Documentation of Tic Tac Toe Game

Christof Renner and Manuel Friedl

June 22, 2024

Contents

1 Introduction **2**

2 Architecture **2**
2.1 Overview 2
2.2 UML-Diagrams 3
2.2.1 Class Diagram 3
2.2.2 Activity-Diagram 4

3 Serialization and Deserialization **5**
3.1 Implementation Details 5
3.1.1 Serialization 5
3.1.2 Deserialization 5
3.1.3 .JSON-File 7

4 Game AI **7**
4.1 Minimax Algorithm 8

5 Tests **9**
5.1 Achieving 90% Line Coverage 9
5.2 PyLint 10

6 Contributions **10**

7 Conclusion and prospects **10**

1 Introduction

This document aims to provide a comprehensive documentation of the Tic Tac Toe game, covering its gameplay, architecture, AI implementation, serialization mechanisms, testing strategies, contributions, and future prospects. Tic Tac Toe is a classic two-player game played on a 3x3 grid. Each player takes turns marking a cell in the grid, with one player using "X" and the other using "O." The objective is to be the first to get three of their marks in a row, which can be horizontally, vertically, or diagonally. If all cells are filled without either player achieving three in a row, the game ends in a draw.

2 Architecture

2.1 Overview

The game's architecture is designed to separate concerns, enhance code reusability, and simplify maintenance. It is divided into three main components: the Model, the View, and the Controller, following the MVC design pattern. The Model represents the data and business logic, the View displays the data (UI), and the Controller handles the user input and updates the Model.

2.2 UML-Diagrams

2.2.1 Class Diagram

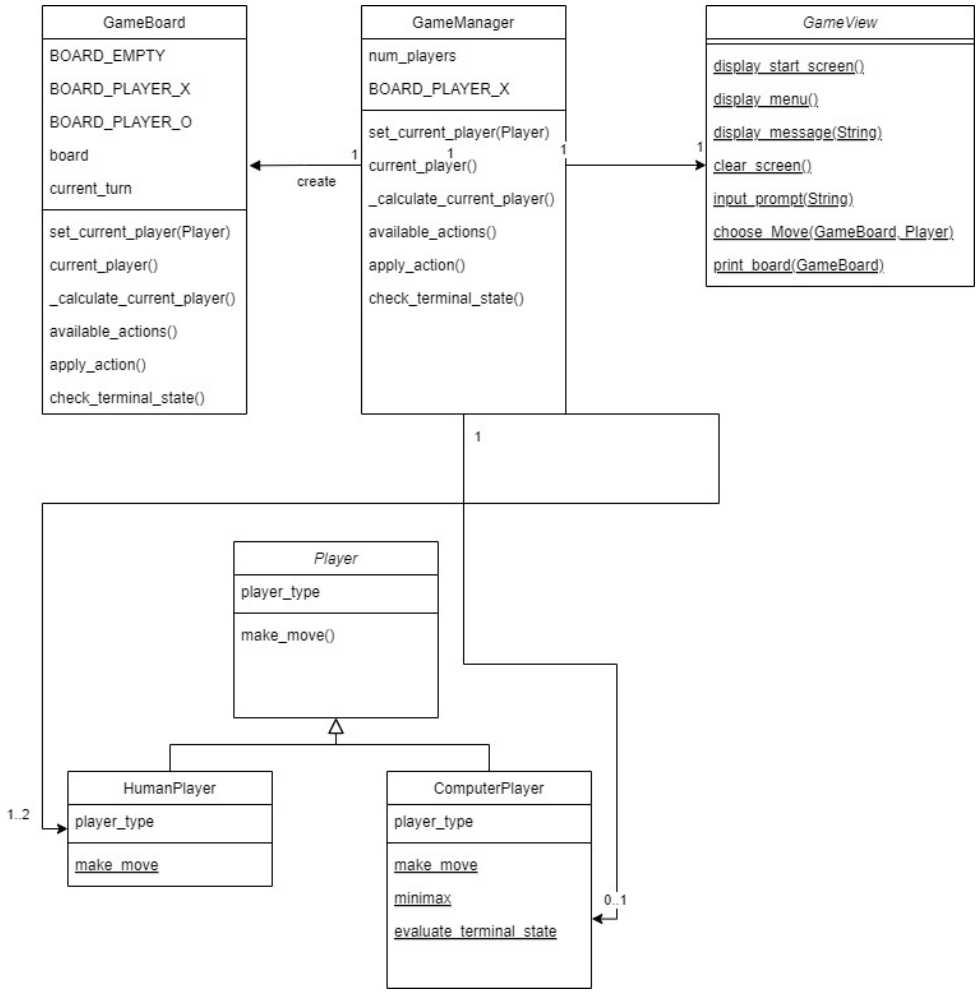

Figure 1: Class-Diagram

2.2.2 Activity-Diagram

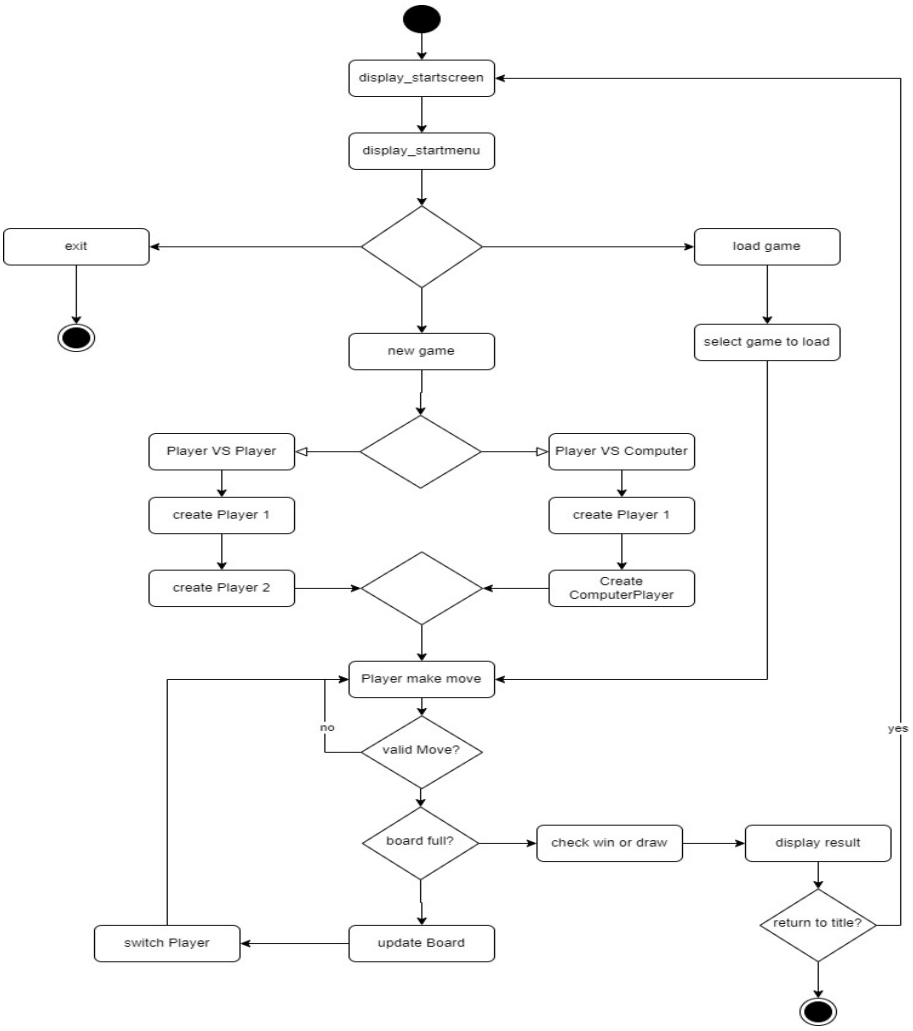

Figure 2: Activity-Diagram

3 Serialization and Deserialization

Serialization and deserialization mechanisms are crucial for saving and loading game states. This functionality is implemented through Python's JSON module, allowing game states to be saved to disk and retrieved later.

3.1 Implementation Details

3.1.1 Serialization

This code snippets showing how the game state is serialized into JSON format and deserialized back into game objects. JSON serialization in Python involves converting a Python object into a JSON formatted string using the json.dumps()-method.

```
import json

def save_game_state(self):
    game_state = {
        'board': self.board.board,
        'num_players': self.num_players,
        'current_turn': self.board.current_player()
    }
    with open(filename, 'w') as file:
        json.dump(game_state, file)
```

Listing 1: Serialization Example

3.1.2 Deserialization

Loading the game is a little bit more complicated so it is split up in several parts to give a better understanding of the code.
The method starts by defining the directory where the saved game files are stored.

```
import json

def save_game_state(self):
   saved_games_dir = 'savedGames'
```

Listing 2: Inititialization

It attempts to list all files in the "savedGames" directory. If the directory doesn't exist or there are no files, it displays an appropriate message and returns FALSE to indicate failure.

```
try:
    saved_files = os.listdir(saved_games_dir)
    if not saved_files:
        GameView.display_message("No saved games found.")
        return False
except FileNotFoundError:
    GameView.display_message("No saved games directory found.")
    return False
```

Listing 3: Checking for saved Games

If there are saved games, it generates a dynamic list and print the saved games in the UI. The user is prompted to enter the number corresponding to the game they want to load or type 'exit' to return to the main menu.

```
try:
    saved_files = os.listdir(saved_games_dir)
    if not saved_files:
        GameView.display_message("No saved games found.")
        return False
except FileNotFoundError:
    GameView.display_message("No saved games directory found.")
    return False
```

Listing 4: Checking for saved Games

If the user chooses to exit, the screen is cleared, the start menu is displayed, and the method returns False. Otherwise, it tries to convert the user's input to an integer and use it to select a file. If the input is invalid, it displays an error message and returns FALSE.

```
GameView.display_message("Please select a game to load:")
for i, file in enumerate(saved_files, 1):
    GameView.display_message(f"{i}. {file}")
choice = GameView.input_prompt("Enter the number of the game you
    "
                               "want to load"
                               "('exit' to return to main menu):
    ")
```

Listing 5: Handling User Input

The selected file is opened, and its contents are read and parsed as JSON to reconstruct the game state. A new game board will be initialized and restores the board configuration, number of players, and current turn from the loaded state.

```
with open(f'{saved_games_dir}/{selected_file}', 'r', encoding='
    utf-8') as file:
    state = json.load(file)
self.board = GameBoard()
self.board.board = state['board']
self.num_players = state['num_players']
current_turn = state['current_turn']
```

Listing 6: Handling User Input

The end of the method is depending on the number of players and the current turn, it initializes the appropriate player objects (human or computer) and sets the current player. After that a message comes up that the game is loaded successfully.

3.1.3 .JSON-File

The .json-File is quite simple. There are only three key-value-plairs, that are stored in the .json-File.

```
{"board": [0, 0, 0, 0, "X", 0, 0, 0, "O"], "num_players": 2, "
    current_turn": "X"}
```

Listing 7: Test.json File

- **"board":** is a array that represents the game board with its 9 positions (0 - 8). The number '0' represents an empty slot, 'X' represents the Player-X and the character 'O' represents the Player-O.
- **"num_player":** indicate the number of players in the game. If the value is 1, a player is playing against the AI. Is it 2, its a PVP game.
- **"current_turn":** shows the player which is currently taking a turn.

4 Game AI

The game AI uses the Minimax algorithm to determine the optimal move for the computer player. It evaluates the game board at each step to make the most advantageous move.

4.1 Minimax Algorithm

The Minimax algorithm is a recursive algorithm used for minimizing the possible loss for a worst-case scenario. For a better understanding how the algorithm works here are some code snippets with explanations.
In the first line will be checked if the current board state is a terminal state (win, loss, or draw). If it is a terminal state, it evaluates this state using the evaluate_terminal_state()-method and returns the score.

```
terminal_state = board.check_terminal_state()
if terminal_state is not None:
    return self.evaluate_terminal_state(terminal_state, depth)
```

Listing 8: Check Terminal State

If is_maximizing is true, it means the current player is the AI. The variable best_score will be initialized to negative infinity. Then it iterates through all possible actions that can be taken from the current board state. For each action, it:

- Applies the action to the board.
- Calls the minimax()-method recursively to evaluate the resulting board state, switching to the minimizing player.
- Undoes the action (resets the board).
- Updates the variable best_score with the maximum score returned from the recursive calls.

```
if is_maximizing:
    best_score = float('-inf')
    for action in board.available_actions():
        board.apply_action(action)
        score = self.minimax(board, depth + 1, False)
        board.board[action[1]] = GameBoard.BOARD_EMPTY
        best_score = max(score, best_score)
    return best_score
```

Listing 9: is_minimizing is true

The other case simulates, that the current player is the the player, it initializes the variable best_score to positive infinity. Similar to the maximizing case, it iterates through all possible actions and does the same as above.

```
best_score = float('inf')
        for action in board.available_actions():
            board.apply_action(action)
            score = self.minimax(board, depth + 1, True)
            board.board[action[1]] = GameBoard.BOARD_EMPTY
            best_score = min(score, best_score)
        return best_score
```

Listing 10: is_minimizing is false

5 Tests

Our testing strategy focused on achieving high code coverage and ensuring the reliability of the game's core functionalities. We used the Python `unittest` module and the `coverage` utility to achieve our line coverage goals. The tests can be run by executing the following command in the root of the repository:

```
coverage run -m unittest discover -s ./tests/ -p "test_*"
```

Listing 11: Run tests

To display the line coverage output, use the following command:

```
coverage report
```

Listing 12: Show coverage report

For a visual HTML output, use:

```
coverage html
```

Listing 13: Generating html files

The corresponding HTML files are also present in the repository for convenience.

5.1 Achieving 90% Line Coverage

To ensure that at least 90% of our business logic was covered by tests, we employed the following strategies:

- Writing unit tests for all models and utility functions.
- Implementing integration tests to cover scenarios involving user interactions.

We are currently achieving a total line coverage of 98%. The line coverage of the individual files is shown here:

```
PS C:\Repos\Python-TicTacToe> python -m coverage report
Name                          Stmts   Miss  Cover
-------------------------------------------------
board.py                         43      0   100%
controller.py                   158     14    91%
player.py                        51      0   100%
tests\test_board.py             102      1    99%
tests\test_controller.py        241      2    99%
tests\test_view.py              149      1    99%
view.py                          49      1    98%
-------------------------------------------------
TOTAL                           793     19    98%
```

Figure 3: Coverage

5.2 PyLint

As the project has to be linted, we have set up a workflow on GitHub. We use the Pylint workflow from GitHub Actions for this. Every time someone pushes code, it starts and analyzes the code with Pylint. A report is generated automatically. Our current PyLint score is 9.59.

6 Contributions

This section details the contributions of each team member to the project.

7 Conclusion and prospects

Tic Tac Toe is a rather simple game, but there is still room for improvement and many more features can be added. Here are some future improvements or changes:

- Add a difficulty system when playing against the AI. This can be done, for example, through the "intelligence" of the AI or even by changing the size of the game board.

- Another point would be the customization of the symbols and the playing field with colors and symbols that the player can choose himself.
- It would also be a visual improvement if you could build a real GUI and place your tile on the corresponding field with a click of the mouse.
- A scoreboard which stores the Wins of the Player or the Computer would also be a nice feature. To identify the Player it would be possible to create an account system with a database in the background. So if a Player wants to play a game he has to first log in to his account.
- And of course there are always some code optimizations that improve the performance, security and maintenance of the code.