**Computer Mathematics and Declarative Programming**

**Interactive Tic Tac Toe Command Line Game**

**S18155900**

Table of Contents

[TASK 1 (Designing and Planning) 3](#_Toc141274011)

[User Identification: 3](#_Toc141274012)

[User Interaction: 3](#_Toc141274013)

[Data Model: 9](#_Toc141274014)

[Behaviours based on Mathematical Relations, Mathematical Functions, and Graph Theory: 11](#_Toc141274015)

[Task 2 (Implementation) 13](#_Toc141274016)

[Behavior Step Implementations 13](#_Toc141274017)

[Task 3 (Testing) 16](#_Toc141274018)

[Manual Testing 16](#_Toc141274019)

[1-Player Mode Tests 16](#_Toc141274020)

[2 Player Testing 18](#_Toc141274021)

[Unit Testing 20](#_Toc141274022)

[Automated Testing 23](#_Toc141274023)

[TASK 4(Version Control) 26](#_Toc141274024)

[References 28](#_Toc141274025)

# TASK 1 (Designing and Planning)

In this section of the logbook, I will be talking about the design and planning stages of the creation of my tictactoe program. This section is mainly split into 4 sections as seen below as we dive into the creation process, focusing on the user and how they’ll interact with it, how gherkin specifications based in Hoare logic provide a guide for users and how their inputs work with the program and then user behaviours.

## User Identification:

The primary user(s) of this application would be anyone between the ages of 5-99 truly, Tic Tac Toe is a game for all ages that lets one to let loose and while away time while having fun. This game is purely for entertainment. Using the command line interface, the user(s) will have to enter in commands/inputs manually, turn by turn 9in the case of 2 player games).

## User Interaction:

The Tic Tac Toe game can be designed with the following user interactions:

a) Start the game: The user should be able to start a new game by running the program.

b) Choose the game mode: The user should be prompted to select the game mode, either 2-player or single-player mode.

c) Enter player names: The user should be prompted to enter the names of both players in 2-player mode.

d) Display the game board: The program should display the current state of the game board after each move.

e) Prompt the player for their move: The current player should be prompted to enter their move by specifying the row and column of the cell they want to place their token.

f) Check for a winner: After each move, the program should check if the game has been won by either player.

g) End the game: The game should end when a player has won or if there are no more moves left to make.

**Gherkin Specifications:**

To specify these interactions using Gherkin specifications based on Hoare logic, we can use the following examples:

1. Feature: Main Menu Navigation

As a command line game player, I want to navigate the main menu of the Tic Tac Toe program, So I can select a relevant game mode

**Scenario:** User selects a single player game option from the main menu.

Given that the main menu is displayed via the stdout (standard output)

And I have provided the input “1” via the stdin (standard input)

When I confirm my choice by pressing the Enter/Return Key

Then the game will transfer to the “Start single player game”.

**Scenario:** User selects a single player game option from the main menu.

Given that the main menu<main menu> is displayed via the stdout (standard output)

And I have provided the input “2” via the stdin (standard input)

When I confirm my choice by pressing the Enter/Return Key

Then the game will transfer to the “Start two player game”.

The first user interaction is described in this scenario, when the programme has begun, and the user is presented with the main menu where they have to select the game mode. It gives a precise description of the user interaction required and guarantees that the programme will properly handle the user's input which could either be ‘1’ for single player or ‘2’ for multiplayer.

These scenarios are specified to make sure the user is presented with the options available to them (1 for single player mode or 2 for multiplayer mode). This is a very vital part of the program as the user cannot play the game without it.

1. Feature: Making a move in-game

As a command line game player,

I want to be able to make a move on the Tic Tac Toe board,

so that I can play the game.

**Scenario**: User makes a valid move on the Tic Tac Toe board.

**Given** that a game board <game board> is displayed via the stdout (standard output)

And it is currently player <player 1>'s turn to make a move

And I have provided the input <coordinate> via the stdin (standard input) When I confirm my move by pressing the Enter/Return Key

**Then** the game board should be updated with my move <move>

And it should be player <next player>'s turn to make a move.

**Scenario**: It is the AI's turn to make a move (Single player mode)

**Given** it's AI's turn after the user has made a valid move on the game board and the game board has been updated accordingly.

**And** it is the AI’s turn.

**When** the AI calculates and selects a valid move

**Then** the program updates the board accordingly

**Scenario**: User makes an invalid move on the Tic Tac Toe board. **Given** that a game board <game board> is displayed via the stdout (standard output)

And it is currently player <player>'s turn to make a move

And I have provided an invalid input <invalid coordinate> via the stdin (standard input)

**When** I confirm my move by pressing the Enter/Return Key

**Then** an error message <error message> should be displayed via the stdout (standard output)

And it should still be player <player>'s turn to make a move

And the game board should remain the same until a valid move is made/coordinate is input.

These scenarios describe the interaction between a player and the program when both a valid and invalid move has been made. In the case of an invalid move being made, it prompts the user to make a move in a valid row/column and once that has been done, the program will update the game board accordingly.

The second scenario highlights the interaction between the computer and the game board once it is its turn to make a move (when playing in single player mode)

These scenarios are specified in this manner to show the importance of valid input from players (and AI). Without a valid move being made, the game cannot progress. By specifying the “invalid move” scenario, we have shown that the program handles such errors correctly and asks the user to make a valid move.

1. Feature: Restart the game

As a command line game player, I want to be able to restart the game, so that I can play again.

**Scenario**: User restarts the game.

Given that the Tic Tac Toe game is currently in progress

And I have provided the input "r" via the stdin (standard input)

When I confirm my decision to restart by pressing the Enter/Return Key

Then the game board should be reset to its initial state <initial state> And player <starting player> should start the game.

1. Feature: Winner declared

As a command line game player, I want the program to detect when a player has won the game, so that the game can end, and the winner can be declared.

**Scenario**: Player wins the game.

Given that a game board <game board> is displayed via the stdout (standard output)

And it is currently player <player>'s turn to make a move

And player <player> has made a move that resulted in a winning game state

When the program checks the game state

Then the program should declare player <player> as the winner via the stdout (standard output)

And the program should end the game

**Scenario**: Computer wins the game.

Given that a game board <game board> is displayed via the stdout (standard output)

And it is currently CPU 's turn to make a move

And CPU has made a move that resulted in a winning game state

When the program checks the game state

Then the program should declare CPU as the winner via the stdout (standard output)

And the program should end the game

These scenarios specify the conditions for a winner to be announced, be it a human user or the computer (AI), the program should show the final game board and announce the results.

These scenarios have been specified this way to make sure and verify the program can recognize a winning move made either by a user or the AI and thus announce the correct winner.

1. Feature: Invalid move made.

As a command line game player, I want the program to detect when an invalid move has been made, so that the player can be informed and prompted to make a valid move.

**Scenario**: Player makes an invalid move.

Given that a game board <game board> is displayed via the stdout (standard output)

And it is currently player <player>'s turn to make a move

When the player inputs an invalid move <invalid move> via the stdin (standard input)

Then the program should display an error message via the stdout (standard output)

And the program should prompt the player to input a valid move

And it should still be player <player>'s turn to make a move.

1. Feature: Tie Game

As a command line game player, I want the program to detect when a game ends in a tie, so that the game can end, and the result can be declared.

**Scenario**: Game ends in a tie.

Given that a game board <game board> is displayed via the stdout (standard output)

And there are no more valid moves that can be made

When the program checks the game state

Then the program should declare the game as a tie via the stdout (standard output)

And the program should end the game

And the program displays the option to restart the game

1. Feature: Exit Game

As a command line game player, I want to be able to exit the game at any time, so that I can quit playing.

**Scenario**: User exits the game.

Given that the Tic Tac Toe game is currently in progress

And I have provided the input "q" via the stdin (standard input) When I confirm my decision to quit by pressing the Enter/Return Key Then the program should display a goodbye message via the stdout (standard output)

And the program should terminate.

All the specifications above have been provided to cover the most essential interactions between the user and program to make sure the game functions as it should.

By adding these gherkin specifications based on Hoare logic to my program, I believe I have provided the appropriate and clear guidelines for the expected user inputs/interactions and their results. This ensures that the program behaves as intended and gives the user the best experience possible and helps with testing the games functionality by serving as test cases for different scenarios.

## Data Model:

To represent the data for the Tic Tac Toe game, we can use the following data model:

1. Game Board: A 3x3 grid to represent the Tic Tac Toe game board. Each cell will have one of three values: ‘X’, ‘O’, ‘’(null/empty). The game board will be represented by a triple tuple (updated because my first attempt was to use a lists of lists), where each element/move will be one cell on the board. Its represented as so:

[

[(0, 0, ' '), (0, 1, ' '), (0, 2, ' ')],

[(1, 0, ' '), (1, 1, ' '), (1, 2, ' ')],

[(2, 0, ' '), (2, 1, ' '), (2, 2, ' ')]

]

Each cell in the 2D list represents a position on the Tic-Tac-Toe board. The row index (0, 1, or 2) and column index (0, 1, or 2) identify the position of the cell on the board. The symbol ' ' represents an empty cell, 'X' represents a move made by player X, and 'O' represents a move made by player O. In Set Theory, the definition would be Board= {(row, col, player) | 0 <= row, col <= 2, player ∈ Players}. The appropriate python definition would be Board= Tuple [int, int Players]

To view the game board at any point in the game, we have a function called ‘print\_board(board)’. This data model allows for the clear representation of a tictactoe game board (grid), making it easy to understand and update during the game, by analysing the placements and symbols in the cells, it also makes it easier to check for winning moves and determine whether the board is full (resulting in a draw).

The game board consists of nine cells and the state of each cell can either be classified as occupied or unoccupied. An occupied cell must contain the symbol either ‘x’ or ‘o’ for the player that made a move in said cell. In python we would use enumeration to define the data types as (1) for occupied and (2) for unoccupied. In Set theory, it would be defined as

Cell\_State = {occupied, unoccupied}.

1. Player: A player object that contains the player's name and token (either X or O).

We have two participants in the game: Player X and Player O. In Set theory, Everyone in the game takes turns, and we can visualise this as a group of players: "Player X" and "Player O", Players= {X, O}.

Since we'll use a straightforward integer variable to monitor who is taking whose turn, We can give Player X (1) and Player O (2).In python we would use enumeration for this, so it would be represented as:

Class Players(enum):

X=1

O=2

1. Move: We must examine the game board's rows, columns, and diagonals in order to determine whether a player has won. These winning combinations can be viewed as collections of cells that make a successful pattern. A player wins if his or her symbols ('X' or 'O') appear in every cell of a winning combination. In Set theory this would be defined as

Winning\_Combinations = { {(r1, c1), (r2, c2), (r3, c3)} | 0 <= r1, c1, r2, c2, r3, c3 <= 2 }.

The python definition/representation would be Winning\_Combinations = Set[Tuple[int, int]].

We would be using a set of tuples to define the winning\_combinations data type.

## Behaviours based on Mathematical Relations, Mathematical Functions, and Graph Theory:

Our game behaviours are like the actions the game takes based on different inputs and states. In this section, I will list out the game behaviours and their definitions.

1. **Valid move during turn:**

A valid move on the board during the users turn is to integers, representing the rc (row and column) indices on the game board/grid, (0 <= r, c <= 2) where a player can place their symbol (X or O).

1. **Winning move:**

A winning move on the board is when either a user or the AI places their symbol (X or O) in a cell that forms a winning combination with other cells either horizontally, vertically or diagonally on the game board/grid.

1. **Draw:**

A draw is when all cells on the game board/grid are occupied without any winning combinations being present. This results in a draw/tie between the 2 players/player and AI and thus the game ends.

1. **Board State:**

The Board state represent the current layout of the game board/grid. The board is represented as a triple tuple ( T, T, T) where each element of T is a triple tuple (r, c, s) to represent the symbols (‘’, x, o) placed at positions (r, c) (r stands for row, column is represented by c)

1. **Player Turn:**

The player turn refers to when a player either represented by X or O can make a move on the board. A valid move would be the appropriate symbol placed in a cell that is unoccupied.

1. **AI Move:**

An AI move would be in the single player mode where the user has made a valid move, and then the AI would make a calculation based on its algorithm to try and claim a win against the user.

1. **Game Result:**

The game outcome is determined when one of the players achieves a winning move or when the board is full and no one (either player or AI in the case of single player mode) has won (draw). The game ends, and the result is displayed and thus the game ends.

1. **Game Restart:**

After the game has ended, the players have the option to restart and play again by selecting a new game mode or continuing in the same mode.

1. **Graph Representation:**

The TicTacToe board can be represented as a graph where each cell is a node, and there are edges connecting adjacent cells. Players' moves correspond to the traversal of these edges, forming paths in the graph.

1. **Graph Connectivity:**

The graph representation of the TicTacToe board is a connected graph. Every cell is reachable from any other cell through a sequence of valid moves made by the players.

These mathematical and graph theory-based definitions help to formalize and understand the behaviors of the Tic-Tac-Toe program. They provide clear definitions of important elements, such as valid moves, winning conditions, and board states. Additionally, graph theory concepts illustrate the structure of the Tic-Tac-Toe board as a connected graph, where player moves create paths through the graph.

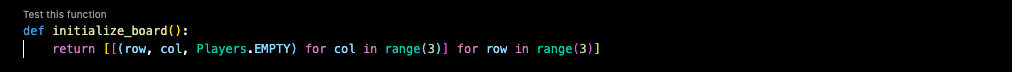
By using these mathematical definitions, we gain insights into the program's behavior and interactions, making it easier to reason about the game's mechanics and properties. Furthermore, these formal definitions can be used to verify the correctness and integrity of the program's logic and facilitate testing and analysis.

# Task 2 (Implementation)

This section will talk about the the implementation of the program. I chose Python for this project because honestly it seemed the easier language out of all the choices, even though I still stand by this decision, it has been a big struggle to make this. This

## Behavior Step Implementations

1.Function to initialize an empty game board



With this line of code, I have created a function that basically prints a 3x3 game grid/board with each cell being a triple tuple (r,c,p) (rows,columns,players.empty). row and column are the row and column indices of each cell while the “Players.empty” is an enumeration representing an empty cell. This function is pure and total because it does not rely on any external input and successfully returns a 3x3 game board/grid without any exceptions each time it is called.It also avoids any mutability as it does not modify any external data structures.

2. Function for checking a winning move

A black background with colorful text

Description automatically generated

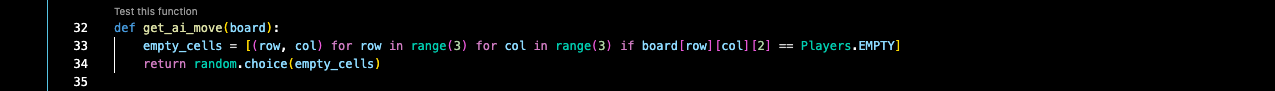
With these lines of code, I have created a function that checks the game board/grid for any winning combinations. This function is pure and total because it does not rely on any external input and only returns a Boolean value (True if there is a winning combination or False if a win has not yet been achieved by either the player(s) or AI.It also avoids any mutability as it does not modify any external data structures and only reads the state of the board.

3.Function to check if board is full



With these lines of code, I have created a function that checks the game board if its full and no more moves can be made. This function is pure and total because it does not rely on any external input and only returns a Boolean value (True if the game board/grid is full or False if the board still has empty cells). It also avoids any mutability as it does not modify any external data structures and only reads the state of the board.

4.Function to get AI move



This function gets the AI’s valid move after its calculation. It uses list comprehension to assess all empty/unoccupied cells and places its token. This function is impure as it relies on a random number generator which leads to a side effect (the ai making a move).

The “get\_ai\_move” function takes the “board” as an input and then creates a list of empty cells available for the AI’s token, then randomly selecting one to make a move on. This function is also a total function because as long as there’s an empty cell, the AI will return a valid move.  
There is some no mutability in this function since it only reads the board to find empty cells then makes a list of said empty cells for the AI to make its valid move during its turn.

5. Main Function to play the game

A screen shot of a computer

Description automatically generated

This is the main function that runs the game loop and coordinates the game. it’s a loop where players take turns (or the player and AI go back and forth) placing tokens on the game board/grid until it is a draw or someone has won. Enumeration in python Is used to represent the players. The game grid/board is made with a triple tuple. These programming techniques are used because we are able to have a more structured codebase and also concise and easy to understand code all in an elegant manner. Most of this code focuses on the single player mode when you are playing versus the AI, if you want to play 2 player mode, all I do is on the last line as seen above in the bracket you type “play\_tic\_tac\_toe(ai\_mode=False)” and then save it and run as usual.

The “play\_tic\_tac\_toe” function contains a mix of pure and impure elements. It mostly operates on data from the board (and its state) and the “current\_player” be it a user or the AI is determined by “players” function. It also interacts with the command line interface through the print() and input() , which makes its an impure function since it has side effects outside of the main function. It’s also through the interaction with the command line interface through (print() and input() ) where “input()” having the possibility of errors for example with the entering of a non-integer input. This means the “play\_tic\_tac\_toe” function is not a total function.

There is mutability in this function because it updates board once a player/AI has made a valid move, but it is justified since the board needs to be updated after every valid move for the game itself to progress and a winner/draw be declared.

# Task 3 (Testing)

## Manual Testing

Manual testing is the process of manually testing out sections of my code to find any errors/vulnerabilities. This form of testing is done without any automated tools/extensions.

### 1-Player Mode Tests

1. Test by inputting a valid integer/coordinate :

A black rectangular object with a brown border

Description automatically generated

In this test, I wanted to make sure if a valid move was made against AI, the program would accept it and it would switch to the AI’s Turn.As seen in the screenshot above, it placed my token (X) in the right cell. This test was a **PASS.**

1. Test by inputting an invalid integer/coordinate:

A screenshot of a computer screen

Description automatically generated

In this test, I wanted to try inputting an invalid coordinate (1,5) against AI, the program ended up crashing as it was out of the lists range and was not acceptable.As seen in the screenshot above, my program crashed. This test was a **FAIL.**

1. Test to place my coordinate in an already occupied cell:

A screenshot of a computer

Description automatically generated

In this test, I wanted to make sure if a valid move was made against AI but in an already occupied cell, an error message would pop up and it would tell me to choose another cell to place my token in.As seen in the screenshot above, it did exactly that and it is still my turn. This test was a **PASS.**

1. Test to input a letter into one/both coordinates instead of a valid integer coordinate:A screenshot of a computer

   Description automatically generated

In this test, I wanted to try inputting an invalid letter coordinate (a b) against AI, the program ended up crashing as it was out of the lists range and was not acceptable.As seen in the screenshot above, my program crashed. This test was a **FAIL.**

### 2 Player Testing

1. Testing to see if both players can input valid coordinates

A screenshot of a computer

Description automatically generated

In this test, I wanted to make sure if a valid move was made against the second player, the program would accept it and it would switch to the Second players turn and they could make a valid move as well.As seen in the screenshot above, it placed my token (X) in the right cell and did the same for the second player. This test was a **PASS.**

1. **Test to see if Player 1 can place token on already occupied cell:**

**A screenshot of a computer

Description automatically generated**

In this test, I wanted to make sure player 2 could not place their token in already occupied cell, an error message would pop up and it would tell them to choose another cell to place my token in. As seen in the screenshot above, it did exactly that and it is still their turn. This test was a **PASS.**

1. Testing to see one player input correct coordinates and the second player inputs alphanumeric characters

A screenshot of a computer

Description automatically generated

In this test, I wanted to try inputting an invalid letter coordinate (0 b) against Player 1, the program ended up crashing as it was out of the lists range and was not acceptable.As seen in the screenshot above, my program crashed. This test was a **FAIL.**

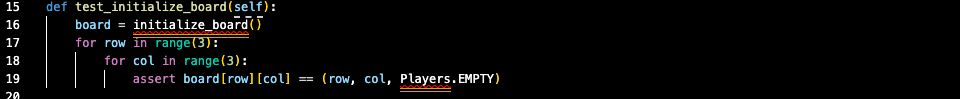
## Unit Testing

In this section, I used the pytest framework to show that individual parts of my code were working as intended.

1.A screen shot of a computer program

Description automatically generated

This test checks that the when the initialise\_board() function is called, it correctly returns a 3x3 Game board/grid (list of tuples).

2.

This test checks that the initialize\_board function returns a board with tuples containing the correct values for row, col, and Players.EMPTY.

3.A screen shot of a computer program

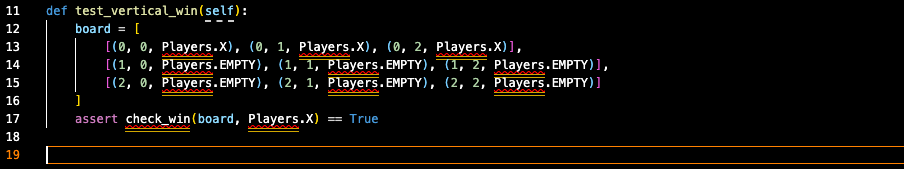
Description automatically generated

This test checks that the print board function correctly prints the game board/grid with all cells empty.

4.A screen shot of a computer program

Description automatically generated

This tests checks that the check\_win function correctly identifies a winning horizontal pattern from the player/AI.

5.

This test checks that the check\_win function correctly identifies a winning vertical pattern from the player/AI.

6.A screen shot of a computer

Description automatically generated

This test checks that the check\_win function correctly identifies a winning diagonal pattern from the player/AI.

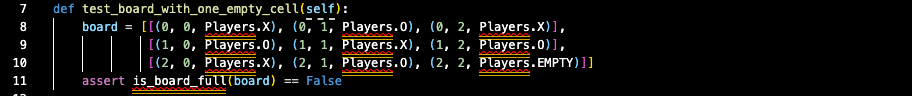
7.A screen shot of a computer

Description automatically generated

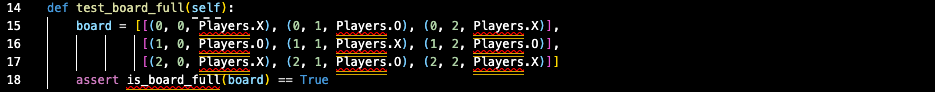
This test checks that the check\_win function correctly identifies when the player/AI does not win and thus returns a False.

8.

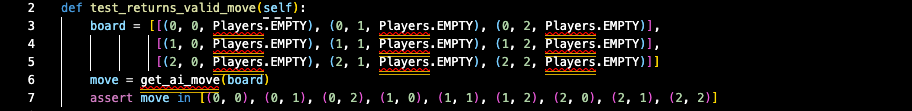
This test checks that an empty game board/grid returns True in regards to the is\_board\_full() function

9.

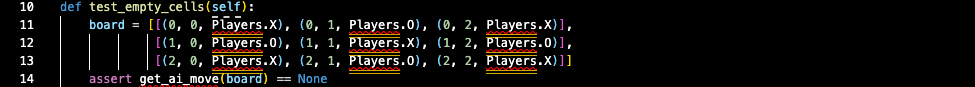
This test checks that one empty board cell/grid returns False in regards to the is\_board\_full() function.

10.

This test checks that the is\_board\_full() function returns True when all board cells are occupied.

11.

This test checks that the get\_ai\_move() function returns a valid move when there are empty cells on the board.

12.

This test checks that the get\_ai\_move() function returns None when there are no empty cells on the board.

## Automated Testing

When it comes to automated testing, This part for me was quite tricky. From the research I did, Automated testing was said to be a very quick way of testing different sets of user input of various levels, with it being able to handle a huge number of tests. The tests I wrote out are using user behaviours to test the code.

I first had to make a “.feature” file using the gherkin specifications I made earlier as a guide.

A screenshot of a computer program

Description automatically generated

Using these specifications and others, I then had to make a”.py” file with code that follows the scenarios set out to test the program. It was tough at first because I didn’t understand the concept of BDD. frameworks because I had only ever heard of TDD. Even though this is automated testing which is supposed to be quicker in comparison to manual testing, BDD is very time consuming to create. In the end I was not able to test it all out as my system (MacBook) ran into problems and was not being compliant. The following is my test code.

A screenshot of a computer screen

Description automatically generatedA screen shot of a computer program

Description automatically generated

# TASK 4(Version Control)

Version control systems like Git, are a huge part of managing software projects with multiple personnel contributing to it. It offers so many benefits and perks that enhance the teamwork, streamline development of the project.

1.Collaboration and Concurrent Development:

Version control systems like Git enable collaborators to work on said project all at the same time. In Git each collaborator creates their own branch of code to work on.

2.Tracking Change History:

Git keeps a history of all changes made to the code and who effected those changes, and at what time. This helps to track evolution of code and provides accountability.

3.Branching: Branching in Git allows developers to create isolated environments for developing new features or fixing bugs. This feature management approach enables seamless integration of new features into the main project once they are complete and tested.

4.Code Review and Quality Assurance:

Git clears the way for code reviews, allowing other team members to analyse and criticise your code, before merging your branch with the main branch.

5.Merging Conflicts:

Git has tools that handle conflicts that may arise from the merging of different branches.This all allows for the main codebase to remain functional.

6.Backup Recovery:

Due to Git having remote repositories, this helps prevent loss of data providing an additional level of backup.

7.Remote Collaboration:

Git allows for collaborators from different areas and time zones to work together due to its remote collaboration. Additionally, remote repositories hosted on platforms like GitHub offer a secure and centralized place to store the project's code and facilitate collaboration.

If I were to apply Git to the previous development tasks for the Tic-Tac-Toe project, I would leverage its version control capabilities to enhance collaboration, organization, and code management throughout the development process. Here's how I would utilize Git in different aspects of the project:

1.Making a repository:

I would make a repository to hold all my different files.It would be the main place for working on my code.

2. Branches:

Id use branches where I would work on each new feature of the game. For example, I might have separate branches for setting up the rules of the game, and adding the ability for the computer to play.

3.Review Code:

Before merging any of the branches with the main branch, I would use some of its tools for reviewing the code to make sure the code is good quality and follows the specifications I wrote out.

4.Automated testing: Since this was an issue for me, I know if I had employed Git it would have made the work a whole lot easier, I would use GitHub Actions or GitLab CI/CD pipelines. This means id be able to run tests automatically with the addition of new features or changes made, so as to make sure the code is always fully functional and follows the rules set out.

# References

Knight, A. *et al.* (2021) *BDD 101: Writing good gherkin*, *Automation Panda*. Available at: https://automationpanda.com/2017/01/30/bdd-101-writing-good-gherkin/ (Accessed: 10 June 2023).

*Loops in python* (2023) *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/loops-in-python/ (Accessed: 15 June 2023).

Pandey, A. (2022) *Beginner’s Guide for testing in Python*, *Astaqc Consulting*. Available at: https://www.astaqc.com/software-testing-blog/beginners-guide-for-testing-in-python#:~:text=Automated%20vs.&text=Beginner%27s%20guide%20for%20Testing%20in%20Python%2C%20When%20the%20user%20checks,complete%20set%20of%20manual%20tests. (Accessed: 15 July 2023).

Ojigbo, R. (2019) *Git basics - an idiot’s guide.*, *Medium*. Available at: https://medium.com/@rukeeojigbo/git-basics-an-idiots-guide-99445a56bd65 (Accessed: 16 July 2023).