ЛЕКЦИЯ 12

- Модуль 2
- 13.12.2023
- SOLID

ЦЕЛИ ЛЕКЦИИ

- Обобщить знания, полученные по ООП
- Познакомиться с SOLID-принципами игровой разработки

SOLID

- **S** [Single Responsibility Principle] – принцип единственной обязанности
- **O** [Open-Closed Principle] – принцип открытости/закрытости
- **L** [Liskov Substitution Principle] – принцип подстановки Барбары Лисков
- **I** [Interface Segregation Principle] – принцип разделения интерфейсов
- **D** [Dependency Inversion Principle] – принцип инверсии зависимостей

S – THE SINGLE RESPONSIBILITY PRINCIPLE

- **Название:** Принцип единственной ответственности
- **Определение:** У класса/модуля должна быть лишь одна причина для изменения
- **Смысл принципа:** Борьба со сложностью, важность которой резко возрастает при развитии логики приложения
- **Краткое описание:** Любой сложный класс должен быть разбит на несколько простых составляющих, отвечающих за определенный аспект поведения, что упрощает как понимание, так и будущее развитие

Single Responsibility Principle (SRP) in 100 seconds (https://dev.to/richardwynn/single-responsibility-principle-srp-in-100-seconds-3b1d)

S – THE SINGLE RESPONSIBILITY PRINCIPLE

Примеры нарушения:

1. смешение логики и инфраструктуры: бизнес-логика смешана с представлением, слоем персистентности, находится внутри WCF или windows-сервисов и т.п.
2. класс/модуль решает задачи разных уровней абстракции: вычисляет CRC и отправляет уведомления по электронной почте; разбирает json-объект и анализирует его содержимое и т.п.

https://dev.to/richardwynn/single-responsibility-principle-srp-in-100-seconds-3b1d

Anti-SRP – Принцип размытой ответственности. Чрезмерная любовь к SRP ведет к обилию мелких классов/методов и размазыванию логики между ними

O – THE OPEN-CLOSED PRINCIPLE

- **Название:** принцип открытости / закрытости
- **Определение:** Программные сущности (классы, модули, функции и т.п.) должны быть открытыми для расширения, но закрытыми для модификации
- **Смысл принципа:**
  - ограничить распространение изменений минимальным числом классов/модулей
  - позволить вести параллельную разработку путем фиксации интерфейсов классов и открытости реализаций
- **Краткое описание:**
  - закрытость модулей означает стабильность интерфейса и возможность использования классов / модулей клиентами
  - открытость модулей означает возможность внесения изменений в поведении, путем изменения реализации или же путем переопределения поведения в наследниках
  - борьба с изменениями заключается в ограничении количества изменений минимальным числом классов / модулей и не подразумевает возможность изменения поведения без перекомпиляции
    - На практике требуемая «гибкость» обеспечивается за счет наследования и сопоставления с образцом (pattern matching), в зависимости от того, какую операцию мы хотим упростить – добавление нового подтипа в иерархию наследования или добавление новой операции в семейство типов

O – THE OPEN-CLOSED PRINCIPLE

Примеры нарушения:

- размазывание информации об иерархии типов по всему приложению

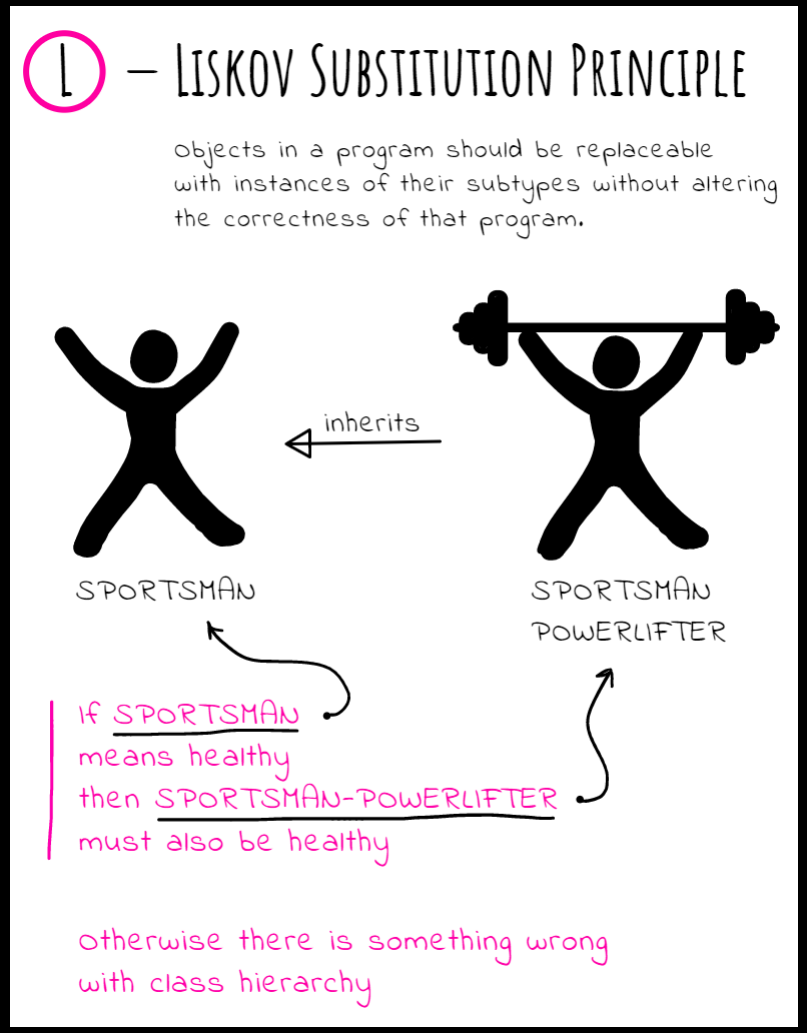
Open/closed principle [https://dev.to/satansdeer/openclosed-principle-86a]

Anti-OCP – Принцип фабрики-фабрик: Чрезмерная любовь к OCP ведет к переусложненным решениям с чрезмерным числом уровней абстракции

L – THE LISKOV SUBSTITUTION PRINCIPLE

- **Название:** Принцип подстановки Барбары Лисков
- **Определение:** Должна быть возможность вместо базового типа подставить любой его подтип
- **Смысл принципа:** Реализуйте наследование подтипов правильно
- **Краткое описание:**
  - для корректной реализации отношения ЯВЛЯЕТСЯ (IS A), наследник может ослаблять предусловие и усиливать постусловие (требовать меньше и гарантировать больше), при этом инварианты базового класса должны выполняться наследником
  - При нарушении этих правил подстановка экземпляров наследника в метод, принимающий базовый класс будет приводить к непредсказуемым последствиям

L – THE LISKOV SUBSTITUTION PRINCIPLE



Примеры нарушения:

- несогласованное поведение наследников, что приводит к необходимости приводить экземпляры базового класса к конкретным типам наследников

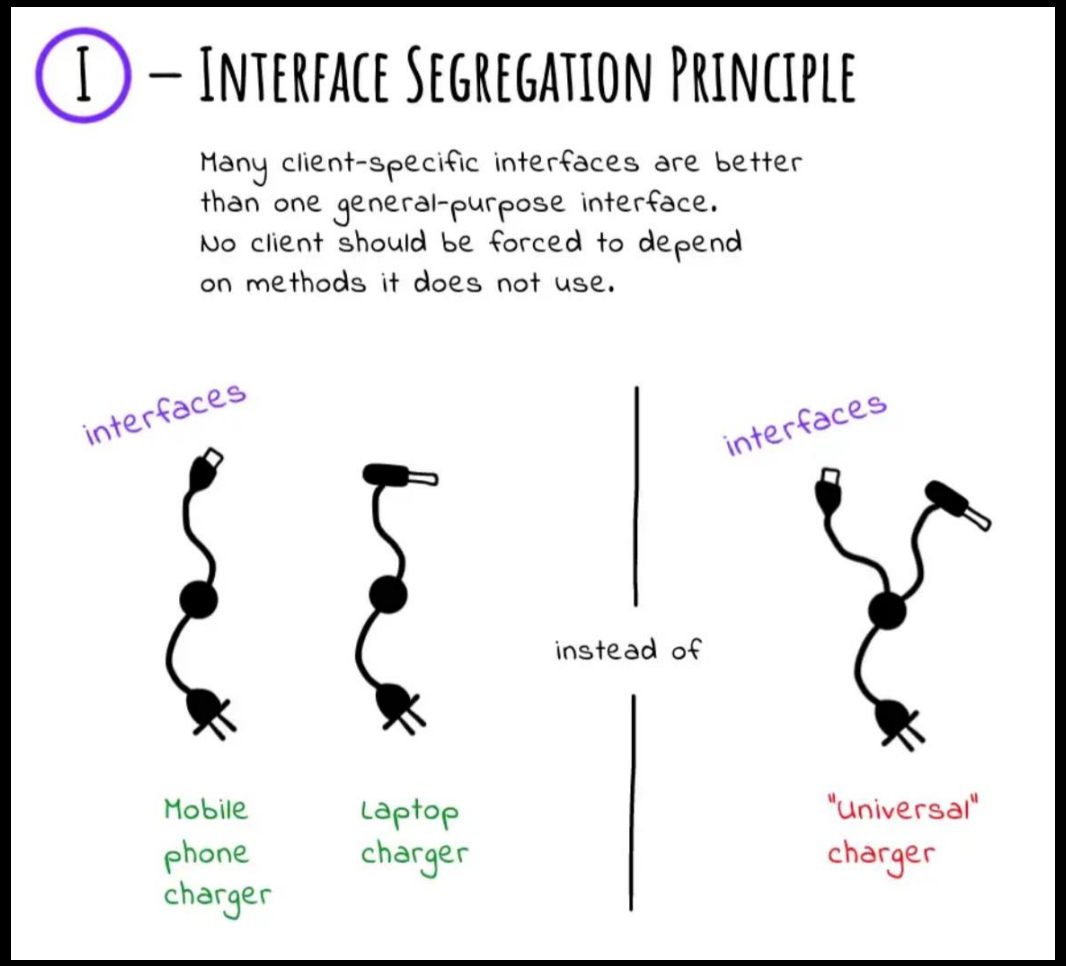
Anti-LSP – Принцип непонятного наследования. Данный анти-принцип проявляется либо в чрезмерном количестве наследования, либо в его полном отсутствии, в зависимости от опыта и взглядов местного главного архитектора

https://okso.app/showcase/solid/page/2899b427-2bd8-4899-16a7-0ce855331ba2

I – INTERFACE SEGREGATION PRINCIPLE

- **Название:** Принцип разделения интерфейсов
- **Определение:** клиенты не должны вынужденно зависеть от методов, которыми не пользуются
- **Смысл принципа:** класс должен предоставлять удобный интерфейс с точки зрения его разнообразных клиентов
- **Краткое описание:**
  - интерфейс класса должен быть цельным и согласованным не зависимо от числа клиентов
  - Несколько разных клиентов могут использовать лишь подмножество методов класса, до тех пор, пока интерфейс класса будет оставаться согласованным
  - Проблемы появляются тогда, когда интерфейс класса начинает распухать или появляются разные методы с похожей семантикой лишь для того, чтобы ими было удобно пользоваться определенным клиентам

I – INTERFACE SEGREGATION PRINCIPLE



https://itnext.io/solid-principles-sketches-a38865e771f0

Примеры нарушения:

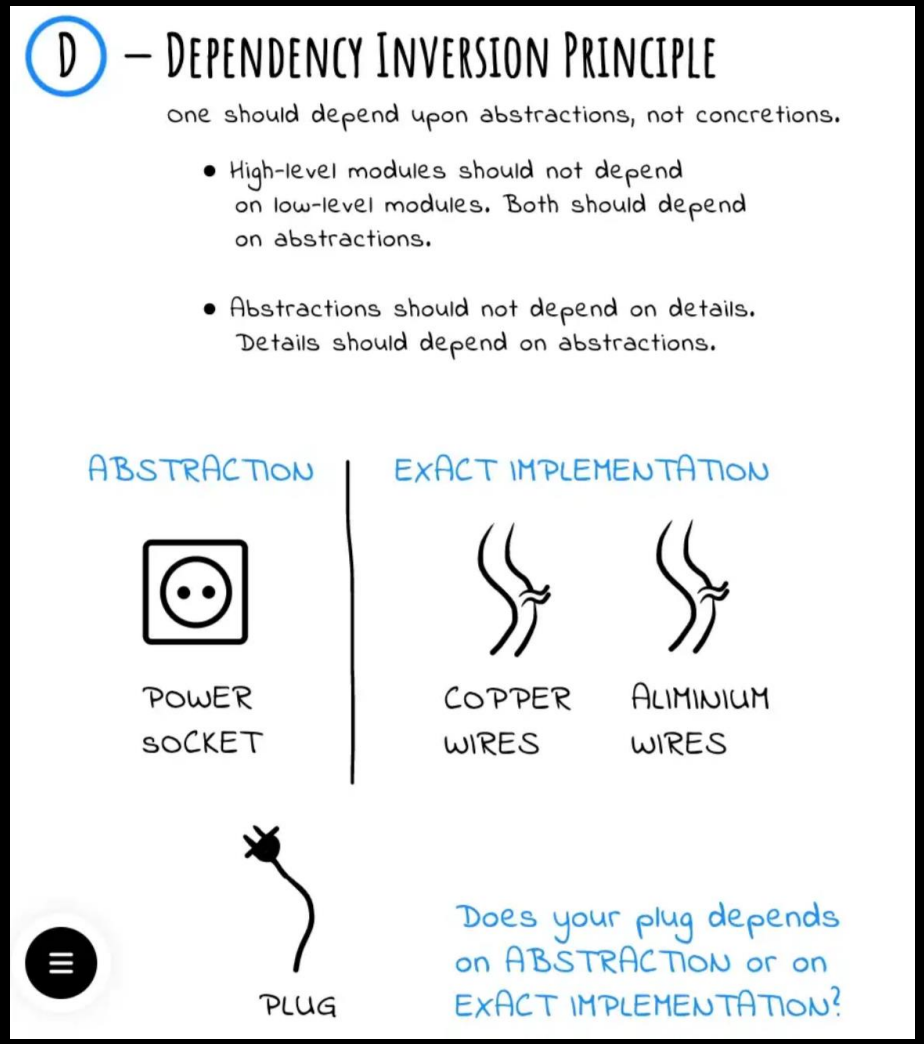
1. класс или интерфейс содержит несколько методов со схожей семантикой, которые используются разными клиентами
2. интерфейс класса слишком разнороден и содержит методы, отвечающие за слабосвязанные операции

Anti-ISP – Принцип тысячи интерфейсов. Интерфейсы классов разбиваются на слишком большое число составляющих, что делает их неудобными для использования всеми клиентами

D – THE DEPENDENCY INVERSION PRINCIPLE

- **Название:** принцип инверсии зависимостей
- **Определение:**
  - Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций
  - Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций
- **Смысл принципа:** сделать ключевые и/или изменчивые зависимости класса явными
- **Краткое описание:**
  - слишком большое число зависимостей класса говорит о проблемах в дизайне
    - Возможно класс делает слишком многое, или же текущий класс не удачен, что приводит к необходимости дергания по одному методу у слишком большого числа зависимостей
    - Любой объектный дизайн представляет собой некоторый граф взаимодействующих объектов, при этом некоторые зависимости являются частью реализации и должны создаваться напрямую (композиция), а некоторые – передаваться ему извне (агрегация)
    - Выделять зависимости особенно полезно, когда они являются изменчивыми (завязаны на окружения), или же представляют собой некоторую форму «стратегий»

D – THE DEPENDENCY INVERSION PRINCIPLE



Типичные примеры нарушения:

- использование синглтонов (одиночек), сервис-локаторов или же создание ключевых зависимостей класса по ходу дела в закрытых методах

Anti-DIP – Принцип инверсии сознания или DI-головного мозга. Интерфейсы выделяются для каждого класса и пачками передаются через конструкторы. Понять, где находится логика становится практически невозможно.

ПЕРЕДАЧА И УПРАВЛЕНИЕ ЗАВИСИМОСТЯМИ

Существует три понятия, связанных с передачей и управлением зависимостями, в каждом из которых есть слово “инверсия” (inversion) или “зависимость” (dependency):

- IoC – Inversion of Control (Инверсия управления)
- DI – Dependency Injection (Внедрение зависимостей)
- DIP – Dependency Inversion Principle (Принцип инверсии зависимостей)

DI vs. DIP vs. IoC [http://sergeyteplyakov.blogspot.com/2014/11/di-vs-dip-vs-ioc.html#more]

injection != inversion

Dependency Injection Is NOT The Same As The Dependency Inversion Principle (https://lostechies.com/derickbailey/2011/09/22/dependency-injection-is-not-the-same-as-the-dependency-inversion-principle/)

ПРИНЦИПЫ УПРАВЛЕНИЯ ЗАВИСИМОСТЯМИ

- важные зависимости должны быть явными
- минимизируем число зависимостей
  - если класс может о чем-то не знать, то избавьте его от этой лишней информации
- изменчивые зависимости должны быть выявлены и по возможности сделаны высокоуровневыми
- выделяем стратегии

Стремитесь к неизменяемости: объекты-значения (Value Objects) и функции без побочных эффектов – это идеальный строительный материал

DEPENDENCY INJECTION (DI) (ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ)

Внедрение зависимостей (DI, Dependency Injection) – это механизм передачи классу его зависимостей

Способы передачи информации о зависимостях реализуются шаблонами DI:

- Constructor injection (required dependency) – передача зависимостей через конструктор
- Setter injection (optional, may have default value) – передача зависимостей через сеттеры
- Constructor + setter (required and changeable)
- Method injection – передача зависимостей через аргументы метода

ПЕРЕДАЧА ЗАВИСИМОСТЕЙ ЧЕРЕЗ КОНСТРУКТОР

Цель шаблона Constructor injection – разорвать жесткую связь между классом и его обязательными зависимостями

Все зависимости, требуемые классу, передаются ему в качестве параметров конструктора, представленных в виде интерфейсов или абстрактных классов

- декоратор
- стратегия
- команда
- абстрактная фабрика
- состояние

DI паттерны. Constructor Injection [http://sergeyteplyakov.blogspot.com/2012/12/di-constructor-injection.html]

ПЕРЕДАЧА ЗАВИСИМОСТИ ЧЕРЕЗ СЕТТЕР

Цель паттерна – разорвать жесткую связь между классом и его необязательными зависимостями

Необязательная зависимость, требуемая некоторому классу, может быть определена вызывающим кодом путём установки ее через свойство

Изменяемость по своей природе усложняет код и серьезно влияет на простоту использования и сопровождаемость. При выборе этого паттерна вы должны себе отдавать отчет в том, что вы готовы на все это пойти и преимущества перевешивают недостатки

DI Паттерны. Property Injection (http://sergeyteplyakov.blogspot.com/2013/01/di-property-injection.html)

ПЕРЕДАЧА ЗАВИСИМОСТЕЙ ЧЕРЕЗ МЕТОД

Цель паттерна – предоставить классу сервиса дополнительную информацию для выполнения определённой задачи

Передача зависимости, необходимой для успешной работы метода, в него

Существует в двух видах:

1. Установка зависимостей объекта с помощью вызова метода (с параметром с типом интерфейса)
2. Передача зависимости в метод, чтобы он использовал эту зависимость для решения текущей задачи, а я не для сохранения в состоянии объекта и последующего использования
    1. Метод является статическим и другие варианты не подходят
    2. Зависимость может изменяться от операции к операции
    3. Передача локального контекста для выполнения операции

DI Паттерны. Method Injection (http://sergeyteplyakov.blogspot.com/2013/02/di-method-injection.html)

IOC CONTAINER

- Специального вида контейнер для упрощения создания фреймворков
- Объект IoC контейнер содержит множество зависимостей, которые внедряются в него (inject) или извлекаются (resolve)

ПРОВЕРЯЕМ СЕБЯ

ВОПРОС (1)

В программе описан класс A

```
class A {
    public short i, j = 0;
    int num;
    protected char ch, chTemp;
    private double account;
}
```

Какие члены будут доступны объектам любого производного от A класса?

ВОПРОС (2)

В результате выполнения следующего фрагмента кода:

```
using System;
class A
{
    public int[] ar;
    public A() { ar = new int[3]; }
    public A(A o) { ar = o.ar; }
}
class test_cs
{
    static void Main()
    {
        A obj = new A();
        obj.ar[0] = 1;
        A obj1 = new A(obj);
        obj.ar[0] = obj1.ar[2] = obj.ar.Length;
        foreach (int temp in obj1.ar)
            Console.Write(temp);
    }
}
```

на экран будет выведено:

ВОПРОС (3)

Базовым для перечисления может быть тип:

1) char
2) int
3) string
4) byte
5) sbyte

ВОПРОС (4)

При выполнении следующей программы:

```
using System;
class test_cs  {
    static void Main()     {
        uint n=3;
        string line = "";
        try {
            Console.Write("Введите натуральное число: ");
            line = Console.ReadLine();
            n = uint.Parse(line);
        }
        catch (Exception) {  Console.Write(line+n++);    }
        finally {  Console.Write(line+n++);   }
    }
}
```

пользователь в ответ на приглашение программы «Введите натуральное число: » ввел значение -4. Что будет выведено в консольное окно?

ВОПРОС (5)

В результате выполнения следующей программы:

```
using System;
  class Program  {
      static Program()     {
          ch = (char)(ch - j);
      }
      static string str = "AaBbCc";
      static char ch = str[j + 4];
      static void Main()    {
          Console.Write(ch + str + j);
      }
      static int j = 2;
}
```

на экран будет выведено:

ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

- Тепляков С. Шпаргалка по SOLID принципам (http://sergeyteplyakov.blogspot.com/2014/10/solid.html)
- Martin Fowler, Inversion of Control Containers and the Dependency Injection pattern, 2004
- Robert Martin, The Dependency Inversion Principle, C++ Report, 1996 (http://www.objectmentor.com/resources/articles/dip.pdf)
- Принципы SOLID в картинках (https://habr.com/ru/company/productivity_inside/blog/505430/)
- Solid Principles with C# .NET Core with Practical Examples & Interview Questions (https://procodeguide.com/design/solid-principles-with-csharp-net-core/)
- SOLID Principle Sketches (https://itnext.io/solid-principles-sketches-a38865e771f0)
- Dependency injection in ASP.NET Core (https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-7.0)
- Understanding SOLID Principles: Dependency Inversion (https://codeburst.io/understanding-solid-principles-dependency-injection-d570c15560ab)
- Корректный ASP.NET Core (https://habr.com/ru/post/437002/)
- How To Use Third Party (Ioc) Containers In ASP.NET Core MVC (https://www.c-sharpcorner.com/article/how-to-use-third-party-ioc-container-in-asp-net-core-mvc/)