DESARROLLO WEB FULL STACK

Javascript

Patrones de diseño

DWFS COR

Javascript: Patrones de diseño

Los **patrones de diseño** son unas técnicas para resolver problemas comunes en el desarrollo de *software* y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño resulta ser una solución a un problema de diseño.
Para que una solución sea considerada un patrón debe poseer ciertas características:

- Debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores.
- Debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Problema de diseño

Requerimientos arquitectónicos que encontramos a la hora de desarrollar una solución de software.

Patrón

Metodología estandarizada para resolver un problema.

Objetivos

Los patrones de diseño pretenden:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Asimismo, no pretenden:

- Imponer ciertas alternativas de diseño frente a otras.
- Eliminar la creatividad inherente al proceso de diseño.

Patrón Módulo

Nos permite definir métodos privados y públicos dentro de un objeto.

En javascript este patrón nos permite emular el comportamiento de una clase como en otros lenguajes de software orientados a objetos.

El mismo habilita que métodos y propiedades estén protegidos del scope global lo brinda una reducción general de nombres de función y variables entrando en conflicto con scripts adicionales en la página.

Ejemplo

Implementación de un módulo autocontenido con dos métodos y una propiedad privada.

```
var testModule = (function () {
  var counter = 0;

  function incrementCounter () {
    return counter++;
  }

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

  return {
    incrementCounter: incrementCounter,
    resetCounter: resetCounter,
  };
})();

//Incrementar el contador
testModule.incrementCounter();

//Ver el valor del contador y reiniciar
testModule.resetCounter();
```

Patrón Observer

Es un patrón donde un objeto (conocido como el subject/sujeto) mantiene una lista de objetos dependientes de él (observers/observadores), los cuales **automáticamente notifican sobre cualquier cambio de estado**.

Cuando un sujeto necesita notificar a los observadores sobre algo interesante que está pasando transmite una notificación a los mismos (los cuales pueden incluir datos específicos relacionados al tópico de la notificación).

Ejemplo

Implementación de un observer simple.

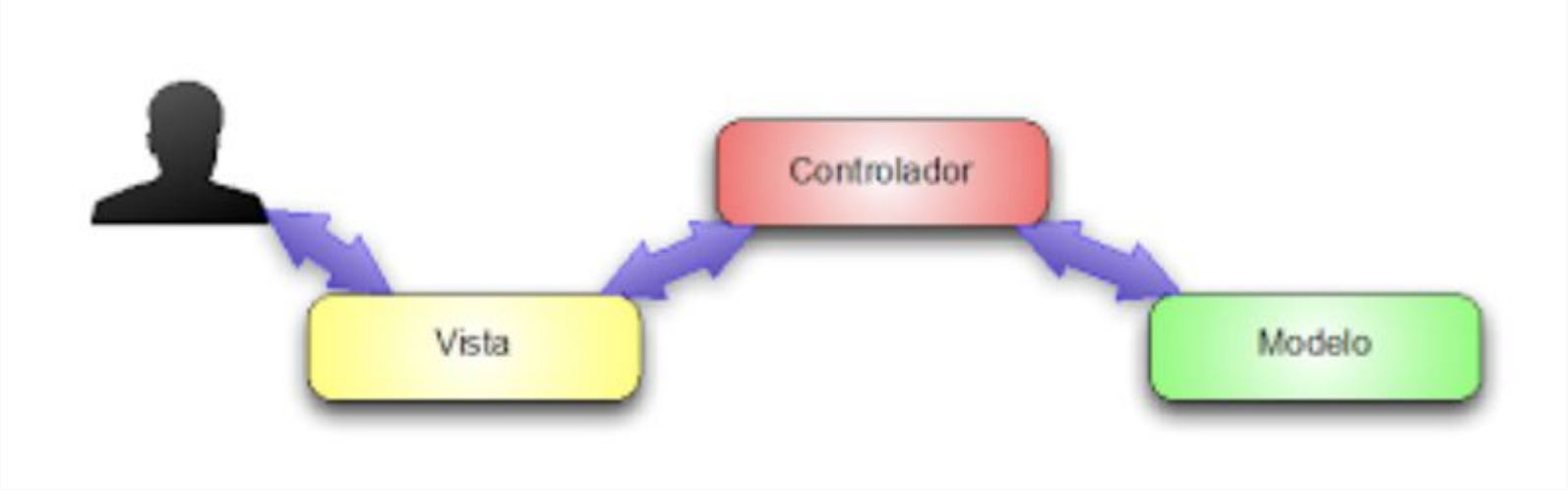
```
var Subject = function() {
  this.observers = [];
}

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

Subject.prototype.notify = function(news) {
  for(var i = 0; i < this.observers.length; i++) {
    if(this.observers[i].listener && this.observers[i].news === news) {
      this.observers[i].listener(news);
    }
  }
}

var Observer = function(news, listener) {
  this.news = news
  this.listener = listener;
}

var onPlay = new Observer('play', function(event) {
  console.log("ejecutar canción");
});

var subject = new Subject();

subject.addObserver(onPlay);
subject.notify('play');
```

Patrón MVC

Modelo-vista-controlador es un patrón de arquitectura de software, que separa los datos y la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones.

- **El Modelo:** contiene una representación de los datos que maneja el sistema, su lógica de negocio, y sus mecanismos de persistencia.
- **La Vista:** o interfaz de usuario, que compone la información que se envía al cliente y los mecanismos interacción con éste.
- **El Controlador:** actúa como intermediario entre el Modelo y la Vista, gestionando el flujo de información entre ellos y las transformaciones para adaptar los datos a las necesidades de cada uno.

Patrón MVC



Modelo

Contiene una representación de los datos que maneja el sistema

Controlador

Gestiona el flujo de información entre el modelo y la vista

```
const controlador = {
    getTareas: function(tipo) {
        return modelo.tareas[tipo];
    },

    agregarTarea: function(tipo, tarea) {
        modelo.tareas[tipo].push(tarea.toUpperCase());
    },

    eliminarTarea: function(tipo, tareaAEliminar) {
        modelo.tareas[tipo].filter(tarea => tarea === tareaAEliminar);
    },
};
```

Vista

Compone la información que se envía al cliente y los mecanismos interacción con éste.

```
const vista = {
    inicializar: function () {
        document.body.innerHTML = `
            <table>
              <tr>
                <td>Por Hacer<td>
                ${getTareas('porHacer').map(tarea => `<td>${tarea}</td>`)}
              </tr>
              <tr>
                <th>En Progreso<th>
                ${getTareas('enProgreso').map(tarea => `<td>${tarea}</td>`)}
              </tr>
              <tr>
                <th>Listas<th>
                ${getTareas('listas').map(tarea => `<td>${tarea}</td>`)}
              </tr>
            </table>
        `; // template string
    },
};
```