



ACTIVIDAD CUATRO

COMPARATIVA DE

PATRONES

AITANA GAONA

ÍNDICE

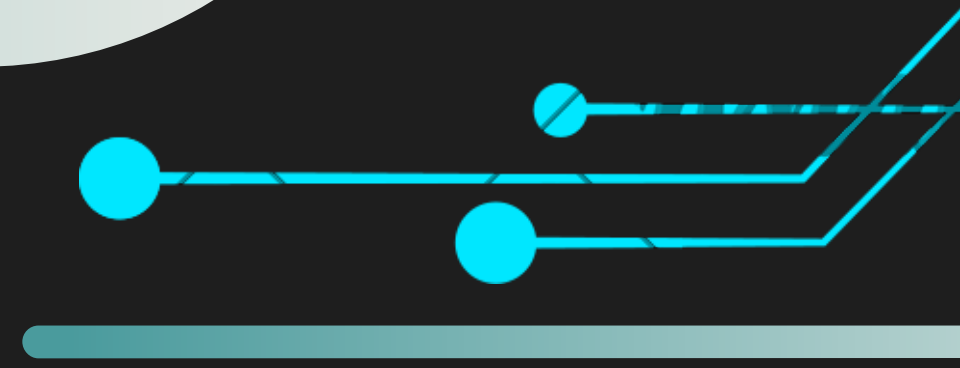
- 01** INTRODUCCIÓN
- 02** DIFERENCIAS
- 03** VENTAJAS
- 04** SITUACIONES DE USO





INTRODUCCIÓN

En este texto se va a realizar una comparativa de los diferentes patrones de diseño que se pueden aplicar en el lenguaje de programación JavaScript. Los patrones de diseño son soluciones reutilizables a problemas comunes que se presentan en el desarrollo de software.



CREACIONALES

Los patrones de diseño creacionales son aquellos que se encargan de la creación de objetos. Estos patrones tienen como objetivo abstraer la lógica de la instanciación y ocultar los detalles de cómo se crean y se inicializan los objetos. Algunos ejemplos de patrones de diseño creacionales son el patrón singleton, el patrón factory, el patrón builder y el patrón prototype.





ESTRUCTURALES

Los patrones de JavaScript estructurales son formas de organizar el código para facilitar su mantenimiento, reutilización y extensibilidad. Algunos ejemplos de estos patrones son el módulo, el revelador, el singleton y el facade. Estos patrones permiten encapsular datos y comportamientos, exponer solo las partes necesarias, evitar la contaminación del espacio de nombres global y simplificar la interfaz con otros componentes.



VENTAJAS



CREACIONALES

- FACILITAN EL MANTENIMIENTO Y LA EXTENSIBILIDAD DEL CÓDIGO, AL SEPARAR LA LÓGICA DE CREACIÓN DE LA LÓGICA DE USO.
- PROMUEVEN EL PRINCIPIO DE RESPONSABILIDAD ÚNICA, AL DELEGAR LA TAREA DE CREAR OBJETOS A UNA ENTIDAD ESPECÍFICA.
- PERMITEN VARIAR EL TIPO Y EL NÚMERO DE OBJETOS CREADOS EN TIEMPO DE EJECUCIÓN, SEGÚN LAS NECESIDADES DEL CONTEXTO.

ESTRUCTURALES

- MEJORAN LA LEGIBILIDAD Y LA MANTENIBILIDAD DEL CÓDIGO, AL HACERLO MÁS CLARO Y CONSISTENTE.
- FACILITAN LA REUTILIZACIÓN Y LA EXTENSIÓN DEL CÓDIGO, AL EVITAR LA DUPLICACIÓN Y EL ACOPLAMIENTO.
- PREVIENEN ERRORES Y BUGS, AL REDUCIR LA COMPLEJIDAD Y EL ALCANCE DE LAS VARIABLES.
- PROMUEVEN EL USO DE PARADIGMAS Y TÉCNICAS AVANZADAS, COMO LA PROGRAMACIÓN FUNCIONAL, ORIENTADA A OBJETOS O REACTIVA.

CREACIONALES SINGLETON

EL PATRÓN SINGLETON ES UNO DE LOS PATRONES DE DISEÑO MÁS UTILIZADOS EN LA INDUSTRIA DEL DESARROLLO DE SOFTWARE. EL PROBLEMA QUE PRETENDE RESOLVER ES MANTENER UNA ÚNICA INSTANCIA DE UNA CLASE. ESTO PUEDE RESULTAR ÚTIL CUANDO SE INSTANCIAN OBJETOS QUE CONSUMEN MUCHOS RECURSOS, COMO LOS MANEJADORES DE BASES DE DATOS.

JS

```
1 - function SingletonFoo() {
2
3     let fooInstance = null;
4
5     // For our reference, let's create a counter that will track the number of active instances
6     let count = 0;
7
8     function printCount() {
9         console.log("Number of instances: " + count);
10    }
11
12    function init() {
13        // For our reference, we'll increase the count by one whenever init() is called
14        count++;
15
16        // Do the initialization of the resource-intensive object here and return it
17        return {}
18    }
19
20    function createInstance() {
21        if (fooInstance == null) {
22            fooInstance = init();
23        }
24        return fooInstance;
25    }
26
27    function closeInstance() {
28        count--;
29        fooInstance = null;
30    }
31
32    return {
33        initialize: createInstance,
34        close: closeInstance,
35        printCount: printCount
36    }
37 }
38
39 let foo = SingletonFoo();
40
41 foo.printCount() // Prints 0
42 foo.initialize()
43 foo.printCount() // Prints 1
44 foo.initialize()
45 foo.printCount() // Still prints 1
46 foo.initialize()
47 foo.printCount() // Still 1
48 foo.close()
49 foo.printCount() // Prints 0
```


ESTRUCTURALES

ADAPTER

EL PATRÓN DE DISEÑO ADAPTER TE PROPORCIONA UNA ABSTRACCIÓN QUE SIRVE DE PUENTE ENTRE LOS MÉTODOS Y PROPIEDADES DE LA NUEVA CLASE Y LOS MÉTODOS Y PROPIEDADES DE LA ANTIGUA. TIENE LA MISMA INTERFAZ QUE LA CLASE ANTIGUA, PERO CONTIENE LÓGICA PARA ASIGNAR LOS MÉTODOS ANTIGUOS A LOS NUEVOS PARA EJECUTAR OPERACIONES SIMILARES. ESTO ES SIMILAR A CÓMO UNA TOMA DE CORRIENTE ACTÚA COMO ADAPTADOR ENTRE UN ENCHUFE DE ESTILO ESTADOUNIDENSE Y UN ENCHUFE DE ESTILO EUROPEO.

```
JS 2Punches change X
1 // Old bot
2 function Robot() {
3
4     this.walk = function(numberOfSteps) {
5         // code to make the robot walk
6         console.log("walked " + numberOfSteps + " steps")
7     }
8     this.sit = function() {
9         // code to make the robot sit
10        console.log("sit")
11    }
12 }
13 // New bot that does not have the walk function anymore
14 // but instead has functions to control each step independently
15 function AdvancedRobot(botName) {
16     // the new bot has a name as well
17     this.name = botName
18
19     this.sit = function() {
20         // code to make the robot sit
21         console.log("sit")
22     }
23
24     this.rightStepForward = function() {
25         // code to take 1 step from right leg forward
26         console.log("right step forward")
27     }
28
29     this.leftStepForward = function () {
30         // code to take 1 step from left leg forward
31         console.log("left step forward")
32     }
33 }
34 function RobotAdapter(botName) {
35     // No references to the old interface since that is usually
36     // phased out of development
37     const robot = new AdvancedRobot(botName)
38
39     // The adapter defines the walk function by using the
40     // two step controls. You now have room to choose which leg to begin/end with,
41     // and do something at each step.
42     this.walk = function(numberOfSteps) {
43         for (let i=0; i<numberOfSteps; i++) {
44
45             if (i % 2 === 0) {
46                 robot.rightStepForward()
47             } else {
48                 robot.leftStepForward()
49             }
50         }
51     }
52     this.sit = robot.sit
53 }
```