TicTacToe - The playing and winning logic

Bot playing strategies

So far, we have created a player class that has an abstract method `play`. The method is called by the game controller to ask the player to make a move. The `HumanPlayer` class implements this method by asking the user to enter the coordinates of the cell they want to mark.

```
@dataclass
class Player(ABC):
    symbol: Symbol

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

The `BotPlayer` class has to automatically make a move on the basis of the level of the bot. Since, there can be multiple different algorithms for the bot to make a move, we can abstract the logic using the strategy pattern. We can create a `BotStrategy` interface that has a single method `get_move` which returns the next move for the bot. We can then create different classes that implement this interface and provide different strategies for the bot to make a move.

```python
from abc import ABC, abstractmethod

class BotStrategy(ABC):
    @abstractmethod
    def get_move(self, board: Board, symbol: Symbol) -> Cell:
        pass
```

Let us create the easy strategy in such a way that the bot will randomly choose a cell to mark. First, we will identify all the available cells on the board, and then randomly choose one of them.

The `get_available_cells` method will look like this:

```python
@dataclass
class Board:
    size: int
    cells: List[List[Cell]] = field(init=False)

    ...

    def get_available_cells(self) -> List[Cell]:
        return [cell for row in self.cells for cell in row if cell.symbol is None]
```

> **List comprehensions are a very powerful feature of Python. They allow us to create a list in a single line of code. The syntax is as follows:**
>
> **`[expression for item in iterable]`**
>
> **The `expression` is evaluated for each item in the `iterable` and the result is added to the list. For example, the following code will create a list of squares of the numbers from 1 to 10.**
>
> **`squares = [x * x for x in range(1, 11)]`**

Now, we can create the `RandomBotStrategy` class that implements the `BotStrategy`

interface. The `get_move` method will look like this:

```
from random import choice

class RandomBotStrategy(BotStrategy):
    def get_move(self, board: Board, symbol: Symbol) -> Cell:
        available_cells = board.get_available_cells()
        return choice(available_cells)
```

The `choice` method from the `random` module returns a random element from the list passed to it.

Now, we can create the `BotPlayer` class that will use the strategy to make a move. The `play` method will look like this:

```
@dataclass
class BotPlayer(Player):
    user: User
    strategy: BotStrategy

    def play(self, board: Board) -> Cell:
        return self.strategy.get_move(board, self.symbol)
```

The `play` method

Now that we have the playing functionalities implemented for each type of the player, we can implement the `play` method in the `Game` class. The steps to play a move are as follows:

1. Get the current player
2. Ask the player to make a move
3. Validate the move
4. If the move is valid, mark the cell on the board
5. Check if the game has been won or is a draw
6. If the game is won or a draw, set the game status
7. If the game is not over, switch the current player

The `play` method will look like this:

```
def play(self):

    current_player = self.get_current_player()
    move: Cell = current_player.play(self.board, self.current_player.symbol)
    validate_move(move)

    board.update(move)

    if self.has_won():
        self.status = GameStatus.FINISHED
        return

    if self.is_draw():
        self.status = GameStatus.DRAW
        return

    self.current_player_index = (self.current_player_index + 1) % len(self.players)
```

Here, we have used the `%` operator to switch the current player. The `%` operator returns the remainder of the division of the first operand by the second. The remainder will always be less than the second operand. So, we can use this to switch the current player.

Winning strategies

Similar to the bot playing strategies, we can also create different strategies to check if the game has been won. We can create a `WinningStrategy` interface that has a single method `has_won` which returns a boolean value indicating if the game has been won. We can then create different classes that implement this interface and provide different strategies.

```
from abc import ABC, abstractmethod

class WinningStrategy(ABC):
    @abstractmethod
    def has_won(self, board: Board, symbol: Symbol) -> bool:
        pass
```

Let us create the `RowWinningStrategy` class that checks if any row has been completely

marked by the given symbol. The `has_won` method will look like this:

```
class RowWinningStrategy(WinningStrategy):
    def has_won(self, board: Board, symbol: Symbol) -> bool:
        for row in board.cells:
            if all(cell.symbol == symbol for cell in row):
                return True
        return False
```

The `all` method returns `True` if all the elements of the iterable are `True`. Otherwise, it returns `False`.
We can similarly create the `ColumnWinningStrategy` and `DiagonalWinningStrategy` classes.

One small difference between the `BotPlayer` and the `WinningStrategy` classes is that the `Game` class has to check if any of the strategies has won the game. So, we need to maintain a list of strategies in the `Game` class. The `has_won` method will look like this:

```
@dataclass
class Game:
    ...
    winning_strategies: List[WinningStrategy] = field(default_factory=list)

    ...

    def has_won(self) -> bool:
        for strategy in self.winning_strategies:
            if strategy.has_won(self.board, self.current_player.symbol):
                return True
        return False
```

Checking for a draw could be a strategy as well, but we will keep it simple for now and implement it in the `Board` class. A game is a draw if all the cells on the board have been marked and no player has won.

Assuming that the `has_won` method is called before the `is_draw` method, we can check if any cell has the symbol as `None`. If it does, then the game is not a draw. Otherwise, the game is a draw. The `is_draw` method will look like this:

```
def is_draw(self) -> bool:
    return len(self.board.get_available_cells()) == 0
```

Conclusion

We have successfully implemented TicTacToe end to end. Here are the things we implemented:

1. Data classes for the entities
2. A game controller to take input from the user and play the game
3. A human player to take input from the user
4. Strategies for the bot to play the game
5. Strategies to check if the game has been won
6. A game class to orchestrate the game

Some of the things that we can do to improve the game are:

1. Take more input from the user like the size of the board, the number of players, the difficulty level of the bot, etc.
2. Add more strategies for the bot to play the game
3. Add more strategies to check if the game has been won i.e. column and diagonal
4. Implement the undo functionality

💻 You can find the associated code here.