**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**

**Сигнатура 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.

**Интерфейс класса**

В Java можно определить абстрактный интерфейс (интрефейс != класс), который описывает только сигнатуры методов (название метода, параметры на вход и выход), но не включает атрибуты (свойства) или тело методов

Перед названием абстрактного интерфейса ставят «I». Классы, которые наследуют интерфейс, обязаны наследовать его полностью, при этом самостоятельно определить поведение каждого из методов. Один класс может наследовать несколько интрефейсов и это называется **полиморфизмом**

public interface IPublicSpeaking {

public void givePresentation();

public void speak();

}

public interface IPrivateConversation {

public void lowerVoiceVolume();

public void speak();

}

public class Person implements IPublicSpeaking, IPrivateConversation {

public void speak(){

System.out.println("hello");

}

}

**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**

*не является,* *а содержит*

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)**

*не является, а использует*

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.

Loose partnership — свободное партнерство, никак не связаны между собой

**Агрегирование**

*является*

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 inheritance 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'}

**Acquaintance**

Один обьект свободно использует другой, может быть и наоборот

const sibling = {

play() { console.log('playing') }

};

const aggregat = {

play: sibling.play

};

aggregat.play();

**Aggregation** —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** —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** —объект делегирует выполнение другому объекту

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

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

const requestHandler = (req, logger) => {

const data = req.body;

logger.write({date: new Date.now(), data})

};

const logger = {

write (date, reqData){

return console.log(reqData + ' received at ' + date)

}

};

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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.

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**

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

****

****

**Creational Patterns**

**Factory**

Делится на 2 вида: Factory Object и Factory pattern

**Factory pattern**

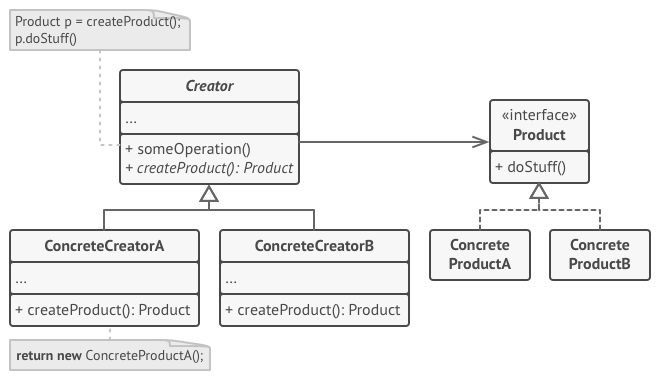
Основан на классах.

Делегирует процесс создания обьектов своим подклассам, сам же определяет для них только интерфейс. Часто используется, когда нужно манипулировать коллекциями обьектов которые похожи между собой

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



**Factory object**

Обьект создает обьекты. Принцип — вынести процесс создания обьектов из продукта в отдельный обьект-фабрику, продукт же будет делегировать процесс создания этой фабрике

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();

**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**

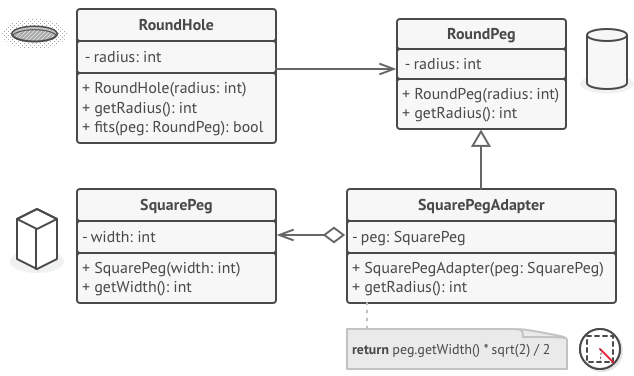
**Adapter**

Интерфейс одного класса транслируется 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

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

В примере Адаптер преобразует один интерфейс в другой, позволяя совместить квадратные колышки и круглые отверстия.



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();

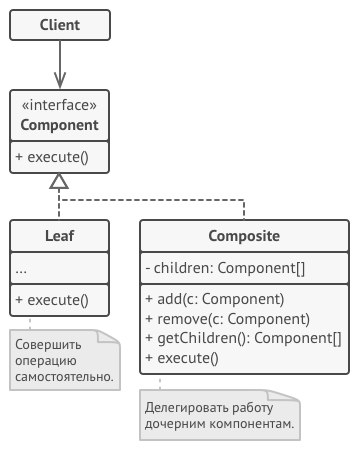
adapter.operations(5, 10, 'add');



**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.

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



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

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**

Используется для сокрытия сложного функционала, предоставления единого и простого публичного интерфейса. Фасад создаёт упрощённый интерфейс к подсистеме, не внося в неё никакой добавочной функциональности. Сама подсистема не знает о существовании Фасада. Классы подсистемы общаются друг с другом напрямую.

* 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

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 }

}

**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:

* Усложнаяет 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() {

return ( this.driver.age < 18) ? "too young to drive" : 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:

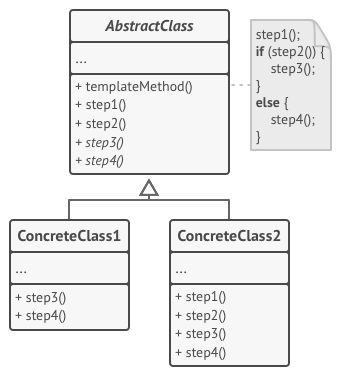
* Когда подклассы должны расширять базовый алгоритм, не меняя его структуры.
* Когда у вас есть несколько классов, делающих одно и то же с незначительными отличиями.

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.

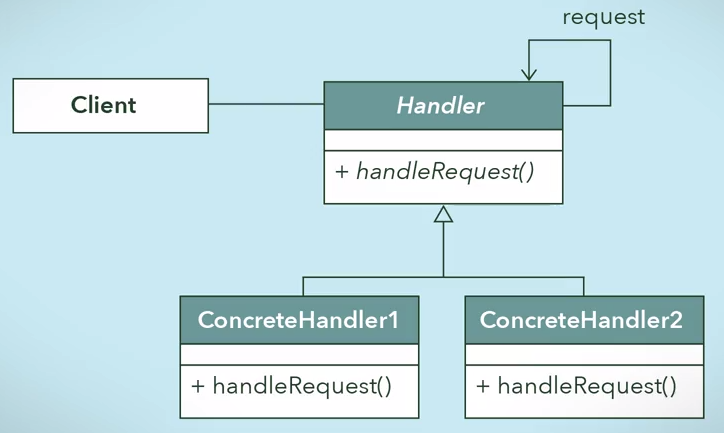
Используется:

* 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.



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 pattern**

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.

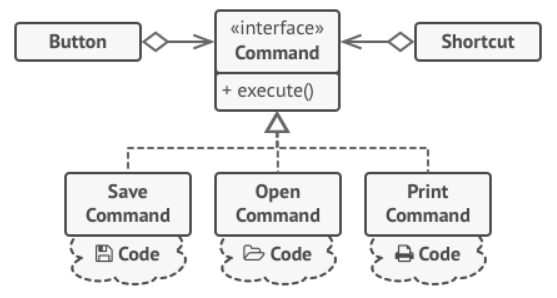
К объекту интерфейса можно будет привязать объект команды, который знает, кому и в каком виде следует отправлять запросы. Когда объект интерфейса будет готов передать запрос, он вызовет метод команды, а та — позаботится обо всём остальном.

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"});

**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.

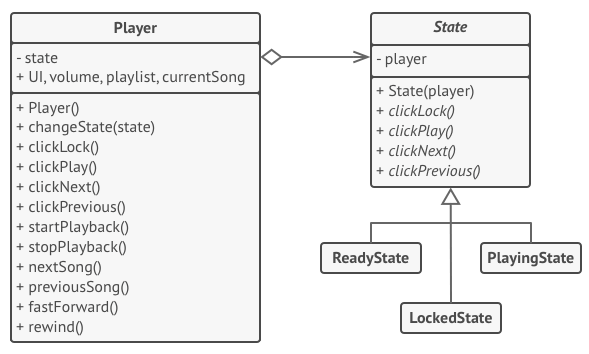
Вместо того, чтобы хранить код всех состояний, первоначальный объект, называемый контекстом, будет содержать ссылку на один из объектов-состояний и делегировать ему работу, зависящую от состояния.

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.

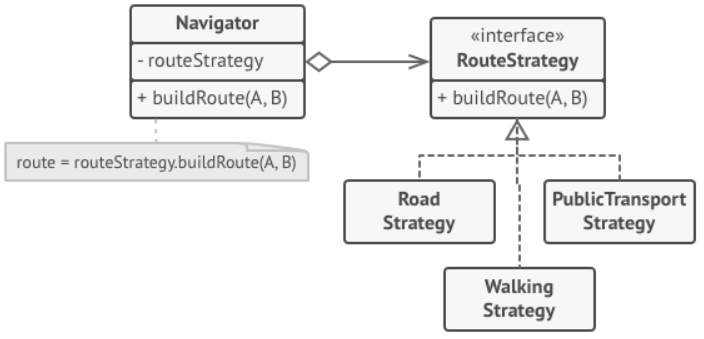
Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста.

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

**Mediator pattern**

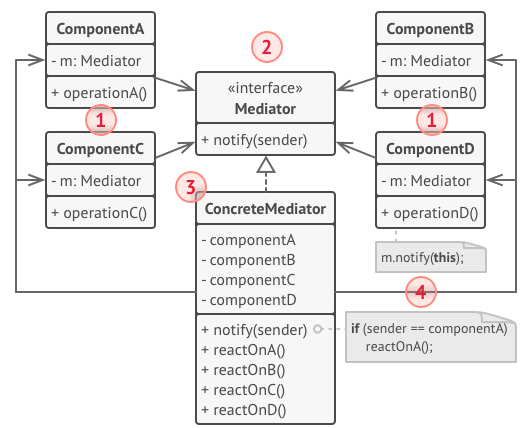
Позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник. Компоненты системы будут зависеть только от посредника, а не от десятков других компонентов. Аналогия из жизни — пилоты, которые общаются только через диспетчера

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.



**Observer pattern**

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

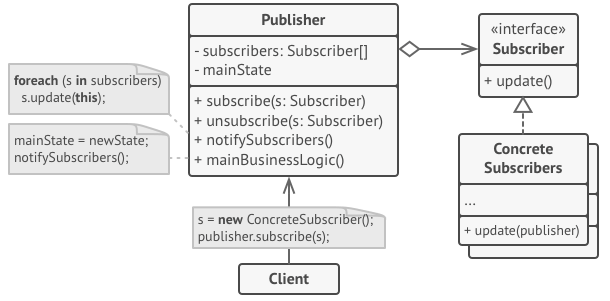
Observer предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.

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);

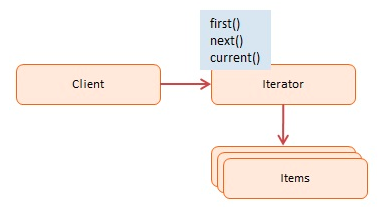
**Iterator pattern**

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));

**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.

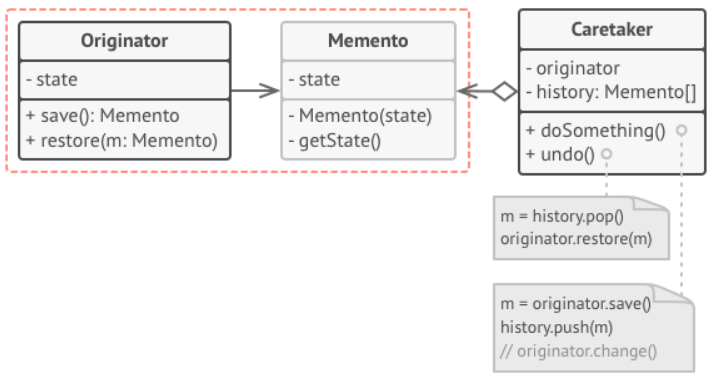
Такая схема позволяет создателям производить снимки и отдавать их для хранения другим объектам, называемым опекунами. Опекунам будет доступен только ограниченный интерфейс снимка, поэтому они никак не смогут повлиять на «внутренности» самого снимка. В нужный момент опекун может попросить создателя восстановить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

**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

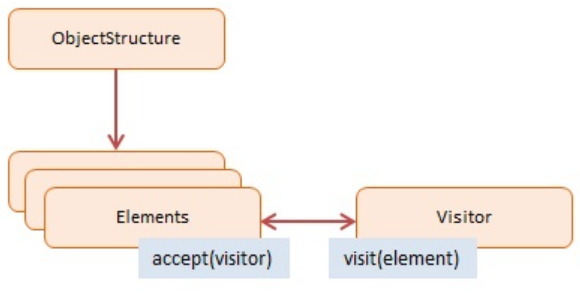
*Двойная диспетчеризация:* вместо того, чтобы самим искать нужный метод, мы можем поручить это объектам, которые передаём в параметрах посетителю. А они уже вызовут правильный метод посетителя.

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")

}