Code Reuse Patterns

By David Martin Gonzalez, Enrico Gigantiello

1. Intro

This document is the written transcript of the seminar Javascript Patterns and Antipatterns held in April 2015 on the subject of Code Reuse Patterns. We will present the two main categories in which these patterns are divided and present the refactoring of some Javascript code which was lacking or making wrong use of these patterns.

It is clear to all JS developers that over the last years this functional object oriented language has tried to provide some syntactical sugar to ‘look like’ classical object oriented languages (classical meaning using classes). The reason for this was probably to encourage developers from other OO languages to use JS. But as we all know JS has no actual classes but only prototypes.

For this reason we will in this paper distinguish between patterns using the concept of class to obtain code reuse (Classical) and the ones using JS’s proper features to obtain the same goal (Modern).

1. Classical Code Reuse Patterns

Code reuse in classical OO languages is obtained through inheritance. So given the following code example our focus in this chapter will be on the implementation of the inherit function at the bottom:

// the parent constructor

**function** **Parent**(name) {

**this**.name = name || 'Adam';

}

// adding functionality to the prototype

Parent.prototype.say = **function** () {

**return** **this**.name;

};

// empty child constructor

**function** **Child**(name) {}

// inheritance magic happens here

inherit(Child, Parent);

* 1. Default Pattern

Or first approach which would come to mind knowing JS features:

**function** **inherit**(C, P) {

C.prototype = **new** P();

}

**var** kid = **new** Child(); //gets functionality from the Parent()

//instance via the prototype

kid.say(); // "Adam"

* As you can see in the code snippet we are setting the child’s prototype to a new parent object.
* In the last line kid can call the method say and look up his prototype chain to use it’s parents method.
  + 1. Drawbacks:
* Inherits both own properties (added to "this") and prototype properties and methods (such as say()). However, own properties are likely to be specific to one instance and not reusable
* Doesn’t enable you to pass parameters to the child constructor:

**var** s = **new** Child('Seth');

s.say(); // "Adam"

* 1. Rent a Constructor

Solves the problem of passing arguments to the parent.

Only inherit properties added to "this" inside the parent constructor (by copy, not by reference).

**function** **Child**(a, b, c, d) {

Parent.apply(**this**, **arguments**);

}

* The function uses the Function.prototype.apply() method to use the parent constructor on an instance of the child (the ‘this’ passed as first argument) while passing an array of arguments to it. We will see later how the apply method can be invoked to reuse any other method of an object.
  + 1. Multiple Inheritance

This pattern can easily be extended to inherit properties from more than one object:

**function** **Mom**() {

**this**.hair = "brown";

}

**function** **Dad**() {

**this**.eyes = "green";

}

**function** **Child**() {

Mom.apply(**this**);

Dad.apply(**this**);

}

**var** kid = **new** Child();

**console**.dir(kid);

// Child

// eyes: "green"

// hair: "brown"

// \_\_proto\_\_: Child

* + The child has now both green eyes from the father and brown hair from the mother
    1. Drawbacks

This pattern doesn’t consider the prototype link at all while inheriting the properties. In the following example the kid.say method will be undefined because it was set on the parent’s prototype which is not reachable by the kid:

// the parent constructor

**function** **Parent**(name) {

**this**.name = name || 'Adam';

}

// adding functionality to the prototype

Parent.prototype.say = **function** () {

**return** **this**.name;

};

// child constructor

**function** **Child**(name) {

Parent.apply(**this**, **arguments**);

}

**var** kid = **new** Child("Patrick");

kid.name; // Patrick

**typeof** kid.say; // und­­efined

* 1. Rent And Set Prototype

Combines the previous two: we borrow the constructor and set the prototype to a new instance of the constructor.

**function** **Child**(a, c, b, d) {

Parent.apply(**this**, **arguments**);

}

Child.prototype = **new** Parent();

* The result objects get copies of the parent’s own members and references to the parent’s reusable functionality.
* The child can also pass any arguments to the parent constructor.
* A minor drawback is that the parent constructor is called twice.
* If both the Child and the Parent have a ‘name’ property the one in Child will be looked up first
  1. Share the prototype

Among the good principles of code reuse in JS is the idea of placing the reusable parts of code of an object in it’s prototype and not in it’s own properties. We can apply this principle by making the child’s and parent’s prototype to be the same object:

**function** **inherit**(C, P) {

C.prototype = P.prototype;

}

* + 1. Drawbacks:
* If someone modifies the prototype for one object type, all the other associated objects would change as well. This might be awfull in long inheritance chains.
  1. A temporary constructor (Proxy)

We can share a copy of the parent’s prototype by setting it to an empty function and then copying it on the child prototype:

**function** **inherit**(C, P) {

**var** F = **function** () {};

F.prototype = P.prototype;

C.prototype = **new** F();

}

//child only inherits from prototype

* This solves the drawback of the previous pattern
  + 1. Enriching Proxy:

This pattern can be enhanced by adding some features typical to classical OO languages:

* + - 1. Uber:
* We will add a reference to the parent’s prototype to access it with the obj.super notation.
* "super" is a reserved word and we should not use "superclass" because there are no classes in JavaScript. Let's use "uber" instead.

C.uber = P.prototype;

* + - 1. Resetting Constructor Pointer:
* Let's reset the pointer to the constructor function in case you need it later.

C.prototype.constructor = C;

If we don’t reset the pointer to the constructor, then all children objects will report that Parent() was their constructor.

* + 1. Complete Proxy pattern

Last optimization: avoid creating the temporary (proxy) constructor every time you need inheritance:

**var** inherit = (**function** () {

**var** F = **function** () {};

**return** **function** (C, P) {

F.prototype = P.prototype;

C.prototype = **new** F();

C.uber = P.prototype;

C.prototype.constructor = C;

}

}());

* 1. Klass - emulating classes

Klasses provide syntactic sugar for emulating actual classes in JavaScript. Many different libraries provide different implementations of Klasses, but they all share:

* Convention on the name of the constructor method
* Inheritance between classes
* Access to superclass

We wont dig into the Klass’s implementation but show two use examples. An example implementation can be found in Stoyanov's JavaScript Patterns.

* + 1. Creating a klass:

**var** Man = klass(null, {

\_\_construct: **function** (what) {

**console**.log("Man's constructor");

**this**.name = what;

},

getName: **function** () {

**return** **this**.name;

}

});

**var** first = **new** Man('Adam'); // logs "Man's constructor"

first.getName(); // "Adam"

* Two parameters: a parent class to be inherited and the actual implementation of the new class.
* Convention: the class's constructor must be a method called \_\_construct.
  + 1. Extending a klass:

**var** SuperMan = klass(Man, {

\_\_construct: **function** (what) {

**console**.log("SuperMan's constructor");

},

getName: **function** () {

**var** name = SuperMan.uber.getName.call(**this**);

**return** "I am " + name;

}

});

**var** clark = **new** SuperMan('Clark Kent'); // logs "SuperMan's constructor"

clark.getName(); // "I am Clark Kent"

clark **instanceof** Man; // true

clark **instanceof** SuperMan; // true

* Inherits from Man.
* getName() is called first by using the uber (super) static property of SuperMan.
  1. Remark about new

Rather then a pattern this remark is added to this context as a confortable workaround to the problem of forgetting to use the ‘new’ when creating an object:

**function** **Person**(name) {

**this**.name = name;

}

**var** john = **new** Person('John');

**var** lars = Person('Lars');

* Lars would be undefined and "name" would be bound to the global object!
  + 1. Workaraund

**function** **Person**(name) {

**if** (!(**this** **instanceof** Person))

{

**return** **new** Person(name);

}

**this**.name = name;

}

* 1. Final Consideration

We have showed once again the extreme flexibility of JS, to the point that it can emulate a differently Object Oriented language.

But do we really want to force this dynamic language to copy another language’s syntax and semantics? The next group of patterns will show us the brilliant native ways JS has to solve code reuse problems.