JAVASCRIPT FRONTEND AVANZADO

CLASE 10 : PATRONES DE DISEÑO

PATRONES DE DISEÑO

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 de JavaScript. Otra forma de ver los patrones es como plantillas para la forma en que resolvemos los problemas, que pueden usarse en bastantes situaciones diferentes.

Entonces, ¿por qué es importante entender los patrones y estar familiarizado con ellos? Los patrones de diseño tienen tres beneficios principales:

- **Los patrones son soluciones comprobadas**: proporcionan enfoques sólidos para resolver problemas en el desarrollo de software utilizando técnicas comprobadas que reflejan la experiencia y los conocimientos que los desarrolladores que ayudaron a definir traen al patrón.
- **Los patrones se pueden reutilizar fácilmente:** un patrón generalmente refleja una solución lista para usar que se puede adaptar a nuestras necesidades. Esta característica los hace bastante robustos.
- **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 recordar 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, sí los admiten. 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 la aplicación**. Lo que esto significa es que cuando el código se basa en patrones probados, podemos permitirnos perder menos tiempo preocupándonos por la estructura de nuestro código y más tiempo concentrándose en la calidad de nuestra solución general. Esto se debe a que los patrones pueden alentarnos a codificar de una manera más estructurada y organizada, evitando la necesidad de refactorizarla 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 del lenguaje de programación) con la que trabajemos, se pueden aplicar patrones de diseño para mejorar la estructura de nuestro código.
- **Ciertos patrones en realidad pueden disminuir el tamaño total del tamaño de archivo de nuestro código al evitar la repetición.** Al alentar a los desarrolladores a mirar más de cerca sus soluciones para las áreas donde se pueden realizar reducciones instantáneas en la repetición, p. reduciendo el número de funciones que realizan procesos similares a favor de una única función generalizada, se puede disminuir el tamaño total de nuestra base de código. Esto también se conoce como hacer que el código sea más SECO.
- **Los patrones se agregan al vocabulario del desarrollador, lo que hace que la comunicación sea más rápida.**
- **Los patrones que se utilizan con frecuencia se pueden mejorar con el tiempo mediante el aprovechamiento de las experiencias colectivas de otros desarrolladores que utilizan 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 pueden ser mejor utilizados. Esto puede garantizar que las soluciones basadas en patrones continúen siendo más robustas que las soluciones ad-hoc [1].

CATEGORIAS

Patrones de diseño creacional

Los patrones de diseño creacional se enfocan en manejar 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 llevar a una complejidad adicional en un proyecto mientras estos patrones intentan resolver este problema controlando el proceso de creación.

Algunos de los patrones que se incluyen en esta categoría son:

- Constructor
- Factory
- Abstract
- Prototype
- Singleton
- Builder

Patrones de diseño estructural
Los patrones estructurales están relacionados con la composición de objetos e identifican formas simples de realizar relaciones entre diferentes objetos. Ayudan a garantizar que cuando una parte del sistema cambia, la estructura completa del sistema no tenga que hacer lo mismo. También ayudan a refundir partes del sistema que no se ajustan a un propósito particular en las que sí lo hacen.

Los patrones que se incluyen en esta categoría incluyen:

- Decorador
- Fachada(facade)
- Flyweight
- Adaptador
- Proxy

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

Algunos patrones de comportamiento incluyen:

- Iterator
- Mediator
- Observer
- Visitor

PATRÓN REVELADOR

Ahora que estamos un poco más familiarizados con el patrón del módulo, echemos un vistazo a una versión ligeramente mejorada: el patrón Revealing Module de Christian Heilmann.

El patrón Revealing Module surgió cuando Heilmann se sintió frustrado por 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 de tener que cambiar a la notación literal del objeto para las cosas que deseaba hacer públicas.

El resultado de su esfuerzo fue un patrón actualizado donde 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:

```
var myRevealingModule = (function () {

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

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

        function publicSetName( strName ) {
            privateVar = strName;
        }

        function publicGetName() {
            privateFunction();
        }


        // Reveal public pointers to
        // private functions and properties

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

    })();

myRevealingModule.setName( "Paul Kinlan" );
```

Como podemos observar, seguimos teniendo un módulo de todas formas pero esta vez ya comenzamos con el namespace global y luego solo exponemos un objeto literal.

PATRÓN SINGLETON

El patrón Singleton se conoce 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 crea una nueva instancia de la clase si no existe. En el caso de que una instancia ya exista, simplemente devuelve una referencia a ese objeto.

Los singletons difieren de las clases (u objetos) estáticos 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 para el código que no conoce una referencia previa a ellos para recuperarlos fácilmente. Esto se debe a que no es ni el objeto ni la "clase" devueltos por 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, los Singleton sirven 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:

```
var singleton = (function () {
  // Instance stores a reference to the Singleton
  var instancia;
  function init() {
    // Singleton
    var numeroRandomPrivado = Math.random();
    return {
      //Metodos y variables publicas
      metodoPublico : function () {
        console.log( "Todos pueden verme!" );
      },
      propiedadPublica : "Tambien soy publicoa",
      getNumeroRandom : function() {
        return numeroRandomPrivado;
      }
    };
  };
  return {
    getInstancia: function () {
      if ( !instancia ) {
        instancia= init();
```

```
      }
       return instancia;
    }
  };
})();
```

De esta forma, cada vez que siempre obtendremos el mismo objeto singleton y nunca uno distinto ya que estamos guardando una referencia del mismo en el closure sin definir un setter salvo al comienzo de la ejecución del código .

PATRÓN PUB/SUB

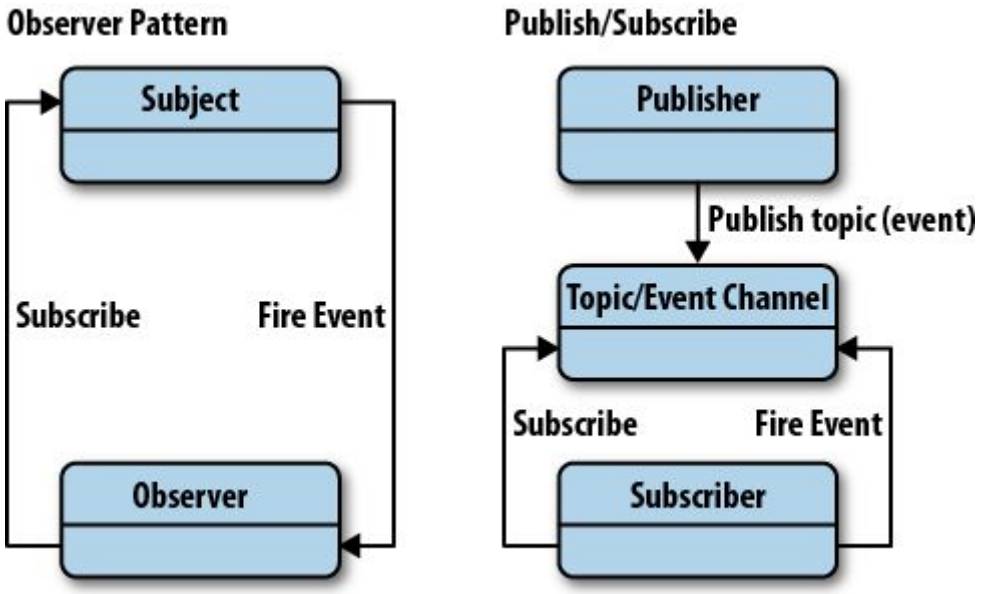
La idea general detrás del patrón Observer ó Publicación-Subscripción es la promoción del acoplamiento flexible (o desacoplamiento como también se lo denomina). En lugar de objetos únicos que invocan los métodos de otros objetos, un objeto se suscribe a una tarea o actividad específica de otro objeto y se notifica cuando ocurre. Los observadores también se llaman Suscriptores y nos referimos al objeto que se observa como el Editor (o el sujeto). Los editores notifican a los suscriptores cuando ocurren eventos.

Cuando los objetos ya no están interesados en ser notificados por el sujeto con el que están registrados, pueden anular el registro (o anular su suscripción). El sujeto, a su vez, los eliminará de la colección del observador.

A menudo es útil referirse a las definiciones publicadas de patrones de diseño que son independientes del idioma para obtener un sentido más amplio de su uso y ventajas a lo largo del tiempo. La definición del patrón de observador proporcionado en el libro de la Pandilla de Cuatro, Patrones de diseño: Elementos de software reutilizable orientado a objetos, es:

'Uno o más observadores están interesados en el estado de una materia y registran su interés con el tema al unirse ellos mismos. 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 separarse ".

Básicamente, el patrón describe sujetos y observadores que forman una relación publicación-suscripción. Un número ilimitado de objetos puede observar eventos en la materia registrándose. Una vez registrado en eventos particulares, el sujeto notificará a todos los observadores cuando el evento haya sido despedido [2].



Diferencia entre el patrón Observer y Pub/Sub

Sí, el Sujeto en el Patrón de Observador es como un Editor y el Observador puede relacionarse totalmente con un Suscriptor y sí, el Sujeto notifica a los Observadores como un Publicador generalmente notifica a sus suscriptores. Es por eso que la mayoría de los libros o artículos de Design Pattern usan la noción 'Publisher-Subscriber' para explicar el patrón de diseño del observador. Pero hay otro patrón popular llamado 'Publisher-Subscriber' y es conceptualmente muy similar al patrón Observer. La principal diferencia entre el patrón (real) 'Publisher-Subscriber' y el patrón 'Observer' es el siguiente:

En el patrón 'Publisher-Subscriber', los remitentes de mensajes, llamados editores, no programan los mensajes para enviarlos directamente a receptores específicos, llamados suscriptores.

Esto significa que el editor y el suscriptor no conocen la existencia del otro. Hay un tercer componente, llamado intermediario o agente de mensajes o bus de eventos, que es conocido tanto por el editor como por el suscriptor, que filtra todos los mensajes entrantes y los distribuye en consecuencia. En otras palabras, pub-sub es un patrón usado para comunicar mensajes entre diferentes componentes del sistema sin que estos componentes sepan algo sobre la identidad del otro. ¿Cómo filtra el agente todos los mensajes? En realidad, hay varios procesos para el filtrado de mensajes. Los métodos más populares son: basados en temas y basados en contenido.

Propongamos el siguiente problema :

```
var Orden = function(params) {
   this.params = params;
 };
  Orden.prototype = {
   guardar: function() {
     // guardar order
     console.log("Orden guardada!");
     this.enviarMail();
   },
    enviarMail: function() {
     var mailer = new Mailer();
     mailer.enviarEmailDeCompra(this.params);
   }
 };
  var Mailer = function() {};
  Mailer.prototype = {
   enviarEmailDeCompra: function(params) {
     console.log("Email enviado a " + params.userEmail);
   }
 };

var order = new Orden({ userEmail: 'juan@gmail.com'  })
order.guardar();
//"Orden guardada!"
//"Email enviado a juan@gmail.com"
```

Entendés la idea ¿no? Este código tiene algunos problemas. Probablemente, el más importante es que Order y Mailer están estrechamente relacionados. Por lo general, usted sabe que dos componentes están acoplados cuando un cambio a uno requiere un cambio en el otro, y ese es el caso. Si queremos cambiar el nombre del método enviarMailDeCompra o sus parámetros, también tendremos que cambiar la implementación de Orden. Este cambio puede parecer bastante sencillo, pero en una aplicación grande esta mala práctica significa tener una aplicación estrechamente unida donde un pequeño cambio puede terminar fácilmente en una cascada de cambios.

Otro problema que encontrará al usar ese enfoque es que tendrá que burlarse de muchas cosas en sus pruebas, convirtiéndose en una pesadilla en aplicaciones grandes.

Ahora que conocemos los problemas de este enfoque, echemos un vistazo a cómo podemos mejorar este código utilizando el patrón Publicar / Suscribir.

Vamos a comenzar creando nuestro objeto interfaz de la aplicación :

```
var EventBus = {
  topics: {},

  suscribir: function(topic, listener) {
    //crea el tópico si aun no existe
    if(!this.topics[topic]) this.topics[topic] = [];

    //Agrega un listener
    this.topics[topic].push(listener);
  },

  publicar: function(topic, data) {
    // return si el evento no existe o no tiene listeners
    if(!this.topics[topic] || this.topics[topic].length < 1)
       return;
    //Envía el evento a todos los listeners
    this.topics[topic].forEach(function(listener) {
      listener(data || {});
    });
  }
};
```

Esta es una implementación simple del patrón pub / sub, puede encontrar algunas bibliotecas que hacen una implementación completa del patrón como Radio.js, Amplify.js y muchas otras.

Ahora podemos usar nuestro nuevo objeto EventBus para enviar mensajes entre objetos

```
EventBus.suscribir ('foo', alert);
EventBus.publicar ('foo', '¡Hola, mundo!');
```

Ahora veamos cómo usar EventBus para desacoplar Order y Mailer

```
var Mailer = function() {
  EventBus.suscribir('order/new', this.enviarMailDeCompra);
};

Mailer.prototype = {
```

```
  enviarMailDeCompra : function(userEmail) {
    console.log("Email enviado a " + userEmail);
  }
};

var Orden = function(params) {
  this.params = params;
};

Orden.prototype = {
  guardarOrden: function() {
    EventBus.publicar('order/new', this.params.userEmail);
  }
};

> var mailer = new Mailer();
> var order = new Orden({userEmail: 'juan@gmail.com'});
> order.guardarOrden();
> "Email enviado a juan@gmail.com"
```

Puede ver cómo Orden y Mailer no saben nada el uno del otro. Ahora podemos cambiar la implementación de ambas funciones sin preocuparnos por el impacto de los cambios, solo se trata de preservar los eventos que estamos disparando o escuchando. Además, podemos probar ambas funciones de forma aislada sin la necesidad de objetos simulados, lo que siempre es una gran cosa [3].

1. https://addyosmani.com/resources/essentialjsdesignpatterns/book/#whatisapattern
2. https://msdn.microsoft.com/en-us/magazine/hh201955.aspx
3. http://dev.housetrip.com/2014/09/15/decoupling-javascript-apps-using-pub-sub-pattern/