Creating a comprehensive exercise that incorporates all these design patterns (Mediator, Command, Proxy, Facade, Decorator) and providing a full implementation in C# would be quite extensive. However, I can sketch a scenario and provide some abbreviated C# code to give you a head start.

**Scenario:**

Imagine we have a "SmartHome" system where various devices communicate. A central hub mediates communication. Devices can send commands to other devices, some devices might require a proxy for security, there's a facade to make interacting with subsystems simpler, and devices can be decorated to add new features.

**Mediator**: CentralHub mediates the communication between devices.

**Command**: Devices send commands to operate other devices.

**Proxy**: SecurityCamera has a proxy to control access to the feed.

**Facade**: SmartHomeFacade provides simplified access to multiple subsystems.

**Decorator**: Decorate devices to add additional functionalities like logging.

**Simplified Implementation:**

csharp

// Command Pattern

public interface ICommand

{

void Execute();

}

public class TurnOnCommand : ICommand

{

private Light \_light;

public TurnOnCommand(Light light)

{

\_light = light;

}

public void Execute()

{

\_light.TurnOn();

}

}

// Mediator Pattern

public interface IMediator

{

void Notify(object sender, string ev);

}

public class CentralHub : IMediator

{

public void Notify(object sender, string ev)

{

// Handle notifications and facilitate communication

}

}

// Proxy Pattern

public interface ICamera

{

void DisplayFeed();

}

public class SecurityCamera : ICamera

{

public void DisplayFeed()

{

Console.WriteLine("Displaying camera feed...");

}

}

public class SecurityCameraProxy : ICamera

{

private SecurityCamera \_camera;

private string \_password = "secret";

public SecurityCameraProxy(SecurityCamera camera)

{

\_camera = camera;

}

public void DisplayFeed(string password)

{

if(password == \_password)

{

\_camera.DisplayFeed();

}

else

{

Console.WriteLine("Access denied!");

}

}

}

// Facade Pattern

public class SmartHomeFacade

{

private Light \_light;

private SecurityCameraProxy \_camera;

public SmartHomeFacade(Light light, SecurityCameraProxy camera)

{

\_light = light;

\_camera = camera;

}

public void EveningRoutine(string cameraPassword)

{

\_light.TurnOn();

\_camera.DisplayFeed(cameraPassword);

}

}

// Decorator Pattern

public abstract class DeviceDecorator : Device

{

protected Device \_device;

public DeviceDecorator(Device device)

{

\_device = device;

}

}

public class LoggingDeviceDecorator : DeviceDecorator

{

public LoggingDeviceDecorator(Device device) : base(device)

{}

public void Operate()

{

Console.WriteLine("Logging operation...");

\_device.Operate();

}

}

public abstract class Device

{

public abstract void Operate();

}

public class Light : Device

{

public override void Operate()

{

Console.WriteLine("Light is operated.");

}

public void TurnOn()

{

Console.WriteLine("Light is turned on.");

}

}

// Usage

var hub = new CentralHub();

var light = new Light();

var camera = new SecurityCamera();

var cameraProxy = new SecurityCameraProxy(camera);

var facade = new SmartHomeFacade(light, cameraProxy);

facade.EveningRoutine("secret");

This is a quite simplified representation. In a real-world application, there would be more intricate interactions, error handling, possibly more layers, etc. It's also important to remember that design patterns should be applied where they make sense and not enforced.