Assignment 3

Software Development 2022

Department of Computer Science
University of Copenhagen

Aditya Fadhillah <hjg708@alumni.ku.dk>

Friday, March 08, 21:30

Introduction

In the assignment that has been given to me I am required to go through testing and implementing the missing functionality of the game TicTacToe. For this assignment I have used Visual Studio Code on the .NET Core version. The missing functionalities is located in the TicTacToe file where the actual game is run and in the TicTacToeTest file where the various methods of the main file are tested. This assignment focuses on a test-driven approach, that means I need to write tests for the missing methods before actually implementing the methods themselves. For my first task I need to extend the code for the movements of the cursor, the task consists of five functions and their tests. For my second task I need to extend the code for the ending condition of the game.

User's Guide

Assuming that the user received the file as a .zip file, they would first need to extract the contents of the .zip file. After extracting the files, the user can run the game TicTacToe by go to the TicTacToe file directory in their terminal, and then type in: 'dotnet run'. If all goes well the user will then see a 3 x 3 board. This board is where the game will be played. The cross player will always start first. For the cross player to choose a location they will be using: 'w','a','s' and 'd'. For the naught player it is: 'i','j','k' and 'l'. And to confirm a location and ending a move press: 'spacebar'. The game will end when a player have made a line with three of their symbols. The line can be horizontal, vertical and diagonal. When a player have achieved this the screen will display: '⟨ player symbol ⟩ Wins!'. When the board is full be there are still no winner the screen will display: 'The game was a tie!'.

Implementation

Implementing codes for methods in Cursor.cs

I will be using MoveUp() as an example of the implementation of move methods in Cursor.cs. The MoveUp() method works very simple. It checks if the Y value of the cursor is larger than the minimum value of the board. If it is true the Y value of the cursor is reduced by one. The other move methods will function similarly to MoveUp(). The other part is to implement MoveCursor(Input input)-function. I solved this task by using a Switch-statement. I made it so the InputType.PerformMove case returns False and all other cases return True.

```
public void MoveUp() {
    if(Y > min) {
        Y -= 1;
    }
}
```

Implementing codes for methods in BoardChecker.cs

The method IsRowWin will be an example on how I have implement the winning conditions in BoardChecker.cs. Because the size of the board is a changeable variable it will not be wise to compare all the possible position the symbols can be placed. Instead I used for-loop to check whether

or not the board have a line with the length that is equal to board.Size, and also consists of identical non null PlayerIdentifier. In the first for-loop when j increase by 1 it means the code moves down 1 row. In the second for-loop when i increase by 1 it means the code moves 1 column to the right. So the code scans the board from top left to bottom right. When a position have a symbol the counter for that symbol goes up by one. In IsRowWin when a counter variable for PlayerIdentifier have the same value as board.Size, IsRowWin will return true.

```
private bool IsRowWin(Board board) {
        for(int j = 0; j <= (board.Size - 1); j++ ) {
            var x = 0;
            var o = 0;
            for(int i = 0; i <= (board.Size - 1); i++ ) {
                if(board.Get(i,j) == PlayerIdentifier.Cross){
                    x++;
                }
                else if(board.Get(i,j) == PlayerIdentifier.Naught){
                    o++;
                }
            }
            if (board.Size==x || board.Size==o) {
                return true;
            }
        }
        return false;
    }
```

Implementing codes for CheckBoardState method

The other part of the task in BoardChecker.cs is to implement the CheckBoardState(Board board) method. This method changes the BoardState to Winner if a win condition if fulfilled. In other word when IsRowWin(), IsColWin() or IsDiagWin() returns true. CheckBoardState() changes to Tied if the board is full without a winner, otherwise it will return BoardState.Inconclusive.

```
public BoardState CheckBoardState(Board board) {
        if(IsRowWin(board) || IsColWin(board) || IsDiagWin(board)) {
            return BoardState.Winner;
        }
        else if(board.IsFull()) {
            return BoardState.Tied;
        }
        else {
            return BoardState.Inconclusive;
        }
    }
```

Evaluation

In CursorTest.cs I tests whether or not the move methods I have implemented actually changes the position of the cursor in the game. I implemented tests for all the move methods, and they all passed. This means that the move methods that I have implemented changes the position of the cursor.
In BoardCheckerTest.cs I tests for the state of the game and checks whether or not it matches with the desired board state. The first line of the test consists a Setup(). This Setup() creates a new board with its size set to 3 and a new boardchecker. In this empty board I begin to fill the necessary positions with symbols according to that specific test. I accomplish this by using TryInsert() method that is located in Board.cs. This method lets me insert a symbol in the desired position. With Assert.AreEqual() I can compare whether or not CheckBoardState(board) will have the same value as that test desired BoardState. If the test passed that means the method for BoardChecker is coded correctly.
So in DiagonalWinTest() I insert the naught symbol so that it makes a diagonal line across the board from top left to bottom right. For RowWinTest() I insert the cross symbols in the middle row. And for ColumnWinTest() I insert the cross symbols in the middle column. In all three tests I then compare the board's BoardState with BoardState.Winner. For InconclusiveTest() I insert two naught and one cross symbol in the top row. I compare the board's BoardState with BoardState.Inconclusive. For TiedTest() I insert all the possible position in the board without making any line with three identical symbols. I compare the board's BoardState with BoardState.Tied. All of the tests that I implemented passed. Other than the five obligatory tests that I need to implement in BoardCheckerTest, I also add three new tests. The three tests is for the opposite PlayerIdentifier winning condition, this is to test that both player type can win the game.

Conclusion

With the implementation I have applied to the files TicTacToe and TicTacToeTest. In Cursor.cs I implemented the methods that made it possible for me to move the cursor to select a position. In BoardChecker.cs I implemented various winning condition for the game, and also a CheckBoardState() method that can change BoardState to Winning, Tied or Inconclusive. With the help of the tests I implemented in CursorTest and BoardCheckerTest I have made it so that I can run and end the game without issues and according to the specification that have been given to me.