

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:

- **Single Responsibility Principle:** A class should have **only one reason to change.**
- **Open/Closed Principle:** **Classes should be open for extension but closed for modification.**
- **Liskov Substitution Principle:** **Subtypes must be substitutable for their base types.**
- **Interface Segregation Principle:** **Clients should not be forced to depend on interfaces they don't use.**
- **Dependency Inversion Principle:** **High-level modules should not depend on low-level modules; both should depend on abstractions.**

2.2 Example: Applying SOLID Principles

```
1 // Single Responsibility: Separate concerns
2 public class Order {
3     public int Id { get; set; }
4     public decimal Total { get; set; }
5
6     public Order(int id, decimal total) {
```

```

7         Id = id;
8         Total = total;
9     }
10 }
11
12 public class OrderRepository {
13     public void Save(Order order) {
14         Console.WriteLine($"Saving order {order.Id} to database.");
15     }
16 }
17
18 // Open/Closed: Extend behavior without modifying
19 public interface IPaymentProcessor {
20     bool ProcessPayment(decimal amount);
21 }
22
23 public class CreditCardProcessor : IPaymentProcessor {
24     public bool ProcessPayment(decimal amount) {
25         Console.WriteLine($"Processing credit card payment of
26             {amount:C}");
27         return true;
28     }
29 }
30
31 public class PayPalProcessor : IPaymentProcessor {
32     public bool ProcessPayment(decimal amount) {
33         Console.WriteLine($"Processing PayPal payment of
34             {amount:C}");
35         return true;
36     }
37 }
38
39 // Liskov Substitution: Use base type without altering behavior
40 public class OrderProcessor {
41     public void Process(Order order, IPaymentProcessor processor) {
42         if (processor.ProcessPayment(order.Total)) {
43             Console.WriteLine($"Order {order.Id} processed
44                 successfully.");
45         }
46     }
47 }

```

2.3 SOLID in Action

```

1 class Program {
2     static void Main() {
3         Order order = new Order(1, 99.99m);
4         OrderRepository repo = new OrderRepository();
5         IPaymentProcessor processor = new CreditCardProcessor();
6
7         OrderProcessor orderProcessor = new OrderProcessor();

```

```

8      orderProcessor.Process(order, processor); // Works with any
          IPaymentProcessor
9      repo.Save(order);
10
11      // Swap processor without modifying OrderProcessor
12      processor = new PayPalProcessor();
13      orderProcessor.Process(order, processor);
14  }
15 }

```

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.

```

1 // Singleton pattern
2 public class Logger {
3     private static Logger instance;
4     private static readonly object lockObject = new object();
5
6     private Logger() {} // Private constructor prevents
        instantiation
7
8     public static Logger Instance {
9         get {
10             lock (lockObject) {
11                 if (instance == null) {
12                     instance = new Logger();
13                 }
14                 return instance;
15             }
16         }
17     }
18
19     public void Log(string message) {
20         Console.WriteLine($"Log: {message} at {DateTime.Now}");
21     }
22 }

```

3.2 Factory Pattern

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

```

1 // Factory pattern
2 public interface INotification {
3     void Send(string message);

```

```

4 }
5
6 public class EmailNotification : INotification {
7     public void Send(string message) {
8         Console.WriteLine($"Email: {message}");
9     }
10 }
11
12 public class SMSNotification : INotification {
13     public void Send(string message) {
14         Console.WriteLine($"SMS: {message}");
15     }
16 }
17
18 public class NotificationFactory {
19     public INotification CreateNotification(string type) {
20         return type.ToLower() switch {
21             "email" => new EmailNotification(),
22             "sms" => new SMSNotification(),
23             _ => throw new ArgumentException("Invalid notification
24                                     type");
25         };
26     }
27 }

```

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.

```

1 // Observer pattern
2 public interface IObserver {
3     void Update(string message);
4 }
5
6 public class Subscriber : IObserver {
7     private string name;
8
9     public Subscriber(string name) {
10         this.name = name;
11     }
12
13     public void Update(string message) {
14         Console.WriteLine($"{name} received: {message}");
15     }
16 }
17
18 public class NewsAgency {
19     private List<IObserver> observers = new List<IObserver>();
20     private string news;
21 }

```

```

22     public string News {
23         get => news;
24         set {
25             news = value;
26             Notify();
27         }
28     }
29
30     public void Subscribe(IObserver observer) {
31         observers.Add(observer);
32     }
33
34     public void Unsubscribe(IObserver observer) {
35         observers.Remove(observer);
36     }
37
38     private void Notify() {
39         foreach (var observer in observers) {
40             observer.Update(news);
41         }
42     }
43 }

```

3.4 Using Patterns

```

1  class Program {
2      static void Main() {
3          // Singleton
4          Logger logger = Logger.Instance;
5          logger.Log("Application started");
6
7          // Factory
8          NotificationFactory factory = new NotificationFactory();
9          INotification notification =
10             factory.CreateNotification("email");
11             notification.Send("Meeting at 3 PM");
12
13             // Observer
14             NewsAgency agency = new NewsAgency();
15             Subscriber sub1 = new Subscriber("Alice");
16             Subscriber sub2 = new Subscriber("Bob");
17
18             agency.Subscribe(sub1);
19             agency.Subscribe(sub2);
20             agency.News = "Breaking: New product launch!";
21     }
22 }

```

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.

```
1 // Example of organized code with documentation
2 /// <summary>
3 /// Manages customer data and operations.
4 /// </summary>
5 public class CustomerService {
6     private readonly ICustomerRepository repository;
7
8     public CustomerService(ICustomerRepository repository) {
9         this.repository = repository;
10    }
11
12    /// <summary>
13    /// Adds a customer to the repository.
14    /// </summary>
15    /// <param name="customer">The customer to add.</param>
16    public void AddCustomer(Customer customer) {
17        repository.Add(customer);
18    }
19 }
```

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.