Building a test controller

In this recipe, we will see how to build a controller for our test automat framework. As outlined in the previous recipe, the controller will load t list of actions, and for each Action, it will import the corresponding datasheet. For each data-driven iteration, it will initialize the Action an invoke its `Run` method.

Most often, a controller is implemented as a function. Here, we will implement it as a class. The reason is that, this way, we can instantia a controller during runtime to support dynamic branching of the test f

Getting ready

Create a folder structure, as follows:

- `C:\Automation`
- `C:\Automation\Data`
- `C:\Automation\Lib`
- `C:\Automation\Tests`
- `C:\Automation\Config`
- `C:\Automation\Results`
- `C:\Automation\Solutions`

Create a new test and save it as `Framework_MasterDriver` under the subfolder `C:\Automation\Tests`. You can also save the solution under the `Solutions` subfolder. Under the `Data` subfolder, create a subfolder named `Framework_MasterDriver` and create an Excel file named `TestScenario.xls`.

Create a new function library. From the **File** menu, navigate to **New | Function Library...**, or use the *Alt* + *Shift* + *N* shortcut. Save the file as `cls.Controller.vbs`.

An Excel file named `TestScenario.xls` with a datasheet named `Steps` is required to be able to use the controller, as shown in the following example:

STEP_ID	ACTION_NAME	RUN	DATASHEET	ITERATIONS	ON_FAILURE
10	`OpenApp`	TRUE	N/A		`ExitTest`
20	`Login`	TRUE			`ExitTest`
30	`Search`	TRUE			`ExitAction`
40	`AddToCart`	TRUE		1-3	`ExitAction`
50	`Checkout`	TRUE			`ExitAction`
60	`Logout`	TRUE	N/A		`ExitAction`
70	`CloseApp`	TRUE	N/A		`ExitTest`

For each step that is data driven, the Excel file should include a specific datasheet named by the Action name (the name of the class that implements the action. Refer to the *Building a reusable component (action)* recipe). If the `ITERATIONS` parameter is left empty, then the controller will run only one iteration.

For example, for the `AddToCart` action, a datasheet named `AddToCart` is required, and we wish to run it three times as shown in the following datasheet example:

PRODUCT_NAME
My Book 1
My Book 2
My Book 3

It is possible to share datasheets with different actions, by specifying a `DATASHEET` value that is different from the Action name. If the Action is not data driven, then `N/A` should be entered.

How to do it...

Proceed with the following steps:

1. Add the following Environment variables to the test (of course, it would be most efficient to export these to an XML file to allow for reusability for all the tests):
    - `DATA_FOLDER` with the root path value of the folder in which the automation input data is stored. In our case it will be `C:\Automation\Data\`.
    - `ON_FAILURE` with the value of the action to be taken if a problem is found. It is used by the `ASSERT_RESULT` function. It's possible that the values are `ExitTest` and `ExitAction`.
2. In the controller function library, `cls.Controller.vbs`, write the following code:

```
Const C_STR_TEST_SCENARIO_XLS = "TestScenario.xls"
Const C_OBJ_OF_CLASS_MSG = "--- Object of Class "
Const C_OBJ_LOADED_MSG = " was loaded ---"
Const C_OBJ_UNLOADED_MSG = " was unloaded ---"
```

These constants are auxiliary, and they are used to log/report:

```
Class Controller
    Public Status
    Public Details

    Function Run(ByVal strTestSetsPathName)
        ' ----------------------------------------------------
        ' Function     :    Run
        ' Purpose      :    Runs the steps (procedures implemented as
        ' Args         :     ByVal strTestSetsPathName
        ' Returns      :     0 on success; 1 on failure
        ' ----------------------------------------------------
        ' Usage         :    Run("C:\Automation\Test_Sets\")
        ' Notes         :    1) Uses a Local DataSheet to control th
        '                    2) Uses GetClassInstance
        '                    3) Uses CNum
        '                    4) Uses ASSERT_RESULT
        '                    5) Uses GetIterations
        '                    6) Uses PrintReportInfo
        '                    7) Uses GetNormalizedStatus
        '                    8) Uses Timestamp
        ' ----------------------------------------------------
        Const C_STEPS_DATASHEET = "Steps"
        Dim iTestStatus, iStepStatus, iIterationStatus 'Statuses at

        Dim dt, rowcount 'Datasheet with the steps list
        Dim bExitAction, bExitTest, bRun, iStep, iter, oAction, sAc
        Dim arrIterations 'To support iterations
        Dim sFolder, sDatasheet 'For datasheet import
        ' ----------------------------------------------------

        ' ----------------------------------------------------
        '--- Get the name of the folder from which to import datash
        sFolder = Environment("TestName")
        '--- Add sheet
        DataTable.AddSheet(C_STEPS_DATASHEET)
        '--- Import steps datasheet
        Call DataTable.ImportSheet(strTestSetsPathName & "\" & sFol

        Set iTestStatus    = [As Num](0)
        Set dt             = DataTable.GetSheet(C_STEPS_DATASHEET)
        rowcount        = dt.GetRowCount
        bExitTest       = False

        PrintReportInfo "Test " & Environment("TestName"), "Startec
```

Until this point we had some initialization commands. Now, comes

the main `For` loop that manages the run session:

```
'--- Loop on all steps defined in the datasheet
       For iStep = 1 To rowcount
           bExitAction = False
           dt.SetCurrentRow(iStep)
           sActionName = dt.GetParameter("ACTION_NAME").Value
           bRun = dt.GetParameter("RUN").Value
```

Within the loop, we initialize the flag `bExitAction`, set the row in the `Steps` datasheet, and retrieve the name of the current action. We also get the value of the `RUN` parameter, which is used to check if the current Action is planned for execution.

```
'--- Check if the step is planned to be executed
           If CStr(bRun) = "TRUE" Then
             '--- Get an instance of the sActionName class
              ASSERT_RESULT(GetClassInstance(oAction, "[" & sActi
              '--- Reset Step status
              Set iStepStatus    = [As Num](0)
              '--- Assign Step id
              oAction.StepNum = dt.GetParameter("STEP_ID").Value
              '--- Get datasheet name to import (for data-driven
              sDatasheet = dt.GetParameter("DATASHEET").Value
              If Trim(sDatasheet) = "" Then
                  sDatasheet = sActionName
              End If

              '--- Check if the Action is data-driven
              If sDatasheet <> "N/A" Then
                  '--- Import datasheet to local
                  Call DataTable.ImportSheet(strTestSetsPathName
                  '--- Assign the new sheet to the step
                  Set oAction.dt = DataTable.LocalSheet
              End If
```

code uses the `ASSERT_RESULT` function to ensure that the ıested `Action` is valid (that is, the returned object by `ClassInstance` is not equal to `Nothing`). The `iStepStatus` able is initialized as a `CNum` object (a custom class that enables ct-oriented operations such as ++, and --), using the `[As Num]` hod, which acts as the `CNum` constructor. We then assign the current Action its number (or ID) from the `STEP_NUM` parameter, and if it is a data-driven action, we assign the Action its corresponding datasheet as well.

```
'--- Get list of iterations (e.g., "1-3,7,13-17") as System.Collect
              Set arrIterations = GetIterations(dt.GetParameter
                ("ITERATIONS").Value)
              arrIterations.Sort()
              '--- Reset iterations status
              Set iIterationStatus = [As Num](0)
              '--- Send start Step to the log
              PrintReportInfo "Step " & oAction.StepNum & " - Act
```

We then get the list of rows from which the Action will retrieve its input data. The number of items in the list determines the number of iterations in the Action. Take note that the list of rows can include a mix of single rows and ranges separated by commas. Next, we reach the inner `For` loop that controls the iterations flow for each action.

This will check if the Action is data driven, and if so, it sets the datasheet row for the current iteration and the Action's `Iteration` field. Then, it simply invokes the `Run` method of the Action and gets the status of the iteration.

Note that we use the `On Error Resume Next` directive just before invoking the action's `Run` method, in order to catch any exception and redirect it to `ErrorHandler` (refer to the *Building an event handler* recipe):

```
'--- Loop for each iteration
              For Each iter In arrIterations
                  PrintReportInfo "Step " & oAction.StepNum & " -
                  '--- Check if the Action is data-driven
                  If sDatasheet <> "N/A" Then
                      '--- Set the row that corresponds to the cu
                      oAction.dt.SetCurrentRow(iter)
                  End If
                  '--- Set the Iteration field of the Action
                  oAction.Iteration = iter
                  ' ----------------------------------------
                    '--- Execute the Action
                  ' ----------------------------------------
                  On Error Resume Next '--- Try
                  oAction.Run
                  ' ----------------------------------------
                  If Err.Number <> 0 Then 'Catch
                    me.ErrorHandler.RunMappedProcedure(Err.Numbe
                  End If
                  On Error Goto 0
                  ' ----------------------------------------
                  '--- Get the Action status
                  iIterationStatus.[+=]oAction.Status
```

Next, we send the result to the log. The `GetNormalizedStatus` function accepts an integer and checks if it represents success or failure. It is possible to customize such a function, depending on the requirements of the test automation framework. If the status is a failure, then we check with the `Eval` statement as to what we should do, as defined in the `ON_FAILURE` parameter.

For example, if `ExitAction` was set, then the next iteration of the Action will not be run, and the controller will attempt to execute the next Action (of course, one must ensure beforehand that the actions are independent). If the test flow cannot be continued, we can set the value of the `ON_FAILURE` parameter in the datasheet to `ExitTest`.

```
    '--- Send iteration result to the log
                PrintReportInfo "Step " & oAction.StepNum & " -
        '--- Check the status of the iteration
                If GetNormalizedStatus(iIterationStatus) > 0 Th
          '--- Evaluate if a failure condition occurred
                    Eval("b" & dt.GetParameter("ON_FAILURE") &
                    '--- Check the Exit flags
                    If bExitAction Then Exit For
                    If bExitTest Then Exit For
                End If
            Next '--- Iteration

      '--- Update the Step status with the iteration status
            iStepStatus.[+=]iIterationStatus
      '--- Send Action result (end) to the log
            PrintReportInfo "Step " & oAction.StepNum & " - Act
            '--- Dispose of the oAction object
            Set oAction = Nothing
```

If the Action is not planned to be executed, it is reported to the results so that the person analyzing them will be aware of this fact. If the Action `RUN` parameter is empty, then the controller will report that it was undefined.

```
        ElseIf CStr(bRun) = "FALSE" Then
                '--- Send skip Step to the log
                PrintReportInfo "Step " & dt.GetParameter("STEP
        Else
            '--- Send no directive for Step to the log
            PrintReportInfo "Step " & dt.GetParameter("STEP_ID"
        End If
```

Next, the `iTestStatus` variable will be updated with the status of the step (Action), which, as previously indicated, stores the accrued status of its iterations.

The `ExitTest` flag is checked, and if set, then the main `For` loop is terminated. The result is sent to the log again and returned by the `Run` function.

```
'--- Update the Test status with the iteration status
            iTestStatus.[+=]GetNormalizedStatus(iStepStatus)
            '--- Check the Exit flag
            If bExitTest Then Exit For
        Next '--- Step (Action)
        '--- Send Test result (end) to the log
        PrintReportInfo "Test " & Environment("TestName"), "Ended a
        '--- Return status
        Run = GetNormalizedStatus(iTestStatus)
    End Function
    ' ---------------------------------------------------------
    ' End: Run
    ' ---------------------------------------------------------
End Class
```

3. To use the controller, add the following function in the same library:

```
Function RunTest()
    Dim oTestRunner

    ASSERT_RESULT(GetClassInstance(oTestRunner, "Controller"))

    RunTest = oController.Run(Environment("DATA_FOLDER"))
End Function
```

4. The `RunTest` function uses the `GetClassInstance` function to get an instance of the controller. To use it, just write the following line of code in your test (`Action`):

```
ExitTest(RunTest())
```

When the `RunTest` function is invoked, the controller will roll the Actions as described, and its `Run` method will return the status of the test. Finally, the test will exit and the status will be returned.

How it works...

It is quite evident that a test automation framework implementing such a

design for the controller module covers most of the requirements for flow control, error handling, reporting, and data loading.