



PRINCIOS SOLID Y PATRONES MV*

© 2017, ACTIBYTI PROJECT SLU, Barcelona
Autor: Ricardo Ahumada



MINISTERIO
DE ENERGÍA, TURISMO
Y AGENDA DIGITAL

red.es



ESTRATEGIA DE
EMPRENDIMIENTO Y
EMPLEO JUVENIL
garantía juvenil



UNIÓN EUROPEA

Fondo Social Europeo
“El FSE invierte en tu futuro”

ÍNDICE DE CONTENIDOS

1. Caso práctico
2. Principios SOLID
3. Patrones de diseño
4. Patrón MVC
5. Patrón MVP
6. Patrón MVVM

1

CASO PRÁCTICO



Caso Práctico: BananaTube SOLID

“BananaTube” es el proyecto estrella de Banana Apps.

BananaTube será el próximo boom! de las redes sociales; permitirá a sus usuarios gestionar videos, exponerlos en su muro, comentar videos propios y de sus amigos, calificarlos y compartirlos en varios canales.

El proyecto está creciendo en complejidad, y por eso en esta etapa del proyecto se garantizar que los mecanismos de arquitectura que estamos implementando son correctos y si no modificarlos para que lo sean. Queremos que la arquitectura del proyecto sea flexible y esté abierto a una evolución ordenada.



Discutamos

- Qué es la arquitectura de la solución?
- Qué significa que sea flexible a las evoluciones?
- Qué buenas prácticas o principios deberíamos seguir?
- Es posible hacerlo en Javascript?

2

Principios SOLID

SOLID

- Solid es un acrónimo inventado por Robert C.Martin (Uncle Bob) para establecer los cinco principios básicos de la programación orientada a objetos y diseño.
- Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.
- Los 5 principios son:
 - SRP: Single Responsibility
 - OCP: Open/Close
 - LSP: Liskov's Substitution
 - ISP: Interface Segregation
 - DIP: Dependency Inversion

Consideraciones de diseño

- El valor primario del software es su facilidad para cambiar, debido sobre todo a la incorporación de nuevas funcionalidades
- Un software que no cambia o evoluciona está destinado a morir.
- Por tanto el valor secundario es la funcionalidad

- NOTA: Examinaremos una serie de ejemplos propuestos por Derek Greer
 - <http://aspiringcraftsman.com/2011/12/08/solid-javascript-single-responsibility-principle/>

SRP: Single Responsibility Principle

Una clase debe tener sólo (una responsabilidad) una razón para cambiar

- Principio de Responsabilidad Única
- Robert C. Martin → Libro Agile Software Development, Principles, Patterns, and Practices
- **Porqué:**
 - Si hay 2 razones para cambiar → 2 equipos pueden trabajar en el mismo código por 2 razones diferentes
 - Extensión de requerimientos

Ejemplo: Carrito compra (derekgreer)

<http://jsfiddle.net/derekgreer/Yvhwy/>

The screenshot shows a JSFiddle interface with two code snippets. The first snippet, titled 'Product', defines a constructor function that returns an object with methods to get the ID and description. The second snippet, titled 'Cart', defines a constructor function that maintains an array of items and provides a method to add new items.

```
JavaScript   HTML   CSS   Result   Edit in JSFiddle   Cloud
```

```
function Product(id, description) {
    this.getId = function () {
        return id;
    };
    this.getDescription = function () {
        return description;
    };
}

function Cart(eventAggregator) {
    var items = [];

    this.addItem = function (item) {
        items.push(item);
    };
}
```

- Este código **mezcla lógica de negocio y presentación** → Si queremos JSON en lugar de HTML?
- Separar responsabilidades

Ejemplo: Carrito compra

- Separando la lógica de negocio y generando una clase específica para la parte de presentación
- Para ello se debe generar otra clase: cartView, que se dedicaría solo a generar la salida en distintos formatos.
- De esta manera, cada clase tiene su responsabilidad

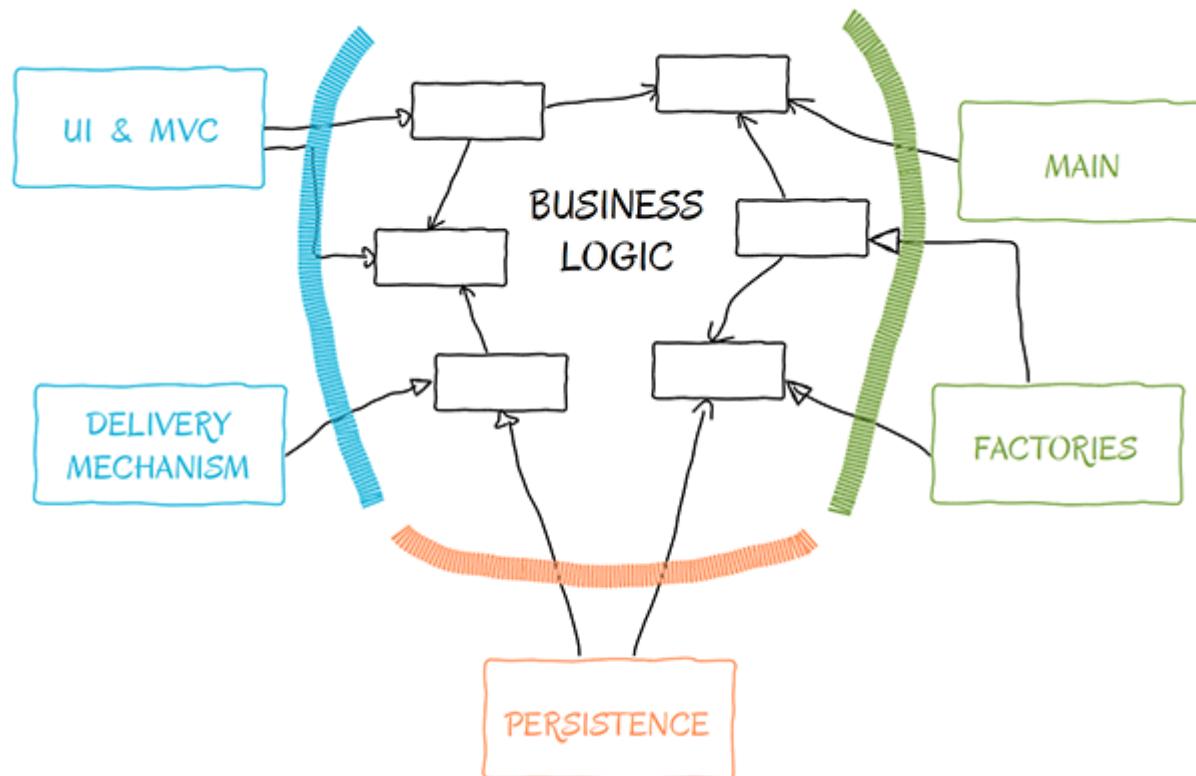
<http://jsfiddle.net/derekgreer/RPuNx/>?utm_source=website&utm_medium=embed&utm_campaign=RPuNx

The screenshot shows a JSFiddle interface with the following code:

```
JavaScript   HTML   CSS   Result   Edit in JSFiddle
```

```
function Event(name) {
    this._handlers = [];
    this.name = name;
}
Event.prototype.addHandler = function(handler) {
    this._handlers.push(handler);
};
Event.prototype.removeHandler = function(handler) {
    for (var i = 0; i < handlers.length; i++) {
        if (this._handlers[i] == handler) {
            this._handlers.splice(i, 1);
            break;
        }
    }
};
Event.prototype.fire = function(eventArgs) {
```

En un nivel superior



OCP: Open/Close Principle

Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para la extensión, pero cerradas para su modificación.

- Principio Abierto-Cerrado
- Bertrand Meyer → Libro Object-Oriented Software Construction
- Complementario con SRP
- Porqué:
 - Cuando una nueva funcionalidad es necesaria, no deberíamos modificar nuestro software, si no escribir nuevo código que añada funcionalidad al existente
 - Nuevas características sin “redeploy” de los binarios existentes

OCP: Open/Close Principle

- Una relación directa entre dos clases (fig) viola el OCP



- *User* usa *Logic* directamente
- Problema: cuando necesitemos implementar un segundo tipo de *Logic* y queramos que *User* pueda usar ambas, sería necesario cambiar la *Logic* existente
- No tenemos manera de proveer nueva *Logic* sin afectar la actual
- Además en lenguajes compilados, seguramente *User* tendría que cambiar

Ejemplo: Renderizador de preguntas

<http://jsfiddle.net/derekgreer/YeFcJ/>?utm_source=website&utm_medium=embed&utm_campaign=YeFcJ

The screenshot shows a JSFiddle interface with the following code:

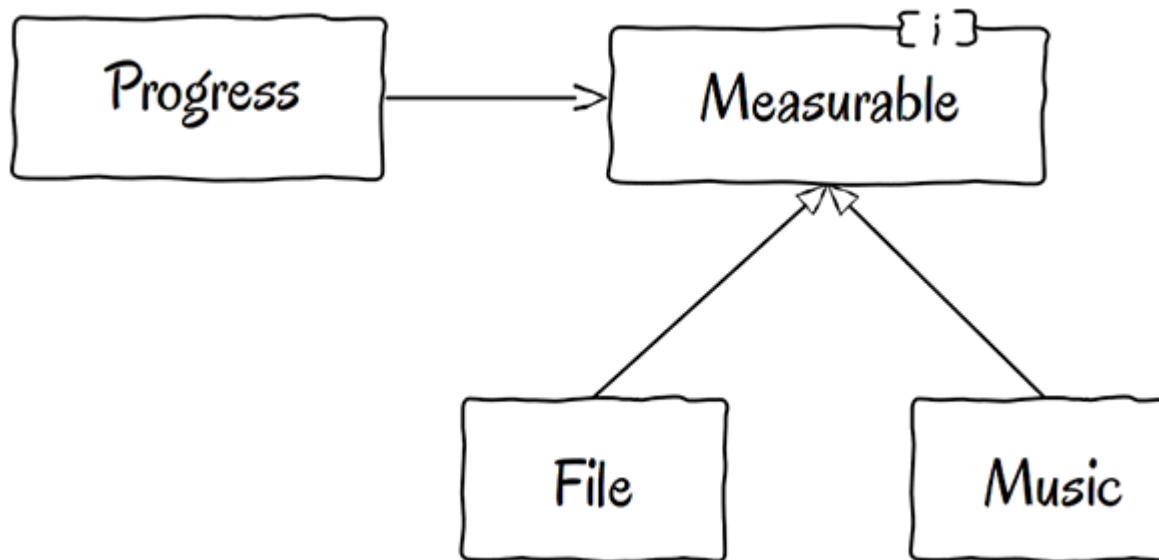
```
JavaScript   HTML   CSS   Result   Edit in JSFiddle   Cloud icon
```

```
var AnswerType = {
    Choice: 0,
    Input: 1
};

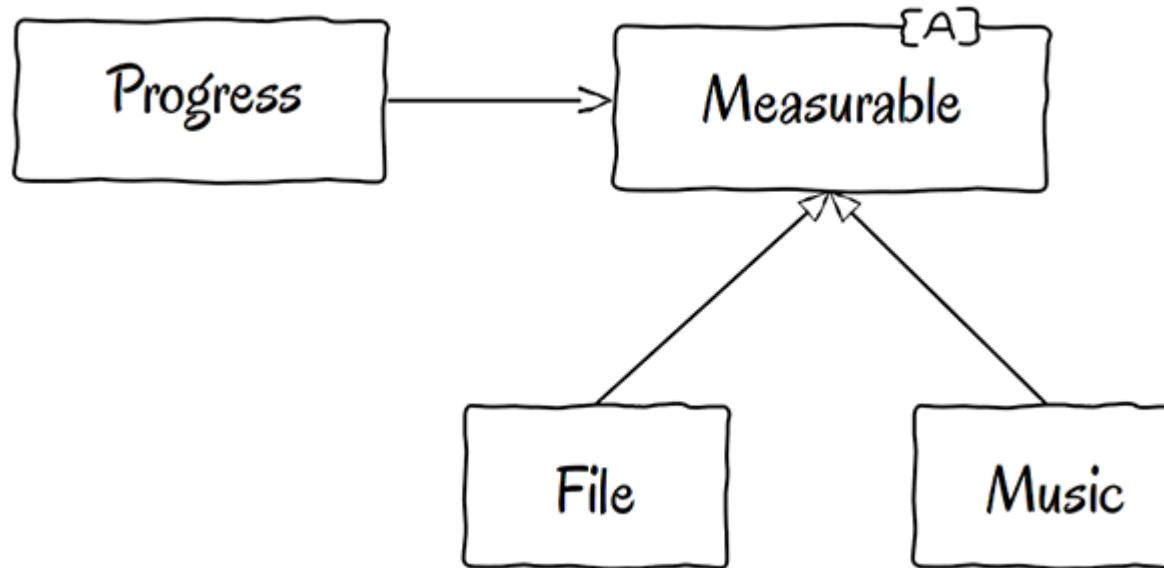
function question(label, answerType, choices) {
    return {
        label: label,
        answerType: answerType,
        choices: choices
    };
}
```

- En este ejemplo, un objeto de vista contiene un método de render que genera preguntas basadas en cada tipo de pregunta recibida.
- Añadir nuevos tipos de entrada requeriría agregar nuevas condiciones dentro del método render.
- Esto violaría el principio abierto/cerrado.

Solución: usar Strategy Design Pattern



Solución II: usar Template Method Design Pattern



Ejemplo: Renderizador de preguntas

<http://jsfiddle.net/derekgreer/eNcVp/>



The screenshot shows a JSFiddle interface with the 'JavaScript' tab selected. The code in the editor is:

```
function questionCreator(spec, my) {
    var that = {};

    my = my || {};
    my.label = spec.label;

    my.renderInput = function() {
        throw "not implemented";
    };

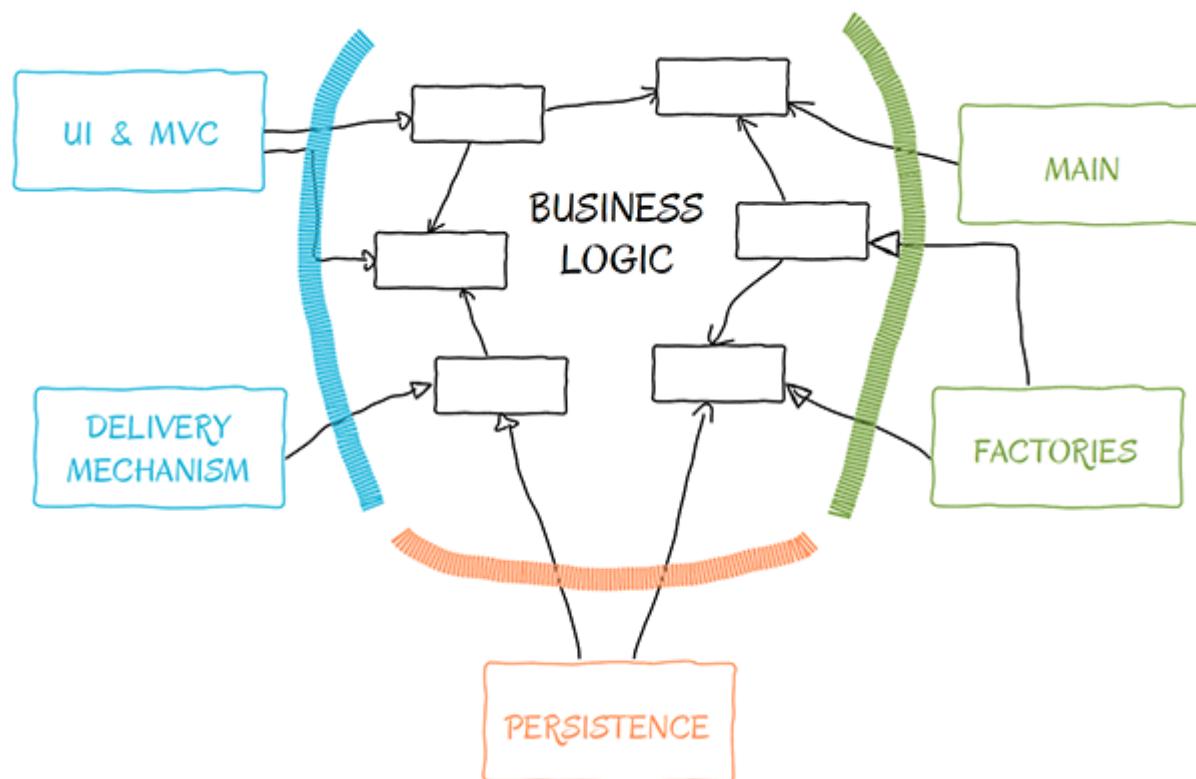
    that.render = function(target) {
        ...
    };
}
```

- Se ha factorizado el código responsable de crear preguntas en un constructor funcional denominado `questionCreator`, que utiliza el patrón Template Method para delegar la creación de cada respuesta a los tipos que extienden.

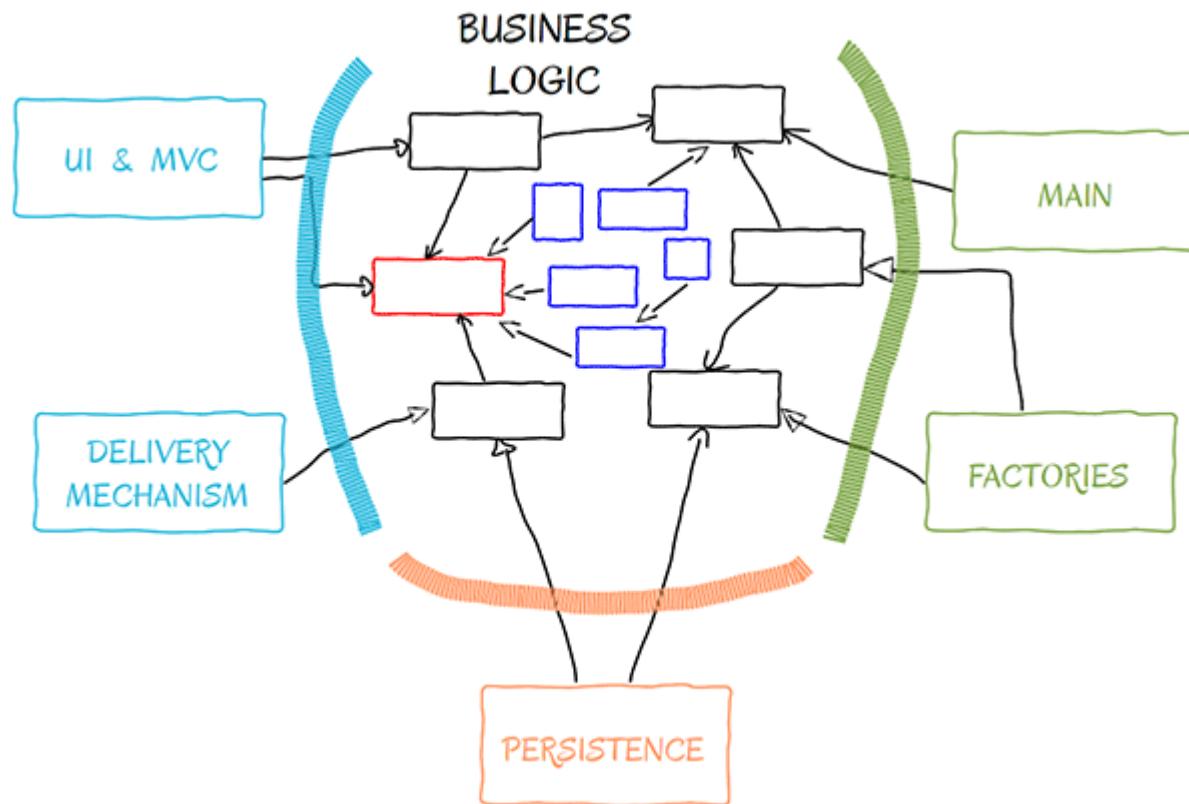
OCP: Los límites

- Cualquier clase usando directamente otra viola el OCP → es necesario desacoplarlas a través de interfaces
- **El límite:**
 - Cuando es más difícil respetar el OCP que modificar el código existente, o
 - El coste de arquitectura no justifica el cambio del código existente

En un nivel superior



En un nivel superior



LSP: Liskov Substitution Principle

- Sea $q(x)$ una propiedad probable sobre objetos x de tipo T . Entonces $q(y)$ debería ser probable para objetos y de tipo S donde S es un subtipo de T . (Barbara Liskov)

Los subtipos deben ser sustituibles
por sus tipos de base

- Robert C. Martin
- Porqué:
 - Una subclase debe sobreescribir los métodos de la clase padre de manera que no rompa la funcionalidad desde el punto de vista del cliente.

Ejemplo: Vehículo

```
function Vehicle(my) {
    my = my || {};
    my.speed = 0;
    my.running = false;

    this.speed = function() {
        return my.speed;
    };
    this.start = function() {
        my.running = true;
    };
    this.stop = function() {
        my.running = false;
    };
    this.accelerate = function() {
        my.speed++;
    };
    this.decelerate = function() {
        my.speed--;
    };
    this.state = function() {
        if (!my.running) {
            return "parked";
        } else if (my.running && my.speed) {
            return "moving";
        } else if (my.running) {
            return "idle";
        }
    };
}
```

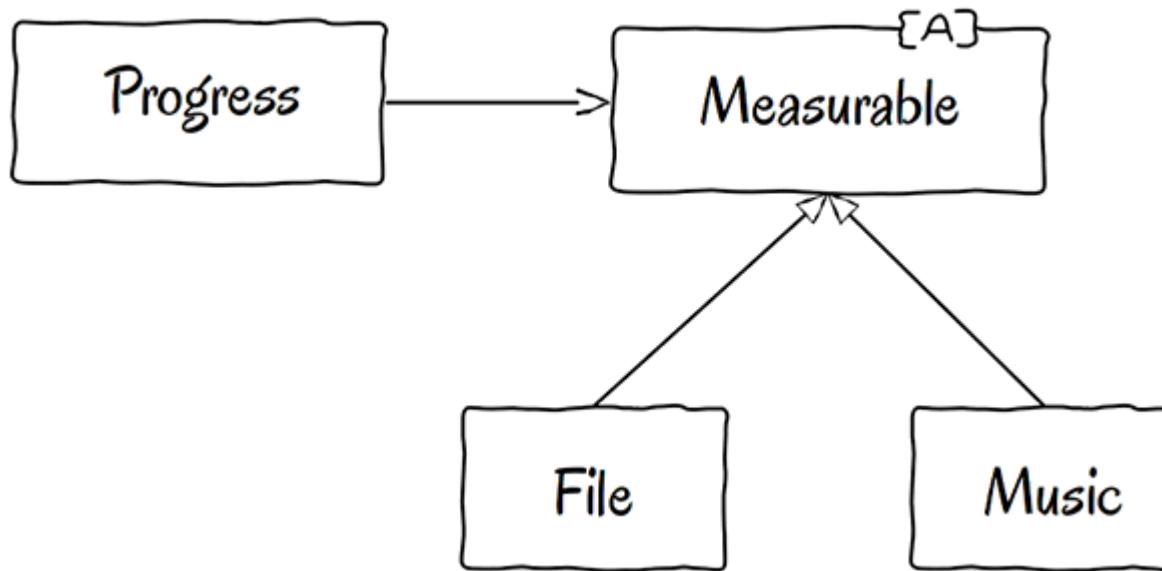
```
function FastVehicle(my) {
    my = my || {};

    var that = new Vehicle(my);
    that.accelerate = function() {
        my.speed += 3;
    };
    return that;
}
```

```
var maneuver = function(vehicle) {
    write(vehicle.state());
    vehicle.start();
    write(vehicle.state());
    vehicle.accelerate();
    write(vehicle.state());
    write(vehicle.speed());
    vehicle.decelerate();
    write(vehicle.speed());
    if (vehicle.state() != "idle") {
        throw "The vehicle is still moving!";
    }
    vehicle.stop();
    write(vehicle.state());
};
```

- Si usamos objetos FastVehicle con la función maneuver, todo va OK.
- Pero si usamos objetos Vehicle, falla.
- Los objetos FastVehicle no son sustituibles por Vehicle (clase base)

Ejemplo: Abstracción de Vehículo



- LSP está fuertemente relacionado con OCP:
 - Una violación del LSP es una violación latent del OCP.

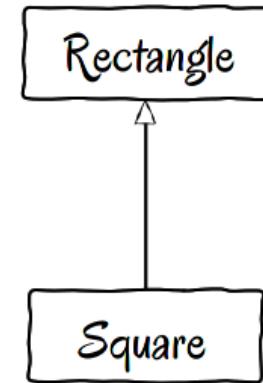
Mitigando la violación de LSP

- ¿Cómo podemos evitar las violaciones del principio de sustitución de Liskov? → Lamentablemente, esto no siempre es posible.
- Para evitar las violaciones de LSP por completo, necesitaríamos anticipar cada forma única en que una biblioteca podría ser utilizada.
- Hay algunas estrategias para evitar las violaciones de LSP:
 - Contratos → usar el método de Diseño por Contratos de Meyer (http://en.wikipedia.org/wiki/Design_by_contract)
 - Evitar la herencia → Favorecer la composición sobre la herencia

Problema: Rectángulos y cuadrados

```
var rectangle = {  
    length: 0,  
    width: 0  
};
```

```
var square = {};  
  
(function() {  
    var length = 0, width = 0;  
    Object.defineProperty(square, "length", {  
        get: function() { return length; },  
        set: function(value) { length = width = value; }  
    });  
    Object.defineProperty(square, "width", {  
        get: function() { return width; },  
        set: function(value) { length = width = value; }  
    });  
})();
```



➤ Función de prueba

```
var g = function(rectangle) {  
    rectangle.length = 3;  
    rectangle.width = 4;  
    write(rectangle.length);  
    write(rectangle.width);  
    write(rectangle.length * rectangle.width);  
};
```

Problema: Rectángulos y cuadrados

- Si probamos g() con un objeto rectángulo funciona bien
- Si enviamos un cuadrado al cliente, no debería romperse, no?
 - Cuando el método se invoca con cuadrado, el producto es 16 en lugar del valor esperado de 12.
 - Nuestro objeto cuadrado viola el principio de sustitución de Liskov con respecto a la función g.
 - En este caso, la presencia de las propiedades de longitud y anchura fue una indicación de que nuestra cuadricula podría no llegar a ser 100% compatible con rectángulo, pero no siempre tendremos tales consejos obvios.
- Corregir esta situación probablemente requeriría un rediseño de los objetos de forma y de la aplicación consumidora. Un enfoque más flexible podría ser definir rectángulos y cuadrados en términos de polígonos.
- Independientemente de ello, lo importante de este ejemplo es que el Principio de sustitución de Liskov no es sólo relevante para la herencia, sino para cualquier enfoque en el que un comportamiento esté sustituyendo por otro.

ISP: Interface Segregation Principle

Ninguna clase cliente debe ser obligado a depender de métodos que no utiliza.

- Su objetivo es comunicar al código de la clase cliente cómo usar el modulo
- Complemento de SRP
- Porqué:
 - En todas las aplicaciones modulares debe haber algún tipo de interfaz de la que el cliente pueda depender.
 - Interfaces o patrones de diseño - Façades
 - Nos fuerza a pensar desde el punto de vista del cliente de nuestro código

Interfaces JavaScript

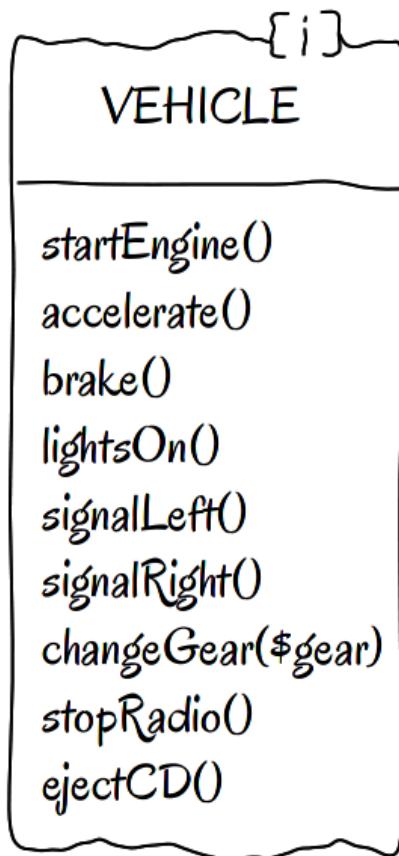
- Entonces, ¿qué tiene que ver este principio con JavaScript? Después de todo, JavaScript no tiene interfaces, ¿verdad? Si por interfaces se entiende algún tipo abstracto proporcionado por el lenguaje para establecer contratos y permitir el desacoplamiento, entonces tal afirmación sería correcta.
- Sin embargo, JavaScript tiene otro tipo de interfaz.
- En el libro Diseño de patrones - Elementos de software orientado a objetos reutilizables, Gamma et al., Encontramos la siguiente definición de una interfaz:

Every operation declared by an object specifies the operation's name, the objects it takes as parameters, and the operation's return value. This is known as the operation's signature. The set of all signatures defined by an object's operations is called the interface to the object. An object's interface characterizes the complete set of requests that can be sent to the object.

- Es decir, todos los objetos tienen una interface implícita

ISP: Interface Segregation Principle

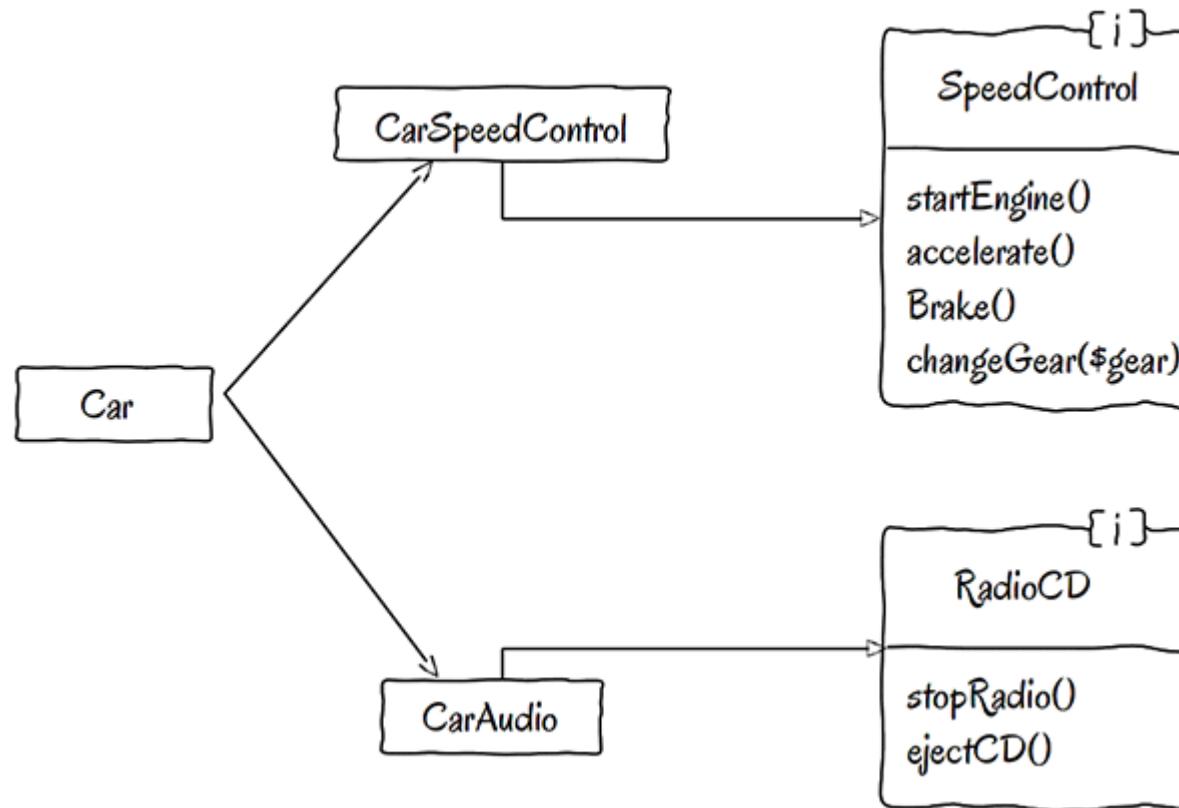
- Cómo deben ser estas interfaces?



- Exponer todas las funcionalidades de nuestro modulo
- Consecuencias:
 - Una clase gigante (Car o Bus) que implementa todos los métodos de la interface → Evitar a toda costa.
 - O, Muchas pequeñas clases (LightsControl, SpeedControl, or RadioCD) que implementan la interface, pero proveyendo utilidad solo para las partes que ellas implementan.
- Ninguna es buena

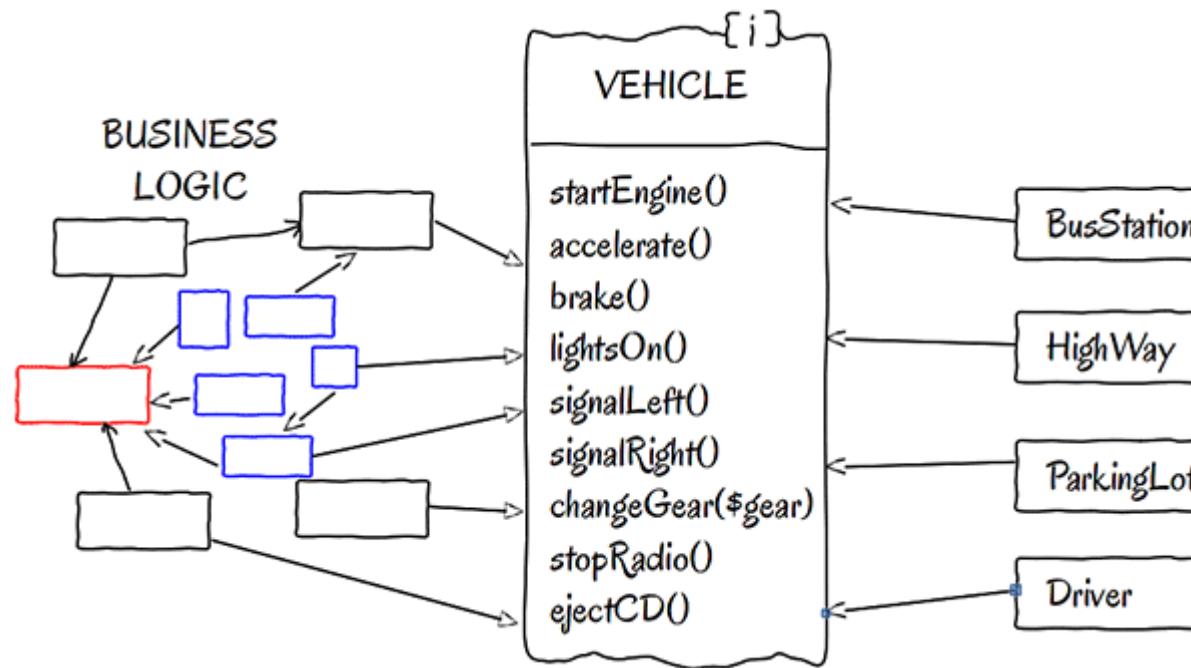
ISP: Interface Segregation Principle

- Romper la interface en piezas especializadas:



ISP: Interface Segregation Principle

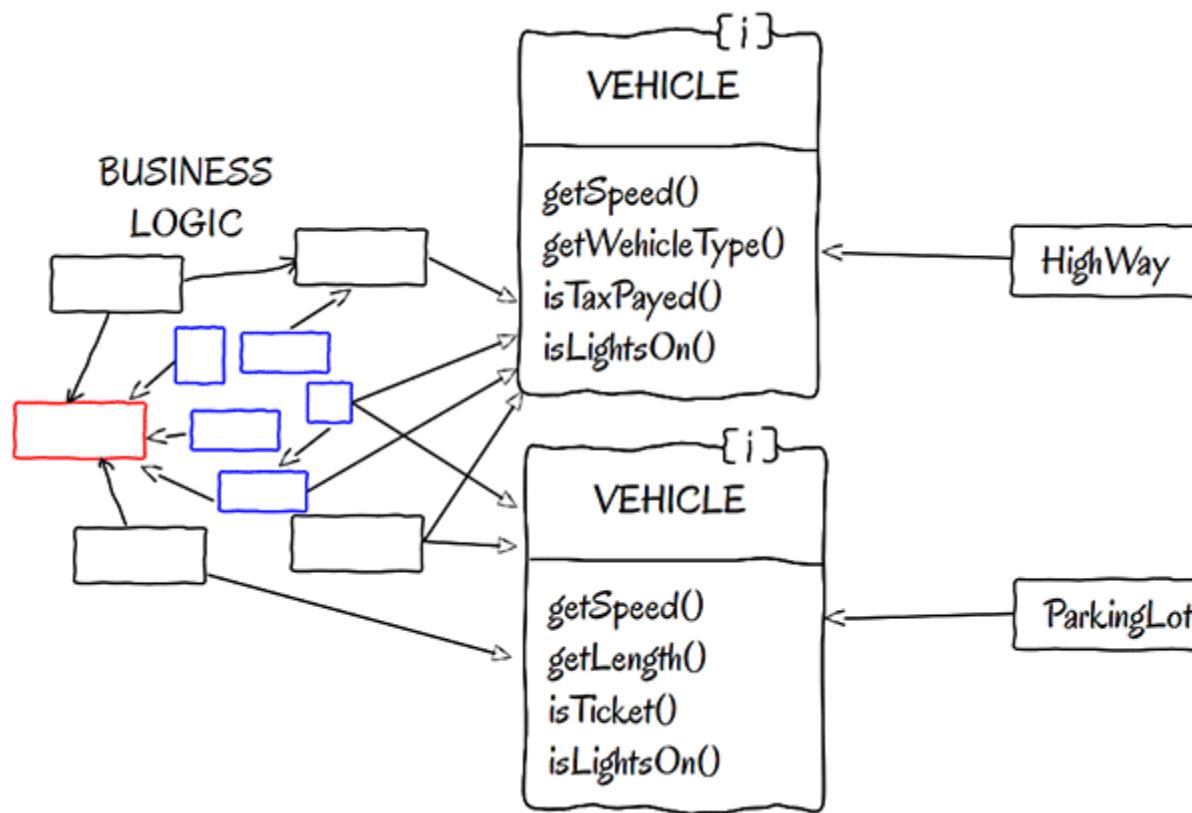
- Queremos proveer a nuestro cliente métodos para usar nuestro modulo (de tipo vehículo)



- Pero...no todos los clientes dependen de todos los métodos
- En consecuencia cambios en métodos que ni siquiera usa el cliente, pueden suponer cambios en el propio cliente

ISP: Interface Segregation Principle

- La solución es segregar la Interface



ISP: Interface Segregation Principle

- La interfaces pertenecen a los clientes, no a la lógica de negocio.
- Debemos diseñarlas de manera que se ajusten a los clientes.
- Pueden usar/necesitar varias interfaces
- En la lógica de negocio, una clase puede implementar varias interfaces, si es necesario.

Ejemplo two-way data binding

```
var exampleBinder = {};
exampleBinder.modelObserver = (function() {
  /* private variables */
  return {
    observe: function(model) {
      /* code */
      return newModel;
    },
    onChange: function(callback) {
      /* code */
    }
  }
})();

exampleBinder.viewAdaptor = (function() {
  /* private variables */
  return {
    bind: function(model) {
      /* code */
    }
  }
})();
```

```
exampleBinder.bind = function(model) {
  /* private variables */
  exampleBinder.modelObserver.onChange(/* callback */);
  var om = exampleBinder.modelObserver.observe(model);
  exampleBinder.viewAdaptor.bind(om);
  return om;
};
```

Ejemplo two-way data binding

- El listado anterior presenta una librería llamada exampleBinder cuyo propósito es facilitar la vinculación de datos de dos vías.
- La interfaz pública de la biblioteca está representada por el método bind.
- Las responsabilidades de notificación de cambio y de interacción de vista se han separado en los objetos modelObserver y viewAdaptor respectivamente para permitir implementaciones alternativas.
- Estos objetos representan las implementaciones de las interfaces esperadas por el método bind.
- Cualquier objeto que se adhiera al comportamiento semántico representado por estas interfaces puede ser sustituido por las implementaciones por defecto.
- Aunque el lenguaje JavaScript no puede proporcionar tipos de interfaz para ayudar a especificar el contrato de un objeto, la interfaz implícita del objeto sigue siendo un contrato para los clientes dentro de una aplicación.

DIP: Dependency Inversion Principle

- A. Módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
- B. Abstracciones no deben depender de detalles. Los detalles deben depender de abstractciones.

- Robert C. Martin
- Porqué:
 - Nos guía/ayuda a respetar todos los otros principios
 - (Casi) obliga a respetar OCP
 - Permiten separar las responsabilidades.
 - Hace utilizar correctamente los subtipos.
 - Ofrecerle la oportunidad de segregar interfaces

DIP y JavaScript

- Como lenguaje dinámico, JavaScript no requiere el uso de abstracciones para facilitar el desacoplamiento.
- Por lo tanto, la estipulación de que las abstracciones no deben depender de detalles no es particularmente relevante para las aplicaciones JavaScript.
- Sin embargo, la estipulación de que los módulos de alto nivel no deben depender de módulos de bajo nivel es relevante.
- Si un módulo de alto nivel está acoplado a un módulo de bajo nivel, está acoplado tanto a la interfaz semántica como a la definición física de la interfaz definida dentro del módulo de bajo nivel.
- Por tanto, las dependencias de módulos de alto nivel deben invertirse tanto para dependencias de librerías de terceros, como para módulos nativos de bajo nivel.

Ejemplo: Track Map

```

$.fn.trackMap = function(options) {
  var defaults = {
    /* defaults */
  };
  options = $.extend({}, defaults, options);

  var mapOptions = {
    center: new google.maps.LatLng(options.latitude,options.longitude),
    zoom: 12,
    mapTypeId:google.maps.MapTypeId.ROADMAP
  },
    map = new google.maps.Map(this[0], mapOptions),
    pos = new google.maps.LatLng(options.latitude,options.longitude);

  var marker = new google.maps.Marker({
    position: pos,
    title: options.title,
    icon: options.icon
  });

  marker.setMap(map);

  options.feed.update(function(latitude, longitude) {
    marker.setMap(null);
    var newLatLng = new google.maps.LatLng(latitude, longitude);
    marker.position = newLatLng;
    marker.setMap(map);
    map.setCenter(newLatLng);
  });

  return this;
};

```

```

var updater = (function() {
  // private properties

  return {
    update: function(callback) {
      updateMap = callback;
    }
  };
})();

$("#map_canvas").trackMap({
  latitude: 35.044640193770725,
  longitude: -89.98193264007568,
  icon: 'http://bit.ly/zjnGDe',
  title: 'Tracking Number: 12345',
  feed: updater
});

```

Ejemplo: Track Map - implicaciones

- Dado que la función trackMap está acoplada semánticamente a la API de Google Maps, cambiar a un proveedor de mapas diferente requeriría que la función trackMap se reescribiera o; que se escribiera un adaptador para adaptar otro proveedor de mapas a la interfaz específica de Google.
- Para invertir el acoplamiento semántico a la librería de Google Maps, necesitamos rediseñar la función trackMap para tener un acoplamiento semántico a una interfaz implícita que representa de forma abstracta la funcionalidad necesaria para un proveedor de mapeo.
- Luego solo necesitaríamos implementar un objeto que adapta esta interfaz a la API de Google Maps.

Ejemplo: Track Map

```
$.fn.trackMap = function(options) {  
    var defaults = {  
        /* defaults */  
    };  
  
    options = $.extend({}, defaults, options);  
  
    options.provider.showMap(  
        this[0],  
        options.latitude,  
        options.longitude,  
        options.icon,  
        options.title);  
  
    options.feed.update(function(latitude, longitude) {  
        options.provider.updateMap(latitude, longitude);  
    });  
  
    return this;  
};
```

```
$("#map_canvas").trackMap({  
    latitude: 35.044640193770725,  
    longitude: -89.98193264007568,  
    icon: 'http://bit.ly/zjnGDe',  
    title: 'Tracking Number: 12345',  
    feed: updater,  
    provider:  
        trackMap.googleMapsProvider  
});
```

Ejemplo: Track Map

- En esta versión, se ha rediseñado la función trackMap para expresar sus necesidades en términos de una interfaz genérica de un proveedor de mapas
- Se ha trasladado los detalles de la implementación a un componente googleMapsProvider separado que se puede agrupar como un módulo JavaScript independiente.

```
trackMap.googleMapsProvider = (function() {  
    var marker, map;  
  
    return {  
        showMap: function(element, latitude, longitude, icon, title) {  
            var mapOptions = {  
                center: new google.maps.LatLng(latitude, longitude),  
                zoom: 12,  
                mapTypeId: google.maps.MapTypeId.ROADMAP  
            },  
            pos = new google.maps.LatLng(latitude, longitude);  
  
            map = new google.maps.Map(element, mapOptions);  
  
            marker = new google.maps.Marker({  
                position: pos,  
                title: title,  
                icon: icon  
            });  
  
            marker.setMap(map);  
        },  
        updateMap: function(latitude, longitude) {  
            marker.setMap(null);  
            var newLatLng = new google.maps.LatLng(latitude, longitude);  
            marker.position = newLatLng;  
            marker.setMap(map);  
            map.setCenter(newLatLng);  
        }  
    };  
})();
```



Pongámoslo en práctica

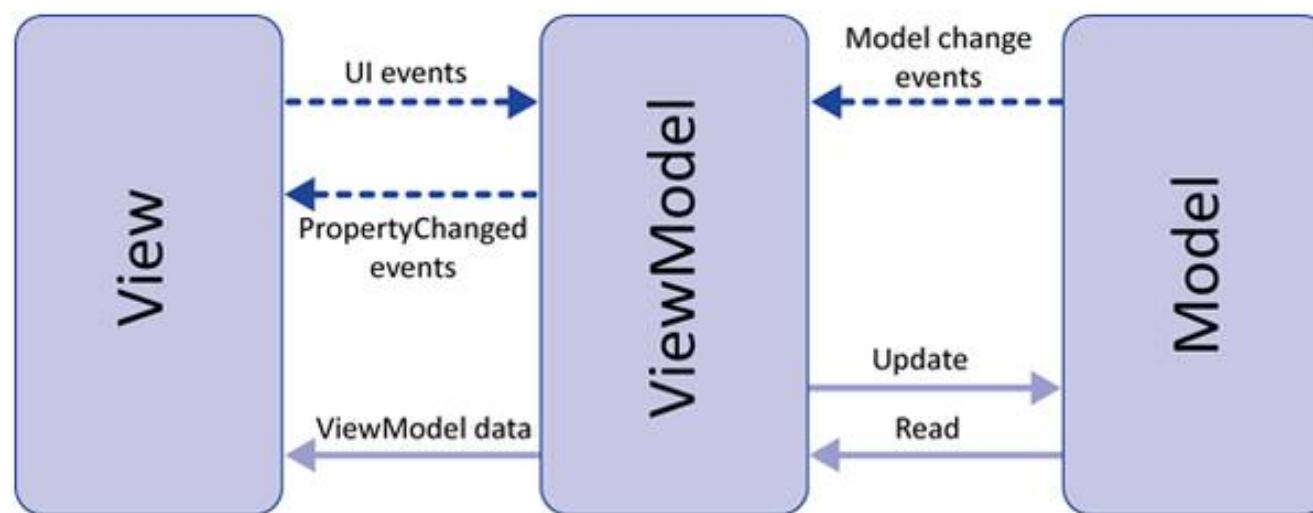
- Tenemos un cliente que, pasándole un array de formas, calcula su área: AreaCalculator.
- Queremos añadir formas tridimensionales (esferas, cuboides) y poder calcular su volumen (además de su área).
- Implementa una arquitectura que permita cumplir con los principios: SRP, OCP, LSP, ISP e IDP

3

Patrones de diseño

Patrones de diseño

- Un patrón es una solución reutilizable que puede aplicarse a problemas comunes en el diseño de software -en nuestro caso- en el desarrollo de aplicaciones web JavaScript.
- Son plantillas para resolver problemas de desarrollo, que se pueden utilizar en diferentes situaciones.
- Los patrones no son una solución exacta. Es importante que recordemos que el papel de un patrón es simplemente proveernos con un esquema de solución.



Ventajas

- **Los patrones son soluciones probadas:** Proporcionan enfoques sólidos para resolver problemas de desarrollo de software, usando técnicas probadas que reflejan la experiencia y las ideas que los desarrolladores que ayudaron a definirlos.
- **Los patrones se pueden reutilizar fácilmente:** Un patrón por lo general refleja una solución que puede adaptarse a nuestras propias necesidades. Esta característica los hacen bastante robustos.
- **Los patrones pueden ser expresivos:** cuando observamos un patrón generalmente hay una estructura y vocabulario definidos para la solución presentada, que ayuda a expresar soluciones muy grandes de manera elegante.

Patrones de diseño Javascript

- Comprender bien los patrones es importante para poder identificar los problemas específicos para los que mejor se adaptan. Esto hará más fácil integrarlos en nuestras arquitecturas de aplicación.
- Los patrones se pueden dividir en tres tipos:
 - **Patrones creacionales:** se enfocan en formas de crear objetos o clases. Esto puede sonar sencillo (y lo es en algunos casos). Las aplicaciones grandes necesitan controlar el proceso de creación de objetos.
 - **Patrones de diseño estructural:** se centran en formas de gestionar las relaciones entre objetos para que la aplicación se diseñe de forma escalable. Un aspecto clave de los patrones estructurales es asegurar que un cambio en una parte de su aplicación no afecta a todas las demás partes.
 - **Patrones de comportamiento:** se centran en la comunicación entre objetos.

Las clases en Javascript

- Al leer acerca de los patrones de diseño, a menudo se hacen referencias a clases y objetos.
- Esto puede ser confuso, ya que JavaScript realmente no tiene el constructo de “clase”, un término más correcto es **“tipo de datos”**.

Tipos de datos en JavaScript

- JavaScript es un lenguaje orientado a objetos donde los objetos heredan de otros objetos en un concepto conocido como herencia prototípica.
- Se puede crear un tipo de datos definiendo lo que se llama una función constructora:

```
function Person(config) {  
    this.name = config.name;  
    this.age = config.age;  
}  
  
Person.prototype.getAge = function() {  
    return this.age;  
};
```

```
var tilo = new Person({name:"Tilo", age:23 });  
console.log(tilo.getAge());
```

Privacidad

- Otro problema común en JavaScript es que no hay un verdadero sentido de las variables privadas.
- Podemos usar “**closures**” para simular la privacidad.

➤ Declaración de closure

```
var retinaMacbook = (function() {  
    //Private variables  
    var RAM, addRAM;  
  
    RAM = 4;  
  
    //Private method  
    addRAM = function (additionalRAM) {  
        RAM += additionalRAM;  
    };  
  
    return {  
        //Public variables and methods  
        USB: undefined,  
        insertUSB: function (device) {  
            this.USB = device;  
        },  
  
        removeUSB: function () {  
            var device = this.USB;  
            this.USB = undefined;  
            return device;  
        }  
    };  
});
```

Ejemplo

➤ Uso

```
retinaMacbook.insertUSB("myUSB");  
console.log(retinaMacbook.USB); //logs out "myUSB"  
console.log(retinaMacbook.RAM) //logs out undefined
```

Patrones de diseño Creacionales

Builder Pattern

- La aplicación del patrón del Builder nos permite construir objetos sólo especificando el tipo y el contenido del objeto. No tenemos que crear explícitamente el objeto.

```
var myDiv = $('

This is a div.</div>');


```

//myDiv now represents a jQuery object referencing a DOM node.

```
var someText = $(''');
```

//someText is a jQuery object referencing an HTMLElement

```
var input = $(''');
```

- La variable **\$** adopta el patrón Builder en jQuery.
- En el ejemplo, se devolvió un objeto jQuery DOM y se tuvo acceso a todos los métodos proporcionados por la librería jQuery, pero en ningún momento llamamos explícitamente a *document.createElement*.
- La librería JS manejó todo eso por debajo.

Prototype Pattern

- Se crean objetos basados en una plantilla de objeto existente a través de la clonación.

```
var Person = {  
    numFeet: 2,  
    numHeads: 1,  
    numHands:2  
};
```

//Object.create takes its first argument and applies it to the prototype of your new object.
var tilo = Object.create(Person);

```
console.log(tilo.numHeads); //outputs 1  
tilo.numHeads = 2;  
console.log(tilo.numHeads) //outputs 2
```

Patrones de diseño Estructurales

Composite Pattern

- El patrón Composite dice que un grupo de objetos puede ser tratado de la misma manera que un objeto individual del grupo.

```
$('.myList').addClass('selected');
$('#myItem').addClass('selected');

//dont do this on large tables, it's just an example.
$("#dataTable tbody tr").on("click", function(event){
    alert($(this).text());
});

$('#myButton').on("click", function(event) {
    alert("Clicked.");
});
```

- La mayoría de las librería JavaScript proporcionan una API consistente, independientemente de si estamos tratando con un único elemento DOM o un array de elementos DOM.

Facade Pattern

- El patrón de fachada proporciona al usuario una interfaz sencilla, mientras que oculta la complejidad subyacente.

```
$(document).ready(function() {  
  
    //all your code goes here...  
  
});
```

- El método `ready ()` implementa una fachada.
- Si se echa un vistazo a la fuente, se encontraría toda la complejidad subyacente (ver siguiente diapositva).

Código fuente método ready()

```
ready: (function() {
    ...
    //Mozilla, Opera, and Webkit
    if (document.addEventListener) {
        document.addEventListener("DOMContentLoaded", idempotent_fn, false);
        ...
    }
    //IE event model
    else if (document.attachEvent) {
        // ensure firing before onload; maybe late but safe also for iframes
        document.attachEvent("onreadystatechange", idempotent_fn);

        // A fallback to window.onload, that will always work
        window.attachEvent("onload", idempotent_fn);

        ...
    }
})
```

Patrones de diseño de Comportamiento

Observer Pattern

- En el patrón Observer, un sujeto puede tener una lista de observadores que están interesados en su ciclo de vida.
- Cada vez que el sujeto hace algo interesante, envía una notificación a sus observadores.
- Si un observador ya no está interesado en escuchar el tema, el sujeto puede eliminarlo de su lista.
- Necesitamos tres métodos para describir este patrón:
 - **publish(data)**: Llamado por el sujeto cuando tiene una notificación que hacer. Algunos datos pueden ser pasados por este método.
 - **subscribe(observer)**: Llamado por el sujeto para agregar un observador a su lista de observadores.
 - **unsubscribe(observer)**: Llamado por el sujeto para remover un observador de su lista de observadores.

Observer Pattern (II)

- La mayoría de las librerías JavaScript modernas soportan estos tres métodos como parte de su infraestructura de eventos personalizados.
- Normalmente hay un método *on()* o *attach()*; un método *trigger()* o *fire()*; y un método *off()* o *detach()*

Ejemplo

- Creamos una asociación entre los métodos de eventos jQuery
- Usamos

```
//those prescribed by the Observer Pattern but you don't have to.  
var o = $( {} );  
$.subscribe = o.on.bind(o);  
$.unsubscribe = o.off.bind(o);  
$.publish = o.trigger.bind(o);
```

```
document.on( 'tweetsReceived', function(tweets) {  
    //perform some actions, then fire an event  
    $.publish('tweetsShow', tweets);  
});
```

```
//We can subscribe to this event and then fire our own event.  
$.subscribe( 'tweetsShow', function() {
```

```
    //display the tweets somehow
```

```
    ..
```

```
    //publish an action after they are shown.
```

```
    $.publish('tweetsDisplayed');
```

```
});
```

```
$.subscribe('tweetsDisplayed', function() {
```

```
    ..
```

```
});
```

Mediator Pattern

- El Patrón Mediator promueve el uso de un solo sujeto compartido que maneja la comunicación con múltiples objetos.
- Todos los objetos se comunican entre sí a través del **mediador**.
- Mediante la colocación de mediadores, la comunicación se puede manejar a través de un solo objeto, en lugar de tener múltiples objetos de comunicación entre sí.
- En este sentido, se puede usar un patrón mediador para reemplazar un sistema que implementa el patrón observador.

Ejemplo

- Se puede observar una implementación simplificada por Addy Osmani en este gist
 - <https://gist.github.com/1794823>
- Podemos usarlo de la siguiente manera

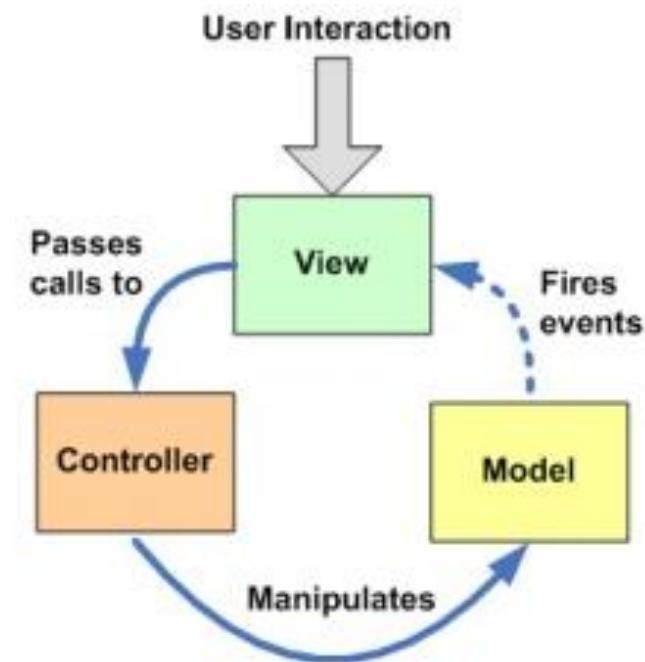
```
$('#album').on('click', function(e) {  
  e.preventDefault();  
  var albumId = $(this).id();  
  mediator.publish("playAlbum", albumId);  
});  
  
var playAlbum = function(id) {  
  ...  
  mediator.publish("albumStartedPlaying", {songList: [...], currentSong: "Without You"});  
};  
  
var logAlbumPlayed = function(id) {  
  //Log the album in the backend  
};  
  
var updateUserInterface = function(album) {  
  //Update UI to reflect what's being played  
};  
  
//Mediator subscriptions  
mediator.subscribe("playAlbum", playAlbum);  
mediator.subscribe("playAlbum", logAlbumPlayed);  
mediator.subscribe("albumStartedPlaying", updateUserInterface);
```

4

Patrón MVC

MVC (Modelo-Vista-Controlador)

- MVC es un patrón de diseño de arquitectura que propone la organización de los componentes mediante la separación de responsabilidades.
- Propone separar los datos de negocio (Modelos) de las interfaces de usuario (Vistas), usando un tercer componente (Controladores) el cual administra la lógica en función de las acciones del usuario.
- Muchísimos frameworks de JS tienen soporte para MVC, poniendo los medios para que los desarrolladores generen estructuras de aplicaciones con el mínimo esfuerzo.
- Entre los más famosos están Backbone, Ember y AngularJS.



Beneficios de MVC

- Uno de los principales beneficios de usarlos es **evitar el código espagueti**. Este término describe un código muy difícil de mantener o leer debido a que no tiene una buena estructura.
- Mejor mantenimiento, cuando haya actualizaciones en la aplicación, sabemos que si el cambio es visual tendremos que modificar la vista y si es de datos, los modelos.
- Es mas fácil escribir tests para la lógica (controladores) al estar separada.
- Eliminación la duplicación de código.
- Se puede trabajar en diferentes componentes simultáneamente.

Modelos

- Los modelos administran los datos de la aplicación. No se preocupan de las interfaces ni capas de presentación sino que representan formas únicas de datos que la aplicación requerirá.
- Cuando un modelo cambia (ej. se actualiza) va notificar a los Observers (ej. vistas) que un cambio ha ocurrido y estos a su vez deben reaccionar.
- Por ejemplo, en una aplicación de fotos, en la galería de fotos el concepto de una foto representa un modelo que posee su información única. Este modelo puede contener atributos como descripción, fuente de la imagen y meta-datos.
- Una foto específica sería guardada en una instancia de un modelo y el modelo será reusable.

Ejemplo Backbone

```
var Photo = Backbone.Model.extend({  
    // Default attributes for the photo  
    defaults: {  
        src: "placeholder.jpg",  
        caption: "A default image",  
        viewed: false  
    },  
    // Ensure that each photo created has an `src`.  
    initialize: function() {  
        this.set( { "src": this.defaults.src } );  
    }  
}
```

Vistas

- Las vistas son la representación visual de los modelos en estado actual.
- Una vista por lo general observa a un modelo y es notificada cuando el modelo cambia, permitiendo a la vista actualizarse.
- En los patrones de diseño las vistas son “**tontas**” ya que su conocimiento de los modelos y controladores de la aplicación son limitados.
- Los usuarios pueden interactuar con las vistas y esto incluye la habilidad de leer y editar modelos.
- Por ejemplo, en la aplicación de la galería de fotos, la vista permitiría al usuario editar los meta-datos de la foto seleccionada mediante un botón ‘edit’.

Templating en las vistas

- Crear grandes bloques de HTML en los JS no es una buena práctica, ya que enturbia el código, es propenso a errores y es poco eficiente.
- Hay soluciones como Handlebars.js y Mustache que permiten definir **templates** en archivos externos, lo cuál es mucho mejor.
- Ejemplo.

```
<li class="photo">
  <h2>{{caption}}</h2>
  
  <div class="meta-data">
    {{metadata}}
  </div>
</li>
```

Controladores

- Los controladores son un intermediario entre los modelos y las vistas. Son responsables de actualizar el modelo cuando el usuario manipula la vista.
- En nuestra aplicación de galería de fotos, el controlador sería el responsable de gestionar los cambios que nuestro usuario haya hecho al editar; *actualizando* el estado del modelo de la foto.
- Los controladores cumplen un rol en MVC: facilitar el **patrón Strategy**. En este patrón, la vista delega eventos al controlador.
- En resumen, el controlador administra, gestiona la lógica y coordina la relación entre los modelos y vistas de nuestra aplicación.



Analizando una implementación MVC

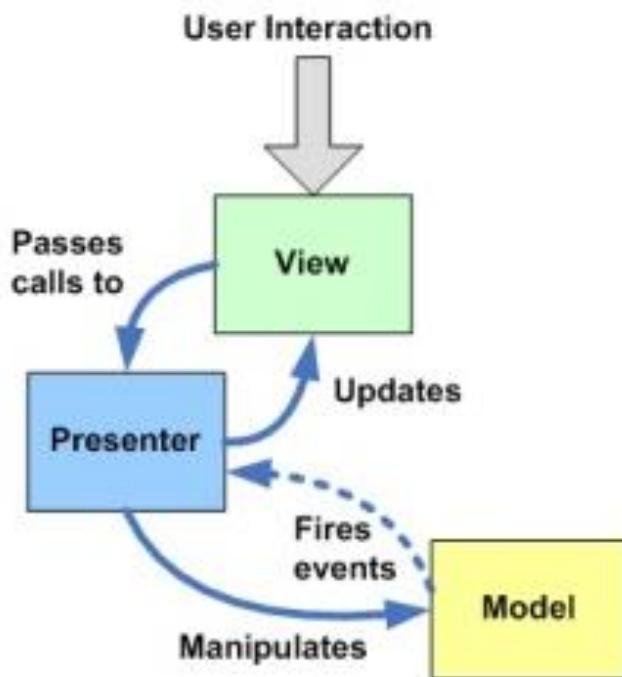
- Accede al ejemplo propuesto por Alex Netkachov en fiddle
 - http://jsfiddle.net/alex_netkachov/ZgBrK/
- Identifica los modelos, la vista y el controlador
- Examina cómo se interrelacionan
- Identifica el proceso que sigue a un evento de usuario

5

Patrón MVP

MVP (Modelo-Vista-Presentador)

- Es un derivado de MVC, y se centra en mejorar la lógica de presentación.
- La entrada comienza con la vista
- Hay una correspondencia uno-a-uno entre la vista y el presentador asociado.
- La vista contiene una referencia al presentador. El presentador también está reaccionando a los eventos que se disparan desde la vista, por lo que es consciente de la vista asociada.
- El presentador actualiza la vista en función de las acciones solicitadas que realiza en el modelo, pero la vista no es consciente del modelo.
- Un framework que implementa MVP es Riot.js



MVP o MVC?

- MVP generalmente es mas usado por aplicación empresariales donde se necesita re-utilizar la presentación y su lógica lo menos posible.
- Las aplicaciones con vistas complejas y mucha interacción con el usuario encontraran que MVC no encaja totalmente y se tendrá que usar muchos controladores.
- En MVP todo esto puede ser encapsulado en un único presentador.
- A final, las diferencias serán mínimas, mientras organicemos todo bien podremos ver todos los beneficios que nos ofrecen.



Analizando una implementación MVC

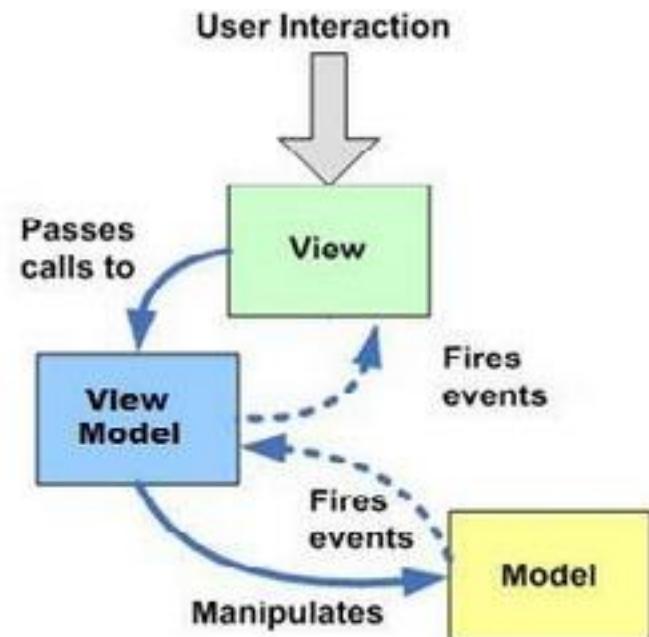
- Accede al ejemplo codepen
 - <https://codepen.io/anon/pen/zBbQYW>
- Identifica los modelos, la vista y el presenter
- Examina cómo se interrelacionan
- Identifica el proceso que sigue a un evento de usuario

6

Patrón MVVM

MVVM (Modelo-Vista-ViewModel)

- Model-View-ViewModel en Inglés.
- Es un patrón de arquitectura basado en MVC y MVP, el cual trata de separar las interfaces de usuario (UI) de la lógica y el comportamiento (behavior) de la aplicación.
- MVVM es útil cuando el estado puede ser mantenido entre las distintas reuquests de UI.
- Un framework que implementa claramente este patrón es Knockout



Modelo y Vista

➤ **Modelo**

- Así como los otros miembros de la familia de MV*, el modelo MVVM representa los datos o información de nuestra aplicación (con la que estará trabajando).

➤ **Vista**

- Como en MVC, la vista es la parte de la aplicación con la interactúa el usuario.
- La vista sólo conoce el ViewModel y, de hecho, obtiene sus datos del ViewModel, no el propio modelo.
- Esto se logra generalmente mediante data-binding (enlaces automáticos entre interface y datos).

VistaModelo

- El viewModel puede ser considerado un controlador especializado, una especie de convertidor de datos: Cambia la información del modelo para ser consumido por la vista y pasa órdenes/eventos de la vista al modelo.
- Por tanto el estado y la lógica de negocio existen en el ViewModel
- El ViewModel se puede considerar como una representación abstracta de nuestro UI, expone los datos necesitados por una vista (de un modelo) y puede ser visto como la fuente a la que las vistas irán a por datos y acciones.
- Las vistas y los ViewModels se comunican usando **data-bindings** y **eventos**

Ventajas y desventajas

Ventajas

- MVVM facilita el desarrollo en paralelo de una UI
- Abstacta la vista y reduce la cantidad de lógica requerida en el código
- El viewModel puede ser testeado fácilmente

Desventajas

- Si tenemos UIs simples, MVVM será excesivo
- Los data-bindings son más complicados de hacer debug.

MVC vs MVP vs MVVM

Cada uno de estos patrones trata de abordar una serie de problemas:

- ¿Dónde y cómo mantenemos el estado en la interfaz de usuario?
- ¿Dónde vive la lógica de negocio en la aplicación y cómo se invoca?
- ¿Cómo podemos mantener la interfaz de usuario sincronizada con los cambios en los datos y entre los elementos de la interfaz de usuario?
- ¿Cómo podemos asegurar una buena separación de los cometidos y crear un código testable sin acoplamiento?

Comparativa

MVC	MVP	MVVM
La entrada se enruta al Controlador.	La entrada se enruta a la vista.	La entrada se enruta a la vista.
El controlador decide qué vista renderizar y construye el modelo que está enlazado a la vista.	El presentador actualiza la vista normalmente en respuesta a los eventos generados por la vista.	La Vista solo conoce el ViewModel.
Un controlador puede optar por representar una de las muchas vistas.	El estado se almacena efectivamente en la vista.	El ViewModel solo conoce el Modelo.
La vista no tiene conocimiento de su controlador.	La lógica de negocio está en el presentador.	La Visat obtiene sus datos del ViewModel, no del Modelo en sí mismo. Se usa data binding.
La lógica de negocio existe en el controlador.		El Estado y la lógica de negocio existen en el ViewModel.
MVC es útil cuando el estado no puede mantenerse entre las solicitudes de entrada del usuario (por ejemplo, a través de HTTP - un protocolo sin estado).		El ViewModel puede ser considerado como una representación abstracta de la UI.
		MVVM es útil cuando el estado puede ser mantenido entre requests de la UI (por ejemplo Silverlight, WPF).



Analizando una implementación MVVM

- Clona el ejemplo propuesto por Alberto Monteiro
 - <https://gist.github.com/AlbertoMonteiro/1370875>
- Identifica los modelos, la vista y la vista-modelo
- Examina cómo se interrelacionan
- Identifica el proceso que sigue a un evento de usuario



Analizando un ejemplo Knockout

- Accede al tutorial de Knockout.js y léelo
 - <http://learn.knockoutjs.com/#/?tutorial=intro>
- Accede al ejemplo en fiddle
 - <http://jsfiddle.net/rniemeyer/LkqTU/>
- Identifica los modelos, la vista y la vista-modelo
- Examina cómo se interrelacionan
- Identifica el proceso que sigue a un evento de usuario



Usando plantillas en BananaTube

- Modifica BananaTube para que use un sistema de plantilla en los puntos donde hay interacción asíncrona



[...]**netmind**

WeKnowIT

Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es



GOBIERNO
DE ESPAÑA

MINISTERIO
DE ENERGÍA, TURISMO
Y AGENDA DIGITAL

red.es



ESTRATEGIA DE
EMPRENDIMIENTO Y
EMPLEO JUVENIL
garantía juvenil



Agenda Digital para España



UNIÓN EUROPEA

Fondo Social Europeo
"El FSE invierte en tu futuro"