

# объектно-ориентированное программирование

принципы **SOLID**

**single responsibility principle**

# SRP

## пример несоблюдения

```
public record OperationResult(...);

public class ReportGenerator
{
    public void GenerateExcelReport(OperationResult result)
    {
        ...
    }

    public void GeneratePdfReport(OperationResult result)
    {
        ...
    }
}
```

# SRP

## пример соблюдения

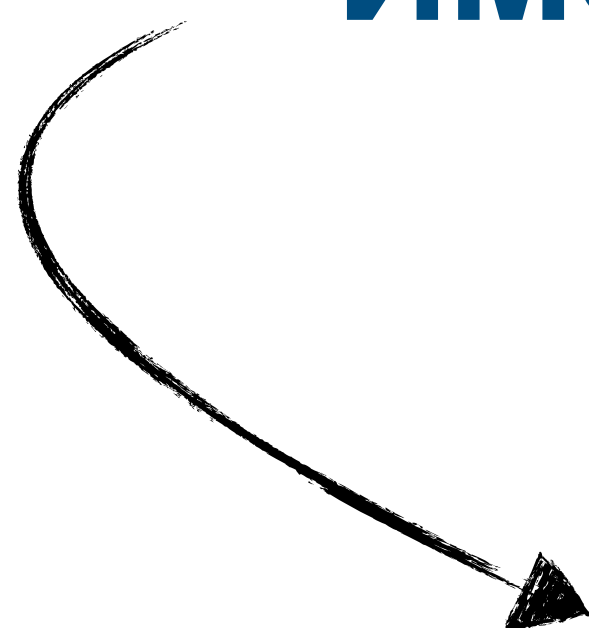
```
public record OperationResult(...);

public interface IReportGenerator
{
    void GenerateReport(OperationResult result);
}

public class ExcelReportGenerator : IReportGenerator
{
    public void GenerateReport(OperationResult result)
    {
        ...
    }
}

public class PdfReportGenerator : IReportGenerator
{
    public void GenerateReport(OperationResult result)
    {
        ...
    }
}
```

**проектирование типов, таким образом что они  
имеют единственную причину для изменения**



**single responsibility principle**

# SRP

## преимущества несоблюдения

- простота  
нет необходимости в абстракциях  
низкий порог для онбординга
- переиспользование логики  
часто логика в типах не соблюдающих SRP имеет общие части, вызвать приватный метод типа в нескольких местах проще чем реализовывать грамотную декомпозицию

# SRP

## последствия несоблюдения

- сильная связанность реализации различных бизнес требований от простого: загрязнённый контекст для IntelliSense до тяжёлого: усложнение тестирования
- усложнённая кастомизация отдельных реализаций изменения в общем коде могут поломать другие решения

open/closed principle



# ОСР

## пример несоблюдения

```
public enum BinaryOperation
{
    Summation,
    Subtraction,
}

public class BinaryOperand
{
    private readonly int _left;
    private readonly int _right;

    // ...

    public int Evaluate(BinaryOperation operation)
    {
        return operation switch
        {
            BinaryOperation.Summation => _left + _right,
            BinaryOperation.Subtraction => _left - _right,
        };
    }
}
```

# ОСР

## пример соблюдения

```
public interface IBinaryOperation
{
    int Evaluate(int left, int right);
}

public class Summation : IBinaryOperation
{
    public int Evaluate(int left, int right)
        ⇒ left + right;
}

public class Subtraction : IBinaryOperation
{
    public int Evaluate(int left, int right)
        ⇒ left - right;
}
```

```
public sealed class BinaryOperand
{
    private readonly int _left;
    private readonly int _right;

    // ...

    public int Evaluate(IBinaryOperation operation)
        ⇒ operation.Evaluate(_left, _right);
}
```

проектирование типов, таким образом что их логику можно расширять, не изменяя их исходный код  
тип должен быть открытым для расширения, но закрытым для изменений



open/closed principle

# ОСР

## проверка соблюдения

представьте что разрабатываете библиотеку

если потребители могут её расширить без изменения исходников – ОСР соблюдается

если нет – не соблюдается

**liskov substitution principle**

# LSP

## пример несоблюдения

```
public record Coordinate(int X, int Y);

public class Creature
{
    public void Die()
    {
        Console.WriteLine("I am dead now");
    }
}

public class Bird : Creature
{
    public virtual void FlyTo(Coordinate coordinate)
    {
        coordinate
        Console.WriteLine("I am flying");
    }
}

public class Penguin : Bird
{
    public override void FlyTo(Coordinate coordinate)
    {
        Die();
    }
}
```

# LSP

## пример несоблюдения

```
StartMigration(birds, new Coordinate(420, 69));
```

```
void StartMigration(IEnumerable<Bird> birds, Coordinate coordinate)
{
    foreach (var bird in birds)
    {
        bird.FlyTo(coordinate);
    }
}
```

```
var birds = new[] { new Penguin() };
```

# LSP

## пример несоблюдения

```
public class Bat : Creature
{
    public void FlyTo(Coordinate coordinate)
    {
        Console.WriteLine("I bat and am flying");
    }
}
```

```
void StartMigration(
    IEnumerable<Creature> creatures,
    Coordinate coordinate)
{
    foreach (var creature in creatures)
    {
        if (creature is Bird bird)
        {
            bird.FlyTo(coordinate);
        }

        if (creature is Bat bat)
        {
            bat.FlyTo(coordinate);
        }
    }
}
```



# LSP

## пример соблюдения

```
public record Coordinate(int X, int Y);

public interface ICreature
{
    void Die();
}

public interface IFlyingCreature : ICreature
{
    void FlyTo(Coordinate coordinate);
}

public class CreatureBase : ICreature
{
    public void Die()
    {
        Console.WriteLine("I am dead now");
    }
}
```

```
public class Bird : CreatureBase { }

public class Penguin : Bird { }

public class Colibri : Bird, IFlyingCreature
{
    public void FlyTo(Coordinate coordinate)
    {
        Console.WriteLine("I am colibri and I'm flying");
    }
}

public class Bat : CreatureBase, IFlyingCreature
{
    public void FlyTo(Coordinate coordinate)
    {
        Console.WriteLine("I am bat and I'm flying");
    }
}
```

# LSP

## пример соблюдения

```
void StartMigration(IEnumerable<IFlyingCreature> creatures, Coordinate coordinate)
{
    foreach (var creature in creatures)
    {
        if (Random.Shared.NextDouble() < 0.8)
        {
            creature.FlyTo(coordinate);
        }
        else
        {
            creature.Die();
        }
    }
}

var creatures = new IFlyingCreature[] { new Colibri(), new Bat() };

StartMigration(creatures, new Coordinate(420, 69));
```

проектирование иерархий типов, таким образом  
что логика дочерних типов не нарушает инвариант  
и интерфейс родительских типов



**liskov substitution principle**

interface segregation principle

# ISP

## пример несоблюдения

```
public interface ICanAllDevice
{
    void Print();

    void PlayMusic();

    void BakeBread();
}
```

# ISP

## почему это плохо

абстракции обрастают лишними, не нужными для всех её пользователей, поведением

это приводит к большому пространству для ошибок

ISP – по факту SRP для интерфейсов, его нарушение, приводит к нарушению SRP у реализаций

# ISP

## пример соблюдения

```
public interface IPrinter
{
    void Print();
}
```

```
public interface IMusicPlayer
{
    void Play();
}
```

```
public interface IBakery
{
    void BakeBread();
}
```

проектирование маленьких абстракций, которые  
ответственны за свой конкретный функционал, а не  
одной всеобъемлющей, содержащий много различного



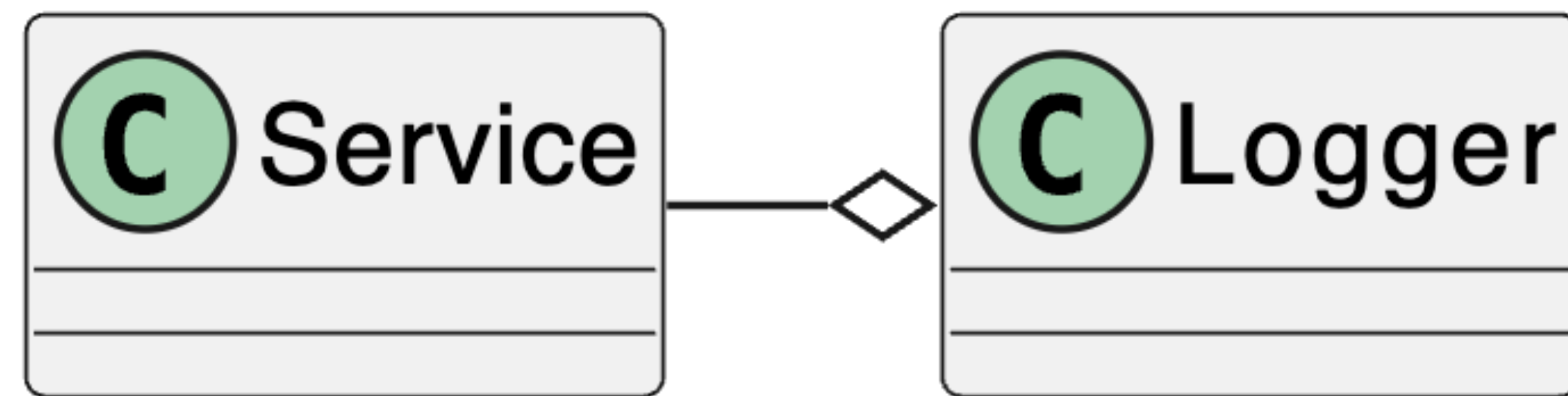
**interface segregation principle**



dependency inversion principle

# DIP

## пример несоблюдения

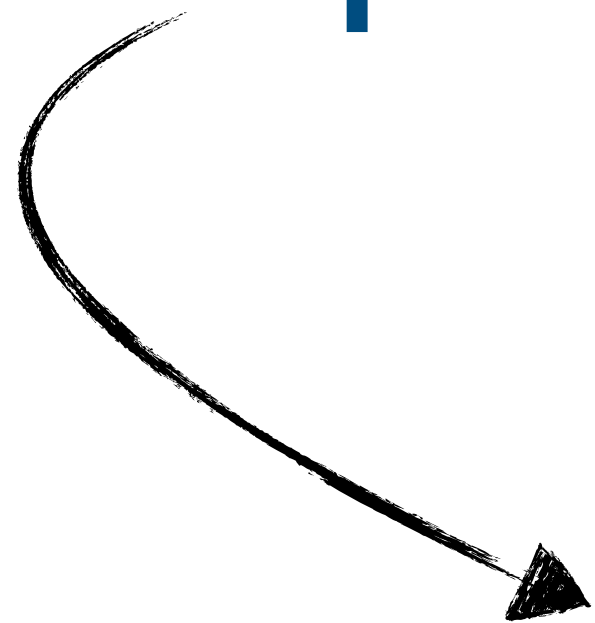


# DIP

пример соблюдения



**проектирование типов, таким образом что одни реализации не зависят от других напрямую**



**dependency inversion principle**

# DIP

## последствия несоблюдения

- сильная связанность между типами  
замена реализации требует изменения кода зависимого типа
- ограничивает возможности расширения типов
- сложности при тестировании  
явная зависимость не позволит провести по-настоящему  
изолированный тест