Tic Tac Toe Project Report

Introduction

- What is your application?

The Tic Tac Toe application is a classic two-player game implemented in Python using the Tkinter library for the graphical user interface. The game allows for two modes: Human vs Human and Human vs AI. Players take turns marking a 3x3 grid with 'X' or 'O', and the goal is to be the first to get three of their marks in a row, column, or diagonal.

- How to run the program?

1. Ensure you have Python installed on your system.
2. Save the provided code in a file named `tictactoe.py`.
3. Run the program using the command:

```sh
python tictactoe.py
```

- How to use the program?

1. Select the game mode (Human vs Human or Human vs AI) using the radio buttons.
2. Enter the player names in the input fields.
3. Click on the grid buttons to make a move.
4. The game will announce the winner or declare a tie when the game ends.
5. Use the reset button to start a new game.

Body/Analysis

- Polymorphism

Polymorphism is implemented through the `Player` base class and its subclasses `HumanPlayer` and `AIPlayer`. Each subclass implements the `make_move` method differently:

```python
class Player:
    def __init__(self, symbol):
        self.symbol = symbol

    def make_move(self, buttons, row=None, col=None):
        pass

class HumanPlayer(Player):
    def make_move(self, buttons, row, col):
        if buttons[row][col]['text'] == " ":
            buttons[row][col]['text'] = self.symbol

class AIPlayer(Player):
    def make_move(self, buttons, row=None, col=None):
        empty = [(r, c) for r in range(3) for c in range(3) if buttons[r][c]['text'] == " "]
        if empty:
            row, col = random.choice(empty)

            buttons[row][col]['text'] = self.symbol
```

- *Abstraction*

Abstraction is achieved by defining the `Player` base class, which provides an abstract interface for the `make_move` method. This allows for different implementations of the method in subclasses.

- *Inheritance*

Inheritance is used with the `Player` base class and its subclasses `HumanPlayer` and `AIPlayer`:

```python
class HumanPlayer(Player):
    # Implementation for human player


class AIPlayer(Player):
    # Implementation for AI player
```

- *Encapsulation*

Encapsulation is demonstrated by keeping the game state (e.g., the game board and player moves) within the `TicTacToe` and `Player` classes. Methods are provided to interact with these states, keeping the internal details hidden.

- *Singleton Pattern*

The Singleton Pattern is used to ensure only one instance of the `TicTacToe` class is created:

```python
class TicTacToe:
    _instance = None

    def__new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super(TicTacToe, cls).__new__(cls)
        return cls._instance
```

- *Factory Method Pattern*

The Factory Method Pattern is used to create player objects:

```python
class PlayerFactory:
    @staticmethod
    def create_player(player_type, symbol='O'):
        if player_type == 'human':
            return HumanPlayer(symbol)
        elif player_type == 'ai':
            return AIPlayer(symbol)
```

- *Reading from and Writing to File*

The game state is saved to and loaded from a file (`game_state.txt`):

```python
def save_game(self):
    with open('game_state.txt', 'w') as f:
        for row in self.buttons:
            f.write(''.join(button['text'] for button in row) + '\n')

def load_game(self):
    if os.path.exists('game_state.txt'):
        with open('game_state.txt', 'r') as f:
            for i, line in enumerate(f):
                for j, char in enumerate(line.strip()):
                    self.buttons[i][j]['text'] = char
```

- *Testing*

Basic unit tests are written using the `unittest` framework to ensure core functionality works as expected:

```python
import unittest
from unittest.mock import Mock
import tkinter as tk
import random
```

```
class TestTicTacToe(unittest.TestCase):
    def setUp(self):
        self.root = tk.Tk()
        self.game = TicTacToe(self.root)

    def tearDown(self):
        self.root.destroy()

    def test_initial_state(self):
        self.assertEqual(self.game.flag, 0)
        self.assertTrue(self.game.bclick)

    def test_player_move(self):
        self.game.btn_click(0, 0)
        self.assertEqual(self.game.buttons[0][0]['text'], 'X')
        self.assertFalse(self.game.bclick)

    def test_ai_move(self):
        self.game.bclick = False
        self.game.ai_move()
        move_made = any(self.game.buttons[r][c]['text'] == 'O' for r in range(3) for c in range(3))
        self.assertTrue(move_made)

    def test_reset_game(self):
        self.game.btn_click(0, 0)
        self.game.reset_game()
        for row in self.game.buttons:
            for button in row:
                self.assertEqual(button['text'], " ")


if__name__== '__main__':
    unittest.main()
```

- *Code Style*

The program is written in Python and follows PEP8 style guidelines to ensure readability and maintainability.

Results and Summary

- *Results*

- The game correctly alternates between 'X' and 'O' for human players.

- The AI makes valid moves when playing against a human.

- The game state can be saved and loaded from a file.

- The game correctly detects win conditions and ties.

- *Summary*

The Tic Tac Toe project demonstrates the use of OOP principles, design patterns, file I/O, and unit testing. The game is functional and meets the requirements of the coursework. Future improvements could include enhancing the AI to use a more sophisticated strategy and adding more comprehensive tests

Conclusions

The Tic Tac Toe project successfully demonstrates the implementation of key object-oriented programming principles, including encapsulation, inheritance, polymorphism, and abstraction. By using the Singleton and Factory Method design patterns, the project effectively manages game state and player creation. The program includes functionality for reading and writing the game state to a file, ensuring that progress can be saved and resumed. Additionally, the project follows PEP8 coding standards and includes unit tests to verify the core functionality.

This work has achieved the creation of a functional and interactive Tic Tac Toe game that supports both Human vs Human and Human vs AI modes. The use of design patterns and OOP principles not only enhances the maintainability and scalability of the code but also provides a clear structure for future extensions.

Future prospects for this project could include improving the AI to use more advanced strategies, adding more comprehensive unit tests, and potentially extending the game to support different grid sizes or additional game modes. The project lays a solid foundation for further exploration and enhancement of game development using Python and Tkinter.

- Optional: Resources and References

- [PEP8 Style Guide](https://peps.python.org/pep-0008/)

- [Tkinter Documentation](https://docs.python.org/3/library/tkinter.html)

- [unittest Documentation](https://docs.python.org/3/library/unittest.html)

- [Design Patterns](https://refactoring.guru/design-patterns)