Building an event handler

One risk we must handle during run sessions is exceptions, as they c have a fatal impact on the robustness of our automated tests. UFT provides the recovery scenario as a built-in solution (refer to Chapter *Event and Exception Handling*), but it is quite complex to implement a may hinder the performance of the test.

In this recipe, we will see how to implement a simple recovery mechanism that is integrated with the controller (described previously and that utilizes the same design pattern used for the regular actions implement recovery procedures.

Getting ready

Add an Environment variable to the test. Name it `ERR_DEFAULT` and s its value to `StopRunSession`. Add another Environment variable nam `ERR_9` and set its value to `ClearError`. Create a function library nam `cls.EventHandler.vbs` in the `Lib` folder, as already described in the previous recipe.

How to do it...

The following code shows a sample procedure implemented as an Action. The `StopRunSession` class is used in our framework to handle an exception by stopping the run session, and it is used as default. Write the following code in the `cls.EventHandler.vbs` function library:

```
Class StopRunSession
    ' ----------------------------------------------------
    ' Reusable Action: StopRunSession
    ' Description: Stops the run in case of an unhandled error/exception
    ' ----------------------------------------------------
    Public Status
    Public Iteration
    Public StepNum
    Public dt
    Public Details

    Public Function Run()
        me.Details = "Ended with "
        me.Status.[=]Reporter.RunStatus

        '--- Report
        Call ReportActionStatus(me)

        '--- Stops the run session
        ExitTest(Reporter.RunStatus)
    End Function

    Private Sub Class_Initialize
        Call InfoClassInstance(me, C_OBJ_LOADED_MSG)
        Set me.Status = [As Num](0)
    End Sub
    Private Sub Class_Terminate
        Call InfoClassInstance(me, C_OBJ_UNLOADED_MSG)
        Set me.Status = Nothing
    End Sub
End Class
```

The procedure was built based on the same command wrapper pattern as the regular reusable Actions. The procedure will be invoked any time by the RunMappedProcedure method (shown in the following code snippet) of the EventHandler class, which will not find a matching procedure for a given error code.

Note that in this sample implementation, the value of the procedure associated with the error number is taken from the environment, but more elaborate design patterns could have been mapped into an XML file or DB:

```
Class EventHandler
    Function RunMappedProcedure(ByVal strError)
        Dim oProcedure

'--- Try to execute the procedure associated with the error (if exists)
        If GetClassInstance(oProcedure, Environment("ERR_" & CStr(Abs(str
            RunMappedProcedure = oProcedure.Run
            Exit Function
        End If
'--- Try to execute the default procedure to handle errors (if exists)
        If GetClassInstance(oProcedure, Environment("DEFAULT_ERROR_HANDLE
            RunMappedProcedure = oProcedure.Run
            Exit Function
        End If
    End Function
End Class
```

The following code shows another sample procedure implemented as an Action. It is used in our test automation framework to handle a specific exception by clearing the error, and the procedure is mapped to error code number `9` (`Subscript out of range`). This can be written in the `cls.EventHandler.vbs` function library file, as follows:

```
Class ClearError
    ' ----------------------------------------------------
    ' Reusable Action: ClearError
    ' Description: Clears the error in case of an unhandled error/excepti
    ' ----------------------------------------------------
    Public Status
    Public Iteration
    Public StepNum
    Public dt
    Public Details

    Public Function Run()
        me.Details = "Ended with "
        me.Status.[=]0

        '--- Report
        Call ReportActionStatus(me)

        '--- Clears the error
        Err.Clear
    End Function

    Private Sub Class_Initialize
        Call InfoClassInstance(me, C_OBJ_LOADED_MSG)
        Set me.Status = [As Num](0)
    End Sub
    Private Sub Class_Terminate
        Call InfoClassInstance(me, C_OBJ_UNLOADED_MSG)
        Set me.Status = Nothing
    End Sub
End Class
```

How it works...

When the controller tries to execute the Action, it sets a kind of try-catch mechanism with `On Error Resume Next`, as shown here:

```
' ---------------------------------------
'--- Execute the Action
' ---------------------------------------
On Error Resume Next
'--- Try
oAction.Run
' ---------------------------------------
If Err.Number <> 0 Then 'Catch
  me.ErrorHandler.RunMappedProcedure(Err.Number)
End If
On Error Goto 0
```

So if an error occurs, it will be passed to `ErrorHandler` via the `RunMappedProcedure` method, and it will use either the specifically defined procedure for the error or the default procedure. This ensures that no exceptions will be left unhandled.