Software Development, Assignment 3

Lauritz Andersen, nmd988

February 25, 2022

```
(* MoveUp()-Method *)
public void MoveUp() {
        if (this.position.Y > this.min) {
            this.position.Y--;
        }
    }
```

Figure 1: MoveUp()-Method.

1 Introduction

This report covers the design and implementation of the well-known game Tic-Tac-Toe. A large part of the code, with some missing parts was given beforehand. The aim of the assignment was then to implement these missing parts utilizing test-driven-development. The solution was made in C in Visual Studio Code, .NET Core version. A part of the solution was to implement different methods such that the Cursor and BoardChecker-classes were functional. The Cursor-class had to be able to be moved around the board without going out of bounds. This was implemented with four methods MoveUp, MoveDown, MoveLeft, MoveRight, which makes moving the cursor accessible in all directions, which were then implemented in MoveCursor-method. The aim of the BoardChecker-class was to check whether there was a winner on the Board. It was implemented by checking whether there is a row, column or diagonal win on the board. If neither of those are apparent, the game is either inconclusive or tied. When both of these classes were functional, they had to be tested thoroughly. All possible test-cases were tested for a 3x3 board in terms of ways to win. However for the inconclusive board, only a few cases were tested.

2 User's guide

To run the tests, the User must locate the TicTacToe and the TicTacToeTest folder. The user must open them in several windows in Visual Studio Code. The user can then in the folder of the TicTacToeTest open a terminal and run "dotnet test". The user will then see that the 12 different tests have been passed.

3 Implementation

To solve the Cursor-problem, the positions of the X and Y-values had to be mutated, according to which direction the Cursor had to go to. On top of this, there also needed to be a conditional statement such that the Cursor would not go out of bounds. The solution to this was to make an if-statement with the condition that as long as the Cursor's position is larger than the minimum possible size of the board, the Position of Y will be decremented by 1 and thereby the Cursor will move up. See the solution in Figure 1. When this part was solved, the next methods were quite uncomplicated to code as they followed the same logic. If this.Position.Y did not exceed this.max, then this.Position.Y would be incremented by one. For X this was the same condition, where as MoveLeft() would be decremented, if Position.X was larger than this.max and MoveUp() would be incremented if it did not exceed this.max, and in that sense the cursor problem was solved.

Next all these methods had to be collected into a final method called MoveCursor(), which was achieved with implemented a switch-statement. In case of the input type being PerformMove(), MoveCursor() should return false. If however, the type was of Up, Down, Left, Right, the corresponding MoveUp()/Down()/Left()/Right() should be called and then return true. Exit and Undefined should also return true. See Figure 2 for the code.

Next the functions Row-win, Col-win and Diagonal-win had to be implemented. Row-win and Diagonal-win will be demonstrated, as Col-win is fundamentally the exact same solution as Row-win.

To implement Row-win, the game-board had to be traversed through in terms of row and cols. The most straightforward approach was to implement a double for-loop, such that it was possible to go through each row for each col and check if there should be a win. To do this an if-condition was implemented that "disproves" a win. It does so by comparing board. Get with board.Get(x + 1, y), meaning that it always checks two fields at a time. If the two are different from each other, we already know there can be no win. E.g if there is a Naught and a Cross in the same row, we instantly know there cannot be

```
(* MoveCursor()-Method *)
private bool MoveCursor(InputType inputType) {
    switch (inputType) {

        case InputType.PerformMove:
            return false;

        case InputType.Up:
            this.MoveUp();
            return true;

        ...

        default:
            throw new Exception("Invalid␣Input␣type.....")
    }
}
```

Figure 2: MoveCursor()-Method. (Snippet)

```
(* IsRowWin()-Method *)
private bool IsRowWin(Board board) {
    for (int y = 0; y < board.Size; y++) {
        bool win = true;
        for (int x = 0; x < board.Size - 1; x++) {
            if (board.Get(x, y) == null || board.Get(x, y) != board.Get(x + 1, y)) {
                win = false;
                break;
            }
        }
        if (win) {
            return true;
        }
    }
    return false;
}
```

Figure 3: IsRowWin()-Method.

a full row of Crosses. The condition also checks if board.Get(x,y) is == null, because if this is the case, obviously there is no win. See figure 3 for the solution.

To implement diagonal win, the coding was a bit trickier. There were two cases in which a diagonal win could occur. 1) // From top left to bottom right. And 2) From top right to bottom right. As opposed to Row and Col-win, here there was no need to make a double for-loop. Instead the cases here were explored separately. For case 1), it follows a similar approach as RowWin, but checks for $board.Get(i+1, i+1)$), as the coordinates for a diagonal win from top left to bottom right would be (0,0) (1,1), (2,2). For the second case, we check for board.Get(x - 1, y + 1)), as we have to go one x-value left and one y-value up as to go in the diagonal direction. The code is a bit too comprehensive to demonstrate in its full form, so the second part is left out here. See figure 5

Then to make the CheckBoardState, check for a win, a simple If-Elif-Else-structure was implemented. This would check if there is each of the three winning types, the player has won, and if there isn't, there is either a tie or inconclusive result. A helper function "IsTied" was implemented. This checks if the board was full and if none of the winning coincidences has occurred yet. If this was is case, the game is tied. See figure: 4

```
(* CheckBoardState-Method *)
public BoardState CheckBoardState(Board board) {
      if (this.IsRowWin(board) || this.IsColWin(board) || this.IsDiagWin(board)) {
         return BoardState.Winner;
      } else if (this.isTied(board)) {
         return BoardState.Tied;
      } else {
         return BoardState.Inconclusive;
      }
   }
}
```

Figure 4: CheckBoardState Method.

```
(* IsDiagWin()-Method *)
private bool IsDiagWin(Board board) {
      // From top left to bottom right
      bool win = true;
      for (int i = 0; i < board.Size - 1; i++) {
         if (board.Get(i, i) == null || board.Get(i, i) != board.Get(i + 1, i + 1)) {
            win = false;
            break;
         }
      }
      if (win) {
         return true;
      }
      //From top right to bottom right
      (left out here)
```

Figure 5: DiagWin()-Snippet

4 Evaluation

For the testing, all possible causes of Row, Column and Diagonal win for a Board of size 3 for each Player Identifier was tested. For instance for a row, for each column, every row was tested for both Naught and Cross. All these passed. Apart from this, three Inconclusive configurations of the board were tested. One where the board was empty, one with only two pieces on the board and one with only one piece on the board. All tests passed, however there could have been run more tests in order to make testing more comprehensive. Whether the game was a tie was only tested with one example, where the board was full with different configurations of Naught and Crosses, but there was no diagonal, row or column-win. This test also passed.

5 Conclusion

The TicTacToe game was implemented succesfully and the comprehensive testing succeeded. The missing methods in the Cursor-class was implemented succesfully, and the tests passed such that the Cursor is able to be moved around the board without going out of bounds. The BoardChecker-class, that checks whether there was a winner, or whether there is a tie or inconclusive result, was implemented succesfully as well and tested thoroughly. Test-driven-development was a good tool for this assignment, as it made possible always to track the development of the project. The tests were always written before the code itself, and in this sense, especially in Row/Col/DiagWin(), it was advantageous. The tests would fail at first, however, being able to visualize what the outcome of the method should be, and which results should outcome in wins (specifically, which combinations of rows and cols should be made), it was extremely helpful when creating the for-loops and conditional statements of the methods.