НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Кафедра автоматизованих систем обробки інформації і систем управління

**ЛАБОРАТОРНА РОБОТА № 1**

з дисципліни «Об’єктно-орієнтоване програмування»

на тему: «Основні концепції об’єктно-орієнтованого програмування»

Виконав:

Студент 2-го курсу групи ІП-72

Федоров О.В.

Перевірив:

Київ – 2018 рік

Зміст

[Ціль роботи 3](#_Toc524973571)

[Постановка задачі 4](#_Toc524973572)

[Алгоритм задачі 5](#_Toc524973573)

[Діаграма класів 7](#_Toc524973574)

[Код програми 8](#_Toc524973575)

[Приклади роботи 11](#_Toc524973576)

[Висновки 13](#_Toc524973577)

# Ціль роботи

Вивчити основні концепції об’єктно-орієнтованого програмування. Вивчити особливості використання класів і об’єктів, а також особливості застосування конструкторів і деструкторів.

# Постановка задачі

Визначити клас «Шахова дошка» із закритим елементом: масив клітинок – *А* (8х8). Визначити такі конструктори:

* Конструктор ініціалізації, що розміщує в заданій клітинці шахову фігуру – коня.

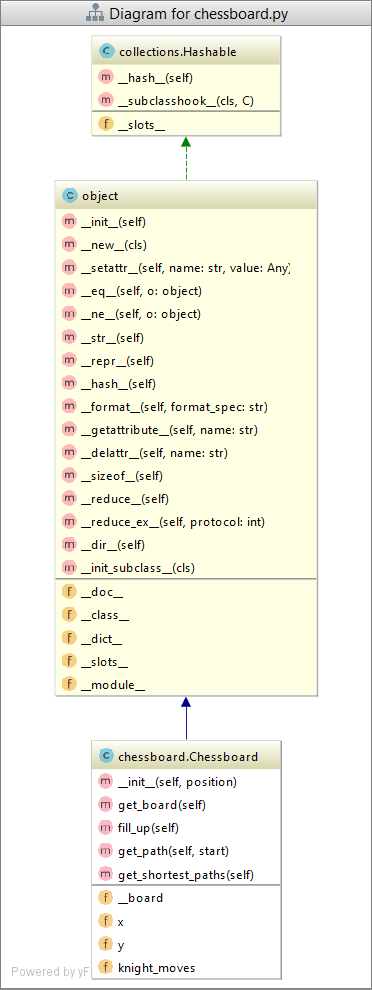
Визначити метод для пошуку шляхів коня до чотирьох кутових клітинок дошки за найменшу кількість кроків. Шахова фігура – кінь – може переміщуватися на 1 клітинку по горизонталі і 2 по вертикалі, або на 2 по горизонталі та 1 по вертикалі.

Якщо за початкову клітинку задана кутова, знайти шляхи до трьох, що залишилися.

# Алгоритм задачі

1. ПОЧАТОК
2. Заповнити дошку значеннями, які відповідають мінімальній кількості кроків, яку треба зробити аби потрапити у дану точку із точки відліку
   1. ПОЧАТОК
   2. Створити масив кортежів *visited* із значенням координати Х, координати Y і кількості попередньо-використаних кроків.
   3. Для кожного кортежу в масиві під індексами від 0 до 63 включно:
      1. ПОЧАТОК
      2. Взяти координати X та Y
      3. Згенерувати масив *possible* можливих та неможливих точок, в які можна потрапити із клітинки (X, Y)
      4. Видалити всі не існуючі на дошці або вже відвідані точки із масиву *possible*
      5. Відмітити точки, що залишилися на дошці
      6. Додати їх до масиву *visited*
      7. КІНЕЦЬ
   4. КІНЕЦЬ
3. Створити пустий масив *result*
4. Для кожної координати кутової клітинки:
   1. ПОЧАТОК
   2. Викликати рекурсивний метод пошуку зворотних шляхів
      1. ПОЧАТОК
      2. Згенерувати масив *back\_tracks* із можливих і неможливих точок, із яких можна потрапити в клітинку, що розглядається
      3. Видалити з масиву *back\_tracks* всі точки, що не існують на дошці або не є частиною найкоротшого шляху від координати коня (якщо в точку, що розглядається можна потрапити за *n* кроків, то шлях до попередньої точки має складається з *(n-1)* кроків коня)
      4. ЯКЩО масив *back\_tracks* не пустий:
         1. Рекурсивно викликати цей метод на кожній точці масиву, генеруючи зворотні шляхи між ними
         2. До кожного отриманого зворотного шляху додати точку, що розглядається на даному етапі рекурсії
         3. ПОВЕРНУТИ масив зворотних шляхів
      5. ІНАКШЕ:
         1. ПОВЕРНУТИ масив із шляхом, що складається із точки, що розглядається на даному етапі рекурсії
      6. КІНЕЦЬ
   3. Додати результуючий масив шляхів до масиву *result*
   4. КІНЕЦЬ
5. ПОВЕРНУТИ масив *result*
6. КІНЕЦЬ

# UML-діаграма класів



# Код програми

class Chessboard:

"""

Конструктор (метод ініціалізації) екземплярів класу "Шахова дошка".

Вхід: рядок із координатами коня

Вихід: відсутній

"""

def \_\_init\_\_(self, position):

self.\_\_A = [[None] \* 8 for \_ in range(8)] # згенерувати матрицю розміром 8х8

self.knight\_moves = ((2, 1), (2, -1), (-2, 1), (-2, -1), # перелік можливих змін координат

(1, 2), (1, -2), (-1, 2), (-1, -2)) # по осях X та Y

self.x, self.y = normalize\_position(position)

self.\_\_A[self.x][self.y] = 0 # "встановити" коня у вказану точку

"""

Метод, що повертає шахову дошку у вигляді масиву масивів кількості кроків до певної точки.

Вхід: відсутній

Вихід: копія шахової дошки

"""

def get\_board(self):

return [line[:] for line in self.\_\_A]

"""

Метод для знаходження найкоротшого шляху у всі точки шахової дошки.

Вхід: відсутній

Вихід: відсутній

"""

def fill\_up(self):

"""

Допоміжна функція для пошуку точок, в які можна попасти із координати, що розгяладається.

Вхід: кортеж із координатою і відстанню точки від коня

Вихід: значення True або False

"""

def valid(tpl):

row, col, \_ = tpl

in\_range = 0 <= row < 8 and 0 <= col < 8

return in\_range and self.\_\_A[row][col] is None

visited = [(self.x, self.y, 0)] # черга відмічених точок

for i in range(64):

x, y, n = visited[i]

# знайти всі точки, в які може переміститися кінь, і які наявні на дошці

possible = [\*filter(valid, ((x+dx, y+dy, n+1) for dx, dy in self.knight\_moves))]

# відмітити точки на дошці

for x0, y0, m in possible:

self.\_\_A[x0][y0] = m

# додати відмічені точки до черги

visited.extend(possible)

"""

Рекурсивний метод для знаходження зворотнього шляху від однієї координати до іншої. Рекурсія

припиняється при досягненні точки, в якій знаходиться кінь.

Вхід: координата у вигляді кортежу

Вихід: масив масивів із найкоротшими шляхами із довільної координати до місцезнаходження коня

"""

def get\_path(self, start):

"""

Допоміжна функція для пошуку точок, із яких можна попасти в координату, що розгяладається.

Вхід: кортеж із координатою і відстанню точки від коня

Вихід: значення True або False

"""

def closer(tpl):

row, col, m = tpl

in\_range = 0 <= row < 8 and 0 <= col < 8

return in\_range and self.\_\_A[row][col] == m

x, y, n = start

stringified = stringify\_position(\*start)

# знайти всі існуючі на дошці точки, з яких можна потрапити в точку, що розглядається

back\_tracks = filter(closer, ((x+dx, y+dy, n-1) for dx, dy in self.knight\_moves))

# додати точку, що розглядається до кожного шляху, знайденого під час рекурсії

paths = [self.get\_path(point)[0] + [stringified] for point in back\_tracks]

return paths or [[stringified]]

"""

Метод для знаходження найкоротших шляхів коня до країв дошки.

Вхід: відсутній

Вихід: масив із масивами усіх найкоротших шляхів коня до країв дошки, де шлях - масив із

координатами кроків від точки відліку до краю

"""

def get\_shortest\_paths(self):

self.fill\_up()

result = []

# для кожної точки: якщо вона не є початкова, знайти найкоротший шлях до неї

for coord in ["A1", "A8", "H1", "H8"]:

x, y = normalize\_position(coord)

if self.\_\_A[x][y]:

result.append(self.get\_path((x, y, self.\_\_A[x][y])))

return result

"""

Функція конвертації координати шахової дошки "рядок у кортеж".

Вхід: координата у вигляді рядка

Вихід: координата у вигляді кортежу

"""

def normalize\_position(position):

letter, number = list(position)

return (8 - int(number), ord(letter.upper()) - 65)

"""

Функція конвертації координати шахової дошки "кортеж у рядок".

Вхід: обов'язкові аргументи - координати Х та Y; необов'язкові аргументи на випадок, якщо

разом із координатою подається її "відстань" від координати розташування коня

Вихід: координата у вигляді рядка

"""

def stringify\_position(x, y, \*args):

return "{}{}".format(chr(65 + y), 8 - x)

"""

Функція для покращеного показу шляху до краю дошки.

Вхід: масив координат точок на дошці

Вихід: рядок, що візуалізує шлях краю дошки

"""

def visualize\_path(path):

return " -> ".join(path)

"""

Функція для показу дошки із нанесеним на неї шляхом.

Вхід: масив координат точок на дошці

Вихід: рядок, що візуалізує дошку, із нанесеними на неї точками

"""

def visualize\_path\_on\_board(path):

# згенерувати матрицю пробілів розміром 8х8

board = [[" "] \* 8 for \_ in range(8)]

# замінити всі точки, що є частинами шляху відповідним символом:

# "K" ("knight") - кінь

# i - номер кроку

# "T" ("target") - ціль

for i, coord in enumerate(path):

x, y = normalize\_position(coord)

s = ["K", i, "T"][(i > 0) + (i == len(path) - 1)]

board[x][y] = s

full\_lines = []

# далі: конвертувати матрицю у рядок, що має вигляд шахової дошки з розміткою

for i, line in enumerate(board):

full\_line = "|".join(" {} ".format(x) for x in line) + " {}".format(8 - i)

full\_lines.append(full\_line)

non\_empty\_line = "\n{}\n".format("+".join(["-" \* 3] \* 8))

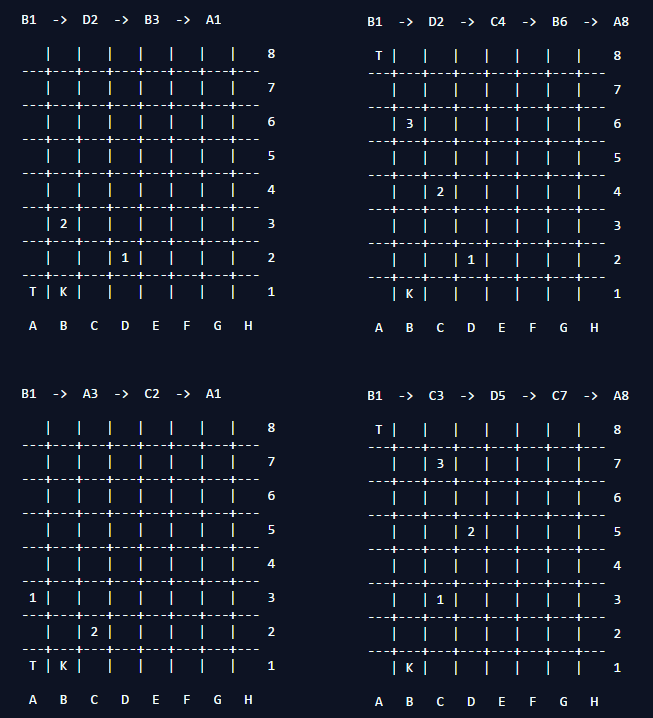
board\_without\_letters = non\_empty\_line.join(full\_lines)

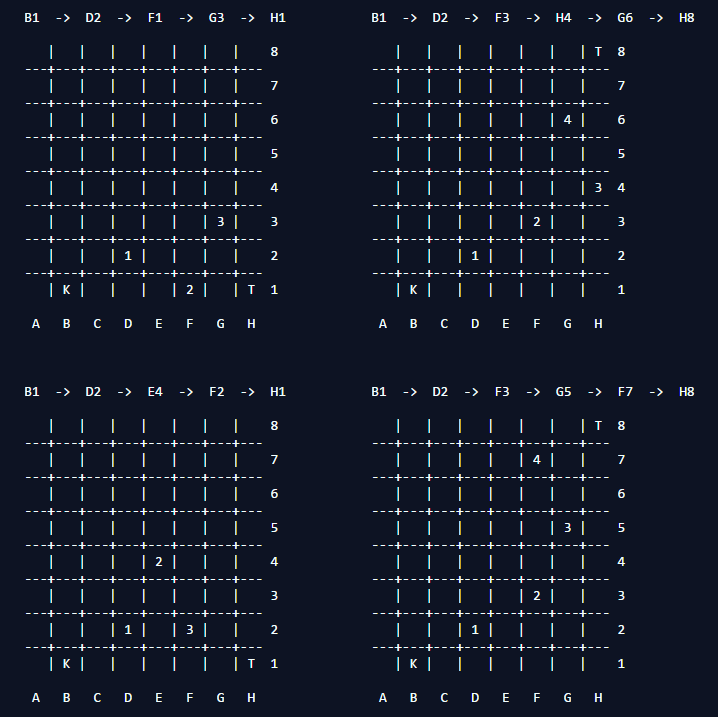
letters = " ".join(" {} ".format(chr(x)) for x in range(65, 73))

board = board\_without\_letters + "\n\n" + letters

return board

# Приклади роботи





# Висновки

Під час виконання даної лабораторної роботи було вивчено головні концепції та особливості об’єктно-орієнтованого підходу у програмуванні, а також були розглянуті основи використання даної парадигми для написання програм. У ролі індивідуального практичного завдання було обрано, досліджено та реалізовано алгоритм пошуку найкоротших шляхів між 2 точками за фіксованої «ціни» кроків із використанням принципів ООП, заради зручного зберігання, обробки інформації та захисту її від зовнішнього впливу інших частин програми.