JavaScript - ООП

ООП:

- Инкапсуляция;
- Наследование;
- Полиморфизм;

Агрегация и композиция

Dependency Injection (DI)

Принципы:

- YAGNI;
- KISS;
- DRY;
- Separation of Concerns;
- Law of Demeter;
- SOLID.

Процедурное программирование

Процедурное программирование — это парадигма программирования, которая основана на концепции вызова процедур, также называемых подпрограммами, методами или функциями. Основная идея заключается в разделении программы на небольшие, повторно используемые и управляемые блоки кода, которые выполняют конкретные задачи.

Вот несколько ключевых характеристик процедурного программирования:

- **Модулярность:** Программа разбивается на модули или функции, каждая из которых выполняет определённую задачу.
- **Последовательное выполнение:** Код выполняется последовательно, строка за строкой, с возможностью передачи управления другим функциям.
- **Локальные и глобальные переменные:** Переменные могут быть объявлены внутри функций (локальные переменные) или вне функций (глобальные переменные).
- **Повторное использование кода:** Функции могут быть вызваны несколько раз в разных местах программы, что способствует повторному использованию кода.
- **Управление потоком выполнения:** Используются управляющие структуры, такие как циклы (for, while) и условные операторы (if, switch).

Объектно ориентированное программирование

Объектно-ориентированное программирование (ООП) — это парадигма программирования, основанная на концепции объектов, которые могут содержать данные и методы для их обработки.

Основные принципы ООП включают:

- **Классы и объекты (синтаксис реализации)**
- **Инкапсуляция**
- **Наследование**
- **Полиморфизм**
- **Абстракция**

Классы и объекты

Классы — это шаблоны для создания объектов. Они определяют свойства (данные) и методы (функции), которые объекты этого класса будут иметь.

Объекты — это экземпляры классов, которые могут обладать своим собственным состоянием и поведением.

```
// Определение класса "Car"
class Car {
    constructor(brand, model) {
        this.brand = brand;
        this.model = model;
    }

    // Метод для вывода информации о машине
    displayInfo() {
        console.log(`Марка: ${this.brand}, Модель: ${this.model}`);
    }
}

// Создание объекта класса "Car"
const myCar = new Car('Toyota', 'Corolla');
myCar.displayInfo();  // Марка: Toyota, Модель: Corolla
```

Инкапсуляция

Инкапсуляция — это принцип сокрытия внутреннего состояния объекта и предоставление доступа к этому состоянию только через публичные методы. Это помогает защитить данные объекта от некорректного использования и модификации извне.

Реализуется при помощи модификаторов доступности:

- Публичный метод/свойство
- Приватный метод/свойство
- Защищенный метод/свойство
- Статический метод/свойство

Публичный метод/свойство

Публичные методы и свойства доступны из любого места в коде, где доступен объект. Их можно вызывать и изменять как внутри класса, так и за его пределами.

```
class Example {
    constructor(value) {
        this.publicValue = value; // Публичное свойство
    }

    publicMethod() { // Публичный метод
        console.log('This is a public method.');
    }
}

const obj = new Example('Hello');
console.log(obj.publicValue); // Hello
obj.publicMethod(); // This is a public method.
```

Приватный метод/свойство

Приватные методы и свойства доступны только внутри класса, в котором они объявлены. В ES2022 и выше они обозначаются с помощью символа #.

```
class Example {
    #privateValue; // Приватное свойство

    constructor(value) {
        this.#privateValue = value;
    }

    #privateMethod() { // Приватный метод
        console.log('This is a private method.');
    }

    publicMethod() {
        console.log(this.#privateValue); // Доступ к приватному свойству
        this.#privateMethod(); // Вызов приватного метода
    }
}

const obj = new Example('Hello');
obj.publicMethod(); // Hello \n This is a private method
// console.log(obj.#privateValue); // Ошибка
```

Защищенный метод/свойство

Защищенные методы и свойства не имеют специальной синтаксической конструкции в JavaScript, но обычно обозначаются одним подчеркиванием (_). Они подразумевают, что эти элементы предназначены для использования только в классе и его подклассах, хотя технически доступны извне.

```
class Base {
    constructor(value) {
        this._protectedValue = value; // Защищенное свойство (по соглашению)
    }

    _protectedMethod() { // Защищенный метод (по соглашению)
        console.log('This is a protected method.');
    }
}

class Derived extends Base {
    useProtected() {
        console.log(this._protectedValue); // Доступ к защищенному свойству
        this._protectedMethod(); // Вызов защищенного метода
    }
}

const obj = new Derived('Hello');
obj.useProtected(); // Hello \n This is a protected method
```

Статический метод/свойство

Статические методы и свойства принадлежат самому классу, а не его экземплярам. Их можно вызывать и использовать без создания экземпляра класса.

```
class Example {
    static staticValue = 'This is a static value'; // Статическое свойство

    static staticMethod() { // Статический метод
        console.log('This is a static method.');
    }
}

console.log(Example.staticValue); // This is a static value
Example.staticMethod(); // This is a static method

const obj = new Example();
// obj.staticMethod(); // Ошибка: obj.staticMethod is not a function
```

Наследование

Наследование — это один из основных принципов объектно-ориентированного программирования (ООП), который позволяет создавать новые классы на основе существующих. Это способствует повторному использованию кода, упрощает его поддержку и позволяет создавать иерархии классов.

Основные аспекты наследования:

- **Переиспользование кода:** Дочерние классы (производные классы) могут наследовать свойства и методы родительских классов (базовых классов), что позволяет избегать дублирования кода.
- **Расширяемость:** Дочерние классы могут добавлять свои собственные свойства и методы или переопределять методы родительского класса для реализации специфичного поведения.
- **Иерархия классов:** Наследование позволяет организовывать классы в иерархии, что помогает лучше структурировать код и моделировать реальный мир.

```
// Базовый класс (родительский класс)
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a noise.`);
    }
}
// Производный класс (дочерний класс) - Dog
class Dog extends Animal {
    speak() {
        console.log(`${this.name} barks.`);
    }
}
// Производный класс (дочерний класс) - Cat
class Cat extends Animal {
    speak() {
        console.log(`${this.name} meows.`);
    }
}
// Создаем экземпляры классов
const dog = new Dog('Rex');
const cat = new Cat('Whiskers');
// Используем методы
dog.speak(); // Rex barks.
cat.speak(); // Whiskers meows.
```

Полиморфизм

Полиморфизм — это один из ключевых принципов объектно-ориентированного программирования (ООП), который позволяет объектам разных классов обрабатывать данные через общий интерфейс. Он обеспечивает возможность использования одних и тех же методов для объектов различных типов, что делает код более гибким и расширяемым.

Основные аспекты полиморфизма:

- **Подмена типов:** Полиморфизм позволяет использовать экземпляры подклассов там, где ожидаются экземпляры родительского класса. Это позволяет создавать более общие функции и методы.
- **Методы с одинаковыми именами:** Разные классы могут иметь методы с одинаковыми именами, но с различной реализацией. Это называется переопределением методов.
- **Интерфейсы и абстрактные классы:** Полиморфизм часто реализуется через интерфейсы и абстрактные классы, что позволяет определить общий интерфейс для различных реализаций.

```
// Базовый класс (родительский класс)
class Animal {
    speak() {
        console.log('Animal makes a sound');
    }
}
// Производный класс (дочерний класс) - Dog
class Dog extends Animal {
    speak() {
        console.log('Dog barks');
    }
}
// Производный класс (дочерний класс) - Cat
class Cat extends Animal {
    speak() {
        console.log('Cat meows');
    }
}
// Функция, использующая полиморфизм
function makeAnimalSpeak(animal) {
    animal.speak(); // Вызов метода speak
}
// Создаем экземпляры классов
const dog = new Dog();
const cat = new Cat();
// Используем полиморфизм
makeAnimalSpeak(dog); // Dog barks
makeAnimalSpeak(cat); // Cat meows
```

Абстракция

Абстракция в объектно-ориентированном программировании (ООП) — это процесс выделения общих характеристик и поведения объектов, при этом скрывая сложные детали реализации. Она позволяет сосредоточиться на том, что объект делает, а не на том, как он это делает.

Основные аспекты абстракции:

- **Сокрытие деталей:** Абстракция помогает скрыть ненужные детали реализации, предоставляя пользователю только важные интерфейсы. Это упрощает взаимодействие с объектами.
- **Общие интерфейсы:** Абстракция позволяет создавать общие интерфейсы для различных объектов. Например, можно определить метод draw() для всех графических фигур, не вдаваясь в детали реализации каждой фигуры.
- **Упрощение разработки:** Разделяя интерфейс и реализацию, разработчики могут изменять реализацию без изменения интерфейса, что облегчает поддержку и расширение кода.

```
// Абстрактный класс (по соглашению)
class Shape {
    constructor() {
        if (new.target === Shape) {
            throw new Error("Cannot instantiate abstract class Shape");
        }
    }

    // Абстрактный метод
    area() {
        throw new Error("Method 'area()' must be implemented.");
    }
}
// Производный класс
class Circle extends Shape {
    constructor(radius) {
        super();
        this.radius = radius;
    }

    area() {
        return Math.PI * this.radius ** 2;
    }
}
// Другой производный класс
class Square extends Shape {
    constructor(side) {
        super();
        this.side = side;
    }

    area() {
        return this.side ** 2;
    }
}
// Использование
const circle = new Circle(5);
console.log(circle.area()); // 78.53981633974483
const square = new Square(4);
console.log(square.area()); // 16
// const shape = new Shape(); // Ошибка: Cannot instantiate abstract class Shape
```

TypeScript и JavaScript

JavaScript является не строго типизированным языком

TypeScript — это открытый язык программирования, основанный на JavaScript, который добавляет статическую типизацию и другие возможности.

В JavaScript нет явных абстракций (есть только по соглашению)

В TypeScript есть явные абстракции в виде типов и интерфейсов

Исходя из этого ООП в JavaScript не имеет явного преимущества в виде использования принципа абстракций, но данную возможность предоставляет TypeScript.

Сравнение на примере

Рассмотрим задачу управления списком студентов, включая добавление, удаление и отображение студентов. Мы решим эту задачу сначала с использованием процедурного программирования, а затем с использованием объектно-ориентированного программирования.

Процедурное программирование

- Логика разделена на функции, которые манипулируют глобальным состоянием (массив students).
- Все функции и данные находятся в глобальном пространстве имен.

```
// Массив для хранения студентов
let students = [];

// Функция для добавления студента
function addStudent(name) {
    students.push(name);
}

// Функция для удаления студента
function removeStudent(name) {
    const index = students.indexOf(name);
    if (index !== -1) {
        students.splice(index, 1);
    }
}

// Функция для отображения всех студентов
function displayStudents() {
    console.log("Список студентов:");
    students.forEach(student => console.log(student));
}

// Пример использования
addStudent("Алиса");
addStudent("Боб");
displayStudents();
removeStudent("Боб");
displayStudents();
```

Объектно-ориентированное программирование

- Логика и данные инкапсулированы внутри класса StudentList.
- Методы класса предоставляют интерфейс для работы со списком студентов.
- Создание экземпляра класса позволяет легко создавать несколько независимых списков студентов.

```
// Класс "StudentList" для управления списком студентов
class StudentList {
    constructor() {
        this.students = [];
    }

    addStudent(name) {
        this.students.push(name);
    }

    removeStudent(name) {
        const index = this.students.indexOf(name);
        if (index !== -1) {
            this.students.splice(index, 1);
        }
    }

    displayStudents() {
        console.log("Список студентов:");
        this.students.forEach(student => console.log(student));
    }
}

// Пример использования
const studentList = new StudentList();
studentList.addStudent("Алиса");
studentList.addStudent("Боб");
studentList.displayStudents();
studentList.removeStudent("Боб");
studentList.displayStudents();
```

Сравнение

Обе подхода решают одну и ту же задачу, но используют разные парадигмы программирования. ООП обеспечивает большую модульность и повторное использование кода, в то время как процедурное программирование более прямолинейно и может быть проще для небольших задач.

Агрегация и композиция

Агрегация и композиция — это два важных понятия в объектно-ориентированном программировании, которые описывают отношения между объектами. Оба подхода позволяют моделировать сложные структуры, но делают это по-разному.

Агрегация и композиция — это важные концепции для моделирования отношений между объектами в программировании. Понимание различий между ними позволяет разработчикам создавать более организованный и поддерживаемый код, выбирая подходящий метод в зависимости от требований к взаимосвязям между объектами.

Параметр	Агрегация	Композиция
Связь	Слабая (части могут существовать отдельно)	Сильная (части не могут существовать без целого)
Уничтожение	Части могут жить независимо	Части уничтожаются вместе с целым
Пример	Отдел и сотрудники	Автомобиль и двигатель

Агрегация

Агрегация — это тип отношения "часть-целое", при котором одна сущность (целое) состоит из одной или нескольких других сущностей (частей), но части могут существовать независимо от целого. То есть, части могут быть созданы и уничтожены отдельно от целого.

Пример:

Представим, что у нас есть класс Department (отдел) и класс Employee (сотрудник). Один отдел может иметь несколько сотрудников, но сотрудники могут существовать и без отдела.

```
class Employee {
    constructor(name) {
        this.name = name;
    }
}

class Department {
    constructor(name) {
        this.name = name;
        this.employees = []; // Массив сотрудников
    }

    addEmployee(employee) {
        this.employees.push(employee);
    }
}

// Использование
const emp1 = new Employee("Alice");
const emp2 = new Employee("Bob");
const department = new Department("HR");
department.addEmployee(emp1);
department.addEmployee(emp2);
```

Композиция

Композиция — это более строгий тип отношения "часть-целое", при котором часть не может существовать без целого. Если целое уничтожается, то и его части также уничтожаются. Это создаёт более сильную связь между объектами.

Пример:

Представим класс Car (автомобиль) и класс Engine (двигатель). Двигатель не имеет смысла без автомобиля, и если автомобиль уничтожается, то его двигатель тоже.

```
class Engine {
    constructor(horsepower) {
        this.horsepower = horsepower;
    }
}

class Car {
    constructor(make, model, horsepower) {
        this.make = make;
        this.model = model;
        this.engine = new Engine(horsepower); // Композиция
    }
}

// Использование
const myCar = new Car("Toyota", "Camry", 268);
// Если myCar будет уничтожен, то и его двигатель тоже.
```

Dependency Injection (DI)

Внедрение зависимостей (Dependency Injection, DI) — это шаблон проектирования, который позволяет управлять зависимостями между классами и объектами. Основная идея заключается в том, что объекты не создают свои зависимости самостоятельно, а получают их извне, что упрощает тестирование, поддержку и изменение кода.

Основные концепции внедрения зависимостей

- Зависимости: Это объекты, от которых зависит другой объект. Например, если класс Car зависит от класса Engine, то Engine является зависимостью для Car.
- Инъекция зависимостей: Зависимости передаются (или "внедряются") в класс через его конструктор, методы или свойства, вместо того чтобы создавать их внутри класса.

Преимущества внедрения зависимостей

- Упрощение тестирования: Можно легко подменять зависимости на моки или стабы, что упрощает модульное тестирование.
- Снижение связанности: Компоненты становятся менее зависимыми друг от друга, что упрощает модификацию и замену.
- Улучшение читаемости: Ясно, какие зависимости требуются классу, что делает код более понятным.
- Гибкость: Легко заменять зависимости на альтернативные реализации без изменения кода самого класса.

Примеры внедрения зависимостей

- Конструкторная инъекция
- Методическая инъекция
- Свойственная инъекция

Конструкторная инъекция (Constructor Injection)

Конструкторная инъекция — это способ внедрения зависимостей, при котором зависимости передаются в класс через его конструктор. Это позволяет классу использовать эти зависимости в своих методах.

Объяснение работы примера

- **Интерфейс UserRepository:** Это класс, который содержит методы для работы с данными пользователей.
- **Класс UserService:** Этот класс использует UserRepository для получения данных пользователей. Он принимает UserRepository как зависимость через свой конструктор.
- **Создание экземпляров:** При создании экземпляра UserService мы передаем UserRepository. Это позволяет UserService использовать методы UserRepository без необходимости его создания внутри себя.

```
// Интерфейс репозитория пользователей
class UserRepository {
    getUser(id) {
        // Логика получения пользователя из базы данных
        console.log(`Getting user with id: ${id}`);
    }
}

// Сервис, который зависит от UserRepository
class UserService {
    constructor(userRepository) {
        this.userRepository = userRepository; // Внедрение зависимости через конструктор
    }

    getUser(id) {
        return this.userRepository.getUser(id);
    }
}

// Использование
const userRepository = new UserRepository();
const userService = new UserService(userRepository); // Внедрение зависимости

userService.getUser(1); // Getting user with id: 1
```

Методическая инъекция (Method Injection)

Методическая инъекция — это способ внедрения зависимостей, при котором зависимости передаются в методы класса вместо конструктора. Это позволяет передавать необходимые зависимости только тогда, когда они нужны.

Объяснение работы примера

- **Интерфейс UserRepository:** Этот класс содержит метод для получения пользователя по идентификатору.
- **Класс UserService:** Вместо того чтобы хранить UserRepository как состояние класса, метод getUser принимает его как аргумент. Это позволяет передавать различные реализации репозитория.
- **Создание экземпляров:** При вызове метода getUser мы передаем конкретную зависимость UserRepository.

```
class UserRepository {
    getUser(id) {
        console.log(`Getting user with id: ${id}`);
    }
}

class UserService {
    getUser(id, userRepository) {
        return userRepository.getUser(id); // Внедрение зависимости в метод
    }
}

// Использование
const userRepository = new UserRepository();
const userService = new UserService();

userService.getUser(1, userRepository); // Getting user with id: 1
```

Свойственная инъекция (Property Injection)

Свойственная инъекция — это метод внедрения зависимостей, при котором зависимости устанавливаются через свойства объекта после его создания. Это позволяет добавлять зависимости в объект в любое время.

Объяснение работы примера

- **Интерфейс UserRepository:** Класс, который содержит метод для получения пользователя.
- **Класс UserService:** Имеет свойство userRepository, которое инициализируется как null. Метод setUserRepository устанавливает зависимость.
- **Создание экземпляров:** Объект UserService создается без зависимостей, и UserRepository устанавливается позже.

```
class UserRepository {
    getUser(id) {
        console.log(`Getting user with id: ${id}`);
    }
}

class UserService {
    constructor() {
        this.userRepository = null; // Инициализация свойства
    }

    setUserRepository(userRepository) {
        this.userRepository = userRepository; // Установка зависимости
    }

    getUser(id) {
        if (!this.userRepository) {
            throw new Error("UserRepository is not set.");
        }
        return this.userRepository.getUser(id);
    }
}

// Использование
const userRepository = new UserRepository();
const userService = new UserService();

userService.setUserRepository(userRepository); // Внедрение зависимости
userService.getUser(1); // Getting user with id: 1
```

Основные принципы программирования

- YAGNI
- KISS
- DRY
- Separation of Concerns (Разделение обязанностей)
- Law of Demeter (Закон Деметры)
- SOLID

YAGNI (You Aren't Gonna Need It)

YAGNI — это принцип, который утверждает, что разработчики не должны добавлять функциональность, которая на данный момент не нужна. Он напоминает, что необходимо сосредоточиться только на текущих требованиях и избегать создания избыточного кода.

Преимущества:

- Упрощает код, делает его более понятным и легким в поддержке.
- Снижает риск возникновения ошибок и неопределенности.
- Позволяет сосредоточиться на текущих требованиях и потребностях проекта.

```
// Пример с ненужной функциональностью
class User {
    constructor(name) {
        this.name = name;
        this.isAdmin = false; // Зачем добавлять, если это не нужно сейчас?
    }

    // Код для администраторских функций, который еще не нужен
    grantAdminRights() {
        this.isAdmin = true;
    }
}

// Лучше просто реализовать, когда это потребуется
class User {
    constructor(name) {
        this.name = name;
    }
}
```

KISS (Keep It Simple, Stupid)

KISS — это принцип, который гласит, что системы и решения должны быть простыми. Сложные решения увеличивают вероятность ошибок и усложняют понимание кода.

Преимущества:

- Упрощает отладку и тестирование.
- Повышает скорость разработки, так как разработчики могут быстрее понять и изменить код.
- Уменьшает риск возникновения ошибок, связанных с избыточной сложностью.

```
// Сложный код
function calculateTotal(items) {
    let total = 0;
    items.forEach(item => {
        if (item.price > 0) {
            total += item.price * item.quantity;
        } else {
            throw new Error("Invalid item price");
        }
    });
    return total;
}

// Простой код
function calculateTotal(items) {
    return items.reduce((total, item) => total + item.price * item.quantity, 0);
}
```

DRY (Don't Repeat Yourself)

DRY — это принцип, который гласит, что код не должен дублироваться. Если один и тот же фрагмент кода встречается в нескольких местах, это может привести к ошибкам и трудностям при его изменении.

Преимущества:

- Упрощает внесение изменений, так как нужно изменять код только в одном месте.
- Улучшает читабельность и поддержку кода.
- Снижает вероятность ошибок, связанных с несоответствием.

Пример DRY

```
function createUser(name, age) {
    const user = { name: name, age: age };
    console.log(`User created: ${user.name}, Age: ${user.age}`);
    return user;
}

function updateUser(user, newName, newAge) {
    user.name = newName;
    user.age = newAge;
    console.log(`User updated: ${user.name}, Age: ${user.age}`);
}

function displayUser(user) {
    console.log(`User: ${user.name}, Age: ${user.age}`);
}

// Использование
const user1 = createUser("Alice", 30);
updateUser(user1, "Alicia", 31);
displayUser(user1);
```


```
function createUser(name, age) {
    const user = { name: name, age: age };
    console.log(`User created: ${user.name}, Age: ${user.age}`);
    return user;
}

function updateUser(user, newName, newAge) {
    user.name = newName;
    user.age = newAge;
    console.log(`User updated: ${user.name}, Age: ${user.age}`);
}

function displayUser(user) {
    console.log(`User: ${user.name}, Age: ${user.age}`);
}

// Использование
const user1 = createUser("Alice", 30);
updateUser(user1, "Alicia", 31);
displayUser(user1);
```

Что изменилось?

Вынесение логики: Мы вынесли логику вывода информации о пользователе в отдельную функцию logUserInfo. Теперь мы можем переиспользовать ее в различных функциях, что уменьшает дублирование кода.

Упрощение изменений: Если мы захотим изменить формат вывода или добавить новую логику, нам нужно сделать это только в одном месте — в функции logUserInfo.

Separation of Concerns (Разделение обязанностей)

Разделение обязанностей — это принцип, который утверждает, что различные аспекты функциональности программы должны быть разделены на отдельные модули или компоненты. Это позволяет упростить разработку, тестирование и поддержку.

Каждая часть программы должна быть ответственна за конкретную задачу или аспект. Это может быть разделение на слои (например, представление, бизнес-логика, доступ к данным) или на модули (например, обработка платежей, аутентификация, управление пользователями).

Преимущества:

- Упрощает понимание системы, так как каждая часть отвечает за свою функциональность.
- Облегчает тестирование и отладку, так как можно тестировать каждую часть отдельно.
- Позволяет более гибко изменять и масштабировать приложение, добавляя новые модули или изменяя существующие.

```
// Модель
class Task {
    constructor(title) {
        this.title = title;
        this.completed = false;
    }

    complete() {
        this.completed = true;
    }
}
// Сервис
class TaskService {
    constructor() {
        this.tasks = [];
    }

    addTask(task) {
        this.tasks.push(task);
    }

    getTasks() {
        return this.tasks;
    }
}
// Контроллер
class TaskController {
    constructor(taskService) {
        this.taskService = taskService;
    }

    addTask(title) {
        const task = new Task(title);
        this.taskService.addTask(task);
    }

    listTasks() {
        return this.taskService.getTasks();
    }
}
// Презентация (например, в UI)
const taskService = new TaskService();
const taskController = new TaskController(taskService);
taskController.addTask("Buy milk");
console.log(taskController.listTasks());
```

Law of Demeter (Закон Деметры)

Закон Деметры, также известный как "принцип наименьшей осведомленности", гласит, что объект должен взаимодействовать только с непосредственными зависимостями, а не с объектами, которые они возвращают. Это означает, что объект не должен "знать" о внутренностях других объектов.

Если класс A зависит от класса B, а класс B зависит от класса C, то класс A не должен напрямую взаимодействовать с классом C. Вместо этого класс A должен взаимодействовать только с классом B. Это помогает уменьшить связанность между классами и упрощает их изменение и тестирование.

Преимущества:

- Уменьшает связанность между классами.
- Упрощает понимание и поддержку кода.
- Облегчает модульное тестирование, поскольку объекты имеют меньшую зависимость от других объектов.

```
class Engine {
    start() {
        console.log("Engine starting...");
    }
}

class Car {
    constructor() {
        this.engine = new Engine();
    }

    start() {
        this.engine.start();
    }
}

// Неправильный подход
class Driver {
    constructor(car) {
        this.car = car;
    }

    drive() {
        this.car.engine.start(); // Прямое взаимодействие с внутренностью Car
    }
}

// Правильный подход
class Driver {
    constructor(car) {
        this.car = car;
    }

    drive() {
        this.car.start(); // Взаимодействуем только с Car
    }
}
```

SOLID

Принципы SOLID — это набор пяти принципов объектно-ориентированного проектирования, которые помогают разработчикам создавать более гибкие, поддерживаемые и понятные системы.

Принципы SOLID помогают разработчикам создавать более чистый, понятный и поддерживаемый код. Их соблюдение приводит к уменьшению связанности, повышению гибкости и упрощению модульного тестирования, что делает архитектуру приложения более устойчивой к изменениям.

Расшифровка:

- Single responsibility — принцип единственной ответственности
- Open-closed — принцип открытости / закрытости
- Liskov substitution — принцип подстановки Барбары Лисков
- Interface segregation — принцип разделения интерфейса
- Dependency inversion — принцип инверсии зависимостей

Single Responsibility Principle (SRP)

Принцип единственной ответственности

Каждый класс должен иметь одну и только одну причину для изменения. Это значит, что класс должен отвечать только за одну задачу или функциональность.

Преимущества:

- Упрощает поддержку и тестирование.
- Снижает связанность и повышает модульность.

Пример нарушения принципа SRP

Представим класс, который отвечает за управление пользователями и отправку уведомлений. Он делает больше одной задачи, что нарушает SRP.

Проблемы с таким подходом

1. Несоответствие принципу SRP: Класс User отвечает и за сохранение данных, и за уведомление. Если потребуется изменить логику уведомлений, придется изменять класс User.
2. Сложность тестирования: Тестировать сохранение и уведомление вместе сложнее.

```
class User {
    constructor(name) {
        this.name = name;
    }

    save() {
        // Логика сохранения пользователя в базе данных
        console.log(`User ${this.name} saved.`);
    }

    notify() {
        // Логика уведомления пользователя
        console.log(`Notify ${this.name}.`);
    }
}

// Использование
const user = new User("Alice");
user.save();
user.notify();
```

Правильное решение с соблюдением принципа SRP

Теперь мы разделим ответственность на два отдельных класса: один для управления пользователями, а другой — для уведомлений.

Преимущества нового подхода

- Соблюдение SRP: Каждый класс теперь отвечает только за свою задачу.
- Упрощение поддержки и тестирования: Изменения в логике уведомлений не затрагивают класс User, и наоборот.
- Чистота кода: Код становится более читаемым и структурированным, что упрощает его понимание.

Следуя принципу SRP, мы создаем более устойчивую и легко поддерживаемую архитектуру.

```
class User {
    constructor(name) {
        this.name = name;
    }

    save() {
        // Логика сохранения пользователя в базе данных
        console.log(`User ${this.name} saved.`);
    }
}

class UserNotifier {
    notify(user) {
        // Логика уведомления пользователя
        console.log(`Notify ${user.name}.`);
    }
}

// Использование
const user = new User("Alice");
user.save();

const notifier = new UserNotifier();
notifier.notify(user);
```

Open/Closed Principle (OCP)

Принцип открытости/закрытости

Сущности (классы, модули, функции и т.д.) должны быть открыты для расширения, но закрыты для модификации. Это позволяет добавлять новую функциональность без изменения существующего кода.

Преимущества:

- Упрощает добавление новых функций.
- Минимизирует риск появления ошибок в существующем коде.

Пример нарушения принципа OCP

Предположим, у нас есть класс, который вычисляет площадь фигур. Если мы добавим новую фигуру, нам нужно будет изменить существующий код.

Проблемы с таким подходом

1. Несоответствие принципу OCP: Каждый раз, когда мы добавляем новую фигуру, нам нужно изменять класс AreaCalculator.
2. Риск ошибок: Изменения в одном месте могут привести к ошибкам в другом.

```
class Rectangle {
    constructor(width, height) {
        this.width = width;
        this.height = height;
    }

    area() {
        return this.width * this.height;
    }
}

class Circle {
    constructor(radius) {
        this.radius = radius;
    }

    area() {
        return Math.PI * this.radius * this.radius;
    }
}

class AreaCalculator {
    calculate(shapes) {
        let totalArea = 0;
        for (const shape of shapes) {
            if (shape instanceof Rectangle) {
                totalArea += shape.area();
            } else if (shape instanceof Circle) {
                totalArea += shape.area();
            }
        }
        return totalArea;
    }
}

// Использование
const shapes = [new Rectangle(10, 20), new Circle(5)];
const calculator = new AreaCalculator();
console.log(calculator.calculate(shapes)); // Выводит общую площадь
```

Правильное решение с соблюдением принципа OCP

Теперь давайте сделаем так, чтобы наш код был открыт для расширения, но закрыт для модификации. Мы создадим базовый класс Shape и будем использовать полиморфизм.

Преимущества нового подхода

- Соблюдение OCP: Теперь мы можем добавлять новые фигуры, создавая новые классы, не изменяя существующий код AreaCalculator.
- Улучшение расширяемости: Легко добавлять новые фигуры, просто создавая новый класс, который наследует Shape.
- Снижение риска ошибок: Меньше изменений в существующем коде означает меньше шансов на введение ошибок.

Следуя принципу OCP, мы создаем более гибкую и устойчивую архитектуру.

```
class Shape {
    area() {
        throw new Error("Method not implemented.");
    }
}

class Rectangle extends Shape {
    constructor(width, height) {
        super();
        this.width = width;
        this.height = height;
    }

    area() {
        return this.width * this.height;
    }
}

class Circle extends Shape {
    constructor(radius) {
        super();
        this.radius = radius;
    }

    area() {
        return Math.PI * this.radius * this.radius;
    }
}

// Добавление новой фигуры не требует изменения существующего кода
class Triangle extends Shape {
    constructor(base, height) {
        super();
        this.base = base;
        this.height = height;
    }

    area() {
        return 0.5 * this.base * this.height;
    }
}

shapes.push(new Triangle(10, 5));
console.log(calculator.calculate(shapes));
// Теперь также учитывает площадь треугольника
```

```
class AreaCalculator {
    calculate(shapes) {
        let totalArea = 0;
        for (const shape of shapes) {
            totalArea += shape.area(); // Используем полиморфизм
        }
        return totalArea;
    }
}

// Использование
const shapes = [new Rectangle(10, 20), new Circle(5)];
const calculator = new AreaCalculator();
console.log(calculator.calculate(shapes)); // Выводит общую площадь
```

Liskov Substitution Principle (LSP)

Принцип подстановки Лисков

Объекты подкласса должны быть взаимозаменяемыми с объектами базового класса без изменения правильности программы. Это означает, что подклассы должны наследовать поведение базового класса.

Преимущества:

- Обеспечивает правильное поведение наследования.
- Упрощает использование и тестирование кода.

Пример нарушения принципа LSP

Представим класс, представляющий фигуру, и два подкласса: Square (квадрат) и Rectangle (прямоугольник). В этом случае мы нарушим принцип Лисков, если Square будет вести себя не как ожидается.

Проблемы с таким подходом

1. Несоответствие LSP: Square не может полностью заменить Rectangle, так как его поведение отличается от ожидаемого. Это приводит к неожиданным результатам.
2. Проблемы с предсказуемостью: Использование Square вместо Rectangle нарушает ожидания, что может привести к логическим ошибкам.

```
class Rectangle {
    constructor(width, height) {
        this.width = width;
        this.height = height;
    }

    setWidth(width) {
        this.width = width;
    }

    setHeight(height) {
        this.height = height;
    }

    area() {
        return this.width * this.height;
    }
}

class Square extends Rectangle {
    setWidth(width) {
        this.width = width;
        this.height = width; // Квадрат всегда имеет равные стороны
    }

    setHeight(height) {
        this.width = height;
        this.height = height; // Квадрат всегда имеет равные стороны
    }
}

// Использование
function printArea(rectangle) {
    rectangle.setWidth(5);
    rectangle.setHeight(10);
    console.log(rectangle.area()); // Ожидается 50, но...
}

const rectangle = new Rectangle(5, 10);
printArea(rectangle); // Выводит 50

const square = new Square(5);
printArea(square); // Выводит 100 (неправильный результат)
```

Правильное решение с соблюдением принципа LSP

Чтобы исправить ситуацию, можно изменить структуру классов, чтобы избежать нарушения LSP. Вместо того, чтобы наследовать Square от Rectangle, можно создать отдельный класс для каждого типа фигуры.

Преимущества нового подхода

- Соблюдение LSP: Теперь Square и Rectangle оба являются подклассами Shape и могут заменять друг друга без изменения поведения.
- Упрощение предсказуемости: Каждый класс реализует свои методы, и поведение остается ожидаемым.
- Гибкость: Легко добавлять новые фигуры без риска нарушения существующего кода.

Следуя принципу LSP, мы обеспечиваем, что подклассы могут использоваться вместо базовых классов без изменения корректности программы.

```
class Shape {
    area() {
        throw new Error("Method not implemented.");
    }
}

class Rectangle extends Shape {
    constructor(width, height) {
        super();
        this.width = width;
        this.height = height;
    }

    area() {
        return this.width * this.height;
    }
}

class Square extends Shape {
    constructor(side) {
        super();
        this.side = side;
    }

    area() {
        return this.side * this.side;
    }
}

// Использование
function printArea(shape) {
    console.log(shape.area());
}

const rectangle = new Rectangle(5, 10);
printArea(rectangle); // Выводим 50

const square = new Square(5);
printArea(square); // Выводим 25
```

Interface Segregation Principle (ISP)

Принцип разделения интерфейса

Клиенты не должны зависеть от интерфейсов, которые они не используют. Вместо создания одного большого интерфейса, лучше создавать несколько специализированных.

Преимущества:

- Упрощает реализацию интерфейсов.
- Позволяет избегать ненужных зависимостей.

Пример нарушения принципа ISP

Предположим, у нас есть интерфейс для многофункционального устройства, который включает в себя методы для печати, сканирования и факса. Если класс реализует интерфейс, он должен реализовать все методы, даже если не использует их.

Проблемы с таким подходом

1. Несоответствие принципу ISP: Класс Printer вынужден реализовывать методы, которые ему не нужны, что усложняет код.
2. Проблемы с поддержкой и тестированием: Код становится менее читабельным, так как он включает методы, которые не имеют смысла для данного класса.

```
class MultiFunctionDevice {
    print() {
        throw new Error("Method not implemented.");
    }

    scan() {
        throw new Error("Method not implemented.");
    }

    fax() {
        throw new Error("Method not implemented.");
    }
}

class Printer extends MultiFunctionDevice {
    print() {
        console.log("Printing...");
    }

    scan() {
        throw new Error("Scan not supported."); // Нарушение ISP
    }

    fax() {
        throw new Error("Fax not supported."); // Нарушение ISP
    }
}

// Использование
const printer = new Printer();
printer.print(); // Работает, но scan и fax не поддерживаются
```

Правильное решение с соблюдением принципа ISP

Теперь давайте разделим интерфейсы на более мелкие, специализированные.

Преимущества нового подхода

- Соблюдение ISP: Каждый класс реализует только те методы, которые ему нужны, что улучшает читаемость и поддержку.
- Упрощение тестирования: Легче тестировать каждый класс отдельно, так как они меньше зависят друг от друга.
- Гибкость: Можно легко добавлять новые классы с нужными функциональностями без изменения существующего кода.

Следуя принципу ISP, мы создаем более модульный и понятный код, который легче поддерживать и расширять.

```
class Printer {
    print() {
        console.log("Printing...");
    }
}

class Scanner {
    scan() {
        console.log("Scanning...");
    }
}

class Fax {
    fax() {
        console.log("Faxing...");
    }
}

// Использование
const printer = new Printer();
printer.print(); // Работает

const scanner = new Scanner();
scanner.scan(); // Работает

const faxMachine = new Fax();
faxMachine.fax(); // Работает
```

Dependency Inversion Principle (DIP)

Принцип инверсии зависимостей

Зависимости должны зависеть от абстракций, а не от конкретных реализаций. Это позволяет уменьшить связанность между модулями.

Преимущества:

- Упрощает тестирование и замену компонентов.
- Позволяет создавать более гибкие и масштабируемые архитектуры.

Пример нарушения принципа DIP

Предположим, у нас есть класс, который зависит от конкретной реализации сервиса уведомлений. Это нарушает принцип инверсии зависимостей, так как классы зависят от конкретных реализаций.

Проблемы с таким подходом

1. Несоответствие принципу DIP: Класс User зависит от конкретного класса EmailService, что делает его менее гибким и усложняет тестирование.
2. Трудности с заменой: Если потребуется изменить способ уведомления, придется изменять класс User.

```
class EmailService {
    sendEmail(message) {
        console.log(`Sending email: ${message}`);
    }
}

class User {
    constructor(emailService) {
        this.emailService = emailService; // Зависимость от конкретной реализации
    }

    notify(message) {
        this.emailService.sendEmail(message);
    }
}

// Использование
const emailService = new EmailService();
const user = new User(emailService);
user.notify("Hello, User!"); // Sending email: Hello, User!
```

Правильное решение с соблюдением принципа DIP

Чтобы исправить ситуацию, давайте введем абстракцию с помощью интерфейса (или абстрактного класса) для сервиса уведомлений.

Преимущества нового подхода

- Соблюдение DIP: Класс User теперь зависит от абстракции (NotificationService), а не от конкретных реализаций.
- Упрощение тестирования: Легко подменять зависимости на моки или стабы для тестирования.
- Гибкость: Можно легко добавлять новые способы уведомлений (например, PushNotificationService), не изменяя класс User.

Следуя принципу DIP, мы создаем более гибкую и модульную архитектуру, что упрощает поддержку и расширение приложения.

```
// Абстракция
class NotificationService {
    send(message) {
        throw new Error("Method not implemented.");
    }
}

class EmailService extends NotificationService {
    send(message) {
        console.log(`Sending email: ${message}`);
    }
}

class SMSService extends NotificationService {
    send(message) {
        console.log(`Sending SMS: ${message}`);
    }
}

class User {
    constructor(notificationService) {
        this.notificationService = notificationService; // Зависимость от абстракции
    }

    notify(message) {
        this.notificationService.send(message);
    }
}

// Использование
const emailService = new EmailService();
const smsService = new SMSService();

const user1 = new User(emailService);
user1.notify("Hello, User!"); // Sending email: Hello, User!

const user2 = new User(smsService);
user2.notify("Hello, User!"); // Sending SMS: Hello, User!
```

Ресурсы

О ООП - ТЫК

SOLID - ТЫК

YAGNI - ТЫК

KISS - ТЫК

DRY - ТЫК

Интересная статья про ООП в JS - ТЫК