BIOINFORMATICS

Vol. 00 no. 00 2015
Pages 1–5

JavaScript Technology: Module Pattern

Sattler Patrick, Andreas Kammerloher
Bioinformatics and Computational Biology Departement
Rostlab

Received on 08.05.2015

Associate Editor: Guy Yachdav

ABSTRACT
Motivation: This paper provides an in depth description of the JavaScript Module Pattern. The Module Pattern is a way to implement classlike behaviour in JavaScript by using the closure of anonymous functions to provide privacy and scope. It also explains the advantages and disadvantages that come with using the Module Pattern for your application. Finally it presents an application that has used the Module Pattern.

Contact: sattler.patrick@posteo.de
andreas.kammerloher@arcor.de

1 INTRODUCTION

Logically separating pieces of code allows for easier reading and understanding of said code. It is common for programming languages to implement this through object orientation.

Since JavaScript was not designed with object orientation in mind certain hacks are required. The Module Pattern is one such hack. It implements classlike behaviour by allowing public and private properties in one single datastructure. This keeps the properties out of the global namespace and prevents unwanted modification from outside the structure. The Module Pattern is an entirely separate alternative to JavaScript prototype based class emulation.

2 MODULE PATTERN:

2.1 Overview:

As mentioned in the Introduction, the Module Pattern allows public and private properties in one datastructure. This is achieved by creating an object inside an anonymous function, called ”module” in the code example, that is executed immediately after its definition. The object is returned at the end of the function. Everything in this function exists inside a closure, providing privacy and state.

All public methods and variables are defined as part of the object, all private ones are created independent of it. The return value is saved in a variable thus allowing its properties to be accessed from outside the functions scope. The state provided by the closure remains consistent over all invocations of the Module. The following codes illustrates the basic Module Pattern.

```
//define anonymous function
var My_Module = (function(){
  var module = {};
  var private_variable = 3;
  module.public_variable = 9;

  //return public part of the module
  return module;

//execute anonymous function immediately
})();
```

Public properties can now be accessed like this:

```
My_Module.public_property
```

While private properties can not be accessed from outside the anonymous functions closure, the following code will not produce an error, but instead create a new public variable named ”private_variable”

```
My_Module.private_variable
```

2.2 Strengths and Weaknesses of the Module Pattern:

A big advantage of the Module Pattern is scalabilty. Modules are isolated pieces of code and can be added or removed fairly easily since they are mostly independent of other code. The isolation also allows for a simple distribution of work among several programmers as they can be assigned different Modules to implement and can work separately.

Restricting variables to a local scope leads to less clutter in the global namespace. This, in addition to the fact that the public variables are bound to one module variable, prevents variable name conflicts which can be a problem when importing libraries or working with a team of developers. On top of that Modules can be augmented to add more methods and variables when required.

The Module Pattern also has a few downsides. For one, inheritance requires the inheriting Module to explicitly copy all properties of the super Module. Also it is not possible to manipulate the private properties of a Module from outside the Module’s scope. Not even while augmenting it. Since private properties only exist within the anonymous function’s closure they can not be accessed from the outside at all.

Another problem is that changing the visibility of a property requires the programmer to edit every line of code that contains said property. This is the case because visibility is not defined by a single keyword but by whether the property is part of the returned object inside the Module so it is either accessed by “module.property” if it is public or just “property” if it is private.

2.3 Global Variables:

Global variables can make code hard to read or maintain since it is difficult for humans to determine where in the code they are used. The Module Pattern allows to import global variables into a Module explicitly by using them as arguments for the Module’s anonymous function. Of course global variables can be accessed inside the functions closure regardless of whether they are imported explicitly or not. Using global variables as arguments is done for increased readability and is highly recommended (2).

```javascript
//use global variable as function parameter
var My_Module = (function(global_variable){
  var module = {};

  //access global variable
  module.global_incremented = function(){
    return global_variable + 1;
  }

  return module;

//use global variable as argument
})(global_variable);
```

2.4 Namespacing

To make the whole project even better organized it’s possible to add sub modules. This means its possible to create something similar to a namespace, from other programming languages you may know and its as easier as you probably think.

```javascript
    My_Module.my_new_submodule = (function(){
      var submodule = {};
        //...
        return submodule;
    })();
```

This is a pretty obvious concept, but it’s also important to know. Obviously the submodules can be seen as normal modules and they do have the same features. That means it’s possible to create more levels in your namespace.

2.5 Inheritance:

Inheritance allows a new Module to derive all public properties from another Module. It is not only useful to reduce the lines of code in a program but it also increases readablity as related Modules are clearly recognizable as such.

Modules inherit from other Modules by copying all their non-private properties. To achieve this the super Module’s public part is imported into the inheriting Module as an argument. A reference to the super Module is saved and all its properties are recreated.

```javascript
//define anonymous function with
//super module as parameter
var My_Module = (function(super_module){
  var module = {};

  //create reference to super module
  module.super = super_module;

  //copy all properties from the
  //super module to the new module
  for (key in super_module) {
    if (super_module.hasOwnProperty(key) {
      module[key] = super_module[key];
    }
  }

//import super module as argument
//to new anonymous function
})(super_module);
```

2.6 Augmentation:

As was briefly touched upon in the ”Strengths and Weaknesses of the Module Pattern” section, Modules can be augmented after their original definition. A new anonymous function is defined and executed immediately. It takes the variable that held the original Module as an argument. Instead of creating a new object to contain all public properties, new public properties are added to the original modules object. The public properties of the original Module can also be modified. The original modules object is returned at the end of the function and the return value is used to overwrite the variable that held the original Module’s public properties. The new public properties can now be accessed along the original ones through this variable.

```javascript
//define anonymous function with
//original Module as parameter
var Original_Module = (function(original_Module){

  //add new properties
  original_Module.new_variable = 25;
  original_Module.new_method = function(){
    return 3+5;
  };

  //overwrite original properties
  original_Module.original_variable = 18;
  original_Module.original_method = function(){
    return 4+2;
  }

  return original_Module;

//import original Module as argument
//to new anonymous function
})(Original_Module);
```

It is worth noting that the original private properties can not be accessed in the new anonymous function, since they are only accessible from inside the original anonymous functions closure. New anonymous properties can be defined but they only exist in their own functions closure and can not be accessed from anywhere else.

These augmentations can be even be done from different files and can even be used when the original Module has not been created. This will be explained in detail in the following section.

2.7 Advanced Module Pattern concepts

Here we would like to introduce you to some advanced Module Pattern concepts like separating your module on to different files. Using more files for one module can open you a whole bunch of new possibilities, for example it is simplifying the project structure. We will also talk about the loading order of the files and the possibilities you have there. In the end we'll show you how to manage the private scope over several files.

2.7.1 A Module in different files Like we mentioned above it's possible to organize your modules different tasks in more files. But more important is that it's possible now to independently load your module extensions. That means if you can't discover which module extensions you need, at page load, you can do it now dynamically as soon as you know it. It's only necessary to include a dynamic loading framework and you can use it. This means you can load your different strategies dynamically and on user demand. Reducing data traffic is one good reason to use this concept.

2.7.2 Augmentation possibilities Splitting the module in to different files implies also you have to choose between two different augmentation possibilities:

- loose augmentation (no defined loading order)
- tight augmentation (a defined loading order by your code)

This augmentation possibilities differ in the function argument. The loose augmentation has the following structure:

```
var module_name = function(module) {
    ...
    return module
} (module_name || {})
```

This means if the module is already instanced we use it to extend and overwrite it. Otherwise we use a new empty object. If you use loose augmentation the var keyword is mandatory. The loose augmentation has no strict loading order and it's possible to load them as you wish.

On the other side there is the tight augmentation structured like this:

```
[var] module_name = function(module) {
    ...
    return module
} (module_name)
```

Here the var keyword is optional and not needed. The base of your module isn't structured like this but has the base structure without the module as an argument. Only the extensions for that base module need this tight augmentation structure. This implies that the base of your module has to be loaded before the extensions. But for the extensions there is no defined loading order (2).

There is a problem you could face if you load the extensions randomly, because of some relations betweeen your extensions. That's a really important thing you have to consider. So if there are constraints related to your module extensions, it's necessary to adapt also your loading order in order to match the constraints.

2.7.3 Cross-File private state So now you probably want to share your private variables, which normally are only accessible inside the generator function, to other extensions of your module. Even if you have more module extensions in one file (in my opinion there is no point in that, but netherless), it isn't possible to access private variables of other extensions.

But there is a solution for this problem: create a private and public object (here named _private), both point to the same object. Its properties are in the cross-file private scope. After the initialization you have to call a "seal" function (called _seal in our example) which deletes the public variables _private, _seal and _unseal. The unseal function (called _unseal in our example) recreates the public variables _private, _seal and _unseal. To fullfill all this requirements the following code has to be in every module extension function (2).

```
var _private = module._private = module._private
    || {};
var _seal = module._seal = module._seal ||
    function () {
        delete module._private;
        delete module._seal;
        delete module._unseal;
    };
var _unseal = module._unseal = module._unseal ||
    function () {
        module._private = _private;
        module._seal = _seal;
        module._unseal = _unseal;
    };
```

This code creates a local "private" object which points to the same as the public "private" object. They both point to the public "private" object if it exists, or to a new object if it doesn't. The same workflow is valid for the "seal" and the "unseal" functions, if the public function exists point to it, if not create the function.

Important is also if it's necessary to load an extension not directly on page loading, but later on, before loading it, the "unseal" function must be called and afterwards the "seal" function. Now you have a full working cross-file private scope.

3 OUR WEB APP - TIC TAC TOE OVERVIEW:

We developed a small web app where it's possible to play the famous Tic Tac Toe game. For a better layout and to simplify our lifes we added the Bootstrap framework. It is a Open source HTML, CSS and JavaScript framework and they advertise it as the most popular. To load our files dynamically we used the 3rd party framework LABjs.

3.1 A simple overview of our Tic Tac Toe app

The webseite has a simple structure on the navigation bar on the top it's possible to select the AI strenght. To show the loading part of our talk we have links there instead of JavaScript handling. The navigation bar is made with Bootstrap. It's really easy to add there different items and links.

All strategies are in different files and extend the main module, with at least the AI computation method "selectNextKiField". The default page loads all files parallel, which means that a random file should be loaded as last, but locally there is no difference between parallel and sequentially loading. Thats partly owed to the small file sizes and partly to the immediate access to the files on the local filesystem. But with larger filesizes and/or a slower Internet connection, e.g. a mobile connetion, there will be differences on every page load. And if a strategy is chosen only the corresponding file gets loaded and there isn't any parallel loading to take care of.

Of course our game has a 3x3 field, filled with buttons. A click on a button sets the field as a user field and directly triggers the AI field selection. The computation is obviously influenced by the chosen AI strategy. After every move we check for a winner, if there is one the user gets a alert as notification. The game can be restarted by reloading the page.

3.2 Why LAB.js and how does it work

LABjs (3) is an all-purpose, on-demand JavaScript loader. It reduces the resource blocking during page-load, to optimize the websites performance. We chose this framework because it's in maintenance mode, which means there won't be any mayjor changes. This is important if you don't want to change the code with each update of the framework. The <script> tags in the header or at the end of the file will be replaced by $LAB calls to load the JavaScript files. In our case this transformed this code:

```
<script src="https://ajax.googleapis.com/ajax/
    libs/jquery/1.11.2/jquery.min.js"></script
    >
<script src="js/bootstrap.min.js"></script>
<script src="tictactoe.js"></script>
<script src="tictactoeNoStrategy.js"></script>
<script src="tictactoeStrategyRandom.js"></
    script>
<script src="tictactoeGoodStrategy.js"></
    script>
```

into this:

```
<script src="js/LAB.src.js"></script>
<script>
    $LAB
    .script('https://ajax.googleapis.com/ajax/
        libs/jquery/1.11.2/jquery.min.js').
        wait()
    .script('js/bootstrap.min.js')
    .script('tictactoe.js');
</script>
```

With the "script" function called on the $LAB Module it's possible to load a file. The Bootstrap framework requires the jQuery framework to be loaded first. This means they can't be loaded asynchronously so by calling the "wait" function, the jQuery framework gets loaded fully and afterwords the following files get loaded. This code implies that LABjs supports method chaining.

You probably noticed the lack of our strategy JavaScript files, with LABjs we load them now in the "tictactoe.js" file. We decide there by the URL parameter "strategy" which files to load.

```
// get Parameter value
var strategy = GetURLParameter("strategy");
if (strategy === "none") {
    // load only the files really needed
    TICTACTOEAreaModule._unseal()
    $LAB.script('tictactoeNoStrategy.js').wait
        (function () {
        TICTACTOEAreaModule._seal();
    });
} else
    ...
```

At first we get the parameter value and afterwards we check the value and load the corresponding JavaScript file. You propably also noticed the unseal and seal method calls to get the cross file private state. The anonymous function gets called after the the file is fully loaded.

4 WHICH ASPECTS OF THE MODULE PATTERN WERE USED IN OUR WEBAPP?

The new private scope is obviously very usefull. Now its possible to hide e.g. the game area. This means it's protected against unauthorized modifications.

It's also really important that its possible now to have private functions. For maintainability reasons you probably want to split large methods into smaller ones, without sharing them to the whole project. This simplifys the nameing of methods and variables.

We mentioned above that we have a file for each strategy and one for the base module. Because of that we also wanted to share some of our private variables and methods to other files. We did that exactly as described above.

We used the loose augmentation possibility but we have here some special case. Our module with all basic functionallity is in the "tictactoe.js" file, but this module needs always at least one extension to work correctly. At the other side the extensions with the strategies obviously can't work correctly without the base madule. So we have some extra dependencies, which can't be solved only by the choice of the right augmentation type. This is probably an error in our webapp design.

We didn't used inheritance since there was no use of it. We also didn't imported global variables, because there was no demand, neither for the namespacing.

REFERENCES

[1]Addy Osmani: *Learning JavaScript Design Patterns*, O'Reilly Media http://addyosmani.com/resources/essentialjsdesignpatterns/book/#modulepatternjavascript

[2]*adequatelygood.com on the Module Pattern, May 8th '15*, http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.htm

[3]*labjs.com on LABjs, May 8th '15*, http://labjs.com