JavaScript design patterns – Part 3: Proxy, observer, and command

This is the final part of my JavaScript Design Patterns series. I've described the singleton, composite, and façade patterns in Part 1, and then in Part 2, explored the adapter, decorator, and factory patterns. This time around I'll try to cram the Proxy, Observer, and Command patterns into your brains.

Proxy

The definition of the word "proxy" is *substitute*, which does a pretty decent job of explaining the job of the proxy pattern. So, what do you substitute and why not just stick with the original?

Why do you need a proxy?

You're essentially creating objects that take the place of other objects. There are several reasons why you might want to create proxy objects:

1. to delay the instantiation of a large object until it is absolutely necessary;
2. to provide access to a remote original object;
3. to control access to the original object.

The virtual proxy

Virtual proxies cover the first case listed above: delaying instantiation of a large object. Pretend you have a massive object, whether that means that it has many functions with long complicated logic or that it just has a lot of data within it. Using a virtual proxy is probably a good idea if there's a decent chance that the object would get instantiated and not used, or if you want to postpone the intensive work required to actually create it until a point where you'll really need it because doing that work immediately might slow things down at a point when you want things to be fast and responsive. As an example, suppose you have an object called `VehicleList`. This will be the original object. This object, when instantiated, contains a massive list of every single vehicle make and model since the Model T. So obviously this would be expensive to create. This code creates a virtual proxy to delay that instantiation.

```
var VehicleListProxy = function() {
    // Variable to hold real VehicleList when it's instatiated
    this.vehicleList = null;
};
VehicleListProxy.prototype = {
    // Whenever a method requires that the real VehicleList is created
    // it should call this function to initialize it (if not already
    // initialized)
    _init: function() {
        if ( !this.vehicleList ) {
            this.vehicleList = new VehicleList();
        }
    },
    getCar: function( ... ) {
        this._init();
        return this.vehicleList.getCar( ... );
    },
    ...
}
```

All this code does is create an object that passes all requests through to the original object, but doesn't instantiate the original object until one of those requests is made. You've created a method that every other method calls to ensure that the original object is instantiated when you need it.

Remote proxies

Remote proxies are essentially the same, except instead of delaying the instantiation of the original object, it already exists in a remote location across the internet. So instead of just passing the requests on through to the original object to which you have direct access via an object property, you need to do an AJAX request. It doesn't matter if there is remote object that follows the same interface on the server. It doesn't even matter if there are any objects at all. Anything can be on the server as long as the AJAX request to that URL will do

what you expect it to do and return the values you expect. Take a look at an example using the `VehicleListProxy` again.

```
var VehicleListProxy = function() {
    this.url = "http://www.welistallcarmodels.com";
};
VehicleListProxy.prototype = {
    getCar: function( ... ) {
        // Skip the rest of the implementation to just show the
        // important stuff
        ajax( this.url + "/getCar/" + args);
        // Package up the data that you got back and return it
    },
    ...
}
```

It's perfectly simple. Models and collections in Backbone.js operate much like proxies, though they have explicit functions to tell them when to deal with the server and when to just keep things local. If you don't know what I'm taking about when I say "models," "collections," or "Backbone.js," then you might find it interesting to read Christophe Coenraets' Backbone.js tutorials here on the Adobe Developer Connection or go through my Backbone.js Video Tutorial series. Or you can just forget I said anything about them.

Access control via proxy

And now the final use case of the proxy: controlling access to the original object. The first thing you have to realize with this type of proxy in JavaScript is that to actually control access to the object, you have to have a closure around it; otherwise, it would just be accessible from the global scope anyway. Then you have to make sure that your proxy is available to whoever needs it. For our example, we'll make sure no one has access until the planned release date. Take a look below to see how it's done.

```
// Close it off in a function
(function() {
// We already know what the VehicleList looks like, so I
// won't rewrite it here
var VehicleList = ...

var VehicleListProxy = function() {
    // Don't initialize the VehicleList yet.
    this.vehicleList = null;
    this.date = new Date();
};
VehicleListProxy.prototype = {
    // this function is called any time any other
    // function gets called in order to initialize
    // the VehicleList only when needed. The VehicleList will
    // not be initialized if it isn't time to yet.
    _initIfTime: function() {
        if ( this._isTime() ) {
            if ( !this.vehicleList ) {
                this.vehicleList = new VehicleList();
            }
            return true;
        }
        return false;
    },

    // Check to see if we've reached the right date yet
    _isTime() {
        return this.date > plannedReleaseDate;
    },

    getCar: function(...) {
        // if _initIfTime returns a falsey value, getCar will
        // return a falsey value, otherwise it will continue
```

```
        // and return the expression on the right side of the
        // && operator
        return this._initIfTime() && this.vehicleList.getCar(...);
    },
    ...
}

// Make the VehicleListProxy publicly available
window.VehicleListProxy = VehicleListProxy;

// you could also do the below statement so people don't even
// know they're using a proxy:
window.VehicleList = VehicleListProxy;

}());
```

Looking at the above code, you can see that every public method will call `this.initIfTime()` to first check if they should have access, and then, if they are allowed access, make sure that the `vehicleList` is instantiated. If `initIfTime()` returns `false`, then the conditional expression in the `return` statement will return `false` and won't run the next expression; but if the method receives `true` back from `initIfTime`, then it executes the next expression and returns its value.

Finally in the last few lines, the proxy is exposed to the rest of the code by attaching it to `window`. As shown, you can do this in a way that the users don't even realize it's a proxy.

If you'd like to read up some more about the proxy pattern, you can read the article on Joe Zim's JavaScript Blog, "JavaScript Design Patterns: Proxy."

Observer

Next up: Observer! The observer pattern (also known as Publish/Subscribe or Pub/Sub for short) is one of the most commonly used patterns in JavaScript. This is because event handling on DOM elements, which is quite common in JavaScript applications of all sizes, is done via the Observer pattern.

The observer pattern explained

The Observer pattern is simple and consists of two entities: an object to observe (known as an observable) and an object that observes the observable (called the observer), both shown in Figure 1. In the case of DOM events, the observer is just a callback function, which is common in JavaScript because it doesn't require the observer object to follow a certain interface. A lot of languages don't have this luxury because functions/methods are not first-class objects and cannot be passed around, but JavaScript is super lucky in this respect.

Figure 1. Structure of the Observer Pattern

Types of observers

There are two methodologies for implementing the observer pattern: push and pull. For the push method, an observer subscribes to the observable object and, when something noteworthy happens to the observable, it reaches out and lets the observer know, which is how DOM events work. In the pull method, observables are added to a list of subscriptions that the observer has. Either periodically or when told to, the observer will check to see if any changes have happened within the observable, and if so, do something in response to that change. This is how a lot of desktop software does its updates. When an application starts up, it checks the server (observable) to see if there are any updates, and if any are present, it will start installing the update.

The push observer

First, I'll walk you through a simple example for building the Observer pattern via the push method. I won't bother showing any examples of the pull method because it's mostly used for communicating with the server, in

which case you just query a server with AJAX; it has limited use in front-end web development.

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

Cbservable.prototype = {
    subscribe: function(callback) {
        // Just add the callback to the subscribers list
        this.subscribers.push(callback);
    },
    unsubscribe: function(callback) {
        var i = C,
            len = this.subscribers.length;

        // Iterate through the array and if the callback is
        // found, remove it from the list of subscribers.
        for (; i < len; i++) {
            if (this.subscribers[i] === callback) {
                this.subscribers.splice(i, 1);
                // Cnce we've found it, we don't need to
                // continue, so just return.
                return;
            }
        }
    },
    publish: function(data) {
        var i = C,
            len = this.subscribers.length;

        // Iterate over the subscribers array and call each of
        // the callback functions.
        for (; i < len; i++) {
            this.subscribers[i](data);
        }
    }
};

// The observer is simply a function
var Cbserver = function (data) {
    console.log(data);
}

// Here's where it gets used.
observable = new Cbservable();
observable.subscribe(Cbserver);
observable.publish('We published!');
// 'We published!' will be logged in the console
observable.unsubscribe(Cbserver);
observable.publish('Another publish!');
// Nothing happens because there are no longer any subscribed observers
```

Walk through this a little at a time. First you create the `Cbservable` constructor, which just creates an empty array for the subscribers/observers to reside in. Then you create the prototype functions `subscribe`, `unsubscribe`, and `publish`. These three functions are necessary for pretty much any type of push observable, though of course you can change the names of them. The `subscribe` function just adds the observer function to the array of subscribers. The `unsubscribe` function will search for a given observer and remove it from the list, and `publish` iterates through the whole list of observers and executes them.

If you create the `Cbservable` object using an object literal rather than via a prototype, you can use mixins to make any object observable. Also, if you wanted to learn how to use the observer pattern with the pull method, or just a little more information about the observer pattern in general you can read the "JavaScript Design Patterns: Observer" post on Joe Zim's JavaScript Blog.

Command

The final pattern for this post and the whole series is the command pattern. Within the context of traditional object-oriented programming, this pattern is very much against the norm. Normally objects represent some type of noun (hence the name "object"), but with the command pattern, objects represent verbs. The object's job is to encapsulate the invocation of a method. It is simply an abstraction layer between the object that implements a method and the object that wishes to invoke the method. This tends be more useful in more traditional object-oriented languages and most often for user interfaces. With JavaScript, a command object can just be a function, which greatly simplifies things.

Commanding an alarm clock

Here's an example that gives a better understanding of how the command pattern is traditionally used. The example will be extremely simple to show how the command pattern works without confusing you with all the extraneous code. The example will be an alarm clock.

```
var EnableAlarm = function(alarm) {
    this.alarm = alarm;
}
EnableAlarm.prototype.execute = function () {
    this.alarm.enable();
}

var DisableAlarm = function(alarm) {
    this.alarm = alarm;
}
DisableAlarm.prototype.execute = function () {
    this.alarm.disable();
}

var SetAlarm = function(alarm) {
    this.alarm = alarm;
}
SetAlarm.prototype.execute = function () {
    this.alarm.set();
}
```

In this code, it's assumed that there is already an alarm object created with at least three methods: `enable`, `disable`, and `set`. I created three different classes to encapsulate those three methods. Each class follows a certain interface, which in this case just has the `execute` method.

The next bit of code assumes there's a `Button` class, which recieves two arguments: the string label for the button and a command object. The command object's `execute` method will be called when the button is pressed. Essentially, all I'm doing is taking a traditional object-oriented approach to binding events to UI elements:

```
var alarms = [/* array of alarms */],
    i = 0,
    len = alarms.length;

for (; i < len; i++) {
    var enable_alarm = new EnableAlarm(alarms[i]),
        disable_alarm = new DisableAlarm(alarms[i]),
        set_alarm = new SetAlarm(alarms[i]);

    new Button('Enable', enable_alarm);
    new Button('Disable', disable_alarm);
    new Button('Set', set_alarm);
}
```

There's a list of alarms and each of them needs some buttons to control them. So I create the three command objects for each alarm and send them in as parameters for three buttons for each alarm. With this example and the flexibility of JavaScript, though, you can shorten down all the command objects into just a couple lines of code:

```
var makeCommand = function( object, methodName ) {
    return {
        execute: function() {
            object[methodName]();
        }
    }
}

// This is how it's used now:
var alarms = [/* array of alarms */],
    i = C, len = alarms.length;

for (; i < len; i++) {
    var enable_alarm = makeCommand(alarms[i], 'enable'),
        disable_alarm = makeCommand(alarms[i], 'disable'),
        set_alarm = makeCommand(alarms[i], 'set');

    new Button('enable', enable_alarm);
    new Button('disable', disable_alarm);
    new Button('set', set_alarm);
}
```

Now it just returns an object literal with an `execute` method, rather than an object of a specific type. All that matters is that the same interface is used either way. Still, though, this seems like more work than it is worth, right? You could just set it up to use simple callback functions so it'd look like this:

```
var alarms = [/* array of alarms */],
    i = C, len = alarms.length;

for (; i < len; i++) {
    new Button('enable', alarm.enable);
    new Button('disable', alarm.disable);
    new Button('set', alarm.set);
}
```

You'll run into an issue here, though: when these callbacks are run, they'll lose their context, so `this` won't be a reference to the alarm, which will probably be a big deal. So you have to either create the `Button` constructor to take a context variable, create proxies of the alarm method (which is pretty much what you're doing with the command pattern), or rewrite the `Alarm` code with closures so that you can use a variable such as `self` or `that` to refer to the alarm within those methods. Well, if you noticed, the only option that doesn't require making changes to the existing classes is to use a proxy or some other type of abstraction layer between the alarms and buttons, which makes it a pretty good option, even though it requires more lines of code.

The simplicity of the matter

The biggest thing you have to realize about the example, like I said earlier, is that it is a relatively simple example to show you how the command pattern works, so it doesn't really show its true power and usefulness. The command pattern can get much more useful. In the example above, the `set` method should probably receive some parameters so that the alarm knows what to get set to. In this case, the command object can be responsible for figuring out what those parameters are supposed to be and passing them into `alarm.set` .

Another way the command pattern can be more useful is if it was in charge of calling more than just one method on `alarm` or on other objects. Basically adding any amount of complexity to the `execute` function makes the command object more useful.

Finally, the biggest way the command pattern can be useful is if it contains two (or more) functions, such as `execute` and `undo` . This way the `undo` function can be created to undo any actions that the `execute` functions took. To utilize this, when the button calls the `execute` method on the command object, it then puts that command object onto a stack of other command objects that have had their `execute` methods called. When the user hits Ctrl + Z to undo the most recent action, a command object will be pulled off of the stack and have its `undo` method called. This makes the command pattern far more useful and powerful than in the examples above.

Taking command

That brings the command pattern to a close for this article. If you wish to read more about the command pattern, and more specifically another means of creating command objects in JavaScript, you can read the "JavaScript Design Patterns: Command" article on Joe Zim's JavaScript Blog.