**S.O.L.D.I.D principles**

<https://medium.com/@cramirez92/s-o-l-i-d-the-first-5-priciples-of-object-oriented-design-with-javascript-790f6ac9b9fa>

**Single responsibility principle**

A class should have one and only one reason to change, meaning that a class should only have one job.

Один класс — одна задача

Если задача класса — преобразовывать фигуры, он не должен заниматься тем, что определяет как отдать их пользователю. Отдавать пользователю должен другой класс

**Open-closed principle**

An API should be open for extension, but closed for modification.

Open for extension means that we should be able to add new features or components to the application without breaking existing code.

Открыт для добавления новых методов, закрыт для модификации

Closed for modification means that we should not introduce breaking changes to existing functionality, because that would force you to refactor a lot of existing code — Eric Elliott

**Liskov substitution pronciple**

All this is stating is that every subclass/derived class should be substitutable for their base/parent class.

Вместо родительского класса можно подставить наследуемый.

In other words, as simple as that, a subclass should override the parent class methods in a way that does not break functionality from a client’s point of view.

**Interface segregation principle**

A client should never be forced to implement an interface that it doesn’t use or clients shouldn’t be forced to depend on methods they do not use.

Под каждый клиент должен создаваться свой интерфейс

**Dependency inversion principle**

Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

Классы должны наследовать абстраккции и не должны зависеть от классов нижнего уровня

**ANTI-PATTERNS**

* Загрязнять глобальными переменными
* Передача строк вместо функций в setTimeout или setInterval
* Изменять базовый прототип Object
* Встраивать JS inline
* Использовать document.write

**JAVASCRIPT PATTERNS**

**Constructor**

Создает новые обьекты в их собственной области видимости. Часто используется как простой паттерн для создания обьектов подобного ему типа

class Hero {

constructor(name, specialAbility) {

this.\_name = name;

this.\_specialAbility = specialAbility;

this.getDetails = function (){

return `${this.\_name} can ${this.\_specialAbility}`

}

}

};

const Superman = new Hero('Superman', 'fly');

Superman.getDetails();

**Module**

Задача модуля — инкапсуляция, скрыть приватные и публичные методы

**var myModule = (function() {**

**var memes = ['cats', 'doge', 'harambe'];**

**var getMemes = function() { return memes };**

**return { getMemes: getMemes };**

**})();**

**console.log(myModule.getMemes()); // 'cats', 'doge', 'harambe'**

**console.log(myModule.memes); // undefined**

**GOF DESIGN PATTERNS**

****

****

**Creational Patterns**

**Factory**

Основан на классах.

Определяет интерфейс для создания объекта, но оставляет подклассам реше-

ние о том, какой класс инстанцировать. Часто используется, когда нужно манипулировать коллекциями обьектов которые похожи между собой

class MachineFactory {

constructor() {

this.create = function (type){

let machine;

// subclasses has the same interface

if (type === 'auto') machine = new Automobile(4, 500);

if (type === 'moto') machine = new Motorcycle(2, 200);

machine.start = function (){ };

machine.stop = function (){ };

return machine;

}

}

}

class Automobile {

constructor(wheels, hp) {

this.wheels = wheels;

this.horsePower = hp;

this.turnOn4x4 = function (){ };

}

}

class Motorcycle {

constructor(wheels, hp) {

this.wheels = wheels;

this.horsePower = hp;

// subclasses can have differ methods

this.rideOnRearWheel = function (){ }

}

}

const factory = new MachineFactory();

const bike = factory.create('moto');

const car = factory.create('auto');

bike.rideOnRearWheel();

car.turnOn4x4();

**Abstract factory**

Основана на обьектах.

Представляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов

class carFactory {

constructor (model) {

let car;

switch(model) {

case 'Cayman':

car = new Cayman();

break;

case 'Panamera':

car = new Panamera();

break;

default:

car = new Cayman();

break;

}

if (typeof car.printModel === 'undefined') {

car.printModel = function () {

console.log('This car model is:', car.model);

}

}

return car;

}

}

class Cayman {

constructor(){

this.model = 'Cayman';

this.createDoor = function (side) {

return CaymanDoor(side);

};

}

}

class Panamera {

constructor(){

this.model = 'Panamera';

this.createDoor = function (side) {

return PanameraDoor(side);

};

}

}

function CaymanDoor(side) {

const build = function() {

console.log(`Build a ${side} door for Cayman`);

};

return {

build: build

}

}

function PanameraDoor(side) {

const build = function() {

console.log(`Build a ${side} door for Panamera`);

};

return {

build: build

}

}

const factory = new carFactory();

const panameraCar = factory.createCar('Panamera');

panameraCar.printModel();

panameraCar.createDoor('right').build();

panameraCar.createDoor('left').build();

**Prototype**

Основан на обьектах.

Использует скелет для создания новых обьектов.

JS нативно поддерживает прототипное наследование вместо обьектно-ориентированого

const Car = {

start() { },

stop() { },

};

// добавили в новый обьект новое свойство – функцию turbo

const bmw = Object.create(Car, {turbo: () => { }});

console.log(bmw.\_\_proto\_\_); // Car

Так же можно реализовать с классом, тогда он ничем не будет отличаться от паттерна constructor:

class Sheep {

constructor(name, weight) {

this.name = name;

this.weight = weight;

}

clone() {

return new Sheep(this.name, this.weight);

}

}

const dolly = Sheep.clone(‘Dolly’, 60)

**Singleton**

Может существовать только 1 экземпляр обьекта, и все должны с ним работать, не создавая новые

Используется в Mongoose.

Несмотря на то, что Одиночка действительно имеет применение, обычно, если мы замечаем, что он нам необходим нам в JavaScript, это признак того, что нам следует переоценить наш дизайн.

Обычно это свидетельствует о том, что модули в системе либо тесно связаны, либо, что эта логика чрезмерно распространена по всем частям кода. Одиночку сложнее тестировать из-за проблем связанных с скрытыми зависимостями, сложностью с созданием нескольких экземпляров, трудностями в установлении зависимостей и т.д.

class Database {

constructor(data) {

if (Database.exists) return Database.instance;

this.\_data = data;

// Вместо this используем только имя класса!

// Иначе каждый раз будут создаваться новые обьекты

Database.instance = this;

Database.exists = true;

return this;

}

getData(){ return this.\_data }

setData(data){ this.\_data = data}

}

const mongo = new Database('mongo');

console.log(mongo.getData()); // mongo

const mysql = new Database('mysql');

console.log(mysql.getData()); // mongo;

**Structural Patterns**

**Adapter**

Интерфейс одного класса тралслируется другому. Этот паттерн позволяет классам работать вместе что в противном случае было бы невозможным ввиду несовместимых интерфейсов

Часто используется для создания оболочек для новых отрефакторенных API, что бы старые API могли работать вместе с ними

class OldCalculator {

constructor() {

this.operations = function (term1, term2, operation) {

switch (operation) {

case 'add':

return term1 + term2;

case 'sub':

return term1 - term2;

default:

return NaN;

}

}

}

}

class NewCalculator {

constructor() {

this.add = function (term1, term2) {

return term1 + term2;

};

this.sub = function (term1, term2) {

return term1 - term2;

};

}

}

class CalculatorAdapter {

constructor() {

const newCalc = new NewCalculator();

this.operations = function (term1, term2, operation) {

switch (operation) {

case 'add':

return newCalc.add(term1, term2);

case 'sub':

return newCalc.sub(term1, term2);

default:

return NaN;

}

}

}

}

adapter = new CalculatorAdapter();

adapter.operations(5, 10, 'add');



**Composite**

Структурный паттерн, который обьединяет обьекты в в деревовидные структуры для представления ирерархии целых частей. В этом паттерне, каждая ветвь структуры может быть как одним обьектом, так и коллекцией.

class Component {

constructor(name) {

this.\_name = name;

}

getNodeName() { return this.\_name }

// abstract methods that need to be overridden

getType() { }

addChild(component) { }

getChildByIndex(index) { }

numberOfChildren() { }

static logTreeStructure(root) {

let treeStructure = '';

function traverse(node, indent = 0) {

treeStructure += `${'--'.repeat(indent)}${node.getNodeName()}\n`;

indent++;

for (let i = 0, length = node.numberOfChildren(); i < length; i++) {

traverse(node.getChildByIndex(i), indent);

}

}

traverse(root);

return treeStructure;

}

}

class Leaf extends Component {

constructor(name) {

super(name);

this.\_type = 'Leaf Node';

}

getType() { return this.\_type }

numberOfChildren() { return 0 }

}

class Composite extends Component {

constructor(name) {

super(name);

this.\_type = 'Composite Node';

this.\_children = [];

}

getType() { return this.\_type }

addChild(component) { this.\_children = [...this.\_children, component] }

getChildByIndex(index) { return this.\_children[index] }

numberOfChildren() { return this.\_children.length }

}

// usage

const tree = new Composite('root');

tree.addChild(new Leaf('leaf'));

const branch = new Composite('branch');

tree.addChild(branch);

branch.addChild(new Leaf('branch-leaf-left'));

branch.addChild(new Leaf('branch-leaf-right'));

const branchMid = new Composite('branch-mid');

branch.addChild(branchMid);

branchMid.addChild(new Leaf('branch-mid-leaf-left'));

branchMid.addChild(new Leaf('branch-mid-leaf-right'));

// log

console.log(Component.logTreeStructure(tree));

/\*

root

--leaf

--branch

----branch-leaf-left

----branch-leaf-right

----branch-mid

------branch-mid-leaf-left

------branch-mid-leaf-right

\*/

**Decorator**

Структурный паттерн, сфокусированный на возможности добавить функционал к существующим классам динамически. Альтернатива наследованию классов

Его легко реалиховать в JS, т. к. JS позволяет добавлять методы и свойства обьектам динамически

class Book {

constructor(title, author, price) {

this.\_title = title;

this.\_author = author;

this.price = price;

}

getDetails() { return `${this.\_title} by ${this.\_author}` }

}

// decorator

function hardbindBook(book) {

book.isHardbound = true;

book.price += 5;

return book;

}

// usage

const inferno = hardbindBook(new Book('Inferno', 'Dan Brown', 15));

console.log(inferno.isHardbound); // true

console.log(inferno.price); // 20

**Decorator**

Структурный паттерн, сфокусированный на возможности добавить функционал к существующим классам динамически. Альтернатива наследованию классов

Его легко реалиховать в JS, т. к. JS позволяет добавлять методы и свойства обьектам динамически

class Book {

constructor(title, author, price) {

this.\_title = title;

this.\_author = author;

this.price = price;

}

getDetails() { return `${this.\_title} by ${this.\_author}` }

}

// decorator

function hardbindBook(book) {

book.isHardbound = true;

book.price += 5;

return book;

}

// usage

const inferno = hardbindBook(new Book('Inferno', 'Dan Brown', 15));

console.log(inferno.isHardbound); // true

console.log(inferno.price); // 20

**Facade**

Используется для сокрытия сложного функционала, предоставления единого и простого публичного интерфейса. Часто используется в библиотеках наподобие jQuery

class ShopFacade {

constructor() {

this.discount = new Discount();

this.shipping = new Shipping();

this.fees = new Fees();

}

calc(price) {

price = this.discount.calc(price);

price = this.fees.calc(price);

price += this.shipping.calc();

return price;

}

}

class Discount {

calc(value) { return value \* 0.9 }

}

class Shipping {

calc() { return 5 }

}

class Fees {

calc(value) { return value \* 1.05 }

}