

Assignment01 Final

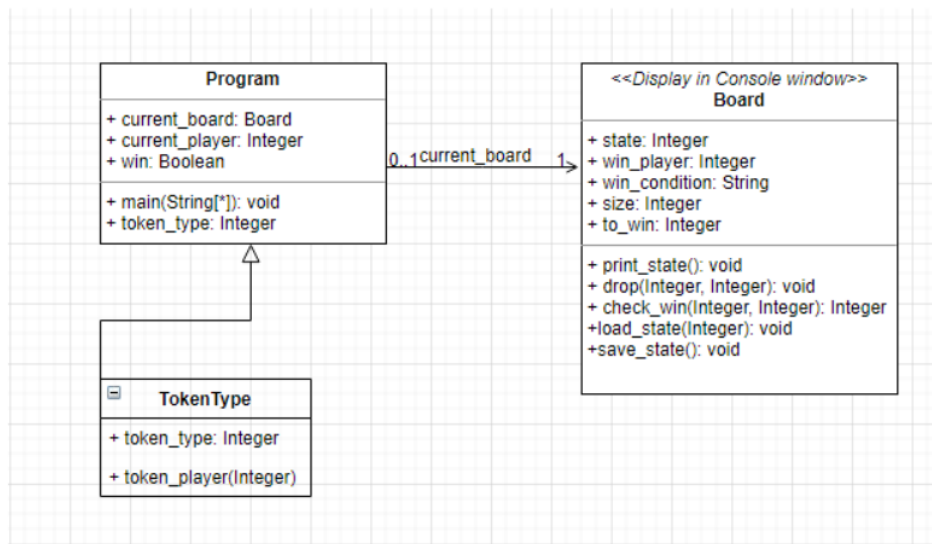
by Christopher Dean Jones

FILE	14910_CHRISTOPHER_DEAN_JONES_ASSIGNMENT01FINAL_339836_92254333.DOCX (1.38M)		
TIME SUBMITTED	15-APR-2021 04:29PM (UTC+0100)	WORD COUNT	2005
SUBMISSION ID	149458313	CHARACTER COUNT	10275

Initial Concept Design

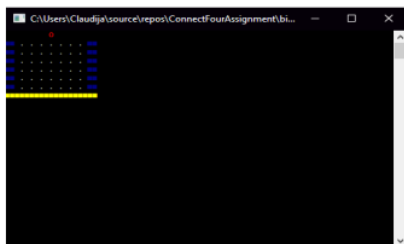
The first real difficulty with this assignment was starting. The task was in context, simple – design a potential code for a Connect 4 game that then needs to be tested to ensure the code would in practice if not actuality – work.

I approached the initial design by deciding on just two classes initially, the core Game Class that would create the overall game and a Board Class that would provide the rules for the board itself.



I came up with a very basic CRC setup that would evolve throughout the development, another obstacle I found – to outlay the main design. The concept is simplistic as I did not wish to complicate things, the code needs to be efficient to the End User and not full of confusing possibly redundant code. I set up the main two classes using a free online design tool called Draw.io from their website <https://app.diagrams.net/> and immediately noticed the first problem.

Whilst I had accounted for the win conditions and the board size, I had not anticipated the presentation. We were set a text-based assignment and yet mine developed into a console game with an interface –

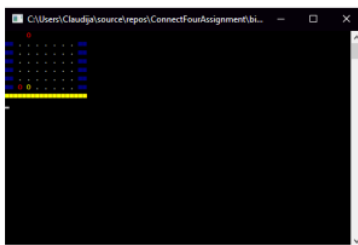


So that presented an adjustment to the CRC which I had not anticipated in that I now created a third class, 'TokenType' which would simply control the design of the virtual counters representing the

player pieces from the traditional game made by MB Games and others. To account for this I adjusted my CRD to the one presented here when I added the TokenType class to the overall design.

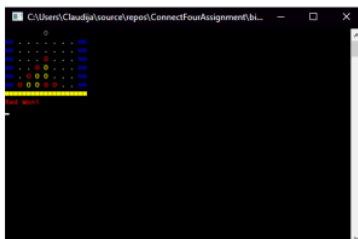
I found the CRC therefore, could be a useful guide but ultimately was not the overall guiding principle in the development of the program simply a means to demonstrate the core basic logic that would be applied in the final product.

Once the initial display was in place, the next step was to code in the logic. Whilst we are being taught a test-first thinking, I was in the old frame – build it then test. I applied natural black box testing without writing up formal tests. For example, deciding the board size. The classic game is traditionally 6 x 7 counters. I tried it with 9 but the board was too large and so set it at the classic. Then it was deciding on counter colours. I had tested varying colours including green, orange and even pink but felt to an end user it would be unfamiliar so set it back to classic red and yellow –

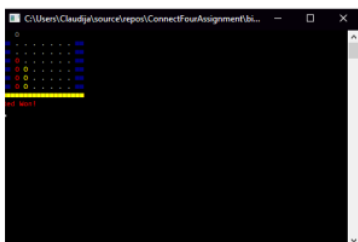


The board boundaries were again tested with varying colours from white to dark red but again I felt that if I were playing this myself, the familiarity of the original would be much more suitable than a customised layout. With the layout set it was on to the game rules themselves.

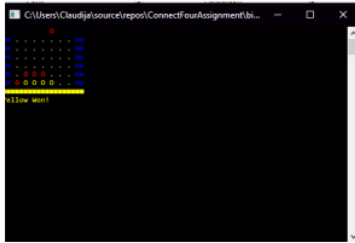
For this assignment we were tasked with developing the game and including new rules. In the classic version, the player wins by placing four counters of their colour in a row either diagonally –



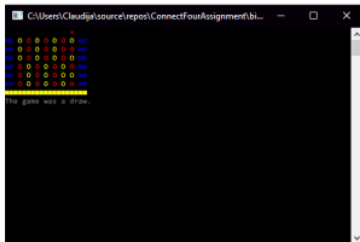
vertically –



or horizontally –



Very rarely a game could end in a draw –



The outcome of these tests during development, showed that my logic was working and my game would display the classic version of Connect 4; accounting for all three possible game outcomes.

Testing with XUnitTests

The tests I settled on were to ensure that the program would recognise the boundaries of the grid itself and that it would not accept counters being played outside those borders. First was to test the columns and the rows would not exceed the set values in the design. When these values were set as shown the program returned an error as expected when attempting to compile as the code was being given exceptions.

```
using System;
using Xunit;

namespace XunitConnectFour
{
    [References]
    public class UnitTest1
    {
        [Fact]
        [References]
        public void Test1()
        {
            private bool CheckHorizontally(int row, int column, TokenType type)
            {
                // If there aren't even four more spots before leaving the grid,
                // we know it can't be.
                if (column + 3 >= Columns) { return true; }

                for (int distance = 0; distance < 4; distance++)
                {
                    if (grid[row, column + distance] != type) { return true; }
                }

                return true;
            }
        }
    }
}
```

Each test I used a similar method to attempt to break the code by causing errors. The tests were as follows –

```
[References]
public void Test2()
{
    private bool CheckVertically(int row, int column, TokenType type)
    {
        // If there aren't even four more spots before leaving the grid,
        // we know it can't be.
        if (row + 3 >= Rows) { return true; }

        for (int distance = 0; distance < 4; distance++)
        {
            if (grid[row + distance, column] != type) { return true; }
        }

        return true;
    }
}
```



```
[References]
public void Test3()
{
    private bool CheckDiagonallyDown(int row, int column, TokenType type)
    {
        // If there aren't even four more spots before leaving the grid,
        // we know it can't be.
        if (row + 3 >= Rows) { return true; }
        if (column + 3 >= Columns) { return true; }

        for (int distance = 0; distance < 4; distance++)
        {
            if (grid[row + distance, column + distance] != type) { return true; }
        }

        return true;
    }
}
```

```

public void Test4()
{
    private bool CheckDiagonallyUp(int row, int column, TokenType type)
    {
        // If there aren't even four more spots before leaving the grid,
        // we know it can't be.
        if (row - 3 < 0) { return true; }
        if (column + 3 >= Columns) { return true; }

        for (int distance = 0; distance < 4; distance++)
        {
            if (grid[row - distance, column + distance] != type) { return true; }
        }

        return true;
    }
}

```

0 references

```

public void Test5()
{
    public bool IsFull()
    {
        for (int column = 0; column < Columns; column++)
        {
            if (CanAdd(column)) { return true; }
        }

        return true;
    }
}

```

0 references

```

public void Test6()
{
    public bool CanAdd(int column)
    {
        if (grid[0, column] == TokenType.Empty) { return false; }
        else { return true; }
    }
}

```

In each case, you can see that the code will fail and highlights errors which we then corrected in the final program code.

I needed to expand on my game to include the rules requested in the brief. If we look at Yellow's win, the brief is asking to have had Red win. Had Red gone to block as is traditional, all the yellow counters would be removed, dropping Red into a winning row of five counters. Despite multiple attempts at this, I did not find a resolution within the assignment duration. The base game is working properly, and I did alter the counters to be graphically more interesting and appeal to a younger player perhaps as the counters are now smiling faces instead of simple coloured "O" to represent counters. I am still looking into developing the brief's new rule to apply to the game alongside a rematch rule as currently any conclusion will result in the console requesting a termination of the window instead of simply offering a choice of rematch or closure.

Appendix

Here is the code I developed in conclusion. It does cover the core rules of the original game and has been edited to ensure the program does run as expected. I have covered the initial two User stories and subsequent patches would include the additional features noted above that I did not resolve within the allotted assessment period:–

Program.cs -

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConnectFourAssignment
{
    class Program
    {

        /// A console window displayed Connect 4 Game for 2 players.

        static void Main(string[] args)
        {
            Board board = new Board();

            int currentColumn = 3;

            TokenType type = TokenType.Red;

            /// Loop until one win condition is met.

            while (!board.IsWinner(TokenType.Red) &&
                !board.IsWinner(TokenType.Yellow) &&
                !board.IsBoardFull())
            {
```

```
/// Display the board in current state.

Console.Clear();

DrawHeader(board, currentColumn, type);

board.DrawBoard();

/// Either press a number 1-7 to select a column or

/// Pressing the Left and Right Arrow Keys will move the cursor one column in either
direction.

ConsoleKeyInfo key = Console.ReadKey();

if (key.KeyChar == '1') { currentColumn = 0; }

else if (key.KeyChar == '2') { currentColumn = 1; }

else if (key.KeyChar == '3') { currentColumn = 2; }

else if (key.KeyChar == '4') { currentColumn = 3; }

else if (key.KeyChar == '5') { currentColumn = 4; }

else if (key.KeyChar == '6') { currentColumn = 5; }

else if (key.KeyChar == '7') { currentColumn = 6; }

else if (key.Key == ConsoleKey.LeftArrow && currentColumn > 0)

{

    currentColumn--;

}

else if (key.Key == ConsoleKey.RightArrow && currentColumn < 6)

{

    currentColumn++;

}

else if (key.Key == ConsoleKey.Enter)

{

    /// Pressing the <Enter> key will verify token can be 'dropped' at current position.

    if (board.CanAdd(currentColumn))
```

```
        {  
            board.Add(currentColumn, type);  
            if (type == TokenType.Red) { type = TokenType.Yellow; }  
            else { type = TokenType.Red; }  
        }  
    }  
}
```

```
/// Win Condition is met, display final board.
```

```
Console.Clear();
```

```
DrawHeader(board, currentColumn, type);
```

```
board.DrawBoard();
```

```
PrintWinConditions(board);
```

```
Console.ReadKey();
```

```
}
```

```
/// Game Conclusion Confirmation Message.
```

```
private static void PrintWinConditions(Board board)
```

```
{
```

```
    if (board.IsWinner(TokenType.Red))
```

```
    {
```

```
        Console.ForegroundColor = ConsoleColor.Red;
```

```
        Console.WriteLine("Red Player Won!");
```

```
    }
```

```
    else if (board.IsWinner(TokenType.Yellow))
```

```

{
    Console.ForegroundColor = ConsoleColor.Yellow;

    Console.WriteLine("Yellow Player Won!");
}

else
{
    Console.ForegroundColor = ConsoleColor.DarkGray;

    Console.WriteLine("This game is a draw.");
}
}

/// Token placement methods to display visually to the player where their token will be placed.
private static void DrawHeader(Board board, int currentColumn,
    TokenType currentType)
{
    string symbol = board.CanAdd(currentColumn) ? "⬤" : "x";

    Console.ForegroundColor = currentType == TokenType.Red ?
        ConsoleColor.Red : ConsoleColor.Yellow;

    Console.WriteLine(" {0} {1} {2} {3} {4} {5} {6}",
        currentColumn == 0 ? symbol : " ",
        currentColumn == 1 ? symbol : " ",
        currentColumn == 2 ? symbol : " ",
        currentColumn == 3 ? symbol : " ",
        currentColumn == 4 ? symbol : " ",
        currentColumn == 5 ? symbol : " ",
        currentColumn == 6 ? symbol : " ");
}

```

```
}  
}
```

Board.cs –

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
  
namespace ConnectFourAssignment  
{  
    /// This helps to keep track of the state of a board and determines if a player has won.  
    public class Board  
    {  
        /// The number of rows for our board  
        public static readonly int Rows = 7;  
  
        /// The number of columns we need for our board.  
        public static readonly int Columns = 6;  
  
        /// Board contents.  
        private TokenType[,] board;  
  
        /// Create a new, empty board.  
        public Board()  
        {  
            board = new TokenType[Rows, Columns];  
        }  
    }  
}
```

```

        PopulateEmptyBoard();
    }

    /// Basic layout for a new board.
    private void PopulateEmptyBoard()
    {
        for (int row = 0; row < Rows; row++)
        {
            for (int column = 0; column < Columns; column++)
            {
                board[row, column] = TokenType.Empty;
            }
        }
    }

    /// Win Condition Methods to check which player has won the game.
    public bool IsWinner(TokenType type)
    {
        /// These methods verify starting point for win condition.
        for (int row = 0; row < Rows; row++)
        {
            for (int column = 0; column < Columns; column++)
            {
                if (CheckVertically(row, column, type)) { return true; }
                if (CheckHorizontally(row, column, type)) { return true; }
                if (CheckDiagonallyDown(row, column, type)) { return true; }
            }
        }
    }

```

```
        if (CheckDiagonallyUp(row, column, type)) { return true; }
    }
}

return false;
}

/// Checking for horizontal win condition.
private bool CheckHorizontally(int row, int column, TokenType type)
{

    if (column + 3 >= Columns) { return false; }

    for (int distance = 0; distance < 4; distance++)
    {
        if (board[row, column + distance] != type) { return false; }
    }

    return true;
}

/// Checking for vertical win condition.
private bool CheckVertically(int row, int column, TokenType type)
{

    if (row + 3 >= Rows) { return false; }
```

```
        for (int distance = 0; distance < 4; distance++)
        {
            if (board[row + distance, column] != type) { return false; }
        }

        return true;
    }

    /// Check for downward diagonal win condition.
    private bool CheckDiagonallyDown(int row, int column, TokenType type)
    {

        if (row + 3 >= Rows) { return false; }
        if (column + 3 >= Columns) { return false; }

        for (int distance = 0; distance < 4; distance++)
        {
            if (board[row + distance, column + distance] != type) { return false; }
        }

        return true;
    }

    /// Check for upward diagonal win condition.
    private bool CheckDiagonallyUp(int row, int column, TokenType type)
    {
```

```

        if (row - 3 < 0) { return false; }

        if (column + 3 >= Columns) { return false; }

        for (int distance = 0; distance < 4; distance++)
        {
            if (board[row - distance, column + distance] != type) { return false; }
        }

        return true;
    }

    /// Verify if the board is full.Determines if the board is full.
    /// A full board does not automatically guarantee a draw win condition.
    public bool IsBoardFull()
    {
        for (int column = 0; column < Columns; column++)
        {
            if (CanAdd(column)) { return false; }
        }

        return true;
    }

    /// Determines token placement and adds it if available.
    public void Add(int column, TokenType type)
    {
        /// If board is full terminate process.

```

```
if (!CanAdd(column)) { return; }

/// Check from the top number of rows token can 'drop'.
int currentRow = 0;
while (currentRow < Rows - 1 &&
    board[currentRow + 1, column] == TokenType.Empty)
{
    currentRow++;
}

/// Check passed, program will place current token in selected column.
board[currentRow, column] = type;
}

/// Verify a token can be added to the top of selected row.
public bool CanAdd(int column)
{
    if (board[0, column] == TokenType.Empty) { return true; }
    else { return false; }
}

/// Board display in console window.
public void DrawBoard()
{
    for (int row = 0; row < Rows; row++)
    {
        Console.ForegroundColor = ConsoleColor.DarkBlue;
```

```

Console.Write("■ ");

for (int column = 0; column < Columns; column++)
{
    TokenType type = board[row, column];

    switch (type)
    {
        case TokenType.Empty:
            Console.ForegroundColor = ConsoleColor.Gray;

            Console.Write("O ");

            break;

        case TokenType.Red:
            Console.ForegroundColor = ConsoleColor.Red;

            Console.Write("☹ ");

            break;

        case TokenType.Yellow:
            Console.ForegroundColor = ConsoleColor.Yellow;

            Console.Write("☹ ");

            break;
    }
}

Console.ForegroundColor = ConsoleColor.DarkBlue;

Console.WriteLine("■");
}

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("■■■■■■■■■■■■■■■■■■■■");

```

```
    }  
    }  
}
```

TokenType.cs –

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
  
namespace ConnectFourAssignment  
{  
    /// Controls the colours of tokens used in the game.  
    public enum TokenType  
    {  
  
        /// Player 1 Red Token.  
        Red,  
  
        /// Player 2, Yellow Token.  
        Yellow,  
  
        /// Empty Space, Unoccupied.  
        Empty  
    }  
}
```

Assignment01 Final

GRADEMARK REPORT

FINAL GRADE

43/100

GENERAL COMMENTS

Instructor

Eval design (10): 4

User Stories (20): 0

CRC (20): 8

SW Testing (30): 4

Coding (20): 16

Feedback:

=====

There was little evaluation included in the report. No user stories or analysis was provided, but a class diagram has been provided. Software testing is little strange, the test code has not been submitted and the code in the report shows compile time errors as a result of coding problems. I think the reference to the application has not been added to the test project and the namespace for the application is not visible at the top of the file.

PAGE 1

PAGE 2

PAGE 3

PAGE 4



Comment 1

These are compile time errors, not errors caused by running the code????

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17
