CS202 Software Tools and Techniques for CSE
Lab Report 12: Advanced Event Handling and Custom EventArgs in C# Windows Forms

Md Sibtain Raza (22110148) | Github Link

November 23, 2025

1 Objective

The objective of this lab is to deepen understanding of advanced event-driven mechanisms in C# Windows Forms applications, especially event chaining, conditional firing, and context sharing through custom `EventArgs` types. Students design a modular GUI application named `OrderPipeline` that models an order-processing workflow using chained custom events, filtered event invocation, and multicast subscriptions.

2 Lab Requirements

- Operating System: Windows.
- Software: Visual Studio 2022 (Community Edition) with .NET SDK.
- Programming Language and platform: C# (latest stable version), Windows Forms.

3 Theory

Events and delegates in C# implement the publisher–subscriber pattern, where a delegate specifies the signature of subscribers and an event exposes a safe interface for adding or removing handlers. Custom `EventArgs` classes allow the publisher to send contextual information such as customer name, product, quantity, or shipping mode from one part of the UI to another without tight coupling.

Multicast delegates enable a single event invocation to call multiple methods in sequence, which is used to separate validation, status updates, and notifications for the same logical action in this lab. Advanced scenarios include event chaining (one event firing another) and dynamic subscription, where handlers are added or removed at runtime to implement filtering based on user choices such as express shipping.

4 Experiment 1: Multi-Stage Event Chaining (OrderPipeline Task 1)

4.1 Problem Statement

Develop a Windows Forms application named `OrderPipeline` that models a small order-processing workflow using chained custom events. The form contains a `TextBox` (`txtCustomerName`), `ComboBox` (`cmbProduct` with "Laptop", "Mouse", "Keyboard"), `NumericUpDown` (`numQuantity`), a button `btnProcessOrder` and a label `lblStatus`.

Clicking `Process Order` must raise a custom event `OrderCreated`, which has two subscribers: `ValidateOrder()` and `DisplayOrderInfo()`. If validation fails, `ValidateOrder()` triggers another

event `OrderRejected`; if it succeeds, it chains a third event `OrderConfirmed` whose subscribers update the status label appropriately.

4.2 Design

The design introduces an `OrderEventArgs` class carrying `CustomerName`, `Product`, and `Quantity` so that all subscribers receive consistent contextual data. `OrderCreated` is raised whenever the user clicks `btnProcessOrder`, and its subscribers are responsible for validation, status changes, and displaying order details without embedding business logic directly in the button handler.

`ValidateOrder()` checks whether `Quantity > 0`; if invalid, it fires `OrderRejected` so that `ShowRejection()` can set `lblStatus` to "Order Invalid – Please retry". If the order is valid, `ValidateOrder()` chains into `OrderConfirmed`, and `ShowConfirmation()` finally writes "Order Processed Successfully for <Customer>" to the label.

4.3 Implementation: OrderEventArgs

Listing 1: OrderEventArgs.cs – custom event data for orders

```
using System;

namespace OrderPipeline
{
    public class OrderEventArgs : EventArgs
    {
        public string CustomerName { get; }
        public string Product { get; }
        public int Quantity { get; }

        public OrderEventArgs(string customerName, string product, int
            quantity)
        {
            CustomerName = customerName;
            Product = product;
            Quantity = quantity;
        }
    }
}
```

4.4 Implementation: Form1.cs (Task 1 core logic)

Listing 2: Form1.cs – Task 1 events and chaining

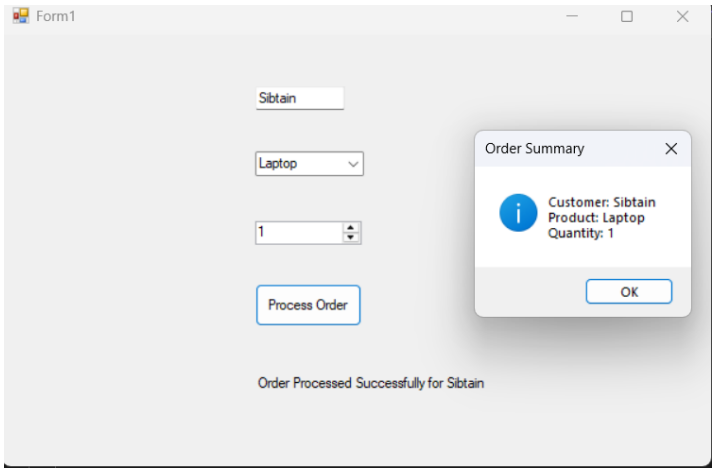
```
using System;
using System.Windows.Forms;

namespace OrderPipeline
{
    public partial class Form1 : Form
    {
        // Task 1 events
        public event EventHandler<OrderEventArgs> OrderCreated;
        public event EventHandler<OrderEventArgs> OrderRejected;
        public event EventHandler<OrderEventArgs> OrderConfirmed;

        // Track if last order was confirmed
        private bool lastOrderConfirmed = false;
```

```
public Form1()
{
    InitializeComponent();

    if (cmbProduct.Items.Count == 0)
    {
        cmbProduct.Items.AddRange(
            new object[] { "Laptop", "Mouse", "Keyboard" });
        cmbProduct.SelectedIndex = 0;
    }

    // Subscribe Task 1 handlers
    OrderCreated += ValidateOrder;
    OrderCreated += DisplayOrderInfo;

    OrderRejected += ShowRejection;
    OrderConfirmed += ShowConfirmation;

    btnProcessOrder.Click += btnProcessOrder_Click;
}
// Publisher for Process Order
private void btnProcessOrder_Click(object sender, EventArgs e)
{
    string customer = txtCustomerName.Text.Trim();
    string product = cmbProduct.SelectedItem?.ToString() ?? string.
        Empty;
    int quantity = (int)numQuantity.Value;

    var args = new OrderEventArgs(customer, product, quantity);

    // Reset confirmation flag; will be true only after
        ShowConfirmation
    lastOrderConfirmed = false;

    OrderCreated?.Invoke(this, args);
}

// Subscriber 1: validation and event chaining
private void ValidateOrder(object sender, OrderEventArgs e)
{
    if (e.Quantity > 0)
    {
        lblStatus.Text = "Validated";
        OrderConfirmed?.Invoke(this, e);
    }
    else
    {
        OrderRejected?.Invoke(this, e);
    }
}
// Subscriber 2: MessageBox summary
private void DisplayOrderInfo(object sender, OrderEventArgs e)
{
    string msg = $"Customer: {e.CustomerName}\n" +
                 $"Product: {e.Product}\n" +
                 $"Quantity: {e.Quantity}";
    MessageBox.Show(msg, "Order Summary",
```

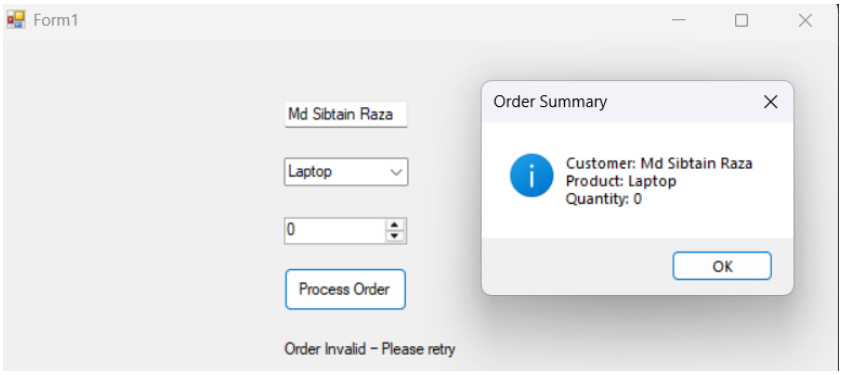
```
                MessageBoxButtons.OK, MessageBoxIcon.Information);
        }

        // Handles OrderRejected
        private void ShowRejection(object sender, OrderEventArgs e)
        {
            lastOrderConfirmed = false;
            lblStatus.Text = "Order Invalid - Please retry";
        }
        // Handles OrderConfirmed
        private void ShowConfirmation(object sender, OrderEventArgs e)
        {
            lastOrderConfirmed = true;
            lblStatus.Text =
                $"Order Processed Successfully for {e.CustomerName}";
        }
    }
}
```

4.5 Execution and Observations

On startup, the user can enter a customer name, pick a product, and set the quantity using the numeric control. When the `Process Order` button is clicked, `OrderCreated` is raised, which triggers validation, shows a summary MessageBox, and either rejects or confirms the order based on the quantity.



For invalid orders (quantity ≤ 0), `ShowRejection()` sets the status label to "Order Invalid – Please retry".



5 Experiment 2: Event Filtering and Dynamic Subscription (Task 2)

5.1 Problem Statement

Extend `OrderPipeline` to introduce an additional shipping stage that illustrates event filtering, dynamic handler subscription, and conditional event firing. The form adds a `CheckBox chkExpress` for express delivery and a second button `btnShipOrder` that should only work when the last order has been confirmed.

A new event `OrderShipped` is defined with a custom `ShipEventArgs` containing `Product` and `Express`. `OrderShipped` has two subscribers: `ShowDispatch()` (always subscribed) and `NotifyCourier()`, which should be dynamically added or removed depending on whether express shipping is selected.

5.2 Implementation: ShipEventArgs

Listing 3: ShipEventArgs.cs – shipping context

```
using System;

namespace OrderPipeline
{
    public class ShipEventArgs : EventArgs
    {
        public string Product { get; }
        public bool Express { get; }

        public ShipEventArgs(string product, bool express)
        {
            Product = product;
            Express = express;
        }
    }
}
```

5.3 Implementation: Shipping Logic in Form1

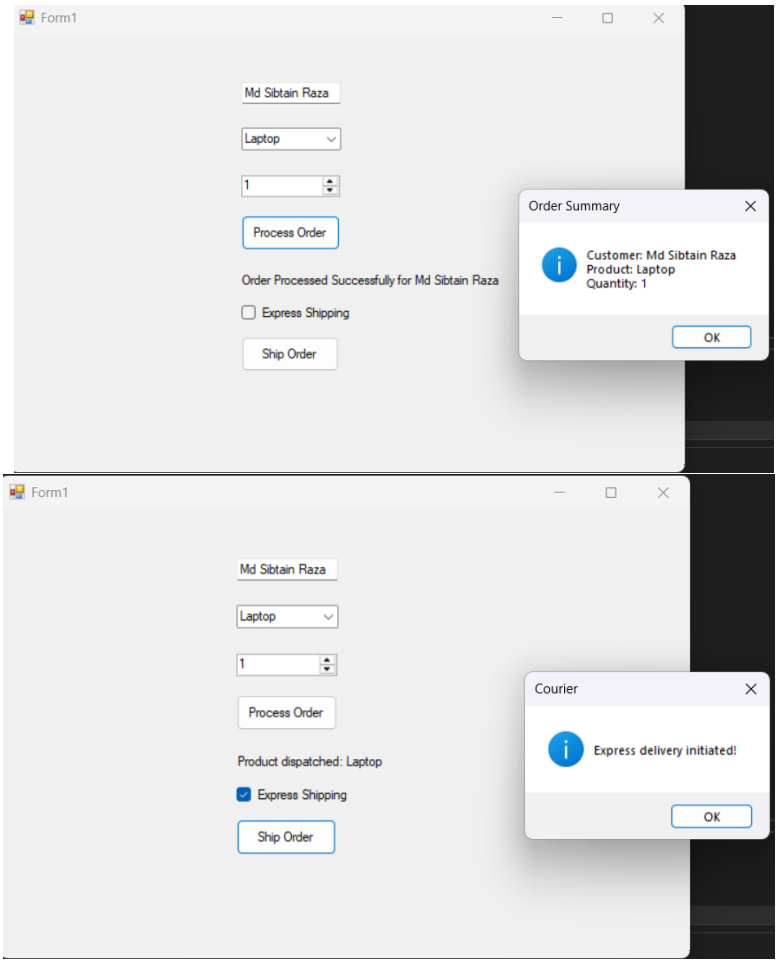
Listing 4: Form1.cs – Task 2 shipping, filtering and dynamic subscription

```
namespace OrderPipeline
{
    public partial class Form1 : Form
    {
        // ... Task 1 members omitted for brevity ...

        // Task 2 event
        public event EventHandler<ShipEventArgs> OrderShipped;

        public Form1() {
            // ... Task 1 members omitted for brevity ...

            // Subscribe Task 2 base handler
            OrderShipped += ShowDispatch;

            // Publisher
            btnProcessOrder.Click += btnProcessOrder_Click;
            btnShipOrder.Click += btnShipOrder_Click;
        }
        // ... Task 1 members omitted for brevity ...
```

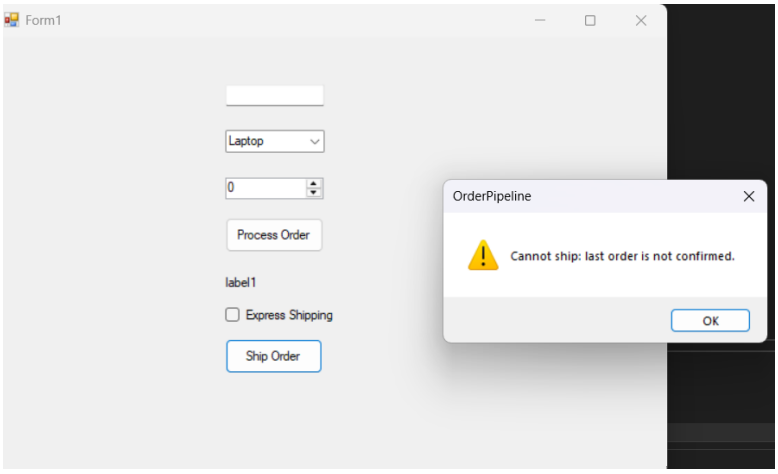
```
        // Publisher for OrderShipped
        private void btnShipOrder_Click(object sender, EventArgs e)
        {
            // Event fires only if previous order was confirmed
            if (!lastOrderConfirmed)
            {
                MessageBox.Show("Cannot ship: last order is not confirmed.",
                    "OrderPipeline",
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Warning);
                return;
            }

            string product = cmbProduct.SelectedItem?.ToString() ?? "Unknown
                ";
            bool express = chkExpress.Checked;

            // Event filtering via dynamic subscription
            OrderShipped -= NotifyCourier;   // avoid duplicate handlers
            if (express)
            {
                OrderShipped += NotifyCourier;
            }

            var shipArgs = new ShipEventArgs(product, express);
            OrderShipped?.Invoke(this, shipArgs);
        }

        // Subscriber 1: always active
        private void ShowDispatch(object sender, ShipEventArgs e)
        {
            lblStatus.Text = $"Product dispatched: {e.Product}";
        }

        // Subscriber 2: added/removed depending on Express flag
        private void NotifyCourier(object sender, ShipEventArgs e)
        {
            if (e.Express)
            {
                MessageBox.Show("Express delivery initiated!",
                    "Courier",
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Information);
            }
        }
    }
}
```

5.4 Execution and Observations

After a successful order confirmation, clicking `Ship Order` with the express checkbox unchecked raises `OrderShipped` and executes `ShowDispatch()`, updating the label to "Product dispatched: [Product]" without any courier notification. When express shipping is selected, the handler `NotifyCourier()` is dynamically added before the event fires, so in addition to the dispatch label, a MessageBox indicates that express delivery has been initiated.



If the user attempts to ship before confirming an order, the button handler detects that `lastOrderConfirmed` is `false` and shows a warning MessageBox instead of raising the event.



6 Output Reasoning

This section records answers and reasoning for the problems provided in the lab sheet.

6.1 Level 0

Problem 1:

```
public delegate void AuthCallback(bool validUser);
public static AuthCallback loginCallback = Login;
public static void Login()
{
    Console.WriteLine("Valid user!");
}


public static void Main(string[] args)
{
    loginCallback(true);
}
```

Solution. This code does not compile because the delegate type `AuthCallback` expects a method with signature `void(bool)`, while `Login` has signature `void()` with no parameters, so the assignment `loginCallback = Login` fails overload resolution.

Problem 2

```
using System;

delegate void Notify(string msg);

class Program
{
    static void Main()
    {
        Notify handler = null;

        handler += (m) => Console.WriteLine("A: " + m);
        handler += (m) => Console.WriteLine("B: " + m.ToUpper());

        handler("hello");

        handler -= (m) => Console.WriteLine("A: " + m);
        handler("world");
    }
}
```

Solution. The first call `handler("hello")` invokes both lambdas, printing "A: hello" and "B: HELLO" on separate lines, in that order. The `-=` operation uses a new lambda instance that does not match the existing delegate reference, so no subscriber is actually removed and the second call prints "A: world" and "B: WORLD" as well, giving four lines of output.

```
A: hello
B: HELLO
A: world
B: WORLD
```

6.2 Level 1

Problem 1

```
using System;

class Program
{
    static string txtAge;
    static DateTime selectedDate;
    static int parsedAge;

    static void Main(string[] args)
    {
        try
        {

            Console.WriteLine(txtAge == null ? "txtAge is null" : txtAge);



            Console.WriteLine(selectedDate == default(DateTime)
                ? "selectedDate is default"
                : selectedDate.ToString());


            if (string.IsNullOrEmpty(txtAge))
            {
                Console.WriteLine("txtAge is null or empty, cannot parse");
            }
            else
            {
                parsedAge = int.Parse(txtAge);
                Console.WriteLine($"Parsed Age: {parsedAge}");
            }
        }
        catch (FormatException)
        {
            Console.WriteLine("Format Exception Caught");
        }
        catch (ArgumentNullException)
        {
            Console.WriteLine("ArgumentNull Exception Caught");
        }
        finally
        {
            Console.WriteLine("Finally block executed");
        }
    }
}
```

Solution. Reference fields like `txtAge` default to `null`, so the first `Console.WriteLine` prints "txtAge is null". `selectedDate` is a value type with default value `default(DateTime)`, so the second line prints "selectedDate is default".

Since `string.IsNullOrEmpty(txtAge)` is true, the program prints "txtAge is null or empty, cannot parse" and does not call `int.Parse`, so no exception is thrown and no catch block executes. The `finally` block always runs, printing "Finally block executed", giving a total of four lines.

```
txtAge is null
selectedDate is default
txtAge is null or empty, cannot parse
Finally block executed
```

Problem 2

```
using System;

delegate void Operation();


class Program
{
    static void Main()
    {
        Operation ops = null;


        ops += Step1;
        ops += Step2;
        ops += Step3;


        try
        {
            ops();
        }
        catch (Exception ex)
        {
            Console.WriteLine("Caught: " + ex.Message);
        }


        Console.WriteLine("End of Main");
    }


    static void Step1()
    {
        Console.WriteLine("Step 1");
    }


    static void Step2()
    {
        Console.WriteLine("Step 2");
        throw new InvalidOperationException("Step 2 failed!");
    }


    static void Step3()
    {
        Console.WriteLine("Step 3");
    }
}
```

Answer. `Step1` runs first, printing "Step 1" and returning normally, then `Step2` prints "Step 2" and throws an `InvalidOperationException("Step 2 failed!")`, so `Step3` is never invoked. The exception is caught in the `catch` block, which prints "Caught: Step 2 failed!", and finally "End of Main" is printed after the `try/catch`.

```
Step 1
Step 2
Caught:  Step 2 failed
End of Main
```

6.3 Level 2

Problem 1

```
using System;

namespace MethodOverloadingExample
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5;
            new Base().F(x);
            new Derived().F(x);


            Console.ReadKey();
        }
    }


    class Base
    {
        public void F(int x)
        {
            Console.WriteLine("Base.F(int)");
        }
    }

    class Derived : Base
    {
        public void F(double x)
        {
            Console.WriteLine("Derived.F(double)");
        }
    }
}
```

Answer. In `Base`, only `F(int)` exists, so `new Base().F(x)` clearly calls `Base.F(int)` and prints "Base.F(int)". `Derived` inherits `Base.F(int)` and introduces an overload `F(double)`, not an override, so overload resolution for the `int` argument still prefers the exact `int` match, invoking `Base.F(int)` again.

```
Base.F(int)
Base.F(int)
```

Problem 2

```
using System;

class StepEventArgs : EventArgs
{
    public int Step { get; }
    public StepEventArgs(int s) => Step = s;
}

class Workflow
{
    public event EventHandler<StepEventArgs> StepStarted;
    public event EventHandler<StepEventArgs> StepCompleted;

    public void Run()
    {
        for (int i = 1; i <= 3; i++)
        {
            StepStarted?.Invoke(this, new StepEventArgs(i));
            Console.Write($"[{i}]");
            StepCompleted?.Invoke(this, new StepEventArgs(i));
        }
    }
}

class Program
{
    static void Main()
    {
        Workflow wf = new Workflow();

        wf.StepStarted += (s, e) =>
        {
            Console.Write("<S" + e.Step + ">");
            if (e.Step == 2)
                ((Workflow)s).StepCompleted += (snd, ev)
                    => Console.Write("(Dyn" + ev.Step + ")");
        };
        wf.StepCompleted += (s, e) => Console.Write("<C" + e.Step + ">");

        wf.Run();
    }
}
```

Answer. For step 1, the output is "<S1[1]<C1>" because only the static `StepCompleted` handler is present. For step 2, `StepStarted` first prints "<S2>" and then adds the dynamic handler, so `StepCompleted` prints "<C2>(Dyn2)" after the loop writes "[2]".

For step 3, both handlers are still attached, so the sequence is "<S3[3]<C3>(Dyn3)". Concatenating everything, the single-line output is:

```
<S1[1]<C1><S2[2]<C2>(Dyn2)<S3[3]<C3>(Dyn3)
```

7 Conclusion

This lab extended basic understanding of delegates and events to more advanced patterns such as multi-stage workflows, event chaining, and dynamic subscription management in a realistic order-processing GUI. By implementing the `OrderPipeline` application and analysing multiple output-tracing problems, it became clear how custom `EventArgs`, multicast delegates, and conditional event firing can decouple UI elements while maintaining clear, traceable control flow.