Final Project

Algorithm and Programming

Project Name: “ChessPy”

Student Name: Peter Nelson Subrata

Student ID: 2502023562

Class: L1CC

Table of Contents

A. Description

I. Introduction

Early in the semester it was made aware to me that some sort of project would determine the final percentage of the total course grade for the semester and at that time and point I had already thought of some ideas for what to make. Before I decided to make a simple chess program, I had to shuffle through many ideas that didn't make it such as a sky map program and a web scraper, but after deciding that either the task was too difficult or too simple in such a way that it didn't feel like it met the requirements, I decided to look for alternatives and that's when I thought to make a chess program on python. A project idea that was the perfect fit for me.

I begin this project on the 23rd of December 2021. I chose to use the PyCharm IDE for this project and I will be posting all of the code along with this report on my repository which can be accessed through my GitHub account here:

https://github.com/PewterZz/ChessPy---A-python-chess-program

II. The function of this program

The function that this program serves is that of a simple chess program that can provide entertainment for the player(s). It works just like a normal chess game would and has similar gameplay and rules overall.

The game is played on an 8x8 board and has over 32 pieces each with their own specific movement sets and rules. The way to play would be first to choose between a colour white or black and move pieces accordingly to your colour. Each turn a player would move starting from white and each player will have to react accordingly to the plays made by the other player.

This encourages strategic thinking that goes well beyond the current state of the board being played, you will have to think ahead of your opponents to win.

B. Design

I. Parts of the program and requirements

Requirements:

- PyGame
  - Used to design the overall game and visualize it into a window.
- OS
  - Used to access file locations and to use it to display sprites.

In this program I used a window of dimensions 626 width and 626 height. The window starts out with the starting screen which is just a sprite picture which tells you to click anywhere on the screen to initiate the game process. Once clicked program will load and it will start the game which is also using the same dimensions as the window for the board. The board is a sprite which contains several other sprites in the form of some text and the chess pieces.

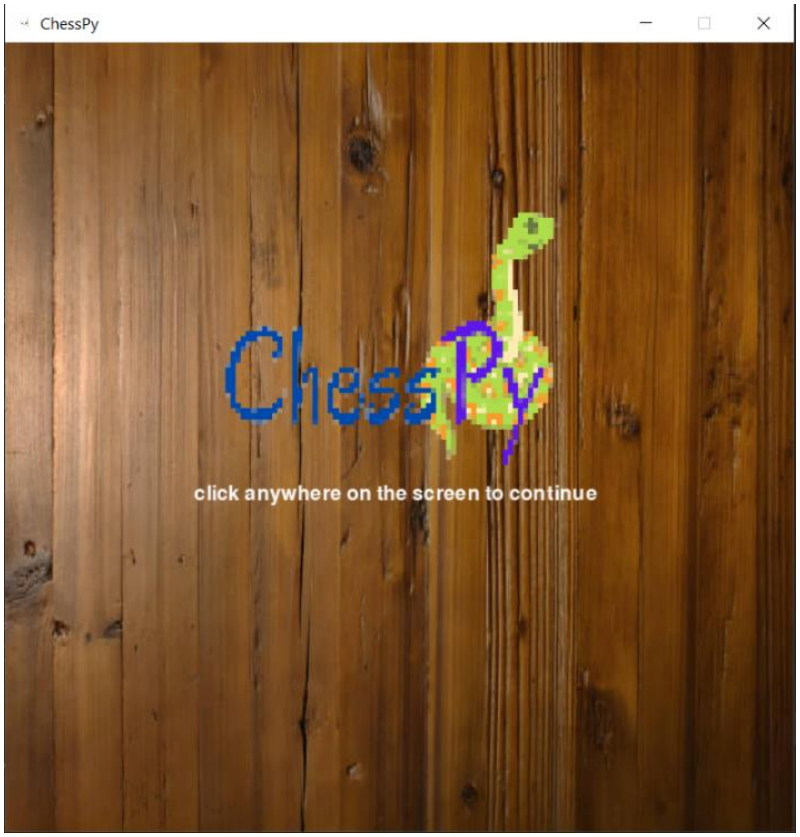

Figure 1: Starting screen for window

II. Function of each part of the program

Each chess piece can be clicked to select it, which will show in the program and it will also show you the actions that each piece can take. You left click to select a piece and right click on a red dot to move the piece in your desired location. Once the piece has moved new moves will appear that will adjust to your location on the board and if an enemy is nearby to attack. If a piece attacks another piece of a different colour the piece that is attacked disappears and won't come back.

Figure 2: Game window showing the game being run and its pieces

Figure 3: Game window showing moving pieces and red dot

C. Implementation

I. Classes Diagram

Chessboard
-rows -columns
+ get_board() + draw(win, board) + selectinboard(columns, rows) + updater() + return_restricted_moves(color) + reset() + do_move_chesspiece(org, end, turn, color)

Figure 4: UML Class diagram of the chessboard class.

Chesspieces
-row -column -color
+ get_color() + get_selected() + possible_moves(board) + move_pos(pos) + border(win, board)

Figure 5: UML Class diagram of the chesspieces class.

Bishop, Pawn, Knight, Queen, King, Rook
-row -column -color
+ (inherited methods from chesspieces class) + move_valid(board)

Figure 6: UML Class diagram for individual pieces.

II. Flowcharts

Starting window screen flowchart

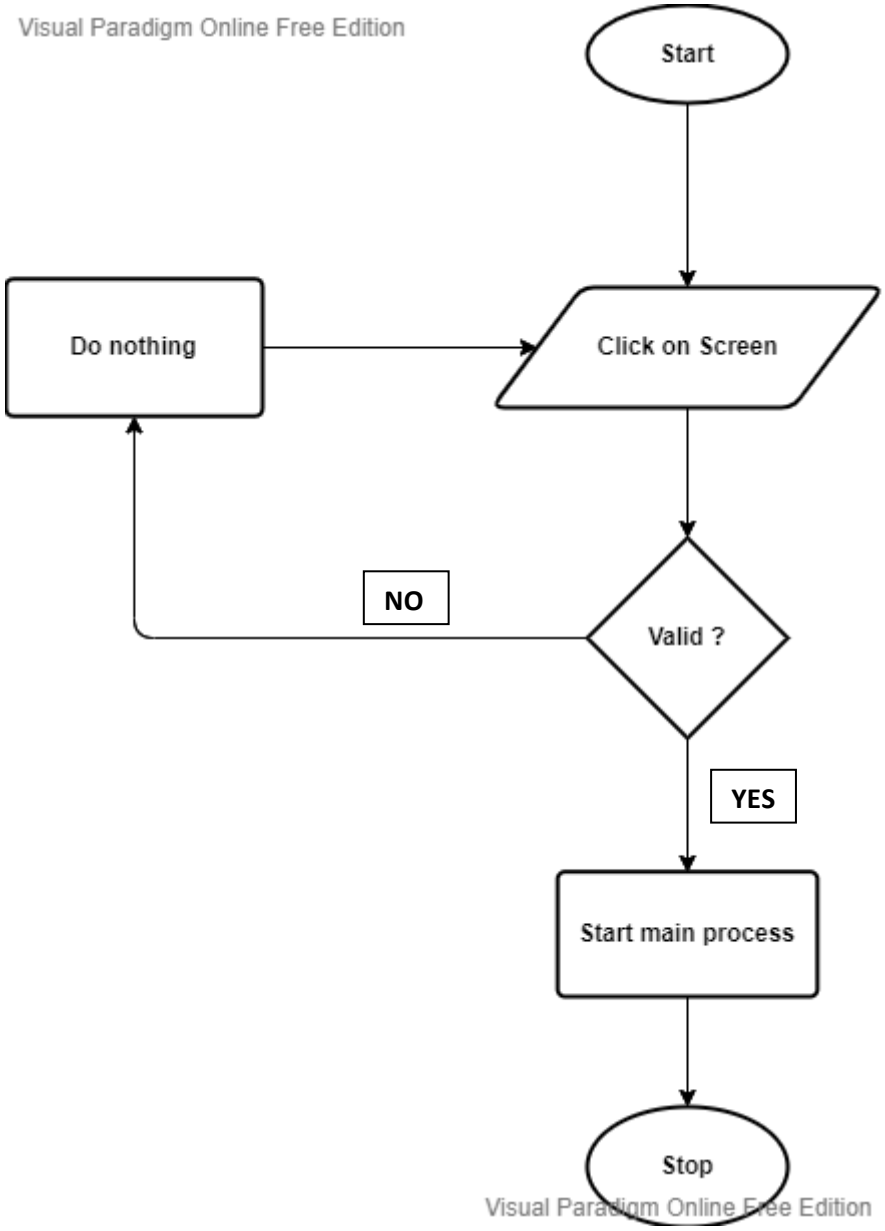

Figure 7: Flowchart of starting screen before main game process.

Main game process

Clicking on a piece

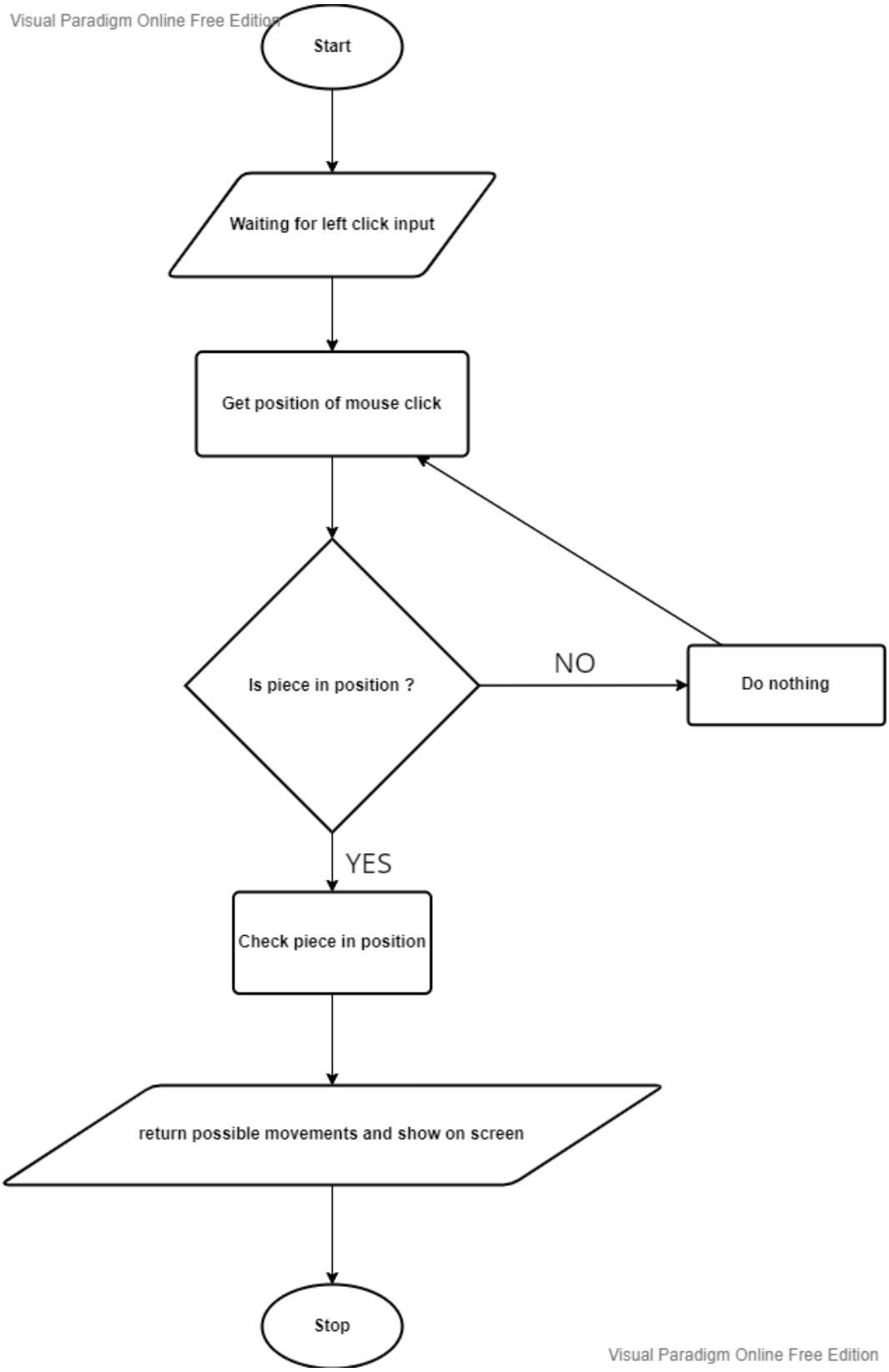

Figure 8: Clicking on a piece in the board flowchart.

Checking move list of selected pieces

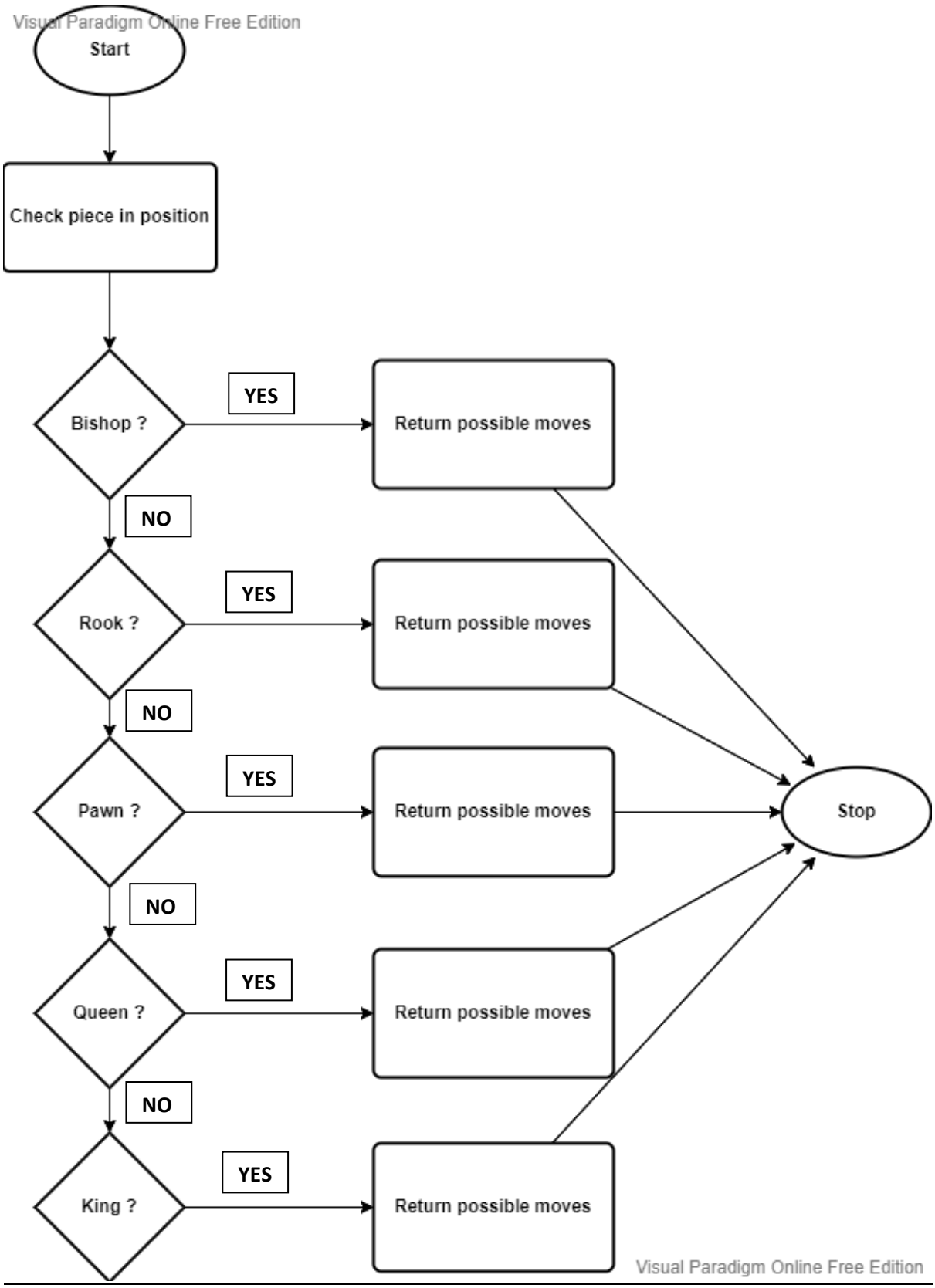

Figure 9: Checking move list of piece flowchart.

Moving a selected piece

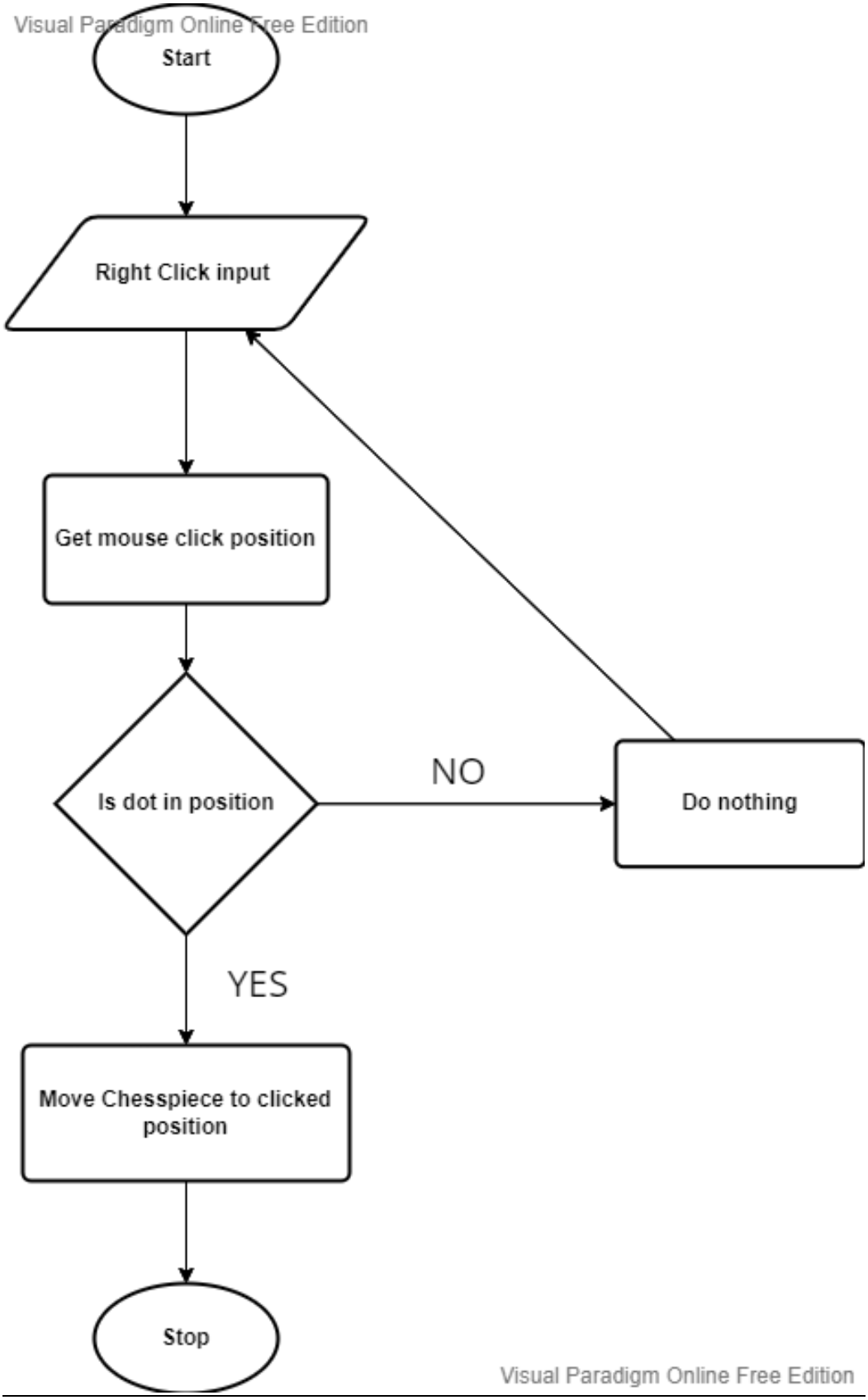

Figure 10: Moving a piece flowchart.

III. Extensibility

1. Comment lines are used to to further explain the functions of each function and method and the purpose of each of them to help other programmers.
2. Variables use meaningful identifiers to help other coders to understand easier.
3. Class diagram and flowchart to make it easier for others to understand and to easily identify features.

IV. Explanation of all the functions made and used

chessboard.py

- __init__(self, rows, columns):
    - Attributes of the class
    - Rows for the rows of the board
    - Columns for the columns in the board
    - Extra attribute self. board which is a matrix consisting of 8 arrays which contain 8 arrays which all contain the integer 0.
- get_board(self):
    - returns self.board for draw function
- draw(self, win, board):
    - Checks to see if an object is in the board and then calls the border function if condition is met.
    - win and board are used as parameter values within the border function.
- selectinboard(self, columns, rows):
    - starts out by turning all the objects in the matrix as (self.selected = False) which is done to make sure that when a piece is selected the border that is surrounding it to show that you are selecting it doesn't linger after selecting another piece. The second part of the function turns (self.selected = True) to the singular object in the columns and rows given from the parameters.
- updater(self):
    - updates the possible moves list based on the current position of each of the pieces in the board.
- reset(self):
    - turns all objects in the matrix as (self.selected = False).
- return_king_pos(self):
    - Returns a list which contains the positions of both kings.

- do_move_chesspiece(self, org, end, turn, color):
    - starts by call the updater method from before to update all possible moves of all current chesspieces. It then checks if four conditions are true, whether or not the turn is even or odd, whether or not the color of the piece is the same as the current color's turn, if the position where the piece will go is in the possible movements list of the object piece, and if the current position of the piece is the same as the position where it will go. If the turn is odd, the color is the same as the color of the piece, the position of where the piece will go is in the possible movements list and the position selected is different from the position where it will go, then white will move. Otherwise the turn is even, and other conditions are met then black will move.
    - This function also checks if the selected piece in the board is a pawn. If it is then turn the attribute self.first_movement, false. Once false that particular pawn object will stay with that condition for the rest of the game and it will only be given one singular move after its first movement, just like in normal chess.
    - The movement of the piece is done by first checking if the movement is possible by checking the possible movements list for that piece that is selected in the board then checking it with the two coordinate positions given by the end parameter. If it is then call the function move_pos with the parameters as the two coordinates given by the end parameter. Then subsequently leave an integer 0 behind where the previous object was located.
    - After all is done the updater function is called to update all the moves list of the pieces with their new positions.
    - Increase turn value by one and change the color to either Black or White depending on which condition was met before. Then return the turn and color.
    - If the move list contains king_pos define king as 1.
    - If there is only one king in king_pos list set condition to zero
- (Additional notes for this module):
    - Imports all the pieces from the MoveableUnitesandSprites.py file.
    - Each piece is immediately assigned the proper starting position as is in a usual chessboard.

MoveableUnitsandSprites.py

- __init__(self, row, column, color):
  - Attributes for the class chessboard.
  - Includes other attributes such as self.all_possible_movements_list which contains an empty array, self.pawn which starts out as False, self.selected_piece which starts out as False, and self.king which also starts out as False.
- get_color(self):.
  - returns self.color.
- get_selected(self):
  - returns self. selected_piece.
- possible_moves(self, board):
  - assigns move_valid function as the new list for all possible movements
- move_pos(self, pos):
  - assigns new positions for row and column of the piece.
- border(self, win, board):
  - the function that places all the sprites on the board and also draws a blue coloured border to define when a piece is selected as well as drawing red dots to show where the pieces can move depending on its current position.
- move_valid(self, board):
  - defined differently for each piece, essentially is all the logical operations that calculate where a piece can go and how it can attack other pieces. Returns all the moves in the form of a list with each value containing two values representing the x and the y coordinates.
- (Additional notes for this module):
  - Multiple of the sprites are loaded in this module and scaled to fit the chessboard.
  - The attributes self.pawn and self.king return true when either a pawn or king is selected.
  - Most of the lines of code in this module is just the logical operations for the move set of each of the pieces.
  - Imports pygame and OS, pygame for the blits and sprite loading, OS for searching the location of the files for the sprites.

maingame.py

- game_window():
  - places the sprites on the window, and updates the display. Always running and updating as the game continues.
- select(pos):
  - Used to determine mouse position on board. The pos parameter is the position of the mouse in x and y values given by using the pygame.mouse.get_pos() function. This function gives the exact value of the mouse click in pixels of the given window dimensions. Returns the coordinates of the piece being selected.
- main():
  - Starts the overall process of the game. Begins with defining starting rules and includes the tick rate which controls the fps of the game. It then runs a while loop that will run until the window is closed manually. Within the while loop the game_window function is run and a for loop which waits for events in this case mouse clicks and such, which will then determine what action will be done, such as left click to select and right click to move. It also adds the turn sprite as well as the turn counter.

- (Additional notes for this module):
  - Before the main function is called a bunch of different things are set. One of these things is the game field in which the actual game of chess takes place. It is defined by four values which represent the starting and ending coordinates of the game field.
  - Win is defined as the window in which the game takes place. It has dimensions 626 x 626.
  - A precursor while loop starts before the main function to include the main starting screen of the game which asks the user to click anywhere on the screen to proceed.
  - If the condition is set to 0 from the function do_move_chesspiece then initiate the endgame screen where a delay will be set for 5 seconds before the game ends.

D. Lessons learned

I. Using Pygame

Pygame is the module I used to make this game and as it was not a part of the main topic of the class, I had to learn it by watching tutorials and reading the documentation all of which was provided on the internet. Using pygame at first was very simple and seamless and it made me realize just how fun making something from scratch really is. I learnt how to load sprites and place them in particular positions of the window and also how to manage actions such as left clicking and right clicking.

```
100  blackturn = pygame.transform.scale(pygame.image.load(os.path.join("Sprites","blacks-turn.png")), (200, 38))
101  whiteturn = pygame.transform.scale(pygame.image.load(os.path.join("Sprites","whites-turn.png")), (200, 38))
102  logo = pygame.transform.scale2x(pygame.image.load(os.path.join("Sprites","ChesspyLogo1.png")))
103  pygame.display.set_icon(logo)
104  win = pygame.display.set_mode((width,height))
105  pygame.display.set_caption("ChessPy")
106  white = (255, 255, 255)
107
```

Figure 11: Snippet of code used to define sprites and to create the window.

```
45
46      for k in pygame.event.get():
47
48          if k.type == pygame.QUIT:
49              quit()
50              pygame.quit()
51              break
52
53          if k.type == pygame.MOUSEMOTION:
54              pass
55
56          if k.type == pygame.MOUSEBUTTONDOWN:
57              click = pygame.mouse.get_pressed()
58              if click[0]:
59                  pes = pygame.mouse.get_pos()
```

Figure 12: Snippet of code to show event handling using Pygame.

II. Using the OS module

I used the OS module to interact with the operating system. This allows me to locate certain file locations which will be used for my sprites. I use the os.path.join(FolderName, Filename) to search for what I need.

```
import os

# Loading up all of the sprite images to use for display
chessboard = pygame.transform.scale(pygame.image.load((os.path.join("Sprites","Chess_board.jpg"))), (626,626))
chesspylogo = pygame.transform.scale(pygame.image.load(os.path.join("Sprites","ChesspyLogo1.png")), (160,120))
madeby = pygame.transform.scale(pygame.image.load(os.path.join("Sprites","madeby.png")), (260, 38))
woodboard = pygame.transform.scale(pygame.image.load((os.path.join("Sprites","woodboard.jpg"))), (626,626))

black_bishop = pygame.image.load(os.path.join("Sprites", "blackBishop.png"))
black_king = pygame.image.load(os.path.join("Sprites", "blackKing.png"))
black_knight = pygame.image.load(os.path.join("Sprites", "blackKnight.png"))
black_pawn = pygame.image.load(os.path.join("Sprites", "blackPawn.png"))
black_rook = pygame.image.load(os.path.join("Sprites", "blackRook.png"))
black_queen = pygame.image.load(os.path.join("Sprites", "blackQueen.png"))

white_bishop = pygame.image.load(os.path.join("Sprites", "whiteBishop.png"))
white_king = pygame.image.load(os.path.join("Sprites", "whiteKing.png"))
white_knight = pygame.image.load(os.path.join("Sprites", "whiteKnight.png"))
white_pawn = pygame.image.load(os.path.join("Sprites", "whitePawn.png"))
white_rook = pygame.image.load(os.path.join("Sprites", "whiteRook.png"))
white_queen = pygame.image.load(os.path.join("Sprites", "whiteQueen.png"))

```

Figure 13: Snippet of code to show loading sprites using OS module.

III. Error Handling and Debugging

One of the most frustrating parts when becoming a coder is having to deal with debugging your own program. I feel the same way too, as having to solve a problem that you caused can be quite confusing. I have devised my own way to figure out what went wrong in my code to lead to something not going the way I want to. The first method I did is writing down a print function where I expect things to be. If it is printed out on the console that means that I was right in thinking so, if not then something clearly went wrong and I will need to check elsewhere to solve the problem. Another thing I did was to use the try and except functions in python. These functions come very handy when having to deal with something that doesn't seem to be that big of a deal. I used multiple of these functions when some sort of error occurred that doesn't seem to be because of a logic error. Finally, if all things mentioned before didn't work, I would probably try to look for a solution online and if that still didn't work then I'd have to read the documentation of any external modules I was using for a full understanding of all of the functions included.

E. Evaluation

I. Does the program work properly?

The ChessPy program is essentially just a traditional chess game, that currently has nothing different to do with a normal game of chess. The program works really well as a game and has basically no errors that can occur that hasn't been dealt with. The program is also really fast with determining movement and other things as well. Overall, I would say that this program is good in terms of not being to complicated to understand its features as well its functionality.

II. Future Improvements that can be done

Before I decided on writing this documentation I was already working on three other parts of this program. I decided not to include them as I felt it was not needed and would also be really difficult to explain. The three additions I would like to add to this game would be a network functionality which allows for multiplayer between a host and another player, power up system which would add a lot more variety and differentiate this game from a normal chess game and also a TensorFlow based chess AI that you could go up against. Asides from those new functionalities the existing ones could also be improves such as a fullscreen function and better-looking graphics.

III. Reflection

I feel as if with this project I have grown stronger as a programmer and I wish that I can never stop growing my skills. With that being said I am satisfied with how my program ended up being, even though I know that there are many improvements that can still be done with the existing program. I will continue to work on this program after submitting this report and I will keep improving it so that it can be something that I am proud of. After all this is one my first python projects and I will remember the lessons, I learnt when making this all throughout my life.

F. Evidence of working program

I. Testing and pictures

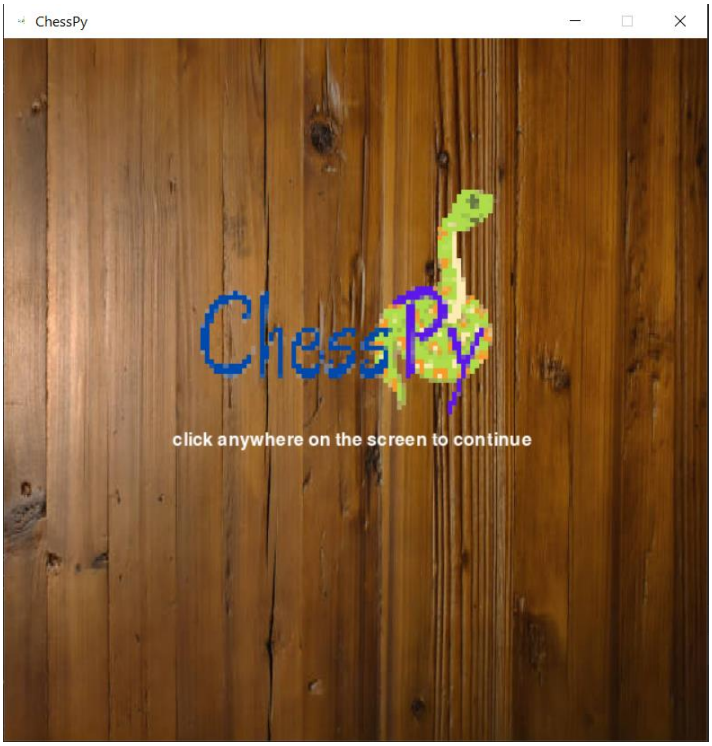

Figure 14: Starting screen of the program.

Figure 15: Main process of the game.

Figure 16: Showing pawns initial movements.

Figure 17: Pawns movements after initial movement.

Figure 18: Movement of knight.

Figure 19: Movement of the Queen.

Figure 20: Sprite when the king is checked.

Figure 21: Endgame screen when king is checkmated.