append(bestMoves, m)

            }

            beta = min(beta, bestScore)

            if beta <= alpha {

                updateKillerAndHistory(b, m, depth, color)

                break

            }

        }

    }

    if len(bestMoves) == 0 {

        fmt.Println("Не удалось найти лучшие ходы для", color)

        return SearchResult{Score: evaluation.Evaluate(b)}

    }

    res := SearchResult{

        BestMoves: bestMoves,

        Score:     bestScore,

    }

    transpositionTable.Lock()

    transpositionTable.data[hash] = res

    transpositionTable.Unlock()

    return res

}

func QuiescenceSearch(b board.Board, alpha int, beta int, maximizingPlayer bool, maxDepth int, deadline time.Time, stats \*SearchStats) int {

    if time.Now().After(deadline) || maxDepth <= 0 {

        stats.NodesEvaluated++

        return evaluation.Evaluate(b)

    }

    standPat := evaluation.Evaluate(b)

    stats.NodesEvaluated++

    if maximizingPlayer {

        if standPat >= beta {

            return beta

        }

        alpha = max(alpha, standPat)

    } else {

        if standPat <= alpha {

            return alpha

        }

        beta = min(beta, standPat)

    }

    color := board.Black

    if maximizingPlayer {

        color = board.White

    }

    moves := move.GenerateMoves(b, color)

    sortMoves(moves, b, 0)

    for \_, m := range moves {

        targetPiece, \_, \_ := b.GetPiece(m.ToX, m.ToY)

        piece, \_, \_ := b.GetPiece(m.FromX, m.FromY)

        newBoard := b.Copy()

        if err := move.MakeMove(&newBoard, m); err != nil {

            continue

        }

        if targetPiece != board.Empty || move.IsKingInCheck(newBoard, board.Black) || move.IsKingInCheck(newBoard, board.White) ||

            (piece == board.Pawn && (m.ToX == 0 || m.ToX == 7)) {

            score := QuiescenceSearch(newBoard, alpha, beta, !maximizingPlayer, maxDepth-1, deadline, stats)

            if maximizingPlayer {

                alpha = max(alpha, score)

                if alpha >= beta {

                    break

                }

            } else {

                beta = min(beta, score)

                if beta <= alpha {

                    break

                }

            }

        }

    }

    if maximizingPlayer {

        return alpha

    }

    return beta

}

var timeLimit time.Duration = 15 \* time.Second

func FindBestMove(b board.Board, depth int, boardColor board.Color) (move.Move, SearchStats) {

    rand.Seed(time.Now().UnixNano())

    maximizingPlayer := (boardColor == board.White)

    start := time.Now()

    deadline := start.Add(timeLimit)

    stats := SearchStats{}

    res := Minimax(b, depth, math.MinInt, math.MaxInt, maximizingPlayer, deadline, &stats)

    stats.SearchTime = time.Since(start)

    if len(res.BestMoves) == 0 {

        fmt.Println("Minimax вернул пустой список лучших ходов для", boardColor)

        moves := move.GenerateMoves(b, boardColor)

        if len(moves) == 0 {

            fmt.Println("GenerateMoves вернул пустой список для", boardColor)

            return move.Move{}, stats

        }

        return moves[0], stats // Возвращаем первый доступный ход

    }

    // Ограничиваем рандомизацию топ-2 ходами (или всеми, если их меньше)

    maxChoices := 2

    if len(res.BestMoves) < maxChoices {

        maxChoices = len(res.BestMoves)

    }

    // Сортируем ходы по эвристике для дебюта

    sort.Slice(res.BestMoves, func(i, j int) bool {

        return moveHeuristic(res.BestMoves[i]) > moveHeuristic(res.BestMoves[j])

    })

    // Выбираем случайный из топ-2

    return res.BestMoves[rand.Intn(maxChoices)], stats

}

// moveHeuristic добавляет приоритет центральным ходам в дебюте

func moveHeuristic(m move.Move) int {

    score := 0

    // Предпочтение центральным клеткам (d4, d5, e4, e5)

    centerSquares := map[int]bool{27: true, 28: true, 35: true, 36: true} // e4, e5, d4, d5

    toSquare := m.ToX\*8 + m.ToY

    if centerSquares[toSquare] {

        score += 30

    }

    return score

}

func max(a, b int) int {

    if a > b {

        return a

    }

    return b

}

func min(a, b int) int {

    if a < b {

        return a

    }

    return b

}

func GetTimeLimit() time.Duration {

    return timeLimit / time.Second

}

func SetTimeLimit(k int) {

    timeLimit = time.Duration(k) \* time.Second

}

func sortMoves(moves []move.Move, b board.Board, depth int) {

    sort.Slice(moves, func(i, j int) bool {

        moveI, moveJ := moves[i], moves[j]

        if moveI.FromX < 0 || moveI.FromX >= 8 || moveI.FromY < 0 || moveI.FromY >= 8 ||

            moveI.ToX < 0 || moveI.ToX >= 8 || moveI.ToY < 0 || moveI.ToY >= 8 ||

            moveJ.FromX < 0 || moveJ.FromX >= 8 || moveJ.FromY < 0 || moveJ.FromY >= 8 ||

            moveJ.ToX < 0 || moveJ.ToX >= 8 || moveJ.ToY < 0 || moveJ.ToY >= 8 {

            return false

        }

        pieceI, colorI, \_ := b.GetPiece(moveI.FromX, moveI.FromY)

        pieceJ, colorJ, \_ := b.GetPiece(moveJ.FromX, moveJ.FromY)

        targetPieceI, \_, \_ := b.GetPiece(moveI.ToX, moveI.ToY)

        targetPieceJ, \_, \_ := b.GetPiece(moveJ.ToX, moveJ.ToY)

        scoreI := 0

        if targetPieceI != board.Empty {

            targetValue := evaluation.PieceValues[targetPieceI]

            pieceValue := evaluation.PieceValues[pieceI]

            scoreI += targetValue - pieceValue/10

        }

        if pieceI == board.Pawn && (moveI.ToX == 0 || moveI.ToX == 7) {

            scoreI += 900

        }

        if pieceI == board.Knight || pieceI == board.Bishop {

            scoreI += 20

        }

        if pieceI == board.Pawn && (moveI.ToY == 3 || moveI.ToY == 4) && targetPieceI == board.Empty {

            scoreI += 20

        }

        if depth < len(killerMoves) {

            if moveI == killerMoves[depth][0] {

                scoreI += 1000

            } else if moveI == killerMoves[depth][1] {

                scoreI += 900

            }

        }

        pieceIndexI := int(pieceI) + 6\*int(colorI)

        if pieceIndexI < 12 {

            scoreI += history[pieceIndexI][moveI.ToX\*8+moveI.ToY] / 100

        }

        scoreJ := 0

        if targetPieceJ != board.Empty {

            targetValue := evaluation.PieceValues[targetPieceJ]

            pieceValue := evaluation.PieceValues[pieceJ]

            scoreJ += targetValue - pieceValue/10

        }

        if pieceJ == board.Pawn && (moveJ.ToX == 0 || moveJ.ToX == 7) {

            scoreJ += 900

        }

        if pieceJ == board.Knight || pieceJ == board.Bishop {

            scoreJ += 20

        }

        if pieceJ == board.Pawn && (moveJ.ToY == 3 || moveJ.ToY == 4) && targetPieceJ == board.Empty {

            scoreJ += 20

        }

        if depth < len(killerMoves) {

            if moveJ == killerMoves[depth][0] {

                scoreJ += 1000

            } else if moveJ == killerMoves[depth][1] {

                scoreJ += 900

            }

        }

        pieceIndexJ := int(pieceJ) + 6\*int(colorJ)

        if pieceIndexJ < 12 {

            scoreJ += history[pieceIndexJ][moveJ.ToX\*8+moveJ.ToY] / 100

        }

        return scoreI > scoreJ

    })

}

func boardToString(b board.Board) string {

    var s string

    for i := 0; i < 8; i++ {

        for j := 0; j < 8; j++ {

            piece, color, \_ := b.GetPiece(i, j)

            s += fmt.Sprintf("%d%d", piece, color)

        }

    }

    return s

}

func updateKillerAndHistory(b board.Board, m move.Move, depth int, color board.Color) {

    if depth < len(killerMoves) {

        killerMoves[depth][1] = killerMoves[depth][0]

        killerMoves[depth][0] = m

    }

    piece, \_, \_ := b.GetPiece(m.FromX, m.FromY)

    pieceIndex := int(piece) + 6\*int(color)

    if pieceIndex < 12 {

        history[pieceIndex][m.ToX\*8+m.ToY] += depth \* depth

    }

}

**Файл custom\_button.go**

package ui

import (

    "fyne.io/fyne/v2"

    "fyne.io/fyne/v2/widget"

)

type CustomButton struct {

    widget.Button

    onRightClick func()

}

func NewCustomButton(label string, onLeftClick, onRightClick func()) \*CustomButton {

    b := &CustomButton{

        Button: widget.Button{

            Text:     label,

            OnTapped: onLeftClick,

        },

        onRightClick: onRightClick,

    }

    b.ExtendBaseWidget(b)

    return b

}

func (b \*CustomButton) TappedSecondary(\*fyne.PointEvent) {

    if b.onRightClick != nil {

        b.onRightClick()

    }

}

**Файл ui.go**

package ui

import (

    "chess-engine/board"

    "chess-engine/evaluation"

    "chess-engine/move"

    "chess-engine/search"

    "fmt"

    "image/color"

    "log"

    "os"

    "time"

    "fyne.io/fyne/v2"

    "fyne.io/fyne/v2/app"

    "fyne.io/fyne/v2/canvas"

    "fyne.io/fyne/v2/container"

    "fyne.io/fyne/v2/widget"

    "github.com/faiface/beep"

    "github.com/faiface/beep/mp3"

    "github.com/faiface/beep/speaker"

)

const (

    cellSize = 80

)

var (

    pieceColors = map[board.Color]color.Color{

        board.Black: color.Black,

        board.White: color.White,

    }

    selectedColor      = color.RGBA{R: 255, G: 255, B: 0, A: 50}

    availableMoveColor = color.RGBA{R: 0, G: 255, B: 0, A: 50}

)

type ChessApp struct {

    currentBoard         board.Board

    selectedX, selectedY int

    window               fyne.Window

    grid                 \*fyne.Container

    infoLabel            \*widget.Label

    logText              \*widget.Entry

    positions            map[string]int

    gameOver             bool

    aiThinking           bool

    moveCount            int

    paused               bool

    aiDepth              int

}

func NewChessApp() \*ChessApp {

    app := &ChessApp{

        currentBoard: board.NewBoard(),

        selectedX:    -1,

        selectedY:    -1,

        logText:      widget.NewEntry(),

        positions:    make(map[string]int),

        gameOver:     false,

        aiThinking:   false,

        moveCount:    0,

        paused:       false,

        aiDepth:      20,

    }

    app.positions[boardToString(app.currentBoard)] = 1

    return app

}

func (appl \*ChessApp) Run() {

    search.LoadData()

    myApp := app.New()

    appl.window = myApp.NewWindow("Шахматы")

    icon, err := fyne.LoadResourceFromPath("Icon.png")

    if err != nil {

        log.Printf("Ошибка загрузки иконки: %v", err)

    } else {

        appl.window.SetIcon(icon)

    }

    appl.grid = appl.createBoardGrid()

    appl.infoLabel = widget.NewLabel("Ваш ход. Выберите фигуру.")

    // Настраиваем logText

    appl.logText.MultiLine = true

    appl.logText.Wrapping = fyne.TextWrapWord

    appl.logText.Disable()

    // Оборачиваем logText в контейнер с тёмным фоном

    logContainer := container.NewMax(

        canvas.NewRectangle(color.RGBA{R: 30, G: 30, B: 30, A: 255}),

        appl.logText,

    )

    content := container.NewBorder(

        nil,

        container.NewVBox(appl.infoLabel, logContainer),

        nil,

        nil,

        appl.grid,

    )

    appl.window.SetContent(content)

    appl.window.Resize(fyne.NewSize(cellSize\*8, cellSize\*8+100))

    // Сохраняем данные ИИ при закрытии окна

    appl.window.SetCloseIntercept(func() {

        search.SaveData()

        appl.window.Close()

    })

    appl.window.Show()

    myApp.Run()

}

func (app \*ChessApp) logMessage(msg string) {

    if app.moveCount != 0 || app.moveCount%2 != 0 {

        go func() {

            fyne.DoAndWait(func() {

                log.Println(msg)

                app.logText.SetText(app.logText.Text + msg + "\n")

            })

        }()

    } else {

        log.Println(msg)

        app.logText.SetText(app.infoLabel.Text + msg + "\n")

    }

}

func boardToString(b board.Board) string {

    var s string

    for i := 0; i < 8; i++ {

        for j := 0; j < 8; j++ {

            piece, color, \_ := b.GetPiece(i, j)

            s += fmt.Sprintf("%d%d", piece, color)

        }

    }

    return s

}

func (app \*ChessApp) playMoveSound() {

    go func() {

        file, err := os.Open("moveSound.mp3")

        if err != nil {

            return

        }

        defer file.Close()

        streamer, \_, err := mp3.Decode(file)

        if err != nil {

            app.logMessage(fmt.Sprintf("Ошибка декодирования MP3: %v", err))

            return

        }

        defer streamer.Close()

        speaker.Play(beep.Seq(streamer, beep.Callback(func() {})))

        time.Sleep(500 \* time.Millisecond)

    }()

}

func (app \*ChessApp) handleCellClick(x, y int) {

    if app.aiThinking && app.moveCount > 0 {

        app.infoLabel.SetText("Подождите, ИИ думает...")

        return

    }

    if app.gameOver {

        app.infoLabel.SetText("Игра завершена. Начните новую игру.")

        return

    }

    if app.selectedX == -1 {

        piece, color, err := app.currentBoard.GetPiece(x, y)

        if err != nil {

            app.logMessage(fmt.Sprintf("Ошибка при получении фигуры: %v", err))

            return

        }

        if piece != board.Empty && color == board.White && !app.paused {

            app.selectedX, app.selectedY = x, y

            app.infoLabel.SetText(fmt.Sprintf("Выбрана фигура на %c%d \n", 'a'+y, x+1))

            app.updateBoard()

        }

    } else {

        piece, color, err := app.currentBoard.GetPiece(x, y)

        if err != nil {

            app.logMessage(fmt.Sprintf("Ошибка при получении фигуры: %v", err))

            return

        }

        if piece != board.Empty && color == board.White {

            app.infoLabel.SetText("Невозможно ходить в клетку с фигурой того же цвета!")

            return

        }

        availableMoves := app.getAvailableMoves(app.selectedX, app.selectedY)

        isValidMove := false

        for \_, m := range availableMoves {

            if m.ToX == x && m.ToY == y {

                isValidMove = true

                break

            }

        }

        if !isValidMove {

            app.infoLabel.SetText("Невозможно ходить в эту клетку!")

            return

        }

        m := move.Move{FromX: app.selectedX, FromY: app.selectedY, ToX: x, ToY: y}

        if err := move.MakeMove(&app.currentBoard, m); err != nil {

            app.infoLabel.SetText("Некорректный ход: " + err.Error())

        } else {

            app.logMessage(fmt.Sprintf("Ход игрока (белые): %s%d-%s%d", string('a'+app.selectedY), app.selectedX+1, string('a'+y), x+1))

            app.playMoveSound()

            app.selectedX, app.selectedY = -1, -1

            app.moveCount++

            positionHash := boardToString(app.currentBoard)

            app.positions[positionHash]++

            app.updateBoard()

            if move.IsKingInCheck(app.currentBoard, board.Black) && app.isCheckmate(board.Black) {

                app.infoLabel.SetText("Мат! Белые победили.")

                app.logMessage("Игра завершена: мат чёрным. Победитель: Белые")

                app.gameOver = true

                search.SaveData()

                return

            } else if app.isCheckmate(board.Black) {

                app.infoLabel.SetText("Пат! Ничья.")

                app.logMessage("Игра завершена: пат для чёрных")

                app.gameOver = true

                search.SaveData()

                return

            } else if app.positions[positionHash] >= 3 {

                app.infoLabel.SetText("Ничья по правилу трёхкратного повторения!")

                app.logMessage("Игра завершена: ничья по правилу трёхкратного повторения")

                app.gameOver = true

                search.SaveData()

                return

            }

            app.makeAIMove()

        }

    }

}

func (app \*ChessApp) makeAIMove() {

    if app.gameOver {

        app.infoLabel.SetText("Игра завершена. Начните новую игру.")

        return

    }

    app.aiThinking = true

    app.infoLabel.SetText("ИИ думает...")

    go func() {

        bestMove, \_ := search.FindBestMove(app.currentBoard, app.aiDepth, board.Black)

        var message string

        if bestMove.FromX == 0 && bestMove.FromY == 0 && bestMove.ToX == 0 && bestMove.ToY == 0 {

            app.logMessage("ИИ не нашёл допустимых ходов")

            if move.IsKingInCheck(app.currentBoard, board.Black) {

                message = "Мат! Белые победили."

            } else {

                message = "Пат! Ничья."

            }

        } else {

            if err := move.MakeMove(&app.currentBoard, bestMove); err != nil {

                app.logMessage(fmt.Sprintf("Ошибка при выполнении хода ИИ: %v", err))

                message = "Ошибка ИИ: " + err.Error()

            } else {

                app.logMessage(fmt.Sprintf("Ход ИИ (чёрные): %s%d-%s%d", string('a'+bestMove.FromY), bestMove.FromX+1, string('a'+bestMove.ToY), bestMove.ToX+1))

                app.playMoveSound()

                app.moveCount++

                positionHash := boardToString(app.currentBoard)

                app.positions[positionHash]++

                app.updateBoard()

                if move.IsKingInCheck(app.currentBoard, board.White) && app.isCheckmate(board.White) {

                    message = "Мат! Чёрные победили."

                } else if app.isCheckmate(board.White) {

                    message = "Пат! Ничья."

                } else if app.positions[positionHash] >= 3 {

                    message = "Ничья по правилу трёхкратного повторения!"

                } else {

                    message = "ИИ сделал ход. Ваш ход."

                }

            }

        }

        fyne.DoAndWait(func() {

            app.infoLabel.SetText(message)

        })

        app.aiThinking = false

        app.gameOver = message != "ИИ сделал ход. Ваш ход."

        if app.gameOver {

            search.SaveData()

        }

    }()

}

func (app \*ChessApp) createCell(x, y int) fyne.CanvasObject {

    lightColor := color.RGBA{R: 240, G: 217, B: 181, A: 255}

    darkColor := color.RGBA{R: 181, G: 136, B: 99, A: 255}

    cellColor := darkColor

    if (x+y)%2 == 1 {

        cellColor = lightColor

    }

    background := canvas.NewRectangle(cellColor)

    background.SetMinSize(fyne.NewSize(cellSize, cellSize))

    var figure fyne.CanvasObject

    piece, pieceColor, err := app.currentBoard.GetPiece(x, y)

    if err != nil {

        log.Printf("Ошибка при получении фигуры: %v", err)

    }

    if piece != board.Empty {

        figure = app.createFigure(piece, pieceColor)

    } else {

        figure = canvas.NewRectangle(color.Transparent)

    }

    cellContainer := container.NewMax(

        background,

        container.NewCenter(figure),

    )

    if x == app.selectedX && y == app.selectedY {

        highlight := canvas.NewRectangle(selectedColor)

        highlight.SetMinSize(fyne.NewSize(cellSize, cellSize))

        cellContainer.Add(highlight)

    }

    if app.selectedX != -1 && app.selectedY != -1 {

        availableMoves := app.getAvailableMoves(app.selectedX, app.selectedY)

        for \_, m := range availableMoves {

            if m.ToX == x && m.ToY == y {

                availableHighlight := canvas.NewRectangle(availableMoveColor)

                availableHighlight.SetMinSize(fyne.NewSize(cellSize, cellSize))

                cellContainer.Add(availableHighlight)

            }

        }

    }

    button := NewCustomButton("", func() {

        app.handleCellClick(x, y)

    }, func() {

        app.handleRightClick()

    })

    button.Importance = widget.LowImportance

    button.Resize(fyne.NewSize(cellSize, cellSize))

    cellContainer.Add(button)

    return cellContainer

}

func (app \*ChessApp) createFigure(piece board.Piece, color board.Color) fyne.CanvasObject {

    var figure fyne.CanvasObject

    figureColor := pieceColors[color]

    switch piece {

    case board.Pawn:

        figure = canvas.NewText("♙", figureColor)

    case board.Knight:

        figure = canvas.NewText("♘", figureColor)

    case board.Bishop:

        figure = canvas.NewText("♗", figureColor)

    case board.Rook:

        figure = canvas.NewText("♖", figureColor)

    case board.Queen:

        figure = canvas.NewText("♕", figureColor)

    case board.King:

        figure = canvas.NewText("♔", figureColor)

    default:

        figure = canvas.NewText("", figureColor)

    }

    figure.(\*canvas.Text).TextSize = cellSize / 2

    figure.(\*canvas.Text).Alignment = fyne.TextAlignCenter

    figure.Resize(fyne.NewSize(cellSize, cellSize))

    return figure

}

func (app \*ChessApp) handleRightClick() {

    if app.gameOver {

        app.infoLabel.SetText("Игра завершена. Начните новую игру.")

        return

    }

    if app.aiThinking && app.moveCount > 0 {

        app.infoLabel.SetText("Подождите, ИИ думает...")

        return

    }

    app.selectedX, app.selectedY = -1, -1

    app.infoLabel.SetText("Выбранная фигура сброшена")

    app.updateBoard()

}

func (app \*ChessApp) getAvailableMoves(x, y int) []move.Move {

    piece, color, err := app.currentBoard.GetPiece(x, y)

    if err != nil || piece == board.Empty {

        return nil

    }

    allMoves := move.GenerateMoves(app.currentBoard, color)

    var availableMoves []move.Move

    for \_, m := range allMoves {

        if m.FromX == x && m.FromY == y {

            availableMoves = append(availableMoves, m)

        }

    }

    return availableMoves

}

func (app \*ChessApp) isCheckmate(color board.Color) bool {

    moves := move.GenerateMoves(app.currentBoard, color)

    if len(moves) == 0 {

        if move.IsKingInCheck(app.currentBoard, color) {

            return true

        }

        return true

    }

    return false

}

func (app \*ChessApp) updateBoard() {

    app.grid = app.createBoardGrid()

    go func() {

        fyne.DoAndWait(func() {

            app.window.SetContent(container.NewBorder(nil, container.NewVBox(app.infoLabel, container.NewMax(canvas.NewRectangle(color.RGBA{R: 30, G: 30, B: 30, A: 255}), app.logText)), nil, nil, app.grid))

            app.window.Content().Refresh()

        })

    }()

}

func (app \*ChessApp) createBoardGrid() \*fyne.Container {

    grid := container.NewGridWithColumns(8)

    for i := 7; i >= 0; i-- {

        for j := 0; j < 8; j++ {

            cell := app.createCell(i, j)

            grid.Add(cell)

        }

    }

    return grid

}

// Консольные команды

func (app \*ChessApp) PrintLastMoveEval() {

    if app.moveCount == 0 {

        log.Println("Нет ходов для оценки")

        return

    }

    score := evaluation.Evaluate(app.currentBoard)

    log.Printf("Оценка позиции: %d (положительно для белых)", score)

}

func (app \*ChessApp) Pause() {

    if app.aiThinking {

        log.Println("Невозможно поставить на паузу. ИИ делает ход")

        return

    }

    app.paused = !app.paused

    if app.paused {

        log.Println("Игра приостановлена")

        go func() {

            fyne.DoAndWait(func() {

                app.infoLabel.SetText("Игра приостановлена")

            })

        }()

    } else {

        log.Println("Игра возобновлена")

        go func() {

            fyne.DoAndWait(func() {

                app.infoLabel.SetText("Ваш ход. Выберите фигуру.")

            })

        }()

    }

}

func (app \*ChessApp) SetAIDepth(depth int) {

    app.aiDepth = depth

    log.Printf("Глубина поиска ИИ установлена на %d", depth)

}

func (app \*ChessApp) Reset() {

    app.currentBoard = board.NewBoard()

    app.selectedX, app.selectedY = -1, -1

    app.positions = make(map[string]int)

    app.positions[boardToString(app.currentBoard)] = 1

    app.gameOver = false

    app.aiThinking = false

    app.moveCount = 0

    app.paused = false

    app.updateBoard()

    app.infoLabel.SetText("Игра сброшена. Ваш ход.")

    log.Println("Игра сброшена")

}

func (app \*ChessApp) Exit(flag int) {

    if flag == 0 {

        fyne.DoAndWait(func() {

            app.window.Close()

        })

        return

    } else {

        search.SaveData()

        fyne.DoAndWait(func() {

            app.window.Close()

        })

    }

}

func (app \*ChessApp) GetAiDepth() int {

    return app.aiDepth

}

**Файл popcount.go**

package util

// PopCount возвращает количество единичных битов в x

func PopCount(x uint64) int

**Файл popcount.s**

// popcount.s

#include "textflag.h"

// func PopCount(x uint64) int

TEXT ·PopCount(SB), NOSPLIT, $0-16

MOVQ x+0(FP), AX // Загружаем аргумент x (uint64) в регистр AX

POPCNTQ AX, AX // Используем инструкцию POPCNT для подсчета битов

MOVQ AX, ret+8(FP) // Сохраняем результат в возвращаемое значение

RET

1. файл с записями о событиях в хронологическом порядке, простейшее средство обеспечения журналирования. [↑](#footnote-ref-1)
2. совершенно новая копия исходного объекта вместе со всеми вложенными объектами, которые он содержит [↑](#footnote-ref-2)
3. ситуация, при которой поиск останавливается на позиции, которая кажется стабильной, но на самом деле может быть нестабильной из-за потенциальных тактических угроз. [↑](#footnote-ref-3)
4. заготовка, готовая модель в программировании для быстрой разработки, на основе которой можно дописать собственный код [↑](#footnote-ref-4)
5. графический пользовательский интерфейс [↑](#footnote-ref-5)
6. протокол языкового сервера, набор правил, по которому среда программирования связывается с сервером и получает от него инструкции. Протокол нужен, чтобы программы для разработки могли автоматически подставлять данные в код [↑](#footnote-ref-6)