LevelUP
IT-образование
в Петербурге
и удалённо

Towards the .NET Junior Developer

The extremely solid course

Lesson 9

SOLID principles and patterns

Agenda

- SOLID principles
  - Single responsibility principle
  - Open-closed principle
  - Liskov substitution principle
  - Interfaces segregation principle
  - Dependency inversion principle
- Patterns
  - Creational patterns
  - Behavioral patterns
  - Structural patterns
- Books of the day
- Links of the day
- Hometask

SOLID principles

Single responsibility principle

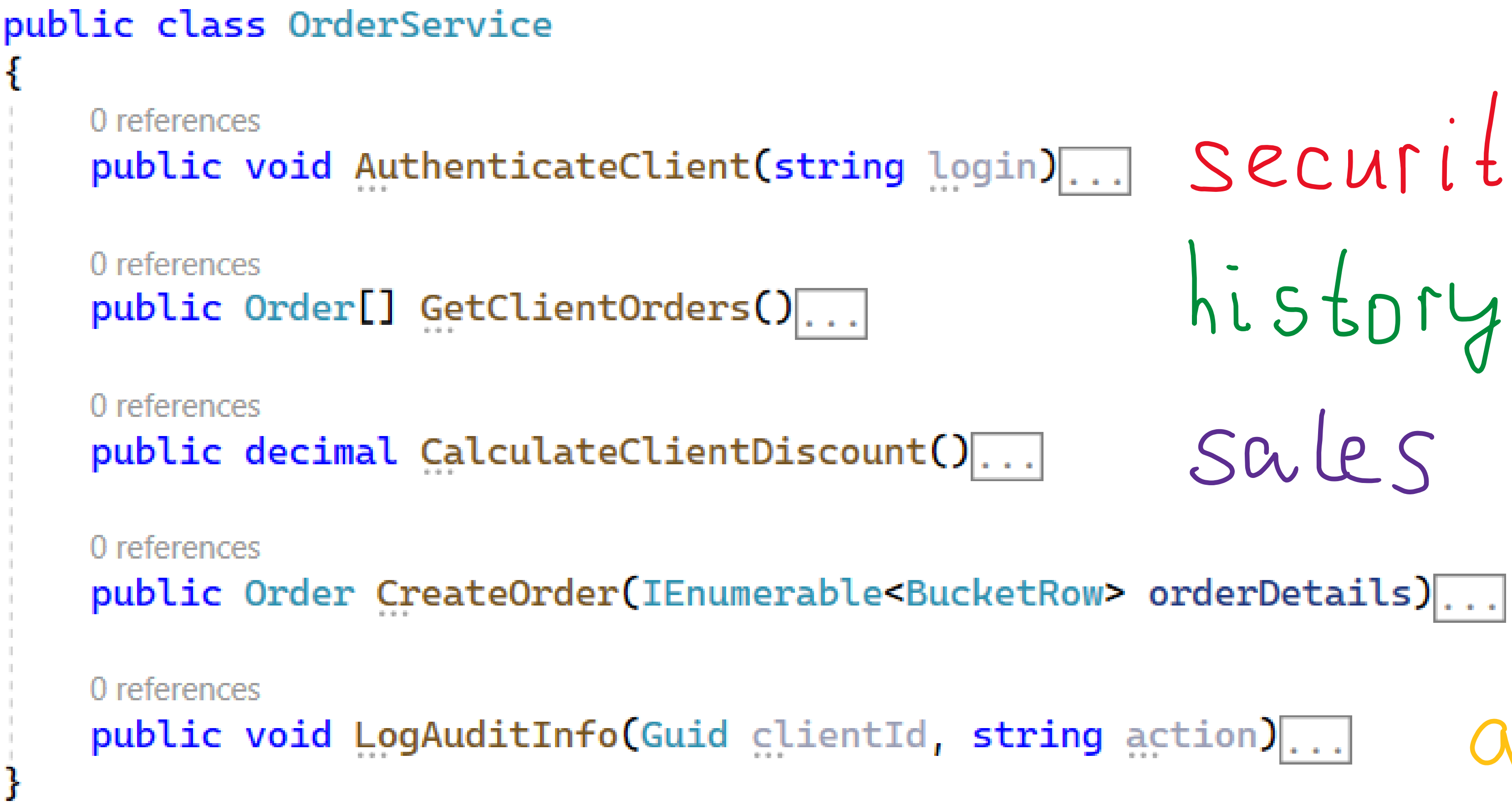
“There should never be more than one reason for a class to change”

Classic

“A module should be responsible to one, and only one, actor”

R. Martin

Single responsibility principle



```
public class OrderService
{
    0 references
    public void AuthenticateClient(string login)...

    0 references
    public Order[] GetClientOrders()...

    0 references
    public decimal CalculateClientDiscount()...

    0 references
    public Order CreateOrder(IEnumerable<BucketRow> orderDetails)...

    0 references
    public void LogAuditInfo(Guid clientId, string action)...
}
```

security

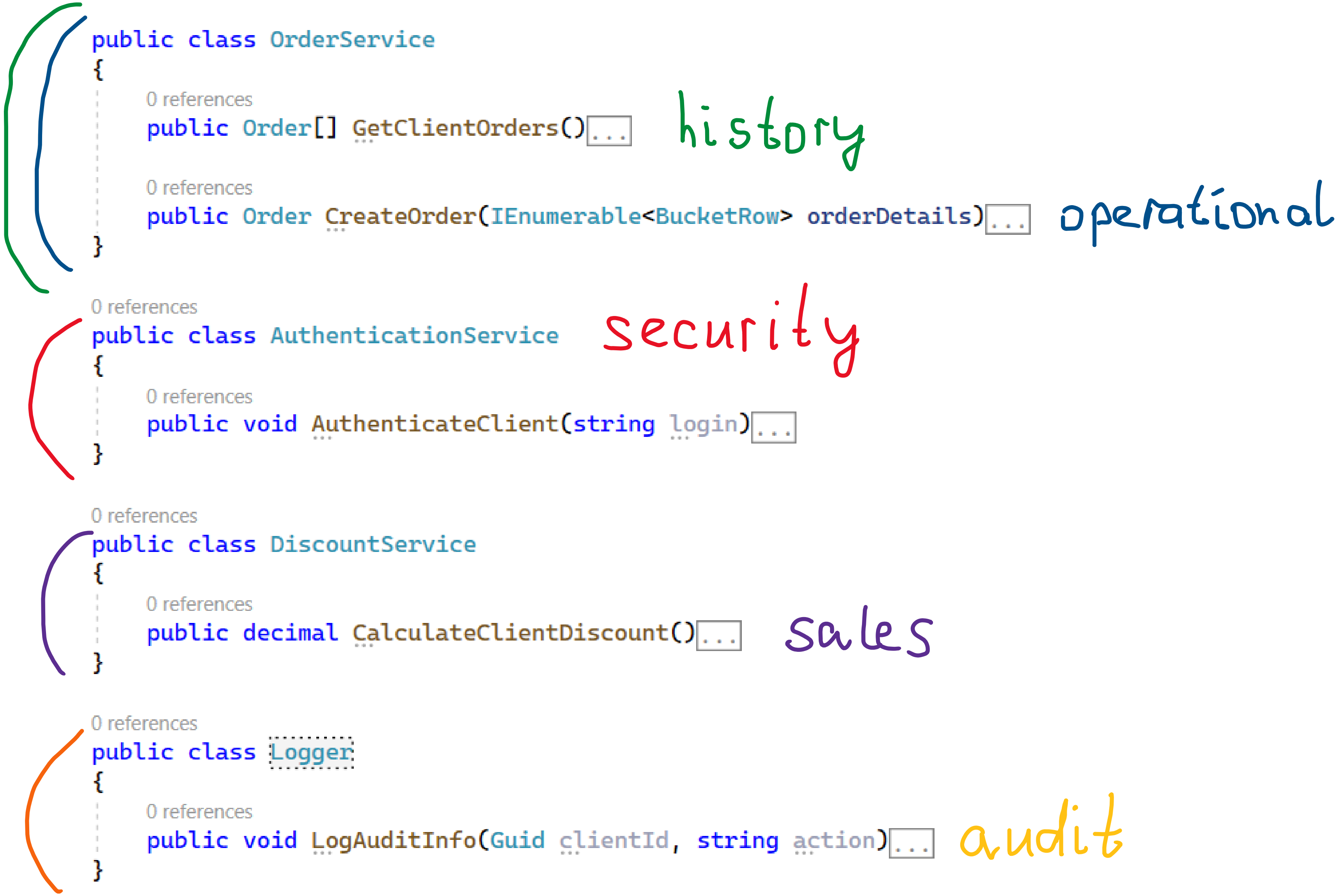
history

sales

operational

audit

Single responsibility principle

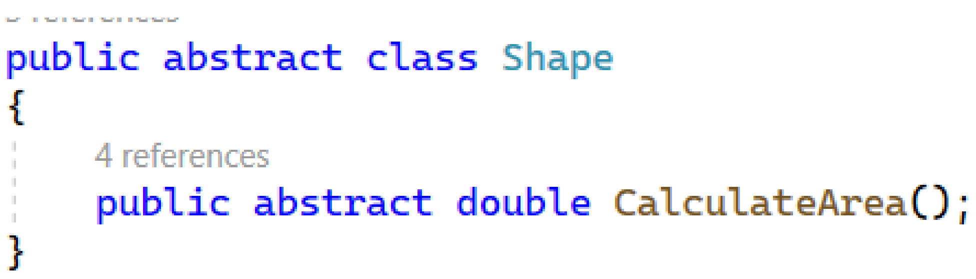


```
public class OrderService
{
    0 references
    public Order[] GetClientOrders()[...]

    0 references
    public Order CreateOrder(IEnumerable<BucketRow> orderDetails)[...]
}

0 references
public class AuthenticationService
{
    0 references
    public void AuthenticateClient(string login)[...]
}

0 references
public class DiscountService
{
    0 references
    public decimal CalculateClientDiscount()[...]
}

0 references
public class Logger
{
    0 references
    public void LogAuditInfo(Guid clientId, string action)[...]
}
```

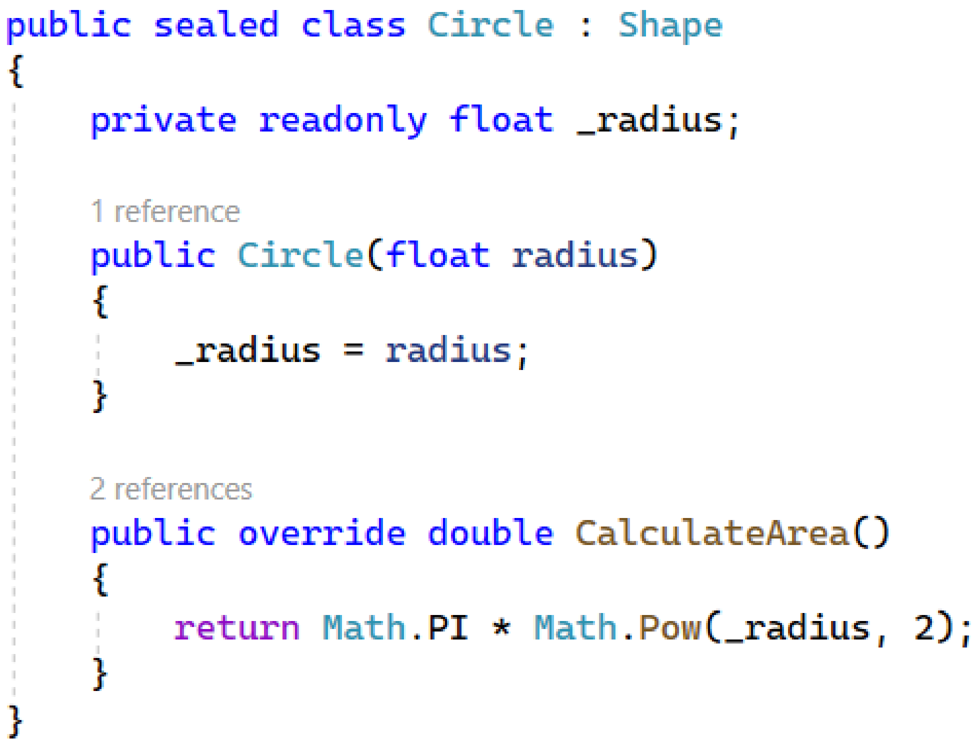
Open-closed principle

“Software entities should be open for extension, but closed for modification”

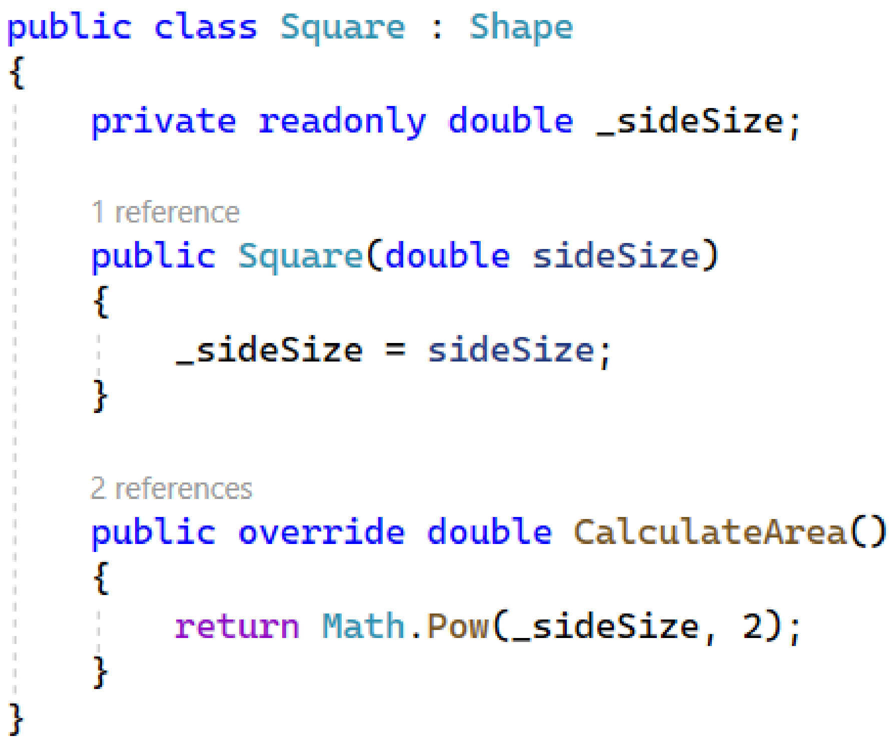
Open-closed principle



```
public abstract class Shape
{
    4 references
    public abstract double CalculateArea();
}
```

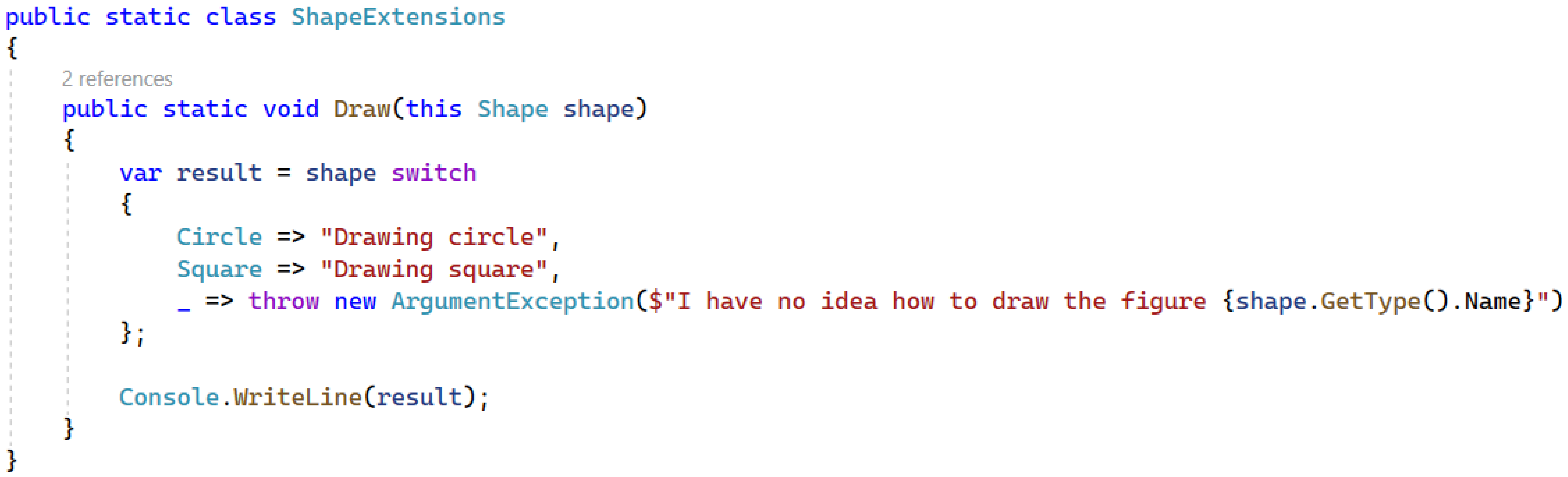


```
public sealed class Circle : Shape
{
    private readonly float _radius;

    1 reference
    public Circle(float radius)
    {
        _radius = radius;
    }

    2 references
    public override double CalculateArea()
    {
        return Math.PI * Math.Pow(_radius, 2);
    }
}
```



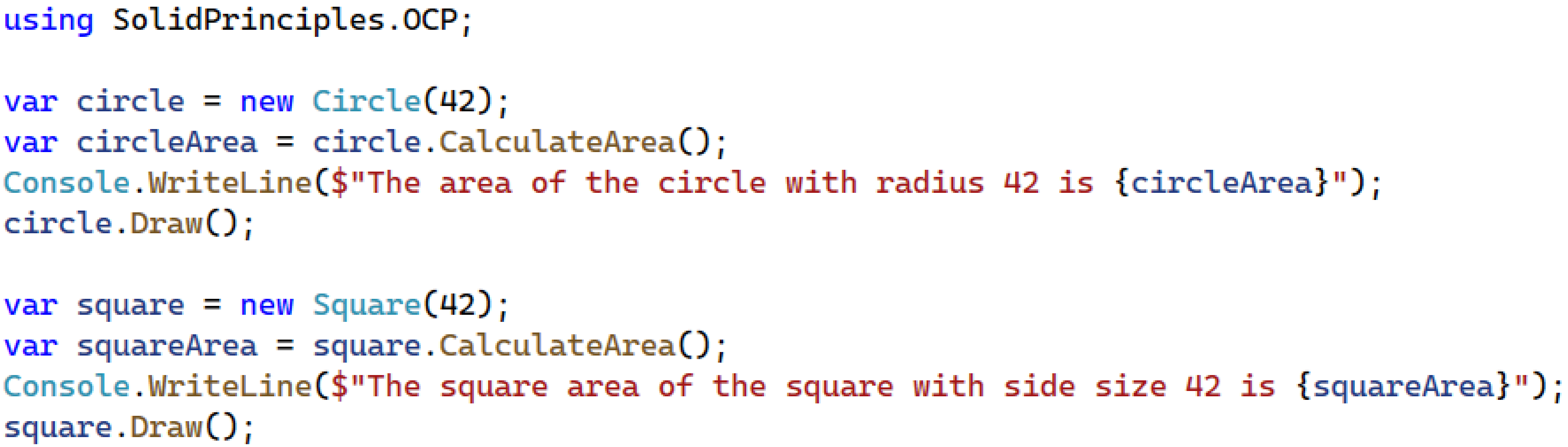
```
public class Square : Shape
{
    private readonly double _sideSize;

    1 reference
    public Square(double sideSize)
    {
        _sideSize = sideSize;
    }

    2 references
    public override double CalculateArea()
    {
        return Math.Pow(_sideSize, 2);
    }
}
```

Open-closed principle

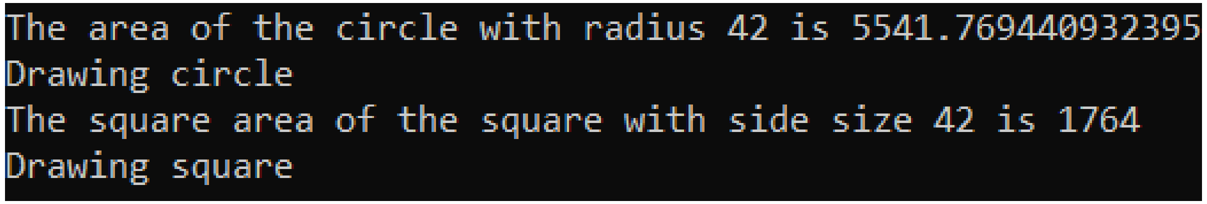


```
public static class ShapeExtensions
{
    2 references
    public static void Draw(this Shape shape)
    {
        var result = shape switch
        {
            Circle => "Drawing circle",
            Square => "Drawing square",
            _ => throw new ArgumentException($"I have no idea how to draw the figure {shape.GetType().Name}")
        };

        Console.WriteLine(result);
    }
}
```

Open-closed principle



```
using SolidPrinciples.OCP;

var circle = new Circle(42);
var circleArea = circle.CalculateArea();
Console.WriteLine($"The area of the circle with radius 42 is {circleArea}");
circle.Draw();

var square = new Square(42);
var squareArea = square.CalculateArea();
Console.WriteLine($"The square area of the square with side size 42 is {squareArea}");
square.Draw();
```



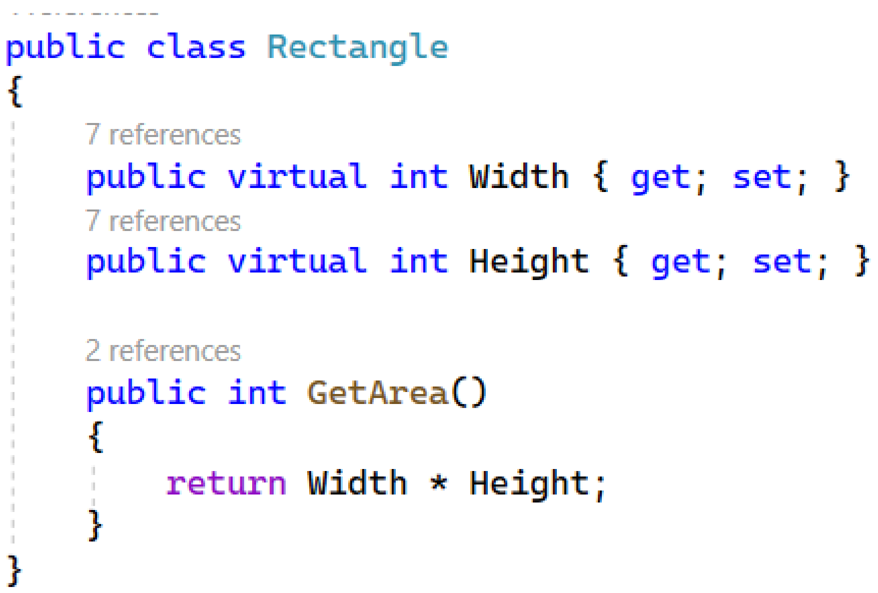
```
The area of the circle with radius 42 is 5541.769440932395
Drawing circle
The square area of the square with side size 42 is 1764
Drawing square
```

Liskov substitution principle

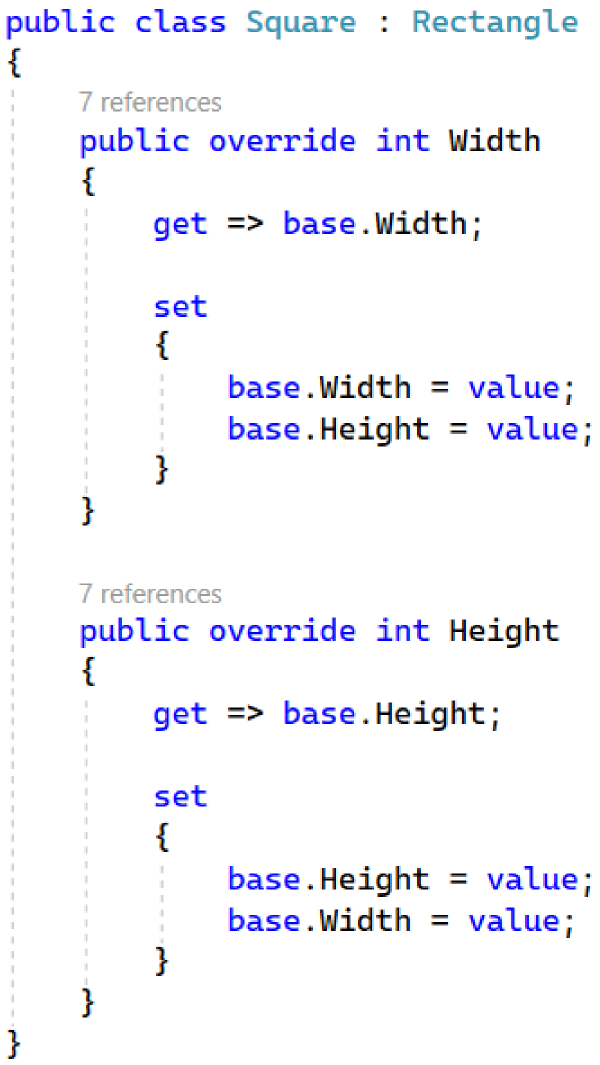
“Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it”

Barbara Liskov

Liskov substitution principle

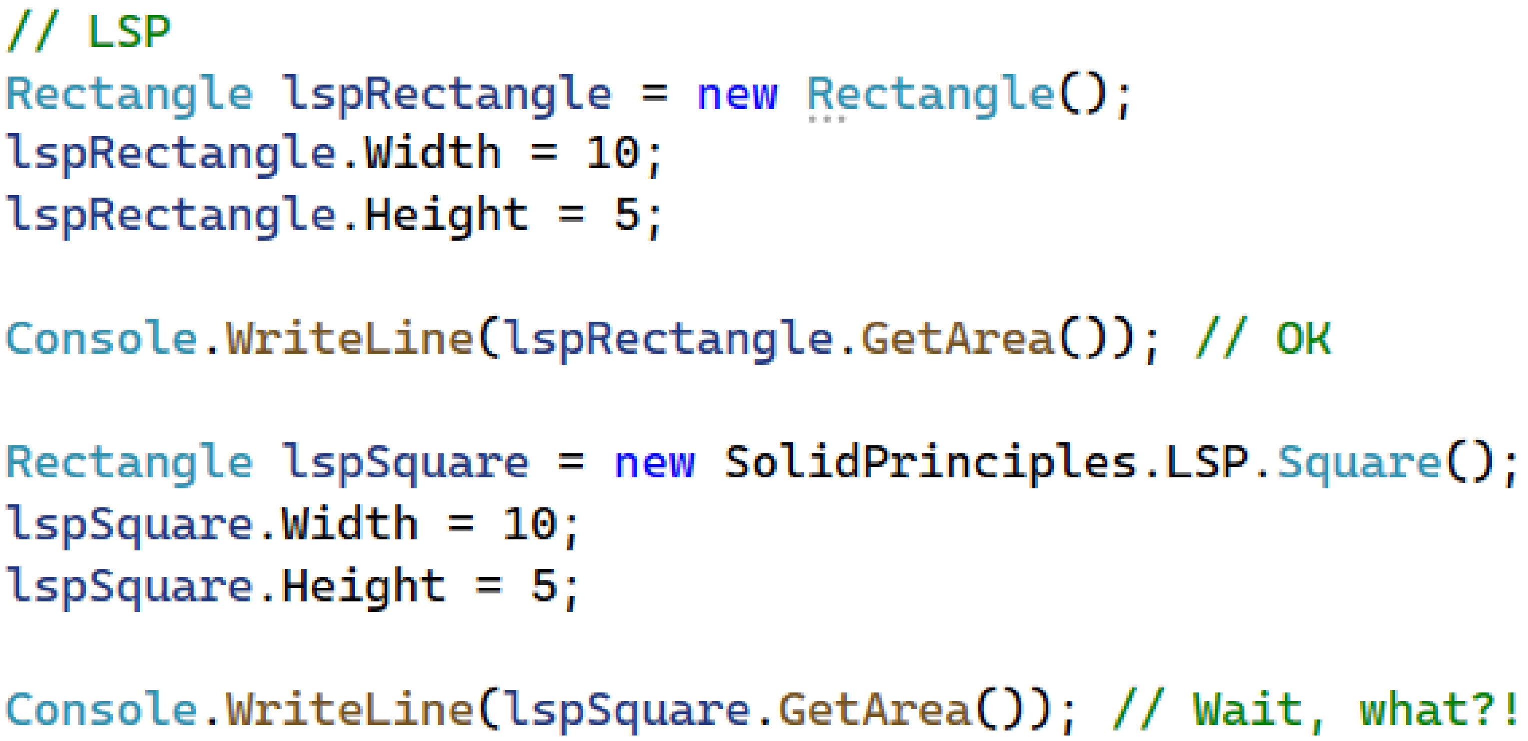


```
public class Rectangle
{
    7 references
    public virtual int Width { get; set; }
    7 references
    public virtual int Height { get; set; }

    2 references
    public int GetArea()
    {
        return Width * Height;
    }
}
```

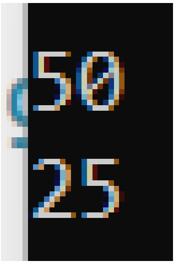


```
public class Square : Rectangle
{
    7 references
    public override int Width
    {
        get => base.Width;

        set
        {
            base.Width = value;
            base.Height = value;
        }
    }

    7 references
    public override int Height
    {
        get => base.Height;

        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```

Liskov substitution principle



```
// LSP
Rectangle lspRectangle = new Rectangle();
lspRectangle.Width = 10;
lspRectangle.Height = 5;

Console.WriteLine(lspRectangle.GetArea()); // OK

Rectangle lspSquare = new SolidPrinciples.LSP.Square();
lspSquare.Width = 10;
lspSquare.Height = 5;

Console.WriteLine(lspSquare.GetArea()); // Wait, what?!
```

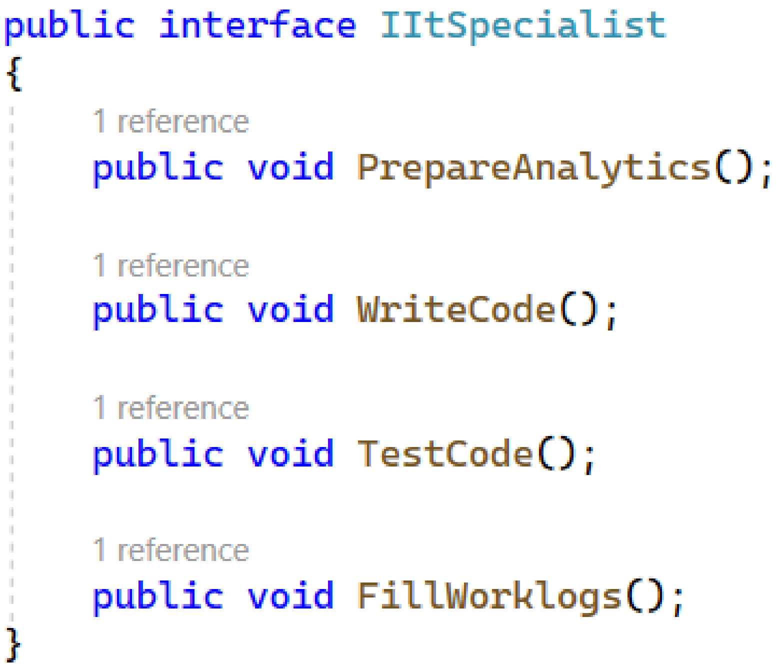


```
50
25
```

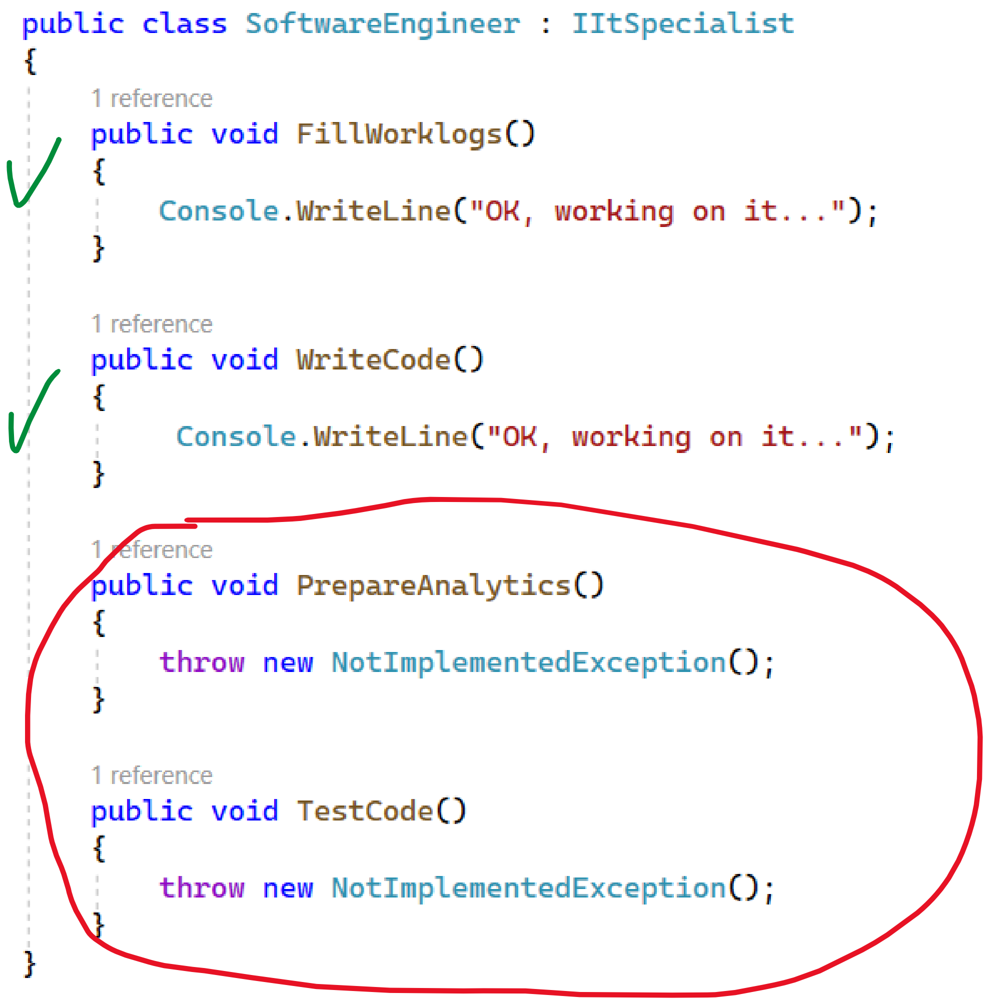
Interfaces segregation principle

“Clients should not be forced to depend upon interfaces that they do not use”

Interfaces segregation principle

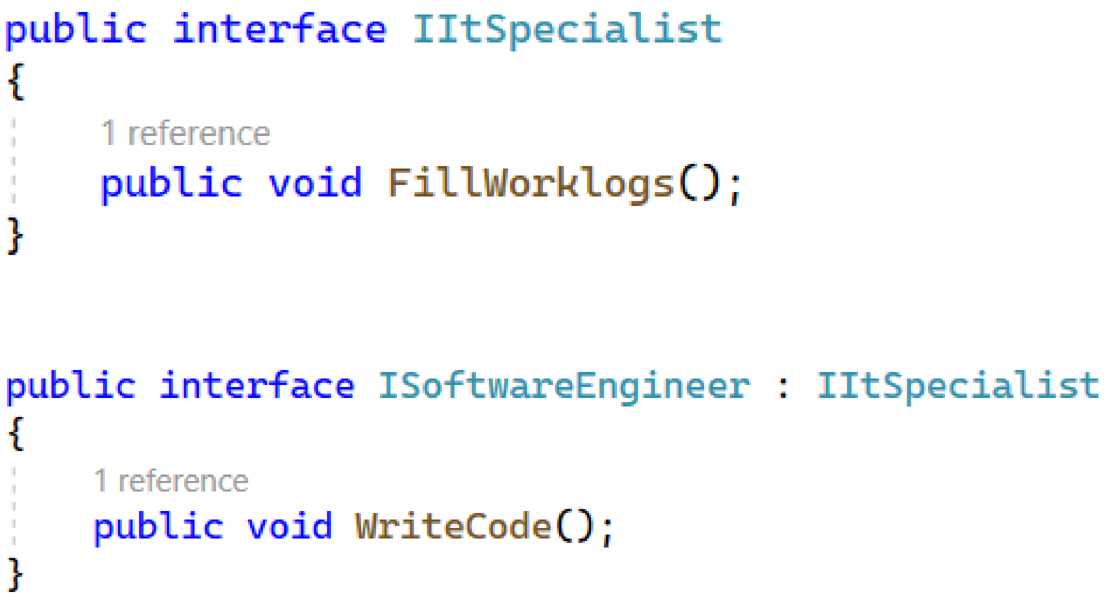


```
public interface IItSpecialist
{
        1 reference
        public void PrepareAnalytics();

        1 reference
        public void WriteCode();

        1 reference
        public void TestCode();

        1 reference
        public void FillWorklogs();
}
```

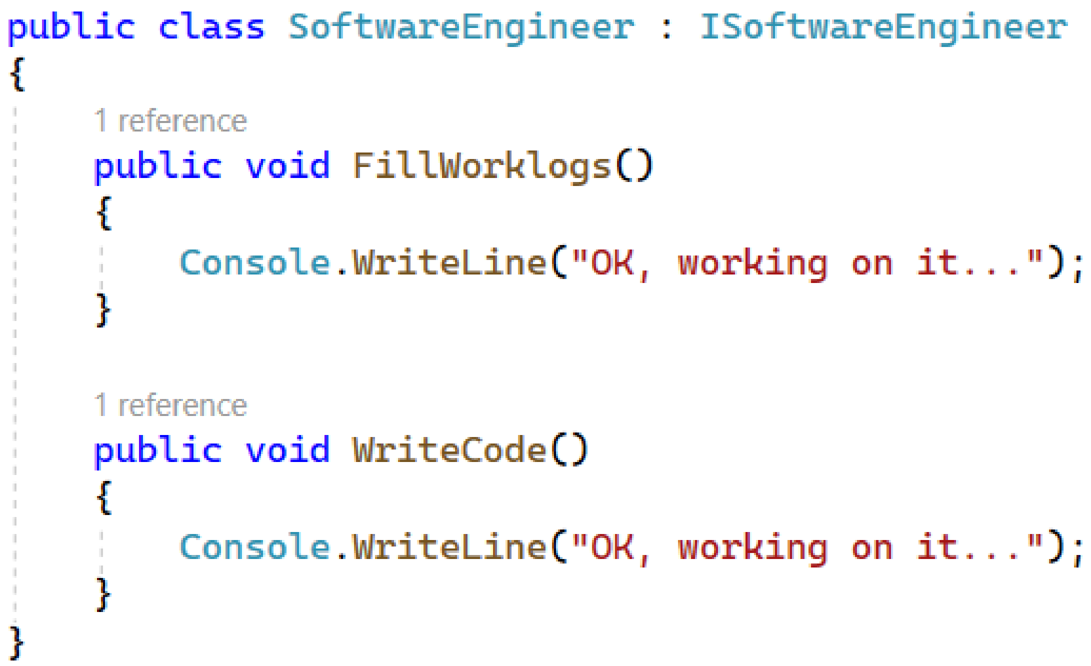


```
public class SoftwareEngineer : IItSpecialist
{
        1 reference
        public void FillWorklogs()
✓       {
            Console.WriteLine("OK, working on it...");
        }

        1 reference
        public void WriteCode()
✓       {
             Console.WriteLine("OK, working on it...");
        }

        1 reference
        public void PrepareAnalytics()
        {
            throw new NotImplementedException();
        }

        1 reference
        public void TestCode()
        {
            throw new NotImplementedException();
        }
}
```

Interfaces segregation principle

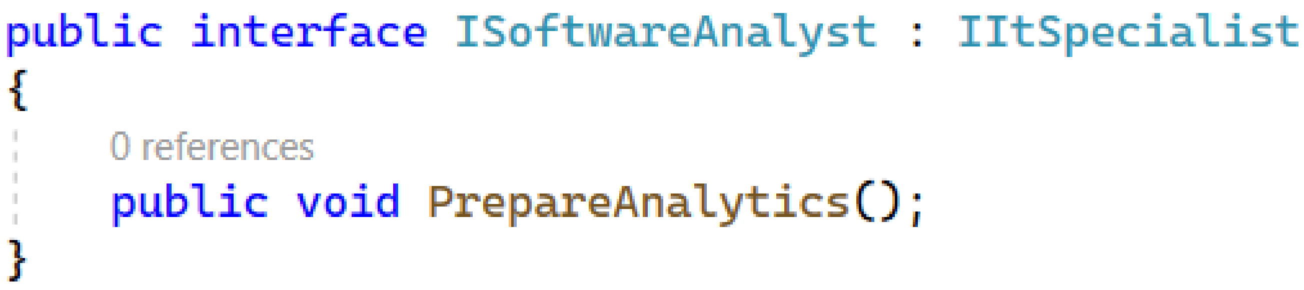


```
public interface IItSpecialist
{
        1 reference
        public void FillWorklogs();
}


public interface ISoftwareEngineer : IItSpecialist
{
        1 reference
        public void WriteCode();
}
```

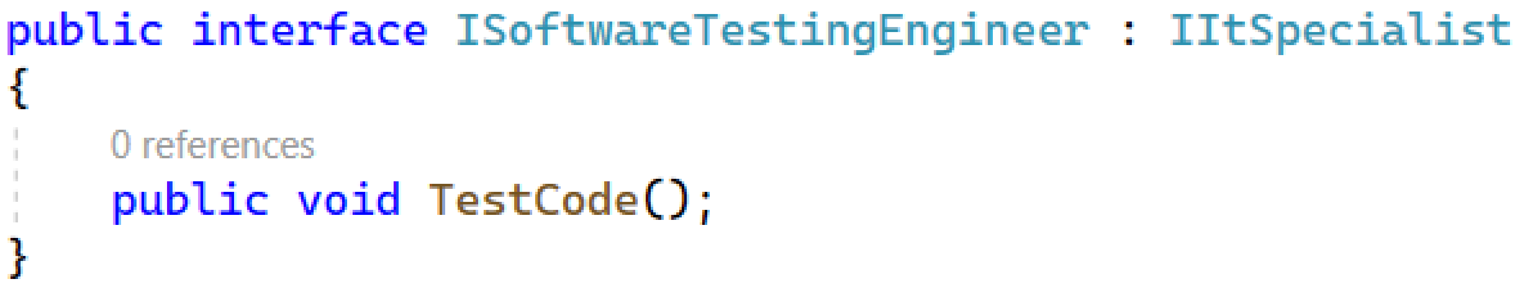


```
public class SoftwareEngineer : ISoftwareEngineer
{
        1 reference
        public void FillWorklogs()
        {
            Console.WriteLine("OK, working on it...");
        }

        1 reference
        public void WriteCode()
        {
            Console.WriteLine("OK, working on it...");
        }
}
```

Interfaces segregation principle



```
public interface ISoftwareAnalyst : IItSpecialist
{
        0 references
        public void PrepareAnalytics();
}
```

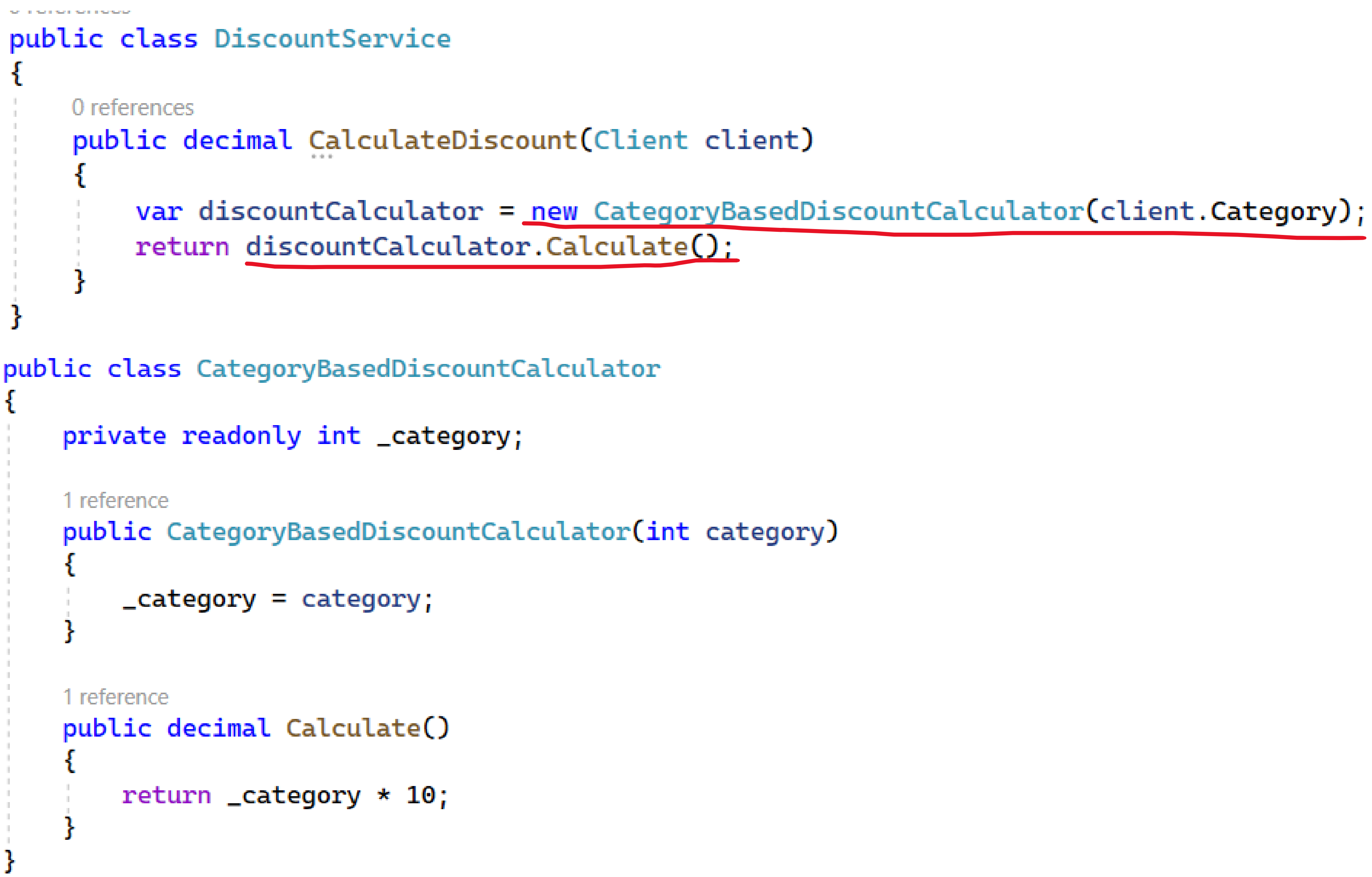


```
public interface ISoftwareTestingEngineer : IItSpecialist
{
        0 references
        public void TestCode();
}
```

Dependency inversion principle

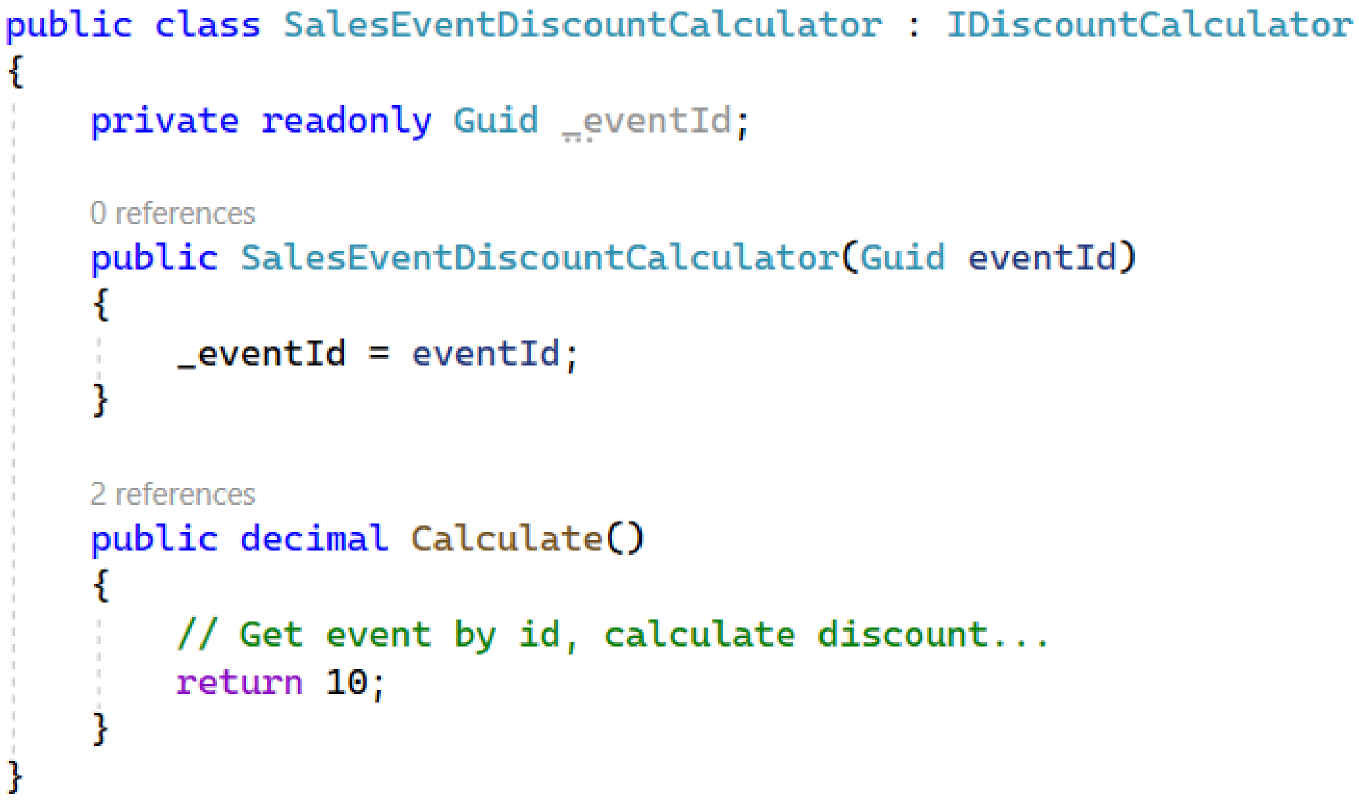
“Depend upon abstractions, [not] concretions”

Dependency inversion principle



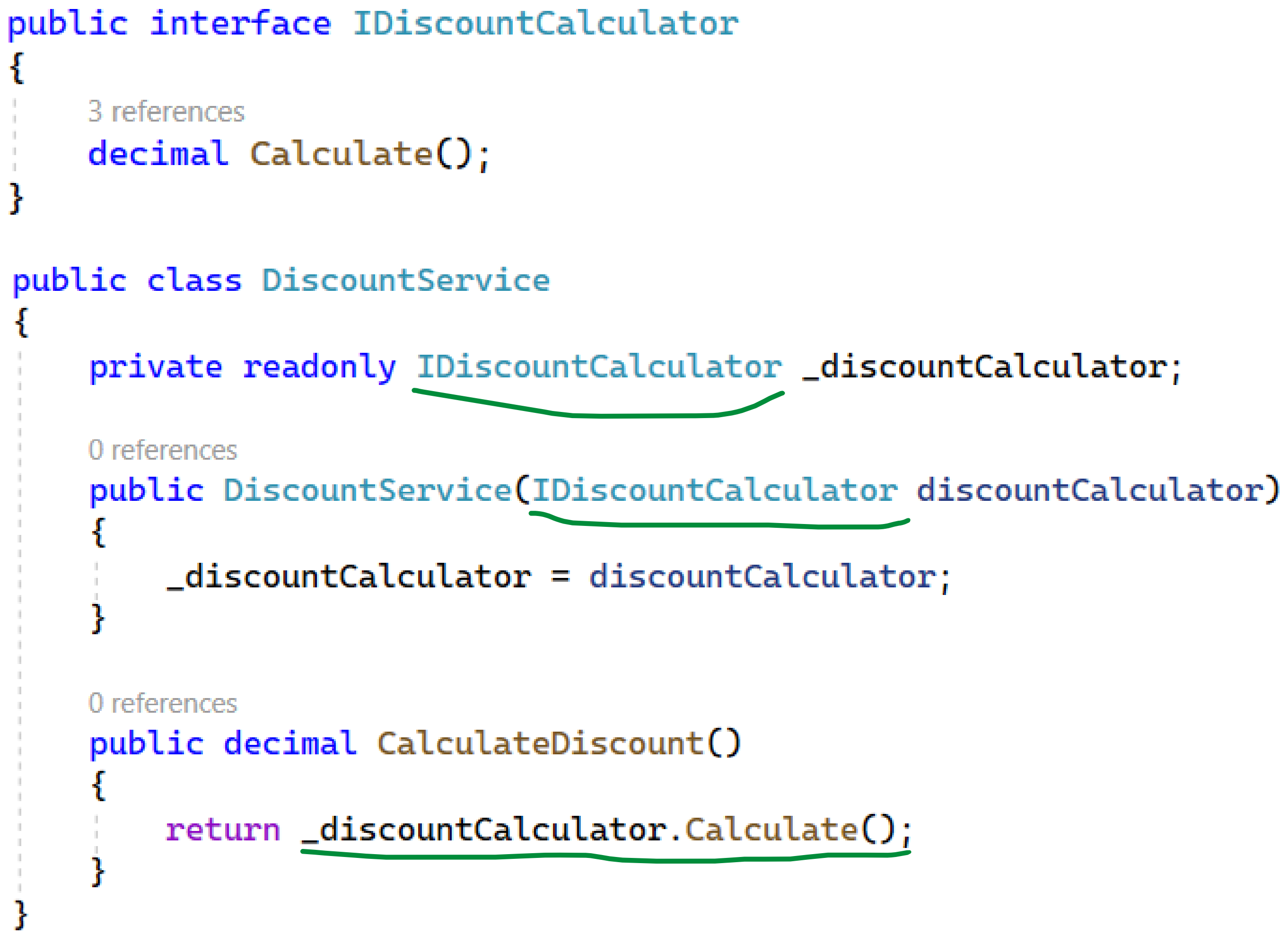
```
public class DiscountService
{
    0 references
    public decimal CalculateDiscount(Client client)
    {
        var discountCalculator = new CategoryBasedDiscountCalculator(client.Category);
        return discountCalculator.Calculate();
    }
}

public class CategoryBasedDiscountCalculator
{
    private readonly int _category;

    1 reference
    public CategoryBasedDiscountCalculator(int category)
    {
        _category = category;
    }

    1 reference
    public decimal Calculate()
    {
        return _category * 10;
    }
}
```

Dependency inversion principle



```
public class SalesEventDiscountCalculator : IDiscountCalculator
{
    private readonly Guid _eventId;

    0 references
    public SalesEventDiscountCalculator(Guid eventId)
    {
        _eventId = eventId;
    }

    2 references
    public decimal Calculate()
    {
        // Get event by id, calculate discount...
        return 10;
    }
}
```

Dependency inversion principle



```
public interface IDiscountCalculator
{
    3 references
    decimal Calculate();
}

public class DiscountService
{
    private readonly IDiscountCalculator _discountCalculator;

    0 references
    public DiscountService(IDiscountCalculator discountCalculator)
    {
        _discountCalculator = discountCalculator;
    }

    0 references
    public decimal CalculateDiscount()
    {
        return _discountCalculator.Calculate();
    }
}
```

DI

SOLID principles demo

Visual Studio

Patterns

Creational patterns

“Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation”

Abstract Factory pattern

Factory method pattern

Builder pattern

Prototype pattern

Singleton pattern

Abstract Factory demo

Visual Studio

Behavioral patterns

“Behavioral design patterns are design patterns that identify common communication patterns among objects"

- Chain of responsibility pattern
- Command pattern
- Interpreter pattern
- Iterator pattern
- Mediator pattern
- Memento pattern
- Observer pattern
- State pattern
- Strategy pattern
- Template method pattern
- Visitor pattern

Strategy demo

Visual Studio

Structural patterns

“Structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships among entities”

Adapter pattern

Bridge pattern

Composite pattern

Decorator pattern

Facade pattern

Flyweight pattern

Proxy pattern

Proxy demo

Books of the day

Freeman, Robson, Sierra, Bates - Head First. Design Patterns

Gamma, Helm, Johnson, Vlissides - Design Patterns: Elements of Reusable Object

Martin R. – Clean Architecture

Teplyakov S. – Design Patterns on the .NET Platform

Links of the day

Single responsibility principle (Habr)

Open-closed principle (Habr)

Liskov substitution principle (Habr)

Interfaces segregation principle

Dependency inversion principle

Patterns cheat sheet (Habr)

Strategy Design Pattern Using C# (c-sharpcorner.com)

Hometask

1. Create console application with the class DeliveryAddressBuilder. It should build the client's delivery address from the index, country, city/region, street, house and apartments number. Information should be entered by the user. Add validation for the entered parameters as needed. Use Builder pattern.

2. Let's use the Observer pattern! Create console application with the behavior above. User can enter several numbers in the console with the whitespace between the numbers. Create the class NumbersProcessor and the event "OnNumbersEntered" in it. When user press Enter, event should be raised.
Add two listeners for the event.
The first one should calculate the sum of the elements and print this sum to the console.
The second one should invert the array and print it.

That’s all for this time!