Assessment task 2

OOP and Version Control

Project Proposal & Report

Tic Tac Toe

Clarity of Project Domain

This project falls within the domain of interactive digital Games, with a specific focus on recreating the timeless game of TIc Tac Toe in a digital format. This project will be implemented using python, utilising its ability to create a user-friendly interactive game. Tic Tac Toe serves as more than just a simple game, it is a fun tool that can help teach and explore a variety of concepts across different fields, however, the traditional games such as, rock paper scissors, battleships, connect four and Tic Tac Toe can become monotonous and lacks the engaging elements that can make it a more compelling experience. The goal of this project addresses the challenge of a boring generic game and innovates that concept by providing a more exciting and stimulating experience for users. This will be achieved through adding a wider variety of aspects to the basic game, including a 1 player mode against AI and an aesthetic visual interface for the player to enjoy. These additional features can enhance the user experience making a more pleasant experience for anyone enjoying the game.

Class and Object Design

Relevant classes and objects are identified. Some of the main classes in the code are

1. Board (manages the game board its size and grid)
2. Player(Represents the player in the game)
3. Game(manages the functionality of the overall game logic)
4. UI(handles the graphical user interface using pygame)

Board Class:

- Attributes:
  - ***size***: Represents the dimensions of the game board.
  - ***grid***: Represents the state that each cell is in, containing either the player mark(X/O) or be empty.
- Behaviours:
  - ***initialise_board(size)****:* Initialises the game board with the specified dimensions.
  - ***display_board():*** Displays the current state of the game board, including player marks and empty cells.
  - ***check_win_combination(player)****:* Checks if the specified player has achieved a winning combination on the board.
  - ***is_full()***: Checks if the game board is completely filled with player marks, indicating a draw.
  - ***update_board(move, player)****:* Updates the board with the player's move, marking the corresponding cell with the player's symbol.

Player Class

- Attributes
    - ***Name****:* Represents the name of the player participating in the game.
    - ***Symbol***: Represents the symbol (e.g., 'X' or 'O') chosen by the player to mark their moves on the game board.
    - **Type**:Represents the type of player(“human” or “AI”)
- Behaviours
    - ***make_move(board, move)***: Allows the player to pick a spot to mark by specifying the cell.
    - **player_type():**For HumanPlayer, these behaviours handle user input for making moves and symbol selection. For AIPlayer, they implement AI-specific logic for making moves based on the chosen difficulty level.

Players Subclass

HumanPlayer

- Attributes:
    - **name**: Represents the name of the player participating in the game.
    - **symbol**: Represents the symbol (e.g., 'X' or 'O') chosen by the player to mark their moves on the game board.
    - **player_type**: Fixed to 'human', indicating that this player is a human player.
- Behaviours
    - **make_move(board, move):** Allows the human player to make a move by specifying the cell to mark on the game board.

AIPlayer

- Attributes:
    - **name**: Represents the name of the player (e.g., 'Computer').
    - **symbol**: Represents the symbol (e.g., 'X' or 'O') chosen by the player to mark their moves on the game board.
    - **player_type**: Fixed to 'AI', indicating that this player is an AI player.
    - **difficulty**: Represents the difficulty level of the AI (e.g., 'easy', 'medium', 'hard') which affects the AI's decision-making process.
- Behaviours:
    - **make_move(board)**: Allows the AI player to make a move on the game board. The AI uses the current state of the board and its internal logic, influenced by the difficulty level, to decide the best move to make.

Game Class

- Attributes
    - ***Board***: Represents the game board in which the gameplay occurs.
    - ***Players*** : It contains 1 if not both of the players participating in the game.
    - ***Current player***: Represents the player whose turn it is currently.
    - ***Game mode***: Responsible for the choice of either single player or vs AI in the game.
    - ***Difficulty***: Represents the difficulty of the AI in single player mode.
    - ***Game state***: shows the state of the game providing information about the current game status(Win, Draw, Lose).

- Behaviours
    - ***start_game()***: Initializes and starts the game by setting up the board and player objects.
    - ***current_player()***: Alternates between players, switching to the next player after each move.
    - ***check_game_status()***: Checks the current status of the game, such as detecting a win, draw, or ongoing play.
    - ***end_game(winner)***: Ends the game and displays the result, indicating the winning player or a draw.
    - ***select_difficulty()****:* Gives the player the option to choose how difficult the AI will be between three choices(easy,medium,hard).

MAIN Classes

- Attributes
    - ***Window****:* Represents the graphical user interface window where the game is displayed.
    - **Exiting**: Exiting is used to control the termination of the main game loop in order to exit the game.
- Behaviours
    - ***draw_board()***: Draws the game board on the user interface.
    - ***get_player_move()***: Retrieves the player's move input from the user interface.
    - ***display_winner(winner)***: Displays the winner of the game on the user interface.
    - **exit_game()**: Enables the user to exit the game.

Menu Classes

- Attributes
    - **Menu**:Represents the Initial screen to select the game mode.
- Behaviours
    - **select_gamemode**: Allows the user to select Between single player or vs AI

OOP Concepts

Encapsulation

Encapsulation consists of bundling data and methods into a single unit and restricting access to specific functions and data stored inside of the class or the object's components. This helps to prevent accidental interface and misuse such as deletion of class data. In my code, each class encapsulates its own attributes and behaviours in a single unit. For example the **'Board'** class encapsulates the game board's size and grid, along with methods for configuring the board, displaying it and updating it. Other classes interact with the **'Board'** class because it contains essential data necessary for the game's operation as it contains the player interaction, game logic and User interface. However other classes interact with the **'Board'** class through its defined interface without directly accessing its data contained within the class.

Abstraction

Abstraction is the concept in python that simplifies complex concepts and focuses on the necessary details, meaning, abstraction is about hiding complicated implementation details and exposing only the required functionalities. It focuses on the essential characteristics that distinguish different kinds of objects. In my game Abstraction is utilised by hiding complex implementation details and provides a simple interface for interacting with objects. An example of this is shown in the **'Player'** class. This class hides the details of how a player makes a move or chooses a symbol showing only the relevant methods, being:**'make_move'** to place the chosen mark in a cell and '**choose_symbol'** in order to choose either "**X**" or "**O**" as the mark.

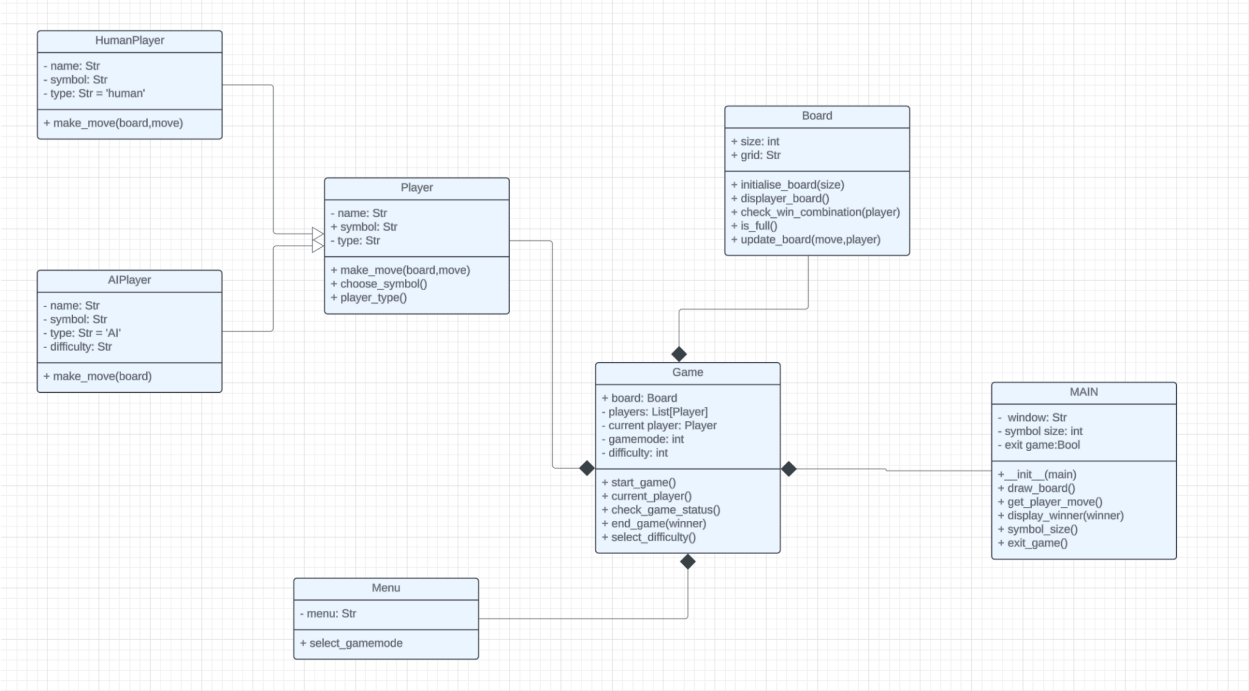
Inheritance

Inheritance is a mechanism that allows a new class(child class/derived class) that inherits the methods and properties from another existing class(parent class/base class). This promotes code reusability, saves time and resources and creates better connections between classes. Inheritance is demonstrated in my code through the specialised sub-classes from the parent class **'Player'**. These child classes represent different players (' **HumanPlayer**', '**AIPlayer**') with specialised behaviours.Such as the **'AIPlayer'** having difficulty levels to suit the ability of the '**HumanPlaye**r'.

Polymorphism

Polymorphism is a term in python that refers to an object's ability to take on multiple forms, but also be able to respond to the same method of function in different ways. This allows objects of a different class to be treated as objects from a common superclass. This is again applied through the **'Player'** class and its "**HumanPlayer**" and "**AIPlayer**" sub classes. By defining a common interface in the **'Player'** class, this allows the sub classes to override methods such as **'make_move()**', as both subclasses provide their own specific logic for making moves on the game board with **'AIPlayer'** utilising AI algorithms to determine the best move based on the current game state and the chosen difficulty level. The code enables different player types to be treated uniformly while accommodating their specific behaviours.

This is my UML diagram depicting the relationships between classes:



Project Report

Development process

This project report documents the process of developing a Tic Tac Toe game. The project was carried out over the course of four days. The primary objective was to design and implement a functional and user-friendly Tic Tac Toe game using Python.

- Planning/ Design:
    - The development of the Tic Tac Toe(TTT) game began with thorough planning and design to ensure that there were no significant mistakes during the coding process. The primary goal was to create a digital version of TTT that offers both a classic two player mode and a single player mode against an AI opponent with 3 different difficulties. During the planning phase the scope of the game was defined focussing on creating a 3x3 grid game board, implementing player turns and checking for win conditions or draws. During the planning phase most of the time was spent on identifying the classes that needed to be made as well as the main functions in order to maximise time efficiency during the coding. There were not too many issues with planning the game, however, the main issue was that I was heavily underestimating the coding prowess necessary to make a TTT game with the overwhelming amount of features that I had mentioned previously, this resulted in having to make some minor adjustments to the features that my game was supposed to contain. UML diagrams were created to map out the game

logic, including the specific classes and main functions contained within the classes.

- Coding:
    - The coding started off by implementing the basic structure of the game. The ‘Board’ class was created to manage the game board, its size and the grid. This was so that I had a concrete floor in order to start coding and building up from the board. Functions for displaying the board and updating it with player moves were implemented within the class. Next, the ‘Menu’ class was created so that I had a menu with 3 choices being: Single player vs AI, Two player mode or Exit. Next, the ‘Game’ class was created to manage the game flow. Alternating turns and checking for win conditions or draws. This was also so that my game could actually run. Next, The ‘Player’ class as well as the ‘HumanPlayer’ and ‘AIPlayer’ sub classes were created to ensure that the person running the game could play as well as have the ability to vs another person on the same device with Two player mode, or vs an AI with 3 difficulty levels. This was probably the hardest part to code mainly because of the complex algorithms that were necessary to implement the “hard” AI. Although I had a lot of trouble, classmates and the internet helped me overcome this issue. Lastly the ‘Main/UI’ class was created. The ‘Main’ class consists of the graphical interface or the window in which the game would be shown through pygame.

- Testing:
    - Once the initial coding was completed, extensive testing was conducted to ensure that the game ran smoothly with no issues. Functional testing involved playing multiple games to verify that win and draw conditions were correctly detected. Testing also included trying the different types of AI at various difficulty levels to ensure it provided a challenging opponent for the human player. This phase also involved debugging and optimising the code to improve performance and user experience. The two player mode also had no issue when played with classmates on the same device. However there was one issue that was seemingly hard to fix. This was the issue that the lines between the cells were not visible during the game and only appeared once the game had ended just before the game closes, initially I thought that It was a bug in the colour so i tried multiple times to fix the colour however the chosen colour would pop up once again after the game had finished. I fixed this issue by solving some bugs that made it so that the lines showed after the game ended, and made sure that the width parameter of the lines were bigger than one as well as changing the colour to black to ensure the visibility of the lines.

Effectiveness of OOP concepts

The implementation of my Tic Tac Toe(TTT) game effectively leverages several key Object oriented programming (OOP) concepts, enhancing both the structure and functionality of the game.

Encapsulation:
Encapsulation is demonstrated throughout my classes(‘Board’, ‘Player’, ‘Game’,’Main’) encapsulating attributes and behaviours within each class while restricting direct access to internal data. This approach ensures that each component operates independently, minimising unintended interactions and enhancing code modularity. For example, the Board class encapsulates attributes like size and grid, along with methods for managing and displaying the game board, maintaining a clear separation of concerns.

Abstraction:
Abstraction simplifies complex implementation details and exposes only necessary functionalities, which is displayed in the Player class. By providing a streamlined interface (make_move(), choose_symbol()) for interacting with players, abstraction hides the specifics of how moves are made or symbols are chosen. This abstraction allows the game logic (Game class) to treat different player types uniformly (HumanPlayer, AIPlayer), facilitating easier maintenance and extensibility.

Inheritance:
Inheritance is effectively used to promote code reuse and establish hierarchical relationships between classes. Subclasses (HumanPlayer, AIPlayer) inherit common functionalities from the Player superclass, enabling shared behaviours (make_move(), choose_symbol()) while allowing for specific implementations tailored to each player type. This approach not only reduces redundancy but also enhances flexibility facilitating the addition of new player types or modifications to existing ones without restructuring the entire codebase.

Polymorphism:
Polymorphism further enhances flexibility by allowing objects to be treated as instances of their superclass (Player), while still invoking specific behaviours defined in their subclasses (HumanPlayer, AIPlayer). This flexibility is crucial in my game, where different player types interact with the game logic (Game class) using a common interface, despite varying implementations of moves and decision-making processes.

Benefits of OOP
Object oriented programming offers a number of advantages when applied to the domain of interactive digital games such as Tic Tac Toe(TTT)

- Encapsulation
    - In game development, encapsulation ensures that each game component or class (e.g ‘Board’, ‘Player’ or ‘Game’ class in my game) encapsulates its own state and behaviour. For example, the ‘Board’ class encapsulates the grid state

and the methods behind managing it(updating, displaying). This containment prevents accidental interference from other parts of the program, ensuring the “safety” of game logic and data. Encapsulation also allows for easier debugging and maintenance as each component can be tested individually and modified without affecting the other parts.However, Python's lack of strict access control mechanisms can sometimes lead to accidental access or modification of supposedly private attributes, impacting encapsulation. To prevent this, developers can employ Python's properties and descriptors to enforce controlled access to attributes and methods, thereby enhancing encapsulation by explicitly defining getters, setters, and deleters.

- Abstraction
    - Abstraction in game development hides complex details behind simplified interfaces. This helps to manage game complexity and enhances the productivity of the developer. For instance the ‘player’ Class hides the specifics of how moves are made on the board, providing a consistent interface(‘make_move()’) regardless of whether the player is human or AI. This simplifies the code helping the developers to forget about the low level implementation details. While abstraction simplifies code and reduces complexity, it can also obscure underlying implementation details, potentially complicating debugging and maintenance efforts. To address this, thorough documentation, code comments, and comprehensive unit testing are crucial to clarify abstracted interfaces and ensure correct functionality across different implementations.

- Inheritance
    - Games often include various types of players, levels or game modes. Inheritance controls code reuse and promotes order among the classes, allowing developers to extend existing functionality without repeating the code in turn making it longer and more confusing. For example ‘HumanPlayer’ and ‘AIPlayer’ are child classes that inherit from the bigger parent class being the ‘Player’ class. Therefore, they inherent common behaviours like ‘make_move()’ while implementing specific logic for human input or AI decision making according to the difficulty. The action of reusing code not only saves time but it also supports complex game interactions and allows for new features or player types to be added without restructuring the existing code. However, deep inheritance hierarchies can result in tightly coupled designs that are difficult to refactor or extend as game features evolve. To mitigate this issue, game developers can adopt composition-based designs using mixins or component-based architectures. This approach promotes flexibility and modularity by allowing components to be composed dynamically, rather than relying on rigid class structures dictated by inheritance.
- Polymorphism
    - Polymorphism allows objects of different types to be treated uniformly through a common interface. In games, this malleable nature of polymorphism allows handling game elements dynamically as in, polymorphism enables a ‘Game’ class to interact with different types of players (Player subclasses) using the

same set of methods (make_move(), check_win_condition()), adapting behaviour based on the specific player instance (human or AI). This adaptability supports complex game interactions and facilitates the addition of new features or player types without restructuring existing code.In Python, dynamic dispatching can introduce performance overhead during method resolution, impacting real-time game responsiveness. To optimise performance, developers can utilise static methods or class methods where applicable, minimising the overhead associated with dynamic dispatch. Additionally, implementing robust type checking and assertions ensures that polymorphic behaviour adheres to defined interfaces and expected object types, thereby enhancing clarity and reliability in game development.