

# 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

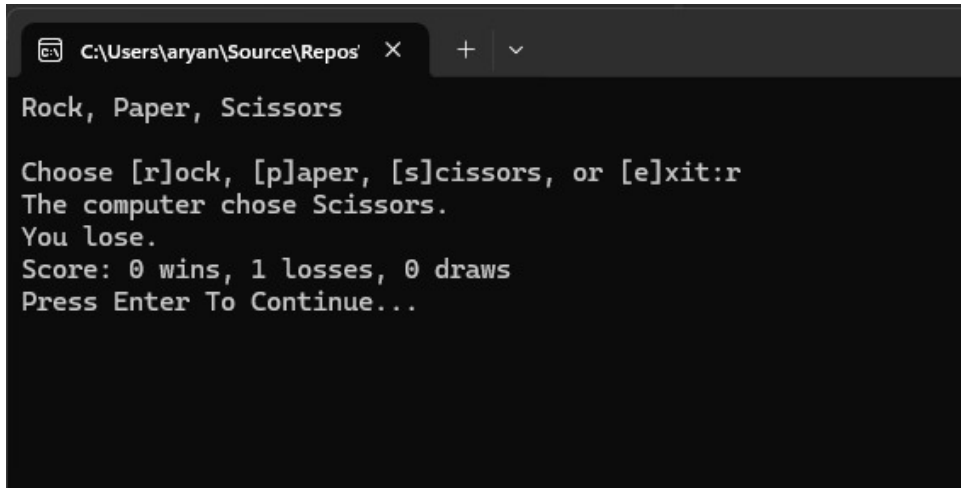
The debugging process followed a consistent methodology for each game:

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".



```
Rock, Paper, Scissors

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

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

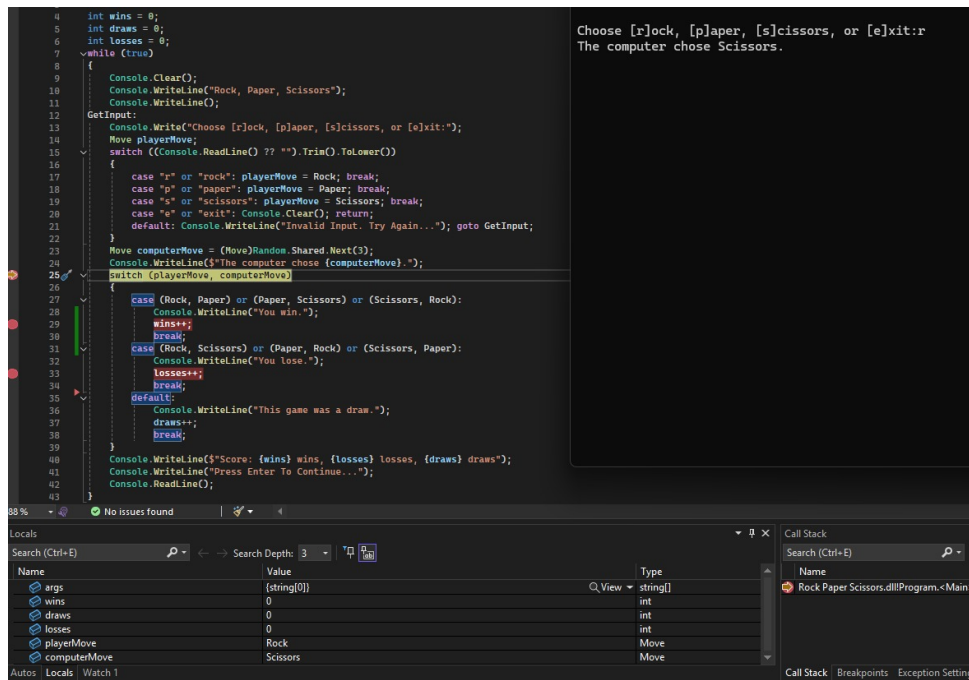


Figure 2: Illustration of using step into and step over to analyze the execution of the program

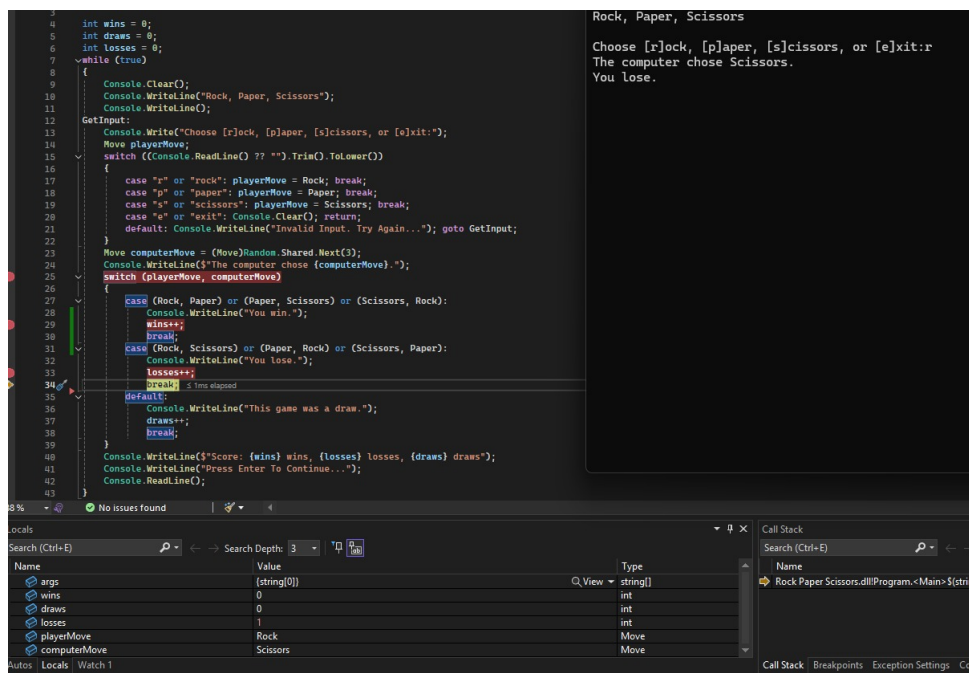


Figure 3: Observing the local variables I am able to identify the issue of the incorrect logic

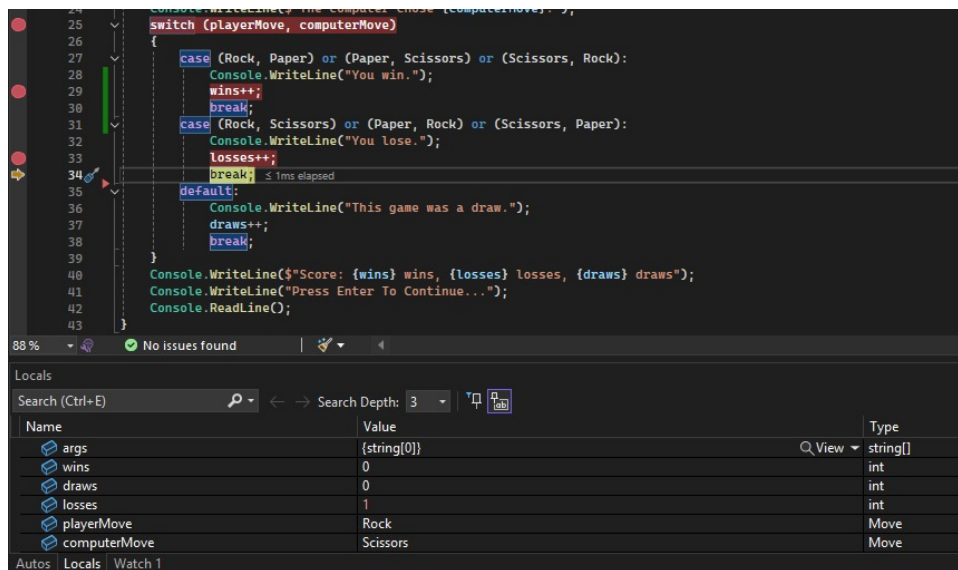


Figure 4: The issue is that the logic is reversed

### 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

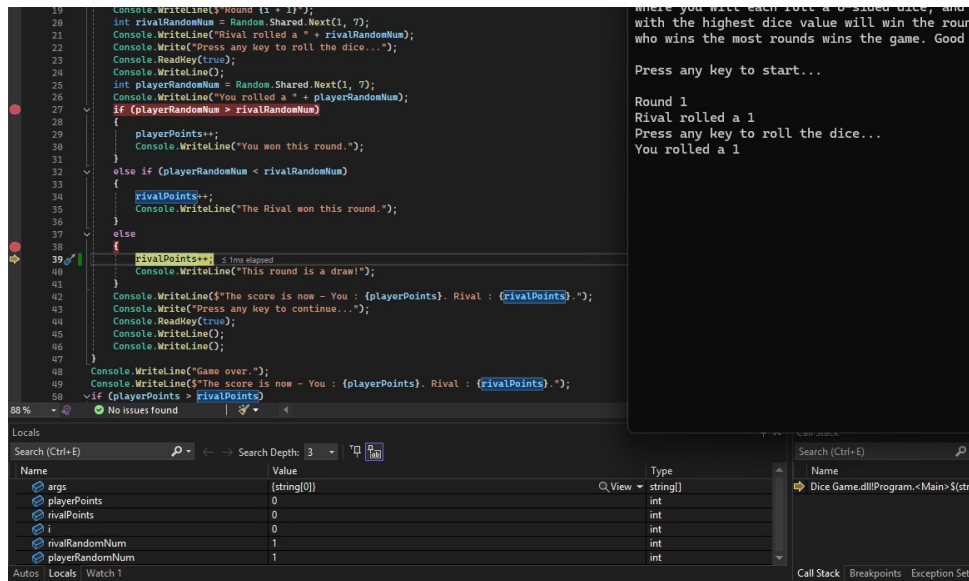


Figure 6: Further insertion of breakpoints and debugging using step into, step over and step out and observing the local variables to pin point the issue, which is the invalid increase of rivals score in the else statement



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

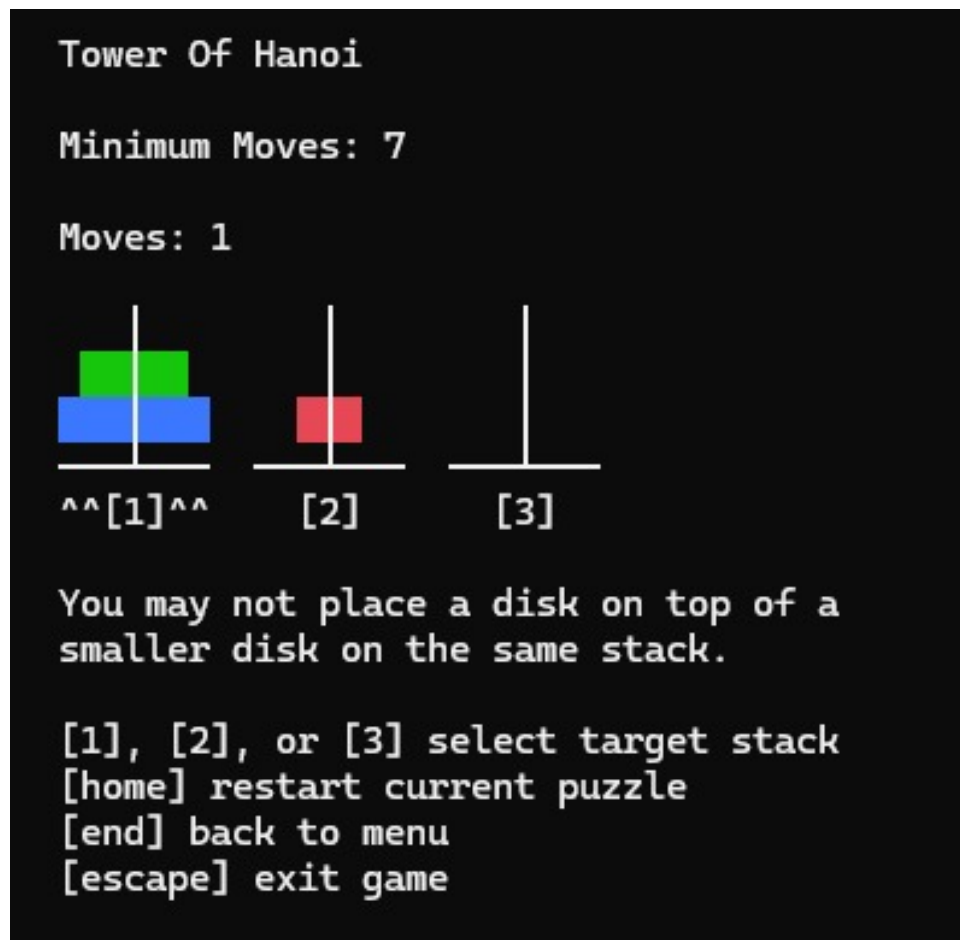


Figure 9: Pertaining to the second move 1-2, this moves the middle disk to the central tower, which is an invalid move. See the output it seems that the program correctly tells us that this is an invalid move and would have them actually negated this move



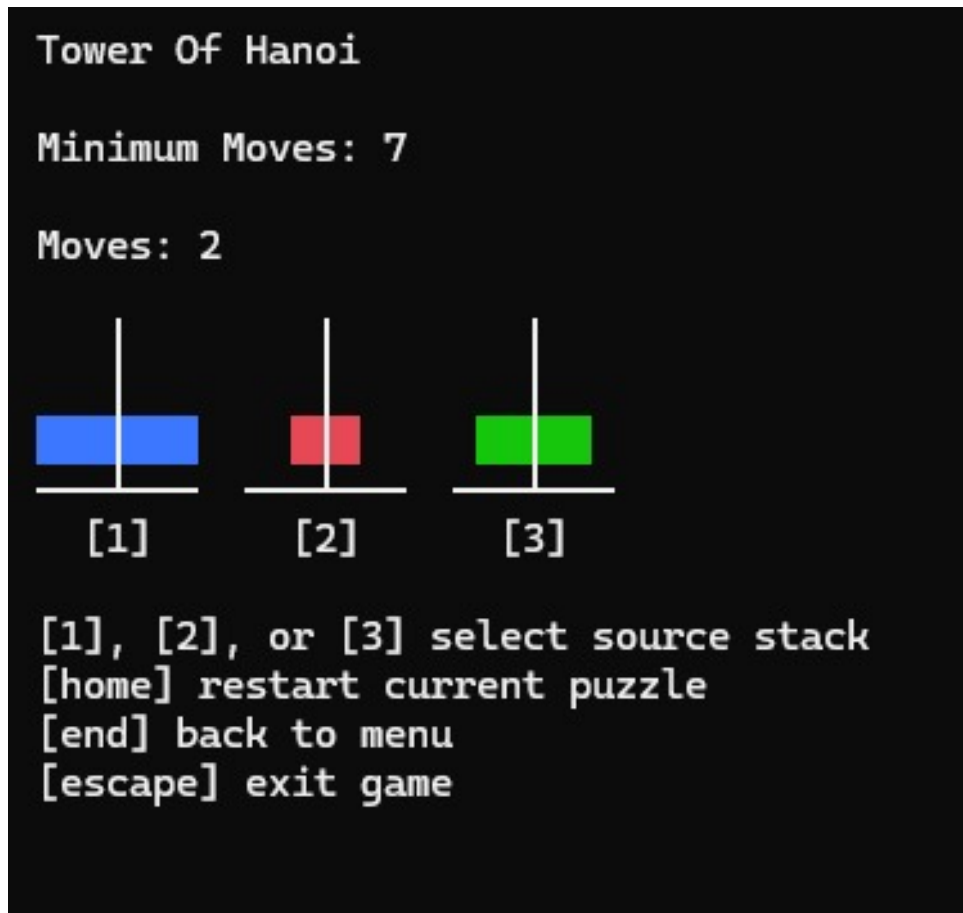


Figure 10: Pertaining to the move 2-3. This is where the bug is identified. Even when I made the move 2-3, the program "appears" to have executed the move 1-3 - a bug.

### 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

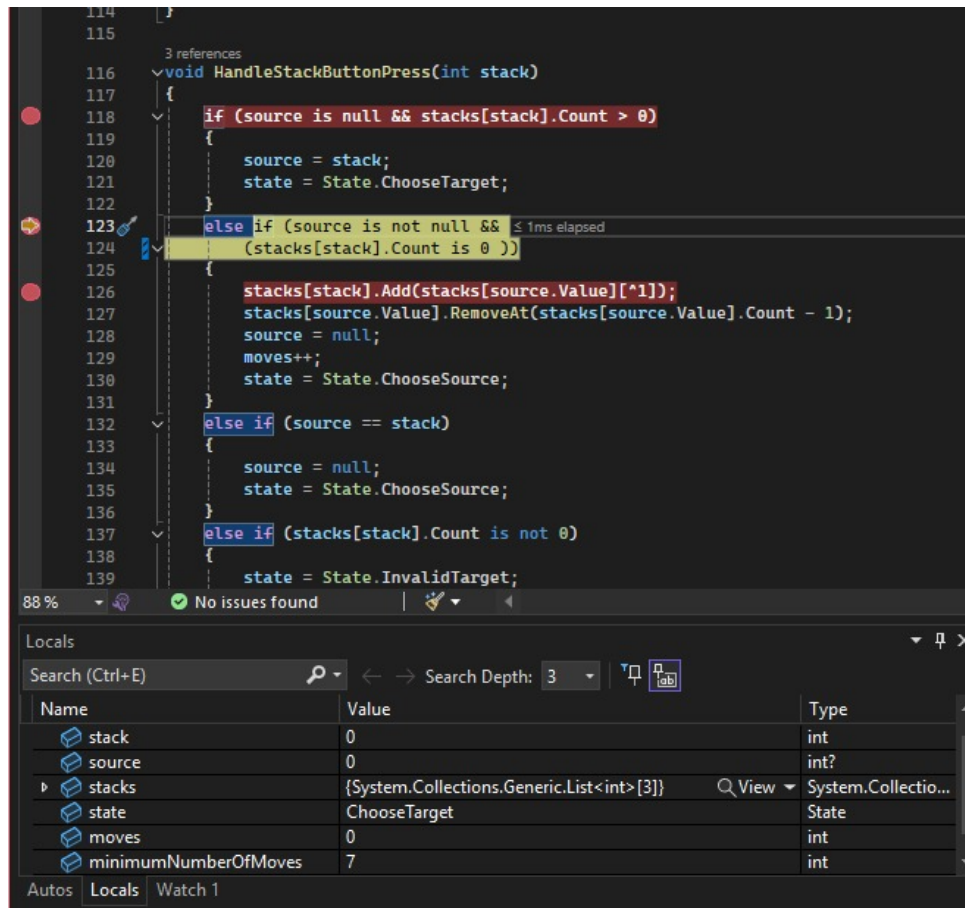


Figure 11: The issue was actually in the second step itself although it seemed to have occurred at the third move. Debugging procedure helped identify that the program lacks a check condition and as a result although it displays correctly, internally it is allowing the movement of a larger disk over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

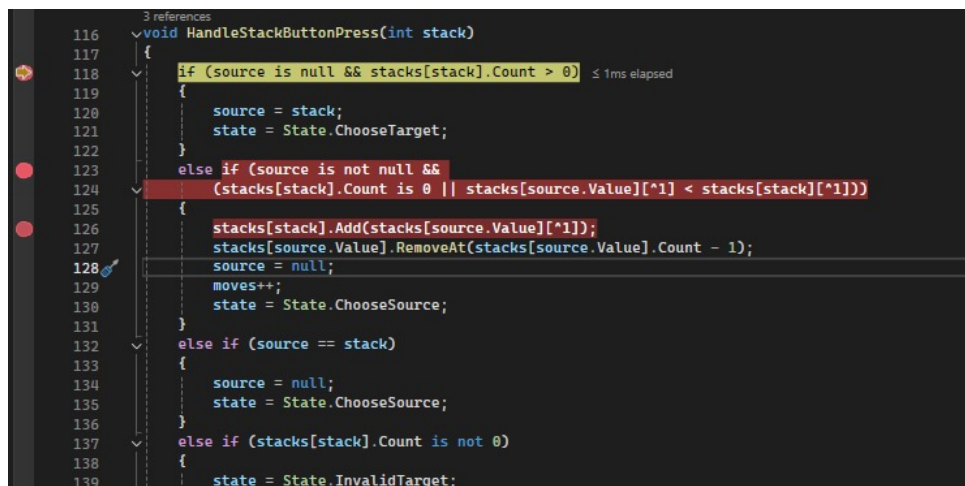


Figure 12: Inserting the additional check condition fixes the bug

### 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

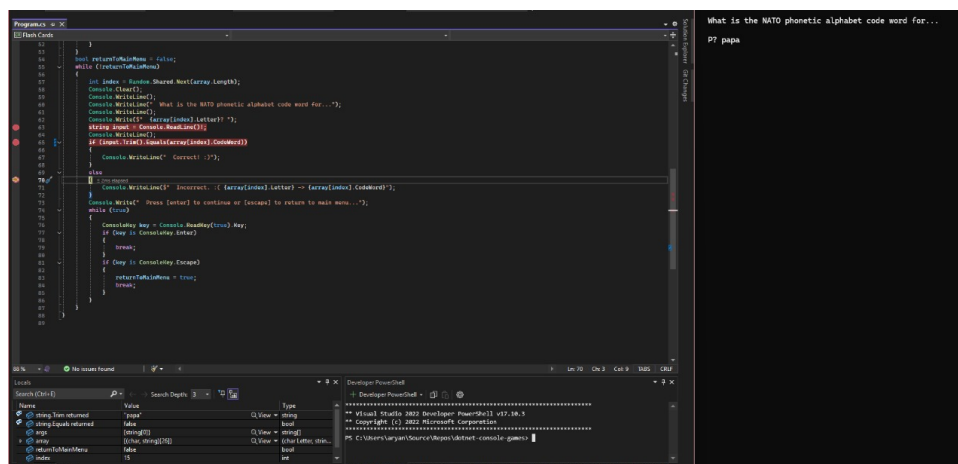


Figure 14: Debugging by inserting breakpoints and observing the local variables I identified that `returnToMainMenu` is incorrectly set to false even after correct input because the first character is lowercase it the program tries to match it to the exact string "Papa"

```

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

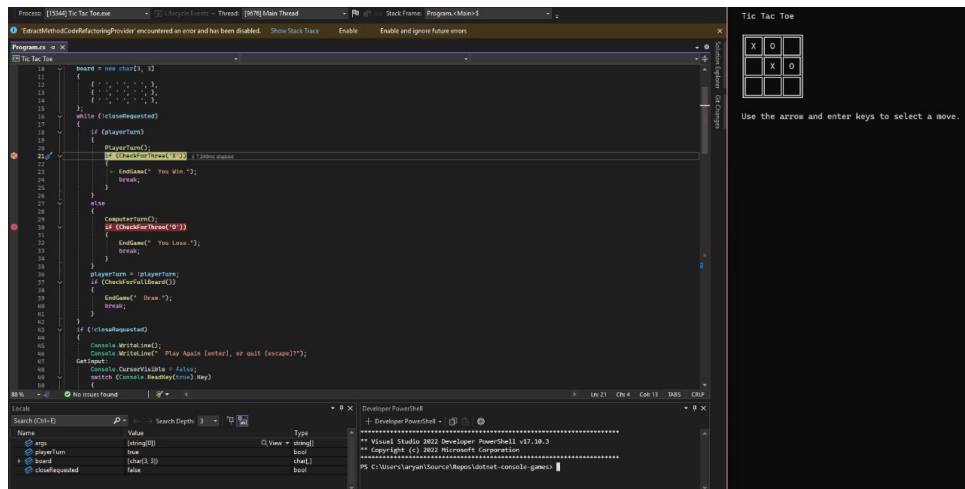


Figure 17: Debugging and inserting breakpoints helps reach the point where I can simulate the buggy situation.

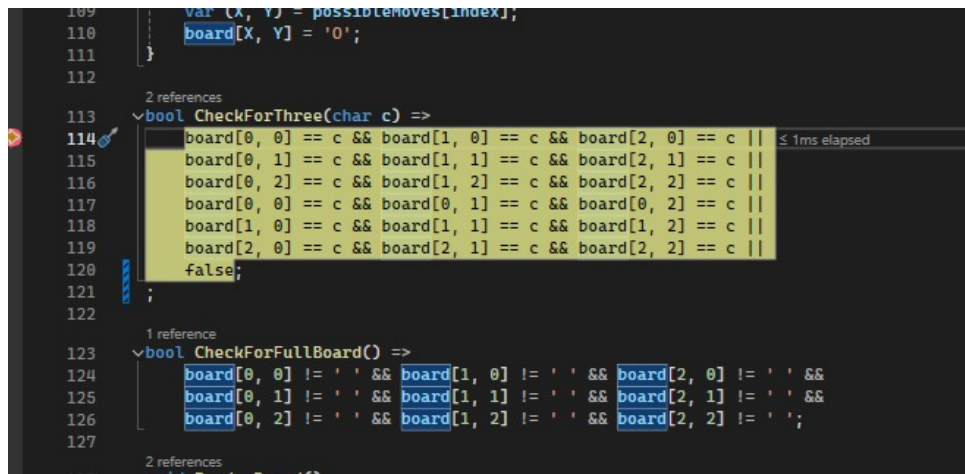


Figure 18: Analyzing the cases, I identify the cases for identifying diagonal wins are missing. This was resulting in the buggy working of the program

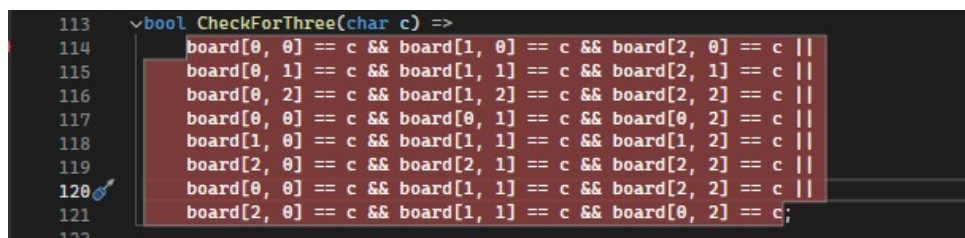


Figure 19: Added the conditions for diagonal trio which fixes the bug

### 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.