**ПОНЯТИЕ «ОБЬЕКТ»**

Объектно-ориентированные программы состоят из объектов. **Объект** сочетает данные и процедуры для их обработки. Такие процедуры обычно называют **методами** или **операциями.** Объект выполняет операцию, когда получает запрос (или сообщение) от клиента.

**Посылка запроса** - это единственный способ заставить объект выполнить опе-

рацию.

**Выполнение операции** - единственный способ изменить внутреннее со-

стояние объекта.

Имея в виду два эти ограничения, говорят, что внутреннее состояние объекта **инкапсулировано**: к нему нельзя получить непосредственный

доступ, то есть представление объекта закрыто от внешней программы.

**ИНТЕРФЕЙСЫ**

**Сигранатура операции**: при объявлении объектом любой операции должны быть заданы: имя операции, объекты, передаваемые в качестве параметров, и значение, возвращаемое операцией

**Интерфейсом объекта** называетсямножество сигнатур всех определенных для объекта операций. Интерфейс описывает все множество запросов, которые можно отправить объекту. Любой запрос, сигнатура которого соответствует интерфейсу объекта, может быть ему послан.

**Тип** — это имя, используемое для обозначения конкретного интерфейса. Говорят, что объект имеет тип Window, если он готов принимать запросы на выполнение любых операций, определенных в интерфейсе с именем Window.У одного объекта может быть много типов. Напротив, сильно отличающиеся объекты могут разделять общий тип. Часть интерфейса объекта может быть охарактеризована одним типом, а часть - другим. Два объекта одного и того же типа должны разделять только часть своих интерфейсов. Интерфейсы могут содержать другие интерфейсы в качестве подмножеств. Мы говорим, что один тип является подтипом другого, если интерфейс первого содержит интерфейс второго. В этом случае второй тип называется **супертипом** для первого. Часто говорят также, что **подтип** **наследует интерфейс своего супертипа.**

Два объекта с различными реализациями могут иметь одинаковые интерфейсы.

Когда объекту посылается запрос, то операция, которую он будет выполнять,

зависит как от запроса, так и от объекта-адресата. Разные объекты, поддерживающие одинаковые интерфейсы, могут выполнять в ответ на такие запросы разные операции. Ассоциация запроса с объектом и одной из его операций во время выполнения называется **динамическим связыванием**.

Динамическое связывание означает, что отправка некоторого запроса не определяет никакой конкретной реализации до момента выполнения. Следовательно, допустимо написать программу, которая ожидает объект с конкретным интерфейсом, точно зная, что любой объект с подходящим интерфейсом сможет принять этот запрос.

**Полиморфизм** — возможность, возникающая при динамическом связывании, позволяющая прямо во время выполнения подставить вместо одного объекта другой, если он имеет точно такой же интерфейс.

Он позволяет клиенту не делать почти никаких предположений об объектах, кроме того, что они поддерживают определенный интерфейс.

**Наследование класса** — включение определений всех данных и операций, определенных в родительском классе.

**Абстрактный класс** — класс, единственное назначение которого — определить общий интерфейс для всех своих подклассов.

**Абстрактные операции** — операции, объявленные, но не реализованные в абстрактном классе

**МЕХАНИЗМЫ ПОВТОРНОГО ИСПОЛЬЗОВАНИЯ**

**Наследование класса —** определение реализации одного класса в терминах другого. Повторное использование за счет порождения подкласса называют еще **прозрачным ящиком** (white-box reuse). Такой термин подчеркивает, что внутреннее устройство родительских классов видимо подклассам.

Поскольку подклассу доступны детали реализации родительского класса,

то часто говорят, что наследование нарушает инкапсуляцию

**Композиция объектов** — это Композиция обьектов — альтернатива наследованию. Композиция объектов определяется динамически во время выполнения за счет того, что объекты получают ссылки на другие объекты. Композицию можно применить, если объекты соблюдают интерфейсы друг друга. Такой способ повторного использования называют **черным ящиком** (black-box reuse), поскольку детали внутреннего устройства объектов остаются скрытыми.

Поскольку доступ к объектам осуществляется только через их интерфейсы,

мы не нарушаем инкапсуляцию. Во время выполнения программы любой объект

можно заменить другим, лишь бы он имел тот же тип.

**Виды композиции**

**Делегирование обьектов** — мощный иструмент композиции, в процесс обработки запроса вовлечено два объекта: получатель поручает выполнение операций другому объекту — уполномоченному. Получатель передает указатель на самого себя соответствующему объекту, дабы при выполнении делегированной операции последний мог обратиться к непосредственному адресату запроса.

Например, вместо того чтобы делать класс Window (окно) подклассом класса

Rectangle (прямоугольник) - ведь окно является прямоугольником, - мы можем воспользоваться внутри Window поведением класса Rectangle, поместив в класс Window переменную экземпляра типа Rectangle и делегируя ей операции, специфичные для прямоугольников. Другими словами, окно *не является* прямоугольником, *а содержит* его.

Делегирование используется в паттернах: состояние, стратегия, посетитель.

**Осведомленность (acquaintance)** означает, что объект, владеющий указателем, не несет никакой ответственности за объект, на который он ссылается. Он просто имеет доступ к его публичным полям и методам, но время жизни этого объекта не под его контролем.

Пример — в функция-обработчик HTTP запроса может передаваться ссылка на логгер — *использует, а не является*

**Агрегирование** же обозначает, что объект, владеющий ссылкой, несет ответственность за уничтожение объекта, на который он ссылается. Объект содержит другой объект или является его частью. Агрегирование означает, что агрегат и его составляющие имеют одинаковое время жизни.

**Виды композиции в JS (Eric Elliot)**

**Class ingeritance composites**:

class Foo {

constructor () {

this.a = 'a'

}

}

class Bar extends Foo {

constructor (options) {

super(options);

this.b = 'b'

}

}

const myBar = new Bar(); // {a: 'a', b: 'b'}

**Aggregation** —когда объект содержит другие объекты. Каждый подобъект содержит ссылку на себя, потому может быть деструктурирован из агрегации без потери информации (в переменную например)

A *has-a* relationship, e.g., DOM children are component elements in a DOM node — a DOM node *has* children

Examples: Arrays, Maps, Sets, WeakMaps , Graphs, Treess, TypedArrays, DOM nodes etc

const aggregator = {

age: 21,

name: 'Alex',

location: {

city: 'Kyiv',

address: 'Khreschatic, 28'

}

}

**Concatenation** —когда объект формируется добавлением новых свойств в существующий обьект.

Сконкатенированные обьекты могут вести к багам, т.к лучше несколько мелких обьектов чем один большой — по сути это похоже на наследование через класс.

Examples: objects and arrays concatenation, state reducers in Redux, Functional mixins

const c = {...user, ...anonymous};

**Delegation** —когда объект делегирует выполнение другому объекту

Встроенное прототипное наследование JS является делегированием — [].map делегирует Array.prototype.map() и тд.

Обработчик HTTP запроса может делегировать логирование переданному в него логгеру

* Экономит память, т.к. не надо создавать новые объекты, используем уже существующие
* динамическое обновление множества экземпляров

Определение различных видов композиции объектов не является взаимоисключающими. Делегирование — это подмножество агрегации, конкатенация может быть использована для формирования делегатов и агрегатов и тд.

**НАИБОЛЕЕ ЧАСТЫЕ ПРИЧИНЫ РЕФАКТОРИНГА**

* **при создании объекта явно указывается класс.** Задание имени класса привязывает вас к конкретной реализации, а не к конкретному интерфейсу. Это может осложнить изменение объекта в будущем. Чтобы уйти от такой проблемы, создавайте объекты косвенно.

Паттерны проектирования: абстрактная фабрика, фабричный метод, прототип

* **зависимость от конкретных операций**. Задавая конкретную операцию, вы ограничиваете себя единственным способом выполнения запроса. Если же не включать запросы в код, то будет проще изменить способ удовлетворения запроса как на этапе компиляции, так и на этапе выполнения.

Паттерны проектирования: цепочка обязанностей, команда;

* **зависимость от аппаратной и программной платформ.** Внешние интерфейсы операционной системы и интерфейсы прикладных программ (API) различны на разных программных и аппаратных платформах. Если программа зависит от конкретной платформы, ее будет труднее перенести на другие. Даже на «родной» платформе такую программу трудно поддерживать. Поэтому при проектировании систем так важно ограничивать платформенные зависимости.

Паттерны проектирования: абстрактная фабрика, мост;

* **зависимость от представления или реализации объекта.** Если клиент «знает», как объект представлен, хранится или реализован, то при изменении объекта может оказаться необходимым изменить и клиента. Сокрытие этой информации от клиентов поможет уберечься от каскада изменений.

Паттерны проектирования: абстрактная фабрика, мост, хранитель, заместитель;

* **зависимость от алгоритмов.** Во время разработки и последующего использования алгоритмы часто расширяются, оптимизируются и заменяются. Зависящие от алгоритмов объекты придется переписывать при каждом изменении алгоритма. Поэтому алгоритмы, вероятность изменения которых высока, следует изолировать.

Паттерны проектирования: мост, итератор, стратегия, шаблонный метод,

посетитель;

* **сильная связанность.** Сильно связанные между собой классы трудно использовать порознь, так как они зависят друг от друга. Сильная связанность приводит к появлению монолитных систем, в которых нельзя ни изменить, ни удалить класс без знания деталей и модификации других классов. Такую систему трудно изучать, переносить на другие платформы и сопровождать. Слабая связанность повышает вероятность того, что класс можно будет повторно использовать сам по себе. При этом изучение, перенос, модификация и сопровождение системы намного упрощаются. Для поддержки слабо связанных систем в паттернах проектирования применяются такие методы, как абстрактные связи и разбиение на слои.

Паттерны проектирования: абстрактная фабрика, мост, цепочка обязанностей, команда, фасад, посредник, наблюдатель;

* **расширение функциональности за счет порождения подклассов.** Специализация объекта путем создания подкласса часто оказывается непростым делом. С каждым новым подклассом связаны фиксированные издержки реализации (инициализация, очистка и т.д.). Для определения подкласса необходимо так же ясно представлять себе устройство родительского класса. Например, для замещения одной операции может потребоваться заместить и другие. Замещение операции может оказаться необходимым для того, чтобы можно было вызвать унаследованную операцию. Кроме того, порождение подклассов ведет к комбинаторному росту числа классов, поскольку даже для реализации простого расширения может понадобиться много новых подклассов.

Композиция объектов и делегирование - гибкие альтернативы наследованию для комбинирования поведений. Приложению можно добавить новую функциональность, меняя способ композиции объектов, а не определяя новые подклассы уже имеющихся классов. С другой стороны, при интенсивном использовании композиции объектов проект может оказаться трудным для понимания. С помощью многих паттернов проектирования удается построить такое решение, где специализация достигается за счет определения одного подкласса и комбинирования его экземпляров с уже существующими.

Паттерны проектирования: мост, цепочка обязанностей, компоновщик, декоратор, наблюдатель, стратегия

* **неудобства при изменении классов.** Иногда нужно модифицировать класс, но делать это неудобно. Допустим, вам нужен исходный код, а его нет (так обстоит дело с коммерческими библиотеками классов). Или любое изменение тянет за собой модификации множества существующих подклассов. Благодаря паттернам проектирования можно модифицировать классы и при таких условиях.

Паттерны проектирования: адаптер, декоратор, посетитель.

**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.

Делегирует процесс создания обьектов своим подклассам, сам же определяет для них только интерфейс. 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();

**Builder**

Основан на обьектах.

Отделяет конструирование сложного объекта от его представления, так что

в результате одного и того же процесса конструирования могут получаться разные представления.

class Request {

constructor() {

this.url = '';

this.method = '';

this.payload = {};

}

}

class RequestBuilder {

constructor() {

this.request = new Request();

}

forUrl(url) {

this.request.url = url;

return this;

}

useMethod(method) {

this.request.method = method;

return this;

}

payload(payload) {

this.request.payload = payload;

return this;

}

build() {

return this.request;

}

}

**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 }

}