
Programacion Orientada a Objetos 2

Ing. Jose Rusca
Ing. Federico Stulich
Ing. Francisco De Grandis

Que tecnología vamos a utilizar?

Lenguaje

JavaScript



Entorno de ejecución

Node.js



Editor de Código

Visual Studio Code



Que tecnologia vamos a utilizar?

Para Node descargar desde la pagina oficial.

<https://nodejs.org/>

Para Visual Studio Code desde la pagina oficial.

<https://code.visualstudio.com/>

Pueden utilizar otro editor de código si lo desean como Atom, Sublime o Notepad++

Que tecnología vamos a utilizar?

¿Todos tenemos
instaladas las
herramientas Básicas para
comenzar la proxima
clase?



Que tecnología vamos a utilizar?

También podemos ejecutar nuestro código JavaScript desde la consola del navegador.

Este modo no es tan práctico para la ejecución de un programa completo, pero nos puede servir para realizar pruebas.

Como abrir la consola del navegador.



Ctrl + Shift + J

o

F12 y seleccionar Console



Ctrl + Shift + K

o

F12 y seleccionar Console



Ctrl + Alt + I (opciones avanzadas “Show Develop menu in menu bar”)



F12



Ctrl + Alt + I

Programación Orientada a Objetos

Programación Orientada a Objetos

“Un paradigma es un modelo que especifica un marco lógico que contiene supuestos, verdades, metodologías y herramientas para la resolución de problemas”

Elementos de la POO

Objeto

Representación de un concepto de la realidad; puede ser visto como una unidad funcional. Posee características y comportamientos asociados.

El estado interno de un objeto está definido por el valor de sus características en un momento dado.

Elementos de la POO

Mensaje

Aquello que un objeto sabe responder, representa un comportamiento que el objeto puede realizar.

Elementos de la POO

Método

Define cómo el objeto responde a un mensaje, es decir, cómo lleva a cabo el mensaje recibido.

Elementos de la POO

Programa Orientado a Objetos

Colección de objetos que interactúan entre sí a través de mensajes para realizar tareas y resolver problemas.

Características de la POO

- Abstracción
- Encapsulamiento
- Ocultamiento
- Polimorfismo
- Herencia

Características de la POO

Abstracción

Permite enfocarse en lo relevante al dominio del problema, dejando de lado factores que no lo son.

Características de la POO

Encapsulamiento

Permite definir bajo una única unidad (cápsula) un concepto en particular y todo lo relacionado a él.

Características de la POO

Ocultamiento

Le da la posibilidad al objeto de definir cómo se mostrará e interactuar con el resto de los objetos, ocultando su estado interno.

Características de la POO

Polimorfismo

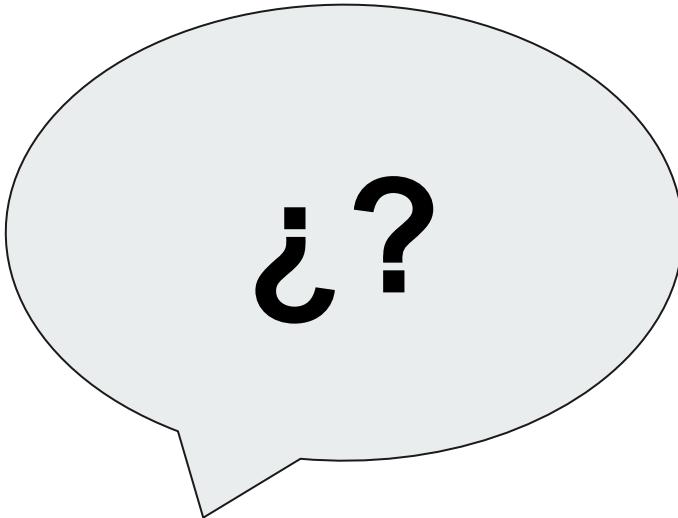
Define la capacidad que tienen distintos objetos de responder al mismo mensaje de manera distinta.

Características de la POO

Herencia

Permite que un concepto adquiera o herede características y comportamientos de otro de orden superior.

Desventajas de la POO

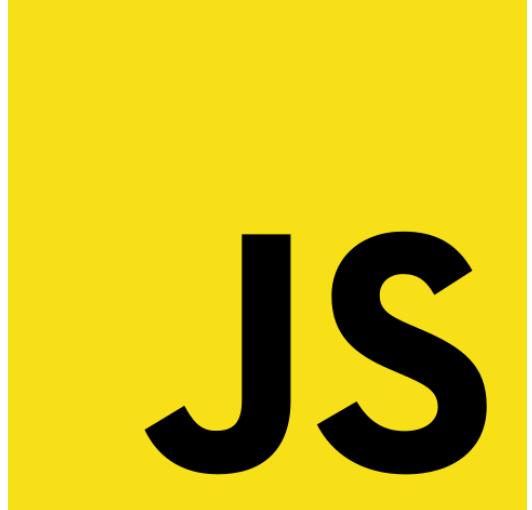


¿?

Próxima Clase

Conceptos Fundamentales

JavaScript



JS

Programacion Orientada a Objetos 2

Ing. Jose Rusca
Ing. Federico Stulich
Ing. Francisco De Grandis

Conceptos Fundamentales

1. Alcance de Variables
2. Funciones anónimas
3. Callbacks
4. Funciones flecha
5. Clausuras
6. Spread Operators
7. Manejo de arreglos

Variables - Alcance

Declaración de variables:

1. var -> Alcance a nivel de función
2. let -> Alcance a nivel de bloque
3. const -> Alcance a nivel de bloque

Variables - ejemplo var

```
function ejemploVar() {  
  
    var var1 = 10;  
  
    if(var1 === 10) {  
  
        var var2 = 40;  
  
        console.log("var1: " + var1 );  
  
        console.log("var2: " + var2 );  
  
    }  
  
    console.log("var1: " + var1 );  
  
    console.log("var2: " + var2 );  
  
}
```

{}

Variables - ejemplo let

```
function ejemploLet() {  
  
    let let1 = 10;  
  
    if(let1 === 10) {  
  
        let let2 = 20;  
  
        console.log("let1: " + let1 );  
  
        console.log("let2: " + let2 );  
  
    }  
  
    console.log("let1: " + let1 );  
  
    console.log("let2: " + let2 );  
  
}
```

Va a producir un error en
tiempo de ejecución.

Funciones anónimas

Son funciones sin nombre o identidad, que pueden ser utilizadas para ser pasadas como parámetros o en el caso que se tenga que retornar una función.

También conocidas como funciones lambda.

Funciones anónimas - con parámetro

{

```
var respuesta = function (nombre) {  
    console.log("Hola " + nombre + "! Soy una funcion  
anonima");  
}  
  
respuesta("Juan");
```

Funciones Flecha (Arrow Functions)

Una función flecha es una alternativa compacta a una función tradicional, pero con ciertas limitaciones.

Nos va a permitir escribir una función de forma más rápida y práctica pero debemos tener en cuenta los siguientes puntos.

Funciones Flecha (Arrow Functions)

Diferencias con las funciones clásicas:

1. La función flecha no define su propio contexto (this)
2. No puede ser utilizada como constructor (debido al punto 1)
3. Poseen un return implícito si no se define un bloque con llaves.

En la especificación del lenguaje vamos a encontrar más diferencias, pero las indicadas son las necesarias para el presente curso.

Funciones Flecha (Arrow Functions)

```
() => { ... instrucciones ...}
```

```
(a,b,c) = > {let res=a+b+c;  
               return res;}
```

{}

```
(a,b,c) => a+b+c;
```

```
a => a + 10;
```

Funciones Flecha (Arrow Functions)

`() => { ... instrucciones ... }`

```
(a,b,c) => {let res=a+b+c;  
              return res;}
```

{}

`(a,b,c) => a+b+c;`

`a => a + 10;`

Con corchetes y valor de
retorno explicito.

Funciones Flecha (Arrow Functions)

```
() => { ... instrucciones ...}
```

```
(a,b,c) = > {let res=a+b+c;  
               return res;}
```

{}

```
(a,b,c) => a+b+c;
```

```
a => a + 10;
```

Sin corchetes y valor de
retorno implicito.

Funciones Flecha (Arrow Functions)

```
() => { ... instrucciones ...}
```

```
(a,b,c) => {let res=a+b+c;
```

```
        return res;}
```

```
{}
```

```
(a,b,c) => a+b+c;
```

```
a => a + 10;
```

Con un parámetro
podemos omitir los
paréntesis.

Funciones anónimas - como parametro (callback)

```
setTimeout(function () {  
    console.log("Saludo en 1 segundo.");  
}, 1000);
```

Callbacks

Los Callbacks son funciones que se pasan como parámetros y van a ser utilizadas por la función.

Al igual que en el TimeOut, podemos definir un parámetro donde recibimos una función y la invocamos.

Callbacks

```
let presentacionFormal = (nombre, apellido)  
    => "En este acto, presentamos a " + nombre + " " + apellido;
```

{}

```
let presentacionInformal = (nombre, apellido)  
    => "Bienvenido " + nombre + " " + apellido;
```

Callbacks

```
function presentarPersona(nombre, apellido, funcionPresentacion) {  
    console.log(funcionPresentacion(nombre, apellido));  
}  
  
{  
presentarPersona("Juan", "Garcia", presentacionFormal);  
  
presentarPersona("Juan", "Garcia", presentacionInformal);
```

Clausuras (Closure)

Una clausura es una función que guarda referencias del estado adyacente.

Una clausura permite acceder al ámbito de una función exterior desde una función interior.

En **JavaScript**, las clausuras se crean cada vez que una función es creada.

Clausuras (Closure)

```
function creaFunc() {  
  
    var nombre = "Programacion Objetos 2";  
  
    function muestraNombre() {  
  
        console.log(nombre); //Llamamos a la funcion.  
  
    }  
  
    return muestraNombre;  
  
}
```

```
var miFunc = creaFunc();  
  
miFunc();
```

Clausuras (Closure) - Un ejemplo interesante

```
function creaSumador(x) {  
  
    return function(y) {  
  
        return x + y;  
  
    };  
  
}  
  
var suma5 = creaSumador(5);  
  
var suma10 = creaSumador(10);
```

// llamamos a la funcion.

```
console.log(suma5(2));  
// muestra 7  
  
console.log(suma10(2));  
// muestra 12
```

Sintaxis Spread

La sintaxis extendida permite a un elemento iterable tal como un arreglo o cadena ser expandido en lugares donde cero o más argumentos o elementos son esperados.

Puede ser utilizado en llamadas a función o arreglos literales

Sintaxis Spread - para pasar parametros

```
function sum(x, y, z) {  
    return x + y + z;  
}  
  
const numbers = [1, 2, 3];  
console.log(sum(...numbers));
```

Sintaxis Spread - concatenando dos arreglos

```
var valores = [1,2,3];  
  
var numeros = [4,5,6];  
  
var totales = [...valores, ...numeros];  
  
{}  
  
console.log(valores);  
  
console.log(...valores, 4);  
  
console.log(totales);
```

Arreglos

Como crear arreglos, tres formas distintas:

```
let frutas = ["Manzana", "Banana"]
```

```
let frutas = new Array("Manzana", "Banana")
```

```
let frutas = new Array(10); //En este caso el  
arreglo no contiene items.
```

Arreglos

Vamos a ver varios métodos necesarios para el manejo de arreglos, muchos de ellos se utilizan pasando una función para el procesamiento de cada uno de los elementos del arreglo.

Arreglos

Métodos para el manejo de arreglos:

foreach

map

filter

includes

concat

find

findindex

indexOf

join

pop

push

reduce

shift

slice

splice

sort

Arreglos - `foreach`

El método `forEach()` ejecuta la función indicada una vez por cada elemento del array.

```
const array1 = ['a', 'b', 'c'];

array1.forEach(
  element => console.log(element)
);
```

Arreglos - map

El método `map()` crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos.



```
var numeros = [1, 5, 10, 15];

var dobles = numeros.map(function(x)
{
    return x * 2;
});

// dobles      [2, 10, 20, 30]
// numeros [1, 5, 10, 15]
```

Arreglos - filter

El método **filter()** crea un nuevo array con todos los elementos que cumplan la condición implementada por la función dada.



```
const words = ['spray', 'limit',
'elite', 'exuberant', 'destruction',
'present'];

const result = words.filter(word =>
word.length > 6);

// Array ["exuberant",
"destruction", "present"]
```

Arreglos - includes

El método **includes()** determina si una matriz incluye un determinado elemento, devuelve true o false según corresponda.



```
const pets = ['cat', 'dog', 'bat'];

console.log(pets.includes('cat'));
//true

console.log(pets.includes('at'));
//false
```

Arreglos - concat

El método **concat()** se usa para unir dos o más arrays. Este método no cambia los arrays existentes, sino que devuelve un nuevo array.



```
const array1 = ['a', 'b', 'c'];
const array2 = ['d', 'e', 'f'];
const array3 = array1.concat(array2);

console.log(array3);
//Array ["a", "b", "c", "d", "e", "f"]
```

Arreglos - find

El método **find()** devuelve el valor del primer elemento del array que cumple la función de prueba proporcionada.

```
const array1 = [5, 12, 8, 130, 44];  
  
const found = array1.find(  
    element => element > 10  
);  
  
console.log(found); //12
```

Arreglos - `findIndex`

El método `findIndex()` devuelve el índice del primer elemento de un array que cumpla con la función de prueba proporcionada. En caso contrario devuelve -1.

```
const array1 = [5, 12, 8, 130, 44];  
  
const isLargeNumber =  
    (element) => element > 13;  
  
console.log(  
    array1.findIndex(isLargeNumber)  
)  
// 3
```

Arreglos - indexOf

El método **indexOf()** retorna el primer índice en el que se puede encontrar un elemento dado en el array, ó retorna -1 si el elemento no esta presente.



```
var array = [2, 9, 9];  
  
array.indexOf(2); // 0  
  
array.indexOf(7); // -1  
  
array.indexOf(9, 2); // 2  
  
array.indexOf(2, -1); // -1  
  
array.indexOf(2, -3); // 0
```

Arreglos - join

El método **join()** une todos los elementos de una matriz (o un objeto similar a una matriz) en una cadena y devuelve esta cadena.



```
const elements = ['Fire', 'Air',  
'Water'];  
  
console.log(elements.join());  
  
// "Fire,Air,Water"  
  
console.log(elements.join(' '));  
  
// "FireAirWater"  
  
console.log(elements.join('-'));  
  
// "Fire-Air-Water"
```

Arreglos - pop

El método **pop()** elimina el último elemento de un array y lo devuelve. Este método cambia la longitud del array.

```
{  
const plants = ['broccoli',  
'cauliflower', 'cabbage', 'kale',  
'tomato'];  
  
console.log(plants.pop());  
// "tomato"  
  
console.log(plants);  
// Array ["broccoli", "cauliflower",  
"cabbage", "kale"]}
```

Arreglos - push

El método **push()** añade uno o más elementos al final de un array y devuelve la nueva longitud del array.



```
const animals = ['pigs', 'goats',  
'sheep'];  
  
const count = animals.push('cows');  
  
console.log(count); // 4  
  
console.log(animals);  
  
// Array ["pigs", "goats", "sheep",  
"cows"]
```

Arreglos - reduce

El método **reduce()** ejecuta una función reductora sobre cada elemento de un array, devolviendo como resultado un único valor.



```
const array1 = [1, 2, 3, 4];  
  
const reducer =  
  
  (accumulator, currentValue) =>  
  accumulator + currentValue;  
  
// 1 + 2 + 3 + 4  
  
console.log(array1.reduce(reducer));  
  
// 10
```

Arreglos - shift

El método **shift()** elimina el primer elemento del array y lo retorna. Este método modifica la longitud del array.



```
var peces = ['ángel', 'payaso',  
'mandarín', 'cirujano'];  
  
var eliminado = peces.shift();  
  
console.log(peces);  
  
// payaso,mandarín,cirujano
```

Arreglos - slice

El método **slice()** devuelve una copia de una parte del array dentro de un nuevo array empezando por inicio hasta fin (fin no incluido). El array original no se modificará.



```
var nombres = ['Rita', 'Pedro',  
'Miguel', 'Ana', 'Vanesa'];
```

```
var masculinos = nombres.slice(1, 3);  
  
// ['Pedro', 'Miguel']
```

Arreglos - splice

El método **splice()** cambia el contenido de un array eliminando elementos existentes y/o agregando nuevos elementos.



```
const months = ['Jan', 'March',  
'April', 'June'];  
  
months.splice(1, 0, 'Feb');  
  
// Array ["Jan", "Feb", "March",  
"April", "June"]  
  
months.splice(4, 1, 'May');  
  
// Array ["Jan", "Feb", "March",  
"April", "May"]
```

Arreglos - sort

El método **sort()** ordena los elementos de un arreglo (array) localmente y devuelve el arreglo ordenado.

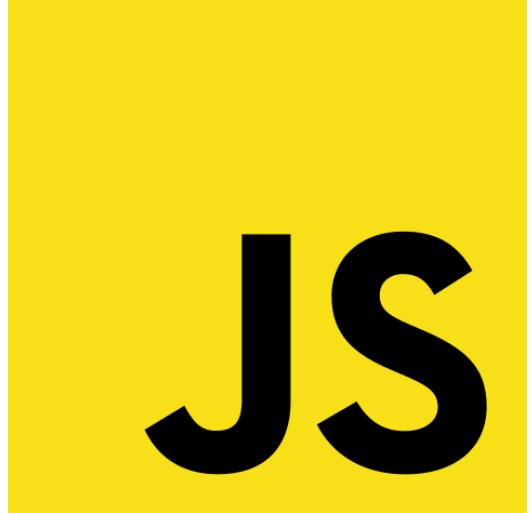


```
var valores =  
[1,3,45,1,23,25,1,8,25,1,9,10,25,23];  
  
valores.sort();  
  
//1,1,1,1,10,23,23,25,25,25,3,45,8,9  
  
valores.sort( (a,b)=> a - b);  
  
//1,1,1,1,3,8,9,10,23,23,25,25,25,45
```

Próxima Clase

Definición de Clases y Objetos en

JavaScript



JS

Programacion Orientada a Objetos 2

Ing. Jose Rusca
Ing. Federico Stulich
Ing. Francisco De Grandis

Programacion Orientada a Prototipos



Clase de hoy

- Prototipos
- Creación de objetos sin definición de clases
(Literales, Constructores y Clonación)
- El objeto Object
- Propiedades de Objetos (Propiedades enumerables, Constructor, Prototipo y mutación)

Prototipos (Definicion)

Un objeto **prototipo** es un objeto que actúa como un objeto plantilla que hereda métodos y propiedades.

Un objeto prototipo del **objeto** puede tener a su vez otro objeto prototipo, el cual hereda métodos y propiedades, y así sucesivamente.

Esto es conocido con frecuencia como la **cadena de prototipos**, y explica por qué objetos diferentes pueden tener disponibles propiedades y métodos definidos en otros objetos.

Clases vs prototipos

Categoría	Basado en clases (Java)	Basado en prototipos (JavaScript)
Clase vs. Instancia	La clase y su instancia son entidades distintas.	Todos los objetos pueden heredar de otro objeto.
Definición	Define una clase con una definición <i>class</i> ; se instancia una clase con métodos constructores.	Define y crea un conjunto de objetos con funciones constructoras.
Creación de objeto	Se crea un objeto con el operador <i>new</i> .	Se crea un objeto con el operador <i>new</i> .
Construcción de jerarquía de objetos	Se construye una jerarquía de objetos utilizando definiciones de clases para definir subclases de clases existentes.	Se construye una jerarquía de objetos mediante la asignación de un objeto como el prototipo asociado a una función constructora.
Herencia	Se heredan propiedades siguiendo la cadena de clases.	Se heredan propiedades siguiendo la cadena de prototipos.
Extensión de propiedades	La definición de una clase especifica <i>todas</i> las propiedades de todas las instancias de esa clase. No se puede añadir propiedades dinámicamente en tiempo de ejecución.	El conjunto <i>inicial</i> de propiedades lo determina la función constructor o el prototipo. Se pueden añadir y quitar propiedades dinámicamente a objetos específicos o a un conjunto de objetos.

Creación de objetos sin clases

1. Literales
2. Constructores (new Object)
3. Clonación (Object.Create)

Objetos Literales

```
var javier = {  
    nombre: "Javier",  
    Apellido: "Baez"  
    fechaNacimiento: new Date(1998, 6,  
15)  
}
```

Objetos Literales

```
var javier = {  
    nombre: "Javier",  
    Apellido: "Baez"  
    fechaNacimiento: new Date(1998, 6,  
    15)  
}
```

Cada propiedad es un par clave/valor

Objetos Literales - Métodos

Repasemos.....

En JavaScript, los métodos son las propiedades de un objeto que contienen una función.

O tambien podemos decir que, los métodos son funciones almacenadas como una propiedad.

Objetos Literales - Métodos

```
var javier = {  
    nombre: "Javier",  
    Apellido: "Baez",  
    fechaNacimiento: new Date(1998, 6, 15),  
    nombreCompleto: function() {  
        return this.apellido + ", " + this.nombre + " (" +  
            this.fechaNacimiento.toLocaleDateString() +")"  
    }  
}
```

Objetos Literales - Métodos

```
var javier = {  
    nombre: "Javier",  
    Apellido: "Baez",  
    fechaNacimiento: new Date(...)
```

}

```
    nombreCompleto: function() {  
  
        return this.apellido + ", " + this.nombre + " (" +  
        this.fechaNacimiento.toLocaleDateString() +")"  
    }
```

}

Por medio de la propiedad **nombreCompleto** accedemos a la función que construye un string y retorna el valor

Objetos Literales - Métodos

```
console.log(javier);  
  
{  
  
  nombre: 'Javier',  
  
  apellido: 'Baez',  
  
  fechaNacimiento: 1998-07-15T03:00:00.000Z,  
  
  nombreCompleto: [Function: nombreCompleto]  
  
}  
  
console.log(javier.nombreCompleto());  
  
Baez, Javier (7/15/1998)
```

Accesos - Setters y Getters

Al igual que definimos en las clases, aca tambien podemos utilizar **setters** y **getters**.

Pero..... debemos recordar que todas las propiedades (y métodos) en JavaScript son públicos.

¿Y las propiedades con hash #.....



Accesos - Setters y Getters

Al trabajar sin clases no tenemos forma de indicar la propiedad como privada, por lo tanto vamos a ver más adelante cómo hacer “privada” una variable de nuestro objeto por medio de clausuras.



Accesos - Setters y Getters

```
.....  
  
get nombreCompleto() {  
  
    return this.apellido + ", " + this.nombre + "(" + this.fechaNacimiento.toLocaleDateString() +")"  
  
},  
  
set nombreCompleto(value) {  
  
    var arreglo = value.split(', ');  
  
    this.apellido = arreglo[0];  
  
    this.nombre = arreglo[1];  
  
},  
  
.....
```

```
console.log(javier.nombreCompleto);  
// Baez, Javier (7/15/1998)  
  
javier.nombreCompleto = "Carlos, Baez";  
  
console.log(javier.nombreCompleto)  
//Carlos, Sanchez (7/15/1998)
```

Constructores

Un **constructor** en JavaScript es una función que se invoca con el operador **new** y nos va servir de plantilla para la creación de objetos.

Trabajando con constructores debemos tener mucho cuidado de no olvidar incluir el operador **new**.

Constructores

```
function Persona(nombre, apellido, fechaNacimiento) {  
    this.nombre = nombre;  
    this.apellido = apellido;  
    this.fechaNacimiento = fechaNacimiento;  
    this.nombreCompleto = function() {  
        return this.apellido + ", " + this.nombre + " (" + this.fechaNacimiento  
            .toLocaleDateString() +")"  
    }  
}
```

Constructores

```
var carlos = new Persona("Carlos", "Alvarez", new Date());
```

```
console.log(carlos.nombreCompleto());
```

{}

Alvarez, Carlos (9/5/2021)

No debemos olvidar el operador **new** si no las propiedades se van a crear en el objeto global.

Constructores

A primera vista los constructores parecieran ser una alteración del concepto de una clase.

Junto al problema que puede derivar en olvidarnos el operador new, pareciera esto no ser el mejor camino posible si vamos a enfocarnos en una programación orientada a prototipos.

Constructores y prototipos

JavaScript está en conflicto con su naturaleza prototípica. Su mecanismo es oscurecido por algún asunto sintáctico complicado que parece vagamente clásico.

En lugar de que los objetos hereden directamente de otros objetos, se inserta un nivel innecesario de indirección de manera que los objetos sean producidos por funciones de constructor.

Prototipos

En un patrón puramente **prototípico** vamos a prescindir de las clases. Por lo tanto vamos a concentrarnos en objetos. La herencia prototípica es conceptualmente más simple que la **herencia clásica**. El **nuevo objeto** puede heredar las propiedades de un **objeto antiguo**.

Esto a primera vista puede parecer extraño, pero es realmente fácil de entender.

Clonacion

Los objetos los vamos a crear de forma literal o bien creando un objeto vacío y luego agregando sus propiedades y métodos.

Utilizando el método `Object.create(object)`.



Vamos a evitar utilizar el operador `new`.

Clonacion

```
var javier = ...
```

```
var adam = Object.create(javier);
```

```
adam.nombre = "Adam";
```

```
adam.apellido = "Morgan";
```

```
console.log(adam.nombreCompleto());
```

```
//Morgan, Adam (7/15/1998)
```

{}

Vamos a utilizar el objeto literal que definimos al comienzo.

Clonacion

```
var javier = ...  
  
var adam = Object.create(javier);  
  
adam.nombre = "Adam";  
  
adam.apellido = "Morgan";  
  
console.log(adam.nombreCompleto());  
  
//Morgan, Adam (7/15/1998)
```

Si no modificamos la fecha de nacimiento se mantiene la fecha de javier.

El objeto Object

Los objetos en JavaScript son mutables por lo tanto vamos a poder modificarlos en tiempo de ejecución.

Esta mutación/extensión la podemos hacer de distintas formas.

El objeto Object - métodos

assign	getOwnPropertyDescriptor	isFrozen
create	getOwnPropertyDescriptors	isSealed
defineProperty	getOwnPropertyNames	keys
defineProperties	getOwnPropertySymbols	preventExtensions
entries	getPrototypeOf	seal
freeze	is	setPrototypeOf
fromEntries	isExtensible	values

Objeto - Propiedades enumerables

Desde EmacScript 5 se pueden enumerar las propiedades de un objeto de forma nativa mediante alguna de las siguientes formas.

1. Bucle for...in
2. Object.keys(objeto*)
3. Object.getOwnPropertyNames(objeto*)

*sobre el objeto que se quiere obtener las propiedades

Objeto - Propiedades enumerables

```
for(const clave in javier) {  
    console.log(javier[clave]);  
}
```

{}

Javier

Baez

1998-07-15T03:00:00.000Z

[Function: nombreCompleto]

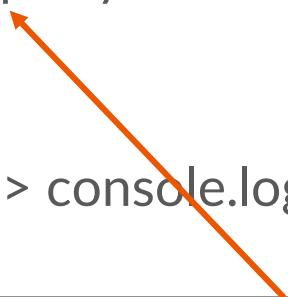
Objeto - Propiedades enumerables

```
console.log(Object.keys(javier));
```

```
console.log(Object.getOwnPropertyNames(javier));
```

{}

```
Object.keys(javier).map( item => console.log(javier[item]));
```



Ambos casos nos devuelven el siguiente arreglo

```
[ 'nombre', 'apellido', 'fechaNacimiento', 'nombreCompleto'  
 ]
```

Objeto - Propiedades enumerables

```
console.log(Object.keys(javier));
```

```
console.log(Object.getOwnPropertyNames(javier));
```

{}

```
Object.keys(javier).map( item => console.log(javier[item]));
```



En ambos casos, podemos mostrar los valores utilizando el método map.

Propiedad constructor

Un objeto que fue construido por medio de un **constructor**, posee una propiedad constructor que apunta a la función constructora.

`daniela.constructor.toString()`

Propiedad prototype

prototype permite identificar el **objeto** utilizado como prototipo. Se puede obtener y asignar por medio de dos métodos que veremos mas adelante.

No debemos confundir con la propiedad prototype de las **funciones** (constructoras) que especifica el prototipo de los objetos que construyó.

Objeto - Mutación

Los objetos en JavaScript son mutables por lo tanto vamos a poder modificarlos en tiempo de ejecución.

Esta mutación/extensión la podemos hacer de distintas formas.

Objeto - Mutación

Agregando propiedades/métodos

1. con el operador punto (.)
2. aplicándolo a la propiedad prototipo
3. Utilizando Object.defineProperty(....)

Objeto - Mutación

Object.defineProperty(obj, prop, descriptor)

obj: el objeto al cual se le va a aplicar la propiedad.

prop: nombre de la propiedad.

descriptor: objeto literal con la configuración de la propiedad.

Objeto - Mutación

Descriptor:

- configurable
- enumerable
- value
- writable
- set
- get

Ejercicio

- Empleado
- Piezas de Ajedrez

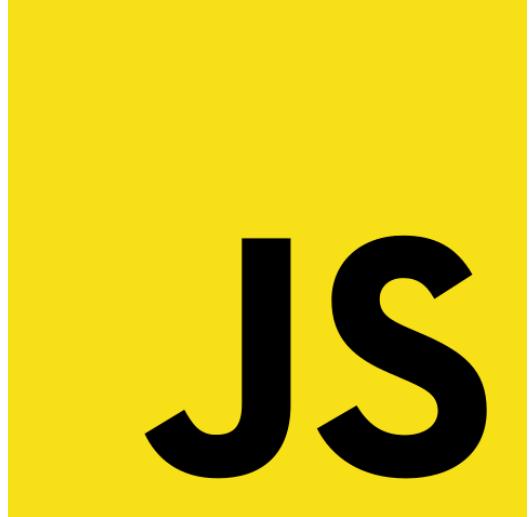
Referencia

- https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/Object_prototypes
- Crockford D. (2008). JavaScript: The Good Parts
- <https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/Inheritance>
- https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

Próxima Clase

Prototipos - Herencia y Accesos

JavaScript



JS

Programacion Orientada a Objetos 2

Ing. Jose Rusca
Ing. Federico Stulich
Ing. Francisco De Grandis

Programacion Orientada a Prototipos



Clase de hoy

- Herencia y Delegación
- Cadena de prototipos
- Miembros privados
- Miembros estaticos (públicos y privados)
- Evitando extensiones

Herencia y Delegación

A simple vista las clases y los prototipos parecen ser similares, pero distan bastante de serlo. Como en ambos casos estamos tratando de modelar objetos al no tener claro los conceptos podemos fácilmente confundirnos. Recordemos lo último que vimos la clase pasada.....

Herencia y Delegación

Nos habíamos quedado con que la clase constructora generaba un objeto.... (la función constructora tenía una propiedad **prototype**).

Para ello vamos a analizar bien las clases y los prototipos y su diferencia.

Herencia y Delegación

El enfoque tradicional para compartir conocimientos entre objetos es conocido como **Herencia**. En este caso estamos obligados a diseñar un objeto clase que nos va a permitir crear/instanciar los objetos en donde esta toda la estructura del objeto “padre” va a ser copiada al objeto “hijo”.

Herencia y Delegación

Si en cambio utilizamos prototipos para compartir conocimientos entre objetos, lo vamos a denominar **Delegación**. Con este enfoque vamos a poder ir modelando lo particular para luego poder ir generalizando. Si veo que todos los objetos comparten características o comportamiento puedo pasar esto al prototipo.

Herencia y Delegación

En los prototipos vamos a encontrarnos con una forma dinámica (method look-up) en donde vamos a recorrer los objetos hacia “arriba” buscando que algún prototipo pueda resolver el problema, en JavaScript esta conexión se conoce como **cadena de prototipos**.

Cadena de Prototipos

En JavaScript cada objeto tiene una propiedad que apunta a su prototipo (no confundir con prototype de la clase constructora). Los objetos (que no son funciones) obtienen su prototipo por medio del objeto Object.

`Object.getPrototypeOf(obj);`

Cadena de Prototipos

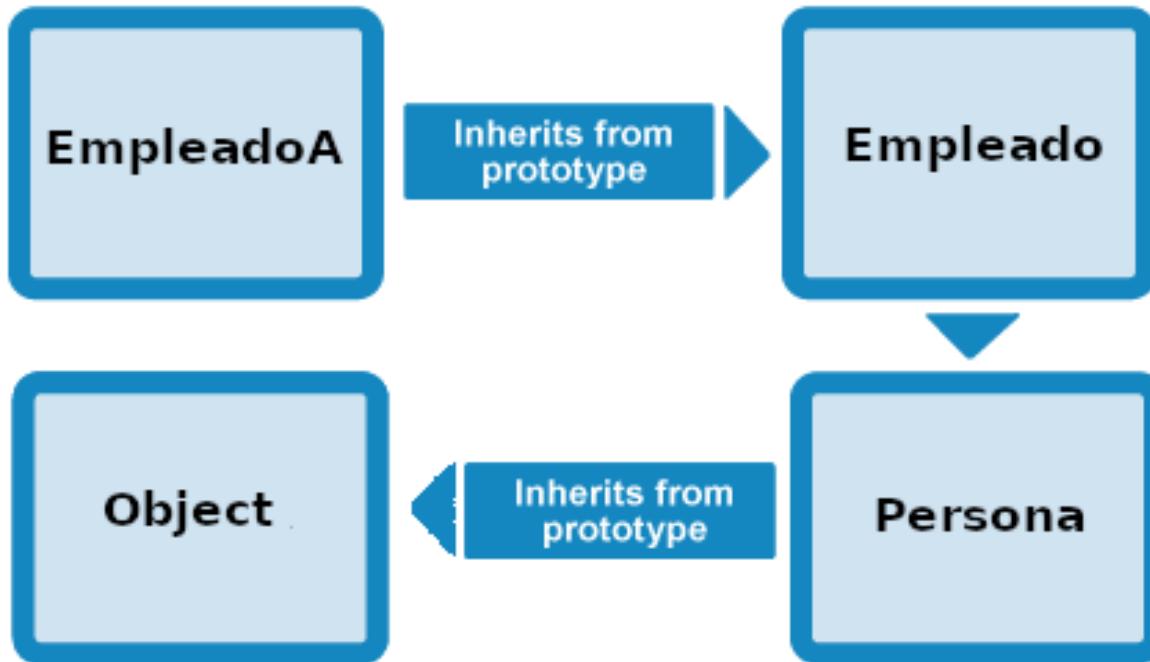
Cuando clonamos un objeto, no tenemos problemas porque el prototipo del objeto creado va a apuntar siempre a su prototipo.

El inconveniente surge cuando utilizamos funciones constructoras y tenemos herencia.

Cadena de Prototipos



Cadena de Prototipos



Cadena de Prototipos

Vamos a ver un ejemplo donde podamos visualizar toda la cadena de prototipos.



Ejemplo

```
function prototypeChain(obj) {
    console.log(obj.constructor.name);
    console.log("  >>> " + JSON.stringify(obj));
    var proto = Object.getPrototypeOf(obj);
    while(proto!=null){
        if(Object.getPrototypeOf(proto)==null) {
            console.log("  ↳ " + proto.constructor.name);
            console.log("    >>> " + JSON.stringify(proto));
        }
        else {
            console.log("  ↳ " + proto.constructor.name);
            console.log("    >>> " + JSON.stringify(proto));
        }

        proto = Object.getPrototypeOf(proto);
    }
}
```

Cadena de Prototipos

```
//Relacionamos los prototipos de Persona y Empleado  
Empleado.prototype = Object.create(Persona.prototype);  
  
{ //Asignamos el constructor al prototipo  
    Empleado.prototype.constructor = Empleado;
```

Miembros privados

Pensando en prototipos, JavaScript no necesita de ninguna instrucción en particular para definir miembros privados, lo resuelve fácilmente por medio de una clausura.

Miembros privados

```
function Persona(nombre, apellido, fechaNacimiento) {  
    var fecha = fechaNacimiento;  
  
    this.nombre = nombre;  
  
    this.apellido = apellido;  
  
    this.calcularEdad = function() {  
  
        return parseInt((new Date() - fecha) / 31536000000);  
  
    }  
....
```

Solo la función puede
acceder a la variable.

{}

Miembros privados

```
function Persona(nombre, apellido, fechaNacimiento) {  
    var fecha = fechaNacimiento;  
  
    this.nombre = nombre;  
    this.apellido = apellido;  
  
    var calcularEdad = function() {  
        return parseInt((new Date() - fecha) / 31536000000);  
    }  
}
```

Solo la función puede acceder a la variable.

....

Miembros estáticos

Al igual que pasa con los miembros privados, cuando pensamos en prototipos no vamos a disponer de palabras claves. Para ello vamos a trabajar sobre la función constructora.

Miembros estáticos públicos

```
Persona.saluteYou = function() {  
    return "Hi I'm a Persona!";  
}
```

{}

```
Persona.prototype.saluteYou = Persona.saluteYou;
```

Definimos el método a nivel
de la función constructora

Miembros estáticos públicos

```
Persona.saluteYou = function() {  
    return "Hi I'm a Persona!";  
}
```

{}

```
Persona.prototype.saluteYou = Persona.saluteYou;
```

Le damos acceso para que las instancias puedan ver el método definido en la función constructora.

Miembros estáticos privados

Vamos a ver un ejemplo donde tengamos un contador de objetos persona y que se acceda desde cada objeto.

Miembros estáticos privados

```
var Persona = (function() {  
    var contador = 0;  
  
    newPersona = function(nombre, apellido, fechaNacimiento) {  
  
        this.nombre = nombre;  
  
        this.apellido = apellido;  
  
        this.fechaNacimiento = fechaNacimiento;  
  
        contador += 1;  
  
    }  
  
    //Continua
```

Variable
contadora

{}

Miembros estáticos privados

```
var Persona = (function() {  
    var contador = 0;  
  
    newPersona = function(nombre, apellido, fechaNacimiento) {  
        this.nombre = nombre;  
  
        this.apellido = apellido;  
  
        this.fechaNacimiento = fechaNacimiento;  
  
        contador += 1;  
    };  
  
    //Continua
```

{}

Función
constructora

Miembros estáticos privados

//Continua

```
newPersona.prototype.nombreCompleto = function() {  
    return this.apellido + ", " +  
        this.nombre + " (" +  
        this.fechaNacimiento.toLocaleDateString() +")";  
}  
  
newPersona.prototype.cantidad = function() {  
    return contador;  
}  
  
return newPersona;  
})();
```

{}

Metodos

Miembros estáticos privados

```
//Continua

newPersona.prototype.nombreCompleto = function() {
    return this.apellido + ", " +
        this.nombre + " (" +
        this.fechaNacimiento.toLocaleDateString() +")";
}

newPersona.prototype.cantidad = function() {
    return contador;
}

return newPersona;
})();
```

Retornamos la
función
constructora

Miembros estáticos privados

//Continua

```
newPersona.prototype.nombreCompleto = function() {  
    return this.apellido + ", " +  
        this.nombre + " (" +  
        this.fechaNacimiento.toLocaleDateString() +")";  
}  
  
newPersona.prototype.cantidad = function() {  
    return contador;  
}  
  
return newPersona;  
})();
```

No debemos olvidar
llamar a la función
incluyendo los
paréntesis.

Evitando Extensiones

`Object.preventExtensions(obj)`

Previene que nuevas propiedades sean agregadas a un objeto e impide que se reasigne el prototipo del objeto.

`Object.isExtensible(obj)`

Nos permite saber si el objeto es extensible

Evitando Extensiones

`Object.seal(obj)`

Previene que nuevas propiedades puedan ser añadidas al objeto y marcando los existentes como no configurables.

`Object.isSealed(obj);`

Nos permite saber si un objeto esta sellado.

Evitando Extensiones

Object.freeze(obj)

Previene que nuevas propiedades puedan ser añadidas al objeto y impidiendo que las existentes se puedan sobreescribir o configurar.

Object.isFrozen(obj);

Nos permite saber si un objeto esta congelado.

Igualdad e identidad

Dos objetos son **iguales** cuando los valores de sus propiedades son iguales.

Un objeto es **idéntico** sólo a sí mismo, por lo tanto si tenemos dos referencias(con distinto nombre) al mismo objeto, la comparación de esas referencias nos debería indicar que el objeto es el mismo.

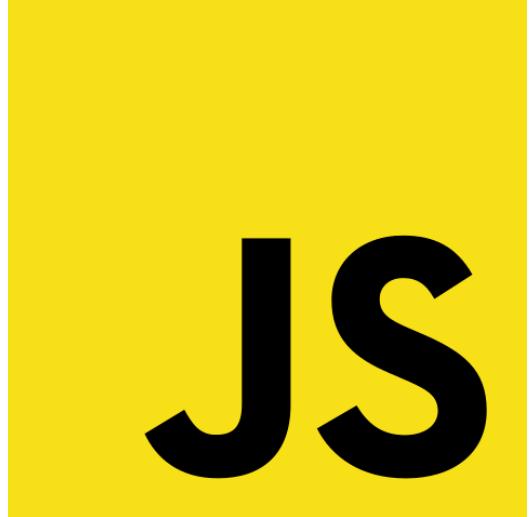
Referencia

- https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/Object_prototypes
- Crockford D. (2008). JavaScript: The Good Parts
- <https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/Inheritance>
- https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Details_of_the_Object_Model
- Lieberman, H. (1986) Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems
- Stefanou, S. (2010) JavaScript Patterns

Próxima Clase

Prototipos - Module & Creational Patterns

JavaScript



JS

Programacion Orientada a Objetos 2

Ing. Jose Rusca
Ing. Federico Stulich
Ing. Francisco De Grandis

Module & Creational Patterns



Clase de hoy

- Creational Pattern (Creación de Objetos)
- IIFE
- Modulos Literales
- Module Pattern
- Revealing Module Pattern

Creational Pattern

Los patrones de diseño creacional se enfocan en manejar los mecanismos de creación de objetos donde los objetos se crean de una manera adecuada para la situación en la que está trabajando.

El enfoque básico para la creación de objetos podría conducir a una mayor complejidad en un proyecto, mientras que estos patrones tienen como objetivo resolver este problema controlando el proceso de creación.

Creación de objetos

Repasemos dos puntos sobre creación de objetos.

Esto nos va a permitir entender mejor el Module Pattern.

- Funciones que devuelven objetos.
- Constructores (con el problema de olvidar el new).

Funciones (que devuelven objetos)

{}

```
function Persona(nombre, apellido, fechaNacimiento) {  
    var persona = {};  
  
    persona.nombre = nombre;  
    persona.apellido = apellido;  
    .... //continua la funcion  
    return persona;  
}
```

Constructores (con el problema de olvidar el new)

```
function Persona(nombre, apellido, fechaNacimiento) {  
  
    if(!(this instanceof Persona))  
        return new Persona(nombre, apellido,  
fechaNacimiento);  
  
    this.nombre = nombre;  
    this.apellido = apellido;  
.... //continua la funcion
```

IIFE (Immediately invoked function expression).

Un IIFE es una función sin nombre que se invoca inmediatamente después de haber sido definida.

Debido a que solo se puede acceder a las variables y funciones definidas explícitamente dentro de dicho contexto, la invocación de funciones proporciona un medio fácil para lograr la privacidad.

IIFE (Immediately invoked function expression).

Esto lo utilizamos la clase pasada para devolver el constructor cuando se necesitaba disponer de una propiedad estática.

Pero si en vez de devolver el constructor devolvemos el objeto, vamos a obtener comportamiento similar a un patrón singleton (en singleton tenemos delay initialization).

IIFE

```
var john = function() {  
  
    var obj = {};  
  
    var name = "John"  
  
    var age = 10;  
  
    obj.greet = function() {  
  
        console.log(name + " has " + age + " years.");  
  
    }  
  
    return obj;  
  
}  
  
var j = john();  
j.greet();
```

{}

IIFE

```
var john = (function() {  
  
    var obj = {};  
  
    var name = "John"  
  
    var age = 10;  
  
    obj.greet = function() {  
  
        console.log(name + " has " + age + " years.");  
  
    }  
  
    return obj;  
})();  
  
john.greet();
```

{}

IIFE

No confundir Singleton con Clases estáticas.

La clase estática es un conjunto de métodos estáticos.

El Singleton es una única instancia de un objeto.

Module & Creational patterns

La semana pasada vimos una forma de crear valores estáticos y aunque todavía no lo sabíamos, esto está relacionado a lo que denominamos patrones creacionales y módulos.

Module & Creational patterns

El patrón de módulo se usa ampliamente porque proporciona estructura y ayuda a organizar el código a medida que crece. JavaScript no tiene una sintaxis especial para paquetes, pero el patrón de módulo proporciona las herramientas para crear piezas de código independientes y desacopladas, que pueden tratarse como cajas negras de funcionalidad y agregarse, reemplazarse o eliminarse según los requisitos del software que se está escribiendo.

Módulos Literales

Por medio de objetos literales podemos crear/organizar la funcionalidad necesaria para nuestra aplicación. Una de las desventajas es que toda la funcionalidad va a permanecer pública. Si necesito ocultar algo no voy a poder hacerlo.

Utilizado tambien para namespacing pattern.

Módulos Literales

```
var moduloMatematico = {  
    calcularResta: function(a,b) {  
        return a - b;  
    }  
    calcularSuma: function(a,b) {  
        return a + b;  
    }  
}
```

Module Pattern

El patrón de módulo se usa para emular aún más el concepto de clases de tal manera que podamos incluir tanto métodos públicos / privados como variables dentro de un solo objeto, protegiendo así partes particulares del alcance global.

Module Pattern

```
var counter = (function () {  
  
    var counter = 0;  
  
    return {  
  
        incrementCounter: function () {  
  
            return counter++;  
  
        },  
  
        resetCounter: function () {  
  
            console.log('counter value prior to reset:' + counter);  
  
            counter = 0;  
  
        } };  
  
})();
```

{}

Revealing Module Pattern

El patrón del módulo revelado surgió cuando Heilmann (Christian) estaba frustrado con el hecho de que si tenía que repetir el nombre del objeto principal cuando quería llamar a un método público desde otro o acceder a variables públicas. Tampoco le gustó el requisito del patrón Module de tener que cambiar a la notación literal de objeto para las cosas que deseaba hacer públicas.

Revealing Module Pattern

```
var counter = (function () {  
  
    var counter = 0;  
  
    function iCounter() { return counter++; }  
  
    function reset() {  
  
        console.log('counter value prior to reset:' + counter);  
  
        counter = 0;  
    }  
  
    return {  
  
        incrementCounter: iCounter,  
  
        resetCounter: reset  
  
    };  
})();
```

{}

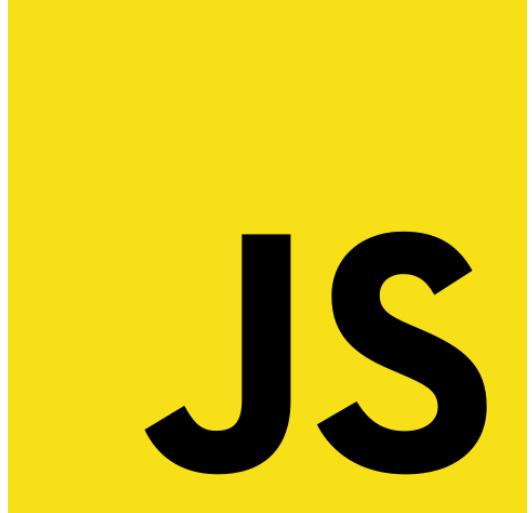
Referencia

- Crockford D. (2008). JavaScript: The Good Parts
- Lieberman, H. (1986) Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems
- Stefanou, S. (2010) JavaScript Patterns
- Osmani, A. (2012) JavaScript Desing Patterns

Próxima Clase

Test Driven Development

JavaScript

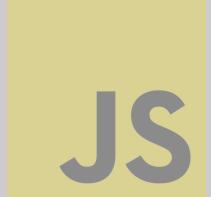


JS

Programacion Orientada a Objetos 2

Ing. Jose Rusca
Ing. Federico Stulich

Design Patterns



JS

Clase de hoy

- Definición
- Patrones creacionales: Singleton, Factory.
- Estructurales: Composite, Decorator.
- De Comportamiento: State, Strategy, Observer.

What Is a Design Pattern?

Un patrón describe un problema que se repite constantemente en nuestro entorno. Describe la esencia de la solución a dicho problema, de tal manera que se pueda aplicar esa solución innumerables veces, sin tener que repetirla exactamente igual cada vez.

Creational Patterns

Singleton

Factory



JS

Creational Patterns - Singleton

Asegura que una clase tenga una única instancia y proporciona un punto de acceso global a la misma.

Creational Patterns - Singleton - Aplicación

- Debe haber solo una instancia de una clase, y esta debe ser accesible para los clientes desde un punto de acceso conocido.
- Cuando la única instancia debe ser extensible mediante subclases, y los clientes deben poder usar una instancia extendida sin modificar su código.

Creational Patterns - Singleton - Características

- Acceso controlado a una única instancia.
- Espacio de nombres reducido.
- Permite la optimización de las operaciones y la representación.
- Permite un número variable de instancias.
- Más flexible que las operaciones de clase.

Creational Patterns - Singleton - conClases

```
class Singleton {  
    constructor() {  
        if (Singleton.instance) {  
            return Singleton.instance;  
        }  
        this.value = Math.random();  
        Singleton.instance = this;  
    }  
    getValue() {  
        return this.value;  
    }  
}  
  
// Ejemplo de uso:  
const singletonA = new Singleton();  
const singletonB = new Singleton();  
  
console.log(singletonA === singletonB); // true  
console.log(singletonA.getValue()); // Por ejemplo: 0.123456  
console.log(singletonB.getValue()); // El mismo valor que singletonA
```



Creational Patterns - Singleton - conPrototipos

```
const Singleton = (function() {  
    let instance;  
    return function() {  
        if (!instance) instance = {};  
        return instance;  
    };  
})();
```

{}

```
const a = Singleton();  
const b = Singleton();  
console.log(a === b); // true
```

Creational Patterns - Factory

Define una interfaz para la creación de objetos, pero deja que las subclases decidan qué clase se debe instanciar. El patrón de diseño Factory Method permite que una clase delegue la creación de instancias a las subclases.

Creational Patterns - Factory - Aplicación

- Una clase no puede anticipar la clase de objetos que debe crear.
- Una clase quiere que sus subclases especifiquen los objetos que crea.
- Las clases delegan la responsabilidad a una de varias subclases auxiliares, y se desea localizar el conocimiento de qué subclase auxiliar es la delegada.

Creational Patterns - Factory - Características

- Permite la creación de métodos específicos para las subclases.
Crear objetos dentro de una clase mediante un método que siempre resulta más flexible que crear un objeto directamente.
- Facilita la integración de jerarquías de clases paralelas.
En los ejemplos vistos hasta ahora, el *factory method* sólo es invocado por la clase creadora. Sin embargo, no es necesario que sea así; los clientes también pueden beneficiarse de los *factory method*, especialmente en el caso de jerarquías de clases paralelas.

Creational Patterns - Factory - Ejemplos Simple

```
// Constructores de los diferentes tipos
function Admin(name) {
    this.role = 'admin';
    this.name = name;
}
Admin.prototype.sayHello = function() {
    return `Hola, soy ${this.name} y soy administrador.`;
};

function Guest(name) {
    this.role = 'guest';
    this.name = name;
}
Guest.prototype.sayHello = function() {
    return `Hola, soy ${this.name} y soy invitado.`;
};
```



Creational Patterns - Factory

```
function Member(name) {  
    this.role = 'member';  
    this.name = name;  
}  
Member.prototype.sayHello = function() {  
    return `Hola, soy ${this.name} y soy miembro.`;  
};  
// La función Factory  
function UserFactory(role, name) {  
    switch (role) {  
        case 'admin':  
            return new Admin(name);  
        case 'guest':  
            return new Guest(name);  
        case 'member':  
            return new Member(name);  
        default:  
            throw new Error('Rol no válido');  
    } }  
}
```

Creational Patterns - Factory

```
// Uso del patrón Factory

var user1 = UserFactory('admin', 'Ana');

var user2 = UserFactory('guest', 'Luis');

var user3 = UserFactory('member', 'Carlos');

}

console.log(user1.sayHello()); // Hola, soy Ana y soy administrador.

console.log(user2.sayHello()); // Hola, soy Luis y soy invitado.

console.log(user3.sayHello()); // Hola, soy Carlos y soy miembro.
```

Estructural Patterns

Composite
Decorator

JS

Estructural Patterns - Composite

Organiza los objetos en estructuras de tipo árbol para representar jerarquías y elementos.

Permite que los clientes traten los objetos individuales y las agrupaciones de objetos de forma uniforme.

Estructural Patterns - Composite - Aplicación

- Desea representar jerarquías de objetos que combinan distintos tipos de objetos.
- Desea que los clientes puedan ignorar la diferencia entre composiciones de objetos y objetos individuales. Los clientes tratarán todos los objetos de la estructura compuesta de forma uniforme.

Estructural Patterns - Composite - Características

- Define jerarquías de clases compuestas por objetos primitivos y objetos compuestos.
Los objetos primitivos pueden combinarse para formar objetos más complejos, los cuales, pueden combinarse entre sí, y así sucesivamente. En cualquier lugar donde el código cliente espere un objeto primitivo, también puede recibir un objeto compuesto.

Estructural Patterns - Composite - Características

- Simplifica el código cliente.

Los clientes pueden tratar las estructuras compuestas y los objetos individuales de forma uniforme. Normalmente, el cliente no necesita saber (ni debería preocuparse por) si está trabajando con un objeto simple o un componente compuesto. Esto simplifica el código cliente, ya que evita la necesidad de escribir funciones con estructuras condicionales (como switch-case) para las clases que definen la composición.

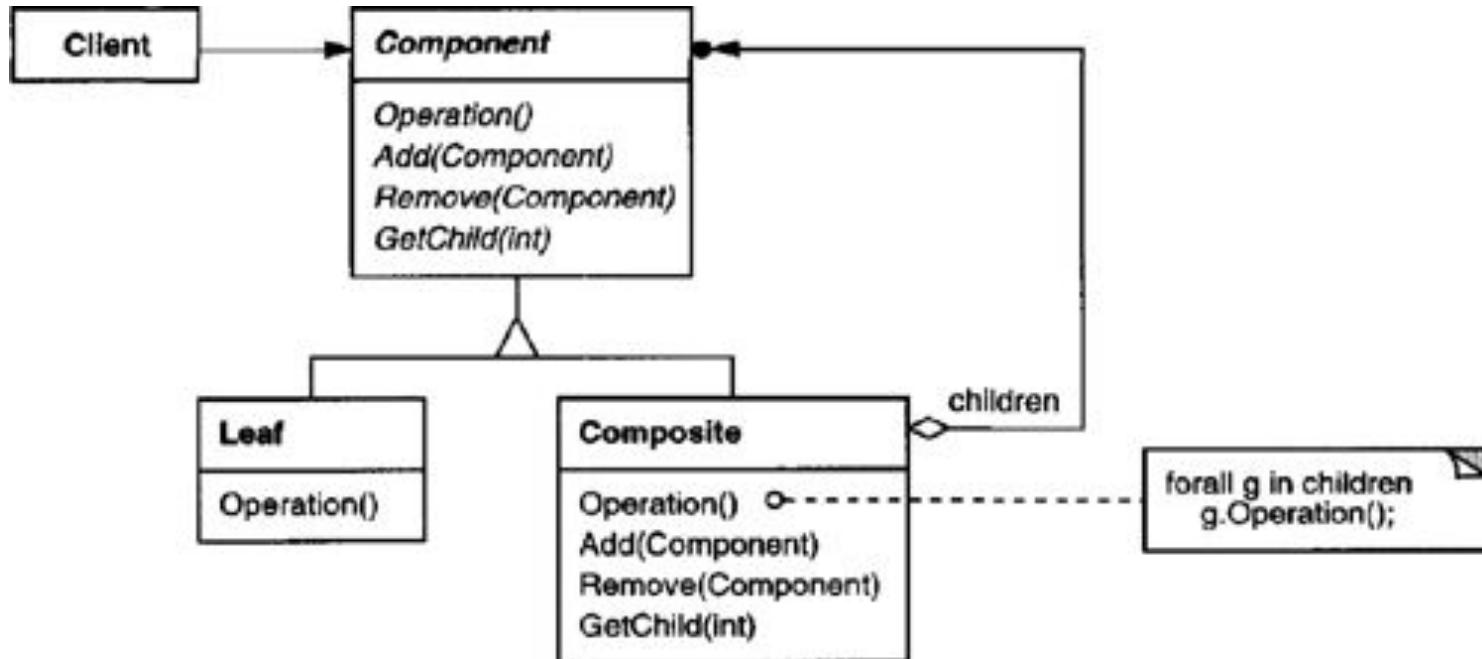
Estructural Patterns - Composite - Características

- Facilita la incorporación de nuevos tipos de componentes.
Las nuevas subclases de componentes (tanto compuestos como simples) funcionan automáticamente con las estructuras y el código de cliente existentes. No es necesario modificar el código de cliente para incorporar nuevas clases de componentes.

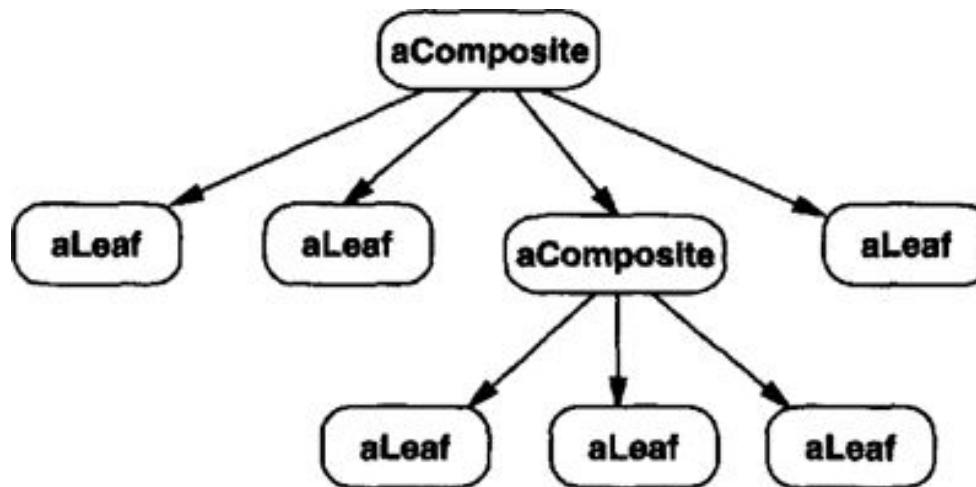
Estructural Patterns - Composite - Características

- Puede hacer que el diseño sea demasiado genérico.
La desventaja de facilitar la incorporación de nuevos componentes es que dificulta la restricción de los componentes de un objeto compuesto. A veces, es necesario que un objeto compuesto contenga únicamente ciertos componentes. Con el patrón Composite, no se puede confiar en el sistema de tipos para imponer estas restricciones. En su lugar, será necesario implementar comprobaciones en tiempo de ejecución.

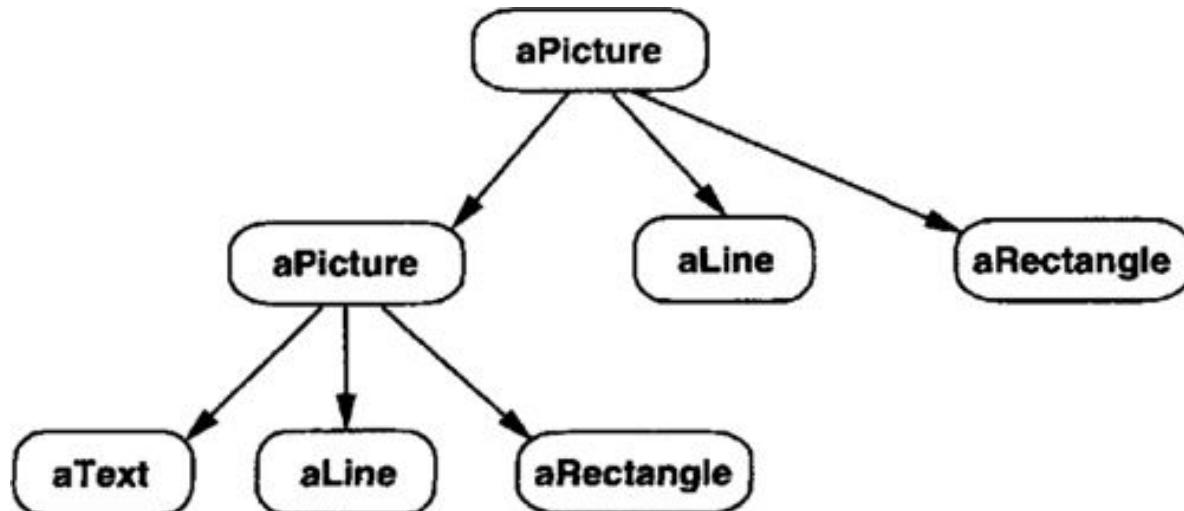
Estructural Patterns - Composite



Estructural Patterns - Composite



Estructural Patterns - Composite



Estructural Patterns - Composite

```
function createLeaf() {  
    return { show: () => console.log("Leaf") };  
}  
  
function createComposite() {  
    const children = [];  
    return {  
        add: c => children.push(c),  
        show: () => {  
            console.log("Composite");  
            children.forEach(child => child.show());  
        } };}  
  
const root = createComposite();  
root.add(createLeaf());  
root.add(createLeaf());  
root.show();  
// Composite  
// Leaf  
// Leaf
```



Estructural Patterns - Decorator

Asignar responsabilidades adicionales a un objeto dinámicamente. Los decoradores ofrecen una alternativa flexible a la subclasiﬁcación para ampliar la funcionalidad.

Estructural Patterns - Decorator - Aplicación

- Añadir responsabilidades a objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos.
- Eliminar responsabilidades
- Cuando la extensión mediante subclases no es práctica.
A veces, es posible un gran número de extensiones independientes, lo que produciría una proliferación de subclases para todas las combinaciones.

Estructural Patterns - Decorator - Características

- Mayor flexibilidad que la herencia estática.
El patrón Decorator proporciona una forma más flexible de añadir responsabilidades a los objetos que la herencia estática (múltiple).

Con los decoradores, se pueden añadir y eliminar responsabilidades en tiempo de ejecución simplemente asociándolas y desvinculándolas. En cambio, la herencia requiere la creación de una nueva clase para cada responsabilidad adicional.

Estructural Patterns - Decorator - Características

- Evita clases con muchas funciones en los niveles superiores de la jerarquía.

Decorator ofrece un enfoque de pago por uso para añadir responsabilidades. En lugar de intentar incluir todas las funciones previsibles en una clase compleja y personalizable, puedes definir una clase simple y añadir funcionalidad gradualmente con objetos Decorator.

Estructural Patterns - Decorator - Características

- Un decorator y su componente no son idénticos.

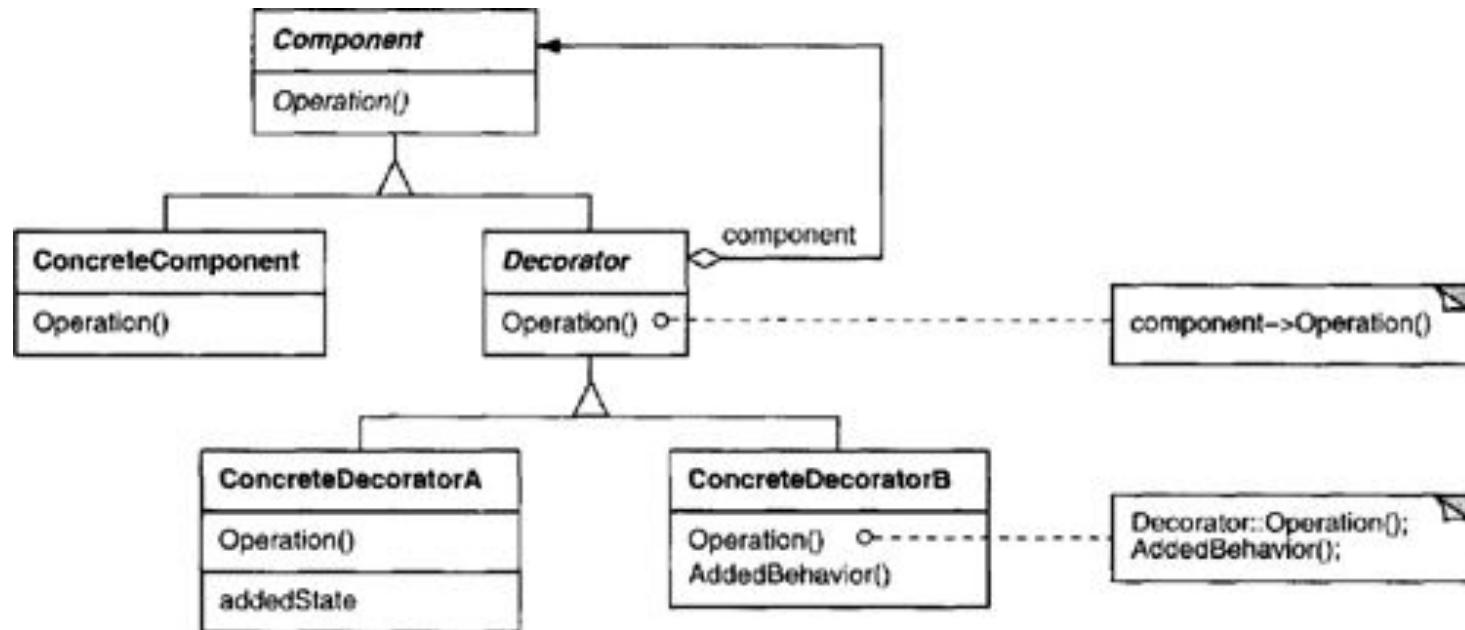
Un decorator actúa como un contenedor transparente. Sin embargo, desde el punto de vista de la identidad del objeto, un componente decorado no es idéntico al componente en sí. Por lo tanto, no se debe confiar en la identidad del objeto al usar decorators.

Estructural Patterns - Decorator - Características

- Muchos objetos pequeños.

Un diseño que utiliza Decorator suele generar sistemas compuestos por muchos objetos pequeños que se parecen entre sí. Los objetos difieren únicamente en su interconexión, no en su clase ni en el valor de sus variables. Si bien estos sistemas son fáciles de personalizar para quienes los entienden, pueden ser difíciles de aprender y depurar.

Estructural Patterns - Decorator



Estructural Patterns - Decorator

```
function coffee() {  
    return { cost: () => 5, description: () => 'Café' };  
}  
  
function milkDecorator(coffeeObj) {  
    return {  
        cost: () => coffeeObj.cost() + 2,  
        description: () => coffeeObj.description() + ' + leche'  
    };  
}  
  
function sugarDecorator(coffeeObj) {  
    return {  
        cost: () => coffeeObj.cost() + 1,  
        description: () => coffeeObj.description() + ' + azúcar'  
    };  
}
```



Estructural Patterns - Decorator

```
// Café simple
let c = coffee();
console.log(c.cost(), c.description());           // 5 'Café'

// Café con leche
let milkCoffee = milkDecorator(c);
console.log(milkCoffee.cost(), milkCoffee.description()); // 7 'Café + leche'

// Café con leche y azúcar
let milkSugarCoffee = sugarDecorator(milkDecorator(c));
console.log(milkSugarCoffee.cost(), milkSugarCoffee.description()); // 8 'Café + leche + azúcar'

// Café solo con azúcar
let sugarCoffee = sugarDecorator(c);
console.log(sugarCoffee.cost(), sugarCoffee.description()); // 6 'Café + azúcar'
```

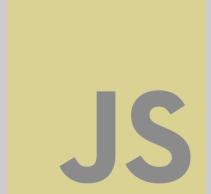


Behavioral Patterns

State

Strategy

Observer



JS

Behavioral Patterns - State

Permite que un objeto modifique su comportamiento cuando cambia su estado interno. El objeto parecerá cambiar de clase.

Behavioral Patterns - State - Aplicación

El comportamiento de un objeto depende de su estado y debe cambiar su comportamiento en tiempo de ejecución dependiendo de ese estado.

Las operaciones tienen sentencias condicionales extensas y dependen del estado del objeto.

Este estado suele representarse mediante una o más constantes enumeradas. A menudo, varias operaciones contienen la misma estructura condicional.

El patrón "Estado" asigna cada rama de la condicional a una clase independiente.

Behavioral Patterns - State - Características

- Localiza el comportamiento específico de cada estado y lo divide en particiones para diferentes estados.

El patrón Estado integra todo el comportamiento asociado a un estado específico en un solo objeto. Dado que todo el código específico de cada estado reside en una subclase Estado, se pueden añadir fácilmente nuevos estados y transiciones definiendo nuevas subclases.

Behavioral Patterns - State - Características

- Hace explícitas las transiciones de estado.

Cuando un objeto define su estado actual únicamente en términos de valores de datos internos, sus transiciones de estado no tienen una representación explícita; solo se muestran como asignaciones a algunas variables.

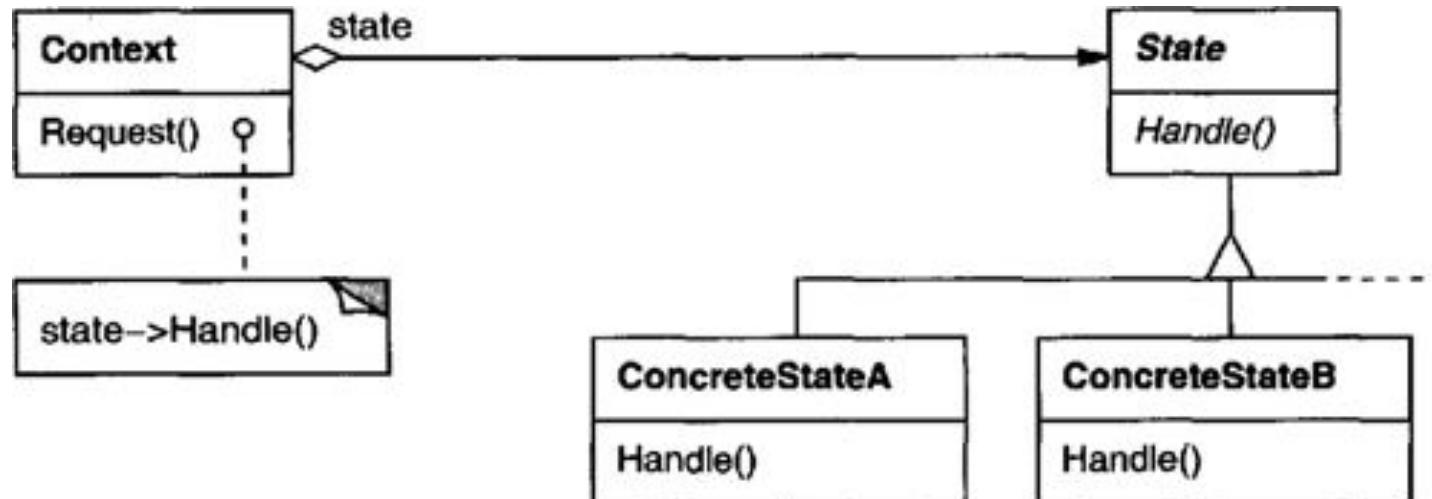
Introducir objetos separados para diferentes estados hace que las transiciones sean más explícitas.

Behavioral Patterns - State - Características

- Los objetos de estado se pueden compartir.

Si los objetos de estado no tienen variables de instancia (es decir, el estado que representan está codificado completamente en su tipo), los contextos pueden compartir un objeto de estado.

Behavioral Patterns - State



Behavioral Patterns - State

```
const onState = { handle: () => console.log("ON") };
const offState = { handle: () => console.log("OFF") };

function createSwitch() {
    let state = offState;
    return {
        press: function () {
            state.handle();
            state = (state === offState) ? onState : offState;
        }
    };
}

const s = createSwitch();
s.press(); // OFF
s.press(); // ON
s.press(); // OFF
```



Behavioral Patterns - Strategy

Define una familia de algoritmos, encapsula cada uno y los hace intercambiables.

El patrón Strategy permite que el algoritmo varíe independientemente de los clientes que lo utilizan.

Behavioral Patterns - Strategy - Aplicación

Muchas clases relacionadas difieren únicamente en su comportamiento.

Las estrategias permiten configurar una clase con uno de varios comportamientos.

Una clase define muchos comportamientos, que aparecen como múltiples sentencias condicionales en sus operaciones.

En lugar de muchas sentencias condicionales, mueva las ramas condicionales relacionadas a su propia clase Strategy.

Behavioral Patterns - Strategy - Aplicación

Se necesitan diferentes variantes de un algoritmo.

Por ejemplo, se podrían definir algoritmos que reflejen diferentes equilibrios espacio-tiempo. Se pueden utilizar estrategias cuando estas variantes se implementan como una jerarquía de clases de algoritmos.

Un algoritmo utiliza datos que los clientes no deberían conocer.

Utilice el patrón Strategy para evitar exponer estructuras de datos complejas y específicas del algoritmo.

Behavioral Patterns - Strategy - Características

- Familias de algoritmos relacionados.

Las jerarquías de clases de estrategia definen una familia de algoritmos o comportamientos que los contextos pueden reutilizar. La herencia puede ayudar a identificar la funcionalidad común de los algoritmos.

- Una alternativa a la subclasicación.

Encapsular el algoritmo en clases Strategy independientes permite modificarlo independientemente de su contexto, lo que facilita su cambio, comprensión y extensión.

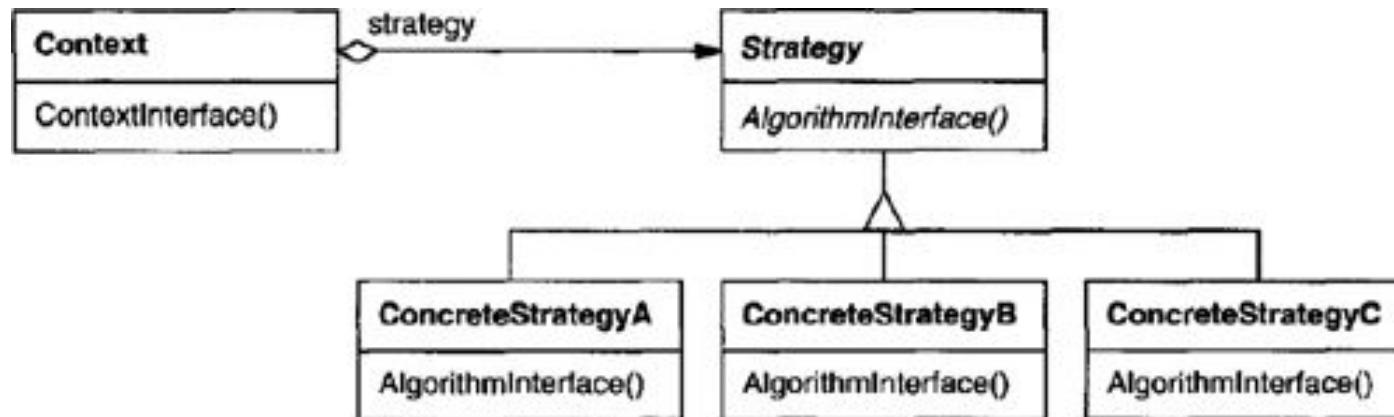
Behavioral Patterns - Strategy - Características

- Las estrategias eliminan las declaraciones condicionales.
El patrón Estrategia ofrece una alternativa a las declaraciones condicionales para seleccionar el comportamiento deseado.
- Selección de implementación.
Las estrategias pueden proporcionar diferentes implementaciones del mismo comportamiento. El cliente puede elegir entre estrategias con diferentes compensaciones de tiempo y espacio.

Behavioral Patterns - Strategy - Características

- Los clientes deben conocer las diferentes estrategias.
Este patrón presenta una posible desventaja: el cliente debe comprender cómo difieren las estrategias antes de poder seleccionar la adecuada.
- Mayor número de objetos.
Las estrategias aumentan el número de objetos en una aplicación. A veces, se puede reducir esta sobrecarga implementando estrategias como objetos sin estado que los contextos pueden compartir.

Behavioral Patterns - Strategy



Behavioral Patterns - Strategy

```
const add = { execute: arr => arr.reduce((a,b)=>a+b, 0) };
const multiply = { execute: arr => arr.reduce((a,b)=>a*b, 1) };

function createContext(strategy) {
  let strat = strategy;
  return {
    setStrategy: s => strat = s,
    execute: arr => strat.execute(arr)
  };
}

const c = createContext(add);
console.log(c.execute([1,2,3])); // 6
c.setStrategy(multiply);
console.log(c.execute([1,2,3])); // 6
```



Behavioral Patterns - Observer

Define una dependencia de uno a muchos objetos así cuando un objeto cambie de estado, todos sus dependientes sean notificados y actualizados automáticamente.

Behavioral Patterns - Observer - Aplicación

- Cuando una abstracción tiene dos enfoques, uno dependiente del otro, encapsularlos en objetos separados permite modificarlos y reutilizarlos de forma independiente.
- Cuando un cambio en un objeto requiere cambiar otros y no se sabe cuántos objetos deben cambiarse.

Behavioral Patterns - Observer - Aplicación

- Cuando un objeto debería poder notificar a otros objetos sin hacer suposiciones sobre quiénes son.
En otras palabras, no se desea que estos objetos estén estrechamente acoplados.

Behavioral Patterns - Observer - Características

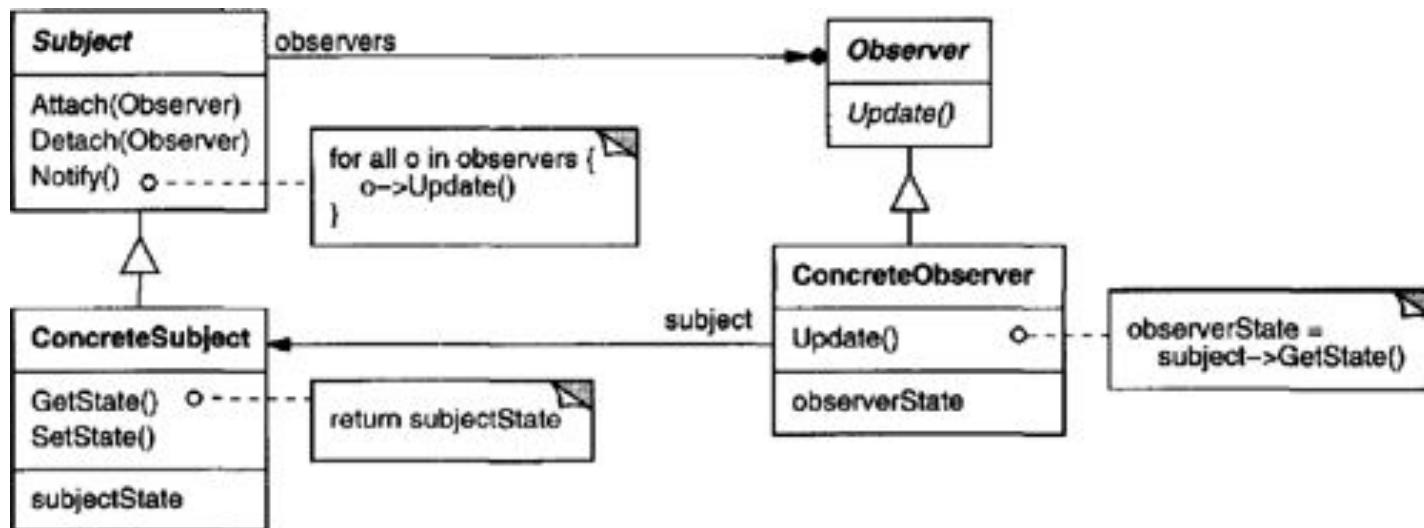
- Acoplamiento abstracto entre sujeto y observador.
Un sujeto solo sabe que tiene una lista de observadores, cada uno conforme a la interfaz simple de la clase abstracta Observador. El sujeto desconoce la clase concreta de ningún observador. Por lo tanto, el acoplamiento entre sujetos y observadores es abstracto y mínimo.
- Compatibilidad con Broadcasting.
A diferencia de una solicitud normal, la notificación que envía un sujeto no necesita especificar su destinatario. La notificación se difunde automáticamente a todos los objetos interesados que se suscribieron.

Behavioral Patterns - Observer - Características

- Actualizaciones inesperadas.

Dado que los observadores desconocen la presencia de los demás, pueden pasar por alto el coste final de cambiar de tema. Una operación aparentemente inocua sobre el tema puede provocar una cascada de actualizaciones para los observadores y sus objetos dependientes.

Behavioral Patterns - Observer



Behavioral Patterns - Observer

```
function createSubject() {  
    const observers = [];  
    return {  
        attach: obs => observers.push(obs),  
        notify: msg => observers.forEach(o => o.update(msg))  
    };  
}  
  
function createObserver() {  
    return { update: msg => console.log("Received:", msg) };  
}  
  
const s = createSubject();  
const o1 = createObserver(), o2 = createObserver();  
s.attach(o1);  
s.attach(o2);  
s.notify("Hello"); // Ambos observadores reciben "Hello"
```



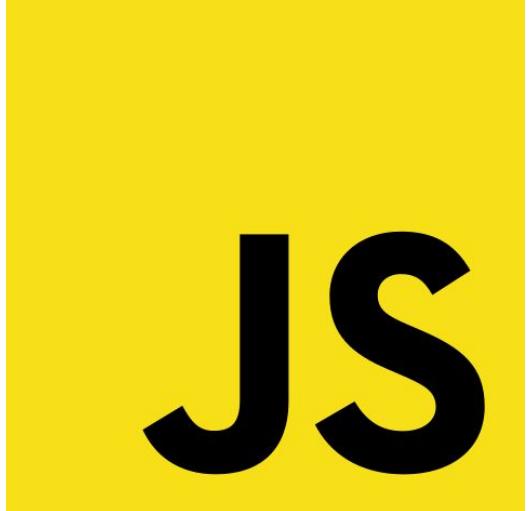
Referencia

- Freeman E, Robson E (2020), Head First Design Patterns, 2nd Edition, O'Reilly Media, Inc.
- Gamma E, Helm R, Johnson R, Vlissides J (1994), Design Patterns: Elements of Reusable Object-Oriented Software, O'Reilly Media, Inc.

Próxima Clase

Module & Creational Pattern

JavaScript



JS

Programacion Orientada a Objetos 2

Ing. Jose Rusca

Programacion Orientada a Prototipos



Clase de hoy

- Sistema de control de versiones
- Comandos principales de Git
- Flujo de Git (Git-Flow)

Sistema de control de Versiones

Un sistema de **control de versiones** (a veces llamado control de revisión) es una herramienta que le permite rastrear el historial de cambios de los archivos del proyecto a lo largo del tiempo (almacenados en un repositorio) y ayuda a los desarrolladores a trabajar en equipo.

Software de control de Versiones

- SVN - Sub-version
- Team Foundation (Microsoft)
- Clear Case (IBM)
- GIT (Linus Torvalds -> Linux)
- otros....

Comandos de Git

- Init
- Clone
- Status
- Add
- Commit
- Push/Pull
- Stash
- Merge/Rebase
- Fetch
- Branch
- Checkout
- reset/revert

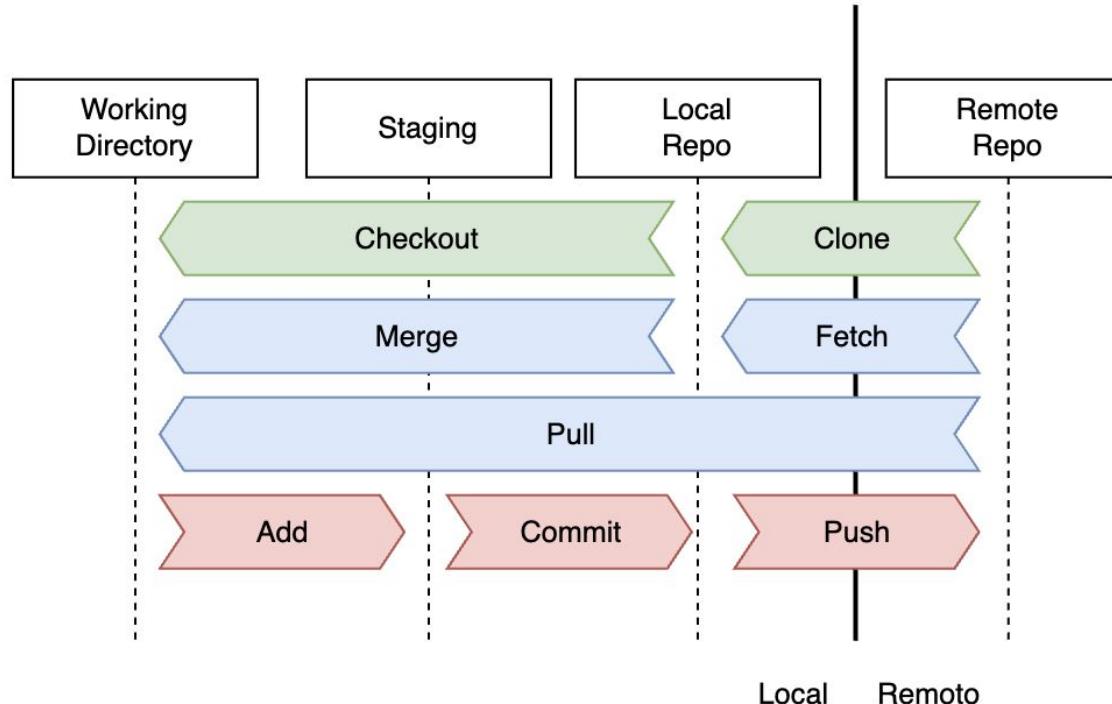
Comandos de Git

Init	Crea un repositorio de forma local
Clone	Copiar el repositorio remoto al local
Status	Ver el estado actual del proyecto
Add	Agregar contenido al stage
Commit	Enviar cambios al repo local
Push	Enviar cambios del repositorio remoto
Fetch	Trae los cambios al repositorio local

Comandos de Git

Pull	Hace un fetch y luego trae los cambios al directorio de trabajo
Stash	Guarda cambios temporales
Merge/Rebase	Une dos ramas distintas
Branch	Permite realizar acciones sobre la rama (renombrar, etc)
Checkout	Cambia/Crea una rama nueva
reset/revert	Revierte cambios

Stage Architecture of Git



Git - Ramas - Definicion

Una rama representa una línea independiente de desarrollo dentro de un repositorio y nos permite trabajar de manera aislada, sin afectar el código principal.

Git - Pull Request

Un PR (pull request) es un mecanismo que nos permite llevar cambios de una rama a otra, generalmente de una rama de desarrollo a una principal. Este mecanismo dispara un proceso de code review informando a los miembros del equipo para su revisión y posterior aprobación.

Git - Code Review

Es un proceso en donde uno o más desarrolladores examinan el código escrito por otro miembro del equipo para identificar errores, refactorizar y asegurar el cumplimiento de los estándares de calidad, antes de integrarlo a la rama de código principal.

Git - Pasos

- Crear una rama feature, desde la rama de develop.
- Realizar los cambios en la nueva rama (feature).
- Finalizado los cambios se crea una solicitud para incorporar cambios, (Pull request)
- La PR muestra las diferencias entre las ramas de origen y destino.
- Los miembros del equipo revisan los cambios propuestos, los debaten y dejan comentarios.
- El desarrollador puede responder a los comentarios, realizar nuevas confirmaciones y actualizar la PR.
- Una vez aprobados, los cambios se incorporan a la rama principal.

Git Flow

Es un modelo de ramas para GIT que se caracteriza por poseer ramas principales y de soporte, que permite organizar el desarrollo de software.

<https://nvie.com/posts/a-successful-git-branching-model/>

Git Flow - Ramas

Principales:

- Develop
- Master

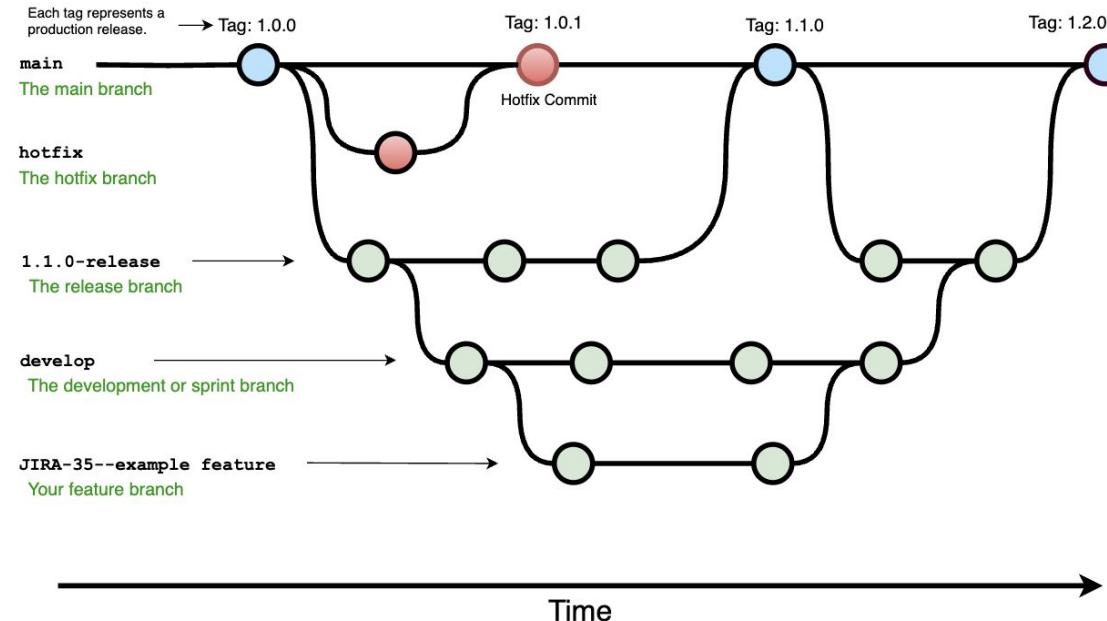
De soporte:

- Feature
- Release
- Hotfix

Git Flow

Example diagram for a workflow similar to "Git-flow" :

See: <https://nvie.com/posts/a-successful-git-branching-model/>



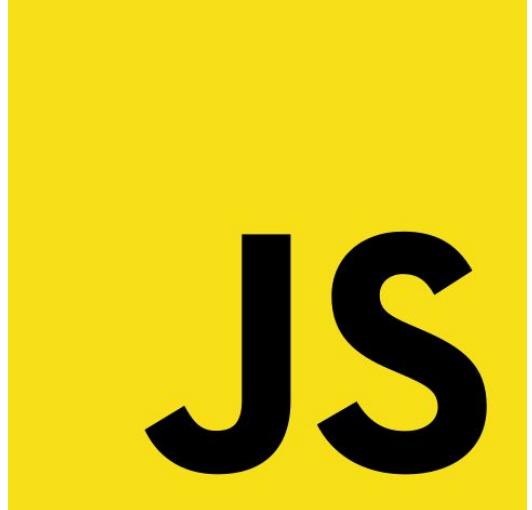
Referencia

- <https://git-scm.com/book>
- www.github.com
- Jakub Narębski (2024) - Mastering Git - Second Edition
- Vincent Driessen (2010) -
<https://nvie.com/posts/a-successful-git-branching-model/>

Próxima Clase

Prototipos - Module & Creational Patterns

JavaScript



JS

Programacion Orientada a Objetos 2

Ing. Jose Rusca
Ing. Federico Stulich
Ing. Francisco De Grandis

Desarrollo Iterativo e Incremental



Clase de hoy

- Testing Framework - Jest
- Unit Test
- UT - Desventajas
- Test Driven Development

JavaScript Testing Framework

Para testing vamos a utilizar Jest como framework de testing.



<https://jestjs.io/>

JavaScript Testing Framework

Su instalación es muy sencilla, solo debemos tener cuidado de instalarlo para el entorno de desarrollo.

Con yarn: yarn add --dev jest

Con npm: `npm install --save-dev jest`

Unit test

Los test nos permiten automatizar la forma de probar el código y mantener la funcionalidad intacta del programa y son ejecutados por los mismos programadores.

Se denomina Test Unitario por que permite probar “Unidades” (bloques) de código (funciones).

Unit Test

Estructura del test:

- Setup Se establece la precondition del código. (Los valores de ingreso).
- Exercise Se “ejercita” el código que se está probando.
- Assert Se compara los resultados obtenidos con los esperados.

También lo van a encontrar como AAA patern (Arrange-Act-Assert)

Unit Test

Vamos a realizar nuestra primera prueba con un simple ejercicio, una función que tome dos valores y los sume.

```
function sum(a, b) {  
    return a + b;  
}  
  
module.exports = sum;
```

sum.js

Unit Test

Vamos a probar nuestra primera prueba con un simple ejercicio, tomemos una función que tome dos valores y los sume.

```
const sum = require('./sum');
```

sum.test.js

{}

```
test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Unit Test

Ejemplo de Pruebas

Unit Test

¿Para qué vamos a utilizar los test?

Para asegurar la integridad del programa, para que las modificaciones que realizamos no modifiquen o generen un comportamiento indeseado.

Los test nos nos van a asegurar código libre de fallas, no debemos pensar que al utilizar esta técnica mi código va a ser perfecto.

Unit Test - Desventajas

Lleva tiempo mantener los test, principalmente con programas complejos.

Es complejo desarrollar buenas pruebas.

Los test suelen escribirse para el código nuevo.

Unit Test - Desventajas

Resistencia al cambio por parte de los distintos actores.

Los **programadores** pueden no sentirse cómodos.. E.E.E
(Educate, Evangelise, Encourage)

Los **gerentes** pueden considerar las pruebas como una pérdida de tiempo.

Test
Driven
Development



TDD

TDD se basa en dos simples premisas

- No escribir una línea de código nuevo a menos que primero tenga una prueba fallida.
- Eliminar la duplicación.

TDD

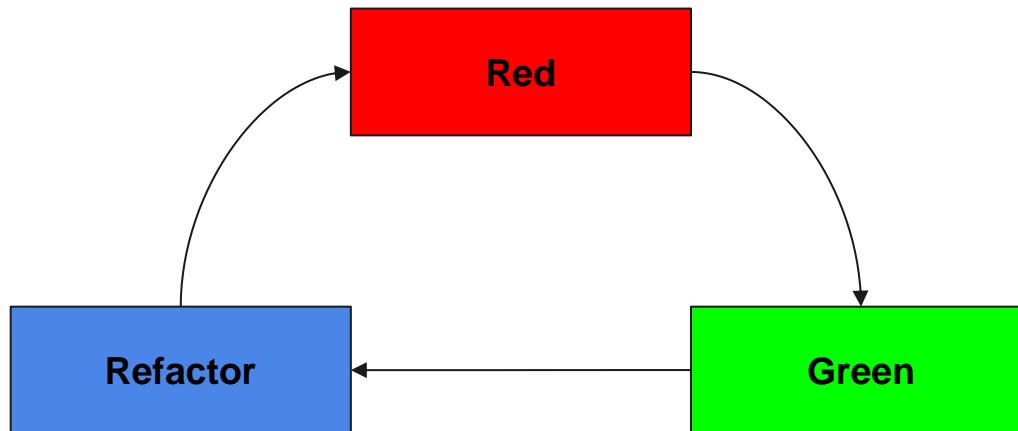
Vamos a poner las dos simples reglas ordenadas para poder programar:

Rojo: Escribir una prueba que no funcione o quizas tampoco compile.

Verde: Hacer que la prueba funcione de forma rápida y cometiendo todos los pecados necesarios en el proceso.

Refactorizar: Eliminar/Mejorar lo realizado para que la prueba funcione.

Test Driven Development

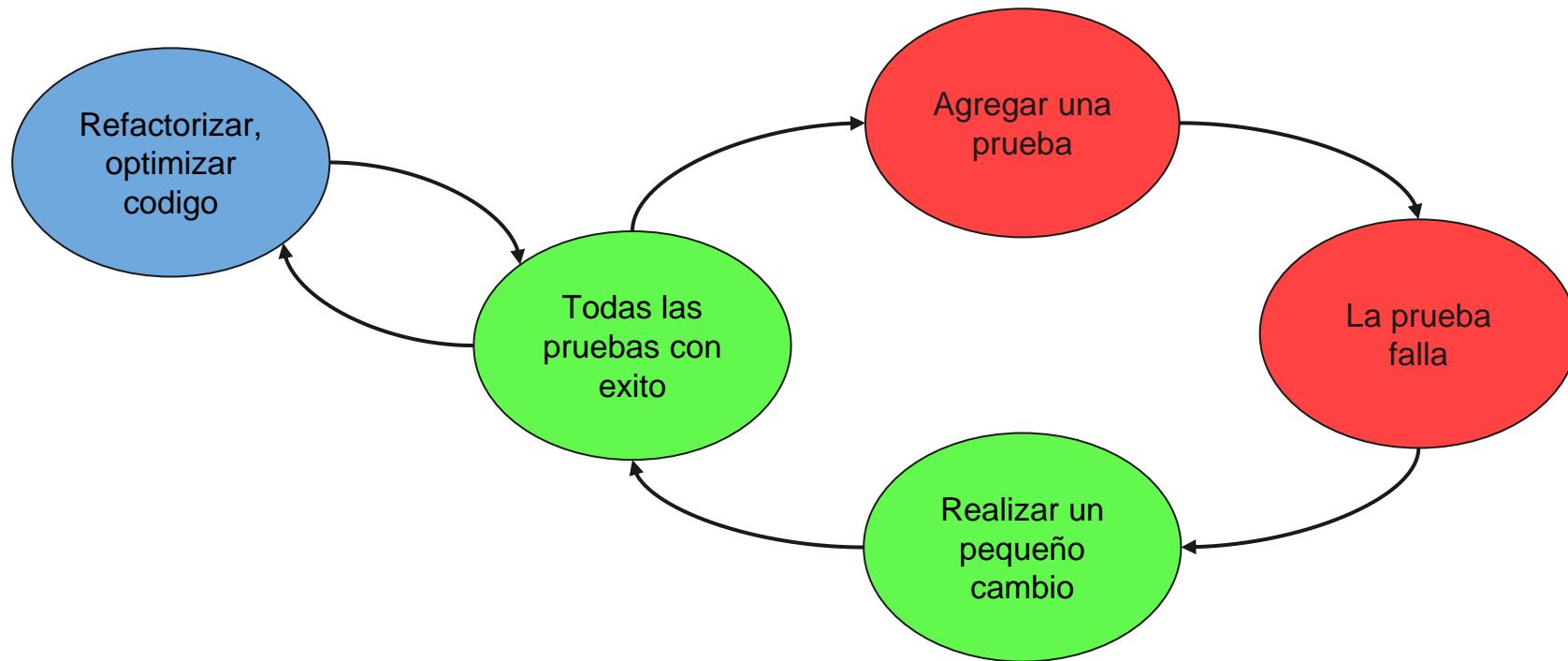


TDD

Del diagrama anterior podemos mejorar y especificar más puntos

1. Agregar una prueba rápidamente
2. Ejecutar todas las pruebas y ver que la nueva falla
3. Realizar un pequeño cambio
4. Ejecutar todas las pruebas y compruebe que todas tienen éxito
5. Refactorizar para optimizar el código (eliminar lo duplicado)

TDD - Cycle



TDD - En la práctica

Vamos entonces a la práctica. Supongamos que necesitamos desarrollar una función que nos devuelva los valores de la serie de **Fibonacci**. En donde voy a tener un valor de entrada **n** que va a ser el número en la serie que quiero obtener y va a devolver dicho valor.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597

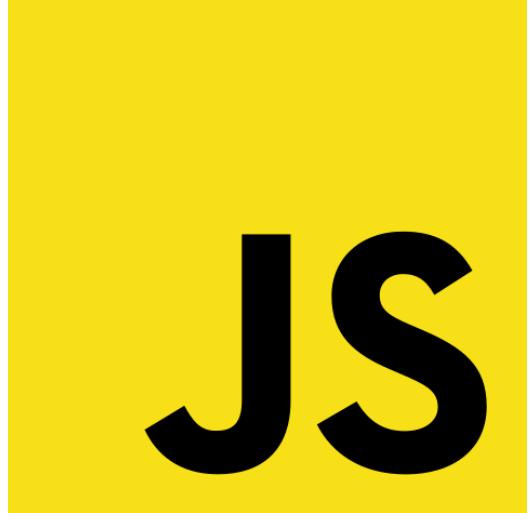
Referencia

- Beck, K (2002) Test-Driven Development by Example, Addison-Wesley (ISBN 978-0321146533)
- <https://jestjs.io/>
- <http://people.apache.org/~stevel/slides/testing.pdf>

Próxima Clase

Test Driven Development

JavaScript



JS

Programacion Orientada a Objetos 2

Ing. Jose Rusca
Ing. Federico Stulich
Ing. Francisco De Grandis

Diseño Avanzado Orientado a Objetos



JS

Clase de hoy

- Nuestro objetivo
- Refactorización - Definiciones
- Motivos de Refactorización
- Cuando no refactorizar
- Bad Smells in Code (Code Smells)

Cual es nuestro objetivo?



Cual es nuestro objetivo?

Desarrollar software....

- más Robusto
- más Claro
- más Reutilizable

Cual es nuestro objetivo?

Hasta ahora demostramos que nuestro código, si bien es bueno (cumple su funcionalidad), podría mejorar ...

... Cómo lo vamos a mejorar? ...

... Refactorizando

Refactorizar - Definición (Sustantivo)

Es un cambio realizado en la estructura interna del software para que sea más fácil de entender y más económico de modificar sin cambiar su comportamiento observable.

Refactorizar - Definición (Verbo)

Reestructurar el software aplicando una serie de modificaciones (refactorizaciones) sin cambiar su comportamiento observable.

The Two Hats

Al momento de desarrollar debemos enfocarnos en dos actividades.

- Agregar funcionalidad a lo existente
- Refactorizar

A qué nos recuerda esto?

¿Por qué debemos Refactorizar?

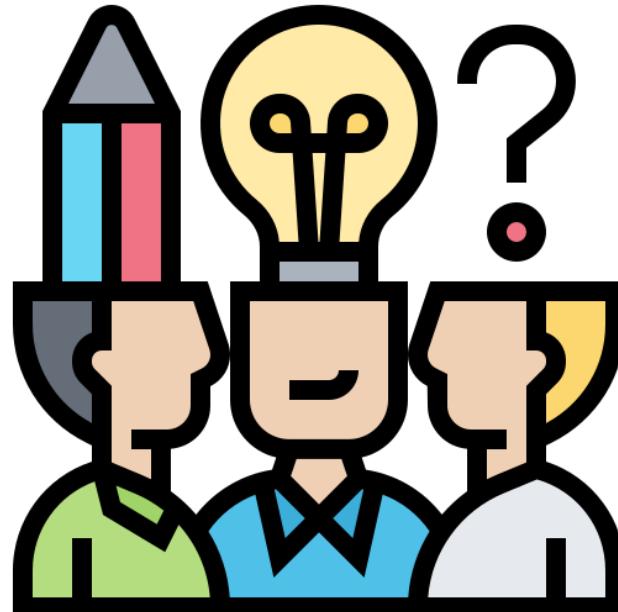
- Mejora el diseño del software.
- Software facil de entender/comprender.
- Ayuda a encontrar errores (Bugs).
- Ayuda a programar más rápido.

Regla de los Tres

La primera vez que haces algo, simplemente, lo haces, la segunda vez que tienes que hacer algo similar te altera realizar la duplicación, pero la haces.

La tercera vez... Lo refactorizas.

Cuando debemos Refactorizar?



Cuando debemos Refactorizar?

Siempre...!

En qué escenario debemos Refactorizar?

- Preparando el terreno (Antes de agregar funcionalidad).
- Comprensión de Código.
- Sacando la basura (una variante del punto anterior).
- Refactorización planificada y oportunista.
- Refactorización a largo plazo.
- Refactorizando en una revisión de código.

Cómo justificar con el gerente?

Podemos encontrarnos con dos tipos de gerentes. Los que tienen un perfil técnico y se preocupan por mejorar el código, en cuyo caso no va a ser difícil convencerlo de que hay que mejorar el código y que esto va a ser un beneficio a largo plazo.

Cómo justificar con el gerente?

En el caso de que el gerente no tenga un enfoque técnico y prefiera apegarse más a lo funcional, podemos tomar una actitud un poco controversial....

... “No decir nada”

Cómo justificar con el gerente?

Es esto una actitud subversiva?

No, por que somos profesionales y lo que estamos modificando es para mejorar el producto final que estamos desarrollando.

Está de más decir que esto aplica para refactorizaciones pequeñas y no para un enfoque a largo plazo.

Cuando no Refactorizar

Si bien dijimos que siempre debemos refactorizar, cuando estamos agregando una nueva funcionalidad y nos referimos al proceso de programar y que en la mayoría de los pasos que damos debemos refactorizar. Hay situaciones delicadas que debemos no refactorizar.

Cuando no Refactorizar

No debemos refactorizar cuando:

- Se vea afectado el rendimiento del programa.
- No podemos traspasar las fronteras de nuestro código (Código propio).
- Trabajamos en distintas ramas (Branches -> Merging).

Cuando no Refactorizar

No debemos refactorizar cuando:

- Asegurar el comportamiento de la aplicación (Testing).
- Legacy Code (Código complejo, viejo, sin pruebas y escrito por alguien que no está en el equipo).
- Bases de Datos (No imposible pero complejo, debemos refactorizar la base de datos).

Bad Smells in Code (Code Smells)

*If it stinks,
... change it!*



Bad Smells in Code (Code Smells)

Con lo visto hasta ahora tenemos una idea de cómo refactorizar, ahora tenemos que detectar que es lo que vamos a refactorizar. Para ello debemos identificar ciertas situaciones en las que el código “Apestá” y poder refactorizar para solucionarlas.

Bad Smells in Code (Code Smells)

Los vamos a agrupar de la siguiente manera

- Bloaters
- Object Oriented Abusers
- Change Preventers
- Disposable
- Couplers

Code Smells - Bloaters

Código, métodos y clases que han crecido de forma gigantesca.

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps

Code Smells - OO Abusers

Incorrecta o incompleta implementación del paradigma.

- Alternative Classes with Different Interfaces
- Refused Bequest (Herencia rechazada)
- Switch Statements
- Temporary Field

Code Smells - Change Preventers

Si vamos a realizar un cambio en un solo lugar, debemos modificar muchos más lugares del código.

- Divergent Change
- Parallel Inheritance Hierarchies
- Shotgun Surgery

Code Smells - Dispensables (Prescindibles)

Código sin sentido

- Comments
- Duplicate Code
- Data Class
- Dead Code
- Lazy Class
- Speculative Generality

Code Smells - Couplers (Acopladores)

Excesivo acoplamiento entre clases

- Feature Envy
- Inappropriate Intimacy
- Incomplete Library Class
- Message Chains
- Middle Man

Code Smells

Vamos a ver en detalle cada caso y como enfocar la refactorización de cada caso.

Code Smells

MYSTERIOUS NAME

- Change Function Declaration
- Rename Variable
- Rename Field

Code Smells

DUPLICATED CODE

- **Extract Function** Si esta en dos métodos dentro de la misma clase.
- **Slide Statements** Si uno se encuentra con que el código no es idéntico.
- **Pull Up Method** Si el código se encuentra en subclases

Code Smells

LONG FUNCTION Funciones con más de diez líneas de código deben empezar a hacernos dudar.

- **Extract Function** Para acortar la función.
- **Replace Temp with Query** Si posee demasiados temporales.
- **Introduce Parameter Object / Preserve Whole Object** Si posee demasiados parametros.
- **Replace Function with Command** Si lo anterior no funciona y seguimos con muchos temporales y parámetros.

Code Smells

LONG PARAMETER LIST

Más de tres o cuatro parámetros en el método.

- Replace Parameter with Query(method)
- Preserve Whole Object Evitar temporales y pasar todo el objeto.
- Introduce Parameter Object

Code Smells

Global Data Los datos globales pueden ser accedidos desde cualquier parte.

- Encapsulate Variable

Code Smells

MUTABLE DATA Cambios en los datos pueden desencadenar errores inesperados y bugs muy difíciles de identificar.

- Encapsulate Variable
- Split Variable
- Slide Statements & Extract Function
- En APIs: Separate Query from Modifier & Remove Setting Method

Code Smells

DIVERGENT CHANGE Un cambio dispara muchos cambios en una sola clase.

- Split Phase
- Move Function
- Extract Function
- Extract Class

Code Smells

SHOTGUN SURGERY

Un cambio dispara muchos cambios en diferentes clases (opuesto a Cambio Divergente)

- Move Function & Move Field
- Combine Functions into Class
- Combine Functions into Transform
- Split Phase

Code Smells

FEATURE ENVY

Sucede cuando un módulo se pasa mas hablando con otro modulo que con sus propias características.

- Move Function
- Extract Function

Code Smells

DATA CLUMPS Muchas clases contienen información repetida (grupos de variables). Deberían agruparse en una clase propia.

- Extract Class
- Introduce Parameter Object
- Preserve Whole Object

Code Smells

PRIMITIVE OBSESSION

Muchos lenguajes están construidos sobre un conjunto de datos primitivos y luego las bibliotecas agregan un conjunto de objetos. Debemos evitar el uso de primitivos.

- Replace Primitive with Object
- Replace Type Code with Subclasses
- Replace Conditional with Polymorphism
- Extract Class & Introduce Parameter Object

Code Smells

REPEATED SWITCHES

Cualquier evangelista orientado a objetos tarde o temprano hablará de lo malo que son las sentencias de decisión.

- Replace Conditional with Polymorphism

Code Smells

Loops Debemos dejar que las colecciones se encarguen del proceso. Hoy en dia todos los lenguajes con funciones de primera clase pueden manejarlo.

- Replace Loop with Pipeline

Code Smells

LAZY ELEMENT

Un elemento que es una simple función, seguramente cuando se diseño se pensó que iba a crecer pero nunca llegó a nada y debemos dejarla morir con dignidad.

- Inline Function
- Inline Class
- Collapse Hierarchy

Code Smells

SPECULATIVE GENERALITY

Una funcionalidad que, al momento de diseño, se pensó que iba a ser genial, pero en realidad nunca fue necesaria.

- Collapse Hierarchy
- Inline Function and Inline Class
- Change Function Declaration
- Remove Dead Code

Code Smells

TEMPORARY FIELD

Un campo de un objeto que solo es útil en determinadas circunstancias.

- Extract Class
- Move Function
- Introduce Special Case

Code Smells

MESSAGE CHAINS Cuando se produce una llamada consecutiva de llamadas a diferentes objetos.

- Hide Delegate
- Extract Function
- Move Function

Code Smells

MIDDLE MAN Una clase que lo único que hace es delegar la tarea a otra clase.

- Remove Middle Man
- Inline Function
- Replace Superclass with Delegate
- Replace Subclass with Delegate

Code Smells

INSIDER TRADING Los módulos deben ser diseñado de forma sólida y al intercambiar datos en exceso aumenta el acoplamiento. Para que las cosas funcionen, es necesario que se produzca algún intercambio, pero debemos reducirlo al mínimo y mantenerlo todo por encima de la mesa.

- Move Function
- Move Field
- Hide Delegate
- Replace Subclass with Delegate
- Replace Superclass with Delegate

Code Smells

LARGE CLASS Una clase con demasiados campos, métodos y líneas de código.

- Extract Class
- Extract Superclass
- Replace Type Code with Subclasses

Code Smells

ALTERNATIVE CLASSES WITH DIFFERENT INTERFACES

Dos clases realizan la misma funcionalidad pero sus interfaces son diferentes (nombre o parámetros).

- Change Function Declaration
- Move Function
- Extract Superclass

Code Smells

DATA CLASS

Una clase que solo contiene valores en sus propiedades y su funcionalidad solo se limita a setters y getters.

- Encapsulate Record
- Remove Setting Method
- Move Function
- Extract Function

Code Smells

REFUSED BEQUEST (Herencia Rechazada) Una clase utiliza solo algunos métodos de los que hereda, los métodos no utilizados no son necesarios o en muchos casos darían una excepción.

- Push Down Method
- Push Down Field
- Replace Subclass with Delegate
- Replace Superclass with Delegate

Code Smells

COMMENTS

No estamos diciendo que no hay que utilizar comentarios, sino que muchas veces el comentario se utiliza como desodorante.

- Extract Function
- Change Function Declaration
- Introduce Assertion

Referencia

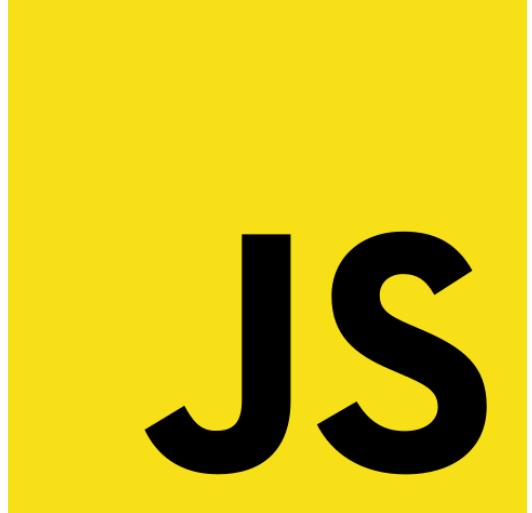
- Fowler, M. (2019) Improving the Design of Existing Code - Second Edition - Addison-Wesley
- <https://refactoring.com/>
- <https://refactoring.guru/>

Iconos de www.flaticon.com

Próxima Clase

Refactoring - Patterns

JavaScript



JS

Programacion Orientada a Objetos 2

Ing. Jose Rusca
Ing. Federico Stulich
Ing. Francisco De Grandis

Diseño Avanzado Orientado a Objetos



JS

Clase de hoy

- Null Object Pattern
- Catálogo de Refactorización
 - Replace Conditional with Polymorphism
 - Replace Superclass with Delegate
 - Replace Subclass with Delegate
- Composition
 - Interface
 - Forwarding Methods/Delegate
 - Mixins

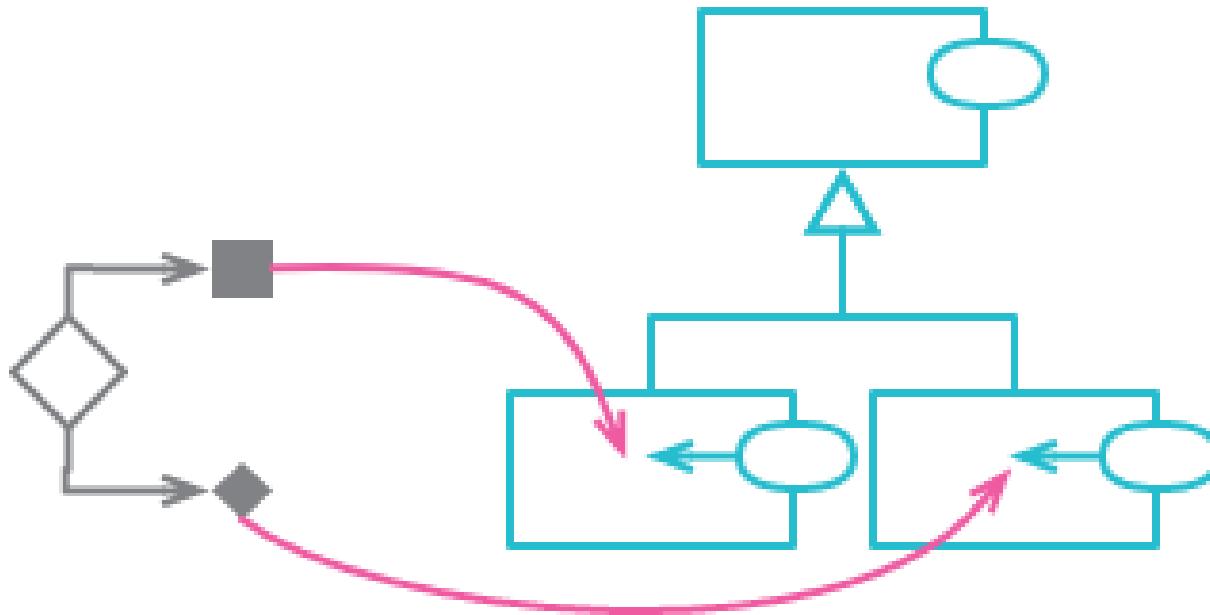
Null Object Pattern

Proporcione una alternativa para un objeto que comparte la misma interfaz pero no hace nada. El objeto nulo encapsula las decisiones de implementación de cómo "no hacer nada" y oculta esos detalles a sus colaboradores.

Null Object Pattern

```
function NullBird() {  
    this.sound = function() {  
        return "Silence.";  
    } }  
  
function getAnimal(type) {  
    return type === 'Pidgeon' ? new Pidgeon() : new NullBird();  
}  
  
['Pidgeon', null].map( (bird) => getAnimal(bird).sound());  
//[ 'Tweet!!! Tweet!!!!', 'Silence.' ]  
  
function Pidgeon(name) {  
    this.sound = function() {  
        return "Tweet!!! Tweet!!!!";  
    } }  
}
```

Replace Conditional with Polymorphism



Replace Conditional with Polymorphism

Es muy común que nos encontremos con un código, que tome decisiones en base a diferentes características de los objetos, en este caso (y en muchos otros) podemos reemplazar el condicional por un concepto polimórfico.

Replace Conditional with Polymorphism

```
function calcularSueldo(empleado) {  
  
    switch(empleado.type) {  
  
        case 'Salario':  
  
            return empleado.salario;  
  
        case 'Comision':  
  
            return empleado.ventasTotales * empleado.comision;  
  
        case 'SalarioXComision':  
  
            return empleado.salario + (empleado.ventasTotales * empleado.comision);  
  
        default:  
  
            return 0;  
  
    }  
}
```

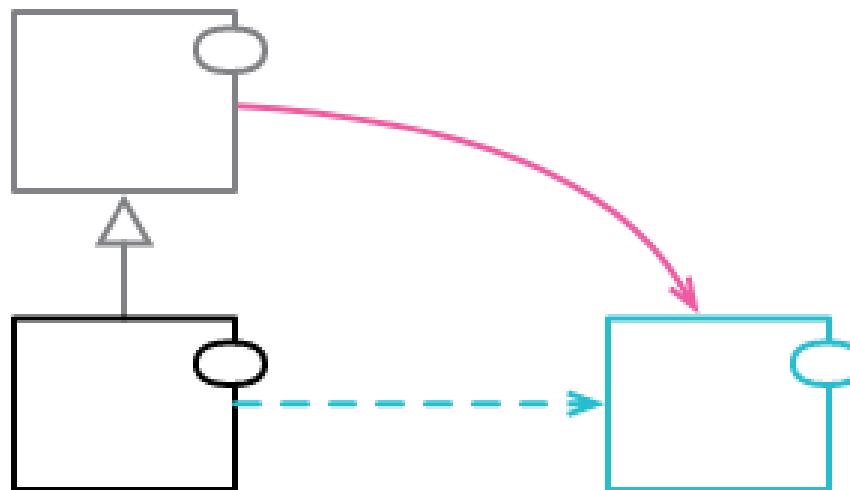
Replace Conditional with Polymorphism

Refactoricemos....

Replace Conditional with Polymorphism

```
function calcularSueldo(empleado) {  
    return empleado.calcularSueldo();  
};  
  
{  
    Const empleadoConSalario = function () {  
        ...  
        this.calcularSueldo = function () {  
            return this.salario;  
        }  
    };  
  
    Const empleadoConComision = function () {  
        ...  
        this.calcularSueldo = function () {  
            return this.ventasTotales *  
                this.comision;  
        }  
    };  
  
    Const empleadoConSalarioYComision =  
        function () {  
            ...  
            this.calcularSueldo = function () {  
                return this. salario +  
                    (this.ventasTotales *  
                     empleado.comision);  
            }  
        };  
}
```

Replace Superclass with Delegate



Replace Superclass with Delegate

La herencia es una herramienta muy poderosa y que permite fácilmente reutilizar código. Tomamos una clase, anulamos y agregamos algunas cosas.....

Pero..... Todo esto puede llevar a un código muy complicado y confuso.

Alias: Replace Inheritance with Delegation

Replace Superclass with Delegate

{}

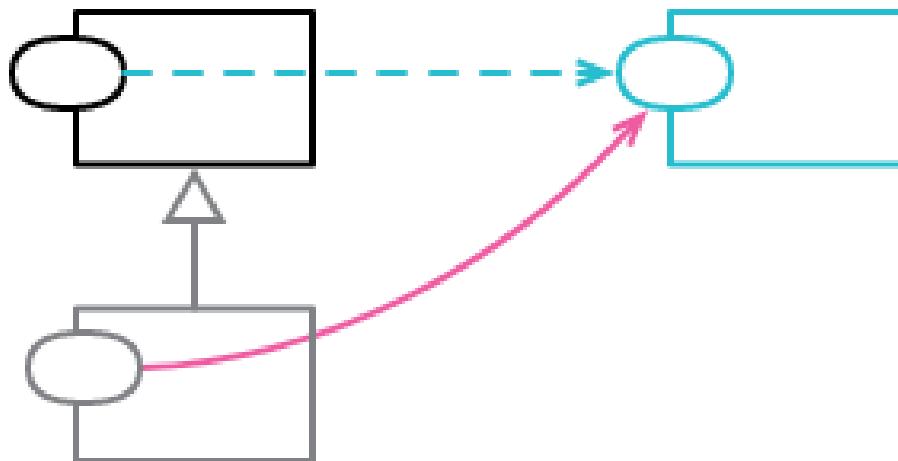
```
class List {...}  
class Stack extends List {...}
```



{}

```
class Stack {  
    constructor() {  
        this._storage = new List();  
    } }  
class List {...}
```

Replace Subclass with Delegate



Replace Subclass with Delegate

Lo natural es que el comportamiento cambie de categoría a categoría. Ponemos datos y comportamiento en la superclase y luego la sobreescrivimos en las subclases. Esto es simple de implementar y un mecanismo que venimos aplicando hace tiempo.

Replace Subclass with Delegate

Pero... que pasa si debo definir distintos comportamientos en base a los rangos de esas categorias.

Replace Subclass with Delegate

```
class Order {  
  
    get daysToShip() {  
  
        return this._warehouse.daysToShip;  
  
    } }  
  
{  
  
class PriorityOrder extends Order {  
  
    get daysToShip() {  
  
        return this._priorityPlan.daysToShip;  
  
    } }  
  
}
```

Replace Subclass with Delegate

```
class Order {  
  
    get daysToShip() {  
  
        return (this._priorityDelegate)  
            ? this._priorityDelegate.daysToShip  
            : this._warehouse.daysToShip;  
  
    } }  
  
class PriorityOrderDelegate {  
  
    get daysToShip() {  
  
        return this._priorityPlan.daysToShip  
  
    } }
```

{}

Composition

*Favor object composition
over class inheritance.*

The GOF design Patterns.

Inheritance versus Composition

Las dos técnicas más comunes para reutilizar código en programación orientada a objetos son

- la herencia de clases
- la composición de objetos.

Inheritance versus Composition

Como hemos explicado, la herencia de clases permite definir la implementación de una clase en términos de otra.

En la reutilización por subclases las partes internas de las clases principales suelen ser visibles para las subclases. A esto se lo suele denominar “reutilización de caja blanca”

Inheritance versus Composition

La **composición** de objetos es una alternativa a la herencia de clases. La nueva funcionalidad se obtiene ensamblando o “componiendo” objetos para obtener una funcionalidad más compleja.

Este estilo de reutilización se denomina reutilización de caja negra, porque no se ven detalles internos de los objetos.

Inheritance advantages

Se define estáticamente en tiempo de compilación, es fácil de usar, ya que es compatible directamente con el lenguaje de programación. También facilita la modificación de la implementación que se está reutilizando.

Inheritance disadvantages

No se puede cambiar las implementaciones heredadas de las clases principales en tiempo de ejecución, porque la herencia se define en tiempo de compilación.

Debido a que la herencia expone los detalles de la implementación de su padre a la subclase, a menudo se dice que "la herencia rompe el encapsulamiento".

Inheritance disadvantages

La implementación de una subclase está tan ligada a la implementación de su clase padre que cualquier cambio en la implementación del padre forzará a la subclase a cambiar.

Esta dependencia limita la flexibilidad y la reutilización. Una cura para esto es heredar solo de clases abstractas, ya que generalmente proporcionan poca o ninguna implementación.

Composition advantages

Mantener cada clase encapsulada y enfocada en una tarea. Las clases seguirán siendo pequeñas y será menos probable que se conviertan en monstruos ingobernables. El comportamiento del sistema dependerá de las interrelaciones de los objetos en lugar de estar definido en una clase.

Composition disadvantages

Idealmente, no deberíamos tener que crear nuevos componentes para lograr la reutilización. La funcionalidad que necesitamos la deberíamos obtener ensamblando componentes existentes, pero esto rara vez pasa, porque el conjunto de componentes disponibles nunca es lo suficientemente grande. La reutilización por herencia facilita la creación de nuevos componentes que se pueden componer con los antiguos. Por tanto, la herencia y la composición de objetos trabajan juntas.

Composition

Formas de composición:

- Interface
- Forwarding Methods/Delegate
- Mixins

Composition - Interface

Como hemos comentado previamente, JavaScript no implementa interfaces.

En POO1 vimos, cómo trabajan las interfaces (en Java). La interfaces no implementa código, recién en versiones más recientes de Java se puede definir código en las interfaces.

Composition - Forwarding Methods/Delegate

Vamos a definir propiedades y métodos que se encuentran definidos en otros objetos (forward). En los métodos no suele aplicarse otro comportamiento y solo se procede a ejecutar el método del otro objeto.

Este caso genera mucho acoplamiento debido a la dependencia que se genera entre los objetos.

Composition - Forwarding Methods/Delegate

Una forma más compleja es vía la implementación del patrón decorator. Donde podemos asignar una propiedad como el objeto que implementa la interfaz y ejecutarla garantizando de esta manera un entorno menos propenso a fallas... ya que la interfaz asegura los tipos de datos. (En Java).

Composition - Forwarding Methods/Delegate

Hagamos una composición....

JS Mixins

Object.assign()

Copia todas las propiedades enumerables de uno o más objetos fuente a un objeto destino. Devuelve el objeto destino.

JS Mixins

De este modo podemos definir objetos para cada comportamiento y luego componer un objeto nuevo en base a estos.

```
var perro = Object.assign(perro, saludar, caminar);
```

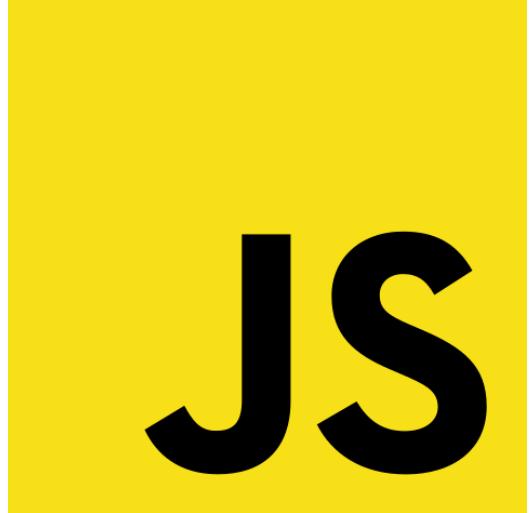
Referencia

- Fowler, M. (2019) Improving the Design of Existing Code - Second Edition - Addison-Wesley
- <https://refactoring.com/>
- <https://refactoring.guru/>
- Bobby Woolf (1996) The Null Object Pattern
- <https://alligator.io/js/class-composition/>
- <https://www.dofactory.com/javascript/design-patterns/>

Próxima Clase

Refactoring - Patterns

JavaScript



JS

Programacion Orientada a Objetos 2

Ing. Jose Rusca
Ing. Federico Stulich
Ing. Francisco De Grandis

Diseño Avanzado Orientado a Objetos



JS

Clase de hoy

- Metodologías ágiles
- Principios SOLID

Symptoms of Poor Design

- R rigidity—The design is hard to change.
- Fragility—The design is easy to break.
- Immobility—The design is hard to reuse.
- Viscosity—It is hard to do the right thing.
- Needless Complexity—Overdesign.
- Needless Repetition—Mouse abuse.
- Opacity—Disorganized expression.

Como Triunfar en el Desarrollo de Software

Debemos construir equipos de desarrollo colaborativos y auto-organizados.

Las organizaciones que provean estos tipos de equipos son las que van a tener éxito sobre las otras.

Pero, como vamos a hacer esto?

Metodologías Ágiles

Las metodologías ágiles son un conjunto de prácticas que se basan en el desarrollo iterativo e incremental con equipos multidisciplinarios y autogestionados.

El objetivo de estos métodos es evitar los procesos lentos y burocráticos implementados hasta el momento que solían fracasar en el desarrollo de software.

The Manifesto of the Agile Alliance

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Individuals and interactions over processes and tools.

Las personas son el ingrediente más importante para el éxito. Podemos tener los mejores procesos, pero si las personas no son buenas, las metodologías no van a servir de nada.

Un integrante bueno, no es solo un gran conocedor técnico, sino una persona que puede relacionarse correctamente con sus compañeros de equipo.

Working software over comprehensive documentation

El equipo necesita producir documentos legibles por humanos que describan el sistema y las decisiones de diseño.

Demasiada documentación, va a producir en el equipo una pérdida de tiempo y generar información que no va a ser necesaria.

Working software over comprehensive documentation

Formas para transferir información a los nuevos miembros del equipo:

- el código
- el equipo.

El código es la única fuente de información inequívoca, aunque puede ser difícil extraer el fundamento y su intención.

Los miembros del equipo tienen en la cabeza la hoja de ruta del sistema en constante cambio.

Working software over comprehensive documentation

Pero....

.... ¿Qué pasa si los miembros del equipo se van?

Customer collaboration over contract negotiation

Los modelos en los cuales se presenta una propuesta, es aceptada por el cliente y luego de un tiempo se entrega el software terminado, demostraron ser de poca calidad y poco exitosos.

Los proyectos exitosos están basados en la retroalimentación de los clientes de forma regular y frecuente.

Los mejores contratos son aquellos que definen la forma en que el equipo de desarrollo y el cliente trabajan de forma conjunta.

Responding to change over following a plan

La habilidad para responder a los cambios es lo que va a determinar el éxito del proyecto.

Nuestra planificación debe ser flexible para poder acaptarse.

¿Por que? Por que los negocios no son rígidos y esto genera cambios en las necesidades.

Responding to change over following a plan

Suena tentador crear un GANTT(o similar) y pegarlo en la pared para que todo el equipo pueda verlo y seguir la planificación esperada al principio del proyecto.

Lo que sucede es que la planificación se va degradando a medida que el equipo de desarrollo va conociendo el sistema y el cliente va evolucionando en sus necesidades.

¿Entonces no tengo que planificar?

Esto es un error muy común que se ve hoy en dia en muchos equipos de desarrollo, pensar que las metodologías ágiles es, vamos viendo y en función de eso corregimos.

Cuando en realidad se habla de una planificación orientada al cambio, donde el desarrollo no debe verse afectado por cambios en las necesidades del cliente.

Principios SOLID

SRP—The Single Responsibility Principle

OCP—The Open–Closed Principle.

LSP—The Liskov Substitution Principle.

ISP—The Interface Segregation Principle.

DIP—The Dependency Inversion Principle.

The Single Responsibility Principle

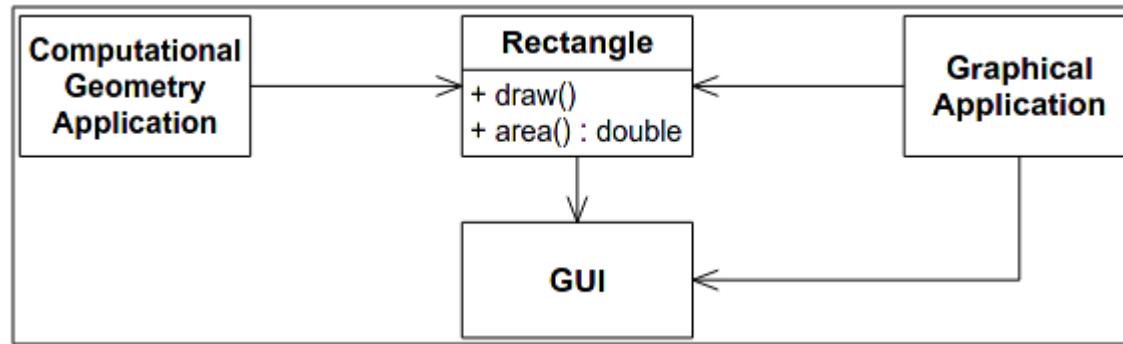
A class should have only one reason to change.

Si una clase posee más de una responsabilidad se vuelve acoplada.

Los cambios en una responsabilidad pueden afectar o inhibir la capacidad de la clase para cumplir con las demás.

The Single Responsibility Principle

Ejemplo



The Single Responsibility Principle

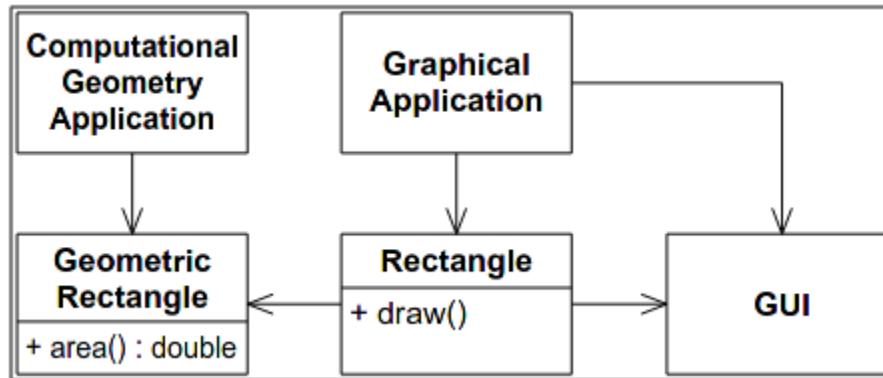
El rectángulo posee dos responsabilidades

- Dibujarse en la GUI
- Calcular su area

Lo mejor sería separar las responsabilidades en dos clases separadas.

The Single Responsibility Principle

Ejemplo



The Open—Closed Principle

Software entities should be open for extension, but closed for modification.*

Cuando un solo cambio da como resultado una cascada de cambios en los módulos dependientes, el diseño huele a rigidez. Por lo tanto debemos refactorizar el sistema para que más cambios de ese tipo no provoquen más modificaciones.

*classes, modules, functions, etc.

OCP - Open for extension

Esto significa que se puede ampliar el comportamiento del módulo a medida que cambian los requisitos de la aplicación. Pudiendo de esta manera satisfacer con nuevos comportamientos los cambios del negocio.

OCP - Closed for modification

Cuando extendemos el comportamiento de un módulo no debe dar como resultado cambios en el código fuente o el binario.

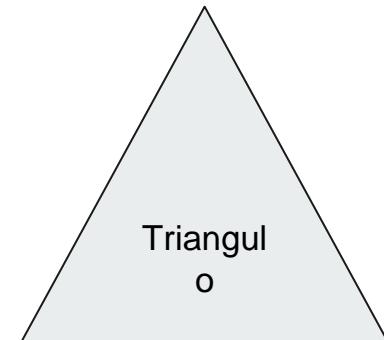
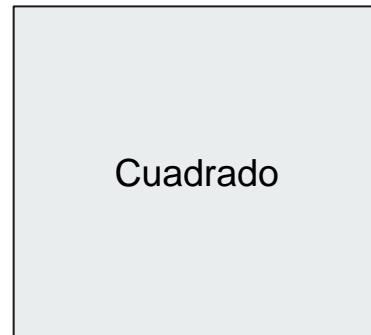
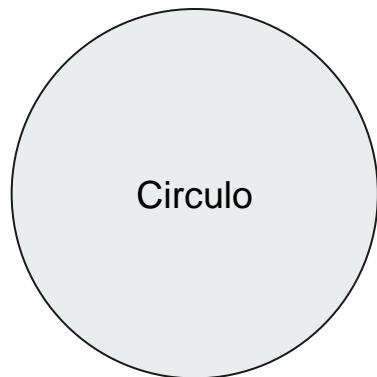
La versión binaria ejecutable del módulo, ya sea en una biblioteca enlazable, una DLL o un .jar de Java, permanece intacta.

OCP - Abstraction Is the Key

El módulo puede manipular la abstracción y puede cerrarse para modificaciones, ya que depende de una abstracción fija.

Sin embargo, el comportamiento se puede ampliar creando nuevos derivados de la abstracción.

OCP - Ejemplo DrawShape



The Liskov Substitution Principle

*Subtypes must be substitutable for
their base types.*

The Liskov Substitution Principle

“What is wanted here is something like the following substitution property: If for each object O_1 of type S there is an object O_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O_1 is substituted for O_2 then S is a subtype of T.”

Barbara Liskov

The Liskov Substitution Principle

La manera que tenemos para manejar OCP es por medio de la abstracción y el polimorfismo, de este modo nos damos cuenta que la herencia va a ser una herramienta fundamental para la resolución del problema.

Esto nos lleva a preguntarnos, de qué modo tenemos que utilizar correctamente la herencia y no abusar de ella.

LSP - Preguntas que surgen por OCP

¿Cuáles son las reglas de diseño que gobiernan este uso particular de la herencia?

¿Cuáles son las características del mejores jerarquías de herencia?

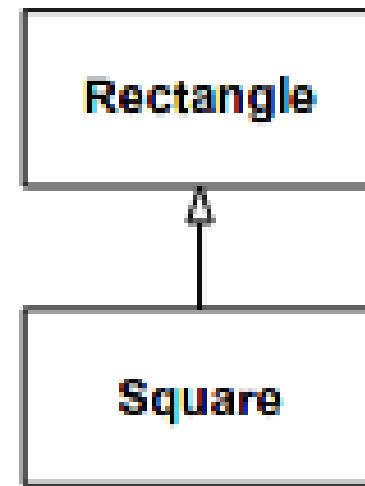
¿Cuáles son las trampas que harán que creemos jerarquías que no se ajusten a la OCP?

LSP - Ejemplo del Problema

Con propiedades Ancho y Alto, nos encontramos con el problema de cómo vamos a setear los valores en el cuadrado?

Se setea el Ancho?

Se setea el Alto?



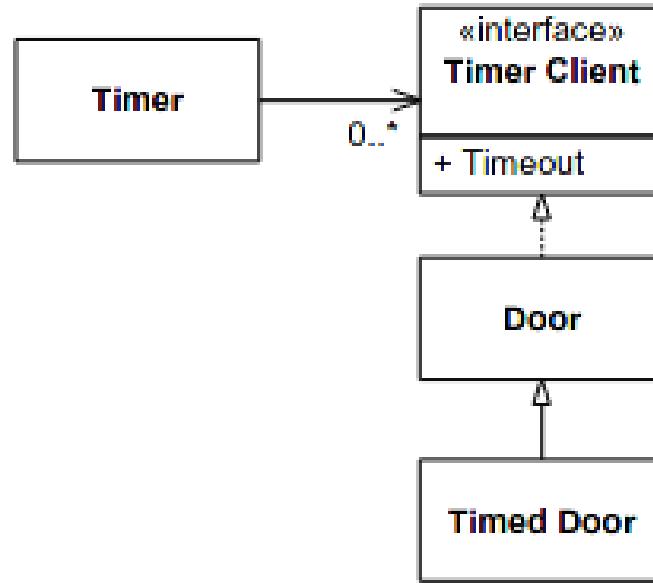
The Interface Segregation Principle

Clients should not be forced to depend on methods that they do not use.

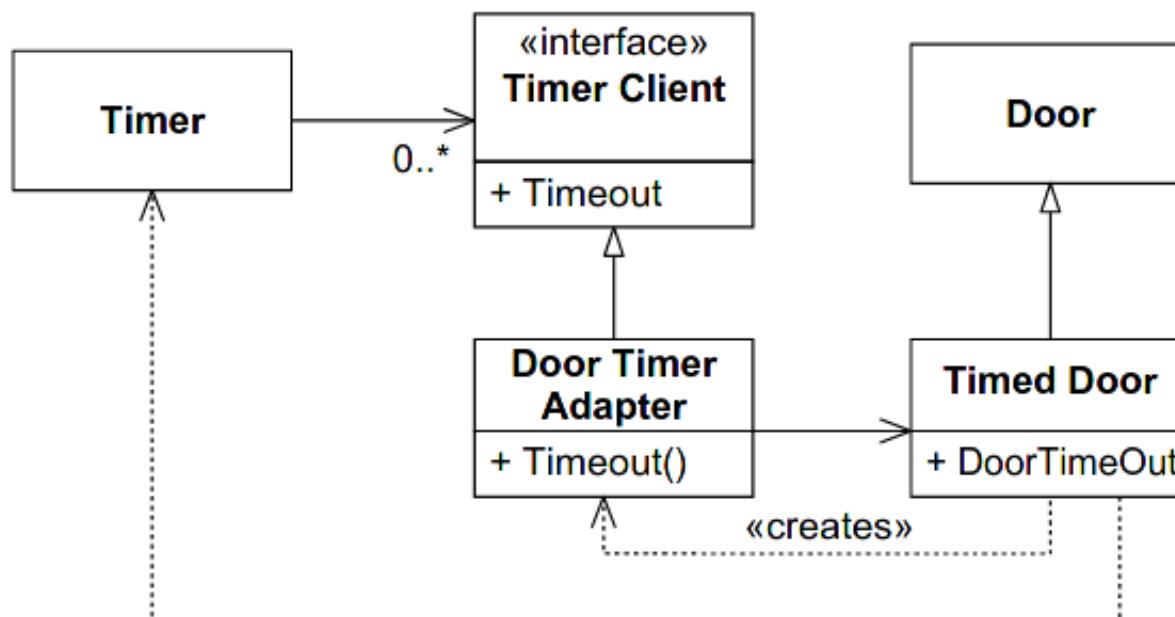
Este es el principio de la desventaja de las interfaces gordas. En pocas palabras la interfaz debe ser separada en grupos de métodos.

De este modo cada cliente puede utilizar un grupo de métodos/funciones mientras que otro cliente, otro grupo.

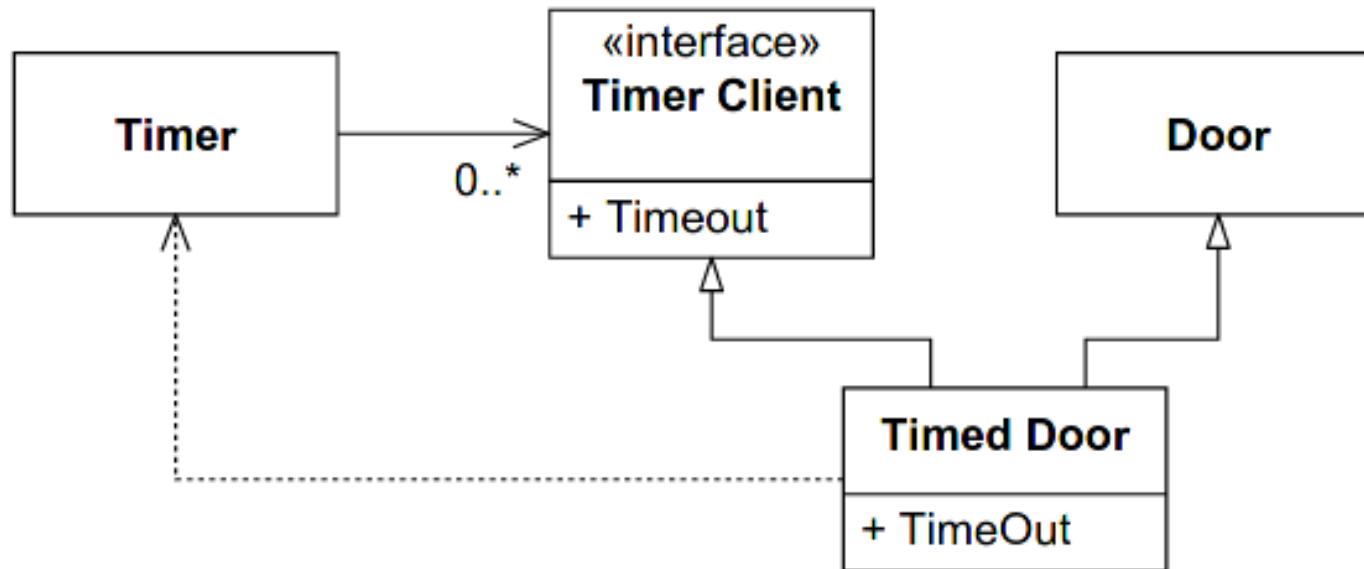
ISP - Interface Pollution



ISP - Separation through Delegation



ISP - Separation through Multiple Inheritance



The Dependency Inversion Principle

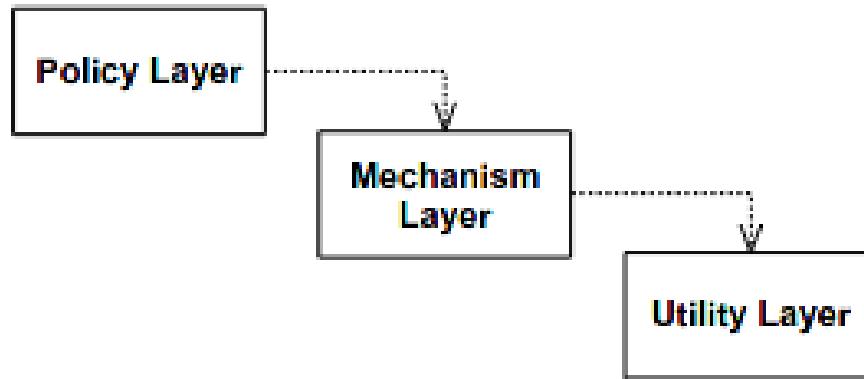
- A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*

- B. Abstractions should not depend on details. Details should depend on abstractions.*

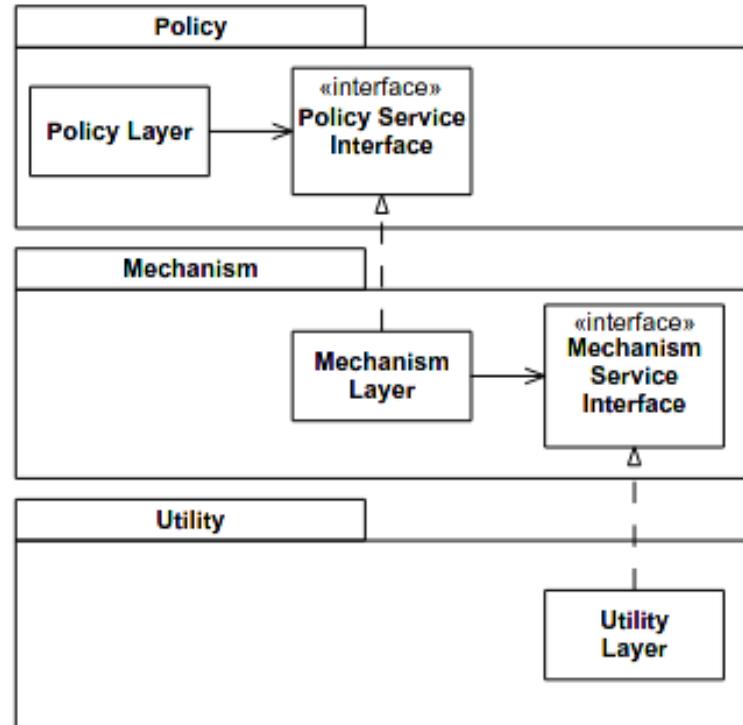
The Dependency Inversion Principle

Los métodos de desarrollo de software más tradicionales, tienden a crear estructuras de software en las que los módulos de alto nivel dependen de los módulos de bajo nivel y en las que la política depende de los detalles.

DIP - Ejemplo Capas



Ejemplo Capas Invertidas



The Dependency Inversion Principle

El principio de inversión de dependencia es el mecanismo fundamental de bajo nivel.

Su correcta aplicación es necesaria para la creación de frameworks reutilizables.

Es vital importancia para la construcción de un código resistente al cambio. Dado que las abstracciones y los detalles están aislados entre sí, el código es mucho más fácil de mantener.

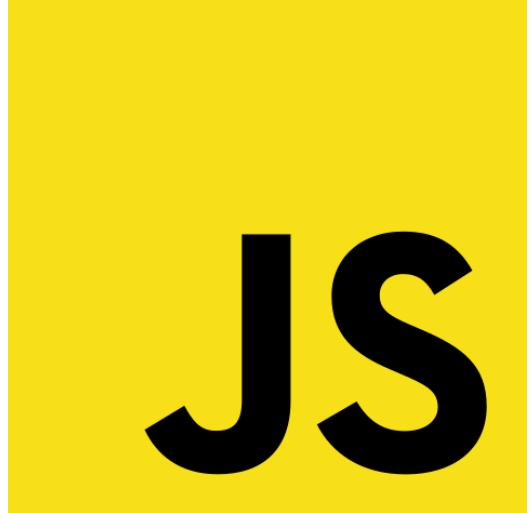
Referencia

- Martin, R. C. (2014) Agile Software Development, Principles, Patterns, and Practices - First Edition - Pearson Education Limited
- Fowler, M. (2019) Improving the Design of Existing Code - Second Edition - Addison-Wesley
- <https://refactoring.com/>
- <https://refactoring.guru/>
- <https://www.dofactory.com/javascript/design-patterns/>

Próxima Clase

Refactoring - Patterns

JavaScript



JS