Stage 3: Explicit Eligibility Tiers (No More Booleans!)

Branch: `demo/stage-3-pattern-matching` | **Time:** 20-30 min | **Goal:** Make tiers first-class domain concepts

F# Code (Library.fs)

```
namespace CustomerLiveDomain

type RegisteredCustomer = { Id: string }  // Removed IsEligible!
type UnregisteredCustomer = { Id: string }

type Customer =
    | Standard of RegisteredCustomer    // Eligible tier EXPLICIT
    | Registered of RegisteredCustomer  // Registered but not eligible
    | Guest of UnregisteredCustomer
```

C# Code (Program.cs)

```
using static CustomerLiveDomain.Customer;

var john = NewStandard(new RegisteredCustomer { Id = "John" });
var mary = NewStandard(new RegisteredCustomer { Id = "Mary" });
var richard = NewRegistered(new RegisteredCustomer { Id = "Richard" });
var sarah = NewGuest(new UnregisteredCustomer { Id = "Sarah" });

static decimal CalculateTotal(Customer customer, decimal spend)
{
    // Simpler logic - just check the tier!
    if (customer.IsStandard && spend >= 100)
        return spend * 0.9m;
    return spend;
}
```

Live Refactoring Steps

1. **Remove boolean:** Delete `IsEligible: bool` from `RegisteredCustomer`
2. **Add Standard tier:** Add `| Standard of RegisteredCustomer` to DU (above Registered)
3. **Update C#:** Change `NewRegistered()` → `NewStandard()` for eligible customers
4. **Simplify logic:** Replace boolean check with `if (customer.IsStandard ...)`

VIP Extension (Live Demo)

```
// Add VIP tier (15% discount)
type Customer =
```

```
    | VIP of RegisteredCustomer         // NEW!
    | Standard of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
```

```
var alice = NewVIP(new RegisteredCustomer { Id = "Alice" });

static decimal CalculateTotal(Customer customer, decimal spend)
{
    if (customer.IsVIP && spend >= 100)       // NEW!
        return spend * 0.85m;                 // 15% discount
    if (customer.IsStandard && spend >= 100)
        return spend * 0.9m;                  // 10% discount
    return spend;
}
```

Key Improvements Over Stage 2

- ☑ **Domain language explicit** - Standard, Registered, Guest, VIP
- ☑ **No boolean flags** - Can't forget to check anything
- ☑ **Simpler logic** - `IsStandard` is clearer than checking boolean
- ☑ **Type names match business** - "Standard tier" is a real concept
- ☑ **Compiler enforces** - Adding VIP forces updating all pattern matches

Key Talking Points

- "The more we break down F# types, the cleaner the C# gets!"
- "Standard IS a tier - not a registered customer WITH a flag"
- "Domain language lives in the type system"
- "No boolean checks to forget - compiler guides us"

Pattern Matching Alternative (Show if time)

```
var discount = customer switch
{
    VIP _ when spend >= 100 => spend * 0.15m,
    Standard _ when spend >= 100 => spend * 0.1m,
    _ => 0m
};
return spend - discount;
```

Recovery

```
git reset --hard; git checkout demo/stage-3-pattern-matching
```

Step-by-Step Code Evolution

Starting Point: Stage 2 Code (Before Refactoring)

Library.fs (Stage 2)

```
namespace CustomerLiveDomain

type RegisteredCustomer = {
    Id: string
    IsEligible: bool
}

type UnregisteredCustomer = {
    Id: string
}

type Customer =
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
```

Program.cs (Stage 2)

```
using CustomerLiveDomain;
using static CustomerLiveDomain.Customer;

class Program
{
    static void Main(string[] args)
    {
        var john = NewRegistered(new RegisteredCustomer { Id = "John", IsEligible
= true });
        var mary = NewRegistered(new RegisteredCustomer { Id = "Mary", IsEligible
= true });
        var richard = NewRegistered(new RegisteredCustomer { Id = "Richard",
IsEligible = false });
        var sarah = NewGuest(new UnregisteredCustomer { Id = "Sarah" });

        Console.WriteLine($"John (£100): £{CalculateTotal(john, 100m)}");
        Console.WriteLine($"Mary (£99): £{CalculateTotal(mary, 99m)}");
        Console.WriteLine($"Richard (£100): £{CalculateTotal(richard, 100m)}");
        Console.WriteLine($"Sarah (£100): £{CalculateTotal(sarah, 100m)}");
    }

    static decimal CalculateTotal(Customer customer, decimal spend)
    {
        // Still checking boolean flag
        if (customer.IsRegistered)
        {
```

```
            var registered = (Customer.Registered)customer;
            if (registered.Item.IsEligible && spend >= 100)
                return spend * 0.9m;
        }
        return spend;
    }
}
```

Problem: Still using boolean `IsEligible` flag - can forget to check it!

Step 1: Remove Boolean from F# Domain Model

Library.fs (Stage 3 - Step 1)

```
namespace CustomerLiveDomain

type RegisteredCustomer = {
    Id: string
    // ❌ Removed: IsEligible: bool
}

type UnregisteredCustomer = {
    Id: string
}

type Customer =
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
```

Talking Point: "We removed the boolean, but we haven't added the tiers yet. Let's see what breaks..."

Step 2: Add Explicit Tiers to DU

Library.fs (Stage 3 - Step 2)

```
namespace CustomerLiveDomain

type RegisteredCustomer = { Id: string }

type UnregisteredCustomer = { Id: string }

type Customer =
    | Standard of RegisteredCustomer    // ☑ NEW: Eligible tier explicit!
    | Registered of RegisteredCustomer  // Registered but not eligible
    | Guest of UnregisteredCustomer
```

Talking Point: "Now eligibility is a TIER, not a property. 'Standard' customers ARE eligible - it's in the type name!"

Step 3: Update C# Factory Method Calls

Program.cs (Stage 3 - Step 3: Main Method)

```
static void Main(string[] args)
{
    // Clean factory methods - domain language is explicit!
    var john = NewStandard(new RegisteredCustomer { Id = "John" });      // ☑
Standard = eligible
    var mary = NewStandard(new RegisteredCustomer { Id = "Mary" });      // ☑
Standard = eligible
    var richard = NewRegistered(new RegisteredCustomer { Id = "Richard" }); //
Registered, not eligible
    var sarah = NewGuest(new UnregisteredCustomer { Id = "Sarah" });     // Guest

    Console.WriteLine($"John (£100): £{CalculateTotal(john, 100m)}");      // £90
    Console.WriteLine($"Mary (£99): £{CalculateTotal(mary, 99m)}");        // £99
    Console.WriteLine($"Richard (£100): £{CalculateTotal(richard, 100m)}"); //
£100
    Console.WriteLine($"Sarah (£100): £{CalculateTotal(sarah, 100m)}");    // £100
}
```

Talking Point: "Look how readable this is! `NewStandard` - no boolean needed!"

Step 4: Simplify Business Logic

Program.cs (Stage 3 - Step 4: CalculateTotal)

```
static decimal CalculateTotal(Customer customer, decimal spend)
{
    // ☑ Simpler logic - no boolean checks needed!
    if (customer.IsStandard && spend >= 100)
        return spend * 0.9m;
    return spend;
}
```

Talking Point: "The business logic got SIMPLER! Just check `IsStandard`. The F# type system did the heavy lifting."

Final Code: Stage 3 Complete

Library.fs (Final)

```
namespace CustomerLiveDomain

type RegisteredCustomer = { Id: string }

type UnregisteredCustomer = { Id: string }

type Customer =
    | Standard of RegisteredCustomer    // Eligible tier explicit
    | Registered of RegisteredCustomer  // Registered but not eligible
    | Guest of UnregisteredCustomer
```

Program.cs (Final)

```
using CustomerLiveDomain;
using static CustomerLiveDomain.Customer;

class Program
{
    static void Main(string[] args)
    {
        // Clean factory methods - domain language explicit
        var john = NewStandard(new RegisteredCustomer { Id = "John" });
        var mary = NewStandard(new RegisteredCustomer { Id = "Mary" });
        var richard = NewRegistered(new RegisteredCustomer { Id = "Richard" });
        var sarah = NewGuest(new UnregisteredCustomer { Id = "Sarah" });

        Console.WriteLine($"John (£100): £{CalculateTotal(john, 100m)}");      //
£90
        Console.WriteLine($"Mary (£99): £{CalculateTotal(mary, 99m)}");        //
£99
        Console.WriteLine($"Richard (£100): £{CalculateTotal(richard, 100m)}"); //
£100
        Console.WriteLine($"Sarah (£100): £{CalculateTotal(sarah, 100m)}");    //
£100
    }

    static decimal CalculateTotal(Customer customer, decimal spend)
    {
        // Simpler logic - no boolean checks!
        if (customer.IsStandard && spend >= 100)
            return spend * 0.9m;
        return spend;
    }
}
```

Live Demo Extension: Add VIP Tier

Time: 5 minutes
Goal: Show compiler enforcement when domain changes

Step 1: Add VIP to F# DU

Library.fs (VIP Extension)

```
namespace CustomerLiveDomain

type RegisteredCustomer = { Id: string }

type UnregisteredCustomer = { Id: string }

type Customer =
    | VIP of RegisteredCustomer         // ☑ NEW!
    | Standard of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
```

Talking Point: "Let's add a VIP tier with 15% discount. Watch what happens in C#..."

Step 2: C# Shows Warning!

Build the solution. C# compiler may show warning about non-exhaustive pattern matching (depending on C# version).

Step 3: Fix C# Business Logic

Program.cs (VIP Extension)

```
static void Main(string[] args)
{
    // Add VIP customer
    var alice = NewVIP(new RegisteredCustomer { Id = "Alice" });  // ☑ NEW!
    var john = NewStandard(new RegisteredCustomer { Id = "John" });
    var mary = NewStandard(new RegisteredCustomer { Id = "Mary" });
    var richard = NewRegistered(new RegisteredCustomer { Id = "Richard" });
    var sarah = NewGuest(new UnregisteredCustomer { Id = "Sarah" });

    Console.WriteLine($"Alice (£100): £{CalculateTotal(alice, 100m)}");    // £85
(15% discount)
    Console.WriteLine($"John (£100): £{CalculateTotal(john, 100m)}");      // £90
(10% discount)
    Console.WriteLine($"Mary (£99): £{CalculateTotal(mary, 99m)}");        // £99
    Console.WriteLine($"Richard (£100): £{CalculateTotal(richard, 100m)}"); //
£100
    Console.WriteLine($"Sarah (£100): £{CalculateTotal(sarah, 100m)}");    // £100
}
```

```
static decimal CalculateTotal(Customer customer, decimal spend)
{
    // Add VIP case
    if (customer.IsVIP && spend >= 100)           // ☑ NEW!
        return spend * 0.85m;                      // 15% discount
    if (customer.IsStandard && spend >= 100)
        return spend * 0.9m;                       // 10% discount
    return spend;
}
```

Talking Point: "The type system forces us to handle the new case. Can't forget it!"

Alternative: Pattern Matching Version (Show if Time Permits)

```
static decimal CalculateTotal(Customer customer, decimal spend)
{
    var discount = customer switch
    {
        VIP _ when spend >= 100 => spend * 0.15m,       // 15% discount
        Standard _ when spend >= 100 => spend * 0.1m,   // 10% discount
        _ => 0m
    };
    return spend - discount;
}
```

Talking Point: "Modern C# has pattern matching too! Both approaches work beautifully with F# types."

Key Messages for Stage 3

- ☑ **Domain language is explicit** - Standard, Registered, Guest, VIP
- ☑ **No boolean flags to check or forget** - Types encode the rules
- ☑ **Simpler C# logic** - The more we refine F#, the cleaner C# becomes
- ☑ **Compiler-enforced correctness** - Can't create illegal states
- ☑ **Reads like the business requirement** - Code matches domain language

---