**Лекция 13.**

**JavaScript: Основные JavaScript-паттерны.**

[Что такое паттерн? 3](#_Toc518867876)

[Порождающие шаблоны 4](#_Toc518867877)

[Простая фабрика (Simple Factory) 4](#_Toc518867878)

[Одиночка (Singleton) 5](#_Toc518867879)

[Строитель (Builder) 6](#_Toc518867880)

[Прототип (Prototype) 7](#_Toc518867881)

[Структурные шаблоны 9](#_Toc518867882)

[Декоратор (Decorator) 9](#_Toc518867883)

[Компоновщик (Composite) 11](#_Toc518867884)

[Фасад (Facade) 12](#_Toc518867885)

[Поведенческие шаблоны 14](#_Toc518867886)

[Наблюдатель (Observer) 14](#_Toc518867887)

[Посредник (Mediator) 15](#_Toc518867888)

[Мемоизация 18](#_Toc518867889)

[Дополнительные материалы 19](#_Toc518867890)

[Задание для закрепления материала 20](#_Toc518867891)

# Что такое паттерн?

Шаблон проектирования или паттерн (англ. design pattern) в разработке программного обеспечения — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Проще говоря, дизайн паттерн представляет собой многократно используемое программное решение для определенного типа проблемы, которая часто возникает при разработке программного обеспечения. За много лет практики разработки ПО, эксперты выяснили пути решения подобных проблем. Эти решения были инкапсулированы в шаблоны дизайна.

Итак:

* шаблоны – это проверенное решения проблем разработки программного обеспечения
* шаблоны являются масштабируемыми, поскольку они обычно структурированы и имеют правила, которых вы должны придерживаться
* шаблоны могут быть повторно использованы для аналогичных задач
* Разновидности шаблонов проектирования

При разработке программного обеспечения, дизайн паттерны, как правило, сгруппированы в несколько категорий. Вот, например, три наиболее важных:

1. Порождающие шаблоны (creational patterns) сосредоточены на способах создания объектов или классов. Это может показаться простым (и это в некоторых случаях оно так и есть), но большие приложения должны контролировать процесс создания объекта.
2. Структурные шаблоны (structural design patterns) сосредоточены на том, чтобы управлять отношениями между объектами так, чтобы ваше приложение было построено масштабируемым способом. Ключевым аспектом структурной модели является обеспечение того, что изменение в одной части приложения не влияет на все другие части.
3. Поведенческие шаблоны (behavioral patterns) сосредоточены на связи между объектами

# Порождающие шаблоны

Порождающие шаблоны — шаблоны проектирования, которые абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов.

Существует много различных видов порождающих шаблонов (абстрактная фабрика, строитель, фабричный метод, отложенная инициализация, пул одиночек, объектный пул, прототип, получение ресурса есть инициализация, одиночка), но мы рассмотрим только основные из них, которые используются довольно часто.

## Простая фабрика (Simple Factory)

Собственно, основной задачей фабрики в статически типизируемых языках является создание разных объектов с одинаковым интерфейсом, в зависимости от ситуаций, в JavaScript этак проблема так остро не стоит, так что появляется вопрос — зачем эта фабрика тут вообще нужна?

Все просто — помимо этой, первой, цели, у нее есть еще и вторая — фабрика может проводить какую-то первичную инициализацию объектов.

Собственно, для того чтобы фабрика могла создавать какие-то объекты, для них сначала неплохо бы задать конструкторы:

var Shapes = {

Circle: function (param){

console.log("new " + param.color + " circle created with radius " + param.radius + "px");

},

Square: function (param){

console.log("new " + param.color + " square created with " + param.side + "px on a side ");

},

Triangle: function (param){

console.log("new " + param.color + " triangle created with " + param.side + "px on a side ");

}

}

А теперь можно сделать и саму фабрику — выглядеть она может так:

function ShapeFactory(size, color){

this.size = size;

this.color = color;

}

ShapeFactory.prototype = {

constructor: ShapeFactory,

makeCircle: function () {

return new Shapes.Circle({ radius: this.size / 2, color: this.color });

},

makeSquare: function () {

return new Shapes.Square({ side: this.size, color: this.color });

},

makeTrinagle: function () {

return new Shapes.Triangle({ side: this.size, color: this.color });

}

}

И проверяем:

var factory = new ShapeFactory(100, "red")

factory.makeSquare();

factory.makeCircle();

factory.makeTrinagle();

## Одиночка (Singleton)

Синглтон - это объект, который есть в системе в одном экземпляре с глобальной точкой доступа.

Есть два способа создания:

**Способ первый — тривиальный**

var singleton\_A = {

log: function( text ){ console.log(text); }

}

Любой объект, созданный в глобальной области видимости, является синглтоном, так как даже при присвоение его в другую переменную – сохраняется ссылка на первый и любое изменение ведет к изменению инициализирующего объекта.

Способ второй — хитрый

var Singleton\_B;

(function(){ // данная обертка не позволяет «злоумышленнику» изменить instance из вне

var instance;

var anticlone\_proxy;

Singleton\_B = function(){

if( instance ){ return instance; }

instance = { //создаем экземпляр и инкапсулируем данные

\_counter: 0,

log: function( text ){

this.\_counter++; console.log( text + this.\_counter );

}

}

anticlone\_proxy = { //в первый раз в публичное пространство отдаем копию, чтобы никто не изменил первоначальный функционал

log: function( text ){ return instance.log( text ); }

}

return anticlone\_proxy;

};

})();

var singleton = new Singleton\_B();

singleton.log("3..2..1... ignition!");

## Строитель (Builder)

Паттерн Строитель (Builder pattern) инкапсулирует конструирование продукта и позволяет разделить его на этапы, тем самым инкапсулируется процесс создания сложного объекта. Данный паттерн относится к порождающим паттернам, дает возможность поэтапного конструирования объекта с переменным набором этапов (в отличие от "одноэтапных" фабрик). Также данный паттерн скрывает внутреннее представление продукта от клиента. Часто используется для создания составных структур. Предназначен для решения проблемы антипаттерна «Телескопический конструктор» (когда метод конструктор начинает разрастаться большим кол-вом параметров).

Простыми словами: Шаблон позволяет вам создавать различные виды объекта, избегая засорения конструктора. Он полезен, когда может быть несколько видов объекта или когда необходимо множество шагов, связанных с его созданием.

var Builder = function() {

var a = "defaultA";

var b = "defaultB";

return {

withA : function(anotherA) {

a = anotherA;

return this;

},

withB : function(anotherB) {

b = anotherB;

return this;

},

build : function() {

return "A is: " + a +", B is: " + b;

}

};

};

var builder = new Builder();

console.log(builder.build());

var first = builder.withA("newA").withB("newB").build();

var second = builder.withB("secondB").build();

var third = builder.withA("someA").build();

console.log(first);

console.log(second);

console.log(third);

## Прототип (Prototype)

Прототипный шаблон – это шаблон, в котором объекты создаются на основе шаблона существующего объекта путем клонирования.

Это простой и естественный способ реализации наследования в JavaScript. Например:

var Person = {

numFeet: 2,

numHeads: 1,

numHands:2

};

var tilo = Object.create(Person); //Object.create берет свой ​​первый аргумент и применяет его к прототипу нового объекта.

console.log(tilo.numHeads); //результат 1

tilo.numHeads = 2;

console.log(tilo.numHeads) //результат 2

Свойства (и методы) в Person объекте, применяются к прототипу объекта tilo. Мы можем переопределить свойства объекта tilo, если хотим, чтобы они были разными.

В приведенном выше примере, мы использовали Object.create (). Однако, Internet Explorer 8 не поддерживает новый метод. В этих случаях мы можем имитировать его поведение:

var vehiclePrototype = {

init: function (carModel) {

this.model = carModel;

},

getModel: function () {

console.log( "The model of this vehicle is " + this.model);

}

};

function vehicle (model){

function F() {};

F.prototype = vehiclePrototype;

var f = new F();

f.init(model);

return f;

}

var car = vehicle("Ford Escort");

car.getModel();

# Структурные шаблоны

Структурные шаблоны действительно полезны, особенно при выяснении, как система должна работать. Они позволяют приложениям легко масштабироваться и оставаться управляемыми. Из этой немалой группы (адаптер, мост, компоновщик, декоратор, фасад, единая точка входа, приспособленец, заместитель), мы рассмотрим опять же только основные

## Декоратор (Decorator)

Декоратор – приём программирования, который позволяет взять существующую функцию и изменить/расширить ее поведение. Декоратор получает функцию и возвращает обертку, которая делает что-то своё «вокруг» вызова основной функции.

Допустим, у нас есть такой код:

function Ball( param ) {

this.\_radius = param.radius;

this.\_color = param.color;

}

Ball.prototype = {

constructor: Ball,

INCREMENTATION\_STEP: 5,

draw: function(){

console.log("ball drawn with radius:" + this.\_radius + " and color: " + this.\_color)

},

inc: function(){

this.\_radius += this.INCREMENTATION\_STEP

}

}

new Ball({ radius:100, color:"red"});

Здесь мы создаем новый красный мячик, а что делать если мячик нужен не просто красный, а красный в полоску? Вот тут на сцену и выходит Decorator. И самое главное первоначальный Ball вообще не подозревает о том, что он может быть в полоску, или что у него могут быть какие-то там декораторы.

Реализовать паттерн можно несколькими способами:

**Способ первый — комплексный**

function StripedBall( ball ){

this.\_ball = ball

}

StripedBall.prototype = {

constructor: StripedBall,

draw: function(){

this.\_ball.draw();

console.log("and with stripes");

},

inc: function(){

return this.\_ball.inc();

}

}

В каждом декораторе нужно воссоздать все функции которые должны быть в объекте родителе, и в тех из них, поведение которых мы менять не хотим, нужно просто перенаправлять запрос родителю. Этот способ лучше применять когда происходят серьезные изменения, которые затрагивают > 1 — 2 функций.

Проверяем результат:

var ball1 = new StripedBall( new Ball({ radius:100, color:"red"}));

ball1.draw();

ball1.inc();

ball1.draw();

**Способ второй — легковесный**

function MakeStripedBall( ball ) {

var function\_name = "draw";

var prev\_func = ball[ function\_name ];

ball[ function\_name ] = function(){

prev\_func.apply( this, arguments )

console.log("and with stripes");

};

return ball;

}

Кода, конечно, нужно меньше чем в первом случае, зато, если изменяемых функций больше чем 1-2, или изменения комплексные — разобраться во всем этом будет намного сложнее.

Проверим результат:

var ball2 = MakeStripedBall( new Ball({ radius: 150, color: "blue" }));

ball2.draw();

ball2.inc();

ball2.draw();

## Компоновщик (Composite)

Компоновщик — структурный шаблон проектирования, объединяющий объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково. Паттерн определяет иерархию классов, которые одновременно могут состоять из примитивных и сложных объектов, упрощает архитектуру клиента, делает процесс добавления новых видов объекта более простым.

Пример из жизни: Каждая организация скомпонована из сотрудников. У каждого сотрудника есть одинаковые свойства, такие как зарплата, обязанности, отчётность и т.д. Компановщик позволит, например, поднять всем зарплату с помощью одного метода или только одному сотруднику или определенной группе лиц.

Простыми словами: Шаблон компоновщик позволяет клиентам работать с индивидуальными объектами в едином стиле.

Давайте, рассмотрим следующий пример в jQuery

$('.myList').addClass('selected');

$('#myItem').addClass('selected');

//Не делайте этого на больших таблицах, это всего лишь пример.

$("#dataTable tbody tr").on("click", function(event){

alert($(this).text());

});

$('#myButton').on("click", function(event) {

alert("Clicked.");

});

Большинство библиотек JavaScript обеспечивают последовательное API, независимо от того, имеем мы дело с одним элементом DOM или массивом DOM элементов. В первом примере, мы можем добавить selected класс ко все элементам подобраных селектором .myList, но мы также можем использовать этот же метод, когда речь идет об еденичном DOM элементе, #myItem. Точно так же можно приложить обработчик событий с помощью on() метода на нескольких узлах, или на одном узле через тот же API.

Благодаря использованию композитных макетов, jQuery (и многие другие библиотеки) предоставляют нам упрощенный API.

## Фасад (Facade)

Фасад — структурный шаблон проектирования, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

Пример из жизни: Как вы включаете компьютер? Нажимаю на кнопку включения, скажете вы. Это то, во что вы верите, потому что вы используете простой интерфейс, который компьютер предоставляет для доступа снаружи. Внутри же должно произойти гораздо больше вещей. Этот простой интерфейс для сложной подсистемы называется фасадом.

Простыми словами: Шаблон фасад предоставляет упрощенный интерфейс для сложной системы.

Фасад шаблон, почти всегда улучшает удобство использования части программного обеспечения. Использование jQuery в качестве примера, одним из наиболее распространенных методов библиотеки, является ready() метод:

$(document).ready(function() {

//весь ваш код идет сюда...

});

Метод ready() фактически реализует фасад. Если взглянуть на источник, вот что вы найдете:

ready: (function() {

...

//Mozilla, Opera, и Webkit

if (document.addEventListener) {

document.addEventListener("DOMContentLoaded", idempotent\_fn, false);

...

}

//IE модель событий

else if (document.attachEvent) {

// обеспечьте firing до onload; может быть поздно, но безопасно для iframes

document.attachEvent("onreadystatechange", idempotent\_fn);

// Резерв для window.onload, который всегда работает

window.attachEvent("onload", idempotent\_fn);

...

}

})

Метод ready() не такой уж простой. jQuery нормализует непостоянство браузера, чтобы ready() сработал в нужный момент. Однако, как разработчик, вы будете представлены с простым интерфейсом.

Большинство примеров фасад шаблонов, следует этому принципу. Для его реализации, мы обычно полагаемся на условные операторы, но представляем его в виде простого интерфейса для пользователя.

# Поведенческие шаблоны

Любая объектно-ориентированная система программного обеспечения будет иметь связи между объектами. Не организация таких связей может привести к ошибкам, которые трудно найти и исправить. Поведенческие шаблоны проектирования преписывают различные методы организации связи между объектами (цепочка ответственности, команда, интерпретатор, итератор, посредник, хранитель, наблюдатель, слуга, спецификация, состояние, стратегия, шаблонный метод, посетитель и т.д.). В этом раздел, мы рассмотрим Наблюдатель (Observer) и Посредник (Mediator) паттерны.

## Наблюдатель (Observer)

Наблюдатель — поведенческий шаблон проектирования, также известен как «подчинённые» (Dependents). Создает механизм у класса, который позволяет получать экземпляру объекта этого класса оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними.

Пример из жизни: Хороший пример: люди, ищущие работу, подписываются на публикации на сайтах вакансий и получают уведомления, когда появляются вакансии подходящие по параметрам.

Простыми словами: Шаблон определяет зависимость между объектами, чтобы при изменении состояния одного из них зависимые от него узнавали об этом.

Нам нужно три метода для описания этого паттерна:

* raise(data): вызывается объектом, когда у него есть уведомление. Некоторые данные могут быть переданы с помощью этого метода.
* subscribe(observer): вызывается объектом, для добавления наблюдателя в свой список наблюдателей.
* unsubscribe(observer): вызывается объектом, чтобы удалить наблюдателя из списка наблюдателей.

Event = function(){

this.\_observers = [];

}

Event.prototype = {

raise: function (data){

for (var i in this.\_observers){

var item = this.\_observers[i];

item.observer.call(item.context, data);

}

},

subscribe: function (observer, context){

var ctx = context || null;

this.\_observers.push({ observer: observer, context: ctx });

},

unsubscribe: function (observer, context ){

for (var i in this.\_observers)

if ( this.\_observers[i].observer == observer &&

this.\_observers[i].context == context )

delete this.\_observers[i];

}

}

var someEvent = new Event();

someEvent.subscribe(function ( data ) { console.log("wohoooooo " + data ) });

var someObject = {

\_topSecretInfo: 42,

observerFunction: function () { console.log("Top Secret:" + this.\_topSecretInfo) }

}

someEvent.subscribe(someObject.observerFunction, someObject);

someEvent.raise("yeaah!");

someEvent.raise();

## Посредник (Mediator)

Объяснить, что представляет собой паттерн «медиатор» достаточно просто на примере следующей аналогии — представьте себе контроль трафика в аэропорту: все решения о том, какие самолеты могут взлетать или садиться, принимает диспетчер. Для этого, все сообщения, исходящие от самолетов, поступают в башню управления, вместо того, чтобы пересылаться между самолетами напрямую. Такой централизованный контроллер — это и есть ключ к успеху нашей системы. Это и есть «медиатор».

Mediator — это шаблон проектирования, который помогает в особо сложных случаях взаимодействия между объектами, например, когда у нас, скажем, 5 объектов более-менее разного типа, и все почему-то знают друг о друге, стоит серьезно задуматься о медиаторе.

Медиатор выступает в качестве посредника в общении между различными модулями, инкапсулируя их взаимодействие. Кроме того, этот шаблон проектирования, предотвращая прямое взаимодействие различных компонентов системы, способствует ослаблению связей в коде.

В качестве подготовки сначала сделаем несколько классов, которые в перспективе медиатор будут использовать (подсказка: в данном случае медиатор будет называться kitchen)

function Daddy() { }

Daddy.prototype = {

constructor: Daddy,

getBeer: function (){

if (!kitchen.tryToGetBeer()){

console.log("Daddy: Who the hell has drank all my beer?");

return false;

}

console.log("Daddy: Yeeah! My beer!");

kitchen.oneBeerHasGone();

return true;

},

argue\_back: function () {

console.log("Daddy: it's my last beer, for sure!");

}

}

function Mammy() { }

Mammy.prototype = {

constructor: Mammy,

argue: function (){

console.log("Mammy: You are an alcoholic!");

kitchen.disputeStarted();

}

}

function BeerStorage(beer\_bottle\_count){

this.\_beer\_bottle\_count = beer\_bottle\_count;

}

BeerStorage.prototype = {

constructor: BeerStorage,

takeOneBeerAway: function (){

if (this.\_beer\_bottle\_count == 0) return false;

this.\_beer\_bottle\_count--;

return true;

}

}

А теперь пора написать и сам медиатор:

var kitchen = {

daddy: new Daddy(),

mammy: new Mammy(),

refrigerator: new BeerStorage(3),

stash: new BeerStorage(2),

tryToGetBeer: function (){

if (this.refrigerator.takeOneBeerAway()) return true;

if (this.stash.takeOneBeerAway()) return true;

return false

},

oneBeerHasGone: function (){ this.mammy.argue(); },

disputeStarted: function (){ this.daddy.argue\_back(); }

}

И так, у нас есть 4 объекта работа со взаимодействием между которыми, могла бы превратиться в неплохое наказание, если бы проходила не через Mediator.

Пишем код проверки:

var round\_counter = 0;

while (kitchen.daddy.getBeer()){

round\_counter++

console.log( round\_counter + " round passed");

}

# Мемоизация

Мемоизация — сохранение результатов выполнения функций для предотвращения повторных вычислений. Это один из способов оптимизации, применяемый для увеличения скорости выполнения компьютерных программ. Перед вызовом функции проверяется, вызывалась ли функция ранее: если не вызывалась, функция вызывается и результат её выполнения сохраняется; если вызывалась, используется сохранённый результат.

Мемоизация полезна, когда вы передаёте в функцию заранее известный набор аргументов и когда результат функции будет всегда одинаковым при одинаковых аргументах. Если же функция не даёт одинакового результата при тех же аргументах, то мемоизация будет бесполезна.

function calculation(x, y){

var key = x.toString() + "|" + y.toString();

var result = 0;

if (!calculation.memento[key]){

for (var i = 0; i < y; ++i) result += x;

calculation.memento[key] = result;

}

return calculation.memento[key];

}

calculation.memento = {};

# Дополнительные материалы

Паттерны в JS

* <https://habr.com/post/132472/>
* <https://lpgenerator.ru/blog/2016/03/11/ponimanie-shablonov-proektirovaniya-v-javascript/>

Мемоизация

* <https://habr.com/company/ruvds/blog/332384/>
* <http://langtoday.com/?p=220>

Общая информация:

* <https://tproger.ru/translations/design-patterns-simple-words-1/>
* <https://tproger.ru/translations/design-patterns-simple-words-2/>
* <https://tproger.ru/translations/design-patterns-simple-words-3/>

# 

# Задание для закрепления материала

1. Вывести на экран два калькулятора, работающие независимо друг от друга
2. Описать в ReadMe файле как минимум три паттерна, которые вы используете в вашем приложении (с указанием места расположения куска кода), если их нет, то реализовать