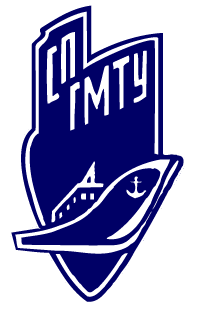
**МИНИСТЕРСВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

“САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ МОРСКОЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ”

ФАКУЛЬТЕТ ЦИФРОВЫХ ПРОМЫШЛЕННЫХ ТЕХНОЛОГИЙ

КАФЕДРА КИБЕРФИЗИЧЕСКИХ СИСТЕМ

КУРСОВОЙ ПРОЕКТ

ПО ДИСЦИПЛИНЕ “ПРОГРАММИРОВАНИЕ”

ТЕМА: “АНАЛИЗ ХОДОВ ФИГУРЫ НА ШАХМАТНОЙ ДОСКЕ”

Выполнил работу студент 20121 группы:

Крылов Илья Сергеевич

Санкт-Петербург 2024

# 

# Оглавление

[Оглавление 2](#_Toc1865139528)

[Цели и формулировка задания 2](#_Toc1721804767)

[Цель 3](#_Toc1474327355)

[Формулировка задания 3](#_Toc337675396)

[Разработка UML-схемы 6](#_Toc262580431)

[Демонстрация работы программы 9](#_Toc1409199131)

[Листинг кода 12](#_Toc546307662)

[Заключение 19](#_Toc379519500)

# Цели и формулировка задания

## Цель:

Анализ ходов фигуры на шахматной доске

При работе над данным лабораторным проектом будут затронуты следующие вопросы:

1. Работа с основами функционального программирования языка Python
2. Работы с основами ООП языка Python
3. Разработка классов и UML диаграмма
4. Работы с пакетами Python
5. Создание GUI приложения с использованием tkinter (или customtkinter) + pygame (+OpenGl).

## Формулировка задания

Дана квадратная шахматная доска размером N x N. На доске уже размещено K фигур. Фигуры размещены так, что находятся не под боем друг друга. Необходимо расставить на доске еще L фигур так, чтобы никакая из фигур на доске не находилась под боем любой другой фигуры. Требуется найти одно решение для визуализации и все возможные решения для вывода в файл. Если решение не найдено, то необходимо вывести соответствующее сообщение.

1) Необходимо создать UML диаграмму взаимодействия классов. При необходимости можно делать несколько UML диаграмм. Для диаграммы создается табличка взаимодействия (Имя класса: Имя базового класса: Описание). Также для каждого класса создается таблица методов и атрибутов (Методы и атрибуты: Описание). При описании методов обязательно прописывать тип параметров, а также выходной тип данных. При описании атрибутов необходимо прописывать его тип.

2) Необходимо составить программу с использованием функционального программирования языка Python, где:

1. Входные данные в файле input.txt. На первой строке файла записаны три числа: N L K (через пробел). Далее следует K строк, содержащих числа x и y (через пробел) - координаты уже стоящей на доске фигуры (фигуры стоят правильно). Координаты отсчитываются от 0 до N-1. 1 <= N <= 20.

2. Выходные данные в файл output.txt. На каждое найденное решение необходимо записать в файл одну строку. Строка состоит из пар (x,y) - координаты фигур на доске. В решение следует вывести координаты всех фигур, находящихся на доске. Каждую фигуру необходимо записать в виде пары координат, разделенных запятой и обрамленных скобками. Координаты отсчитываются от 0 до N- 1.

3) Необходимо составить программу с использованием ООП языка Python, а также разработать интерфейса при помощи пакетов tkinter (или customtkinter) + pygame (+OpenGl).

1. tkinter (или customtkinter) + pygame(+OpenGl)).
   1. tkinter (или customtkinter)
      1. На интерфейсе есть два поля ввода и кнопка. В первом поле вводится размер доски N, во втором – количество фигур, которые необходимо расставить с помощью алгоритма L, и с помощью кнопки создается новое окно. Необходимо проверять правильность ввода данных.
   2. pygame (+OpenGl)
      1. В данном интерфейсе есть шахматная доска и кнопка. На созданной шахматной доске с помощью ПКМ/ЛКМ необходимо расставить/убрать стоящие на доске K фигур. Необходимо визуализировать саму фигуру и ее ходы (визуализацию фигуры и ее ходов выбираете сами, но так, чтобы они отличались). Также необходимо проверить, чтобы поставленная вами фигура не находилась под боем. С помощью кнопки создается новое окно.
      2. В данном интерфейсе есть шахматная доска и кнопка. На созданной шахматной доске показываются фигуры, расставленные пользователем и найденные алгоритмом, (нужно показать любое найденное решение). Визуализация пользовательских и найденных алгоритмом фигур должна отличаться (визуализация ходов одинаковая). Если решение не было найдено, то вывести сообщение об этом и закрыть данный интерфейс. С помощью кнопки происходит вывод данных в файл output.txt. На каждое найденное решение необходимо записать в файл одну строку. Строка состоит из пар (x,y) - координаты фигур на доске. В решение следует вывести координаты всех фигур, находящихся на доске. Каждую фигуру необходимо записать в виде пары координат, разделенных запятой и обрамленных скобками. Координаты отсчитываются от 0 до N. Порядок, в котором фигуры перечислены в решении, не имеет значения.

Дополнительная информация:

1. Обязательное использование аннотации и комментариев
2. Проверка ввода данных.

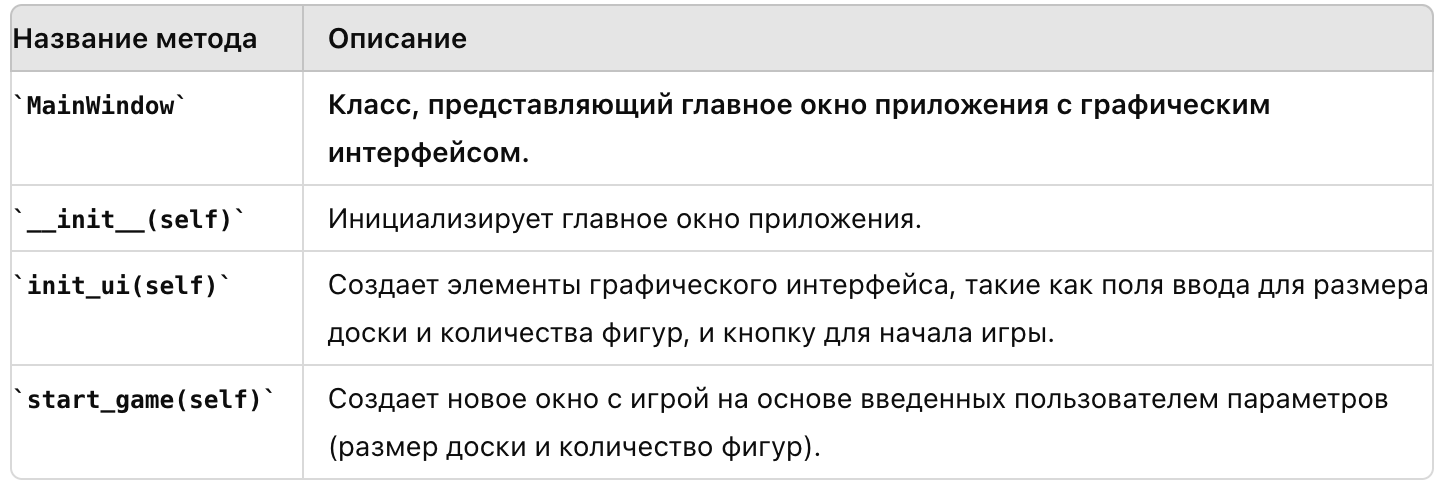
# Разработка UML-схемы

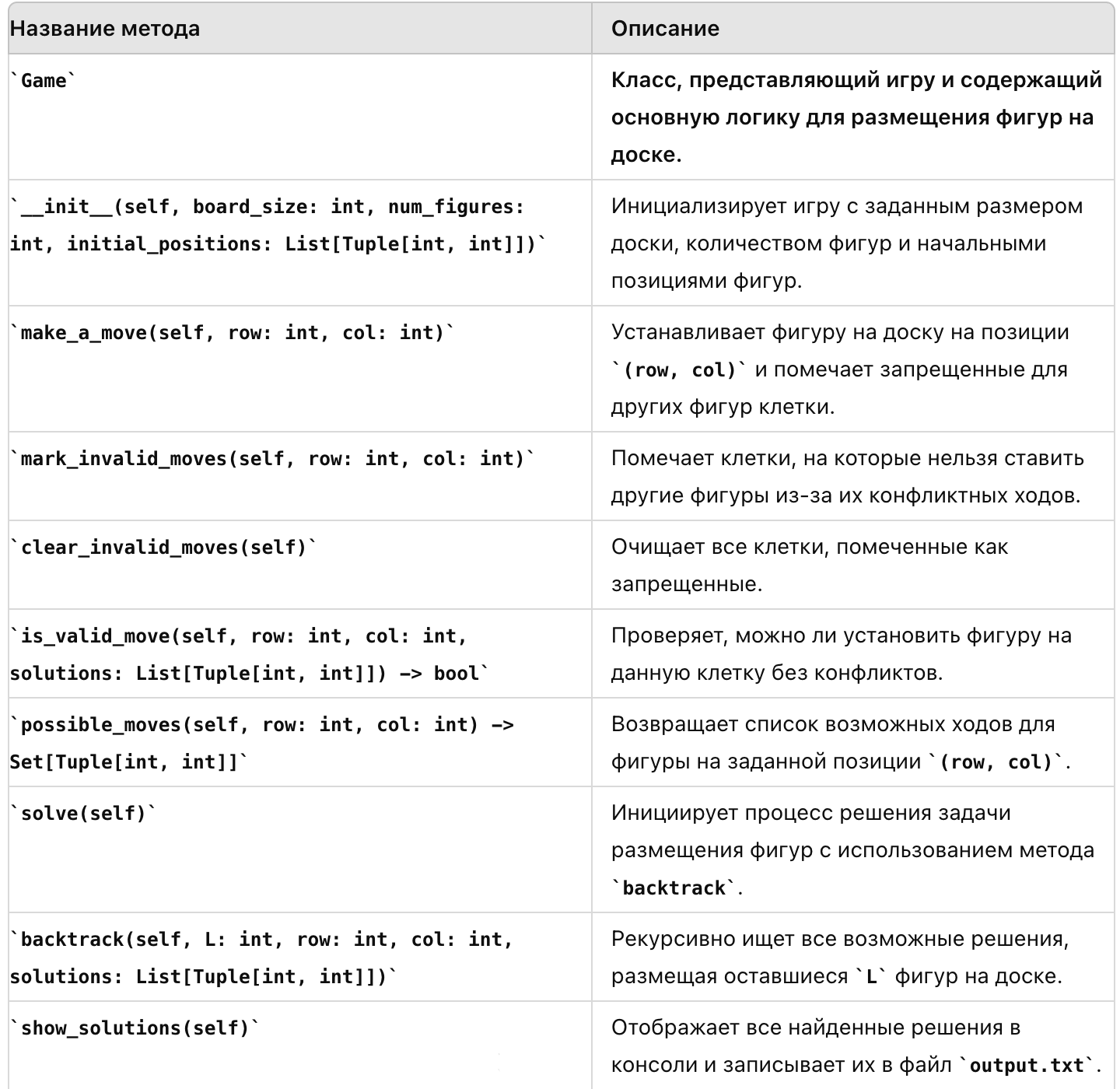
В UML-схеме указаны взаимосвязи между классами, а также основные атрибуты и методы классов.

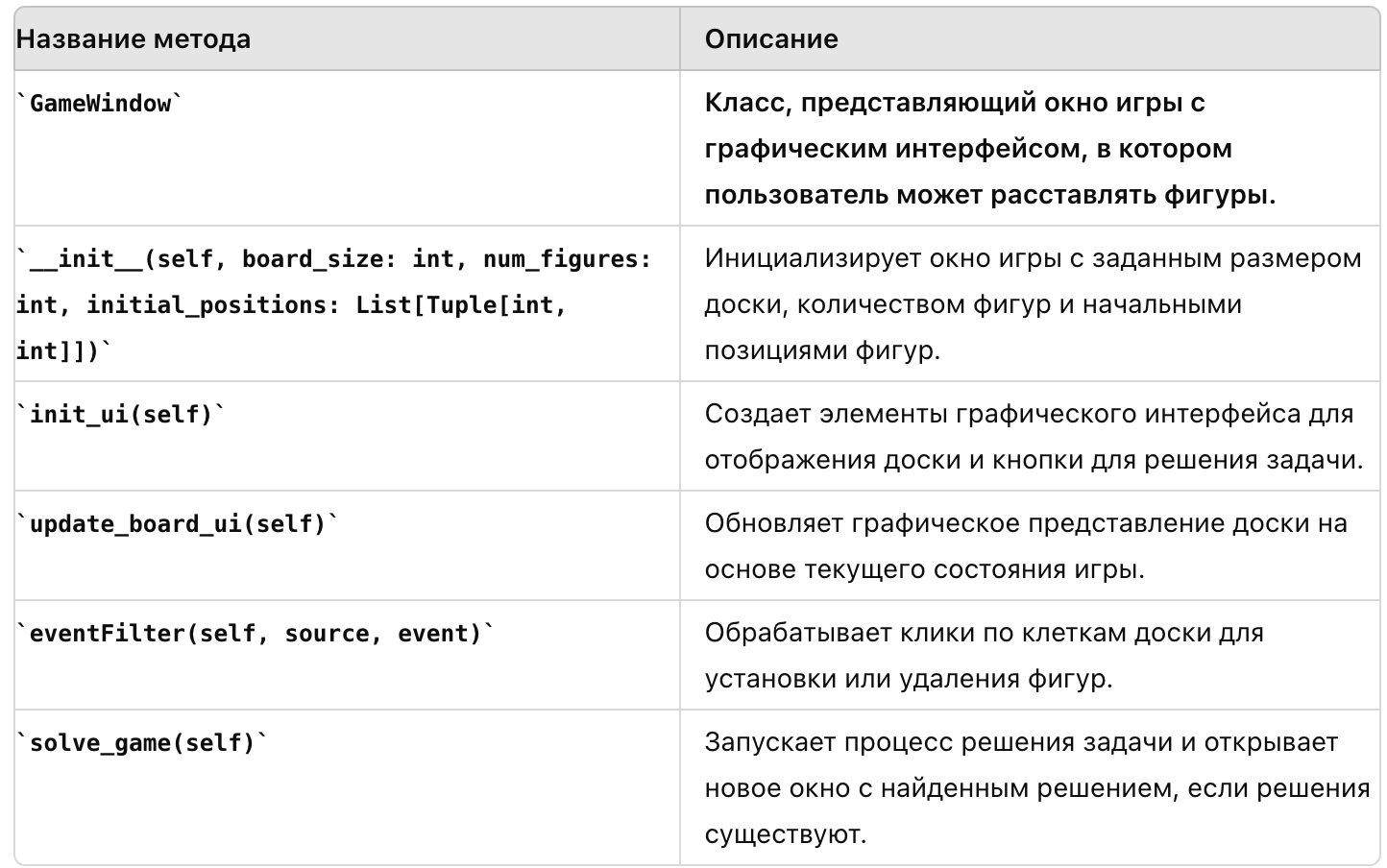


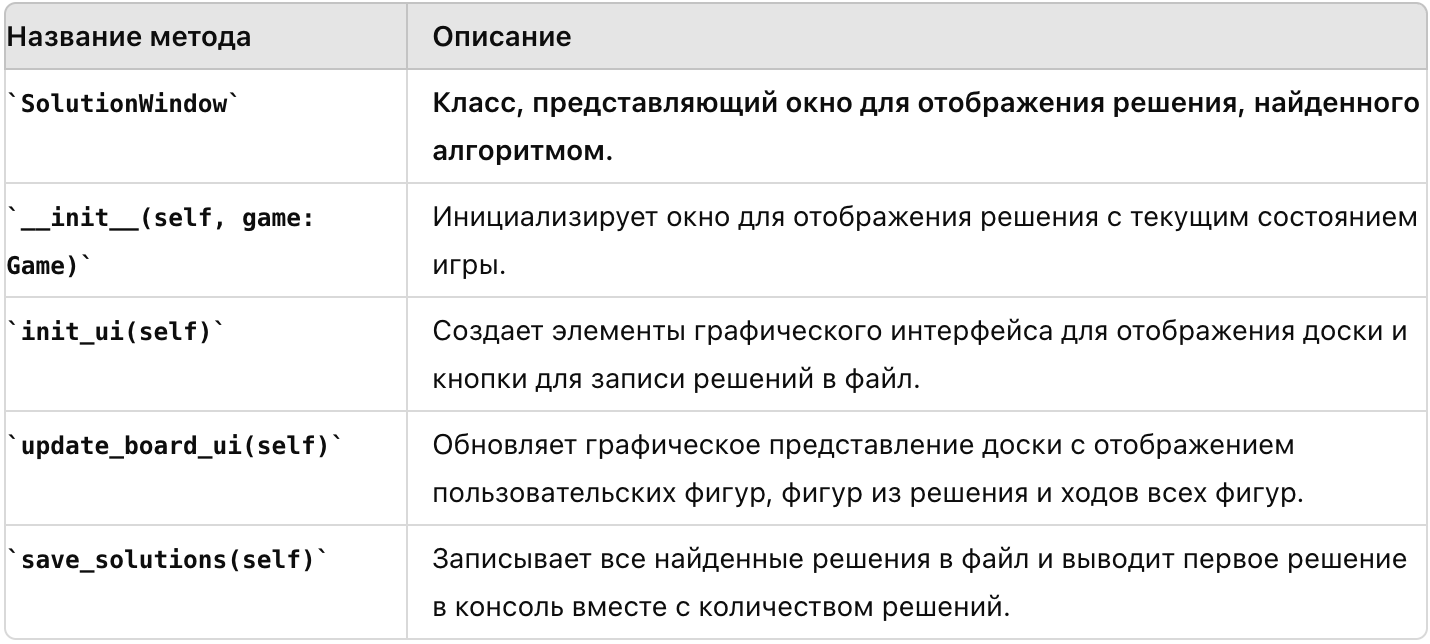
Более подробное описание программы (методов классов) представлено ниже:





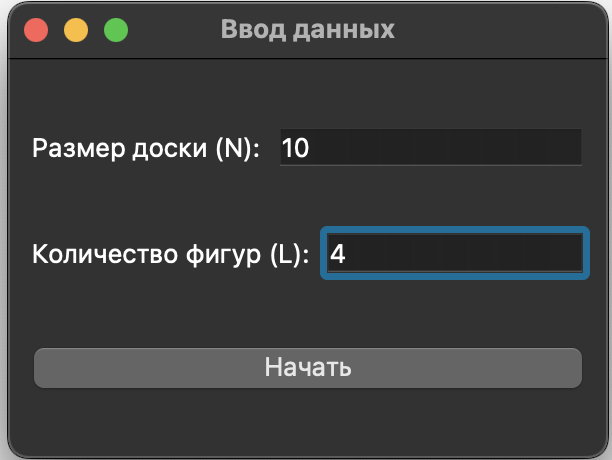






# Демонстрация работы программы

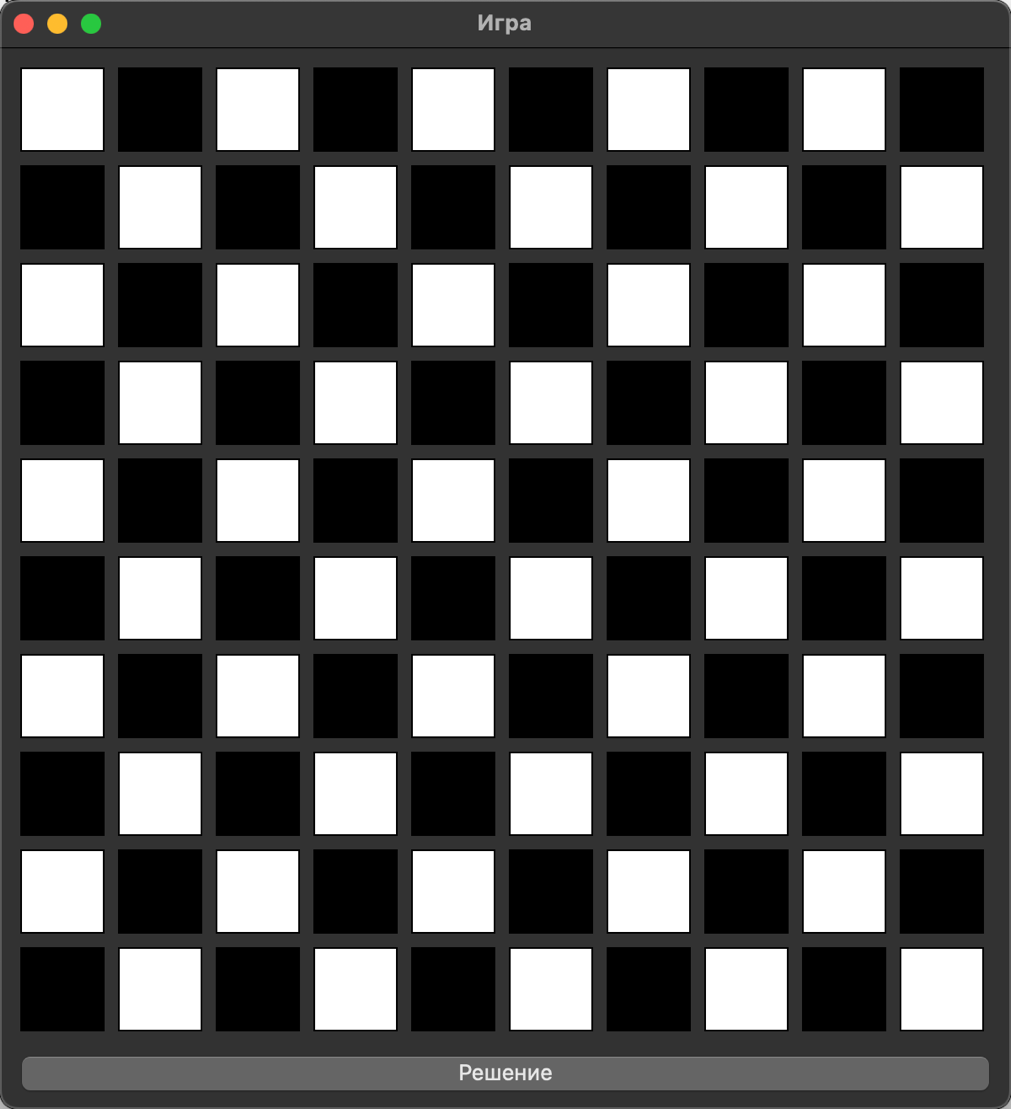
Окно ввода начальных данных: размера доски N и количество фигур L, которое необходимо расставить:



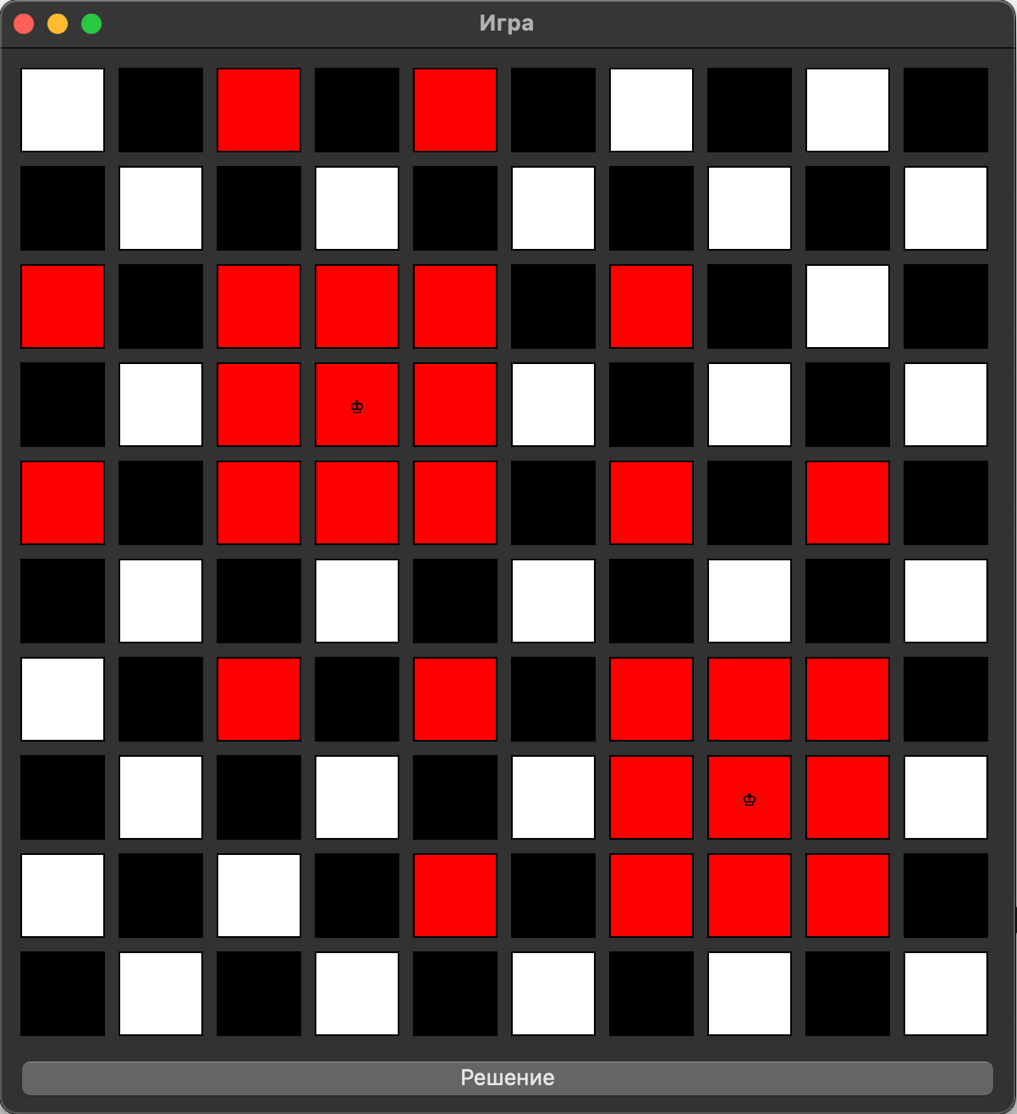
При неправильном вводе может выводиться ошибка:



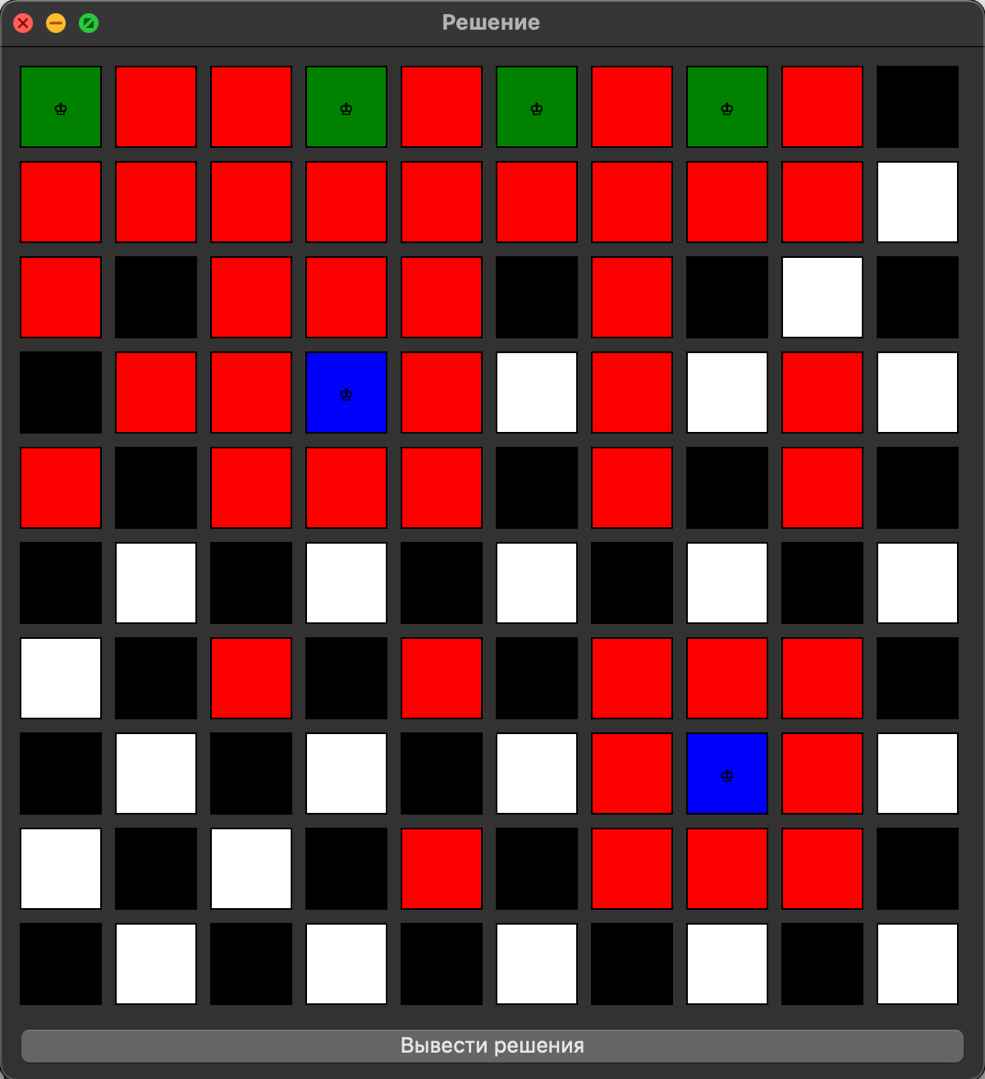
Окно расстановки фигур:



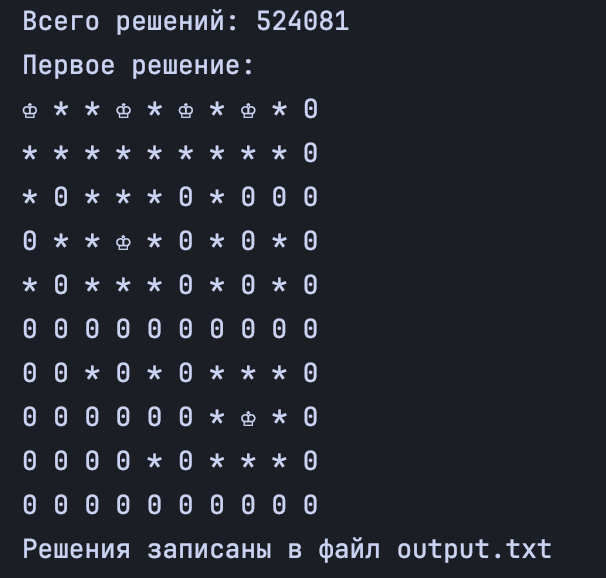
То же окно с расставленными фигурами пользователя:



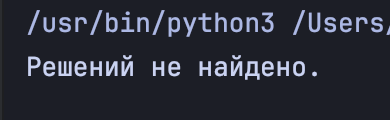
Окно вывода одного из решений:



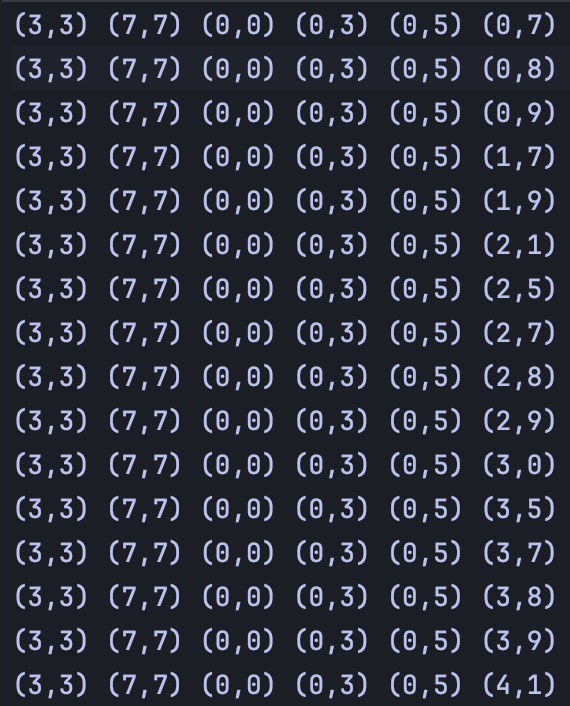
При нажатии на кнопку «Вывести решения», все решение записываются в файл «output.txt» и вместе с первым решением в консоль также выводится соответствующее уведомление:



В случае, когда решений нет, окно с доской закрывается и в консоль выводиться соответствующее уведомление:



Сохранённые данные в файл:



# Листинг кода

```  
*import* time  
*from* typing *import* List, Tuple, Set  
*from* PySide6.QtWidgets *import* (  
 QApplication, QMainWindow, QWidget, QVBoxLayout, QLabel, QPushButton, QGridLayout, QLineEdit, QHBoxLayout  
)  
*from* PySide6.QtCore *import* Qt  
*from* PySide6.QtGui *import* QMouseEvent  
  
  
*class* Board:  
  
 # Класс для представления шахматной доски  
  
 *def \_\_init\_\_*(*self*, size: *int*): # Инициализация доски заданного размера  
 *self*.size = size  
 *self*.grid = [['0'] \* size *for* \_ *in range*(size)]  
  
 *def* update\_board(*self*, row: *int*, col: *int*, char: *str*): # Обновление клетки доски символом char  
 *self*.grid[row][col] = char  
  
 *def* get\_board(*self*) -> List[List[str]]: # Получение текущего состояния доски  
 *return self*.grid  
  
 *def* display(*self*): # Отображение доски в консоли  
 *for* row *in self*.grid:  
 *print*(" ".join(row))  
  
  
*class* Game:  
  
 # Класс для представления игры и её логики  
  
 *def \_\_init\_\_*(*self*, board\_size: *int*, num\_figures: *int*, initial\_positions: List[Tuple[int, int]]): # Инициализация игры  
 *self*.board = Board(board\_size)  
 *self*.num\_figures = num\_figures  
 *self*.initial\_positions = initial\_positions  
 *self*.user\_positions = []  
 *self*.all\_solutions = []  
  
 *for* row, col *in* initial\_positions:  
 *self*.make\_a\_move(row, col)  
  
 *def* make\_a\_move(*self*, row: *int*, col: *int*): # Установка фигуры на доску и пометка запрещенных клеток  
 *self*.board.update\_board(row, col, '♔')  
 *self*.mark\_invalid\_moves(row, col)  
 *self  
  
 def* mark\_invalid\_moves(*self*, row: *int*, col: *int*): # Пометка клеток, на которые нельзя ставить фигуры  
 moves = *self*.possible\_moves(row, col)  
 *for* r, c *in* moves:  
 *if* 0 <= r < *self*.board.size *and* 0 <= c < *self*.board.size *and self*.board.get\_board()[r][c] == '0':  
 *self*.board.update\_board(r, c, '\*')  
  
 *def* clear\_invalid\_moves(*self*): # Очистка всех помеченных клеток  
 *for* i *in range*(*self*.board.size):  
 *for* j *in range*(*self*.board.size):  
 *if self*.board.get\_board()[i][j] == '\*':  
 *self*.board.update\_board(i, j, '0')  
  
 *def* is\_valid\_move(*self*, row: *int*, col: *int*, solutions: List[Tuple[int, int]]) -> *bool*:  
 # Проверка, можно ли установить фигуру на данную клетку  
 *if self*.board.get\_board()[row][col] != '0':  
 *return False* moves = *self*.possible\_moves(row, col)  
 *for* r, c *in* moves:  
 *if* 0 <= r < *self*.board.size *and* 0 <= c < *self*.board.size:  
 *if* (r, c) *in* solutions *or self*.board.get\_board()[r][c] == '♔':  
 *return False  
 return True  
  
 def* possible\_moves(*self*, row: *int*, col: *int*) -> Set[Tuple[int, int]]: # Получение списка возможных ходов для фигуры  
 moves = {  
 (row + 1, col + 1), (row - 1, col - 1),  
 (row + 1, col - 1), (row - 1, col + 1),  
 (row - 1, col), (row + 1, col),  
 (row, col - 1), (row, col + 1),  
 (row + 3, col + 1), (row + 3, col - 1),  
 (row - 3, col + 1), (row - 3, col - 1),  
 (row + 1, col + 3), (row + 1, col - 3),  
 (row - 1, col + 3), (row - 1, col - 3)  
 }  
 *return* moves  
  
 *def* solve(*self*): # Решение задачи размещения фигур  
 *self*.backtrack(*self*.num\_figures, 0, -1, *self*.initial\_positions + *self*.user\_positions)  
  
 *def* backtrack(*self*, L: *int*, row: *int*, col: *int*, solutions: List[Tuple[int, int]]):  
 # Рекурсивный метод для нахождения всех возможных решений  
 *if* L == 0:  
 *self*.all\_solutions.append(solutions.copy())  
 *return  
  
 for* r *in range*(row, *self*.board.size):  
 start\_col = col + 1 *if* r == row *else* 0  
 *for* c *in range*(start\_col, *self*.board.size):  
 *if* (r, c) *not in* solutions *and self*.is\_valid\_move(r, c, solutions):  
 solutions.append((r, c))  
 *self*.backtrack(L - 1, r, c, solutions)  
 solutions.pop()  
  
 *def* show\_solutions(*self*): # Отображение решений в консоли и запись их в файл  
 *if not self*.all\_solutions:  
 *print*("No solutions found.")  
 *return  
  
 print*("Всего решений:", *len*(*self*.all\_solutions))  
 *print*("Первое решение:")  
 first\_solution = *self*.all\_solutions[0]  
 *for* row, col *in* first\_solution:  
 *self*.board.update\_board(row, col, '♔')  
 *self*.mark\_invalid\_moves(row, col)  
 *self*.board.display()  
  
 # Записываем все решения в файл  
 *with open*("output.txt", "w") *as* output\_file:  
 *for* solution *in self*.all\_solutions:  
 solution\_str = " ".join([f"({row},{col})" *for* row, col *in* solution])  
 output\_file.write(solution\_str + "\n")  
  
 *def* create\_solution\_board(*self*, solution: List[Tuple[int, int]]) -> List[List[str]]:  
  
 # Создание доски с указанным решением  
  
 board = Board(*self*.board.size)  
 *for* row, col *in self*.initial\_positions:  
 board.update\_board(row, col, '♔')  
 *self*.mark\_invalid\_moves(row, col)  
  
 *for* row, col *in* solution:  
 board.update\_board(row, col, '♔')  
 *self*.mark\_invalid\_moves(row, col)  
 *return* board.get\_board()  
  
  
*class* MainWindow(QMainWindow):  
  
 # Класс главного окна приложения с графическим интерфейсом  
  
 *def \_\_init\_\_*(*self*): # Инициализация главного окна  
 *super*().\_\_init\_\_()  
 *self*.setWindowTitle("Ввод данных")  
 *self*.setGeometry(100, 100, 300, 200)  
  
 *self*.init\_ui()  
  
 *def* init\_ui(*self*): # Инициализация графического интерфейса  
 central\_widget = QWidget()  
 *self*.setCentralWidget(central\_widget)  
  
 layout = QVBoxLayout()  
  
 size\_layout = QHBoxLayout()  
 size\_label = QLabel("Размер доски (N):")  
 *self*.size\_input = QLineEdit()  
 size\_layout.addWidget(size\_label)  
 size\_layout.addWidget(*self*.size\_input)  
  
 figures\_layout = QHBoxLayout()  
 figures\_label = QLabel("Количество фигур (L):")  
 *self*.figures\_input = QLineEdit()  
 figures\_layout.addWidget(figures\_label)  
 figures\_layout.addWidget(*self*.figures\_input)  
  
 start\_button = QPushButton("Начать")  
 start\_button.clicked.connect(*self*.start\_game)  
  
 layout.addLayout(size\_layout)  
 layout.addLayout(figures\_layout)  
 layout.addWidget(start\_button)  
  
 central\_widget.setLayout(layout)  
  
 *def* start\_game(*self*): # Обработка нажатия кнопки "Начать" для создания нового окна с игрой  
 *try*:  
 board\_size = *int*(*self*.size\_input.text())  
 num\_figures = *int*(*self*.figures\_input.text())  
 *except ValueError*:  
 *print*("Пожалуйста, введите корректные числовые значения.")  
 *return* initial\_positions = [] # Начальные позиции фигур могут быть пустыми или заданы пользователем  
  
 *self*.game\_window = GameWindow(board\_size, num\_figures, initial\_positions)  
 *self*.game\_window.show()  
  
  
*class* GameWindow(QMainWindow):  
  
 # Класс окна игры с графическим интерфейсом  
  
 *def \_\_init\_\_*(*self*, board\_size: *int*, num\_figures: *int*, initial\_positions: List[Tuple[int, int]]): # Инициализация окна игры  
 *super*().\_\_init\_\_()  
 *self*.game = Game(board\_size, num\_figures, initial\_positions)  
 *self*.setWindowTitle("Игра")  
 *self*.setGeometry(100, 100, 600, 600)  
  
 *self*.init\_ui()  
  
 *def* init\_ui(*self*): # Инициализация графического интерфейса  
 central\_widget = QWidget()  
 *self*.setCentralWidget(central\_widget)  
  
 layout = QVBoxLayout()  
  
 *self*.board\_layout = QGridLayout()  
  
 *self*.update\_board\_ui()  
  
 solve\_button = QPushButton("Решение")  
 solve\_button.clicked.connect(*self*.solve\_game)  
  
 layout.addLayout(*self*.board\_layout)  
 layout.addWidget(solve\_button)  
  
 central\_widget.setLayout(layout)  
  
 *def* update\_board\_ui(*self*): # Обновление графического интерфейса доски  
 *for* i *in range*(*self*.board\_layout.count()):  
 *self*.board\_layout.itemAt(i).widget().deleteLater()  
  
 board = *self*.game.board.get\_board()  
 *for* i *in range*(*self*.game.board.size):  
 *for* j *in range*(*self*.game.board.size):  
 label = QLabel()  
 label.setAlignment(Qt.AlignCenter)  
 label.setFixedSize(50, 50)  
 *if* (i, j) *in self*.game.user\_positions:  
 label.setText('♔')  
 label.setStyleSheet("background-color: red; color: black; border: 1px solid black;")  
  
 *elif* (i, j) *in self*.game.initial\_positions:  
 label.setText('♔')  
 label.setStyleSheet("background-color: green; color: black; border: 1px solid black;")  
  
 *elif* board[i][j] == '\*':  
 label.setStyleSheet("background-color: red; border: 1px solid black;")  
  
 *else*:  
 color = 'white' *if* (i + j) % 2 == 0 *else* 'black'  
 label.setStyleSheet(f"background-color: {color}; border: 1px solid black;")  
  
 label.installEventFilter(*self*)  
 label.row = i  
 label.col = j  
 *self*.board\_layout.addWidget(label, i, j)  
  
 *def* eventFilter(*self*, source, event): # Обработка кликов по клеткам доски  
 *if* event.type() == QMouseEvent.MouseButtonPress:  
 row, col = source.row, source.col  
  
 *if* event.button() == Qt.LeftButton:  
 *if self*.game.board.get\_board()[row][col] == '0':  
 *self*.game.make\_a\_move(row, col)  
 *self*.game.user\_positions.append((row, col))  
 *elif self*.game.board.get\_board()[row][col] == '♔' *and* (row, col) *in self*.game.user\_positions:  
 *self*.game.board.update\_board(row, col, '0')  
 *self*.game.user\_positions.remove((row, col))  
 *self*.game = Game(*self*.game.board.size, *self*.game.num\_figures, *self*.game.initial\_positions)  
 *for* pos *in self*.game.user\_positions:  
 *self*.game.make\_a\_move(\*pos)  
 *self*.update\_board\_ui()  
  
 *elif* event.button() == Qt.RightButton:  
 *if self*.game.board.get\_board()[row][col] == '♔' *and* (row, col) *in self*.game.user\_positions:  
 *self*.game.board.update\_board(row, col, '0')  
 *self*.game.user\_positions.remove((row, col))  
 *self*.game = Game(*self*.game.board.size, *self*.game.num\_figures, *self*.game.initial\_positions)  
 *for* pos *in self*.game.user\_positions:  
 *self*.game.make\_a\_move(\*pos)  
 *self*.update\_board\_ui()  
 *return super*().eventFilter(source, event)  
  
 *def* solve\_game(*self*): # Обработка нажатия кнопки "Решение" для решения задачи  
 *self*.game.solve()  
 *if not self*.game.all\_solutions:  
 *print*("Решений не найдено.")  
 *self*.close()  
 *return* # Создаем новое окно для отображения первого решения  
 *self*.solution\_window = SolutionWindow(*self*.game)  
 *self*.solution\_window.show()  
  
  
*class* SolutionWindow(QMainWindow):  
  
 # Класс окна для отображения решения  
  
 *def \_\_init\_\_*(*self*, game: Game): # Инициализация окна с решением  
 *super*().*\_\_init\_\_*()  
 *self*.setWindowTitle("Решение")  
 *self*.setGeometry(100, 100, 600, 600)  
  
 *self*.game = game  
 *self*.solution = *self*.game.all\_solutions[0]  
  
 *self*.init\_ui()  
  
 *def* init\_ui(*self*): # Инициализация графического интерфейса  
 central\_widget = QWidget()  
 *self*.setCentralWidget(central\_widget)  
  
 layout = QVBoxLayout()  
  
 *self*.board\_layout = QGridLayout()  
  
 *self*.update\_board\_ui()  
  
 save\_button = QPushButton("Вывести решения")  
 save\_button.clicked.connect(*self*.save\_solutions)  
  
 layout.addLayout(*self*.board\_layout)  
 layout.addWidget(save\_button)  
  
 central\_widget.setLayout(layout)  
  
 *def* update\_board\_ui(*self*): # Обновление графического интерфейса доски  
 *for* i *in range*(*self*.board\_layout.count()):  
 *self*.board\_layout.itemAt(i).widget().deleteLater()  
  
 board = *self*.game.create\_solution\_board(*self*.solution)  
 possible\_moves = *set*()  
  
 # Собираем все возможные ходы фигур  
 *for* row, col *in self*.solution:  
 moves = *self*.game.possible\_moves(row, col)  
 possible\_moves.update(moves)  
  
 *for* i *in range*(*self*.game.board.size):  
 *for* j *in range*(*self*.game.board.size):  
 label = QLabel()  
 label.setAlignment(Qt.AlignCenter)  
 label.setFixedSize(50, 50)  
  
 *if* (i, j) *in self*.game.user\_positions:  
 label.setText('♔')  
 label.setStyleSheet("background-color: blue; color: black; border: 1px solid black;")  
  
 *elif* (i, j) *in self*.solution:  
 label.setText('♔')  
 label.setStyleSheet("background-color: green; color: black; border: 1px solid black;")  
  
 *elif* (i, j) *in* possible\_moves:  
 label.setStyleSheet("background-color: red; border: 1px solid black;")  
  
 *else*:  
 color = 'white' *if* (i + j) % 2 == 0 *else* 'black'  
 label.setStyleSheet(f"background-color: {color}; border: 1px solid black;")  
  
 *self*.board\_layout.addWidget(label, i, j)  
  
 *def* save\_solutions(*self*): # Обработка нажатия кнопки "Вывести решения" для записи решений в файл  
 *self*.game.show\_solutions()  
 *print*("Решения записаны в файл output.txt")  
  
  
*def* main(): # Основная функция программы  
 app = QApplication([])  
  
 main\_window = MainWindow()  
 main\_window.show()  
  
 app.exec()  
  
  
*if* \_\_name\_\_ == "\_\_main\_\_":  
 main()

# Заключение

Мною было проведена работа по созданию GUI (Graphical User Interface), для алгоритма расстановки фигур шахмат, с возможностью последующей работы с ней.

Этот код представляет собой комплексное решение для задачи размещения фигур на шахматной доске с использованием рекурсивного алгоритма поиска всех возможных решений. Он включает в себя графический интерфейс для взаимодействия с пользователем, где пользователь может вводить параметры задачи, расставлять фигуры на доске, и видеть решения, найденные алгоритмом.

В процессе работы я ознакомился с модулями языка Python: PySide6. Мною была создана UML-схема, упорядочивающая программу.