T1A3: Terminal Application

Terminal Connect Four by Deandre Sugandhi

Hi, my name is Deandre Sugandhi. I'm going to showcase my Terminal Application, which is a connect four clone app for the terminal.

Main Features

- Connect Four Match
- User Account System
- High-Score System
- Customizable Piece
- Game Hubs: Player Lounge & Main Lobby

Terminal Connect Four is a terminal application for playing the traditional Connect Four game, with several added functionalities. The following is the main features of the game.

Connect Four Match

- Generates 2 players PvP (Player vs Player) Connect Four match
- Involves 2 classes, Board & Piece
- Turn-taking mechanism
- Winning conditions checked every turn
- Input column number, “clear”, or “surrender”

Figure 1.1: A Connect Four match in progress.

So the first feature of the app is a Connect Four match. For now, this app is purely PvP for 2 players. If it is developed further, I might add AI functionalities to allow single player matches against AI with varying difficulties.

This feature consists of three components, namely the game board, game piece, and a referee object to check for game results every turn. The game board, made as an instance of the Board class, displays a 6 x 7 array in the form of 7 columns and 6 rows. Players take turns dropping their game piece, an instance of the Piece class, into the board, just like the traditional Connect Four game. This is done using the drop method of the Piece class, which prompts for user input, inputting a column number where they would want to drop their game piece. For column inputs, only the numbers 1-7 is allowed; numbers outside that is considered invalid, prompting the user to input another command that is valid. The drop method of the piece class also detects when a column is full, which will then render input commands of that column invalid. Other than inputting column number, a player can also type in the commands “clear” and “surrender”. “Clear”, as the name suggests, clears the board, acting like the slider under a physical Connect Four board. “Surrender” allows a player to forfeit a match and grant automatic victory to the other player. The logic behind these classes will be explained later on in the presentation.

The winning conditions is also represented using a class, namely the VictoryChecker class. An instance of this class is assigned to the variable “referee”, which as the name suggests, acts as the referee of the match. I decided to implement these 3 classes because I find it intuitive to imagine this application as a sort of simulation of the physical Connect Four game. Through object-oriented programming, the Connect Four match feature can be visualized like a physical Connect Four game; there is a referee “object” to check for win conditions, there is an empty game board “object” with slots where pieces can be dropped, and there are the pieces themselves that have their own properties and can be dropped into the board. The winning conditions are horizontal victory (4 pieces stacked horizontally), vertical victory, diagonal victory, and surrenders. There is also a draw condition, when the whole board is filled up with pieces. The logic behind these conditions will be discussed further in the coming slides.

User Account System

- Stores user data
- Allows use of guest accounts

```
Welcome! Before starting the game, both players must be logged in.
Player 1! Are you an existing user? (y / n):
```

```
Please enter a 4-digit PIN for your account. This will be used
to verify your future sign-ins.
Note: For now, PINS and usernames are stored in a .json file,wh
ich is a human-readable file format. Due to the risk of unautho
rized access, please use a unique PIN that is not associated wi
th your other external accounts.
PIN: ****
```

```
Enter your username: tester123
Enter your PIN: ****
```

Figure 2.1: Creating a new account


```
[
    {
        "username": "tester123",
        "pin": "1923",
        "games_played": 5,
        "wins": 2,
        "losses": 1,
        "win_ratio": 60.0,
        "color": "black",
        "piece_type": "Z",
        "logged_in": "n"
    },
    {
        "username": "tester321",
        "pin": "1009",
        "games_played": 2,
        "wins": 1,
        "losses": 0,
        "win_ratio": 75.0,
        "color": "light_yellow",
        "piece_type": "B",
        "logged_in": "n"
    }
]
```


Figure 2.2: Sample contents of users.json

The next feature is the user account system. This allows a user to create their own account to store personal game data such as total games played, wins, losses, win ratio, piece color, and piece type. All user data is stored in a JSON file named users.json.

At the start of the game, before being able to access the game's main features, both players are prompted to setup their accounts. They can either login to their existing personal accounts, create new personal accounts, or use guest accounts which do not have full access to the functionalities of the game such as piece customization, game statistics, etc. Creating personal accounts requires users to input a unique username (with a specific format) and a four-digit PIN for registration. PIN inputs are masked for security purposes. Username / PIN will be refused, errors raised and handled accordingly, if either one is in an invalid format or the username is already associated with another account. The program also stores the login status of each user account, meaning that a player cannot attempt to login to an account that is already currently logged in.]

For now, account information including username and PIN are not encrypted when stored into the JSON file. This means that anyone having access to the JSON file can

easily read all account information. This warning is displayed in-game when a user attempts to create a personal account.

High-Score System

- Gathers user game records from JSON file, sort them, and display top 5 users based on sorting keyword (wins, games played, or win ratio)

Figure 3.1: Sample of high-score system, with games played as the sorter

The high-score system is a feature that gathers game records from the JSON file and sorts them in descending order based on a keyword inputted by the player accessing it. Players can then view the top 5 user accounts with most games played, most wins, or highest win ratio. The top users' username, games played, wins, losses, and win ratio details are shown. Guest accounts can access this feature but are not included in it as their game histories are not recorded into the JSON file. This feature is available in the player lounge, a featured explained later in the presentation.

Customizable Piece

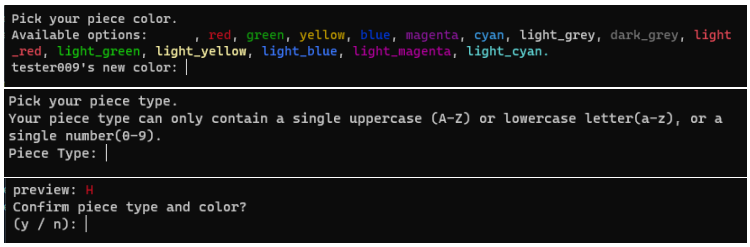


Figure 4.1: Customizing a Piece instance

- Game piece is defined by piece type and color
- User accounts, but not guest accounts, can customize them

Players can also choose to customize how their game pieces are displayed on the game board during matches. This includes the piece type and piece color. The piece type is any single alphanumeric character chosen by the player that will be used to represent their game piece on the board, while the piece color defines the text color of the alphanumeric character. Players on a match are allowed to have the same piece color or piece type, but not both, to prevent confusion; if such a case is detected, the players are prompted to make changes to their piece properties. Players using guest accounts cannot access the piece customization feature, using the default guest account properties of white "O" piece for player 1 and white "X" piece for player 2. This customization will be stored in the JSON file, so the system remembers a user's customization for the next time they login

Game Hubs: Player Lounge & Main Lobby

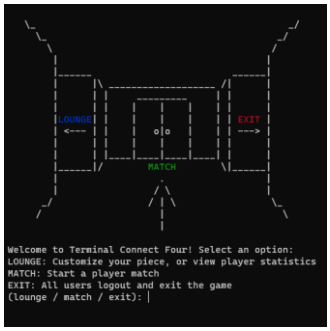


Figure 5.1: Game Lobby

Figure 5.2: Player Lounge

- Main Lobby: Landing area after game is launched. Provide access to Player Lounge, Match, and Exit Game.
- Player Lounge: Provides access to piece customization, access user info, high-score board, and exit to lobby.

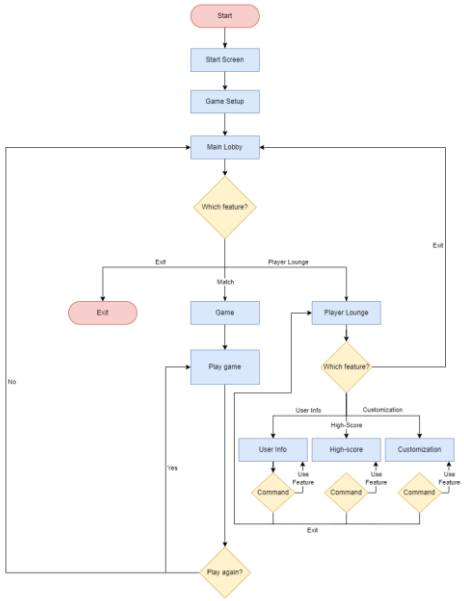
The game implements a class of objects called Game Hub, which represents lobbies or hubs with a selection of features that a user can enter and access. One can visualize it as a sort of room, hall, or corridor, with, as visualized in the ASCII art image above. The Player Lounge is one such hub, with features including piece customization, high-score board, and user information kiosk. Piece customization, as explained earlier, is a feature that allows users to customize their piece and store it on their user data. High-score board is the visualization of the high-score system, displaying the top 5 players from all user records based on the keyword inputted by users accessing the feature. Access user info can be imagined like a user information kiosk, but for user data. Users can input a username and view their game records as fetched from the JSON file. Of course, information like PIN and log in status is not displayed.

The Main Lobby is another Game Hub instance, which is the landing area after both players have logged into their respective accounts. This is the hub from which players can choose to start a Connect Four match, enter the player lounge, login to a different account, or exit the game.

Both of these are actually their own respective classes, inheriting from the Game Hub

class. This will be discussed later on in the coming slides.

Flowchart



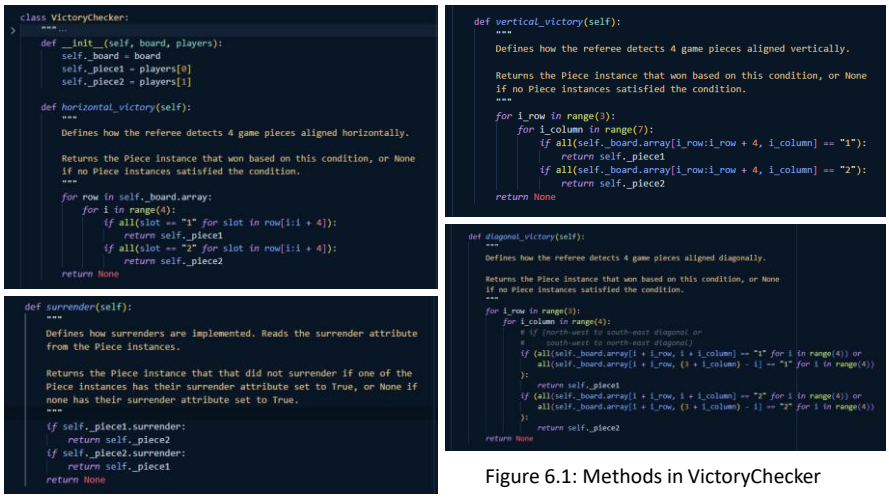
So this is a general flowchart for how the app roughly works. On start, we go to the start screen, where users are prompted to press enter to continue. Then, game setup is performed, in which both players must login either to their existing accounts, create new accounts, or use guest accounts. Then, users are brought to the main lobby, where they are given the choice to either enter a player match, enter the player lounge, or exit the game. If the users decide to enter a match, the game progresses, game record is stored on completion of game, and users are given the choice to play again or go back to the lobby. If the users decide to enter a player lounge from the lobby, they are given the choice to use the following 3 features: access user info, high-score board, or piece customization. These 3 features ask each of their unique sets of input commands, until the users decide to type in the exit command, in which they will be brought back to the player lounge. If the player exits from the player lounge, they go back to the lobby.

Highlights and Application Logic

- Connect Four match & winning conditions
- Game Engine
- Board & Piece classes
- GameHub class
- Data storage in users.json

Next I'm going to describe the logic behind some of the main features of the app.

Connect Four Match & Winning Conditions



```
class VictoryChecker:
    """...
    def __init__(self, board, players):
        self._board = board
        self._piece1 = players[0]
        self._piece2 = players[1]

    def horizontal_victory(self):
        """
        Defines how the referee detects 4 game pieces aligned horizontally.

        Returns the Piece instance that won based on this condition, or None
        if no Piece instances satisfied the condition.
        """
        for row in self._board.array:
            for i in range(4):
                if all(slot == "1" for slot in row[i:i + 4]):
                    return self._piece1
                if all(slot == "2" for slot in row[i:i + 4]):
                    return self._piece2
        return None
```

```
def vertical_victory(self):
    """
    Defines how the referee detects 4 game pieces aligned vertically.

    Returns the Piece instance that won based on this condition, or None
    if no Piece instances satisfied the condition.
    """
    for i_row in range(3):
        for i_column in range(7):
            if all(self._board.array[i_row:i_row + 4, i_column] == "1"):
                return self._piece1
            if all(self._board.array[i_row:i_row + 4, i_column] == "2"):
                return self._piece2
    return None
```

```
def surrender(self):
    """
    Defines how surrenders are implemented. Reads the surrender attribute
    from the Piece instances.

    Returns the Piece instance that that did not surrender if one of the
    Piece instances has their surrender attribute set to True, or None if
    none has their surrender attribute set to True.
    """
    if self._piece1.surrender:
        return self._piece2
    if self._piece2.surrender:
        return self._piece1
    return None
```

```
def diagonal_victory(self):
    """
    Defines how the referee detects 4 game pieces aligned diagonally.

    Returns the Piece instance that won based on this condition, or None
    if no Piece instances satisfied the condition.
    """
    for i_row in range(3):
        for i_column in range(4):
            # if (north-west to south-east diagonal or
            #     south-west to north-east diagonal)
            if (all(self._board.array[i + i_row, i + i_column] == "1" for i in range(4)) or
                all(self._board.array[i + i_row, (3 + i_column) - i] == "1" for i in range(4))
            ):
                return self._piece1
            if (all(self._board.array[i + i_row, i + i_column] == "2" for i in range(4)) or
                all(self._board.array[i + i_row, (3 + i_column) - i] == "2" for i in range(4))
            ):
                return self._piece2
    return None
```

Figure 6.1: Methods in VictoryChecker

The winning conditions, as described earlier, is its own class called VictoryChecker. It uses two parameters, namely board (a Board instance) and players, which is a list containing the 2 Piece instances involved in the game. The VictoryChecker instance is assigned to variable “referee.” All winning conditions are class methods that use for loops to do the checks. How it works is that it first checks a slot, starting from the top left corner of the board (row 1 column 1), and find out if it is empty or filled. Empty slots are represented as a 0 in the board’s array, player 1 represented as a 1, and player 2 represented as a 2, all strings. Depending on the winning conditions, it then slices the array and finds out if all elements in the slice is a 1 or a 2. If so, victory is called and the winning Piece instance is returned. This is looped until the whole board is covered. The horizontal and vertical victory methods essentially work like this, though the diagonal victory method is slightly different and more complex. Instead, it does two checks per iteration, one for north-west to south-east diagonal, and another one for south-west to north-east diagonal, and iteration is looped until whole board is covered. Surrender is detected by accessing the surrender attributes of the Piece instances, which is set to True by the game engine if a player inputs the “surrender” command during their turn. All these methods are performed every turn.

To make the code simpler, I decided to remove the draw condition method from this

class and put it in another python file created exclusively for game engines. That way, I can set draw condition as when the piece drop count has reached 42 (there are 42 slots in the game board), and set it as the condition for the game engine's while loop.

Board & Piece Classes

```
class Board:
    """
    Represents a game board used in the game.

    The game board is a 6 x 7 Connect Four board. It functions as a NumPy array
    which is translated and displayed on the terminal through various methods.

    Attributes:
    1. _array (np.array): A NumPy array with 6 lists (representing rows) with
       7 elements each (representing columns). 0 represents an empty space
       in the cage.
    2. _edge (str): Component used to display the board. It is the bottom edge
       of the game board, representing the slider to clear the board.
    3. _divider (str): Component used to display the board. Represents the
       dividers between each row of the game board.
    4. _column (list): A list of components to display the board. Represents
       each "cage" of the board, i.e. the columns.
    5. players (list): A list of Piece class instances, defining the pieces and
       players that is playing on the board, accessing its properties such as
       piece color, piece type, player name, etc to be displayed by the board.
    """
    def __init__(self, players):
        self._array = np.array([["0" for i in range(7)] for i in range(6)])
        self._edge = "+===" *7 + "+"
        self._divider = "+---"*7 + "+"
        self._column = [f"  {i+1} " for i in range(7)]
        self._players = players
```

```
class Piece:
    """
    Represents a game piece used in the game.

    The game piece is a visualization of a player's piece on the board. The
    appearance can be customized by changing a Piece instance's attributes.

    Attributes:
    1. _player_name (str): The name of the player owning the piece.
    2. _color (str): The color of the piece.
    3. _piece_type (str): The visual representation of the piece
       (i.e. its shape)
    4. _player (Player): The player number of the piece (i.e. p1 or p2)
    5. surrender (bool): Represents whether the owner of the piece surrendered.
    """
    def __init__(self, player_name, color, piece_type, player):
        self._player_name = player_name
        self._color = color
        self._piece_type = piece_type
        self._player = player
        self.surrender = False
```

```
def drop(self, board, column):
    """
    A method to drop the piece onto the board. Players 1's piece would be
    represented as a "1" in the board's array, and players 2's piece would
    be represented as a "2" in the board's array.

    Args:
    1. board (Board): An instance of the Board class, defining the board
       onto which the piece will be dropped.
    2. column (str): A string digit representing the column number of the
       board to drop the piece onto
    """
    row = -1
    while board.array[row, column - 1] != "0":
        row -= 1
        if row < -6:
            raise ColumnFullError("Column is full. Please try again: ")
    board.array[row, column - 1] = self.player
```

Figure 7.1: Board & Piece class attributes

The Board class has several attributes. The array attribute is a NumPy array of 6 lists with 7 elements each, representing 6 rows and 7 columns. As mentioned earlier, a “0” element represents an empty slot, “1” for player 1 piece, and “2” for player 2 piece. This array attribute is the core of the board class, and this is what is accessed by other functions such as the VictoryChecker methods and the game engine. The rest of the attributes, except self.players, are used exclusively to display self.array on the terminal in a cleaner, clearer, and more attractive way. Self.players is a list of all Piece instances that are playing the game, defined by placing the list in the class parameter. The board then detects properties such as the player’s name, the colors and the piece type and display it accordingly.

The Piece class has several attributes. All of them have been explained before except for self.player. Self.player is a string of either “1” or ”2” that defines whether the Piece instance is player 1 or player 2. This is the string that will be inputted into the board’s array, replacing the 0’s whenever a piece is dropped. It does this using the drop method.

Game Engine

```
def game_in_progress(board, players, referee):
    """
    Function to decide how a game is run from start until when game result is
    detected.

    Args:
    1. board (Board): An instance of the Board class, defining the board in
       which the game will be played.
    2. players (Piece): An instance of the Piece class, defining the players
       playing the game.
    3. referee (VictoryChecker): An instance of the VictoryChecker class,
       defining the referee that checks the state of the board each game turn
       to see if the game should be over.
    """
    player_turn = 0
    move_count = 0

    # Game loops as long as referee does not detect a winner and there are
    # less than 42 move counts, which is the total number of slots available in
    # the board.
    while referee.check_victory() is None and move_count < 42:
        # Refreshes the screen and board state
        reset_screen(board)

        player_command = input(
            f"\n{colored(players[player_turn].player_name, players[player_turn].color)}"
            f"'s turn: "
        )
```

Figure 8.1: game_in_progress part 1

```
    # Input loops until the player makes a valid move"
    while True:
        if re.fullmatch("^([1-7]|clear|surrender)$", player_command.lower()):
            # If it's a digit between 1-7, within game columns limit,
            # drop is attempted
            if player_command.isdigit():
                try:
                    players[player_turn].drop(board, int(player_command))
                # If column is full, piece drop and input is refused.
                except ColumnFullError as error:
                    player_command = input(error)
                # If piece drop is successful, alternate player turn and
                # increase move count by 1.
                else:
                    player_turn = int(not player_turn)
                    move_count += 1
                    break
            # If "clear", game is reset. No wins are recorded.
            elif player_command.lower() == "clear":
                board.clear_board()
                player_turn = 0
                move_count = 0
                break
            # If "surrender", the player's surrender state is turned on,
            # and the player loses the game.
            elif player_command.lower() == "surrender":
                players[player_turn].surrender = True
                reset_screen(board)
                # Returns the winning Piece and the surrendering Piece,
                return referee.check_victory(), players[player_turn]
        else:
            player_command = input("Invalid input, please try again: ")

    reset_screen(board)
    # Returns referee's result (None if no winners hence draw), and surrendering
    # Piece (which is None) as a tuple.
    return referee.check_victory(), None
```

Figure 8.2: game_in_progress part 2

The game engine, represented as the whole module of game_engine.py, is basically a series of functions that compiles of all the methods and objects involved in the game and defines how the main phases of the game works. It consists of four main functions, namely game_start, game_in_progress, game_complete, and game_reset. With the time constraints, it would not be feasible to describe all functions in detail, but I would like to look at the most important one, which is game_in_progress. This function defines what happens during the major portion of the game. First, two variables are set: player_turn representing which player's turn it is, and move_count representing how many turns has it been since the start of the game. The game is in a while loop, looping until referee have detected one of the winning conditions, or move_count has reached 42, meaning a draw. After the screen is reset (the old board state cleared and the new board state displayed), a player is prompted to input a command. Player_turn alternates between 0 and 1 each turn, 0 representing player 1's turn and 1 representing player 2's turn. Another while loop is created after, which loops as long as a player's command is invalid, and breaks if it is valid and satisfy certain conditions. If it is a column input, and column is not full and between 1-7, the piece is dropped unto the board, player_turn alternates, move_count increases, and this child loop is broken, looping back from the start of the parent loop. If command is "clear", the board is cleared (all elements in the array becomes "0"), move_count

and player_turn is reset. If “surrender” is called, the game immediately ends, and the function returns both winning Piece and surrendering Piece. Otherwise, the game loops until a winner is detected by the referee, after which the function will return the winning Piece and the surrendering Piece (as None, since no one surrendered).

Another feature to note is in the game_complete function which is executed after game_in_progress, game records are updated and stored in the JSON file based on who wins, who loses, and whether it is a draw.

Game Hub Classes

Figure 9.1: GameHub enter_logic method

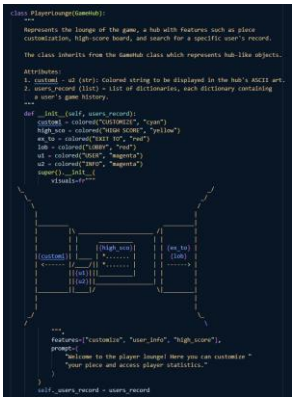


Figure 9.2: PlayerLounge class

Figure 9.3: MainLobby class

The GameHub class is the parent class that defines how hub-like objects behave. Again, due to time constraints, I'm going to explain only its most important method, which is the enter_logic method. It describes the logic of entering a hub and accessing its features. As a user enters a hub, the code will loop as long as the users are still accessing a feature. When exit command is entered while they are in the hub, the loop will break and "exit" is going to be returned, and then there will be a custom command depending on the hub they are in. If they are in the main lobby, "exit" will exit the user from the game. If they are in the PlayerLounge, "exit" will move them back to the MainLobby. When they are accessing the features, with commands defined by the feature dictionary, the code will loop as long as they are still inputting commands that access the feature. For example, in the Player Lounge's high-score board feature, this includes inputting the sorter keyword such as "games_played" or "wins". When "exit" is inputted, they will be moved back to the hub they are in.

The PlayerLounge and MainLobby are their own classes that inherit methods and attributes from the GameHub class. The commands, features list, ASCII art, and other methods will be customized according to the unique behaviour of each class inherting from the GameHub class.

Data Storage in JSON file

```
[
    {
        "username": "tester123",
        "pin": "1923",
        "games_played": 5,
        "wins": 2,
        "losses": 1,
        "win_ratio": 60.0,
        "color": "black",
        "piece_type": "Z",
        "logged_in": "n"
    },
    {
        "username": "tester321",
        "pin": "1009",
        "games_played": 2,
        "wins": 1,
        "losses": 0,
        "win_ratio": 75.0,
        "color": "light_yellow",
        "piece_type": "B",
        "logged_in": "n"
    }
]
```

Figure 10.1: Sample contents of users.json

A list of dictionaries, each dictionary containing the following key-value pairs:

- username
- pin
- games_played
- wins
- losses
- win_ratio
- color
- piece_type
- logged_in

users.json is basically a list of dictionaries, each dictionary element representing data of a unique user. All key-value pairs are basically as the name suggests. A specific key-value pair I want to highlight is the logged_in key. This key informs whether or not the user is currently logged in. If they are, subsequent login attempts are prohibited. This ensures that the two players playing the game are not using the same account, which would break the game logic. On each initialization and termination of the program, the user login is reset, meaning that all accounts are automatically logged off.

Testing

```python
def test_game_in_progress_p1wins_horizontal_victory(monkeypatch):
    """
    Tests the following sequence during game_in_progress: players 1 and 2
    putting their pieces on columns 1 - 4 alternatingly. This should result in
    player 1 horizontal victory.
    """
    # Define required variables
    player1 = Piece("Test1","red", "O", 1)
    player2 = Piece("Test2","blue", "X", 2)
    players = [player1, player2]
    board = Board(players)
    referee = VictoryChecker(board, players)

    # Sequence of user input to achieve the desired sequence of commands
    input_sequence = iter(["1", "1", "2", "2","3", "3", "4"])

    # Replaces default input function to enter input_sequence in sequence
    monkeypatch.setattr("builtins.input", lambda _: next(input_sequence))

    # Main function
    result = game_in_progress(board, players, referee)

    # Function should return player 1 as a winner, with no surrendered players
    assert result == (player1, None)
```

Figure 11.1: One of the tests from game_engine.py

```
test_report.txt
1  ============================= test session starts ==============================
2  platform linux -- Python 3.10.12, pytest-7.4.3, pluggy-1.3.0
3  rootdir: /home/deandresugandhi/projects/T1A3/DeandreSugandhi_T1A3/src
4  plugins: mock-3.12.0
5  collected 13 items
6
7  test_game_engine.py .....                                                [ 38%]
8  test_hubs.py ....                                                        [ 69%]
9  test_utilities.py ....                                                   [100%]
10
11 ============================== 13 passed in 0.16s ==============================
12
```

Figure 11.2: Successful test report

The test is executed for some of the most important functions and methods of the app, including those in game_engine.py, hubs.py, and utilities.py. Pytest is used to conduct the tests, and mocks are made using Pytest's moneypatch. Test files are commented to explain why and how the test is executed. Report on the successful test results can be found in test_report.txt in the root directory. Because the test depends on the user records stored in users.json as of the time of test, it might not work in the future if users.json has been modified such as by generating new users, changes in game records from playing matches, etc.

Development Process

Challenges
- Coming up with the general structure of the app
- Input validation
- Debugging

Ethical issues:
- Username and PIN storage, which is not encrypted as for now

Favourite parts:
- The GameHub and the object-oriented features in general

Firstly, I find one of the most challenging parts of the game to be coming up with the general structure of the app. As I progress through the app development, I find myself rewriting many parts of the code that I have written to ensure they work together well. One example is the creation of the GameHub class to be parent classes of PlayerLounge and MainLobby. Originally, PlayerLounge was an independent class, while MainLobby wasn't a class, but a function instead. However, after I noticed how the two behaves similarly, I decided to create the GameHub class for their shared behaviour. Because the inner workings of PlayerLounge and the main lobby function are already defined, I find it challenging to come up with how the GameHub class should work, and to change the code of PlayerLounge and main lobby function to fit this new idea. This was one of the most time-consuming process of the entire app development.

Input validation also takes a lot of time, because initially I want all input validation to share the same function. However, because of the different types of commands and behaviours of different player inputs (such as case sensitivity, password masking, regex matching, etc.), when I implemented conditionals to cover all these different behaviours, the unified input validation command becomes unintuitive to read (for me) and loses its robustness. Thus, I come up with a compromise, and created some

variations for the input validation function, such as validate_color, validate_username, etc. Also, in the middle of development, I realized that I should have raised errors and handle them instead when inputs are considered invalid, instead of purely using conditionals to prevent invalid input, so this is something I changed midway as well. Debugging is another time-consuming process, especially considering the different classes and functions that relates to one another; one error or code change on one part of the program causes unintended, unexpected consequences on the other parts.

Some ethical issues I encountered include the storage of user data in a JSON file. Because I don't really understand how encrypting works, and researching it made me quite confused, I decided to skip on this feature for now due to time and knowledge constraints. This was something that bothered me though, as the idea of having user PINS that can be easily accessed and very readable is an ethical problem and a huge limitation for the implemented user account system.

The GameHub, despite something I find challenging to implement, ended up being one of my favourite features of the app. I liked the idea that accessing in-game features is like how people access features in the physical realm; entering a space / room, looking at the features they like, and looking into it further should they be interested. It also allowed me to visualize the program in a clearer way. The same goes for the other object-oriented features of the program, such as Board and Piece class, which visualizes Connect Four not as a set of functions or rules, but as a set of "objects" doing different things such as the game board, the game pieces, and the referee.