Aprendizaje de Patrones de Diseño de JavaScript

Un libro de Addy Osmani

Volumen 1.7.0

Prefacio

Los patrones de diseño son soluciones reutilizables para problemas comunes en el diseño de software. Son temas emocionantes y fascinantes para explorar en cualquier lenguaje de programación.
Una razón para esto es que nos ayudan a construir sobre la experiencia combinada de muchos desarrolladores que nos precedieron y aseguran que estructuramos nuestro código de manera optimizada, satisfaciendo las necesidades de los problemas que intentamos resolver.
Los patrones de diseño también nos proporcionan un vocabulario común para describir soluciones. Esto puede ser significativamente más simple que describir la sintaxis y la semántica cuando intentamos transmitir a los demás una forma de estructurar una solución en forma de código.

En este libro exploraremos la aplicación de patrones de diseño clásicos y modernos al lenguaje de programación JavaScript.

Público Objetivo

Este libro está dirigido a desarrolladores profesionales que deseen mejorar su conocimiento de los patrones de diseño y cómo se pueden aplicar al lenguaje de programación JavaScript. Algunos de los conceptos cubiertos (cierres, herencia prototípica) asumirán un nivel de conocimiento previo básico y comprensión. Si necesita leer más sobre estos temas, se proporciona una lista de títulos sugeridos para su comodidad.

Si desea aprender a escribir código hermoso, estructurado y organizado, creo que este es el libro para usted.

Agradecimientos

Siempre estaré agradecido por los talentosos revisores técnicos que ayudaron a revisar y mejorar este libro, incluidos los de la comunidad en general. El conocimiento y el entusiasmo que aportaron al proyecto fue simplemente increíble. Los tweets y blogs oficiales de los revisores técnicos también son una fuente regular de ideas e inspiración y recomiendo sinceramente que los revisen.

- Nicholas Zakas (http://nczonline.net, @slicknet)
- Andrée Hansson (http://andreehansson.se, @peolanha)
- Luke Smith (http://lucassmith.name, @ls_n)
- Eric Ferraiuolo (http://ericf.me/, @ericf)
- Peter Michaux (http://michaux.ca, @petermichaux)
- Alex Sexton (http://alexsexton.com, @slexaxton)

También me gustaría agradecer a Rebecca Murphey (http://rmurphey.com, @rmurphey) por proporcionar la inspiración para escribir este libro y, lo que es más importante, seguir haciéndolo disponible en GitHub y en O'Reilly.

Finalmente, me gustaría agradecer a mi maravillosa esposa Ellie, por todo su apoyo mientras preparaba esta publicación.

Créditos

Si bien algunos de los patrones cubiertos en este libro se implementaron en base a la experiencia personal, muchos de ellos han sido identificados previamente por la comunidad JavaScript. Este trabajo es como tal la producción de la experiencia combinada de varios desarrolladores. Similar al enfoque lógico de Stoyan Stefanov para evitar la interrupción de la narrativa con créditos (en patrones de JavaScript), he enumerado créditos y sugerido leer cualquier contenido cubierto en la sección de referencias.

Si se ha perdido algún artículo o enlace en la lista de referencias, acepte mis más sinceras disculpas. Si me contactas, me aseguraré de actualizarlos para incluirte en la lista.

Leyendo

Si bien este libro está dirigido a desarrolladores principiantes e intermedios, se asume una comprensión básica de los fundamentos de JavaScript. Si desea obtener más información sobre el idioma, me complace recomendar los siguientes títulos:

- *JavaScript: The Definitive Guide* by David Flanagan
- *Eloquent JavaScript* by Marijn Haverbeke
- *JavaScript Patterns* by Stoyan Stefanov
- *Writing Maintainable JavaScript* by Nicholas Zakas
- *JavaScript: The Good Parts* by Douglas Crockford

Tabla de Contenido

- Patrones de Diseño del Complemento jQuery
- Patrones de Espacio de Nombres JavaScript
- Conclusiones
- Referencias

Introducción

Uno de los aspectos más importantes de escribir código mantenible es ser capaz de notar los temas recurrentes en ese código y optimizarlos. Esta es un área donde el conocimiento de los patrones de diseño puede resultar invaluable.

En la primera parte de este libro, exploraremos la historia y la importancia de los patrones de diseño que realmente se pueden aplicar a cualquier lenguaje de programación. Si ya está vendido o está familiarizado con esta historia, no dude en pasar al capítulo "¿Qué es un patrón?" para seguir leyendo.

Los patrones de diseño se remontan a los primeros trabajos de un arquitecto llamado Christopher Alexander. A menudo escribía publicaciones sobre su experiencia en la resolución de problemas de diseño y cómo se relacionaban con edificios y ciudades. Un día, se le ocurrió a Alexander que cuando se usa una y otra vez, ciertas construcciones de diseño conducen a un efecto óptimo deseado.

En colaboración con Sara Ishikawa y Murray Silverstein, Alexander produjo un lenguaje de patrones que ayudaría a empoderar a cualquiera que desee diseñar y construir a cualquier escala. Esto fue publicado en 1977 en un artículo titulado "A Pattern Language", que luego fue lanzado como un libro completo de tapa dura.

Hace unos 30 años, los ingenieros de software comenzaron a incorporar los principios sobre los que Alexander había escrito en la primera documentación sobre patrones de diseño, que sería una guía para los desarrolladores novatos que buscan mejorar sus habilidades de codificación. Es importante tener en cuenta que los conceptos detrás de los patrones de diseño han existido en la industria de la programación desde su inicio, aunque de forma menos formal.

Uno de los primeros trabajos y más formales icónicos posiblemente publicados sobre patrones de diseño en ingeniería de software fue un libro en 1995 titulado Patrones de diseño: elementos de software orientado a objetos reutilizables. Esto fue escrito por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides - un grupo que se hizo conocido como la Banda de los Cuatro (the Gang of Four o GoF para abreviar).

La publicación del GoF se considera bastante instrumental para impulsar aún más el concepto de patrones de diseño en nuestro campo, ya que describe una serie de técnicas de desarrollo y dificultades, así como también proporciona veintitrés patrones de diseño orientados a objetos centrales que se usan con frecuencia en todo el mundo en la actualidad. Cubriremos estos patrones con más detalle en la sección "Categorías de patrones de diseño".

En este libro, veremos una serie de patrones de diseño de JavaScript populares y exploraremos por qué ciertos patrones pueden ser más adecuados para sus proyectos que otros. Recuerde que los patrones se pueden aplicar no solo a JavaScript estándar (es decir, código JavaScript estándar), sino

también a bibliotecas abstractas como jQuery or dojo. Antes de comenzar, veamos la definición exacta de un "patrón" en el diseño de software.

¿Qué es un patrón?

Un patrón es una solución reutilizable que se puede aplicar a problemas comunes en el diseño de software, en nuestro caso, al escribir aplicaciones web JavaScript. Otra forma de ver los patrones son las plantillas de cómo resolvemos los problemas, que se pueden usar en diferentes situaciones. Entonces, ¿por qué es importante entender los patrones y estar familiarizado con ellos? Los patrones de diseño tienen tres beneficios principales:

1. **Los patrones son soluciones probadas:** Proporcionan enfoques sólidos para resolver problemas en el desarrollo de software utilizando técnicas comprobadas que reflejan la experiencia y los conocimientos que los desarrolladores ayudaron a definirlos para llevarlos al patrón.
2. **Los patrones se pueden reutilizar fácilmente:** Un patrón generalmente refleja una solución lista para usar que se puede adaptar para satisfacer nuestras propias necesidades. Esta característica los hace bastante robustos.
3. **Los patrones pueden ser expresivos:** Cuando observamos un patrón, generalmente hay una estructura establecida y un *vocabulario* para la solución presentada que puede ayudar a expresar soluciones bastante grandes con bastante elegancia.

Los patrones **no** son una solución exacta. Es importante que recordemos que el papel de un patrón es simplemente proporcionarnos un esquema de solución. Los patrones no resuelven todos los problemas de diseño ni reemplazan a los buenos diseñadores de software, sin embargo, los **apoyan**. A continuación, veremos algunas de las otras ventajas que los patrones tienen para ofrecer.

- **La reutilización de patrones ayuda a prevenir problemas menores que pueden causar problemas importantes en el proceso de desarrollo de aplicaciones.** Esto significa que cuando el código se basa en patrones probados, podemos permitirnos pasar menos tiempo preocupándonos por la estructura de nuestro código y más tiempo enfocándonos en la calidad de nuestra solución general. Esto se debe a que los patrones pueden alentarnos a codificar de manera más estructurada y organizada, evitando la necesidad de refactorizarlo para fines de limpieza en el futuro.
- **Los patrones pueden proporcionar soluciones generalizadas que están documentadas de una manera que no requiere que estén vinculadas a un problema específico.** Este enfoque generalizado significa que, independientemente de la aplicación (y en muchos casos el lenguaje de programación) con el que estamos trabajando, los patrones de diseño se pueden aplicar para mejorar la estructura de nuestro código.
- **Ciertos patrones en realidad pueden disminuir la huella general del tamaño de archivo de nuestro código al evitar la repetición.** Al alentar a los desarrolladores a mirar más de cerca sus soluciones para áreas donde se pueden hacer reducciones instantáneas en la

repetición, ej. Al reducir el número de funciones que realizan procesos similares en favor de una sola función generalizada, el tamaño total de nuestra base de código se puede disminuir. Esto también se conoce como hacer que el código sea más DRY (seco).

- **Los patrones se agregan al vocabulario de un desarrollador, lo que agiliza la comunicación.**
- **Los patrones que se usan con frecuencia se pueden mejorar con el tiempo aprovechando las experiencias colectivas que otros desarrolladores que usan esos patrones contribuyen a la comunidad de patrones de diseño.** En algunos casos, esto conduce a la creación de patrones de diseño completamente nuevos, mientras que en otros puede conducir a la provisión de pautas mejoradas sobre cómo los patrones específicos se pueden utilizar mejor. Esto puede garantizar que las soluciones basadas en patrones continúen siendo más sólidas de lo que pueden ser las soluciones ad-hoc.

Ya usamos patrones todos los días

Para comprender cuán útiles pueden ser los patrones, repasemos un problema de selección de elementos muy simple que la biblioteca jQuery nos resuelve..

Imagine que tenemos un script donde para cada elemento DOM encontrado en una página con la clase "foo" deseamos incrementar un contador. ¿Cuál es la forma más eficiente de consultar esta colección de elementos? Bueno, hay algunas maneras diferentes de abordar este problema:

1. Seleccione todos los elementos en la página y luego almacene referencias a ellos. Luego, filtre esta colección y use expresiones regulares (u otro medio) para almacenar solo aquellos con la clase "foo".
2. Use una característica moderna del navegador nativo como `querySelectorAll()` para seleccionar todos los elementos con la clase "foo".
3. Use una función nativa como `getElementsByClassName()` para recuperar de manera similar la colección deseada. alternatives

Entonces, ¿cuál de estas opciones es la más rápida? En realidad, es la opción 3, por un factor de 8-10 veces las alternativas. Sin embargo, en una aplicación del mundo real, 3 no funcionará en versiones de Internet Explorer inferiores a 9 y, por lo tanto, es necesario usar 1 donde no se admiten 2 y 3.

Sin embargo, los desarrolladores que usan jQuery no tienen que preocuparse por este problema, ya que afortunadamente se nos abstrae usando el patrón *Facade*. Como revisaremos con más detalle más adelante, este patrón proporciona un conjunto simple de interfaces abstraídas (ej. `$el.css()`, `$el.animate()`) a varios cuerpos de código subyacentes más complejos. Como hemos visto, esto significa menos tiempo para preocuparse por los detalles del nivel de implementación.

Detrás de escena, la biblioteca simplemente opta por el enfoque más óptimo para seleccionar elementos dependiendo de lo que admita nuestro navegador actual y solo consumimos la capa de abstracción.

Probablemente todos también estamos familiarizados con `$("selector")` de jQuery. Esto es significativamente más fácil de usar para seleccionar elementos HTML en una página en lugar de tener que optar manualmente por `getElementById()`, `getElementsByClassName()`, `getElementsByTagName()` y así sucesivamente.

Although we know that `querySelectorAll()` attempts to solve this problem, compare the effort involved in using jQuery's Facade interfaces vs. selecting the most optimal selection paths ourselves. There's no contest! Abstractions using patterns can offer real-world value.

Aunque sabemos que `querySelectorAll()` intenta resolver este problema, compare el esfuerzo involucrado en el uso de las interfaces Facade de jQuery versus seleccionando nosotros mismos los caminos de selección más óptimos. ¡No hay concurso! Las abstracciones que usan patrones pueden ofrecer valor en el mundo real.

Veremos esto y más patrones de diseño más adelante en el libro.

"Pruebas de "patrón", prototipos y la regla de tres.

Recuerde que no todos los algoritmos, mejores prácticas o soluciones representan lo que podría considerarse un patrón completo. Puede haber algunos ingredientes claves que faltan y la comunidad de patrones generalmente desconfía de algo que dice ser uno a menos que haya sido investigado en gran medida. Incluso si se nos presenta algo que **parece** cumplir con los criterios para un patrón, no debe considerarse uno hasta que haya pasado por períodos adecuados de escrutinio y prueba por parte de otros.

Al recordar el trabajo de Alexander una vez más, afirma que un patrón debería ser tanto un proceso como una "cosa". Esta definición es obtusa a propósito, ya que sigue diciendo que es el proceso el que debería crear la "cosa". Esta es una razón por la cual los patrones generalmente se enfocan en abordar una estructura visualmente identificable, es decir, deberíamos poder representar visualmente (o dibujar) una imagen que represente la estructura en la que se pone en práctica el patrón.

Al estudiar patrones de diseño, no es irregular encontrar el término "proto-patrón". ¿Que es esto? Bueno, un patrón que aún no se sabe que pasa las pruebas de calidad de "patrón" generalmente se conoce como un proto-patrón. Los proto-patrones pueden ser el resultado del trabajo de alguien que ha establecido una solución particular que es digna de compartir con la comunidad, pero que aún no ha tenido la oportunidad de haber sido examinada en gran medida debido a su corta edad.

Alternativamente, las personas que comparten el patrón pueden no tener el tiempo o el interés de pasar por el proceso de "patrón" y, en su lugar, pueden publicar una breve descripción de su proto-patrón. Breves descripciones o fragmentos de este tipo de patrón se conocen como paletas.

El trabajo involucrado en documentar completamente un patrón calificado puede ser bastante desalentador. Mirando hacia atrás en algunos de los primeros trabajos en el campo de los patrones de diseño, un patrón puede considerarse "bueno" si hace lo siguiente:

- **Resuelve un problema particular**: Se supone que los patrones no solo capturan principios o estrategias. Necesitan capturar soluciones. Este es uno de los ingredientes más esenciales para un buen patrón.
- **La solución a este problema no puede ser obvia**: Podemos encontrar que las técnicas de resolución de problemas a menudo intentan derivar de los primeros principios bien conocidos. Los mejores patrones de diseño generalmente brindan soluciones indirectas a los problemas. Esto se considera un enfoque necesario para los problemas más desafiantes relacionados con el diseño.
- **El concepto descrito debe haber sido probado**: Los patrones de diseño requieren pruebas de que funcionan como se describe y, sin esta prueba, el diseño no puede considerarse seriamente. Si un patrón es de naturaleza altamente especulativa, solo los valientes pueden intentar usarlo.

- **Debe describir una relación**: En algunos casos, puede parecer que un patrón describe un tipo de módulo. Aunque una implementación puede aparecer de esta manera, la descripción oficial del patrón debe describir estructuras y mecanismos de sistema mucho más profundos que explican su relación con el código.

Seríamos perdonados por pensar que no vale la pena aprender un proto-patrón que no cumple con las pautas, sin embargo, esto está lejos de la verdad. Muchos proto-patrones son realmente bastante buenos. No estoy diciendo que valga la pena mirar todos los proto-patrones, pero hay bastantes útiles en la naturaleza que podrían ayudarnos con futuros proyectos. Use el mejor criterio con la lista anterior en mente y estará bien en su proceso de selección.

Uno de los requisitos adicionales para que un patrón sea válido es que muestran algún fenómeno recurrente. Esto es a menudo algo que puede calificarse en al menos tres áreas clave, lo que se conoce como la *regla de tres*. Para mostrar la recurrencia usando esta regla, uno debe demostrar:

1. **Aptitud de propósito** - ¿cómo se considera exitoso el patrón?
2. **Utilidad** - ¿por qué el patrón se considera exitoso?
3. **Aplicabilidad** - ¿el diseño es digno de ser un patrón porque tiene una aplicabilidad más amplia? Si es así, esto necesita ser explicado. Al revisar o definir un patrón, es importante tener en cuenta lo anterior.

La estructura de un patrón de diseño

Es posible que tenga curiosidad acerca de cómo un autor de patrones podría abordar la estructura, la implementación y el propósito de un nuevo patrón. Un patrón se presenta inicialmente en forma de una **regla** que establece una relación entre:

- Un **contexto**
- Un sistema de **fuerzas** que surge en ese contexto y
- Una **configuración** que permite que estas fuerzas se resuelvan en contexto

Con esto en mente, echemos un vistazo a un resumen de los elementos componentes de un patrón de diseño. Un patrón de diseño debe tener un:

- **Nombre del patrón** y una **descripción**
- **Trazado del contexto** - los contextos en los que el patrón es eficaz para responder a las necesidades de los usuarios
- **Declaración del problema** - una declaración del problema que se está abordando para que podamos entender la intención del patrón
- **Solution** – una descripción de cómo se resuelve el problema del usuario en una lista comprensible de pasos y percepciones
- **Diseño** – una descripción del diseño del patrón y, en particular, el comportamiento del usuario al interactuar con él
- **Implementación** – una guía sobre cómo se implementaría el patrón
- **Ilustraciones** – una representación visual de las clases en el patrón (ej., un diagrama)
- **Ejemplos** – una implementación del patrón en una forma mínima
- **Co-requisitos** – ¿qué otros patrones pueden ser necesarios para apoyar el uso del patrón que se describe?
- **Relaciones** – ¿a qué patrones se parece este patrón? ¿se parece mucho a los demás?
- **Uso conocido** – ¿Se está utilizando el patrón en la *naturaleza*? si es así, ¿dónde y cómo?
- **Discusiones** – Las ideas del equipo o del autor sobre los interesantes beneficios del patrón

Los patrones de diseño son un enfoque bastante poderoso para lograr que todos los desarrolladores de una organización o equipo estén en la misma página al crear o mantener soluciones. Si está considerando trabajar en un patrón propio, recuerde que aunque pueden tener un alto costo inicial en las fases de planificación y redacción, el valor devuelto de esa inversión puede valer la pena. Sin embargo, siempre investigue a fondo antes de trabajar en nuevos patrones, ya que puede resultarle más beneficioso usar o construir sobre los patrones probados existentes que comenzar de nuevo.

Escribir patrones de Diseño

Aunque este libro está dirigido a los nuevos patrones de diseño, una comprensión fundamental de cómo se escribe un patrón de diseño puede ofrecer una serie de beneficios útiles. Para empezar, podemos obtener una apreciación más profunda del razonamiento detrás de por qué se necesita un patrón. También podemos aprender a saber si un patrón (o proto-patrón) está a la altura cuando lo revisamos para nuestras propias necesidades.

Escribir buenos patrones es una tarea desafiante. Los patrones no solo necesitan (idealmente) proporcionar una cantidad sustancial de material de referencia para los usuarios finales, sino que también deben poder defender por qué son necesarios.

Después de leer la sección anterior sobre qué es un patrón, podemos pensar que esto en sí mismo es suficiente para ayudarnos a identificar patrones que vemos en la naturaleza. Esto en realidad no es completamente cierto. No siempre está claro si un fragmento de código que estamos viendo sigue un patrón establecido o si accidentalmente aparece como lo hace.

Cuando observamos un cuerpo de código que creemos que puede estar usando un patrón, deberíamos considerar escribir algunos de los aspectos del código que creemos que se incluyen en un patrón o conjunto de patrones existente en particular.

En muchos casos de análisis de patrones, podemos encontrar que solo estamos viendo código que sigue buenos principios y prácticas de diseño que podrían superponerse con las reglas de un patrón por accidente. Recuerde: las soluciones en las que no aparecen interacciones ni reglas definidas *no* son patrones.

Si está interesado en aventurarse en el camino de escribir sus propios patrones de diseño, le recomiendo aprender de otros que ya han pasado por el proceso y lo han hecho bien. Pase tiempo absorbiendo la información de varias descripciones de patrones de diseño diferentes y asimile lo que es significativo para usted.

Explore la estructura y la semántica: esto se puede hacer examinando las interacciones y el contexto de los patrones que le interesan para que pueda identificar los principios que ayudan a organizar esos patrones en configuraciones útiles.

Una vez que nos hayamos expuesto a una gran cantidad de información sobre literatura de patrones, es posible que deseemos comenzar a escribir nuestro patrón utilizando un formato *existente* y ver si podemos generar nuevas ideas para mejorarlo o integrar nuestras ideas allí.

Un ejemplo de un desarrollador que hizo esto en los últimos años es Christian Heilmann, quien tomó el existente patrón del *Módulo* e hizo algunos cambios fundamentalmente útiles para crear el patrón del *Módulo Revelador* (este es uno de los patrones cubiertos más adelante en este libro).

Los siguientes son consejos que sugeriría si está interesado en crear un nuevo patrón de diseño:

- **¿Qué tan práctico es el patrón?**: Asegúrese de que el patrón describa soluciones comprobadas a problemas recurrentes en lugar de solo soluciones especulativas que no han sido calificadas.

- **Tenga en cuenta las mejores prácticas:** Las decisiones de diseño que tomamos deben basarse en los principios que derivamos de la comprensión de las mejores prácticas.
- **Nuestros patrones de diseño deben ser transparentes para el usuario**: Los patrones de diseño deben ser completamente transparentes para cualquier tipo de experiencia del usuario. Están principalmente allí para servir a los desarrolladores que los usan y no deben forzar cambios en el comportamiento en la experiencia del usuario que no se incurriría sin el uso de un patrón.
- **Recuerde que la originalidad no es clave en el diseño del patrón**: Al escribir un patrón, no necesitamos ser el descubridor original de las soluciones que se están documentando ni preocuparse por la superposición de nuestro diseño con piezas menores de otros patrones. Si el enfoque es lo suficientemente fuerte como para tener una aplicabilidad amplia y útil, tiene la posibilidad de ser reconocido como un patrón válido.
- **Los patrones necesitan un sólido conjunto de ejemplos:** Una buena descripción del patrón debe ser seguida por un conjunto igualmente sólido de ejemplos que demuestren la aplicación exitosa de nuestro patrón. Para mostrar un uso amplio, los ejemplos que exhiben buenos principios de diseño son ideales.

La escritura de patrones es un equilibrio cuidadoso entre crear un diseño que sea general, específico y, sobre todo, útil. Intente asegurarse de que, si escribe un patrón, cubra las áreas de aplicación más amplias posibles y esté bien. Espero que esta breve introducción a los patrones de escritura te haya dado algunas ideas que te ayudarán en tu proceso de aprendizaje en las próximas secciones de este libro.

Antipatrones

Si consideramos que un patrón representa una mejor práctica, un antipatrón representa una lección que se ha aprendido. El término antipatrones fue acuñado en 1995 por Andrew Koenig en el Informe C ++ de noviembre de ese año, inspirado en el libro de GoF *Design Patterns*. En el informe de Koenig, se presentan dos nociones de antipatrones. Antipatrones:

- Describa una *mala* solución a un problema particular que resultó en una mala situación
- Describa *cómo* salir de dicha situación y cómo pasar de allí a una buena solución.

Sobre este tema, Alexander escribe sobre las dificultades para lograr un buen equilibrio entre una buena estructura de diseño y un buen contexto:

"Estas notas son sobre el proceso de diseño; el proceso de inventar cosas físicas que muestran un nuevo orden físico, organización, forma, en respuesta a la función ... cada problema de diseño comienza con un esfuerzo por lograr la adecuación entre dos entidades: la forma en cuestión y su contexto. La forma es la solución al problema; el contexto define el problema".

Si bien es bastante importante conocer los patrones de diseño, puede ser igualmente importante comprender los antipatrones. Califiquemos la razón detrás de esto. Al crear una aplicación, el ciclo de vida de un proyecto comienza con la construcción, sin embargo, una vez que haya realizado el lanzamiento inicial, debe mantenerse. La calidad de una solución final será buena o mala, dependiendo del nivel de habilidad y el tiempo que el equipo haya invertido en ella. Aquí, lo bueno y lo malo se consideran en contexto - un diseño 'perfecto' puede calificar como un antipatrón si se aplica en el contexto incorrecto.

Los desafíos más grandes suceden después de que una aplicación ha alcanzado la producción y está lista para entrar en modo de mantenimiento. Un desarrollador que trabaje en un sistema de este tipo y que no haya trabajado antes en la aplicación puede introducir un *mal* diseño en el proyecto por accidente. Si dichas *malas* prácticas se crean como antipatrones, les permiten a los desarrolladores un medio para reconocerlas de antemano para que puedan evitar errores comunes que pueden ocurrir; esto es paralelo a la forma en que los patrones de diseño nos proporcionan una forma de reconocer las técnicas comunes que son *útiles*.

Para resumir, un antipatrón es un mal diseño que es digno de documentarse. Ejemplos de antipatrones en JavaScript son los siguientes:

- Contaminar el espacio de nombres global definiendo una gran cantidad de variables en el contexto global
- Pasar cadenas en lugar de funciones a setTimeout o setInterval ya que esto activa el uso de `eval()` internamente
- Modificación del prototipo de la clase `Object` (este es un antipatrón particularmente malo)
- Usando JavaScript en un formulario en línea ya que esto es inflexible
- El uso de document.write donde las alternativas DOM nativas como document.createElement son más apropiadas. document.write ha sido mal utilizado durante años y tiene algunas desventajas, incluyendo que si se ejecuta después de cargar la página, puede sobrescribir la página en la que estamos, mientras que document.createElement no.

Podemos ver aqui un ejemplo en vivo de esto en acción. Tampoco funciona con XHTML, que es otra razón por la que optar por métodos más compatibles con DOM, como document.createElement, es favorable.

El conocimiento de los antipatrones es crítico para el éxito. Una vez que podemos reconocer tales antipatrones, podemos refactorizar nuestro código para negarlos, de modo que la calidad general de nuestras soluciones mejore instantáneamente.

Categorías de Patrones de Diseño

Un glosario del conocido libro de diseño, *Domain-Driven Terms* , establece con razón que:

"*Un patrón de diseño nombra, resume e identifica los aspectos clave de una estructura de diseño común que lo hace útil para crear un diseño orientado a objetos reutilizable. El patrón de diseño identifica las clases participantes y sus instancias, sus roles y colaboraciones, y la distribución de responsabilidades.*

Cada patrón de diseño se centra en un asunto o problema de diseño orientado a objetos en particular. Describe cuándo se aplica, si se puede aplicar o no en vista de otras restricciones de diseño, y las consecuencias y compensaciones de su uso. Dado que eventualmente debemos implementar nuestros diseños, un patrón de diseño también proporciona un código de muestra ... para ilustrar una implementación.

Aunque los patrones de diseño describen diseños orientados a objetos, se basan en soluciones prácticas que se han implementado en lenguajes de programación orientados a objetos convencionales...."

Los patrones de diseño se pueden dividir en varias categorías diferentes. En esta sección revisaremos tres de estas categorías y mencionaremos brevemente algunos ejemplos de los patrones que se incluyen en estas categorías antes de explorar las específicas con más detalle.

Patrones de Diseño Creacionales (Creational)

Los patrones de diseño creacionales se centran en manejar los mecanismos de creación de objetos donde los objetos se crean de manera adecuada para la situación en la que estamos trabajando. El enfoque básico para la creación de objetos podría conducir a una mayor complejidad en un proyecto, mientras que estos patrones apuntan a resolver este problema *controlando* el proceso de creación.

Algunos de los patrones que entran en esta categoría son: Constructor, Factory, Abstract, Prototype, Singleton y Builder.

Patrones de Diseño Estructurales (Structural)

Los patrones estructurales se refieren a la composición de objetos y típicamente identifican formas simples de realizar relaciones entre diferentes objetos. Ayudan a garantizar que cuando una parte de un sistema cambia, toda la estructura del sistema no necesita hacer lo mismo. También ayudan a refundir partes del sistema que no se ajustan a un propósito particular en los que sí lo hacen.

Los patrones que entran en esta categoría incluyen : Decorator, Facade, Flyweight, Adapter y Proxy.

Patrones de Diseño Conductuales (Behavioral)

Los patrones de comportamiento se centran en mejorar o racionalizar la comunicación entre objetos dispares en un sistema.

Algunos patrones de comportamiento incluyen: Iterator, Mediator, Observer and Visitor.

Categorización de Patrones de Diseño

En mis primeras experiencias de aprendizaje sobre patrones de diseño, personalmente encontré en la siguiente tabla un recordatorio muy útil de lo que una serie de patrones tiene para ofrecer: cubre los 23 patrones de diseño mencionados por GoF. La tabla original fue resumida por Elyse Nielsen en 2004 y la modifiqué cuando fue necesario para adaptarla a nuestra discusión en esta sección del libro. Recomiendo usar esta tabla como referencia, pero recuerde que hay una serie de patrones adicionales que no se mencionan aquí, pero que se analizarán más adelante en el libro.

Una breve nota sobre las clases.

Nota: ES2015 introdujo el soporte nativo para las clases de JavaScript, sin embargo, son principalmente azúcar sintáctica sobre el modelo de herencia basado en prototipos existente de JavaScript. No cubriremos las clases de ES2015 en este libro, pero **MDN** tiene una excelente introducción a ellas.

Tenga en cuenta que habrá patrones en esta tabla que hacen referencia al concepto de "clases". En ES5, JavaScript es un lenguaje sin clases, sin embargo, las clases se pueden simular mediante funciones.

El enfoque más común para lograr esto es definiendo una función de JavaScript donde luego creamos un objeto usando la palabra clave `new`.

`this` se puede utilizar para ayudar a definir nuevas propiedades y métodos para el objeto de la siguiente manera:

```
// A car "class"
function Car( model ) {

  this.model = model;
  this.color = "silver";
  this.year = "2012";

  this.getInfo = function () {
    return this.model + " " + this.year;
  };

}
```

Entonces podemos instanciar el objeto usando el constructor Car que definimos anteriormente de esta manera: ?

```
var myCar = new Car("ford");

myCar.year = "2010";

```

```
5 console.log( myCar.getInfo() );
```

Para más formas de definir "clases" usando JavaScript, vea la publicación útil de Stoyan Stefanov sobre ellas.

Pasemos ahora a revisar la tabla.

Creational	Basado en el concepto de crear un objeto.
Class	
Factory Method	Esto hace una instancia de varias clases derivadas basadas en datos o eventos interconectados.
Object	
Abstract Factory	Crea una instancia de varias familias de clases sin detallar clases concretas.
Builder	Separa la construcción de objetos de su representación, siempre crea el mismo tipo de objeto.
Prototype	Una instancia completamente inicializada utilizada para copiar o clonar.
Singleton	Una clase con una sola instancia con puntos de acceso global.
Structural	Basado en la idea de construir bloques de objetos.
Class	
Adapter	Haga coincidir las interfaces de diferentes clases, por lo tanto las clases pueden funcionar juntas a pesar de las interfaces incompatibles.
Object	
Adapter	Haga coincidir las interfaces de diferentes clases, por lo tanto las clases pueden funcionar juntas a pesar de las interfaces incompatibles.
Bridge	Separa la interfaz de un objeto de su implementación para que los dos puedan variar independientemente.
Composite	Una estructura de objetos simples y compuestos que hace que el objeto total sea más que la suma de sus partes.
Decorator	Agregue dinámicamente procesamiento alternativo a objetos.
Facade	Una simple clase que oculta la complejidad de un subsistema completo.
Flyweight	Una instancia detallada utilizada para compartir de manera eficiente la información contenida en otro lugar.
Proxy	Un objeto marcador de posición que representa el objeto verdadero.
Behavioral	Basado en la forma en que los objetos juegan y trabajan juntos.
Class	
Interpreter	Una forma de incluir elementos del lenguaje en una aplicación para que coincida con la gramática del idioma deseado.
Template Method	Crea la capa de un algoritmo en un método, luego difiere los pasos exactos a una subclase
Object	
Chain of Responsibility	Una forma de pasar una solicitud entre una cadena de objetos para encontrar el objeto que puede manejar la solicitud.

Command	Encapsula una solicitud de comando como un objeto para habilitar, registrar y / o poner en cola solicitudes, y proporciona manejo de errores para solicitudes no manejadas.
Iterator	Acceda secuencialmente a los elementos de una colección sin conocer el funcionamiento interno de la colección.
Mediator	Define la comunicación simplificada entre clases para evitar que un grupo de clases se refiera explícitamente entre sí.
Memento	Captura el estado interno de un objeto para poder restaurarlo más tarde.
Observer	Una forma de notificar los cambios en varias clases para garantizar la coherencia entre las clases..
State	Alterar el comportamiento de un objeto cuando cambia su estado.
Strategy	Encapsula un algoritmo dentro de una clase que separa la selección de la implementación.
Visitor	Agrega una nueva operación a una clase sin cambiar la clase.

Patrones de Diseño JavaScript

En esta sección, exploraremos las implementaciones de JavaScript de varios patrones de diseño clásicos y modernos.

Los desarrolladores comúnmente se preguntan si existe un patrón ideal o un conjunto de patrones que deberían usar en su flujo de trabajo. No hay una verdadera respuesta única a esta pregunta; Es probable que cada script y aplicación web en la que trabajemos tenga sus propias necesidades individuales y debemos pensar dónde creemos que un patrón puede ofrecer un valor real a una implementación.

Por ejemplo, algunos proyectos pueden beneficiarse de los beneficios de desacoplamiento ofrecidos por el patrón Observador (que reduce la dependencia de las partes de una aplicación entre sí), mientras que otros simplemente pueden ser demasiado pequeños para que el desacoplamiento sea una preocupación.

Dicho esto, una vez que tenemos una sólida comprensión de los patrones de diseño y los problemas específicos a los que se adaptan mejor, se hace mucho más fácil integrarlos en nuestras arquitecturas de aplicaciones.

Los patrones que exploraremos en esta sección son:

- Constructor Pattern
- Module Pattern
- Revealing Module Pattern
- Singleton Pattern
- Observer Pattern
- Mediator Pattern
- Prototype Pattern
- Command Pattern
- Facade Pattern
- Factory Pattern
- Mixin Pattern
- Decorator Pattern
- Flyweight Pattern

Constructor Pattern

En los lenguajes de programación orientados a objetos clásicos, un constructor es un método especial utilizado para inicializar un objeto recién creado una vez que se le ha asignado memoria. En JavaScript, como casi todo es un objeto, a menudo nos interesan los constructores de objetos. Los constructores de objetos se usan para crear tipos específicos de objetos, tanto para preparar el objeto para su uso como para aceptar argumentos que un constructor puede usar para establecer los valores de las propiedades y métodos de los miembros cuando el objeto se crea por primera vez.

Creación de objetos

Las tres formas comunes de crear nuevos objetos en JavaScript son las siguientes:

```
1 // Cada una de las siguientes opciones creará un nuevo objeto vacío:
2
3 var newObject = {};
4
5 // or
6 var newObject = Object.create( Object.prototype );
7
8 // or
9 var newObject = new Object();
```

Cuando el constructor "Object" en el ejemplo final crea un contenedor de objetos para un valor específico, o cuando no se pasa ningún valor, creará un objeto vacío y lo devolverá.

Existen cuatro formas de asignar claves y valores a un objeto:

ej:01

```
// ECMAScript 3 enfoques compatibles

// 1. Sintaxis de punto

// Establecer propiedades
newObject.someKey = "Hello World";

// Obtener propiedades
var value = newObject.someKey;



// 2. Sintaxis de corchetes

// Establecer propiedades
newObject["someKey"] = "Hello World";

// Obtener propiedades
var value = newObject["someKey"];



// ECMAScript 5 solo enfoques compatibles
// Para más información, ver: http://kangax.github.com/es5-compat-table/

// 3. Object.defineProperty
```

```
// Set properties
Object.defineProperty( newObject, "someKey", {
    value: "para un mayor control del comportamiento de la propiedad",
    writable: true,
    enumerable: true,
    configurable: true
});

// Si lo anterior se siente un poco difícil de leer, se puede abreviar de la
// siguiente manera:

var defineProp = function ( obj, key, value ){
  var config = {
    value: value,
    writable: true,
    enumerable: true,
    configurable: true
  };
  Object.defineProperty( obj, key, config );
};

// Para usar, entonces creamos un nuevo objeto "person" vacío
var person = Object.create( Object.prototype );

// Rellenar el objeto con propiedades
defineProp( person, "car", "Delorean" );
defineProp( person, "dateOfBirth", "1981" );
defineProp( person, "hasBeard", false );

console.log(person);
// Outputs: Object {car: "Delorean", dateOfBirth: "1981", hasBeard: false}


// 4. Object.defineProperties

// Set properties
Object.defineProperties( newObject, {

  "someKey": {
    value: "Hello World",
    writable: true
  },

  "anotherKey": {
    value: "Foo bar",
    writable: false
  }

});

// Obtener propiedades para 3. y 4. se puede hacer usando cualquiera de los
// opciones en 1. y 2.
```

Como veremos un poco más adelante en el libro, estos métodos incluso se pueden usar para la herencia, de la siguiente manera:

```
// Uso:

// Crea un piloto de carreras que herede del objeto person
var driver = Object.create( person );
```

```
// Establecer algunas propiedades para el conductor (driver)
defineProp(driver, "topSpeed", "100mph");

// Obtener una propiedad heredada (1981)
console.log( driver.dateOfBirth );

// Obtener la propiedad que establecimos (100mph)
console.log( driver.topSpeed );
```

Constructores Básicos

Como vimos anteriormente, JavaScript no admite el concepto de clases, pero sí admite funciones especiales de construcción que funcionan con objetos. Simplemente anteponiendo una llamada a una función de constructor con la palabra clave "new", podemos decirle a JavaScript que nos gustaría que la función se comporte como un constructor e instanciar un nuevo objeto con los miembros definidos por esa función.

Dentro de un constructor, la palabra clave *this* hace referencia al nuevo objeto que se está creando. Revisando la creación de objetos, un constructor básico puede verse de la siguiente manera:

ej:002

```
function Car( model, year, miles ) {

  this.model = model;
  this.year = year;
  this.miles = miles;

  this.toString = function () {
    return this.model + " has done " + this.miles + " miles";
  };
}

// Uso:

// Podemos crear nuevas instancias del auto (car)
var civic = new Car( "Honda Civic", 2009, 20000 );
var mondeo = new Car( "Ford Mondeo", 2010, 5000 );

// y luego abra nuestra consola del navegador para ver la
// salida del método toString() que se llama en
// estos objetos
console.log( civic.toString() );
console.log( mondeo.toString() );
```

Lo anterior es una versión simple del patrón de constructor pero sufre algunos problemas. Una es que dificulta la herencia y la otra es que funciones como `toString()` se redefinen para cada uno de los nuevos objetos creados con el constructor Car. Esto no es muy óptimo ya que la función idealmente debería compartirse entre todas las instancias del tipo Car.

Afortunadamente, como hay una serie de alternativas compatibles con ES3 y ES5 para construir objetos, es trivial evitar esta limitación.

Constructores con Prototipos

Las funciones, como casi todos los objetos en JavaScript, contienen un objeto "prototype". Cuando llamamos a un constructor de JavaScript para crear un objeto, todas las propiedades del prototipo del constructor se ponen a disposición del nuevo objeto. De esta manera, se pueden crear múltiples objetos Car que acceden al mismo prototipo. Así podemos extender el ejemplo original de la siguiente manera:

ej:003

```
function Car( model, year, miles ) {

  this.model = model;
  this.year = year;
  this.miles = miles;

}

// Tenga en cuenta que aquí estamos usando Object.prototype.newMethod en
// lugar de Object.prototype para evitar redefinir el objeto prototipo

Car.prototype.toString = function () {
  return this.model + " has done " + this.miles + " miles";
};

// Usage:

var civic = new Car( "Honda Civic", 2009, 20000 );
var mondeo = new Car( "Ford Mondeo", 2010, 5000 );

console.log( civic.toString() );
console.log( mondeo.toString() );
```

Arriba, ahora se compartirá una sola instancia de `toString()` entre todos los objetos Car.

El Patrón de Módulo
Módulo

Los módulos son una parte integral de la arquitectura de cualquier aplicación robusta y, por lo general, ayudan a mantener las unidades de código para un proyecto separadas y organizadas de manera limpia.

En JavaScript, hay varias opciones para implementar módulos. Éstos incluyen:

- El patrón Módulo
- Notación literal de objeto
- Módulos AMD
- Módulos CommonJS
- Módulos Armonía ECMAScript

Exploraremos las últimas tres de estas opciones más adelante en el libro en la sección *Patrones de Diseño Modulares Modernos de JavaScript*.

El patrón del Módulo se basa en parte en literales de objetos, por lo que tiene sentido actualizar primero nuestro conocimiento de ellos.

Literales de Objetos

En notación literal de objeto, un objeto se describe como un conjunto de pares de nombre/valor separados por comas encerrados entre llaves (`{}`). Los nombres dentro del objeto pueden ser cadenas o identificadores seguidos de dos puntos. No debe usarse una coma después del par de nombre/valor final en el objeto, ya que esto puede provocar errores.

```
var myObjectLiteral = {

    variableKey: variableValue,

    functionKey: function () {
      // ...
    }
};
```

Los literales de objeto no requieren instanciación usando el operador `new`, pero no deben usarse al comienzo de una declaración ya que la apertura `{` puede interpretarse como el comienzo de un bloque. Fuera de un objeto, se pueden agregar nuevos miembros mediante la asignación de la siguiente manera `myModule.property = "someValue";`

A continuación podemos ver un ejemplo más completo de un módulo definido usando notación literal de objeto:
ej:004

```
var myModule = {

  myProperty: "someValue",

 // Los literales de objeto pueden contener propiedades y métodos.
 // ej. podemos definir otro objeto para la configuración del módulo:
  myConfig: {
    useCaching: true,
    language: "en"
  },

  // un método muy básico
  saySomething: function () {
    console.log( "Where in the world is Paul Irish today?" );
  },

  // generar un valor basado en la configuración actual
  reportMyConfig: function () {
    console.log( "Caching is: " + ( this.myConfig.useCaching ? "enabled" :
"disabled") );
  },

  // anular la configuración actual
  updateMyConfig: function( newConfig ) {

    if ( typeof newConfig === "object" ) {
      this.myConfig = newConfig;
      console.log( this.myConfig.language );
    }
  }
};

// Salidas: someValue
console.log(myModule.myProperty);

// Salidas: Where in the world is Paul Irish today?
myModule.saySomething();

// Salidas: Caching is: enabled
myModule.reportMyConfig();

// Salidas: fr
myModule.updateMyConfig({
  language: "fr",
  useCaching: false
});

// Salidas: Caching is: disabled
myModule.reportMyConfig();
```

El uso de literales de objetos puede ayudar a encapsular y organizar su código, y Rebecca Murphey ha escrito anteriormente sobre este tema en profundidad si desea leer más en literales de objetos.

Dicho esto, si estamos optando por esta técnica, podemos estar igualmente interesados en el patrón del Módulo. Todavía usa literales de objeto, pero solo como el valor de retorno de una función de alcance.

El Patrón Módulo

El patrón Módulo se definió originalmente como una forma de proporcionar encapsulación pública y privada para las clases de ingeniería de software convencional.

En JavaScript, el patrón Módulo se usa para emular aún más el concepto de clases de tal manera que podamos incluir tanto métodos públicos/privados como variables dentro de un solo objeto, protegiendo así partes particulares del alcance global. Lo que esto resulta es una reducción en la probabilidad de que nuestros nombres de funciones entren en conflicto con otras funciones definidas en scripts adicionales en la página.

Privacidad
El patrón Módulo encapsula "privacidad", estado y organización mediante cierres. Proporciona una forma de envolver una mezcla de métodos y variables públicas y privadas, evitando que las piezas se filtren en el ámbito global y colisionen accidentalmente con la interfaz de otro desarrollador. Con este patrón, solo se devuelve una API pública, manteniendo todo lo demás dentro del cierre privado.

Esto nos da una solución limpia para la lógica de blindaje que hace el trabajo pesado mientras expone la interfaz que deseamos que utilicen otras partes de nuestra aplicación. El patrón utiliza una expresión de función invocada inmediatamente (IIFE - consulte la sección sobre patrones de espacios de nombres para obtener más información sobre esto) donde se devuelve un objeto.

Cabe señalar que no existe realmente un sentido explícitamente verdadero de "privacidad" dentro de JavaScript porque, a diferencia de algunos lenguajes tradicionales, no tiene modificadores de acceso. Técnicamente, las variables no pueden declararse como públicas ni privadas, por lo que utilizamos el alcance de la función para simular este concepto. Dentro del patrón Módulo, las variables o métodos declarados solo están disponibles dentro del módulo gracias al cierre. Sin embargo, las variables o métodos definidos dentro del objeto de retorno están disponibles para todos.

Historia
Desde una perspectiva histórica, el patrón del Módulo fue desarrollado originalmente por varias personas, incluido Richard Cornford en 2003. Más tarde, Douglas Crockford lo popularizó en sus conferencias. Otra curiosidad es que si alguna vez has jugado con la biblioteca YUI de Yahoo, algunas de sus características pueden parecer bastante familiares y la razón de esto es que el patrón del Módulo fue una gran influencia para YUI al crear sus componentes.

Ejemplos
Comencemos a ver una implementación del patrón Módulo creando un módulo que sea autónomo.
Ej:005

```
var testModule = (function () {

  var counter = 0;
```

```
  return {

    incrementCounter: function () {
      console.log( "counter value in this moment: " + counter );
      return counter++;
    },

    resetCounter: function () {
      console.log( "counter value prior to reset: " + counter );
      counter = 0;
    }
  };

})();

// Uso:

// Incrementa nuestro contardor
testModule.incrementCounter(); //counter value in this moment: 0
testModule.incrementCounter(); //counter value in this moment: 1
testModule.incrementCounter(); //counter value in this moment: 2

// Verifique el valor del contador y reinicie
// Salidas: valor del contador antes de reiniciar: 1
testModule.resetCounter(); //counter value in this moment: 3
```

Aquí, otras partes del código no pueden leer directamente el valor de nuestro `incrementCounter()` o `resetCounter()`. La variable de contador está completamente protegida de nuestro alcance global, por lo que actúa como lo haría una variable privada: su existencia está limitada dentro del cierre del módulo, de modo que el único código capaz de acceder a su alcance son nuestras dos funciones. Nuestros métodos tienen un espacio de nombres efectivo, por lo que en la sección de prueba de nuestro código, necesitamos anteponer cualquier llamada con el nombre del módulo (ej., "testModule").

Cuando trabaje con el patrón Módulo, podemos encontrarnos útil para definir una plantilla simple que usamos para comenzar con él. Aquí hay uno que cubre el espacio de nombres, las variables públicas y privadas:
ej:006

```
var myNamespace = (function () {

  var myPrivateVar, myPrivateMethod;

  // Una variable de contador privado
  myPrivateVar = 0;

  // Una función privada que registra cualquier argumento
  myPrivateMethod = function( foo ) {
      console.log( foo, myPrivateVar );
  };

  return {

    // Una variable pública
```

```
    myPublicVar: "foo",

    // Una función pública que utiliza privados
    myPublicFunction: function( bar ) {

      // Incrementar nuestro contador privado
      myPrivateVar++;

      // Llama a nuestro método privado usando bar
      myPrivateMethod( bar );

    }
  };

})();

// Uso
myNamespace.myPublicFunction('myCount goes by:'); //myCount goes by: 1
myNamespace.myPublicFunction('myCount goes by:'); //myCount goes by: 2
myNamespace.myPublicFunction('myCount goes by:'); //myCount goes by: 3
```

Mirando otro ejemplo, a continuación podemos ver una cesta de la compra implementada usando este patrón. El módulo en sí está completamente autocontenido en una variable global llamada basketModule. El conjunto de la cesta en el módulo se mantiene privado y, por lo tanto, otras partes de nuestra aplicación no pueden leerlo directamente. Solo existe con el cierre del módulo y, por lo tanto, los únicos métodos que pueden acceder a él son aquellos con acceso a su alcance (es decir, `addItem()`, `getItemCount()`, etc.).

```
var basketModule = (function () {

    // privadas

    var basket = [];

    function doSomethingPrivate(obj) {
        return ( typeof obj == 'object'
            && obj.hasOwnProperty('article')
            && obj.hasOwnProperty('price') )
                ? true : false;
    }

    function doSomethingElsePrivate() {
        return 'It is not a valid article';
    }

    function getPreItemCountPrivate() {
        return basket.length + 1;
    }

    // Devolver un objeto expuesto al público.
    return {

        // Agregar artículos a nuestra cesta
        addItem: function( obj ) {
            if (doSomethingPrivate(obj)) {
```

```
                basket.push(obj);
                return obj.article + ', price: ' + obj.price;
            } else
                return doSomethingElsePrivate();
        },

        // Obtener el contador de artículos en la cesta
        getItemCount: function () {
            return basket.length;
        },

        // Alias público para una función privada
        getPreItemCount: getPreItemCountPrivate,

        // Obtener el valor total de los artículos en la cesta
        getTotal: function () {

            var q = this.getItemCount(),
            p = 0;

            while (q--) {
                p += basket[q].price;
            }

            return p;
        }
    };
})();
```

Dentro del módulo, puede haber notado que devolvemos un objeto. Esto se asigna automáticamente a `basketModule` para que podamos interactuar con él de la siguiente manera:

```
// basketModule devuelve un objeto con una API pública que podemos usar

console.log(basketModule.getPreItemCount(), basketModule.addItem(''));
console.log(basketModule.getPreItemCount(), basketModule.addItem({}));
console.log(basketModule.getPreItemCount(), basketModule.addItem({article:'article1'}));
console.log(basketModule.getPreItemCount(), basketModule.addItem({price:100}));
console.log(basketModule.getPreItemCount(), basketModule.addItem({article:'article1', price:100}));
console.log(basketModule.getPreItemCount(), basketModule.addItem({article:'article2', price:200}));
console.log(basketModule.getPreItemCount(), basketModule.addItem({article:'article3', price:300}));
console.log('Total article:',basketModule.getItemCount(), 'Total amount:', basketModule.getTotal());

// Salidas:

// 1 It is not a valid article
// 1 It is not a valid article
// 1 It is not a valid article
// 1 It is not a valid article
// 1 article1, price: 100
// 2 article2, price: 200
// 3 article3, price: 300
// Total article: 3 Total amount: 600

// Sin embargo, lo siguiente no funcionará:

// Salidas: undefined
// Esto se debe a que basket en sí no está expuesta como parte de nuestro
// API pública
console.log( basketModule.basket );

// Esto tampoco funcionará, ya que solo existe dentro del alcance de nuestro
```

```
// cierre basketModule, pero no en el objeto público devuelto
console.log( basket );
```

Los métodos anteriores son efectivamente espacios de nombres dentro de `basketModule`.

Observe cómo la función de alcance en el módulo de canasta anterior se envuelve alrededor de todas nuestras funciones, que luego llamamos e inmediatamente almacenamos el valor de retorno de. Esto tiene una serie de ventajas que incluyen:

- La libertad de tener funciones privadas y miembros privados que solo pueden ser consumidos por nuestro módulo. Como no están expuestos al resto de la página (solo lo está nuestra API exportada), se consideran verdaderamente privados.
- Dado que las funciones se declaran normalmente y se nombran, puede ser más fácil mostrar las pilas de llamadas en un depurador cuando intentamos descubrir qué funciones arrojaron una excepción.
- Como T.J Crowder ha señalado en el pasado, también nos permite devolver diferentes funciones dependiendo del entorno. En el pasado, he visto a los desarrolladores usar esto para realizar pruebas de UA con el fin de proporcionar una ruta de código en su módulo específico para IE, pero actualmente podemos optar fácilmente por la detección de características para lograr un objetivo similar.

Variaciones del Patrón del Módulo

Importar mezcla (mixins)

Esta variación del patrón demuestra cómo los globales (ej., myObjA, myObjB) se pueden pasar como argumentos a la función anónima de nuestro módulo. Esto efectivamente nos permite importarlos y aplicarles alias localmente como lo deseamos.

Ej:008

```
var myObjA = { id:'id_a' },
    myObjB = {
        id : 'id_b',
        func: function () {
            console.log( 'Id de B:', this.id )
        }
    };

// Módulo global
var myModule = (function ( obj1, obj2 ) {

    function privateMethod1(){
      console.log( 'Id de A:', obj1.id );
    }

    function privateMethod2(){
      obj2.func();
    }

    return{
        publicMethod1: function(){
```

```
            privateMethod1();
        },
        publicMethod2: function(){
            privateMethod2();
        }
    };

})( myObjA, myObjB ); // Importar myObjA y myObjB

// Uso:

myModule.publicMethod1(); // Salida: Id de A: id_a
myModule.publicMethod2(); // Salida: Id de B: id_b
```

Exportaciones

Esta próxima variación nos permite declarar globales sin consumirlos y podría apoyar de manera similar el concepto de importaciones globales visto en el último ejemplo.

Ej:009

```
// Módulo global
var myModule = (function () {

    var privateVariable = "Hello World",
    module = {
        publicProperty : "Foobar",
        publicMethod : function () {
            privateMethod();
        }
    };

    function privateMethod() {
        console.log( privateVariable );
    }

    return module;

})();

// Uso:

console.log(myModule.publicProperty);
myModule.publicMethod();

// Salidas:

// Foobar
// Hello World
```

Kit de Herramientas e Implementaciones de Patrones de Módulos específicos del framework Dojo

Dojo proporciona un método conveniente para trabajar con objetos llamados dojo.setObject (). Esto toma como primer argumento una cadena separada por puntos como myObj.parent.child que se refiere a una propiedad llamada "child" dentro de un objeto "parent" definido dentro de "myObj". El

uso de setObject() nos permite establecer el valor de los hijos, creando cualquiera de los objetos intermedios en el resto de la ruta pasada si aún no existen.

Por ejemplo, si quisiéramos declarar basket.core como un objeto del espacio de nombres de la tienda, esto podría lograrse de la siguiente manera utilizando la forma tradicional:

```
var store = window.store || {};

if ( !store["basket"] ) {
  store.basket = {};
}

if ( !store.basket["core"] ) {
  store.basket.core = {};
}

store.basket.core = {
  // ...rest of our logic
};
```

O de la siguiente manera usando Dojo 1.7 (versión compatible con AMD) y superior:

```
require(["dojo/_base/customStore"], function( store ){

    // using dojo.setObject()

    store.setObject( "basket.core", (function() {

        var basket = [];

        function privateMethod() {
            console.log(basket);
        }

        return {
            publicMethod: function(){
                privateMethod();
            }
        };

    })());

});
```

Para obtener más información sobre `dojo.setObject()`, consulte la documentación oficial.

ExtJS
Para aquellos que usan ExtJS de Sencha, a continuación se puede encontrar un ejemplo que demuestra cómo usar correctamente el patrón Módulo con el framework.

Aquí, vemos un ejemplo de cómo definir un espacio de nombres que luego se puede completar con un módulo que contiene una API pública y privada. Con la excepción de algunas diferencias semánticas, está bastante cerca de cómo se implementa el patrón Módulo en JavaScript vainilla:

```
// create namespace
Ext.namespace("myNameSpace");

// create application
myNameSpace.app = function () {

  // NO acceda a DOM desde aquí; los elementos aún no existen

  // private variables
  var btn1,
      privVar1 = 11;

  // funciones privadas
  var btn1Handler = function ( button, event ) {
      console.log( "privVar1=" + privVar1 );
      console.log( "this.btn1Text=" + this.btn1Text );
    };

  // espacio público
  return {
    // propiedades públicas, ej. cadenas para traducir
    btn1Text: "Button 1",

    // méthodos públicos
    init: function () {

      if ( Ext.Ext2 ) {

        btn1 = new Ext.Button({
          renderTo: "btn1-ct",
          text: this.btn1Text,
          handler: btn1Handler
        });

      } else {

        btn1 = new Ext.Button( "btn1-ct", {
          text: this.btn1Text,
          handler: btn1Handler
        });

      }
    }
  };
}();
```

YUI

Del mismo modo, también podemos implementar el patrón Módulo al crear aplicaciones utilizando YUI3. El siguiente ejemplo se basa en gran medida en la implementación del patrón del módulo YUI original por Eric Miraglia, pero, de nuevo, no es muy diferente de la versión JavaScript de vainilla:

```
Y.namespace( "store.basket" ) ;
Y.store.basket = (function () {

    var myPrivateVar, myPrivateMethod;

    // private variables:
    myPrivateVar = "I can be accessed only within Y.store.basket.";

    // private method:
    myPrivateMethod = function () {
        Y.log( "I can be accessed only from within YAHOO.store.basket" );
    }

    return {
        myPublicProperty: "I'm a public property.",

        myPublicMethod: function () {
            Y.log( "I'm a public method." );

            // Within basket, I can access "private" vars and methods:
            Y.log( myPrivateVar );
            Y.log( myPrivateMethod() );

            // The native scope of myPublicMethod is store so we can
            // access public members using "this":
            Y.log( this.myPublicProperty );
        }
    };

})();
```

jQuery

Hay varias formas en que el código jQuery inespecífico para los complementos se puede ajustar dentro del patrón del Módulo. Ben Cherry sugirió previamente una implementación en la que se usa un contenedor de funciones alrededor de las definiciones de módulos en caso de que haya una serie de puntos en común entre los módulos.

En el siguiente ejemplo, se define una función de `library` que declara una nueva librería y vincula automáticamente la función `init` para `document.ready` cuando se crean nuevas librerías (es decir, módulos).

```
function library( module ) {

    $( function() {
        if ( module.init ) {
            module.init();
        }
    });

    return module;
}

var myLibrary = library(function () {
```

```
    return {
        init: function () {
            // module implementation
        }
   };

}());
```

Ventajas

Hemos visto por qué el patrón Constructor puede ser útil, pero ¿por qué el patrón Módulo es una buena opción? Para empezar, es mucho más limpio para los desarrolladores que provienen de un entorno orientado a objetos que la idea de una verdadera encapsulación, al menos desde una perspectiva de JavaScript.

En segundo lugar, admite datos privados, por lo que, en el patrón Módulo, las partes públicas de nuestro código pueden tocar las partes privadas, sin embargo, el mundo exterior no puede tocar las partes privadas de la clase (¡no se ría! Ah, y gracias a David Engfer por el chiste).

Desventajas

Las desventajas del patrón Módulo son que a medida que accedemos a los miembros públicos y privados de manera diferente, cuando deseamos cambiar la visibilidad, en realidad tenemos que hacer cambios en cada lugar donde se usó el miembro.

Tampoco podemos acceder a miembros privados en métodos que se agregan al objeto en un momento posterior. Dicho esto, en muchos casos el patrón del Módulo sigue siendo bastante útil y, cuando se usa correctamente, ciertamente tiene el potencial de mejorar la estructura de nuestra aplicación.
Otras desventajas incluyen la incapacidad de crear pruebas unitarias automatizadas para miembros privados y una complejidad adicional cuando los errores requieren correcciones urgentes. Simplemente no es posible parchear privados. En cambio, uno debe anular todos los métodos públicos que interactúan con los privados con errores. Los desarrolladores tampoco pueden extender fácilmente los privados, por lo que vale la pena recordar que los privados no son tan flexibles como pueden parecer inicialmente.

Para leer más sobre el patrón Módulo, vea el excelente artículo en profundidad de Ben Cherry sobre él.

El Patrón Módulo Revelador

Ahora que estamos un poco más familiarizados con el patrón del módulo, echemos un vistazo a una versión ligeramente mejorada: el patrón del módulo revelador de Christian Heilmann. El patrón del Módulo Revelador surgió cuando Heilmann estaba frustrado con el hecho de que tenía que repetir el nombre del objeto principal cuando queríamos llamar a un método público desde otro o acceder a variables públicas. Tampoco le gustó el requisito del patrón del Módulo para tener que cambiar a la notación literal de objetos para las cosas que deseaba hacer públicas. El resultado de sus esfuerzos fue un patrón actualizado en el que simplemente definiríamos todas nuestras funciones y variables en el ámbito privado y devolveríamos un objeto anónimo con punteros a la funcionalidad privada que deseábamos revelar como pública.

Un ejemplo de cómo usar el patrón *Revealing Module* se puede encontrar a continuación:

ej:010

```
var myRevealingModule = (function () {

    var privateVar = "Ben Cherry",
        publicVar = "Hey there!";

    function privateFunction() {
        console.log( "Name:" + privateVar );
    }

    function publicSetName( strName ) {
        privateVar = strName;
    }

    function publicGetName() {
        privateFunction();
    }

    // Revelar punteros públicos para
    // funciones privadas y propiedad

    return {
        setName: publicSetName,
        greeting: publicVar,
        getName: publicGetName
    };

})();

// Uso:

console.log(myRevealingModule.greeting); // Salida:
myRevealingModule.getName(); // Salida: Ben Cherry
```

```
myRevealingModule.setName( "Paul Kinlan" );
myRevealingModule.getName(); // Salida: Paul Kinlan
```

El patrón también se puede usar para revelar funciones y propiedades privadas con un esquema de nombres más específico si preferimos:
ej:011

```
var myRevealingModule = (function () {

    var privateCounter = 0;

    function privateFunction() {
        privateCounter++;
    }

    function publicFunction() {
        publicIncrement();
    }

    function publicIncrement() {
        privateFunction();
    }

    function publicGetCount(){
        return privateCounter;
    }

    // Revelar punteros públicos para
    // funciones y propiedades privadas

    return {
        start: publicFunction,
        increment: publicIncrement,
        count: publicGetCount
    };

})();

// Uso:

console.log(myRevealingModule.count()); // 0
myRevealingModule.start();
console.log(myRevealingModule.count()); // 1
myRevealingModule.increment();
console.log(myRevealingModule.count()); // 2
```

Ventajas

Este patrón permite que la sintaxis de nuestros scripts sea más consistente. También deja más claro al final del módulo cuáles de nuestras funciones y variables pueden accederse públicamente, lo que facilita la legibilidad.

Desventajas

Una desventaja de este patrón es que si una función privada se refiere a una función pública, esa función pública no puede anularse si es necesario un parche. Esto se debe a que la función privada seguirá haciendo referencia a la implementación privada y el patrón no se aplica a los miembros públicos, solo a las funciones.

Los miembros de objetos públicos que se refieren a variables privadas también están sujetos a las notas de la regla sin parche anteriores.

Como resultado de esto, los módulos creados con el patrón Revealing Module pueden ser más frágiles que los creados con el patrón Module original, por lo que se debe tener cuidado durante el uso.

El Patrón Singleton (Únic@)

El patrón Singleton se conoce así porque restringe la creación de instancias de una clase a un solo objeto. Clásicamente, el patrón Singleton se puede implementar creando una clase con un método que cree una nueva instancia de la clase si no existe. En el caso de una instancia ya existente, simplemente devuelve una referencia a ese objeto.

Los singletons difieren de las clases estáticas (u objetos) ya que podemos retrasar su inicialización, generalmente porque requieren cierta información que puede no estar disponible durante el tiempo de inicialización. No proporcionan una forma de código que desconoce una referencia previa a ellos para recuperarlos fácilmente. Esto se debe a que no es el objeto o la "clase" lo que devuelve un Singleton, es una estructura. Piense en cómo las variables cerradas no son realmente cierres: el alcance de la función que proporciona el cierre es el cierre.

En JavaScript, Singletons sirve como un espacio de nombres de recursos compartidos que aísla el código de implementación del espacio de nombres global para proporcionar un único punto de acceso para las funciones.

Podemos implementar un Singleton de la siguiente manera:

ej:12

```
var MySingleton = (function () {

    // La instancia almacena una referencia al Singleton
    var instance;

    function init() {

        // Singleton

        // Métodos y variables privad@s
        function privateMethod(){
            console.log( "I am private" );
        }

        var privateVariable = "Im also private";

        var privateRandomNumber = Math.random();

        return {

            // Métodos y variables públic@s
            publicMethod: function () {
                console.log( "The public can see me!" );
            },

            publicProperty: "I am also public",

            getRandomNumber: function() {
                return privateRandomNumber;
            }

        };

    };
```

```
    return {

        // Obtener la instancia única (Singleton) si existe
        // o crea una si no es así
        getInstance: function () {

            if ( !instance ) {
                instance = init();
                console.log(instance); // Comprobar que se crea la instancia sólo una vez
            }

            return instance;
        }

    };

})();

// Uso:

var obj1 = MySingleton.getInstance();
var obj2 = MySingleton.getInstance();
obj1.publicMethod();
console.log(obj2.publicProperty);
console.log(obj1.getRandomNumber());
console.log(obj2.privateVariable);
console.log(obj1.privateRandomNumber());

// Salidas:

// Object { publicMethod: init/<.publicMethod(), publicProperty: "I am also public",...
// ...getRandomNumber: init/<.getRandomNumber() }
// The public can see me!
// I am also public
// 0.729069521305126 (ej: de valor aliatorio)
// undefined
// TypeError: obj1.privateRandomNumber is not a function[Conocer más]
```

Lo que hace que Singleton sea el acceso global a la instancia (generalmente a través de `MySingleton.getInstance()`) ya que no (al menos en lenguajes estáticos) llamamos a `new MySingleton()` directamente. Sin embargo, esto es posible en JavaScript.

En el libro GoF, la *aplicabilidad* del patrón Singleton se describe de la siguiente manera:

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

El segundo de estos puntos se refiere a un caso en el que podríamos necesitar código como:
ej:013

```
var MySingleton = (function () {

    this._instance;
    let _isFoo = false;

    let basicSingleton = function ( model, year, miles ) {

        this.model = model;
        this.year = year;
```

```
        this.miles = miles;

    }

    let fooSingleton = function ( model, year, miles ) {

        basicSingleton.prototype.toString = function () {
            return this.model + " has done " + this.miles + " miles";
        };

        basicSingleton.prototype.setNew = function (model, year, miles) {

            this.model = model;
            this.year = year;
            this.miles = miles;

        };

        _isFoo = true;

        return new basicSingleton( model, year, miles );

    }

    return {

        getInstance: function(model, year, miles) {
            if ( this._instance == undefined )
                this._instance = new basicSingleton( model, year, miles );
            else if ( !_isFoo )
                this._instance = fooSingleton( model, year, miles );
            return this._instance;
        },

    };

})();

// Uso y salida:

var obj1 = MySingleton.getInstance("Honda Civic", 2009, 20000);
console.log(obj1);            // Object { model: "Honda Civic", year: 2009, miles: 20000 }
console.log(obj1.toString()); // [object Object]  index.js:48:1

var obj2 = MySingleton.getInstance("Ford Mondeo", 2010, 5000);
console.log(obj2);            // Object { model: "Ford Mondeo", year: 2010, miles: 5000 }
console.log(obj2.toString()); // Ford Mondeo has done 5000 miles  index.js:52:1

var obj3 = MySingleton.getInstance("Jeep CJ5", 1975, 15000);
console.log(obj3);            // Object { model: "Ford Mondeo", year: 2010, miles: 5000 }
console.log(obj3.toString()); // Ford Mondeo has done 5000 miles  index.js:56:1

obj3.setNew("Jeep CJ5", 1975, 15000);
console.log(obj3);            // Object { model: "Jeep CJ5", year: 1975, miles: 15000 }
console.log(obj3.toString()); // Jeep CJ5 has done 15000 miles  index.js:60:1

var obj4 = MySingleton.getInstance("Toyota Celica", 2019, 19000);

console.log(obj4);            // Object { model: "Jeep CJ5", year: 1975, miles: 15000 }
console.log(obj4.toString()); // Jeep CJ5 has done 15000 miles
```

Aquí, `getInstance` se parece un poco al método Factory y no necesitamos actualizar cada punto de nuestro código para acceder a él. El `fooSingleton` anterior sería una subclase de `basicSingleton` e implementaría la misma interfaz.

¿Por qué la ejecución diferida se considera importante para un Singleton?

> En C ++ sirve para aislar de la imprevisibilidad del orden de inicialización dinámica, devolviendo el control al programador.

Es importante notar la diferencia entre una instancia estática de una clase (objeto) y un Singleton: mientras que un Singleton puede implementarse como una instancia estática, también puede construirse perezosamente, sin la necesidad de recursos ni memoria hasta que esto sea realmente necesario.

Si tenemos un objeto estático que se puede inicializar directamente, debemos asegurarnos de que el código siempre se ejecute en el mismo orden (por ejemplo, en caso de que `objCar` necesite `objWheel` durante su inicialización) y esto no se escala cuando tiene un gran número de archivos fuentes.
Tanto Singletons como los objetos estáticos son útiles, pero no deberían usarse en exceso, de la misma manera en que no deberíamos usar en exceso otros patrones.
En la práctica, el patrón Singleton es útil cuando se necesita exactamente un objeto para coordinar otros a través de un sistema. Aquí hay un ejemplo con el patrón que se utiliza en este contexto:
ej:014

```
var SingletonTester = (function () {

    // options: un objeto que contiene opciones de configuración para el singleton
    // ej. var options = { name: "test", pointX: 5};
    function Singleton( options ) {

        // establecer opciones a las opciones suministradas
        // o un objeto vacío si no se proporciona ninguno
        options = options || {};

        // establecer algunas propiedades para nuestro singleton
        this.name = "SingletonTester";

        this.pointX = options.pointX || 6;

        this.pointY = options.pointY || 10;

    }

    // nuestro asimiento de instancia
    let instance;

    // una emulación de variables estáticas y métodos
    let _static = {

        name: "SingletonTester",

        // Método para obtener una instancia. Devuelve
        // una instancia singleton de un objeto singleton
        getInstance: function( options ) {

            if( instance === undefined ) {
```

```
                instance = new Singleton( options );
            }

            return instance;

        }
    };

    return _static;

})();

var singletonTest = SingletonTester.getInstance({ pointX: 5, pointY: 9 });

// Registre la salida de pointX y pointY solo para verificar que sea correcta

console.log( singletonTest.pointX, ',', singletonTest.pointY);

// Salidas: 5 , 9
```

Si bien el Singleton tiene usos válidos, a menudo cuando nos encontramos necesitándolo en JavaScript, es una señal de que podemos necesitar reevaluar nuestro diseño.
A menudo son una indicación de que los módulos en un sistema están estrechamente acoplados o que la lógica está demasiado extendida en varias partes de una base de código. Los Singletons pueden ser más difíciles de probar debido a problemas que van desde dependencias ocultas, la dificultad para crear múltiples instancias, dificultades para tropezar dependencias, etc.
Miller Medeiros ha recomendado previamente este excelente artículo sobre Singleton y sus diversos temas para leer más, así como los comentarios a este artículo, discutiendo cómo Singletons puede aumentar el acoplamiento estrecho. Me complace respaldar estas recomendaciones, ya que ambas piezas plantean muchos puntos importantes sobre este patrón que también vale la pena señalar.

El Patrón Observador

El observador es un patrón de diseño en el que un objeto (conocido como Sujeto <<Subject>>) mantiene una Lista de Objetos dependiendo de él (Observadores), notificándoles automáticamente cualquier cambio de estado.
Cuando un Sujeto necesita notificar a los Observadores sobre algo interesante que sucede, transmite una notificación a los Observadores (que puede incluir datos específicos relacionados con el tema de la notificación).
Cuando ya no deseamos que un Observador en particular sea notificado de los cambios por el Sujeto con el que está registrado, el Sujeto puede eliminarlo de la Lista de Observadores.
A menudo es útil volver a consultar las definiciones publicadas de patrones de diseño que son independientes del lenguaje para tener una idea más amplia de su uso y ventajas a lo largo del tiempo. La definición del patrón Observador que se proporciona en el libro GoF, *Patrones de Diseño: Elementos de Software Orientado a Objetos Reutilizables*, es:

"Uno o más observadores están interesados en el estado de un sujeto y registran su interés con el sujeto al adjuntarse. Cuando algo cambia en nuestro tema que el observador puede estar interesado, se envía un mensaje de notificación que llama al método de actualización en cada

observador. Cuando el observador ya no está interesado en el estado del sujeto, simplemente puede desprenderse ".

Ahora podemos ampliar lo que hemos aprendido para implementar el patrón Observador con los siguientes componentes:

- **Subject**: mantiene una lista de Observadores, facilita la adición o eliminación de Observadores
  ObserverList: proporciona una interfaz de actualización para los Objetos que necesitan ser notificados de los cambios de estado de un Sujeto
- **ConcreteSubject**: transmite notificaciones a los Observadores sobre cambios de estado, almacena el estado de ConcreteObservers
- **ConcreteObserver**: almacena una referencia al ConcreteSubject, implementa una interfaz de actualización para el observador para garantizar que el estado sea coherente con el del sujeto

Primero, modelemos la Lista De Observadores dependientes que un Sujeto puede tener:
ej:015:

```
function ObserverList() {
    this.observerList = [];
}

ObserverList.prototype.add = function( obj ) {
    return this.observerList.push( obj );
};

ObserverList.prototype.count = function() {
    return this.observerList.length;
};

ObserverList.prototype.get = function( index ) {
    if( index > -1 && index < this.observerList.length ){
        return this.observerList[ index ];
    }
};


ObserverList.prototype.indexOf = function( obj, startIndex ) {
    var i = startIndex;

    while( i < this.observerList.length ) {
        if( this.observerList[i] === obj ) {
          return i;
        }
        i++;
    }

    return -1;
```

```
};

ObserverList.prototype.remove = function(position) {
    return this.observerList.splice( position, 1  );
};
```

A continuación, modelemos el Sujeto y la capacidad de agregar, eliminar o notificar a los observadores en la lista de observadores.

```
function Subject() {
    this.observers = new ObserverList();
}

function Subject() {
    this.observers = new ObserverList();
}

Subject.prototype.addObserver = function( observer ) {
    this.observers.add( observer );
};

Subject.prototype.notify = function( context ) {
    var observerCount = this.observers.count();
    for(var i=0; i < observerCount; i++){
        this.observers.get(i).update( context );
    }
};

Subject.prototype.removeObserver = function( observer ) {
    this.observers.remove( this.observers.indexOf( observer, 0 ) );
};
```

En nuestra aplicación de muestra que utiliza los componentes Observador anteriores, ahora definimos:

- Un botón para agregar nuevas casillas de verificación observables a la página
- Una casilla de verificación de control que actuará como un sujeto, notificando a otras casillas de verificación que deben marcarse
- Un contenedor para las nuevas casillas de verificación que se agregan

html

```
  <script src="ObserverList.js"></script>
  <script src="Subject.js"></script>

  <button id="addNewObserver">Add New Observer checkbox</button>
```

```
  <input id="mainCheckbox" type="checkbox"/>
  <div id="observersContainer"></div>

  <script src="index.js"></script>
```

Luego definimos los controladores ConcreteSubject y ConcreteObserver para agregar nuevos observadores a la página e implementar la interfaz de actualización. Consulte a continuación los comentarios en línea sobre lo que hacen estos componentes en el contexto de nuestro ejemplo.

```
// Referencias a nuestros elementos DOM
var controlCheckbox = document.getElementById( "mainCheckbox" ),
    container = document.getElementById( "observersContainer" ),
    addBtn = document.getElementById( "addNewObserver" );

// Extiende un objeto con una extensión
var extend = function ( obj, extension ){
    for ( let key in extension ){
        obj[key] = extension[key];
        console.log(obj, obj[key]);
    }
}

// El Observador
var Observer = function (){
    this.update = function(){
        // ...
    };
}

// Observador Concreto
var addNewObserver = function (){

    // Crea un nuevo checkbox para ser agregado
    let check = document.createElement( "input" );
    check.type = "checkbox";

    // Extiende el checkbox con la clase Observador
    extend( check, new Observer() );

    // Anular con comportamiento de actualización personalizado
    check.update = function( value ){
        this.checked = value;
    };

    // Agrega el nuevo observador a nuestra lista de observadores
    // para nuestro sujeto principal
    controlCheckbox.addObserver( check );

    // Append the item to the container
    container.appendChild( check );
}

addBtn.onclick = addNewObserver;

// Sujeto Concreto
// Extienda el checkbox de control con la clase Subject
extend( controlCheckbox, new Subject() );

// Al hacer clic en la casilla de verificación, se activarán notificaciones para sus observadores
controlCheckbox.onclick = function(){
    controlCheckbox.notify( controlCheckbox.checked );
```

```
};
```

Note que definimos un esqueleto para crear nuevos observadores (`Observer`) . La funcionalidad de actualización aquí se sobrescribirá más adelante con un comportamiento personalizado.

Script de muestra:
En este ejemplo, vimos cómo implementar y utilizar el patrón Observador, cubriendo los conceptos de un Sujeto, Observador, ConcreteSubject y ConcreteObserver.

Diferencias Entre El Observador y El Patrón de Publicación/Suscripción

Si bien es útil tener en cuenta el patrón Observer, con bastante frecuencia en el mundo de JavaScript, lo encontraremos comúnmente implementado usando una variación conocida como el patrón Publicar/Suscribir. Si bien es muy similar, hay diferencias entre estos patrones que vale la pena señalar.
El patrón de observador requiere que el observador (u objeto) que desee recibir notificaciones de temas suscriba este interés al objeto que dispara el evento (el sujeto).
Sin embargo, el patrón Publicar/Suscribir utiliza un canal de tópico/evento que se encuentra entre los objetos que desean recibir notificaciones (suscriptores) y el objeto que dispara el evento (el editor). Este sistema de eventos permite que el código defina eventos específicos de la aplicación que pueden pasar argumentos personalizados que contienen valores necesarios para el suscriptor. La idea aquí es evitar las dependencias entre el suscriptor y el editor.
Esto difiere del patrón Observador, ya que permite a cualquier suscriptor que implemente un controlador de eventos apropiado registrarse y recibir notificaciones de temas transmitidas por el editor.
Aquí hay un ejemplo de cómo se podría usar Publicar/Suscribir si se proporciona una implementación funcional que impulse `publish()`,`subscribe()` y `unsubscribe()` detrás de escena:

```
// Un nuevo administrador de correo muy simple

// Un contador de la cantidad de mensajes recibidos
var mailCounter = 0;

// Inicializar suscriptores que escucharán un tópico
// con el nombre "inbox/newMessage".

// Renderizar una vista previa de nuevos mensajes
var subscriber1 = subscribe( "inbox/newMessage", function( topic, data ) {

    // Log del tópico para fines de depuración
    console.log( "A new message was received: ", topic );

    // Utilice los datos que se pasaron de nuestro sujeto.
    // para mostrar una vista previa del mensaje al usuario
    $( ".messageSender" ).html( data.sender );
```

```
    $( ".messagePreview" ).html( data.body );

});

// Aquí hay otro suscriptor que usa los mismos datos
//para realizar una tarea diferente.

// Actualice el contador que muestra el número de mensajes
// nuevos recibidos a través del editor (publisher)

var subscriber2 = subscribe( "inbox/newMessage", function( topic, data ) {

    $('.newMessageCounter').html( ++mailCounter );

});

publish( "inbox/newMessage", [{
    sender: "hello@google.com",
    body: "Hey there! How are you doing today?"
}]);

// Luego podríamos dar de baja a nuestros suscriptores
// de recibir notificaciones de nuevos temas de la siguiente manera:
// unsubscribe( subscriber1 );
// unsubscribe( subscriber2 );
```

La idea general aquí es la promoción del acoplamiento flojo. En lugar de que los objetos individuales recurran directamente a los métodos de otros objetos, se suscriben a una tarea o actividad específica de otro objeto y reciben una notificación cuando ocurre.

Ventajas

Los patrones Observador y Publicar/Suscribir nos alientan a pensar mucho sobre las relaciones entre las diferentes partes de nuestra aplicación. También nos ayudan a identificar qué capas contienen relaciones directas que podrían reemplazarse con conjuntos de sujetos y observadores. Esto podría usarse efectivamente para dividir una aplicación en bloques más pequeños y más flojos para mejorar la administración del código y las posibilidades de reutilización.
La motivación adicional detrás del uso del patrón Observador es donde necesitamos mantener la coherencia entre los objetos relacionados sin hacer que las clases estén estrechamente vinculadas. Por ejemplo, cuando un objeto necesita poder notificar a otros objetos sin hacer suposiciones con respecto a esos objetos.
Pueden existir relaciones dinámicas entre observadores y sujetos cuando se usa cualquiera de los patrones. Esto proporciona una gran flexibilidad que puede no ser tan fácil de implementar cuando partes dispares de nuestra aplicación están estrechamente acopladas.
Si bien puede no ser siempre la mejor solución para cada problema, estos patrones siguen siendo una de las mejores herramientas para diseñar sistemas desacoplados y deben considerarse una herramienta importante en el cinturón de utilidades de cualquier desarrollador de JavaScript.

Desventajas

En consecuencia, algunos de los problemas con estos patrones en realidad se derivan de sus principales beneficios. En Publicar/Suscribir, al desacoplar a los editores de los suscriptores, a veces puede ser difícil obtener garantías de que partes particulares de nuestras aplicaciones funcionan como podríamos esperar.
Por ejemplo, los editores pueden suponer que uno o más suscriptores los están escuchando. Digamos que estamos usando tal suposición para registrar o generar errores con respecto a algún proceso de solicitud. Si el suscriptor que realiza el registro falla (o por alguna razón no funciona), el editor no tendrá una forma de ver esto debido a la naturaleza desacoplada del sistema.
Otro inconveniente del patrón es que los suscriptores ignoran bastante la existencia de los demás y son ciegos al costo de cambiar de editor. Debido a la relación dinámica entre suscriptores y editores, la dependencia de la actualización puede ser difícil de rastrear.

Implementación de Publicar /Subcribir

Publicar/Suscribirse encaja muy bien en los ecosistemas de JavaScript, en gran parte porque, en el fondo, las implementaciones de ECMAScript están basadas en eventos. Esto es particularmente cierto en entornos de navegador, ya que el DOM utiliza eventos como su principal API de interacción para la creación de secuencias de comandos.
Dicho esto, ni ECMAScript ni DOM proporcionan objetos centrales o métodos para crear sistemas de eventos personalizados en el código de implementación (con la excepción quizás del DOM3 CustomEvent, que está vinculado al DOM y, por lo tanto, no es genéricamente útil).
Afortunadamente, las bibliotecas JavaScript populares como dojo, jQuery (eventos personalizados) y YUI ya tienen utilidades que pueden ayudar a implementar fácilmente un sistema de Publicación/Suscripción con muy poco esfuerzo. A continuación podemos ver algunos ejemplos de esto:

```
// Publish

// jQuery: $(obj).trigger("channel", [arg1, arg2, arg3]);
$( el ).trigger( "/login", [{username:"test", userData:"test"}] );

// Dojo: dojo.publish("channel", [arg1, arg2, arg3] );
dojo.publish( "/login", [{username:"test", userData:"test"}] );

// YUI: el.publish("channel", [arg1, arg2, arg3]);
el.publish( "/login", {username:"test", userData:"test"} );


// Subscribe

// jQuery: $(obj).on( "channel", [data], fn );
$( el ).on( "/login", function( event ){...} );

// Dojo: dojo.subscribe( "channel", fn);
var handle = dojo.subscribe( "/login", function(data){..} );
```

```
// YUI: el.on("channel", handler);
el.on( "/login", function( data ){...} );


// Unsubscribe

// jQuery: $(obj).off( "channel" );
$( el ).off( "/login" );

// Dojo: dojo.unsubscribe( handle );
dojo.unsubscribe( handle );

// YUI: el.detach("channel");
el.detach( "/login" );
```

Para aquellos que deseen utilizar el patrón Publicar/Suscribir con JavaScript vainilla (u otra biblioteca), AmplifyJS incluye una implementación limpia y libre de bibliotecas que se puede usar con cualquier biblioteca o kit de herramientas. Radio.js (http://radio.uxder.com/), PubSubJS (https://github.com/mroderick/PubSubJS) o Pure JS PubSub de Peter Higgins (https://github.com/phiggins42/bloody-jquery-plugins/blob/55e41df9bf08f42378bb08b93efcb28555b61aeb/pubsub.js) también son alternativas similares que vale la pena consultar.

Los desarrolladores de jQuery en particular tienen bastantes otras opciones y pueden optar por usar una de las muchas implementaciones bien desarrolladas que van desde el complemento jQuery de Peter Higgins hasta la publicación (optimizada) Pub / Sub jQuery de Ben Alman en GitHub. Los enlaces a algunos de estos se pueden encontrar a continuación.

- Ben Alman's Pub/Sub gist https://gist.github.com/661855 (recomendado)
- Rick Waldron's jQuery-core style take on the above https://gist.github.com/705311
- Peter Higgins" plugin http://github.com/phiggins42/bloody-jquery-plugins/blob/master/pubsub.js.
- AppendTo's Pub/Sub in AmplifyJS http://amplifyjs.com
- Ben Truyman's gist https://gist.github.com/826794

Para que podamos apreciar cuántas de las implementaciones de JavaScript vainilla del patrón Observer podrían funcionar, veamos una versión minimalista de Publicar/Suscribir que publiqué en GitHub en un proyecto llamado pubsubz. Esto demuestra los conceptos básicos de suscripción, publicación y el concepto de cancelación de suscripción.

Opté por basar nuestros ejemplos en este código, ya que se adhiere estrechamente a las firmas de métodos y al enfoque de implementación que esperaría ver en una versión de JavaScript del patrón clásico de Observer.

Implementación de Publicar /Subcribir

Ej:016:index.js

```
var pubsub = {};

(function(myObject) {
```

```
// Almacenamiento de temas que pueden
//transmitirse o escucharse
var topics = {};

// Un identificador de tópico
var subUid = -1;

// Publique o transmita eventos de interés
// con un nombre de tópico específico y argumentos
// tales como los datos a pasar a lo largo

myObject.publish = function( topic, args ) {

    if ( !topics[topic] ) {
        return false;
    }

    var subscribers = topics[topic],
        len = subscribers ? subscribers.length : 0;

    while (len--) {
        subscribers[len].func( topic, args );
    }

    return this;
};

// Suscríbase a eventos de interés
// con un nombre de tópico específico y una
// función de devolución de llamada, que se ejecutará
// cuando se observe el tópico/evento
myObject.subscribe = function( topic, func ) {

    if (!topics[topic]) {
        topics[topic] = [];
    }

    var token = ( ++subUid ).toString();
    topics[topic].push({
        token: token,
        func: func
    });
    return token;
};

// Des-subscribe de un tema
// específico, basado en una referencia
// simbólica a la suscripción
myObject.unsubscribe = function( token ) {
    for ( var m in topics ) {

        if ( topics[m] ) {
```

```
                for ( var i = 0, j = topics[m].length; i < j; i++ ) {
                    if ( topics[m][i].token === token ) {
                        topics[m].splice( i, 1 );
                        return token;
                    }
                }
            }
        }
        return this;
    };
}( pubsub ));
```

Ejemplo: uso de nuestra implementación

Ahora podemos usar la implementación para publicar y suscribirse a eventos de interés de la siguiente manera:

Ej:016:implement.js

```
// Otro manejador de mensajes simple

// Un simple loggeador de mensajes que logea cualquier tópico y datos recibidos a través de nuestro
// suscriptor

var messageLogger = function ( topics, data ) {
    console.log( "Logging: " + topics + ": " + data );
};

// Los suscriptores escuchan los tópicos a los que se han suscrito e
// invocan una función (callback) de devolución de llamada (ej., messageLogger)
// una vez que se transmite una nueva notificación sobre ese tópico
var subscription = pubsub.subscribe( "inbox/newMessage", messageLogger );

// Publishers are in charge of publishing topics or notifications of
// interest to the application. e.g:

// Los Publicistas se encargan de publicar tópicos o notificaciones de
// interés para la aplicación. ej:
pubsub.publish( "inbox/newMessage", "hello world!" );

// or
pubsub.publish( "inbox/newMessage", ["test", "a", "b", "c"] );

// or
pubsub.publish( "inbox/newMessage", {
  sender: "hello@google.com",
  body: "Hey again!"
});

// También podemos cancelar la suscripción si ya no deseamos que se notifique
// a nuestros suscriptores
pubsub.unsubscribe( subscription );

// Una vez que se haya dado de baja, esto, por ejemplo, no dará como resultado
// que se ejecute nuestro messageLogger ya que el suscriptor
// ya no escucha
pubsub.publish( "inbox/newMessage", "Hello! are you still there?" );
```

Ejemplo: notificaciones de interfaz de usuario

A continuación, imaginemos que tenemos una aplicación web responsable de mostrar información de stock en tiempo real.

La aplicación puede tener una cuadrícula para mostrar las estadísticas de stock y un contador para mostrar el último punto de actualización. Cuando el modelo de datos cambia, la aplicación necesitará actualizar la cuadrícula y el contador. En este escenario, nuestro tema (que será la publicación de tópicos/notificaciones) es el modelo de datos y nuestros suscriptores son la cuadrícula y el contador.

Cuando nuestros suscriptores reciben una notificación de que el modelo en sí ha cambiado, pueden actualizarse en consecuencia.

En nuestra implementación, nuestro suscriptor escuchará el tema "newDataAvailable" para averiguar si hay nueva información disponible. Si se publica una nueva notificación sobre este tema, activará `gridUpdate` para agregar una nueva fila a nuestra cuadrícula que contenga esta información. También actualizará un *último* contador *actualizado* para registrar la última vez que se agregaron datos.

Ej:017:implement2.js

```
// Devuelve la hora local actual para usar en nuestra UI más tarde
getCurrentTime = function (){

   var date = new Date(),
         m = date.getMonth() + 1,
         d = date.getDate(),
         y = date.getFullYear(),
         t = date.toLocaleTimeString().toLowerCase();

        return (m + "/" + d + "/" + y + " " + t);
};

// Agrega una nueva fila de datos a nuestro componente de cuadrícula ficticia
function addGridRow( data ) {

   // ui.grid.addRow( data );
   console.log( "updated grid component with:" + data );

}

// Actualiza nuestra grilla ficticia para mostrar la hora en que se actualizó
// por última vez
function updateCounter( data ) {

   // ui.grid.updateLastChanged( getCurrentTime() );
   console.log( "data last updated at: " + getCurrentTime() + " with " + data);

}

// Actualiza la cuadrícula utilizando los datos pasados a nuestros suscriptores
gridUpdate = function( topic, data ){

  if ( data !== undefined ) {
     addGridRow( data );
     updateCounter( data );
   }

};

// Creea una suscripción al tópico newDataAvailable
```

```
var subscriber = pubsub.subscribe( "newDataAvailable", gridUpdate );

// Lo siguiente representa actualizaciones de nuestra capa de datos. Esto podría ser
// impulsado por solicitudes ajax que transmiten que los nuevos datos están disponibles
// para el resto de la aplicación.

// Publicar cambios al tópico "gridUpdated" representando nuevas entradas

pubsub.publish( "newDataAvailable", {
  summary: "Apple made $5 billion",
  identifier: "APPL",
  stockPrice: 570.91
});

pubsub.publish( "newDataAvailable", {
  summary: "Microsoft made $20 million",
  identifier: "MSFT",
  stockPrice: 30.85
});
```

Ejemplo: desacoplar aplicaciones usando la implementación Pub/Sub de Ben Alman

En el siguiente ejemplo de clasificación de películas, utilizaremos la implementación jQuery de Ben Alman de Publicar/Suscribir para demostrar cómo podemos desacoplar una interfaz de usuario. Observe cómo enviar una calificación solo tiene el efecto de publicar el hecho de que hay nuevos usuarios y datos de calificación disponibles.

Depende de los suscriptores de esos temas delegar lo que sucede con esos datos. En nuestro caso, estamos introduciendo esos nuevos datos en las matrices existentes y luego los procesamos utilizando el método `.template()` de la biblioteca Underscore para crear plantillas.

HTML/Templates

```
<script id="userTemplate" type="text/html">
   <li><%= name %></li>
</script>


<script id="ratingsTemplate" type="text/html">
   <li><strong><%= title %></strong> was rated <%= rating %>/5</li>
</script>


<div id="container">

   <div class="sampleForm">
       <p>
           <label for="twitter_handle">Twitter handle:</label>
           <input type="text" id="twitter_handle" />
       </p>
       <p>
           <label for="movie_seen">Name a movie you've seen this year:</label>
           <input type="text" id="movie_seen" />
       </p>
       <p>

           <label for="movie_rating">Rate the movie you saw:</label>
           <select id="movie_rating">
                 <option value="1">1</option>
                  <option value="2">2</option>
                  <option value="3">3</option>
                  <option value="4">4</option>
                  <option value="5" selected>5</option>

         </select>
        </p>
        <p>
```

```
            <button id="add">Submit rating</button>
        </p>
    </div>



    <div class="summaryTable">
        <div id="users"><h3>Recent users</h3></div>
        <div id="ratings"><h3>Recent movies rated</h3></div>
    </div>

</div>
```