**Курс " Javascript Fullstack Developer" и "Javascript Frontend"**

# **Кожевников Сергей 2024 год**

# **Задание 6**

Опциональное дз:

Выучите принципы SOLID наизусть.

Попробуйте по памяти воспроизвести примеры JS кода для демонстрации каждого из

этих принципов.

Выполните практические задачи по алгоритмам из Хендбука Яндекса

(https://education.yandex.ru/handbook/algorithms).

# 1) Реализовать алгоритм Флойда–Уоршелла или алгоритм Беллмана-Форда в графе (на матрице смежности или на списках смежности). 2) Реализовать с использованием TypeScript паттерны Строитель, Прокси, Медиатор. Привести примеры использования собственных классов. 3) Пройти хэндбук от Яндекса: <https://education.yandex.ru/handbook/algorithms>

# **Решение**

Выучите принципы SOLID наизусть.

Вот как расшифровывается акроним SOLID:

* S: Single Responsibility Principle (Принцип единственной ответственности).
* O: Open-Closed Principle (Принцип открытости-закрытости).
* L: Liskov Substitution Principle (Принцип подстановки Барбары Лисков).
* I: Interface Segregation Principle (Принцип разделения интерфейса).
* D: Dependency Inversion Principle (Принцип инверсии зависимостей).

Попробуйте по памяти воспроизвести примеры JS кода для демонстрации каждого из

этих принципов.

## **Принцип единственной ответственности**

Класс должен быть ответственен лишь за что-то одно. Если класс отвечает за решение нескольких задач, его подсистемы, реализующие решение этих задач, оказываются связанными друг с другом. Изменения в одной такой подсистеме ведут к изменениям в другой.

class Animal {

   constructor(name: string){ }

   getAnimalName() { }

}

class AnimalDB {

   getAnimal(a: Animal) { }

   saveAnimal(a: Animal) { }

}

## **Принцип открытости-закрытости**

Программные сущности (классы, модули, функции) должны быть открыты для расширения, но не для модификации.

class Animal {

       makeSound();

       //...

}

class Lion extends Animal {

   makeSound() {

       return 'roar';

   }

}

class Squirrel extends Animal {

   makeSound() {

       return 'squeak';

   }

}

class Snake extends Animal {

   makeSound() {

       return 'hiss';

   }

}

//...

function AnimalSound(a: Array<Animal>) {

   for(int i = 0; i <= a.length; i++) {

       a[i].makeSound();

   }

}

AnimalSound(animals);

## **Принцип подстановки Барбары Лисков**

Цель этого принципа заключаются в том, чтобы классы-наследники могли бы использоваться вместо родительских классов, от которых они образованы, не нарушая работу программы. Если оказывается, что в коде проверяется тип класса, значит принцип подстановки нарушается.

function AnimalLegCount(a: Array<Animal>) {

   for(let i = 0; i <= a.length; i++) {

**a[i].LegCount();**

   }

}

AnimalLegCount(animals);

class Animal {

       makeSound();

**LegCount();**

       //...

}

class Lion extends Animal {

   makeSound() {

       return 'roar';

   }

**LegCount();**

}

## **Принцип разделения интерфейса**

*Создавайте узкоспециализированные интерфейсы, предназначенные для конкретного клиента. Клиенты не должны зависеть от интерфейсов, которые они не используют.*  
  
Этот принцип направлен на устранение недостатков, связанных с реализацией больших интерфейсов.

interface Shape {

   draw();

}

interface ICircle {

   drawCircle();

}

interface ISquare {

   drawSquare();

}

interface IRectangle {

   drawRectangle();

}

interface ITriangle {

   drawTriangle();

}

class Circle implements ICircle {

   drawCircle() {

       //...

   }

}

class Square implements ISquare {

   drawSquare() {

       //...

   }

}

class Rectangle implements IRectangle {

   drawRectangle() {

       //...

   }

}

class Triangle implements ITriangle {

   drawTriangle() {

       //...

   }

}

class CustomShape implements Shape {

  draw(){

     //...

  }

}

## **Принцип инверсии зависимостей**

1. Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
2. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

interface Connection {

   request(url: string, opts:any);

}

class Http {

   constructor(private httpConnection: Connection) { }

   get(url: string , options: any) {

       this.httpConnection.request(url,'GET');

   }

   post() {

       this.httpConnection.request(url,'POST');

   }

   //...

}

class XMLHttpService implements Connection {

   const xhr = new XMLHttpRequest();

   //...

   request(url: string, opts:any) {

       xhr.open();

       xhr.send();

   }

}

class NodeHttpService implements Connection {

   request(url: string, opts:any) {

       //...

   }

}

class MockHttpService implements Connection {

   request(url: string, opts:any) {

       //...

   }

}