Lab Assignment 11

Lab Topic: Analyzing C#Console Games for Bugs

1. Introduction

This lab focuses on practical debugging skills within the Visual Studio environment using C# console game examples from the dotnet-console-games collection. The core activity involves utilizing the Visual Studio Debugger to analyze program execution flow, identify existing or injected runtime bugs, and implement code corrections, thereby enhancing understanding of fault diagnosis and code verification.

Environment Setup:

- Operating System: Windows
- Software: Visual Studio 2022 (Community Edition), Visual Studio with .NET SDK
- Programming Language: C# 13 or dotnet SDK 9.0.102

Key Concepts:

Runtime Error: An error that occurs only when the program is executing, often due to specific data, user input, or environmental conditions (e.g., NullReferenceException, DivideByZeroException). Contrasts with compile-time errors.

Logic Error: A flaw where the program compiles and runs without crashing but produces incorrect or unexpected results due to faulty reasoning or implementation in the code.

Mutation (Bug Injection): Intentionally introducing small changes (e.g., altering operators, constants) into working code to create bugs for the purpose of testing detection or practicing debugging.

Top-Level Statements: A C# feature (used in these projects) allowing executable code directly in Program.cs without the traditional Main method boilerplate, simplifying simple console applications.

---

2. Methodology and Execution

This is the file structure in the folder Lab9.

File Structure:

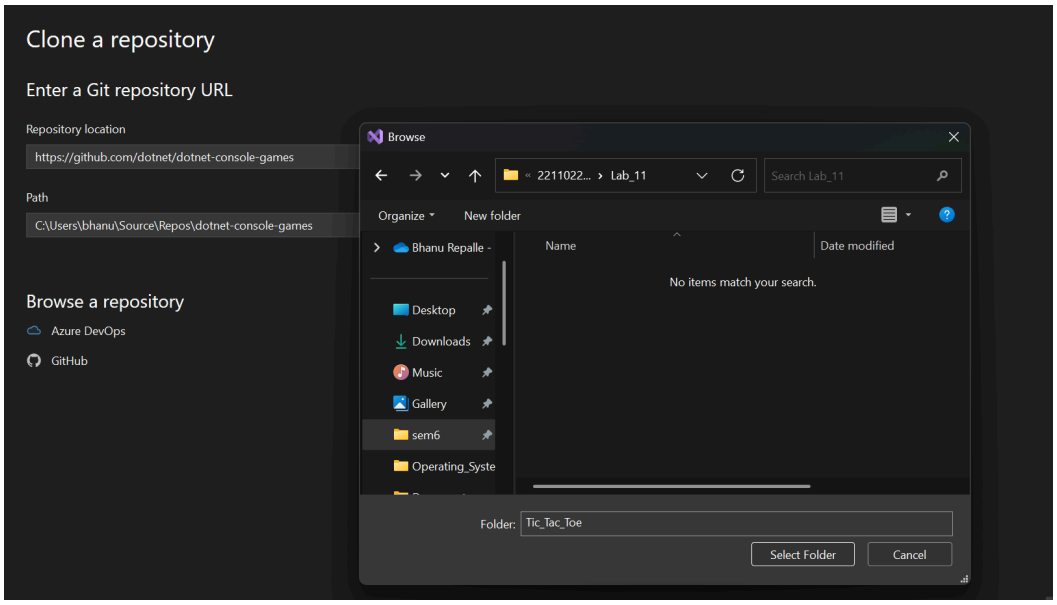
```
Lab11/
├── dotnet-console-games.sln # VS Solution file (groups all games)
├── dotnet-console-games.slnf# VS Solution filter file (optional view settings)
├── Directory.Build.props   # Common build properties (repo-level)
└── Projects/               # Folder containing individual game projects
    ├── Rock Paper Scissors/ # Project folder for Rock Paper Scissors
    │   ├── Program.cs      # Game logic and entry point
```

```
│   ├── README.md       # Specific game info (optional)
│   ├── Rock Paper Scissors.csproj # Project build settings
│   ├── bin/            # Compiled output folder
│   └── obj/            # Intermediate build files folder
│
└── Tic Tac Toe/        # Project folder for Tic Tac Toe
    ├── Program.cs      # Game logic and entry point
    ├── README.md       # Specific game info (optional)
    ├── Tic Tac Toe.csproj # Project build settings
    ├── bin/            # Compiled output folder
    └── obj/            # Intermediate build files folder
```

Below is the step-by-step procedure I followed.

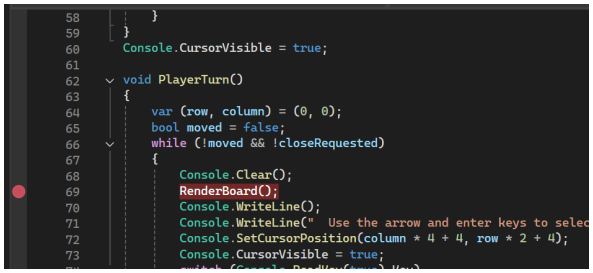
Step 1: Initial Setup

First, I cloned the projects folder.

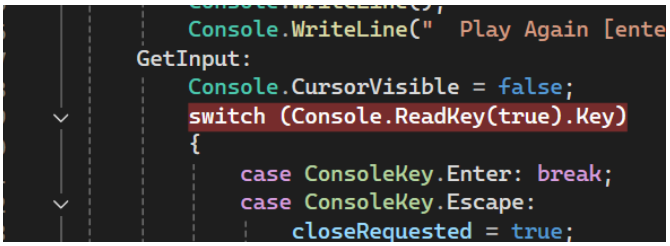


I chose the 2 games: Tic Tac Toe and Rock Paper Scissors. I then opened the folder with visual studio.

I started by inserting breakpoints at a few critical points and played the game.



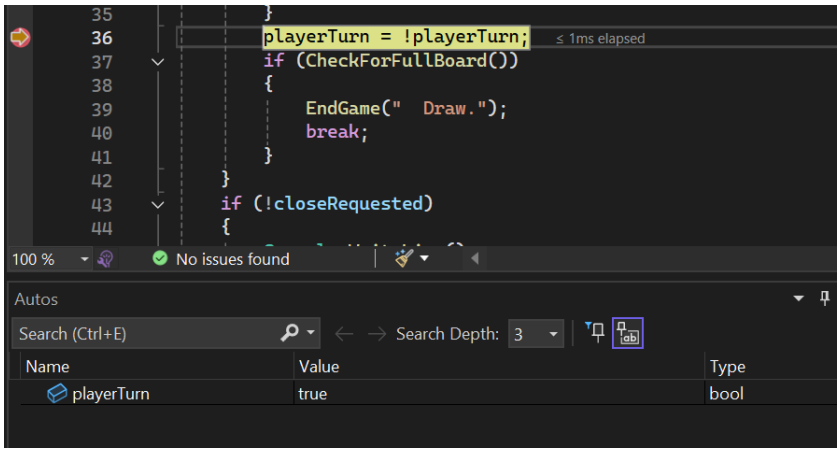
```
    }
}
Console.CursorVisible = true;

void PlayerTurn()
{
    var (row, column) = (0, 0);
    bool moved = false;
    while (!moved && !closeRequested)
    {
        Console.Clear();
        RenderBoard();
        Console.WriteLine();
        Console.WriteLine("  Use the arrow and enter keys to selec
        Console.SetCursorPosition(column * 4 + 4, row * 2 + 4);
        Console.CursorVisible = true;
```



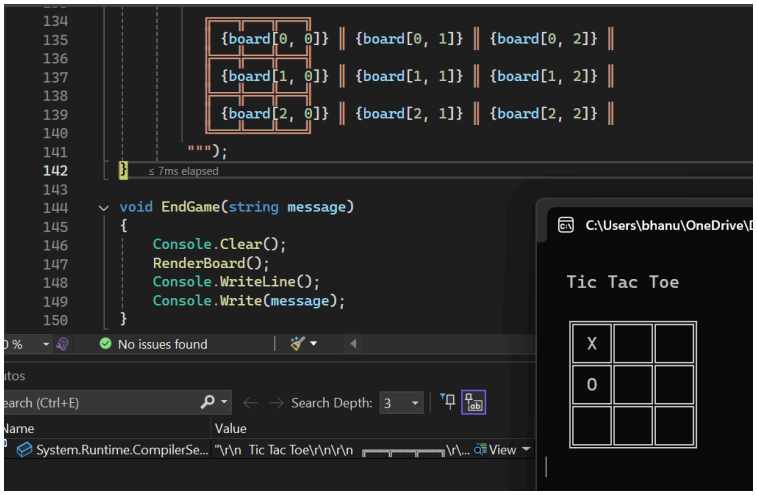
```
        Console.WriteLine("  Play Again [ente
GetInput:
    Console.CursorVisible = false;
    switch (Console.ReadKey(true).Key)
    {
        case ConsoleKey.Enter: break;
        case ConsoleKey.Escape:
            closeRequested = true;
```

Step 2: Analysing Tic Tac Toe

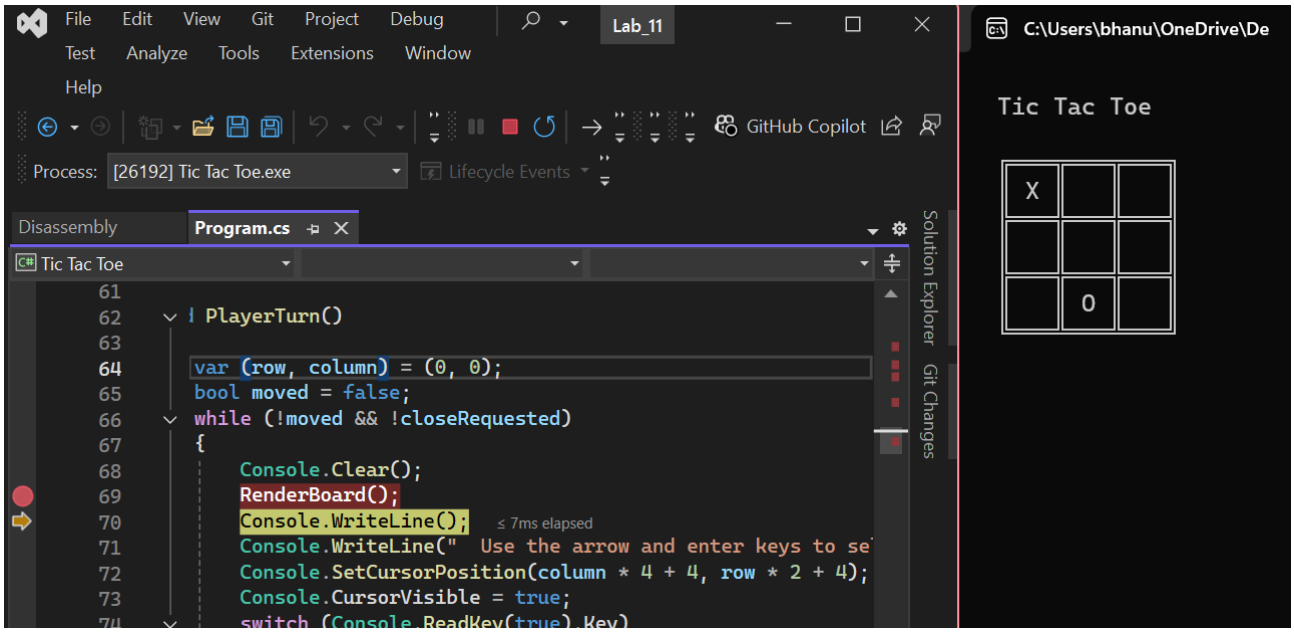
The input collection is very strict so it was not possible to give any invalid input. I played multiple rounds.
Below are a few screenshots of the output at break points, they are as expected.



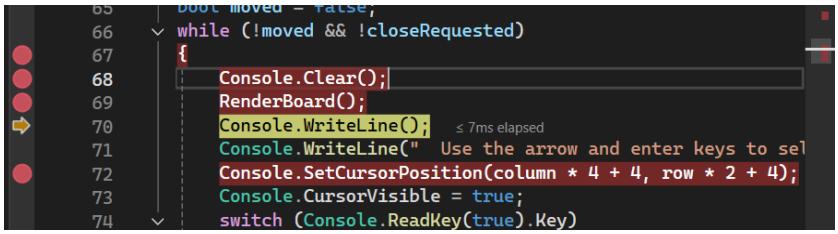
There was one issue which occurred which is not exactly the game logic but it was because of the interaction between console and the debugger and how screen clearing/cursor positioning works.
The cursor is out of the board



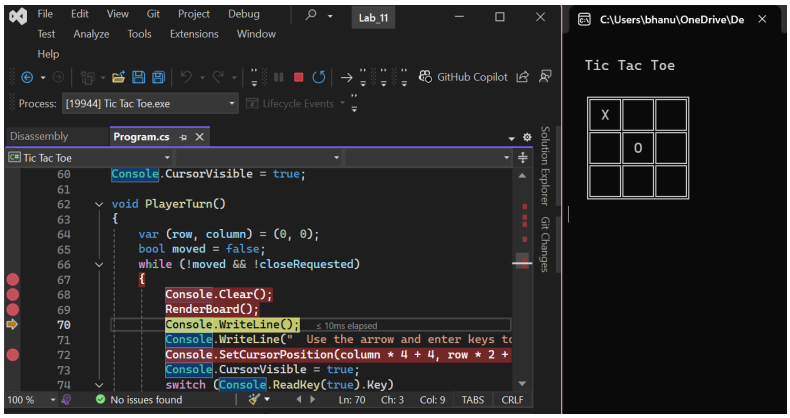
This is observed at the RenderBoard() breakpoint



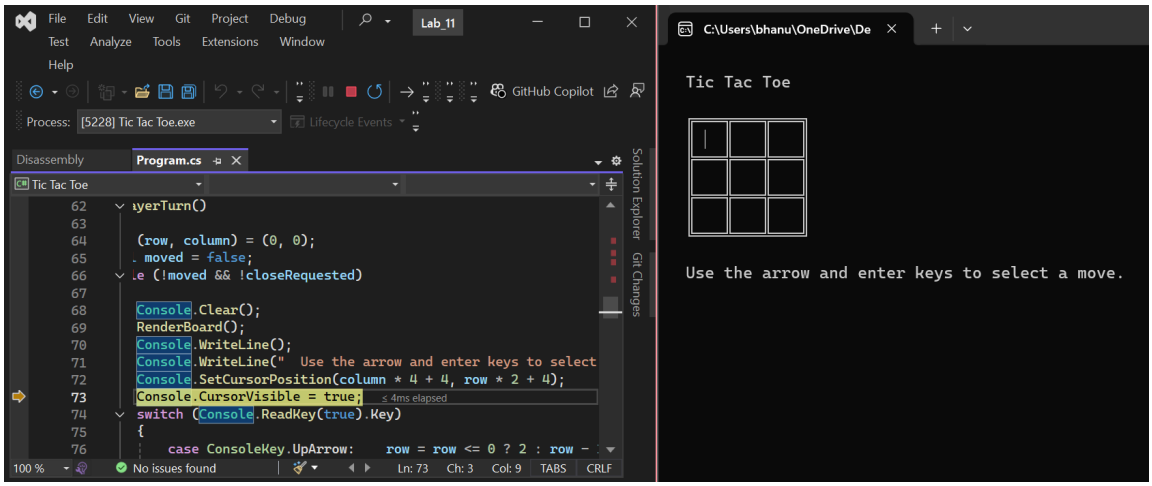
Debugging:
I set multiple bugs along the lines.

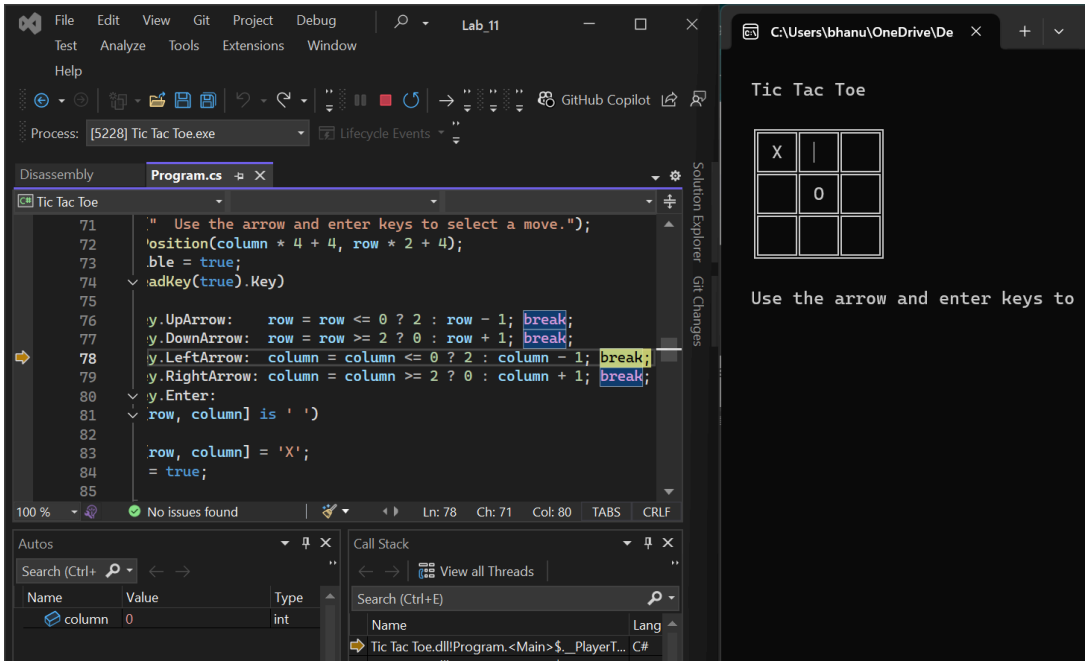


This is the exact position when the cursor is seen outside the board.



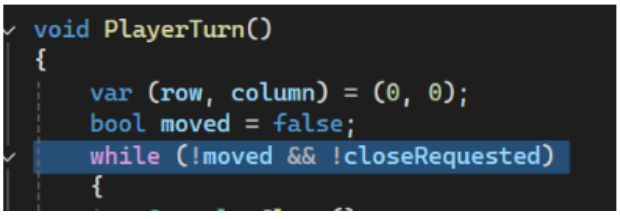
I believe that the error because of the selection of the break points and using step over
So removed all the breakpoints in the area and used step in then the arrow location is as expected.



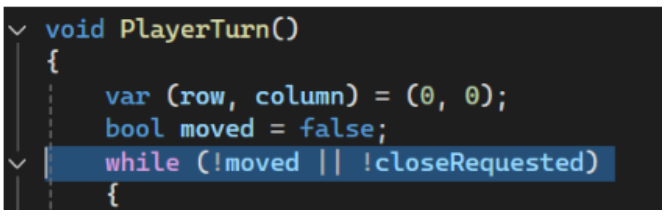


Other than that, I could not find any logical bugs so I inserted 5 major bugs into program.cs code.
Tic Tac Toe:

1. Changed && to || in PlayerTurn()



```
void PlayerTurn()
{
    var (row, column) = (0, 0);
    bool moved = false;
    while (!moved && !closeRequested)
    {
```



```
void PlayerTurn()
{
    var (row, column) = (0, 0);
    bool moved = false;
    while (!moved || !closeRequested)
    {
```

2. Change == to != in ComputeTurn()

```
void ComputerTurn()
{
    var possibleMoves = new List<(int X, i
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (board[i, j] == ' ')
```

```
void ComputerTurn()
{
    var possibleMoves = new List<(int
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (board[i, j] != ' ')
```

3. Changes winning logic by switching 2 with 3

```
113     bool CheckForThree(char c) =>
114         board[0, 0] == c && board[1, 0] == c && board[2, 0] == c ||
115         board[0, 1] == c && board[1, 1] == c && board[2, 1] == c ||
116         board[0, 2] == c && board[1, 2] == c && board[2, 2] == c ||
117         board[0, 0] == c && board[0, 1] == c && board[0, 2] == c ||
```

```
bool CheckForThree(char c) =>
    board[0, 0] == c && board[1, 0] == c && board[3, 0] == c ||
    board[0, 1] == c && board[1, 1] == c && board[2, 1] == c ||
    board[0, 2] == c && board[1, 2] == c && board[2, 2] == c ||
    board[0, 0] == c && board[0, 1] == c && board[0, 2] == c ||
```

4. Changed CloseRequested to !CloseRequested

```
    };
    while (closeRequested)
    {
```

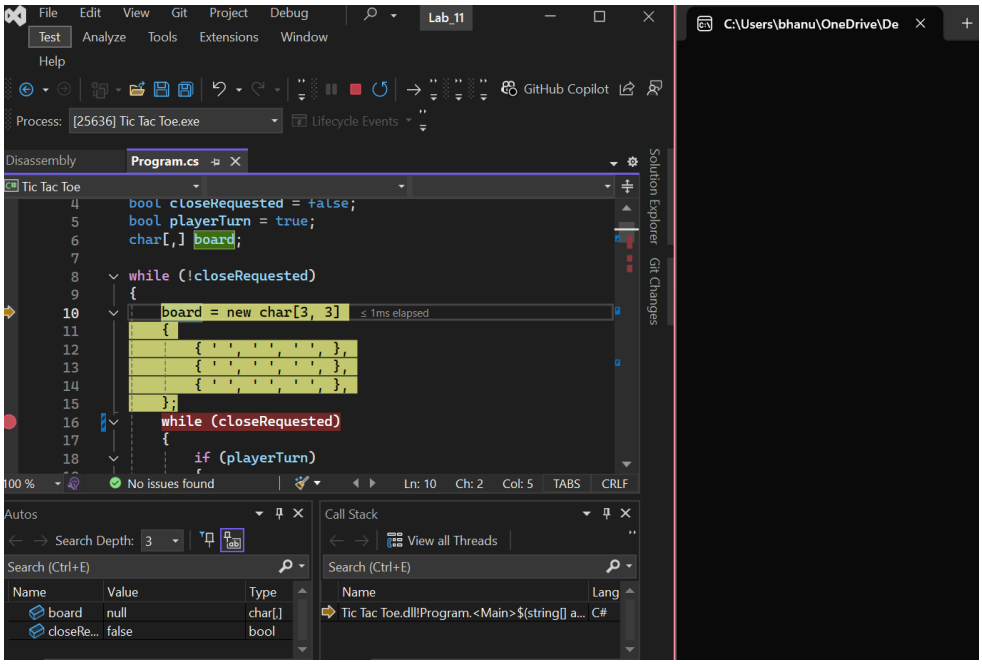
```
    };
    while (!closeRequested)
    {
```

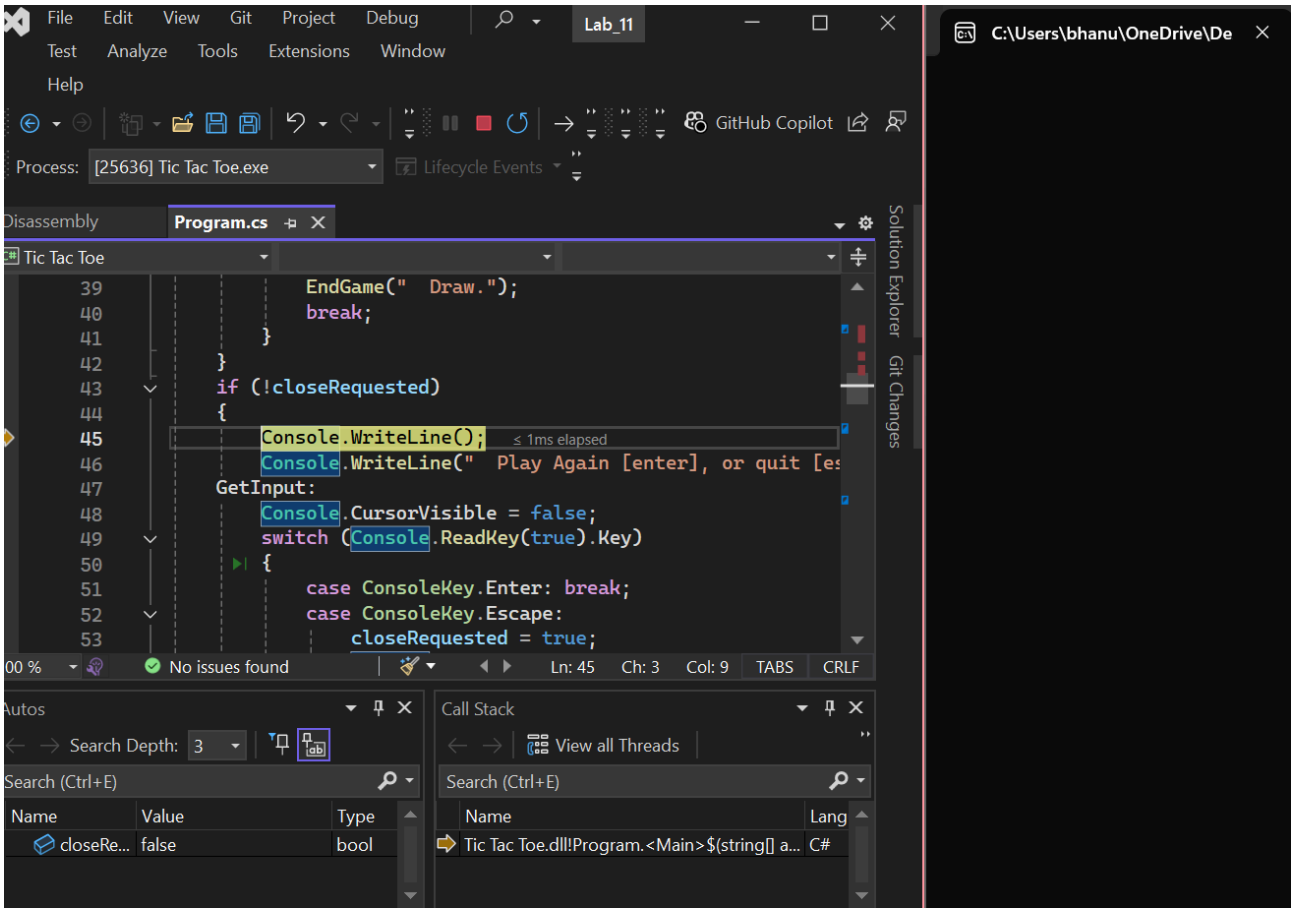
5. Checked for ‘X’ twice and did not check ‘O’

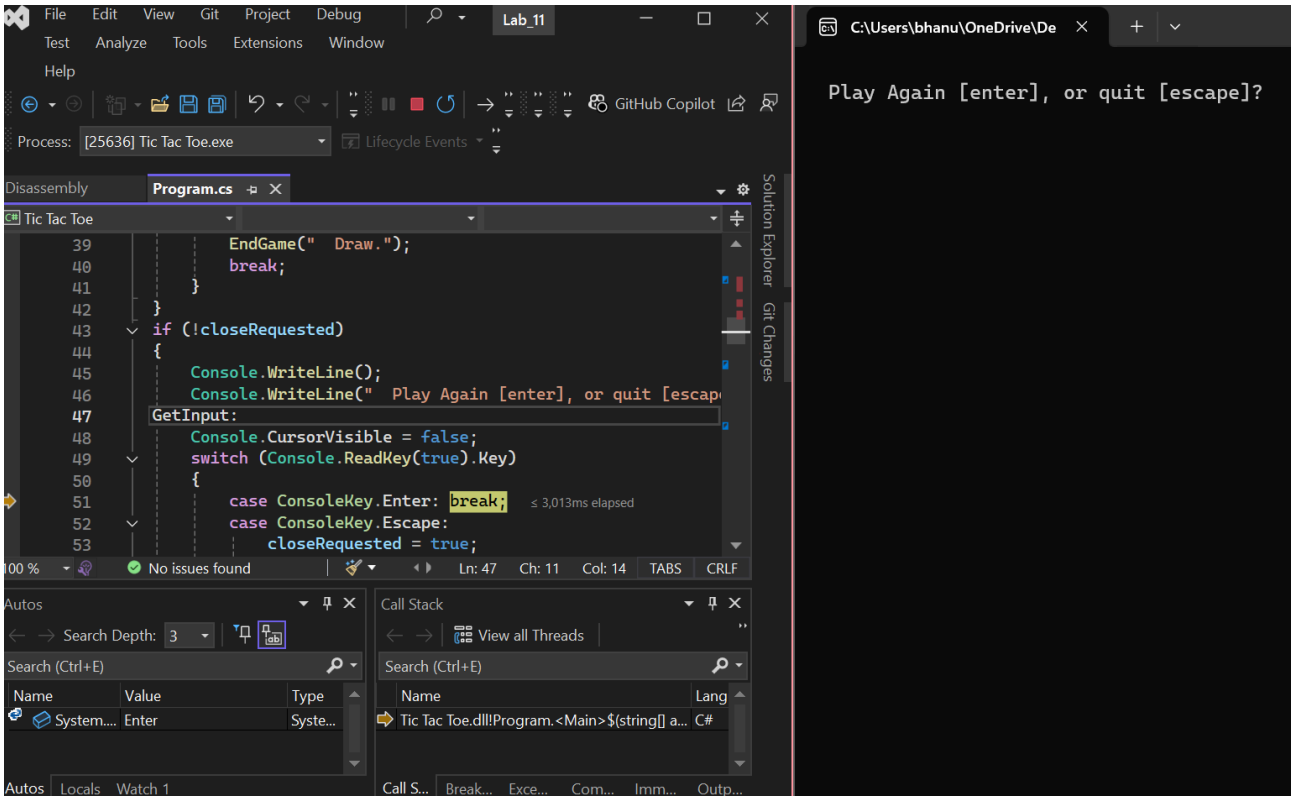
```
    PlayerTurn();
    if (CheckForThree('X'))
    {
        EndGame("  You Win.");
        break;
    }
}
else
{
    ComputerTurn();
    if (CheckForThree('O'))
```

```
    if (CheckForThree('X'))
    {
        EndGame("  You Win.");
        break;
    }
}
else
{
    ComputerTurn();
    if (CheckForThree('X'))
```

Step 3: Debugging Tic Tac Toe



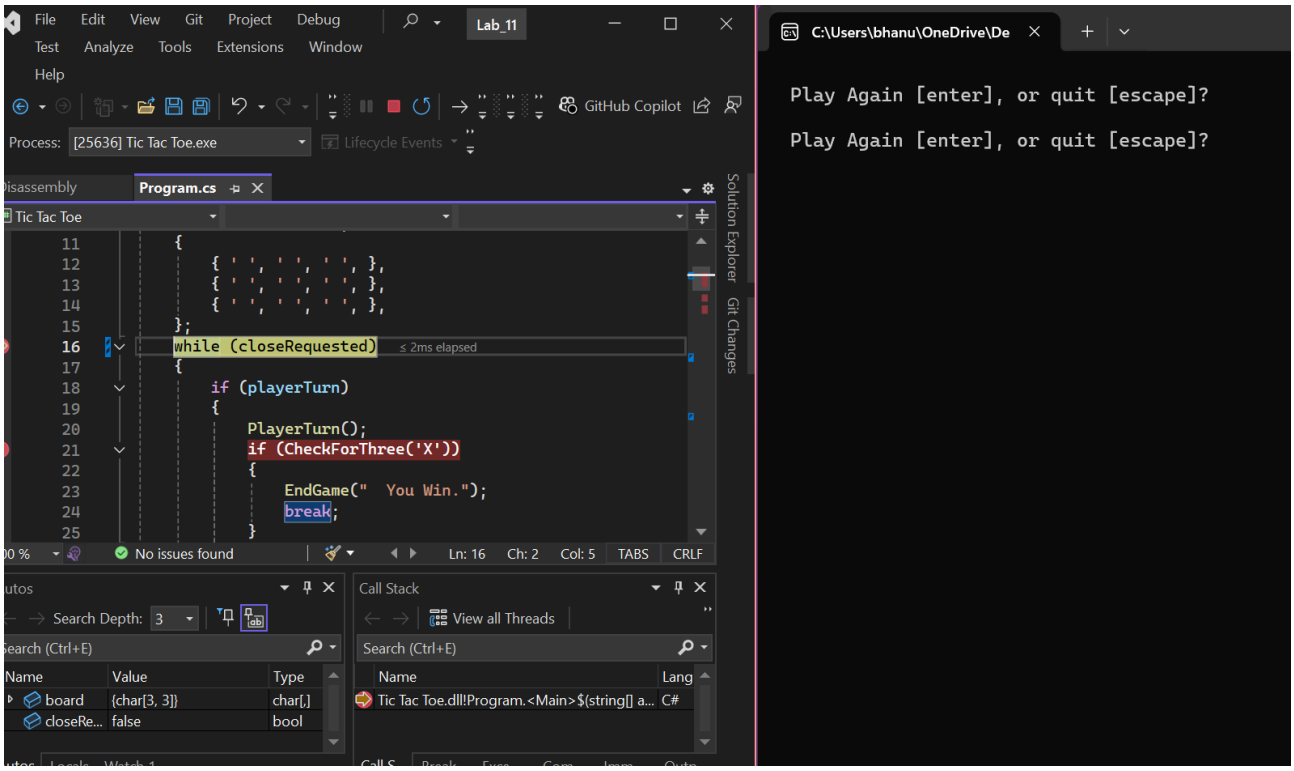




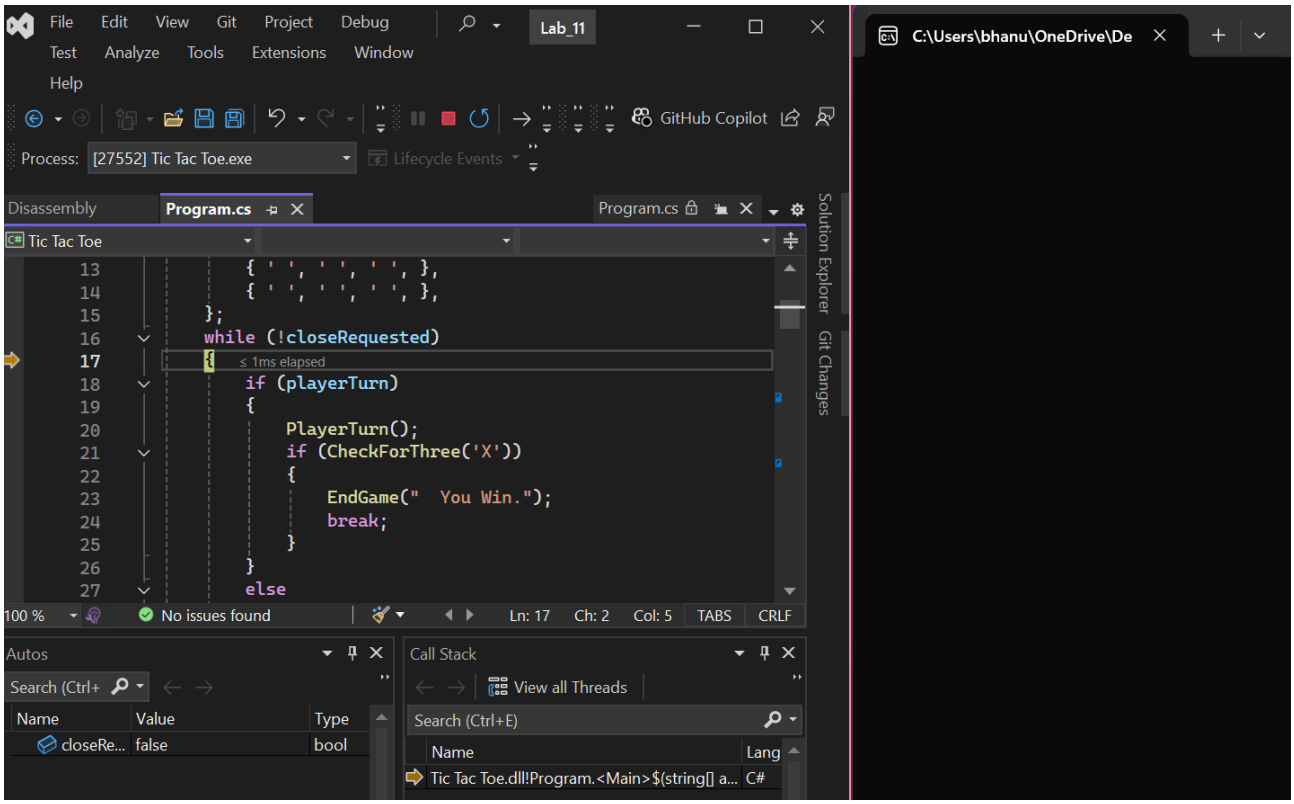
Bug 1:

After realising that I am going in a loop and reading the console output, it seems that the game is thinking that it has ended and is asking the player for another round. Without the player even playing 1 round. And It is going in a loop.

So I checked and loop and noticed that at this point, we need the game to enter this while loop but since it is set to closeRequested and not !closRequested, we are unable to enter, So our 1st bug is found and corrected.

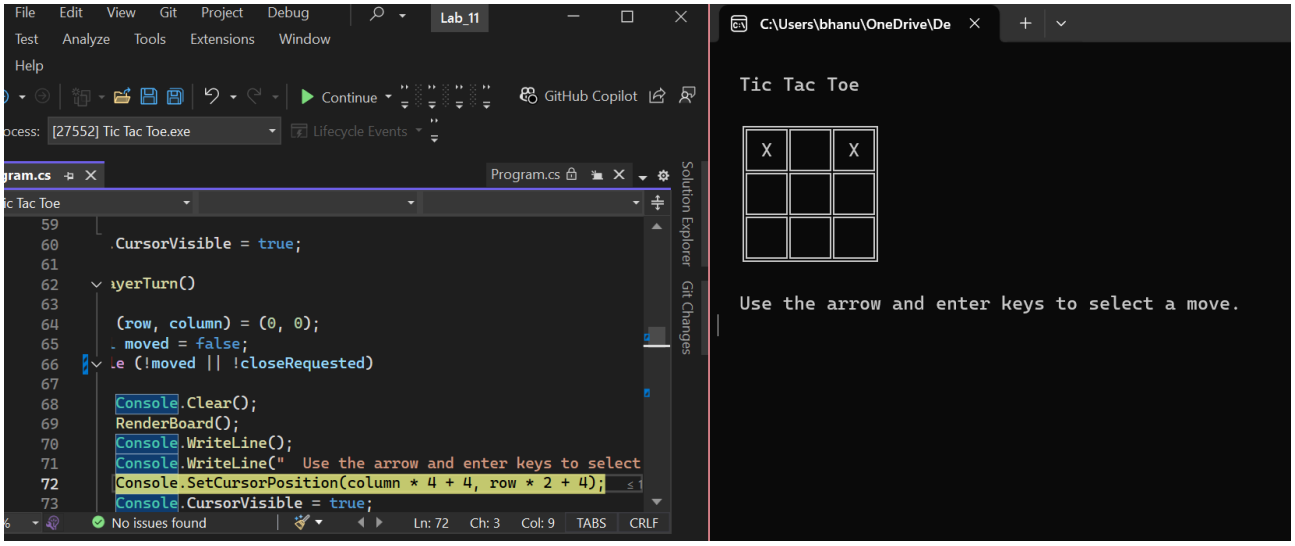


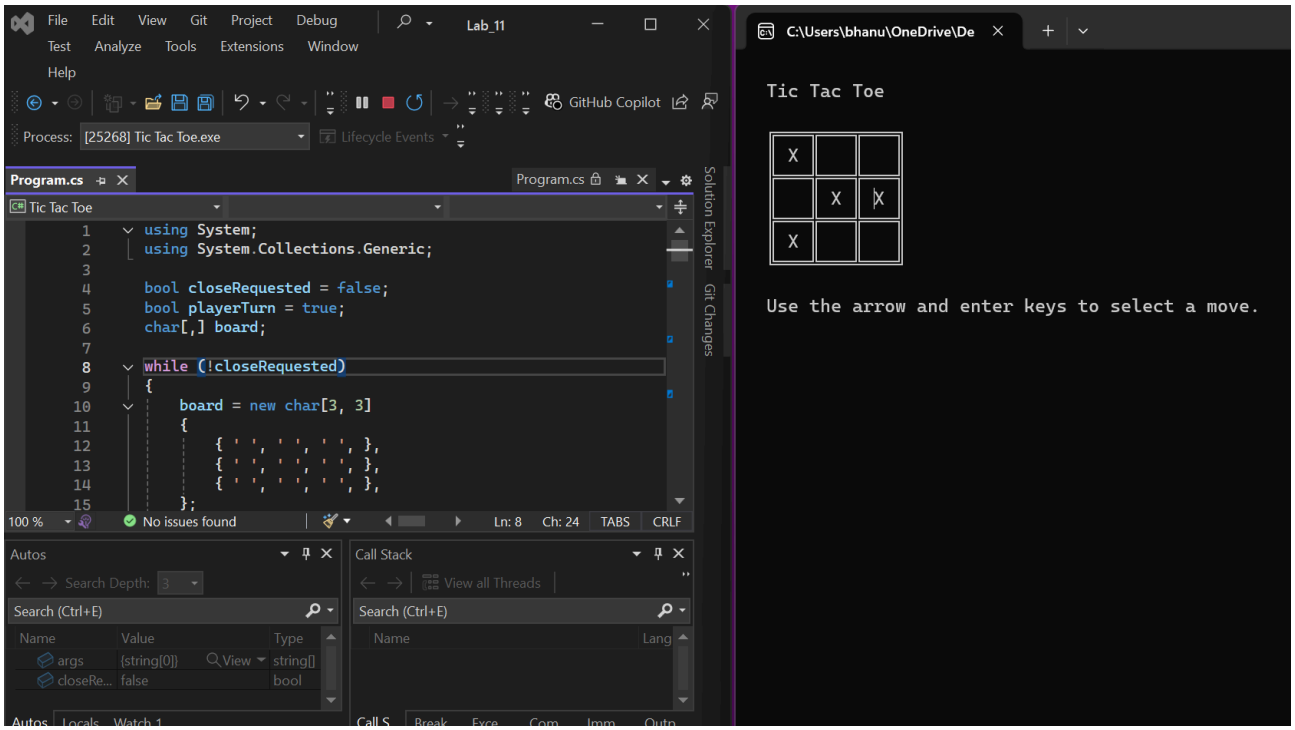
After correcting, it entered the loop as expected.

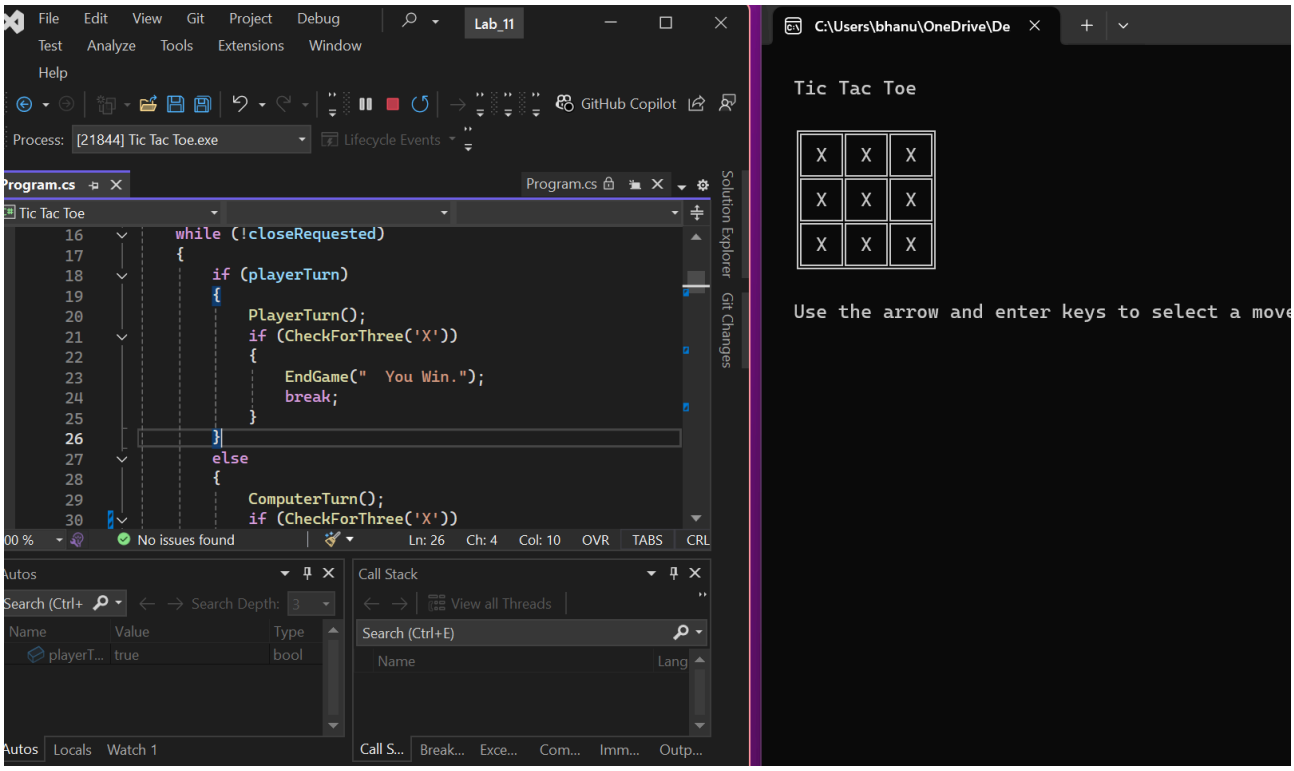


Bug 2:

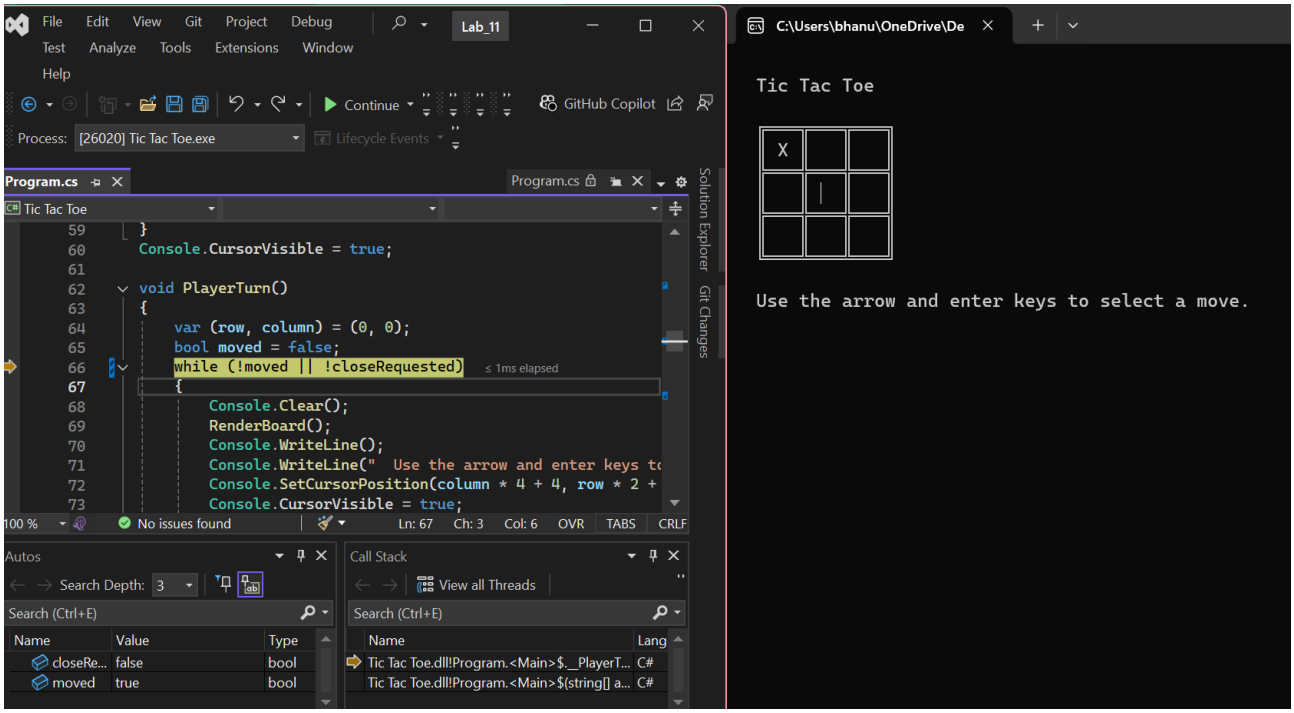
After fixing the bug, I went to play another round and found that the computer was not playing any input.





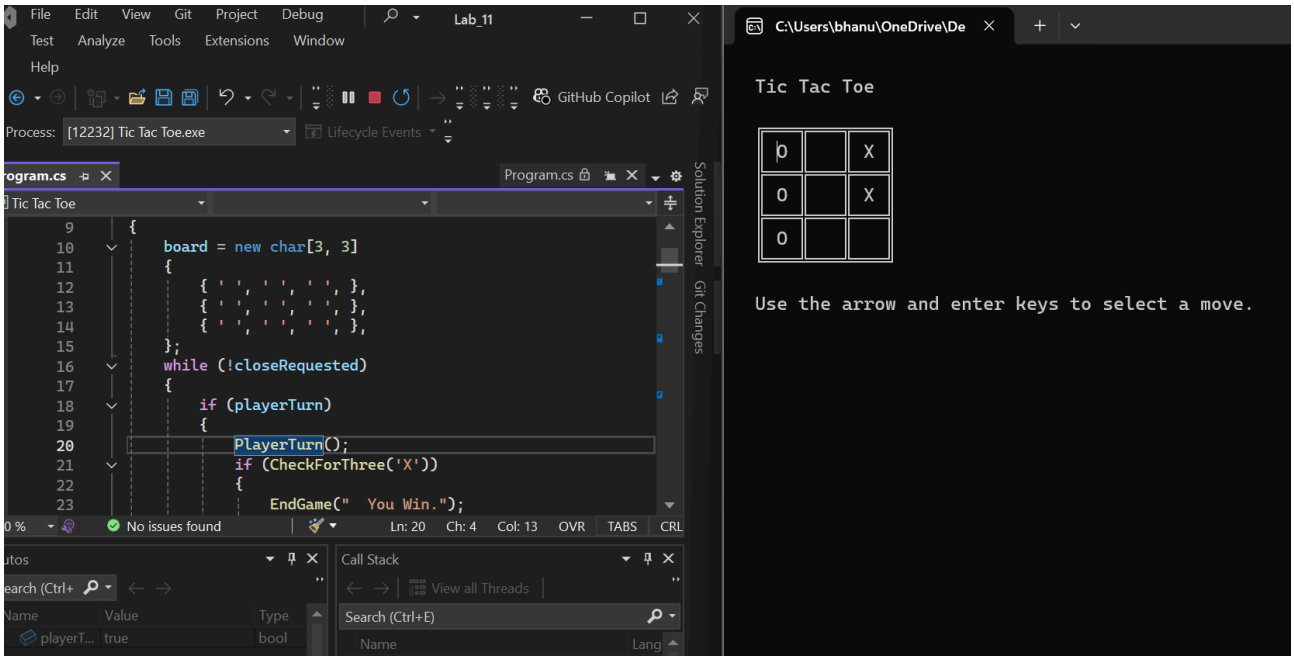


So I checked if Computer Turn was being called or not. But this was happening because I am going in an infinite loop within playerTurn, because in the below line it should be &&.

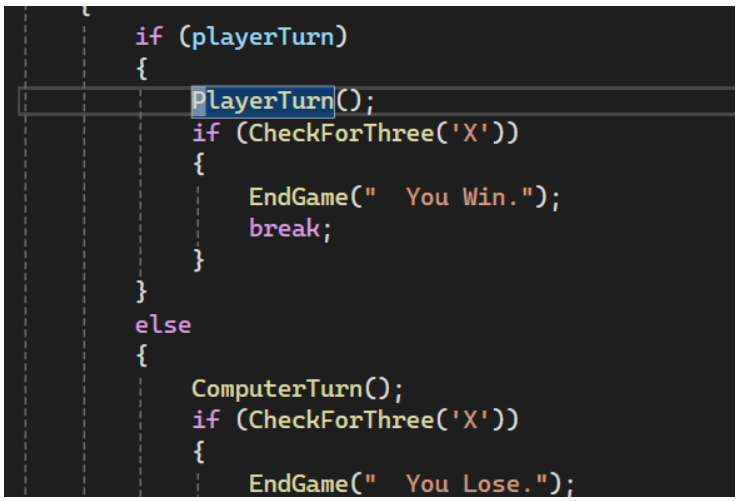


Bug 3:

I continued playing and I noticed that even after ‘O’ has won the game is still continuing.

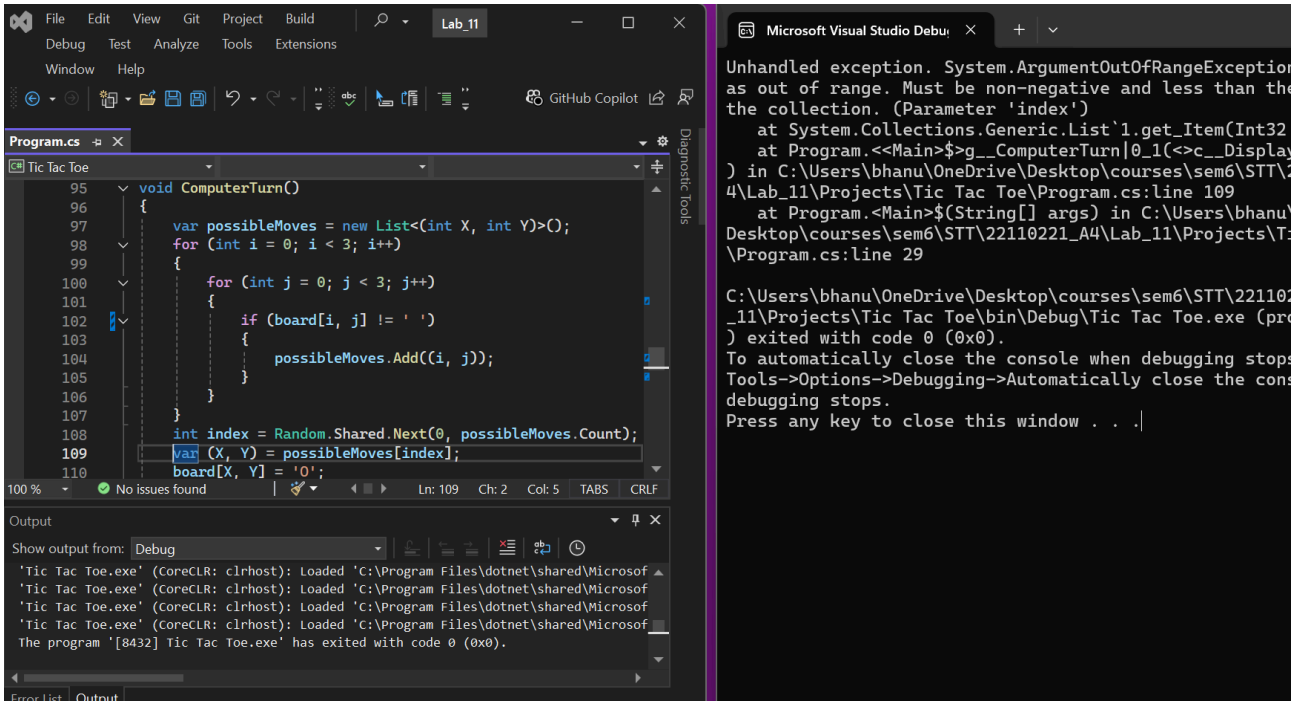


So I checked the winning conditions and the winning function for ‘O’ and there is no checking done for ‘O’ winning. So I changed it.

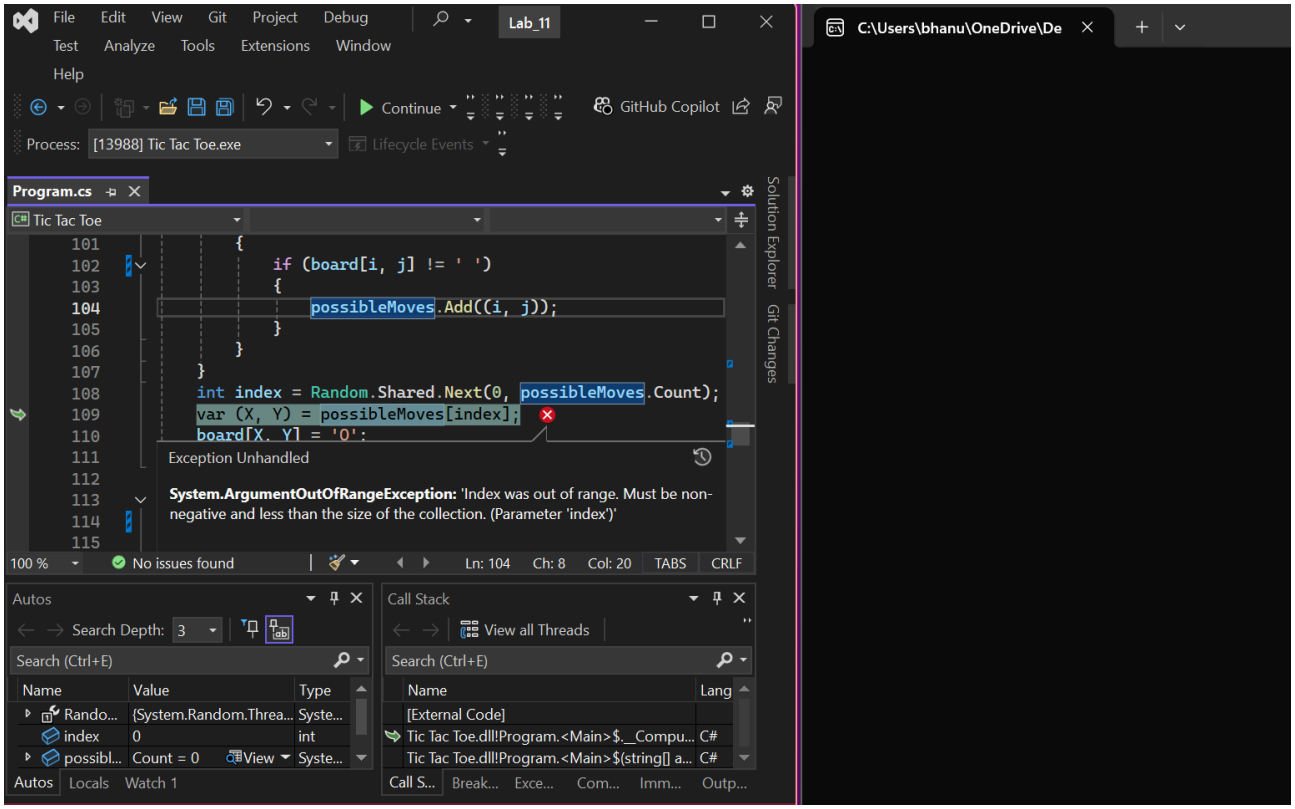


Bug 4:

As I continued playing the game, I got the error that the index is out of range. This arised in ComputerTurn().



I couldn't find anything using step over so I used step-into. Then I found this



In the bottom, it says possible moves **count = 0**. As we are trying to read from an empty array this error is arising but why is possible moves empty before the game is played.

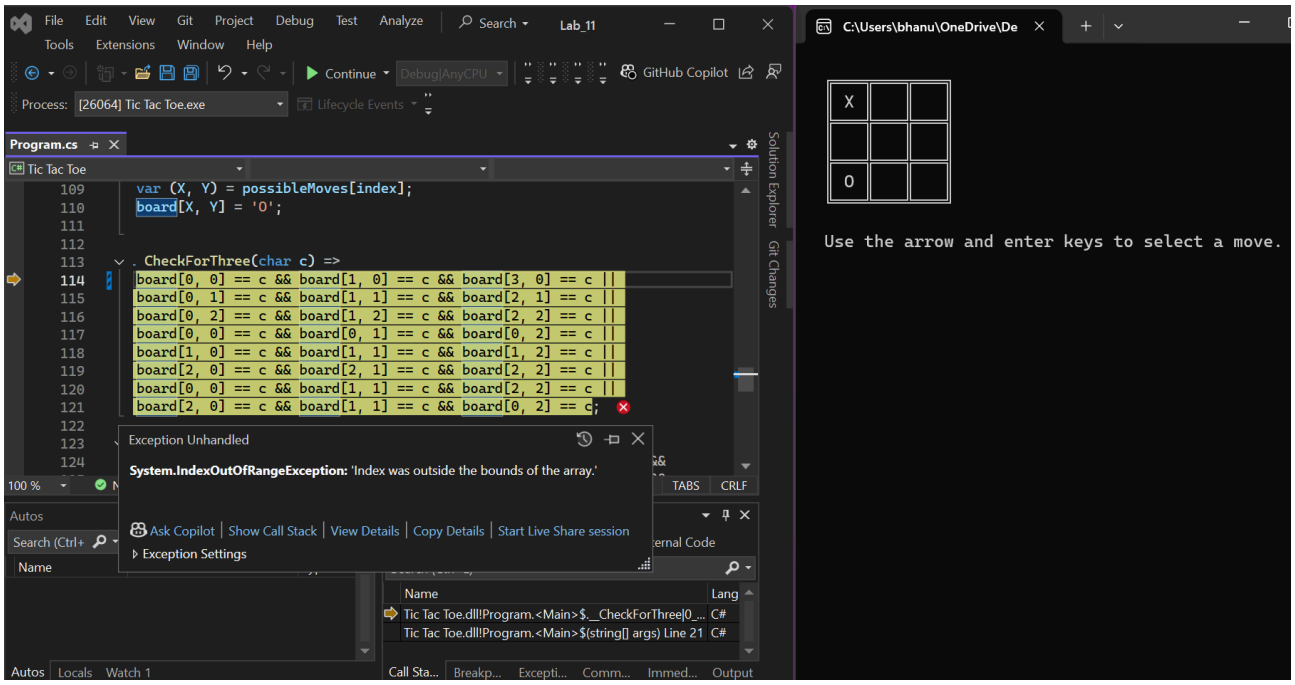
Computer was not able to find any turn because it is not adding the possible moves which it found into the array

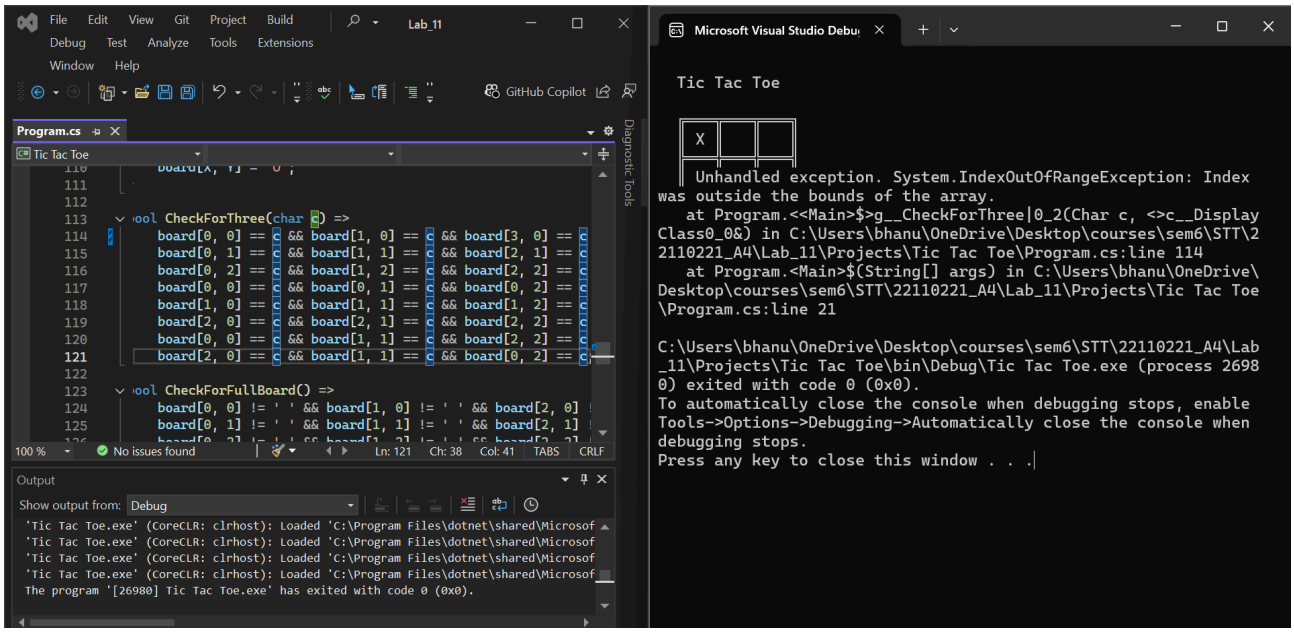
```
void ComputerTurn()
{
    var possibleMoves = new List<(int X, int Y)>();
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (board[i, j] != ' ')
            {
                possibleMoves.Add((i, j));
            }
```

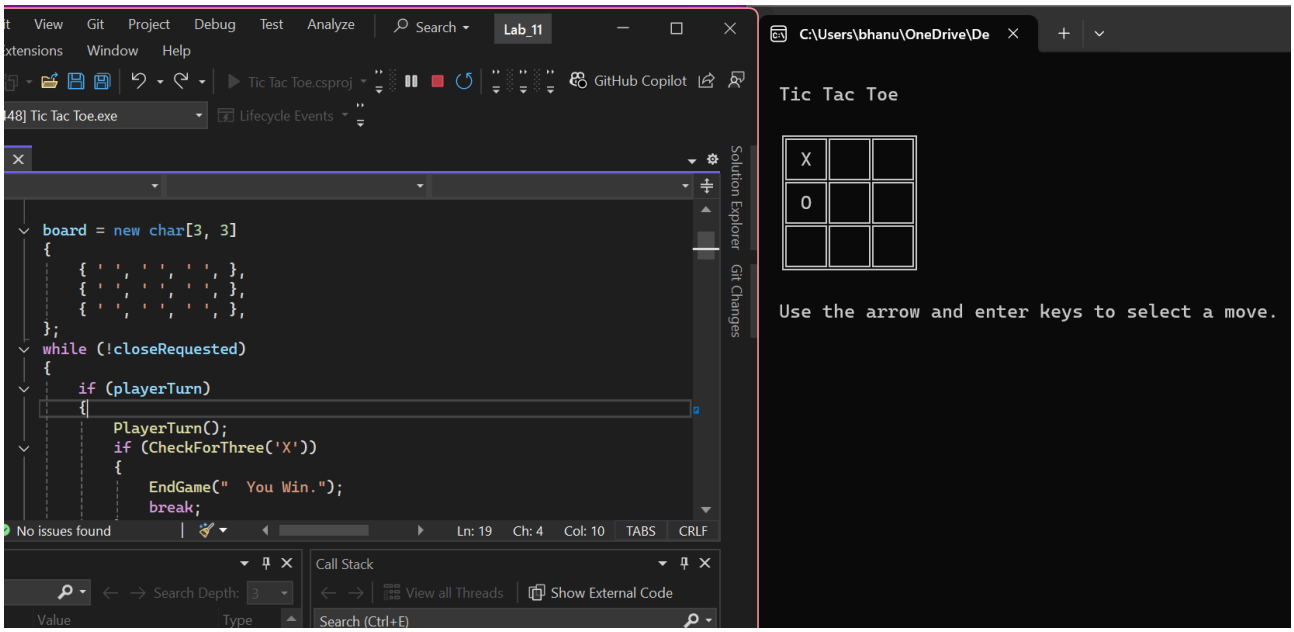
So I corrected it. And continued playing the game.

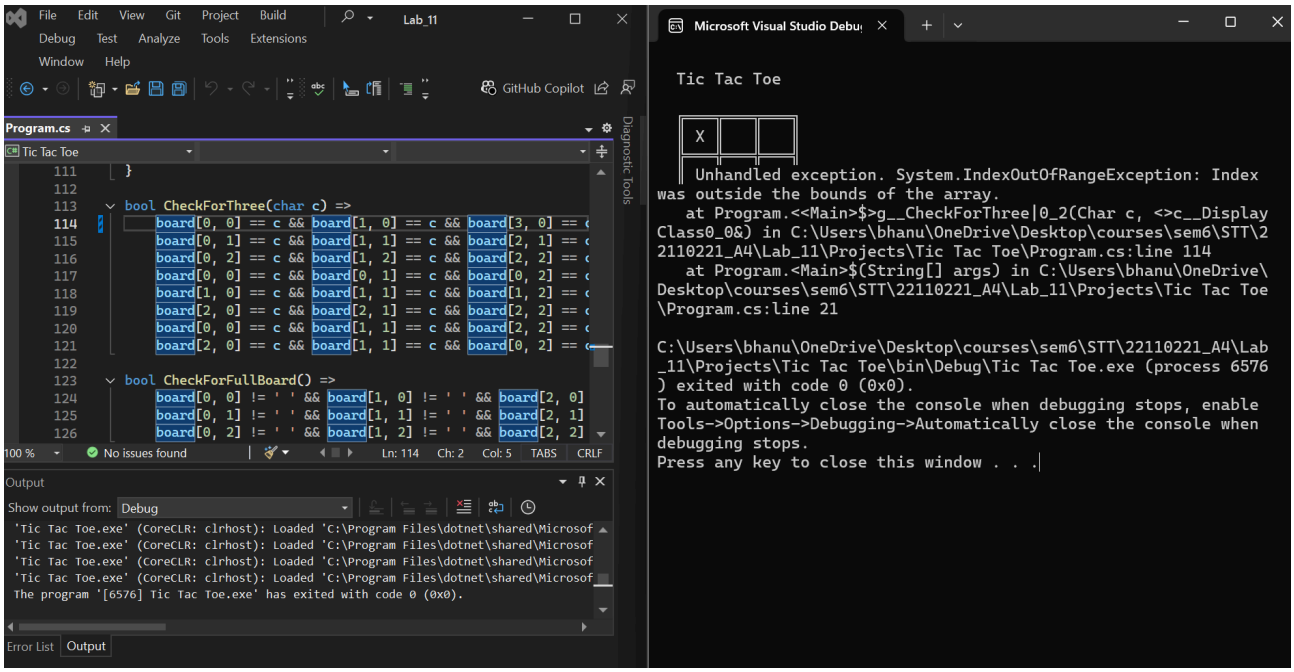
Bug 5:

For a few cases, I faced index out of range exception, so I kept playing to notice the pattern.

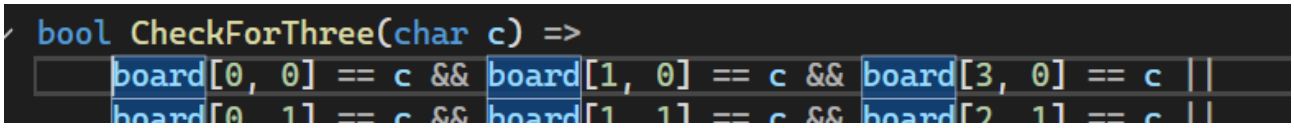



File Edit View Git Project Build
Debug Test Analyze Tools Extensions
Window Help
Lab_11
GitHub Copilot
Program.cs
Tic Tac Toe
bool CheckForThree(char c) =>
board[0, 0] == c && board[1, 0] == c && board[3, 0] == c
board[0, 1] == c && board[1, 1] == c && board[2, 1] == c
board[0, 2] == c && board[1, 2] == c && board[2, 2] == c
board[0, 0] == c && board[0, 1] == c && board[0, 2] == c
board[1, 0] == c && board[1, 1] == c && board[1, 2] == c
board[2, 0] == c && board[2, 1] == c && board[2, 2] == c
board[0, 0] == c && board[1, 1] == c && board[2, 2] == c
board[2, 0] == c && board[1, 1] == c && board[0, 2] == c
bool CheckForFullBoard() =>
board[0, 0] != ' ' && board[1, 0] != ' ' && board[2, 0]
board[0, 1] != ' ' && board[1, 1] != ' ' && board[2, 1]
100 % No issues found Ln: 121 Ch: 38 Col: 41 TABS CRLF
Output
Show output from: Debug
'Tic Tac Toe.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsof
'Tic Tac Toe.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsof
'Tic Tac Toe.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsof
'Tic Tac Toe.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsof
The program '[26980] Tic Tac Toe.exe' has exited with code 0 (0x0).
Microsoft Visual Studio Debu
Tic Tac Toe
X
Unhandled exception. System.IndexOutOfRangeException: Index was outside the bounds of the array.
at Program.<<Main>$>g__CheckForThree|0_2(Char c, <>c__DisplayClass0_0&) in C:\Users\bhanu\OneDrive\Desktop\courses\sem6\STT\22110221_A4\Lab_11\Projects\Tic Tac Toe\Program.cs:line 114
at Program.<Main>$(String[] args) in C:\Users\bhanu\OneDrive\Desktop\courses\sem6\STT\22110221_A4\Lab_11\Projects\Tic Tac Toe\Program.cs:line 21
C:\Users\bhanu\OneDrive\Desktop\courses\sem6\STT\22110221_A4\Lab_11\Projects\Tic Tac Toe\bin\Debug\Tic Tac Toe.exe (process 26980) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .


View Git Project Debug Test Analyze
Search
Lab_11
Extensions Window Help
Tic Tac Toe.csproj
GitHub Copilot
Lifecycle Events
board = new char[3, 3]
{
{ ' ', ' ', ' ', },
{ ' ', ' ', ' ', },
{ ' ', ' ', ' ', },
};
while (!closeRequested)
{
if (playerTurn)
{
PlayerTurn();
if (CheckForThree('X'))
{
EndGame(" You Win.");
break;
No issues found Ln: 19 Ch: 4 Col: 10 TABS CRLF
Solution Explorer
Git Changes
Call Stack
Search Depth:
View all Threads
Show External Code
Search (Ctrl+E)
Value
Type
C:\Users\bhanu\OneDrive\De
Tic Tac Toe
X
O
Use the arrow and enter keys to select a move.



This error is arising only in the case when both board[0][0] and board[1][0] are filled (with any symbol). This is because the bug here is only identified when the first two statements are true then it needs to check the 3rd statement which is out of range.



All the bugs are found!

Step 4: Analyzing Rock Paper Scissors

I started by playing a few rounds without debugging.

I lost

```
Rock, Paper, Scissors

Choose [r]ock, [p]aper, [s]cissors, or [e]xit:r
The computer chose Paper.
You lose.
Score: 0 wins, 1 losses, 0 draws
Press Enter To Continue...
```

I won

```
Rock, Paper, Scissors

Choose [r]ock, [p]aper, [s]cissors, or [e]xit:p
The computer chose Rock.
You win.
Score: 1 wins, 1 losses, 0 draws
Press Enter To Continue...
```

Draw

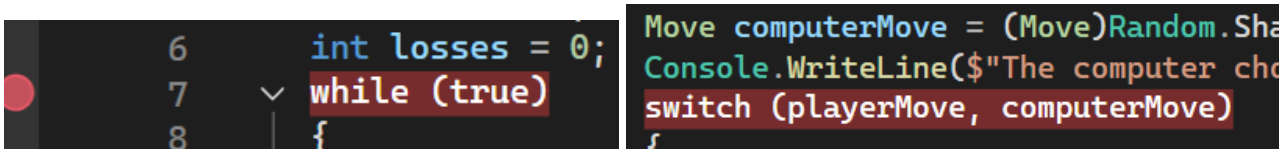
```
Rock, Paper, Scissors

Choose [r]ock, [p]aper, [s]cissors, or [e]xit:s
The computer chose Scissors.
This game was a draw.
Score: 3 wins, 3 losses, 1 draws
Press Enter To Continue...
```

Invalid

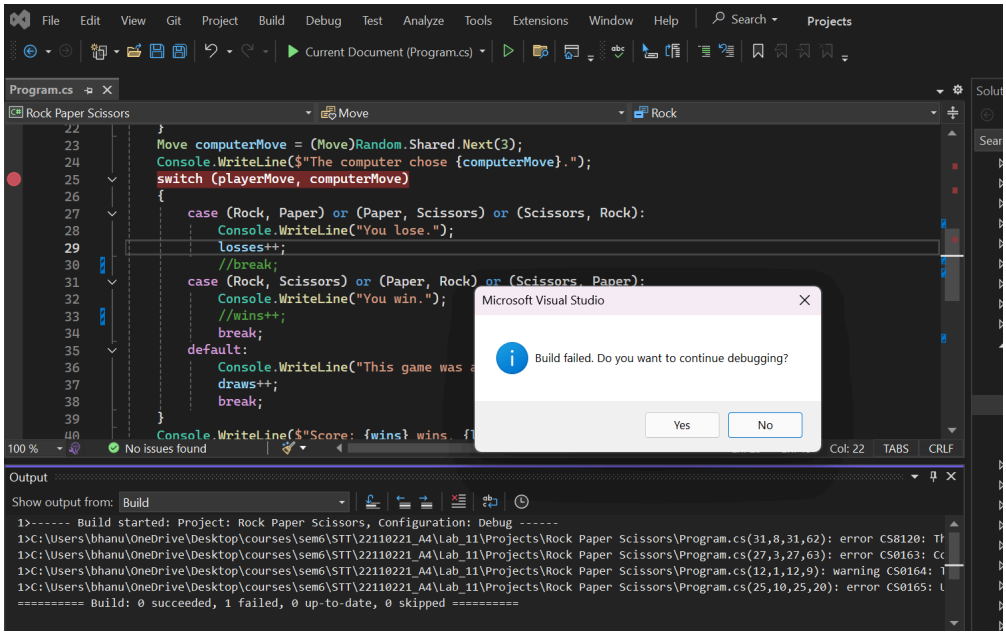
```
Rock, Paper, Scissors

Choose [r]ock, [p]aper, [s]cissors, or [e]xit:f
Invalid Input. Try Again...
Choose [r]ock, [p]aper, [s]cissors, or [e]xit:
```

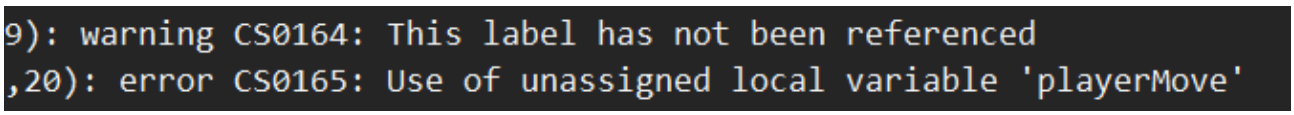
Then, added break points.

Breakpoints

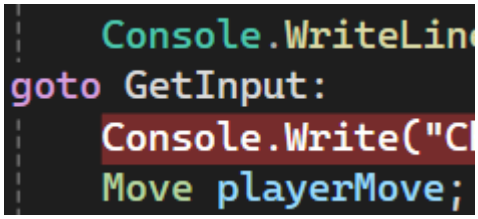


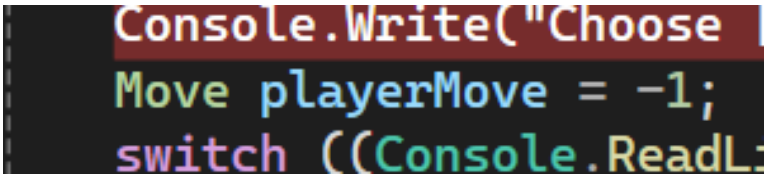
And started playing in debug mode but before I could play I got these errors.





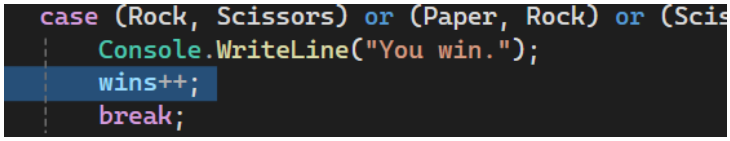
These errors refer to not using goto for a label, hence I added it. And the other one says playerMove is not initialized so I initialised it to -1 (invalid).

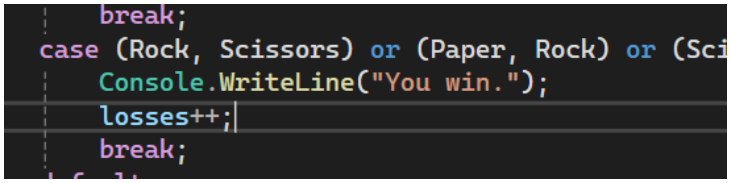




Everything is working as expected so I inserted 5 bugs

1. I replace wins with losses





2. Removing one of the win conditions

```
case (Rock, Scissors) or (Paper, Rock) or (Scissors, Paper):
    Console.WriteLine("You win.");
    losses++;
    break;
```

```
case (Rock, Scissors) or (Paper, Rock):
    Console.WriteLine("You win.");
    losses++;
    break;
```

3. I replaced case ‘e’ return with break;

```
case "p" or "paper": playerMove = Paper; break;
case "s" or "scissors": playerMove = Scissors; break;
case "e" or "exit": Console.Clear(); return;
default:Console.WriteLine("Invalid Input. Try Again...");
goto GetInput;
```

```
case "r" or "rock": playerMove = Rock; break;
case "p" or "paper": playerMove = Paper; break;
case "s" or "scissors": playerMove = Scissors; break;
case "e" or "exit": Console.Clear(); break;
default:Console.WriteLine("Invalid Input. Try Again...");
goto GetInput;
```

4. Make the computer generate an invalid move

```
}
Move computerMove = (Move)Random.Shared.Next(3);
Console.WriteLine($"The computer chose {computerMove}.");
```

```
Move computerMove = (Move)Random.Shared.Next(4);
Console.WriteLine($"The computer chose {computerMove}.");
```

5. Consider Rock vs Rock as a lose

```
case (Rock, Paper) or (Paper, Scissors) or (Scissors, Rock):
    Console.WriteLine("You lose.");
    losses++;
    break;
```

```
case (Rock, Rock) or (Paper, Scissors) or (Scissors, Rock):
    Console.WriteLine("You lose.");
    losses++;
    break;
```

Step 5: Debugging Rock Paper Scissors

Before I could start debugging, the build failed.

First I started with step-over debugging.

Bug 1:

And as I played the game. It showed a draw when I chose rock and the computer chose paper. This is incorrect.

Upon checking the draw or default case i.e by checking the winning and losing condition, I noticed that there is no (Rock,Paper) condition. It should be in the losing condition. 1st bug detected.

Bug 2:

Then by playing a few rounds, I noticed another issue. It is saying Rock vs Rock as a lose. Reviewing the winning and losing condition once again I noticed the issue and removed the Rock vs Rock condition as the lose condition.

```
    case (Rock, Scissors) or (Scissors, Paper) or (Paper, Rock) or (Rock, Rock):
        Console.WriteLine("You lose.");
        losses++;
        break;
    case (Rock, Scissors) or (Paper, Rock):
```

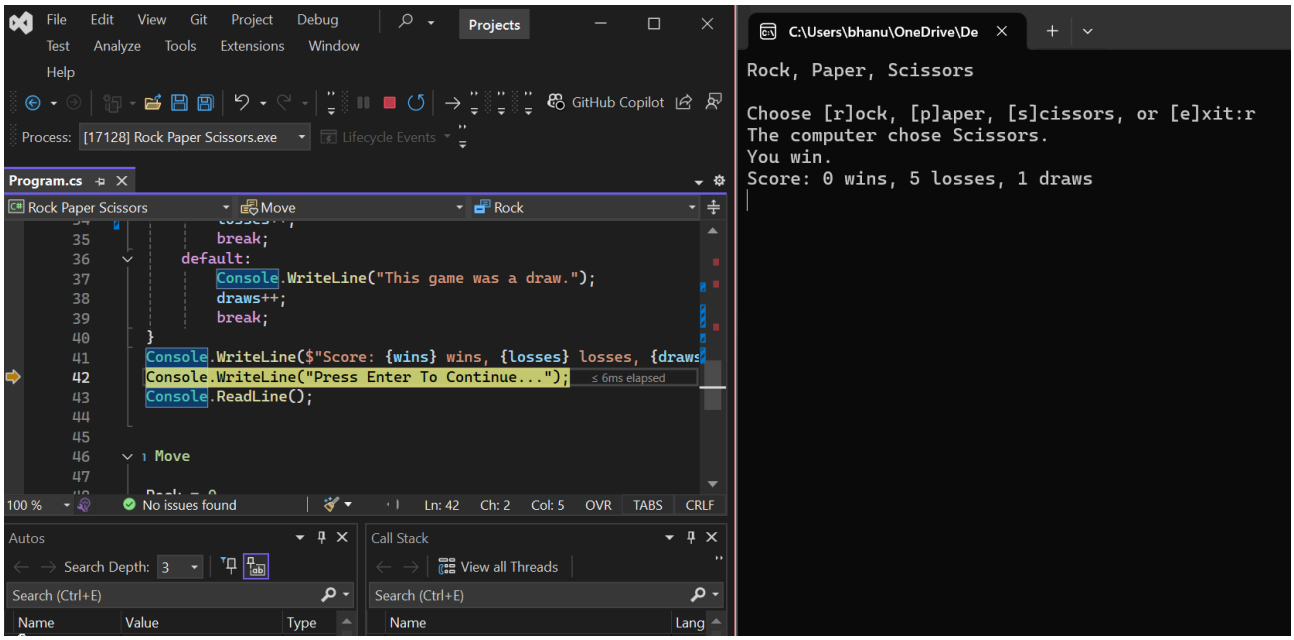
```
switch (playerMove, computerMove)
{
    case (Rock, Scissors) or (Scissors, Paper) or (Paper, Rock):
        Console.WriteLine("You lose.");
        losses++;
        break;
```

Finally I made sure all the condition are correct

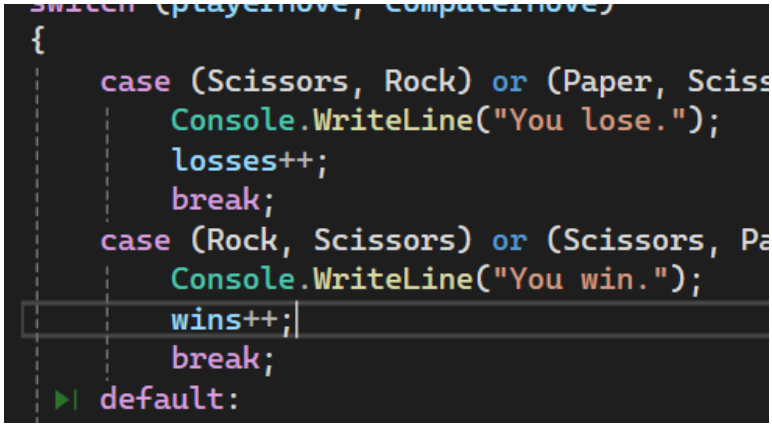
```
switch (playerMove, computerMove)
{
    case (Scissors, Rock) or (Paper, Scissors) or (Rock, Paper):
        Console.WriteLine("You lose.");
        losses++;
        break;
    case (Rock, Scissors) or (Scissors, Paper) or (Paper, Rock):
        Console.WriteLine("You win.");
        losses++;
        break;
    default:
        Console.WriteLine("This game was a draw.");
        draws++;
        break;
}
```

Bug 3:

After playing several rounds, I noticed that even after winning a few rounds the computer is viewing no wins by player. But it acknowledged that I won that round.

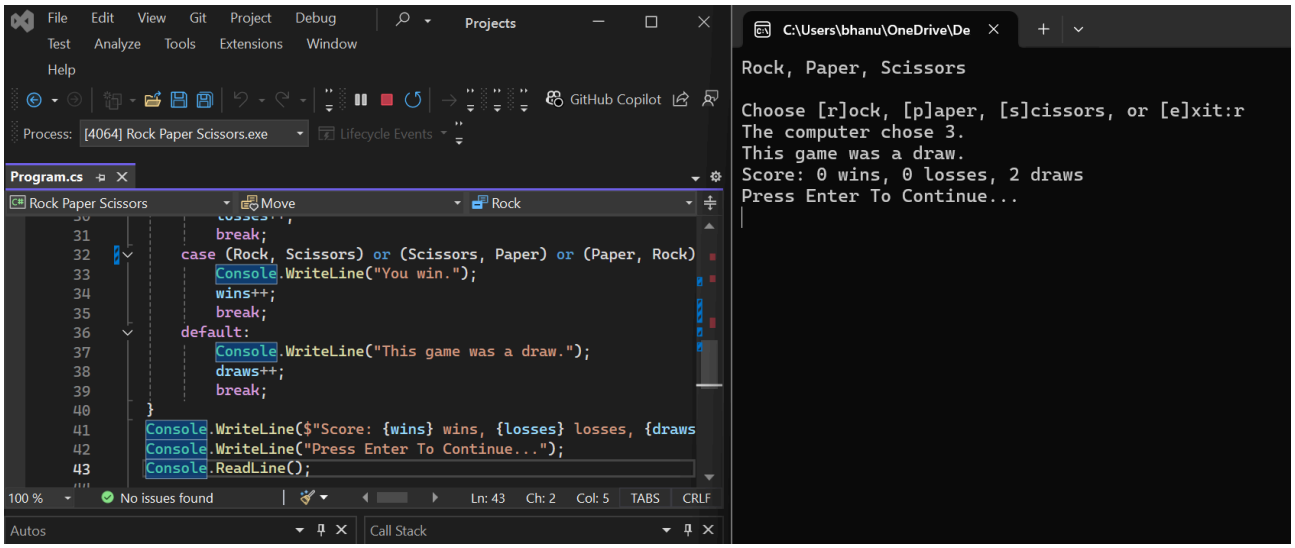


Upon checking the case, It seems that no matter who wins, the score is being added to the computer i.e losses are getting incremented every time. So I corrected it.



Bug 4:

When I was playing I found that the computer is sometimes showing a valid move and sometimes as invalid move like 3.



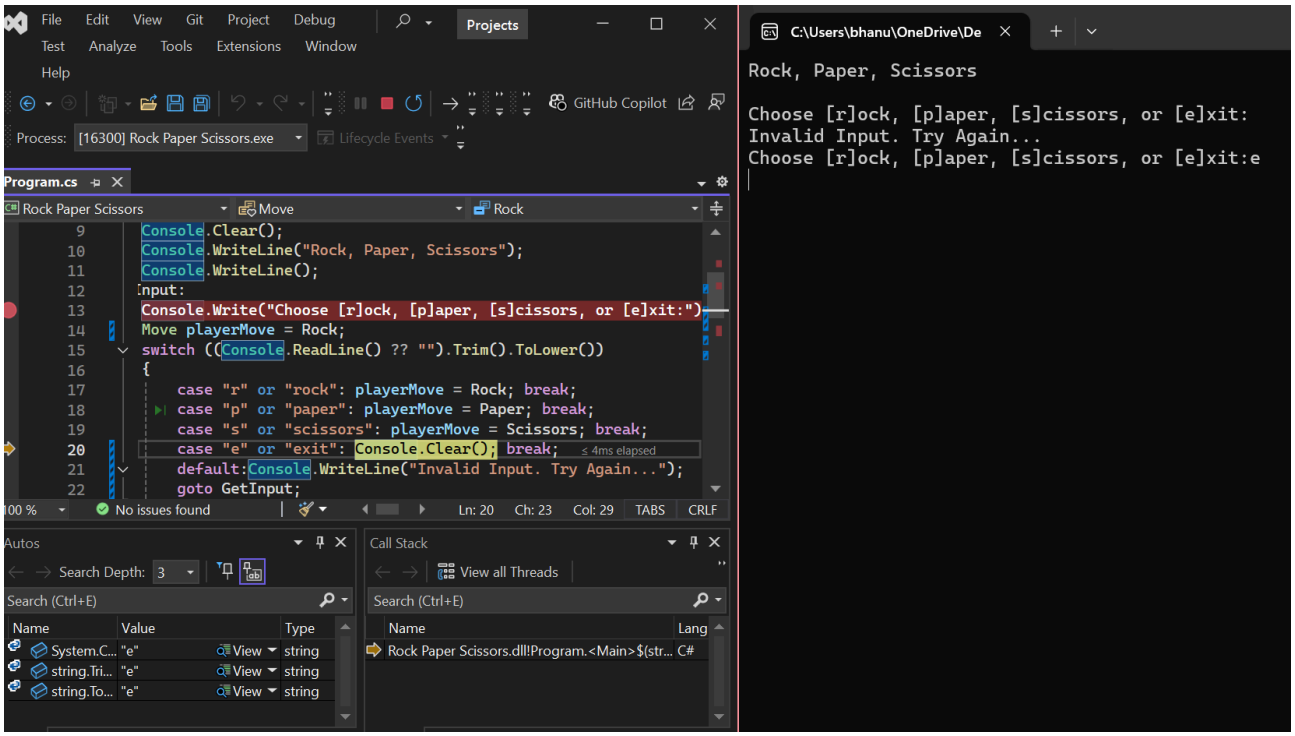
This is because of :

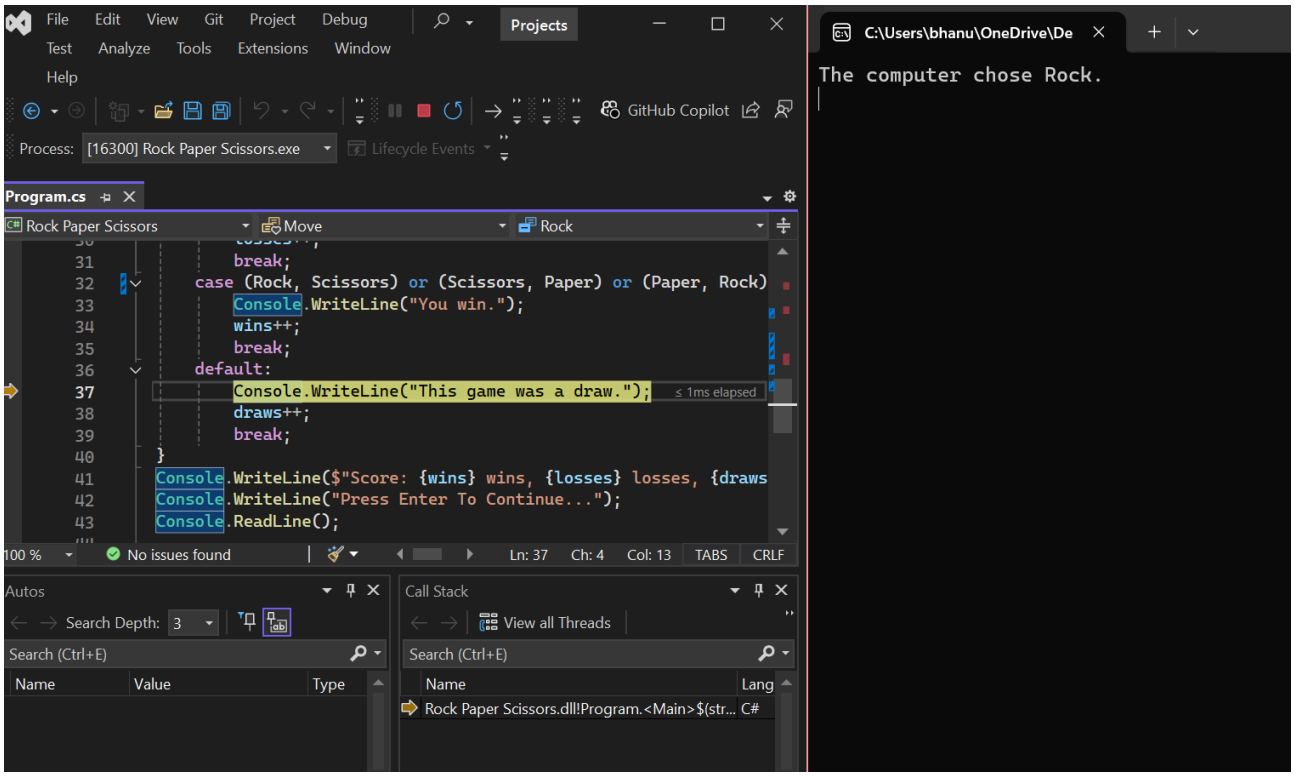
```
Move computerMove = (Move)Random.Shared.Next(4);
```

So I set it to 3.

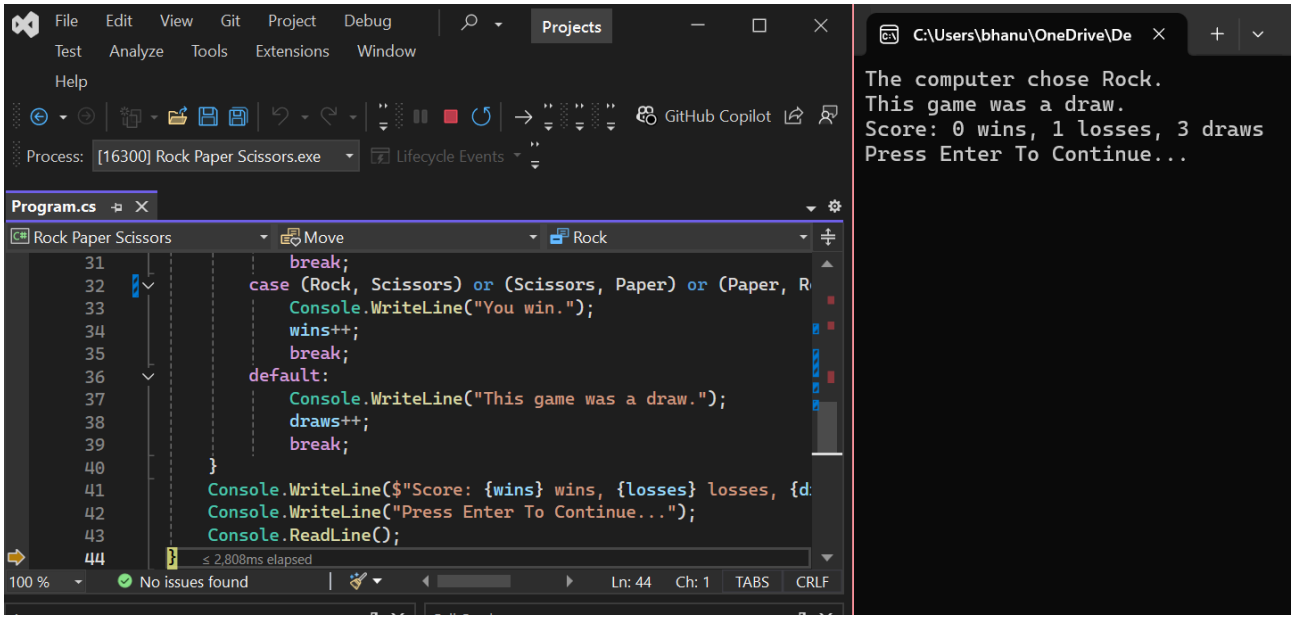
Bug 5:

This time all the cases are working correctly so I wanted to exit





But the game kept on going, the computer still chose a player.



Upon checking the exit condition. It is set to break rather than return.
All bugs are found!

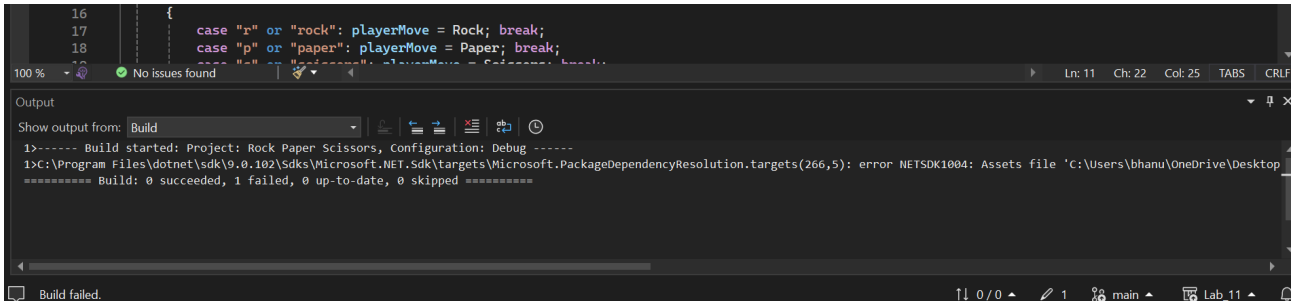
---

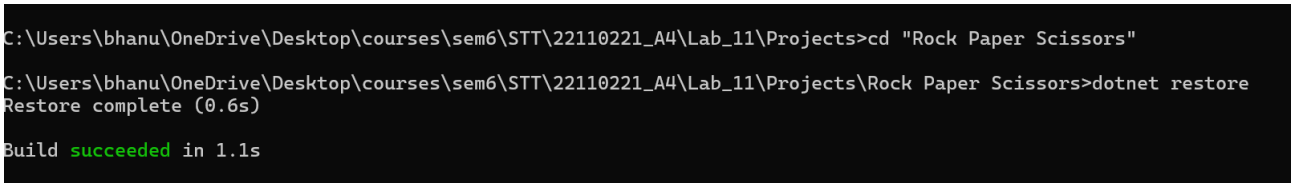
3. Results and Analysis

4. Discussion and Conclusion

Challenges and Reflections

Initially I was getting a build failed error and it got resolved using restore dotnet





Initial challenges included differentiating compiler errors from runtime bugs and understanding debugger interactions with console input (Console.ReadKey). Intentionally creating non-compiler-breaking bugs required careful thought about runtime logic rather than just syntax. Effectively tracing complex conditional logic (like the game's win/loss switch) using the debugger took practice.
I created bugs like removing default statements or removing the body from the case statements and these were easily detected by the compiler itself.

The hands-on debugging process significantly clarified how code executes dynamically. Witnessing variable changes and control flow shifts in real-time was highly instructive. Injecting bugs proved to be a valuable exercise in understanding code fragility and the importance of thorough testing.

Lessons Learned

Debugger Navigation: Proficient use of Step Into (F11), Step Over (F10), and Step Out (Shift+F11) for controlling execution flow.

State Inspection: Utilizing debugger windows (Locals, Autos, Watch) to monitor variable values during runtime.

Breakpoint Strategy: Placing breakpoints effectively to isolate problems and analyze specific code sections.

Runtime vs. Compile-time: Clearly distinguishing between errors caught by the compiler (syntax, static analysis) and those needing the debugger (runtime exceptions, logic flaws).

Bug Causation: Analyzing how small code changes (mutations) can lead to significant runtime errors or incorrect behavior.

Summary

This lab provided essential hands-on experience with the Visual Studio Debugger using a C# console game. Activities involved tracing program execution, identifying and fixing injected runtime bugs, and documenting the process. Key debugging techniques were practiced, enhancing understanding of code analysis, runtime behavior, and fault diagnosis in C# applications.

---

5. References

1. https://github.com/dotnet/dotnet-console-games
2. https://visualstudio.microsoft.com
3. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/top-level-statements
4. https://learn.microsoft.com/en-us/visualstudio/debugger
5. Google AI studio for debugging, refining the grammar and formatting of this report

Lab Assignment 12

Lab Topic: Event-driven Programming for Windows Forms Apps. in C#

1. Introduction

This project explores fundamental C# programming concepts through the development of two alarm clock applications. The first task utilizes a console interface to demonstrate core event handling and time monitoring using the publisher/subscriber pattern. The second task transitions to a graphical user interface (GUI) using Windows Forms, showcasing event-driven programming, UI responsiveness, and visual feedback common in desktop applications. Both tasks involve handling user input, validating data, working with date/time structures, and triggering actions based on time comparisons, providing practical experience in different C# application models.

Environment Setup:

- Operating System: Windows
- Software: Visual Studio 2022 (Community Edition), Visual Studio with .NET SDK
- Programming Language: C# 13 or dotnet SDK 9.0.102

Key Concepts:

Windows Forms (WinForms): A .NET GUI framework for building desktop applications with visual elements like forms, buttons, text boxes, etc. It operates on an event-driven model.

Event-Driven Programming: A programming paradigm where the flow of the program is determined by events (e.g., user clicks a button, a timer ticks). Code execution happens in response to these events.

Events and Delegates (Action, EventHandler): A mechanism in C# for enabling communication between objects (publisher/subscriber). Delegates define the required signature for event handler methods, and events allow classes to notify subscribers when something significant happens.

Partial Classes: A C# feature allowing a class definition to be split across multiple source files (e.g., Form1.cs and Form1.Designer.cs), commonly used by WinForms designer.

---

2. Methodology and Execution

This is the file structure in the folder Lab12.

File Structure:

```
├── Lab12/              # Console Alarm Project Folder
│   ├── Lab12.csproj      # Project build settings/dependencies
│   ├── Lab12.sln         # Solution file (groups projects)
│   ├── Program.cs        # Main application code/entry point
│   ├── bin/            # Compiled output binaries folder
│   └── obj/            # Intermediate build files folder
```

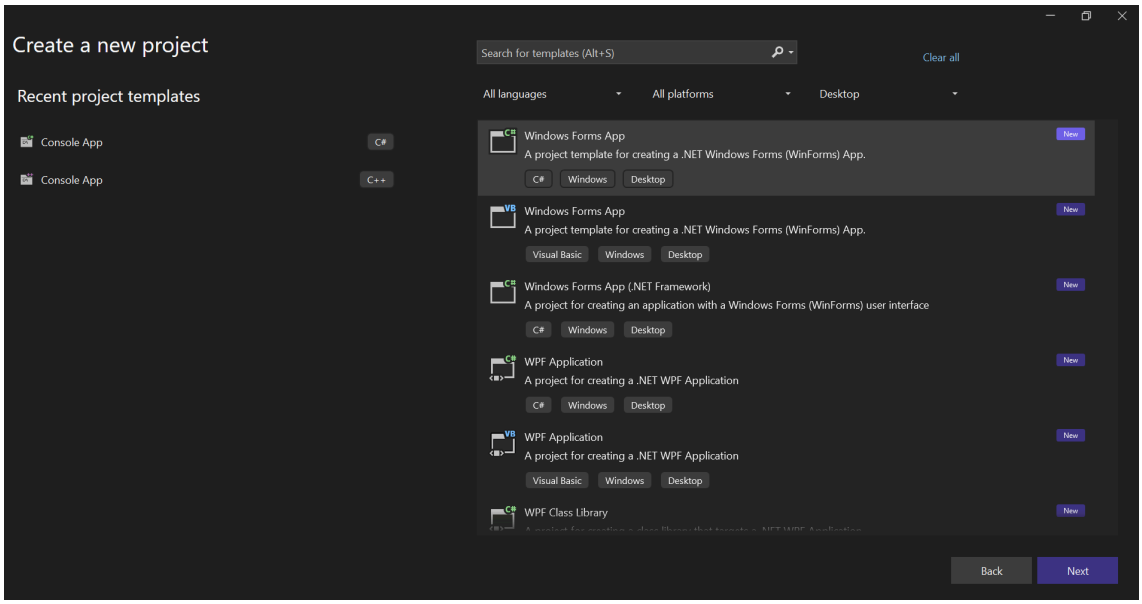
```
│
└── Lab12_2/              # WinForms Alarm Project Folder
    ├── Lab12_2.csproj      # Project build settings/dependencies
    ├── Lab12_2.sln         # Solution file (groups projects)
    ├── Form1.cs            # Form's user code-behind logic
    ├── Form1.Designer.cs   # Designer-generated UI code
    ├── Form1.resx          # Form resources (images, strings)
    ├── Program.cs          # Main application entry point
    ├── Lab12_2.csproj.user # User-specific project settings (optional)
    ├── bin/                # Compiled output binaries folder
    └── obj/                # Intermediate build files folder
```

Below is the step-by-step procedure I followed.

Step 1: Initial Setup

For task1: I created a new C# "Console App" project in Visual Studio, named the project Lab12 and ensured the target .NET framework was suitable. Visual Studio generated the initial project structure, including Program.cs.

For task2: I Created a new C# "Windows Forms App" project in Visual Studio, named the project Lab12_2 and confirmed the target .NET framework. Visual Studio generated the initial project files (Program.cs, Form1.cs, Form1.Designer.cs).



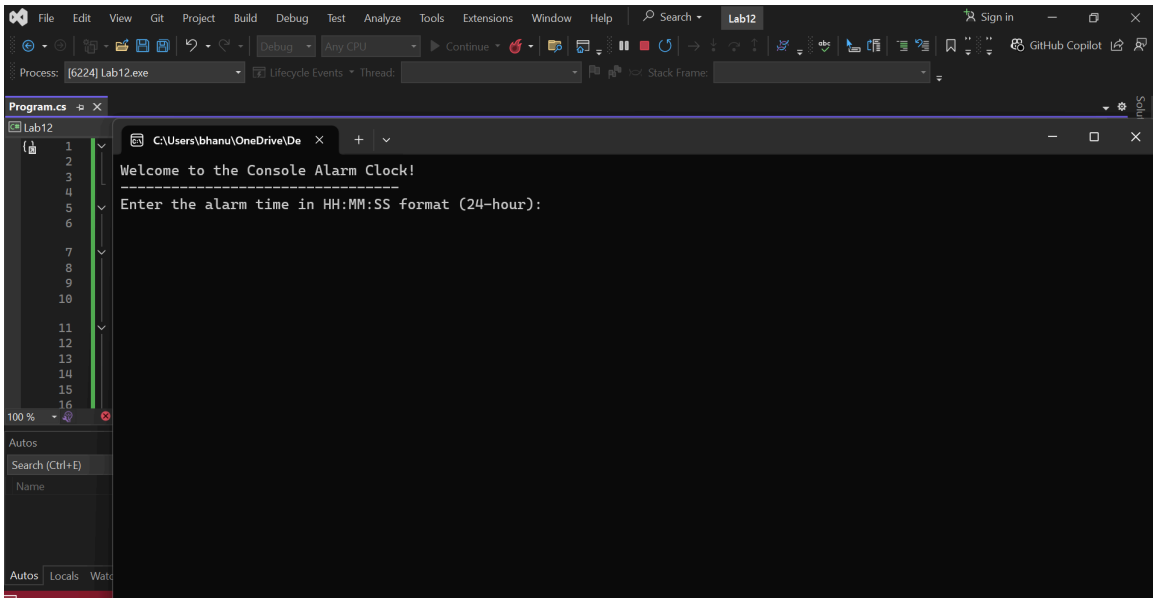
Step 2: Implementing Input, Validation, and Core Logic (Program.cs)

Inside the Main method of Program.cs, I first set up the publisher/subscriber mechanism:

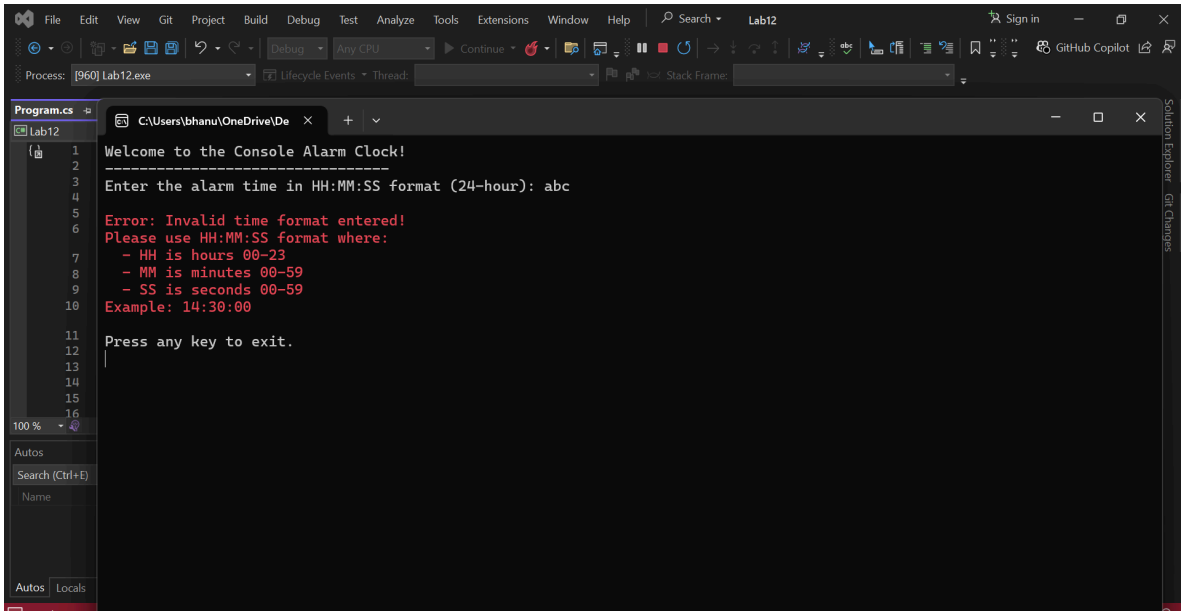
- Declared a static event: public static event Action RaiseAlarm; (Publisher).
- Defined the static event handler method: static void RingAlarm() which prints the alarm message (Subscriber logic).
- Subscribed the RingAlarm method to the RaiseAlarm event: RaiseAlarm += RingAlarm;.

Next, I implemented the user interaction and time handling:

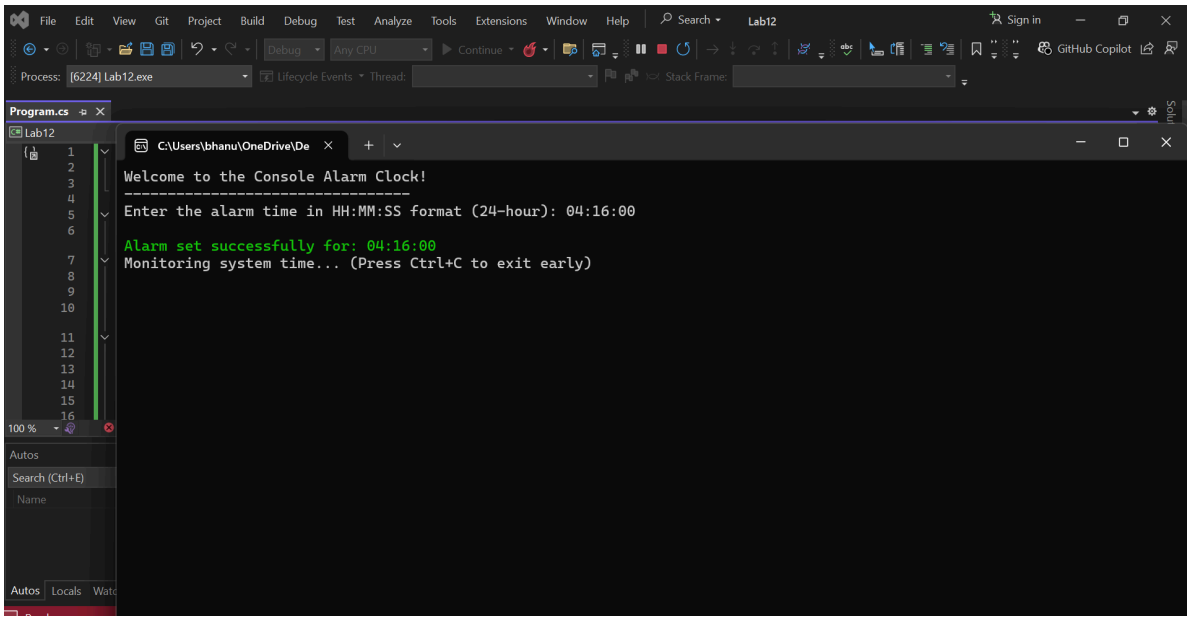
- Prompted the user to enter the alarm time in HH:MM:SS format using Console.WriteLine.



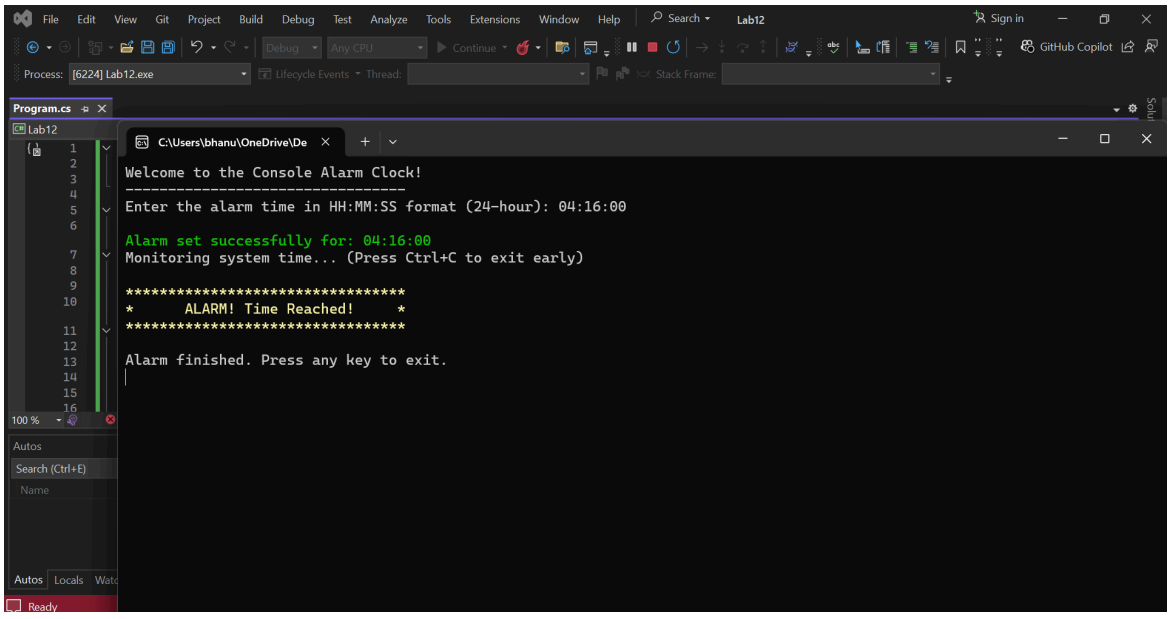
- Read the user's input using Console.ReadLine.
- Validated the input string using a helper method (IsValidTimeFormat) which employed Regular Expressions to ensure the format was strictly HH:MM:SS (24-hour).
- If the input was invalid, I displayed a detailed error message explaining the correct format and exited the application.



- If the input was valid, I parsed the string into a TimeSpan object (alarmTime = TimeSpan.Parse(inputTime);) and confirmed the set time back to the user.



Then, I implemented the time-checking loop by starting an infinite while(true) loop. Inside the loop, I retrieved the current system time's TimeOfDay. To ensure accurate comparison only down to the second (matching the input precision), I truncated the milliseconds/ticks from the current TimeOfDay. I compared the truncated current time (hours, minutes, seconds) with the alarmTime. If they matched, I invoked the RaiseAlarm event (RaiseAlarm?.Invoke();), which triggered the subscribed RingAlarm method, and then used break; to exit the loop.

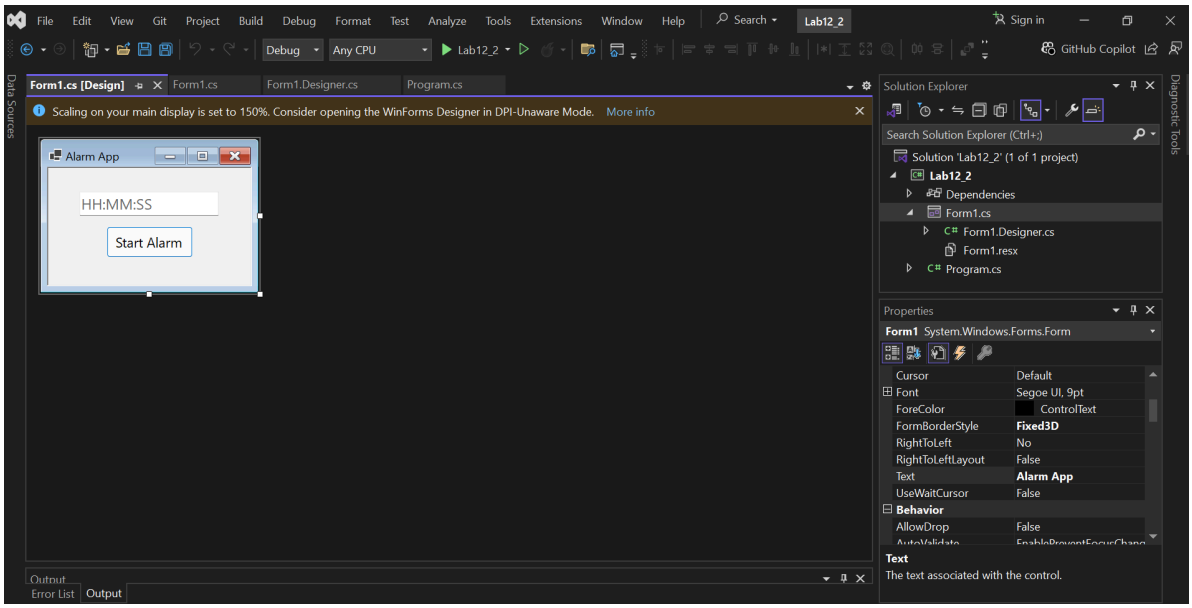


If they didn't match, I paused the execution for approximately one second using Thread.Sleep(1000) before the next iteration of the loop.

I tested various scenarios: entering invalid time formats (e.g., "10:5", "25:00:00", "abc"), entering a valid time a minute or two in the future, and waiting for the alarm to trigger to verify correct validation, event firing, message display, and loop termination.

Step 3: Designing the User Interface (Form1)

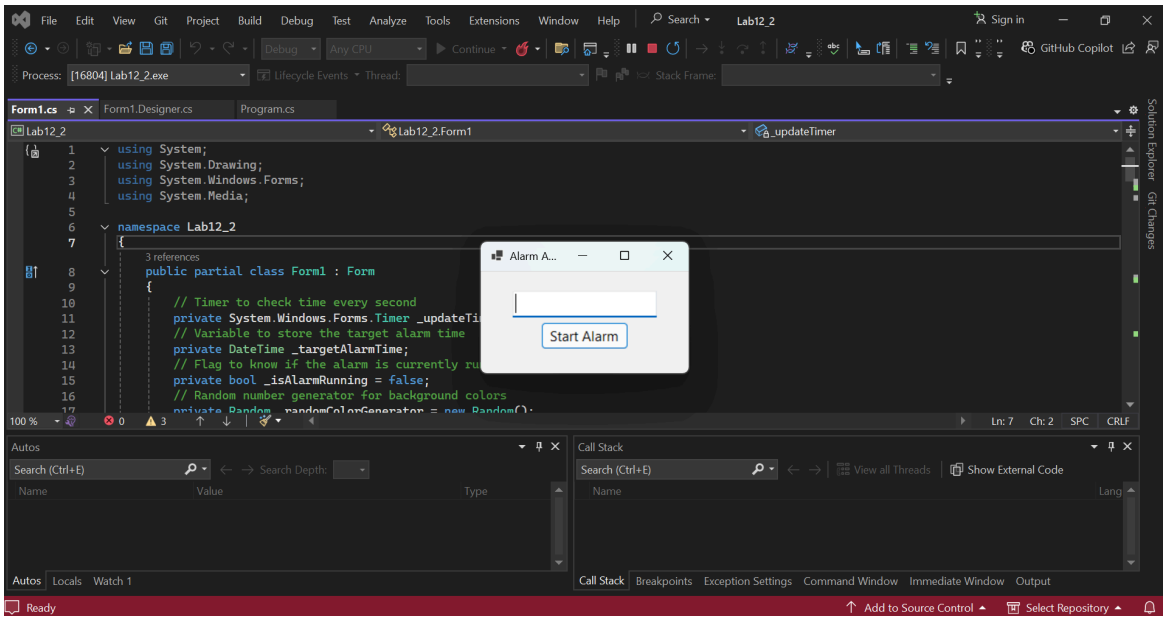
I opened Form1.cs in the Designer and added a TextBox (named txtTime) for time input and a Button (named btnStart) to initiate the alarm from the Toolbox.Then, configured control properties (PlaceholderText, Button Text, Font, Size, Location) via the Properties window for layout and usability. Set Form properties like Title (Text), FormBorderStyle, StartPosition, and AcceptButton.

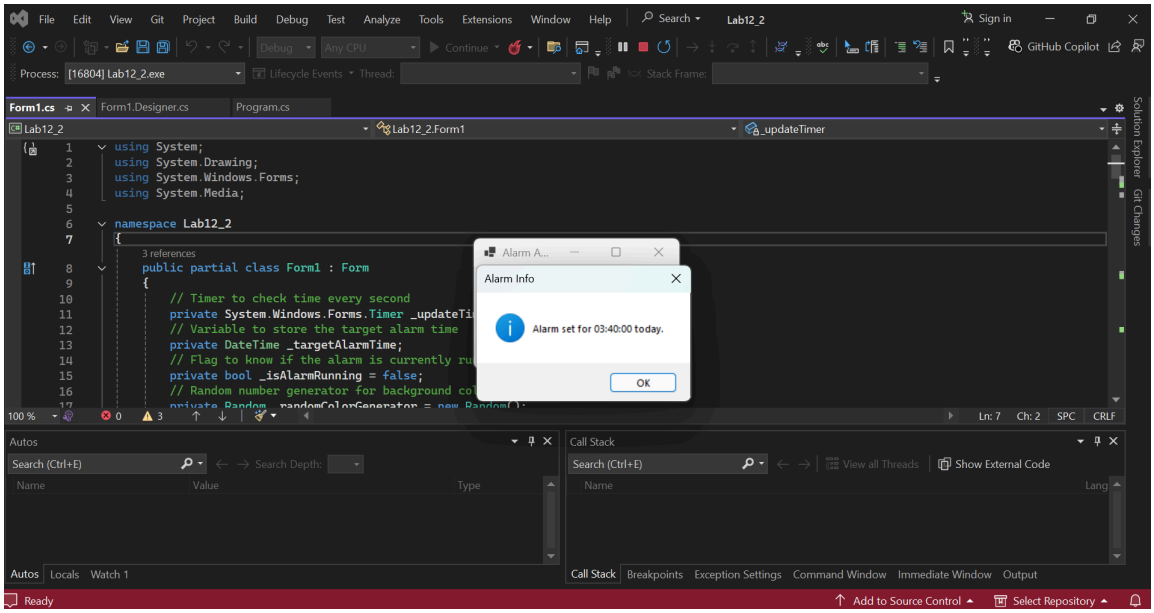


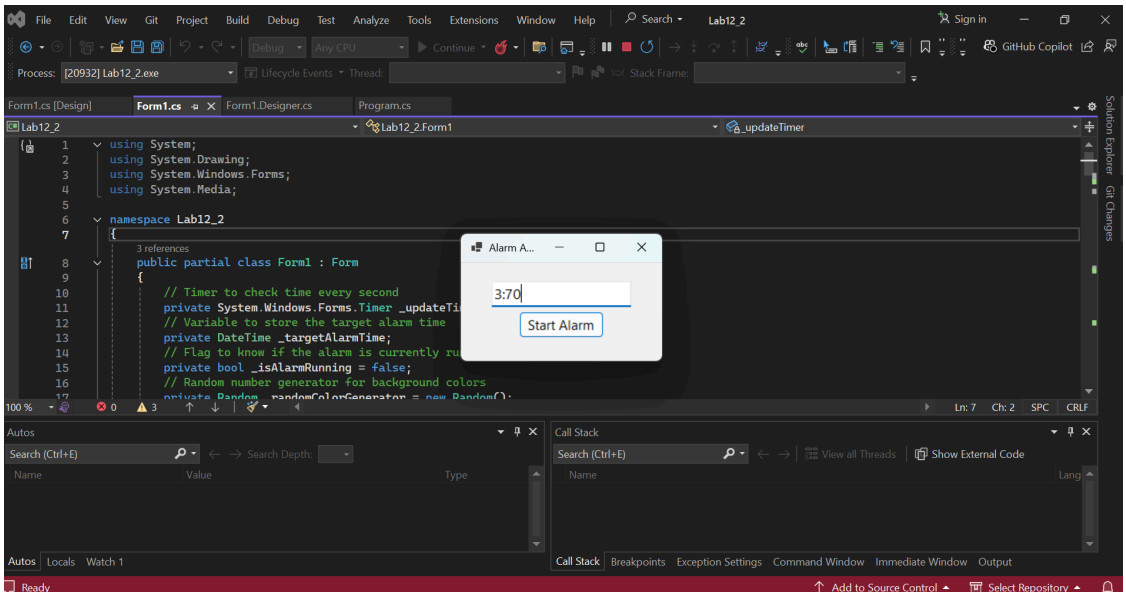
Step 3: Implementing Core Logic (Form1.cs - Code-Behind)

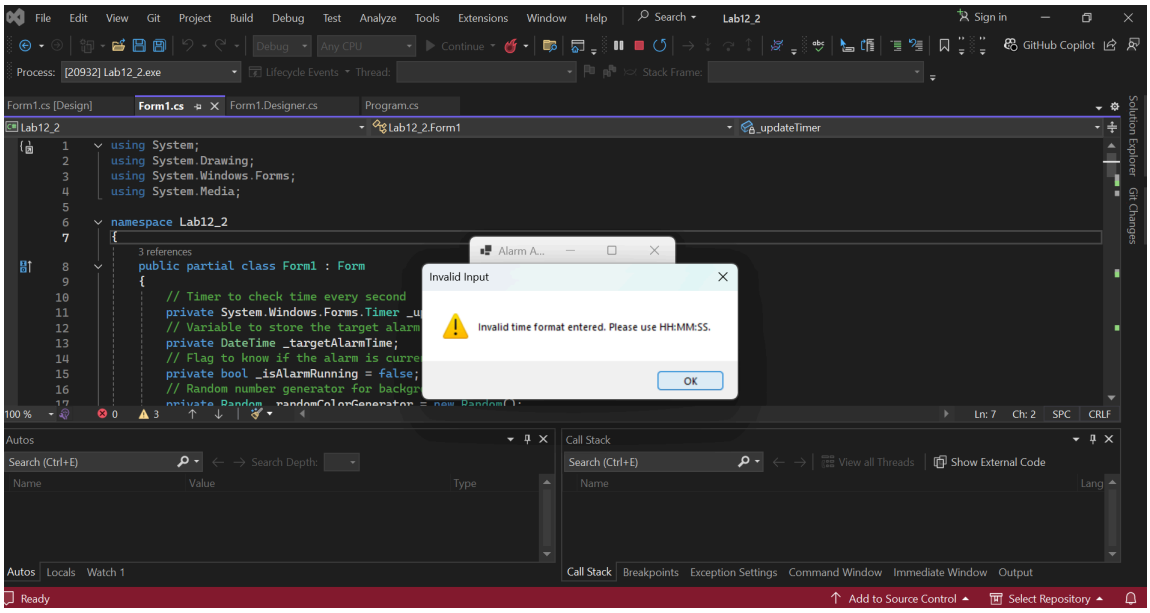
After setting up the visual elements, I switched to the code view of Form1.cs. First, I declared the necessary class-level variables to manage the application's state, such as _updateTimer, _targetAlarmTime, and _isAlarmRunning.

Then, I generated the btnStart_Click event handler by double-clicking the start button in the designer. Inside this btnStart_Click method, I implemented the logic to parse the time input from the txtTime textbox using TimeSpan.TryParse, making sure to handle any invalid input by showing a MessageBox.

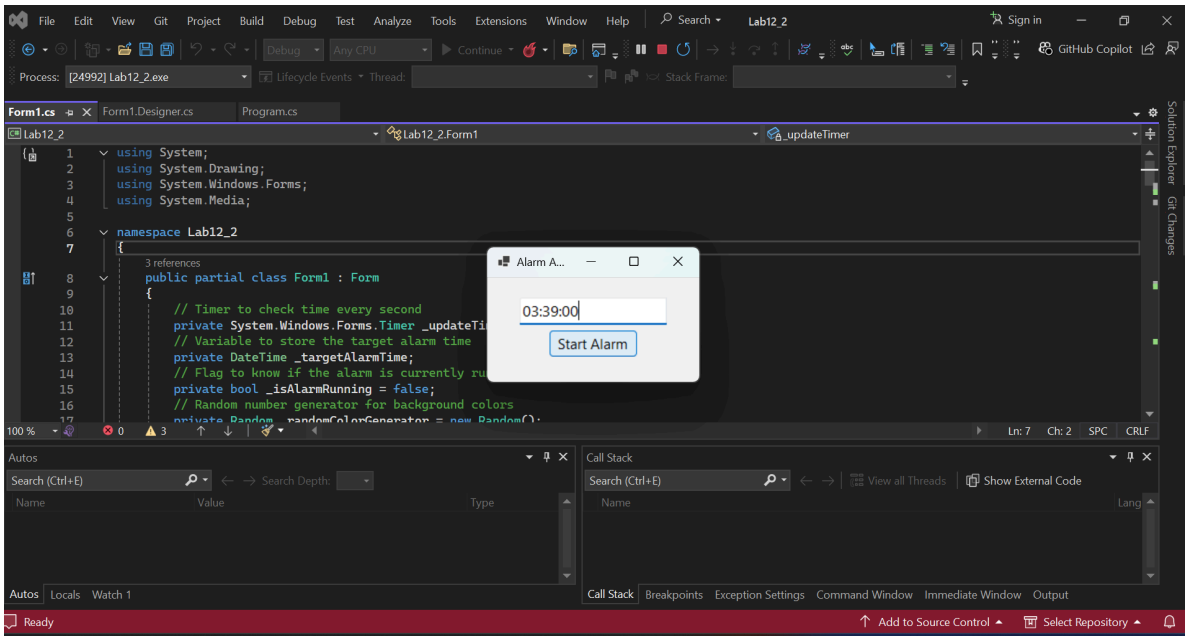

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search
Lab12_2
Sign in
GitHub Copilot
Process: [16804] Lab12_2.exe
Lifecycle Events
Thread:
Stack Frame:
Form1.cs
Form1.Designer.cs
Program.cs
Lab12_2
Lab12_2.Form1
_updateTimer
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Media;
namespace Lab12_2
{
3 references
public partial class Form1 : Form
{
// Timer to check time every second
private System.Windows.Forms.Timer _updateTi
// Variable to store the target alarm time
private DateTime _targetAlarmTime;
// Flag to know if the alarm is currently ru
private bool _isAlarmRunning = false;
// Random number generator for background colors
Alarm A...
Start Alarm
100 %
Ln: 7
Ch: 2
SPC
CRLF
Autos
Search (Ctrl+E)
Search Depth:
Name
Value
Type
Call Stack
View all Threads
Show External Code
Lang
Autos
Locals
Watch 1
Call Stack
Breakpoints
Exception Settings
Command Window
Immediate Window
Output
Ready
Add to Source Control
Select Repository

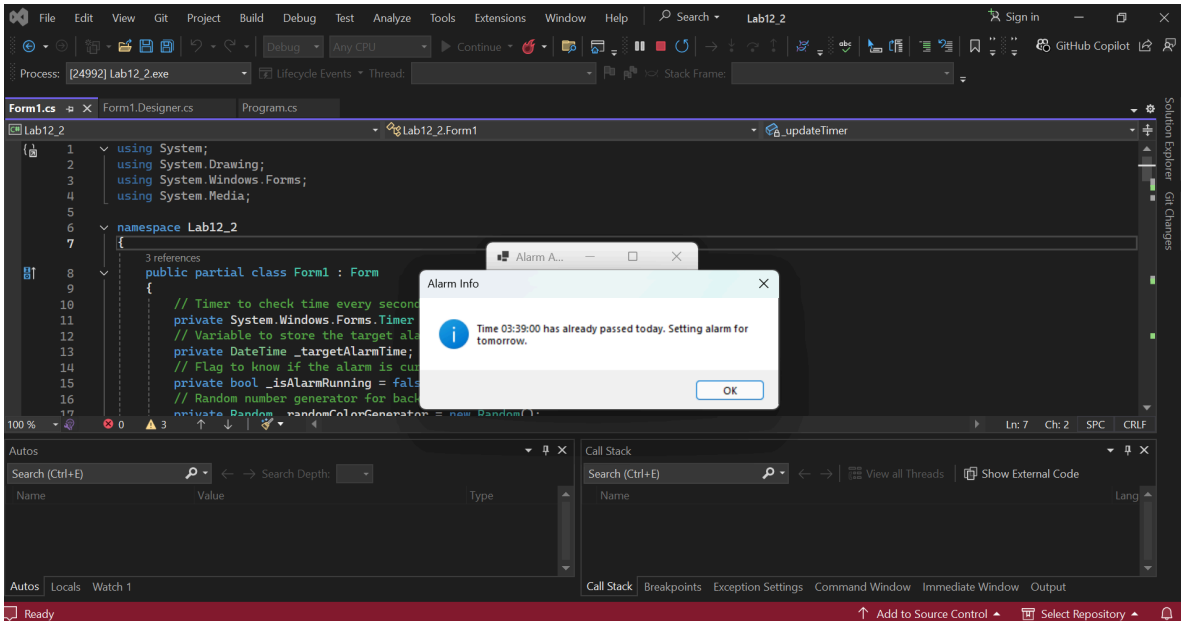

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search
Lab12_2
Sign in
GitHub Copilot
Process: [16804] Lab12_2.exe
Lifecycle Events
Thread:
Stack Frame:
Form1.cs
Form1.Designer.cs
Program.cs
Lab12_2
Lab12_2.Form1
_updateTimer
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Media;
namespace Lab12_2
{
3 references
public partial class Form1 : Form
{
// Timer to check time every second
private System.Windows.Forms.Timer _updateTi
// Variable to store the target alarm time
private DateTime _targetAlarmTime;
// Flag to know if the alarm is currently ru
private bool _isAlarmRunning = false;
// Random number generator for background co
Alarm A...
Alarm Info
Alarm set for 03:40:00 today.
OK
100 %
Ln: 7
Ch: 2
SPC
CRLF
Autos
Search (Ctrl+E)
Search Depth:
Name
Value
Type
Call Stack
View all Threads
Show External Code
Lang
Autos
Locals
Watch 1
Call Stack
Breakpoints
Exception Settings
Command Window
Immediate Window
Output
Ready
Add to Source Control
Select Repository





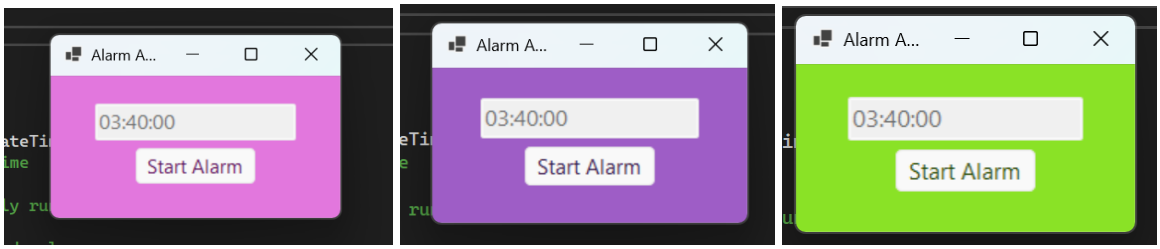
If the input was valid, I calculated the target alarm time (_targetAlarmTime), carefully checking if the time had already passed for today and setting it for tomorrow if needed, before showing a confirmation MessageBox.



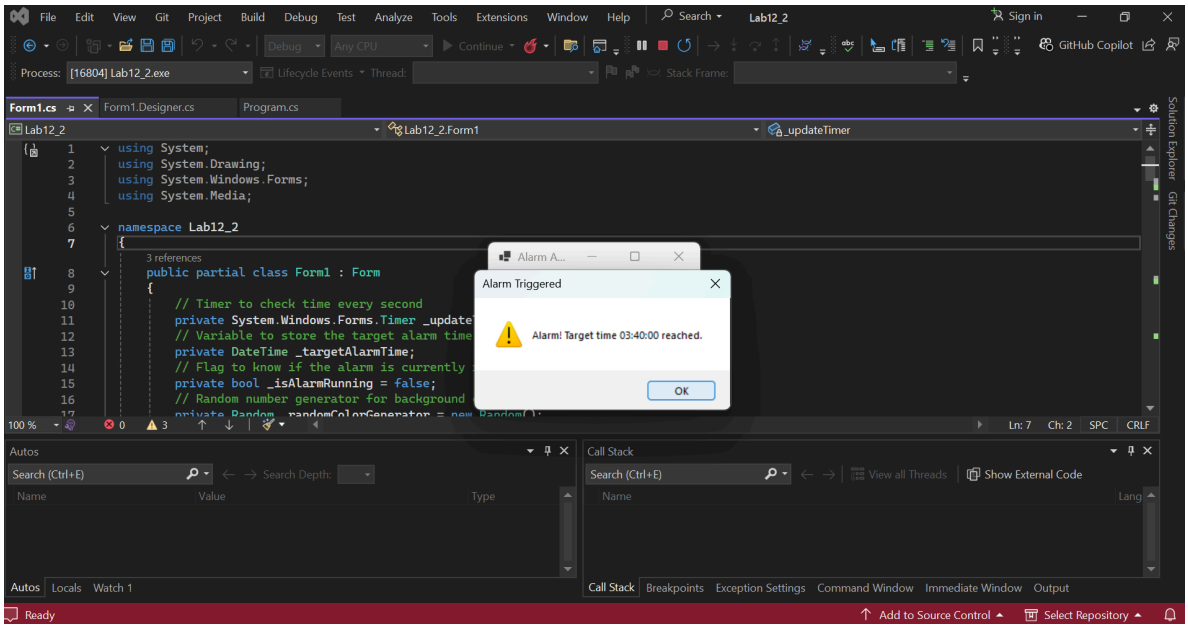


Next, I initialized and started the _updateTimer, set the _isAlarmRunning flag to true, and disabled the input controls. Following that, I created the UpdateTimer_Tick event handler method and assigned it to the timer's Tick event.

In this method, which runs every second, I checked the current time against the target: if the time hadn't been reached, I changed the form's BackColor to a random color.



Once the time was reached, I stopped the timer, reset the running flag, restored the original background color, showed the final "Alarm Triggered" MessageBox, re-enabled the controls, and disposed of the timer.



Lastly, I verified that Program.cs contained the correct application initialization code for the .NET framework I was using and confirmed the Application.Run(new Form1()); call was present to launch the form.

Tested various scenarios: invalid input, valid future time, time already passed today, attempting to restart while running, and reaching the alarm time to verify correct behavior, color changes, messages, and control state changes.

---

3. Results and Analysis

The project successfully resulted in the development of two distinct C# applications fulfilling the specified alarm clock requirements:

Task 1 (Console Application): A functional console application was produced. It correctly prompts the user for a time in HH:MM:SS format, validates the input using regular expressions, and continuously monitors the system time. Upon matching the target time, it successfully raises a

custom event (RaiseAlarm). This event triggers the subscribed RingAlarm method, which displays the specified alarm message in the console.

Task 2 (Windows Forms Application): A functional GUI application using Windows Forms was developed. The application presents a user-friendly interface with a TextBox for time input and a Button to initiate the alarm. Input validation (TimeSpan.TryParse) occurs upon button click. When started, the application correctly disables input controls and utilizes a System.Windows.Forms.Timer to trigger an event every second. This event handler changes the form's background color randomly, providing continuous visual feedback. The timer's event handler also checks the current time against the target time. Upon reaching the target, the timer stops, the background color reverts to its original state, controls are re-enabled, and a confirmation MessageBox is displayed. It also handles the edge case of the entered time having already passed for the current day.

---

4. Discussion and Conclusion

Challenges and Reflections

- Initially, in Task 2, configuring the WinForms project correctly involved resolving compiler errors related to missing method definitions (btnStart_Click) and ensuring the Form1 class properly inherited from System.Windows.Forms.Form. Differences between .NET Framework and modern .NET (.NET 6+) initialization (ApplicationConfiguration) also required clarification.
- Implementing accurate time comparison, especially handling the case where the target time might have already passed today and needed to be set for the next day, required careful logic.
- Ensuring the GUI remained responsive in Task 2 necessitated understanding the difference between Thread.Sleep (unsuitable for GUI) and the System.Windows.Forms.Timer. Correctly implementing the event wiring (both custom events in Task 1 and control/timer events in Task 2) was also a key learning point.

Lessons Learned

The process highlighted the importance of understanding the underlying framework and programming paradigm (console vs. event-driven GUI). Debugging compile-time errors, particularly in the WinForms designer interaction, emphasized the need for careful code structure and understanding how partial classes work. The transition from the direct execution flow of the console app to the asynchronous, event-based nature of the WinForms app was a significant conceptual step. It became clear how crucial appropriate background processing techniques (like the WinForms Timer) are for creating usable GUI applications.

Summary

This project involved the development of two C# alarm clock applications using different approaches: a console application and a Windows Forms GUI application. Both applications successfully accepted user-defined time input, monitored system time, and triggered an alarm

notification upon reaching the target time. Key C# concepts including event handling (custom events and control events), time manipulation (DateTime, TimeSpan), input validation (Regex, TryParse), and timing mechanisms (Thread.Sleep, System.Windows.Forms.Timer) were utilized. The project effectively demonstrated the implementation of time-based, event-driven logic in both console and graphical environments, meeting all specified objectives.

---

5. References

- Lecture 11 Slides
- https://learn.microsoft.com/en-us/dotnet/csharp
- Google ai studio