МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования «Санкт-Петербургский государственный морской технический университет»  
 (СПБГМТУ)

|  |
| --- |
| ФАКУЛЬТЕТ ЦИФРОВЫХ ПРОМЫШЛЕННЫХ ТЕХНОЛОГИЙ  КАФЕДРА КИБЕРФИЗИЧЕСКИХ СИСТЕМ |

КУРСОВОЙ ПРОЕКТ

по дисциплине “программирование”

|  |
| --- |
|  |

|  |
| --- |
| Выполнил студент группы 20121 |
| Кузьмин Кирилл Алексеевич |
| Проверил |
| Поделенюк Павел Петрович |

Санкт-Петербург

2025 год.

содержание

[Цели 3](#_Toc138023417)

[Формулировка задачи 4](#_Toc138023418)

[Разработка классов и UML диаграмма 8](#_Toc138023419)

[Результаты работы 10](#_Toc138023420)

[Листинг кода 1](#_Toc138023421)8

[Заключение](#_Toc138023422) 35

# Цели

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 PySide6

# Формулировка задачи

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:

• 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.

• 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-

• Порядок, в котором фигуры перечислены в решении, не имеет значения. Если не было найдено ни одного решения, в файл необходимо записать no solutions.

• Выходные данные на консоль — это доска N\*N, где фигура обозначается #, ее ходы обозначаются \*, а пустые клетки обозначаются 0.

3) Необходимо составить программу с использованием объектноориентированного программирования (ООП) на языке Python. Каждый класс должен чётко обозначать своё предназначение. Также необходимо разработать интерфейс с использованием пакета PySide6.

На интерфейсе есть три поля ввода и три кнопки.

• В первом поле вводится размер доски N.

• Во втором поле — количество фигур, которые необходимо расставить с помощью алгоритма L.

• В третьем поле — количество фигур, расставленных пользователем, K. Кнопки и их функции:

• Первая кнопка вызывает новое окно для ввода координат для каждой из K фигур.

• Вторая кнопка вызывает новое окно для отрисовки доски.

• Третья кнопка отвечает за выход из приложения. Условия работы кнопок:

• Если поля пустые, то все кнопки (кроме кнопки выхода) должны быть неактивными.

• Если K > 0 и координаты для фигур не введены, то кнопка, отвечающая за отрисовку доски, должна быть неактивной.

• Если K = 0, то кнопка отрисовки доски может быть активной. В этом случае кнопка ввода координат может либо не открывать окно, но активировать кнопку отрисовки доски, либо остаться без действия, либо скрыта и т.д. (на усмотрение разработчика).

2 В окне ввода координат должно быть K полей ввода и две кнопки.

• В каждое поле необходимо ввести числа x и y (через пробел), представляющие координаты фигуры, уже размещённой на доске (или другой любой способ).

• Если указанные координаты фигур попадают под атаку друг друга, это должно быть отображено (способ отображения на усмотрение разработчика). Кнопки и их функции:

• Кнопка подтверждения фиксирует введённые координаты.

• Кнопка выхода закрывает окно ввода координат без сохранения изменений. Условия работы кнопки подтверждения:

• Если хотя бы одно поле пустое, кнопка подтверждения должна быть неактивной.

• Если координаты какой-либо фигуры попадают под атаку или атакуют координаты другой фигуры, кнопка подтверждения также должна быть неактивной. В окне вывода доски отображаются фигуры, расставленные пользователем, и фигуры, найденные алгоритмом (необходимо показать любое найденное решение).

Визуализация:

• Фигуры, расставленные пользователем, и фигуры, найденные алгоритмом, должны иметь различную визуализацию (например, разный цвет, стиль или значок).

• Визуализация ходов (если применимо) может быть одинаковой для всех фигур или отличаться на усмотрение разработчика. Отсутствие решения: • Если алгоритм не нашёл решения, окно вывода доски не открывается, а пользователю отображается сообщение об отсутствии решения. Функционал окна:

• С помощью кнопки данные записываются в файл. Для каждого найденного решения в файл записывается одна строка, содержащая координаты всех фигур на доске (как пользовательских, так и найденных алгоритмом). Каждая фигура представлена парой координат (x,y), где x и y — числа от 0 до N-1. Пары координат разделены запятыми и заключены в круглые скобки. Порядок фигур в строке не имеет значения.

• Окно содержит кнопку для закрытия окна вывода доски.

Дополнительная информация:

1. Обязательное использование аннотации и комментариев

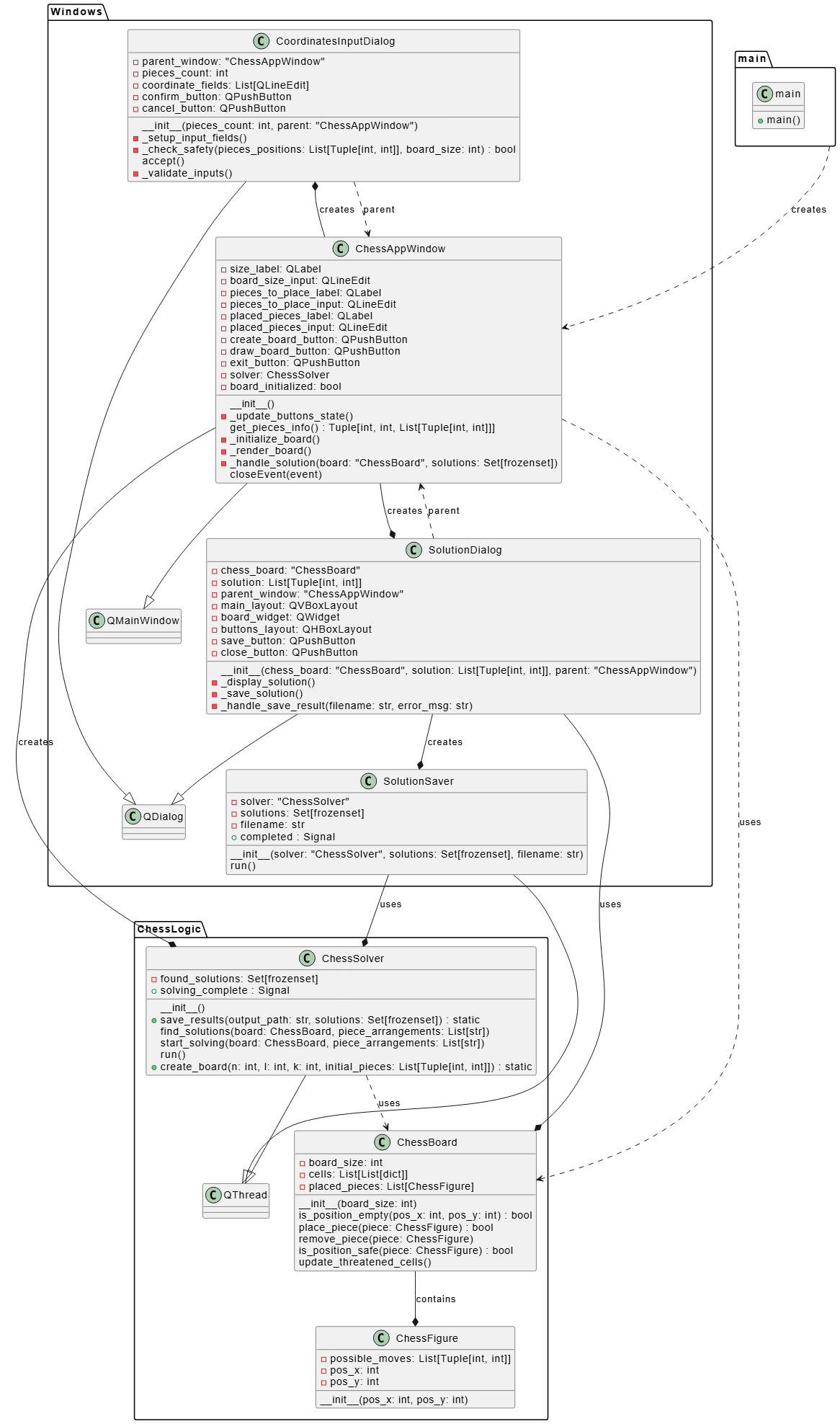
2. Проверка ввода данных (особенно для первого способа, где вводим координаты вручную).

3. Новые открывающиеся окна делайте модальными, чтобы главное окно было не активное.

4. Чтобы приложение оставалось отзывчивым во время вывода данных в файл или выполнения алгоритма, рекомендуется использовать многопоточность, например, QRunnable или QtConcurrent из библиотеки PySide6.

5. Классы, отвечающие за реализацию окон интерфейса, и классы, отвечающие за выполнение алгоритма, должны быть разделены и выполнять различные функции.

# Разработка классов и UML диаграмма



CoordinatesInputDialog – Класс, отвечающий за окно для ввода координат фигур (L)

ChessAppWindow – Класс, отвечающий за главное окно

DeskWindow – Класс, отвечающий за окно с отрисованным шахматным полем

SolutionSaver – Класс, отвечающий за сохранение решений в файл

ChessFigure – Класс, отвечающий за фигуру с координатами (X Y)

ChessBoard – Класс, отвечающий за логику отрисовки доски  
ChessSolver – Класс, отвечающий за нахождение всевозможных расстановок фигур

# Результаты работы

1. Реализация программы с использованием функционального программирования языка Python.

## Ход работы

1. Проанализировав задачу, я обнаружил следующие полезные особенности:
   1. Если клетка не находится под боем другой фигуры и не занята ею, то можно поставить на эту клетку фигуру и она не будет под боем.
   2. Все возможные ходы фигуры, поставленной на клетку с координатами x и y можно выразить как:  
      (x + 1, y), (x, y - 1), (x - 1, y), (x, y + 1), (x, y + 2), (x, y - 2), (x - 2, y), (x + 2, y)
2. Создаю функцию figure\_moves – генератор, убирающий фигуры с указанной координаты, выходящие за рамки нашего поля
3. Создаю функцию place\_figure, ставящую фигуру на клетку с указанными координатами.
4. Создаю функцию remove\_figure, убирающую фигуру с клетки с указанными координатами.
5. Обе точки остановки возвращают None, но в первой ситуации мы записываем получившуюся расстановку в файл
6. Переменные side и file делаем глобальными для того, чтобы не добавлять размерность игрового поля в параметры каждой функции, и для того, чтобы записывать данные в файл в рамках рекурсии.
7. Считываю данные из файла, расставляем фигуры из файла на поле
8. Вызываю рекуррентную функцию
9. В рекуррентной функции, реализуется следующий алгоритм:
   1. Ставлю фигуру на поле, перехожу на более низкий уровень рекурсии
   2. Если это была последняя фигура из тех, которые необходимо было поставить, то записываю получившуюся расстановку в файл, вывожу поля с расстановками в консоль (если не закомментировано), и возвращаюсь на предыдущий уровень рекурсии
   3. Если вернулся с более низкого уровня рекурсии, снимаю фигуру и перехожу к следующей свободной клетке
10. Точками остановки в рекурсии будут две ситуации:
    1. Поставлены все «дополнительные» фигуры записываю расстановку в файл
    2. Мы не можем поставить на поле больше ни одной фигуры
11. Выводим no solution в файл если не было решений

## Демонстрация работы программы

## 

## Пример входных данных:

## 2 1 1

## 0 1

## Вывод данных в консоль:

\* #

# \*

**Вывод данных в файл:**

(0, 1), (1, 0)

## Пример входных данных:

2 1 2

0 1

1 0

## Вывод данных в консоль:

## Вывод данных в файл:

no solution

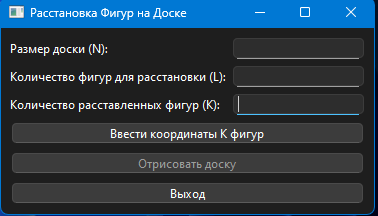
## Листинг кода

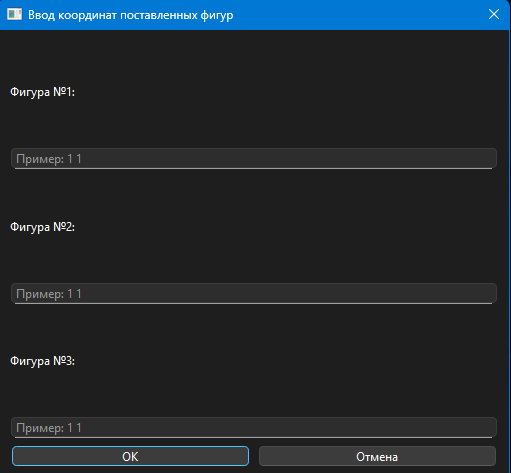
1. from typing import Generator
2. # Функция для вывода нынешнего состояния доски: "#" - фигура, "0" - свободная клетка, '\*' - клетка под боем(число указывает сколько фигур бьют на эту клетку)
3. def print\_field(field: list[list[int]]) -> None:
4. for i in range(side):
5. for j in field[i]:
6. if j==-1:
7. print('#', sep='', end=' ')
8. elif j>0:
9. print('\*', sep='', end=' ')
10. else:
11. print('0', sep='', end=' ')
12. print()
13. print('-' \* (2 \* len(field[0]) - 1))
14. # Генератор, отсекающий ходы фигуры, выходящие за рамки игрового поля
15. def figure\_moves(x: int, y: int) -> Generator[tuple[int, int], None, None]:
16. for i in ((x + 1, y), (x, y - 1), (x - 1, y), (x, y + 1), (x, y + 2), (x, y - 2), (x - 2, y), (x + 2, y)):
17. if not (i[0] > side - 1 or i[1] > side - 1 or i[0] < 0 or i[1] < 0):
18. yield i
19. # Функция, ставящая фигуру на указанную клетку
20. def place\_figure(x: int, y: int, field: list[list[int]])-> list[list[int]]:
21. # Расстановка клеток боя на поле
22. for i in figure\_moves(x, y):
23. field[i[1]][i[0]] += 1
24. # Ставим саму фигуру
25. field[y][x] = -1
26. return field
27. # Функция, убирающая фигуру с указанной клетки
28. def remove\_figure(x: int, y: int, field: list[list[int]]) -> list[list[int]]:
29. # Убираем бой этой фигуры с тех клеток, на которых он был
30. for i in figure\_moves(x, y):
31. field[i[1]][i[0]] -= 1
32. # Убираем саму фигуру
33. field[y][x] = 0
34. return field
35. # Рекуррентная функция, с точкой остановки "Все дополнительные фигуры поставлены"
36. def recursion(place: int, current\_pos: int, field: list[list[int]], figures:list = [])-> None:
37. # Точка остановки, если нам не нужно больше ставить фигуры на поле, мы записываем нынешнюю расстановку в файл и выходим из этой ветки рекурсии
38. if place==0:
39. # Здесь мы выводим удачные расстановки в консоль, по умолчанию закомментировано для большей оптимизации
40. print\_field(field)
41. file.write(",".join(map(str, const\_figures + figures)) + '\n')
42. return
43. # Тело рекурсии
44. else:
45. # Представляем нашу шахматную доску в виде последовательности клеток, избавляясь от двухмерности. Идти по доске мы начинаем с "Нулевой позиции"
46. for pos in range(current\_pos, side \*\* 2):
47. #Если мы находим пустую клетку, то ставим на неё фигуру, записываем её местоположение, меняем значение "Нулевой позиции" на следующую клетку после той, на которую мы поставили фигуру
48. if field[pos // side][pos % side] == 0:
49. field = place\_figure(pos % side, pos // side, field)
50. figures.append((pos % side, pos // side))
51. recursion(place - 1, pos + 1, field, figures)
52. # По достижению точки остановки мы возвращаемся сюда и переходим в другую ветку, снимая поставленную ранее на эту клетку фигуру
53. figures.pop(len(figures) - 1)
54. field = remove\_figure(pos % side, pos // side, field)
56. # Объявление глобальных переменных. 1)side - Сторона поля 2)file - Само шахматное поле 3)const\_figures - Фигуры, которые я не могу двигать
57. global side
58. global file
59. global const\_figures
60. const\_figures: list = list()
62. # Считывание данных из файла
63. with open('input.txt','r') as f:
64. side, place, field\_side = map(int, f.readline().split())
65. # Генерация поля, расстановка фигур из файла
66. field: list[list[int]] = [[0] \* side for i in range(side)]
67. for i in range(field\_side):
68. x, y = map(int, f.readline().split())
69. const\_figures.append((x, y))
70. field: list[list[int]] = place\_figure(x, y, field)
71. #Запуск рекурсии, запись всех возможных расстановок в файл
72. file = open("output.txt", 'w')
73. recursion(place, 0, field)
74. file.close()
75. #Вывод no solutions в файл если нет решений
76. with open("output.txt",'r') as f:
77. file = f.readlines()
78. if len(file) == 0:
79. with open("output.txt", 'w') as f:
80. f.write('no solutions')
81. *Реализация программы с использованием ООП языка Python.*
    1. Ход работы.

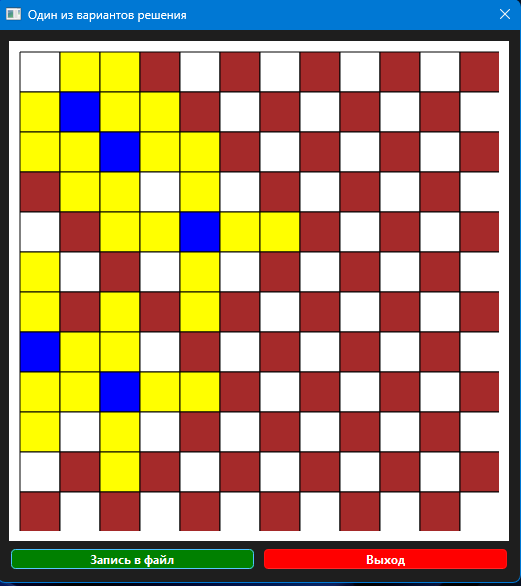
Сначала создаётся объект класса ChessAppWindow (), который при инициализации вызовет окно. В открывшемся окне необходимо ввести размер доски (1-20), количество фигур, расставленных игроком и количество фигур, которые должен расставить алгоритм. После этого, если K>0 нажать на кнопку ‘Ввести координаты K фигур’. Далее создается объект класса ‘CoordinatesInputDialog’ В котором необходимо ввести координаты K фигур. После этого нажимаем кнопку ‘Ок’ и сохраняем их. Дальше создаются экземпляры классов ChessFigure() и ChessBoard(). После нажатия кнопки ‘отрисовать доску’ инициализируется объект класса DeskWindow() и находятся варианты расстановки фигур с помощью методов класса ChessSolver. ChessSolver использует модифицированный алгоритм поиска решений из 1 пункта курсовой работы. В открывшемся окне будет продемонстрирована одна из возможных расстановок фигур. При нажатии кнопки ‘Запись в файл’

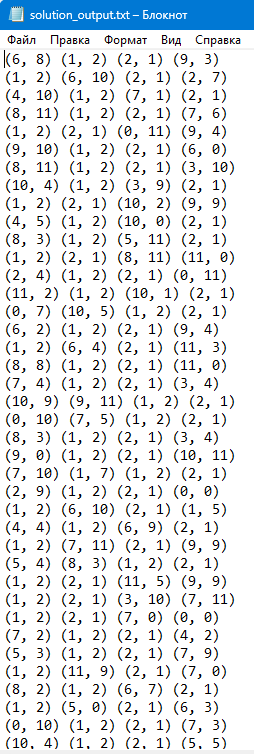
инициализируется объект класса SolutionSaver. Появляется выбор файла .txt. При выборе файла верного формата в этот файл записываются все варианты расстановки.

* 1. Демонстрация работы программы.









*(И так далее)*

# Листинг кода

main.py

from PySide6.QtWidgets import QApplication

from Windows import ChessAppWindow

if \_\_name\_\_ == "\_\_main\_\_":

    chess\_app = QApplication([])

    main\_window = ChessAppWindow()

    main\_window.show()

    chess\_app.exec()

Windows.py

from typing import List, Tuple, Optional, Set

from PySide6.QtCore import QThread, Signal, QRectF

from PySide6.QtGui import QPen, QColor, QPainter

from PySide6.QtCore import Qt

from PySide6.QtWidgets import (

    QMainWindow,

    QDialog,

    QPushButton,

    QLabel,

    QLineEdit,

    QVBoxLayout,

    QHBoxLayout,

    QMessageBox,

    QWidget,

    QGridLayout,

    QGraphicsRectItem,

    QGraphicsScene,

    QGraphicsView,

    QApplication,

    QFileDialog

)

from ChessLogic import ChessFigure, ChessBoard, ChessSolver

import uuid

def create\_unique\_id(input\_data: str) -> str:

    """Создает уникальный идентификатор на основе входных данных."""

    return str(uuid.uuid5(uuid.NAMESPACE\_DNS, input\_data))

def create\_board\_cell(x: int, y: int, cell\_state: int) -> QGraphicsRectItem:

    """

    Создает и возвращает представление клетки шахматной доски.

    Args:

        x: Координата X клетки.

        y: Координата Y клетки.

        value: Значение клетки (0, 1: пустая, 2: фигура, 3: под угрозой).

    Returns:

        Объект QGraphicsRectItem, представляющий клетку.

    """

    CELL\_COLORS = {

        0: QColor("white"),

        1: QColor("brown"),

        2: QColor("blue"),

        3: QColor("yellow")

    }

    cell\_item = QGraphicsRectItem(0, 0, 40, 40)

    cell\_item.setPen(QPen(QColor("black")))

    cell\_item.setPos(x \* 40, y \* 40)

    cell\_item.setBrush(CELL\_COLORS.get(cell\_state, QColor("white")))

    return cell\_item

class ChessAppWindow(QMainWindow):

    """

    Главное окно приложения.

    """

    def \_\_init\_\_(self):

        """

        Инициализация главного окна.

        """

        super().\_\_init\_\_()

        self.setWindowTitle("Расстановка Фигур на Доске")

        self.size\_label = QLabel("Размер доски (N):")

        self.board\_size\_input = QLineEdit()

        self.pieces\_to\_place\_label = QLabel("Количество фигур для расстановки (L):")

        self.pieces\_to\_place\_input = QLineEdit()

        self.placed\_pieces\_label = QLabel("Количество расставленных фигур (K):")

        self.placed\_pieces\_input = QLineEdit()

        self.create\_board\_button = QPushButton("Ввести координаты K фигур")

        self.draw\_board\_button = QPushButton("Отрисовать доску")

        self.exit\_button = QPushButton("Выход")

        self.create\_board\_button.clicked.connect(self.\_initialize\_board)

        self.draw\_board\_button.clicked.connect(self.\_render\_board)

        self.exit\_button.clicked.connect(self.close)

        layout = QGridLayout()

        layout.addWidget(self.size\_label, 0, 0)

        layout.addWidget(self.board\_size\_input, 0, 1)

        layout.addWidget(self.pieces\_to\_place\_label, 1, 0)

        layout.addWidget(self.pieces\_to\_place\_input, 1, 1)

        layout.addWidget(self.placed\_pieces\_label, 2, 0)

        layout.addWidget(self.placed\_pieces\_input, 2, 1)

        layout.addWidget(self.create\_board\_button, 3, 0, 1, 2)

        layout.addWidget(self.draw\_board\_button, 4, 0, 1, 2)

        layout.addWidget(self.exit\_button, 5, 0, 1, 2)

        central\_widget = QWidget()

        central\_widget.setLayout(layout)

        self.setCentralWidget(central\_widget)

        self.draw\_board\_button.setEnabled(False)

        self.solver = ChessSolver()

        self.board\_size\_input.textChanged.connect(self.\_update\_buttons\_state)

        self.pieces\_to\_place\_input.textChanged.connect(self.\_update\_buttons\_state)

        self.placed\_pieces\_input.textChanged.connect(self.\_update\_buttons\_state)

        self.board\_initialized = False

    def \_update\_buttons\_state(self):

        """

        Включает/выключает кнопки в зависимости от заполненности полей.

        """

        has\_size = self.board\_size\_input.text() != ""

        has\_to\_place = self.pieces\_to\_place\_input.text() != ""

        has\_placed = self.placed\_pieces\_input.text() != ""

        self.create\_board\_button.setEnabled(has\_placed and self.placed\_pieces\_input.text() != "0" and has\_size)

        try:

            placed\_count = int(self.placed\_pieces\_input.text()) if self.placed\_pieces\_input.text().isdigit() else 0

            self.draw\_board\_button.setEnabled(has\_size and has\_to\_place and has\_placed and

                                           (placed\_count == 0 or (self.board\_initialized and hasattr(self, 'pieces\_positions') and

                                            len(self.pieces\_positions) == placed\_count)))

        except ValueError:

            self.draw\_board\_button.setEnabled(False)

    def get\_pieces\_info(self) -> Tuple[int, int, List[Tuple[int, int]]]:

        """

        Возвращает размер доски, требуемое количество фигур и координаты размещенных фигур.

        Returns:

            Кортеж (размер доски, кол-во фигур, координаты).

        """

        return int(self.board\_size\_input.text()), int(self.pieces\_to\_place\_input.text()), self.pieces\_positions

    def \_initialize\_board(self):

        """

        Создание доски и активация кнопки рисования.

        """

        try:

            size\_text = self.board\_size\_input.text()

            to\_place\_text = self.pieces\_to\_place\_input.text()

            placed\_text = self.placed\_pieces\_input.text()

            try:

                board\_size = int(size\_text)

                pieces\_to\_place = int(to\_place\_text)

                placed\_count = int(placed\_text)

            except ValueError:

                QMessageBox.critical(self, "Ошибка", "Введены некорректные числовые значения.")

                self.draw\_board\_button.setEnabled(False)

                return

            if placed\_count > 0:

                input\_dialog = CoordinatesInputDialog(placed\_count, self)

                if input\_dialog.exec() == QDialog.Accepted:

                    self.pieces\_positions = input\_dialog.result\_positions

            else:

                self.pieces\_positions = []

            self.board\_initialized = True

            self.draw\_board\_button.setEnabled(True)

        except Exception as e:

            QMessageBox.critical(self, "Ошибка", f"Доска не создалась: {str(e)}")

            self.draw\_board\_button.setEnabled(False)

    def \_render\_board(self):

        """

        Отрисовка доски.

        """

        if self.board\_initialized:

            try:

                size\_text = self.board\_size\_input.text()

                to\_place\_text = self.pieces\_to\_place\_input.text()

                placed\_text = self.placed\_pieces\_input.text()

                try:

                    board\_size = int(size\_text)

                    pieces\_to\_place = int(to\_place\_text)

                    placed\_count = int(placed\_text)

                except ValueError:

                    QMessageBox.critical(self, "Ошибка", "Введены некорректные числовые значения.")

                    return

                board, placement\_options = self.solver.create\_board(board\_size, pieces\_to\_place, placed\_count, self.pieces\_positions)

                self.solver.start\_solving(board, placement\_options)

                self.solver.solving\_complete.connect(self.\_handle\_solution)

            except Exception as e:

                QMessageBox.critical(self, "Ошибка", f"Ошибка при отрисовке доски: {str(e)}")

        else:

            QMessageBox.warning(self, "Ошибка", "Создайте доску")

    def \_handle\_solution(self, board: "ChessBoard", solutions: Set[frozenset]):

        """

        Обработчик сигнала `finished` потока решателя.

        Args:

            board: Шахматная доска.

            solutions: Множество решений.

        """

        if solutions:

            solution\_coordinates = list(next(iter(solutions)))

            solution\_dialog = DeskWindow(board, solution\_coordinates, self)

            solution\_dialog.exec()

    def closeEvent(self, event):

        """

        Обработчик события закрытия окна.

        Остановка всех потоков при закрытии.

        """

        active\_threads = [

            getattr(self, attr) for attr in dir(self)

            if attr.endswith('\_thread') and isinstance(getattr(self, attr), QThread)

        ]

        for thread in active\_threads:

            if thread.isRunning():

                thread.quit()

                thread.wait(500)

        event.accept()

class CoordinatesInputDialog(QDialog):

    """

    Диалоговое окно для ввода координат фигур.

    """

    def \_\_init\_\_(self, pieces\_count: int, parent: "ChessAppWindow"):

        """

        Инициализация диалогового окна.

        Args:

            num\_figures: Количество фигур, для которых нужно ввести координаты.

            parent: Родительское окно (ChessAppWindow).

        """

        super().\_\_init\_\_(parent)

        self.parent\_window = parent

        self.pieces\_count = pieces\_count

        self.setWindowTitle("Ввод координат поставленных фигур")

        self.coordinate\_fields: List[QLineEdit] = []

        self.confirm\_button = QPushButton("OK")

        self.cancel\_button = QPushButton("Отмена")

        main\_layout = QVBoxLayout()

        self.coordinates\_layout = QVBoxLayout()

        main\_layout.addLayout(self.coordinates\_layout)

        buttons\_layout = QHBoxLayout()

        buttons\_layout.addWidget(self.confirm\_button)

        buttons\_layout.addWidget(self.cancel\_button)

        main\_layout.addLayout(buttons\_layout)

        self.setLayout(main\_layout)

        self.\_setup\_input\_fields()

        self.confirm\_button.clicked.connect(self.accept)

        self.cancel\_button.clicked.connect(self.reject)

    def \_setup\_input\_fields(self):

        """

        Создание полей ввода для координат фигур.

        """

        for i in range(self.pieces\_count):

            label = QLabel(f"Фигура №{i + 1}:")

            input\_field = QLineEdit()

            input\_field.setPlaceholderText("Пример: 1 1")

            self.coordinate\_fields.append(input\_field)

            self.coordinates\_layout.addWidget(label)

            self.coordinates\_layout.addWidget(input\_field)

            self.coordinate\_fields[-1].textChanged.connect(self.\_validate\_inputs)

    def \_check\_safety(self, pieces\_positions: List[Tuple[int, int]], board\_size: int) -> bool:

        """

        Проверка, находятся ли фигуры в безопасных позициях на доске.

        Args:

            pieces: Список координат фигур.

            n: Размер доски.

        Returns:

            True, если все фигуры в безопасных позициях, иначе False.

        """

        temp\_board = ChessBoard(board\_size)

        for x, y in pieces\_positions:

            piece = ChessFigure(x, y)

            if not temp\_board.place\_piece(piece):

                return False

        return True

    def accept(self):

        """

        Обработка нажатия кнопки "OK".

        Извлечение координат из QLineEdit и сохранение в `resFig`.

        Проверка безопасного расположения фигур.

        """

        coordinates = []

        is\_valid = True

        board\_size = 0

        try:

            board\_size = int(self.parent\_window.board\_size\_input.text())

        except ValueError:

            QMessageBox.warning(self, "Ошибка", "Неверный формат ввода N.")

            return

        for field in self.coordinate\_fields:

            text = field.text().strip()

            try:

                x, y = map(int, text.split())

                if not (0 <= x < board\_size and 0 <= y < board\_size):

                    QMessageBox.warning(self, "Ошибка", "Координаты выходят за пределы доски.")

                    is\_valid = False

                    break

                coordinates.append((x, y))

            except ValueError:

                QMessageBox.warning(self, "Ошибка", "Неверный формат ввода координат.")

                is\_valid = False

                break

        if is\_valid:

            if self.\_check\_safety(coordinates, board\_size):

                self.result\_positions = coordinates

                super().accept()

            else:

                QMessageBox.warning(self, "Ошибка", "Невозможно расположение, фигуры под ударом")

    def \_validate\_inputs(self):

        """

        Проверка валидности введенных координат.

        Включение/выключение кнопки "OK" в зависимости от валидности всех полей.

        """

        all\_valid = True

        for field in self.coordinate\_fields:

            text = field.text().strip()

            if text:

                try:

                    x, y = map(int, text.split())

                    n = int(self.parent\_window.board\_size\_input.text())

                    if not (0 <= x < n and 0 <= y < n):

                        all\_valid = False

                except ValueError:

                    all\_valid = False

            else:

                all\_valid = False

        self.confirm\_button.setEnabled(all\_valid)

class DeskWindow(QDialog):

    """

    Окно для отображения решения на шахматной доске.

    """

    def \_\_init\_\_(self, chess\_board: "ChessBoard", solution: List[Tuple[int, int]], parent: Optional[QWidget] = None):

        """

        Инициализация диалогового окна.

        Args:

            table: Объект шахматной доски.

            solution: Решение в виде списка кортежей (x, y).

            Parent: Родительский виджет.

        """

        super().\_\_init\_\_(parent)

        self.chess\_board = chess\_board

        self.solution = solution

        self.parent\_window = parent

        self.main\_layout = QVBoxLayout(self)

        self.board\_widget = QWidget()

        self.buttons\_layout = QHBoxLayout()

        self.save\_button = QPushButton("Запись в файл")

        self.save\_button.setStyleSheet("background-color: green; font-weight: bold;")

        self.close\_button = QPushButton("Выход")

        self.close\_button.setStyleSheet("background-color: red; font-weight: bold;")

        self.buttons\_layout.addWidget(self.save\_button)

        self.buttons\_layout.addWidget(self.close\_button)

        self.main\_layout.addWidget(self.board\_widget)

        self.main\_layout.addLayout(self.buttons\_layout)

        self.board\_widget.setStyleSheet("background-color: white;")

        self.save\_button.clicked.connect(self.\_save\_solution)

        self.close\_button.clicked.connect(self.close)

        self.setWindowTitle("Один из вариантов решения")

        self.\_display\_solution()

    def \_display\_solution(self):

        """

        Визуализация решения на шахматной доске.

        """

        board\_size = len(self.chess\_board.cells)

        scene = QGraphicsScene()

        view = QGraphicsView()

        view.setScene(scene)

        view.setRenderHint(QPainter.Antialiasing)

        view.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff)

        view.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOff)

        layout = QVBoxLayout(self.board\_widget)

        layout.addWidget(view)

        scene\_dimension = board\_size \* 40

        view.setSceneRect(QRectF(-1, -1, scene\_dimension + 2, scene\_dimension + 2))

        scene.setSceneRect(QRectF(view.sceneRect()))

        view.setFixedSize(scene\_dimension + 2, scene\_dimension + 2)

        temporary\_pieces = []

        for x, y in self.solution:

            if self.chess\_board.is\_position\_empty(x, y):

                piece = ChessFigure(x, y)

                if self.chess\_board.place\_piece(piece):

                    temporary\_pieces.append(piece)

        for x in range(board\_size):

            for y in range(board\_size):

                if self.chess\_board.cells[y][x]['threatened']:

                    cell = create\_board\_cell(x, y, 3)

                elif self.chess\_board.is\_position\_empty(x, y) and (y + x) % 2 == 0:

                    cell = create\_board\_cell(x, y, 0)

                elif self.chess\_board.is\_position\_empty(x, y) and (y + x) % 2 != 0:

                    cell = create\_board\_cell(x, y, 1)

                else:

                    cell = create\_board\_cell(x, y, 2)

                scene.addItem(cell)

        for piece in temporary\_pieces:

            self.chess\_board.remove\_piece(piece)

    def \_save\_solution(self):

        """

        Сохранение решения в файл.

        """

        filename, \_ = QFileDialog.getSaveFileName(

            self,

            "Сохранить решение",

            "",

            "Текстовые файлы (\*.txt);;Все файлы (\*)"

        )

        if filename:

            try:

                self.save\_thread = SolutionSaver(self.parent\_window.solver, self.parent\_window.solver.found\_solutions, filename)

                self.save\_thread.completed.connect(self.\_handle\_save\_result)

                self.save\_thread.start()

            except Exception as e:

                QMessageBox.critical(self, "Ошибка", f"Сохранение не удалось: {e}")

    def \_handle\_save\_result(self, filename: str, error\_msg: str):

        """

        Обработчик сигнала `finished` потока сохранения.

        Args:

            filename: Имя файла, в который было сохранено решение.

            error\_message: Сообщение об ошибке, если возникла.

        """

        print(len(self.parent\_window.solver.found\_solutions))

        if error\_msg:

            QMessageBox.critical(self, "Ошибка", f"Сохранение не удалось: {error\_msg}")

        else:

            QMessageBox.information(

                self,

                "Успешно",

                f"Результат сохранен в {filename}"

            )

class SolutionSaver(QThread):

    """

    Поток для сохранения решения в файл.

    """

    completed = Signal(str, str)

    def \_\_init\_\_(self, solver: "ChessSolver", solutions: Set[frozenset], filename: str):

        """

        Инициализация потока.

        Args:

            solver: Объект решателя.

            solutions: Множество решений.

            filename: Имя файла для сохранения.

        """

        super().\_\_init\_\_()

        self.solver = solver

        self.solutions = solutions

        self.filename = filename

    def run(self):

        """

        Выполнение сохранения решения в файл.

        """

        try:

            self.solver.save\_results(self.filename, self.solutions)

            self.completed.emit(self.filename, "")

        except Exception as e:

            self.completed.emit(self.filename, str(e))

if \_\_name\_\_ == '\_\_main\_\_':

    app = QApplication([])

    window = ChessAppWindow()

    window.show()

    app.exec()

ChessLogic.py

from itertools import combinations\_with\_replacement

from typing import List, Tuple, Set

from PySide6.QtCore import QThread, Signal

class ChessFigure:

    def \_\_init\_\_(self, pos\_x: int, pos\_y: int):

        """

        Инициализация фигуры.

        Args:

            x: Координата X фигуры.

            y: Координата Y фигуры.

            possible\_moves: Возможные ходы фигуры.

        """

        self.possible\_moves = [(-1, 0), (1, 0), (0, -1), (0, 1), (-2, 0), (2, 0), (0, -2), (0, 2)]

        self.pos\_x = pos\_x

        self.pos\_y = pos\_y

class ChessBoard:

    """

    Класс для представления шахматной доски и ее клеток.

    """

    def \_\_init\_\_(self, board\_size: int):

        """

        Инициализация доски.

        Args:

            size: Размер доски.

        """

        self.board\_size = board\_size

        self.cells: List[List[dict]] = [[{'x': x, 'y': y, 'piece': None, 'threatened': False} for x in range(board\_size)] for y in range(board\_size)]

        self.placed\_pieces: List[ChessFigure] = []

    def is\_position\_empty(self, pos\_x: int, pos\_y: int) -> bool:

        """

        Проверяет, пуста ли клетка.

        Args:

            x: Координата X клетки.

            y: Координата Y клетки.

        Returns:

            True, если клетка пуста, иначе False.

        """

        return self.cells[pos\_y][pos\_x]['piece'] is None

    def place\_piece(self, piece: ChessFigure) -> bool:

        """

        Добавляет фигуру на доску с проверкой безопасности.

        Args:

            piece: Фигура для добавления.

        Returns:

            True, если фигура успешно добавлена, иначе False.

        """

        if not self.is\_position\_empty(piece.pos\_x, piece.pos\_y) or not self.is\_position\_safe(piece):

            return False

        self.cells[piece.pos\_y][piece.pos\_x]['piece'] = piece

        self.placed\_pieces.append(piece)

        self.update\_threatened\_cells()

        return True

    def remove\_piece(self, piece: ChessFigure):

        """

        Удаляет фигуру с доски.

        Args:

            piece: Фигура для удаления.

        """

        cell = self.cells[piece.pos\_y][piece.pos\_x]

        if cell['piece'] == piece:

            cell['piece'] = None

            self.placed\_pieces.remove(piece)

            self.update\_threatened\_cells()

    def is\_position\_safe(self, piece: ChessFigure) -> bool:

        """

        Проверка безопасности позиции для фигуры.

        Args:

            piece: Фигура для проверки.

        Returns:

            True, если позиция безопасна, иначе False.

        """

        if not (0 <= piece.pos\_x < self.board\_size and 0 <= piece.pos\_y < self.board\_size):

            return False

        if not self.is\_position\_empty(piece.pos\_x, piece.pos\_y):

            return False

        for other\_piece in self.placed\_pieces:

            for dx, dy in other\_piece.possible\_moves:

                if other\_piece.pos\_x + dx == piece.pos\_x and other\_piece.pos\_y + dy == piece.pos\_y:

                    return False

        return True

    def update\_threatened\_cells(self):

        """

        Обновляет клетки под боем на доске.

        """

        # Сброс всех клеток под боем

        for row in self.cells:

            for cell in row:

                cell['threatened'] = False

        # Установка клеток под боем всех фигур

        for piece in self.placed\_pieces:

            for dx, dy in piece.possible\_moves:

                new\_x, new\_y = piece.pos\_x + dx, piece.pos\_y + dy

                if 0 <= new\_x < self.board\_size and 0 <= new\_y < self.board\_size:

                    self.cells[new\_y][new\_x]['threatened'] = True

class ChessSolver(QThread):

    """

    Класс для решения задачи расстановки фигур на шахматной доске.

    """

    solving\_complete = Signal(object, object)

    def \_\_init\_\_(self):

        """

        Инициализация расстановщика.

        """

        super().\_\_init\_\_()

        self.found\_solutions: Set[frozenset] = set()

    @staticmethod

    def save\_results(output\_path: str, solutions: Set[frozenset]):

        """

        Записывает результаты в файл.

        Args:

            file\_path: Путь к файлу для записи.

            solutions: Множество решений.

        """

        with open(output\_path, 'w') as output\_file:

            if solutions:

                for solution in solutions:

                    output\_file.write(" ".join(f"({x}, {y})" for x, y in solution) + "\n")

            else:

                output\_file.write("no solutions\n")

    def find\_solutions(self, board: ChessBoard, piece\_arrangements: List[str]):

        """

        Находит все решения с помощью рекурсии.

        Args:

            board: Шахматная доска.

            combinations: Список комбинаций фигур для размещения.

        """

        def recursive\_solver(remaining\_pieces: List[str]):

            if not remaining\_pieces:

                self.found\_solutions.add(frozenset((p.pos\_x, p.pos\_y) for p in board.placed\_pieces))

                return

            current\_piece = remaining\_pieces[0]

            for y in range(board.board\_size):

                for x in range(board.board\_size):

                    new\_piece = ChessFigure(x, y)

                    if board.place\_piece(new\_piece):

                        recursive\_solver(remaining\_pieces[1:])

                        board.remove\_piece(new\_piece)

        for arrangement in piece\_arrangements:

            recursive\_solver(arrangement)

    def start\_solving(self, board: ChessBoard, piece\_arrangements: List[str]):

        """

        Запускает решение задачи в отдельном потоке.

        Args:

            board: Шахматная доска.

            combinations: Список комбинаций фигур для размещения.

        """

        self.chess\_board = board

        self.piece\_arrangements = piece\_arrangements

        self.start()

    def run(self):

        """

        Выполнение решения задачи расстановки фигур на шахматной доске.

        """

        self.find\_solutions(self.chess\_board, self.piece\_arrangements)

        self.solving\_complete.emit(self.chess\_board, self.found\_solutions)

    @staticmethod

    def create\_board(n: int, l: int, k: int, initial\_pieces: List[Tuple[int, int]]) -> Tuple[ChessBoard, List[str]]:

        """

        Генерирует шахматную доску и список комбинаций фигур для размещения.

        Args:

            n: Размер доски.

            l: Количество фигур для размещения.

            k: Количество предварительно размещенных фигур.

            pieces: Координаты предварительно размещенных фигур.

        Returns:

            Кортеж (шахматная доска, список комбинаций фигур).

        """

        board = ChessBoard(n)

        # Добавляем предварительно размещенные фигуры

        if k > 0:

            for coordinates in initial\_pieces:

                piece = ChessFigure(coordinates[0], coordinates[1])

                board.place\_piece(piece)

        # Генерация вариантов расстановки фигур для размещения

        placement\_options = set(combinations\_with\_replacement(' ', l))

        return board, list(placement\_options)

# Заключение

В результате работы над курсовым проектом я научился проектировать программу в рамках ООП, создал свою первую модульную программу, UML схему, овладел несколькими библиотеками Pyside6.