JavaScript Design Pattern

DIALLO Hady, BAH Alhassane, DIALLO Amadou, NGALEU Harold Fred

Patterns & Catégories

- Moyen d’accomplir un objectif
- Offre une solution à un problème
- Fondamentalement destiné à être imité.

- Architectural Patterns
- Design Patterns
- Anti-patrons
- Idioms ou coding patterns
- Organizational patterns

Historique

- 1987 - Cunningham et Beck utilisent les idées d'Alexander pour développer un petit langage de patrons pour Smalltalk.
- 1990 - Le Gang des 4 (« Gang of Four » : Gamma, Helm, Johnson and Vlissides) commence à travailler à la compilation d'un catalogue de patrons de conceptions
- 1995 - Le Gang des 4 (GoF) publie le livre *Design Patterns – Elements of Reusable Object-Oriented Software*

Formalisme & Objectifs

- Nom : Vocabulaire de conception
- But : Description du problème à résoudre
- Moyen : Description de la solution
- Conséquences : effets résultants de la mise en oeuvre du patron.

- Concevoir de bons modèles OO
- Simplifier la conception
- Moyen de documentation

Famille de design patterns

- Patterns structuraux:
  - Définissent la façon de composer des objets pour obtenir une structure plus large.
  - Composite, Adaptateur, Façade, Décorateur, Pont
- Patterns Comportement:
  - Définissent la collaboration entre objets.
  - Stratégie, Itérateur, Patron de méthode, Observateur
- Patterns création :
  - Définissent la façon d'instancier des classes/objets
  - Singleton, Fabrique & Fabrique Abstraite

Principaux patterns JS

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

Trois Manières de créer un Objet

```
    //Creation d'Objet vide
var person={}
console.log(typeof(person)) //Object
// ou

var person=Object.create(Object.prototype)
console.log(typeof(person))  //Object

// ou
var person=new objet()
console.log(typeof(person))  //Object
```

```
// Set properties
person.name = "toto";

// Get properties
var name = person.name;

console.log(name);// titi


// Object.defineProperty

// Set properties
Object.defineProperty( person, "Prenom", {
    value: "titi"});
var prenom= person.prenom;
console.log(prenom); //titi
```

Constructor Pattern :Exemple

Déclaration

```
function Animal(name) {
  this.name = name
  this.run = function() {alert("running "+this.name)  }
}
var animal = new Animal('Foxie');
animal.run();
```

Héritage

```
function Rabbit(name,prenom) {
Animal.apply(this, name); //héritage
this.bounce = function() {alert("bouncing "+this.prenom)}
  }
rabbit = new Rabbit("Rab","PRab");
rabbit.bounce() // méthode de la fille
rabbit.run()    // méthode de la mére
```

Constructor Pattern

Polymorphisme

```
function Rabbit(name) {
  Animal.apply(this, arguments)
  var parentRun = this.run  // Methode de la clesse mére
  this.run = function() {
    alert("bouncing "+this.name)
    parentRun.apply(this)  // appel de la méthode le classe mére
  }
}
rabbit = new Rabbit("Rab")
rabbit.run()
```

Prototype Pattern

Ajout externe de Méthode grâce à Prototype

```
function Animal(name) {
    this.name=name;
}
animal = new Animal("foxies");
animal.run();// Undefined
animal.MAJ();//Undefined
Animal.prototype.run= function() {alert("bouncing "+this.name)};
Animal.prototype.MAJ= function() {alert(this.name.toUpperCase())};
animal.run();// bouncing foxies
animal.MAJ();//FOXIES
```

Prototype Pattern

Prototype Pattern

```
function Animal(name) {
    this.name=name;
}
Animal.prototype = {
        run : function() {alert("bouncing "+this.name)},
        MAJ : function() {alert(this.name.toUpperCase())}
}
animal = new Animal("foxies");
animal1.run();// bouncing foxies
animal.MAJ();//FOXIES
```

Pattern Commande

- Encapsulation de l’invocation de méthodes
- Paramétrage d’un objet
- Réversibilité des opérations
- La création et l'ajout de nouvelles commandes ne posent aucun problème.

Command Pattern

```
var Light = (function () {
    // private variables and functions

    // constructor
    var Light = function () {
        this.bOn = false;
    };
    // prototype
    Light.prototype = {
        constructor: Light,
        on: function () {
            this.bOn = true;
            console.log("Light is on");
        },
        off: function () {
            this.bOn = false;
            console.log("Light is off");
        }
    };
    // return Light
    return Light;
})();
```

```
var Command = (function () {
    // private variables and functions

    // constructor
    var Command = function () {
    };
    // prototype
    Command.prototype = {
        constructor: Command,
        execute: function () {
            throw new Error("This method must be overwritten!");
        },
        undo: function () {
            throw new Error("This method must be overwritten!");
        }
    };
    // return Command
    return Command;
})();
```

Command Pattern

```
var LightOnCommand = (function () {
    // private variables and functions

    // constructor
    var LightOnCommand = function (oLight) {
        this.oLight = oLight;
    };
    // prototype
    LightOnCommand.prototype = new Command();
    LightOnCommand.prototype = {
        constructor: LightOnCommand,
        execute: function () {
            this.oLight.on();
        },
        undo: function () {
            this.oLight.off();
        }
    };
    // return LightOnCommand
    return LightOnCommand;
})();
```

```
var SimpleRemoteControl = (function () {
    // private variables and functions

    // constructor
    var SimpleRemoteControl = function () {
        this.oCommand = null;
    };
    // prototype
    SimpleRemoteControl.prototype = {
        constructor: SimpleRemoteControl,
        setCommand: function (oCommand) {
            this.oCommand = oCommand;
        },
        buttonWasPressed: function () {
            this.oCommand.execute();
        },
        buttonUndoWasPressed: function () {
            this.oCommand.undo();
        }
    };
    // return LightOnCommand
    return SimpleRemoteControl;
})();
```

Pattern Singleton

```
var mySingleton = (function () {

    function Singleton(  )  {
    this.X = 4;
    this.Y = 5;
   }
   var instance;
  var result  = {
      getInstance:  function(  ) {
      if( instance  ===  undefined )  {
        instance = new Singleton(  );
      }
      return  instance;
    }
  };

  return  result;

})();
```

Module Pattern

```
var module = (function () {

  // private attribut
  var tab = [];

  function PrivateMethod() {
    console.log("private methode");
  }
  // retour de l'objet expose au public
  return {
    // Add values
    addValues: function( values ) {
      tab.push(values);
    },

    // length of tab
    getLength: function () {
      return tab.length;
    },

  };

})();
```

Revealing Module Pattern

```
var revealingModule = (function () {
 // private attribut
  var tab = [];
  function PrivateMethod() {
    console.log("private methode");
  }

  function addValues ( values ) {
      tab.push(values);
    }
    // length of tab
   function getLength () {
      return tab.length;
    }
  // Reveal public pointers to
  // private functions and properties
  return {
    add :     addValues,
    nbElement : getLength

  };

})();
```

Import Module Pattern

```
// import module
var moduleImport = (function ( jQ, _ ) {

    function privateMethod(){
        jQ(".container").html("module");

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

// Pull in jQuery and Underscore
})( jQuery, _ );

myModule.publicMethod();
```

Export Module Pattern

```
var moduleExport = (function () {

  // Module object
  var module = {},
  // private variable
    variable = "private variable";

  function privateMethod() {
    console.log( "private method" );
  }

  module.modProperty = "public property";
  module.publicMethod = function () {
    console.log( variable );
  };

  return module;

})();
```

AMD & Common JS

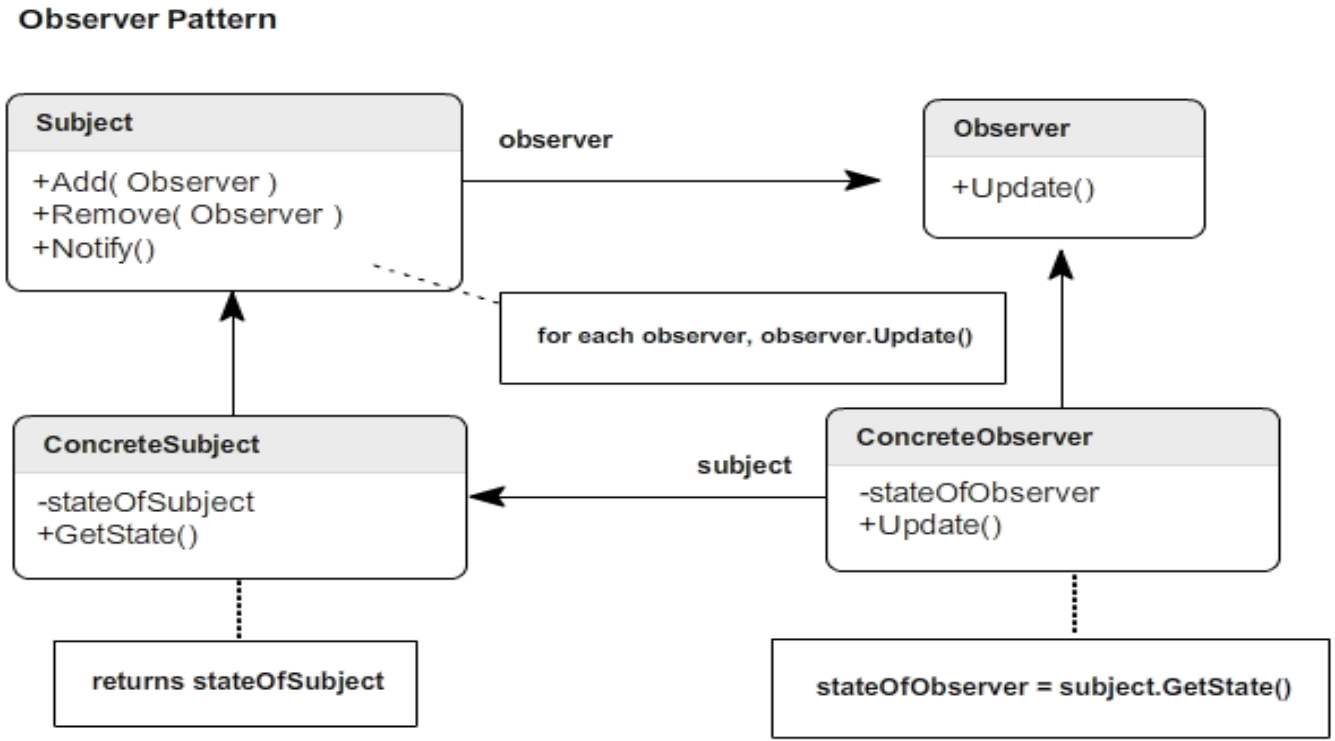
```
define(function (require) {
    var dep1 = require('dependence1'),
    var dep2 = require('dependence2');

    return function () {};
});
```

```
// lib : contenu exporter au chargement du module
var lib = require( "package/lib" );

function myFunction(){
    lib.log( "my Fuction!" );
}

// exporter myFunction au autre module
exports.myFunction = myFunction;
```

Observer Pattern

- Définit une dépendance de un-à-plusieurs entre les objets.
- Le sujet peut avoir une liste d’observateurs qui s'intéressent a son cycle de vie.
- A chaque fois que le sujet change d'état, il envoie une notification à ses observateurs.
- Lorsqu’un observateur n’est plus intéressé à écouter le sujet, le sujet peut le retirer de sa liste.

Pattern Observer



Observer Pattern :Observers

```
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.removeAt = function( index ){ this.observerList.splice( index, 1 );};
```

Observer Pattern : Subject

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

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

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

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

Observer Pattern : Observer

```
// The Observer
function Observer(){
  this.update = function(){
    // ...
  };
}
```

Observer & Publish/Subscribe

- Le pattern Observer est souvent très utilisé en Javascript.
- Le pattern Publish/Subscribe est la variante la plus couramment utilisée pour la mise en oeuvre du pattern Observer.
- Similaires, mais il existe une différence.

Difference : Observer & Pub/Subs

- Pattern Observer:

  Exige que l'observer qui souhaite recevoir des notifications de sujet doit souscrire cet intérêt pour l'objet déclenchant l'événement (sujet).

- Pattern Publish/Subscribe:

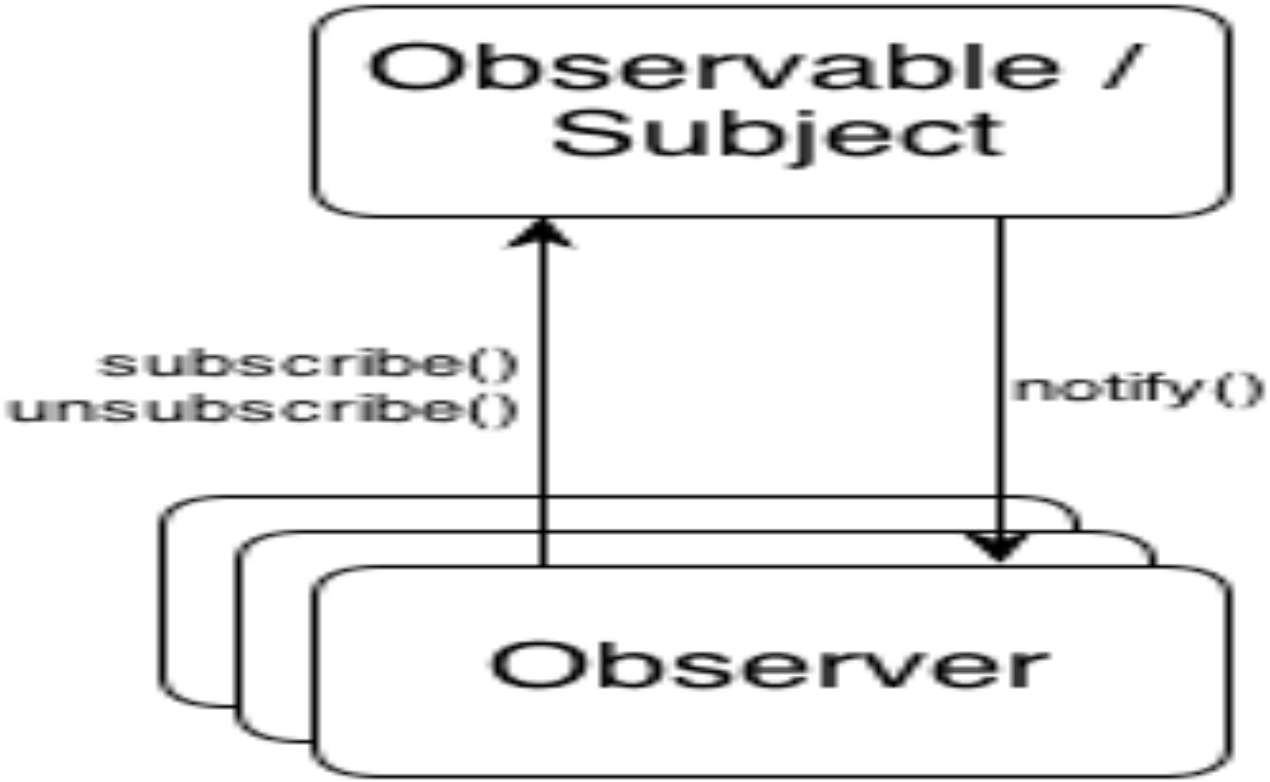
  Utilise cependant un canal sujet/évènement qui se trouve entre les objets qui souhaite recevoir des notifications(subscribes) et l’objet déclenchant

Difference : Observer & Pub/Subs

l’évènement(publisher).

Ce système d’évènement permet de définir le code des évènements spécifiques à l’application qui peut passer des arguments personnalisés contenant des valeurs requises par le subscribe.

Publish/Subscribe Pattern



Publish/Subscribe Pattern

- publish(data)

  Le sujet(Observable) notifie les objets(Observers).

  Des données peuvent être transmisent.

- Subscribe(observer)

  Le sujet(Observable) ajoute “observer” dans sa liste.

- Unsubscribe(observer)

  Le sujet(Observable) supprime “observer” dans sa liste

Publish/Subscribe Pattern

- Deux méthodes de base pour les observers pour obtenir des informations sur du sujet: push et pull.
  - Push : A chaque changement d'état du sujet, il notifie automatiquement tous ses observers.
  - Pull : L'observer vérifie avec l'observable(sujet) s'il y' a changement d'état du sujet à chaque qu'il estime cela nécessaire.

Publish/Subscribe Pattern : Push

```
var Observable = function() {
    this.subscribers = [];
}

Observable.prototype = {
    subscribe: function(callback) {
        this.subscribers.push(callback);
    },
    unsubscribe: function(callback) {
        var i = 0,
        len = this.subscribers.length;
        for (; i < len; i++) {
            if (this.subscribers[i] === callback) {
                this.subscribers.splice(i, 1);
                return;
            }
        }
    },
    publish: function(data) {
        var i = 0,
        len = this.subscribers.length;
        for (; i < len; i++) {
            this.subscribers[i](data);
        }
    }
};

var Observer = function (data) {
  console.log(data);
}

observable = new Observable();
observable.subscribe(Observer);
observable.publish('We published!');
```

Publish/Subscribe Pattern : Pull

```
Observable = function() {
    this.status = "constructed";
}
Observable.prototype.getStatus = function() {
 return this.status;
}

Observer = function() { this.subscriptions = [];}
Observer.prototype = {
    subscribeTo: function(observable) {
        this.subscriptions.push(observable);
    },
    unsubscribeFrom: function(observable) {
        var i = 0,
            len = this.subscriptions.length;

        for (; i < len; i++) {
            if (this.subscriptions[i] === observable) {
                this.subscriptions.splice(i, 1);
                return;
            }
        }
    }
    doSomethingIfOk: function() {
        var i = 0;
            len = this.subscriptions.length;

        for (; i < len; i++) {
            if (this.subscriptions[i].getStatus() === "ok") {

            }
        }
    }
}

var observer = new Observer(),
    observable = new Observable();
observer.subscribeTo(observable);

observer.doSomethingIfOk();
observable.status = "ok";
observer.doSomethingIfOk();
```

Conclusion

- Avantages :
  - Solution réutilisable, robuste et éprouvée
- Inconvénient :
  - Abstraits.