4 JavaScript Design Patterns You Should Know

Every developer strives to write maintainable, readable, and reusable code. Code structuring becomes more important as applications become larger. Design patterns prove crucial to solving this challenge – providing an organization structure for common issues in a particular circumstance.

JavaScript web developers frequently interact with design patterns, even unknowingly, when creating applications.

Although there is a diverse list of design patterns used in certain circumstances, JavaScript developers tend to use some patterns customarily more than others.

In this post, I want to discuss these common patterns to expose ways to improve your programming repertoire and dive deeper into the JavaScript internals.

The design patterns in question include the following:

- Module
- Prototype
- Observer
- Singleton

Each pattern consists of many properties, though, I will emphasize the following key points:

1. **Context**: Where/under what circumstances is the pattern used?
2. **Problem**: What are we trying to solve?
3. **Solution**: How does using this pattern solve our proposed problem?
4. **Implementation**: What does the implementation look like?

#Module Design Pattern

JavaScript modules are the most prevalently used design patterns for keeping particular pieces of code independent of other components. This provides loose coupling to support well-structured code.

For those that are familiar with object-oriented languages, modules are JavaScript “classes”. One of the many advantages of classes is *encapsulation* – protecting states and behaviors from being accessed from other classes. The module pattern allows for public and private (plus the lesser-know protected and privileged) access levels.

Modules should be Immediately-Invoked-Function-Expressions (IIFE) to allow for private scopes – that is, a closure that protect variables and methods (however, it will return an object instead of a function). This is what it looks like:

```
(function() {

    // declare private variables and/or functions

    return {
      // declare public variables and/or functions
    }

})();
```

Here we instantiate the private variables and/or functions before returning our object that we want to return. Code outside of our closure is unable to access these private variables since it is not in the same scope. Let’s take a more concrete implementation:

```
var HTMLChanger = (function() {
  var contents = 'contents'

  var changeHTML = function() {
    var element = document.getElementById('attribute-to-
change');
    element.innerHTML = contents;
  }

  return {
    callChangeHTML: function() {
      changeHTML();
      console.log(contents);
    }
  };

})();

HTMLChanger.callChangeHTML();        // Outputs: 'contents'
console.log(HTMLChanger.contents);   // undefined
```

Notice that `callChangeHTML` binds to the returned object and can be referenced within the `HTMLChanger` namespace. However, when outside the module, contents are unable to be referenced.

REVEALING MODULE PATTERN

A variation of the module pattern is called the **Revealing Module Pattern**. The purpose is to maintain encapsulation and reveal certain variables and methods returned in an object literal. The direct implementation looks like this:

```
var Exposer = (function() {
  var privateVariable = 10;
```

```
  var privateMethod = function() {
    console.log('Inside a private method!');
    privateVariable++;
  }

  var methodToExpose = function() {
    console.log('This is a method I want to expose!');
  }

  var otherMethodIWantToExpose = function() {
    privateMethod();
  }

  return {
      first: methodToExpose,
      second: otherMethodIWantToExpose
  };
})();

Exposer.first();        // Output: This is a method I want to
expose!
Exposer.second();       // Output: Inside a private method!
Exposer.methodToExpose; // undefined
```

Although this looks much cleaner, an obvious disadvantage is unable to reference the private methods. This can pose unit testing challenges. Similarly, the public behaviors are non-overridable.

#Prototype Design Pattern

Any JavaScript developer has either seen the keyword **prototype**, confused by the prototypical inheritance,

or implemented prototypes in their code. The Prototype design pattern relies on the JavaScript prototypical inheritance. The prototype model is used mainly for creating objects in performance-intensive situations.

The objects created are clones (shallow clones) of the original object that are passed around. One use case of the prototype pattern is performing an extensive database operation to create an object used for other parts of the application. If another process needs to use this object, instead of having to perform this substantial database operation, it would be advantageous to clone the previously created object.

Prototype Design Pattern on *Wikipedia*

This UML describes how a prototype interface is used to clone concrete implementations.

To clone an object, a constructor must exist to instantiate the first object. Next, by using the keyword **prototype** variables and methods bind to the object's structure. Let's look at a basic example:

```
var TeslaModelS = function() {
  this.numWheels    = 4;
  this.manufacturer = 'Tesla';
  this.make         = 'Model S';
}

TeslaModelS.prototype.go = function() {
  // Rotate wheels
}

TeslaModelS.prototype.stop = function() {
```

```
  // Apply brake pads
}
```

The constructor allows the creation of a single TeslaModelS object. When a creating new TeslaModelS object, it will retain the states initialized in the constructor. Additionally, maintaining the function **go** and **stop** is easy since we declared them with **prototype**. A synonymous way to extend functions on the prototype as described below:

```
var TeslaModelS = function() {
  this.numWheels    = 4;
  this.manufacturer = 'Tesla';
  this.make         = 'Model S';
}

TeslaModelS.prototype = {
  go: function() {
    // Rotate wheels
  },
  stop: function() {
    // Apply brake pads
  }
}
```

REVEALING PROTOTYPE PATTERN

Similar to Module pattern, the Prototype pattern also has a revealing variation. The Revealing Prototype Pattern provides encapsulation with public and private members since it returns an object literal.

Since we are returning an object, we will prefix the prototype object with a **function**. By extending our example above, we can choose what we want to expose in the current prototype to preserve their access levels:

```
var TeslaModelS = function() {
  this.numWheels    = 4;
  this.manufacturer = 'Tesla';
  this.make         = 'Model S';
}

TeslaModelS.prototype = function() {

  var go = function() {
    // Rotate wheels
  };

  var stop = function() {
    // Apply brake pads
  };

  return {
    pressBrakePedal: stop,
    pressGasPedal: go
  }

}();
```

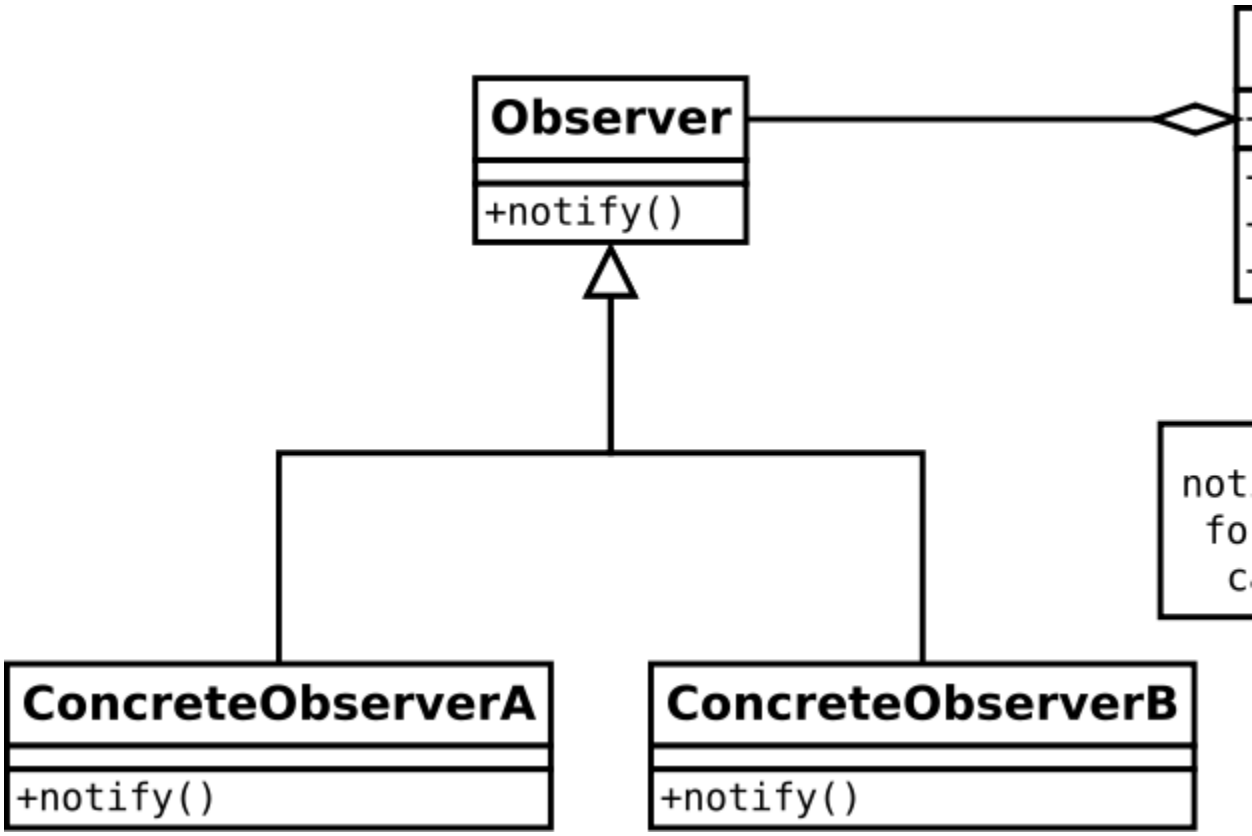
Note how the functions stop and go will be shielded from the returning object due to being outside of returned object's scope. Since JavaScript natively supports prototypical inheritance, there is no need to rewrite underlying features.

#Observer Design Pattern

There are many times when one part of the application changes, other parts needs to be updated. In AngularJS, if the `$scope` object updates, an event can be triggered to notify

another component. The observer pattern incorporates just that – if an object is modified it **broadcasts** to dependent objects that a change has occurred.

Another prime example is the model-view-controller (MVC) architecture; The view updates when the model changes. One benefit is decoupling the view from the model to reduce dependencies.



Observer Design Pattern on *Wikipedia*

As shown in the UML diagram, the necessary objects are the `subject`, `observer`, and `concrete` objects. The subject contains references to the concrete observers to notify for any changes. The Observer object is an abstract class that allows for the concrete observers to implements the notify method.

Let’s take a look at an AngularJS example that encompasses the observer pattern through event management.

```
// Controller 1
$scope.$on('nameChanged', function(event, args) {
    $scope.name = args.name;
});

...

// Controller 2
$scope.userNameChanged = function(name) {
    $scope.$emit('nameChanged', {name: name});
};
```

With the observer pattern, it is important to distinguish the independent object or the **subject**.

It is important to note that although the observer pattern does offer many advantages, one of the disadvantages is a significant drop in performance as the number of observers increased. One of the most notorious observers is **watchers**. In AngularJS, we can **watch** variables, functions, and objects. The **$$digest** cycle runs and notifies each of the watchers with the new values whenever a scope object is modified.

We can create our own Subjects and Observers in JavaScript. Let’s see how this is implemented:

```
var Subject = function() {
  this.observers = [];

  return {
    subscribeObserver: function(observer) {
      this.observers.push(observer);
    },
```

```
    unsubscribeObserver: function(observer) {
      var index = this.observers.indexOf(observer);
      if(index > -1) {
        this.observers.splice(index, 1);
      }
    },
    notifyObserver: function(observer) {
      var index = this.observers.indexOf(observer);
      if(index > -1) {
        this.observers[index].notify(index);
      }
    },
    notifyAllObservers: function() {
      for(var i = 0; i < this.observers.length; i++){
        this.observers[i].notify(i);
      };
    }
  };
};

var Observer = function() {
  return {
    notify: function(index) {
      console.log("Observer " + index + " is notified!");
    }
  }
}

var subject = new Subject();

var observer1 = new Observer();
var observer2 = new Observer();
var observer3 = new Observer();
var observer4 = new Observer();

subject.subscribeObserver(observer1);
subject.subscribeObserver(observer2);
subject.subscribeObserver(observer3);
subject.subscribeObserver(observer4);
```

```
subject.notifyObserver(observer2); // Observer 2 is notified!

subject.notifyAllObservers();
// Observer 1 is notified!
// Observer 2 is notified!
// Observer 3 is notified!
// Observer 4 is notified!
```

PUBLISH/SUBSCRIBE

The Publish/Subscribe pattern, however, uses a topic/event channel that sits between the objects wishing to receive notifications (subscribers) and the object firing the event (the publisher). This event system allows code to define application-specific events that can pass custom arguments containing values needed by the subscriber. The idea here is to avoid dependencies between the subscriber and publisher.

This differs from the Observer pattern since any subscriber implementing an appropriate event handler to register for and receive topic notifications broadcast by the publisher.

Many developers choose to aggregate the publish/subscribe design pattern with the observer though there is a distinction. Subscribers in the publish/subscribe pattern are notified through some messaging medium, but observers are notified by implementing a handler similar to the subject.

In AngularJS, a subscriber ‘subscribes’ to an event using $on(‘event’, callback), and a publisher ‘publishes’ an event using $emit(‘event’, args) or $broadcast(‘event’, args).

#Singleton

A Singleton only allows for a single instantiation, but many instances of the same object. The Singleton restricts clients from creating multiple objects, after the first object created, it will return instances of itself.

Finding use cases for Singletons is difficult for most who have not yet used it prior. One example is using an office printer. If there are ten people in an office, and they all use one printer, ten computers share one printer (instance). By sharing one printer, they share the same resources.

```
var printer = (function () {

  var printerInstance;

  function create () {

    function print() {
      // underlying printer mechanics
    }

    function turnOn() {
      // warm up
      // check for paper
    }

    return {
      // public + private states and behaviors
      print: print,
      turnOn: turnOn
    };
  }

  return {
    getInstance: function() {
```

```
        if(!printerInstance) {
          printerInstance = create();
        }
        return printerInstance;
      }
    };

  function Singleton () {
    if(!printerInstance) {
      printerInstance = intialize();
    }
  };

})();
```

The create method is private because we do not want the client to access this, however, notice that the *getInstance* method is public. Each officer worker can generate a printer instance by interacting with the *getInstance* method, like so:

```
var officePrinter = printer.getInstance();
```

In AngularJS, Singletons are prevalent, the most notable being services, factories, and providers. Since they maintain state and provides resource accessing, creating two instances defeats the point of a shared service/factory/provider.

Race conditions occur in multi-threaded applications when more than one thread tries to access the same resource. Singletons are susceptible to race conditions, such that if no instance were initialized first, two threads could then create two objects instead of returning and instance. This defeats the purpose of a singleton. Therefore, developers must be privy to synchronization when implementing singletons in multithreaded applications.

#Conclusion

Design patterns are frequently used in larger applications, though to understand where one might be advantageous over another, comes with practice.

Before building any application, you should thoroughly think about each actor and how they interact with one another. After reviewing the `Module`, `Prototype`, `Observer`, and `Singleton`design patterns, you should be able to identify these patterns and use them in the wild.