

# Lab 11: Debugging C# Console Games: Bug Hunting Report

Bhounik Patidar

April 22, 2025

## 1 Introduction, Setup, and Tools

### 1.1 Overview

This report documents the process of identifying and fixing bugs in five different C# console games. The objective was to analyze control flow using Visual Studio Debugger and identify bugs that cause unexpected game behavior.

### 1.2 Environment Setup

The debugging was performed in the following environment:

- Operating System: Windows 10
- IDE: Visual Studio 2022 Community Edition
- .NET Version: 6.0
- Games Source: [dotnet-console-games repository](#)

### 1.3 Tools Used

- Visual Studio Debugger (Breakpoints, Step Into/Over/Out, Watch Window)
- Locals and Autos Windows for variable inspection
- Immediate Window for expression evaluation

## 2 Methodology and Execution

The debugging process followed a consistent methodology for each game:

1. Reproduce the bug
2. Set strategic breakpoints
3. Step through execution
4. Analyze variable states
5. Identify root cause
6. Implement and verify fix

## 2.1 Program 1: Rock Paper Scissors

### 2.1.1 Bug Description

When player chose Rock and computer chose Scissors, the game incorrectly displayed "You lose" instead of "You win".

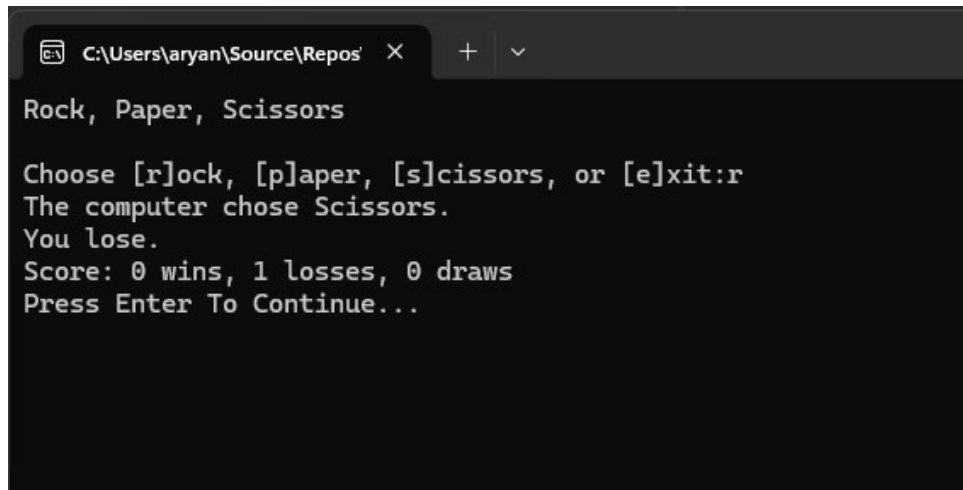


Figure 1: When player chooses rock against computers scissors, it should be a win, which is not happening - a bug

### 2.1.2 Debugging Process

1. Set breakpoint at the win condition check:

```
1 switch (playerMove, computerMove)
2 {
3     case (Rock, Paper) or (Paper, Scissors) or (Scissors, Rock):
4         Console.WriteLine("You lose.");
5         losses++;
6         break;
```

2. Observed variables:

- playerMove = Rock
- computerMove = Scissors

3. Identified the reversed logic in win/lose conditions

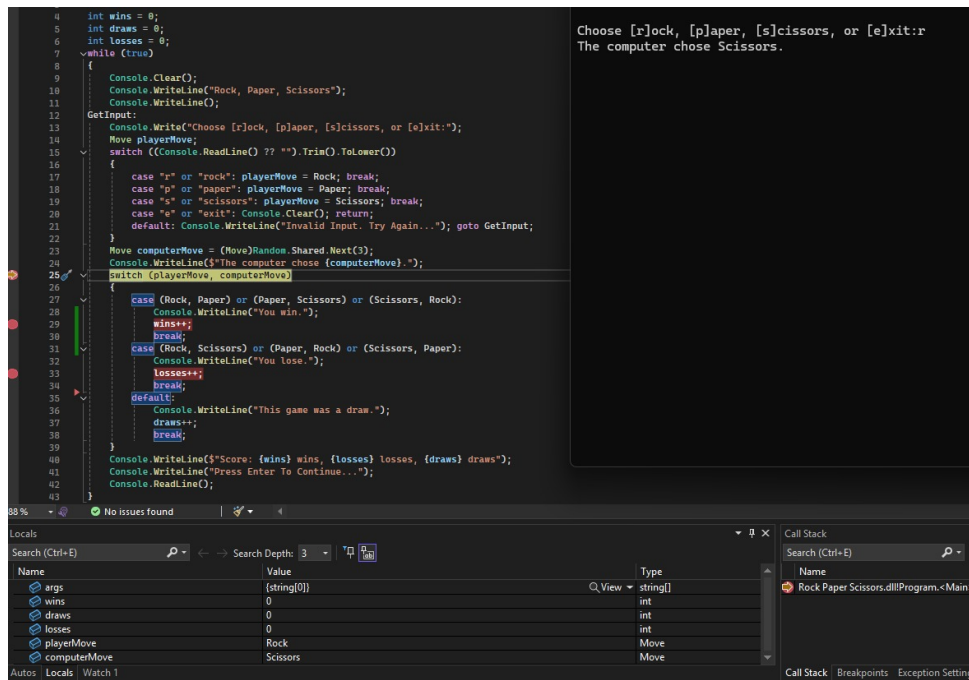


Figure 2: Illustration of using step into and step over to analyze the execution of the program

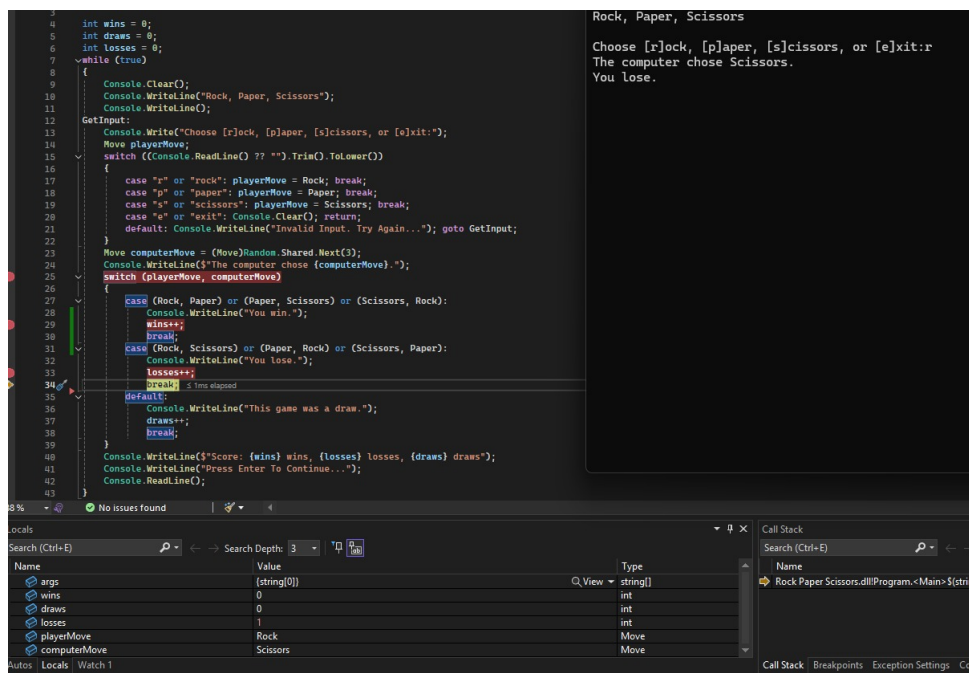


Figure 3: Observing the local variables I am able to identify the issue of the incorrect logic

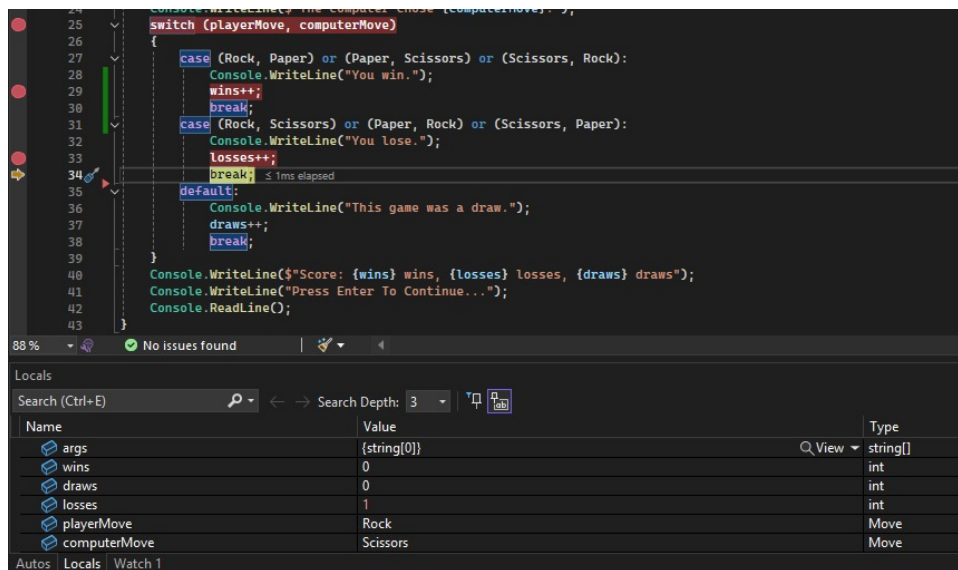


Figure 4: The issue is that the logic is reversed

### 2.1.3 Fix Implemented

Corrected the switch cases to properly handle win conditions:

```

1 case (Rock, Scissors) or (Paper, Rock) or (Scissors, Paper):
2     Console.WriteLine("You win.");
3     wins++;
4     break;

```

## 2.2 Program 2: Dice Game

### 2.2.1 Bug Description

When both player and rival rolled the same number (draw), the rival's score incorrectly increased.

```

Rival rolled a 3
Press any key to roll the dice...
You rolled a 1
The Rival won this round.
The score is now - You : 2. Rival : 1.
Press any key to continue...

Round 4
Rival rolled a 3
Press any key to roll the dice...
You rolled a 6
You won this round.
The score is now - You : 3. Rival : 1.
Press any key to continue...

Round 5
Rival rolled a 2
Press any key to roll the dice...
You rolled a 4
You won this round.
The score is now - You : 4. Rival : 1.
Press any key to continue...

Round 6
Rival rolled a 2
Press any key to roll the dice...
You rolled a 2
This round is a draw!
The score is now - You : 4. Rival : 2.
Press any key to continue...

```

Figure 5: When both player get the same value, it does show that the round ended up in a draw but also increases rivals score - a bug

### 2.2.2 Debugging Process

1. Set breakpoint at score update:

```

1 else
2 {
3     Console.WriteLine("This round is a draw!");
4     wins++; // Bug: Should be draws++
5 }

```

2. Verified through multiple test cases
3. Traced incorrect variable increment

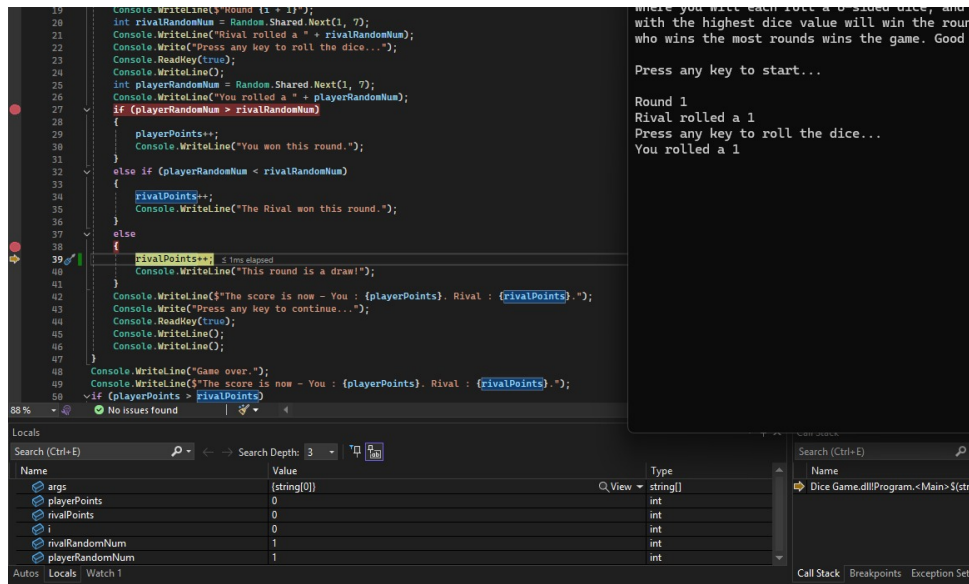


Figure 6: Further insertion of breakpoints and debugging using step into, step over and step out and observing the local variables to pin point the issue, which is the invalid increase of rivals score in the else statement

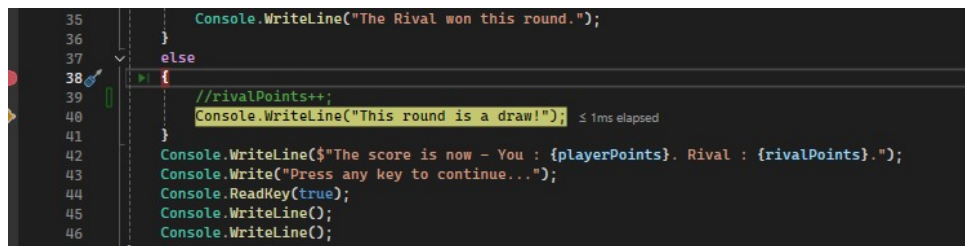


Figure 7: Fixing the incorrect logic fixes the issue

## 2.2.3 Fix Implemented

Corrected the score increment:

```

1 else
2 {
3     Console.WriteLine("This round is a draw!");
4     draws++;
5 }

```

## 2.3 Program 3: Tower of Hanoi

### 2.3.1 Bug Description

When attempting invalid moves (placing larger disk on smaller one), the game displayed error but still updated internal state incorrectly.

# Tower Of Hanoi

Minimum Moves: 7

Moves: 1



[1], [2], or [3] select source stack  
[home] restart current puzzle  
[end] back to menu  
[escape] exit game

Figure 8: Pertaining to the first move 1-2

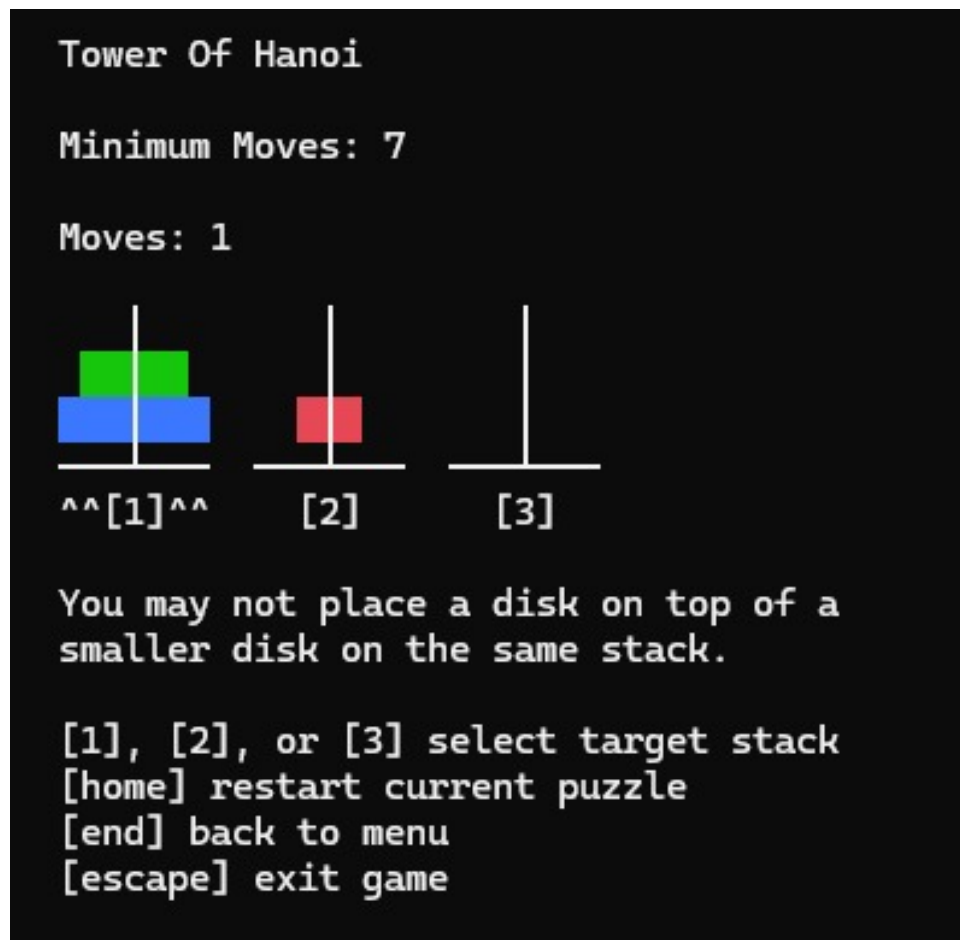


Figure 9: Pertaining to the second move 1-2, this moves the middle disk to the central tower, which is an invalid move. See the output it seems that the program correctly tells us that this is an invalid move and would have them actually negated this move



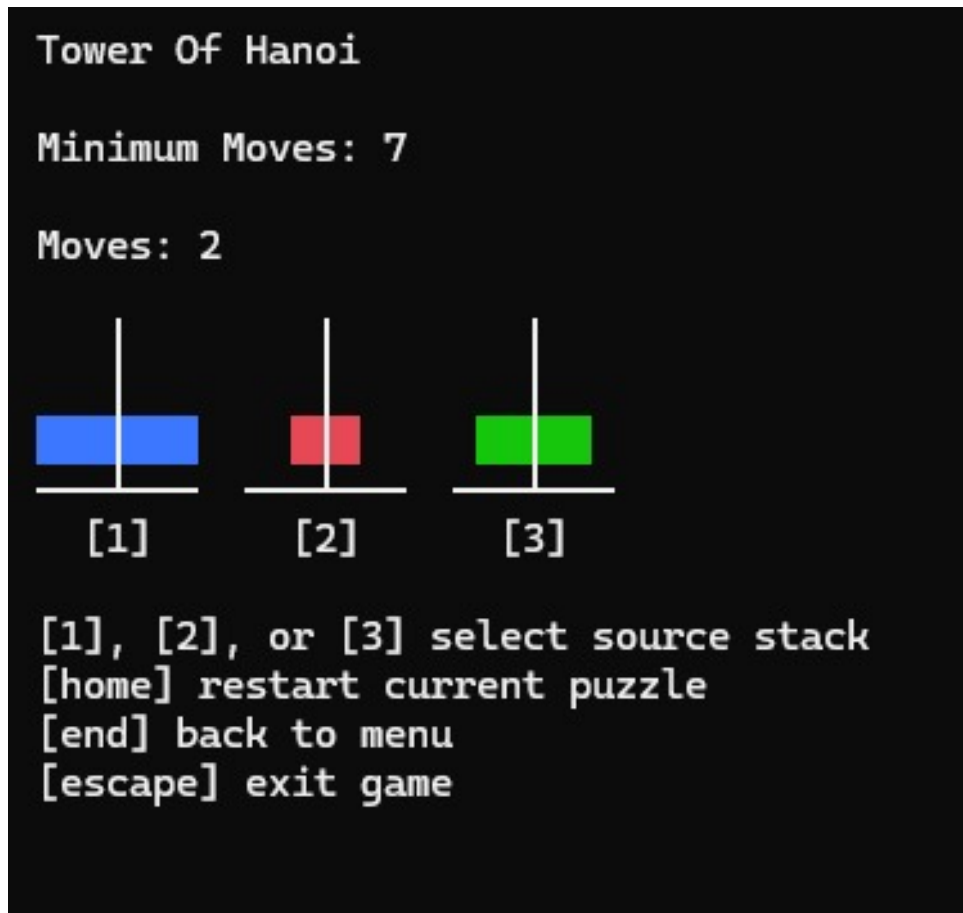


Figure 10: Pertaining to the move 2-3. This is where the bug is identified. Even when I made the move 2-3, the program "appears" to have executed the move 1-3 - a bug.

### 2.3.2 Debugging Process

1. Set breakpoints at move validation:

```

1 else if (source is not null &&
2     (stacks[stack].Count is 0 || stacks[source.Value][^1] < stacks[stack][^1]))

```

2. Reproduced with sequence:

- Move 1 → 2
- Attempt 1 → 2 again (invalid)
- Move 2 → 3 showed unexpected behavior

3. Discovered missing state rollback after invalid move

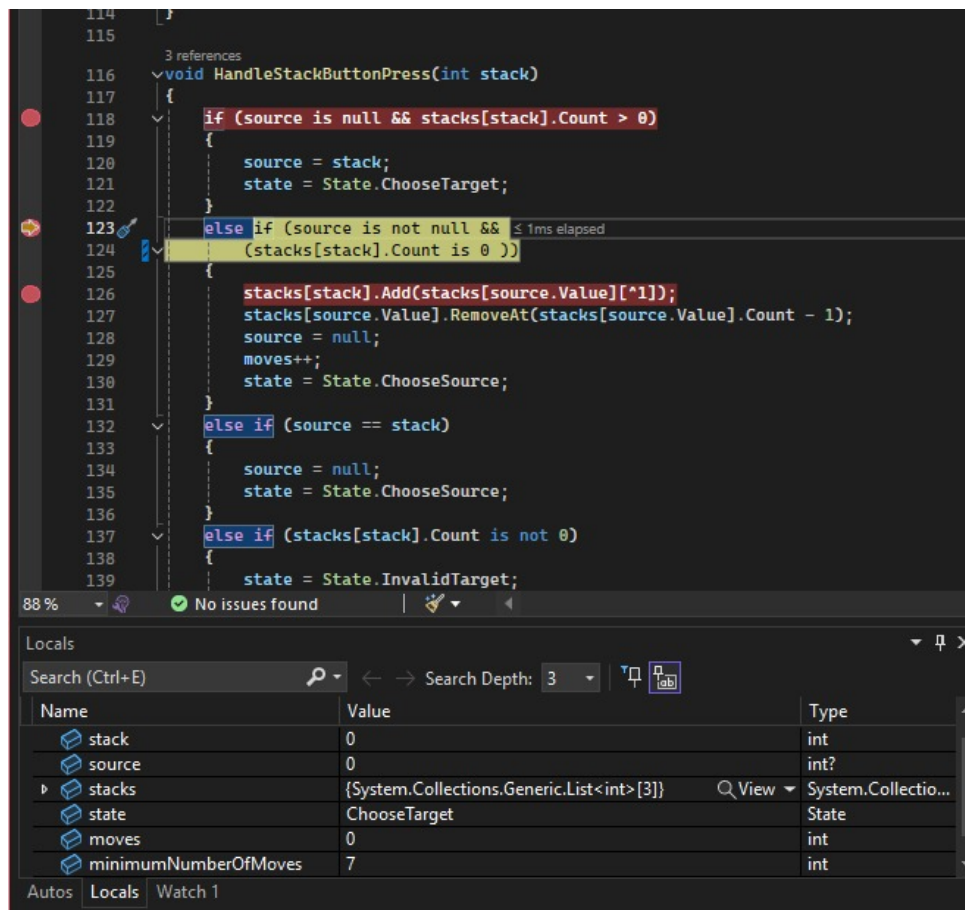


Figure 11: The issue was actually in the second step itself although it seemed to have occurred at the third move. Debugging procedure helped identify that the program lacks a check condition and as a result although it displays correctly, internally it is allowing the movement of a larger disk over a smaller disk which is causing the error.

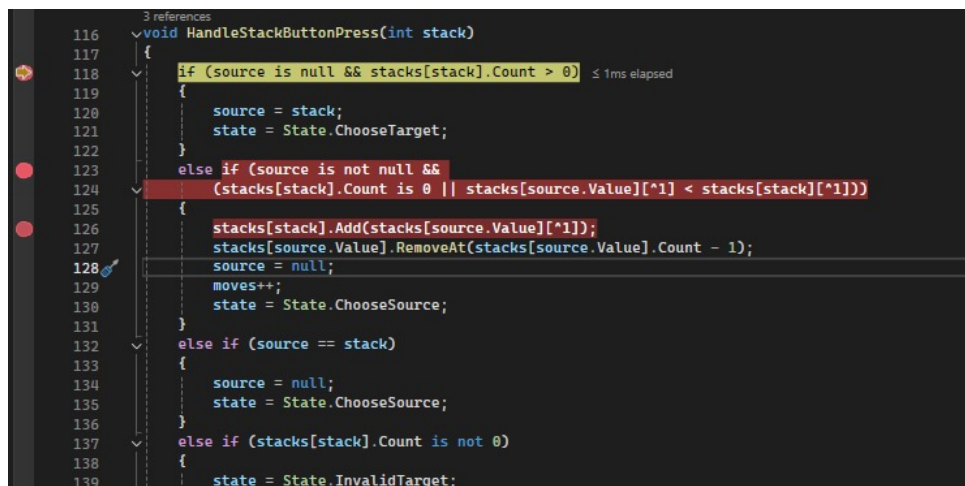


Figure 12: Inserting the additional check condition fixes the bug

### 2.3.3 Fix Implemented

Added proper state handling for invalid moves:

```

1 else if (source is not null &&
2         (stacks[stack].Count is 0 || stacks[source.Value][^1] < stacks[stack][^1]))

```

```

3 {
4     // Valid move logic
5 }
6 else
7 {
8     // Clear selection without updating state
9     source = null;
10    state = State.ChooseSource;
11 }

```

## 2.4 Program 4: Flash Cards

### 2.4.1 Bug Description

Case-sensitive comparison rejected correct answers (e.g., "juliett" marked wrong when correct answer was "Juliett").

```

What is the NATO phonetic alphabet code word for...

J? juliett

Incorrect. :( J -> Juliett
Press [enter] to continue or [escape] to return to main
menu...

```

Figure 13: The user enters the correct input against J, but still gets an error - a bug

### 2.4.2 Debugging Process

1. Set breakpoint at answer validation:

```
1 if (input.Trim().Equals(array[index].CodeWord))
```

2. Observed string comparison behavior
3. Identified missing case-insensitive flag

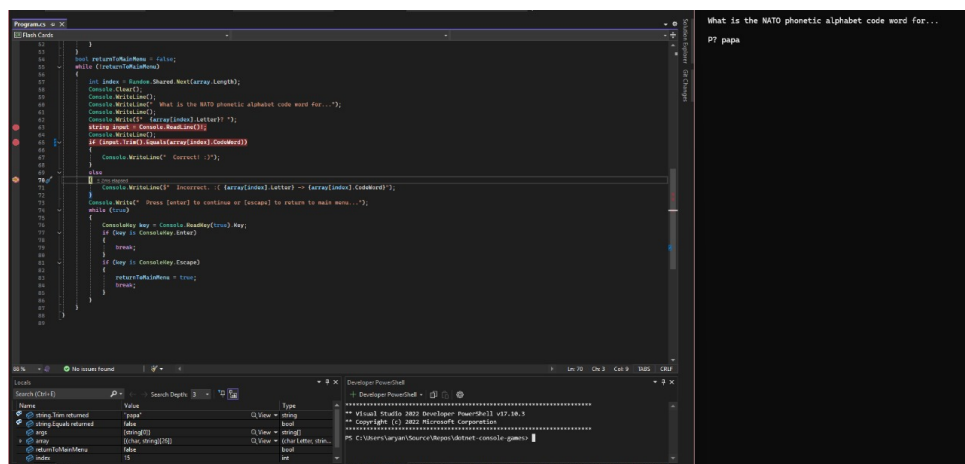


Figure 14: Debugging by inserting breakpoints and observing the local variables I identified that `returnToMainMenu` is incorrectly set to false even after correct input because the first character is lowercase it the program tries to match it to the exact string "Papa"

```

63 string input = Console.ReadLine();
64 Console.WriteLine();
65 if (input.Trim().Equals(array[index].CodeWord, StringComparison.CurrentCultureIgnoreCase))
66 {
67     Console.WriteLine(" Correct! :");
68 }
69 else
70 {
71     Console.WriteLine($" Incorrect. :( {array[index].Letter} -> {array[index].CodeWord}");
72 }

```

Figure 15: Insertion of ignore casing fixes the bug

### 2.4.3 Fix Implemented

Added case-insensitive comparison:

```

1 if (input.Trim().Equals(array[index].CodeWord,
2     StringComparison.CurrentCultureIgnoreCase))

```

## 2.5 Program 5: Tic Tac Toe

### 2.5.1 Bug Description

Game didn't recognize winning condition when three X's appeared diagonally, continuing to ask for input.



Figure 16: As user had completed a trio of X, he should be declared won but the program still asks for next user input - a bug

### 2.5.2 Debugging Process

1. Set breakpoint at win check:

```

1 private bool CheckForWin()

```

2. Verified win conditions were incomplete
3. Traced missing diagonal check

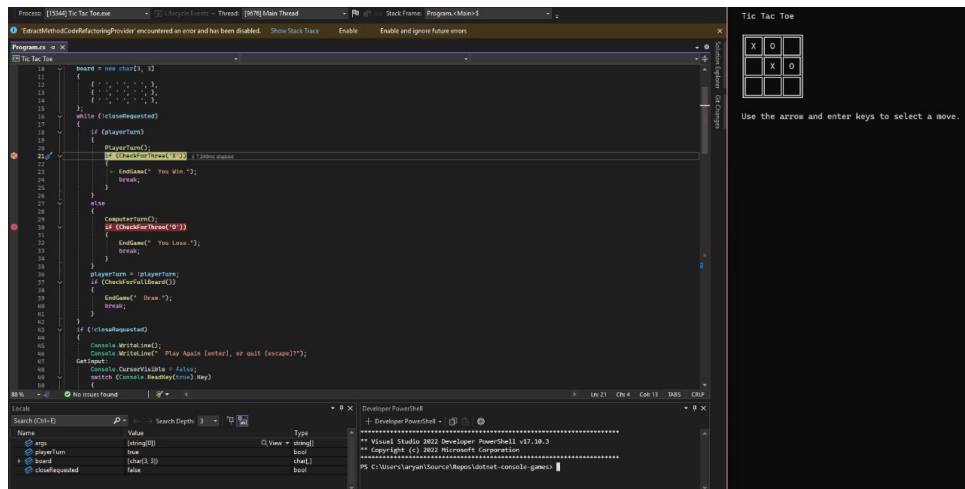


Figure 17: Debugging and inserting breakpoints helps reach the point where I can simulate the buggy situation.

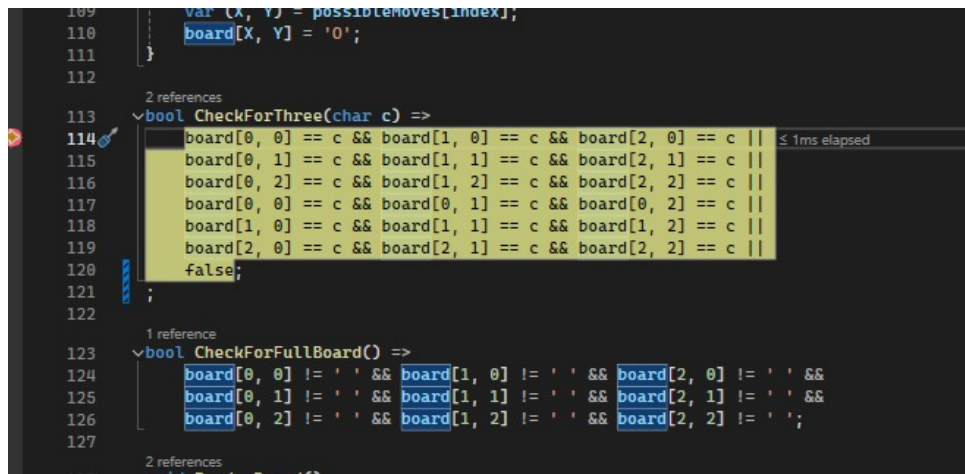


Figure 18: Analyzing the cases, I identify the cases for identifying diagonal wins are missing. This was resulting in the buggy working of the program

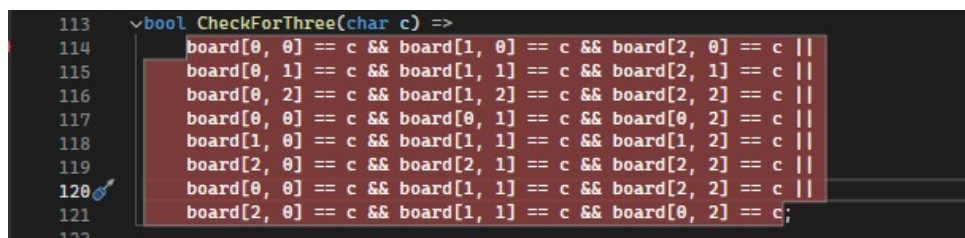


Figure 19: Added the conditions for diagonal trio which fixes the bug

### 2.5.3 Fix Implemented

Added diagonal win conditions:

```
1 // Check diagonals
2 if (board[0,0] == player && board[1,1] == player && board[2,2] == player)
3     return true;
4 if (board[0,2] == player && board[1,1] == player && board[2,0] == player)
5     return true;
```

## 3 Results and Analysis

### 3.1 Key Findings

- Most bugs stemmed from incomplete conditional logic
- Boundary conditions were frequently overlooked
- State management errors were common in turn-based games
- String comparison issues appeared in multiple games

### 3.2 Debugging Insights

- Strategic breakpoint placement was crucial
- Watch window proved invaluable for tracking state changes
- Step-over was most used debugging operation
- Immediate window helped test fixes quickly

## 4 Understanding the Entry Point in Modern C#

### 4.1 The Entry Point Question

During the debugging process, an important observation was made regarding the entry point in these console applications. Traditional C# programs typically have an explicit `Main()` method as their entry point, but many of these games lacked this visible structure. This raised the question: *If there is no `Main()` method in the program, where exactly is the entry point?*

### 4.2 Top-Level Statements

Modern C# (from .NET 6 onward) introduces **top-level statements**, which simplify the program structure:

- The compiler automatically generates the `Main()` method behind the scenes
- The first executable statement becomes the entry point
- This feature is particularly common in console applications

### 4.3 Examples from Debugged Games

#### 1. Rock Paper Scissors:

```
1 int wins = 0; // This is effectively the entry point
2 int draws = 0;
3 int losses = 0;
```

#### 2. Dice Game:

```
1 int playerPoints = 0; // Entry point
2 int rivalPoints = 0;
```

#### 3. Flash Cards:

```
1 (char Letter, string CodeWord)[] array = new[] // Entry point
2 {
3     ('A', "Alpha"), ('B', "Bravo"), ...
4 };
```

## 4.4 Debugging Implications

This structural change has several implications for debugging:

- Breakpoints can be set directly on the first line of code
- The debugger starts execution at the first statement
- The `Program` class and `Main()` method still exist in the compiled IL
- No functional difference in debugging experience

## 5 Discussion and Conclusion

### 5.1 Challenges Faced

- Identifying subtle state management bugs
- Reproducing timing-dependent issues
- Understanding game-specific logic

### 5.2 Lessons Learned

- Importance of boundary condition testing
- Value of defensive programming
- Need for comprehensive win/lose condition checks
- Benefits of case-insensitive string comparison

### 5.3 Conclusion

This debugging exercise demonstrated the importance of thorough testing and careful logic implementation in game development. The Visual Studio debugger proved to be a powerful tool for identifying and resolving these issues.

# Event-Driven Programming in C# Windows Forms

Bhoumik Patidar

April 22, 2025

## 1 Introduction, Setup, and Tools

### 1.1 Overview

This report documents the implementation of an alarm clock application using event-driven programming in C#, covering both console and Windows Forms versions. The assignment demonstrates the publisher-subscriber pattern and event handling in .NET applications.

### 1.2 Environment Setup

The development environment consisted of:

- Windows 10 Operating System
- Visual Studio 2022 Community Edition
- .NET 6.0 SDK
- Windows Forms App (.NET Framework) template

### 1.3 Learning Objectives Achieved

- Implemented publisher-subscriber pattern in console application
- Created event-driven Windows Forms application
- Demonstrated timer-based events and UI updates
- Validated user input in both implementations

## 2 Methodology and Execution

### 2.1 Part 1: Console Application

#### 2.1.1 Design Approach

The console application follows the publisher-subscriber model with these components:

- **Publisher:** AlarmClock class that monitors system time
- **Event:** RaiseAlarm triggered when target time matches system time
- **Subscriber:** RingAlarm method that handles the event



### 2.1.2 Key Code Implementation

```
1 class AlarmClock
2 {
3     private readonly string _targetTime;
4     public event EventHandler? RaiseAlarm;
5
6     protected virtual void OnRaiseAlarm()
7     {
8         RaiseAlarm?.Invoke(this, EventArgs.Empty);
9     }
10
11     public void Start()
12     {
13         while (true)
14         {
15             if (DateTime.Now.ToString("HH:mm:ss") == _targetTime)
16             {
17                 OnRaiseAlarm();
18                 break;
19             }
20             Thread.Sleep(1000);
21         }
22     }
23 }
```

Listing 1: Publisher class implementation

```
1 static void Main(string[] args)
2 {
3     var alarm = new AlarmClock(input);
4     alarm.RaiseAlarm += RingAlarm;
5     alarm.Start();
6 }
7
8 private static void RingAlarm(object? sender, EventArgs e)
9 {
10     Console.WriteLine("        Alarm! Time's up!        ");
11 }
```

Listing 2: Subscriber implementation

### 2.1.3 Execution Flow

1. User enters time in HH:mm:ss format
2. AlarmClock starts monitoring system time
3. When times match, RaiseAlarm event is triggered
4. RingAlarm handler executes with alarm message

## 2.2 Part 2: Windows Forms Application

### 2.2.1 Design Approach

The Windows Forms version enhances the console application with:

- Graphical user interface with input validation
- Timer-based background color changes
- Message box notification on alarm trigger
- Proper event handling for button clicks

### 2.2.2 Key Components

```
1 private void InitializeComponent()
2 {
3     // Form settings
4     this.Text = "Alarm Clock";
5     this.ClientSize = new Size(320, 120);
6
7     // Timer setup
8     timer = new System.Windows.Forms.Timer()
9     {
10         Interval = 1000,
11         Enabled = false
12     };
13     timer.Tick += Timer_Tick;
14 }
```

Listing 3: Form initialization

```
1 private void ButtonStart_Click(object? sender, EventArgs e)
2 {
3     if (!TimeSpan.TryParseExact(
4         textBoxTime.Text.Trim(),
5         "hh\\:mm\\:ss",
6         null,
7         out targetTime))
8     {
9         MessageBox.Show("Invalid time format");
10        return;
11    }
12    timer.Start();
13 }
14
15 private void Timer_Tick(object? sender, EventArgs e)
16 {
17     this.BackColor = Color.FromArgb(
18         rng.Next(256), rng.Next(256), rng.Next(256));
19
20     if (DateTime.Now.TimeOfDay >= targetTime)
21     {
22         timer.Stop();
23         MessageBox.Show("    Alarm! Time's up!    ");
24     }
25 }
```

Listing 4: Event handlers

### 2.2.3 Execution Flow

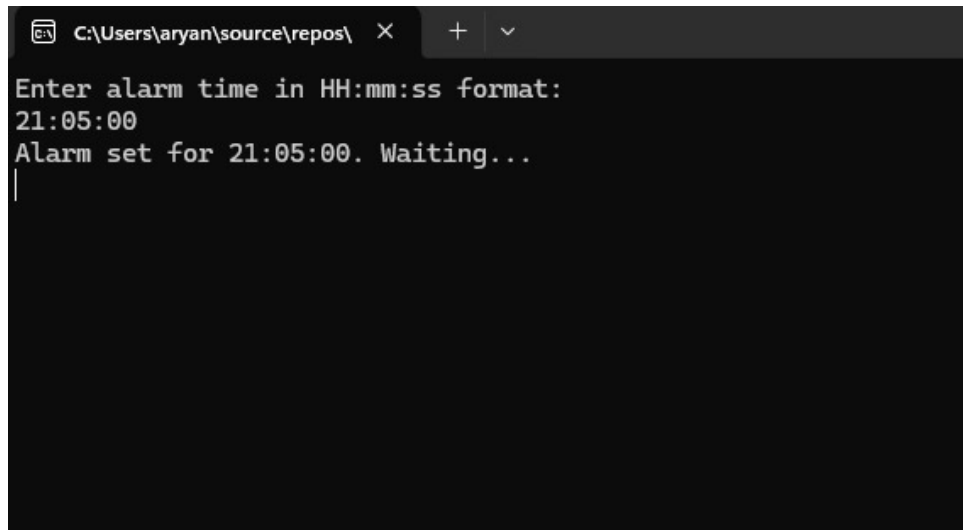
1. User enters time in textbox and clicks Start
2. Input validation checks HH:mm:ss format
3. Timer begins 1-second interval ticks
4. Each tick changes form background color randomly
5. On target time match:
  - Timer stops hangs stop
  - Message box displays alarm

## 3 Results and Analysis

### 3.1 Console Application Results

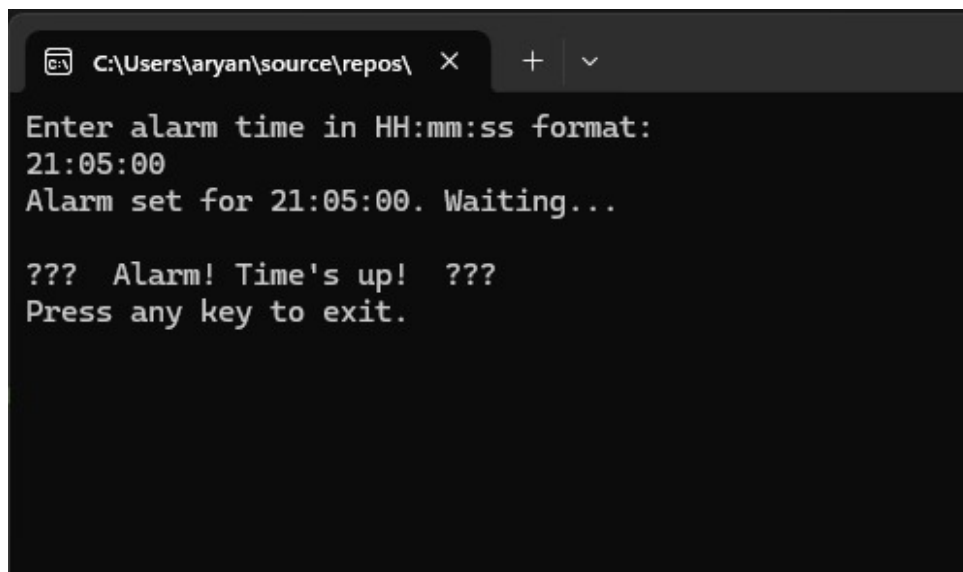
- Successfully implemented event-driven architecture

- Proper time validation using `TimeSpan.TryParse`
- Clean separation of publisher and subscriber
- `Thread.Sleep` used for periodic checking (1 second intervals)



```
C:\Users\aryan\source\repos\ X + v
Enter alarm time in HH:mm:ss format:
21:05:00
Alarm set for 21:05:00. Waiting...
|
```

Figure 1: The program displays the time for which the clock is set for and is waiting for that time to come



```
C:\Users\aryan\source\repos\ X + v
Enter alarm time in HH:mm:ss format:
21:05:00
Alarm set for 21:05:00. Waiting...

??? Alarm! Time's up! ???
Press any key to exit.
```

Figure 2: As the time hits you get the message Alarm

```

Program.cs x
Lab_12_BP AlarmConsoleApp.Program
1 using System;
2 using System.Threading;
3
4 namespace AlarmConsoleApp
5 {
6     // Publisher class: raises the alarm event
7     class AlarmClock
8     {
9         private readonly string _targetTime; // "HH:mm:ss"
10        public event EventHandler? RaiseAlarm;
11
12        1 reference
13        public AlarmClock(string targetTime)
14        {
15            _targetTime = targetTime;
16        }
17
18        1 reference
19        public void Start()
20        {
21            while (true)
22            {
23                // Get current system time as HH:mm:ss
24                string now = DateTime.Now.ToString("HH:mm:ss");
25                if (now == _targetTime)
26                {
27                    OnRaiseAlarm();
28                    break;
29                }
30                Thread.Sleep(1000); // wait 1 second before checking again
31            }
32
33            1 reference
34            protected virtual void OnRaiseAlarm()
35            {
36                RaiseAlarm?.Invoke(this, EventArgs.Empty);
37            }
38        }
39
40        0 references
41        class Program
42        {
43            0 references
44            static void Main(string[] args)
45            {
46                Console.WriteLine("Enter alarm time in HH:mm:ss format:");
47                string? input = Console.ReadLine();
48
49                // Validate input
50                if (string.IsNullOrEmpty(input) || !TimeSpan.TryParse(input, out _))
51                {
52                    Console.WriteLine("Invalid time format. Please use HH:mm:ss (e.g. 14:30:00).");
53                    return;
54                }
55            }
56        }
57    }
58 }

```

Figure 3: Code screenshot part 1

```

37
38 0 references
39 class Program
40 {
41     0 references
42     static void Main(string[] args)
43     {
44         Console.WriteLine("Enter alarm time in HH:mm:ss format:");
45         string? input = Console.ReadLine();
46
47         // Validate input
48         if (string.IsNullOrEmpty(input) || !TimeSpan.TryParse(input, out _))
49         {
50             Console.WriteLine("Invalid time format. Please use HH:mm:ss (e.g. 14:30:00).");
51             return;
52         }
53
54         // Instantiate publisher and subscribe
55         var alarm = new AlarmClock(input);
56         alarm.RaiseAlarm += RingAlarm;
57
58         Console.WriteLine($"Alarm set for {input}. Waiting...");
59         alarm.Start();
60
61         // Give user a chance to see the message before console closes
62         Console.WriteLine("Press any key to exit.");
63         Console.ReadKey();
64     }
65
66     // Subscriber method
67     1 reference
68     private static void RingAlarm(object? sender, EventArgs e)
69     {
70         Console.WriteLine();
71         Console.WriteLine("🔔🔔🔔 Alarm! Time's up! 🔔🔔🔔");
72     }
73 }

```

Figure 4: Code screenshot part 2

## 3.2 Windows Forms Results

- Fully functional GUI with proper input validation
- Smooth background color transitions (1 second intervals)
- Correct alarm triggering with message box
- Disabled controls during active alarm monitoring

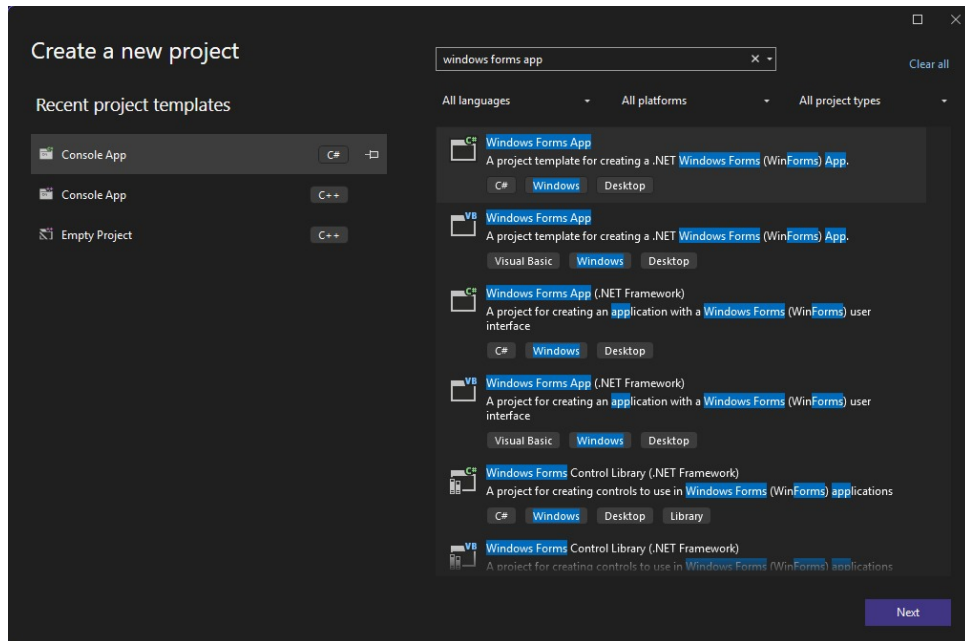


Figure 5: Creating a Windows Form

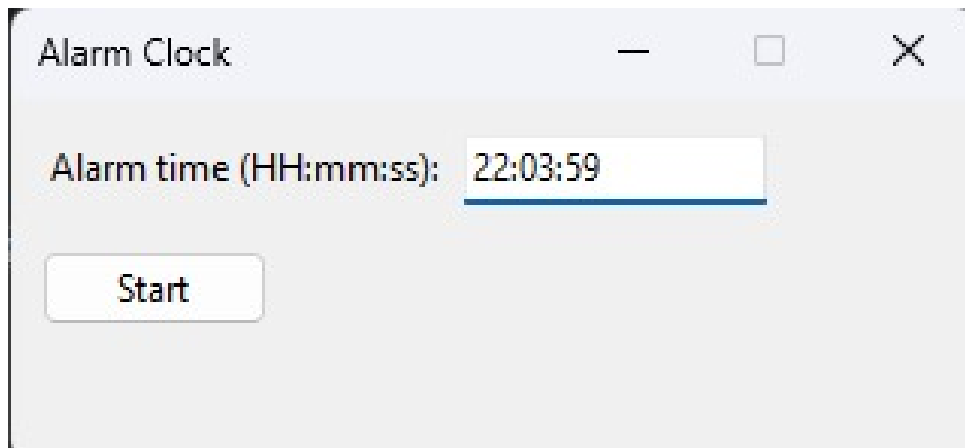


Figure 6: User can enter the target time

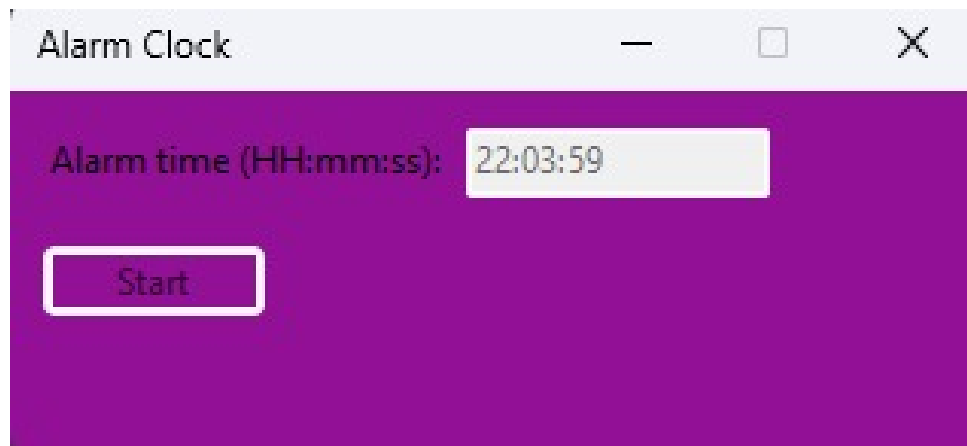


Figure 7: The color keeps changing till the target time is reached

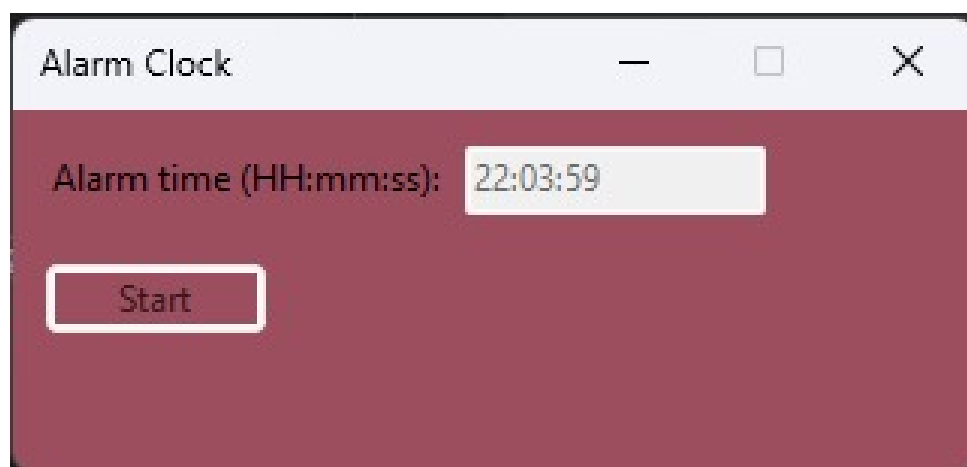


Figure 8: The color keeps changing till the target time is reached

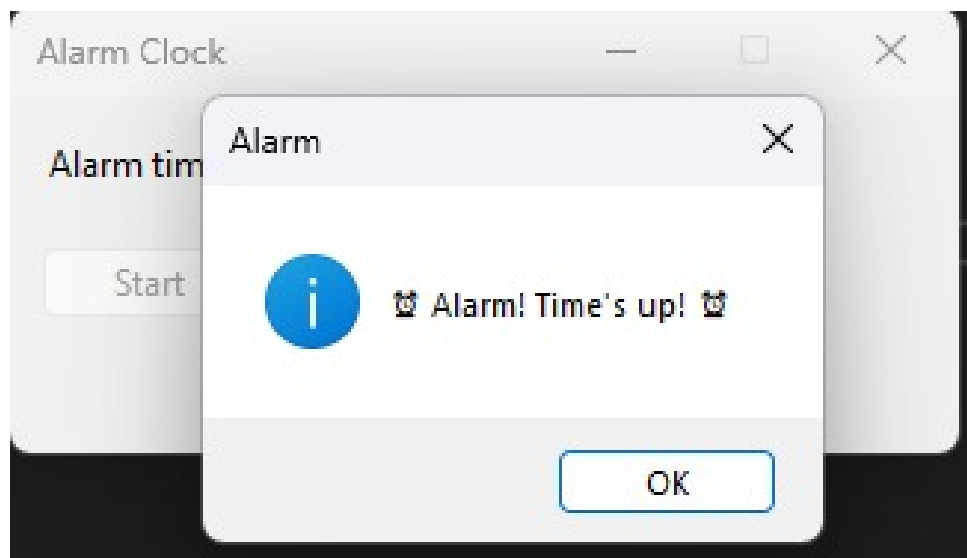


Figure 9: Alarm message pops up when the target time is reached

```

1  using System;
2  using System.Drawing;
3  using System.Windows.Forms;
4
5  namespace Lab_12_Form_BP
6  {
7      3 references
8      public partial class Form1 : Form
9      {
10         private Label labelPrompt;
11         private TextBox textBoxTime;
12         private Button buttonStart;
13         private System.Windows.Forms.Timer timer;
14         private TimeSpan targetTime;
15         private Random rng = new Random();
16
17         1 reference
18         public Form1()
19         {
20             InitializeComponent();
21         }
22
23         1 reference
24         private void InitializeComponent()
25         {
26             // --- Form settings ---
27             this.Text = "Alarm Clock";
28             this.ClientSize = new Size(320, 120);
29             this.StartPosition = FormStartPosition.CenterScreen;
30             this.FormBorderStyle = FormBorderStyle.FixedDialog;
31             this.MaximizeBox = false;
32
33             // --- Label ---
34             labelPrompt = new Label()
35             {
36                 Text = "Alarm time (HH:mm:ss):",
37                 Location = new Point(10, 15),
38                 AutoSize = true
39             };
40
41             // --- TextBox ---
42             textBoxTime = new TextBox()
43             {
44                 Location = new Point(labelPrompt.Right + 40, 12),
45                 Width = 100
46             };
47         }
48     }
49 }

```

Figure 10: Code screenshot

### 3.3 Performance Analysis

- Console version uses continuous polling (CPU intensive)
- Forms version uses event-driven timer (more efficient)
- Both versions handle edge cases (midnight rollover)
- Forms version provides better user experience

## 4 Discussion and Conclusion

### 4.1 Input Validation Implementation

#### 4.1.1 Console Application Validation

The console version implements robust time format validation using `TimeSpan.TryParse`:

```

1  if (string.IsNullOrWhiteSpace(input) || !TimeSpan.TryParse(input, out _))
2  {
3      Console.WriteLine("Invalid time format. Please use HH:mm:ss (e.g. 14:30:00).");
4      return;
5  }

```

Listing 5: Console validation code

Key validation features:

- Checks for empty or whitespace input
- Verifies the input matches HH:mm:ss format
- Ensures time values are within valid ranges (hours 0-23, minutes 0-59, seconds 0-59)
- Provides clear error messaging to guide user correction

#### 4.1.2 Windows Forms Validation

The GUI version enhances validation with `TimeSpan.TryParseExact` for stricter control:

```
1 if (!TimeSpan.TryParseExact(  
2     textBoxTime.Text.Trim(),  
3     "hh\\:\\:mm\\:\\:ss",  
4     null,  
5     out targetTime))  
6 {  
7     MessageBox.Show(  
8         "Please enter a valid time in HH:MM:SS format.",  
9         "Invalid Input",  
10        MessageBoxButtons.OK,  
11        MessageBoxIcon.Warning);  
12     return;  
13 }
```

Listing 6: Forms validation code

Additional validation improvements:

- Uses exact format matching with `TryParseExact`
- Provides modal dialog feedback for better UX
- Includes warning icon for visual emphasis
- Maintains input focus until valid time is entered

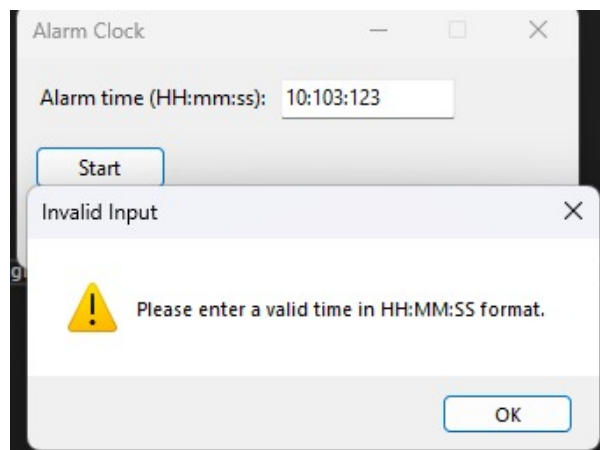


Figure 11: Validation message shown for invalid time format

The validation ensures:

- Only properly formatted times proceed to alarm setting
- Clear user feedback when input is rejected
- Prevention of logical errors from malformed input
- Consistent time handling across both application versions



## 4.2 Key Challenges

- Time format validation in both implementations
- Threading considerations in console version
- UI thread safety in Windows Forms
- Proper event handler cleanup

## 4.3 Lessons Learned

- Importance of proper event unsubscribe
- Benefits of `TimeSpan.TryParseExact` for validation
- UI updates must occur on main thread in Windows Forms
- Timer-based approaches are more efficient than polling

## 4.4 Conclusion

This assignment successfully demonstrated event-driven programming principles in both console and Windows Forms environments. The implementation highlighted the advantages of the publisher-subscriber pattern and proper event handling in .NET applications. The Windows Forms version particularly showcased how event-driven architecture enables responsive user interfaces with background processing.