# Design Patterns In JavaScript

July 15, 2015

## Agenda

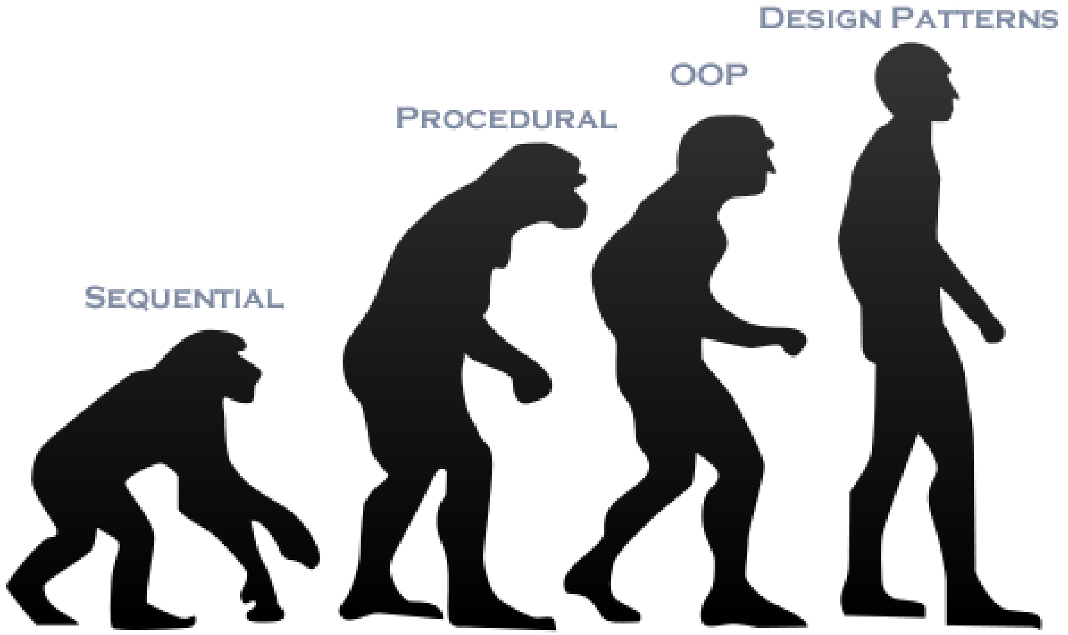
- Creational patterns: factory, factory method
- Structural patterns: decorator, module
- Behavioral patterns: mediator, pub/sub
- MV* patterns in JavaScript
- 2-way binding as design pattern

## What does “design pattern” mean?

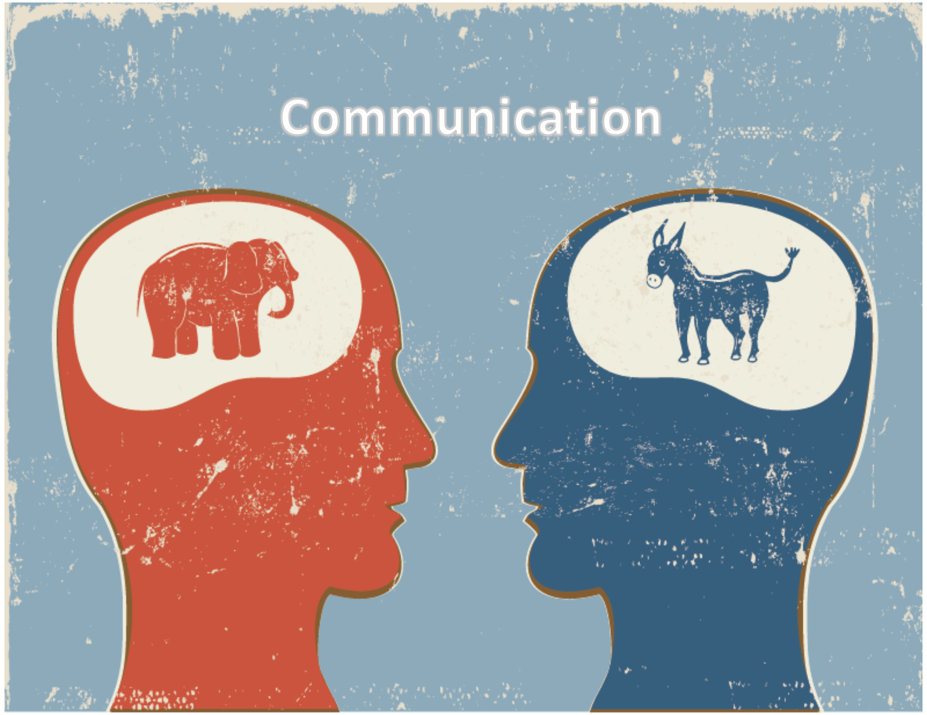
architectural **solution**

to the **frequently occurring problem**

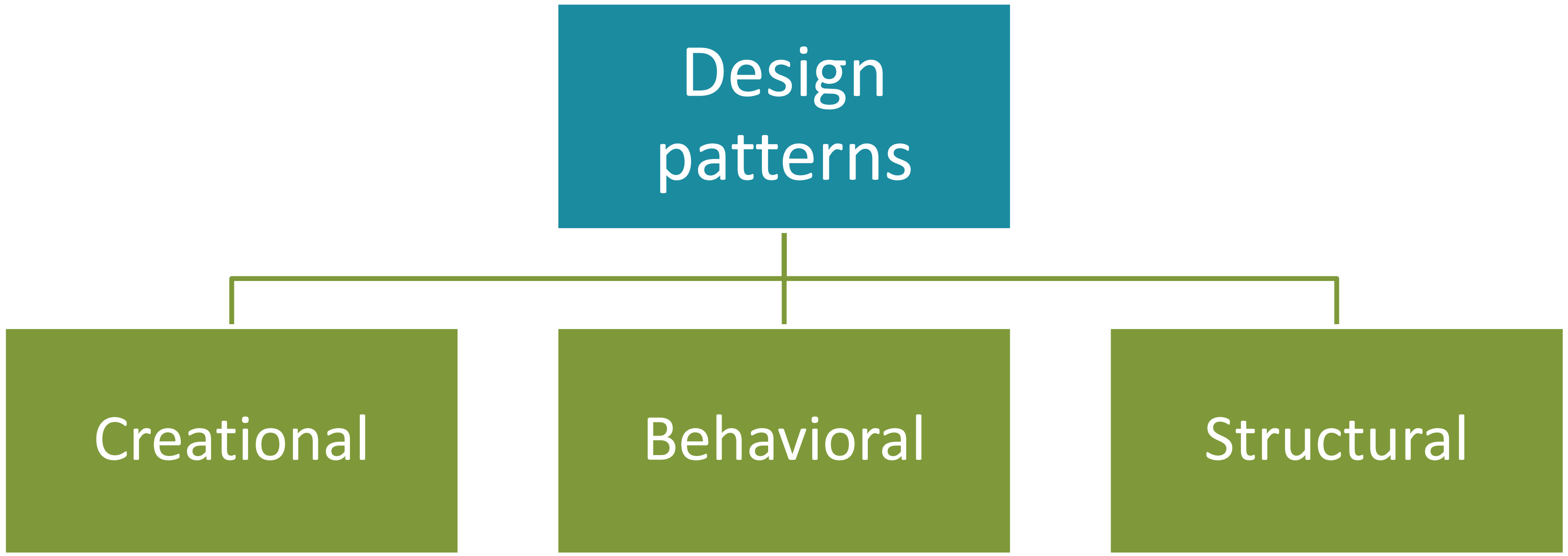
## Programmer Evolution



## Why do we need patterns?



## Classification

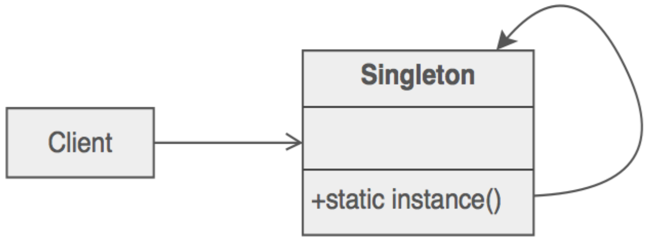


## CREATIONAL PATTERNS

**DEAL WITH OBJECT CREATION MECHANISMS**

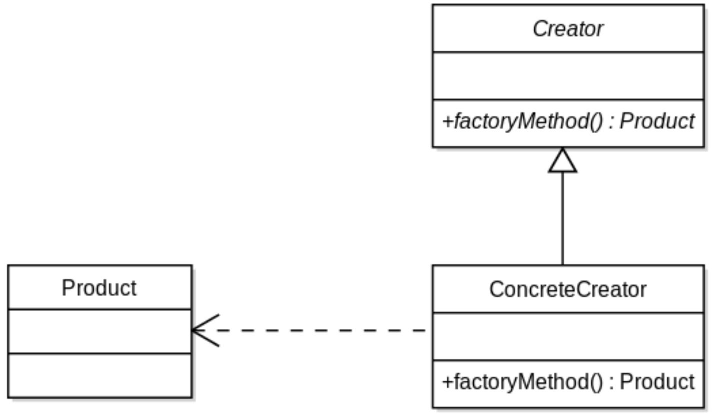
## Singleton

Ensure a class has only one instance, and provide a global point of access to it



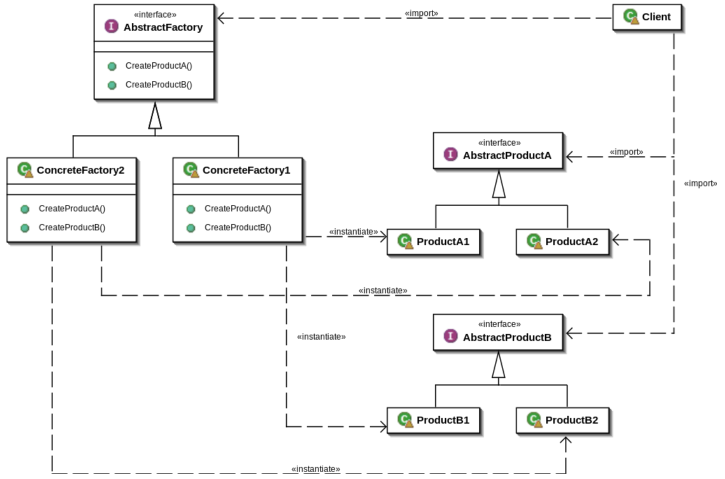
## Factory Method

is a creational pattern which uses factory methods to deal with the problem of creating objects without specifying the exact class of object that will be created



## Abstract Factory

a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes



# Abstract Factory, Factory Method

**Advantages**

Makes complex object creation easy through an interface that can bootstrap this process for you

Great for generating different objects based on the environment

Practical for components that require similar instantiation or methods

Great for decoupling components by bootstrapping the instantiation of a different object to carry out work for particular instances

**Disadvantages**

- Unit testing can be difficult as a direct result of the object creation process being hidden by the factory methods

## STRUCTURAL PATTERNS

**DEAL WITH RELATIONSHIPS BETWEEN MODULES**

## Module

This pattern is used to mimic classes in conventional software engineering and focuses on public and private access to methods & variables

```
( function( window, undefined ) {
    function MyModule() {

        this.myMethod = function () {
            alert( 'my method' );
        };

        this.myOtherMethod = function () {
            alert( 'my other method' );
        };

    }

    window.MyModule = MyModule;

} )( window );

// example usage
var myModule = new MyModule();
myModule.myMethod();
myModule.myOtherMethod();
```

## Revealing Module

This pattern is the same concept as the module pattern in that it focuses on public & private methods. The only difference is that the revealing module pattern was engineered as a way to ensure that all methods and variables are kept private until they are explicitly exposed

```
var MyModule = ( function( window, undefined ) {

    function myMethod() {
        alert( 'my method' );
    }

    function myOtherMethod() {
        _privateMethod();
    }

    function _privateMethod() {
        alert( 'my other method' );
    }

    return {
        myMethod: myMethod,
        myOtherMethod: myOtherMethod
    };
} )( window );

// example usage
var myModule = new MyModule();
myModule.myMethod();
myModule.myOtherMethod();
```

# (Revealing) Module

**Advantages**

Cleaner approach for developers

Supports private data

Less clutter in the global namespace

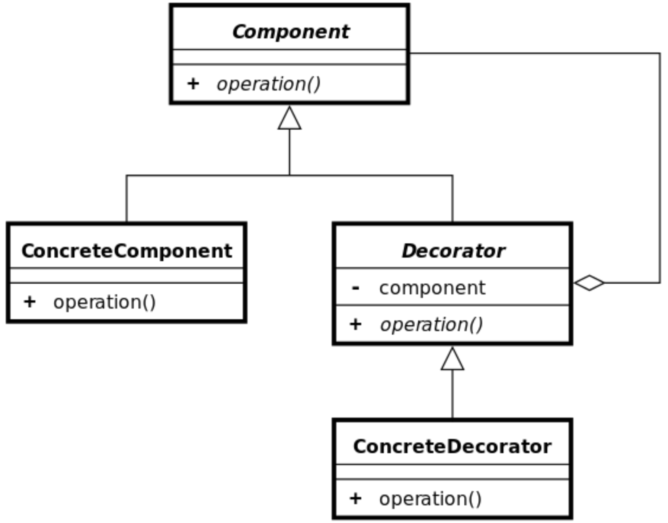
Localization of functions and variables through closures

**Disadvantages**

- Private methods are inaccessible.
- Private methods and functions lose extendibility since they are inaccessible

## Decorator

Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.

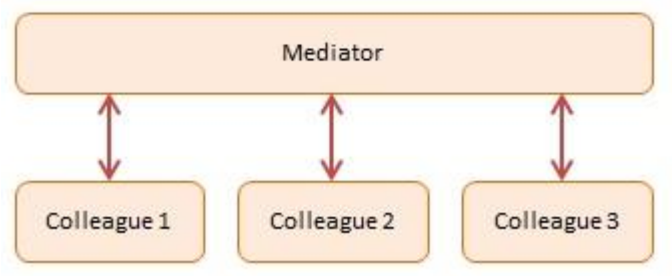


## BEHAVIORAL PATTERNS

**DEAL WITH COMMUNICATION BETWEEN MODULES**

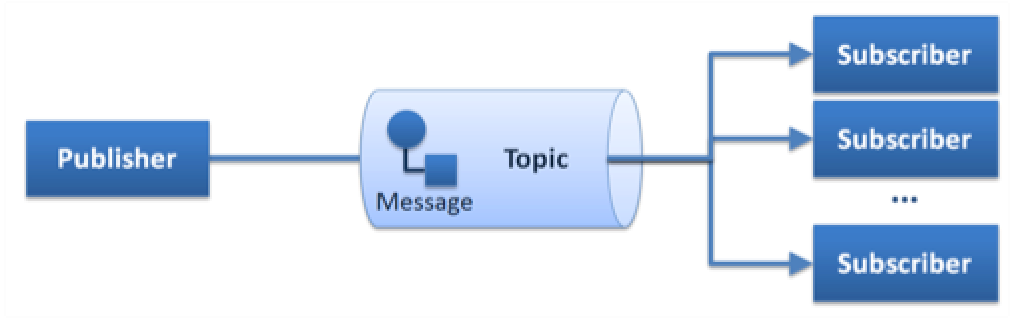
## Mediator

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently



## Publish/Subscribe

Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically



## DEEP DIVE
## INTO DESIGN PATTERNS

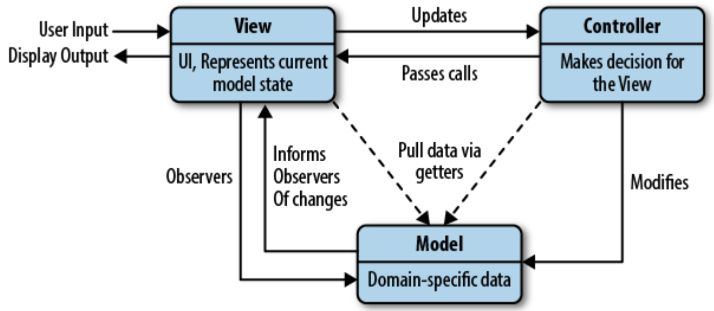
## MV* PATTERNS
## MVC, MVP, MVVM

## MV* patterns

1. MVC – Model-View-Controller
2. MVP – Model-View-Presenter
3. MVVM – Model-View-ViewModel

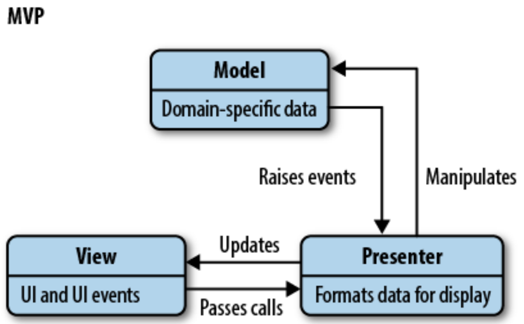
## MVC pattern

is an architectural design pattern that encourages improved application organization through a separation of concerns



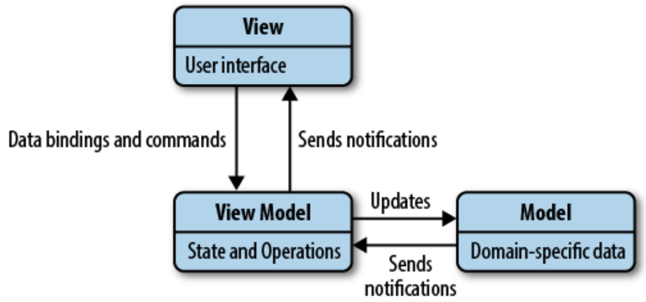
## MVP pattern

is a derivative of the MVC design pattern that focuses on improving presentation logic



## MVVM pattern

is an architectural pattern based on MVC and MVP, which attempts to more clearly separate the development of user interfaces (UI) from that of the business logic and behavior in an application



## MV* Advantages

| | | |
|---|---|---|
| 1 | **Easier overall maintenance** | • When updates need to be made to the application it is very clear whether the changes are data-centric, meaning changes to models and possibly controllers, or merely visual, meaning changes to views<br>• Duplication of low-level model and controller code (i.e., what we may have been using instead) is eliminated across the application. |
| 2 | **Decoupling models and views** | • means that it is significantly more straight-forward to write unit tests for business logic |
| 3 | **Separation of roles** | • Depending on the size of the application and separation of roles, this modularity allows developers responsible for core logic and developers working on the user interfaces to work simultaneously. |

## 2-WAY BINDING

## Imagine a task

User enters some data into a form, presses “submit” button and a page shows loading indicator.
Then (given the input contained some kind of error) invalidated fields get highlighted

## What’s happening behind the scenes

1. Values of fields get stored into a variable
2. Variable gets serialized into JSON and gets sent onto server with an AJAX query
3. DOM gets modified: loading indicator appears
4. As soon as the request finishes we see that status is not 200, then we parse response body
5. DOM gets modified again: loading indicator disappears, invalidated fields get highlighted

## Classical solution

1. A function gets attached to the ‘click’ event of button
2. The function collects fields and puts them into a variable
3. The variables gets serialized into JSON and goes to server
4. We mark “request in process” state using another variable (to not react double clicks mainly)
5. We modify DOM adding indicator
6. As soon as the request finishes we parse response body and get invalidation data
7. We modify DOM adding invalidation information and removing the loading indicator

## Classical solution with 2-way binding

1. A function gets attached to the ‘click’ event of button
2. ~~The function collects fields and puts them into a variable~~
3. The variables gets serialized into JSON and goes to server
4. We mark “request in process” state using another variable (to not react double clicks mainly)
5. ~~We modify DOM adding indicator~~
6. As soon as the request finishes we parse response body and get invalidation data
7. ~~We modify DOM adding invalidation information and removing the loading indicator~~

## Template

```
<% if (entity.errors.name) { %>
<div class="error">
<% } %>
<input name="name" value="<%= entity.name %>">
<% if (entity.errors.name) { %> <%= entity.errors.name %> </div> <% } %>
<% if (entity.loading) { %>
 Sending form...
<% } else {%>
<button>Send!</button>
<% } %>
```

## Problems

1. How to monitor entity modifications recursively
2. How to do not monitor all the modifications for all the variables and constantly redraw the whole DOM of the page.
3. How to update (including the tag itself) two <tr>’s out of 10?
4. How to animate automatic modifications instead of changing DOM instantly?
5. How to implement reflection of input to entity.name?

## Object modifications monitoring

1. Wrap object with setters getters (Ember way)
2. External monitoring (Angular way)

## Page sectioning

1. Declarative templates (Angular way)
2. Manual sectioning (Backbone way)

## Q & A

THANK YOU

**FOR YOUR ATTENTION :)**

## Useful links

https://carldanley.com/javascript-design-patterns/

https://www.safaribooksonline.com/library/view/learning-javascript-design/9781449334840/ch10.html

https://www.dropbox.com/s/iwmkt9f81pkjsjv/Design%20Principles%20and%20Design%20patterns.pdf?dl=0

https://www.dropbox.com/s/0yeernn5vl3ta80/Patterns%20of%20Enterprise%20Application%20Architecture%20-%20Martin%20Fowler.chm?dl=0

https://www.dropbox.com/s/k3byuosimvlelir/Pro%20JavaScript%20Design%20Patterns.pdf?dl=0

https://www.dropbox.com/s/9rgzn2i4r45lj84/%D0%A1%D1%82%D0%BE%D1%8F%D0%BD%20%D0%A1%D1%82%D0%B5%D1%84%D0%B0%D0%BD%D0%BE%D0%B2%20-%20JavaScript.%20%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B%20%28O%27REILLY%29%20-%202011.DjVu.djvu?dl=0

## Contacts

**Author:**

Viktor Pishuk

Lead Software Engineer at “EPAM Systems Ukraine”

**E-mail:** Viktor_Pishuk@epam.com

**Skype:** victor.pishuk