Using a global dictionary for fast shared code access

As we have shown in the recipe *Using a global dictionary for fast shared data access*, it is possible to use a dictionary to store values of different types during runtime, and share them during the test flow with other actions at any level. We mentioned that a dictionary has the capacity to store any type of value, including objects. We further indicated that this opens the possibility to have nested dictionaries (albeit out of the scope of the current chapter).

In a similar fashion, it is possible to load pieces of code globally and hence grant shared access to all actions. In order to achieve this, we will recur to a well-known code design pattern, the command wrapper.

Getting ready

Refer to the *Getting ready* section of the *Using a global dictionary for fast shared data access* recipe. Basically, we can just add the code to the same function library and actions.

How to do it...

The first steps of defining the `GlobalDictionary` variable and the functions to manage its instantiation and disposal are identical, as in the recipe *Using a global dictionary for fast shared data access*, so we can just skip to the next step.

tion deserves special attention. In the
ary that we attached to the test, we will add the following pieces of code:

```
Class MyOperation1
    Function Run()
        Print typename(me) & " is now running..."
    End Function
End Class

Class MyOperation2
    Function Run()
        Print typename(me) & " is now running..."
    End Function
End Class

Function GetInstance(cls)
    Dim obj

    On error resume next
    Execute "set obj = new " & cls
    If err.number <> 0 Then
        reporter.ReportEvent micFail, "GetInstance", "Class " & cls & " i
        Set obj = nothing
    End If
    Set GetInstance = obj
End Function
```

The two classes follow the command wrapper design pattern. Note that they both contain a `Run` function (any name would do). This follows a pattern, which enables us to load an instance of each class and store it in our `GlobalDictionary` variable.

The `GetInstance(cls)` function acts as a generic constructor for our encapsulated functions. It is absolutely necessary to have such a constructor in the function library because UFT does not support instantiating classes with the operator `new` within an action. We use the `Execute` function to make the line of code, resulting from concatenating the command string with the `cls` parameter passed to the function, and hence, it can return an instance of any class contained in any other

associated function library. The function checks if an error occurs while trying to create a new instance of the given class. This could happen if the string naming the class is inaccurate. In such a case, the function returns nothing after reporting a failure to the test report. In such a case, we may wish to halt the test run altogether by using the `ExitTest` command.

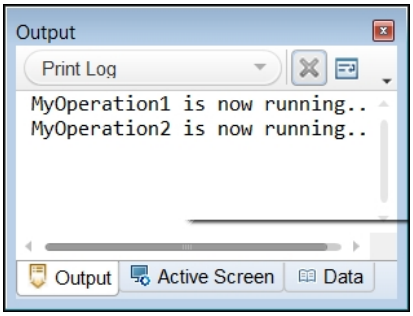
In `Action1`, we will add the following code:

```
GlobalDictionary.Add "Op1", GetInstance("MyOperation1")
GlobalDictionary.Add "Op2", GetInstance("MyOperation2")
```

In `Action2`, we will add the following code:

```
GlobalDictionary("Op1").Run
GlobalDictionary("Op2").Run
```

The output of the test is now as shown in the following screenshot:



How it works...

When you run this test, the initial process of `GlobalDictionary` instantiation is executed, as in the previous recipe. Then, we simply add two keys to the `GlobalDictionary` and assign a reference to each value to an instance of the command wrapper classes `MyOperation1` and `MyOperation2`. When the test flow reaches `Action2`, we access these instances by retrieving the items (or the values) we stored with the keys, and then have access to the classes' public methods, fields, and properties. The code line is as follows:

```
GlobalDictionary("Op1").Run
```

First, it retrieves the reference to the `MyOperation1` object, and then, it applies to the `Op1` operator to access the public `Run` method, which just prints the name of the class and a string.

There's more...

Of course, the `Run` method of the command wrapper pattern may need a variable number of arguments, because different functions meet different requirements. This can easily be resolved by defining the `Run` method as accepting one argument and passing a `Dictionary` object with the keys and values for each variable that is required.

For example, assuming that the `dic` argument is a dictionary:

```
Class MyOperation1
    Function Run(dic)
        Print typename(me) & " is now running..."
        Print dic("var1")
        Print dic("var2")
        Print typename(me) & " ended running..."
    End Function
End Class
```

Now, we would use the following code in `Action2` to call the `Run` method:

```
Set dic = CreateObject("Scripting.Dictionary")
dic.Add "var1", "Some value"
dic.Add "var2", "Some other value"
GlobalDictionary("Op1").Run
```

See also

Also refer to the *Using a global dictionary for fast shared data access* recipe in this chapter. We will also delve more in depth into the command wrapper design pattern in Chapter 7, *Using Classes*.