**Принцип единственной обязанности** (Single Responsibility Principle) можно сформулировать так:

**У класса должна быть только одна причина для изменения**

Под обязанностью здесь понимается набор функций, которые выполняют единую задачу. Суть этого принципа заключается в том, что класс должен выполнять одну единственную задачу. Весь функционал класса должен быть целостным, обладать высокой связностью (high cohesion).

Конкретное применение принципа зависит от контекста. В данном случае важно понимать, как изменяется класс. Если класс выполняет несколько различных функций, и они изменяются по отдельности, то это как раз тот случай, когда можно применить принцип единственной обязанности. То есть иными словами, у класса несколько причин для изменения.

Но если же все функции класса, как правило, изменяются вместе и составляют одно функциональное целое, решают одну задачу, то нет смысла применять данный принцип. Рассмотрим применение принципа на примере.

Допустим, нам надо определить класс отчета, по которому мы можем перемещаться по страницам и который можно выводить на печать. На первый взгляд мы могли бы определить следующий класс:

class Report

{

    public string Text { get; set; }

    public void GoToFirstPage()

    {

        Console.WriteLine("Переход к первой странице");

    }

    public void GoToLastPage()

    {

        Console.WriteLine("Переход к последней странице");

    }

    public void GoToPage(int pageNumber)

    {

        Console.WriteLine("Переход к странице {0}", pageNumber);

    }

    public void Print()

    {

        Console.WriteLine("Печать отчета");

        Console.WriteLine(Text);

    }

}

Первые три метода относятся к навигации по отчету и представляют одно единое функциональное целое. От них отличается метод Print, который производит печать. Что если нам понадобится печатать отчет на консоль или передать его на принтер для физической печати на бумаге? Или вывести в файл? Сохранить в формате html, txt, rtf и т.д.? Очевидно, что мы можем для этого поменять нужным образом метод Print(). Однако это вряд ли затронет остальные методы, которые относятся к навигации страницы.

Также верно и обратное - изменение методов постраничной навигации вряд ли повлияет на возможность вывода текста отчета на принтер или на консоль. Таким образом, у нас здесь прослеживаются две причины для изменения, значит, класс Report обладает двумя обязанностями, и от одной из них этот класс надо освободить.

В этом случае мы могли бы вынести функционал печати в отдельный класс, а потом применить агрегацию:

interface IPrinter

{

    void Print(string text);

}

class ConsolePrinter : IPrinter

{

    public void Print(string text)

    {

        Console.WriteLine(text);

    }

}

class Report

{

    public string Text { get; set; }

    public void GoToFirstPage()

    {

        Console.WriteLine("Переход к первой странице");

    }

    public void GoToLastPage()

    {

        Console.WriteLine("Переход к последней странице");

    }

    public void GoToPage(int pageNumber)

    {

        Console.WriteLine("Переход к странице {0}", pageNumber);

    }

    public void Print(IPrinter printer)

    {

        printer.Print(this.Text);

    }

}

Теперь объект Report получает ссылку на объект IPrinter, который используется для печати, и через метод Print выводится содержимое отчета:

|  |  |
| --- | --- |
| 1  2  3  4 | IPrinter printer = new ConsolePrinter();  Report report = new Report();  report.Text = "Hello Wolrd";  report.Print(printer); |

Побочным положительным действием является то, что теперь функционал печати инкапсулируется в одном месте, и мы сможем использовать его повторно для объектов других классов, а не только Report.

Однако обязанности в классах не всегда группируются по методам. Вполне возможно, что в одном методе сгруппировано несколько обязанностей. Например:

class Phone

{

    public string Model { get; set; }

    public int Price { get; set; }

}

class MobileStore

{

    List<Phone> phones = new List<Phone>();

    public void Process()

    {

        Console.WriteLine("Введите модель:");

        string model = Console.ReadLine();

        Console.WriteLine("Введите цену:");

        int price = 0;

        bool result = Int32.TryParse(Console.ReadLine(), out price);

        if (result == false || price <= 0  || String.IsNullOrEmpty(model))

        {

            throw new Exception("Некорректно введены данные");

        }

        else

        {

            phones.Add(new Phone { Model = model, Price = price });

            // сохраняем данные в файл

            using (System.IO.StreamWriter writer = new System.IO.StreamWriter("store.txt", true))

            {

                writer.WriteLine(model);

                writer.WriteLine(price);

            }

            Console.WriteLine("Данные успешно обработаны");

        }

    }

}

Класс имеет один единственный метод Process, однако этот небольшой метод, содержит в себе как минимум четыре обязанности: ввод данных, их валидация, создание объекта Phone и сохранение. В итоге класс знает абсолютно все: как получать данные, как валидировать, как сохранять. При необходимости в него можно было бы засунуть еще пару обязанностей. Такие классы еще называют "божественными" или "классы-боги", так как они инкапсулируют в себе абсолютно всю функциональность. Подобные классы являются одним из распространенных анти-паттернов, и их применения надо стараться избегать.

Хотя тут довольно немного кода, однако при последующих изменениях метод Process может быть сильно раздут, а функционал усложнен и запутан.

Теперь изменим код класса, инкапсулировав все обязанности в отдельных классах:

class Phone

{

    public string Model { get; set; }

    public int Price { get; set; }

}

class MobileStore

{

    List<Phone> phones = new List<Phone>();

    public IPhoneReader Reader { get; set; }

    public IPhoneBinder Binder { get; set; }

    public IPhoneValidator Validator { get; set; }

    public IPhoneSaver Saver { get; set; }

    public MobileStore(IPhoneReader reader, IPhoneBinder binder, IPhoneValidator validator, IPhoneSaver saver)

    {

        this.Reader = reader;

        this.Binder = binder;

        this.Validator = validator;

        this.Saver = saver;

    }

    public void Process()

    {

        string[] data = Reader.GetInputData();

        Phone phone = Binder.CreatePhone(data);

        if (Validator.IsValid(phone))

        {

            phones.Add(phone);

            Saver.Save(phone, "store.txt");

            Console.WriteLine("Данные успешно обработаны");

        }

        else

        {

            Console.WriteLine("Некорректные данные");

        }

    }

}

interface IPhoneReader

{

    string[] GetInputData();

}

class ConsolePhoneReader : IPhoneReader

{

    public string[] GetInputData()

   {

        Console.WriteLine("Введите модель:");

        string model = Console.ReadLine();

        Console.WriteLine("Введите цену:");

        string price = Console.ReadLine();

        return new string[] { model, price };

    }

}

interface IPhoneBinder

{

    Phone CreatePhone(string[] data);

}

class GeneralPhoneBinder : IPhoneBinder

{

    public Phone CreatePhone(string[] data)

    {

        if(data.Length>=2)

        {

            int price = 0;

            if(Int32.TryParse(data[1], out price))

            {

                return new Phone { Model = data[0], Price = price };

            }

            else

            {

                throw new Exception("Ошибка привязчика модели Phone. Некорректные данные для свойства Price");

            }

        }

        else

        {

            throw new Exception("Ошибка привязчика модели Phone. Недостаточно данных для создания модели");

        }

    }

}

interface IPhoneValidator

{

    bool IsValid(Phone phone);

}

class GeneralPhoneValidator : IPhoneValidator

{

    public bool IsValid(Phone phone)

    {

        if (String.IsNullOrEmpty(phone.Model) || phone.Price <= 0)

            return false;

        return true;

    }

}

interface IPhoneSaver

{

    void Save(Phone phone, string fileName);

}

class TextPhoneSaver : IPhoneSaver

{

    public void Save(Phone phone, string fileName)

    {

        using (System.IO.StreamWriter writer = new System.IO.StreamWriter(fileName, true))

        {

            writer.WriteLine(phone.Model);

            writer.WriteLine(phone.Price);

        }

    }

}

Теперь для каждой обязанности определен свой интерфейс. Конкретные реализации обязанностей устанавливаются в виде интерфейсов в целевом классе.

В то же время кода стало больше, в связи с чем программа усложнилась. И, возможно, подобное усложнение может показаться неоправданным при наличии одного небольшого метода, который необязательно будет изменяться. Однако при модификации стало гораздо проще вводить новый функционал без изменения существующего кода. А все части метода Process, будучи инкапсулированными во внешних классах, теперь не зависят друг от друга и могут изменяться самостоятельно.

**Распространенные случаи нарушения принципа SRP**

Нередко принцип единственной обязанности нарушает при смешивании в одном классе функциональности разных уровней. Например, класс производит вычисления и выводит их пользователю, то есть соединяет в себя бизнес-логику и работу с пользовательским интерфейсом. Либо класс управляет сохранением/получением данных и выполнением над ними вычислений, что также нежелательно. Класс слеует применять только для одной задачи - либо бизнес-логика, либо вычисления, либо работа с данными.

Другой распространенный случай - наличие в классе или его методах абсолютно несвязанного между собой функционала.