C# Branching Lab: State Tables and Decision Logic

Setup Instructions

1. Open your terminal/command prompt
2. Create and navigate to your lab directory:

   ```
   mkdir ~/1301/Lab_branching
   cd ~/1301/Lab_branching
   ```

3. Create a new console project:

   ```
   dotnet new console
   ```

4. Test that it runs:

   ```
   dotnet run
   ```

Quick Syntax Review

Basic if-statement structure in C#:

```
if (condition)
{
    // code block
}
else if (condition2)
{
    // code block
}
else
{
    // code block
}
```

Remember:

- Conditions must be boolean expressions
- Curly braces { } define code blocks
- Multiple conditions use else if
- else is optional

Section 1: Basic Branching - Medal Achievement System

Stage 1: Understanding Basic Branches

Focus: Predicting single if-statement behavior and understanding branch selection.

Paper Task:

Analyze this code:

```
int score = 75;
string medal;
if (score >= 90)
{
    medal = "Gold";
}
else if (score >= 70)
{
    medal = "Silver";
}
else
{
    medal = "Bronze";
}
```

For each line number, fill in the state of variables immediately AFTER that line executes:

State Table for score = 75:

After Line	score	medal	Notes (branch taken & why)
1			
2			

After Line	score	medal	Notes (branch taken & why)
4			
8			
12			

Coding Task:

8. Set up your project:

```
# Navigate to your lab folder
cd ~/1301/Lab_branching

# Create a new project directory
mkdir BasicMedal
cd BasicMedal

# Initialize new C# project
dotnet new console
```

9. Implement the basic medal code.
10. Add assertions to verify behavior:

```
// Verify medal was assigned
Debug.Assert(medal != null, "Medal should be assigned");

// Verify correct medal for score 75
Debug.Assert(medal == "Silver",
    "Score 75 should earn Silver medal");

// Verify not incorrectly earning higher medal
Debug.Assert(medal != "Gold",
    "Score 75 should not earn Gold medal");
```

11. Run your program:

```
dotnet run
```

Stage 2: Branch Analysis with Different Values

Focus: Understanding how different inputs affect branch selection.

Paper Task:

Using the same code, predict outcomes for:

1. score = 95
2. score = 70 (exactly)
3. score = 60

For each case:

1. Circle which branch will execute
2. Write the final value of medal
3. Explain why that branch was chosen

Coding Task:

4. Create a new project:

```
# Navigate to lab folder
cd ~/1301/Lab_branching

# Create project directory
mkdir MedalAnalysis
cd MedalAnalysis

# Initialize new C# project
dotnet new console
```

5. Test each score scenario:

```
// Test score 95
int score = 95;
string medal;
// [branching code here]
Debug.Assert(medal == "Gold",
    "Score 95 should earn Gold");
Debug.Assert(medal != "Silver" && medal != "Bronze",
    "Higher score should not earn lower medals");

// Test exact threshold (70)
score = 70;
// [branching code here]
Debug.Assert(medal == "Silver",
    "Score 70 should earn Silver");
Debug.Assert(medal != "Bronze",
    "Threshold score should earn higher medal");

// Test low score (60)
score = 60;
// [branching code here]
Debug.Assert(medal == "Bronze",
    "Score 60 should earn Bronze");
Debug.Assert(medal != "Silver" && medal != "Gold",
    "Low score should not earn higher medals");
```

6. Compare results to paper predictions

Stage 3: Creating Your Own Branching Logic

Focus: Designing branching conditions for a specific purpose.

Paper Task:

Design a grading system that:

1. Takes a player's score (0-100)
2. Assigns a rank and bonus points:
    - "Master" rank: score >= 90, bonus = 50
    - "Expert" rank: score >= 75, bonus = 30
    - "Skilled" rank: score >= 60, bonus = 15
    - "Beginner" rank: score < 60, bonus = 0

Your paper work should include:

1. Write out your complete code with all branches
2. Create a state table for score = 82
3. Note which branches you expect to execute
4. Calculate the final bonus value

Coding Task:

1. Set up your project:

```
# Navigate to lab folder
cd ~/1301/Lab_branching

# Create project directory
mkdir PlayerRank
cd PlayerRank

# Initialize new C# project
dotnet new console
```

2. Implement your ranking system:

```
int score = 82;
string rank;
int bonus;

// Your branching implementation here
Console.WriteLine($"Score: {score}, Rank: {rank}, Bonus: {bonus}");
```

3. Add comprehensive assertions:

```
// Verify rank assignment
Debug.Assert(rank != null, "Rank must be assigned");
Debug.Assert(bonus >= 0, "Bonus cannot be negative");

// Test score 82 (should be Expert)
Debug.Assert(rank == "Expert",
    "Score 82 should earn Expert rank");
Debug.Assert(bonus == 30,
    "Expert rank should give 30 bonus points");
```

```
// Verify rank and bonus are consistent
Debug.Assert((rank == "Master" && bonus == 50) ||
             (rank == "Expert" && bonus == 30) ||
             (rank == "Skilled" && bonus == 15) ||
             (rank == "Beginner" && bonus == 0),
    "Rank and bonus points must match");
```

4. Run and verify:

```
dotnet run
```

Key Concepts About Assertions:

1. Use assertions to verify:
    - Variables are properly assigned
    - Correct branch was taken
    - Incorrect branches were NOT taken
    - Values are within expected ranges
    - Related variables are consistent
2. Writing Good Assertions:
    - Test one specific thing
    - Include clear error messages
    - Consider edge cases
    - Verify both positive and negative conditions

Section 2: Sequential vs. Chained Branches - Point Calculation

Stage 1: Analyzing Sequential Branches

Focus: Understanding how multiple if-statements execute in sequence.

Paper Task:

Analyze this code:

```
int points = 0;
bool hasSolvedPuzzle = true;
bool hasFoundKey = true;

if (hasSolvedPuzzle)
{
    points = points + 5;
}
if (hasFoundKey)
{
    points = points + 3;
}
```

For each line number, fill in the state of variables immediately AFTER that line executes:

State Table (when both conditions are true):

After Line	points	hasSolvedPuzzle	hasFoundKey	Notes (what happened & why)
1				
2				
3				
7				
11				

Coding Task:

3. Set up your project:

```
# Navigate to your lab folder
cd ~/1301/Lab_branching

# Create a new project directory
mkdir SequentialPoints
cd SequentialPoints
```

```
# Initialize new C# project
dotnet new console
```

4. Implement the sequential branching code:

```
int points = 0;
bool hasSolvedPuzzle = true;
bool hasFoundKey = true;

if (hasSolvedPuzzle)
{
    points = points + 5;
    Console.WriteLine($"After puzzle check: {points} points");
}
if (hasFoundKey)
{
    points = points + 3;
    Console.WriteLine($"After key check: {points} points");
}
```

5. Add assertions to verify behavior:

```
// Test when both are true
Debug.Assert(points == 8, "Both true should give 8 points");

// Reset and test puzzle only
points = 0;
hasSolvedPuzzle = true;
hasFoundKey = false;
// [Code repeats here]
Debug.Assert(points == 5, "Puzzle only should give 5 points");

// Reset and test key only
points = 0;
hasSolvedPuzzle = false;
hasFoundKey = true;
// [Code repeats here]
Debug.Assert(points == 3, "Key only should give 3 points");
```

6. Run your program:

```
dotnet run
```

Stage 2: Analyzing Chained Branches

Paper Task:

Analyze this alternative code:

```
int points = 0;
bool hasSolvedPuzzle = true;
bool hasFoundKey = true;

if (hasSolvedPuzzle && hasFoundKey)
{
    points = points + 8;
}
else if (hasSolvedPuzzle)
{
    points = points + 5;
}
else if (hasFoundKey)
{
    points = points + 3;
}
```

For each line number, fill in the state of variables immediately AFTER that line executes:
State Table (same conditions):

After Line	points	hasSolvedPuzzle	hasFoundKey	Notes (what happened & why)
1				
2				
3				
7				

1. Create a new project:

```
# Navigate back to lab folder
cd ~/1301/Lab_branching

# Create new project directory
mkdir ChainedPoints
cd ChainedPoints

# Initialize new C# project
dotnet new console
```

2. Implement the chained branching code:

```
int points = 0;
bool hasSolvedPuzzle = true;
bool hasFoundKey = true;

if (hasSolvedPuzzle && hasFoundKey)
{
    points = points + 8;
    Console.WriteLine("Both conditions met: 8 points");
}
else if (hasSolvedPuzzle)
{
    points = points + 5;
    Console.WriteLine("Puzzle solved: 5 points");
}
else if (hasFoundKey)
{
    points = points + 3;
    Console.WriteLine("Key found: 3 points");
}
```

3. Add assertions to verify each case:

```
// Test both true
Debug.Assert(points == 8,
    "Both conditions should give 8 points");

// Reset and test puzzle only
```

```
points = 0;
hasSolvedPuzzle = true;
hasFoundKey = false;
// [Code repeats here]
Debug.Assert(points == 5,
    "Puzzle only should give 5 points");
Debug.Assert(points != 8,
    "Should not get both-condition points with only puzzle");

// Reset and test key only
points = 0;
hasSolvedPuzzle = false;
hasFoundKey = true;
// [Code repeats here]
Debug.Assert(points == 3,
    "Key only should give 3 points");
Debug.Assert(points != 8 && points != 5,
    "Should not trigger other conditions");
```

4. Run and verify:

```
dotnet run
```

Stage 3: Design Your Own Comparison

Paper Task:

Design two versions of a damage calculation system:

1. Sequential version that:
    - Adds fire damage (2 points) if target is burning
    - Adds frost damage (3 points) if target is frozen
    - Adds weakness bonus (double damage) if target is weak
2. Chained version that:
    - Handles all combinations in a single chain
    - Ensures consistent results

    - Prevents double-counting

For each version:

- Write complete code
- Create state tables
- Predict outcomes for key scenarios
- Explain why you organized the branches this way

Coding Task:

3. Create a new project for damage calculation:

   ```
   # Navigate to your lab folder
   cd ~/1301/Lab_branching

   # Create a new project directory
   mkdir DamageCalc
   cd DamageCalc

   # Initialize new C# project
   dotnet new console

   # Open Program.cs in your editor
   # Replace the existing code with your solution
   ```

4. Implement both versions
5. Add print statements showing damage calculations
6. Test with these scenarios:
    - Only burning
    - Burning and frozen
    - All conditions true
7. Compare results between versions
8. Document which approach was better and why

Reflection Questions

9. When would you use sequential if-statements?

10. When would you use chained if-statements?
11. How does the order of conditions matter in each approach?
12. What happens when multiple conditions are true in each version?

Key Concepts to Review

- Sequential if-statements can execute multiple times
- Chained if-else statements execute exactly once
- Condition order affects chained but not sequential branches
- Complex conditions might be clearer in one form vs the other

Key Points About Using Assertions:

13. Assertions should verify both:

- Expected outcomes (what should happen)
- Exclusion cases (what shouldn't happen)

14. Test each condition combination separately
15. Reset variables before each new test case
16. Include meaningful error messages in assertions
17. Use assertions to verify branch logic, not just final values

Common Assertion Patterns:

```
// Verify specific values
Debug.Assert(value == expected, "Error message");

// Verify ranges
Debug.Assert(value >= min && value <= max, "Out of range");

// Verify mutually exclusive conditions
Debug.Assert(!(conditionA && conditionB), "Should not both be true");

// Verify required conditions
Debug.Assert(value != null, "Value should be assigned");
```

Section 3: Nested Branching - Item Quality System

Stage 1: Understanding Nested Branches

Focus: Understanding how branches within branches execute.

Paper Task:

Analyze this item quality checking system:

```
int durability = 80;
int age = 5;
string quality;

if (durability > 50)
{
    if (age < 3)
    {
        quality = "Excellent";
    }
    else
    {
        quality = "Good";
    }
}
else
{
    if (age < 3)
    {
        quality = "Fair";
    }
    else
    {
        quality = "Poor";
    }
}
```

For each line number, fill in the state of variables immediately AFTER that line executes:

State Table (for durability = 80, age = 5):

After Line	durability	age	quality	Notes (which nested branch & why)
1				
2				
3				
9				

Coding Task:

18. Set up your project:

```
# Navigate to your lab folder
cd ~/1301/Lab_branching

# Create a new project directory
mkdir ItemQuality
cd ItemQuality

# Initialize new C# project
dotnet new console
```

19. Implement the nested branching code:

```
// [Previous code goes here]
Console.WriteLine($"Item Stats - Durability: {durability}, " +
    $"Age: {age}, Quality: {quality}");
```

20. Add assertions to verify behavior:

```
// Verify quality was assigned
Debug.Assert(quality != null,
    "Quality must be assigned");

// Verify correct path taken
Debug.Assert(quality == "Good",
    "Durability 80, Age 5 should be Good quality");
```

```
// Verify we didn't take wrong paths
Debug.Assert(quality != "Excellent",
    "Age 5 should not be Excellent regardless of durability");
Debug.Assert(quality != "Fair" && quality != "Poor",
    "Durability 80 should not result in Fair or Poor");
```

21. Run your program:

```
dotnet run
```

Stage 2: Analysis of Multiple Paths

Focus: Understanding how different combinations affect nested branch execution.

Paper Task:

Using the same code, create state tables for these scenarios:
22. durability = 30, age = 2
23. durability = 90, age = 1
24. durability = 20, age = 5

For each scenario, trace the execution path through the nested branches.

Coding Task:

25. Create a new project:

```
# Navigate to lab folder
cd ~/1301/Lab_branching

# Create project directory
mkdir QualityPaths
cd QualityPaths

# Initialize new C# project
dotnet new console
```

26. Test each scenario with assertions:

```
// Test low durability, low age
durability = 30;
age = 2;
// [branching code here]
Debug.Assert(quality == "Fair",
    "Low durability, low age should be Fair");

// Test high durability, low age
durability = 90;
age = 1;
// [branching code here]
Debug.Assert(quality == "Excellent",
    "High durability, low age should be Excellent");

// Test low durability, high age
durability = 20;
age = 5;
// [branching code here]
Debug.Assert(quality == "Poor",
    "Low durability, high age should be Poor");
```

Stage 3: Design Complex Nested Logic

Focus: Creating nested branches for complex decision-making.

Paper Task:

Design an enchanted item classification system that considers:
27. Magical power (0-100)
28. Stability (0-100)
29. Age (years)

Requirements:

- Items are "Stable" if stability > 75
- Items are "Powerful" if power > 80
- Items are "Ancient" if age > 100
- Classification rules:
    - Stable AND Powerful:

        - Ancient → "Legendary"
        - Not Ancient → "Superior"
    - Stable only:
        - Ancient → "Venerable"
        - Not Ancient → "Standard"
    - Unstable:
        - Powerful → "Volatile"
        - Not Powerful → "Flawed"

Create state tables for at least three test cases.

Coding Task:

30. Set up your project:

```
# Navigate to lab folder
cd ~/1301/Lab_branching

# Create project directory
mkdir EnchantedItems
cd EnchantedItems

# Initialize new C# project
dotnet new console
```

31. Implement your classification system:

```
int power = 85;
int stability = 80;
int age = 120;
string classification;

// Your nested branching implementation here
```

32. Add comprehensive assertions:

```
// Verify classification was assigned
Debug.Assert(classification != null,
    "Classification must be assigned");

// Test Legendary case (Stable, Powerful, Ancient)
power = 85; stability = 80; age = 120;
// [code here]
Debug.Assert(classification == "Legendary",
    "Stable, Powerful, Ancient should be Legendary");

// Test Superior case (Stable, Powerful, Not Ancient)
age = 50;
// [code here]
Debug.Assert(classification == "Superior",
    "Stable, Powerful, Not Ancient should be Superior");

// Test Volatile case (Unstable, Powerful)
stability = 50;
// [code here]
Debug.Assert(classification == "Volatile",
    "Unstable but Powerful should be Volatile");

// Verify mutual exclusivity
Debug.Assert(classification != "Legendary" ||
    (stability > 75 && power > 80 && age > 100),
    "Legendary must meet all criteria");
```

33. Add verification printing:

```
Console.WriteLine($"Power: {power}, Stability: {stability}, " +
    $"Age: {age}");
Console.WriteLine($"Classification: {classification}");
```

Key Concepts for Nested Branching:

34. Trace each level of nesting separately
35. Test all combinations of conditions
36. Verify mutual exclusivity of classifications
37. Check both outer and inner branch conditions
38. Consider edge cases at each decision point

Common Assertion Patterns for Nested Branches:

```
// Verify all conditions for highest classification
Debug.Assert(classification != "Best" ||
    (condition1 && condition2 && condition3),
    "Best classification requires all conditions");

// Verify exclusivity
Debug.Assert(!(classification1 && classification2),
    "Classifications should be mutually exclusive");

// Verify proper nesting
Debug.Assert(innerCondition || !outerCondition,
    "Inner condition only matters if outer is true");
```