Day 6: Design Patterns (2 Hours)

OOP Course

July 21, 2025

1 Learning Objectives

By the end of today, you will:

- Understand SOLID principles and their importance in OOP
- Implement common design patterns: Singleton, Factory, and Observer
- Apply best practices for organizing and structuring code
- Design maintainable and scalable systems using design patterns

2 Part 1: SOLID Principles Overview (45 minutes)

2.1 What are SOLID Principles?

SOLID is an acronym for five design principles that promote maintainable and scalable code:

- **S**ingle Responsibility Principle: A class should have only one reason to change.
- **O**pen/Closed Principle: Classes should be open for extension but closed for modification.
- **L**iskov Substitution Principle: Subtypes must be substitutable for their base types.
- **I**nterface Segregation Principle: Clients should not be forced to depend on interfaces they don't use.
- **D**ependency Inversion Principle: High-level modules should not depend on low-level modules; both should depend on abstractions.

2.2 Example: Applying SOLID Principles

```
// Single Responsibility: Separate concerns
public class Order {
    public int Id { get; set; }
    public decimal Total { get; set; }

    public Order(int id, decimal total) {
```

```
        Id = id;
        Total = total;
    }
}

public class OrderRepository {
    public void Save(Order order) {
        Console.WriteLine($"Saving order {order.Id} to database.");
    }
}

// Open/Closed: Extend behavior without modifying
public interface IPaymentProcessor {
    bool ProcessPayment(decimal amount);
}

public class CreditCardProcessor : IPaymentProcessor {
    public bool ProcessPayment(decimal amount) {
        Console.WriteLine($"Processing credit card payment of
            {amount:C}");
        return true;
    }
}

public class PayPalProcessor : IPaymentProcessor {
    public bool ProcessPayment(decimal amount) {
        Console.WriteLine($"Processing PayPal payment of
            {amount:C}");
        return true;
    }
}

// Liskov Substitution: Use base type without altering behavior
public class OrderProcessor {
    public void Process(Order order, IPaymentProcessor processor) {
        if (processor.ProcessPayment(order.Total)) {
            Console.WriteLine($"Order {order.Id} processed
                successfully.");
        }
    }
}
```

2.3 SOLID in Action

```
class Program {
    static void Main() {
        Order order = new Order(1, 99.99m);
        OrderRepository repo = new OrderRepository();
        IPaymentProcessor processor = new CreditCardProcessor();

        OrderProcessor orderProcessor = new OrderProcessor();
```

```
        orderProcessor.Process(order, processor); // Works with any
            IPaymentProcessor
        repo.Save(order);

        // Swap processor without modifying OrderProcessor
        processor = new PayPalProcessor();
        orderProcessor.Process(order, processor);
    }
}
```

3 Part 2: Common Patterns: Singleton, Factory, Observer (60 minutes)

3.1 Singleton Pattern

The Singleton pattern ensures a class has only one instance and provides a global point of access to it.

```
// Singleton pattern
public class Logger {
    private static Logger instance;
    private static readonly object lockObject = new object();

    private Logger() {} // Private constructor prevents
        instantiation

    public static Logger Instance {
        get {
            lock (lockObject) {
                if (instance == null) {
                    instance = new Logger();
                }
                return instance;
            }
        }
    }

    public void Log(string message) {
        Console.WriteLine($"Log: {message} at {DateTime.Now}");
    }
}
```

3.2 Factory Pattern

The Factory pattern creates objects without specifying the exact class of object that will be created.

```
// Factory pattern
public interface INotification {
    void Send(string message);
```

```
}

public class EmailNotification : INotification {
    public void Send(string message) {
        Console.WriteLine($"Email: {message}");
    }
}

public class SMSNotification : INotification {
    public void Send(string message) {
        Console.WriteLine($"SMS: {message}");
    }
}

public class NotificationFactory {
    public INotification CreateNotification(string type) {
        return type.ToLower() switch {
            "email" => new EmailNotification(),
            "sms" => new SMSNotification(),
            _ => throw new ArgumentException("Invalid notification
                type")
        };
    }
}
```

3.3 Observer Pattern

The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.

```
// Observer pattern
public interface IObserver {
    void Update(string message);
}

public class Subscriber : IObserver {
    private string name;

    public Subscriber(string name) {
        this.name = name;
    }

    public void Update(string message) {
        Console.WriteLine($"{name} received: {message}");
    }
}

public class NewsAgency {
    private List<IObserver> observers = new List<IObserver>();
    private string news;

```

```
    public string News {
        get => news;
        set {
            news = value;
            Notify();
        }
    }

    public void Subscribe(IObserver observer) {
        observers.Add(observer);
    }

    public void Unsubscribe(IObserver observer) {
        observers.Remove(observer);
    }

    private void Notify() {
        foreach (var observer in observers) {
            observer.Update(news);
        }
    }
}
```

3.4 Using Patterns

```
class Program {
    static void Main() {
        // Singleton
        Logger logger = Logger.Instance;
        logger.Log("Application started");

        // Factory
        NotificationFactory factory = new NotificationFactory();
        INotification notification =
           factory.CreateNotification("email");
        notification.Send("Meeting at 3 PM");

        // Observer
        NewsAgency agency = new NewsAgency();
        Subscriber sub1 = new Subscriber("Alice");
        Subscriber sub2 = new Subscriber("Bob");

        agency.Subscribe(sub1);
        agency.Subscribe(sub2);
        agency.News = "Breaking: New product launch!";
    }
}
```

4 Part 3: Best Practices and Code Organization (15 minutes)

4.1 Guidelines

- **Naming Conventions**: Use clear, descriptive names (e.g., `OrderRepository`, `IPaymentProcessor`).
- **Folder Structure**: Organize code by feature or layer (e.g., Models, Services, Repositories).
- **Single Responsibility**: Ensure classes and methods do one thing well.
- **Documentation**: Use XML comments for public APIs.
- **Testing**: Write unit tests for critical functionality.

```
// Example of organized code with documentation
/// <summary>
/// Manages customer data and operations.
/// </summary>
public class CustomerService {
    private readonly ICustomerRepository repository;

    public CustomerService(ICustomerRepository repository) {
        this.repository = repository;
    }

    /// <summary>
    /// Adds a customer to the repository.
    /// </summary>
    /// <param name="customer">The customer to add.</param>
    public void AddCustomer(Customer customer) {
        repository.Add(customer);
    }
}
```

5 Day 6 Summary

5.1 What You've Learned

1. SOLID principles promote maintainable and scalable code
2. Singleton ensures a single instance, Factory creates objects dynamically, and Observer enables event-driven updates
3. Best practices improve code readability and maintainability

5.2 Best Practices

- Apply SOLID principles to design robust systems
- Use design patterns appropriately to solve common problems
- Organize code logically and document public APIs

- Keep classes focused and loosely coupled

5.3 Tomorrow's Preview

Day 7 will cover a **Real-World Application**, including building a complete project, code review, refactoring, testing, and planning next steps.

6 Additional Practice Exercises

1. Implement a `Decorator` pattern for a coffee shop order system.
2. Create a `Strategy` pattern for different sorting algorithms.
3. Build a `Command` pattern for an undoable text editor.