TicTacToe - The game flow

Game controller

In the previous session, we built the data classes required by our tic-tac-toe game. We will now build the functionality required to play the game. Since we are building a console based game, we require a controller which will be responsible for taking input from the user and displaying the output.

> **In Python, we can use the `input()` function to take input from the user. The `input()` function takes a string as an argument which is displayed to the user. The user can then enter the input and press enter. The `input()` function returns the input entered by the user as a string.**

For now, we will keep it simple and only take the name, email and symbol of the user. To take input from the user, we can define the following method:

```
from typing import Tuple

def get_user_input() -> Tuple[str, str, Symbol]:
    user_name = input("Enter your name: ")
    user_email = input("Enter your email: ")
    user_symbol = input("Enter your symbol: ")
    parsed_symbol: Symbol = Symbol[user_symbol]

    return user_name, user_email, parsed_symbol
```

Apart from the `input` method, another thing worth noting is the `Tuple` type. A tuple is a collection of elements of different types. In the above method, we are returning a tuple of 3 elements - `user_name`, `user_email` and `parsed_symbol`. The `Tuple` type hint is defined in the `typing` module.

Once we have the user input, we can use it to create the `User` object and subsequently all the required objects for the game.

```
def create_game() -> Game:
    name, email, symbol = get_user_input()
    user = User(name, email)
    human = HumanPlayer(symbol, user)

    bot = BotPlayer(decide_bot_symbol(symbol), DIFFICULTY_LEVEL)

    board = Board(GAME_SIZE)
    return Game(0, board, [human, bot])

def decide_bot_symbol(user_symbol: Symbol) -> Symbol:
    return Symbol.X if user_symbol == Symbol.O else Symbol.O
```

The `create_game` method takes the user input and creates the `User` and `HumanPlayer` objects. It also creates the `BotPlayer` object. The `BotPlayer` object is created by passing the symbol of the human player and the difficulty level. The `decide_bot_symbol` method is used to decide the symbol of the bot player. The `create_game` method also creates the `Board` object and returns the `Game` object.

> **When using the `dataclass` decorator with an inherited parent class, the required parent attributes are the first arguments to the constructor. You can also use the named parameters to pass the arguments, which reduces the need for a builder method.**
>
> **`user = User(name="Tantia Tope", email="t@t.com")`**

Game class

We had previously defined the `Game` class from our class diagram as follows:

```
@dataclass
class Game:
    current_player_index: int
    board: Board
    players: List[Player] = field(default_factory=list)
```

The `start` method

We also need to add method to the `Game` class to maintain the lifecycle of the game. Let us start with adding the `start` method, that will be called from the game controller. To start the game, we need to implement the following steps:

1. Randomly select the first player from the list of players
2. Set the `current_player_index` to the index of the first player
3. Set the game status to be in progress

First let us define a status enum for the game:

```
from enum import Enum

class GameStatus(Enum):
    IN_PROGRESS = 1
    FINISHED = 2
    DRAW = 3
```

Now we can define the `start` method:

```
def start(self):
    self.current_player_index = random.randint(0, len(self.players) - 1)
    self.status = GameStatus.IN_PROGRESS
```

The `random.randint` method is used to generate a random integer between the given range. Here, we generate a random integer between 0 and the length of the list of players. We then

set the `current_player_index` to this random integer.

We can also define a `get_current_player` method to get the current player:

```
def get_current_player(self) -> Player:
    return self.players[self.current_player_index]
```

The game loop

Let's also think about the other steps of the game lifecycle.

1. The game asks the user for their details and creates the game objects
2. The game starts
3. The game asks the current player to make a move. If the current player is a bot, make a bot move
4. The game checks if the move is valid
5. If the move is valid, the game updates the board
6. The game checks if the user has won or the game is a draw
7. If the game is not over, the game switches the current player and goes to step 3
8. If the game is over, the game displays the result and exits

Let us add some dummy method for the remaining steps in our `Game` class for now. We shall come back to these methods later.

```
from typing import Optional

@dataclass
class Game:
    current_player_index: int
    board: Board
    players: List[Player] = field(default_factory=list)
    status: GameStatus = GameStatus.FINISHED
    winner: Optional[Player] = None

    def start(self):
        self.current_player_index = random.randint(0, len(self.players) - 1)
        self.status = GameStatus.IN_PROGRESS

    def get_current_player(self) -> Player:
        return self.players[self.current_player_index]

    def play(self):
        pass

    def is_valid_move(self, move: Move) -> bool:
        pass

    def has_won(self) -> bool:
        pass

    def is_draw(self) -> bool:
        pass

    def get_winner(self) -> Optional[Player]:
        return self.winner
```

To implement the game flow, once the game has been created we will run an infinite loop. In each iteration, we shall check if any of the terminating conditions have been met. If they have, we shall break out of the loop. Otherwise, we shall ask the current player to make a move. So far, our game controller looks like this:

```
GAME_SIZE = 3
DIFFICULTY_LEVEL = Level.EASY


def get_user_input() -> Tuple[str, str, Symbol]:
    user_name = input("Enter your name: ")
    user_email = input("Enter your email: ")
    user_symbol = input("Enter your symbol: ")
    parsed_symbol: Symbol = Symbol[user_symbol]

    return user_name, user_email, parsed_symbol


def create_game() -> Game:
    name, email, symbol = get_user_input()
    user = User(name, email)
    human = HumanPlayer(symbol, user)

    bot = BotPlayer(decide_bot_symbol(symbol), DIFFICULTY_LEVEL)

    board = Board(GAME_SIZE)
    return Game(0, board, [human, bot])


def decide_bot_symbol(user_symbol: Symbol) -> Symbol:
    return Symbol.X if user_symbol == Symbol.O else Symbol.O


def main():
    print("Welcome to Tic Tac Toe!")
    # Take user input for player name, email and symbol
    game = create_game()


if __name__ == "__main__":
    main()
```

Let's add the start method and our game loop now:

```
def main():
    print("Welcome to Tic Tac Toe!")

    game = create_game()
    game.start()

    while game.status == GameStatus.IN_PROGRESS:
        current_player = game.get_current_player()
        print(f"Next turn: {current_player.symbol.name}")

        current_player.play()

        if (game.status == GameStatus.FINISHED):
            print(f"{game.get_winner().symbol} has won!")
            break

        if (game.status == GameStatus.DRAW):
            print("The game is a draw!")
            break
```

The `play` method

The `play` method is responsible for asking the current player to make a move. If the current player is a bot, we need to make a bot move. Otherwise, we need to ask the user to make a move. Let's start by implementing it in the human player. We first need an abstract method in the `Player` class since it will be implemented by both the `HumanPlayer` and `BotPlayer` classes.

```
from abc import ABC, abstractmethod
from dataclasses import dataclass

@dataclass
class Player(ABC):
    symbol: Symbol

    @abstractmethod
    def play(self, board: Board) -> Cell:
        pass
```

For the `HumanPlayer` class, we can simply ask the user to enter the row and column of the cell they want to mark. We can then return the cell object corresponding to the row and column entered by the user.

```
from dataclasses import dataclass

@dataclass
class HumanPlayer(Player):
    user: User

    def play(self, board: Board) -> Cell:
        row = int(input("Enter row: "))
        col = int(input("Enter col: "))
        return Cell(row, col)
```

Conclusion

In this session, we implemented the game controller and the game loop. We also added the `play` method to the `HumanPlayer` class. In the next session, we will implement the `play` method for the `BotPlayer` class, and also complete the remaining methods in the game flow.

> 💻 **You can find the associated code here.**