Understanding Design Patterns in JavaScript

Tilo Mitra on Feb 16th 2013 with 48 Comments

Tutorial Details

- - **Difficulty:** Intermediate
  - **Estimated Completion Time:** 45 minutes

Today, we're going to put on our computer science hats as we learn about some common design patterns. Design patterns offer developers ways to solve technical problems in a reusable and elegant way. Interested in becoming a better JavaScript developer? Then read on.

Republished Tutorial

Every few weeks, we revisit some of our reader's favorite posts from throughout the history of the site. This tutorial was first published in July, 2012.

Introduction

Solid design patterns are the basic building block for maintainable software applications. If you've ever participated in a technical interview, you've liked been asked about them. In this tutorial, we'll take a look at a few patterns that you can start using today.

What is a Design Pattern?

> A design pattern is a reusable software solution

Simply put, a design pattern is a reusable software solution to a specific type of problem that occurs frequently when developing software. Over the many years of practicing software development, experts have figured out ways of solving similar problems. These solutions have been encapsulated into design patterns. So:

- patterns are proven solutions to software development problems
- patterns are scalable as they usually are structured and have rules that you should follow
- patterns are reusable for similar problems

We'll get into some examples of design patterns further in to the tutorial.

Types of Design Patterns

In software development, design patterns are generally grouped into a few categories. We'll cover the three most important ones in this tutorial. They are explained in brief below:

1. **Creational** patterns focus on ways to create objects or classes. This may sound simple (and it is in some cases), but large applications need to control the object creation process.

2. **Structural** design patterns focus on ways to manage relationships between objects so that your application is architected in a scalable way. A key aspect of structural patterns is to ensure that a change in one part of your application does not affect all other parts.

3. **Behavioral** patterns focus on communication between objects.

You may still have questions after reading these brief descriptions. This is natural, and things will clear up once we look at some design patterns in depth below. So read on!

A Note About Classes in JavaScript

When reading about design patterns, you'll often see references to classes and objects. This can be confusing, as JavaScript does not really have the construct of "class"; a more correct term is "data type".

Data Types in JavaScript

JavaScript is an object-oriented language where objects inherit from other objects in a concept known as prototypical inheritance. A data type can be created by defining what is called a constructor function, like this:

view plaincopy to clipboardprint?

```
function Person(config) {
    this.name = config.name;
    this.age = config.age;
}

Person.prototype.getAge = function() {
    return this.age;
};

var tilo = new Person({name:"Tilo", age:23 });
console.log(tilo.getAge());
```

Note the use of the `prototype` when defining methods on the `Person` data type. Since multiple `Person` objects will reference the same prototype, this allows the `getAge()` method to be shared by all instances of the `Person` data type, rather than redefining it for every instance. Additionally, any data type that inherits from `Person` will have access to the `getAge()` method.

Dealing with Privacy

Another common issue in JavaScript is that there is no true sense of private variables. However, we can use closures to somewhat simulate privacy. Consider the following snippet:

view plaincopy to clipboardprint?

```
var retinaMacbook = (function() {

    //Private variables
    var RAM, addRAM;

    RAM = 4;

    //Private method
    addRAM = function (additionalRAM) {
        RAM += additionalRAM;
    };

    return {

        //Public variables and methods
        USB: undefined,
        insertUSB: function (device) {
            this.USB = device;
        },

        removeUSB: function () {
            var device = this.USB;
            this.USB = undefined;
            return device;
        }
    };
})();
```

In the example above, we created a `retinaMacbook` object, with public and private variables and methods. This is how we would use it:

view plaincopy to clipboardprint?

```
retinaMacbook.insertUSB("myUSB");
console.log(retinaMacbook.USB); //logs out "myUSB"
console.log(retinaMacbook.RAM) //logs out undefined
```

There’s a lot more that we can do with functions and closures in JavaScript, but we won’t get into all of it in this tutorial. With this little lesson on JavaScript data types and privacy behind us, we can carry along to learn about design patterns.

Creational Design Patterns

There are many different kinds of creational design patterns but we are going to cover two of them in this tutorial: Builder and Prototype. I find that these are used often enough to warrant the attention.

Builder Pattern

The Builder Pattern is often used in web development, and you've probably used it before without realizing it. Simply put, this pattern can be defined as the following:

> Applying the builder pattern allows us to construct objects by only specifying the type and the content of the object. We don't have to explicitly create the object.

For example, you've probably done this countless times in jQuery:

view plaincopy to clipboardprint?

```
var myDiv = $('<div id="myDiv">This is a div.</div>');

//myDiv now represents a jQuery object referencing a DOM node.

var someText = $('<p/>');
//someText is a jQuery object referencing an HTMLParagraphElement

var input = $('<input />');
```

Take a look at the three examples above. In the first one, we passed in a `<div/>` element with some content. In the second, we passed in an empty `<p>` tag. In the last one, we passed in an `<input />` element. The result of all three were the same: we were returned a jQuery object referencing a DOM node.

The `$` variable adopts the Builder Pattern in jQuery. In each example, we were returned a jQuery DOM object and had access to all the methods provided by the jQuery library, but at no point did we explicitly call `document.createElement`. The JS library handled all of that under the hood.

Imagine how much work it would be if we had to explicitly create the DOM element and insert content into it! By leveraging the builder pattern, we're able to focus on the type and the content of the object, rather than explicit creation of it.

Prototype Pattern

Earlier, we went through how to define data types in JavaScript through functions and adding methods to the object's `prototype`. The Prototype pattern allows objects to inherit from other objects, via their prototypes.

> The prototype pattern is a pattern where objects are created based on a template of an existing object through cloning.

This is an easy and natural way of implementing inheritance in JavaScript. For example:

view plaincopy to clipboardprint?

```
var Person = {
    numFeet: 2,
    numHeads: 1,
    numHands:2
};

//Object.create takes its first argument and applies it to the prototype of your new object.
var tilo = Object.create(Person);

console.log(tilo.numHeads); //outputs 1
tilo.numHeads = 2;
console.log(tilo.numHeads) //outputs 2
```

The properties (and methods) in the `Person` object get applied to the prototype of the `tilo` object. We can redefine the properties on the `tilo` object if we want them to be different.

In the example above, we used `Object.create()`. However, Internet Explorer 8 does not support the newer method. In these cases, we can simulate it's behavior:

view plaincopy to clipboardprint?

```
var vehiclePrototype = {

  init: function (carModel) {
    this.model = carModel;
  },

  getModel: function () {
    console.log( "The model of this vehicle is " + this.model);
  }
};
function vehicle (model) {

  function F() {};
  F.prototype = vehiclePrototype;

  var f = new F();

  f.init(model);
  return f;

}

var car = vehicle("Ford Escort");
car.getModel();
```

The only disadvantage to this method is that you cannot specify read-only properties, which can be specified when using `Object.create()`. Nonetheless, the prototype pattern shows how objects can inherit from other objects.

Structural Design Patterns

Structural design patterns are really helpful when figuring out how a system should work. They allow our applications to scale easily and remain maintainable. We're going to look at the following patterns in this group: Composite and Facade.

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):

view plaincopy to clipboardprint?

```
$('.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:

view plaincopy to clipboardprint?

```
$(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:

view plaincopy to clipboardprint?

```
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?

Behavioral Design Patterns

Any object-oriented software systems will have communication between objects. Not organizing that communication can lead to bugs that are difficult to find and fix. Behavioral design patterns prescribe different methods of organizing the communication between objects. In this section, we're going to look at the Observer and Mediator patterns.

Observer Pattern

The Observer pattern is the first of the two behavioral patterns that we are going to go through. Here's what it says:

> In the Observer Pattern, a subject can have a list of observers that are interested in it's lifecycle. Anytime the subject does something interesting, it sends a notification to

its observers. If an observer is no longer interested in listening to the subject, the subject can remove it from its list.

Sounds pretty simple, right? We need three methods to describe this pattern:

- `publish(data)`: Called by the subject when it has a notification to make. Some data may be passed by this method.
- `subscribe(observer)`: Called by the subject to add an observer to its list of observers.
- `unsubscribe(observer)`: Called by the subject to remove an observer from its list of observers.

Well, it turns out that most modern JavaScript libraries support these three methods as part of their custom events infrastructure. Usually, there's an `on()` or `attach()` method, a `trigger()` or `fire()` method, and an `off()` or `detach()` method. Consider the following snippet:

view plaincopy to clipboardprint?

```
//We just create an association between the jQuery events methods
```

view plaincopy to clipboardprint?

```
//and those prescribed by the Observer Pattern but you don't have to.
var o = $( {} );
$.subscribe = o.on.bind(o);
$.unsubscribe = o.off.bind(o);
$.publish = o.trigger.bind(o);

// Usage
document.on( 'tweetsReceived', function(tweets) {
    //perform some actions, then fire an event

    $.publish('tweetsShow', tweets);
});

//We can subscribe to this event and then fire our own event.
$.subscribe( 'tweetsShow', function() {
    //display the tweets somehow
    ..

    //publish an action after they are shown.
    $.publish('tweetsDisplayed);
});

$.subscribe('tweetsDisplayed, function() {
    ...
});
```

The Observer pattern is one of the simpler patterns to implement, but it is very powerful. JavaScript is well suited to adopt this pattern as it's naturally event-based. The next time you develop web applications, think about developing modules that are loosely-coupled with each other and adopt the Observer pattern as a means of communication. The observer pattern can become problematic if there are too many subjects and observers involved. This can happen in large-scale systems, and the next pattern we look at tries to solve this problem.

Mediator Pattern

The last pattern we are going to look at is the Mediator Pattern. It's similar to the Observer pattern but with some notable differences.

> The Mediator Pattern promotes the use of a single shared subject that handles communication with multiple objects. All objects communicate with each other through the mediator.

A good real-world analogy would be an Air Traffic Tower, which handles communication between the airport and the flights. In the world of software development, the Mediator pattern is often used as a system gets overly complicated. By placing mediators, communication can be handled through a single object, rather than having multiple objects communicating with each other. In this sense, a mediator pattern can be used to replace a system that implements the observer pattern.

There's a simplified implementation of the Mediator pattern by Addy Osmani in this gist. Let's talk about how you may use it. Imagine that you have a web app that allows users to click on an album, and play music from it. You could set up a mediator like this:

view plaincopy to clipboardprint?

```
$('#album').on('click', function(e) {
    e.preventDefault();
    var albumId = $(this).id();
    mediator.publish("playAlbum", albumId);
});
var playAlbum = function(id) {
    …
    mediator.publish("albumStartedPlaying", {songList: [..], currentSong: "Without You"});

};

var logAlbumPlayed = function(id) {
    //Log the album in the backend
};

var updateUserInterface = function(album) {
    //Update UI to reflect what’s being played
};

//Mediator subscriptions
mediator.subscribe("playAlbum", playAlbum);
mediator.subscribe("playAlbum", logAlbumPlayed);
mediator.subscribe("albumStartedPlaying", updateUserInterface);
```

The benefit of this pattern over the Observer pattern is that a single object is responsible for communication, whereas in the observer pattern, multiple objects could be listening and subscribing to each other.

In the Observer pattern, there is no single object that encapsulates a constraint. Instead, the Observer and the Subject must cooperate to maintain the constraint. Communication patterns are determined by the way observers and subjects are interconnected: a single subject usually has many observers, and sometimes the observer of one subject is a subject of another observer.

Conclusion

Someone has already applied it successfully in the past.

The great thing about design patterns is that someone has already applied it successfully in the past. There are lots of open-source code that implement various patterns in JavaScript. As developers, we need to be aware of what patterns are out there and when to apply them. I hope this tutorial helped you take one more step towards answering these questions.

Additional Reading

Much of the content from this article can be found in the excellent Learning JavaScript Design Patterns book, by Addy Osmani. It's an online book that was released for free under a Creative Commons license. The book extensively covers the theory and implementation of lots of different patterns, both in vanilla JavaScript and various JS libraries. I encourage you to look into it as a reference when you start your next project.

Tags: design patterns

By Tilo Mitra

Hey there! I'm a front-end engineer who is interested in startups, education and using technology to solve problems that ordinary folks have. You can connect with me on Twitter and peek at my projects on my site.

Note: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more