Python 3 Programming Project Report: Game development

1. Introduction:

In this project, I developed a *Ludo* game, which is a strategy board game played with two or four players, or one player against the computer. The game revolves around a random dice which determines players' moves.

This project requires creating a friendly user interface, to make the game accessible for players of all backgrounds. Also, this project requires designing a solid logic structure, because a *Ludo* game has many detailed rules for every action. A bad logic structure will result in an unplayable game. Additionally, classes and objects are required to create this project, because this game has different objects, which interact with each other under the rules of the game. Furthermore, defining classes with the appropriate properties and behaviours will diminish any bugs during the interactions, and thus increase the robustness of the game. Finally, this project requires constructing an algorithm that can play the game, to create a single-player mode.

In light of the described problems, I used GIMP (an image manipulation software) to create images for the user interface and used Pygame library to handle and render the graphics. Also, I created six classes to represent different objects and to handle various game mechanics. Lastly, I constructed a player algorithm.

2. Requirements:

When running the game on two different computers to test its reliability, I ran into several problems. Firstly, Pygame library was not installed by default with Python. Secondly, the directory I used in my code to access the game assets was not working properly. Thirdly, the game window was larger than the screen borders, which makes the game inaccessible. While troubleshooting these problems, I came to the following conclusions to ensure that the program will run appropriately:

- The computer must have Python 3 installed.
- After installing Python, Pygame package must be installed separately and updated.
- The client's display must be FHD (1920×1080 px) or higher.
- The client's system display settings must be adjusted so it doesn't scale up applications. To apply that in Windows 10, go to settings > system > display > scale, and then set the scale to 100%.
- When running the python script through the terminal, make sure that the terminal is in the same directory as the python script.

Also, I have observed the behaviour of the client when interacting with the user interface. First of all, since the client was a new Ludo player, he found it difficult to know which tokens were available. Thus, a sheet that explains the game rules is necessary. Also, to make it accessible for new players, I consider adding features, such as a token highlighter, or an interactive "how to play" screen. Secondly, the client thought that he should drag a token to move to its place. Hence, it could be helpful to introduce more than one type of input into the game. Finally, it was a little confusing for the client to see the tokens moving suddenly.

Therefore, including animations will help improve the accessibility and aesthetic of the game.

3. Design and implementation:

3.1 Implementation of four screens:

In order to make a friendly interface, I decided to make more than one screen for the game. Every screen needs to refresh every frame in its independent game loop. Therefore, I broke down my main file into four functions, each represents an independent screen (main menu, game, exit, game over), and they are called whenever appropriate.

3.2 Implementation of the game function:

I organized the game function into four parts. Each part is put in a logical order to run the game appropriately. Part #1 is outside the game loop, and it is responsible for loading the necessary game files, and for creating an instance of the game and the players. Inside the game loop, part #2 is responsible for checking the game status (e.g., whether the player finished their turn, the game ended). Part #3 is for updating the graphics on screen. Part #4 is for checking the client's input.

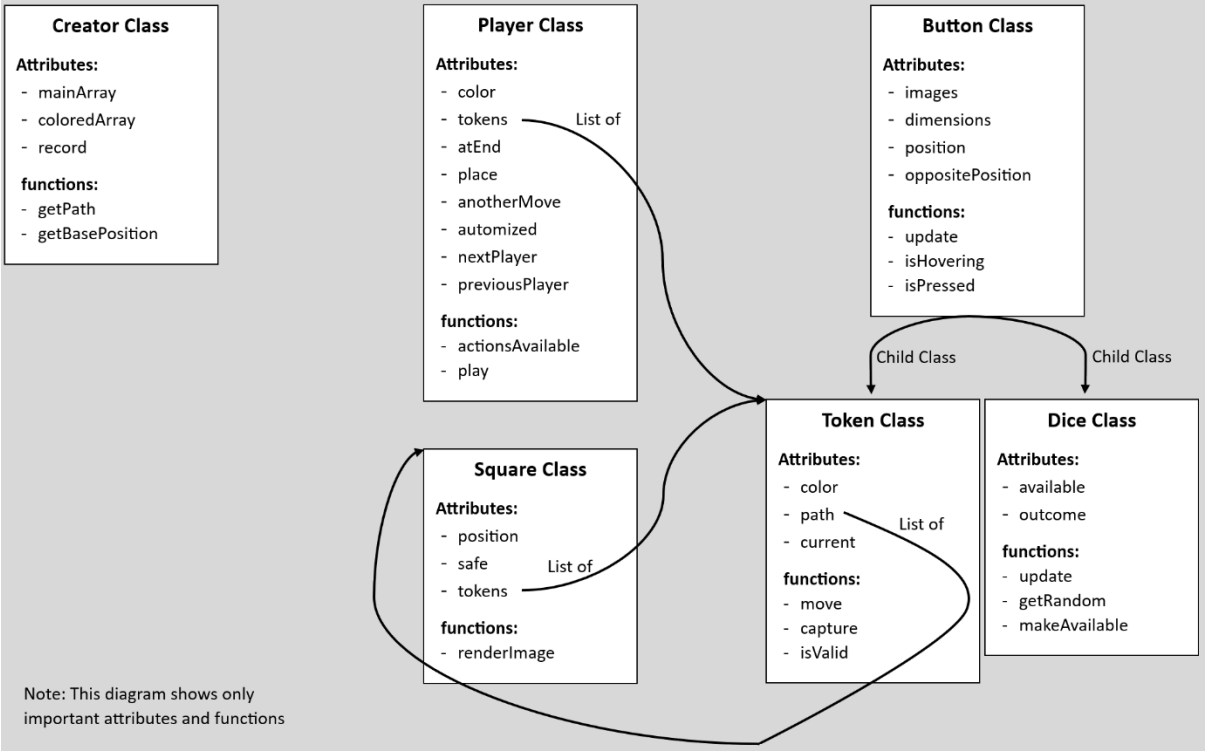
The logical order in the game loop is, the programme checks the game status and refreshes the screen accordingly and then waits for the client's input and so on. However, I didn't include any game validation or logic structures in the game loop, I included these in their respective classes. The reason for that is to make the main game loop stable and flexible for change.

3.3 Design of classes:

The main reason for choosing the class structure as the solution, is that classes organize different game objects. Also, modifying objects is safer than modifying variables in terms of unexpected behaviours and bugs, because the former method changes from the roots (e.g., if the player attribute "anotherMove" changes to false, the player cannot make a move until reset). Thus, it increases the robustness of the game.

I created six classes (as shown in the diagram below):

- Button, Token, Dice classes represent clickable objects.
- Dice class's unique features are producing a random outcome and storing it.
- Token class's unique features are having a path, which is a list of Square objects, and applying related game mechanics, such as moving and capturing.
- Square class represents every square on the board, and it is only responsible for rendering the token graphics.
- Player class represents the players with their appropriate properties. Also, it includes the automatic player algorithm.
- Creator class creates an instance of the game and helps initiate Token objects.
- Player, Token, Dice classes apply the game's logic structures and validation, to fulfil the game rules.



3.4 Debugging design issues:

At one time, I wrote my code so that dice availability and player availability ("anotherMove") are handled separately. And that caused some unwanted behaviour, such as infinite player moves per dice roll. So, I debugged my code and monitored the player and dice attributes. I came to understand that dice availability and player availability are always changing together, so I redesigned my code and created "makeAvailable" function in Dice class to handle both attributes.

Also, the game loop is used to update the screen before checking the game's status. The code was working properly until I made the single-player mode and the graphics stopped updating accurately. After debugging, I learned that the automatic player, unlike human players, finishes its turn in one iteration. That makes the game loop sensitive to the order of parts.

4. Programming techniques used

4.1 The game loop

To create a game, I needed the programme to run continuously and do several checks at a specific rate. To achieve that, I implemented a game loop, which iterates at the rate of 60 times a second using a built-in method in Pygame [1].

```
#to apply all changes on screen and refresh the window
pygame.display.flip()

#to restrict the refresh rate to 60 frames/second
clock.tick(60)
```

4.2 Condition statements

I have used condition statements in my code for different purposes. One of the major implementations of it occurs in the main game loop, and it is used to check the status of the game.

```
##### part (2): check game status #####

#in the linked list, if a player is refering to themselves, that means they are the last one
#and they lose and game ends
if currentPlayer == currentPlayer.nextPlayer:
    pygame.time.wait(1000)
    gameOver()
    return

#if the current player has no more actions to make and the dice is not available, that means their turn ends
if not currentPlayer.actionsAvailable(dice.outcome) and not dice.available:
    currentPlayer = currentPlayer.nextPlayer #go to next player's turn
    dice.makeAvailable(currentPlayer) #reset dice for the current player
```

4.3 Iterable objects

Python has many iterable objects, and I used three of them. I have used tuples to represent coordinates of fixed objects, because tuples are immutable and they preserve order. Also, I have used lists and dictionaries when appropriate. Finally, a square can display a token in 32 ways, so I organized its image data in a 3D array (axes are shown below). That enabled the programme to access data conventionally.


```
#therefore, if different colors coexist, we must render a small image of the tokens
if self.safe and len(track.keys()) > 1:
    for color, quantity in track.items():
        screen.blit(gameSetup.images[color][quantity - 1][1], self.position)

#otherwise, render the regular size
else:
    for color, quantity in track.items():
        screen.blit(gameSetup.images[color][quantity - 1][0], self.position)
```

4.4 Validation

This game checks for user input 60 times a second. At each time, the user input must be validated through different functions, such as functions to validate tokens, the dice and the players' eligibility.

```
def isValid(self, diceValue):

    if diceValue == 0:
        return False

    #if it is at the begining and the dice value is not 6
    if self.current == 0 and diceValue != 6:
        return False

    #if the dice value is greater than the remaining path
    if (57 - self.current) < diceValue:
        return False

    return True
```

4.5 Functions

I used functions widely in the solution (e.g., move, capture and getRandom). I used them to minimize repetition in the code and to break down the solution into different roles, which I can modify and debug easily, without the need to check the entirety of the code.

4.6 Sorting

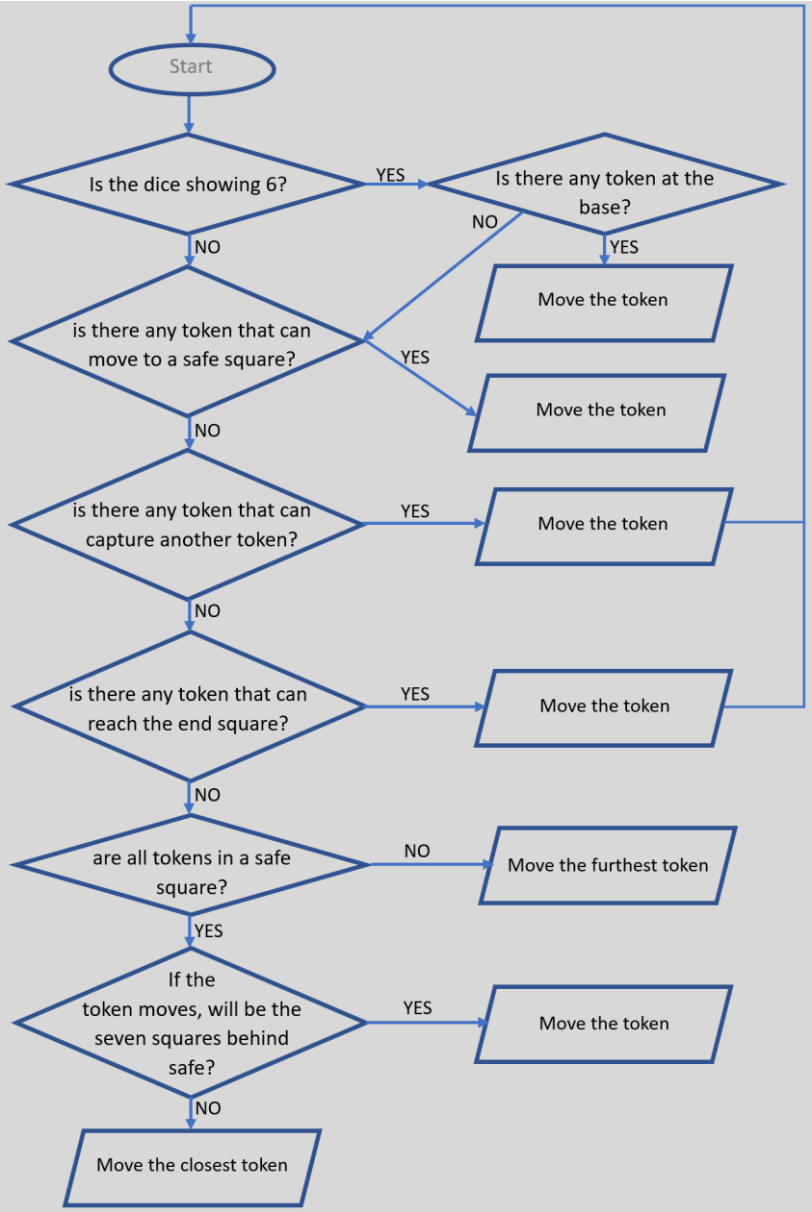
I implemented merge sort (recursive) in my solution, which has a time complexity of O(n*log(n)) [2]. Since my code always sorts lists of four elements (in Player class), this time complexity is more viable.

4.7 Pygame and Random libraries

I used Pygame mainly to render graphics, manage refresh rate and to check for user input. A random object is necessary in a game of chance, so I implemented that in Dice class using the Random library.

4.8 The automatic player algorithm

My solution features an automatic player (in Player class). *Ludo* is not only a game of chance, but also a strategy game. Therefore, implementing an automatic player requires an algorithm. The algorithm flowchart is shown below.



4.9 Circular doubly linked lists

In my solution, I needed to traverse from one player to the next one with the ability to remove a player during the game (because the game continues after the first winners). To apply that, I created a circular linked list of players, where the last player refers to the first one, so the game continues. However, I upgraded the list so every player can refer to the previous player as well, which enables that player to remove their reference from both directions [3].

```
#remove the references to the current player from neighbour players on the list
player.previousPlayer.nextPlayer = player.nextPlayer
player.nextPlayer.previousPlayer = player.previousPlayer
```

4.10 Transforming between data types

One of Python's features is called duck typing, which allows data to be interpreted differently depending on the context [4]. For example, in the snippet below, you can see that I used a list as a boolean expression. Empty lists are interpreted as "False" in this context.

Furthermore, this snippet of code cannot raise the "index out of range" exception, even though it looks like it. That is because if the list was empty in the second condition, the result of the expression would be "True", and the python executer would not need to verify the third expression.

```
if self.path[self.current].safe or not self.path[self.current].tokens or self.path[self.current].tokens[0].color == self.color:
    return captured
```

5. Instructions:

Follow these instructions to run the game:

- Make sure your computer meets all the requirements listed earlier.
- Go to "GameAssets" folder, and extract the compressed file into the same directory (so the images are stored directly under "GameAssets").
- Go back to "LudoWarwick" folder, and open the terminal from there, and then type `py main.py`. Alternatively, you can open the folder in an IDE, and run "main.py".

If the game window is larger than your screen, try this solution:

- Open "main.py" in a text editor. In line 9, remove the comment tag.

6. Test and maintenance:

First of all, to check the quality of the user interface, I tested the Button class, which creates buttons and tokens and a dice. To test it:

- Hover the pointer around the button, then on top of it, then away from it.
- Hover the pointer on top of the button and hold and press on it, move away from it and release the click.

Secondly, it is important to test the logic structure, otherwise the game is unplayable. To test it:

- Click on invalid tokens (including opponent's)
- Roll the dice twice.
- Capture own tokens.
- Capture in a safe square.
- Click on valid tokens.
- Verify the rules for rolling the dice twice.

Finally, I tested the automatic player by playing against it. I compared its moves with the algorithm diagram, until I verified every branch of the diagram. Furthermore, the algorithm

can be tested by changing line 27 in “player.py” to `self.automized = True`, which makes all players automatic by default, and then playing normal game mode.

However, the current design does not support dragging tokens nor animations. Instead, the program creates pauses to simulate animations, which causes the game to be irresponsive. Although this does not cause the game to fail, it makes it feel less robust. Also, the lack of a structure for animation restricts new features from being introduced to the game in the future. To solve the problem, it is suggested to create a class for handling animations and rewrite all code in coordination with it.

7. Reflection and next steps:

In this project, I used the rapid application development (RAD) method, since I did not clearly identify the objectives of the project from the beginning. Instead, I had a few ideas and a vague overall design. Afterwards, I followed the RAD’s cycle of coding, refining, testing and implementing to build my project incrementally.

Using the RAD method, I was able to finish the project on time, and managed to add new features along the way. However, the RAD method resulted in some incapabilities in my solution due to design issues, where it was too late to redesign the entirety of the project. For example, animations could not be implemented properly in the current design. However, the RAD method could be a viable option for my next project only if an early comprehensive design were present.

8. References and open-source declarations:

[1] “Pygame Front Page — pygame v2.0.0.dev15 documentation,” *pygame.org*. https://www.pygame.org/docs/

[2] “Analysis of merge sort (article),” *Khan Academy*. https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort

[3] “Insertion in Doubly Circular Linked List,” *GeeksforGeeks*, Jan. 17, 2023. https://www.geeksforgeeks.org/insertion-in-doubly-circular-linked-list/

[4] “Glossary — Python 3.10.2 documentation,” *python.org*, Apr. 23, 2023. https://docs.python.org/3/glossary.html#term-duck-typing