1. Create a Class Hierarchy Demonstrating Inheritance and Polymorphism

Problem Statement:
Design a class hierarchy for different types of vehicles. Implement inheritance and polymorphism to demonstrate how various vehicle classes can override common methods.

C# Implementation:

```
public abstract class Vehicle
{
    public string Name { get; set; }

    public abstract void Start();
    public abstract void Stop();
}

public class Car : Vehicle
{
    public override void Start()
    {
        Console.WriteLine("Car is starting.");
    }

    public override void Stop()
    {
        Console.WriteLine("Car has stopped.");
    }
}

public class Motorcycle : Vehicle
{
    public override void Start()
    {
        Console.WriteLine("Motorcycle is starting.");
    }

    public override void Stop()
    {
        Console.WriteLine("Motorcycle has stopped.");
    }
}

public class Program
{
    public static void Main()
    {
        List<Vehicle> vehicles = new List<Vehicle>
        {
            new Car { Name = "Toyota" },
            new Motorcycle { Name = "Harley Davidson" }
        };

        foreach (var vehicle in vehicles)
        {
            Console.WriteLine(vehicle.Name);
            vehicle.Start();
            vehicle.Stop();
        }
    }
}
```

Answer Explanation:

This solution demonstrates the principles of inheritance and polymorphism. The `Vehicle` class is abstract with abstract methods `Start()` and `Stop()`. Derived classes `Car` and `Motorcycle` override these methods. Polymorphism is shown when objects of different types are stored in a `List<Vehicle>`, and their respective methods are called based on their runtime type.

Difficulty Rating: Intermediate

2. Implement a LINQ Query to Filter and Transform Data

Problem Statement:
Write a LINQ query to filter employees who earn more than 50,000 and project only their names and salaries.

C# Implementation:

```
using System.Collections.Generic;
using System.Linq;

public class Employee
{
    public string Name { get; set; }
    public int Salary { get; set; }
}

public class Program
{
    public static void Main()
    {
        List<Employee> employees = new List<Employee>
        {
            new Employee { Name = "Alice", Salary = 60000 },
            new Employee { Name = "Bob", Salary = 45000 },
            new Employee { Name = "Charlie", Salary = 70000 }
        };

        var highEarners = from emp in employees
                          where emp.Salary > 50000
                          select new { emp.Name,(emp.Salary) };

        foreach (var emp in highEarners)
        {
            Console.WriteLine($"{emp.Name} earns {emp.Salary}");
        }
    }
}
```

Answer Explanation:

This query uses LINQ to filter employees with a salary over 50,000 and projects their names and salaries. It demonstrates the use of query syntax with `from`, `where`, and `select` clauses, creating an anonymous type for the result.

Difficulty Rating: Intermediate

3. Asynchronous File Reading Using Async/Await

Problem Statement:
Implement an asynchronous method to read the contents of a text file.

C# Implementation:

```
using System;
using System.IO;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {
        string content = await ReadFileAsync("example.txt");
        Console.WriteLine(content);
    }

    public static async Task<string> ReadFileAsync(string path)
    {
        using (StreamReader reader = new StreamReader(path))
        {
            return await reader.ReadToEndAsync();
        }
    }
}
```

Answer Explanation:

This code uses asynchronous methods to read a file. `ReadFileAsync` is an async method that reads the file content asynchronously using `StreamReader.ReadToEndAsync()`. The `Main` method awaits this task, allowing other operations to run concurrently.

Difficulty Rating: Intermediate

4. Implement a Custom Exception and Handle It

Problem Statement:
Create a custom exception for invalid age input and handle it in a method.

C# Implementation:

```
public class InvalidAgeException : Exception
{
    public InvalidAgeException(string message) : base(message)
    {
    }
}

public class Person
{
    public int Age { get; set; }

    public void SetAge(int age)
    {
        if (age < 0 || age > 120)
        {
            throw new InvalidAgeException("Age must be between 0 and 120.");
        }
        Age = age;
    }
}

public class Program
{
    public static void Main()
    {
        try
        {
            Person person = new Person();
            person.SetAge(130);
        }
        catch (InvalidAgeException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
}
```

Answer Explanation:

This solution defines a custom exception `InvalidAgeException` and uses it in the `SetAge` method to validate age input. Proper exception handling is demonstrated with a try-catch block.

Difficulty Rating: Intermediate

5. Generic Stack Implementation Using Collections.Generic.Stack

Problem Statement:
Create a generic stack to manage the order of customer service requests.

C# Implementation:

```
using System.Collections.Generic;

public class Program
{
    public static void Main()
    {
        Stack<CustomerRequest> requestStack = new Stack<CustomerRequest>();

        requestStack.Push(new CustomerRequest("John", "Account Issue"));
        requestStack.Push(new CustomerRequest("Alice", "Technical Support"));

        while (requestStack.Count > 0)
        {
```

```
            CustomerRequest request = requestStack.Pop();
            Console.WriteLine($"Processing: {request.CustomerName} - {request.Issue}");
        }
    }
}

public class CustomerRequest
{
    public string CustomerName { get; set; }
    public string Issue { get; set; }

    public CustomerRequest(string name, string issue)
    {
        CustomerName = name;
        Issue = issue;
    }
}
```

Answer Explanation:

Using `Stack<T>`, this example demonstrates a generic stack to manage customer service requests. Requests are added using `Push` and processed in LIFO order with `Pop`.

Difficulty Rating: Intermediate

6. Serialize and Deserialize an Object to JSON

Problem Statement:
Serialize a product object to JSON and deserialize it back using Newtonsoft's Json.NET.

C# Implementation:

```
using Newtonsoft.Json;
using System;

public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}

public class Program
{
    public static void Main()
    {
        Product product = new Product { Name = "Laptop", Price = 1299.99m };

        string json = JsonConvert.SerializeObject(product);
        Console.WriteLine("Serialized JSON:");
        Console.WriteLine(json);

        Product deserializedProduct = JsonConvert.DeserializeObject<Product>(json);
        Console.WriteLine("\nDeserialized Object:");
        Console.WriteLine(deserializedProduct.Name + " - " + deserializedProduct.Price);
    }
}
```

Answer Explanation:

This example uses `JsonConvert.SerializeObject` to convert a `Product` object to JSON string and `DeserializeObject` to convert it back. It demonstrates the use of Json.NET for serialization.

Difficulty Rating: Intermediate

7. Implement Dependency Injection Using Constructor Injection

Problem Statement:
Create a class that uses dependency injection to manage a logging service.

C# Implementation:

```
public interface ILogger
```

```
{
    void Log(string message);
}

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

public class UserService
{
    private readonly ILogger _logger;

    public UserService(ILogger logger)
    {
        _logger = logger;
    }

    public void CreateUser(string username)
    {
        _logger.Log($"User {username} created.");
    }
}

public class Program
{
    public static void Main()
    {
        ILogger logger = new ConsoleLogger();
        UserService userService = new UserService(logger);

        userService.CreateUser("john_doe");
    }
}
```

Answer Explanation:

This solution uses constructor injection to provide the `ILogger` dependency to `UserService`. The `ConsoleLogger` implements the interface, logging messages to the console.

Difficulty Rating: Intermediate

8. Unit Test a Method Using xUnit

Problem Statement:
Write unit tests for a method that calculates the sum of two integers.

C# Implementation:

```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}

// Unit test using xUnit
public class CalculatorTests
{
    [Fact]
    public void Add_WhenCalledWithTwoNumbers_ReturnsSum()
    {
        Calculator calculator = new Calculator();
        int result = calculator.Add(5, 3);
        Assert.Equal(8, result);
    }

    [Fact]
    public void Add_NegativeNumbers_ReturnsCorrectSum()
    {
        Calculator calculator = new Calculator();
```

```
        int result = calculator.Add(-2, 3);
        Assert.Equal(1, result);
    }
}
```

Answer Explanation:

These unit tests use xUnit to test the `Add` method of a `Calculator`. The tests cover both positive and negative integers, ensuring correct functionality.

Difficulty Rating: Intermediate

9. Read a File Asynchronously Using FileStream and StreamReader

Problem Statement:
Asynchronously read the contents of a large text file using `FileStream` and `StreamReader`.

C# Implementation:

```
using System;
using System.IO;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {
        string content = await ReadLargeFileAsync("largefile.txt");
        Console.WriteLine(content.Length);
    }

    public static async Task<string> ReadLargeFileAsync(string path)
    {
        using (FileStream fileStream = new FileStream(path, FileMode.Open, FileAccess.Read, FileShare.Read,
bufferSize: 4096, useAsync: true))
        using (StreamReader reader = new StreamReader(fileStream))
        {
            return await reader.ReadToEndAsync();
        }
    }
}
```

Answer Explanation:

This code reads a large file asynchronously using `FileStream` with asynchronous mode enabled and `StreamReader.ReadToToEndAsync()`. This is efficient for large files as it reads data in chunks without blocking the main thread.

Difficulty Rating: Intermediate

10. Use Reflection to Dynamically Create an Instance of a Class

Problem Statement:
Use reflection to create an instance of a class without referencing it directly.

C# Implementation:

```
using System;
using System.Reflection;

public class Program
{
    public static void Main()
    {
        string className = "MyClass";

        Type type = Type.GetType($"ReflectionExample.{className}, ReflectionExample");
        if (type != null)
        {
            object instance = Activator.CreateInstance(type);
            Console.WriteLine("Instance created successfully.");
        }
        else
```

```
        {
            Console.WriteLine("Class not found.");
        }
    }
}

public class MyClass
{
    public MyClass()
    {
        Console.WriteLine("MyClass constructor called.");
    }
}
```

Answer Explanation:

This example uses reflection to dynamically create an instance of `MyClass` by its name. It demonstrates obtaining a `Type` object and using `Activator.CreateInstance` to instantiate it.

Difficulty Rating: Intermediate

11. Implement a Singleton Pattern Using Lazy Initialization

Problem Statement:
Create a singleton class that ensures only one instance exists using lazy initialization.

C# Implementation:

```
public sealed class Singleton
{
    private static readonly Lazy<Singleton> _instance = new Lazy<Singleton>(() => new Singleton());

    private Singleton() { }

    public static Singleton Instance
    {
        get { return _instance.Value; }
    }

    public string GetDate()
    {
        return DateTime.Now.ToString();
    }
}

public class Program
{
    public static void Main()
    {
        Singleton instance1 = Singleton.Instance;
        Singleton instance2 = Singleton.Instance;

        Console.WriteLine(instance1.GetDate());
        Console.WriteLine(ReferenceEquals(instance1, instance2)); // True
    }
}
```

Answer Explanation:

This singleton implementation uses `Lazy<T>` for thread-safe lazy initialization. The private constructor prevents instantiation, and the `Instance` property provides access to the single instance.

Difficulty Rating: Intermediate

12. Implement a Factory Pattern for Shape Creation

Problem Statement:
Create a factory class to produce different types of shapes.

C# Implementation:

```
public interface IShape
```

```
{
    void Draw();
}

public class Circle : IShape
{
    public void Draw()
    {
        Console.WriteLine("Drawing a circle.");
    }
}

public class Square : IShape
{
    public void Draw()
    {
        Console.WriteLine("Drawing a square.");
    }
}

public class ShapeFactory
{
    public IShape CreateShape(string shapeType)
    {
        switch (shapeType.ToLower())
        {
            case "circle":
                return new Circle();
            case "square":
                return new Square();
            default:
                throw new ArgumentException("Invalid shape type.");
        }
    }
}

public class Program
{
    public static void Main()
    {
        ShapeFactory factory = new ShapeFactory();

        IShape circle = factory.CreateShape("circle");
        circle.Draw();

        IShape square = factory.CreateShape("square");
        square.Draw();
    }
}
```

Answer Explanation:

The `ShapeFactory` uses the Factory pattern to create different `IShape` implementations based on input. This promotes loose coupling and simplifies object creation.

Difficulty Rating: Intermediate

13. Implement a Custom Event in C#

Problem Statement:
Create a class with a custom event that is triggered when a property changes.

C# Implementation:

```
public class TemperatureMonitor
{
    public event EventHandler<TemperatureChangedEventArgs> TemperatureChanged;

    private double _temperature;

    public double Temperature
    {
        get => _temperature;
        set
        {
```

```
            if (_temperature != value)
            {
                _temperature = value;
                OnTemperatureChanged(new TemperatureChangedEventArgs { NewValue = value });
            }
        }
    }

    protected virtual void OnTemperatureChanged(TemperatureChangedEventArgs e)
    {
        TemperatureChanged?.Invoke(this, e);
    }
}

public class TemperatureChangedEventArgs : EventArgs
{
    public double NewValue { get; set; }
}

public class Program
{
    public static void Main()
    {
        TemperatureMonitor monitor = new TemperatureMonitor();
        monitor.TemperatureChanged += (sender, args) =>
        {
            Console.WriteLine($"Temperature changed to: {args.NewValue}°C");
        };

        monitor.Temperature = 25;
        monitor.Temperature = 30;
    }
}
```

Answer Explanation:

This code defines a `TemperatureMonitor` class with a custom `TemperatureChanged` event. When the `Temperature` property changes, it triggers the event with a new value.

Difficulty Rating: Intermediate

14. Implement IDisposable Correctly for Resource Management

Problem Statement:
Create a class that implements `IDisposable` to manage unmanaged resources.

C# Implementation:

```
public class ResourceHolder : IDisposable
{
    private bool _disposed = false;
    public IntPtr Handle { get; }

    public ResourceHolder()
    {
        Handle = AllocateResource();
        Console.WriteLine("Resource allocated.");
    }

    ~ResourceHolder()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!_disposed)
        {
            if (disposing)
```

```
            {
                // Dispose managed resources here
            }

            FreeResource(Handle);
            _disposed = true;
            Console.WriteLine("Resource disposed.");
        }
    }

    // Simulated resource allocation and freeing
    private IntPtr AllocateResource()
    {
        return new IntPtr(123);
    }

    private void FreeResource(IntPtr handle)
    {
        // Code to free resource
        Console.WriteLine("Resource freed.");
    }
}

public class Program
{
    public static void Main()
    {
        using (ResourceHolder holder = new ResourceHolder())
        {
            // Use the resource
        }

        Console.WriteLine("Main method completed.");
    }
}
```

Answer Explanation:

This `ResourceHolder` class correctly implements `IDisposable` to manage resource cleanup. It includes both deterministic disposal via the `using` statement and finalization. The `Dispose(bool)` method handles resource release, and `GC.SuppressFinalize` prevents unnecessary finalization.

Difficulty Rating: Intermediate

15. Make HTTP GET Request Using HttpClient

Problem Statement:
Use `HttpClient` to make a GET request to an external API and handle the response.

C# Implementation:

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {
        HttpClient client = new HttpClient();

        try
        {
            HttpResponseMessage response = await client.GetAsync("https://api.example.com/data");
            response.EnsureSuccessStatusCode();

            string content = await response.Content.ReadAsStringAsync();
            Console.WriteLine(content);
        }
        catch (HttpRequestException ex)
        {
            Console.WriteLine($"Request failed: {ex.Message}");
        }
    }
}
```

Answer Explanation:

This code uses `HttpClient` to asynchronously make a GET request. It handles potential exceptions with `EnsureSuccessStatusCode()` and catches `HttpRequestException` for error handling.

Difficulty Rating: Intermediate

16. Implement a Thread Pool with Tasks for Parallel Processing

Problem Statement:
Use `ThreadPool` to execute multiple tasks in parallel.

C# Implementation:

```
using System;
using System.Threading;

public class Program
{
    public static void Main()
    {
        for (int i = 0; i < 5; i++)
        {
            ThreadPool.QueueUserWorkItem(ProcessTask, i);
        }

        Console.WriteLine("Main thread continues execution.");
        Thread.Sleep(1000); // Allow tasks to run
    }

    private static void ProcessTask(object state)
    {
        int taskId = (int)state;
        Console.WriteLine($"Task {taskId} is running on thread: {Thread.CurrentThread.ManagedThreadId}");
        Thread.Sleep(500);
    }
}
```

Answer Explanation:

This example uses `ThreadPool.QueueUserWorkItem` to enqueue tasks for parallel execution. Each task runs on a thread from the ThreadPool, demonstrating asynchronous task processing.

Difficulty Rating: Intermediate

17. Create a Custom LINQ Extension Method

Problem Statement:
Write a custom LINQ extension method to count elements greater than a specified value.

C# Implementation:

```
using System.Collections.Generic;
using System.Linq;

public static class MyLinqExtensions
{
    public static int CountGreaterThan<TSource>(
        this IEnumerable<TSource> source,
        TSource value) where TSource : IComparable<TSource>
    {
        return source.Count(item => item.CompareTo(value) > 0);
    }
}

public class Program
{
    public static void Main()
    {
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
        int count = numbers.CountGreaterThan(3);
        Console.WriteLine(count); // Output: 2
```

```
    }
}
```

Answer Explanation:

This custom LINQ extension method `CountGreaterThan` counts elements in a collection that are greater than a specified value. It uses the `IComparable` interface to compare elements.

Difficulty Rating: Intermediate

18. Implement Asynchronous Programming with CancellationToken

Problem Statement:
Implement an asynchronous method that can be canceled using `CancellationToken`.

C# Implementation:

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {
        CancellationTokenSource cts = new CancellationTokenSource();

        Console.WriteLine("Starting the task...");
        Task<int> longTask = DoLongOperation(cts.Token);

        Console.WriteLine("Press any key to cancel...");
        Console.ReadKey();
        cts.Cancel();

        try
        {
            int result = await longTask;
            Console.WriteLine($"Result: {result}");
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("Task was canceled.");
        }
    }

    private static async Task<int> DoLongOperation(CancellationToken token)
    {
        for (int i = 0; i < 10; i++)
        {
            token.ThrowIfCancellationRequested();
            Console.WriteLine($"Processing step {i + 1}");
            await Task.Delay(500, token);
        }
        return 42;
    }
}
```

Answer Explanation:

This code demonstrates an asynchronous task that can be canceled. The `DoLongOperation` method checks the cancellation token periodically and throws an exception if canceled, which is caught in `Main`.

Difficulty Rating: Intermediate

19. Implement a Generic Repository Pattern for Data Access

Problem Statement:
Create a generic repository pattern to encapsulate data access logic.

C# Implementation:

```
public interface IRepository<T>
```

```
{
    T GetById(int id);
    void Add(T entity);
    void Update(T entity);
    void Delete(T entity);
}

public class Repository<T> : IRepository<T>
{
    public T GetById(int id)
    {
        // Implementation to get by ID
        return default;
    }

    public void Add(T entity)
    {
        // Implementation to add entity
    }

    public void Update(T entity)
    {
        // Implementation to update entity
    }

    public void Delete(T entity)
    {
        // Implementation to delete entity
    }
}

public class Program
{
    public static void Main()
    {
        IRepository<Customer> customerRepo = new Repository<Customer>();
        Customer customer = customerRepo.GetById(1);
    }
}

public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Answer Explanation:

This generic repository pattern encapsulates data access methods for any entity type. It provides basic CRUD operations, which can be implemented with specific data access logic (e.g., Entity Framework).

Difficulty Rating: Intermediate

20. Implement a Custom Attribute and Use Reflection to Read It

Problem Statement:
Create a custom attribute to mark methods as debug-only and use reflection to read it.

C# Implementation:

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.Method, Inherited = false)]
public class DebugOnlyAttribute : Attribute
{
    public bool IsEnabled { get; set; }
}

public class MyClass
{
    [DebugOnly(IsEnabled = true)]
    public void DebugMethod()
    {
        Console.WriteLine("This is a debug method.");
```

```
    }
}

public class Program
{
    public static void Main()
    {
        MethodInfo method = typeof(MyClass).GetMethod("DebugMethod");
        DebugOnlyAttribute attribute =
(DebugOnlyAttribute)method.GetCustomAttribute(typeof(DebugOnlyAttribute));

        if (attribute != null && attribute.IsEnabled)
        {
            MyClass instance = new MyClass();
            instance.DebugMethod();
        }
    }
}
```

Answer Explanation:

This example defines a `DebugOnly` attribute and applies it to a method. Using reflection, the program checks for this attribute and conditionally calls the method.

Difficulty Rating: Intermediate

21. Implement Asynchronous File Operations with Parallelism

Problem Statement:
Read multiple files asynchronously using parallel tasks.

C# Implementation:

```
using System;
using System.IO;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {
        string[] files = { "file1.txt", "file2.txt", "file3.txt" };

        var tasks = files.Select(ReadFileAsync);
        await Task.WhenAll(tasks);

        foreach (var content in tasks)
        {
            Console.WriteLine(content.Result);
        }
    }

    public static async Task<string> ReadFileAsync(string path)
    {
        try
        {
            using (StreamReader reader = new StreamReader(path))
            {
                return await reader.ReadToEndAsync();
            }
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine($"File {path} not found.");
            return null;
        }
    }
}
```

Answer Explanation:

This code uses `Task.WhenAll` to read multiple files asynchronously. Each file read is a separate task, and their results are processed once all tasks complete.

Difficulty Rating: Intermediate

22. Implement a Class with Explicit Interface Implementation

Problem Statement:
Create a class that explicitly implements an interface's method.

C# Implementation:

```
public interface IWorker
{
    void Work();
}

public class Employee : IWorker
{
    string IWorker.Work()
    {
        return "Employee is working.";
    }
}

public class Program
{
    public static void Main()
    {
        IWorker worker = new Employee();
        Console.WriteLine(worker.Work());

        // Compiler error: Employee does not contain a definition for 'Work'
        // Employee employee = new Employee();
        // Console.WriteLine(employee.Work());
    }
}
```

Answer Explanation:

The `Employee` class explicitly implements the `Work` method of `IWorker`, making it accessible only through the interface reference. Direct access from an `Employee` instance causes a compiler error.

Difficulty Rating: Intermediate

23. Use LINQ to Group and Aggregate Data

Problem Statement:
Group products by category and calculate the average price.

C# Implementation:

```
using System.Collections.Generic;
using System.Linq;

public class Product
{
    public string Category { get; set; }
    public decimal Price { get; set; }
}

public class Program
{
    public static void Main()
    {
        List<Product> products = new List<Product>
        {
            new Product { Category = "Electronics", Price = 500m },
            new Product { Category = "Electronics", Price = 600m },
            new Product { Category = "Clothing", Price = 100m }
        };

        var groupedProducts = products.GroupBy(p => p.Category)
            .Select(g => new { Category = g.Key, AvgPrice = g.Average(p => p.Price) });

        foreach (var group in groupedProducts)
```

```
        {
            Console.WriteLine($"{group.Category}: {group.AvgPrice:C}");
        }
    }
}
```

Answer Explanation:

This LINQ query groups products by their category and calculates the average price for each group. It demonstrates `GroupBy` and `Select` with aggregation.

Difficulty Rating: Intermediate

24. Implement a Custom Exception Filter in Global.aspx

Problem Statement:
Create a custom exception filter to handle specific exceptions globally.

C# Implementation:

```
using System.Web.Mvc;

public class GlobalFilters : FilterProvider
{
    public override IEnumerable<Filter> GetFilters(ControllerContext controllerContext)
    {
        return new Filter[]
        {
            new HandleErrorAttribute()
        };
    }
}

public class CustomExceptionFilter : IExceptionFilter
{
    public void OnException(ExceptionContext filterContext)
    {
        if (filterContext.Exception is ArgumentException)
        {
            filterContext.Result = new ViewResult
            {
                ViewName = "Error/Argument"
            };
        }
    }
}
```

Answer Explanation:

This setup uses `GlobalFilters` and a custom exception filter to handle specific exceptions globally in an ASP.NET MVC application.

Difficulty Rating: Intermediate

25. Implement a Cache Using Dictionary with Expiration

Problem Statement:
Create a cache that stores values with expiration times.

C# Implementation:

```
using System;
using System.Collections.Generic;

public class Cache<T>
{
    private Dictionary<string, CacheItem<T>> _items = new Dictionary<string, CacheItem<T>>();

    public T Get(string key)
    {
        if (_items.TryGetValue(key, out CacheItem<T> item))
        {
```

```
            if (DateTime.Now < item.ExpiryTime)
                return item.Value;

            _items.Remove(key);
        }

        return default;
    }

    public void Set(string key, T value, int cacheTimeInMinutes)
    {
        DateTime expiry = DateTime.Now.AddMinutes(cacheTimeInMinutes);
        _items[key] = new CacheItem<T> { Value = value, ExpiryTime = expiry };
    }

    private class CacheItem<T>
    {
        public T Value { get; set; }
        public DateTime ExpiryTime { get; set; }
    }
}

public class Program
{
    public static void Main()
    {
        Cache<string> cache = new Cache<string>();
        cache.Set("key1", "Hello, World!", 5);

        string value = cache.Get("key1");
        Console.WriteLine(value); // Output: Hello, World!

        System.Threading.Thread.Sleep(60000);
        value = cache.Get("key1");
        Console.WriteLine(value); // Output:
    }
}
```

Answer Explanation:

This cache implementation uses a dictionary to store values with expiration times. The `Get` method checks if the item is expired and removes it if so, while `Set` adds or updates the cache entry with an expiration time.

Difficulty Rating: Intermediate

26. Implement a Simple REST API Client Using HttpClient

Problem Statement:
Create a client to consume a REST API that returns JSON data.

C# Implementation:

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {
        HttpClient client = new HttpClient();

        try
        {
            HttpResponseMessage response = await client.GetAsync("https://api.example.com/users");
            response.EnsureSuccessStatusCode();

            string content = await response.Content.ReadAsStringAsync();
            Console.WriteLine(content);
        }
        catch (HttpRequestException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
```

```
}
```

Answer Explanation:

This example uses `HttpClient` to asynchronously make a GET request to a REST API. It handles exceptions and prints the JSON response.

Difficulty Rating: Intermediate

27. Implement a Custom Sorting Algorithm Using LINQ

Problem Statement:
Sort a list of strings by their length and then alphabetically.

C# Implementation:

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        List<string> words = new List<string> { "apple", "banana", "cherry", "date" };

        var sortedWords = words.OrderBy(w => w.Length)
                               .ThenBy(w => w);

        foreach (var word in sortedWords)
        {
            Console.WriteLine(word);
        }
    }
}
```

Answer Explanation:

This query sorts words first by their length and then alphabetically. It demonstrates the use of `OrderBy` followed by `ThenBy`.

Difficulty Rating: Intermediate

28. Implement a State Management Pattern Using Enums

Problem Statement:
Implement a state machine using enums to manage the states of an object.

C# Implementation:

```
public enum State
{
    Idle,
    Running,
    Paused,
    Stopped
}

public class StateManager
{
    private State _currentState;

    public event EventHandler<StateChangedEventArgs> StateChanged;

    public State CurrentState
    {
        get => _currentState;
        private set
        {
            if (_currentState != value)
            {
                StateChangedEventArgs args = new StateChangedEventArgs { PreviousState = _currentState,
NewState = value };
                OnStateChanged(args);
```

```
                _currentState = value;
            }
        }
    }

    public StateManager()
    {
        CurrentState = State.Idle;
    }

    protected virtual void OnStateChanged(StateChangedEventArgs e)
    {
        StateChanged?.Invoke(this, e);
    }

    public void Start()
    {
        CurrentState = State.Running;
    }

    public void Stop()
    {
        CurrentState = State.Stopped;
    }
}

public class StateChangedEventArgs : EventArgs
{
    public State PreviousState { get; set; }
    public State NewState { get; set; }
}

public class Program
{
    public static void Main()
    {
        StateManager manager = new StateManager();
        manager.StateChanged += (sender, args) =>
        {
            Console.WriteLine($"State changed from {args.PreviousState} to {args.NewState}");
        };

        manager.Start();
        manager.Stop();
    }
}
```

Answer Explanation:

This state machine uses enums to define possible states and a `StateManager` class to transition between them. Events notify observers of state changes.

Difficulty Rating: Intermediate

29. Implement a Command Pattern for Request Handling

Problem Statement:
Implement the command pattern to encapsulate and execute a request.

C# Implementation:

```
public interface ICommand
{
    void Execute();
}

public class AddCommand : ICommand
{
    private int _a;
    private int _b;

    public AddCommand(int a, int b)
    {
        _a = a;
        _b = b;
```

```
    }

    public void Execute()
    {
        Console.WriteLine($"Sum: {_a + _b}");
    }
}

public class CommandInvoker
{
    private ICommand _command;

    public void SetCommand(ICommand command)
    {
        _command = command;
    }

    public void ExecuteCommand()
    {
        if (_command != null)
            _command.Execute();
    }
}

public class Program
{
    public static void Main()
    {
        CommandInvoker invoker = new CommandInvoker();
        ICommand command = new AddCommand(5, 3);

        invoker.SetCommand(command);
        invoker.ExecuteCommand();
    }
}
```

Answer Explanation:

The Command pattern encapsulates a request as an object, allowing it to be executed at any time. `AddCommand` implements `ICommand`, and `CommandInvoker` executes the command.

Difficulty Rating: Intermediate

30. Implement a Chain of Responsibility Pattern

Problem Statement:
Create a chain where each handler processes a request and passes it on if needed.

C# Implementation:

```
public interface IHandler
{
    void SetNext(IHandler handler);
    void HandleRequest(Request request);
}

public class Request
{
    public string Type { get; set; }
    public string Content { get; set; }
}

public abstract class AbstractHandler : IHandler
{
    protected IHandler Next;

    public void SetNext(IHandler handler)
    {
        Next = handler;
    }

    public abstract void HandleRequest(Request request);
}

public class ConcreteHandler1 : AbstractHandler
```

```
{
    public override void HandleRequest(Request request)
    {
        if (request.Type == "Type1")
            Console.WriteLine("ConcreteHandler1 handled the request.");
        else
            Next.HandleRequest(request);
    }
}

public class ConcreteHandler2 : AbstractHandler
{
    public override void HandleRequest(Request request)
    {
        if (request.Type == "Type2")
            Console.WriteLine("ConcreteHandler2 handled the request.");
        else
            Next.HandleRequest(request);
    }
}

public class Program
{
    public static void Main()
    {
        IHandler handler1 = new ConcreteHandler1();
        IHandler handler2 = new ConcreteHandler2();

        handler1.SetNext(handler2);

        Request request1 = new Request { Type = "Type1", Content = "Content" };
        handler1.HandleRequest(request1); // Output: ConcreteHandler1 handled the request.

        Request request2 = new Request { Type = "Type2", Content = "Content" };
        handler1.HandleRequest(request2); // Output: ConcreteHandler2 handled the request.
    }
}
```

Answer Explanation:

This chain of responsibility pattern allows requests to be passed through a series of handlers until one handles it. Each handler decides whether to process the request or forward it.

Difficulty Rating: Intermediate

31. Implement a Strategy Pattern for Payment Methods

Problem Statement:
Create different payment strategies and switch them at runtime.

C# Implementation:

```
public interface IPaymentStrategy
{
    void Pay(decimal amount);
}

public class CreditCardStrategy : IPaymentStrategy
{
    public void Pay(decimal amount)
    {
        Console.WriteLine($"Paid {amount:C} via credit card.");
    }
}

public class PayPalStrategy : IPaymentStrategy
{
    public void Pay(decimal amount)
    {
        Console.WriteLine($"Paid {amount:C} via PayPal.");
    }
}

public class PaymentContext
{
```

```
    private IPaymentStrategy _strategy;

    public void SetPaymentStrategy(IPaymentStrategy strategy)
    {
        _strategy = strategy;
    }

    public void ProcessPayment(decimal amount)
    {
        if (_strategy == null)
            throw new InvalidOperationException("No payment strategy set.");

        _strategy.Pay(amount);
    }
}

public class Program
{
    public static void Main()
    {
        PaymentContext context = new PaymentContext();

        // Use credit card strategy
        context.SetPaymentStrategy(new CreditCardStrategy());
        context.ProcessPayment(100.00m);

        // Use PayPal strategy
        context.SetPaymentStrategy(new PayPalStrategy());
        context.ProcessPayment(50.00m);
    }
}
```

Answer Explanation:

The Strategy pattern allows payment methods to be selected at runtime. `PaymentContext` uses different strategies to process payments.

Difficulty Rating: Intermediate

32. Implement a Composite Pattern for Tree Structures

Problem Statement:
Create a tree structure where nodes can be either leaves or containers.

C# Implementation:

```
public interface INode
{
    void Add(INode node);
    void Remove(INode node);
    void Display(int depth);
}

public class LeafNode : INode
{
    public string Name { get; set; }

    public LeafNode(string name)
    {
        Name = name;
    }

    public void Add(INode node)
    {
        throw new NotSupportedException("Cannot add to a leaf node.");
    }

    public void Remove(INode node)
    {
        throw new NotSupportedException("Cannot remove from a leaf node.");
    }

    public void Display(int depth)
    {
        Console.WriteLine(new string(' ', depth * 2) + Name);
```

```
    }
}

public class CompositeNode : INode
{
    private string _name;
    private List<INode> _children = new List<INode>();

    public CompositeNode(string name)
    {
        _name = name;
    }

    public void Add(INode node)
    {
        _children.Add(node);
    }

    public void Remove(INode node)
    {
        _children.Remove(node);
    }

    public void Display(int depth)
    {
        Console.WriteLine(new string(' ', depth * 2) + _name);
        foreach (INode node in _children)
        {
            node.Display(depth + 1);
        }
    }
}

public class Program
{
    public static void Main()
    {
        INode root = new CompositeNode("Root");

        root.Add(new LeafNode("Leaf1"));
        root.Add(new LeafNode("Leaf2"));

        CompositeNode composite = new CompositeNode("Composite");
        composite.Add(new LeafNode("Leaf3"));
        root.Add(composite);

        root.Display(0);
    }
}
```

Answer Explanation:

This Composite pattern represents a tree structure where `CompositeNode` contains other nodes, and `LeafNode` is a terminal node. The `Display` method recursively prints the tree structure.

Difficulty Rating: Intermediate

33. Implement a Proxy Pattern for Lazy Loading

Problem Statement:
Create a proxy to load an image only when needed.

C# Implementation:

```
public interface IImage
{
    void Display();
}

public class RealImage : IImage
{
    private string _path;

    public RealImage(string path)
    {
```

```
        Load(path);
    }

    private void Load(string path)
    {
        // Simulate loading time
        Console.WriteLine($"Loading image from {path}...");
        System.Threading.Thread.Sleep(2000);
    }

    public void Display()
    {
        Console.WriteLine("Displaying image.");
    }
}

public class ImageProxy : IImage
{
    private string _path;
    private RealImage _realImage;

    public ImageProxy(string path)
    {
        _path = path;
    }

    public void Display()
    {
        if (_realImage == null)
            _realImage = new RealImage(_path);

        _realImage.Display();
    }
}

public class Program
{
    public static void Main()
    {
        IImage proxy = new ImageProxy("image.jpg");

        Console.WriteLine("Proxy created. No image loaded yet.");
        proxy.Display(); // Image is loaded and displayed
    }
}
```

Answer Explanation:

The Proxy pattern delays the loading of an image until it is actually needed. The `ImageProxy` loads the real image only when `Display()` is called.

Difficulty Rating: Intermediate

34. Implement a Flyweight Pattern for Memory Optimization

Problem Statement:
Use the flyweight pattern to reduce memory usage by sharing data.

C# Implementation:

```
public class FlyweightFactory
{
    private Dictionary<string, Circle> _flyweights = new Dictionary<string, Circle>();

    public Circle GetCircle(string color)
    {
        if (!_flyweights.ContainsKey(color))
            _flyweights[color] = new Circle(color);

        return _flyweights[color];
    }
}

public class Circle
{
```

```
    private string _color;

    public Circle(string color)
    {
        _color = color;
    }

    public void Draw(int x, int y, int radius)
    {
        Console.WriteLine($"Drawing {radius} size circle at ({x}, {y}) with color {_color}");
    }
}

public class Program
{
    public static void Main()
    {
        FlyweightFactory factory = new FlyweightFactory();

        Circle circle1 = factory.GetCircle("Red");
        circle1.Draw(0, 0, 5);

        Circle circle2 = factory.GetCircle("Blue");
        circle2.Draw(10, 10, 10);

        Circle circle3 = factory.GetCircle("Red");
        circle3.Draw(5, 5, 7);

        Console.WriteLine($"\nTotal circles created: {factory._flyweights.Count}");
    }
}
```

Answer Explanation:

The Flyweight pattern minimizes memory usage by reusing shared objects. The factory returns existing circles of the same color to reduce object creation.

Difficulty Rating: Intermediate

35. Implement a Template Method Pattern for Common Algorithms

Problem Statement:
Define the skeleton of an algorithm in a base class and let subclasses implement specific steps.

C# Implementation:

```
public abstract class AlgorithmTemplate
{
    public void TemplateMethod()
    {
        Step1();
        Step2();
        Step3();
    }

    protected abstract void Step1();
    protected abstract void Step2();
    protected virtual void Step3()
    {
        Console.WriteLine("Default step 3 implementation.");
    }
}

public class AlgorithmA : AlgorithmTemplate
{
    protected override void Step1()
    {
        Console.WriteLine("Algorithm A step 1.");
    }

    protected override void Step2()
    {
        Console.WriteLine("Algorithm A step 2.");
    }
```

```
    protected override void Step3()
    {
        Console.WriteLine("Algorithm A step 3.");
    }
}

public class AlgorithmB : AlgorithmTemplate
{
    protected override void Step1()
    {
        Console.WriteLine("Algorithm B step 1.");
    }

    protected override void Step2()
    {
        Console.WriteLine("Algorithm B step 2.");
    }
}

public class Program
{
    public static void Main()
    {
        AlgorithmTemplate templateA = new AlgorithmA();
        templateA.TemplateMethod();

        AlgorithmTemplate templateB = new AlgorithmB();
        templateB.TemplateMethod();
    }
}
```

Answer Explanation:

The Template Method pattern defines an algorithm's skeleton in a base class, allowing subclasses to implement specific steps while reusing common structure.

Difficulty Rating: Intermediate

36. Implement the Observer Pattern for Event Subscription

Problem Statement:
Create an observer pattern where observers subscribe to a subject and react to state changes.

C# Implementation:

```
public interface ISubject
{
    void Subscribe(IObserver observer);
    void Unsubscribe(IObserver observer);
    void Notify();
}

public interface IObserver
{
    void Update();
}

public class WeatherData : ISubject
{
    private List<IObserver> _observers = new List<IObserver>();
    private string _weather;

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

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

    public void Notify()
    {
        foreach (IObserver observer in _observers)
```

```
            observer.Update();
    }

    public void SetWeather(string weather)
    {
        _weather = weather;
        Notify();
    }
}

public class WeatherObserver : IObserver
{
    public void Update()
    {
        Console.WriteLine("Weather updated. Current conditions: ...");
    }
}

public class Program
{
    public static void Main()
    {
        ISubject weatherData = new WeatherData();
        IObserver observer1 = new WeatherObserver();
        IObserver observer2 = new WeatherObserver();

        weatherData.Subscribe(observer1);
        weatherData.Subscribe(observer2);

        weatherData.SetWeather("Sunny"); // Observers get updated
    }
}
```

Answer Explanation:

The Observer pattern allows objects (observers) to subscribe to an event source (subject). When the subject changes state, it notifies all observers.

Difficulty Rating: Intermediate

37. Implement a Mediator Pattern to Reduce Coupling

Problem Statement:
Use the mediator pattern to reduce direct communication between objects.

C# Implementation:

```
public interface IMediator
{
    void Send(string message, Colleague colleague);
}

public class Colleague
{
    protected IMediator _mediator;

    public Colleague(IMediator mediator)
    {
        _mediator = mediator;
    }

    public void Send(string message)
    {
        _mediator.Send(message, this);
    }

    public virtual void Receive(string message)
    {
        Console.WriteLine($"Message received by {GetType().Name}: {message}");
    }
}

public class ConcreteMediator : IMediator
{
    private Colleague _colleague1;
```

```
    private Colleague _colleague2;

    public ConcreteMediator(Colleague colleague1, Colleague colleague2)
    {
        _colleague1 = colleague1;
        _colleague2 = colleague2;
    }

    public void Send(string message, Colleague sender)
    {
        if (sender == _colleague1)
            _colleague2.Receive(message);
        else
            _colleague1.Receive(message);
    }
}

public class Program
{
    public static void Main()
    {
        Colleague colleague1 = new Colleague(new ConcreteMediator(null, null));
        Colleague colleague2 = new Colleague(new ConcreteMediator(colleague1, colleague2));

        colleague1.Send("Hello from Colleague 1");
    }
}
```

Answer Explanation:

The Mediator pattern reduces direct coupling between objects by centralizing their communication through a mediator. Each object communicates via the mediator.

Difficulty Rating: Intermediate

38. Implement a Facade Pattern to Simplify Interfaces

Problem Statement:
Create a facade to simplify the interfaces of subsystems.

C# Implementation:

```
public class SubsystemA
{
    public void Operation1() { }
}

public class SubsystemB
{
    public void Operation2() { }
}

public class Facade
{
    private SubsystemA _subSystemA = new SubsystemA();
    private SubsystemB _subSystemB = new SubsystemB();

    public void Method()
    {
        _subSystemA.Operation1();
        _subSystemB.Operation2();
        // Additional logic
    }
}

public class Program
{
    public static void Main()
    {
        Facade facade = new Facade();
        facade.Method(); // Calls all subsystem operations
    }
}
```

Answer Explanation:

The Facade pattern provides a simplified interface to a complex subsystem. `Facade` class encapsulates the interactions with subsystem classes.

Difficulty Rating: Intermediate

39. Implement a Builder Pattern for Complex Object Construction

Problem Statement:
Use the builder pattern to construct complex objects step-by-step.

C# Implementation:

```
public class Car
{
    public string Model { get; set; }
    public int Year { get; set; }
    public string Color { get; set; }

    private Car()
    {
        // Private constructor to enforce use of builder
    }

    public class Builder
    {
        private Car _car = new Car();

        public Builder SetModel(string model)
        {
            _car.Model = model;
            return this;
        }

        public Builder SetYear(int year)
        {
            _car.Year = year;
            return this;
        }

        public Builder SetColor(string color)
        {
            _car.Color = color;
            return this;
        }

        public Car Build()
        {
            return _car;
        }
    }
}

public class Program
{
    public static void Main()
    {
        Car car = new Car.Builder()
            .SetModel("Sedan")
            .SetYear(2023)
            .SetColor("Red")
            .Build();

        Console.WriteLine($"Car: {car.Model}, {car.Year}, {car.Color}");
    }
}
```

Answer Explanation:

The Builder pattern allows constructing complex objects step-by-step through a fluent interface. The `Builder` class provides methods to set properties and build the final object.

Difficulty Rating: Intermediate

40. Implement a Singleton Pattern to Ensure One Instance

Problem Statement:
Ensure only one instance of a class exists and provide global access to it.

C# Implementation:

```
public sealed class Singleton
{
    private static Singleton _instance = null;

    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
                _instance = new Singleton();

            return _instance;
        }
    }

    private Singleton()
    {
        // Private constructor to prevent instantiation
    }

    public void DoSomething()
    {
        Console.WriteLine("Singleton instance is doing something.");
    }
}

public class Program
{
    public static void Main()
    {
        Singleton instance1 = Singleton.Instance;
        Singleton instance2 = Singleton.Instance;

        Console.WriteLine(Object.ReferenceEquals(instance1, instance2)); // True
    }
}
```

Answer Explanation:

The Singleton pattern ensures a class has only one instance and provides a global point of access. The `Instance` property lazily initializes the singleton instance.

Difficulty Rating: Intermediate

41. Implement a Prototype Pattern for Object Cloning

Problem Statement:
Use the prototype pattern to create objects by cloning existing instances.

C# Implementation:

```
public interface IPrototype<T> where T : class, new()
{
    T Clone();
}

public class PrototypeClass : IPrototype<PrototypeClass>
{
    public int Value { get; set; }

    public PrototypeClass Clone()
    {
        return (PrototypeClass)MemberwiseClone();
    }
}

public class Program
{
    public static void Main()
```

```
    {
        PrototypeClass original = new PrototypeClass { Value = 42 };
        PrototypeClass clone = original.Clone();

        Console.WriteLine(original.Value); // 42
        Console.WriteLine(clone.Value);    // 42

        clone.Value = 100;
        Console.WriteLine(clone.Value);    // 100
    }
}
```

Answer Explanation:

The Prototype pattern creates new objects by cloning existing instances. `Clone()` method performs a shallow copy using `MemberwiseClone()`.

Difficulty Rating: Intermediate

42. Implement a Factory Pattern to Create Objects

Problem Statement:
Use the factory pattern to create objects without specifying their exact class.

C# Implementation:

```
public interface IProduct
{
    void Use();
}

public class ProductA : IProduct
{
    public void Use() { Console.WriteLine("Using Product A."); }
}

public class ProductB : IProduct
{
    public void Use() { Console.WriteLine("Using Product B."); }
}

public class Factory
{
    public IProduct Create(string type)
    {
        switch (type)
        {
            case "A":
                return new ProductA();
            case "B":
                return new ProductB();
            default:
                throw new ArgumentException("Invalid product type.");
        }
    }
}

public class Program
{
    public static void Main()
    {
        Factory factory = new Factory();

        IProduct productA = factory.Create("A");
        productA.Use(); // Output: Using Product A.

        IProduct productB = factory.Create("B");
        productB.Use(); // Output: Using Product B.
    }
}
```

Answer Explanation:

The Factory pattern encapsulates object creation. The factory determines which product to create based on input parameters.

Difficulty Rating: Intermediate

43. Implement a Command Line Parser for Custom Commands

Problem Statement:
Create a parser to handle custom command-line arguments.

C# Implementation:

```
using System;
using System.Collections.Generic;

public class CommandLineParser
{
    private Dictionary<string, string> _arguments = new Dictionary<string, string>();

    public void Parse(string[] args)
    {
        for (int i = 0; i < args.Length; i++)
        {
            if (args[i].StartsWith("--"))
            {
                string key = args[i].Substring(2);
                if (i + 1 < args.Length && !args[i + 1].StartsWith("--"))
                {
                    _arguments[key] = args[++i];
                }
                else
                {
                    _arguments[key] = null;
                }
            }
        }
    }

    public string this[string key]
    {
        get
        {
            if (_arguments.ContainsKey(key))
                return _arguments[key];
            else
                throw new KeyNotFoundException();
        }
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        CommandLineParser parser = new CommandLineParser();
        parser.Parse(args);

        try
        {
            string name = parser["name"];
            Console.WriteLine($"Hello, {name}!");
        }
        catch (KeyNotFoundException)
        {
            Console.WriteLine("Name not provided.");
        }
    }
}
```

Answer Explanation:

This command-line parser processes arguments in the form of `--key value`. It stores them in a dictionary and allows access by key.

Difficulty Rating: Intermediate

44. Implement a URL Shortener Service Using Hashing

Problem Statement:
Create a simple URL shortener that maps long URLs to shorter aliases.

C# Implementation:

```
using System;
using System.Collections.Generic;

public class UrlShortener
{
    private Dictionary<string, string> _longToShort = new Dictionary<string, string>();
    private Dictionary<string, string> _shortToLong = new Dictionary<string, string>();

    private readonly char[] _characters =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789".ToCharArray();

    public string Shorten(string longUrl)
    {
        if (_longToShort.ContainsKey(longUrl))
            return _longToShort[longUrl];

        string shortUrl = GenerateShortCode();
        while (_shortToLong.ContainsKey(shortUrl))
            shortUrl = GenerateShortCode();

        _longToShort[longUrl] = shortUrl;
        _shortToLong[shortUrl] = longUrl;

        return $"https://bit.ly/{shortUrl}";
    }

    public string Expand(string shortUrl)
    {
        if (_shortToLong.TryGetValue(shortUrl.Substring(12), out string longUrl))
            return longUrl;
        else
            throw new ArgumentException("Invalid short URL.");
    }

    private string GenerateShortCode()
    {
        Random random = new Random();
        char[] code = new char[6];
        for (int i = 0; i < code.Length; i++)
            code[i] = _characters[random.Next(_characters.Length)];

        return new string(code);
    }
}

public class Program
{
    public static void Main()
    {
        UrlShortener shortener = new UrlShortener();

        string longUrl = "https://www.example.com/very-long-url";
        string shortened = shortener.Shorten(longUrl);
        Console.WriteLine(shortened); // Output: https://bit.ly/...

        string expanded = shortener.Expand(shortened);
        Console.WriteLine(expanded); // Output: https://www.example.com/very-long-url
    }
}
```

Answer Explanation:

The URL shortener service uses two dictionaries to map long URLs to shortened aliases and vice versa. The `GenerateShortCode` method creates random 6-character codes.

Difficulty Rating: Intermediate

45. Implement a Simple Dependency Injection Container

Problem Statement:

Create a basic dependency injection container to resolve dependencies.

C# Implementation:

```
using System;
using System.Collections.Generic;

public interface IService { }

public class Service : IService { }

public class Component
{
    private readonly IService _service;

    public Component(IService service)
    {
        _service = service;
    }

    public void DoWork() { }
}

public class Program
{
    public static void Main()
    {
        // Register dependencies
        var container = new Dictionary<Type, object>();
        container[typeof(IService)] = new Service();

        // Resolve dependencies
        var component = (Component)Activator.CreateInstance(
            typeof(Component),
            container[typeof(IService)]
        );

        component.DoWork();
    }
}
```

Answer Explanation:

This simple dependency injection container uses a dictionary to store service instances. The `Activator.CreateInstance` method resolves dependencies by passing registered services.

Difficulty Rating: Intermediate

46. Implement a Publish/Subscribe Pattern Using Events

Problem Statement:
Create a pub/sub system where publishers emit events and subscribers react.

C# Implementation:

```
public class EventPublisher
{
    public event EventHandler<string> MessagePublished;

    protected virtual void OnMessagePublished(string message)
    {
        MessagePublished?.Invoke(this, message);
    }

    public void Publish(string message)
    {
        OnMessagePublished(message);
    }
}

public class Subscriber
{
    public void Subscribe(EventPublisher publisher)
    {
```

```
        publisher.MessagePublished += HandleMessage;
    }

    private void HandleMessage(object sender, string message)
    {
        Console.WriteLine($"Subscriber received: {message}");
    }
}

public class Program
{
    public static void Main()
    {
        EventPublisher publisher = new EventPublisher();
        Subscriber subscriber1 = new Subscriber();
       Subscriber subscriber2 = new Subscriber();

        subscriber1.Subscribe(publisher);
        subscriber2.Subscribe(publisher);

        publisher.Publish("Hello subscribers!");
    }
}
```

Answer Explanation:

The Publish/Subscribe pattern allows multiple subscribers to react to events published by a central publisher. Each subscriber can handle the event independently.

Difficulty Rating: Intermediate

47. Implement a Token Bucket Algorithm for Rate Limiting

Problem Statement:
Implement rate limiting using the token bucket algorithm.

C# Implementation:

```
using System;
using System.Threading;

public class TokenBucket
{
    private int _capacity;
    private int _currentTokens;
    private DateTime _lastRefill;

    public TokenBucket(int capacity, int refillRatePerSecond)
    {
        _capacity = capacity;
        _currentTokens = 0;
        _lastRefill = DateTime.UtcNow;
    }

    private void Refill()
    {
        int tokensToAdd = (int)(DateTime.UtcNow - _lastRefill).TotalSeconds;
        _currentTokens += tokensToAdd;

        if (_currentTokens > _capacity)
            _currentTokens = _capacity;

        _lastRefill = DateTime.UtcNow;
    }

    public bool Consume(int tokens)
    {
        Refill();

        if (tokens > _currentTokens)
            return false;

        _currentTokens -= tokens;
        return true;
    }
```

```
}

public class Program
{
    public static void Main()
    {
        TokenBucket bucket = new TokenBucket(10, 1); // Max capacity: 10, refill rate: 1 per second

        if (bucket.Consume(5))
            Console.WriteLine("Consumed 5 tokens.");

        Thread.Sleep(1000); // Wait one second
        if (bucket.Consume(6))
            Console.WriteLine("Consumed 6 tokens.");
    }
}
```

Answer Explanation:

The Token Bucket algorithm maintains a count of available tokens, refilling them at intervals. The `Consume` method checks and deducts tokens when available.

Difficulty Rating: Intermediate

48. Implement a Circuit Breaker Pattern for Fault Tolerance

Problem Statement:
Implement the circuit breaker pattern to handle service failures.

C# Implementation:

```
using System;
using System.Threading;

public enum CircuitState { Open, Closed, HalfOpen }

public class CircuitBreaker
{
    private CircuitState _state = CircuitState.Closed;
    private int _failureCount = 0;
    private readonly int _maxFailures;
    private DateTime _lastFailure;

    public CircuitBreaker(int maxFailures, int resetTimeoutSeconds)
    {
        _maxFailures = maxFailures;
        // Store timeout for later use (not implemented in this example)
    }

    public bool IsOpen()
    {
        return _state == CircuitState.Open;
    }

    public void RecordFailure()
    {
        _failureCount++;
        _lastFailure = DateTime.UtcNow;

        if (_state == CircuitState.Closed && _failureCount >= _maxFailures)
            ChangeState(CircuitState.Open);
    }

    public void RecordSuccess()
    {
        _failureCount = 0;

        if (_state == CircuitState.HalfOpen)
            ChangeState(CircuitState.Closed);
    }

    private void ChangeState(CircuitState newState)
    {
        _state = newState;
        if (newState == CircuitState.HalfOpen)
```

```
            ThreadPool.QueueUserWorkItem(_ => ChangeToClosed());
    }

    private void ChangeToClosed()
    {
        if (_state == CircuitState.HalfOpen)
            ChangeState(CircuitState.Closed);
    }
}

public class Program
{
    public static void Main()
    {
        CircuitBreaker breaker = new CircuitBreaker(3, 5);

        // Simulate multiple failures
        breaker.RecordFailure();
        breaker.RecordFailure();
        breaker.RecordFailure();

        if (breaker.IsOpen())
            Console.WriteLine("Circuit is open.");

        // Wait for half-open state (simplified)
        Thread.Sleep(5000);
        breaker.RecordSuccess();

        if (!breaker.IsOpen())
            Console.WriteLine("Circuit is closed again.");
    }
}
```

Answer Explanation:

The Circuit Breaker pattern monitors service failures and trips the circuit to prevent further attempts. It can transition between states (Open, Closed, Half-Open) based on failure thresholds.

Difficulty Rating: Intermediate

49. Implement a Memoization Decorator for Caching

Problem Statement:
Create a decorator to cache method results and avoid recomputation.

C# Implementation:

```
using System;
using System.Collections.Generic;

public interface IMemoizable<TInput, TOutput>
{
    TOutput Compute(TInput input);
}

public class MemoizationDecorator<TInput, TOutput> : IMemoizable<TInput, TOutput>
{
    private readonly IMemoizable<TInput, TOutput> _memoizable;
    private Dictionary<TInput, TOutput> _cache;

    public MemoizationDecorator(IMemoizable<TInput, TOutput> memoizable)
    {
        _memoizable = memoizable;
        _cache = new Dictionary<TInput, TOutput>();
    }

    public TOutput Compute(TInput input)
    {
        if (_cache.TryGetValue(input, out TOutput result))
            return result;

        result = _memoizable.Compute(input);
        _cache[input] = result;
        return result;
    }
```

```
}

public class ExpensiveCalculator : IMemoizable<int, int>
{
    public int Compute(int input)
    {
        // Simulate expensive computation
        Console.WriteLine("Computing...");
        System.Threading.Thread.Sleep(1000);
        return input * 2;
    }
}

public class Program
{
    public static void Main()
    {
        IMemoizable<int, int> calculator = new MemoizationDecorator<int, int>(new ExpensiveCalculator());

        Console.WriteLine(calculator.Compute(5)); // Computed
        Console.WriteLine(calculator.Compute(5)); // Cached
        Console.WriteLine(calculator.Compute(6)); // Computed
    }
}
```

Answer Explanation:

The Memoization Decorator caches the results of expensive computations. If the same input is provided again, it returns the cached result instead of recomputing.

Difficulty Rating: Intermediate

50. Implement a Throttling Mechanism to Limit Request Rate

Problem Statement:
Limit the rate at which requests can be processed.

C# Implementation:

```
using System;
using System.Collections.Generic;

public class RequestThrottler
{
    private int _maxRequestsPerSecond;
    private Queue<DateTime> _requestTimes;

    public RequestThrottler(int maxRequestsPerSecond)
    {
        _maxRequestsPerSecond = maxRequestsPerSecond;
        _requestTimes = new Queue<DateTime>();
    }

    public bool AllowRequest()
    {
        // Remove old requests
        while (_requestTimes.Count > 0 && (DateTime.UtcNow - _requestTimes.Peek()).TotalSeconds >= 1)
            _requestTimes.Dequeue();

        // Check if we can accept the request
        if (_requestTimes.Count < _maxRequestsPerSecond)
        {
            _requestTimes.Enqueue(DateTime.UtcNow);
            return true;
        }

        return false;
    }
}

public class Program
{
    public static void Main()
    {
        RequestThrottler throttler = new RequestThrottler(2); // Maximum 2 requests per second
```

```
        Console.WriteLine(throttler.AllowRequest()); // True
        Console.WriteLine(throttler.AllowRequest()); // True
        Console.WriteLine(throttler.AllowRequest()); // False

        System.Threading.Thread.Sleep(1000); // Wait one second
        Console.WriteLine(throttler.AllowRequest()); // True
    }
}
```

Answer Explanation:

The Request Throttler limits the number of requests that can be processed within a specific time window. It uses a queue to track recent request times and enforces the rate limit.

Difficulty Rating: Intermediate

51. Implement a Strategy Pattern for Sorting Algorithms

Problem Statement:
Allow different sorting strategies to be used interchangeably.

C# Implementation:

```
using System;

public interface ISortStrategy
{
    void Sort(int[] array);
}

public class BubbleSort : ISortStrategy
{
    public void Sort(int[] array)
    {
        for (int i = 0; i < array.Length - 1; i++)
            for (int j = 0; j < array.Length - i - 1; j++)
                if (array[j] > array[j + 1])
                    Swap(ref array[j], ref array[j + 1]);
    }

    private void Swap(ref int a, ref int b)
    {
        int temp = a;
        a = b;
        b = temp;
    }
}

public class QuickSort : ISortStrategy
{
    public void Sort(int[] array)
    {
        QuickSortHelper(array, 0, array.Length - 1);
    }

    private void QuickSortHelper(int[] array, int left, int right)
    {
        if (left < right)
        {
            int pivotIndex = Partition(array, left, right);
            QuickSortHelper(array, left, pivotIndex - 1);
            QuickSortHelper(array, pivotIndex + 1, right);
        }
    }

    private int Partition(int[] array, int left, int right)
    {
        int pivotValue = array[right];
        int i = left - 1;
        for (int j = left; j < right; j++)
            if (array[j] <= pivotValue)
                Swap(ref array[++i], ref array[j]);
        Swap(ref array[i + 1], ref array[right]);
        return i + 1;
```

```
    }

    private void Swap(ref int a, ref int b)
    {
        int temp = a;
        a = b;
        b = temp;
    }
}

public class Sorter
{
    private ISortStrategy _strategy;

    public void SetSortStrategy(ISortStrategy strategy)
    {
        _strategy = strategy;
    }

    public void Sort(int[] array)
    {
        _strategy.Sort(array);
    }
}

public class Program
{
    public static void Main()
    {
        int[] numbers = { 5, 3, 8, 1, 2 };

        Sorter sorter = new Sorter();

        // Use Bubble Sort
        sorter.SetSortStrategy(new BubbleSort());
        sorter.Sort(numbers.Clone() as int[]);
        Console.WriteLine("Bubble Sort: " + string.Join(", ", numbers));

        // Use Quick Sort
        sorter.SetSortStrategy(new QuickSort());
        sorter.Sort(numbers.Clone() as int[]);
        Console.WriteLine("Quick Sort: " + string.Join(", ", numbers));
    }
}
```

Answer Explanation:

The Strategy pattern allows different sorting algorithms to be selected at runtime. The `Sorter` class uses a strategy object to perform the sorting based on the current algorithm.

Difficulty Rating: Intermediate

52. Implement a Chain of Responsibility for Handling Requests

Problem Statement:
Chain handlers to process requests sequentially until one handles it.

C# Implementation:

```
using System;

public abstract class Handler
{
    private Handler _nextHandler;

    public void SetNext(Handler handler)
    {
        _nextHandler = handler;
    }

    public abstract void HandleRequest(Request request);

    protected void PassToNext(Request request)
    {
        if (_nextHandler != null)
```

```
            _nextHandler.HandleRequest(request);
    }
}

public class ConcreteHandler1 : Handler
{
    public override void HandleRequest(Request request)
    {
        if (request.IsHandledByType1())
        {
            Console.WriteLine("ConcreteHandler1 handled the request.");
        }
        else
        {
            PassToNext(request);
        }
    }
}

public class ConcreteHandler2 : Handler
{
    public override void HandleRequest(Request request)
    {
        if (request.IsHandledByType2())
        {
            Console.WriteLine("ConcreteHandler2 handled the request.");
        }
        else
        {
            PassToNext(request);
        }
    }
}

public class Request
{
    private int _type;

    public Request(int type)
    {
        _type = type;
    }

    public bool IsHandledByType1() => _type == 1;
    public bool IsHandledByType2() => _type == 2;
}

public class Program
{
    public static void Main()
    {
        Handler handler1 = new ConcreteHandler1();
        Handler handler2 = new ConcreteHandler2();

        handler1.SetNext(handler2);

        // Request type 1
        handler1.HandleRequest(new Request(1));

        // Request type 2
        handler1.HandleRequest(new Request(2));

        // Request type 3 (no handler)
        handler1.HandleRequest(new Request(3));
    }
}
```

Answer Explanation:

The Chain of Responsibility pattern allows a series of handlers to process requests. Each handler decides whether to handle the request or pass it along.

Difficulty Rating: Intermediate

53. Implement a State Pattern for Object State Management

Problem Statement:
Change an object's behavior when its internal state changes.

C# Implementation:

```
using System;

public interface State
{
    void Handle();
}

public class StateA : State
{
    public void Handle()
    {
        Console.WriteLine("State A handling.");
    }
}

public class StateB : State
{
    public void Handle()
    {
        Console.WriteLine("State B handling.");
    }
}

public class Context
{
    private State _state;

    public void SetState(State state)
    {
        _state = state;
    }

    public void Request()
    {
        _state.Handle();
    }
}

public class Program
{
    public static void Main()
    {
        Context context = new Context();
        context.SetState(new StateA());
        context.Request(); // Output: State A handling.

        context.SetState(new StateB());
        context.Request(); // Output: State B handling.
    }
}
```

Answer Explanation:

The State pattern encapsulates state-specific behavior into separate classes. The context delegates actions to the current state object.

Difficulty Rating: Intermediate

54. Implement a Visitor Pattern for Object Structure Operations

Problem Statement:
Define a new operation on elements of an object structure without changing their classes.

C# Implementation:

```
using System;

public interface IElement { }
```

```
public class ElementA : IElement { }

public class ElementB : IElement { }

public interface IVisitor
{
    void Visit(ElementA element);
    void Visit(ElementB element);
}

public class ConcreteVisitor : IVisitor
{
    public void Visit(ElementA element)
    {
        Console.WriteLine("Visited Element A.");
    }

    public void Visit(ElementB element)
    {
        Console.WriteLine("Visited Element B.");
    }
}

public class ObjectStructure
{
    private IElement[] _elements = new IElement[0];

    public void Add(IElement element)
    {
        var temp = _elements;
        _elements = new IElement[temp.Length + 1];
        Array.Copy(temp, _elements, temp.Length);
        _elements[temp.Length] = element;
    }

    public void Accept(IVisitor visitor)
    {
        foreach (IElement element in _elements)
        {
            if (element is ElementA a)
                visitor.Visit(a);
            else if (element is ElementB b)
                visitor.Visit(b);
        }
    }
}

public class Program
{
    public static void Main()
    {
        ObjectStructure structure = new ObjectStructure();
        structure.Add(new ElementA());
        structure.Add(new ElementB());

        IVisitor visitor = new ConcreteVisitor();
        structure.Accept(visitor);
    }
}
```

Answer Explanation:

The Visitor pattern allows adding new operations to elements of an object structure without modifying their classes. The visitor class defines methods for each element type.

Difficulty Rating: Intermediate

55. Implement a Proxy Pattern to Control Access

Problem Statement:
Provide a surrogate or placeholder for another object.

C# Implementation:

```
using System;
```

```
public interface IService { void Operation(); }

public class RealService : IService
{
    public void Operation()
    {
        Console.WriteLine("Real service operation.");
    }
}

public class ProxyService : IService
{
    private RealService _realService = new RealService();
    private bool _cacheValid = false;
    private string _cachedResult;

    public void Operation()
    {
        if (!_cacheValid)
        {
            _realService.Operation();
            _cachedResult = "Cached result.";
            _cacheValid = true;
        }
        else
        {
            Console.WriteLine(_cachedResult);
        }
    }
}

public class Program
{
    public static void Main()
    {
        IService service = new ProxyService();
        service.Operation(); // Real service called
        service.Operation(); // Cached result used
    }
}
```

Answer Explanation:

The Proxy pattern provides a substitute for another object. In this case, the proxy caches results to avoid repeated calls to the real service.

Difficulty Rating: Intermediate

56. Implement a Bridge Pattern to Decouple Abstraction from Implementation

Problem Statement:
Decouple an abstraction from its implementation so they can vary independently.

C# Implementation:

```
using System;

public interface Implementor { void Operation(); }

public class ConcreteImplementorA : Implementor
{
    public void Operation() { Console.WriteLine("Concrete Implementor A operation."); }
}

public class ConcreteImplementorB : Implementor
{
    public void Operation() { Console.WriteLine("Concrete Implementor B operation."); }
}

public abstract class Abstraction
{
    protected Implementor _implementor;

    public void SetImplementor(Implementor implementor)
```

```
    {
        _implementor = implementor;
    }

    public abstract void Operation();
}

public class RefinedAbstraction : Abstraction
{
    public override void Operation()
    {
        _implementor.Operation();
    }
}

public class Program
{
    public static void Main()
    {
        RefinedAbstraction abstraction = new RefinedAbstraction();

        abstraction.SetImplementor(new ConcreteImplementorA());
        abstraction.Operation();

        abstraction.SetImplementor(new ConcreteImplementorB());
        abstraction.Operation();
    }
}
```

Answer Explanation:

The Bridge pattern decouples an abstraction from its implementation. The `RefinedAbstraction` class uses the current implementor to perform operations, allowing independent variation.

Difficulty Rating: Intermediate

57. Implement a Chainable Builder for Flexible Object Construction

Problem Statement:
Allow building objects with multiple configurations using a fluent interface.

C# Implementation:

```
public class Car
{
    public string Model { get; private set; }
    public int Year { get; private set; }
    public string Color { get; private set; }

    private Car() { }

    public class Builder
    {
        private Car _car = new Car();

        public Builder SetModel(string model)
        {
            _car.Model = model;
            return this;
        }

        public Builder SetYear(int year)
        {
            _car.Year = year;
            return this;
        }

        public Builder SetColor(string color)
        {
            _car.Color = color;
            return this;
        }

        public Car Build()
        {
```

```
            if (string.IsNullOrEmpty(_car.Model))
                throw new InvalidOperationException("Model must be set.");

            return _car;
        }
    }
}

public class Program
{
    public static void Main()
    {
        Car car = new Car.Builder()
            .SetModel("Sedan")
            .SetYear(2023)
            .SetColor("Red")
            .Build();

        Console.WriteLine($"Car: {car.Model}, {car.Year}, {car.Color}");
    }
}
```

Answer Explanation:

The Chainable Builder allows setting object properties in a fluent manner. Each method returns the builder instance, enabling method chaining.

Difficulty Rating: Intermediate

58. Implement a Composite Pattern for Tree Structures

Problem Statement:
Compose objects into tree structures and treat them uniformly.

C# Implementation:

```
using System.Collections.Generic;

public interface IComponent { void Display(); }

public class Composite : IComponent
{
    private List<IComponent> _components = new List<IComponent>();

    public void Add(IComponent component)
    {
        _components.Add(component);
    }

    public void Remove(IComponent component)
    {
        _components.Remove(component);
    }

    public void Display()
    {
        foreach (IComponent component in _components)
            component.Display();
    }
}

public class Leaf : IComponent
{
    public void Display() { Console.WriteLine("Leaf node display."); }
}

public class Program
{
    public static void Main()
    {
        Composite root = new Composite();
        Leaf leaf1 = new Leaf();
        Leaf leaf2 = new Leaf();
        Composite composite = new Composite();
```

```
        root.Add(leaf1);
        root.Add(composite);

        composite.Add(leaf2);

        root.Display();
    }
}
```

Answer Explanation:

The Composite pattern allows treating individual objects and compositions uniformly. The composite node contains child components, which can be either leaves or other composites.

Difficulty Rating: Intermediate

59. Implement a Decorator Pattern to Add Responsibilities Dynamically

Problem Statement:
Add responsibilities to objects dynamically by wrapping them in decorator classes.

C# Implementation:

```
using System;

public interface IComponent { void Operation(); }

public class ConcreteComponent : IComponent
{
    public void Operation() { Console.WriteLine("Concrete component operation."); }
}

public abstract class Decorator : IComponent
{
    protected IComponent _component;

    public Decorator(IComponent component)
    {
        _component = component;
    }

    public void Operation()
    {
        _component.Operation();
    }
}

public class ConcreteDecorator : Decorator
{
    public ConcreteDecorator(IComponent component) : base(component) { }

    public override void Operation()
    {
        base.Operation();
        Console.WriteLine("Concrete decorator added functionality.");
    }
}

public class Program
{
    public static void Main()
    {
        IComponent component = new ConcreteDecorator(new ConcreteComponent());
        component.Operation(); // Output: Component and decorator
    }
}
```

Answer Explanation:

The Decorator pattern dynamically adds responsibilities to objects. The decorator class wraps the component and enhances its functionality.

Difficulty Rating: Intermediate

60. Implement a Flyweight Pattern to Reduce Object Creation Overhead

Problem Statement:
Minimize memory usage by sharing as much data as possible among similar objects.

C# Implementation:

```
using System.Collections.Generic;

public interface IFlyweight { void Operation(string extrinsicState); }

public class FlyweightFactory
{
    private Dictionary<string, IFlyweight> _flyweights = new Dictionary<string, IFlyweight>();

    public FlyweightFactory()
    {
        _flyweights["X"] = new ConcreteFlyweight();
        _flyweights["Y"] = new ConcreteFlyweight();
    }

    public IFlyweight GetFlyweight(string key)
    {
        return _flyweights[key];
    }
}

public class ConcreteFlyweight : IFlyweight
{
    public void Operation(string extrinsicState)
    {
        Console.WriteLine($"ConcreteFlyweight operation with state: {extrinsicState}");
    }
}

public class Program
{
    public static void Main()
    {
        FlyweightFactory factory = new FlyweightFactory();

        IFlyweight flyweight1 = factory.GetFlyweight("X");
        flyweight1.Operation("State A");

        IFlyweight flyweight2 = factory.GetFlyweight("X");
        flyweight2.Operation("State B");

        IFlyweight flyweight3 = factory.GetFlyweight("Y");
        flyweight3.Operation("State C");
    }
}
```

Answer Explanation:

The Flyweight pattern reduces memory usage by sharing intrinsic state among objects. The factory manages and reuses flyweight instances based on keys.

Difficulty Rating: Intermediate

61. Implement a Interpreter Pattern to Evaluate Expressions

Problem Statement:
Define an interface for interpreting expressions and implement them.

C# Implementation:

```
using System;

public interface IExpression { int Interpret(); }

public class Number : IExpression
{
    private int _number;
```

```
    public Number(int number) { _number = number; }

    public int Interpret() => _number;
}

public class Add : IExpression
{
    private IExpression _left, _right;

    public Add(IExpression left, IExpression right)
    {
        _left = left;
        _right = right;
    }

    public int Interpret() => _left.Interpret() + _right.Interpret();
}

public class Multiply : IExpression
{
    private IExpression _left, _right;

    public Multiply(IExpression left, IExpression right)
    {
        _left = left;
        _right = right;
    }

    public int Interpret() => _left.Interpret() * _right.Interpret();
}

public class Program
{
    public static void Main()
    {
        IExpression expr = new Multiply(
            new Add(new Number(5), new Number(3)),
            new Number(2)
        );

        Console.WriteLine(expr.Interpret()); // (5+3)*2 = 16
    }
}
```

Answer Explanation:

The Interpreter pattern defines a language grammar and interprets sentences in that language. Each expression is an abstract syntax tree node.

Difficulty Rating: Intermediate

62. Implement a Memento Pattern to Save and Restore State

Problem Statement:
Capture an object's internal state to restore it later.

C# Implementation:

```
using System;

public class Originator
{
    private string _state;

    public void SetState(string state)
    {
        Console.WriteLine($"Originator: State set to {state}.");
        _state = state;
    }

    public IMemento SaveState()
    {
        return new Memento(_state);
    }
```

```
    public void RestoreState(IMemento memento)
    {
        _state = memento.State;
        Console.WriteLine($"Originator: State restored to {memento.State}.");
    }

    public interface IMemento
    {
        string State { get; }
    }

    private class Memento : IMemento
    {
        public string State { get; }

        public Memento(string state)
        {
            State = state;
        }
    }
}

public class Caretaker
{
    private Originator. IMemento _memento;

    public void SetMemento(Originator.IMemento memento)
    {
        _memento = memento;
    }

    public Originator.IMemento GetMemento()
    {
        return _memento;
    }
}

public class Program
{
    public static void Main()
    {
        Originator originator = new Originator();
        Caretaker caretaker = new Caretaker();

        originator.SetState("State 1");
        caretaker.SetMemento(originator.SaveState());

        originator.SetState("State 2");
        originator.RestoreState(caretaker.GetMemento());
    }
}
```

Answer Explanation:

The Memento pattern allows saving an object's state and restoring it later. The caretaker holds the memento, which contains the saved state.

Difficulty Rating: Intermediate

63. Implement a State Machine for Finite Automata

Problem Statement:
Create a state machine to model finite automata transitions.

C# Implementation:

```
using System;

public interface IState { }

public class StateA : IState { public override string ToString() => "State A"; }
public class StateB : IState { public override string ToString() => "State B"; }

public enum Transition
```

```
{
    AtoB,
    BtoA,
    None
}

public class StateMachine
{
    private IState _currentState;

    public StateMachine(IState initialState)
    {
        _currentState = initialState;
    }

    public IState CurrentState => _currentState;

    public Transition HandleInput(object input)
    {
        if (_currentState is StateA && input != null)
        {
            _currentState = new StateB();
            return Transition.AtoB;
        }
        else if (_currentState is StateB && input == null)
        {
            _currentState = new StateA();
            return Transition.BtoA;
        }

        return Transition.None;
    }
}

public class Program
{
    public static void Main()
    {
        StateMachine machine = new StateMachine(new StateA());

        Console.WriteLine($"Current state: {machine.CurrentState}");

        machine.HandleInput("something");
        Console.WriteLine($"Current state after transition: {machine.CurrentState}");

        machine.HandleInput(null);
        Console.WriteLine($"Current state after transition: {machine.CurrentState}");
    }
}
```

Answer Explanation:

The State Machine pattern models state transitions based on input. The machine changes its current state according to defined rules.

Difficulty Rating: Intermediate

64. Implement a Command Pattern for Encapsulating Requests

Problem Statement:
Encapsulate a request as an object, allowing logging and queuing.

C# Implementation:

```
using System;

public interface ICommand { void Execute(); }

public class Light
{
    public void TurnOn() => Console.WriteLine("Light turned on.");
    public void TurnOff() => Console.WriteLine("Light turned off.");
}

public class LightOnCommand : ICommand
```

```
{
    private Light _light;

    public LightOnCommand(Light light)
    {
        _light = light;
    }

    public void Execute() => _light.TurnOn();
}

public class LightOffCommand : ICommand
{
    private Light _light;

    public LightOffCommand(Light light)
    {
        _light = light;
    }

    public void Execute() => _light.TurnOff();
}

public class RemoteControl
{
    private ICommand _command;

    public void SetCommand(ICommand command)
    {
        _command = command;
    }

    public void PressButton()
    {
        if (_command != null)
            _command.Execute();
    }
}

public class Program
{
    public static void Main()
    {
        Light light = new Light();
        RemoteControl remote = new RemoteControl();

        remote.SetCommand(new LightOnCommand(light));
        remote.PressButton(); // Output: Light turned on.

        remote.SetCommand(new LightOffCommand(light));
        remote.PressButton(); // Output: Light turned off.
    }
}
```

Answer Explanation:

The Command pattern encapsulates requests as objects, allowing actions to be logged, queued, or executed remotely.

Difficulty Rating: Intermediate

65. Implement a Observer Pattern for Event Subscription and Publication

Problem Statement:
Define a dependency between objects where changes to one object notify others.

C# Implementation:

```
using System;
using System.Collections.Generic;

public interface IObserver { void Update(string message); }

public class Subject
{
    private List<IObserver> _observers = new List<IObserver>();
```

```
    public void Attach(IObserver observer)
    {
        _observers.Add(observer);
    }

    public void Detach(IObserver observer)
    {
        _observers.Remove(observer);
    }

    public void Notify(string message)
    {
        foreach (IObserver observer in _observers)
            observer.Update(message);
    }
}

public class ConcreteObserver : IObserver
{
    public void Update(string message)
    {
        Console.WriteLine($"Observer received: {message}");
    }
}

public class Program
{
    public static void Main()
    {
        Subject subject = new Subject();
        IObserver observer1 = new ConcreteObserver();
        IObserver observer2 = new ConcreteObserver();

        subject.Attach(observer1);
        subject.Attach(observer2);

        subject.Notify("Hello observers!");

        subject.Detach(observer1);
        subject.Notify("Another message.");
    }
}
```

Answer Explanation:

The Observer pattern allows objects to subscribe to and receive updates from a subject. Observers are notified when the subject changes.

Difficulty Rating: Intermediate

66. Implement a Template Method Pattern for Algorithm Skeletons

Problem Statement:
Define the skeleton of an algorithm in a base class, deferring some steps to subclasses.

C# Implementation:

```
public abstract class AbstractClass
{
    public void TemplateMethod()
    {
        BaseOperation();
        Hook();
        DerivedOperation();
    }

    protected virtual void BaseOperation() { Console.WriteLine("Abstract class performing base operation."); }
    protected abstract void DerivedOperation();

    protected virtual void Hook() { Console.WriteLine("Abstract class performing hook operation."); }
}

public class ConcreteClass : AbstractClass
{
```

```
    protected override void DerivedOperation() => Console.WriteLine("Concrete class performing derived operation.");

    protected override void Hook() { }
}

public class Program
{
    public static void Main()
    {
        AbstractClass instance = new ConcreteClass();
        instance.TemplateMethod();
    }
}
```

Answer Explanation:

The Template Method pattern defines an algorithm skeleton in a base class, allowing subclasses to override specific steps while maintaining the overall structure.

Difficulty Rating: Intermediate

67. Implement a Factory Pattern for Object Creation Abstraction

Problem Statement:
Provide an interface for creating objects without specifying their concrete class.

C# Implementation:

```
using System;

public interface IProduct { void Operation(); }

public class ProductA : IProduct
{
    public void Operation() => Console.WriteLine("Product A operation.");
}

public class ProductB : IProduct
{
    public void Operation() => Console.WriteLine("Product B operation.");
}

public interface IFactory { IProduct CreateProduct(); }

public class FactoryA : IFactory
{
    public IProduct CreateProduct() => new ProductA();
}

public class FactoryB : IFactory
{
    public IProduct CreateProduct() => new ProductB();
}

public class Program
{
    public static void Main()
    {
        IFactory factory = new FactoryA();
        IProduct product = factory.CreateProduct();
        product.Operation();

        factory = new FactoryB();
        product = factory.CreateProduct();
        product.Operation();
    }
}
```

Answer Explanation:

The Factory pattern abstracts object creation. The factory interface creates products, which are then used without knowledge of their concrete type.

Difficulty Rating: Intermediate

68. Implement a Abstract Factory for Multiple Product Families

Problem Statement:
Provide an interface for creating families of related objects.

C# Implementation:

```
using System;

public interface IAbstractFactory { IProductA CreateProductA(); IProductB CreateProductB(); }

public interface IProductA { void OperationA(); }
public interface IProductB { void OperationB(); }

public class ProductA1 : IProductA
{
    public void OperationA() => Console.WriteLine("Product A1 operation.");
}

public class ProductB1 : IProductB
{
    public void OperationB() => Console.WriteLine("Product B1 operation.");
}

public class ProductA2 : IProductA
{
    public void OperationA() => Console.WriteLine("Product A2 operation.");
}

public class ProductB2 : IProductB
{
    public void OperationB() => Console.WriteLine("Product B2 operation.");
}

public class Factory1 : IAbstractFactory
{
    public IProductA CreateProductA() => new ProductA1();
    public IProductB CreateProductB() => new ProductB1();
}

public class Factory2 : IAbstractFactory
{
    public IProductA CreateProductA() => new ProductA2();
    public IProductB CreateProductB() => new ProductB2();
}

public class Program
{
    public static void Main()
    {
        IAbstractFactory factory = new Factory1();
        IProductA a = factory.CreateProductA();
        a.OperationA();

        IProductB b = factory.CreateProductB();
        b.OperationB();

        factory = new Factory2();
        a = factory.CreateProductA();
        a.OperationA();

        b = factory.CreateProductB();
        b.OperationB();
    }
}
```

Answer Explanation:

The Abstract Factory pattern creates families of related objects. Each factory produces a set of products that belong together.

Difficulty Rating: Intermediate

69. Implement a Builder Pattern for Complex Object Construction

Problem Statement:
Separate the construction of a complex object from its representation.

C# Implementation:

```
using System;

public interface IBuilder { void BuildPartA(); void BuildPartB(); Product GetProduct(); }

public class Product
{
    private string _partA;
    private string _partB;

    public void SetPartA(string part) => _partA = part;
    public void SetPartB(string part) => _partB = part;

    public override string ToString() => $"Product: PartA={_partA}, PartB={_partB}";
}

public class ConcreteBuilder : IBuilder
{
    private Product _product;

    public void BuildPartA()
    {
        _product = new Product();
        _product.SetPartA("Part A");
    }

    public void BuildPartB() => _product.SetPartB("Part B");

    public Product GetProduct() => _product;
}

public class Director
{
    private IBuilder _builder;

    public void SetBuilder(IBuilder builder) => _builder = builder;

    public Product Construct()
    {
        _builder.BuildPartA();
        _builder.BuildPartB();
        return _builder.GetProduct();
    }
}

public class Program
{
    public static void Main()
    {
        Director director = new Director();
        IBuilder builder = new ConcreteBuilder();

        director.SetBuilder(builder);
        Product product = director.Construct();
        Console.WriteLine(product.ToString());
    }
}
```

Answer Explanation:

The Builder pattern separates the construction of a complex object from its representation. The director orchestrates the builder to construct parts step by step.

Difficulty Rating: Intermediate

70. Implement a Singleton Pattern for Single Instance Control

Problem Statement:

Ensure only one instance of a class exists and provide global access.

C# Implementation:

```
using System;

public sealed class Singleton
{
    private static Singleton _instance = new Singleton();
    private Singleton() { }

    public static Singleton Instance => _instance;

    public void DoSomething()
    {
        Console.WriteLine("Singleton instance doing something.");
    }
}

public class Program
{
    public static void Main()
    {
        Singleton singleton = Singleton.Instance;
        singleton.DoSomething();

        // Attempting to create a new instance will fail due to sealed class
    }
}
```

Answer Explanation:

The Singleton pattern ensures only one instance of a class exists. The class is sealed to prevent inheritance, and the constructor is private.

Difficulty Rating: Intermediate

71. Implement a Multiton Pattern for Limited Instance Control

Problem Statement:
Limit the number of instances to more than one but fewer than unlimited.

C# Implementation:

```
using System;
using System.Collections.Generic;

public sealed class Multiton<T>
{
    private static readonly Dictionary<string, T> _instances = new Dictionary<string, T>();
    private static object _lock = new object();

    private Multiton() { }

    public static T GetInstance(string key, Func<T> factory)
    {
        lock (_lock)
        {
            if (!_instances.ContainsKey(key))
                _instances[key] = factory();
        }
        return _.instances[key];
    }

    public static void RemoveInstance(string key)
    {
        lock (_lock)
        {
            _instances.Remove(key);
        }
    }

    public static void Clear()
    {
```

```
        lock (_lock)
        {
            _instances.Clear();
        }
    }

    public static int Count => _instances.Count;
}

public class MyClass
{
    private readonly string _name;

    public MyClass(string name)
    {
        _name = name;
    }

    public override string ToString() => _name;
}

public class Program
{
    public static void Main()
    {
        var instance1 = Multiton<MyClass>.GetInstance("Key1", () => new MyClass("Instance 1"));
        var instance2 = Multiton<MyClass>.GetInstance("Key2", () => new MyClass("Instance 2"));
        var instance3 = Multiton<MyClass>.GetInstance("Key1", () => new MyClass("Duplicate"));

        Console.WriteLine(instance1.ToString()); // Output: Instance 1
        Console.WriteLine(instance3.ToString()); // Output: Instance 1 (same as instance1)
    }
}
```

Answer Explanation:

The Multiton pattern allows control over the number of instances, each identified by a key. Instances are created using a factory function and stored in a dictionary.

Difficulty Rating: Intermediate

72. Implement a Registry Pattern for Object Registration and Lookup

Problem Statement:
Maintain a collection of objects, indexed by keys for easy access.

C# Implementation:

```
using System;
using System.Collections.Generic;

public class Registry<T>
{
    private Dictionary<string, T> _entries = new Dictionary<string, T>();

    public void Register(string key, T entry)
    {
        if (_entries.ContainsKey(key))
            throw new ArgumentException("Key already exists.");

        _entries[key] = entry;
    }

    public T Lookup(string key)
    {
        if (!_entries.ContainsKey(key))
            throw new KeyNotFoundException("Key not found.");

        return _entries[key];
    }

    public bool Contains(string key) => _entries.ContainsKey(key);
}

public class Program
```

```
{
    public static void Main()
    {
        Registry<string> registry = new Registry<string>();

        registry.Register("Key1", "Value1");
        registry.Register("Key2", "Value2");

        Console.WriteLine(registry.Lookup("Key1")); // Output: Value1

        if (registry.Contains("Key3"))
            Console.WriteLine(registry.Lookup("Key3"));
        else
            Console.WriteLine("Key3 not found.");
    }
}
```

Answer Explanation:

The Registry pattern maintains a collection of objects, allowing registration by key and lookup based on the same keys.

Difficulty Rating: Intermediate

```
▯
```