# #. Design Patterns in JavaScript

## #. Creational Design Pattern

### #. Module Pattern

### #. Revealing Module Pattern

### #. Prototype Pattern

### #. Singleton Pattern

## #. Structural Design Pattern

### #. Composite Pattern

The composite pattern is another pattern that you probably have used before without any realization.

*“The composite pattern says that a group of objects can be treated in the same manner as an individual object of the group”.*

So what does this mean? Well, consider this example in jQuery (most JS libraries will have an equivalent to this):

*$('.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.");*

*});*

Most JavaScript libraries provide a consistent API regardless of whether we are dealing with a single DOM element or an array of DOM elements. In the first example, we are able to add the selected class to all the items picked up by the .myList selector, but we can use the same method when dealing with a singular DOM element, #myItem. Similarly, we can attach event handlers using the on() method on multiple nodes, or on a single node through the same API.

By leveraging the Composite pattern, jQuery (and many other libraries) provide us with a simplified API.

The Composite Pattern can sometimes cause problems as well. In a loosely-typed language such as JavaScript, it can often be helpful to know whether we are dealing with a single element or multiple elements. Since the composite pattern uses the same API for both, we can often mistake one for the other and end up with unexpected bugs. Some libaries, such as YUI3, offer two separate methods of getting elements (Y.one() vs Y.all()).

### #. Facade Pattern

Here's another common pattern that we take for granted. In fact, this one is one of my favorites because it's simple, and I've seen it being used all over the place to help with browser inconsistencies. Here's what the Facade pattern is about:

*The Facade Pattern provides the user with a simple interface, while hiding it's underlying complexity.*

The Facade pattern almost always improves usability of a piece of software. Using jQuery as an example again, one of the more popular methods of the library is the ready() method:

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

The ready() method actually implements a facade. If you take a look at the source, here's what you find:

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);

        ...

    }

})

Under the hood, the ready() method is not all that simple. jQuery normalizes the browser inconsistencies to ensure that ready() is fired at the appropriate time. However, as a developer, you are presented with a simple interface.

Most examples of the Facade pattern follow this principle. When implementing one, we usually rely on conditional statements under the hood, but present it as a simple interface to the user. Other methods implementing this pattern include animate() and css(). Can you think of why these would be using a facade pattern?

Facades can be frequently found across the jQuery library and provide developers easy access to implementations for handling DOM manipulation, animation, and of particular interest, cross-browser Ajax.

The following are facades for jQuery’s $.ajax():

$.get( url, data, callback, dataType );

$.post( url, data, callback, dataType );

$.getJSON( url, data, callback );

$.getScript( url, callback );

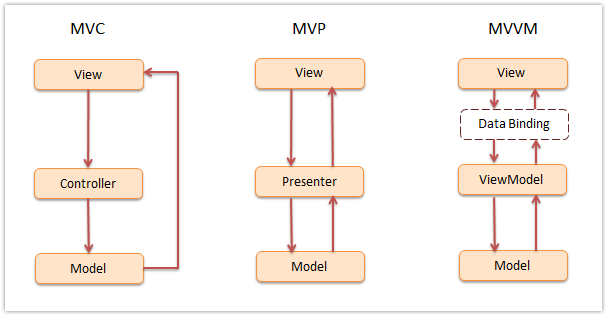
### #. Decorator Pattern

## #. Behavioral Design Pattern

### #. Observer Pattern

## #. Architectural Design Pattern

1. MVC – Model-View-Controller
2. MVP – Model-view-presenter
3. MVVM – Model-View-View-Model



### #. MVC – Model-View-Controller:

### #. MVP – Model-view-presenter

Model-view-presenter (MVP) is a derivative of the MVC design pattern, which focuses on improving presentation logic.

### #. MVVM – Model-View-View-Model