UNIVERSIDAD TECNOLÓGICA DE PANAMÁ

FACULTAD DE INGENIERÍA DE SISTEMAS COMPUTACIONALES

LICENCIATURA EN DESARROLLO DE SOFTWARE

PROF. ERICK AGRAZAL

PATRÓN CONSTRUCTOR Y PATRÓN MÓDULO

EQUIPO #10

INTEGRANTES

ODETH AREVALO

MOISÉS BETANCOURT

GRUPO

1LS242

FECHA DE ENTREGA

19/04/2025

Contenido

Patrón Constructor 3
Concepto 3
¿Cuándo utilizarlo? 4
Estructura y Participantes del Patrón Constructor 5
Implementando el patrón constructor 6
Patrón Módulo 9
Concepto 9
¿Cúando se usa el patrón modulo? 10
Ejemplos de uso 11
Conclusiones 14
Bibliografías 15

Patrón Constructor

Concepto

Los desarrolladores de software, a menudo nos enfrentamos con diversos desafíos a la hora de crear objetos complejos cuando se trata de una gran variedad de configuraciones y características. Para facilitarnos este trabajo, solemos recurrir a la implementación de los patrones de diseño. Uno de ellos el Patrón constructor o también muy conocido como Builder.

Entonces, ¿Qué es el patrón constructor?

El patrón constructor es uno de los patrones de diseño creacional que, como mencionamos anteriormente, nos ofrece una solución a problemas relacionados a la creación de objetos en la programación orientada a objetos.

En esencia, como su nombre nos lo dice, el patrón constructos nos permite tener una política general para construir nuestros objetos, centralizándolos en una clase constructora. Para traducir esto de manera más simple, esto significa que el patrón constructor define una forma estándar y consistente de cómo se deben crear los objetos en nuestros proyectos, por lo que, en lugar de tener la lógica de creación dispersa por todo el código, con este patrón se establece una regla general.

La responsabilidad de crear los objetos se concentra en una clase específica, en este caso, en la "clase constructora", dicha clase contiene toda la lógica que necesitamos para instanciar y configurar los objetos de manera correcta y simple.

Ahora bien, la principal función del patrón constructor es separar la construcción de un objeto complejo en partes, de modo que nos permita crear múltiples objetos o representaciones de esas partes separadas. Para entender esto veamos un ejemplo: Podemos representar ese objeto complejo como una computadora, para construirla necesitamos saber su marca, modelo, procesador, sistema operativo, etc. Gracias a esa separación de partes, nos facilita la construcción de más computadoras siguiendo el patrón constructor que ya definimos.

Crear objetos con múltiples características o dependencias puede llegar a ser tediosos y complejo si lo intentamos hacer desde el mismo constructor de la clase. En lugar de construir un objeto complejo de una sola vez, el patrón Constructor divide el proceso en la creación de partes individuales.

Volviendo a utilizar el ejemplo anterior: en vez de intentar crear una computadora completa en un solo paso, el patrón Constructor nos va a permitir construirla por partes, primero definiendo la marca, luego el modelo, después el procesador, y así sucesivamente.

Al tener la lógica de construcción ya organizada en una clase constructora y dividida en pasos, se vuelve mucho más sencillo crear múltiples instancias de computadoras con sus propias especificaciones.

¿Cuándo utilizarlo?

Es importante que sepamos identificar cuándo utilizar el patrón constructor en nuestro código JavaScript, ya que en muchos casos se puede crear conflictos de inflexibilidad, esto quiere decir que en caso de que más adelante necesitemos definir nuevos pasos o realizar cambios en cuanto a la lógica, también tendremos que modificar el código que definimos dentro de nuestra clase constructora. Esto puede representar un gran desafío cuando nuestra clase tiene muchas otras responsabilidades.

Dicho esto, sabemos que podemos utilizar el patrón constructor cuando:

Nuestros objetos contienen muchos parámetros opcionales

Si la creación de un objeto implica una gran cantidad de parámetros, muchos de los cuales son opcionales, nos podemos encontrar con constructores largos y difíciles de leer, haciendo el código confuso y facilitando que cometamos errores en cuanto al orden. Para evitar este problema, es muy probable que decidamos utilizar muchos constructores para manejar las diferentes combinaciones de estos parámetros, pero esto solo nos puede generar otro problema de sobrecarga y da paso a malos entendidos.

Por otro lado, si empleamos el uso del patrón constructor, podremos crear el objeto paso a paso, utilizando métodos separados para establecer cada parámetro opcional, lo que resulta en una interfaz mucho más clara y amigable con los desarrolladores.

La lógica de creación es compleja

Si el proceso de creación de un objeto implica varios pasos para crearse correctamente como validaciones o lógica condicional y esta lógica se coloca directamente dentro del constructor de la clase del objeto, lo más probable es que la clase del objeto se vuelve grande y difícil de entender, o que no podamos probar la lógica de creación, ya que se vuelve más complicado porque está entrelazada con la lógica del objeto en sí.

En cambio, al utilizar el patrón constructor abordaríamos este problema al crear una clase separada que se encarga exclusivamente de la lógica de creación del objeto complejo. La clase del objeto en sí se enfoca en su responsabilidad principal, que es representar la información y el comportamiento del objeto.

Estructura y Participantes del Patrón Constructor

El patrón constructor en JavaScript generalmente involucra los siguientes componentes:

Producto: Esta es la clase del objeto complejo que se va a construir paso a paso, en ella se contienen los atributos y los métodos del objeto final del proceso de construcción.

Constructor/Builder: Su propósito principal es definir los pasos necesarios para construir las diferentes partes del producto, actuando como una especie de contrato que los constructores concretos deben seguir. Otra de sus responsabilidades es enumerar los métodos que deben existir para poder construir el producto.

Constructor Concreto: Dado que JavaScript no es un lenguaje con interfaces explicitas como Java o TypeScript, se utiliza una clase de constructor concreto, la cual contiene la lógica real para crear cada una de las partes del producto, lo cual nos es útil cundo queremos crear diferentes representaciones o configuraciones del producto.

Director: Esta clase es opcional y es responsable de orquestar el proceso de construcción utilizando un objeto constructor. Aunque el director no sabe cómo se construyen las partes, es el responsable de indicar al constructor qué debe construir y en qué orden, justamente esto es lo que nos va a permitir utilizar el mismo proceso de construcción con diferentes constructores para obtener diferentes representaciones del producto.

```
//Producto
class User {
    constructor(firstName, lastName, age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
    getFullName() {
        return `${this.firstName} ${this.lastName}`;
    }
    getAge() {
        return this.age ? `${this.age} años` : 'Edad no especificada';
    }
}
//Constructor Concreto
class UserBuilder {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = null;
    }
    withAge(age) {
        this.age = age;
        return this;
    }
//Método para Recuperar el Producto (build())
    build() {
        return new User(this.firstName, this.lastName, this.age);
    }
}
```

Figura 1. Componentes del patrón constructor

Implementando el patrón constructor

Ahora veamos la parte práctica, para entender mejor el uso del patrón constructor, veremos dos ejemplos de un código simple, uno antes de implementar el patrón constructos y uno después de utilizarlo, comparando las diferencias y los beneficios que este patrón nos aporta.

Código antes

```
1  class Casa {
2      constructor(direccion, numHabitaciones, numBanos, tieneJardin = false, tienePiscina = false, tieneGaraje = false) {
3          this.direccion = direccion;
4          this.numHabitaciones = numHabitaciones;
5          this.numBanos = numBanos;
6          this.tieneJardin = tieneJardin;
7          this.tienePiscina = tienePiscina;
8          this.tieneGaraje = tieneGaraje;
9      }
10
11     obtenerDescripcion() {
12         let descripcion = `Casa ubicada en ${this.direccion} con ${this.numHabitaciones} habitaciones y ${this.numBanos} baños.`;
13         if (this.tieneJardin) descripcion += " Tiene jardín.";
14         if (this.tienePiscina) descripcion += " Cuenta con piscina.";
15         if (this.tieneGaraje) descripcion += " Incluye garaje.";
16         return descripcion;
17         }
18     }
19
20   // Creando instancias de la clase Casa sin el patrón constructor
21     const casaBasica = new Casa("El Crisol", 2, 1);
22     const casaConJardin = new Casa("Villa Lucre", 3, 2, true);
23     const casaCompleta = new Casa("Brisas del Golf", 5, 3, true, true, true);
24     const casaConAlgunasOpciones = new Casa("Cerro Azul", 4, 2, false, true);
25
26     console.log("Casa Básica:", casaBasica.obtenerDescripcion());
27     console.log("Casa con Jardín:", casaConJardin.obtenerDescripcion());
28     console.log("Casa Completa:", casaCompleta.obtenerDescripcion());
29     console.log("Casa con Algunas Opciones:", casaConAlgunasOpciones.obtenerDescripcion());
```

Figura 2. Ejemplo de código sin patrón constructor

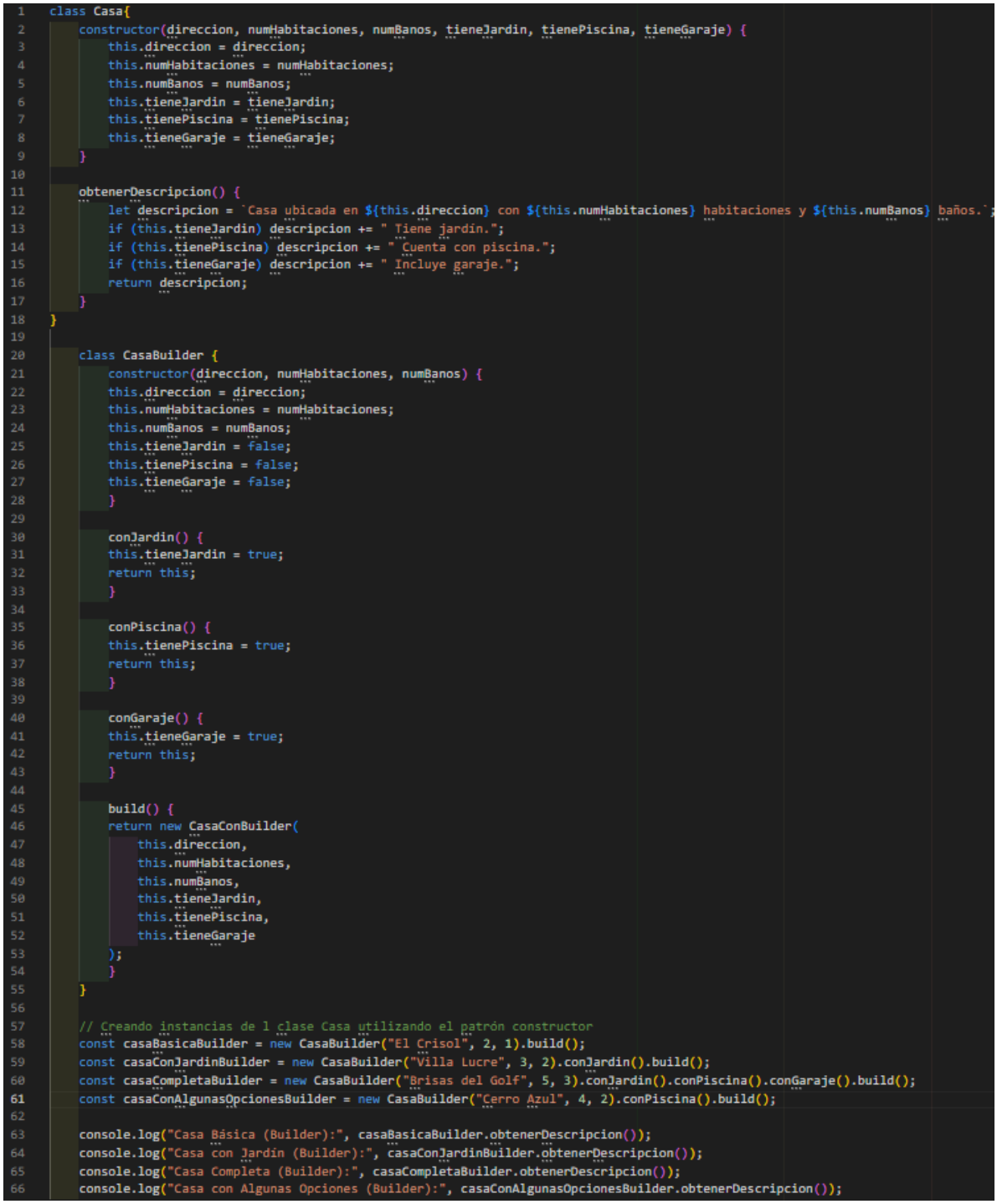
Como se puede observar en la figura 2, el constructor de Casa tiene parámetros para todas las características, incluso para las características opcionales. Al crear una casa básica, se deben pasar valores por defecto undefined para las características que no se desean, lo que puede resultar en un código menos legible.

Otra dificultad sería agregar nuevas características opcionales a la clase Casa en el futuro, las cuales podrían ser, por ejemplo, tieneSotano o tieneTerraza, esto implicaría agregar más parámetros al constructor, lo que lo volvería confuso y difícil de manejar.

Por último, si el código es extenso, se volvería complejo entender las características de la clase a la hora de crear instancias, en la línea 23 de la figura 2, se aprecia el código: const casaCompleta = new Casa("Brisas del Golf", 5, 3, true, true, true);

¿Qué representan esos números? ¿A qué se refieren los “true”? Puede resultar confuso en ciertas situaciones.

Código después



```
class Casa{
    constructor(direccion, numHabitaciones, numBanos, tieneJardin, tienePiscina, tieneGaraje) {
        this.direccion = direccion;
        this.numHabitaciones = numHabitaciones;
        this.numBanos = numBanos;
        this.tieneJardin = tieneJardin;
        this.tienePiscina = tienePiscina;
        this.tieneGaraje = tieneGaraje;
    }

    obtenerDescripcion() {
        let descripcion = `Casa ubicada en ${this.direccion} con ${this.numHabitaciones} habitaciones y ${this.numBanos} baños.`;
        if (this.tieneJardin) descripcion += " Tiene jardín.";
        if (this.tienePiscina) descripcion += " Cuenta con piscina.";
        if (this.tieneGaraje) descripcion += " Incluye garaje.";
        return descripcion;
    }
}

    class CasaBuilder {
        constructor(direccion, numHabitaciones, numBanos) {
        this.direccion = direccion;
        this.numHabitaciones = numHabitaciones;
        this.numBanos = numBanos;
        this.tieneJardin = false;
        this.tienePiscina = false;
        this.tieneGaraje = false;
        }

        conJardin() {
        this.tieneJardin = true;
        return this;
        }

        conPiscina() {
        this.tienePiscina = true;
        return this;
        }

        conGaraje() {
        this.tieneGaraje = true;
        return this;
        }

        build() {
        return new CasaConBuilder(
            this.direccion,
            this.numHabitaciones,
            this.numBanos,
            this.tieneJardin,
            this.tienePiscina,
            this.tieneGaraje
        );
        }
    }

    // Creando instancias de l clase Casa utilizando el patrón constructor
    const casaBasicaBuilder = new CasaBuilder("El Crisol", 2, 1).build();
    const casaConJardinBuilder = new CasaBuilder("Villa Lucre", 3, 2).conJardin().build();
    const casaCompletaBuilder = new CasaBuilder("Brisas del Golf", 5, 3).conJardin().conPiscina().conGaraje().build();
    const casaConAlgunasOpcionesBuilder = new CasaBuilder("Cerro Azul", 4, 2).conPiscina().build();

    console.log("Casa Básica (Builder):", casaBasicaBuilder.obtenerDescripcion());
    console.log("Casa con Jardín (Builder):", casaConJardinBuilder.obtenerDescripcion());
    console.log("Casa Completa (Builder):", casaCompletaBuilder.obtenerDescripcion());
    console.log("Casa con Algunas Opciones (Builder):", casaConAlgunasOpcionesBuilder.obtenerDescripcion());
```

Figura 3. Ejemplo de código utilizando patrón constructor

Si bien el código resulta ser más extenso, a diferencia del código de la figura 2, donde se requerían pasar valores no deseados, con el patrón constructor, la creación de la casa se hace de forma más explícita, indicando claramente qué características se incluyen mediante los métodos conJardin(), conPiscina() y conGaraje(). Otra ventaja es que agregar nuevas características no sería complejo,

ya que solo se necesitaría crear otro método como los ya existentes sin afectar el resto de la lógica.

Tampoco podemos dejar de lado la gran facilidad que nos brinda respecto a la creación de las instancias. Como mencionamos anteriormente, en el código de la figura 2, era fácil confundirse respecto a las características de la clase, en contraparte, utilizando el código de la figura 3, la creación de las instancias es mucho más fácil de leer y entender, ya que se puede ver de inmediato qué características tendrá la casa que se está construyendo.

Otra de las facilidades que obtenemos, es la separación de responsabilidades, en el ejemplo de la figura 3, la lógica de cómo se construye una casa se encuentra encapsulada dentro de la clase CasaBuilder, separada de la clase Casa que representa la casa en sí, permitiendo que el código sea vea más organizado y sea más fácil de mantener.

Patrón Módulo

Concepto

El patrón módulo consiste en un módulo donde se encapsulará toda la lógica de nuestra aplicación o proyecto. Dentro de este módulo estarán declaradas todas las variables o funciones privadas y sólo serán visibles dentro del mismo.

¿Cómo crear un patrón modulo?

Para crear un patrón módulo se declarará una única variable local que se corresponde con el nombre del módulo. Dicha variable se trata realmente de una función autoejecutable que devuelve un objeto con la funcionalidad que necesitamos, donde las variables internas se llamarán propiedades y las funciones serán métodos.

Las funciones autoejecutables son aquellas que una vez declaradas se llaman a sí mismas para inicializarse. La peculiaridad de estas funciones es que una vez inicializadas están disponibles en otras partes de la aplicación.

Una cualidad de este patrón módulo es que podemos decidir que propiedades o métodos serán públicos a través del comando return pasando de esta forma al contexto global.

Beneficios de los sistemas de módulos de JavaScript

Los sistemas de módulos en JavaScript ofrecen una serie de beneficios técnicos que contribuyen a un desarrollo más eficiente y robusto de aplicaciones. A continuación, destacaremos algunos de los principales:

Reutilización de código: Los módulos admiten encapsular y exportar componentes específicos, lo que facilita la reutilización de código en diferentes partes de una aplicación. De esta forma se evita la duplicación y mejora la mantenibilidad del código.

Legibilidad y estructura clara: Al dividir el código en módulos con funciones y responsabilidades específicas, se mejora la legibilidad y la comprensión del código. Esto facilita la colaboración entre desarrolladores y el mantenimiento a largo plazo.

Gestión de dependencias: Los sistemas de módulos son capaces de gestionar explícitamente las dependencias entre módulos. Así pues, se ayuda a identificar y resolver las dependencias necesarias, evitando problemas de acoplamiento y asegurando que las funcionalidades requeridas estén disponibles cuando se necesiten.

Encapsulación y ocultamiento de información: Los módulos permiten encapsular y ocultar información interna, exponiendo solo lo necesario a otros módulos. Gracias a ello, se promueve una mayor seguridad y evita posibles conflictos o accesos no autorizados.

Escalabilidad: Con los sistemas de módulos, las aplicaciones pueden crecer y evolucionar de manera más sencilla. Los módulos facilitan la adición y modificación de funcionalidades sin afectar a otras partes del código, lo que permite una escalabilidad más eficiente.

Aprovechando al máximo los sistemas de módulos en JavaScript

Para aprovechar al máximo los sistemas de módulos, es esencial comprender y utilizar correctamente la sintaxis y las capacidades específicas de cada sistema.

Además, se recomienda seguir buenas prácticas de organización y nomenclatura de módulos, así como optimizar el rendimiento al eliminar código no utilizado y gestionar las dependencias de manera eficiente.

En última instancia, la adopción de sistemas de módulos en el desarrollo web de JavaScript es fundamental para construir aplicaciones sostenibles. Por consiguiente, al utilizar esta arquitectura modular, los desarrolladores pueden optimizar su flujo de trabajo, mejorar la colaboración y asegurar la calidad y eficiencia del código.

¿Cúando se usa el patrón modulo?

El patrón de módulos se utiliza para implementar el concepto de módulos de software, definido por la programación modular, en un lenguaje de programación con soporte directo incompleto para el concepto.

Algunos contextos para los que se usa este patrón:

- Cuando quieres evitar **contaminar el scope global**.

  Nos servirá para no llenar el ámbito global de la aplicación con muchas variables o funciones que están disponibles en todo el programa. Esto puede causar **colisiones de nombres**, errores difíciles de detectar y baja calidad de código.

- Cuando necesitas **encapsular lógica.**

  Por ejemplo cuando necesitamos ocultar detalles internos que no deberían ser accesibles desde afuera del módulo. Esto protege los datos y permite un control total sobre cómo se acceden o modifican.

- Para mejorar la **mantenibilidad** y **reutilización** del código.

Este patrón permite dividir tu aplicación en **bloques reutilizables**, que puedes mantener fácilmente, testear de forma independiente y reutilizar en distintos proyectos.

- En aplicaciones grandes donde necesitas organizar por funcionalidades. Cuando hacemos proyectos grandes, se vuelve esencial separar el código por funcionalidades o módulos: usuarios, productos, pedidos, etc.

Ejemplos de uso

Ejemplo 1, antes de usar el patrón módulo.

Figura 4. Ejemplo de código sin utilizar patrón módulo

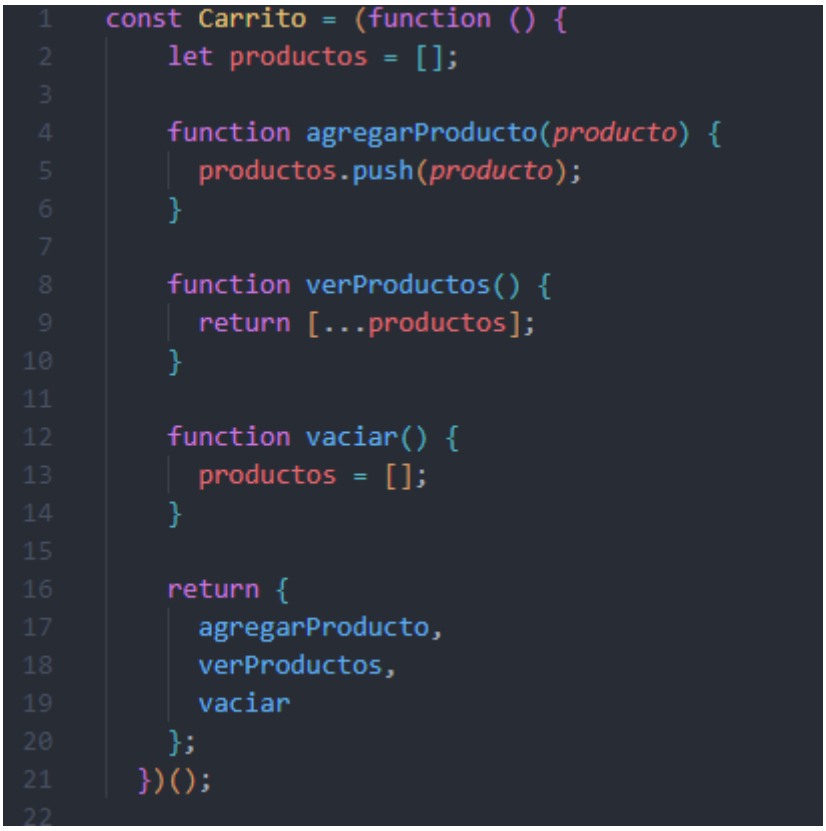
Problemas de este código:

El arreglo productos está en el scope global, cualquier parte del código puede modificarlo sin control.

No hay encapsulamiento.

Puede haber conflictos o errores si otra parte del código también usa una variable llamada productos.

Ejemplo 1, después de usar el patrón módulo



```
const Carrito = (function () {
    let productos = [];

    function agregarProducto(producto) {
      productos.push(producto);
    }

    function verProductos() {
      return [...productos];
    }

    function vaciar() {
      productos = [];
    }

    return {
      agregarProducto,
      verProductos,
      vaciar
    };
  })();
```

Figura 5. Ejemplo de código utilizando patrón módulo

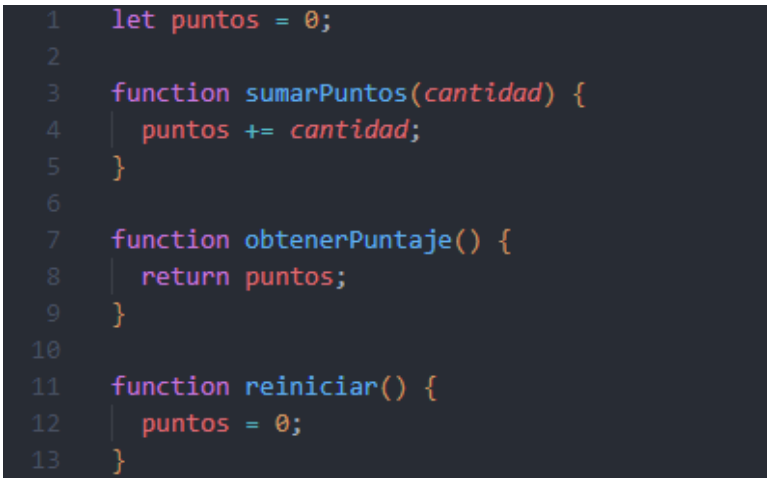
Ventajas que se tienen con esté codigo:

El arreglo de productos está protegido (encapsulado).

Solo se expone la funcionalidad necesaria.

Mejora el mantenimiento y la seguridad del código.

Ejemplo 2, antes de usar el patrón módulo.



```
let puntos = 0;

function sumarPuntos(cantidad) {
  puntos += cantidad;
}

function obtenerPuntaje() {
  return puntos;
}

function reiniciar() {
  puntos = 0;
}
```

Figura 6. Ejemplo de código 2 sin patrón módulo

Problemas de este código:

La variable puntos está en el scope global.

Cualquier código puede hacer puntos = 9999 directamente.

No hay control de validaciones

Ejemplo 2, después de usar el patrón módulo.

```
const Puntaje = (function () {
    let puntos = 0;

    function sumarPuntos(cantidad) {
      if (typeof cantidad === "number" && cantidad > 0) {
        puntos += cantidad;
      }
    }

    function obtenerPuntaje() {
      return puntos;
    }

    function reiniciar() {
      puntos = 0;
    }

    return {
      sumarPuntos,
      obtenerPuntaje,
      reiniciar
    };
  })();
```

Figura 7. Ejemplo de código 2 utilizando patrón módulo

Ventajas de este código:

La variable puntos es privada.

Se valida la entrada antes de modificar el puntaje.

Código más limpio y seguro.

Conclusiones

Odeth: Para concluir, con el desarrollo de la presente investigación, pude comprender que tanto el patrón constructor como el patrón módulo son recursos fundamentales para nuestro código JavaScript y, que con su uso adecuado, nos facilita la vida y nos resuelve muchas complicaciones. Estos dos patrones tienen diferentes utilizades, pero pueden complementarse, el patrón constructor nos ayuda a manejar los obstáculos a la hora de crear objetos con múltiples características, por lo que nos ofrece un proceso de construcción claro, haciendo que nuestro código sea flexible y fácil de mantener. Por otro lado, el patrón módulo nos sirve más que todo para , mejorar la estructura de nuestro código en unidades encapsuladas, lo que nos brinda protección ya su vez, nos facilita la reutilización y la organización de nuestros proyectos.

Moisés: Para finalizar este trabajo de investigación, puedo concluir con que la utilización de estos 2 patrones resulta especialmente valiosa en el desarrollo de proyectos de software, particularmente en aquellos que llegan a ser proyectos muy grandes. Aunque en proyectos pequeños su aplicación puede parecer innecesaria debido a la poca cantidad de código, en sistemas más complejos estos patrones permiten organizar, encapsular y proteger la lógica del programa de forma eficiente, gracias a ellos, es posible mantener un orden claro en el manejo de variables, funciones, clases y estructuras, lo que facilita la escalabilidad, la reutilización del código y la mantenibilidad a largo plazo.

Bibliografías

IONOS editorial team. (2023, 14 julio). *What is the builder pattern?* IONOS Digital Guide. https://www.ionos.com/digitalguide/websites/web-development/what-is-the-builder-pattern/

Builder. (s. f.). https://refactoring.guru/es/design-patterns/builder#:~:text=El%20patr%C3%B3n%20Builder%20se%20puede,s%C3%B3lo%20var%C3%ADan%20en%20los%20detalles.

Durán, E. (2019, 18 marzo). *Building objects progressively with the builder pattern in javascript*. https://enmascript.com/articles/2019/03/18/building-objects-progressively-with-the-builder-pattern-in-javascript/

Wiar8. (2024, 24 marzo). *Construye mejores objetos con BUILDER | Patrones de Diseño | Design Patterns* [Vídeo]. YouTube. https://www.youtube.com/watch?v=L2YtRmG9Ibs

Marluan Espiritusanto. (2020, 31 marzo). *04. Patrones de diseño: Patrón Builder* [Vídeo]. YouTube. https://www.youtube.com/watch?v=5et6PxMDVXo

Dofactory. (s. f.). *JavaScript Builder Design pattern*. https://www.dofactory.com/javascript/design-patterns/builder

Parvati, P. (2023, 25 marzo). *Design pattern - Builder*. DEV Community. https://dev.to/pratikparvati/design-pattern-builder-jcj

Fernández, G. (26 de Abril de 2018). *El Patrón módulo en JavaScript*. Obtenido de Medium: https://medium.com/@gloriafercu/el-patrón-módulo-en-javascript-1cd012a30ad

Guía completa sobre los sistemas de módulos de JavaScript. (15 de Mayo de 2023). Obtenido de VGS: https://vgst.net/blog/development/sistemas-de-modulos-de-javascript