**VBScript++**

Akien MacIain, Test Automation Architect, Delta Dental of California

June 2009

**Table Of Contents**

[Introduction](#h.gjdgxs)

[Overview](#h.30j0zll)

[The BagObject Framework](#h.1fob9te)

[Creating new BagClasses](#h.3znysh7)

[Adding Methods to a BagClass](#h.2et92p0)

[Adding Properties to a BagClass](#h.tyjcwt)

[Creating BagInstances of a BagClass](#h.3dy6vkm)

[Calling methods and properties](#h.1t3h5sf)

[How it's most commonly used](#h.4d34og8)

[Custom Signatures](#h.2s8eyo1)

[Constructors](#h.17dp8vu)

[Debugging](#h.3rdcrjn)

[Other little amusing oddities](#h.26in1rg)

[Summary](#h.lnxbz9)

[Attachments](#h.35nkun2)

## ***Introduction***

For the entire time I’ve been working with HP QuickTest Pro and VBScript, I have had a yearning for real object support. Specifically, inheritance and polymorphism. I am a big fan of reusing every piece of code I can, and these two attributes contribute more to my productivity than any other concept in software development.

In addition, at one point when we were performing a refactoring of our automation framework, I found myself implementing a port of the Log4j logging framework in VBScript (for more on log4j see <http://logging.apache.org/log4j/1.2/index.html> and <http://en.wikipedia.org/wiki/Log4j>). To do that effectively I needed the ability to pass in varying numbers of parameters to methods and I needed inheritance.

## ***Overview***

To implement this kind of functionality, I went back to what I knew of the internals of a typical C++ implementation. Internally, the compiler sees objects as a collection of metadata, pointers to functions, function prototype maps, and so on. To recreate this, I created a VBScript class called BagDict. Internally a BagDict is made up of two standard Scripting.Dictionary objects, one for object data, and one for metadata. The BagDict implements the basic Scripting.Dictionary interface, which access the object data. We then go on to implement a group of additions to the basic interface, which allow us to create VBScript++ classes, instances, add methods and properties to classes, and call those methods and properties. The BagDict is the substrate that BagObjects are built on.

One piece of terminology we need before going on is the distinction between an object class and instance, in the VBScript sense, and a VBScript++ BagObject, BagClass and BagInstance. BagObjects are instances of the BagDict VBScript class which contain additional metadata that makes them either a BagClass or a BagInstance. BagClasses and BagInstances are the VBScript++ counterparts to the traditional VBScript class and instance. BagObject refers to an item which could be either an instance or a class.

Before we start looking at how all this works in practice, there is one more item we need to cover. As I mentioned above, one of the requirements when I started this project was I wanted to be able to pass a variable numbers of parameters into things, so I could have evolving functionality but the static interfaces that VBScript requires. I took my inspiration from Perl, which allows you to pass an anonymous hash in to your functions.

As an example, let’s look at the following code:

Function DoSomething **(**args**)**

BagDictCreate args**,** NULL

Print args**(**“my\_data”**)**

End Function

DoSomething "my\_data=>thing to print|some\_other\_data=>foo"

The BagDictCreate function takes the args variable by reference (which is a string) and converts it into a BagDict. Each key/value pair in the string is separated by a pipe (|) and the => symbol is used to separate the keys from the values. You can also pass in arrays where the first entry is a key, the second a value, the third a key, and so on. BagDict instances created this way are not BagObjects, even though they are instances of the VBScript BagDict class. They lack the metadata to be BagObjects.

As a matter of style, we refer to keys passed in this way in the documentation either as key:my\_data, object key:my\_data or metadata key:my\_data. Whenever object or metadata is not specified, you should assume the key refers to object data. This is reflected in the documentation included in the code.

## ***The BagObject Framework***

### **Creating new BagClasses**

In the VBSCript++ model, everything inherits from ClassBase. To create a new class, we do this:

Dim classFileSystemItem

Set classFileSystemItem **=** ClassBase.MakeClass **(**"ClassFileSystemItem"**,** NULL**, \_**

"self.is\_virtual=><eval>True"**)**

There are three parameters to the MakeClass method:

* The first parameter identifies the name of the new class we are creating, in this case ClassFileSystemItem. Does not have to match the name of the variable it's stored in. (Though in practice I always use the same names for the two to avoid confusion.)
* The second parameter is any object data we wish to add at this point, in the same form that we would pass in to BagDictCreate. In this case, there isn’t any, so we pass in NULL.
* The third parameter is any metadata we might wish to add at this point. In this case, we are adding self.is\_virtual, which we set to True (virtual classes are not allowed to be instantiated, only inherited from). The <eval> keyword means that this expression will be evaluated in the global namespace before the assignment to the new BagDict.

The result of this call is a new BagObject of type BagClass that we store in to the classFileSystemItem variable.

### **Adding Methods to a BagClass**

Now that we have our new class, we can begin adding methods and properties to the class. To add a method to the class, we first need to create a new, non-virtual instance of our base class, and then call the ApplyMethod routine of our new class.

Dim classFile

Set classFile **=** classFileSystemItem.MakeClass**(**"ClassFile"**,** NULL**, \_**

"self.is\_virtual=><eval>False" **)**

We have now created a new concrete class, called ClassFile, that inherits from ClassFileSystemItem. By setting self.is\_virtual to False we’ve ensured that we can actually create BagInstances of this class. Now let's add the method. We add methods to the class by calling ApplyMethod:

classFile.ApplyMethod "Delete"**,** NULL

Function ClassFile\_Delete**(**self**,**args**)**

self.VerifyHasKeys "file\_name"**,** "ClassFile\_Delete fails missing key:file\_name"

If ClassFile\_Exist\_Get**(**self**,** args**)** Then

ClassFileSystemItem\_FSO.DeleteFile**(**self**(**"file\_name"**))**

End If

End Function

Please note that **all** methods and properties in VBScript++ are implemented as functions. They're all called by Eval() which requires them to be functions and not subs. Please note that they don't have to be functions that return values.

The first parameter to AddMethod provides the name this class will use to refer to the method, in this case Delete.

The second parameter is an anonymous Hash. We could use the key:vector to tell VBScript++ what function to call whenever the Delete method of class ClassFile is called. We call this the method vector. This is a VBScript version of function pointers. If the second parameter is NULL, as it is here, then ApplyMethod will generate the method vector by combining the name of the class, an underscore and then the method name. In our example, the method vector is ClassFile\_Delete. If we wanted the Delete method to instead vector to a function called ClassFile\_DeleteFile, our ApplyMethod statement would have looked like this:

classFile.ApplyMethod "Delete"**,** "vector=>ClassFile\_DeleteFile"

It is important to notice that the ClassFile\_Delete method takes two arguments, self and args. The first argument, self, is the BagInstance that has just vectored to this function. The args parameter is any other arguments that the programmer has elected to pass in. However, notice that in our example above, BagDictCreate is NOT called. It is actually called inside the object framework before vectoring to the method code, so args in this case is already a BagDict.

In addition, in the above example, you will notice self.VerifyHasKeys. This method verifies that the specified BagDict has the needed keys before continuing, and if it does not, issues an error message. This is important to do because in VBScript, and in particular in this object implementation, the runtime does not do most of the compiler time checking and verification that some languages provide for you.

**Important note: In QTP, the methods and property routines for BagObjects \*cannot\* be in the test file, they \*must\* be in a library file.**  I don't know why this is, but when you put the methods or properties in the test file, Exec() cannot call them.

### **Adding Properties to a BagClass**

Adding properties to a class is done mostly the same way as methods. In the ClassFile\_Delete method above, there is a call to a Exist property (If ClassFile\_Exist**(**self**,** args**)** Then). We can define this property as follows:

classFile.ApplyProp "Exist"**,"**get**",**"return\_type=>Boolean"

Function ClassFile\_Exist\_Get**(**self**,**args**)**

self.VerifyHasKeys "file\_name"**,** "ClassFile\_Exist fails missing key:file\_name"

ClassFile\_Exist **=** ClassFileSystemItem\_FSO.FileExists**(**self**(**"file\_name"**))**

End Function

It's important to note that using the direct method shown in the example above (If ClassFile\_Exist**(**self**,** args**)** Then) means that the inheritance mechanism is bypassed, if we overrode the mechanism for exist, the override would not be used. Sometimes, this is the behavior we want, but sometimes we also want to take the most recent definition, which we can do like so:

If self.Prop**(**"exist"**,** null**)** Then

...

When we create a property, please note that there are 3 arguments instead of two. In the example above, the second argument is **"**get**"**, this could be get, set or let. Though internally set and let are the same, and whether the return data is an object is handled automatically. So you only need 2 not 3 property handlers.

For Setters, a third value is passed in. This is the only exception to the two argument rule. The third parameter is the value to be used by the setter.

So for covering both bases, we'd do something like this:

myClass.ApplyProp "time"**,"**get**",** "return\_type=>Boolean"

Function myClass\_Time\_Get**(**self**,**args**)**

**...**

myClass.ApplyProp "time"**,"**set**",** NULL

Function myClass\_Time\_Set**(**self**,**args,value**)**

**...**

**Important note: In QTP, the methods and property routines for BagObjects \*cannot\* be in the test file, they \*must\* be in a library file.**  I don't know why this is, but when you put the methods or properties in the test file, Exec() cannot call them.

### **Creating BagInstances of a BagClass**

Now that the ClassFile BagClass has been created, it’s time to look at creating instances (BagInstances) of our new class. We create instances of a class by calling the classes static MakeObject method.

Dim myFile

Set myFile **=** classFile.MakeObject **(**"file\_name=>c:\foo.txt"**,** NULL**)**

That is all there is to it. We ask the class to make a new instance and we pass in any object data (in this case, file\_name) and metadata (in this case NULL). These are applied to the new BagInstance object before it is returned.

### **Calling methods and properties**

Now that we have an instance of the ClassFile class, we can call it’s methods using instance.Method(). The first parameter to Method() is the name of the method we want to call, and the second parameter is any arguments we want to pass in, in a format the BagDictCreate understands.

myFile.Method "Delete"**,** NULL

Methods can return values. If we modified Delete() to return a boolean about the success of the operation, we could do this:

myData = myFile.Method**(**"Delete"**,** NULL**)**

One important note here is that we do have to maintain the VBScript rules about parenthesis or not based on whether or not we're doing anything with the return value (VBScript doesn't actually care of it's a function, if you're not doing anything with the result, you must call it as if it were a Sub).

If we wanted to pass in an argument, we might do it like so:

myFile.Method "Delete"**,** "force=><Eval>True"

If we wanted to get or set a property, we might do it like so:

myVar = self.Prop**(**"date\_accessed"**,** null**)**

or:

self.Prop**(**"date\_accessed"**,** null**) =** myVar

### **How it's most commonly used**

I just wanted to post a little example here. In this case, the objects represent the records that the system under test is designed to manipulate. The UI manipulation happens inside the method call, so the test cases deal only with the data abstractions. This means that as the application under test changes, the libraries change, but the test cases do not.

'Support library

Dim ClassMember

ClassMember = ClassBase.MakeClass**(**"ClassMember"**,** NULL**,** NULL **)**

ClassMember.ApplyMethod "add",NULL

Function ClassMember\_Add**(**self, args**)**

'(function code here)

End Function

'test case code

Dim newMember

newMember = ClassMember.MakeObject **(**NULL**,** NULL**)**

newMember**(**"last\_name"**)** = "Smith"

newMember**(**"first\_name"**)** = "Bob"

newMember.Method "add",NULL

### **Custom Signatures**

Another feature common to most object based languages is polymorphism. This allows the compiler to look at different method calls with the same label and route them based on the types of the data passed in. So you might have an Add() method, that has two different signatures, and the compiler does the right thing with a call based on it's arguments. Add(myMember) might do something very different than Add(myMember, phoneNumber).

Because all of the method and property calls in VBScript++ are required to take only two arguments (self and args), to support the middleware properly, we can't change the behavior based on the number of arguments or the types of the arguments. It's also the case that VBScript++ is a loosely typed language, so typing might not be meaningful in any case.

However, we do have named keys as a way to distinguish between the arguments passed. We can create calls for different methods based on the passed keys. Like so:

myClass.ApplyMethod "Create", \_

"required\_keys=>company\_name|vector=>MyClass\_Create\_Company"

myClass.ApplyMethod "Create", \_

"required\_keys=>last\_name|vector=>MyClass\_Create\_Person"

And then, when we make our calls, we can do like so:

MyObject.Method "Create","company\_name=>foo" ' goes to MyClass\_Create\_Company

MyObject.Method "Create","last\_name=>Smith" ' goes to MyClass\_Create\_Person

Even clearer, we can do this:

Dim myRecord ' make container

BagDictCreate myRecord**,** NULL ' turn it into record

myRecord**(**"last\_name"**)** = "Smith" ' add field

myRecord**(**"first\_name"**)** = "Bob" ' add another field

MyObject.Method "Create",myRecord ' call method

### **Constructors and Destructors**

The framework supports constructors and destructors. Both take the same two arguments as the rest of the bag methods.

Default constructors are called <classname>\_Constructor. If your instance requires instantiation of an instance specific object, it should be created in the constructor, as any created at the class level will be shared amongst all the entities in the class.

Likewise, destructors are called <classname>\_Destroy.

### **Debugging**

There are two important points about debugging. The first is where you set your breakpoints. Because of the exec() call inside method and property calls, you must set your breakpoint in the target routine, you can't assume that you'll be able to step into it.

In the code, there's a special comment you can search for: JumpStation

This string happens just above where the vector call is made. If you set a breakpoint on the vector call statement just below this comment, you can check the value of the variable called stringCommand, and then go and set a breakpoint in that target routine.

The second point was something I stumbled across as I was developing this. I found I needed to be able to "see" into the objects when debugging. I could set watchers for each key, but sometimes I just wanted to see the whole thing at once. Toward that end, I created Debug().

myRecord.Debug()

This returns an array of all the keys and string approximations of their values. Metadata keys are prefixed with M: and object data keys are prefaced with O:

### **Other little amusing oddities**

One of the things I really like about this framework is that I can add a method to an instance, without creating an intermediate class. I can also create a class, create an instance of the class, pass that instance to a caller, and allow the creating class to go out of scope without impacting the instance that was passed back.

For all it's delights however, it also is a framework that must be used with care, because for the most part, it does not protect you from yourself.

*How to shoot yourself in the foot with VBScript:*

*Write a shoot.vbs virus and attach it to an e-mail. Two hundred million Microsoft Outlook users will infect their feet with it.*

*How to shoot yourself in the foot with VBScript++*

*With what?!?!*

## ***Summary***

I've seen some post processing ways to do this in VBScript, but I found them cumbersome to use and difficult to debug. This framework gives me the best of all worlds. I can debug with it, I can create classes and instances that reflect the abstraction structures that are relevant to the applications I'm testing, and most importantly, I can reuse almost all of what I do. Without shooting myself in the foot too much.

You're welcome to use this framework, subject to the GNU General Public License version 3 (GPLv3) (<http://www.opensource.org/licenses/gpl-3.0.html>). I am up for answering questions as time permits, but offer no promises about it's fitness for any particular purpose, nor do I commit to issuing bug fixes (though if you tell me of one, I probably will fix it).

## ***Attachments***

The download for this article contains one file (aggregations.logging.vbs), which is an aggregation of several files:

* ExtensionsBase.vbs: A general library of extensions to VBScript. Handy in lots of ways and is mostly needed for the VBScript++ implementation.
* ExtensionsStrings.vbs: A general library of string specific extensions to VBScript. Also handy.
* ExtensionsFiles.vbs: File related VBScript extensions.
* ExtensionsDates.vbs: Date string manipulation extensions.
* VBScript++.vbs: The actual VBScript++ implementation, including the BagDict class, a number of required support functions, and the basic objects we’ve defined so far.
* Logging.vbs: This file contains Log4VB, which is our limited implementation of Log4J. It does not include the configuration from XML capability, but does provide a useful example of using VBScript++.