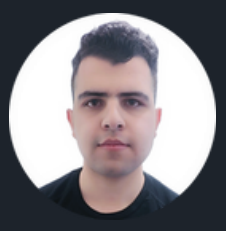
Single Responsibility Principle

A class/method should only have one reason to change.

```
public class Logger
{
    public void Log(string message)
    {
        // Logging logic
    }
}

public class Authenticator
{
    public bool Authenticate(string username, string password)
    {
        // Authentication logic
        return true;
    }
}
```

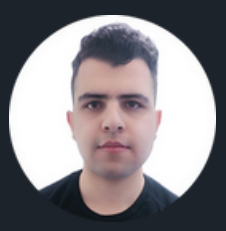


Elliot One

SRP Violation

SRP is violated because the Authenticator handles both authentication and logging.

```
public class LoggerAndAuthenticator
{
    public void Log(string message)
    {
        // Logging logic
    }

    public bool Authenticate(string username, string password)
    {
        // Authentication logic
        return true;
    }
}
```

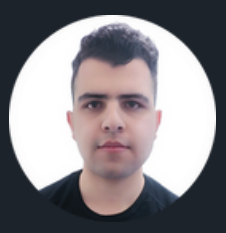


Elliot One

Open/Closed Principle

Software entities (modules, classes, methods) should be open for extension but closed for modification.

```
public interface IShape
{
    double CalculateArea();
}

public class Rectangle : IShape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public double CalculateArea()
    {
        return Width * Height;
    }
}

public class Circle : IShape
{
    public double Radius { get; set; }

    public double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }
}
```

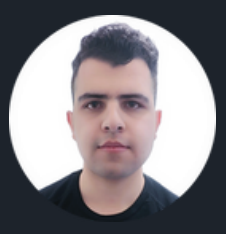


Elliot One

OCP Violation

OCP is violated because modifying the area calculation for different shapes requires changing the AreaCalculator class.

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}

public class AreaCalculator
{
    public double CalculateArea(Rectangle rectangle)
    {
        return rectangle.Width * rectangle.Height;
    }
}
```

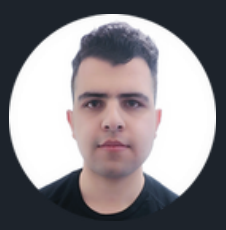


Elliot One

Liskov Substitution Principle

Subtypes must be substitutable for their base types without altering the correctness of the program.

```
IBird parrot = new Parrot();
parrot.Move(); // Parrot can move by either flying or walking

IBird penguin = new Penguin();
penguin.Move(); // Penguin can only move by walking

public interface IBird
{
    void Move();
}

public interface IFlyingBird : IBird
{
    void Fly();
}

public interface IWalkingBird : IBird
{
    void Walk();
}

public class Parrot : IFlyingBird, IWalkingBird
{
    public void Fly()
    {
        Console.WriteLine("Parrot flying...");
    }

    public void Walk()
    {
        Console.WriteLine("Parrot walking...");
    }

    public void Move()
    {
        // Parrot can either fly or walk
        Fly();
    }
}

public class Penguin : IWalkingBird
{
    public void Walk()
    {
        Console.WriteLine("Penguin walking...");
    }

    public void Move()
    {
        // Penguins can only walk
        Walk();
    }
}
```

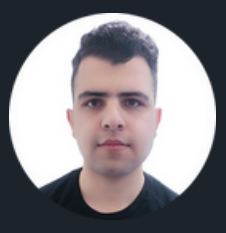


Elliot One

LSP Violation

LSP is violated because Penguin cannot substitute Bird without altering expected behavior by throwing an exception for Fly().

```
public class Bird
{
    public virtual void Fly()
    {
        // Flying logic
    }
}

public class Penguin : Bird
{
    // Penguins can't fly, violating LSP
    public override void Fly()
    {
        throw new InvalidOperationException(
            "Penguins can't fly.");
    }
}
```

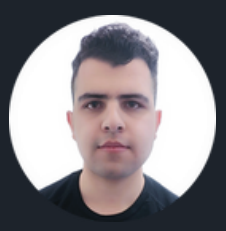


Elliot One

Interface Segregation Principle

A class should not be forced to implement interfaces it does not use.

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

public interface IEater
{
    void Eat();
}

public class Robot : IWorker
{
    public void Work()
    {
        // Work logic
    }
}
```

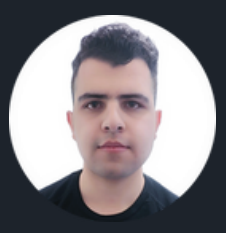


Elliot One

ISP Violation

ISP is violated because Robot must implement the Eat method it doesn't need.

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

public class Robot : IWorker
{
    public void Work()
    {
        // Work logic
    }

    public void Eat()
    {
        // Robot doesn't eat, violating ISP
    }
}
```

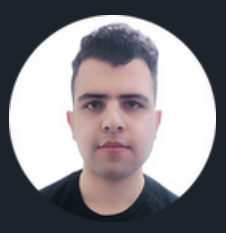


Elliot One

Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions.

```
public interface ISwitchable
{
    void TurnOn();
}

public class LightBulb : ISwitchable
{
    public void TurnOn()
    {
        // Turn on logic
    }
}

public class Switch
{
    private readonly ISwitchable device;

    public Switch(ISwitchable device)
    {
        this.device = device;
    }

    public void Press()
    {
        device.TurnOn();
    }
}
```

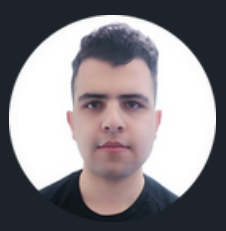


Elliot One

DIP Violation

DIP is violated because Switch directly depends on the concrete LightBulb class instead of an abstraction.

```
public class LightBulb
{
    public void TurnOn()
    {
        // Turn on logic
    }
}

public class Switch
{
    private readonly LightBulb bulb;

    public Switch()
    {
        this.bulb = new LightBulb();
    }

    public void Press()
    {
        bulb.TurnOn();
    }
}
```



Elliot One

Elliot One

Enjoyed Reading This?
Reshare and Spread Knowledge.