**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**

**MVC pattern**

**Model —** содержит данные и логику, с которыми взаимодействует юзер. Обычно бекенд. Model самодостаточна, она имеет все методы и состояния что бы существовать сама по себе.

**View** — возможность юзера увидеть модель, или хотя бы ее часть. Обычно UI. View может быть несколько. View подписываются на изменения модели через паттерн Observer

**Controller** — содержит всю логику для обработки запросов от view и перенаправления их в модель. Controller — управляющий между моделью и view. View сам подставляет нужный контроллер для обработки действий юзера — паттерн Strategy.

**MVP pattern**

**Presenter** отличается от Controller тем, что содержит всю логику работы с View. View становится полностью dumb или passive View, сам Presenter подписывается на изменения модели и транслирует их во view через интерфейс. Это позволяет писать код Presenter не завися от дизайна UI или параллельно с разработкой UI, облегчает написание unit-tests. MVP в основном используется в энтерпрайз приложениях, в которых часто надо повторно использовать презентационную логику.

**View** — dumb или passive, перенаправляет весь пользовательский ввод на Presenter`a, управляется только Presenter`ом.

Отличия MVP от MVC в основном семантические.

**MVVM pattern**

**Model —** содержит данные, например емейл и имя юзера, но обычно не обрабатывает поведение. Они не форматирует информацию и не влияет на то, как данные появляются в браузере — это не ее обязанность. Вместо этого, форматирование данных происходит во View, в то время как поведение рассматривается как бизнес-логика которая должна быть инкапсулирована в другом слое, который взаимодействует с моделью — ViewModel. Единственные методы, которые может содержать Model — валидация данных при внесении оных в модель.

**View** — как и в MVC, view является единственной частью приложения, с которым юзеры могут взаиодействовать. Это интерактивный UI, который представляет состояние ViewModel. В данной модели View активное — оно принимает ввод юзера и содержит привязки данных, евенты и поведение, которое требует знания о ViewModel. View не обрабатывает **state** (состояние) — оно его синхронизирует с ViewModel

**ViewModel** — знает о Model и содержит методы поведения и изменения данных. ViewModel может рассматриваться как специальный контроллер, который работает как преобразователь данных. Оно изменяет информацию полученную из Model в информацию, которую надо передать во View, а так же передает команды из View в Model. View и ViewModel коммуницируют, используя привязки данных и евенты.

**MVC Vs. MVP Vs. MVVM**

В **MVC**, View находится на верхушке архитектуры, рядом с контроллером. View знает о Controller, а Controller знает о Model. View имеет непосредственный доступ к Model, что чревато проблемами безопасности и производительности, в зависимости от сложности приложения.

В **MVP,** Presenter находится на одном уровне с View, слушает евенты и от View, и от модели и обрабатывает действия между ними. В отличии от MVVM, тут отсутствует механизм привязки View к ViewModel, вместо этого создается интерфейс View, через который Presenter передает данные во View.

В **MVVM** позволяет последовательно создавать специфичные подмножества Model в разрезе View, которые содержат состояние модели и логику, избегая нужды показывать всю модель для View. В отличии от Presenter в MVP, ViewModel не требуется ссылаться на View. View может быть привязано к свойствам ViewModel, которые в свою очередь отображают данные Model.

Абстракция View означает меньше логики.

Одной из отрицательных сторон является то, что уровень интерпретации, который требуется между ViewModel и View уменьшает производительность. Сложность зависит от приложения — от простого копирования данных до сложного манипулирования данными для приведения их к тому виду, который требуется для View. В MVC такой проблемы нет, т. к. вся модель доступна сразу, и манипуляции такого вида не требуются. MVVM обычно применяется в крупных приложениях

**GOF DESIGN PATTERNS**

Классные примеры на <https://refactoring.guru/ru/design-patterns>

****

****

**Creational Patterns**

**Factory**

Делится на 2 вида: Simple Factory и Factory Method

**Simple Factory**

Делегирует процесс создания объектов своим подклассам, сам же определяет для них только интерфейс.

Вынос процесса создания объектов из продукта в отдельный объект-фабрику, продукт же будет делегировать процесс создания этой фабрике, а фабрика выбирает нужный класс исходя из переданных параметров и инстанциирует объект. Таким образом, продукт не будет знать деталей процесса инстанциирования, передавая только параметры. Заказывая телевизор Sony с диагональю 50“ и встроенным Smart TV меня в последнюю очередь будут интересовать детали создания этого телевизора.

Используется если планируется создавать много разных реализаций продукта.

const MachineFactory = {

create(type){

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

}

};

class Machine {

constructor(type) {

this.factory = MachineFactory.create;

const machine = this.factory(type);

machine.start = function() { /\* ...some implementation \*/ };

machine.stop = function() { /\* ...some implementation \*/ };

return machine;

}

}

class Automobile {

constructor(wheels, hp) {

this.wheels = wheels;

this.horsePower = hp;

// subclasses can have differ methods

this.turnOn4x4 = function () { };

}

}

// клиент ничего не знает о классе Automobile,

// он пользуется интерфейсом Machine

const car = new Machine('auto');

car.start();

car.turnOn4x4();

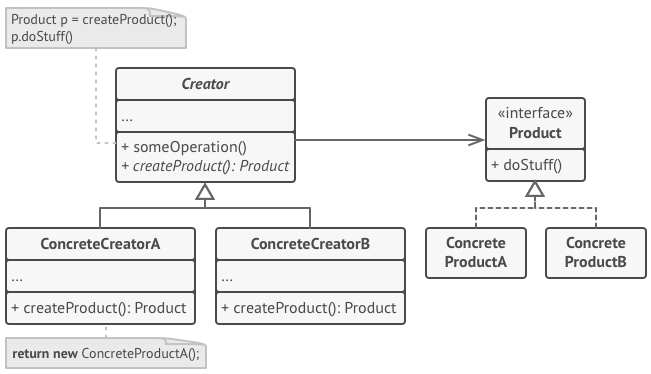
**Factory Method**

Отличается от Simple Factory тем, что может содержать дополнительную логику управления создания объектов подклассами, в то время как Simple Factory только создает обьекты, но не управляет (настраивает) ими. Используется, если нужно контролировать алгоритм/стратегию создания семейств продуктов. Отлично сочетается с *Template Pattern.*

Тоже самое, что абстрактная фабрика.

*Product* — интерфейс, который наследуют все подклассы продукта

*Creator* — абстрактный класс, который реализует общие классы для всех конкретных создателей а так же имеет абстрактный метод *create(),* который каждый подкласс перезаписывает по-своему, что бы вызвать нужный подкласс Product



**Abstract factory**

Основана на классах\объектах

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.

Паттерн предлагает вынести конструирование обьекта за пределы его собственного класса, поручив это классам *builders.* Процесс строительства разбивается на шаги, строители вызываются по очереди, причем у одного строителя может быть несколько вариаций (например *buildWall* может построить деревянную стену, а может каменную).

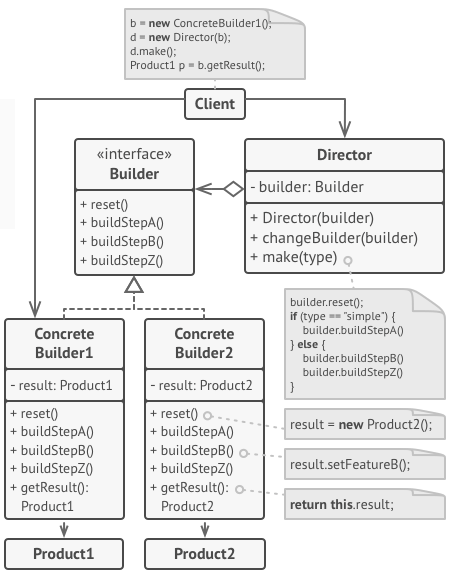
Клиент можно вызывать строителей напрямую, но при наличии нескольких вариаций, вводятся посредники *Directors,* которые определяют, каких строителей нужно вызывать

Используется:

* Когда вы хотите избавиться от «телескопического конструктора», когда в конструктор пихаются все возможные опции продукта
* Когда ваш код должен создавать разные представления какого-то объекта. Например, деревянные и железобетонные дома.
* Когда вам нужно собирать сложные составные объекты, например, деревья *Composite*.

*Builder* концентрируется на построении сложных объектов шаг за шагом. *Abstract factory* специализируется на создании семейств связанных продуктов. *Builder* возвращает продукт только после выполнения всех шагов, а *Abstract factory* возвращает продукт сразу же.

*Builder* может быть построен в виде *Bridge*: *director* будет играть роль абстракции, а *builders* — реализации.



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**

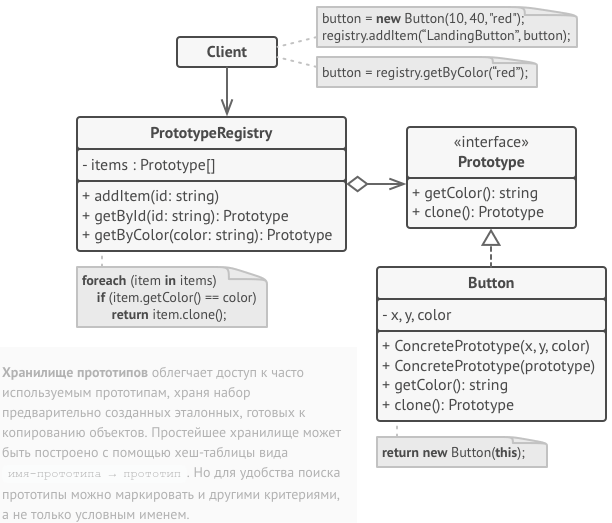
Основан на обьектах.

Позволяет копировать объекты, не вдаваясь в подробности их реализации.

Паттерн поручает создание копий самим копируемым объектам. Он вводит общий интерфейс для всех объектов, поддерживающих клонирование. Это позволяет копировать объекты, не привязываясь к их конкретным классам. Обычно такой интерфейс имеет всего один метод clone.

Метод создаёт новый объект текущего класса и копирует в него значения всех полей собственного объекта. Так получится скопировать даже приватные поля.

JS нативно поддерживает прототипное наследование



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**

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

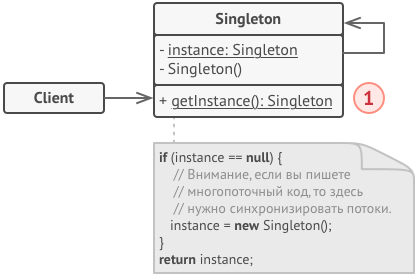
Паттерн скрывает конструктор и создает публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.

Используется в Mongoose.

Не рекомендуется использовать в Javascript.

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) {

// Вместо this используем только имя класса!

// Иначе каждый раз будут создаваться новые обьекты

if (Database.exists) return Database.instance;

this.\_data = data;

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**

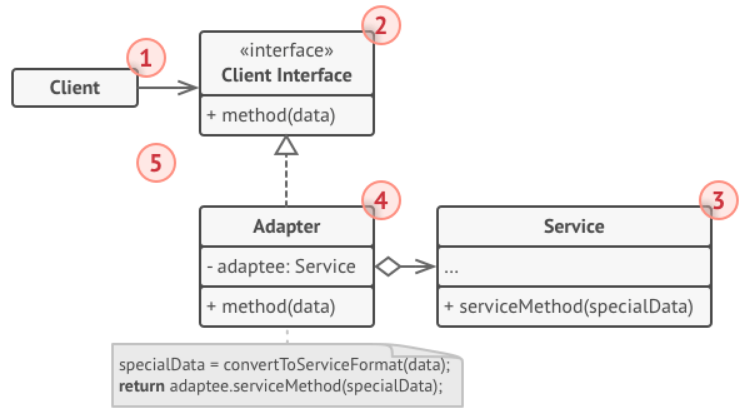
Трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. 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

Двусторонний адаптер — разные классы могут работать друг с другом через него.

*Adapter* предоставляет клиенту альтернативный интерфейс. *Decorator* предоставляет расширенный интерфейс. *Proxy* предоставляет тот же интерфейс.



// old interface

class Shipping {

request(zipStart, zipEnd, weight) {

// ...

return console.log("$49.75");

}

}

// new interface

class AdvancedShipping {

login(credentials) { };

setStart(start) { };

setDestination(destination) { };

calculate(weight) {

// ...

return console.log("$39.50");

};

}

// adapter interface

class ShippingAdapter {

constructor(credentials) {

this.\_shipping = new AdvancedShipping();

this.\_shipping.login(credentials);

}

request(zipStart, zipEnd, weight) {

this.\_shipping.setStart(zipStart);

this.\_shipping.setDestination(zipEnd);

return this.\_shipping.calculate(weight);

}

}

// original shipping object and interface

const shipping = new Shipping();

let cost = shipping.request("78701", "10010", "2 lbs");

// new shipping object with adapted interface

const credentials = {token: "30a8-6ee1"};

const adapter = new ShippingAdapter(credentials);

cost = adapter.request("78701", "10010", "2 lbs");

**Bridge**

Разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

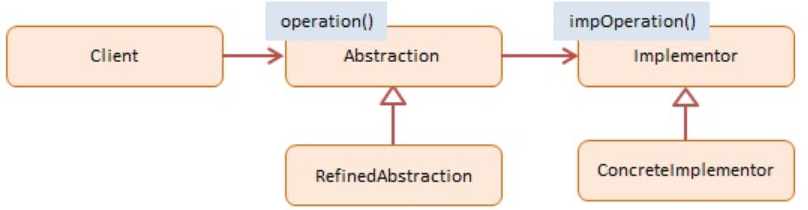
Паттерн предлагает заменить наследование делегированием

В контексте *Bridge*, абстракция (или интерфейс) — это образный слой управления чем-либо. Он не делает работу самостоятельно, а делегирует её слою реализации (иногда называемому платформой).

Например, абстракцией может выступать графический интерфейс программы (GUI), а реализацией — низкоуровневый код операционной системы (API), к которому графический интерфейс обращается по реакции на действия пользователя.

Используется:

* Когда вы хотите разделить монолитный класс, который содержит несколько различных реализаций какой-то функциональности (например, если класс может работать с разными системами баз данных).
* Когда класс нужно расширять в двух независимых плоскостях.
* Когда вы хотите, чтобы реализацию можно было бы изменять во время выполнения программы.



// Remotes contain references to the device they control.

// Remotes delegate most of the work to their device objects.

class Remote {

constructor(device) {

this.device = device

}

togglePower() {

if (this.device.isEnabled()) {

this.device.disable();

} else {

this.device.enable();

}

}

volumeDown() {

this.device.setVolume(this.device.getVolume() - 10)

}

volumeUp() {

this.device.setVolume(this.device.getVolume() + 10)

}

channelDown() {

this.device.setChannel(this.device.getChannel() - 1)

}

channelUp() {

this.device.setChannel(this.device.getChannel() + 1)

}

}

// You can extend remotes' class hierarchy independently

// from devices' classes

class AdvancedRemote extends Remote {

constructor() {

super();

}

mute() {

this.device.setVolume(0)

}

}

// All devices follow the common interface.

// It makes them compatible with all types of remotes.

class Device {

isEnabled(){}

enable(){}

disable(){}

getVolume(){}

setVolume(percent){}

getChannel(){}

setChannel(channel){}

}

// but each concrete device has its own implementation.

class Tv extends Device {

getChannel(){ console.log('TV channel: BBC')}

}

class Radio extends Device {

getChannel(){ console.log('Radio channel: Electro Swing')}

}

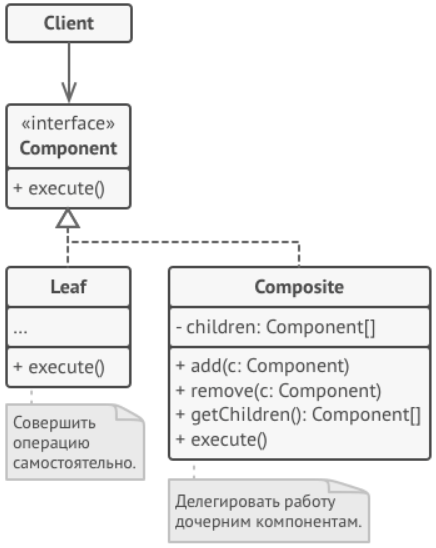
// client code

tv = new Tv();

remote = new Remote(tv);

radio = new Radio();

remote = new AdvancedRemote(radio);

**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.

Основную часть функционала выполняет «лист», которому делегериует обязанности его ближайший «композит» или же сам корень

Используется:

* Когда вам нужно представить древовидную структуру объектов.
* Когда клиенты должны единообразно трактовать простые и составные объекты.

Минусы: создает слишком общий дизайн классов

class Component {

constructor(name) {

this.\_name = name;

}

getNodeName() {

return this.\_name

}

// abstract methods

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**

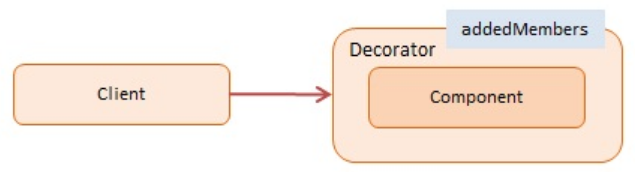
Позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки». Альтернатива наследованию классов

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

Вы помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё.

Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно — результат будет иметь объединённое поведение всех обёрток сразу.

Однако более полезен он в статических языках, в которых обьекты инстанциируются из класса, когда необходима комбинация поведения нескольких обьектов. Decorator создает стек из таких обьектов и вызывает их последовательно.



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**

Предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку, скрывая их сложный функционал

Не вносит добавочной функциональности. Сама подсистема не знает о существовании Фасада. Классы подсистемы общаются друг с другом напрямую.

Фасадов может быть несколько для разных клиентов, что бы не захламлять единственный всем подряд

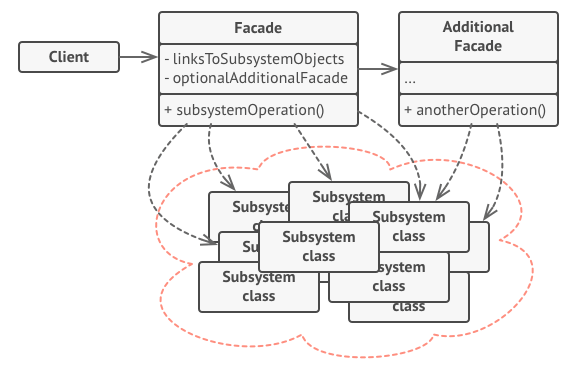
* Incapsulatiuon: инкапуслирует сложность подсистемим за общей оберткой
* Less coupling: избавляет клиента от необходимости самостоятельно управлять подсистемой
* Управляет инстанциированием и перенаправлением задач походящему классу в подсистеме

Часто используется в библиотеках наподобие jQuery

Используется:

* Когда вам нужно представить простой или урезанный интерфейс к сложной подсистеме.
* Когда вы хотите разложить подсистему на отдельные слои. Например, вы хотите разбить на слои работы с аудио и видео сложную систему видеоконвертации. Для каждой из этих частей можно попытаться создать фасад и заставить классы аудио и видео обработки общаться друг с другом через эти фасады, а не напрямую.

Минусы: рискует стать божественным объектом, привязанным ко всем классам программы.



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 } }

const shop = new ShopFacade();

shop.calc(100);

**Flyweight**

Позволяет вместить бóльшее количество объектов в отведённую оперативную память. *Flyweight* экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

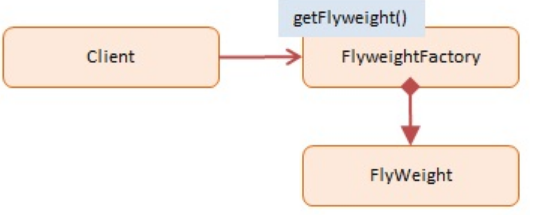
Неизменяемые данные объекта принято называть «*внутренним состоянием*». Все остальные данные — это «*внешнее состояние*».

*Flyweight* предлагает не хранить в классе внешнее состояние, а передавать его в те или иные методы через параметры. Таким образом, одни и те же объекты можно будет повторно использовать в различных контекстах. Но главное — понадобится гораздо меньше объектов, ведь теперь они будут отличаться только внутренним состоянием, а оно имеет не так много вариаций.

Так как объекты легковесов будут использованы в разных контекстах, вы должны быть уверены в том, что их состояние невозможно изменить после создания. Всё внутреннее состояние легковес должен получать через параметры конструктора. Он не должен иметь сеттеров и публичных полей.Так как объекты легковесов будут использованы в разных контекстах, вы должны быть уверены в том, что их состояние невозможно изменить после создания. Всё внутреннее состояние легковес должен получать через параметры конструктора. Он не должен иметь сеттеров и публичных полей.

Для удобства работы с легковесами и контекстами можно создать фабричный метод, принимающий в параметрах всё внутреннее (а иногда и внешнее) состояние желаемого объекта.

Главная польза от этого метода в том, чтобы искать уже созданные легковесы с таким же внутренним состоянием, что и требуемое. Если легковес находится, его можно повторно использовать. Если нет — просто создаём новый. Обычно этот метод добавляют в контейнер легковесов либо создают отдельный класс-фабрику. Его даже можно сделать статическим и поместить в класс легковесов.



class Flyweight {

constructor(make, model, processor){

this.make = make;

this.model = model;

this.processor = processor;

}

}

const FlyWeightFactory = {

\_flyweights: {},

get(make, model, processor) {

if (!this.\_flyweights[make + model]) {

this.\_flyweights[make + model] =

new Flyweight(make, model, processor);

}

return this.\_flyweights[make + model];

},

getCount() {

let count = 0;

for (const f in this.\_flyweights) count++;

return count;

},

};

class Computer {

constructor(make, model, processor, memory, tag){

this.flyweight = FlyWeightFactory.get(make, model, processor);

this.memory = memory;

this.tag = tag;

}

getMake() { return this.flyweight.make }

// ...

}

class ComputerCollection {

constructor() {

this.\_computers = {};

this.\_count = 0;

}

add(make, model, processor, memory, tag) {

this.\_computers[tag] = new Computer(make, model, processor, memory, tag);

this.\_count++;

}

get(tag) { return this.\_computers[tag] }

getCount() { return this.\_count }

}

const computers = new ComputerCollection();

computers.add("Dell", "Studio XPS", "Intel", "5G", "Y755P");

computers.add("Dell", "Studio XPS", "Intel", "6G", "X997T");

computers.add("Dell", "Studio XPS", "Intel", "2G", "U8U80");

computers.add("HP", "Envy", "Intel", "4G", "CNU883701");

computers.add("HP", "Envy", "Intel", "2G", "TXU003283");

console.log("Computers: " + computers.getCount());

console.log("Flyweights: " + FlyWeightFactory.getCount());

**Proxy**

Позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

Decorator отличаются от Proxy тем, что Proxy сам управляет жизнью сервисного объекта и предоставляет тот же интерфейс, а обёртывание Decorator контролируется клиентом и интерфейс Decorator может быть расширен.

Используется:

* ленивая инициализация (виртуальный прокси). Когда у вас есть тяжёлый объект, грузящий данные из файловой системы или базы данных.
* Защита доступа (защищающий прокси). Когда в программе есть разные типы пользователей, и вам хочется защищать объект от неавторизованного доступа.
* Локальный запуск сервиса (удалённый прокси). Когда настоящий сервисный объект находится на удалённом сервере.
* Логирование запросов (логирующий прокси). Когда требуется хранить историю обращений к сервисному объекту.
* Кеширование объектов («умная» ссылка). Когда нужно кешировать результаты запросов клиентов и управлять их жизненным циклом.

Минусы:

* Усложняет код программы из-за введения дополнительных классов.
* Увеличивает время отклика от сервиса.



class Car {

drive() { return "driving" };

}

class CarProxy {

constructor(driver) {

this.driver = driver;

}

drive() {

if( this.driver.age < 18) {

return "too young to drive"

}

return new Car().drive();

};

}

class Driver {

constructor(age) { this.age = age }

}

const driver = new Driver(17);

const car = new CarProxy(driver);

car.drive();

**Behavioural Patterns**

**Template pattern**

Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

Используется:

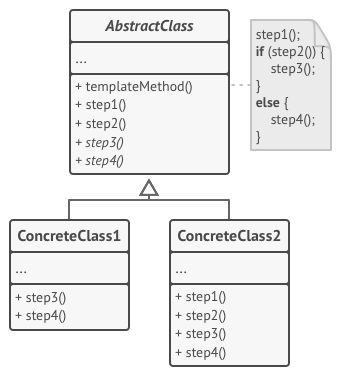
* Когда подклассы должны расширять базовый алгоритм, не меняя его структуры.
* Когда у вас есть несколько классов, делающих одно и то же с незначительными отличиями.

Минусы:

* ограничивает подклассы скелетом существующего алгоритма
* можно нарушить принцип подстановки Лисков, изменяя базовое поведение одного из шагов алгоритма через подкласс
* С ростом шагов шаблонный метод становится сложно поддерживать

Фабричный метод можно рассматривать как частный случай Шаблонного метода. Кроме того, Фабричный метод нередко бывает частью большого класса

с Шаблонными методами.



class Employee {

constructor(name, salary) {

this.\_name = name;

this.\_salary = salary;

}

work() {

this.enterTheOffice();

this.prepareToWork();

this.talkWithTeamLead()

}

// Some of these steps may be implemented right in a base class.

enterTheOffice(){

return this.\_name + ' enters an office'

}

// And some of them may be defined as abstract.

prepareToWork(){ }

talkWithTeamLead(){ }

}

class Developer extends Employee {

constructor(name, salary) {

super(name, salary);

}

// details handled by subclass

prepareToWork() {

return 'Make coffee and start PC';

}

// details handled by subclass

talkWithTeamLead() {

return 'Go to talk with C.J.';

}

}

// usage

const dev = new Developer('Nathan', 100000);

console.log(dev.work()); // 'Nathan handles application development'

**Chain of Responsibility**

Позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Используется:

* если нужны чтобы обработчики выполнялись один за другим в строгом порядке
* когда программа должна обрабатывать разнообразные запросы несколькими способами, но заранее неизвестно, какие конкретно запросы будут приходить и какие обработчики для них понадобятся.
* Когда набор объектов, способных обработать запрос, должен задаваться динамически.

Другой подход: обработчики прерывают цепь только когда они могут обработать запрос. В этом случае запрос движется по цепи, пока не найдётся обработчик, который может его обработать. Очень часто такой подход используется для передачи событий, создаваемых классами графического интерфейса в результате взаимодействия с пользователем (DOM event bubbling).

*Chain of Responsibility* часто используют вместе с *Composite*. В этом случае запрос передаётся от дочерних компонентов к их родителям.

*Chain of Responsibility* и *Decorator* имеют очень похожие структуры. Оба паттерна базируются на принципе рекурсивного выполнения операции через серию связанных объектов. Но есть и несколько важных отличий.

*Chain of Responsibility* могут выполнять произвольные действия, независимые друг от друга, а также в любой момент прерывать дальнейшую передачу по цепочке. С другой стороны *Decorators* расширяют какое-то определённое действие, не ломая интерфейс базовой операции и не прерывая выполнение остальных декораторов.

class CumulativeSum {

constructor(intialValue = 0) {

this.sum = intialValue;

}

add(value) {

this.sum += value;

return this;

}

}

// usage

const sum1 = new CumulativeSum();

console.log(sum1.add(10).add(2).add(50).sum); // 62

**Command**

Превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

Пример: один из объектов интерфейса напрямую вызывает метод одного из объектов бизнес-логики, передавая в него какие-то параметры.

Паттерн Команда предлагает больше не отправлять вызовы напрямую. Вместо этого каждый вызов, отличающийся от других, следует завернуть в собственный класс с единственным методом, который и будет осуществлять вызов. Такие объекты называют командами.

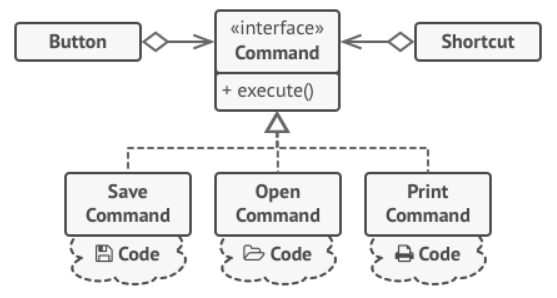
К объекту интерфейса можно будет привязать объект команды, который знает, кому и в каком виде следует отправлять запросы. Когда объект интерфейса будет готов передать запрос, он вызовет метод команды, а та — позаботится обо всём остальном.

Классы команд можно объединить под общим интерфейсом c единственным методом запуска. После этого одни и те же отправители смогут работать с различными командами, не привязываясь к их классам. Даже больше: команды можно будет взаимозаменять на лету, изменяя итоговое поведение отправителей.

Параметры, с которыми должен быть вызван метод объекта получателя, можно загодя сохранить в полях объекта-команды. Благодаря этому, объекты, отправляющие запросы, могут не беспокоиться о том, чтобы собрать необходимые для получателя данные. Более того, они теперь вообще не знают, кто будет получателем запроса. Вся эта информация скрыта внутри команды.

Используется:

* Когда вы хотите параметризовать объекты выполняемым действием.
* Когда вы хотите ставить операции в очередь, выполнять их по расписанию или передавать по сети (обьект команды можно превратить в json)
* Когда вам нужна операция отмены.



const carManager = {

// public method

execute: function ({commandName = '', carModel = '', carId = ''}){

switch (commandName) {

case 'requestInfo':

console.log(this);

this.\_requestInfo(carModel, carId);

break;

case 'buyVehicle':

this.\_buyVehicle(carModel, carId);

break;

default:

console.log('unknown request');

}

},

// commands

\_requestInfo: function( model, id ){

console.log("The information for " + model + " with ID " + id + " is foobar");

},

\_buyVehicle: function( model, id ){

console.log("You have successfully purchased Item " + id + ", a " + model);

},

};

carManager.execute({commandName: "requestInfo", carModel: "Ford Mondeo", carId: "54323"});

carManager.execute({commandName: "buyVehicle", carModel: "Ferrari", carId: "14523"});

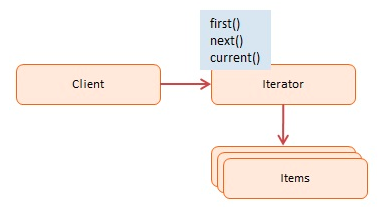
**Iterator pattern**

Даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления, вынося поведение обхода коллекции из самой коллекции в отдельный класс.

Объект-итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом. Идея паттерна Итератор состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс.

Используется:

* Когда у вас есть сложная структура данных, и вы хотите скрыть от клиента детали её реализации (из-за сложности или вопросов безопасности).
* Когда вам нужно иметь несколько вариантов обхода одной и той же структуры данных.
* Когда вам хочется иметь единый интерфейс обхода различных структур данных.



class Iterator {

constructor(items) {

this.index = 0;

this.items = items;

}

first() {

this.reset();

return this.next();

}

next() { return this.items[this.index++] }

hasNext() { return this.index <= this.items.length }

reset(){ this.index = 0 }

each(callback) {

for (let item = this.first(); this.hasNext(); item = this.next()) {

callback(item);

}

}

}

const items = ["one", 2, "circle", true, "Applepie"];

const iterator = new Iterator(items);

for (let item = iterator.first(); iterator.hasNext(); item = iterator.next()) {

console.log(item)

}

iterator.each(item => console.log(item));

**Mediator pattern**

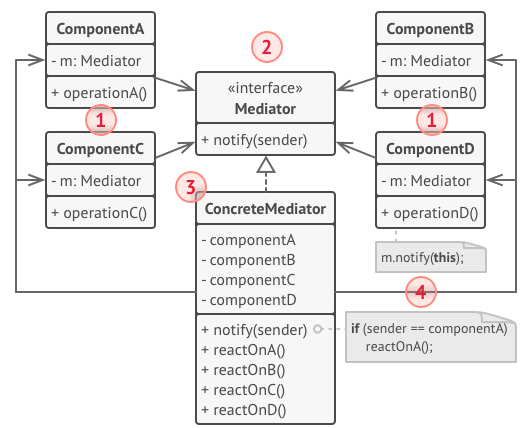
Позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник. Компоненты системы будут зависеть только от посредника, а не от десятков других компонентов. Аналогия из жизни — пилоты, которые общаются только через диспетчера

Используется:

* Когда вам сложно менять некоторые классы из-за того, что они имеют множество хаотичных связей с другими классами.
* Когда вы не можете повторно использовать класс, поскольку он зависит от уймы других классов.
* Когда вам приходится создавать множество подклассов компонентов, чтобы использовать одни и те же компоненты в разных контекстах.

Минусы: посредник может сильно раздуться

В отличии от *Observer,* цель *Medium* — убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависимыми от самого *Medium*. С другой стороны, цель *Observer* — обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других. Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать.



**Memento pattern**

Позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

Паттерн *Memento* поручает создание копии состояния объекта самому объекту, который этим состоянием владеет. Вместо того, чтобы делать снимок «извне», наш редактор сам сделает копию своих полей, ведь ему доступны все поля, даже приватные.

Паттерн предлагает держать копию состояния в специальном объекте-снимке с ограниченным интерфейсом, позволяющим, например, узнать дату изготовления или название снимка. Но, с другой стороны, снимок должен быть открыт для своего создателя, позволяя прочесть и восстановить его внутреннее состояние.

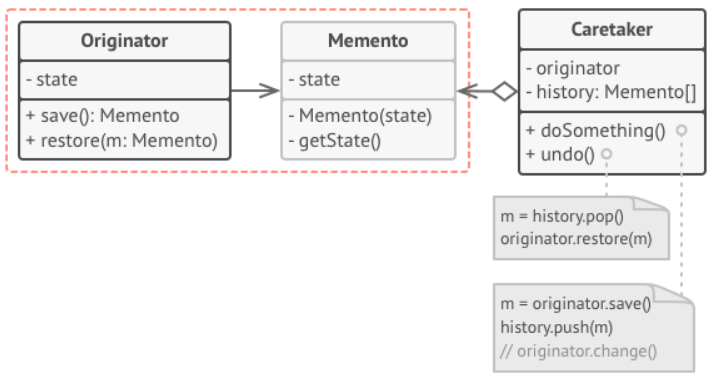
Такая схема позволяет создателям производить снимки и отдавать их для хранения другим объектам, называемым опекунами. Опекунам будет доступен только ограниченный интерфейс снимка, поэтому они никак не смогут повлиять на «внутренности» самого снимка. В нужный момент опекун может попросить создателя восстановить своё состояние, передав ему соответствующий снимок.

Используется:

* Когда вам нужно сохранять мгновенные снимки состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии.
* Когда прямое получение состояния объекта раскрывает приватные детали его реализации, нарушая инкапсуляцию.

Минусы:

* Требует много памяти, если клиенты слишком часто создают снимки.
* Может повлечь дополнительные издержки памяти, если объекты, хранящие историю, не освобождают ресурсы, занятые устаревшими снимками.
* В некоторых языках (например, PHP, Python, JavaScript) сложно гарантировать, чтобы только исходный объект имел доступ к состоянию снимка.



class Person {

constructor(name, street, city, state) {

this.name = name;

this.street = street;

this.city = city;

this.state = state;

}

hydrate() {

return JSON.stringify(this)

}

dehydrate(memento) {

const m = JSON.parse(memento);

this.name = m.name;

this.street = m.street;

this.city = m.city;

this.state = m.state;

}

}

const CareTaker = {

mementos: {},

add(key, memento) {

this.mementos[key] = memento

},

get(key) {

return this.mementos[key]

}

};

const mike = new Person("Mike Foley", "1112 Main", "Dallas", "TX");

const john = new Person("John Wang", "48th Street", "San Jose", "CA");

// save

CareTaker.add(1, mike.hydrate());

CareTaker.add(2, john.hydrate());

// mess up their names

mike.name = "King Kong";

john.name = "Superman";

// restore original state

mike.dehydrate(CareTaker.get(1));

john.dehydrate(CareTaker.get(2));

**Observer pattern**

Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

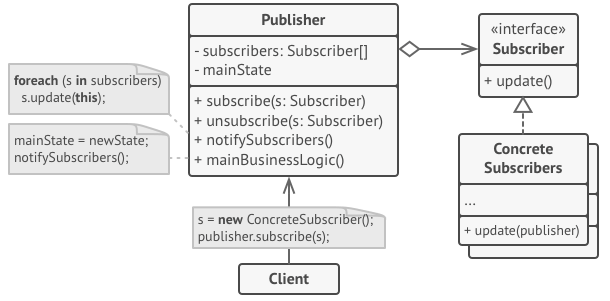
Observer предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.

Когда в издателе будет происходить важное событие, он будет проходиться по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.

Используется:

* Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать.
* Когда одни объекты должны наблюдать за другими, но только в определённых случаях.

Минусы: подписчики оповещаются в случайном порядке.



class Product {

constructor() {

this.\_price = 0;

this.\_actions = [];

}

register(observer) {

this.\_actions.push(observer);

}

unregister(observer) {

this.\_actions.remove.filter(function (el) {

return el !== observer;

});

}

setBasePrice(val) {

this.\_price = val;

this.notifyAll();

}

notifyAll() {

return this.actions.forEach(function (el) {

el.update(this);

}.bind(this));

}

}

const fees = {

update(product) {

product.price = product.price \* 1.2

}

};

const profit = {

update(product) {

product.price = product.price \* 2

}

};

const product = new Product();

product.register(fees);

product.register(profit);

product.setBasePrice(115);

**State pattern**

Позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

State pattern предлагает создать отдельные классы для каждого состояния, в котором может пребывать контекстный объект, а затем вынести туда поведения, соответствующие этим состояниям.

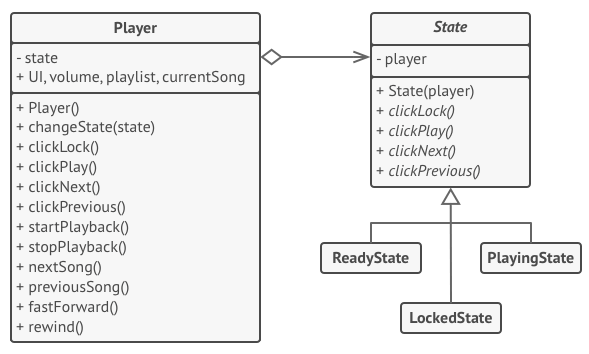
Вместо того, чтобы хранить код всех состояний, первоначальный объект, называемый контекстом, будет содержать ссылку на один из объектов-состояний и делегировать ему работу, зависящую от состояния.

Отличается от Strategy тем, что и контекст, и сами конкретные состояния могут знать друг о друге и инициировать переходы от одного состояния к другому.

Используетя:

* когда у есть объект, поведение которого кардинально меняется в зависимости от внутреннего состояния, причём типов состояний много, и их код часто меняется
* когда код класса содержит множество больших, похожих друг на друга, условных операторов, которые выбирают поведения в зависимости от текущих значений полей класса.
* когда вы сознательно используете табличную машину состояний, построенную на условных операторах, но вынуждены мириться с дублированием кода для похожих состояний и переходов.

Минусы: может неоправданно усложнить код, если состояний мало и они редко меняются.



**Strategy pattern**

Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. Чтобы сменить алгоритм, вам будет достаточно подставить в контекст другой объект-стратегию.

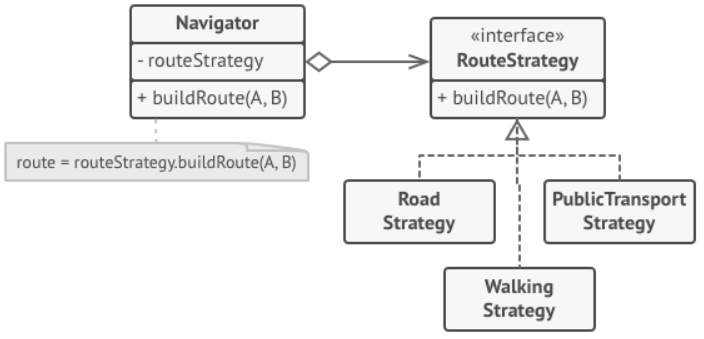
Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста.

Используется:

* Когда вам нужно использовать разные вариации какого-то алгоритма внутри одного объекта.
* Когда у вас есть множество похожих классов, отличающихся только некоторым поведением.
* Когда вы не хотите обнажать детали реализации алгоритмов для других классов.
* Когда различные вариации алгоритмов реализованы в виде развесистого условного оператора. Каждая ветка такого оператора представляет собой вариацию алгоритма.

Минусы:

* Усложняет программу за счёт дополнительных классов.
* Клиент должен знать, в чём состоит разница между стратегиями, чтобы выбрать подходящую.



// strategies

const strategyBubbleSort = {

execute: function (dataset){

// ..sorting with bubble sort

console.log('sorted by bubble sort dataset');

}

};

const strategyQuickSort = {

execute: function (dataset){

// ..sorting with quick sort

console.log('quick sorted dataset');

}

};

// Context (as a client) always works with strategies through

// a common interface.

// It does not know or care which strategy is currently active.

const context = {

\_strategy: {},

setStrategy: function (strategy){

this.\_strategy = strategy;

},

executeStrategy: function (array){

return this.\_strategy.execute();

}

};

// The concrete strategy is picked on a higher level

// (for example, by application config) and passed to the client object.

// At any time, the strategy object can be replaced

// by a different strategy.

const sorter = dataset => {

if(dataset.length > 5){

context.setStrategy(strategyQuickSort);

} else {

context.setStrategy(strategyBubbleSort);

}

return context.executeStrategy(dataset)

};

// And it can be used as

const longDataSet = [1, 5, 4, 3, 2, 8];

const shortDataSet = [1, 5, 4];

const quickSorter = sorter(longDataSet);

const bubbleSorter = sorter(shortDataSet);

quickSorter(longDataSet); // Output : Sorting with quick sort

bubbleSorter(shortDataSet); // Output : Sorting with bubble sort

**Visitor pattern**

Позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

Паттерн *Visitor* предлагает разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого вы будете передавать эти объекты в методы посетителя.

Код поведения, скорее всего, должен отличаться для объектов разных классов, поэтому и методов у посетителя должно быть несколько. Названия и принцип действия этих методов будет схож, но основное отличие будет в типе принимаемого в параметрах объекта

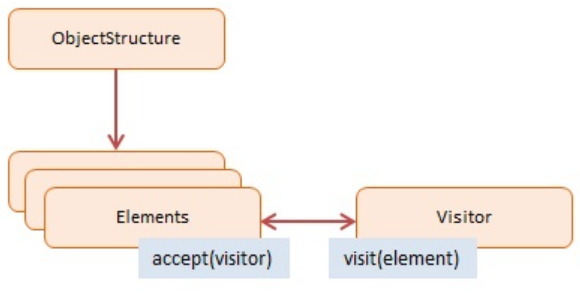
*Двойная диспетчеризация:* вместо того, чтобы самим искать нужный метод, мы можем поручить это объектам, которые передаём в параметрах посетителю. А они уже вызовут правильный метод посетителя.

Используется:

* Когда вам нужно выполнить какую-то операцию над всеми элементами сложной структуры объектов, например, деревом.
* Когда над объектами сложной структуры объектов надо выполнять некоторые не связанные между собой операции, но вы не хотите «засорять» классы такими операциями.
* Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии.

Минусы:

* Паттерн не оправдан, если иерархия компонентов часто меняется.
* Может привести к нарушению инкапсуляции компонентов.



const Employee = function (name, salary, vacation) {

const self = this;

// encapsulated methods

this.accept = function (visitor) {

visitor.visit(self);

};

this.getName = function () {

return name;

};

this.getSalary = function () {

return salary;

};

this.setSalary = function (sal) {

salary = sal;

};

this.getVacation = function () {

return vacation;

};

this.setVacation = function (vac) {

vacation = vac;

};

};

const ExtraSalary = function () {

this.visit = function (emp) {

// visitor can access encapsulated

emp.setSalary(emp.getSalary() \* 1.1);

};

};

const ExtraVacation = function () {

this.visit = function (emp) {

emp.setVacation(emp.getVacation() + 2);

};

};

const employees = [

new Employee("John", 10000, 10),

new Employee("Mary", 20000, 21),

new Employee("Boss", 250000, 51)

];

const visitorSalary = new ExtraSalary();

const visitorVacation = new ExtraVacation();

for (let i = 0, len = employees.length; i < len; i++) {

const emp = employees[i];

emp.accept(visitorSalary);

emp.accept(visitorVacation);

console.log(emp.getName() + ": $" + emp.getSalary() +

" and " + emp.getVacation() + " vacation days")

}