Министерство образования и науки Российской Федерации

Санкт-Петербургский государственный политехнический университет

—

Факультет технической кибернетики

**Кафедра «Компьютерные системы и программные технологии»**

**Курсовая работа за 2 семестр**

**«Шахматы»**

по дисциплине «Программирование на языках С и С++»

Выполнил

студент гр. 1081/3 Скрипаль Б.А.

<*подпись*>

Руководитель

доцент, к.т.н. Пышкин Е.В.

<*подпись*>

Санкт-Петербург

2012

Содержание

1. Техническое задание.
2. Структура проекта.
3. Класс Player.
4. Класс Desk.
5. Класс Figure.
   1. Класс Pawn
   2. Класс Bishop
   3. Класс Rook
   4. Класс King
   5. Класс Queen
   6. Класс Knight
6. Графический интерфейс
7. Летняя практика
   * 1. Техническое задание

Разработать приложение на языке С++ с использованием графической библиотеки Qt, моделирующее игру в шахматы на доске 8х8, позволяющее осуществлять игру между двумя игроками и поддерживающее графический интерфейс.

*Решаемые задачи:*

1. Разработка структуры классов, позволяющую наиболее эффективно реализовать поставленную задачу.
2. Разработка графического интерфейса, позволяющего
   1. Реализовать задуманные функции, поставленные в задаче.
   2. Осуществлять взаимодействие между программой и пользователем.

* + 1. Структура проекта

Для реализации поставленной задачи была выбрана следующая иерархия классов:

1. Класс MainWindow, осуществляющий взаимодействие между пользователем и программой
2. Класс Player, осуществляющий переключение между игроками. Так же класс Player:
   1. Осуществляет переключение между игроками
   2. Выявляет победителя
   3. Начинает новую игру

В классе Player содержится объект класса Desk.

1. Класс Desk, содержащий массив фигур (класс Figure). Класс Desk ещё:
   1. Выполняет проверку на шах и мат
   2. Осуществляет ход фигуры
   3. Осуществляет специфические ходы фигур, такие как:
      1. Рокировка
      2. Взятие на проходе
      3. Смена пешки
2. Класс Figure виртуальный класс, являющийся интерфейсом для следующих классов:
   * 1. Pawn
     2. Rook
     3. Queen
     4. King
     5. Bishop
     6. Knight

Так же в этом классе содержится перечисление с именами фигур, для их различия, а так же основные поля фигуры, такие как координата, цвет и так далее. Этот класс позволяет осуществлять следующие действия:

1. Получать доступ к функции проверки правильности хода конкретной фигуры
2. Вызывать рисование фигуры
3. Классы, производные от класса Figure осуществляют расчёт правильности хода и рисуют себя в окне. Каждый из производных классов будет в дальнейшем будет рассмотрен более подробно.

Примерная архитектура проекта:

(оранжевым показаны обычные классы, синим – графические модули).

MainWindow

Player

Desk

Figure

Pawn

King

Queen

Bishop

Knight

Rook

Test

testwariants

chanchepawn

docastling

Для взаимодействия с пользователем так же реализованы следующие классы форм:

* + 1. Класс MainWindow, реализующий окно, через которое осуществляется основное взаимодействие с пользователем.
    2. Класс ChancePawn, вызывающий диалоговое окно, при помощи которого производится выбор фигуры, на которую необходимо заменить пешку при вызове соответствующего метода.
    3. Класс DoCastling, при помощи которого вызывается диалоговое окно подтверждения совершения рокировки.
    4. Класс TestVariants, позволяющий пользователю выбрать какой тестовый сценарий следует проверить.

Взаимодействие с пользователем происходит через диалоговое окно. Для осуществления хода пользователю необходимо щелчком мыши выбрать фигуру, которой он собирается совершить ход, а затем указать клетку, на которую должна сходить эта фигура. При правильно выбранной фигуре и конечной клетке ход совершается и очередь хода переходит ко второму игроку. В противном же случае выдаётся сообщение об ошибке и пользователю необходимо совершить другой ход.

Алгоритм обработки хода устроен следующим образом:

* Считываются координаты первого и второго щелчка мыши.
* Координаты курсора преобразовываются в координаты клетки и вызывается метод класса Player «makePlayerTurn», который в свою очередь вызывает метод «makeFigureTurn» класса Desk.
* Метод makeFigureTurn
  + Проверяет корректно ли указаны начальные и конечные координаты, точнее не выходят ли они за рамки игрового поля.
  + Затем проверяется находится ли на стартовой и на конечной клетке фигура.
  + Проверяется не совершает ли игрок каких либо специфических ходов. Если да, то вызываются соответствующие методы класса Desk. После каждого из этих ходов вызывается проверка на шах. Если она не проходит, то доска возвращается в изначальное положение и метод makeFigureTurn возвращает ложь.
  + Если не один из специфических ходов не выполнен, то вызывается метод фигуры, которая стоит на стартовой клетке canFigureTurn.
    - Метод canFigureTurn вызывает метод isFinishCellTrue, который проверят не стоит ли на конечной клетке фигура, того же цвета, что и выбранная фигура.
    - Затем проверяется может ли фигура сходить на заданную клетку. Если да, то этот ход совершается и функция возвращает истину. В противном случае она возвращает ложь.
  + Далее идёт проверка на шах. Если шах подтверждается, то ход отменяется.
  + Далее проверяется достигла ли какая-нибудь пешка противоположного края. Если да, то она заменяется на другую фигуру. Эту проверку и замену осуществляет метод класса Desk changePawn.
  + Затем если всё прошло успешно метод makeFigureTurn возвращает истину. Если же что то выполнено не правильно, то возвращается ложь.
* makePlayerTurn проверяет значение makeFigureTurn. Если функция вернула ложь, то выводится сообщение об ошибке и начинается ожидание нового хода.
* Если же нет то ход передаётся другому игроку.
* Далее проводится проверка на мат, и если он есть, то определяется победитель и начинается новая игра. Победы заносятся в обую статистику.
* Проводится проверка на шах, и если она подтверждается, то выводится предупреждение о шахе.
* Далее производится перерисовка доски и программа ожидает новых действий пользователя.

1. Класс Player

Класс Player является наследником класса QObject. Player является некоторой абстракцией, позволяющей взаимодействовать пользователю с доской, чередовать ходы пользователей, а так же определять победителя. Так же этот класс содержит информацию о том, сколько игр выиграл каждый из игроков во время работы программы. Заголовочный файл класса Player имеет следующий вид:

#pragma once

#ifndef \_player\_h\_

#define \_player\_h\_

#include <QtGui>

#include "desk.h"

#include "test.h"

enum Winner

{

white,

black,

twoWinner,

nobody

};

class Player : public QObject

{

Q\_OBJECT

private:

Winner winner;

int whitePlayerWins;

int blackPlayerWins;

bool whitePlayerTurnNow;

bool checkWinner();

void printWinner();

void insertWhitePlayerTurnNow();

signals:

void printShah();

void printFalseTurn();

void printMat();

public:

Desk \*desk;

Player();

bool getWhitePlayerTurnNow();

int getWhitePlayerWins();

int getBlackPlayerWins();

void plusWinner(bool whitePlayer);

void newGame();

Winner getWinner();

void makePlayerTurn(int startX, int startY, int finishX, int finishY);

friend class Test;

};

#endif

Так же этот класс имеет в «Друзьях» класс Test. Это сделано для более удобного тестирования программы. Внутри заголовочного файла имеется перечисление Winner. Оно создано для более удобного чтение программы и используется в качестве поля объекта класса Player, в котором записан текущий победитель. В ходе игры это nobody. Так же в классе Player содержатся следующие поля:

* whitePlayerWins – поле, содержащее количество побед игрока, играющего белыми фигурами.
* blackPlayerWins – поле, содержащее количество побед игрока, играющего чёрными фигурами.
* whitePlayerTurnNow – поле, показывающее какие фигуры ходят в данный момент, белые или чёрные.
* \*desk – указатель на объект класса Desk, представляет из себя некоторую абстракцию игровой доски.

Класс Player cодержит следующие методы:

* Player() – конструктор класса
* bool getWhitePlayerTurnNow() – функция, возвращающая значение, показывающее какие фигуры ходит в данный момент.
* int getWhitePlayerWins() – функция, возвращающая количество побед игрока, играющего белыми фигурами.
* int getBlackPlayerWins() – функция, возвращающая количество побед игрока, играющего чёрными фигурами..
* void plusWinner(bool whitePlayer) – функция, увеличивающая победы белых фигур, если аргумент истина и чёрных фигур, если аргумент отрицательный.
* void newGame() – функция, создающая новую партию.
* Winner getWinner() – функция, возвращающая победителя на данный момент.
* void makePlayerTurn(int startX, int startY, int finishX, int finishY) – функция совершения хода.
* bool checkWinner() – функция проверки наличия победителя. Возвращает истину, если победитель есть и ложь если победителя нет.
* void printWinner() – распечатка победителя (победитель выводится в консоль. Функция используется только для отладки).
* void insertWhitePlayerTurnNow() – функция смены очереди хода.

Так же класс Player имеет три сигнала:

* void printShah();
* void printMat();
* void printFalseTurn();

Они сообщают, что необходимо вывести на экран сообщение о том, что игроку поставлен шах, мат или о том, что он неправильно сходил.

Функция Player()

Функция является конструктором. Тело функции:

Player::Player()

{

desk = new Desk;

winner = nobody;

whitePlayerTurnNow = true;

whitePlayerWins=0;

blackPlayerWins=0;

}

Функция создаёт в динамической памяти объект класса Desk, присваивает победителю значение nobody, ставит очередь хода белого игрока, а так же обнуляет количество побед белых и чёрных фигур.

Функция void makePlayerTurn(int startX, int startY, int finishX, int finishY)

Функция, позволяющая осуществлять совершение хода. Тело функции:

void Player::makePlayerTurn(int startX, int startY, int finishX, int finishY)

{

bool isTurnTrue = desk->makeFigureTurn(startX,startY,finishX,finishY,whitePlayerTurnNow);

if(isTurnTrue == true)

{

insertWhitePlayerTurnNow();

}

else

printFalseTurn();

checkWinner();

if(desk->checkShah(whitePlayerTurnNow)==true)

{

printShah();

}

}

Сначала функция вызывает метод класса Desk, которому передаются координаты начальной и конечной клеток, а так же цвет фигур, чей сейчас ход. Эта функция возвращает истину если такой ход возможен и ложь, если невозможен. При правильном ходе меняется цвет фигур, которые ходят сейчас, затем производятся проверки на шах и мат. При неправильном ходе на экран выводится окно сообщений, которая говорит пользователю о недопустимости хода. Если игроку, чей ход сейчас будет, грозит шах или мат, информация об этом так же выводится на экран в виде диалоговых окон с сообщением.

Функция bool checkWinner()

Функция определения победителя. Тело функции:

bool Player::checkWinner()

{

if(desk->checkMat(whitePlayerTurnNow)==true)

{

if(whitePlayerTurnNow==false)

{

winner = black;

plusWinner(!whitePlayerTurnNow);

printMat();

newGame();

}

else

{

winner = white;

plusWinner(whitePlayerTurnNow);

printMat();

newGame();

}

return true;

}

for(int i=0;i<32;i++)

if(desk->getFigure(i)->getType()!=king && desk->getFigure(i)->isEat() == false)

return false;

else

{

winner = twoWinner;

printMat();

newGame();

return true;

}

return false;

}

Сначала вызывается метод класса Desk, проверяющий наличие мата. Если проверка на мат заканчивается успешно, то смотрится чья сейчас очередь ходить. Этому игроку и засчитывается победа. Далее на экран выводится сообщение о наличие победителя, затем начинается новая игра. Так же при патовых ситуациях победа присваивается обоим игрокам и начинается новая игра.

1. Класс Desk

Класс Desk так же является наследником QObject. По своей сути, класс Desk является некой моделью игровой доски, которая представлена в виде списка фигур с их координатами. Так же класс Desk содержит методы проверки ситуации на игровом поле (проверки на шах и мат), методы передвижения фигур на доске. Они включают в себя так же специфические ходы фигур (рокировка, взятие на проходе, смена пешки). Заголовок класса выглядит следующим образом:

#ifndef \_desk\_h\_

#define \_desk\_h\_

class Figure;

#include <QtGui>

#include "bishop.h"

#include "knight.h"

#include "king.h"

#include "pawn.h"

#include "queen.h"

#include "rook.h"

#include "chanchepawn.h"

#include "docastling.h"

struct PlayingBoard

{

TypeOfFigure figuresType;

bool isFigureWhite;

bool isCellWhite;

Figure \*figureOnCell;

};

class Desk: public QObject

{

Q\_OBJECT

private:

Figure \*wKing;

Figure \*bKing;

PlayingBoard \*\*playingBoard;

Figure \*\*figure;

void cancelTurn(const int coordinateX, const int coordinateY, const int number);

void refreshPlayingBoard();

void chanchePawn(const bool whiteColour);

void setFigure(const int numberOfFigure,const int type);

bool castling(const int startX, const int startY,const int finishX,const int finishY,const bool whitePlayer);

bool enPassant(const int startX,const int startY,const int finishX, const int finishY,const bool whitePlayer);

signals:

void printShah();

public:

Desk();

Figure\* getFigure(const int number) const;

void createNewDesk();

bool checkMat(const bool whitePlayer);

bool checkShah(const bool whitePlayer);

bool makeFigureTurn(const int startX,const int startY,const int finishX,const int finishY, const bool whitePlayerTurn);

void printPlayingBoard();

~*Desk*();

friend class Test;

};

#endif

Класс Test так же сделан дружественным классу Desk. Это сделано для тестирования некоторых функций класса Desk. Так же в одном модуле с классом Desk объявлена структура PlayingBoard. ПО сути она является ещё одним уровнем абстракции, показывающая игровой процесс более наглядно. Класс Desk содержит следующие поля:

* Figure \*wKing – указатель на белого короля
* Figure \*bKing – указатель на чёрного короля
* PlayingBoard \*\*playingBoard – указатель на двумерный массив структуры PlayingBoard
* Figure \*\*figure – двумерный массив фигур

Следующие методы:

* Desk() – конструктор класса.
* Figure\* getFigure(const int number) const; - метод, возвращающий указатель на фигуру по её номеру в массиве.
* void createNewDesk() – функция создания новой доски.
* bool checkMat(const bool whitePlayer) – проверка на мат.
* bool checkShah(const bool whitePlayer) – проверка на шах.
* bool makeFigureTurn(const int startX,const int startY,const int finishX,const int finishY, const bool whitePlayerTurn) – функция совершения фигурой её хода.
* void printPlayingBoard() – функция распечатки доски в консоль.
* ~*Desk*() – деструктор.
* void cancelTurn(const int coordinateX, const int coordinateY, const int number) – функция возвращения состояния доски на начало хода.
* void refreshPlayingBoard() – функция синхронизации PlayingBoard и массива фигур.
* void chanchePawn(const bool whiteColour) – функция смены пешки.
* void setFigure(const int numberOfFigure,const int type) – функция замены фигуры на любую другую из списка.
* bool castling(const int startX, const int startY,const int finishX,const int finishY,const bool whitePlayer) – функция совершения рокировки.
* bool enPassant(const int startX,const int startY,const int finishX, const int finishY,const bool whitePlayer) – функция взятия на проходе.

Так же класс Desk содержит сигнал void printShah(), показывающий, что надо выводить диалоговое окно, предупреждающее игрока о шахе.

Функция Desk()

Конструктор класса Desk. Тело функции:

Desk::Desk()

{

figure = new Figure \*[32];

playingBoard = new PlayingBoard \*[8];

for(int i=0;i<8;i++)

playingBoard[i] = new PlayingBoard[8];

for(int i=0;i<8;i++)

for(int k=0;k<8;k++)

{

if(k%2==0 && i%2==0 || k%2==1 && i%2==1)

playingBoard[i][k].isCellWhite=false;

else

playingBoard[i][k].isCellWhite=true;

playingBoard[i][k].figuresType=none;

playingBoard[i][k].figureOnCell=NULL;

}

createNewDesk();

for(int i=0;i<32;i++)

if(figure[i]->getX()==4 && figure[i]->getY()==1 && figure[i]->getColour()==true)

wKing = figure[i];

for(int i=0;i<32;i++)

if(figure[i]->getX()==4 && figure[i]->getY()==8 && figure[i]->getColour()==false)

bKing = figure[i];

refreshPlayingBoard();

}

Сначала в динамической памяти создаётся двумерная матрица типа PlayingBoard и массив фигур типа Figure. Затем вызывается функция создания новой доски ,которая присваивает элементам массива фигур конкретные фигуры с конкретными координатами. Далее указателям на белого и чёрного короля присваивается белый и чёрный король соответственно. Далее массив фигур и двумерная матрица синхронизируются.

Функция bool checkMat(const bool whitePlayer)

Функция проверяет ситуацию на доске на наличие матовой ситуации. В качестве аргумента функции передаётся цвет игрока, для того что бы знать для какой из сторон проверять наличие мата. Тело функции:

bool Desk::checkMat(const bool whitePlayer)

{

Figure \*king;

if(whitePlayer == true)

king = wKing;

else

king = bKing;

int startX;

int startY;

startX = king->getX();

startY = king->getY();

int numberOfEatFigure =-1;

for(int j=0;j<32;j++)

if(figure[j]->getColour()==king->getColour() && figure[j]->isEat()==false && figure[j]->getType()!=king->getType())

if(checkShah(whitePlayer)==false)

return false;

for(int x=1;x<9;x++)

for(int y=1;y<9;y++)

if(king->*canFigureTurn*(x,y,\*this))

{

for(int i=0;i<32;i++)

if(figure[i]->getX()==x && figure[i]->getY()==y&&figure[i]->isEat()==false&&figure[i]->getColour()!=king->getColour())

{

numberOfEatFigure=i;

break;

}

if(numberOfEatFigure!=-1)

{

king->eatFigure(x,y,\*this);

if(checkShah(king->getColour())==false)

{

king->setX(startX);

king->setY(startY);

king->increaceSteps(false);

figure[numberOfEatFigure]->eatFigure(false);

return false;

}

else

{

king->setX(startX);

king->setY(startY);

king->increaceSteps(false);

}

}

else

{

king->putFigure(x,y);

if(checkShah(king->getColour())==false)

{

king->setX(startX);

king->setY(startY);

king->increaceSteps(false);

return false;

}

else

{

king->setX(startX);

king->setY(startY);

king->increaceSteps(false);

}

}

}

return true;

}

Сначала создаётся указатель типа Figure, которому присваивается указатель на короля того цвета, чей сейчас ход. Далее проверяется если у игрока есть ещё фигуры кроме короля и королю не угрожает шах, то мата нет. Далее проверяются все клетки, в которые король может сходить. Если при нахождении во всех этих клетках королю объявлен шах, то проверка на мат является положительной, если нет – отрицательной.

Функция bool checkShah(const bool whitePlayer)

Функция осуществляет проверку шаха. В качестве аргумента функции передаётся цвет короля, которого надо проверить. Если шах есть, функция возвращает истину, если нету – то ложь. Тело функции:

bool Desk::checkShah(const bool whitePlayer)

{

Figure \*king;

if(whitePlayer == true)

king = wKing;

else

king = bKing;

for(int i=0;i<32;i++)

if(figure[i]->getColour()!=king->getColour() && figure[i]->isEat() == false && figure[i]->*canFigureTurn*(king->getX(),king->getY(),\*this)==true)

{

return true;

}

return false;

}

В начале функции создается указатель типа Figure на короля. Он указывает на белого короля если аргумент истина, на чёрного – если ложь. Далее проверяется, если хоть одна из фигур следующим ходом может съесть короля, то шах есть и функция возвращает истину, а если нету ни одной такой фигуры, то функция возвращает ложь.

Функция bool makeFigureTurn(const int startX,const int startY,const int finishX,const int finishY, const bool whitePlayerTurn)

Функция осуществляет проверку корректности координат, а так же на правильность хода. Если ход совершён верно, то функция возвращает истину, иначе – ложь. В качестве координат функции передаются начальные и конечные координаты фигуры, а так же цвет фигур, которые ходят сейчас. Тело функции:

bool Desk::makeFigureTurn(const int startX,const int startY,const int finishX,const int finishY, const bool whitePlayerTurnNow)

{

int numberOfFirstFigure= -1;

int numberOfSecondFigure =-1;

for(int i=0;i<32;i++)

if(getFigure(i)->getX()==startX && getFigure(i)->getY()==startY && getFigure(i)->isEat()==false)

{

numberOfFirstFigure=i;

break;

}

if(numberOfFirstFigure ==-1)

return false;

for(int i=0;i<32;i++)

if(getFigure(i)->getX()==finishX && getFigure(i)->getY()==finishY && getFigure(i)->isEat()==false)

{

numberOfSecondFigure=i;

break;

}

if(getFigure(numberOfFirstFigure)->getColour()!=whitePlayerTurnNow)

return false;

if(numberOfSecondFigure!=-1 && getFigure(numberOfSecondFigure)->getColour()==whitePlayerTurnNow)

return false;

if(castling(startX,startY,finishX,finishY,whitePlayerTurnNow)==true)

{

if(checkShah(whitePlayerTurnNow)==true)

{

cancelTurn(startX,startY,numberOfFirstFigure);

cancelTurn(finishX,finishY,numberOfSecondFigure);

return false;

}

return true;

}

if(enPassant(startX,startY,finishX,finishY,whitePlayerTurnNow)==true)

{

if(checkShah(whitePlayerTurnNow)==true)

{

cancelTurn(startX,startY,numberOfFirstFigure);

cancelTurn(finishX,finishY,numberOfSecondFigure);

return false;

}

return true;

}

if(getFigure(numberOfFirstFigure)->*canFigureTurn*(finishX,finishY,\*this)==false)

return false;

else

{

if(numberOfSecondFigure!=-1)

getFigure(numberOfFirstFigure)->eatFigure(finishX,finishY,\*this);

else

getFigure(numberOfFirstFigure)->putFigure(finishX,finishY);

if(checkShah(whitePlayerTurnNow)==true)

{

cancelTurn(startX,startY,numberOfFirstFigure);

cancelTurn(finishX,finishY,numberOfSecondFigure);

return false;

}

chanchePawn(whitePlayerTurnNow);

refreshPlayingBoard();

return true;

}

}

В начале функции объявляются две переменные типа integer, которым будут присвоены номера индексов съедаемой и съедающей фигуры соответственно. По умолчанию им задаётся значение -1. Затем массив перебирается два раза в поисках фигуры, которая съедает и фигуры которую съедают. Если значение первой переменной так и осталось равным -1, то это означает что никакая фигура не была выбрана и ход уже совершён неправильно, поэтому функция завершается со значением ложь. Если значение второй переменной осталось равно -1, то это значит, что выбранная фигура просто передвигается, если же значение не равно исходному, то это значит, что выбранная фигура съедает другую фигуру. Далее проверяется не хочет ли игрок совершить рокировку или взятие на проходе. Если да, то производится проверка на шах ,и если она отрицательная, то функция завершается со значением истина. Если же игрок хочет совершить обычный ход, то вызывается метод фигуры, может ли она совершить этот ход. Если да, то этот ход совершается. Далее совершается проверка на то, не достигла ли какая-нибудь пешка противоположного края. Если да, то эта пешка заменяется на другую фигуру. Далее производится проверка на шах. Если шах есть, то ход отменяется и функция возвращает ложь. Если же угрозы шаха нету, то функция завершается со значением истина.

Функция void cancelTurn(const int coordinateX, const int coordinateY, const int number)

Функция возвращает состояние доски в состояние, до совершения хода. Функция вызывается для проверки на мат или когда фигура уже передвинута, но появляется угроза шаха, поэтому такой ход не возможен. В качестве аргумента функции передаются начальные координаты фигуры, а так же индекс этой фигуры в массиве. Тело функции:

void Desk::cancelTurn(const int coordinateX, const int coordinateY, const int number)

{

if(number==-1)

return;

getFigure(number)->setX(coordinateX);

getFigure(number)->setY(coordinateY);

getFigure(number)->eatFigure(false);

getFigure(number)->increaceSteps(false);

return;

}

Сначала проверяется, правильно ли передан номер индекса. Если номер отрицательный то такой фигуры нет и функция завершается. Если же индекс передан верно, то координаты фигуры возвращаются к тем, которые были в начале хода, если фигура была съедена, то она опять становится несъеденной, а количество шагов, совершённых фигурой уменьшается на 1.

Функция void chanchePawn(const bool whiteColour)

Функция осуществляет такой ход пешки, как замена её на любую фигуру, если она достигнет противоположного края. В качестве аргумента функции передаётся цвет пешки, которую нужно поменять. Тело функции:

void Desk::chanchePawn(const bool whiteColour)

{

int type;

for(int i=0;i<32;i++)

if(figure[i]->getType() == pawn &&figure[i]->getY() == 8 && figure[i]->getColour() == true)

{

do

{

ChanchePawn \*dial = new ChanchePawn;

dial->exec();

type=dial->getTypeOfFigure();

delete dial;

}

while(type>4);

setFigure(i,type);

}

for(int i=0;i<32;i++)

if(figure[i]->getType() == pawn &&figure[i]->getY() == 1 && figure[i]->getColour() == false)

{

do

{

ChanchePawn \*dial = new ChanchePawn;

dial->exec();

type=dial->getTypeOfFigure();

delete dial;

}

while(type>4);

setFigure(i,type);

}

refreshPlayingBoard();

return;

}

Сначала создаётся переменная типа integer, которая будет хранить тип фигуры на которую нам нужно будет заменить пешку. Затем проверяется действительно ли пешка дошла до противоположного края доски. Если да, то вызывается диалоговое окно (будет рассмотрено подробнее в дальнейшем), при помощи которого выбирается тип фигуры. Далее вызывается функция setFigure, которая заменяет пешку на выбранную фигуру. Если же пешка не дошла до конца игрового поля ,то функция просто завершается, не внося никаких изменений в массив фигур.

Функция void setFigure(const int numberOfFigure,const int type)

Функция, заменяющая одну фигуру в массиве другой фигурой. В качестве аргументов функции передаются номер фигуры в массиве фигур и тип фигуры на которую нужно заменить. Тело функции:

void Desk::setFigure(const int numberOfFigure,const int type)

{

switch(type)

{

case 1:

{

Figure \*bufFigure;

bufFigure = figure[numberOfFigure];

figure[numberOfFigure] = new Queen(bufFigure->getX(),bufFigure->getY(),bufFigure->getColour());

//delete bufFigure;

break;

}

case 2:

{

Figure \*bufFigure;

bufFigure = figure[numberOfFigure];

figure[numberOfFigure] = new Rook(bufFigure->getX(),bufFigure->getY(),bufFigure->getColour());

//delete bufFigure;

break;

}

case 3:

{

Figure \*bufFigure;

bufFigure = figure[numberOfFigure];

figure[numberOfFigure] = new Knight(bufFigure->getX(),bufFigure->getY(),bufFigure->getColour());

//delete bufFigure;

break;

}

case 4:

{

Figure \*bufFigure;

bufFigure = figure[numberOfFigure];

figure[numberOfFigure] = new Bishop(bufFigure->getX(),bufFigure->getY(),bufFigure->getColour());

//delete bufFigure;

break;

}

default:

{

break;

}

}

return;

}

Создаётся указатель типа Figure. Этому указателю присваивается значение того элемента массива, который необходимо заменить. Затем, в зависимости от типа фигуры, создаётся новая фигура. Если же требуемого типа нет функция просто завершается, не внося изменений в массив фигур.

Функция bool castling(const int startX, const int startY,const int finishX,const int finishY,const bool whitePlayer)

Функция осуществляет такой специфический ход короля ,как рокировка. В качестве аргументов функции передаются начальные и конечные координаты фигуры, а так же цвет фигур игрока ,совершающего рокировку. Функция возвращает истину если рокировка была выполнена успешно и ложь, если рокировку выполнить нельзя. Тело функции:

bool Desk::castling(const int startX, const int startY,const int finishX,const int finishY,const bool whitePlayer)

{

Figure \*king;

if(whitePlayer == true)

king = wKing;

else

king = bKing;

if(king->getX()==startX && king->getY()==startY)

if(king->getStep()==0 && abs(king->getX()-finishX)==2 && king->getY()==finishY)

{

for(int i=0; i<32;i++)

if(figure[i]->getX()==(king->getX()+(finishX-startX)/2) && figure[i]->getY()==startY)

return false;

Figure \*rook;

bool haveRook=false;

if(finishX>startX)

{

for(int i=0;i<32;i++)

if(figure[i]->getX()==8 && figure[i]->getY()==king->getY() && figure[i]->getStep()==0)

{

rook=figure[i];

haveRook = true;

break;

}

}

else

{

for(int i=0;i<32;i++)

if(figure[i]->getX()==1 && figure[i]->getY()==king->getY() && figure[i]->getStep()==0)

{

haveRook = true;

rook=figure[i];

break;

}

}

if(haveRook == false)

return false;

docastling \*cast = new docastling;

cast->exec();

bool doCast = cast->getDoCastling();

delete cast;

if(doCast==false)

return false;

if(rook->*canFigureTurn*(finishX,finishY,\*this)==false)

return false;

else

{

rook->setX((finishX+startX)/2);

rook->increaceSteps(true);

king->setX(finishX);

king->increaceSteps(true);

refreshPlayingBoard();

return true;

}

}

return false;

}

В начале создаётся указатель типа Figure. Он указывает на белого короля если ходит белый игрок или на чёрного, если ходит чёрный игрок. Далее проверяется что бы король не ходил до этого ни разу, что бы расстояние между его начальными и конечными координатами было равно 2 и что бы он перемещался по горизонтали. Далее проверяется что бы между начальной и конечной клеткой короля не находилось никакой фигуры. Затем проверяется что бы в стороне куда ходил король стояла ладья и количество ходов, которые совершила ладья было равно 0. Затем проверяется что бы между королём и ладьёй не находилось каких либо фигур. Далее выводится диалоговое окно с запросом «Совершить рокировку или нет». Если пользователь подтверждает рокировку, то она совершается, если же пользователь нажимает «отмена» или ли же хотя бы одно из вышеперечисленных условий не выполняется, то функция возвращает значение ложь и не вносит никаких изменений.

Функция bool enPassant(const int startX,const int startY,const int finishX, const int finishY,const bool whitePlayer)

Функция осуществляет такой специфический ход пешки ,как взятие на проходе. В качестве аргументов функции передаются начальные и конечные координаты фигуры ,а так же цвет игрока, который совершает ход. Функция возвращает значение истина, если ход совершён верно, и ложь, если ход неверен. Тело функции:

bool Desk::enPassant(const int startX, const int startY, const int finishX,const int finishY,const bool whitePlayer)

{

if(abs(startX-finishX)!=1)

return false;

if(abs(startY-finishY)!=1)

return false;

if(finishY>startY && whitePlayer==false)

return false;

if(finishY<startY && whitePlayer==true)

return false;

for(int i=0;i<32;i++)

if(figure[i]->getX()==startX && figure [i]->getY()==startY && figure[i]->getType()==pawn && figure[i]->getColour()==whitePlayer && figure[i]->isEat()==false)

{

for(int k=0;k<32;k++)

if(figure[k]->getX()==finishX && figure[k]->getY()==finishY && figure[k]->isEat()==true)

return false;

for(int k=0;k<32;k++)

if(figure[k]->getX()==finishX && figure[k]->getY()==startY && figure[k]->getColour()!=figure[i]->getColour() && figure[k]->isEat()==false && figure[k]->getStep()==1)

if(figure[k]->getColour()==true && figure[k]->getY()==4 || figure[k]->getColour()==false &&figure[k]->getY()==5)

{

figure[k]->eatFigure(true);

figure[i]->setX(finishX);

figure[i]->setY(finishY);

figure[i]->increaceSteps(true);

refreshPlayingBoard();

return true;

}

}

return false;

В начале проверяется что бы пешка смещалась по горизонтали и вертикали на одну клетку. Далее проверяется что бы пешка ела в правильном направлении. Затем в массиве фигур ищется пешка с соответствующими начальными координатами. Затем ищется пешка, с необходимыми координатами и проверяется, что бы эта пешка совершила только 1 ход. Если все условия соблюдены то ход совершается и функция возвращает значение истина, в противном случае ход не совершается и функция возвращает ложь.

1. Класс Figure

Класс Figure является наследником класса QObject. Класс Figure является абстрактным классом, объединяющим в себе все фигуры ,использующиеся в шахматах, а если точнее – классы этих фигур. Внутри класса Figure используется перечисление TypeOfFigure. Поле этого типа показывает какая именно это фигура. Класс фигура так же содержит следующие поля:

* int coordinateX – координата фигура по горизонтали.
* int coordinateY – координата фигуры по вертикали.
* int steps – количество шагов, совершённых фигурой.
* bool whiteColour – цвет фигуры (true – белый цвет, false – чёрный).
* bool isFigureEat – съедена ли фигура или нет.
* char symbol – символ, обозначающий фигуру (используется для вывода в консоль).

Так же этот класс содержит следующие методы:

* int getX() – возвращает координату по горизонтали.
* int getY() – возвращает координату по вертикали.
* int getStep() – возвращает количество шагов фигуры.
* TypeOfFigure getType() – возвращает тип фигуры.
* bool getColour() – возвращает цвет фигуры.
* bool isEat() – возвращает съедена ли фигура или нет.
* void setX(const int x) – изменяет координату по горизонтали.
* void setY(const int y) – изменяет координату по вертикали.
* void increaceSteps(const bool plus) – увеличивает или уменьшает количество шагов на 1 в зависимости от значения аргумента.
* void eatFigure(const bool eat) – устанавливает фигуру съеденой или несъеденой в зависимости от аргумента.
* bool isFinishCellTrue(const int xCoordinate,const int yCoordinate,const Desk& desk) – проверяет конечную клетку на корректность.
* void putFigure(const int xCoordinate,const int yCoordinate) – переносит фигуру на конкретную клетку.
* void eatFigure(const int xCoordinate,const int yCoordinate, Desk &desk) – съедает фигуру, стоящую на клетке и переносит туда исходную фигуру.

Так же класс Figure содержит два виртуальных метода:

* virtual bool canFigureTurn(const int,const int,const Desk &)=0 – проверяет может ли фигура сходить на выбранную клетку.
* virtual void printFigure(QPainter &)=0 – вырисовывает фигуру в диалоговом окне.

Функция bool isFinishCellTrue(const int xCoordinate,const int yCoordinate,const Desk& desk)

Функция проверяет конечную клетку на корректность, т.е. что бы координаты клетки не выходили за границы доски и т.д. В случае выбора правильной клетки функция возвращает значение истина, и значение ложь в противоположном случае. Тело функции:

bool Figure::isFinishCellTrue(const int xCoordinate,const int yCoordinate,const Desk& desk)

{

if(xCoordinate>8 || xCoordinate<1 || yCoordinate>8 || yCoordinate<1)

return false;

if(xCoordinate == coordinateX && yCoordinate == coordinateY)

return false;

for(int i=0;i<32;i++)

if(desk.getFigure(i)!=this)

if(desk.getFigure(i)->isFigureEat==false)

if(desk.getFigure(i)->coordinateX == xCoordinate && desk.getFigure(i)->coordinateY == yCoordinate)

if(desk.getFigure(i)->whiteColour==whiteColour)

return false;

return true;

}

Сначала проверяется что бы координаты не выходили за границы допустимых значений, т.е. что бы они не были меньше 1 или больше нуля. Затем проверяется что бы на конечной клетке не стояла фигура одного цвета с той, которой мы ходим. Если эти условия соблюдены, то функция возвращает true, иначе - false.

Функция virtual void printFigure(QPainter &)=0

Функция передаёт в QPainter изображение фигуры и координаты где её следует нарисовать. В качестве аргумента функции выступает ссылка на объект типа QPainter. Пример тела функции:

void Pawn::*printFigure*(QPainter &painter)

{

QPixmap figureIcon;

if(whiteColour==true)

figureIcon.load("pawnwhite.png");

else

figureIcon.load("pawnblack.png");

int x = 78\*coordinateX-30;

int y = 78\*coordinateY-11;

painter.drawPixmap(x,y,figureIcon);

return;

}

Функция virtual bool canFigureTurn(const int,const int,const Desk &)=0

Эта функция проверки может ли фигура сходить на заданную клетку. Для каждого класса эта функция определена по-своему, поэтому она будет рассматриваться отдельно для каждого конкретного класса.

Класс Pawn:

bool Pawn::*canFigureTurn*(const int xCoordinate,const int yCoordinate,const Desk &desk)

{

if(isFinishCellTrue(xCoordinate,yCoordinate,desk)==false)

return false;

if(abs(yCoordinate - coordinateY)>2)

return false;

bool pawnEat = false;

for(int i=0;i<32;i++)

if(desk.getFigure(i)->isEat()!=true)

if(desk.getFigure(i)->getX() == xCoordinate)

if(desk.getFigure(i)->getY() == yCoordinate)

if(desk.getFigure(i)->getColour()!=whiteColour)

{

pawnEat=true;

break;

}

if(pawnEat==false)

for(int i=0;i<32;i++)

if(desk.getFigure(i)->isEat()==false)

if(desk.getFigure(i)->getX()==coordinateX)

if(desk.getFigure(i)->getY()==coordinateY+(abs(coordinateY-yCoordinate)/(yCoordinate-coordinateY)))

return false;

if(pawnEat==false)

if(xCoordinate == coordinateX)

{

if(whiteColour==true && (coordinateY - yCoordinate)==-2 )

{

for (int k =0; k<32;k++)

if(desk.getFigure(k)->getY()==3 && desk.getFigure(k)->getX()==coordinateX && desk.getFigure(k)->isEat()==false)

return false;

if(steps==0)

return true;

}

if(whiteColour==false && (coordinateY - yCoordinate)==2)

{

for (int k =0; k<32;k++)

if(desk.getFigure(k)->getY()==6 && desk.getFigure(k)->getX()==coordinateX && desk.getFigure(k)->isEat()==false)

return false;

if(steps==0)

return true;

}

if(whiteColour==true && (coordinateY - yCoordinate)==-1 )

return true;

if(whiteColour==false && (coordinateY - yCoordinate)==1)

return true;

}

if(pawnEat==true)

if(abs(xCoordinate - coordinateX)==1)

{

if(whiteColour==true && (coordinateY - yCoordinate)==-1 )

return true;

if(whiteColour==false && (coordinateY - yCoordinate)==1)

return true;

}

return false;

}

В начале функции конечная клетка проверяется на корректность. Если проверка не пройдена, то функция завершается со значением ложь. Далее проверяется перемещение пешки по вертикали. Если пешка переместилась более чем на две клетки, то функция так же завершается со значением ложь. Затем создаётся локальная переменная bool pawnEat, показывающая, съедает ли пешка какую-либо фигуру за этот ход или нет. Изначально переменной присваивается значение ложь, и если на конечной клетке стоит фигура противоположного цвета её значение меняется на истину. Затем проверяется значение переменной pawnEat. Если оно ложно, то идёт проверка переместилась ли пешка на две клетки или на одну. Если она переместилась на две клетки и это её первый ход, на финишной клетке нет фигур и направление хода правильное, то ход совершается и функция завершается. Если на одну летку, то проверяется нет ли на конечной клетке фигур и правильное ли направление хода. Если да, то функция завершается. В обоих последних случаях в случае успеха возвращается значение истина.

Класс Bishop:

bool Bishop::*canFigureTurn*(const int xCoordinate,const int yCoordinate,const Desk &desk)

{

if(isFinishCellTrue(xCoordinate,yCoordinate,desk)==false)

return false;

if(abs(xCoordinate - coordinateX)!=abs(yCoordinate - coordinateY))

return false;

int xStep = (xCoordinate - coordinateX)/(abs(xCoordinate - coordinateX));

int yStep = (yCoordinate - coordinateY)/(abs(yCoordinate - coordinateY));

int startX = coordinateX;

int startY = coordinateY;

if(abs(xCoordinate-coordinateX)==1)

return true;

while(abs(xCoordinate - startX)>1)

{

startX+=xStep;

startY+=yStep;

for(int i=0;i<32;i++)

if(desk.getFigure(i)->isEat()==false)

if(desk.getFigure(i)->getX()==startX)

if(desk.getFigure(i)->getY() == startY)

return false;

}

return true;

}

В начале функции конечная клетка проверяется на корректность. Если проверка не пройдена, то функция завершается со значением ложь. Затем идёт проверка переместился ли слон по диагонали или нет. Если нет, то функция завершается, возвращая ложь. Далее если слон переместился на одну клетку, то возвращается значение истина. В противном случае идёт проверка нет ли каких фигур на диагонали, по которой перемещался слон. Если на ней есть какая-либо фигура, то функция завершается, возвращая ложь. Иначе возвращается значение истина.

Класс Rook:

bool Rook::*canFigureTurn*(const int xCoordinate,const int yCoordinate,const Desk &desk)

{

if(isFinishCellTrue(xCoordinate,yCoordinate, desk)==false)

return false;

if((xCoordinate - coordinateX)!=0 && (yCoordinate - coordinateY)!=0)

return false;

int stepOfRook,startX,startY;

bool lineIsClear = true;

if((xCoordinate - coordinateX)!=0)

stepOfRook=abs(xCoordinate-coordinateX)/(xCoordinate - coordinateX);

else

stepOfRook = abs(yCoordinate - coordinateY)/(yCoordinate - coordinateY);

startX = coordinateX;

startY = coordinateY;

if((xCoordinate - coordinateX)!=0)

while(lineIsClear==true && abs(startX-xCoordinate)>1)

{

startX+=stepOfRook;

for(int i=0;i<32;i++)

if(desk.getFigure(i)->isEat()==false)

if(desk.getFigure(i)->getX() == startX && desk.getFigure(i)->getY()==coordinateY)

{

lineIsClear=false;

break;

}

}

if((yCoordinate - coordinateY)!=0)

while(lineIsClear==true && abs(startY-yCoordinate)>1)

{

startY+=stepOfRook;

for(int i=0;i<32;i++)

if(desk.getFigure(i)->isEat()==false)

if(desk.getFigure(i)->getX() == coordinateX && desk.getFigure(i)->getY()==startY)

{

lineIsClear=false;

break;

}

}

if(lineIsClear == true)

return true;

return false;

}

В начале функции конечная клетка проверяется на корректность. Если проверка не пройдена, то функция завершается со значением ложь. Далее проверяется ходит ли ладья по прямой линии или нет. Затем проверяется нет ли на пути ладьи какой либо фигуры. Если есть, то функция завершается и возвращает ложь, если нет, то истину.

Класс King:

bool King::*canFigureTurn*(const int xCoordinate,const int yCoordinate,const Desk &desk)

{

if(isFinishCellTrue(xCoordinate,yCoordinate,desk)==false)

return false;

if(abs(xCoordinate - coordinateX)>1 || abs(yCoordinate - coordinateY)>1)

return false;

return true;

}

Сначала функция проверяет, конечную ячейку на корректность. Если конечная ячейка правильная, то затем проверяется перемещение фигуры по диагонали и вертикали. Если хотя бы по одному из этих направлений фигура передвигается более чем на одну клетку, то следовательно король ходит неверно и функция возвращает значение ложь. Если же ход верен, то функция возвращает значение истина.

Класс Queen:

bool Queen::*canFigureTurn*(const int xCoordinate,const int yCoordinate,const Desk &desk)

{

if(isFinishCellTrue(xCoordinate,yCoordinate,desk)==false)

return false;

if(abs(coordinateX - xCoordinate)!=abs(coordinateY-yCoordinate))

if(abs(coordinateX-xCoordinate)!=0 && (coordinateY-yCoordinate)!=0)

return false;

int stepX,stepY;

if(yCoordinate==coordinateY)

stepY=0;

else

stepY=(yCoordinate-coordinateY)/abs(yCoordinate-coordinateY);

if(xCoordinate==coordinateX)

stepX=0;

else

stepX=(xCoordinate-coordinateX)/abs(xCoordinate-coordinateX);

int startX = coordinateX;

int startY = coordinateY;

while(abs(startX-xCoordinate)>1 || abs(startY-yCoordinate)>1)

{

startX+=stepX;

startY+=stepY;

for(int i=0;i<32;i++)

if(desk.getFigure(i)->isEat()==false)

if(desk.getFigure(i)->getX()==startX && desk.getFigure(i)->getY() == startY)

return false;

}

return true;

}

Сначала функция проверяет, конечную ячейку на корректность. Далее проверяется ходит ли ферзь по диагонали или по прямой линии. Если конечная ячейка не удовлетворяет ни одному из двух условий, то функция возвращает ложь. Если этого не произошло то проверяется нет ли на линии ,по которой ходит ферзь каких-либо фигур. Если есть то функция завершается, возвращая ложь, если нет – то истину.

Класс Knight:

bool Knight::*canFigureTurn*(const int xCoordinate,const int yCoordinate,const Desk &desk)

{

if(isFinishCellTrue(xCoordinate,yCoordinate,desk)==false)

return false;

if(abs(xCoordinate - coordinateX)==1 && abs(yCoordinate - coordinateY) ==2)

return true;

if(abs(xCoordinate - coordinateX)==2 && abs(yCoordinate - coordinateY) ==1)

return true;

return false;

}

Сначала функция проверяет, конечную ячейку на корректность. Затем проверяется правильно ли сходил конь или нет. Если да, то возвращается истина, если нет – ложь.

1. Графический интерфейс

Взаимодействие программы с пользователем происходит при помощи графического интерфейса. Весь интерфейс представлен четырьмя классами формы:

* MainWindow – главное диалоговое окно
* Chancepawn – диалоговое окно, позволяющее пользователю выбрать на какую фигуру следует заменить пешку при выполнении соответствующего хода.
* Docastling –диалоговое окно, позволяющее пользователю совершить рокировку.
* Testvariants – диалоговое окно, позволяющее осуществить выбор конкретного теста из предоставленного списка тестов.

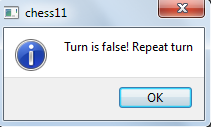
Внешний вид программы:



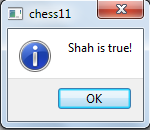
В левой части окна располагается шахматная доска с фигурами. В правой – индикатор, показывающий, какие фигуры ходят в данный момент. Через панель меню возможно осуществление следующих функций: начало новой игры, вывод на экран количество побед каждого игрока за текущий сеанс, возможность сдаться, а так же вызов списка тестов. При выборе фигуры соответствующая ячейка обводится чёрным контуром. 

(в данный момент выбрана чёрная пешка на клетке е-5)

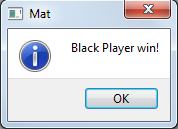
При неправильном ходе происходит вызов диалогового окна, сообщающего об ошибке.



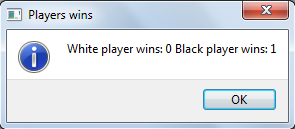
При угрозе шаха так же вызывается диалоговое окно, предупреждающее об этом:



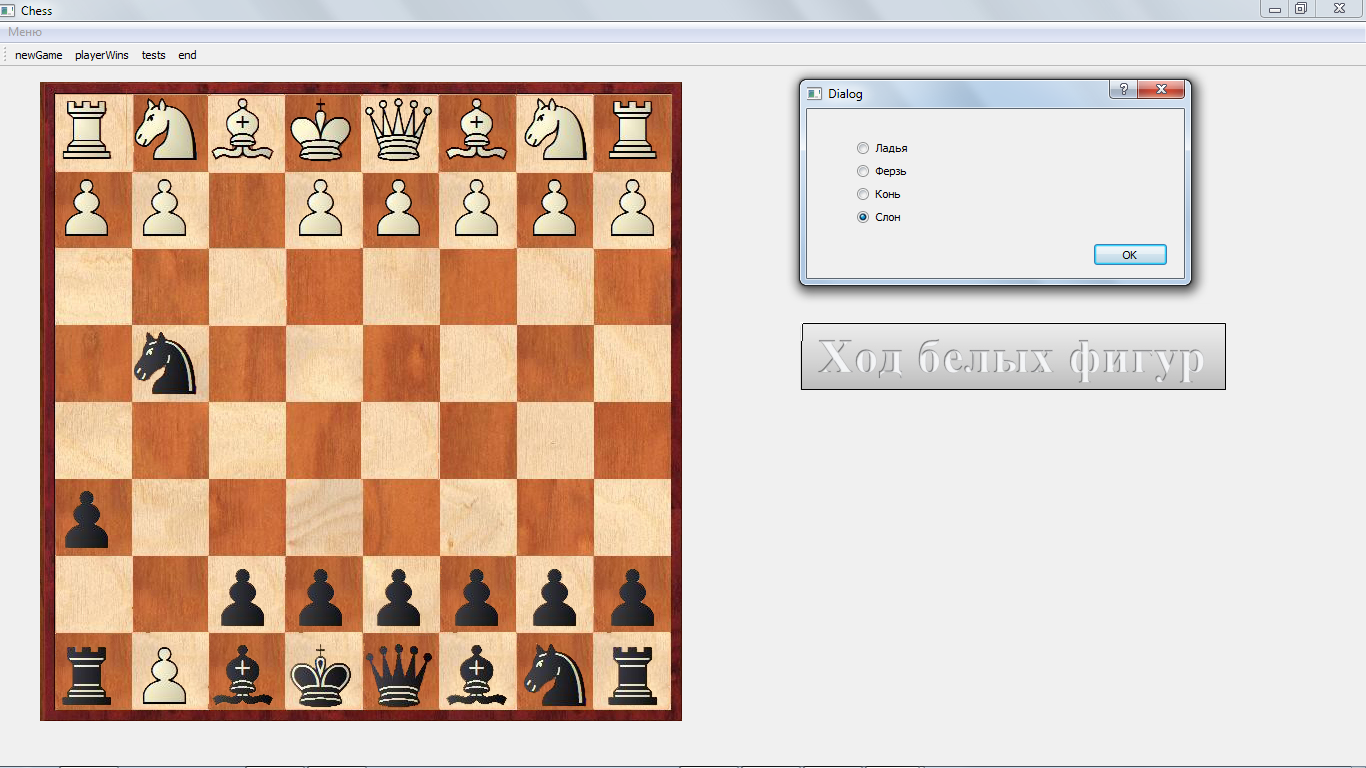
При победе выводится сообщение о том, какая сторона победила:



Статистика побед так же выводится в диалоговом окне:

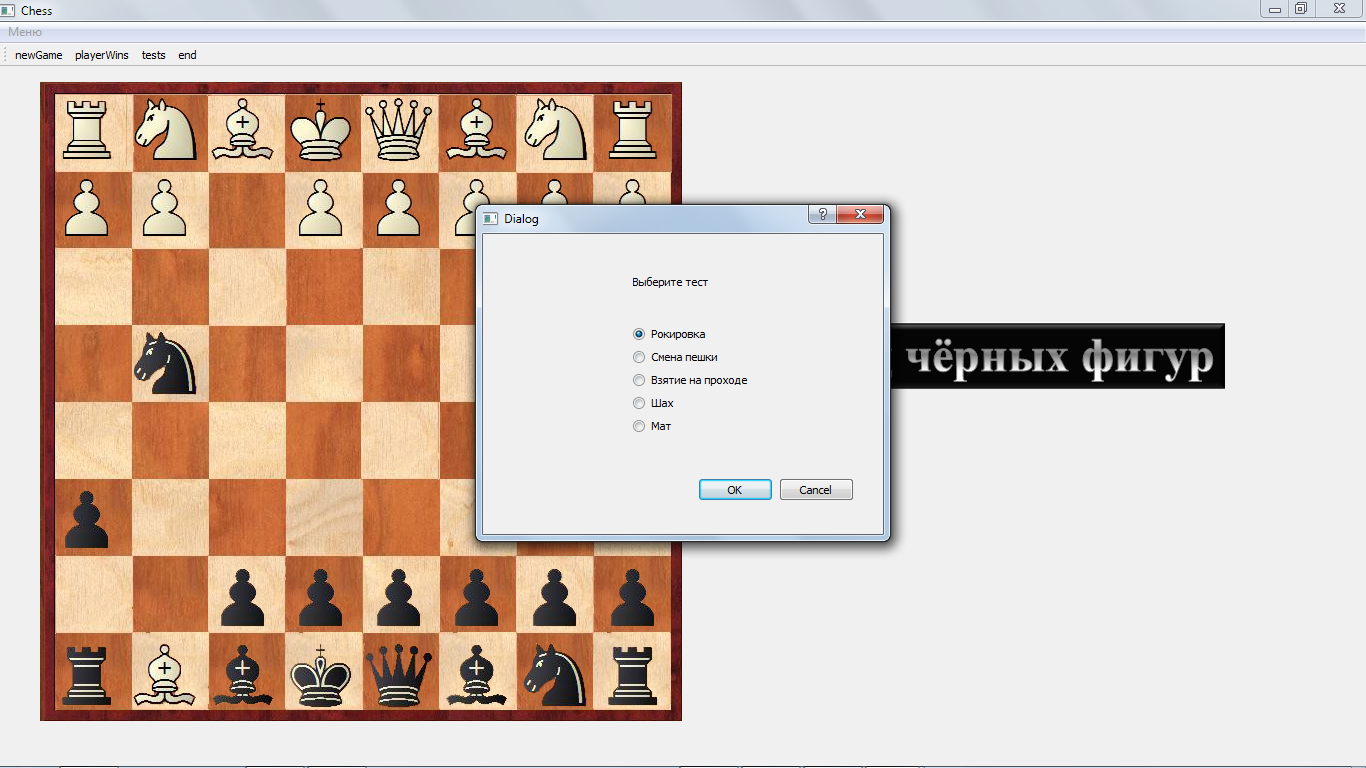


При смене пешки вызывается диалоговое окно:



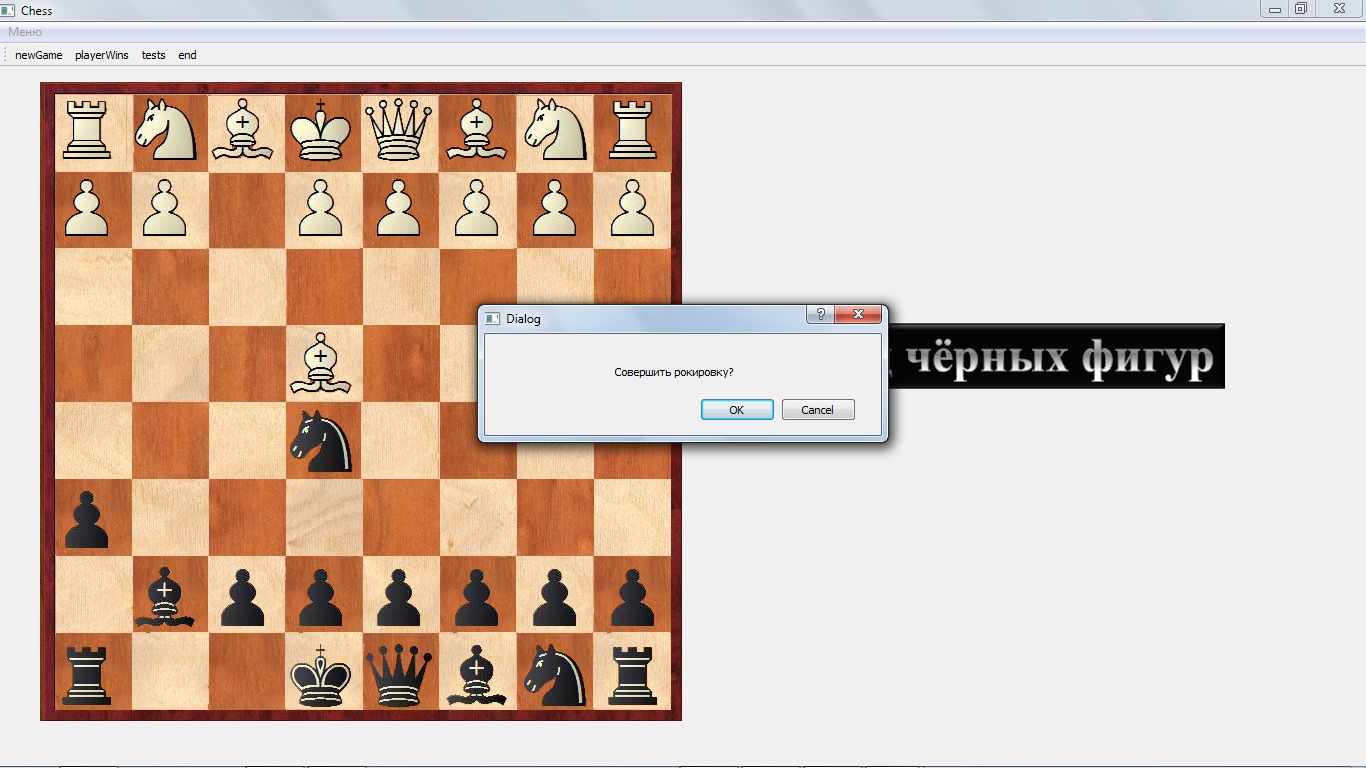
Оно состоит из 4 radioButton, сгруппированных вместе, а так же кнопки OK. Выбор фигуры, на которую следует сменить пешку осуществляется выбором соответствующего пункта и нажатием кнопки ОК.

Окно тестов выглядит похожим образом:



Выбор теста осуществляется выбором соответствующего пункта и нажатием клавиши ОК. При нажатии на клавишу Cancel окно закрывается.

При совершении рокировки вызывается диалоговое окно., запрашивающее подтверждения.



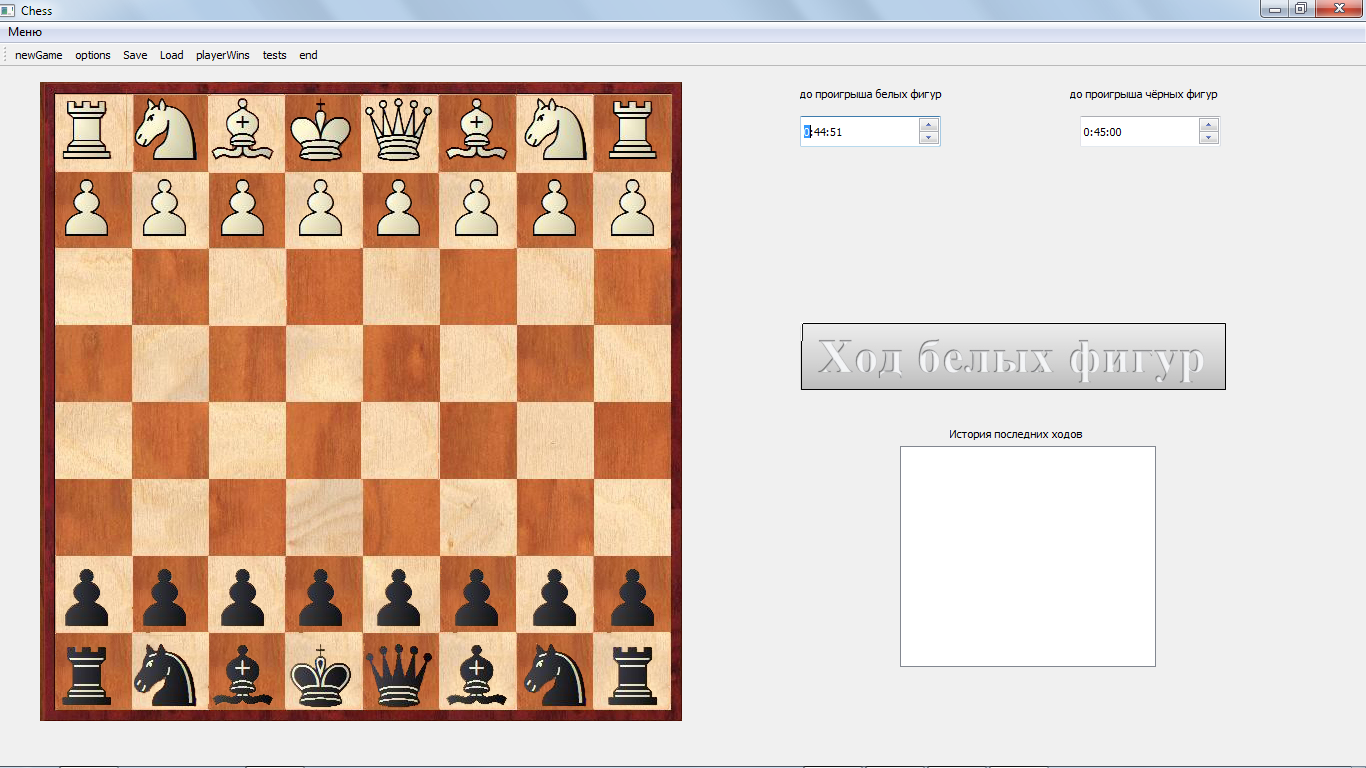
Если нажать кнопку ОК, то рокировка будет совершена, если Cancel, то ход будет отменён

1. Летняя практика.

В рамках летней практики были произведены следующие изменения курсового проекта, выполненного за 2 семестр:

1. Выбранная фигура теперь не выделяет квадратом, а мигает.
2. Добавлены таймеры, отсчитывающие время до окончания игры.
3. Добавлена возможность сохранения и загрузки последней игры.
4. Добавлена история ходов.
5. Добавлено окно настройки таймеров и частоты мигания

После внесения изменений окно программы стало выглядеть следующим образом



1. Мигание выбранной фигуры

При выборе фигуры запускается таймер (blinkTimer) с интервалом, записанным в файле конфигурации. По истечении времени вызывается функция (blinkTimeOut). Фрагмент программы отвечающей за мигание:

blinkTimer = new QTimer;

connect(blinkTimer,SIGNAL(timeout()),this,SLOT(blinkTimeOut()));

if(blinkFigure!=NULL)

blinkTimer->start(blink);

Функция blinkTimeOut:

void MainWindow::blinkTimeOut()

{

blinkFigure->eatFigure(!blinkFigure->isEat());

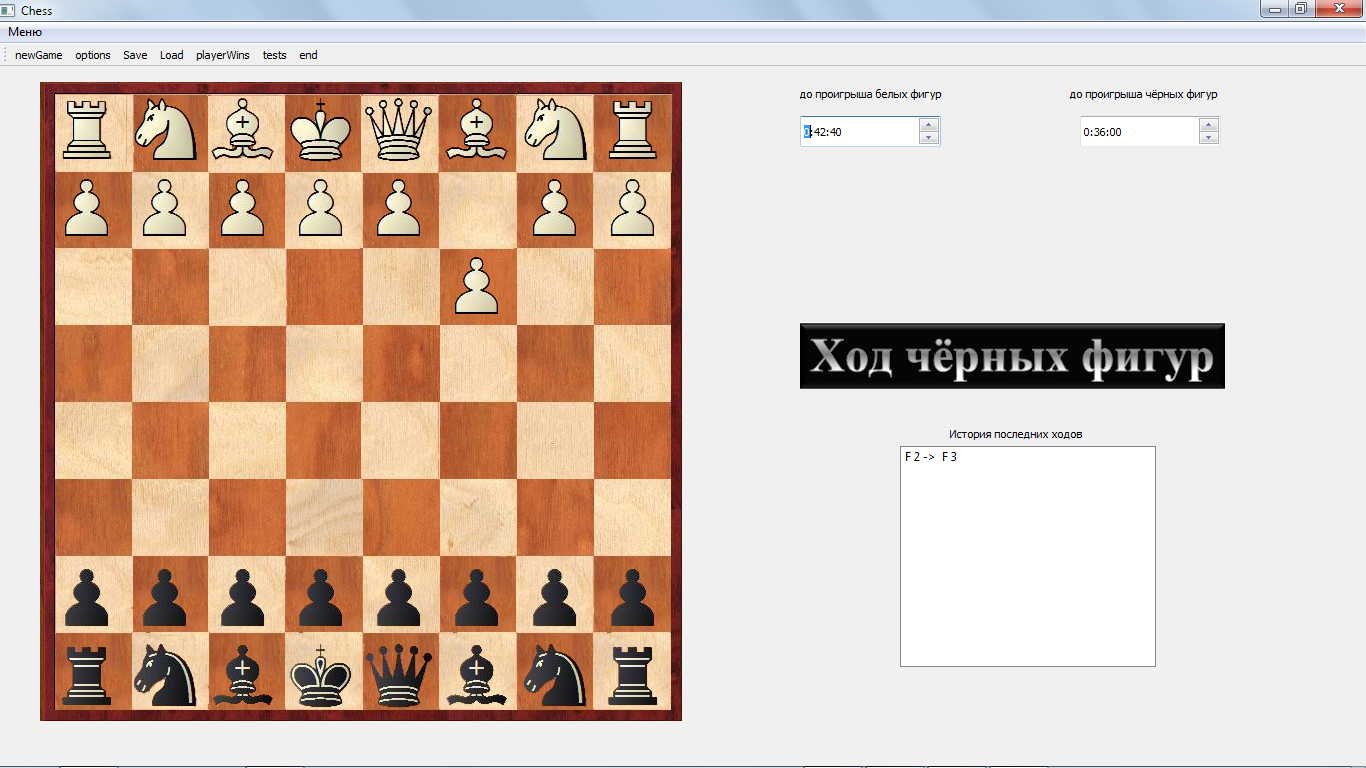
repaint();

}

Как можно заметить, по истечении времени фигура меняет свой «статус» съедена/ не съедена, затем поле перерисовывается. Переменная blink считывается из файла при создании новой игры.

1. Таймеры

Таймеры находятся в правой верхней части окна программы

 значения часов ,минут и секунд считываются из файла при создании новой игры или при загрузке игры. При смене хода таймер, который отсчитывает время останавливается и другой таймер начинает обратный отсчёт. Если таймер достигает нуля, то игрок, чей таймер первым достиг нуля проигрывает. Этой проверкой занимается функция checkTimeOut. Текст функции:

void MainWindow::checkTimeOut()

{

if(timeWhite->hour()==0 && timeWhite->minute()==0 && timeWhite->second()==0)

{

QMessageBox::information(this,"Time is over","Black Player win!");

game->plusWinner(false);

game->newGame();

}

if(timeBlack->hour()==0 && timeBlack->minute()==0&&timeBlack->second()==0)

{

QMessageBox::information(this,"Time is over","White Player win!");

game->plusWinner(true);

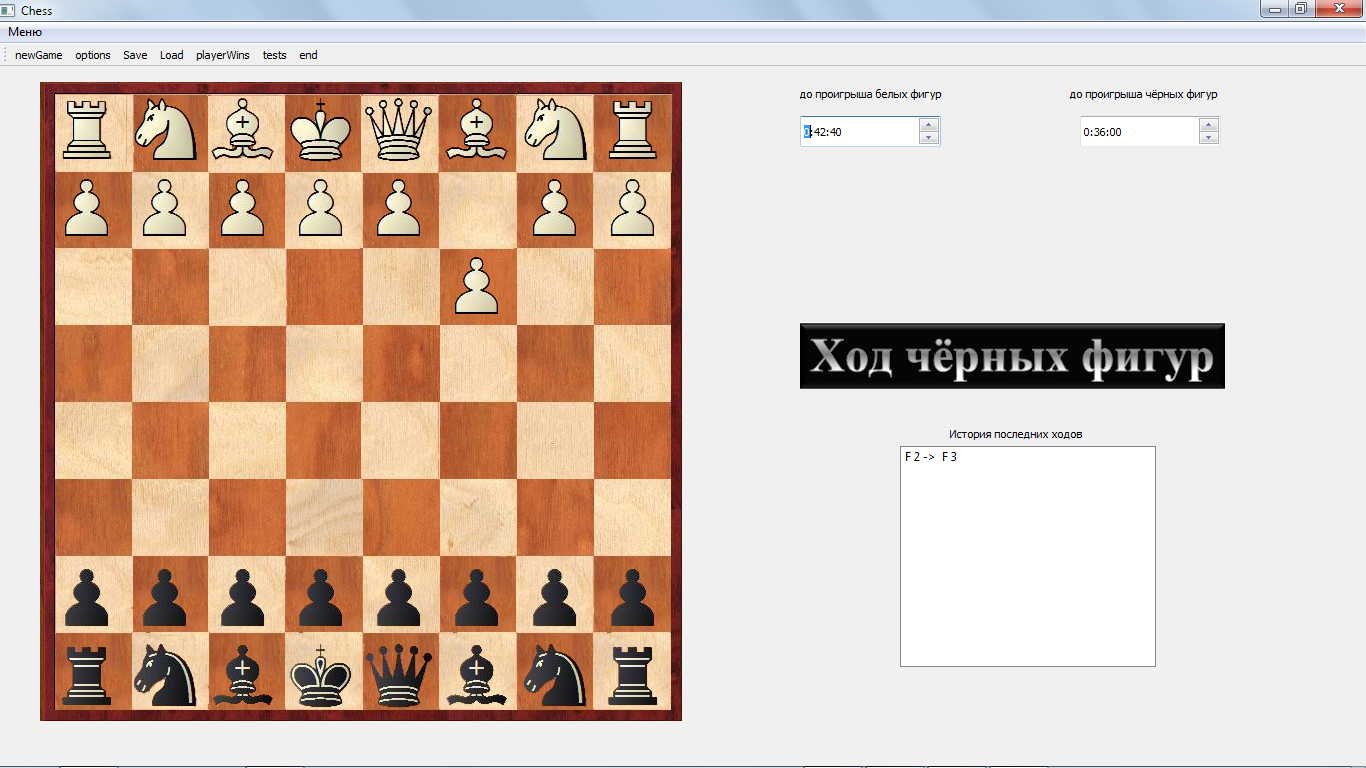
game->newGame();

}

}

1. Сохранение и загрузка игры

Добавлена возможность сохранения и загрузки одной игры. Для этого не панели команд были созданы две кнопки “Save” и “Load”, осуществляющие соответственно сохранения текущей игры и загрузку сохранённой игры.



Сохранение. При нажатии кнопки “Save” выполняется следующее: создаётся текстовый файл “save.txt”, в который записываются

* Количество побед белого и чёрного игрока
* Чей ход (1 если белого игрока, 0 – чёрного)
* Значение таймеров белого и чёрного игрока
* Параметры всех фигур (цвет ( 1 – белая фигура, 0 – чёрная), съедена ли фигура (1 – съедена, 2 – не съедена), позиция фигуры, количество ходов, сделанных фигурой)
* История совершённых фигур

Затем файл “save.txt” закрывается. Текст функции:

void MainWindow::on\_actionSave\_activated()

{

ofstream fout("save.txt");

if(fout.is\_open()==false)

{

QMessageBox::information(this,"Error!", "Save file can not opened!");

return;

}

fout<<game->getWhitePlayerWins()<<" "<<game->getBlackPlayerWins()<<std::endl;

if(game->getWhitePlayerTurnNow()==true)

fout<<1<<std::endl;

else

fout<<0<<std::endl;

fout<<ui->timeEdit->time().hour()<<" "<<ui->timeEdit->time().minute()<<" "<<ui->timeEdit->time().second()<<std::endl;

fout<<ui->timeEdit\_2->time().hour()<<" "<<ui->timeEdit\_2->time().minute()<<" "<<ui->timeEdit\_2->time().second()<<std::endl;

//обозначения при сохранении:

//0 - чёрная фигура, 1 - белая фигура

//0 - фигура не съедена, 1 - фигура съедена

//координаты фигуры

//количество ходов, сделанных фигурой

for(int i=0;i<32;i++)

{

fout<<(int) game->desk->getFigure(i)->getType()<<" ";

if(game->desk->getFigure(i)->getColour()==true)

fout<<1<<" ";

else

fout<<0<<" ";

if(game->desk->getFigure(i)->isEat()==true)

fout<<1<<" ";

else

fout<<0<<" ";

fout<<game->desk->getFigure(i)->getX()<<" ";

fout<<game->desk->getFigure(i)->getY()<<" ";

fout<<game->desk->getFigure(i)->getStep();

if (i!=31)

fout<<std::endl;

}

if(game->desk->getTurnVecSize()==0)

return;

for(int i=0;i<game->desk->getTurnVecSize()-1;i++)

{

fout<<game->desk->getTurnVecElem(i)<<std::endl;

}

fout<<game->desk->getTurnVecElem(game->desk->getTurnVecSize()-1);

fout.close();

return;

}

Загрузка. При загрузке игры открывается файл “save.txt”. Затем создаётся новая игра. Из файла считываются соответствующие значения ,которые задаются в качестве параметров фигур, таймеров и т.д. Текст функции:

void MainWindow::on\_actionLoad\_activated()

{

ifstream fin("save.txt");

if(fin.is\_open()==false)

{

QMessageBox::information(this,"Error!", "Load file can not opened!");

return;

}

game->newGame();

ui->listWidget->clear();

game->desk->clearTurnVec();

int a,b,c,d,e,f;

fin>>a>>b;

game->setWins(true,a);

game->setWins(false,b);

fin>>a;

if(a==1)

game->setWhitePlayerTurn(true);

else

game->setWhitePlayerTurn(false);

fin>>a>>b>>c;

timeWhite->setHMS(a,b,c);

fin>>a>>b>>c;

timeBlack->setHMS(a,b,c);

for(int i=0;i<32;i++)

{

fin>>a>>b>>c>>d>>e>>f;

if(c==1)

game->desk->getFigure(i)->eatFigure(true);

game->desk->getFigure(i)->setX(d);

game->desk->getFigure(i)->setY(e);

for(int k=0;k<f;k++)

game->desk->getFigure(i)->increaceSteps(true);

}

for(int i=0;!fin.eof();i++)

{

fin>>a>>b>>c>>d;

TurnCoordinate coord(a,b,c,d);

game->desk->setTurnVecElem(coord);

ui->listWidget->addItem(QString( QString(a+'A'-1) + " " + QString(b+'0') ) +

" -> "+" "+QString( QString(c+'A'-1) + " " + QString(d+'0') ));

}

repaint();

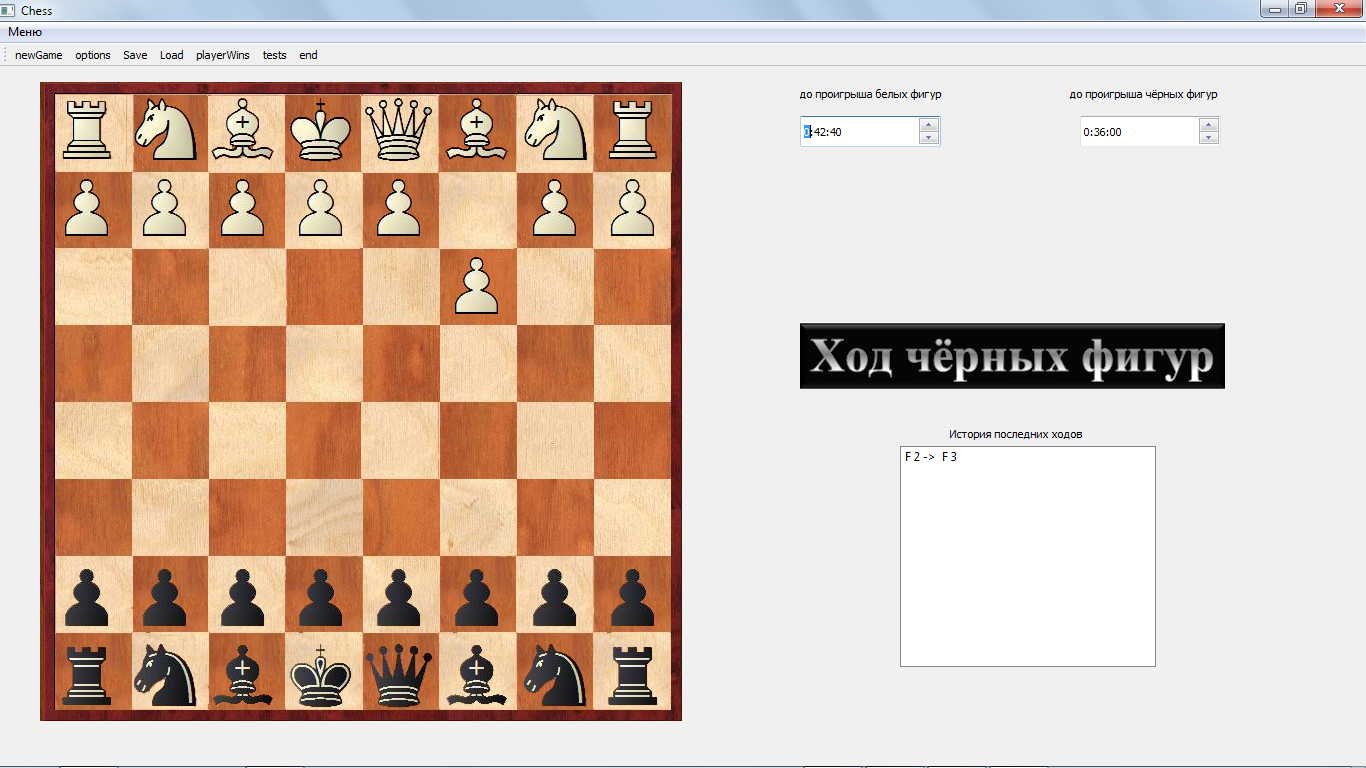
fin.close();

return;

}

1. История ходов

История ходов выводится в QListWidget в формате начальные координаты фигуры –> конечные координаты.



История ходов записывается в вектор (turnVec), хранящийся в классе Desk. Для удобства была создана структура (TurnCoordinate), хранящая в себе координаты, а так же выводящая их при необходимости. Заголовок структуры:

struct TurnCoordinate

{

int startX;

int startY;

int finishX;

int finishY;

TurnCoordinate(int sX, int sY, int fX, int fY);

friend ostream& operator << (ostream &out,const TurnCoordinate &T);

};

Вывод координат в историю ходов производится только если совершённый ход был правильным. При этом эти координаты сохраняются в вектор turnVec, для того, что бы они сохранялись и загружались при сохранении\загрузке игры. Текст программы, отвечающей за вывод:

void MainWindow::appendTurn()

{

ui->listWidget->addItem(QString( QString(readCord.startX+'A'-1) + " " + QString(readCord.startY+'0') ) +

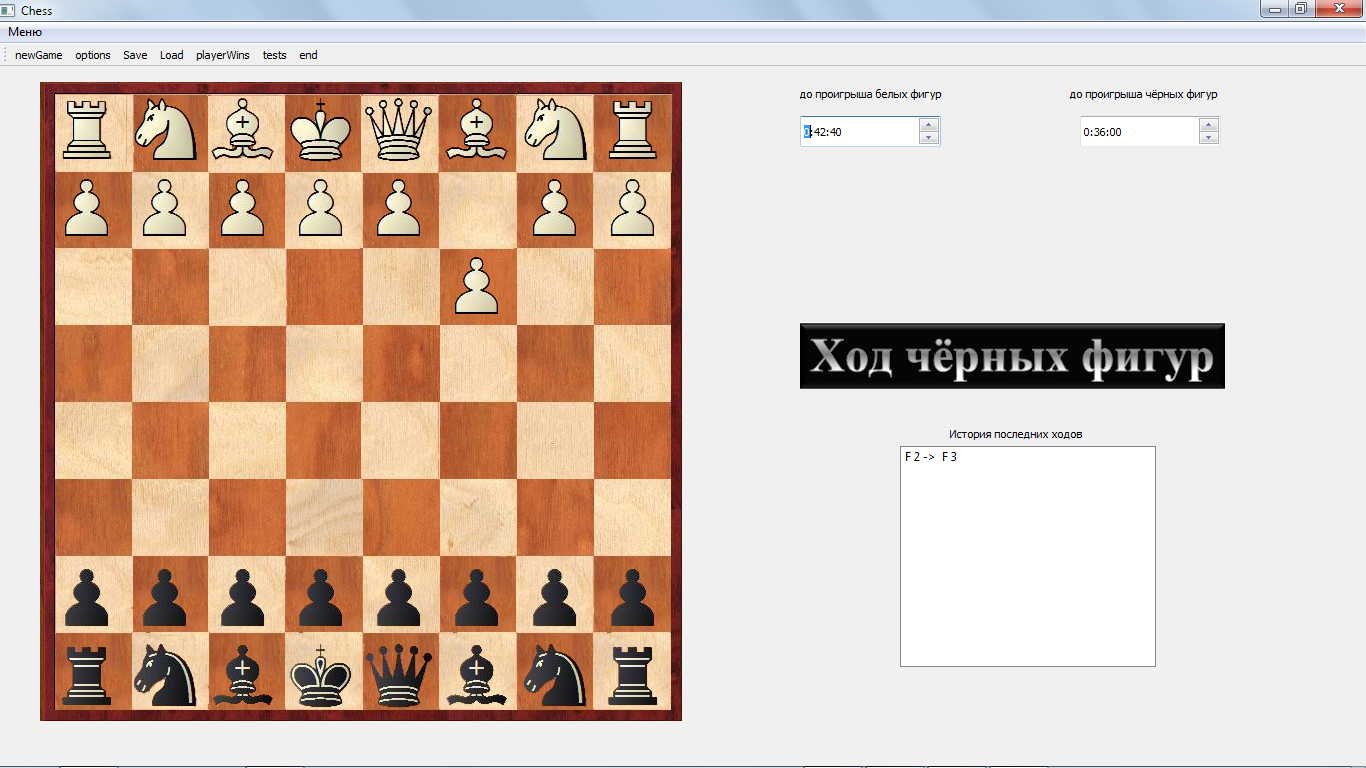
" -> "+" "+QString( QString(readCord.finishX+'A'-1) + " " + QString(readCord.finishY+'0') ));

}

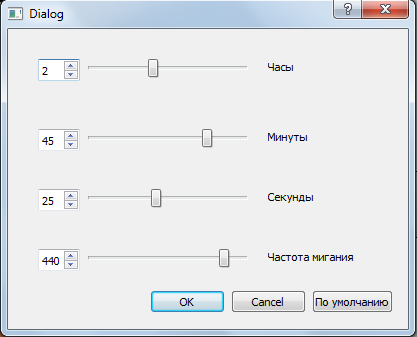
Вывод осуществляется следующим образом: при окончании правильного хода генерируется сигнал TurnWasTrue, который вызывает функцию appendTurn, выводящую в свою очередь координаты в окно.

1. Окно настройки

Окно настройки позволяет задавать частоту мигания фигуры а так же устанавливать значения таймеров. Максимальным значением таймеров является 5 часов 59 минут и 59 секунд. Максимальная частота мигания 500 мс. Окно вызывается нажатием кнопки option.



Вид диалогового окна:



Параметры задаются или при помощи слайдеров или при введении числа в spinBox. Соответствующие spinBox соединены с slider при помощи команды connect. Сохранённые настройки хранятся в файле “setting.txt”. При нажатии кнопки OK начинается новая игра с новыми параметрами. При нажатии на cancel изменения отменяются. При нажатии кнопки По умолчанию задаются «Изначальные» параметры: время игры 45 минут и частота мигания 250. Текст модуля:

#include "optiondialog.h"

#include "ui\_optiondialog.h"

optionDialog::optionDialog(QWidget \*parent) :

QDialog(parent),

ui(new Ui::optionDialog)

{

ui->setupUi(this);

std::ifstream sett("setting.txt");

int hours,minutes,seconds,blink;

sett>>hours>>minutes>>seconds>>blink;

ui->horizontalSlider->setValue(hours);

ui->spinBox->setValue(hours);

ui->horizontalSlider\_2->setValue(minutes);

ui->spinBox\_2->setValue(minutes);

ui->horizontalSlider\_3->setValue(seconds);

ui->spinBox\_3->setValue(seconds);

ui->horizontalSlider\_4->setValue(blink);

ui->spinBox\_4->setValue(blink);

connect(ui->horizontalSlider,SIGNAL(sliderMoved(int)),ui->spinBox,SLOT(setValue(int)));

connect(ui->spinBox,SIGNAL(valueChanged(int)),ui->horizontalSlider,SLOT(setValue(int)));

connect(ui->horizontalSlider\_2,SIGNAL(sliderMoved(int)),ui->spinBox\_2,SLOT(setValue(int)));

connect(ui->spinBox\_2,SIGNAL(valueChanged(int)),ui->horizontalSlider\_2,SLOT(setValue(int)));

connect(ui->horizontalSlider\_3,SIGNAL(sliderMoved(int)),ui->spinBox\_3,SLOT(setValue(int)));

connect(ui->spinBox\_3,SIGNAL(valueChanged(int)),ui->horizontalSlider\_3,SLOT(setValue(int)));

connect(ui->horizontalSlider\_4,SIGNAL(sliderMoved(int)),ui->spinBox\_4,SLOT(setValue(int)));

connect(ui->spinBox\_4,SIGNAL(valueChanged(int)),ui->horizontalSlider\_4,SLOT(setValue(int)));

sett.close();

}

optionDialog::~*optionDialog*()

{

delete ui;

}

void optionDialog::on\_buttonBox\_accepted()

{

std::ofstream sett("setting.txt");

sett<<ui->horizontalSlider->value()<<" "<<ui->horizontalSlider\_2->value()<<" "<<ui->horizontalSlider\_3->value()<<" "<<ui->horizontalSlider\_4->value();

sett.close();

changesAccepted();

}

void optionDialog::on\_pushButton\_clicked()

{

std::ofstream sett("setting.txt");

sett<<0<<" "<<45<<" "<<0<<" "<<250;

sett.close();

changesAccepted();

this->close();

}