Programación por prototipos

Junio 2017, semana 2

Patrones de diseño

Contexto

Se necesita tener una dependencia de uno a muchos entre objetos de manera tal que: cuando un objeto cambia de estado, todas sus dependencias sean notificadas de forma automática.

→ Manteniendo **bajo acoplamiento**

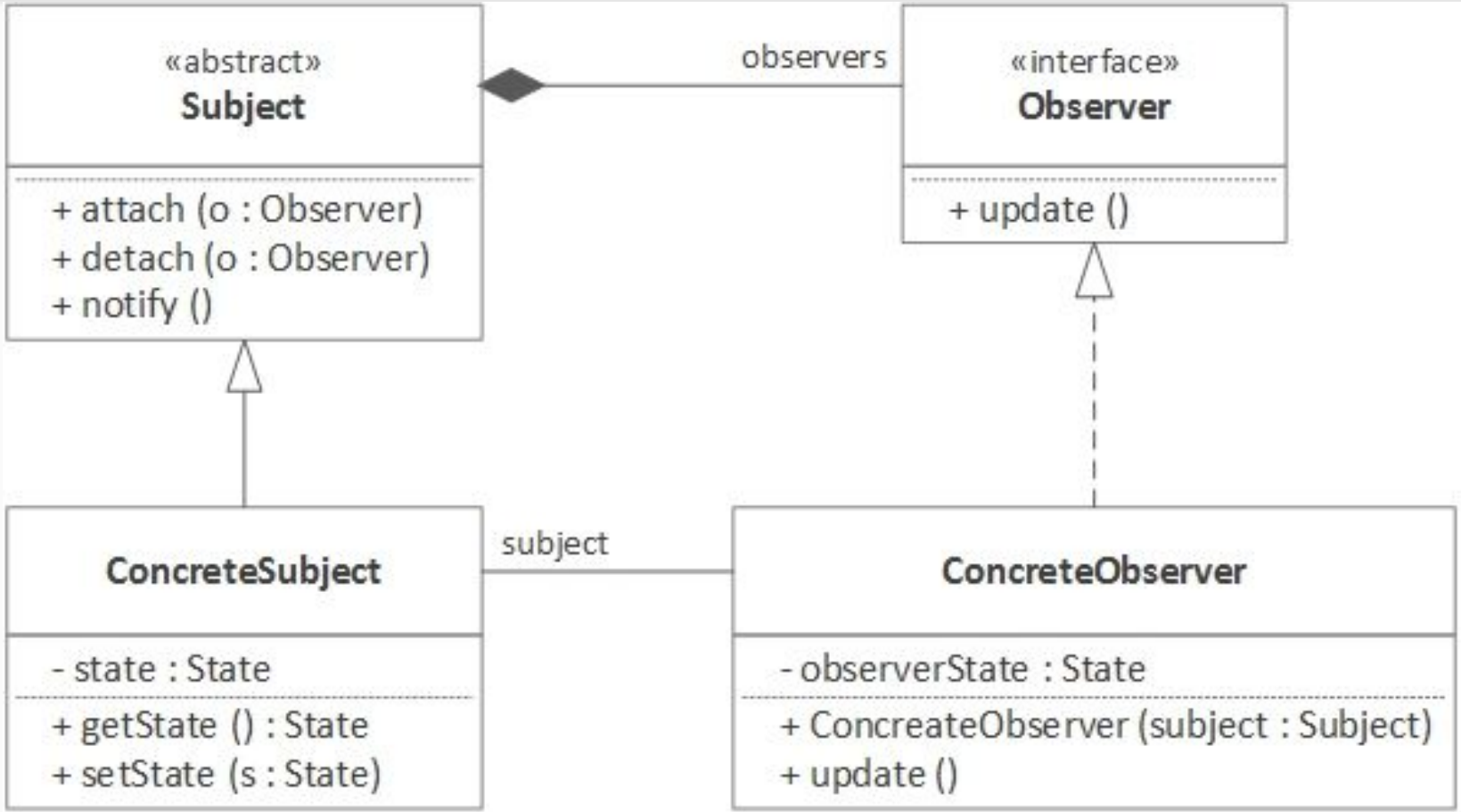
Observer

Una solución es tener dos interfaces: **‘Observador’** y **‘Observable’**.

El **Observable** permite que objetos de la interfaz Observador, se ‘suscriban’ a sus novedades. Cada vez que cambia su estado, notifica a sus observadores.

El **Observador** es quien se suscribe al Observable y recibe las notificaciones. Luego actúa en consecuencia.

Observer con clases

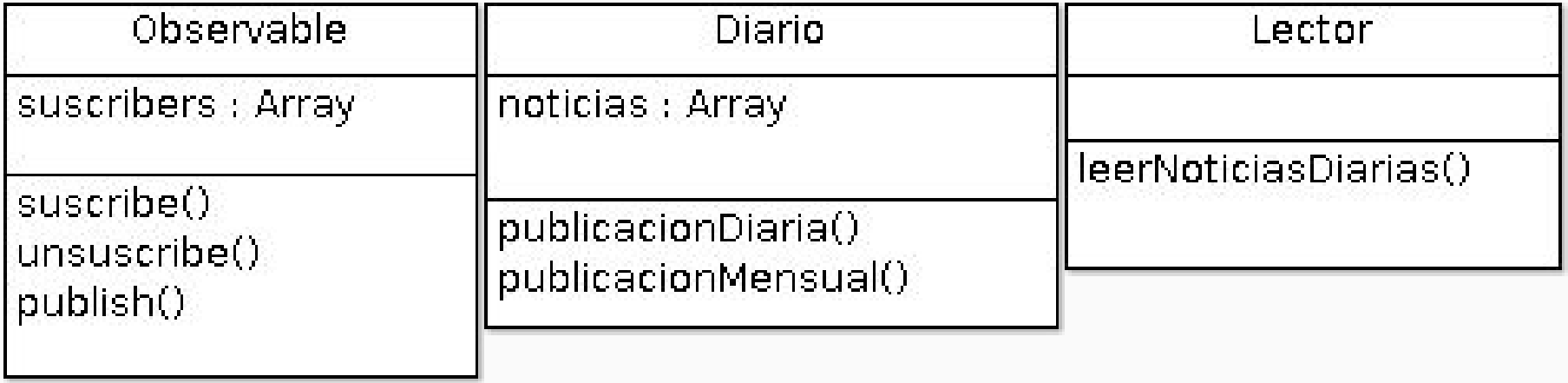


Observer con prototipos

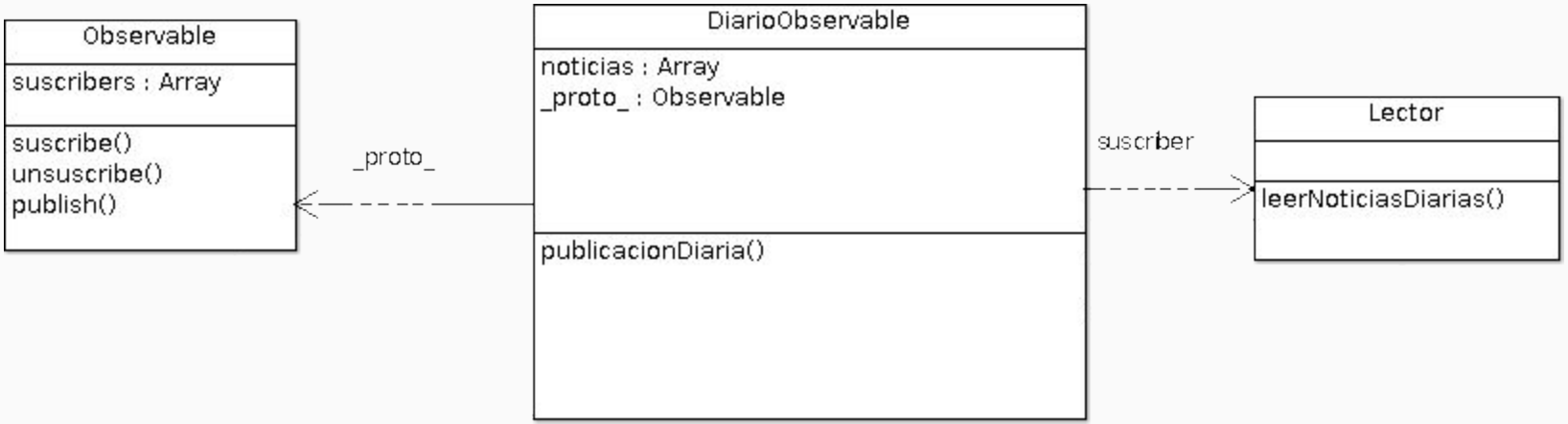
Objeto prototipo de observable

Objeto observable

Objeto observador



Observer con prototipos



Observer con prototipos - ejemplo javascript

```
function Observer() {};
Observer.prototype.subscribers =[];
Observer.prototype.subscribe = function (fn) {
    this.subscribers.push(fn);
};

Observer.prototype.unsubscribe = function (fn) {
    this.visitSubscribers('unsubscribe', fn);
};

Observer.prototype.publish = function (publication) {
    this.visitSubscribers('publish', publication);
};
```

Observer con prototipos - ejemplo javascript

```
Observer.prototype.visitSubscribers = function (action, arg) {
    var i,
    max = this.subscribers.length;
    for (i = 0; i < max; i += 1) {
        if (action === 'publish') {
            this.subscribers[i](arg);
        } else if (subscribers[i] === arg) {
            this.subscribers.splice(i, 1);
        }
    }
}
```

Observer con prototipos - ejemplo javascript

```
function extend ( Child , Parent ) {
    var F = function() {};
    F.prototype = Parent.prototype;
    Child.prototype = new F();
    Child.prototype.constructor = Child;
    Child.uber = Parent.prototype;
}
-----------------------------------------------------------------------
function Paper() {};
extend(Paper, Observer);
Paper.prototype.daily= function () {
    this.publish("big news today");
}
function Person() {};
Person.prototype.drinkCoffee = function (lecture) {
    console.log('Just read ' + lecture);
}
```

Observer con prototipos - ejemplo javascript

```
var paper = new Paper();
var joe = new Person();
paper.subscribe(joe.drinkCoffee);
paper.daily();
```

Contexto

Se necesita asegurar que existe una única instancia de una clase y tener un único punto de acceso a la misma.

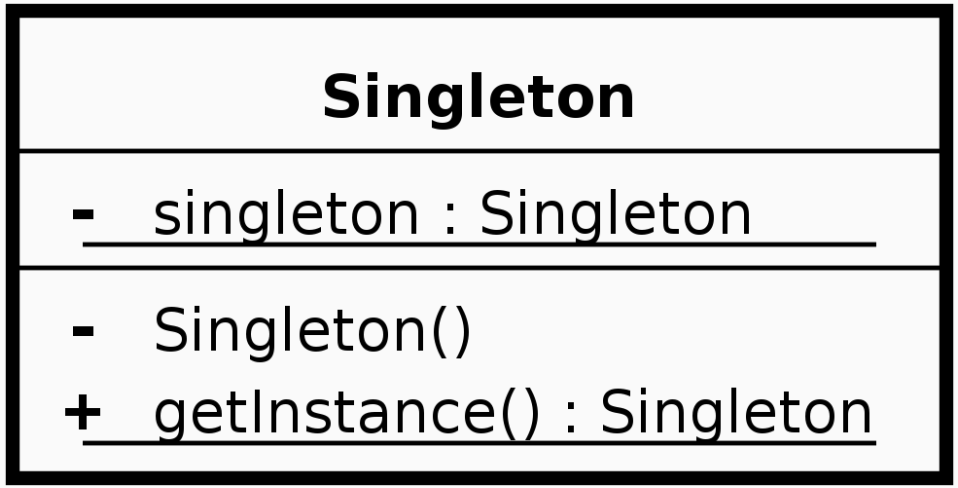
SIngleton

Definir un punto de acceso al recurso que asegure que será creado una única vez.

→ Si es con clases, una clase que se encargue de crear una instancia y controle el acceso a la misma..

→ Si es con prototipos, una función constructora que asegure una única creación..

Singleton con clases



Singleton con prototipos

Se quiere definir la función Universe que siempre devuelva la misma instancia de universe. Deberá responder de esta manera:

```
Universe.prototype.nothing = true;
var uni= new Universe();
Universe.prototype.everything = true;
var uni2 = new Universe();
uni.nothing; // true
uni2.nothing; // true
uni.everything; // true
uni2.everything; // true
uni.constructor.name; // "Universe"
uni.constructor === Universe; // true
```

Singleton con prototipos

```
function Universe() {
     // the cached instance
     var instance = this;
     // proceed as normal
     this.start_time = 0;
     this.bang = "Big";
     // rewrite the constructor
     Universe = function () {
          return instance;
     }
};
```

Singleton con prototipos

```
function Universe() {
     // the cached instance
     var instance = this;
     // proceed as normal
     this.start_time = 0;
     this.bang = "Big";
     // rewrite the constructor
     Universe = function () {
          return instance;
     }
};
```

```
Universe.prototype.nothing = true // true
var uni1 = new Universe();
Universe.prototype.everything = true
var uni2 = new Universe(); // undefined
uni1.nothing //true
uni2.nothing //true
uni1.everything // undefined
uni2.everything // undefined
uni1.constructor.name // "Universe"
uni1.constructor === Universe //false
```

Singleton con prototipos

```
function Universe() {
      // the cached instance
      var instance;
      // rewrite the constructor
      Universe = function Universe() {
            return instance;
      };
      // carry over the prototype properties
      Universe.prototype = this;
      // the instance
      instance = new Universe();
      // reset the constructor pointer
      instance.constructor = Universe;
      // all the functionality
      instance.start_time = 0;
      instance.bang = "Big";
      return instance;
}
```

- La función Universe debe redefinirse al ser invocada por primera vez.
- Se define como ‘this’ al prototipo para que sea el mismo para todas las instancias
- Si no retornara la instancia, automáticamente se retorna ‘this’ cuando se invoca con ‘new’

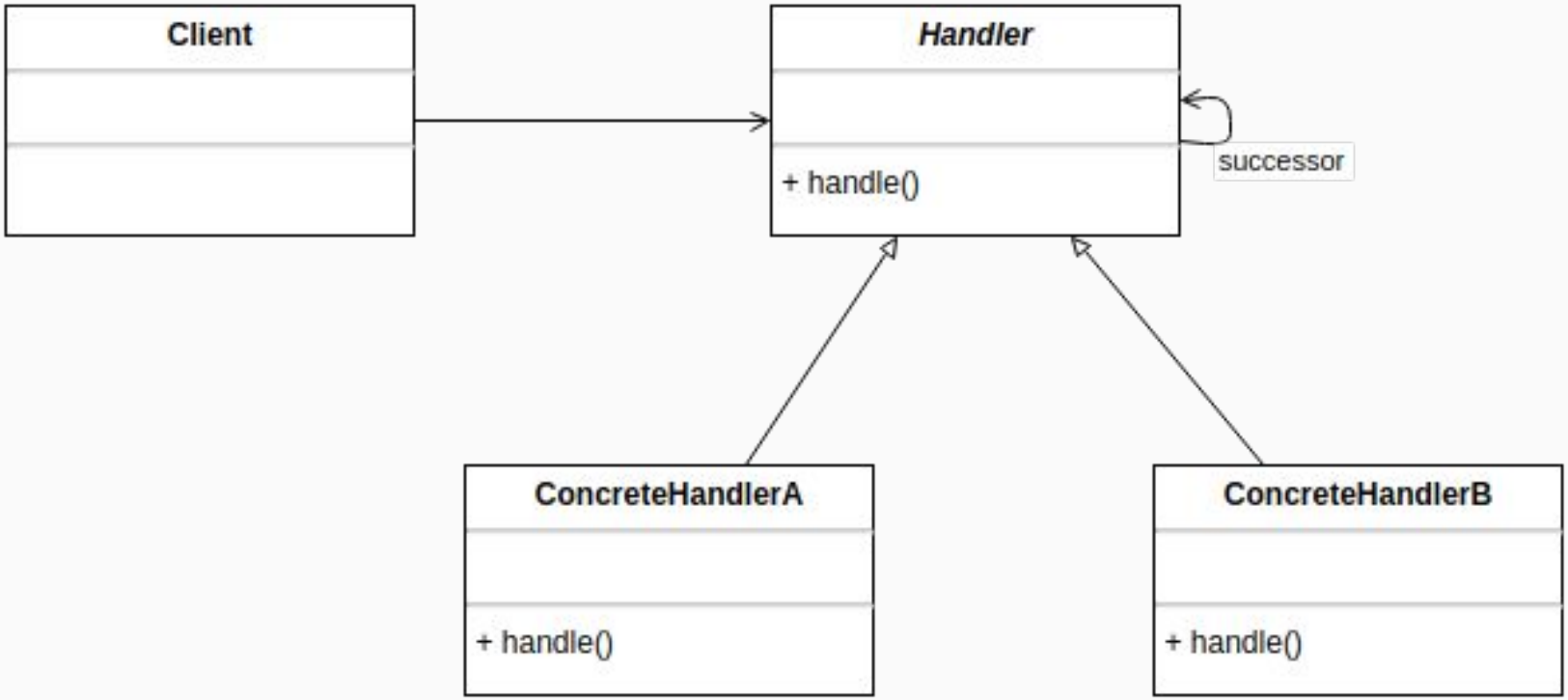
Singleton con prototipos

```
Universe.prototype.nothing = true // true
var uni1 = new Universe(); // undefined
Universe.prototype.everything = true
var uni2 = new Universe(); // undefined
uni1.nothing //true
uni2.nothing //true
uni1.everything // true
uni2.everything // true
uni1.constructor.name // "Universe"
uni1.constructor === Universe //true
```

Ejercicio

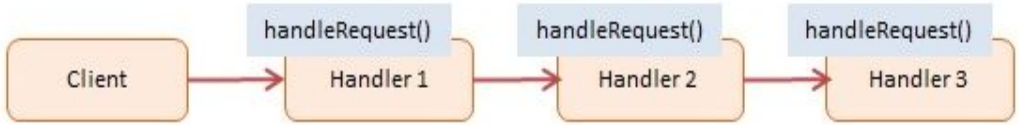
Se desea implementar la lógica de un cajero tipo dispensador de dinero, en montos de 100, 50, 20, 10, 5, 1. Que devuelva billetes tomando los de mayor valor primero, minimizando la cantidad de billetes entregados.

Chain of Responsibility con Clases



Chain of Responsibility

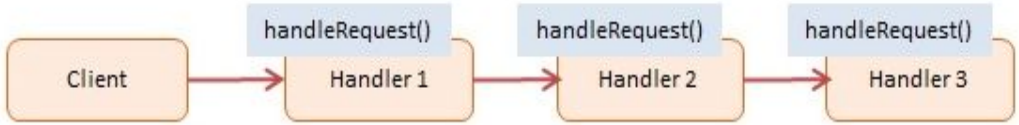
```
var ATM = function() {
     // Create the stacks of money
     var stack100 = new MoneyStack(100);
     var stack50 = new MoneyStack(50);
     var stack20 = new MoneyStack(20);
     var stack10 = new MoneyStack(10);
     var stack5 = new MoneyStack(5);
     var stack1 = new MoneyStack(1);

     // Set the hierarchy for the stacks
     stack100.setNextStack(stack50);
     stack50.setNextStack(stack20);
     stack20.setNextStack(stack10);
     stack10.setNextStack(stack5);
     stack5.setNextStack(stack1);
     // Set the top stack as a property
     this.moneyStacks = stack100;
}
```



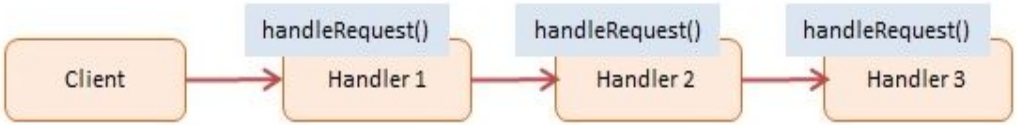

```
ATM.prototype.withdraw = function(request) {
   request.result = [];
   this.moneyStacks.withdraw(request);
   return request.result;
}

var atm = new ATM();
var request = { amount: 186 };
var bills = atm.withdraw(request);
```

Chain of Responsibility

```
var MoneyStack = function(billSize) {
     this.billSize = billSize;
     this.next = null;

     this.withdraw = function(request) {
          var numOfBills = Math.floor(request.amount / this.billSize);
          if (numOfBills > 0) {
               request.amount = request.amount - (this.billSize * numOfBills);
               _ejectMoney(numOfBills, this.billSize, request.result);
          }
          if (request.amount > 0 && this.next!=null)
               this.next.withdraw(request);
     }

     // set the stack that comes next in the chain
     this.setNextStack = function(stack) {
               this.next = stack;
     }
}
```



Chain of Responsibility




```
// private method that ejects the money`
var _ejectMoney = function(numOfBills, billSize, result) {
     result[billSize] = numOfBills;
}
```

Bibliografía

- Java Script Patterns, Stoyan Stefanov, O'Reilly, 2010
- Pro JavaScript Design Patterns, Ross Harmes and Dustin Diaz, Apress, 2008 3
- JavaScript: The Good Parts, Douglas Crockford, O'Reilly, 2008.
- Object-Oriented JavaScript, Stoyan Stefanov, Packt Publishing, 2008.
- Test-Driven JavaScript Development, Christian Johansen, Adisson Wesley, 2011
- Node, https://nodejs.org/dist/latest-v8.x/docs/api/
- ECMA Script 262 https://www.ecma-international.org/publications/standards/Ecma-262.htm
- Mocha Js, https://mochajs.org/
- Design Patterns: Elements of Reusable Object-Oriented Software

Test Unitarios

Unit Test Frameworks:

Karma, Protractor, Buster, TestSwarm,

Jasmine, QUnit, Sinon, Intern,

Mocha, Yolpo, Ava, etc

Test Unitarios

```
var assert = require('assert');


describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      assert.equal(-1, [1,2,3].indexOf(4));
    });
  });
});
```

Test Unitarios

Assertions

- should.js - BDD style shown throughout these docs
- expect.js - expect() style assertions
- chai - expect(), assert() and should-style assertions
- better-assert - C-style self-documenting assert()
- unexpected - “the extensible BDD assertion toolkit”

Test Unitarios

```
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      [1,2,3].indexOf(5).should.equal(-1);
      [1,2,3].indexOf(0).should.equal(-1);
    });
  });
});
```

ASYNCHRONOUS CODE

```
describe('User', function() {
  describe('#save()', function() {
    it('should save without error', function(done) {
      var user = new User('Luna');
      user.save(function(err) {
        if (err) done(err);
        else done();
      });
    });
  });
});
```

HOOKS

```
describe('hooks', function() {

  before(function() {
    // runs before all tests in this block
  });

  after(function() {
    // runs after all tests in this block
  });

  beforeEach(function() {
    // runs before each test in this block
  });

  afterEach(function() {
    // runs after each test in this block
  });

  // test cases
});
```

Recursos

- Java Script Patterns, Stoyan Stefanov, O'Reilly, 2010
- Pro JavaScript Design Patterns, Ross Harmes and Dustin Diaz, Apress, 2008 3
- JavaScript: The Good Parts, Douglas Crockford, O'Reilly, 2008.
- Object-Oriented JavaScript, Stoyan Stefanov, Packt Publishing, 2008.
- Test-Driven JavaScript Development, Christian Johansen, Adisson Wesley, 2011
- Node, https://nodejs.org/dist/latest-v8.x/docs/api/
- ECMA Script 262 https://www.ecma-international.org/publications/standards/Ecma-262.htm
- Mocha Js, https://mochajs.org/
- Design Patterns: Elements of Reusable Object-Oriented Software