Day 7: Real-World Application (2 Hours)

OOP Course

July 21, 2025

1 Learning Objectives

By the end of today, you will:

- Build a complete application using OOP concepts from previous days
- Perform code review and refactoring to improve quality
- Write unit tests to validate your OOP code
- Plan next steps for continued learning in OOP

2 Part 1: Complete Project - Inventory Management System (90 minutes)

2.1 Project Overview

You will build an inventory management system that supports:

- Inheritance and polymorphism for different item types
- Abstract classes and interfaces for extensibility
- Generics for type-safe item storage
- Delegates and events for stock level notifications
- Exception handling for robust operations
- SOLID principles and design patterns (Factory, Observer)

2.2 Solution

```
// Abstract class for inventory items
public abstract class InventoryItem {
    public string Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public int Stock { get; protected set; }

    protected InventoryItem(string id, string name, decimal price,
       int stock) {
```

```
        if (stock < 0) throw new ArgumentException("Stock cannot be
            negative.");
        Id = id;
        Name = name;
        Price = price;
        Stock = stock;
    }

    public abstract void UpdateStock(int quantity);
    public virtual void DisplayDetails() {
        Console.WriteLine($"ID: {Id}, Name: {Name}, Price:
            {Price:C}, Stock: {Stock}");
    }
}

// Interface for notifiable items
public interface INotifiable {
    void NotifyLowStock();
}

// Concrete classes
public class Electronics : InventoryItem, INotifiable {
    public string Brand { get; set; }
    public delegate void StockHandler(string itemId, int stock);
    public event StockHandler OnLowStock;

    public Electronics(string id, string name, decimal price, int
        stock, string brand)
        : base(id, name, price, stock) {
        Brand = brand;
    }

    public override void UpdateStock(int quantity) {
        try {
            if (quantity < -Stock) {
                throw new InvalidOperationException("Cannot reduce
                    stock below zero.");
            }
            Stock += quantity;
            if (Stock < 5) {
                NotifyLowStock();
                OnLowStock?.Invoke(Id, Stock);
            }
            Console.WriteLine($"Updated stock for {Name} to
                {Stock}");
        } catch (InvalidOperationException ex) {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }

    public void NotifyLowStock() {
```

```
55          Console.WriteLine($"Warning: Low stock for {Name} (ID:
                {Id})");
56      }
57
58      public override void DisplayDetails() {
59          base.DisplayDetails();
60          Console.WriteLine($"Type: Electronics, Brand: {Brand}");
61      }
62  }
63
64  public class Clothing : InventoryItem {
65      public string Size { get; set; }
66
67      public Clothing(string id, string name, decimal price, int
            stock, string size)
68          : base(id, name, price, stock) {
69          Size = size;
70      }
71
72      public override void UpdateStock(int quantity) {
73          if (quantity < -Stock) {
74              throw new InvalidOperationException("Cannot reduce
                    stock below zero.");
75          }
76          Stock += quantity;
77          Console.WriteLine($"Updated stock for {Name} to {Stock}");
78      }
79
80      public override void DisplayDetails() {
81          base.DisplayDetails();
82          Console.WriteLine($"Type: Clothing, Size: {Size}");
83      }
84  }
85
86  // Factory for creating items
87  public class ItemFactory {
88      public InventoryItem CreateItem(string type, string id, string
            name, decimal price, int stock, string extra) {
89          return type.ToLower() switch {
90              "electronics" => new Electronics(id, name, price,
                    stock, extra),
91              "clothing" => new Clothing(id, name, price, stock,
                    extra),
92              _ => throw new ArgumentException("Invalid item type")
93          };
94      }
95  }
96
97  // Generic inventory repository
98  public class InventoryRepository<T> where T : InventoryItem {
99      private List<T> items = new List<T>();
```

```
100
101     public void AddItem(T item) {
102         try {
103             if (item == null) throw new ArgumentNullException("Item
                    cannot be null.");
104             items.Add(item);
105             Console.WriteLine($"Added {item.Name} to inventory.");
106         } catch (ArgumentNullException ex) {
107             Console.WriteLine($"Error: {ex.Message}");
108         }
109     }
110
111     public T FindItem(string id) {
112         T item = items.FirstOrDefault(i => i.Id == id);
113         if (item == null) {
114             throw new KeyNotFoundException($"Item with ID {id} not
                    found.");
115         }
116         return item;
117     }
118
119     public void DisplayInventory() {
120         Console.WriteLine("\n=== Inventory ===");
121         foreach (T item in items) {
122             item.DisplayDetails();
123             Console.WriteLine(new string('-', 30));
124         }
125     }
126 }
127
128 // Inventory manager
129 public class InventoryManager {
130     private readonly InventoryRepository<InventoryItem> repository;
131     private readonly ItemFactory factory;
132
133     public InventoryManager() {
134         repository = new InventoryRepository<InventoryItem>();
135         factory = new ItemFactory();
136     }
137
138     public void AddItem(string type, string id, string name,
            decimal price, int stock, string extra) {
139         InventoryItem item = factory.CreateItem(type, id, name,
                price, stock, extra);
140         if (item is INotifiable notifiable) {
141             if (item is Electronics electronics) {
142                 electronics.OnLowStock += (itemId, stock) =>
143                     Console.WriteLine($"Alert: Item {itemId} stock
                            at {stock} units!");
144             }
145         }
```

```
        repository.AddItem(item);
    }

    public void UpdateStock(string id, int quantity) {
        try {
            InventoryItem item = repository.FindItem(id);
            item.UpdateStock(quantity);
        } catch (KeyNotFoundException ex) {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }

    public void DisplayAllItems() {
        repository.DisplayInventory();
    }
}

// Test program
class Program {
    static void Main() {
        InventoryManager manager = new InventoryManager();

        // Add items
        manager.AddItem("electronics", "E001", "Laptop", 999.99m,
            10, "Dell");
        manager.AddItem("clothing", "C001", "T-Shirt", 19.99m, 50,
            "Medium");
        manager.AddItem("electronics", "E002", "Smartphone",
            699.99m, 3, "Samsung");

        // Display inventory
        manager.DisplayAllItems();

        // Update stock
        Console.WriteLine("\□□napp Updating Stock □□app");
        manager.UpdateStock("E001", -5); // Reduce laptop stock
        manager.UpdateStock("E002", -2); // Trigger low stock
            notification
        manager.UpdateStock("C001", 10); // Add T-shirt stock
        manager.UpdateStock("X001", 5); // Non-existent item

        // Display updated inventory
        manager.DisplayAllItems();
    }
}
```

3 Part 2: Code Review and Refactoring (15 minutes)

3.1 Code Review Checklist

- **Single Responsibility**: Each class has a clear purpose (e.g., `ItemFactory` creates items, `InventoryRepository` manages storage).
- **Open/Closed**: System is extensible via `ItemFactory` and interfaces.
- **Naming**: Descriptive names like `Electronics`, `NotifyLowStock`.
- **Error Handling**: Exceptions are caught and handled appropriately.
- **Documentation**: Add XML comments for public methods.

3.2 Refactoring Example

Original code with issues:

```
// Problematic code
public class Inventory {
    public void Add(string type, string id, string name, decimal
       price, int stock) {
        // Too much logic in one method
        if (type == "electronics") {
            // Create electronics item
        } else {
            // Create clothing item
        }
    }
}
```

Refactored code:

```
/// <summary>
/// Manages inventory operations.
/// </summary>
public class InventoryManager {
    private readonly InventoryRepository<InventoryItem> repository;
    private readonly ItemFactory factory;

    public InventoryManager() {
        repository = new InventoryRepository<InventoryItem>();
        factory = new ItemFactory();
    }

    /// <summary>
    /// Adds an item to the inventory.
    /// </summary>
    public void AddItem(string type, string id, string name,
       decimal price, int stock, string extra) {
        InventoryItem item = factory.CreateItem(type, id, name,
           price, stock, extra);
        repository.AddItem(item);
```

```
    }
}
```

3.3 Improvements

- Used Factory pattern to separate object creation
- Applied Single Responsibility by delegating storage to `InventoryRepository`
- Added XML documentation for clarity

4 Part 3: Testing Your OOP Code (15 minutes)

4.1 Unit Testing with MSTest

Create unit tests to validate the inventory system using MSTest.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

[TestClass]
public class InventoryTests {
    private InventoryManager manager;

    [TestInitialize]
    public void Setup() {
        manager = new InventoryManager();
    }

    [TestMethod]
    public void AddItem_ValidElectronicsItem_AddsSuccessfully() {
        manager.AddItem("electronics", "E001", "Laptop", 999.99m,
            10, "Dell");
        Assert.IsNotNull(manager); // Proxy for checking item
            addition
    }

    [TestMethod]
    [ExpectedException(typeof(ArgumentException))]
    public void CreateItem_InvalidType_ThrowsException() {
        manager.AddItem("invalid", "X001", "Invalid Item", 100m,
            10, "");
    }

    [TestMethod]
    [ExpectedException(typeof(InvalidOperationException))]
    public void UpdateStock_ExcessiveWithdrawal_ThrowsException() {
        manager.AddItem("clothing", "C001", "T-Shirt", 19.99m, 5,
            "Medium");
        manager.UpdateStock("C001", -10);
    }

    [TestMethod]
```

```
32      public void
           UpdateStock_LowStockElectronics_TriggersNotification() {
33          // Note: Testing events requires additional setup (e.g.,
               capturing output)
34          manager.AddItem("electronics", "E002", "Smartphone",
               699.99m, 3, "Samsung");
35          manager.UpdateStock("E002", -2); // Should trigger low
               stock event
36          Assert.IsTrue(true); // Placeholder for event validation
37      }
38  }
```

4.2 Testing Best Practices

- Test both success and failure cases
- Use meaningful test names (e.g., $\texttt{AddItem}_{V}alidElectronicsItem_{A}ddsSuccessfully)Mockdepe$
- Test edge cases and exceptions

5 Part 4: Next Steps for Continued Learning (15 minutes)

5.1 Learning Path

- **Advanced Design Patterns**: Study patterns like Decorator, Strategy, and Command.
- **Frameworks**: Explore ASP.NET Core or WPF for real-world OOP applications.
- **TDD**: Practice Test-Driven Development to write tests before code.
- **Clean Code**: Read *Clean Code* by Robert C. Martin for best practices.
- **Open Source**: Contribute to open-source C# projects on GitHub.

5.2 Resources

- Books: *Design Patterns* by Gamma et al., *C# in Depth* by Jon Skeet
- Online: Microsoft Learn, Pluralsight, GitHub
- Practice: Build larger projects (e.g., e-commerce system, game engine)

6 Day 7 Summary

6.1 What You've Learned

1. Built a complete inventory management system using OOP concepts
2. Performed code review and refactoring to improve code quality
3. Wrote unit tests to validate functionality

4. Planned next steps for mastering OOP

6.2 Best Practices

- Integrate multiple OOP concepts for cohesive designs
- Refactor code to adhere to SOLID principles
- Write comprehensive unit tests for reliability
- Continuously learn and apply new patterns and techniques