Software Design Patterns for the Web

CPEN400A – Building Modern Web Applications (Fall-2019)

Julien Gascon-Samson, Assistant Professor

ÉTS Montréal / University of Quebec
Department of Software and IT Engineering
Montréal, Canada

The design patterns mentioned in this lecture and the code samples are taken or adapted from the book ***Learning JavaScript Design Patterns****, by Addy Osmani*

Thursday, November 21, 2019

Last compilation: November 21, 2019

Design Patterns

Design Patterns: Definition

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Source: https://fr.wikipedia.org/wiki/Patron_de_conception

1995: a key book in Software Engineering was published *Design Patterns: Elements of Reusable Oriented-Object Software*, by the "Gang of Four" (GoF) (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides)

- Proposed a set of reusable techniques and object-oriented design principles for solving common problems in OO software development

Web Design Patterns

- You might have seen the "well known" design patterns in another course
- In this lecture, we will cover a key selection of design patterns that are highly relevant in a web development context (using JavaScript)
  - Reusable solutions that can be applied to typical problems in designing and writing web apps
- The patterns can be implemented using "vanilla" JS, or through a framework/library (ex., jQuery, dojo, etc.)
  - In fact, the most popular libraries often provide implementations of some of the design patterns (ex., Observer, Publish/Subscribe, etc.)
  - The Observer design patterns underpins the DOM interactions
- **The patterns discussed in this lecture as well as the code samples are taken from the book Learning JavaScript Design Patterns, by Addy Osmani**

Why design patterns?

1. **Design patterns model proven solutions** by the developer community.
2. **Design pattern can be reused easily** – they provide a generic ready-to-use solution that can be adapted to the context.
3. **Design patterns are expressive** – they integrate structure and vocabulary which eases communications about the solution to a given design problem.

Beware – they do not provide an exact solution to all problems :-)
We must also be careful not to overuse them :-)

Anti-patterns

Describes a **bad solution to a given problem**, and **how to improve the design to reach a better solution**.

Examples of anti-patterns in JS:

- Polluting the global namespace
- Passing strings to methods expecting a number as parameter and relying on implicit type conversion
- Modifying Object.prototype (really bad – why?)
- Using document.write (or document.innerHTML / document.innerText)
- Some will say: using eval

Design patterns – categories

- **Creational design patterns**: mechanisms for supporting the creation / instantiation of objects
  - Factory Method, Abstract Factory, Prototype, Singleton, Builder
- **Structural design patterns**: object composition and their relationships
  - Decorator, Facade, Flyweight, Adapter, Proxy, Bridge, Composite
- **Behavioral design patterns**: aim at improving / simplifying communications between heterogeneous objects
  - Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, Visitor, State, Strategy

Relevant design patterns in a web context

- **Module**
- **Revealing Module**
- **Singleton**
- **Observer**
- **Publish/Subscribe**
- Mediator
- **Constructor**
- **Prototype**
- Command
- **Facade**
- Factory
- **Mixin**
- Decorator
- Flyweight

Web Design Patterns

“Object litteral” syntax

A module allows for keeping the different portions of the code logically separated and organized.

- The simplest option is to use the “object litteral” syntax:

```
var myModule = {

  myProperty: "someValue",

  // object literals can contain properties and methods.
  // e.g we can define a further object for module configuration:
  myConfig: {
    useCaching: true,
    language: "en"
  },

  // a very basic method
  saySomething: function () {
    console.log( "Where in the world is Paul Irish today?" );
  },

  // output a value based on the current configuration
  reportMyConfig: function () {
    console.log( "Caching is: " + ( this.myConfig.useCaching ? "enabled" : "disabled") );
  },

  // override the current configuration
  updateMyConfig: function( newConfig ) {

    if ( typeof newConfig === "object" ) {
      this.myConfig = newConfig;
      console.log( this.myConfig.language );
    }
  }
};
```

Module: invoking


```
1  // Outputs: Where in the world is Paul Irish today?
2  myModule.saySomething();
3
4  // Outputs: Caching is: enabled
5  myModule.reportMyConfig();
6
7  // Outputs: fr
8  myModule.updateMyConfig({
9    language: "fr",
10   useCaching: false
11 });
12
13 // Outputs: Caching is: disabled
14 myModule.reportMyConfig();
```

- What is the limitation of that module definition syntax?

Module: invoking


```
1  // Outputs: Where in the world is Paul Irish today?
2  myModule.saySomething();
3
4  // Outputs: Caching is: enabled
5  myModule.reportMyConfig();
6
7  // Outputs: fr
8  myModule.updateMyConfig({
9    language: "fr",
10   useCaching: false
11 });
12
13 // Outputs: Caching is: disabled
14 myModule.reportMyConfig();
```

- What is the limitation of that module definition syntax?
  - Private data cannot be encapsulated.

Module Design Pattern

- Allows for keeping the data (variables & methods) **private**
  - Can emulate traditional classes

```
1  var testModule = (function () {
2
3    var counter = 0;
4
5    return {
6      incrementCounter: function () { return counter++; },
7      resetCounter: function () { counter = 0; }
8    };
9
10 })(); // IIFE (!) -- "immediately-invoked function expression"
         (un sous-patron) -- pourquoi avons-nous cela?
```

```
1  // Increment our counter
2  testModule.incrementCounter();
3
4  // Check the counter value and reset
5  testModule.resetCounter();
```

Module Pattern – classic "OO" model


```
1  var myNamespace = (function () {
2
3    var myPrivateVar, myPrivateMethod;
4
5    // A private counter variable
6    myPrivateVar = 0;
7
8    // A private function which logs any arguments
9    myPrivateMethod = function( foo ) {
10       console.log( foo );
11   };
12
13   return {
14
15     // A public variable
16     myPublicVar: "foo",
17
18     // A public function utilizing privates
19     myPublicFunction: function( bar ) {
20
21       // Increment our private counter
22       myPrivateVar++;
23
24       // Call our private method using bar
25       myPrivateMethod( bar );
26
27     }
28   };
29
30 })();
```

Module Pattern – shopping cart example (1)


```
1  var basketModule = (function () {
2    // privates
3    var basket = [];
4
5    // Return an object exposed to the public
6    return {
7
8      // Add items to our basket
9      addItem: function( values ) {
10       basket.push(values);
11     },
12
13     // Get the count of items in the basket
14     getItemCount: function () {
15       return basket.length;
16     },
17
18     // Get the total value of items in the basket
19     getTotal: function () { /* ... */ }
20   };
21 })();
```

Module Pattern – shopping cart example (2)


```
1  // basketModule returns an object with a public API we can use
2
3  basketModule.addItem({
4    item: "bread",
5    price: 0.5
6  });
7
8  basketModule.addItem({
9    item: "butter",
10   price: 0.3
11 });
12
13 // Outputs: 2
14 console.log( basketModule.getItemCount() );
15
16 // Outputs: 0.8
17 console.log( basketModule.getTotal() );
```

Module Pattern – “Exports” object


```
1  // Global module
2  var myModule = (function () {
3
4    // Module object
5    var module = {},
6      privateVariable = "Hello World";
7
8    function privateMethod() {
9      // ...
10   }
11
12   module.publicProperty = "Foobar";
13   module.publicMethod = function () {
14     console.log( privateVariable );
15   };
16
17   return module;
18
19 })();
```

Module Pattern – ES6

The export keyword can be used to *export* resources to make them public

- Resources that are not *exported* are private

```
1  // person.js
2
3  export var Person = function( firstName, lastName ){
4    this.firstName = firstName;
5    this.lastName = lastName;
6    this.gender = "male";
7  };
8
9  export var theProf = new Person("Julien", "Gascon-Samson");
10
11 var fourtyTwo = 42;
12 var twentyTwo = 22;
13 var foo = 11;
14 export {fourtyTwo, twentyTwo};
15 export {foo as eleven};
```

Module Pattern – Pros and cons

Pros:

- Provides a better decoupling and a better encapsulation

Cons:

- Given that private data is really private (and completely concealed by the use of closures – even through reflection), executing tests or debugging can be tricky
- No distinction between private and protected
- As private and public data are defined in different manners, altering the *visibility* can be trickier than in traditional OO languages

Revealing Module Patterns

A variant of the Module pattern:

- All data is defined in a "private" manner using closures
- An anonymous object is returned that contains pointers to the private features that should become public

```
1  var myRevealingModule = (function () {
2          var privateVar = "Ben Cherry", publicVar =
               "Hey there!";
3
4          function privateFunction() { console.log( "
               Name:" + privateVar ); }
5
6          function publicSetName( strName ) {
               privateVar = strName; }
7
8          function publicGetName() { privateFunction
               (); }
9
10         // Reveal public pointers to
11         // private functions and properties
12
13         return {
14             setName: publicSetName,
15             greeting: publicVar,
16             getName: publicGetName
17         };
18     })();
19 myRevealingModule.setName( "Paul Kinlan" );
```

Revealing Module Pattern – Pros and cons

Pros:

- Makes the distinction between private and public members more clear
- Brings together the module interface

Inconvénients:

- If a private method points to a public function, the public function cannot be "patched"
  - The private function will still point to the private implementation

Singleton Pattern

- Allows for having only one instance of a given class, in a standard manner
- Differs from a “static” class – a singleton allows for delaying the initialization of a given object when it is first used
- Avoiding putting variables & methods in the global namespace (pollution)

Singleton Pattern – Example


```
1  var mySingleton = (function () {
2    // Instance stores a reference to the Singleton
3    var instance;
4
5    function init() {
6      // Private methods and variables
7      function privateMethod(){  console.log( "I am private" );
           }
8      var privateVariable = "Im also private";
9      var privateRandomNumber = Math.random();
10
11     return {
12       // Public methods and variables
13       publicMethod: function () {
14          console.log( "The public can see me!" ); },
15       publicProperty: "I am also public",
16       getRandomNumber: function() {
17          return privateRandomNumber; }
18     };
19   };
```

Singleton Pattern – Example (2)


```
1   return {
2
3       // Get the Singleton instance if one exists
4       // or create one if it doesn't
5       getInstance: function () {
6
7         if ( !instance ) {
8           instance = init();
9         }
10
11        return instance;
12      }
13
14    };
15
16  })();
17
18  var singleA = mySingleton.getInstance();
19  var singleB = mySingleton.getInstance();
20  console.log( singleA.getRandomNumber() === singleB.
        getRandomNumber() ); // true
```

Singleton Pattern – Bad example


```
1     return {
2
3       // Always create a new Singleton instance
4       getInstance: function () {
5
6         instance = init();
7
8         return instance;
9       }
10
11    };
12
13  })();
14
15  var badSingleA = myBadSingleton.getInstance();
16  var badSingleB = myBadSingleton.getInstance();
17  console.log( badSingleA.getRandomNumber() !== badSingleB.
        getRandomNumber() ); // true
```

Singleton Pattern – reflection points

- Singletons are very convenient, but they should not be overused (like static objects)
- There are many good uses for singletons, but if they are overused, then it might be a sign that the design should be reevaluated
- **A good use case:** a global configuration that pertains to a given webapp (rather than creating several configuration variables in the global namespace)

Observer Pattern

- Allows an object (the *subject*) to have a list of decoupled objects that depend on it (*observers*, or *listeners*), and to notify them of a change of state
- Used a lot in standard web APIs – the whole event system of the W3C DOM is built on a large-scale observer pattern
- Underpins event handling in event-based languages and toolkits

Observer Pattern: main components

- **Subject**: maintains a list of observers, allows for adding and removing observers
- **Observer**: update interface for objects that must be notified of changes
- **ConcreteSubject**: sends notifications to observers when the state changes (ex., DOM node)
- **ConcreteObserver**: implements the update interface

Observer Pattern – observer list


```
1  function ObserverList(){ this.observerList = []; }
2
3  ObserverList.prototype.add = function( obj ){
4    return this.observerList.push( obj );
5  };
6
7  ObserverList.prototype.count = function(){
8    return this.observerList.length;
9  };
10
11 ObserverList.prototype.get = function( index ){
12   if( index > -1 && index < this.observerList.length ){
13     return this.observerList[ index ];
14   }
15 };
16
17 ObserverList.prototype.removeAt = function( index ){
18   this.observerList.splice( index, 1 );
19 };
20
21 ObserverList.prototype.indexOf = function( obj, startIndex ){
       /* ... */ } };
```

Observer Pattern – Subject interface


```
1  function Subject(){
2    this.observers = new ObserverList();
3  }
4
5  Subject.prototype.addObserver = function( observer ){
6    this.observers.add( observer );
7  };
8
9  Subject.prototype.removeObserver = function( observer ){
10   this.observers.removeAt( this.observers.indexOf( observer, 0
         ) );
11 };
12
13 Subject.prototype.notify = function( context ){
14   var observerCount = this.observers.count();
15   for(var i=0; i < observerCount; i++){
16     this.observers.get(i).update( context );
17   }
18 };
```

Observer Pattern – Observer interface


```
1  // The Observer
2  function Observer(){
3    this.update = function(){
4      // ...
5    };
6  }
```

Observer Pattern: usage

Hint: for "adding" a certain "behavior" to an object:

```
1  function extend( obj, extension ){
2    for ( var key in extension ){
3      obj[key] = extension[key];
4    }
5  }
```

- Defining an observer:

```
1  var txt = document.getElementById("mytextbox");
2  extend(txt, new Observer());
3  txt.update = function(context) { txt.value = context; }
```

- 2Defining a subject:

```
1  var btn = document.getElementById("mybutton");
2  extend(btn, new Subject());
3  btn.addObserver( document.getElementById("mytextbox") );
4  btn.onclick = function() { btn.notify("World"); }
```

Observer Pattern – Class activity

Consider the HTML page observer.html which contains a button (id=mybutton) and 10 text fields (<input id="mytextbox1"/> to 10). Consider the jsobserver.js library that is included by the HTML page (it contains the relevant classes for implementing an Observer pattern), as seen in the previous slides. You must implement the following features:

1. Each text box must implement an Observer that must provide an implementation of the update method, which will modify the value of the textbox according to the parameter passed to update.
2. The button must be a Subject. When the user clicks on the button, all observers must be notified – the string "World" must be passed as parameter to the notification.
3. When the user clicks on a text field, you must add this text field (observer) to the list of observers of the button (which is a subject), and set its color to yellow (style="backgroud-color: yellow;"). However, if the text field is already in the list of observers, you must remove it, and reset the style to default (style="").