

# Tabla de contenido

Introducción	1.1
Conocimientos previos	1.1.1
Reseña histórica	1.1.2
Buenas prácticas	1.1.3
Principios de diseño	1.1.4
Patrones de diseño	1.2
Requisitos	1.2.1
Estructura	1.2.2
Categorías	1.2.3
Catálogo de patrones	1.2.4
Antipatrones	1.2.5
El lenguaje Javascript	1.3
Elección del lenguaje	1.3.1
Javascript y programación orientada a objetos	1.3.2
Patrones creacionales	1.4
Abstract Factory	1.4.1
Builder	1.4.2
Factory	1.4.3
Prototype	1.4.4
Singleton	1.4.5
Patrones estructurales	1.5
Adapter	1.5.1
Bridge	1.5.2
Composite	1.5.3
Decorator	1.5.4
Facade	1.5.5
Flyweight	1.5.6
Proxy	1.5.7
Patrones de comportamiento	1.6
Chain of responsibility	1.6.1
Command	1.6.2
Iterator	1.6.3
Mediator	1.6.4
Memento	1.6.5
Observer	1.6.6
Template method	1.6.7
Otros Patrones	1.7

Module Pattern	1.7.1
Revealing module	1.7.2

# Introducción

El diseño, desarrollo y mantenimiento de aplicaciones software puede ser un gran desafío intelectual, sobre todo en sistemas de alta complejidad o criticidad. Un mal diseño puede provocar:

- Sobrecostes.
- Ejecución del proyecto fuera de plazo.
- Software ineficiente.
- Software de mala calidad y frágil.
- Software que no cumple los requisitos.
- Software difícil de mantener.
- Catástrofes humanas<sup>1</sup>.

Por ello, en lugar de reinventar la rueda durante la etapa de diseño de cada sistema, es preferible reutilizar soluciones ampliamente conocidas y validadas, también llamadas "patrones de diseño" en el ámbito de la arquitectura de software. Éstos ofrecen "Una descripción detallada de una solución a un problema recurrente dentro de un contexto"<sup>2</sup>. De forma simplificada, pueden ser vistos como "recetas" que describen cómo resolver problemas en diferentes situaciones.

Este documento recopila y documenta, usando javascript como lenguaje de programación, la aplicación de los patrones de diseño más utilizados en la actualidad.

---

<sup>1</sup>. Relación de fallos software catastróficos ([https://en.wikipedia.org/wiki/List\\_of\\_software\\_bugs](https://en.wikipedia.org/wiki/List_of_software_bugs)) ↩

<sup>2</sup>. Alexander, Christopher; Ishikawa, Sara; Silverstein, Murray; Jacobson, Max; Fiksdahl-King, Ingrid; Angel, Shlomo (1977). A Pattern Language: Towns, Buildings, Construction. New York: Oxford University Press. ISBN 978-0-19-501919-3. ↩

## Conocimientos previos

Para la correcta comprensión de los conceptos que se introducen en este documento se requiere que el lector disponga de nociones de programación orientada a objetos, así como del lenguaje Javascript en su versión ECMAScript 5<sup>1</sup>.

De no ser así, se recomienda como paso previo la adquisición de dichos conocimientos. Internet ofrece numerosos recursos gratuitos al respecto, entre los que se encuentran:

- Programación orientada a objetos (Wikipedia): ([https://es.wikipedia.org/wiki/Programaci%C3%B3n\\_orientada\\_a\\_objetos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos))
- Javascript syntax (Wikipedia): ([https://en.wikipedia.org/wiki/JavaScript\\_syntax](https://en.wikipedia.org/wiki/JavaScript_syntax))
- Introducción a Javascript (librosweb): (<https://librosweb.es/libro/javascript/>)
- Guía de Javascript (Mozilla Developer Network - MDN): (<https://developer.mozilla.org/es/docs/Web/JavaScript/Guide>)
- Object-Oriented JavaScript. Create scalable, reusable high-quality JavaScript applications and libraries. Packt Publishing (2008). Stoyan Stefanov. ISBN 9781847194145

---

<sup>1</sup>. ECMAScript Language Specification (<http://www.ecma-international.org/ecma-262/5.1/>) ↔

## Reseña histórica

Las dificultades para escribir programas comprensibles, mantenibles y libres de defectos fueron expuestas por primera vez en 1968 en la "NATO Software Engineering Conference" bajo el patrocinio de la OTAN <sup>1</sup>, donde se acuñó el término "crisis del software" para referenciar la problemática identificada. Posteriormente, Edsger Dijkstra empleó el mismo término en su artículo "The Humble Programmer"<sup>2</sup>.

La Ingeniería de software surge como respuesta a esta situación, tratando de aplicar ingeniería al proceso de diseño, desarrollo, implementación, pruebas y mantenimiento del software. Dentro de este enfoque sistemático y disciplinado de desarrollo software aparece la aplicación de los patrones de diseño, aunque no obtienen una relevancia significativa hasta la publicación del libro "Design Patterns"<sup>3</sup>.

---

<sup>1</sup>. Artículos de la conferencia (<http://homepages.cs.ncl.ac.uk/brian.randell/NATO/>) ↩

<sup>2</sup>. Dijkstra, Edger. The Humble Programmer (<http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF>) ↩

<sup>3</sup>. Gamma, Erich; Helm, Richard; Johnson, Ralph; Missides, John (1994). Design Patterns:Elements of Reusable Object-Oriented software. Addison-Wesley. ISBN 0-201-63361-2. ↩

## Buenas prácticas de codificación

Como paso previo a la aplicación de patrones de diseño, el lector debería conocer algunas buenas prácticas genéricas de codificación. Aunque existe multitud de bibliografía al respecto, resalta especialmente el libro "Clean Code. A Handbook of Agile Software-Craftsmanship" de Robert C. Martin (editorial Prentice Hall). En él se definen algunas pautas que permiten generar código fuente de mayor calidad (lo que el autor denomina como "código limpio"). De aplicación inmediata son:

- Usar nombres expresivos y fáciles de leer en clases/funciones/variables
- Las funciones deben ser cortas, hacer una única cosa y mantenerse dentro del mismo nivel de abstracción. Además, no deben provocar efectos secundarios en la aplicación.
- Una función debe tener cuanto menos argumentos, mejor.
- Los comentarios no pueden maquillar un mal código. Tampoco debe contradecirlo.
- ES mejor provocar excepciones que devolver códigos de error

# Principios básicos de diseño

Asociados a la Ingeniería de software, aparecen una serie de principios básicos de diseño que favorecen el desarrollo de aplicaciones mantenibles y reutilizables. La lista es bastante extensa, por lo que sólo se mencionarán aquellos de conocimiento popular.

Si desea obtener más información, puede recurrir al listado proporcionado por la wikipedia en el enlace [List of software development philosophies](#)

## Principio KISS (Keep it simple, stupid!)

Establece que el diseño de un programa debe ser cuanto más sencillo, mejor, evitando complejidades innecesarias. <sup>1</sup>

## Principio DRY (Don't repeat yourself)

Propone no escribir código duplicado.

El código duplicado es propenso a errores y difícil de mantener, por lo que es mejor extraerlo y encapsularlo.

## Principio YAGNI (You ain't gonna need it)

Sugiere al programador no agregar funcionalidades a la aplicación/código a menos que estas sean realmente necesarias, evitando la tentación de escribir código sólo porque se prevé su uso.

## Principio de Hollywood (Don't Call Us, We'll Call You!)

Este principio hace referencia al eslogan de los directivos de Hollywood "No nos llames, nosotros te llamaremos". Pretende conseguir un menor acoplamiento entre los componentes de una aplicación a través de la inversión de control <sup>2</sup>

## Ley de Demeter (No hables con extraños)

También conocida como "Principio de Menor Conocimiento" también favorece el bajo acoplamiento entre cada unidad a través de las siguientes pautas:

- Cada unidad debe tener un conocimiento limitado sobre otras unidades
- Cada unidad debe hablar solo a sus amigos inmediatos.

## Principio SOLID

Introducido por Robert C. Martin, hace referencia a cinco principios de diseño <sup>3</sup>:

- (S)ingle responsibility principle: Una clase sólo debe tener una única responsabilidad. Además, dicha clase debe ser la única con esa responsabilidad
- (O)pen/close principle: Las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación. Esto es, una entidad no debería modificar su comportamiento, únicamente ampliarlo (a excepción de corrección de errores).
- (L)iskov substitution principle: Los objetos de un programa deberían ser reemplazables por instancias de sus subtipos. Como consecuencia, una clase derivada no debe modificar el comportamiento de la clase base.
- (I)nterface segregation principle: Muchas interfaces específicas son mejores que una de propósito general. Ninguna clase debería estar forzada a implementar un método que no requiere.

- (D)ependency inversion principle: Una clase debería depender de abstracciones, no de implementaciones.
- 

1. Ficha en FOLDOC (<http://foldoc.org/KISS%20Principle>) ↔

2. Inversión de control. Martin Fowler (<http://martinfowler.com/bliki/InversionOfControl.html>) ↔

3. Artículo de Robert C. Martin "UncleBob" (<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>) ↔

# Patrones de diseño

La RAE define el término patrón como "modelo que sirve de muestra para sacar otra cosa igual". En ingeniería de software, los patrones de diseño proporcionan:

- Buenas soluciones de diseño validas para un problema recurrente en un contexto determinado.
- Código ampliamente verificado, reutilizable y mantenible.
- Un vocabulario común entre diseñadores software.

Debido a la importancia y trascendencia del libro "Design Patterns", cuyos autores son conocidos como GoF (Gang of Four), en los siguientes subapartados se emplearán las definiciones incluidas en el citado documento.

# Requisitos

Para que una solución represente un patrón de diseño debe cumplir una serie de requisitos, a saber:

- Debe ser capaz de resolver un problema particular.
- El problema debe ser recurrente.
- La solución a este problema no debe ser obvia.
- La solución debe describir una relación entre los elementos que la componen.
- El concepto descrito en la solución debe haber sido probado, validado y aceptado en la comunidad.

# Estructura

Según GoF un patrón de diseño se describe a través de una plantilla que debe contener los siguientes elementos:

- Nombre del patrón y clasificación: Nombre estándar.
- Propósito: Frase breve que responde a las cuestiones ¿Qué hace este patrón de diseño? ¿En qué se basa? ¿Cuál es el problema concreto de diseño que resuelve?
- Alias: Nombres alternativos.
- Motivación: Escenario que ilustra el problema de diseño y cómo el patrón resuelve el problema.
- Aplicabilidad: Situaciones en las que se puede aplicar el patrón de diseño. Ejemplos de malos diseños que el patrón puede resolver. Cómo reconocer dichas situaciones.
- Estructura: Representación gráfica de las clases que emplea el patrón (OMT/UML)
- Participantes: Clases y objetos participantes en el patrón, junto con sus responsabilidades
- Colaboraciones: Colaboración entre participantes para llevar a cabo su responsabilidad
- Consecuencias: Descripción de consecución de objetivos. Ventajas e inconvenientes de usar el patrón.
- Implementación: Dificultades, trucos y técnicas que se deben tener en cuenta a la hora de aplicar el patrón. Cuestiones específicas del lenguaje.
- Código de ejemplo: Fragmentos de código que muestran implementaciones del patrón.
- Usos conocidos: Ejemplos de uso en sistemas reales.
- Patrones relacionados: Enumeración de otros patrones estrechamente relacionados. Dependencias y diferencias.

# Categorías

GoF cataloga los patrones de diseños de acuerdo a dos criterios diferentes:

- Propósito: Refleja qué hace un patrón:
  - De creación: Relacionados con los proceso de creación de objetos (cómo se obtienen y construyen)
  - Estructural: Tratan con la composición entre clases u objetos (cómo se acoplan, relacionan, componen y comunican un conjunto de clases/objetos)
  - De comportamiento: Caracterizan el modo en que las clases y objetos interactúan y se reparten las responsabilidades.
- Ámbito: Especifica si el patrón se aplica principalmente a la clase o a objetos.
  - De clases: se ocupan de las relaciones entre las clases y sus subclases. éstas relaciones se establecen a través de la herencia, de modo que son relaciones estáticas.
  - De objetos: Tratan con las relaciones entre objetos, que pueden cambiarse en tiempo de ejecución.

## Catálogo de patrones

El siguiente cuadro resume los principales patrones agrupados según las categorías del apartado anterior.

### Creacionales. Basados en el concepto de creación de objetos

<b>Ámbito de clases</b>	
Factory Method	Define una interfaz para crear un objeto, pero deja que sean las subclasses quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclasses la creación de objetos.
<b>Ámbito de Objetos</b>	
Abstract Factory	Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.
Builder	Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.
Prototype	Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando de este prototipo.
Singleton	Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.

### Estructurales. Basados en la idea de la construcción de bloques de objetos

<b>Ámbito de clases</b>	
Adapter	Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permite que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.
<b>Ámbito de Objetos</b>	
Bridge	Desacopla una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.
Composite	Combina objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.
Decorator	Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.
Facade	Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.
Flyweight	Usa el compartimento para permitir un gran número de objetos de grano fino de forma eficiente.
Proxy	Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.

### De comportamiento. Basados en cómo interaccionan entre sí los objetos

<b>Ámbito de clases</b>	
Interpreter	Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación par interpretar sentencias del lenguaje.
Template Method	Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.
<b>Ámbito de Objetos</b>	
Chain of Responsibility	Evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que ésta sea tratada por algún objeto.
Command	Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer la operación.
Iterator	Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.
Mediator	Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.
Memento	Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.
Observer	Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifica y se actualizan automáticamente todos los objetos que dependen de él.
State	Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.
Strategy	Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varía independientemente de los clientes que lo usan.
Visitor	Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera

# Antipatrones

Así como los patrones de diseño representan buenas prácticas, los antipatrones simbolizan malas práxis. Agrupan aquellos patrones de diseño que provocan inexorablemente una mala solución al problema que tratan de resolver. Conocer los antipatrones de diseño es un factor clave para detectar y evitar soluciones desaconsejadas.

Aunque quedan fuera del alcance de este documento, se recomienda la lectura del siguiente material:

- Catálogo de Antipatrones: (<http://c2.com/cgi/wiki?AntiPatternsCatalog>)
- Listado de Antipatrones más conocidos (Wikipedia): (<https://en.wikipedia.org/wiki/Anti-pattern>)
- [Browm et al 1998] Anti Patterns. Refactoring software, Architectures and Projects in Crisis. W.J. Brown, R.C. Malveau, H.W. "Skip" McCormick III, T. J. Mowbray. Wiley, 1998.

# El lenguaje Javascript

Aunque el nombre del lenguaje sugiere similitudes con Java, poco o nada tiene que ver con éste. Javascript es un lenguaje de alto nivel, dinámico, no tipado, interpretado, basado en prototipos con funciones como ciudadanos de primera clase<sup>1</sup> (lo cual permite, además de la programación imperativa, el estilo de programación funcional).

---

<sup>1</sup>. First-class functions Wikipedia ([https://en.wikipedia.org/wiki/First-class\\_function](https://en.wikipedia.org/wiki/First-class_function)) ↩

## Elección del lenguaje

Existen muchos argumentos para elegir Javascript como lenguaje de programación, pero tal vez el más importante de todos sea su alcance. Junto con HTML y CSS, es una de las tres tecnologías básicas del World Wide Web y está soportado por casi todos los navegadores del mundo sin necesidad de plug-ins, por lo que desarrollar aplicaciones en Javascript facilita su distribución en cualquier ordenador o smartphone.

Aello se le suma la gran expansión que el lenguaje ha sufrido en el lado servidor, gracias a Node.js<sup>1</sup> y al servidor No-SQL MongoDB<sup>2</sup>, permitiendo el desarrollo de aplicaciones cliente/servidor empleando un único lenguaje común: Javascript. De hecho, existe todo un conjunto de de sistemas (conocido como MEAN Stack<sup>3</sup>) que facilitan el desarrollo integral de aplicaciones cliente/servidor.

---

1. Página oficial de NodeJs (<https://nodejs.org/en/>) ↵

2. Página oficial de MongoDB (<https://www.mongodb.com/es>) ↵

3. MEAN Wikipedia (<https://es.wikipedia.org/wiki/MEAN>) ↵

# Javascript y programación orientada a objetos

Los patrones de diseños descritos en el libro "Design Pattern" sustentan su definición en la programación orientada a objetos usando el modelo basado en **clases**. Javascript, en cambio, es un lenguaje de programación orientada a objetos basado en **prototipos**<sup>1</sup> en el que no existen el concepto de clases, por lo que la implementación de los diferentes patrones de diseño requiere de una adaptación a las características propias del lenguaje.

No obstante, Javascript permite emular la mayoría de las características propias de una clase. Gracias a la noción de prototipos y al azúcar sintáctico<sup>2</sup> que proporciona el lenguaje, es posible implementar los conceptos de clase, constructor, propiedades, métodos, herencia, polimorfismo, encapsulamiento y abstracción.

Los siguientes subapartados, extraídos de MDN<sup>3</sup> muestran una metodología para emplear los conceptos anteriores en Javascript, aunque existen múltiples alternativas.

## La clase

Javascript permite usar funciones para definir una clase. Por ejemplo:

```
function Persona() { }
```

Puede encontrar métodos alternativos para describir una clase en el blog de Stoyan Stefanov (<http://www.phpied.com/3-ways-to-define-a-javascript-class/>)

## El Objeto

Para crear un objeto concreto a partir de la clase persona, se utilizará el operador "new":

```
function Persona() { }  
  
var persona1= new Persona();
```

## El constructor

El constructor es el método de la clase que se llama durante la instanciación del objeto. En javascript, la función Persona() se emplea a la vez como definición de la clase y constructor.

## Las propiedades del objeto

Las propiedades son variables contenidas en la clase. Cada instancia de la clase (objeto) tiene dichas propiedades. La palabra reservada "this" permite crear variables propias de la clase. El acceso externo a propiedades de la clase se realiza a través del operador punto '.'.

```
function Persona(nombre){  
    this.nombre = nombre; // Propiedad 'nombre'  
}  
  
var persona1 = new Persona("Rocío"); /// Nuevo objeto persona1  
console.log(persona1.nombre); // Muestra 'Rocío'
```

## Los métodos

Los métodos se definen de la misma forma similar a las propiedades. El acceso externo a un método de la clase se realiza a través del operador '.', pero agregando () al final del nombre del método para ejecutarlo.

```
// Definición de la clase Persona
function Persona(nombre){
  // Definición de la propiedad nombre
  this.nombre = nombre;
}

// Definición del método saludar()
Persona.prototype.saludar = function() {
  console.log('Hola, soy ' + this.nombre);
};

// Creación del objeto persona1
var persona1 = new Persona("Rocío");

// Llamada al método saludar() del objeto persona1
persona1.saludar(); // Muestra "Hola, soy Rocío" en la consola
```

## Herencia

Mediante la herencia se consigue crear una subclase como una versión especializada de otra clase base. En Javascript la herencia se logra mediante la asignación de una instancia de la clase primaria a la clase secundaria y la posterior especialización.

```

// Definimos la clase padre 'Persona'
function Persona(nombre) {
  // Propiedad de la clase padre
  this.nombre = nombre;
}

// Método de la clase padre
Persona.prototype.saludar = function() {
  console.log("Hola, Soy" + this.nombre);
};

// Definimos la clase hija 'Estudiante'
function Estudiante(nombre, asignatura) {
  // Llamamos al constructor de la clase padre
  Persona.call(this, nombre);

  // Propiedad de la clase hija
  this.asignatura = asignatura;
};

// Hacemos que la clase Estudiante herede de la clase Persona
// https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/create
Estudiante.prototype = Object.create(Persona.prototype);

// Establecemos la propiedad "constructor" para referenciar a Estudiante
Estudiante.prototype.constructor = Estudiante;

// Agregamos el método caminar() a la clase hija
Estudiante.prototype.caminar = function() {
  console.log("¡Estoy caminando!");
};

// Ejemplos de uso
var estudiante1 = new Estudiante("Rocío", "Filosofía");
estudiante1.saludar(); // muestra "Hola, Soy Rocío."
estudiante1.caminar(); // muestra "¡Estoy caminando!"

// Comprobamos que la herencia funciona correctamente
console.log(estudiante1 instanceof Persona); // devuelve true
console.log(estudiante1 instanceof Estudiante); // devuelve true

```

Si necesita más información al respecto, Douglas Crockford ofrece varios métodos alternativos para implementar la herencia en Javascript:

- Herencia clásica: (<http://javascript.crockford.com/inheritance.html>)
- Herencia basada en prototipos: (<http://javascript.crockford.com/prototypal.html>)

## Polimorfismo

Las diferentes clases hijas pueden definir métodos con el mismo nombre que la clase padre. Así, en el ejemplo anterior de las clases Persona/Estudiante se puede, por ejemplo, modificar el método saludar de la clase hija para que incluya información de la asignatura:

```

// Definimos la clase padre 'Persona'
function Persona(nombre) {
  // Propiedad de la clase padre
  this.nombre = nombre;
}

// Método de la clase padre
Persona.prototype.saludar = function() {
  console.log("Hola, Soy" + this.nombre);
};

// Definimos la clase hija 'Estudiante'
function Estudiante(nombre, asignatura) {

  // Llamamos al constructor de la clase padre
  Persona.call(this, nombre);

  // Propiedad de la clase hija
  this.asignatura = asignatura;
};

// Hacemos que la clase Estudiante herede de la clase Persona
Estudiante.prototype = Object.create(Persona.prototype);
Estudiante.prototype.constructor = Estudiante;

// Polimorfismo
Estudiante.prototype.saludar = function () {
  console.log("Hola, soy " + this.nombre + ". Estudio " + this.asignatura + ".");
};

var estudiante = new Estudiante('Manuel', 'Ingeniería de Software');
estudiante.saludar();

```

## Encapsulación

El encapsulamiento hace referencia a la agrupación de los elementos de una misma entidad (métodos y propiedades). En el ejemplo anterior, las clases Persona y Estudiante agrupan sus propias funciones y atributos a través de la función constructora y de sus prototipos, consiguiéndose el principio de encapsulamiento.

El término también puede hacer referencia a la restricción de acceso a estos métodos y propiedades fuera del objeto. Una forma de ocultar el estado (propiedades) y métodos es utilizando funciones y variables anidadas. Sirva el siguiente código a modo de ejemplo:

```

function Persona(nombre) {
  this.nombre = nombre // Propiedad pública
  var dni = null; // Propiedad privada

  // Propiedad privada que permite a métodos privados acceder a variables públicas
  var that = this;

  // Método público
  this.establecerDNI = function (nuevoDNI) {
    if( validarDNI() ) {
      dni = nuevoDNI;
      console.log('DNI de ' + that.nombre + ' validado y establecido');
    } else {
      console.log('DNI inválido');
    }
  };

  // Método privado. Devuelve true si el DNI es válido
  function validarDNI(dni) {
    // ...
    return true;
  }
}

var persona1 = new Persona("Rocío"); // Nuevo objeto persona1
persona1.establecerDNI(1234567); // Muestra "DNI de Rocío válido y establecido"
persona1.dni; // muestra "undefined"

```

## Abstracción

Mediante la abstracción se modela un elemento generalizándolo, esto es, eliminando particularidades y detalles complejos del mismo. El modelo puede, posteriormente, acercarse a la realidad a través de una especialización de dicha abstracción.

Javascript permite la especialización por herencia y por composición.

---

<sup>1</sup>. Artículo Wikipedia - Programación basada en prototipos

([https://es.wikipedia.org/wiki/Programaci%C3%B3n\\_basada\\_en\\_prototipos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_basada_en_prototipos)) ↵

<sup>2</sup>. Añadidos a la sintaxis de un lenguaje que facilitan expresar algunas construcciones de una forma más clara o concisa. Wikipedia ([https://es.wikipedia.org/wiki/Az%C3%BAcar\\_sint%C3%A1ctico](https://es.wikipedia.org/wiki/Az%C3%BAcar_sint%C3%A1ctico)) ↵

<sup>3</sup>. Introducción a Javascript orientado a objetos

([https://developer.mozilla.org/es/docs/Web/JavaScript/Introducci%C3%B3n\\_a\\_JavaScript\\_orientado\\_a\\_objetos](https://developer.mozilla.org/es/docs/Web/JavaScript/Introducci%C3%B3n_a_JavaScript_orientado_a_objetos)) ↵

# Abstract Factory

## Propósito

Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas

Se encuentra íntimamente relacionado con el patrón "Factory Method", ya que encapsula a un grupo de factories concretas<sup>1</sup> con un objetivo común.

Mientras que el patrón "Factory Method" usa subclases para crear instancias concretas de cada tipo de objetos, "Abstract Factory" define en una única clase la creación de objetos de diferente tipo (aunque pertenecientes a la misma familia).

## Implementación

El objeto "Creador" del código que se expone permite obtener diferentes "Productos" relacionados (cuadrado, círculo) usando una misma interfaz:

```

// Producto abstracto
function Forma(opciones) {
  this.dibujar = function () {
    throw 'Método dibujar no implementado';
  };
}

// Producto Concreto: Circulo
function Circulo(opciones) {
  // Si no existe opciones.radio, usar el valor 0
  this.radio = opciones.radio || 0;
}

// Circulo es subclase de Forma
Circulo.prototype = Object.create(Forma.prototype);
Circulo.prototype.constructor = Circulo;

// Producto Concreto: Cuadrado
function Cuadrado(opciones) {
  this.lado = opciones.lado || 0;
}

// Circulo es subclase de Forma
Cuadrado.prototype = Object.create(Forma.prototype);
Cuadrado.prototype.constructor = Cuadrado;

// Creador abstracto
function FabricaGenerica() {
  this.crearForma = function (tipo, opciones) {
    throw 'Función no implementada';
  };
}

// Creador Concreto: FabricaDeFormas
function FabricaDeFormas() {
  var formaPorDefecto = Circulo;

  this.crearForma = function (tipo, opciones) {
    var formaAConstruir = formaPorDefecto;
    var opcionesRecibidas = opciones || 0;
    // Seleccionar subclase a construir
    switch (tipo) {
      case 'Circulo':
        formaAConstruir = Circulo;
        break;
      case 'Cuadrado':
        formaAConstruir = Cuadrado;
        break;
    }

    return new formaAConstruir(opcionesRecibidas);
  };
}

// Nueva fábrica
var fabrica = new FabricaDeFormas();

// Solicitud de creación de objetos
var circulo1 = fabrica.crearForma('Circulo', { radio: 5 });
var circulo2 = fabrica.crearForma(); // Devuelve un circulo de radio 0;
var cuadrado = fabrica.crearForma('Cuadrado', { lado: 3 });

// Comprobación sobre objetos obtenidos
console.log(circulo1 instanceof Circulo); // Devuelve True
console.log(circulo2 instanceof Circulo); // Devuelve True
console.log(cuadrado instanceof Cuadrado); // Devuelve True

```

La clase "FabricaDeFormas" se ha implementado fijando los tipos de objetos a construir. Un diseño alternativo, más flexible, permite registrar los tipos de objeto a crear siempre que éstos dispongan del mismo patrón de creación:

```
// Creador Concreto: FabricaDeFormas
function FabricaDeFormas() {
  var formas = {};
  this.registrarForma = function (tipo, constructor) {
    formas[tipo] = constructor;
  };
  this.crearForma = function (tipo, opciones) {
    var parametros = opciones || {};
    var constructor = formas[tipo];
    return (constructor ? new constructor(parametros) : null);
  };
}

// Ejemplo de uso: Nueva fábrica
var fabrica = new FabricaDeFormas();

// Registro de objetos a construir
fabrica.registrarForma('Circulo', Circulo);
fabrica.registrarForma('Cuadrado', Cuadrado);

// Construcción de objetos
var circulo = fabrica.crearForma('Circulo', { radio: 3 });
var cuadrado = fabrica.crearForma('Cuadrado', { lado: 5 });
```

---

1. el concepto de Factoría Concreta se describe en el apartado [Factory Method](#) ←

# Builder

## Propósito

Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción puede crear diferentes representaciones.

Gracias al patrón Constructor se divide la creación de un objeto complejo en pasos simples e independientes que definen cada parte del objeto.

Una clase, denominada Director, podrá crear diferentes objetos complejos (Productos) a través de cualquier clase que implemente los pasos necesarios (Constructor).

Se asume que el protocolo de construcción del objeto es conocido por el Constructor y el Director.

Gracias al patrón Builder se solventan los problemas relacionados con el antipatrón "telescoping constructor", el cual aparece cuando se usa un constructor con todos los parámetros del objeto como argumento. Al usar este anti-patrón, cuando el objeto a crear va aumentando en complejidad (tiene más parámetros), se requiere modificar el constructor existente o aumentar mediante polimorfismo el número de constructores. Como consecuencia, la aplicación debe adaptarse para usar estos nuevos constructores en todos los puntos donde se construya el objeto. En cambio, al usar el patrón Builder se centraliza y simplifica la creación del objeto en cuestión, permitiendo diferentes construcciones a través de Builders personalizados.

## Implementación

Supongamos que un ordenador/tablet queda representado por las siguientes características:

- CPU.
- Memoria Ram.
- Capacidad de almacenamiento.

El siguiente ejemplo muestra una posible implementación de un objeto "Ordenador" y un objeto "Tablet" empleando el patrón Builder.

```

// Builder Concreto: Ordenador
function EnsambladorOrdenador() {
    var producto = Object.create(null);
    producto.tipo = 'Ordenador';

    this.cpu = function(cpu) { producto.cpu = cpu; };
    this.ram = function(ram) { producto.ram = ram; };
    this.capacidad= function(capacidad) { producto.capacidad = capacidad; };
    this.extras = function() {}; // Extras no configurables

    this.ensamblar = function() { return producto; };
}

// Builder Concreto: Tablet
function EnsambladorTablet() {
    var producto = Object.create(null);
    producto.tipo = 'Tablet';

    this.cpu = function(cpu) { producto.cpu = cpu; };
    this.ram = function(ram) { producto.ram = ram; };
    this.capacidad = function(capacidad) { producto.capacidad = capacidad; };
    this.extras = function() { producto.pantalla = '10pulgadas'; };

    this.ensamblar = function() {
        producto.pantalla = '10inches';
        return producto;
    };
}

// Director
function Taller() {
    this.ensamblar = function(ensamblador) {
        ensamblador.cpu('i5');
        ensamblador.ram('8Gb');
        ensamblador.capacidad('1Tb');
        ensamblador.extras();

        return ensamblador.ensamblar();
    }
}

// Builders concretos
var ensambladorOrdenador = new EnsambladorOrdenador();
var ensambladorTablet = new EnsambladorTablet();

// Director que utilizará el builder concreto
var taller = new Taller();

// Cliente
var ordenador = taller.ensamblar(ensambladorOrdenador); // producto obtenido: ordenador
var tablet = taller.ensamblar(ensambladorTablet); // producto obtenido: tablet

```

Gracias a este diseño el cliente obtiene diferentes productos (tablet, ordenador) empleando una misma interfaz Director (Taller). De la misma forma, a través de los Builders concretos la clase Director puede generar productos diferentes siguiendo siempre la misma secuencia de acciones.

Si se requiere un producto diferente, basta con definir y emplear un nuevo Builder, siendo éste cambio transparente en la interfaz Cliente-Director.

# Factory Method

## Propósito

Define una interfaz para crear un objeto, pero deja que sean las subclasses quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclasses la creación de objetos.

En vez de usar directamente el operador "New" o la herencia a través de prototipos usando "Object.create()", Factory Method desacopla la creación del nuevo objeto (llamado Producto) del componente que lo solicita, simplificando y encapsulando los detalles de instanciación en el objeto "Creador". A través de esta interfaz ("Creador") se definen los métodos que las fábricas concretas emplearán para crear nuevos objetos.

A diferencia de Abstract factory, cada fábrica concreta sólo produce un tipo de objetos, por lo que la única forma de crear objetos diferentes es a través de una fábrica concreta distinta que implemente la misma interfaz.

## Implementación

Haciendo uso del patrón Factory Method es posible obtener diferentes objetos (Cuadrado, Circulo) a través de una misma interfaz "FabricaDeFormas":

```

// Producto abstracto
function Forma ( opciones ) {}

// Producto Concreto: Circulo
function Circulo( opciones ) {
  // Si no existe opciones.radio, usar el valor 0
  this.radio = opciones.radio || 0;
}

// Producto Concreto: Cuadrado
function Cuadrado( opciones ) {
  this.lado = opciones.lado || 0;
}

// Creador abstracto
function FabricaGenerica() {
  this.crearForma = function ( tipo, opciones ) {
    throw 'Función no implementada';
  };
}

// Creador Concreto: FabricaDeCirculos
function FabricaDeCirculos() {
  this.crearForma = function ( opciones ) {
    var opcionesRecibidas = opciones || {};
    return new Circulo( opcionesRecibidas );
  };
}

// Creador Concreto: FabricaDeCuadrados
function FabricaDeCuadrados() {
  this.crearForma = function ( opciones ) {
    var opcionesRecibidas = opciones || {};
    return new Cuadrado( opcionesRecibidas );
  };
}

// Creación de fábricas concretas
var fabricaDeCirculos = new FabricaDeCirculos();
var fabricaDeCuadrados = new FabricaDeCuadrados();

// Uso de fábricas
var circulo1 = fabricaDeCirculos.crearForma( { radio: 5 } );
var circulo2 = fabricaDeCirculos.crearForma(); // Devuelve un circulo de radio 0;
var cuadrado = fabricaDeCuadrados.crearForma( { lado: 3 } );

// Comprobación de objetos creados
console.log(circulo1 instanceof Circulo); // Devuelve True
console.log(circulo2 instanceof Circulo); // Devuelve True
console.log(cuadrado instanceof Cuadrado); // Devuelve True

```

Si surge la necesidad de ampliar la fabrica para que contemple otros tipos de formas, basta con crear una subclase de FabricaDeFormas sobrescribiendo el método "crearForma", o bien crear otra nueva fábrica que implemente la interfaz FabricaGenerica.

Observe que javascript no posee el concepto de "interfaz", por lo que las interfaces "FabricaGenerica" y "Forma" se han definido únicamente por convenio; nada impide a las clases concretas no implementar su interfaz correctamente. No obstante, existen mecanismos en el lenguaje para emular el uso de interfaces en javascript<sup>1</sup>

---

<sup>1</sup>. Pro Javascript Design Patterns. Ros Harnes and Dustin Diaz. Apresss. 2008. Capítulo 2 ↩

# Prototype

## Propósito

Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando este prototipo

En vez de crear nuevos objetos a partir de su definición mediante clases y herencia, el patrón clona en el nuevo objeto las propiedades y métodos de otro ya existente. Gracias a Prototype se disminuye la herencia y el número de clases, obteniendo un sistema más simple, flexible y, por tanto, reutilizable.

La implementación del patrón es inmediata al ser una característica propia del lenguaje javascript, el cual se basa en prototipos. Además, el código resultante está optimizado, pues las funciones definidas en el prototipo son referenciadas por los objetos clonados, con el consecuente ahorro de memoria que ello supone.

## Implementación

Como se ha mencionado en el apartado anterior, la herencia a través de prototipos está definida en el estándar ECMAScript 5. Para clonar un objeto se hace uso del método "Object.create"<sup>1</sup>.

```
// Figura: Objeto prototipo
var figura = {
  posicionX: 0,
  posicionY: 0,

  mostrarPosicion: function() {
    console.log('Posición actual: ' + this.posicionX + ',' + this.posicionY);
  },

  mover: function( x, y ) {
    this.posicionX += x;
    this.posicionY += y;
    console.log('Nueva posición: ' + this.posicionX + ',' + this.posicionY);
  },
};

// Rectangulo: Objeto clonado a partir de Figura
var rectangulo = Object.create(figura);

// Usar método clonado mover.
rectangulo.mover(5,5); // Muestra 'Nueva posición: 5,5'

figura.mostrarPosicion(); // Muestra 'Posición actual: 0,0'
rectangulo.mostrarPosicion(); // Muestra 'Posición actual: 5,5'
```

<sup>1</sup>. Referencia de MDN:

[https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\\_globales/Object/create](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/create) ↵

# Singleton

## Propósito

Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.

Existen situaciones en las que existe un único recurso que debe ser compartido por el resto de participantes (por ejemplo, un único puerto de comunicaciones RS232 compartido por varios elementos de la aplicación).

Una variable global, aunque proporciona un punto de acceso global, no garantiza el requisito de única instancia. Por otro lado, una clase/objeto estático, aunque cumple el cometido del patrón, obliga a que la inicialización del mismo se produzca en el mismo momento de su definición. En cambio, el patrón singleton permite postergar dicha inicialización.

Existe una gran controversia respecto al uso de este patrón (llegando a ser considerado por algunos autores como un antipatrón <sup>12</sup>) debido a que introduce restricciones innecesarias en situaciones donde realmente no se requiere una sola clase, además de introducir un estado global en la aplicación y dificultar la capacidad del código de ser probado.

## Implementación

El código que se usa a modo de ejemplo usa la función `Math.random()` para obtener un número aleatorio, gracias al cual se identificará la instancia única creada por el objeto Singleton

```
var singleton = (function() {

    var instancia; // Referencia a instancia única.

    function iniciarInstancia() {

        // Funciones y propiedades - Inicialmente privadas
        function raizCuadrada(numero) { return Math.sqrt(numero); }
        var numeroAleatorio = Math.random();

        // Funciones y propiedades publicas
        return {
            raiz: raizCuadrada,
            identificador: numeroAleatorio
        };
    }

    return {
        obtenerInstancia: function() {
            if ( !instancia ) {
                instancia = iniciarInstancia();
            }
            return instancia;
        }
    };
})();

var objeto1 = singleton.obtenerInstancia();
var objeto2 = singleton.obtenerInstancia();

// objeto1 y objeto2 referencian a la misma instancia,
// por lo que ambos deben tener el mismo identificador
console.log(objeto1.identificador === objeto2.identificador); // true
```

Según el libro "Design Patterns", la instancia debe ser extensible a través de la herencia, y los clientes deberían poder extender instancias sin modificar su código. Para cumplir con este requisito basta con sobrescribir la función "obtenerInstancia" para que cree una instancia diferente en vez de llamar al método privado "iniciarInstancia()"

---

1. Artículo en google Code sobre la controversia sobre el patrón Singleton

(<https://code.google.com/archive/p/google-singleton-detector/wikis/WhySingletonsAreControversial.wiki>) ↩

2. Scott Densmore. Why singletons are evil

(<https://blogs.msdn.microsoft.com/scottdensmore/2004/05/25/why-singletons-are-evil/>) ↩

# Adapter

## Propósito

Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permite que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.

Facilita la reutilización de código, adaptando la interfaz de una clase existente en otra compatible con el cliente que desea usar objetos de dicha clase. El patrón emplea la siguiente terminología:

- **Objetivo:** Interfaz que espera el cliente.
- **Cliente:** usuario que empleará la interfaz objetivo para comunicarse con la clase a adaptar.
- **Adaptable:** clase existente cuya interfaz necesita ser adaptada
- **Adaptador:** clase "envoltorio" que adapta la interfaz del adaptable a la interfaz objetivo.

Adapter presenta similitudes con Facade, ya que ambas envuelve un objeto para modificar la interfaz que éste presenta al exterior. No obstante, mientras que Facade presenta una interfaz simplificada (sin opciones adicionales, haciendo suposiciones con objeto de simplificar la interfaz), Adapter no pretende disminuir la complejidad ni funcionalidad de la interfaz; en su lugar, su objetivo se limita a adaptar dicha interfaz para hacerla compatible con el cliente que desea usar la clase. Se trata pues de una adaptación sintáctica, y no semántica.

## Implementación

La aplicación que se expone requiere la consulta del precio de un producto a diferentes servicios en línea. Cada servicio expone una interfaz diferente al exterior, pero se desea que la aplicación use una API unificada para comunicarse con cada uno de ellos, por lo que el uso del patrón Adapter está justificado

```
// Objetivo: Interfaz
// function obtenerPrecio ( producto )

// Cliente
function Cliente(){
  var servicios = {};
  this.agregarServicio = function ( nombre, adaptador ) {
    servicios[nombre] = adaptador;
  };

  // Lista precios ofrecidos por los diferentes servicios
  // usando la interfaz objetivo
  this.listarPrecios = function ( producto ) {
    for(var servicio in servicios) {
      var precio = servicios[servicio].obtenerPrecio( producto );
      console.log( servicio + ': ' + precio );
    }
  };
}

// Adaptable: API del Servicio 1
function Amazon() {

  this.getPriceOf = function(price, taxes) {
    var prize;
    // ...
    prize = 5;
    // ...
    return ( prize*(1+taxes) );
  };

  // ...
}
```

```

// Adaptador del Servicio 1
function AdaptadorAmazon() {
    this.obtenerPrecio = function ( producto ) {
        return amazon.getPriceOf( producto, 0.21 );
    };
}

// Adaptable: API del Servicio 2
function ElCorteIngles() {

    this.pvp = function (producto) {
        var precio;
        // ...
        precio = 7;
        // ...
        return precio;
    }
}

// Adaptador del Servicio 2
function AdaptadorElCorteIngles() {
    this.obtenerPrecio = function ( producto ) {
        return elCorteIngles.pvp(producto);
    };
}

// Ejemplo de uso

// Instancias de cliente y servicios
var cliente = new Cliente();
var amazon = new Amazon();
var elCorteIngles = new ElCorteIngles();

// Instancias de adaptadores
var adaptadorAmazon = new AdaptadorAmazon();
var adaptadorElCorteIngles = new AdaptadorElCorteIngles();

// Registro de servicios en cliente;
cliente.agregarServicio('Amazon', adaptadorAmazon);
cliente.agregarServicio('El Corte Ingles', adaptadorElCorteIngles);

// Listado de precios de producto
cliente.listarPrecios('Playstation 4');

```

# Bridge

## Propósito

Desacopla una abstracción de su implementación, de manera que ambas puedan variar de forma independiente

En programación orientada a objetos, cuando una abstracción puede tener varias implementaciones se suele crear una clase abstracta que define la interfaz, siendo las subclases quienes lleven a cabo las diferentes implementaciones. Esta solución dificulta modificar, extender y reutilizar la abstracción e implementaciones de forma independiente, pues ambas clases se encuentran fuertemente acopladas.

Para resolverlo, el patrón Bridge desacopla la interfaz y la implementación haciendo que ambas pertenezcan a jerarquías de clases diferentes. Para ello, define los siguientes participantes:

- **Abstracción:** Interfaz de la abstracción. Contiene una referencia a un objeto "Implementador".
- **AbstracciónRefinada:** Subclase de "Abstracción" (extiende la interfaz original).
- **Implementador:** Interfaz de las clases de implementación. Suele proporcionar operaciones primitivas que serán usadas por "Abstracción" en operaciones de más alto nivel.
- **ImplementadorConcreto:** Clase que usa la interfaz "Implementador"

De la enumeración anterior se deduce que Bridge requiere dos interfaces para desacoplar la abstracción de la implementación, por lo que se suele considerar al patrón como una única abstracción formada por dos capas.

La relación entre ambas jerarquías ("Abstracción" e "Implementador") a través de la referencia de la primera hacia la segunda es lo que se conoce como "puente", pues hace de nexo entre ambas interfaces.

Por último, merece la pena destacar las diferencias frente al patrón Adapter: mientras que el primero tiene como objeto adaptar dos interfaces existentes que son incompatibles entre sí, el segundo pretende desacoplar interfaz de implementación, promoviendo "composición sobre herencia" en el diseño.

## Implementación

Suponga que necesita diseñar un sistema de ventanas compatible con los sistemas operativos más populares del mercado. La primera capacidad que se desea implementar es la de crear nuevas ventanas con fondo opaco o transparente. Una posible solución de diseño se muestra en las siguientes líneas:

```
// Con patrón BRIDGE:
//
//   Escritorio   <---   Pintar
//   /           \       /   \
// Windows   Unix   conTransparencia   conRelleno
//
//
// Implementación
//
// Implementador
function Pintar () {

    this.pintarVentana = function () {
        throw "pintarVentana no implementado";
    };
}

// ImplementadorConcreto 1
function PintarConRelleno () {
```

```

    this.pintarVentana = function () {
        // ... Implementación
        console.log('Ventana pintada con fondo opaco');
    };
}

// ImplementadorConcreto 2
function PintarConTransparencia () {

    this.pintarVentana = function () {
        // ... Implementación
        console.log('Ventana pintada con fondo transparente');
    };

}

// Abstracción
function Escritorio( implementador ) {

    // Referencia a objeto con interfaz "Implementador"
    var pintar = implementador;

    // Interfaz de Abstracción
    this.nuevaVentana = function () {
        throw "nuevaVentana no implementado";
    };
}

// Abstracción Refinada 1
function EscritorioWindows ( implementador ) {

    // Referencia a objeto con interfaz "Implementador"
    var pintar = implementador;

    // Interfaz de Abstracción
    this.nuevaVentana = function () {
        console.log('Nueva ventana en S.O Windows. ');
        pintar.pintarVentana();
    };

    // Resto de métodos específicos de Escritorio Windows
    // ...
}

// Abstracción Refinada 2
function EscritorioUNIX ( implementador ) {

    // Referencia a objeto con interfaz "Implementador"
    var pintar = implementador;

    // Interfaz de Abstracción
    this.nuevaVentana = function () {
        console.log('Nueva ventana en S.O UNIX. ');
        pintar.pintarVentana();
    };

    // Resto de métodos específicos de Escritorio UNIX
    // ...
}

// Ejemplo de Uso

// Cliente: Crea Implementadores concretos.
var pintarUsandoTransparencia = new PintarConTransparencia();
var pintarUsandoRelleno = new PintarConRelleno();

// Cliente: crea Abstracciones refinadas usando los Implementadores concretos
var escritorioWindows = new EscritorioWindows ( pintarUsandoRelleno );

```

```

var escritorioUNIX = new EscritorioUNIX ( pintarUsandoTransparencia );

// Cliente: usa la interfaz ofrecida por Abstracción,
// la cual a su vez empleará la interfaz ofrecida por Implementador
escritorioWindows.nuevaVentana(); // Crea una nueva ventana con fondo opaco
escritorioUNIX.nuevaVentana(); // Crea una nueva ventana con fondo transparente

// Sin patrón BRIDGE (usando herencia):
//
//          Escritorio(Interfaz: PintarVentana)
//          /                               \
//        Windows                          Unix
//       /   \                             /   \
//  conTransparencia  conRelleno  conTransparencia  conRelleno
//
//
// Implementación
//
// ...

```

El patrón Bridge se encuentra muy extendido en la programación orientada a eventos y en el diseño de APIs<sup>1</sup>, pues promueve la creación de componentes modulares y verificable.

---

<sup>1</sup>. Puede encontrar ejemplos de ello en el capítulo 8 del libro "Pro Javascript Design Patterns", de Ross Harnes y Dustin Diaz. Editorial Apress.2008 [↩](#)

# Composite

## Propósito

Combina objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y compuestos.

Permite tratar a una colección de objetos de la misma forma en la que se trata a cada objeto particular. Para ello, la colección implementa las mismas operaciones que implementan cada objeto de forma individual, de forma que una operación ejecutada sobre el conjunto se ejecuta en cada uno de sus miembros.

Además, este patrón de diseño organiza los elementos en una estructura en árbol, permite que éste sea recorrido y habilita métodos para recuperar cualquier nodo/hoja del mismo.

En la definición de Composite de GoF se distinguen cuatro colaboradores:

- **Componente:** Interfaz de los objetos individuales y métodos para acceder a los hijos y al padre.
- **Compuesto:** Objeto que almacena componentes hijos. Implementa las operaciones de la interfaz "Componente" relacionadas con los hijos.
- **Hoja:** Representa a un objeto sin hijos dentro de la colección.
- **Cliente:** Manipula a los objetos de la composición usando la interfaz componente.

Cabe destacar que cada composición dentro de la colección tiene una relación de pertenencia<sup>1</sup> (el componente A TIENE el hijo B), y no una relación de generalización/especialización (clase/subclase).<sup>2</sup>

## Implementación

Con el fin de mostrar la implementación de Composite, se ha definido la estructura en árbol de un menú alimenticio compuesto por un plato de arroz con tomate y un flan para el postre. Cada objeto compuesto (es decir, cada plato) está a su vez formado por ingredientes simples. Gracias al patrón de diseño, el cálculo de calorías del menú o la búsqueda de alérgenos se vuelve trivial.

```
// Componente: Objeto básico individual
function IngredienteBasico(nombre, calorias,
    contieneGluten, contieneLactosa) {

    var id = nombre;
    var aporteCalorico = calorias;
    var conGluten = contieneGluten;
    var conLactosa = contieneLactosa;

    // Interfaz de objeto básico
    this.obtenerNombre = function () { return id; };
    this.obtenerCalorias = function () { return aporteCalorico; };
    this.contieneGluten = function () { return conGluten; };
    this.contieneLactosa = function () { return conLactosa; };
}

// Compuesto: Implementa la interfaz de componente,
// aplicada a todos los hijos
function IngredienteCompuesto(nombre) {
    var id = nombre;

    // Referencia a hijos
    var ingredientes = [];

    // Métodos para gestionar hijos
    this.agregarIngrediente = function (ingrediente) {
        ingredientes.push( ingrediente );
    };
}
```

```

}

// Interfaz de objeto básico, considerando a los hijos
this.obtenerNombre = function () { return id; };

this.obtenerCalorias = function () {
    var totalCalorias = 0;
    ingredientes.forEach(function ( ingrediente ) {
        totalCalorias += ingrediente.obtenerCalorias()
    });

    return totalCalorias;
};

this.contieneGluten = function () {
    var conGluten = false;
    ingredientes.forEach(function ( ingrediente ) {
        conGluten = conGluten || ingrediente.contieneGluten() ;
    });

    return conGluten;
};

this.contieneLactosa = function () {
    var conLactosa = false;
    ingredientes.forEach(function ( ingrediente ) {
        conLactosa = conLactosa || ingrediente.contieneLactosa() ;
    });
    return conLactosa;
};
}

// Ejemplo de uso

// Ingredientes básicos
var huevo = new IngredienteBasico('huevo', 155, false, false);
var leche = new IngredienteBasico('leche', 141, false, true);
var pan = new IngredienteBasico('leche', 141, true, false);
var azucar = new IngredienteBasico('azucar', 265, false, false);
var arroz = new IngredienteBasico('arroz', 130, false, false);
var tomate = new IngredienteBasico('tomate', 18, false, false);

// Plato simple
var arroz_con_tomate = new IngredienteCompuesto('arroz_con_tomate');
arroz_con_tomate.agregarIngrediente( arroz );
arroz_con_tomate.agregarIngrediente( tomate );

// Postre simple
var flan = new IngredienteCompuesto('flan');
flan.agregarIngrediente( leche );
flan.agregarIngrediente( huevo );
flan.agregarIngrediente( azucar );

console.log(arroz_con_tomate.obtenerCalorias()); // Muestra '148'
console.log(flan.contieneLactosa()); // Devuelve true;
console.log(arroz_con_tomate.contieneGluten()); // Devuelve 'false'

// Agregamos pan al arroz con tomate
arroz_con_tomate.agregarIngrediente( pan );
console.log(arroz_con_tomate.contieneGluten()); // Devuelve 'true'

```

1. Relación HAS-A(<https://en.wikipedia.org/wiki/Has-a>) ↔

2. Relación IS-A(<https://en.wikipedia.org/wiki/Is-a>) ↔

# Decorator

## Propósito

Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

El patrón es útil cuando necesitamos añadirle responsabilidades a objetos individuales, y no a todos los objetos de una clase. La herencia queda, por tanto, descartada.

Es especialmente útil cuando un objeto puede contener un gran número de características diferentes, por lo que definir una subclase para cada una de las posibles combinaciones generaría demasiado código difícil de manejar, mantener y reusar. En su lugar, Decorator define un objeto base y agrega propiedades al mismo (lo decora).

En lenguajes estáticos la implementación del patrón se realiza envolviendo al objeto con otro nuevo que posee la misma interfaz y añade funcionalidades; se crea un objeto "Decorator" que contiene una referencia del objeto "Decorado", y se redirigen las peticiones hacia éste a través del "Decorator"<sup>1</sup>.

Por la naturaleza de javascript, agregar atributos/métodos a objetos es un proceso sencillo, por lo que la implementación del patrón se simplifica considerablemente<sup>2</sup>.

## Implementación

A continuación se muestra, haciendo uso del patrón Decorator, cómo agregar piezas a un componente base ("ordenador") través de diferentes "Decoradores".

```

// Componente base
function Ordenador() {
  this.cpu = 'i5'; // Nombre comercial
  this.ram = 4; // Gigabytes
  this.hdd = 1; // Terabytes

  this.precio = function () {
    var precioBase = 500;
    return precioBase;
  };
}

// Decorador concreto 1: Agrega tarjeta gráfica dedicada
function graficaDedicada( ordenador ) {
  ordenador.graficaDedicada = 'GeForce GTX 1060';

  // Decora la función precio agregando el coste de la tarjeta.
  var precioActual = ordenador.precio();
  ordenador.precio = function () {
    var precioConGrafica = precioActual + 300;
    return precioConGrafica;
  };
}

// Decorador concreto 2: Agrega impuestos
function impuestosPeninsula ( ordenador ) {
  ordenador.impuestos = 0.21;

  // Decora la función precio agregando impuestos
  var precioActual = ordenador.precio();
  ordenador.precio = function () {
    var precioConImpuestos = precioActual * (1 + ordenador.impuestos);
    return precioConImpuestos;
  };
}

// Ejemplo de uso: nuevo componente base
var ordenador = new Ordenador();

// Decorar agregándole una tarjeta gráfica dedicada
graficaDedicada(ordenador);

// Decorar agregándole los impuestos de la zona.
impuestosPeninsula (ordenador);

// Comprobación del resultado
// Precio total: (500 +300)*1.21
console.log(ordenador.precio()); // Muestra '968'
console.log(ordenador.graficaDedicada); // Muestra 'GeForce GTX 1060'

```

1. Un ejemplo de implementación del diseño clásico se encuentra en el libro "Pro JavaScript Design Patterns". Ross Harmes and Dustin Diaz. Apress 2008. Capítulo 12. [↩](#)

2. Puede encontrar implementaciones alternativas en el libro "Javascript Patterns". Stoyan Stefanov. O-Reilly. 2010. Capítulo 7. [↩](#)

# Facade

## Propósito

Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.

Gracias a Facade la apariencia de un sistema complejo puede simplificarse, exponiendo al exterior una API de alto nivel que abstrae la maraña interna de componentes. Como contrapartida, esta simplificación de la realidad puede llevar consigo la disminución de servicios ofrecidos o una falta de eficiencia del código resultante.

## Implementación

Suponga que una aplicación dispone de una base de datos en la que existe una tabla llamada "Personas" que contiene los campos 'nombre' y 'edad'. Además, la aplicación permite hacer consultas SQL estándar, de forma que para obtener una relación de todas las personas mayores de edad habría que ejecutar la sentencia `"SELECT * FROM PERSONAS WHERE EDAD >= 18"`

Para simplificar la interacción con la base de datos, el siguiente código define, haciendo uso del patrón fachada, la función "seleccionar":

```
// Función compleja Database
function BaseDeDatos() {
  this.connect = function ( usuario, password ) {
    // ... Implementación ...
  };

  this.sql = function ( sentencia ) {
    console.log( 'Consulta realizada: ' + sentencia );
    // ... Implementación ...
  };

  // ... Resto de funciones ...
}

// Instancia de base de datos
var baseDeDatos = new BaseDeDatos();

// Función tipo Facade
function seleccionar( tabla, filtro ) {
  return baseDeDatos.sql("SELECT * FROM " + tabla + " WHERE " + filtro);
}

// Ejemplo de uso:
// Muestra 'Consulta realizada: SELECT * FROM PERSONAS WHERE >= 18`
// Recibe un listado de todas las personas mayores de edad
var personasMayoresDeEdad = seleccionar( 'PERSONAS', '>= 18');
```

El ejemplo anterior, aunque muy simplificado e inválido en un entorno de producción (no se han tenido en cuenta consideraciones de seguridad y se presupone que el usuario empleará nombres de tablas y filtros válidos), expone una interfaz mucho más clara y sencilla para quienes desconozcan el lenguaje SQL.

Muchas librerías populares javascript hacen uso del patrón Facade para simplificar la interacción del usuario con el navegador. Un ejemplo de ello es JQuery y sus "Selectores"<sup>1</sup>.

<sup>1</sup>. Página web de JQuery (<https://api.jquery.com/category/selectors/>) ↩

# Flyweight

## Propósito

Usa el compartimento para permitir un gran número de objetos de grano fino de forma eficiente.

En aplicaciones que contienen un gran número de objetos se requiere una gran capacidad de memoria. Cuando dichos objetos contienen cierta cantidad de código repetitivo, la aplicación está haciendo un uso ineficiente de los recursos, pues cada objeto contiene una porción de código repetida cuando realmente podría ser compartida por todos los objetos.

El patrón describe cómo compartir pequeños objetos (llamados "Peso ligero") para permitir su uso en el resto de componentes, optimizando así el uso de memoria de la aplicación. Tiene gran relevancia, por tanto, en aplicaciones que hacen uso intensivo de la memoria y requieren manejar una gran cantidad de información.

Para separar la información "común", la cual se compartirá, del resto de información, el patrón define el concepto de estado intrínseco/extrínseco de un objeto. Así, el estado intrínseco hace referencia a las propiedades del objeto que serán compartidas, mientras que el extrínseco es la colección de propiedades que quedan fuera del objeto compartido. Por ejemplo, un objeto con las propiedades "marca, modelo, propietario, fecha de compra" puede dividirse en un objeto compartida con estado intrínseco "marca, modelo" y estado extrínseco "propietario, fecha de compra", de forma que sólo existirá un objeto compartido por cada tupla "marca, modelo" donde antes existían muchos más objetos con información repetida.

La información asociada al estado intrínseco se almacena en el objeto compartido, mientras que el estado extrínseco se almacenará en otro lugar externo a dicho objeto (por ejemplo, en los objetos que emplean la información compartida).

Cabe destacar que la separación de las propiedades/métodos de un objeto en estado intrínseco y extrínseco es subjetiva y depende de la aplicación.

Para la creación de los objetos compartidos se suele emplear el patrón Factory Method, haciendo que la fábrica devuelva referencias de los objetos compartidos si estos ya han sido solicitados con anterioridad, o bien creando nuevos objetos compartidos si aún no existe una instancia de los mismos. Efectivamente, el patrón Singleton también aparece en el escenario para crear instancias únicas y compartidas.

## Implementación

Ammodo de ejemplo, el siguiente código optimiza en cuanto a uso de memoria la gestión de todos los vehículos registrados en un país, caracterizados por los parámetros (marca, modelo, Año de fabricación, número de bastidor, NIF del propietario, dirección del propietario, teléfono de contacto del propietario)

```
// Objeto optimizado como peso ligero: coche
function Coche ( marca, modelo, anio ) {

    // Estado intrínseco
    this.marca = marca;
    this.modelo = modelo;
    this.anioDeFabricacion = anio;
}

// Fábrica de objetos ligeros
function FabricaDeCoches() {

    var listadoDeCoches = {}; // marca-modelo-anio: instancia

    this.crearCoche = function ( marca, modelo, anio ) {
```

```

// Implementación singleton
var referencia = marca + '-' + modelo + '-' + anio;
if( listadoDeCoches[referencia] ) {
    return listadoDeCoches[referencia];
} else {
    var coche = new Coche(marca, modelo, anio);
    listadoDeCoches[referencia] = coche;
    return coche;
}
};
}

// Gestión del estado extrínseco
function BaseDeDatosDeVehiculos () {

    var listadoDeVehiculos = {}; // bastidor: {nif, direccion, telefono, coche}

    // Agregar nuevo registro
    this.nuevoRegistro= function (marca, modelo, anio, bastidor,
        nif, direccion, telefono) {
        var coche = fabricaDeCoches.crearCoche(marca, modelo, anio);
        listadoDeVehiculos[bastidor] = {
            // Estado extrínseco
            nif: nif,
            direccion: direccion,
            telefono: telefono,

            // Referencia a objeto peso ligero
            coche: coche
        };
    };

    // Consultar registro en base a identificador único (bastidor)
    this.obtenerRegistro = function ( bastidor ) {
        return listadoDeVehiculos[ bastidor ];
    };
}

// Ejemplo de uso:
// Creación de la fábrica de objetos ligeros
var fabricaDeCoches = new FabricaDeCoches();

// Creación de la base de datos de vehículos
var baseDeDatos = new BaseDeDatosDeVehiculos();

// Registro de los dos primeros vehículos del país
baseDeDatos.nuevoRegistro('clement', 'triciclo', 1900, 1,
    '1234A', 'Palma de Mallorca', 'C/ Automoción', 'N/A');
baseDeDatos.nuevoRegistro('clement', 'triciclo', 1900, 2,
    '5678B', 'Cáceres', 'C/ Marqués', 'N/A');

// Obtenemos registros
var registro1 = baseDeDatos.obtenerRegistro(1);
var registro2 = baseDeDatos.obtenerRegistro(2);

// Comprobamos que ambos registros referencian al mismo objeto "coche"
console.log( registro1.coche === registro2.coche); // Muestra 'True'

```

Además del objeto coche, podríamos haber optimizado la información relativa al usuario convirtiéndola en otro objeto "Peso ligero", minimizando aún más el uso de memoria.

El sistema resultante es, no obstante, más complejo que su versión sin optimizar, pues la información se halla repartida en dos objetos diferentes donde antes sólo existía uno. Además, ha sido necesaria la implementación de una fábrica y dos clases Singleton (baseDeDatos también mantiene un listado de registros único).

# Proxy

## Propósito

Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.

Un proxy es un objeto que puede ser usado para controlar el acceso a otro objeto denominado "Sujeto". El proxy puede ser instanciado en lugar del objeto real al que controla. Puede, además, retrasar la instanciación del objeto real hasta el momento en que éste sea realmente utilizado<sup>1</sup>. Éste último hecho tiene especial interés en componentes cuya inicialización requiere un alto consumo de recursos, pues permite una carga inicial rápida:

- Si el objeto real no se llega a utilizar, entonces nunca será instanciado.
- Si se sabe que el objeto real será requerido, entonces permite una carga en segundo plano mientras continúa el resto de la aplicación.

Mientras que el patrón Facade encapsula la llamada de varios métodos complejos a través de un método simple, Proxy se sitúa entre el cliente de un objeto y el objeto mismo, protegiendo su acceso para favorecer los siguientes aspectos:

- El acceso local a un recurso remoto (proxy remoto).
- La inicialización de objetos costosos (proxy virtual).
- La seguridad en el acceso a un recurso protegido (proxy de protección).
- Conteo de referencias al objeto real.
- Facilitar la concurrencia cuando el objeto real es un objeto compartido.

## Implementación

El código de ejemplo emplea el patrón Proxy con múltiples propósitos

- Retrasar la instanciación de la clase Geolocalizador hasta el momento en que ésta es necesaria.
- Guardar en memoria local los resultados durante las diferentes peticiones al servicio de geolocalización, disminuyendo el número de accesos remotos

```
// Sujeto
function Geolocalizador() {

    // Proceso de inicialización costoso en tiempo.
    this.iniciar = function () {
        // implementación
    };

    // Obtiene coordenadas de un servicio de mapas
    this.coordenadasDe = function ( ciudad ) {
        // Implementación
        // Petición remota a servidor de mapas
        // ...

        // Ejemplo de respuestas
        var ciudadEnMayusculas = ciudad.toUpperCase();
        switch (ciudadEnMayusculas) { // En orden alfabético
            case 'BARCELONA':
                return '41.385064, 2.173403';
                break;
            case 'MADRID':
                return '40.416775, -3.703790';
                break;
            case 'SEVILLA':
                return '37.389092, -5.984459';
                break;
        }
    };
}
```

```

        case 'VALENCIA':
            return '39.469907, -0.376288';
            break;
    }
};
}

// Proxy
function GeoProxy() {
    var geolocalizador; // Referencia a Sujeto

    // Caché local de consultas a geolocalizador
    var cache = {};

    // Dato estadístico. Consultas realizadas a geolocalizador
    var consultasAGeolocalizador = 0;

    this.coordenadasDe = function ( ciudad ) {

        // Variable auxiliar
        var ciudadEnMayusculas = ciudad.toUpperCase();

        // Devolver el resultado almacenado en caché, si existe.
        if( cache[ciudadEnMayusculas] ) {
            return cache[ciudadEnMayusculas];
        }

        // Lazy loading: instanciación del geolocalizador si
        // aún no se ha solicitado ninguna ciudad.
        if( !geolocalizador ) {
            geolocalizador = new Geolocalizador();
            geolocalizador.iniciar();
        }

        // Solicitud de coordenadas
        cache[ciudadEnMayusculas] = geolocalizador.coordenadasDe( ciudad );
        consultasAGeolocalizador++;

        return cache[ciudadEnMayusculas];
    };

    // Estadísticas de solicitudes diarias.
    this.consultasRemotas = function () {
        return consultasAGeolocalizador;
    };
}

// Ejemplo de uso

// Instanciación de geoProxy (retrasa creación de geolocalizador)
var geoProxy = new GeoProxy();

// Solicitud de primera ciudad:
// - Proxy creará instancia de geolocalizador
// - Proxy realizará la primera solicitud a geolocalizador
console.log( geoProxy.coordenadasDe( 'Sevilla' ) );

// Solicitud de segunda ciudad:
// - Proxy usará la instancia existente de geolocalizador
// - Proxy realizará la segunda solicitud a geolocalizador
console.log( geoProxy.coordenadasDe( 'Madrid' ) );

// Solicitud de tercera ciudad:
// - Proxy usará la caché interna para devolver resultado
console.log( geoProxy.coordenadasDe( 'Sevilla' ) );

// Número de accesos a servicio remoto
console.log( geoProxy.consultasRemotas() ); // Devuelve '2'

```

Gracias al patrón se disminuyen los tiempos de acceso a la aplicación y se mejoran los tiempos de acceso a consultas repetidas en la sesión. Como cntrapartida, al igual que sucede con el resto de patrones, Proxy añade complejidad a la solución.

---

<sup>1</sup>. Este concepto se conoce como carga diferida (lazy loading) ([https://es.wikipedia.org/wiki/Lazy\\_loading](https://es.wikipedia.org/wiki/Lazy_loading)) ↵

# Chain of Responsibility

## Propósito

Evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que ésta sea tratada por algún objeto.

El patrón permite a un objeto transmisor enviar una petición a una cadena de uno o varios objetos receptores. Cada elemento de la cadena receptora procesa la petición y/o la traslada al objeto contiguo. Por tanto, cada eslabón de la cadena sólo conoce a sus elementos adyacentes.

El transmisor sólo se comunica con el primer objeto de la cadena. La petición se envía a través de un objeto dedicado que contiene toda la información, o bien llamando a un método del objeto receptor sin argumentos (en cuyo caso la invocación del método es la petición en sí misma). Cualquier nodo de la cadena receptora puede realizar 3 acciones diferentes:

- Procesar completamente la petición y dar por finalizada la operación, en cuyo caso no traslada la petición al siguiente receptor.
- Procesar parcialmente la solicitud, modificarla (si es necesario) y trasladarla al siguiente elemento.
- No procesar la petición y trasladarla íntegramente al nodo contiguo.

En la definición del patrón se denomina "Cliente" al transmisor de la petición, "Manejador" a la interfaz usada por los receptores para tratar las peticiones y enlazarse con el siguiente nodo de la cadena, y "ManejadorConcreto" a cada uno de los eslabones de la cadena que implementan la interfaz "Manejador".

Es recomendable usar este diseño cuando existe más de un objeto que puede manejar una misma petición, pero el cliente no conoce a priori (o no desea especificar) cual de ellos debería procesarla. También se aconseja en escenarios en los que se requiere especificar dinámicamente a los receptores.

En aplicaciones web, Chain of Responsibility se emplea en el manejo de eventos. Cuando un elemento del DOM<sup>1</sup> dispara un evento (por ejemplo, notificando una pulsación de ratón sobre él), el navegador<sup>2</sup> lo captura y propaga de forma jerárquica, comenzando por el elemento que inició el evento, ascendiendo hasta su nodo padre y llegando hasta el nodo raíz en caso de que ningún componente de la cadena maneje la petición.

## Implementación

El funcionamiento de una máquina de cambio de billetes encaja perfectamente con la filosofía del patrón:

```

// Manejador
function Expendedor ( unidades ) {

    var valorDelBillete = unidades;
    var sucesor;

    this.siguiete = function ( expendedor ) {
        sucesor = expendedor;
    };

    this.cambiar = function ( cantidad ) {
        var numeroDeBilletes = Math.floor(cantidad / valorDelBillete);
        var noDispensable = cantidad - (numeroDeBilletes * valorDelBillete);

        if( numeroDeBilletes > 0 ) {
            console.log(numeroDeBilletes + ' billetes de ' +
                valorDelBillete + ' dispensados');
        }

        if ( noDispensable > 0 && sucesor ) {
            sucesor.cambiar ( noDispensable);
        } else if ( noDispensable ) {
            console.log('No se ha podido dar cambio de ' + noDispensable);
        }
    };
}

// Manejadores Concreto
var expendedorDeCincoUnidades = new Expendedor(5);
var expendedorDeDiezUnidades = new Expendedor(10);
var expendedorDeVeinteUnidades = new Expendedor(20);
var expendedorDeCincuentaUnidades = new Expendedor(50);

// Establecer cadena:
expendedorDeCincuentaUnidades.siguiete( expendedorDeVeinteUnidades );
expendedorDeVeinteUnidades.siguiete( expendedorDeDiezUnidades );
expendedorDeDiezUnidades.siguiete( expendedorDeCincoUnidades );

// Cliente
function MaquinaDeCambio( expendedor ) {
    var expendedorInicial = expendedor;

    this.darCambioDe = function ( cantidad ) {
        expendedorInicial.cambiar( cantidad );
    };
}

// Ejemplo de uso

var maquinaDeCambio = new MaquinaDeCambio(expendedorDeCincuentaUnidades);
maquinaDeCambio.darCambioDe(123);

```

Un inconveniente de encadenar a los receptores es que el transmisor no conoce a priori quién ha respondido a la petición, ni si ésta ha sido resuelta satisfactoriamente. Para resolverlo, el último elemento de la cadena puede provocar una excepción en caso de que no pueda satisfacer el resto de la petición.

1. Document Object Model ([https://es.wikipedia.org/wiki/Document\\_Object\\_Model](https://es.wikipedia.org/wiki/Document_Object_Model)) ↩

2. Para entender el por qué del funcionamiento actual de los navegadores web, se recomienda el visionado de "Douglas Crockford: An Inconvenient API - The Theory of the DOM" (<https://www.youtube.com/watch?v=Y2Y0U-2qJM8>) ↩

# Command

## Propósito

Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer las operaciones.

Tal y como menciona el libro "Design Patterns" (GoF), existen situaciones en las que es necesario solicitar operaciones a objetos sin que se conozcan detalles de la acción o al receptor real de la misma.

El patrón Command permite encapsular la operación en un único objeto, el cual puede ser pasado al componente encargado de realizar dicha operación a través de una interfaz conocida. Como consecuencia, el objeto que solicita la operación y el objeto que la ejecuta quedan completamente desacoplados.

Los elementos que intervienen en el patrón son:

- Comando: Declara una interfaz para ejecutar una operación. Por ejemplo, declara el método "Ejecutar".
- ComandoConcreto: Implementa el método Ejecutar, el cual invoca las operaciones necesarias en el Receptor para llevar a cabo la orden.
- Cliente: Crea un objeto ComandoConcreto y establece su Receptor.
- Invocador: Le pide al objeto ComandoConcreto que ejecute la petición
- Receptor: es el objeto en donde se ejecutará realmente el comando.

## Implementación

El siguiente ejemplo muestra el uso del patrón en una calculadora básica que permite las operaciones de suma y resta.

```
// Receptor. Implementa las acciones reales a ejecutar
function Calculadora() {
    var valorActual = 0;

    this.incrementar = function ( numero ) {
        valorActual += numero;
        console.log("Nuevo valor: " + valorActual);
    };

    this.disminuir = function ( numero ) {
        valorActual -= numero;
        console.log("Nuevo valor: " + valorActual);
    };
}

// Interfaz Comando: define funciones ejecutar, deshacer
function Comando () {
    this.ejecutar = function(valor) {
        throw "Interfaz no implementada";
    };

    this.deshacer = function() {
        throw "Interfaz no implementada";
    };
}

// Comando Concreto: implementa funciones ejecutar, deshacer
function ComandoAgregar ( receptor, numero) {
    var claseReceptora = receptor;
    var valor = numero;

    this.ejecutar = function() {
```

```

        claseReceptora.incrementar( valor );
    };

    this.deshacer = function() {
        claseReceptora.disminuir( valor );
    };
}

// Comando Concreto: implementa funciones ejecutar, deshacer
function ComandoSustraer ( receptor, numero ) {
    var claseReceptora = receptor;
    var valor = numero;

    this.ejecutar = function( ) {
        claseReceptora.disminuir( valor);
    };

    this.deshacer = function() {
        receptor.incrementar(valor);
    };
}

// Cliente: crea los comandos y asigna receptor
function Cliente( receptor ) {
    this.comandos = [];

    this.comandos.push( new ComandoAgregar( receptor, 50 ) );
    this.comandos.push( new ComandoAgregar( receptor, 20 ) );
    this.comandos.push( new ComandoSustraer( receptor, 10 ) );
}

// Invocador: ejecuta peticiones
function Invocador ( comandos ) {
    var listaComandos = comandos;

    this.ejecutar = function() {
        listaComandos.forEach( function ( comando ) {
            comando.ejecutar();
        });
    };

    this.deshacerComandos = function ( numeroDeComandos ) {
        var i=0;
        for( /* i=0 */ ; i<numeroDeComandos; i++ ) {
            listaComandos.pop().deshacer();
        }
    };
}

// Ejemplo:
var calculadora = new Calculadora(); // Receptor
var cliente = new Cliente(calculadora); // Cliente
var invocador = new Invocador( cliente.comandos ); // Invocador

// Ejecución de comandos con interfaz común. Salida de consola:
// 'Nueva valor: 50'
// 'Nueva valor: 70'
// 'Nueva valor: 60'
invocador.ejecutar();

// Deshacer último comando. Salida:
// 'Nueva valor: 70'
invocador.deshacerComandos(1);

```

Aunque el código generado es complejo respecto a la tarea que propone (sumar y restar), es muy flexible y fácilmente extensible a través de nuevos comandos. Además, permite crear un historial de acciones, permitiendo deshacerlas en caso necesario.

# Iterator

## Propósito

Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

El punto de partida es un objeto que almacena información en una estructura compleja interna. Iterator proporciona un método fácil para acceder secuencialmente a cada elemento de dicha estructura.

El "Cliente" que requiere acceder a los elementos de la estructura no necesita conocer la implementación de ésta; todo lo que necesita es acceder a los diferentes elementos individuales que la componen. Para ello, el objeto "Iterator" proporciona (entre otros) el método "siguiente", el cual devuelve el siguiente elemento consecutivo de la estructura. El patrón deja abierto el significado de "siguiente" en cada estructura, por lo que es decisión del programador interpretar qué debe devolver dicho método.

Iterator puede definir otros métodos auxiliares para recorrer los elementos de la estructura interna del objeto "Agregado", como por ejemplo

- "haTerminado": Devuelve un valor booleano indicando si existen más elementos en la estructura.
- "reiniciarIterator": reinicia el recorrido de la estructura, comenzando en el primer elemento.
- "elementoActual": devuelve el elemento actual de la estructura

Separar el mecanismo de recorrido del objeto agregado permite definir iteradores con diferentes implementaciones del método "siguiente", por lo que la misma estructura interna puede recorrerse en un orden diferente según el iterador empleado.

El iterador y el objeto agregado están acoplados (iterador necesita conocer la estructura interna), mientras que el cliente únicamente conoce la interfaz Iterator (siguiente, haTerminado, etc.) y la forma de los elementos de la estructura.

Para promover la reutilización de código, el objeto agregado (el que contiene la estructura compleja) puede implementar el método "obtenerIterator", el cual devuelve el iterador con interfaz conocida por el cliente. De este modo, cada subclase puede implementar su propio iterador concreto y proporcionarle al cliente una interfaz conocida.

## Implementación

```

// Objeto agregado
function MejoresCancionesDeRock() {
  // Estructura interna compleja
  var canciones = ['Kashmir',
    'Stairway to Heaven',
    'Smoke on the Water',
    'Bohemian Rhapsody',
    'Paranoid',
    'Highway to Hell',
    'Hotel California',
    'Sultans of Swing',
    'Dream on',
    'Enter Sandman'];

  // Interfaz Iterador implementado por MejoresCancionesDelRock
  this.obtenerIterador = function () {

    var indiceActual = 0;

    function siguiente() {
      if (haTerminado()) {
        return null;
      }
      return canciones[indiceActual++];
    }

    function haTerminado() {
      return indiceActual >= canciones.length;
    }

    function reiniciar() {
      indiceActual = 0;
    }

    function elementoActual() {
      return canciones[indiceActual];
    }

    return {
      siguiente: siguiente,
      haTerminado: haTerminado,
      reiniciar: reiniciar,
      elementoActual: elementoActual
    };
  }
};

// Ejemplo de uso

// Nuevo objeto agregado
var mejoresCancionesDeRock = new MejoresCancionesDeRock();

// Iterador asociado
var iterador = mejoresCancionesDeRock.obtenerIterador();

// Uso de interfaz Iterador para recorrer toda la estructura interna
// Muestra el listado interno completo
while (!iterador.haTerminado()) {
  console.log(iterador.siguiente());
}

// Reinicio del iterador
iterador.reiniciar();

// Obtener elemento actual (primer elemento tras el reinicio)
console.log( iterador.elementoActual() ); // Muestra 'Kashmir'

```

# Mediator

## Propósito

Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.

Según la RAE, un mediador es aquel que "Actúa entre dos o más partes para ponerlas de acuerdo en un pleito o negocio". De igual modo, el patrón Mediator permitirá que las diferentes partes del sistema se comuniquen a través de un componente común, el "Mediator".

Imagine, a modo de ejemplo, un formulario web en el que la habilitación de unos campos dependerá de las opciones previamente seleccionadas. Existirá, por tanto, dependencia entre los elementos del formulario. Si se encapsula el comportamiento colectivo de todos los elementos en un único objeto (el mediador), cada componente sólo tendrá que relacionarse con dicho objeto, evitando que unos objetos se refieran a otros explícitamente y reduciendo el número de interconexiones. Se pasa, pues, de una topología mallada a una topología en estrella.

Aunque este patrón posee muchas similitudes con "Publish/Subscribe", son conceptualmente diferentes. Aunque ambos comunican a todos los elementos del escenario a través de un nuevo componente (llamado "canal" en el caso del patrón Publish/Subscribe, y "mediador" en el patrón Mediator), el flujo de trabajo y la lógica de negocio es radicalmente diferente en cada patrón. Así, el "canal" actúa como un bus de comunicaciones que traslada los eventos notificados por los "publicadores" a todos los suscriptores, siendo éstos quienes controlan la lógica de negocio de la aplicación. En cambio, el "mediador" centraliza por sí mismo toda la lógica de negocio y el flujo de trabajo, coordinando a los objetos del sistema e indicándoles las tareas que deben realizar en cada momento.

Por todo lo anterior, ambos patrones pueden incluso ser usados en conjunto usando Publish/Subscribe para comunicar a todos los objetos con el "mediador" y viceversa, y "Mediator" para centrar la lógica de negocio y enviar las acciones a los diferentes objetos a través del "canal" expuesto por el otro patrón.

## Implementación

Suponga un juego en el que los jugadores tienen que acertar un número secreto, ganando quien acierte dicho número en primer lugar. Toda la información de la partida se muestra en un marcador.

El siguiente código emplea la figura del "mediador" para interconectar a jugadores y marcador

```
// Colega: Jugador
function Jugador ( nombre ) {
  this.nombre = nombre;
  this.intentos = 0;
  this.juez = undefined;

  this.jugar = function ( numero ){
    if ( juez ) {
      this.intentos++;
      juez.nuevoIntento( this, numero );
    }
  };
}

// Colega: Marcador
function Marcador() {

  // Actualizar marcador, mostrando el número de reintentos de cada jugador
  this.actualizar = function ( jugadores ) {
```

```

    jugadores.forEach(function ( jugador ){
        console.log('Intentos fallidos de ' + jugador.nombre +
            ': ' + jugador.intentos);
    });
};

// Informar del ganador de la partida.
this.mostrarGanador = function ( jugador ) {
    console.log('¡El jugador ' + jugador.nombre + ' ha ganado la partida!');
};
}

// Mediador: Juez
function Juez() {

    // Listado de jugadores
    var jugadores = [];

    // Referencia al Marcador
    var marcador;

    // Número secreto.
    var numeroSecreto
    numeroSecreto = Math.floor((Math.random() * 10) + 1);

    // A modo de demostración, fijaremos el número a un valor conocido.
    // Comentar para un juego real
    numeroSecreto = 3;

    // Agregar jugadores a la partida
    this.agregarJugador = function ( jugador ) {
        jugador.juez = this;
        jugadores.push(jugador);
    };

    // Agregar marcador
    this.agregarMarcador = function ( nuevoMarcador ) {
        marcador = nuevoMarcador;
    }

    // Permite a un jugador notificar de un nuevo intento]
    this.nuevoIntento = function ( jugador, numero ) {

        if ( numero == numeroSecreto) {
            marcador.mostrarGanador( jugador );
            // ... Limpiar partida ...
        } else {
            marcador.actualizar( jugadores );
        }
    };
}

// Ejemplo de partida
var jugador1 = new Jugador('Juan');
var jugador2 = new Jugador('Pedro');
var marcador = new Marcador();
var juez = new Juez();

// Establecer al objeto juez como árbitro de la partida
// y controlador del marcador
juez.agregarJugador(jugador1);
juez.agregarJugador(jugador2);
juez.agregarMarcador(marcador);

// Inicio del juego

// Muestra
// 'Intentos fallidos de Juan: 1

```

```
// Intentos fallidos de Pedro: 0''  
jugador1.jugar(1);  
  
// Muestra  
// 'Intentos fallidos de Juan: 1  
// Intentos fallidos de Pedro: 1'  
jugador2.jugar(2);  
  
// Muestra '¡El jugador Pedro ha ganado la partida!'  
jugador2.jugar(3);
```

Gracias a Mediator se ha centralizado la lógica de negocio en un solo elemento, disminuido el número de conexiones entre los mismos y, por tanto, reducido el acoplamiento entre los componentes del sistema.

# Memento

## Propósito

Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.

A veces es necesario implementar mecanismos de deshacer/rehacer que permitan a los usuarios anular o recuperar operaciones. Para ello, el estado de los objetos en cada instante debe ser almacenado.

La solución más sencilla es exponer el estado de los objetos, esto es, hacer todas sus propiedades accesibles al resto de objetos. Pero este diseño rompe el principio de encapsulación, comprometiendo la fiabilidad de la aplicación.

Como alternativa, el patrón Memento guarda una instantánea del estado del objeto y ofrece una interfaz para controlar el acceso a dicha imagen, de forma que el objeto que captura el estado es el único con acceso total al mismo. De esta forma, la encapsulación no se rompe completamente.

Aunque el patrón Command también tiene la habilidad de deshacer operaciones, solo es posible volver al estado original cuando existe un comando inverso que anula las acciones del principal. Por ejemplo, una calculadora implementada usando el patrón Command permite deshacer operaciones de suma y resta, pero no operaciones de elevar al cuadrado, pues durante el proceso se pierde el signo del operando original.

El patrón define los siguientes participantes:

- **Memento:** guarda el estado del objeto "Creador". Protege el acceso a dicho estado a través de dos interfaces. La primera de ellas ofrece al exterior una versión reducida del estado. La segunda, en cambio, permite al "Creador" recuperar el estado completo.
- **Creador:** objeto que contiene el estado interno a capturar. Es quien origina el "Memento" (recuerdo) de su estado interno actual. Proporciona una interfaz para generar nuevas instantáneas.
- **Conserje:** Contiene el almacén de objetos "Memento" y permite el mecanismo de deshacer. Nunca opera sobre el contenido de los objetos "Memento". En este patrón, tiene el rol de cliente.

El momento de creación de mementos depende de la aplicación. Puede interesar que un objeto genere nuevas instantáneas cada vez que su estado interno cambia. En otras ocasiones, la captura del estado se realizará bajo demanda del Conserje.

## Implementación

Una posible implementación que incluye protección de acceso al memento es:

```
// Clase Memento
function Estado(propietario, fecha) {

    // Encapsulación del estado
    var maquinaDelTiempo = propietario;
    var ubicacionTemporal = fecha;

    // Protección de acceso
    this.recuperar = function (objeto) {
        if (objeto === maquinaDelTiempo) {
            return ubicacionTemporal;
        } else {
            return null;
        }
    };
}
```

```

// Creador
function MaquinaDelTiempo(fechaDePartida) {

    // Estado interno
    var fechaActual = fechaDePartida;

    // Crea un nuevo memento
    this.coordenadasTemporales = function () {
        return new Estado(this, fechaActual);
    };

    // Recuperar memento
    this.regresar = function (estado) {
        var fechaDeRegreso = estado.recuperar(this);
        if (fechaDeRegreso) {
            fechaActual = fechaDeRegreso;
            console.log('Ha regresado a ' + fechaActual);
        }
    };

    // Cambia el estado del creador.
    this.viajarA = function (fecha) {
        fechaActual = fecha;
        console.log('Ha viajado a ' + fechaActual);
    };
}

// Conserje
function Bitacora() {

    // Mementos guardados
    var listadoDeViajes = [];

    // Almacena un nuevo Memento
    this.anotar = function (datosDelViaje) {
        listadoDeViajes.push(datosDelViaje);
    };

    // Recupera el último memento
    this.borrarUltimo = function () {
        return listadoDeViajes.pop();
    };
}

// Ejemplo de uso

// Nuevo "Creador" de tipo "Máquina del tiempo"
var DeLorean = new MaquinaDelTiempo(2016);

// Nuevo Conserje de tipo "Bitácora"
var bitacora = new Bitacora();

// Crear y almacenar el primer Memento
bitacora.anotar(DeLorean.coordenadasTemporales());

// Cambio de estado interno: Primer viaje en el tiempo
DeLorean.viajarA(1955); // Muestra 'Ha viajado a 1955'

// Recuperar Memento: Regreso al futuro
DeLorean.regresar(bitacora.borrarUltimo()); // Muestra 'Ha regresado a 2016'

```

Cabe destacar que aunque los métodos definidos se han nombrado teniendo en cuenta la semántica de la aplicación, se ha generado un comentario en cada línea para mostrar la asociación entre cada elemento y los colaboradores genéricos del patrón Memento.

# Observer

## Propósito

Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifica y se actualizan automáticamente todos los objetos que dependen de él.

El patrón observador permite a un objeto (conocido como "Sujeto") mantener una lista de objetos (denominados "observadores") que dependen de él, notificándolos de cualquier cambio en su estado. Para propiciar la reutilización del código, es necesario que el diseño permita un bajo acoplamiento entre todos los participantes.

En primer lugar los observadores se registran en el sujeto a través de la función "agregar" que éste publica. Cuando el sujeto necesite informar a los observadores de cualquier suceso, retransmite una notificación a los observadores a través del método público "actualizar" que éstos exponen. Por último, cuando un observador lo desea, puede darse "desconectarse" del sujeto a través de la función "quitar".

Esta patrón es ampliamente usado en javascript cuando se emplea en aplicaciones Web, pues facilita la gestión de acciones en respuesta a eventos del usuario (por ejemplo, pulsaciones de ratón, teclado, etc. son escuchadas por la aplicación para actuar en consecuencia).

## Implementación

A continuación se muestra el código necesario para crear los objetos colaboradores "Sujeto" y "Observador", así como un ejemplo de uso.

```

// Sujeto.
// Puede agregar, eliminar y notificar a los observadores de su lista.
function Sujeto(){
    this.listaObservadores = []; // Listado de observadores
}

Sujeto.prototype.agregarObservador = function (observador) {
    this.listaObservadores.push(observador);
}

Sujeto.prototype.quitarObservador = function (observador) {
    // Si el observador se encuentra en la lista, eliminar de ella.
    var indiceObservador = this.listaObservadores.indexOf( observador );
    if( indiceObservador != -1 ) {
        this.listaObservadores.splice( indiceObservador, 1 );
    }
}

Sujeto.prototype.notificarAObservadores = function ( mensaje ) {
    this.listaObservadores.forEach( function( observador ) {
        observador.actualizar( mensaje );
    } );
}

// Observador.
// Recibe notificaciones del Sujeto a través de la función "actualizar"
function Observador(nombre) {
    this.nombre = nombre;
    this.actualizar = function ( mensaje ) {
        console.log( nombre + ' ha recibido un mensaje. Contenido: ' + mensaje );
    };

    // Funciones auxiliares para que Observador pueda decidir
    // cuándo registrarse a un Sujeto o anular dicho registro.
    this.observarA = function( sujeto ) {
        sujeto.agregarObservador( this );
    };

    this.dejarDeObservarA = function( sujeto ) {
        sujeto.quitarObservador ( this );
    };
}

// Escenario de ejemplo: 1 sujeto y dos observadores suscritos
var sujeto = new Sujeto();
var observador1 = new Observador( 'Observador 1' );
var observador2 = new Observador( 'Observador 2' );

observador1.observarA( sujeto );
observador2.observarA( sujeto );

// Notificar cambio de estado.
// Salida de la consola:
// Observador 1 ha recibido un mensaje. Contenido: mensaje de echo
// Observador 2 ha recibido un mensaje. Contenido: mensaje de echo
sujeto.notificarAObservadores( 'mensaje de echo' );

// Eliminar a observador 1;
observador1.dejarDeObservarA( sujeto );

// Notificación al resto de observadores. Salida de la consola:
// Observador 2 ha recibido un mensaje. Contenido: observador 2 eliminado
sujeto.notificarAObservadores( 'observador 2 eliminado' );

```

El patrón Observer requiere que el objeto Observador se suscriba al objeto Sujeto, por lo que se requiere visibilidad entre ambos, con la consecuente dependencia entre ellos.

Como alternativa, el patrón Publish/Subscribe, establece un canal de comunicaciones intermedio que desacopla a Sujeto y Observador, conectando a todos los objetos interesados a una interfaz común en la que pueden intercambiar información. Cualquier objeto puede, por tanto, publicar información en el canal o suscribirse al canal para recibir dicha información. Puede encontrar una implementación de éste patrón en el repositorio Github de Addy Osmani (<https://github.com/addyosmani/pubsubz>).

# Template Method

## Propósito

- Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos.
- Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.

El patrón define un algoritmo en términos de operaciones abstractas junto con una implementación inicial de las mismas. Cualquier subclase que implemente la interfaz del algoritmo puede sobrescribir el comportamiento de cualquiera de las operaciones.

En la programación orientada a objetos basada en clases, la herencia proporciona este mecanismo. En javascript, puede ser implementado a través de la herencia por prototipos.

## Implementación

Una posible definición de un algoritmo que comunica dos equipos consta de los siguientes pasos:

1. Reservar recursos del equipo.
2. Iniciar llamada.
  - Depende del protocolo de comunicaciones
3. Enviar información.
  - Depende del protocolo de comunicaciones.
4. Finalizar llamada
5. Liberar recursos del ordenador.

Los pasos 1 y 5 dependen del hardware del equipo y de las capas físicas y de enlace, mientras que los pasos 2, 3 y 4 dependen del protocolo de comunicaciones a implementar. Cualquier equipo que implemente el algoritmo deberá, por tanto, reimplementar estos pasos para adecuarlos a sus características y al protocolo elegido.

```

// Plantilla genérica del algoritmo
function Comunicacion( nombreEquipo ) {
    this.nombre = nombreEquipo;
}

// Interfaz del algoritmo
Comunicacion.prototype.reservarRecursos = function ( parametros ) {
    console.log( 'Recursos reservados' );
};

Comunicacion.prototype.iniciarLlamada = function ( equipo ) {
    console.log( 'Conexión establecida con: ' + equipo.nombre );
};

Comunicacion.prototype.enviarInformacion = function ( mensaje ) {
    console.log( 'Información enviada: ' + mensaje );
};

Comunicacion.prototype.finalizarLlamada = function () {
    console.log( 'Llamada finalizada' );
};

Comunicacion.prototype.liberarRecursos = function () {
    console.log( 'Recursos Liberados' );
};

// Objeto que implementa el algoritmo:
// Extender ComunicacionArduino usando la interfaz Comunicacion
function ComunicacionArduino( nombre ) { this.nombre = nombre; }
ComunicacionArduino.prototype = Object.create(Comunicacion.prototype);
ComunicacionArduino.prototype.constructor = ComunicacionArduino;

// Reimplementar pasos específicos de arduino
ComunicacionArduino.prototype.reservarRecursos = function () {
    console.log( 'Arduino: Recursos reservados' );
};

ComunicacionArduino.prototype.liberarRecursos = function () {
    console.log( 'Arduino: Recursos Liberados' );
};

// Ejemplo de uso

// Objetos con algoritmo concreto
var equipo1 = new ComunicacionArduino('Arduino Nano');
var equipo2 = new ComunicacionArduino('Arduino Mega');

// Ejecución del algoritmo concreto
equipo1.reservarRecursos(); // Muestra 'Arduino: Recursos reservados'
equipo1.iniciarLlamada(equipo2); // Muestra Conexión establecida con Arduino Mega'
equipo1.enviarInformacion('echo'); // Muestra 'Información enviada: echo'
equipo1.finalizarLlamada(); // Muestra 'Llamada finalizada'
equipo1.liberarRecursos(); // Muestra 'Arduino: Recursos liberados'

```

# Module

## Propósito

Separar y organizar la funcionalidad total de un programa en módulos independientes e intercambiables.

Cada módulo contiene todo lo necesario para ejecutar un sólo aspecto de la funcionalidad deseada. Además, establece una interfaz pública con el exterior, ocultando detalles de implementación y facilitando la encapsulación del código y su reusabilidad.

El patrón Module puede considerarse creacional y estructural, pues gestiona la creación, organización y agrupación de otros elementos.

## Implementación

El siguiente código puede emplearse como plantilla para crear un módulo en Javascript:

```
var modulo1 = (function () {  
  
    // Variables privadas  
    var contador = 0;  
  
    // Funciones privadas  
    log = function (msg) { console.log(msg); };  
  
    return {  
  
        // Variables públicas  
        nombreModulo: "miModulo",  
  
        // Funciones públicas  
        incrementarContador: function () { contador++; },  
        reiniciarContador: function () { contador = 0; },  
        imprimirContador: function () { log(contador); }  
    };  
})();  
  
modulo1.incrementarContador(); // Contador = 1;  
modulo1.imprimirContador(); // Muestra '1'  
modulo1.reiniciarContador(); // Contador = 0;
```

El ámbito de las variables y funciones incluidas en el módulo queda definido por la función anónima que envuelve a todo el código, la cual es ejecutada inmediatamente quedando, por tanto, inaccesible desde el ámbito global. Como consecuencia, "modulo1" encapsula variables y funciones privadas y públicas, sirviendo de espacio de nombres.

Además, es posible emplear el patrón para devolver diferentes funciones/variables dependiendo del entorno donde se inicializa el módulo.

Puede encontrar más información sobre el patrón Module en la página personal de Todd Motto (<https://toddmotto.com/mastering-the-module-pattern/>)

# Revealing Module

## Propósito

Separar y organizar la funcionalidad total de un programa en módulos independientes e intercambiables.

Se trata de una variación del patrón Module en la que todas las funciones y variables se definen privadas para posteriormente publicar sólo aquellas de interés.

## Implementación

A continuación se muestra un módulo implementado usando el patrón "Revealing Module":

```
var revealingModule = (function () {

    // Variables - Inicialmente privadas
    var variable1 = 1;
    var variable2 = 2;

    // Funciones - Inicialmente privadas
    function funcion1() {
        console.log('Funcion 1');
    }

    function funcion2() {
        console.log('Funcion 2');
    }

    // Referencias públicas a funciones y propiedades privadas
    return {
        variablePublica: variable1,
        funcionPublica: funcion1
    };

})();

revealingModule.funcionPublica(); // Muestra 'Funcion 1'
revealingModule.variablePublica; // Muestra '1'
```

En el ejemplo anterior destaca la claridad con la que se define el módulo. Como contrapartida, si una función privada hace referencia a una pública que posteriormente se sobrescribe, entonces la función privada seguirá referenciando a la implementación privada, por lo que no empleará la nueva definición. Como resultado, el diseño puede considerarse más frágil que el patrón "Module".