Exercise 1: Implementing the Adapter Pattern

Description: Create an adapter that allows incompatible interfaces to work together.

Task: Given a legacy `PaymentGateway` class with a `ProcessOldPayment` method, write an adapter that allows it to work with a new `IPayment` interface with a `ProcessPayment` method.

Solution:

```csharp
public interface IPayment
{
    void ProcessPayment();
}

public class PaymentGateway
{
    public void ProcessOldPayment()
    {
        Console.WriteLine("Processing payment using the old gateway.");
    }
}

public class PaymentAdapter : IPayment
{
    private PaymentGateway _gateway;

    public PaymentAdapter(PaymentGateway gateway)
    {
        _gateway = gateway;
    }

    public void ProcessPayment()
    {
        _gateway.ProcessOldPayment();
```

```
    }
}

// Usage
var oldGateway = new PaymentGateway();
IPayment payment = new PaymentAdapter(oldGateway);
payment.ProcessPayment();
```

Exercise 2: Implementing the Bridge Pattern

Description: Decouple an abstraction from its implementation.

Task: Create a `Shape` abstraction that can be drawn with different `IRenderer` implementations (e.g., `VectorRenderer`, `RasterRenderer`).

Solution:

```csharp
public interface IRenderer
{
    void RenderShape(string shape);
}

public class VectorRenderer : IRenderer
{
    public void RenderShape(string shape)
    {
        Console.WriteLine($"Rendering {shape} as vectors.");
    }
}

public class RasterRenderer : IRenderer
{
    public void RenderShape(string shape)
```

```
    {
        Console.WriteLine($"Rendering {shape} as pixels.");
    }
}

public abstract class Shape
{
    protected IRenderer renderer;

    protected Shape(IRenderer renderer)
    {
        this.renderer = renderer;
    }

    public abstract void Draw();
}

public class Circle : Shape
{
    public Circle(IRenderer renderer) : base(renderer) { }

    public override void Draw()
    {
        renderer.RenderShape("Circle");
    }
}

// Usage
IRenderer vectorRenderer = new VectorRenderer();
Shape circle = new Circle(vectorRenderer);
circle.Draw();
```

```

### Exercise 3: Implementing the Composite Pattern

**Description**: Treat individual objects and compositions of objects uniformly.

**Task**: Implement a `FileComponent` interface with `File` and `Directory` classes that allow files and directories to be treated uniformly.

**Solution**:

```csharp
public interface IFileComponent
{
    void Display();
}

public class File : IFileComponent
{
    private string _name;

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

    public void Display()
    {
        Console.WriteLine(_name);
    }
}

public class Directory : IFileComponent
{
    private string _name;
```
```

```
    private List<IFileComponent> _components = new List<IFileComponent>();

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

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

    public void Display()
    {
        Console.WriteLine(_name);
        foreach (var component in _components)
        {
            component.Display();
        }
    }
}

// Usage
Directory root = new Directory("root");
root.Add(new File("file1.txt"));
Directory subDir = new Directory("subdir");
subDir.Add(new File("file2.txt"));
root.Add(subDir);
root.Display();
```

Exercise 4: Implementing the Decorator Pattern

Description: Dynamically add responsibilities to an object.

Task: Create a `TextMessage` class and add decorators like `EncryptedMessage` and `CompressedMessage` to modify its behavior.

Solution:

```csharp
public interface IMessage
{
    string GetContent();
}

public class TextMessage : IMessage
{
    private string _text;

    public TextMessage(string text)
    {
        _text = text;
    }

    public string GetContent()
    {
        return _text;
    }
}

public class EncryptedMessage : IMessage
{
    private IMessage _message;

    public EncryptedMessage(IMessage message)
```

```
    {
        _message = message;
    }

    public string GetContent()
    {
        return "Encrypted: " + _message.GetContent();
    }
}

public class CompressedMessage : IMessage
{
    private IMessage _message;

    public CompressedMessage(IMessage message)
    {
        _message = message;
    }

    public string GetContent()
    {
        return "Compressed: " + _message.GetContent();
    }
}

// Usage
IMessage message = new TextMessage("Hello, World!");
message = new EncryptedMessage(message);
message = new CompressedMessage(message);
Console.WriteLine(message.GetContent());
```

Exercise 5: Implementing the Facade Pattern

Description: Simplify the interface to a complex subsystem.

Task: Create a `Computer` facade that simplifies the interaction with `CPU`, `Memory`, and `HardDrive` classes.

Solution:

```csharp
public class CPU
{
    public void Freeze() { Console.WriteLine("CPU Freeze"); }
    public void Jump(long position) { Console.WriteLine($"CPU Jump to {position}"); }
    public void Execute() { Console.WriteLine("CPU Execute"); }
}

public class Memory
{
    public void Load(long position, byte[] data) { Console.WriteLine("Memory Load"); }
}

public class HardDrive
{
    public byte[] Read(long lba, int size) { return new byte[size]; }
}

public class Computer
{
    private CPU cpu;
    private Memory memory;
    private HardDrive hardDrive;

    public Computer()
```

```
    {
        cpu = new CPU();
        memory = new Memory();
        hardDrive = new HardDrive();
    }

    public void Start()
    {
        cpu.Freeze();
        memory.Load(0, hardDrive.Read(0, 1024));
        cpu.Jump(0);
        cpu.Execute();
    }
}

// Usage
Computer computer = new Computer();
computer.Start();
```

Exercise 6: Implementing the Flyweight Pattern

Description: Reduce memory usage by sharing objects.

Task: Implement a `Tree` class that shares common `TreeType` objects to reduce memory usage in a forest.

Solution:

```csharp
public class TreeType
{
    public string Name { get; }
    public string Color { get; }
    public string Texture { get; }
```

```
    public TreeType(string name, string color, string texture)
    {
        Name = name;
        Color = color;
        Texture = texture;
    }

    public void Draw(int x, int y)
    {
        Console.WriteLine($"Drawing a {Color} {Name} at ({x}, {y}) with {Texture} texture.");
    }
}

public class TreeFactory
{
    private static Dictionary<string, TreeType> _treeTypes = new Dictionary<string, TreeType>();

    public static TreeType GetTreeType(string name, string color, string texture)
    {
        var key = $"{name}-{color}-{texture}";
        if (!_treeTypes.ContainsKey(key))
        {
            _treeTypes[key] = new TreeType(name, color, texture);
        }
        return _treeTypes[key];
    }
}

public class Tree
{
```

```
    private int x, y;
    private TreeType type;

    public Tree(int x, int y, TreeType type)
    {
        this.x = x;
        this.y = y;
        this.type = type;
    }

    public void Draw()
    {
        type.Draw(x, y);
    }
}

// Usage
TreeType oak = TreeFactory.GetTreeType("Oak", "Green", "Rough");
Tree tree1 = new Tree(0, 0, oak);
Tree tree2 = new Tree(1, 1, oak);
tree1.Draw();
tree2.Draw();
```

Exercise 7: Implementing the Proxy Pattern

Description: Provide a surrogate or placeholder for another object.

Task: Create a `ProxyImage` class that delays the loading of a `RealImage` until it is actually needed.

Solution:

```csharp
public interface IImage
```

```
{
    void Display();
}

public class RealImage : IImage
{
    private string _filename;

    public RealImage(string filename)
    {
        _filename = filename;
        LoadImageFromDisk();
    }

    private void LoadImageFromDisk()
    {
        Console.WriteLine("Loading " + _filename);
    }

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

public class ProxyImage : IImage
{
    private RealImage _realImage;
    private string _filename;

    public ProxyImage(string filename)
```

```
    {
        _filename = filename;
    }

    public void Display()
    {
        if (_realImage == null)
        {
            _realImage = new RealImage(_filename);
        }
        _realImage.Display();
    }
}

// Usage
IImage image = new ProxyImage("photo.jpg");
image.Display(); // Loading and displaying the image
image.Display(); // Only displaying the image
```

Exercise 8: Implementing an Advanced Composite Pattern

Description: Extend the Composite Pattern to handle additional operations.

Task: Add a `GetSize()` method to the `FileComponent` interface and implement it in `File` and `Directory` classes.

Solution:

```csharp
public interface IFileComponent
{
    void Display();
    long GetSize();
}
```

```
public class File : IFileComponent
{
    private string _name;
    private

 long _size;

    public File(string name, long size)
    {
        _name = name;
        _size = size;
    }

    public void Display()
    {
        Console.WriteLine($"{_name} ({_size} bytes)");
    }

    public long GetSize()
    {
        return _size;
    }
}

public class Directory : IFileComponent
{
    private string _name;
    private List<IFileComponent> _components = new List<IFileComponent>();

    public Directory(string name)
```

```
    {
        _name = name;
    }

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

    public void Display()
    {
        Console.WriteLine(_name);
        foreach (var component in _components)
        {
            component.Display();
        }
    }

    public long GetSize()
    {
        return _components.Sum(c => c.GetSize());
    }
}

// Usage
Directory root = new Directory("root");
root.Add(new File("file1.txt", 500));
Directory subDir = new Directory("subdir");
subDir.Add(new File("file2.txt", 1000));
root.Add(subDir);
root.Display();
```

```
Console.WriteLine($"Total Size: {root.GetSize()} bytes");
```

Exercise 9: Implementing a Logger Using the Decorator Pattern

Description: Use the Decorator Pattern to add logging capabilities to a simple application.

Task: Create a basic `ILogger` interface and implement a `FileLogger` and `DatabaseLogger`. Use decorators to add timestamping and error-level filtering.

Solution:

```csharp
public interface ILogger
{
    void Log(string message);
}

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

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

public class TimestampLogger : ILogger
```

```
{
    private ILogger _logger;

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

    public void Log(string message)
    {
        _logger.Log($"[{DateTime.Now}] {message}");
    }
}

public class ErrorLevelLogger : ILogger
{
    private ILogger _logger;

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

    public void Log(string message)
    {
        _logger.Log($"[ERROR] {message}");
    }
}

// Usage
ILogger logger = new FileLogger();
```

```
logger = new TimestampLogger(logger);
logger = new ErrorLevelLogger(logger);
logger.Log("This is a test message.");
```

Exercise 10: Extending the Proxy Pattern for Caching

Description: Implement caching in the Proxy Pattern.

Task: Modify the `ProxyImage` class to cache the loaded image data and avoid reloading it from disk.

Solution:

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

public class RealImage : IImage
{
    private string _filename;

    public RealImage(string filename)
    {
        _filename = filename;
        LoadImageFromDisk();
    }

    private void LoadImageFromDisk()
    {
        Console.WriteLine("Loading " + _filename);
    }
```

```
    public void Display()
    {
        Console.WriteLine("Displaying " + _filename);
    }
}

public class ProxyImage : IImage
{
    private RealImage _realImage;
    private string _filename;
    private bool _isLoaded = false;

    public ProxyImage(string filename)
    {
        _filename = filename;
    }

    public void Display()
    {
        if (!_isLoaded)
        {
            _realImage = new RealImage(_filename);
            _isLoaded = true;
        }
        _realImage.Display();
    }
}

// Usage
IImage image = new ProxyImage("photo.jpg");
image.Display(); // Loading and displaying the image
```

image.Display(); // Only displaying the image from cache

```

### Exercise 11: Implementing a Shape Drawing Application with the Bridge Pattern

**Description**: Use the Bridge Pattern to separate shape drawing and rendering implementations.

**Task**: Create `Circle` and `Square` shapes with `VectorRenderer` and `RasterRenderer` implementations.

**Solution**:

```csharp
public interface IRenderer
{
    void RenderShape(string shape);
}

public class VectorRenderer : IRenderer
{
    public void RenderShape(string shape)
    {
        Console.WriteLine($"Rendering {shape} as vectors.");
    }
}

public class RasterRenderer : IRenderer
{
    public void RenderShape(string shape)
    {
        Console.WriteLine($"Rendering {shape} as pixels.");
    }
}

public abstract class Shape
```
```

```
{
    protected IRenderer renderer;

    protected Shape(IRenderer renderer)
    {
        this.renderer = renderer;
    }

    public abstract void Draw();
}

public class Circle : Shape
{
    public Circle(IRenderer renderer) : base(renderer) { }

    public override void Draw()
    {
        renderer.RenderShape("Circle");
    }
}

public class Square : Shape
{
    public Square(IRenderer renderer) : base(renderer) { }

    public override void Draw()
    {
        renderer.RenderShape("Square");
    }
}
```

```
// Usage
IRenderer vectorRenderer = new VectorRenderer();
Shape circle = new Circle(vectorRenderer);
circle.Draw();

IRenderer rasterRenderer = new RasterRenderer();
Shape square = new Square(rasterRenderer);
square.Draw();
```

Exercise 12: Creating a Media Player using the Adapter Pattern

Description: Adapt legacy audio player functionality to a new interface.

Task: Given a `LegacyMediaPlayer` class with a `PlayAudioFile` method, create an adapter that allows it to work with an `IMediaPlayer` interface that has a `Play` method.

Solution:

```csharp
public interface IMediaPlayer
{
    void Play(string filename);
}

public class LegacyMediaPlayer
{
    public void PlayAudioFile(string filename)
    {
        Console.WriteLine("Playing audio file: " + filename);
    }
}

public class MediaPlayerAdapter : IMediaPlayer
{
```

```
    private LegacyMediaPlayer _legacyPlayer;

    public MediaPlayerAdapter(LegacyMediaPlayer legacyPlayer)
    {
        _legacyPlayer = legacyPlayer;
    }

    public void Play(string filename)
    {
        _legacyPlayer.PlayAudioFile(filename);
    }
}

// Usage
LegacyMediaPlayer legacyPlayer = new LegacyMediaPlayer();
IMediaPlayer player = new MediaPlayerAdapter(legacyPlayer);
player.Play("song.mp3");
```

Exercise 13: Implementing a UI Component System with the Composite Pattern

Description: Use the Composite Pattern to manage a UI component hierarchy.

Task: Create a `Component` interface with `Button` and `Panel` classes, where `Panel` can contain other `Component` objects.

Solution:

```csharp
public interface IComponent
{
    void Render();
}

public class Button : IComponent
```

```
{
    private string _text;

    public Button(string text)
    {
        _text = text;
    }

    public void Render()
    {
        Console.WriteLine($"Button: {_text}");
    }
}

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

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

    public void Render()
    {
        Console.WriteLine("Rendering Panel:");
        foreach (var component in _components)
        {
            component.Render();
        }
    }
```

```
}

// Usage
Panel panel = new Panel();
panel.Add(new Button("OK"));
panel.Add(new Button("Cancel"));
panel.Render();
```

Exercise 14: Adding Compression to a Data Stream using the Decorator Pattern

Description: Use the Decorator Pattern to add compression to a data stream.

Task: Create a `Stream` interface with `FileStream` and `CompressedStream` implementations, where `CompressedStream` adds compression.

Solution:

```csharp
public interface IStream
{
    void Write(string data);
}

public class FileStream : IStream
{
    public void Write(string data)
    {
        Console.WriteLine($"Writing data: {data}");
    }
}

public class CompressedStream : IStream
{
    private IStream _stream;
```

```
    public CompressedStream(IStream stream)
    {
        _stream = stream;
    }

    public void Write(string data)
    {
        string compressedData = Compress(data);
        _stream.Write(compressedData);
    }

    private string Compress(string data)
    {
        // Simulate compression
        return data.Substring(0, data.Length / 2);
    }
}

// Usage
IStream stream = new FileStream();
stream = new CompressedStream(stream);
stream.Write("This is a test data.");
```

Exercise 15: Simplifying Database Operations with the Facade Pattern

Description: Use the Facade Pattern to simplify database operations.

Task: Create a `DatabaseFacade` class that simplifies interaction with `Connection`, `Command`, and `Transaction` classes.

Solution:

```csharp
```

```
public class Connection
{
    public void Open() { Console.WriteLine("Opening connection."); }
    public void Close() { Console.WriteLine("Closing connection."); }
}

public class Command
{
    public void Execute() { Console.WriteLine("Executing command."); }
}

public class Transaction
{
    public void Begin() { Console.WriteLine("Beginning transaction."); }
    public void Commit() { Console.WriteLine("Committing transaction."); }
    public void Rollback() { Console.WriteLine("Rolling back transaction."); }
}

public class DatabaseFacade
{
    private Connection _connection;
    private Command _command;
    private Transaction _transaction;

    public DatabaseFacade()


    {
        _connection = new Connection();
        _command = new Command();
        _transaction = new Transaction();
```

```
    }

    public void ExecuteOperation()
    {
        _connection.Open();
        _transaction.Begin();
        try
        {
            _command.Execute();
            _transaction.Commit();
        }
        catch
        {
            _transaction.Rollback();
        }
        finally
        {
            _connection.Close();
        }
    }
}

// Usage
DatabaseFacade dbFacade = new DatabaseFacade();
dbFacade.ExecuteOperation();
```

Exercise 16: Implementing a Flyweight Pattern for a Text Editor

Description: Use the Flyweight Pattern to manage character formatting in a text editor.

Task: Create a `Character` class that uses a shared `CharacterStyle` object to minimize memory usage.

Solution:

```csharp
public class CharacterStyle
{
    public string Font { get; }
    public int Size { get; }
    public string Color { get; }

    public CharacterStyle(string font, int size, string color)
    {
        Font = font;
        Size = size;
        Color = color;
    }
}

public class Character
{
    private char _char;
    private CharacterStyle _style;

    public Character(char c, CharacterStyle style)
    {
        _char = c;
        _style = style;
    }

    public void Display()
    {
        Console.WriteLine($"Character: {_char}, Font: {_style.Font}, Size: {_style.Size}, Color: {_style.Color}");
```

```
    }
}

public class CharacterFactory
{
    private Dictionary<string, CharacterStyle> _styles = new Dictionary<string, CharacterStyle>();

    public CharacterStyle GetStyle(string font, int size, string color)
    {
        string key = $"{font}-{size}-{color}";
        if (!_styles.ContainsKey(key))
        {
            _styles[key] = new CharacterStyle(font, size, color);
        }
        return _styles[key];
    }
}

// Usage
CharacterFactory factory = new CharacterFactory();
CharacterStyle style = factory.GetStyle("Arial", 12, "Black");
Character character = new Character('A', style);
character.Display();
```

Exercise 17: Adding a Logging Proxy to a Service

Description: Use the Proxy Pattern to add logging to a service without modifying the service itself.

Task: Create a `LoggingProxy` class that logs method calls to a `Service` class.

Solution:

```csharp
```

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

public class Service : IService
{
    public void Execute()
    {
        Console.WriteLine("Service is executing.");
    }
}

public class LoggingProxy : IService
{
    private IService _service;

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

    public void Execute()
    {
        Console.WriteLine("Logging: Service execution started.");
        _service.Execute();
        Console.WriteLine("Logging: Service execution finished.");
    }
}

// Usage
```

```
IService service = new Service();
IService proxy = new LoggingProxy(service);
proxy.Execute();
```

Exercise 18: Implementing a GUI Component Framework with the Composite Pattern

Description: Use the Composite Pattern to manage a GUI component hierarchy.

Task: Extend the `IComponent` interface with a `Remove` method and implement it in the `Panel` class.

Solution:

```csharp
public interface IComponent
{
    void Render();
    void Remove(IComponent component);
}

public class Button : IComponent
{
    private string _text;

    public Button(string text)
    {
        _text = text;
    }

    public void Render()
    {
        Console.WriteLine($"Button: {_text}");
    }
```

```
    public void Remove(IComponent component)
    {
        // Not applicable for Button
    }
}

public class Panel : 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 Render()
    {
        Console.WriteLine("Rendering Panel:");
        foreach (var component in _components)
        {
            component.Render();
        }
    }
}

// Usage
```

```
Panel panel = new Panel();
IComponent button = new Button("OK");
panel.Add(button);
panel.Render();
panel.Remove(button);
panel.Render();
```

Exercise 19: Implementing a Lazy Loading Proxy

Description: Use the Proxy Pattern to implement lazy loading for a resource-intensive object.

Task: Create a `LazyLoadingProxy` class that only loads the `HeavyResource` object when it is accessed.

Solution:

```csharp
public interface IResource
{
    void Load();
}

public class HeavyResource : IResource
{
    public HeavyResource()
    {
        Console.WriteLine("HeavyResource is being loaded.");
    }

    public void Load()
    {
        Console.WriteLine("HeavyResource is now available.");
    }
}
```

```
public class LazyLoadingProxy : IResource
{
    private HeavyResource _resource;

    public void Load()
    {
        if (_resource == null)
        {
            _resource = new HeavyResource();
        }
        _resource.Load();
    }
}

// Usage
IResource resource = new LazyLoadingProxy();
resource.Load(); // Resource is loaded here
resource.Load(); // Resource is already loaded
```

Exercise 20: Simplifying Network Communication with the Facade Pattern

Description: Use the Facade Pattern to simplify network communication.

Task: Create a `NetworkFacade` class that provides a simple interface for connecting, sending data, and disconnecting from a server.

Solution:

```csharp
public class Connection
{
    public void Connect() { Console.WriteLine("Connecting to server..."); }
    public void Disconnect() { Console.WriteLine("Disconnecting from server..."); }
```

```
}

public class DataTransfer
{
    public void SendData(string data) { Console.WriteLine($"Sending data: {data}"); }
}

public class NetworkFacade
{
    private Connection _connection;
    private DataTransfer _dataTransfer;

    public NetworkFacade()
    {
        _connection = new Connection();
        _dataTransfer = new DataTransfer();
    }

    public void Communicate(string data)
    {
        _connection.Connect();
        _dataTransfer.SendData(data);
        _connection.Disconnect();
    }
}

// Usage
NetworkFacade network = new NetworkFacade();
network.Communicate("Hello, World!");
```