Chapter No 1
"Data-driven Tests"
In this package, you will find:

The author’s biography

A preview chapter from the book, Chapter no.1 "Data-driven Tests"

A synopsis of the book’s content

Information on where to buy this book

About the Author’s

Meir Bar-Tal holds a Master's Degree in Cognitive Psychology from the Ben-Gurion University of the Negev, but he made a swift switch to the software industry in 2000 and is currently an independent test automation architect. In 2007, Meir was one of the cofounders of the popular knowledge sharing site www.advancedqtp.com (originally founded as a personal blog by Yaron Assa) and has been its Editor in Chief and Forum Administrator ever since (sole owner since 2011). The site’s forums were among the final four candidates at the Automated Testing Institute Awards several times, and once were second only to the renowned SQA Forums. Apart from the materials he publishes on his site from time to time, Meir is a regular contributor to several professional online groups and forums and also conducts lectures on UFT at Ness IT Business College (Israel) and other institutions. In 2008, he joined Yaron Assa and others in establishing a small consultancy firm, SOLMAR Knowledge Networks Ltd., which was active until early 2011, when the partners decided to go their separate paths. Since then, Meir worked as an independent freelancer and became a SmartBear Software Authorized Provider. In 2011, he developed a QTP plugin for SeeTest, which enabled interoperability and IntelliSense. In 2014, he joined UGenTech Ltd. as the Associate Director of Automation.

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
Meir has been involved in many projects characterized by a wide array of technologies (COM, Unix, Windows, Web, .NET, Win Forms, WPF, Java, C#, and so on) and business industries (Derivatives, Banking, Medical, Storage, CRM, Billing, VOIP, and so on). The range of services he provides is wide and includes consultancy, project management, design and development, training and coaching, tools evaluation, and extensibility and plugins development.

Meir has provided services to firms such as HP Software, Experitest, Omnisys, IBM XIV, Hermes Logistics, Bank Leumi, YIT, Ginger Software, and Mazor Robotics. Before he co-founded Solmar in 2008, he worked for several companies, including dbMotion, Type Reader, Amdocs, Matrix, and ultimately, Super Derivatives. At Super Derivatives, he led a team of QTP developers to implement an object-oriented framework for QTP, a point that reflects his special interest in the design and development of frameworks and enthusiasm to share the fruits of his research and experience with others. In 2013-2014, he was, as a subcontractor, technical lead of a challenging project at HP Software, in which UFT was used to automate end-to-end scenarios for HP ALM with great success. Besides this, he works in close cooperation with HP Software R&D and periodically contributes his insights to improve UFT.

It would not have been possible to accomplish this piece of work without the assistance (and patience) of several people to whom I wish to express my gratitude and deep appreciation. First and foremost, to my late father who, by his own example, taught me that giving up is not an option; second, to my beloved daughters who always give me strength to accomplish more than what seems feasible; third, to my mother, who has always loved and supported, though not understood, me and all my endeavors; fourth, to my reviewers and editors, who contributed to me personally and to the final quality of this book.

Last, but not least, to all members of Packt Publishing, for giving me this wonderful, exciting, and challenging opportunity to share my knowledge. I wish to extend my special appreciation to Anila Vincent, for her understanding of the difficulties I faced during this project and her virtually unlimited patience, which unfortunately, I was forced to put to the test more than once.

Jonathon Lee Wright has over 15 years of international automation experience with a number of global organizations, including Lehman Brothers, Hitachi Consulting, Siemens, Thomson Reuters, New Zealand Lotteries Commission, PlanIT (Sydney), Unisys (iPSL), Xerox (BJSS), Hewlett Packard (Enterprise), and Microsoft (ALM). Currently, he provides enterprise wide Portfolio Lifecycle Management at Deutsche Bank.

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
as part of the test automation transition initiative, targeting 2,700 applications for test automation across the bank’s global digital enterprise landscape/ecosystem.

Jonathon also contributed to the best-selling book Experiences of Test Automation: Case Studies of Software Test Automation, Dorothy Graham and Mark Fewster, Addison Wesley, and a number of books on Testing as a Service (TaaS) models (epistemic and systemic entropy) and API testing in the cloud (service and network virtualization). He is the founder of Test Automation as a Service (TaaS.net) and Automation Development Services (automation.org.uk) and the Hybrid approach pattern (2004). He has also presented at various international testing conferences, such as Gartner (London), STARWest (California), Fusion (Sydney), ANZTB (Melbourne), EuroSTAR (Gothenburg and Dublin), BCS SIGIST (London). Further details about Jonathon can be found at www.linkedin.com/in/automation.

I would like to thank my father John Wright, without whose support and encouragement none of this would have been possible, for providing the solid foundation for my success over the years. Then, I would like to thank everyone at Packt Publishing, especially Shweta Pant for her support during the technical editing.

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
Advanced UFT 12 for Test Engineers Cookbook

Unlock the full potential of Unified Functional Testing (UFT) 12 with the introduction of new features and functionality. Learn the industry's best kept secrets, enhancing toolset capabilities like you never thought possible. Whether you are a casual user of UFT or an advanced power user searching for new automation design patterns to supercharge your existing framework, look no further as this is the book for you.

Join the authors who, with three decades of automation expertise between them, are ready to share with you ways to make your tests more relevant, effective, maintainable, efficient, manageable, portable, and reliable. This book is designed to be an invaluable source of reference for everyone with its clear and powerful coding examples. Why waste any more time trying to reinvent the wheel? Instead, accelerate straight to the expert level in UFT today!

What This Book Covers

Chapter 1, Data-driven Tests, covers the design patterns required to truly unlock the DataTable functionalities (create, retrieve, store, import, and export).

Chapter 2, Testing Web Pages, covers the design patterns required to manage modern-day browser capabilities and the challenges: handling broken links, websites' downtime, multiple browser instances, cookies, unexpected pop-ups, downloading and uploading files, synchronization, and most importantly, object identification and checking dynamic content through the DOM.

Chapter 3, Testing XML and Database, covers the design patterns required to manage DB connections, executing SQL statements, and a walkthrough of advanced DB and XML checkpoints.

Chapter 4, Method Overriding, covers the design patterns required to override a Test Object method, enriching basic functions, and adding exception handling mechanisms.

Chapter 5, Object Identification, covers the design patterns required to effectively manage the object identification process through techniques such as Inline Descriptive Programming, Description Object, ChildObjects, and native properties.

Chapter 6, Event and Exception Handling, covers the design patterns required to provide robust and maintainable tests that can deal with unexpected events or exceptions by catching errors inside a function or subroutine, recovery scenarios, and how to use the global dictionary for recovery.

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
Chapter 7, Using Classes, covers the design patterns required to implement classes in VBScript, along with illustrative examples and implementing function pointers.

Chapter 8, Utility and Reserved Objects, covers the design patterns required to utilize advanced functionality hidden within the depths of UFT.

Chapter 9, Windows Script Host, covers the design patterns surrounding the underlying infrastructure provided by the platform to execute scripts written in a variety of programming languages.

Chapter 10, Frameworks, covers the design patterns to implement modern-day test automation frameworks exploring modular-driven, data-driven, keyword-driven, model-driven, and hybrid techniques to find the best approach that works for you.

Appendix, Design Patterns, covers the additional design patterns, including auxiliary classes and functions, to enhance the tool set capabilities and unlock the full potential of UFT. It provides powerful examples for both the action and runtime data patterns to put into practice what has been covered in the previous chapters.
1 Data-driven Tests

In this chapter, we will cover the following recipes:

- Creating a DataTable parameter
- Retrieving data from a DataTable
- Storing data in a DataTable
- Importing an Excel file to a test
- Exporting a DataTable
- Parameterizing Test Object properties
- Defining test cases using a DataTable
- Storing data in the Environment object
- Retrieving data from the Environment object
- Reading values from an INI file
- Using a configuration file to manage test environments
- Using a global dictionary for fast shared data access
- Using a global dictionary for fast shared code access

Introduction

This chapter describes several ways by which data can be used to drive automated tests in UFT. Data-driven tests enable us to cover different paths in a test flow, by supplying a coded script with different sets of values to its parameters. These include input data for manipulating GUI objects and, where relevant, also the expected output from the application under test. In other words, a data-driven script is one whose behavior changes when fed with different sets of input data.

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
We can retrieve input data using the global DataTable object. The first seven recipes explain how we can work with a DataTable to attain various goals related to the concept of data-driven tests. The next two recipes deal with Environment variables using the Environment object. The Reading values from an INI file and Using a configuration file to manage test environments recipes show how to retrieve values from INI files and how to manage test environments with them. Finally, the Using a global dictionary for fast shared data access and Using a global dictionary for fast shared code access recipes describe advanced techniques for fast shared data and code access using a Dictionary object.

Creating a DataTable parameter

DataTable is a UFT object that acts as a wrapper to an MS Excel file, and its scope is global. This means that it can be accessed from any action within a test, as well as from function libraries that were attached to the test. When you create a new test or open an existing UFT test, you will notice that the DataTable pane will always show a global and local datasheet, one for each existing action within the test. In this section, we will see how to create a DataTable parameter.

How to do it...

Perform the following steps to create the DataTable parameter LocalParam1 for the local sheet:

1. From the File menu, navigate to New | Test or use the Ctrl + N shortcut. When a new test dialog opens, choose GUI Test and then click on the Create button.

2. We will create a DataTable parameter in the Action1 local sheet from the UFT data pane by double-clicking on the column header and entering the parameter name LocalParam1 in the dialog that opens, as shown in the following screenshot:

Similarly, for the test global sheet we will create a parameter named GlobalParam1.
3. Next, we need to enter our input data in the remaining cells of the parameter column in the global or local sheet, according to the requirements.

How it works...

If we open the Default.xls file in the test folder (which, as its name suggests, is the default data source for a new test), we will notice that there are two worksheets, namely, Global and Action1. In each of these, the first row holds the name of the parameters, so we will see GlobalParam1 in the Global worksheet and LocalParam1 in the Action1 worksheet. You will also notice that the used rows have borders at the bottom of the worksheet (the borders have no real function; UFT identifies the used range by the number of used rows and columns based on the content range).

See also

For information about setting and retrieving values for a DataTable parameter, refer to the next two recipes, Retrieving data from a DataTable and Storing data in a DataTable.

Retrieving data from a DataTable

DataTable is a UFT object that acts as a wrapper to an MS Excel file, and its scope is global. This means that it can be accessed from any action within a test, as well as from function libraries that were attached to the test. When you create a new test or open an existing UFT test, you will notice that the DataTable pane will always show a global datasheet and a local one for each existing action within the test.

Getting ready

Prior to getting started with this recipe, please ensure that you have followed the Creating a DataTable parameter recipe.

How to do it...

We will retrieve the value of a DataTable parameter, namely, LocalParam1, from the Action1 local sheet with the following code written in the code editor inside Action1:

```vbscript
Dim MyLocalParam

MyLocalParam = DataTable.Value("LocalParam1", dtLocalSheet)
Print MyLocalParam
```

For More Information: www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
Similarly, the following code snippet shows how to retrieve the value of a DataTable parameter from the test global sheet:

```vbnet
Dim MyGlobalParam
MyGlobalParam = DataTable("GlobalParam1", dtGlobalSheet)
' We can omit the explicit .Value property as given above since it is the default property
Print MyGlobalParam
MyGlobalParam = DataTable("GlobalParam1")
' We can omit the second parameter as given above (dtGlobalSheet) since the Global sheet is the default
Print MyGlobalParam
```

The result of this code in UFT's console is as follows:

![Output](image)

Of course, we need to ensure beforehand that the parameter exists in the DataTable class as outlined in the previous *Creating a DataTable parameter* recipe.

**How it works...**

By using the `DataTable.Value` property we are referring to the column by the parameter name in the underlying Excel worksheet (be it global or local):

```vbnet
MyLocalParam = DataTable.Value("LocalParam1", dtLocalSheet)
MyGlobalParam = DataTable("GlobalParam1", dtGlobalSheet)
```

For More Information:

www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
As we entered just a single value into the datasheet, the command retrieves just the value in the first row. If multiple values were entered and action iterations were set to run on all rows, then it would have retrieved the values from each row with each iteration.

The dtLocalSheet constant always refers to the datasheet by the name of the current action. The dtGlobalSheet constant always refers to the global datasheet and can be used in any action.

Storing data in a DataTable

Sometimes, data that is collected during a run session might be needed for later use. For example, suppose that Application Under Test (AUT) is a mobile operator management system. We could begin by executing a customer creation process, during which a customer ID is assigned automatically by the system. We then proceed with the other operations, such as selecting a phone number, an IMEI, credit card details, and so on. Later, we may wish to retrieve the customer records and update some personal data such as the mailing address. For this purpose, we will keep the customer ID in the global datasheet, so that any action that is executed, which can be referenced later (for example, one that performs a customer search), will have access to the data.

Data stored in the global datasheet is effective only until the test stops.

To see how to save data persistently for later run sessions, please refer to the Exporting a DataTable and Importing an Excel file to a test recipes.

How to do it...

Proceed with the following steps:

1. From the File menu, select New | Test or use the Ctrl + N shortcut. When the new test dialog opens, choose GUI Test and click on the Create button.

2. We will save the value of a DataTable parameter, CustomerID, to the global sheet with the following code written in the code editor inside Action1:

   Dim CustomerID

   DataTable.GlobalSheet.AddParameter "CustomerID", "990011234"
   CustomerID = DataTable("CustomerID")
   Print Environment("ActionName") & ": " & CustomerID

For More Information:

www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
3. To retrieve the value from another action, we will now create a new action datasheet. In the code editor, right-click on the next empty line and select **Action | Call to New Action**, as shown in the following screenshot:

The following dialog will open:

![Insert Call to New Action dialog]

For More Information:
[www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook](http://www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook)
4. Leave all the fields with the default values and click on **OK**. You will see that a new action named **Action2** will appear in the **Solution Explorer** window and open on the code editor's MDI region:

![Solution Explorer](image)

5. Now, we will retrieve the value of the **CustomerID** parameter from the global sheet with the following code inside **Action2**:

```vbnet
Dim CustomerID

CustomerID = DataTable("CustomerID")
Print Environment("ActionName") & ": " & CustomerID
```

The result of this code in the UFT's console is shown in the following screenshot:

![Output](image)

**How it works...**

When we run the test, UFT first executes **Action1**, and a new parameter named **CustomerID** will be added to **GlobalSheet** (a property of the **DataTable** object that refers to the **GlobalSheet** object) with the value given by the second parameter.

```vbnet
DataTable.GlobalSheet.AddParameter "CustomerID", "990011234"
```

We then immediately assign a variable with the retrieved value and print it to the console (for illustration purposes, we also concatenate the current action's name from the **Environment** object's built-in variables).

```vbnet
CustomerID = DataTable("CustomerID")
Print Environment("ActionName") & ": " & CustomerID
```

Next, UFT executes **Action2** same as **Action1**.

---

**For More Information:**

[www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook](http://www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook)
There's more...

There are other alternative ways of keeping and sharing data during a run session. The simplest is by using public variables declared in a function library attached to the test. The disadvantage of this approach is that these variables must be declared in advance and they are hard coded, but the nature of automation often demands more flexibility to manage such data.

See also

For information on advanced methods to share data among sessions, refer to the Using a global dictionary for fast shared data access recipe.

Importing an Excel file to a test

We can dynamically set the underlying Excel file that will serve as a data source for the whole run session, though this is probably a very rare case, and even switch between such files during the run session. It is possible to import a whole Excel workbook for a test or just a single worksheet for a specific action.

The classical case of importing an Excel file to a test is when the same flow needs to be executed on different environments, such as with multilingual systems. In such a case, the test would require an external parameter to identify the environment, and then load the correct Excel file. Another possibility is that the test identifies the language dynamically, for example, by retrieving the runtime property value of a Test Object (TO), which indicates the current language, or by retrieving the lang attribute of a web page or element.

Getting ready

Ensure that a new test is open and create a new action. Ensure that an external Excel sheet exists with one global worksheet and worksheets named after each action in the test. The Excel sheet will contain three worksheets, namely, Global, Action1, and Action2. The Action2 worksheet will contain data shown in the following screenshot. In our example, we will use the Excel sheet named MyDynamicallyLoadedExcel.xls, and to simplify matters, we will put it under the same test folder (it should be placed in a separate shared folder):
In the Flow pane, make sure that the Action Call properties are set to Run on all rows.

How to do it...

In order to load the MyDynamicallyLoadedExcel.xls file to the test, perform the following steps:

1. We use the DataTable.Import method to load the Excel sheet. In Action1 (the first to be run), we use the following code to ensure that the Excel file is loaded only once (to avoid loading Excel for each iteration in case the test is set to Run on all rows):

   ```
   if cint(Environment("TestIteration")) = 1 then
   DataTable.Import("MyDynamicallyLoadedExcel.xls")
   end if
   ```

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
2. In **Action2**, we use the following code to retrieve the values for all parameters defined in the local datasheet for **Action2**. We first print the number of the current action iteration, so that we may distinguish between the outputs in the console.

```vbscript
Print Environment("ActionName") & " - Action Iteration #" & Environment("ActionIteration")
For p = 1 to DataTable.LocalSheet.GetParameterCount
    print DataTable.LocalSheet.GetParameter(p)
Next
```

3. When a test is set to **Run on all rows**, it means that it will be executed repeatedly for each row having data in **GlobalSheet**.

The output to the console looks like the following screenshot:

![Output Screenshot](image)

**How it works...**

In **Action1**, the `DataTable.Import` method replaces `Default.xls` with the target Excel file. The code in **Action2** retrieves and prints the values for each parameter, and as the action was set to **Run on all rows**, the code repeats this for all rows with data.

---

For More Information:

[www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook](http://www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook)
There's more...

To import just a worksheet for an action, use the `DataTable.ImportSheet` method as follows:

```csharp
DataTable.ImportSheet("MyDynamicallyLoadedExcel.xls", "Action1", "Action1")
```

Here, the first parameter is the Excel filename and the last two are the source datasheet and target datasheet respectively.

See also

For information on saving values collected during a run session, refer to the next recipe, *Exporting a DataTable*.

Exporting a DataTable

We may need to save data that is collected during a run session. For example, a comparison of the current result with previous results might be required. Alternatively, we might wish to update the expected results. In such scenarios, we can save the data to an external Excel sheet for later use.

**How to do it...**

To save the `DataTable` in its current state before the run session ends, we will use the `DataTable.Export` method, which takes the path and name of an Excel file as an argument. There are two options to save the data table:

- Using a hardcoded filename:
  ```csharp
  DataTable.Export(Environment("TestDir") & 
                   "\MyDynamicallySavedExcel.xls")
  ```

- Using a variable filename:
  ```csharp
  DataTable.Export(Environment("TestDir") & "\" & strFileName & ".xls")
  ```

For More Information:

www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
Data-driven Tests

How it works...

The preceding statement saves the current contents of the DataTable (all worksheets) to a new Excel file (if not existent, otherwise it is overwritten). The statement `Environment("TestDir")` returns a string with the path of the current test to which a string with the name of the file we wish to create is concatenated (TestDir is one of the built-in Environment variables covered in detail later in this chapter).

There’s more...

To export just a single worksheet (in our example, the global sheet) for an action, use the `DataTable.ExportSheet` method, as follows:

```vbscript
call DataTable.ExportSheet(Envirronment("TestDir") & 
"\MyDynamicallySavedSheet1.xls", "Global")
```

Here, the first parameter is the Excel filename and the second is the source datasheet. The target datasheet will take the same name as the source.

Parameterizing Test Object properties

The same TOs might appear in different flavors. For example, a Submit button in a localized English version would show the text Einreichen for the German version. If the objects were given abstract IDs not related to their function or displayed text, then life for the automation developer would be easy; however, this is generally not the case. As managing a separate object repository for each language would pose a maintenance impasse for any automation project, a more practical approach should be adopted. A good alternative is to store the values for identification properties that change from one environment to another in an external data source, which is loaded at the beginning of a run session according to the required language. This recipe will show how to parameterize an identification property for a given TO and use the dynamically loaded value to identify the object during runtime.

How to do it...

In this recipe, we will take the Google+ Sign In button, which carries a different text value for each localized version. In our example, we will learn about the button in one language (Afrikaans), store the TO in a single Object Repository (OR), and make it get the value for its name attribute from an Environment variable named Google_Sign_In. The Environment variables for the test will be loaded according to the application language (refer to the Using global variables (Environment) recipe in Chapter 8, Utility and Reserved Objects).

For More Information:

www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
Proceed with the following steps:

1. With UFT already up, the Web add-in loaded and the test using it, navigate to **File | Settings** and then open Internet Explorer. Now, navigate to the Google+ sign-in page.

2. At the bottom of the page, there is a list from which one can select a language; select **Afrikaans**.

3. In UFT, navigate to **Resources | Object Repository...** (or use the **Ctrl + R** shortcut), as shown in the following screenshot:

4. Click on the + icon bearing the tooltip **Add Objects to Local**, select the sign in button with the mouse icon, and then add it to the OR.
5. Click on the text to the right of the name property, and then click on the icon that appears to the right, as shown:

![Test object details](image1)

6. In the **Value Configuration Options** dialog that opens, select **Parameter**. Now, from the drop-down list in the **Parameter** field, select **Environment**, as shown in the following screenshot:

![Value Configuration Options](image2)

7. In the **name** field (combobox), type `<Google_Sign_In>`. Leave the value as is and click on **OK**.

---

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
8. In OR, you will now notice that the value of the `name` property has changed to `<Google_Sign_in>` and the icon on the left represents the `Environment` object.

When running the test, OR will take the value of the `name` property from the `Environment` variable. Hence, if we have a set of such values for each language, then we will be able to test an application in whichever language we choose, without having to change a single line of script code!

As with this method, the value is stored as an internal `Environment` variable, and we wish to load the values according to the language interface under test. We need to export the environment to an external XML file and load it at the beginning of the run session.

**There's more...**

The same basic approach can be used for object identification with the Using Descriptive Programming inline recipe of Chapter 5, Object Identification.
See also

The Importing an Excel file to a test and Using a configuration file to manage test environments recipes.

Defining test cases using a DataTable

As mentioned earlier, a data-driven test is one that is designed to behave as required by different sets of parameter values. Basically, such sets of values actually represent different test cases. When executing a login test action, for example, valid or invalid values for the username and the password will trigger different application responses. Of course, the best is to have a single action (or function) that will handle all cases, with the flow branching according to the input data.

Getting ready

Ensure that you have the Flight Reservation sample application shipped with the installed UFT. You can check this by navigating to Start | All Programs | HP Software | Unified Functional Testing | Sample Applications. You should have a shortcut named Flight GUI that launches flight4a.exe. Create a new test by navigating to File | New | Test from the menu, or by using the Ctrl + N keyboard shortcut. Rename Action1 to FR_Login (optional).

How to do it...

Proceed with the following steps:

1. In the DataTable, select the FR_Login (or Action1 if you decided not to rename it) datasheet. Create the following parameters in the DataTable (as described in the Creating a DataTable parameter recipe):
   - TC_ID
   - Agent
   - Password
   - Button
   - Message1
   - Message2
   - Description
2. We will derive the test cases with reference to the system requirements, as we know (for this example, we will ignore the Cancel and Help buttons):

- The correct login password is always mercury. A wrong password triggers an appropriate message.
- The agent name must be at least four characters long. If shorter, the application prompts the user with an appropriate message.
- An empty agent name triggers an appropriate message.
- An empty password triggers an appropriate message.
- After four consecutive failed login attempts with a wrong password, the application prompts the user with an appropriate message and then closes.

Accordingly, we will enter the following data to represent the test cases:

<table>
<thead>
<tr>
<th>#</th>
<th>TC_ID</th>
<th>Agent</th>
<th>Password</th>
<th>Button</th>
<th>Message1</th>
<th>Message2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AgentEmpty</td>
<td></td>
<td>mercury</td>
<td>OK</td>
<td>Please enter agent name</td>
<td>Empty agent</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>AgentLT4</td>
<td>Mer</td>
<td>mercury</td>
<td>OK</td>
<td>Agent name must be at least 4 characters long</td>
<td>Agent with less than 4 characters</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Agent4EmptyPass</td>
<td>Merc</td>
<td></td>
<td>OK</td>
<td>Please enter password</td>
<td>Wrong password #1 (empty)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Agent4WrongPass</td>
<td>Merc</td>
<td>Merc</td>
<td>OK</td>
<td>Incorrect password. Please try again</td>
<td>Wrong password #2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Agent4WrongPass</td>
<td>Merc</td>
<td>1234</td>
<td>OK</td>
<td>Incorrect password. Please try again</td>
<td>Wrong password #3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Agent4WrongPass</td>
<td>Merc</td>
<td>Gfrgfh</td>
<td>OK</td>
<td>Incorrect password. Please try again</td>
<td>Login unsuccessful. Please try again later.</td>
<td>Wrong password #4; App closes</td>
</tr>
<tr>
<td>7</td>
<td>SuccessfulLogin</td>
<td>mercury</td>
<td>mercury</td>
<td>OK</td>
<td></td>
<td>Correct username and password</td>
<td></td>
</tr>
</tbody>
</table>
3. Apart from learning the TOs for the login and the message dialogs, create two checkpoints for the messages that appear after unsuccessful logins (one for the first and the other for the second type mentioned in the preceding table), and name them Message1 and Message2 respectively.

OR should contain the following TOs (smart identification should be turned off):

- Dialog: Login (parent: none, description: text=Login, nativeclass=#32770, is owned window=False, is child window=False)
- WinEdit: Agent Name (parent: Dialog Login, description: nativeclass=Edit, attached text=Agent Name)
- WinEdit: Password (parent: Dialog Login, description: nativeclass=Edit, attached text=Password)
- WinButton: OK (parent: Dialog Login, description: text=OK, nativeclass=Button)
- Dialog: Flight Reservations (parent: Dialog Login, description: text=Flight Reservations, nativeclass=#32770, is owned window=True, is child window=False)
- Static: Message (parent: Dialog Flight Reservations, description: window id=65535, nativeclass=Static)
- WinButton: OK (parent: Dialog Flight Reservations, description: text=OK, nativeclass=Button)
- Window: Flight Reservation (parent: none, description: regexpwndtitle=Flight Reservation, regexpwndclass=Afx:, is owned window=False, is child window=False)
- WinButton: Delete Order (parent: Window Flight Reservation, description: text=&Delete Order, nativeclass=Button)
- WinButton: Insert Order (parent: Window Flight Reservation, description: text=&Insert Order, nativeclass=Button)
- WinButton: FLIGHT (parent: Window Flight Reservation, description: text=FLIGHT, nativeclass=Button)
- WinRadioButton: First (parent: Window Flight Reservation, description: text=First, nativeclass=Button)
OR should contain the following Checkpoint objects:

- **Message1** and **Message2**: These checkpoints identify the static text appearing in the message that opens after a failed attempt to log in. The checkpoints should verify the `enabled=True` and `text=LocalSheet DataTable` parameters for **Message1** and **Message2** respectively.

- **Flight Reservation**: This checkpoint verifies that the main window opens with the properties `enabled=True` and `text=title=Flight Reservation`.

- **Delete Order, Insert Order, and Update Order**: All three checkpoints should verify that the buttons have the `enabled=False` and `text` properties set while opening the main application window set as their learned text property with the ampersand character (`&`) in the beginning of the string.

- **First**: This checkpoint for the WinRadioButton should verify that upon opening the main application window, the properties `enabled=False` and `checked=OFF` are set.

4. In FR_Login (Action1), write the following code:

```vbs
'Checks if either the Login or the Main window is already open
Function appNotOpen()
    appNotOpen = true
    If Dialog("Login").Exist(0) or Window("Flight Reservation").Exist(0) Then
        appNotOpen = false
    End If
End Function

'Opens the application if not already open
Function openApp()
    If appNotOpen() Then
        SystemUtil.Run "C:\Program Files\HP\Unified Functional Testing\samples\flight\app\flight4a.exe","","C:\Program Files\HP\Unified Functional Testing\samples\flight\app\","
        openApp = Dialog("Login").WaitProperty("enabled", 1, 5000)
    Else
```

For More Information:
[www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook](http://www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook)
Data-driven Tests

openApp = true
End If
End function

'Handles the Login dialog: Enters the Agent Name and the Password and clicks on the OK button
Function login(agentName, password, button)
with Dialog("Login")
  .WinEdit("Agent Name").Set agentName
  .WinEdit("Password").SetSecure password
  .WinButton(button).Click
If .exist(0) Then
  login = false
else
  login = true
End If
End with
End Function

'Performs a standard checkpoint on a message open by the FR application
Function checkMessage(id)
  If Dialog("Login").Dialog("Flight Reservations").Exist(0) Then
    checkMessage = Dialog("Login").Dialog("Flight Reservations").Static("Message").Check(CheckPoint("Message&id"))
    Dialog("Login").Dialog("Flight Reservations").WinButton("OK").Click
  else
    checkMessage = false
  End if
End Function

'Performs several standard checkpoints on the Main window and on several of its child objects
' to verify its initial state
function verifyMainWndInitialState()
  with Window("Flight Reservation")
    if .Check(CheckPoint("Flight Reservation")) then
      .WinButton("FLIGHT").Check CheckPoint("FLIGHT")
      .WinRadioButton("First").Check
      CheckPoint("First")
  end with
End Function

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
.WinButton("Update Order").Check
CheckPoint("Update Order")
.WinButton("Delete Order").Check
CheckPoint("Delete Order")
.WinButton("Insert Order").Check
CheckPoint("Insert Order")
End if
End with
End function

'Variables declaration and initialization
Dim agentName, password, button
agentName = DataTable("AgentName", dtLocalSheet)
password = DataTable("Password", dtLocalSheet)
button = DataTable("Button", dtLocalSheet)

'Tries to open the application
If not openApp() Then
    ExitTest
End If

'Tries to login with the input data
if not login(agentName, password, button) Then
    'Checks if a warning/error message opened, if it's correct in context and closes it
    if checkMessage("1") then
        'Checks if a second warning/error message opened, if it's correct in context and closes it
        if checkMessage("2") then
            If not Dialog("Login").Exist(0) Then
Application closed."
            'If a second message opened, then the number of login trials was exceeded and the application closed, so we need to reopen the application
            call openApp()
        End If
    End if
End if
else
    call verifyMainWndInitialState()
End if 'Tries to login
How it works...

Now, we will explain the flow of the FR_Login action and the local functions.

We declare the variables that we need for the Login operation, namely, AgentName, Password, and Button. We then initialize them by retrieving their values from the local sheet in the DataTable. The button value is parameterized to enable further elaboration of the code to incorporate the cases of clicking on the Cancel and Help buttons.

Next, we call the openApp() function and check the returned value. If it is False, then the Flight Reservation application did not open, and therefore we exit the test.

We attempt to log in and pass the AgentName, Password, and Button parameters to the function. If it returns true, then login was successful and the else block of code is executed where we call the verifyMainWndInitialState() function to assert that the main window opened as expected.

If the login did not succeed, we check the first message with a checkpoint that compares the actual text with the text recorded in the DataTable, which is correct in the context of the planned flow.

If the first message check passes, then we check to see if there is another message. Of course, we could have used a counter for the actual password failures to see if the second message is shown exactly by the fourth attempt. However, as we set the input data, the flow is planned such that it must appear at the right time. This is the true sense of defining test cases with input data. If a message appears, then the checkMessage(id) function closes the message box. We then check if the login dialog box is closed with the code If not Dialog("Login").Exist(0) Then, and it then calls openApp() to begin again for the last iteration.

In the last iteration, with the input data on the seventh row (refer to the table in the previous section), the script performs a successful login, and then calls the function verifyMainWndInitialState(), as mentioned in the previous section.

Storing data in the Environment object

The Environment global object is one of UFT's reserved objects and it can be used to store and retrieve both runtime and design-time data.

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
Chapter 1

**How to do it...**

Proceed with the following steps:

1. To store a parameter at design time, navigate to **File | Settings** from the UFT menu and then select **Environment**. From the **Variable type** list box, select **User-defined**, as shown in the following screenshot:

2. Click on the + button on the right of your **Environment** window. The **Add New Environment Parameter** window will open. Enter the new parameter's (variable) name and value:

   ![Add New Environment Parameter](image)

**For More Information:**

[www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook](http://www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook)
3. Click on the OK button to approve. You will notice that the newly created variable, with its value, now appears on the list. You should pay attention to the Type column, in which it indicates that the variable we just created is Internal:

![Image of Test Settings and Environment](image)

What does it mean? A user-defined variable is Internal when we define it through the UFT GUI. It becomes External when either we export the variables to an XML file or define them directly in such a file and later, load the file with the variables and values to the test.

**How it works...**

Definitions of the types of variable classifications are as follows:

- **Internal variables**: When you open an existing test, which has an Internal variable defined, these will be loaded automatically. Changes made to their values during the run session will not be saved. In this sense, the values given to Internal variables using the GUI can be referred to as default values.

- **External variables**: When you open an existing test, which has its user-defined variables loaded from an external XML file, these will be loaded automatically. Their values cannot be changed during the run session. In this sense, the values given to External variables can be referred to as constant values.

For More Information:

www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
There's more...

We can also store a value to an Environment variable dynamically from the code. Such a variable will have global scope but will be accessible during runtime only. This means that you will not see it in the list of Internal variables, as shown in this recipe. The procedure is equivalent to using the default Add method of the Scripting.Dictionary object, as shown in the following line of code:

```plaintext
Environment("MyEnvParam") = "MyEnvValue"
```

See also

Refer to an article by Yaron Assa at http://www.advancedqtp.com/reserved-objects-as-an-env-object-replacement.

Retrieving data from the Environment object

This recipe will show you how to retrieve data from the Environment object, which is a kind of dictionary that stores key-value pairs. As we will see, unlike a regular dictionary, the Environment object stores two types of variables:

- **Built-in**
- **User-defined**

Built-in Environment variables give access to two types of information:

- **Static data** such as the OS, OSVersion, LocalHostName, SystemTempDir, ProductDir (where UFT is installed), ProductName, ProductVer (UFT version), UserName (the Windows login name), and several settings such as UpdatingCheckpoints and UpdatingTODescriptions. In addition, we can retrieve information about the current test, such as TestName and TestDir (the path to the current test(s) from the Environment object).

- **Runtime data** such as TestIteration, ActionName, and ActionIteration can be retrieved via the Environment object during runtime. The iteration number can be useful, for instance, when we need to perform an initialization procedure that should be done only once. In this case, the iteration number must be equal to the TestIteration parameter value.

Getting ready

Create a user-defined Environment variable named MyEnvParam (see the previous recipe, Storing data in the Environment object).
**Data-driven Tests**

**How to do it...**

The following code shows how to retrieve either a built-in or a user-defined variable:

```vbscript
Print Environment("TestDir")
'Prints the Built-in TestDir (path) Environment variable to the console
Print Environment("MyEnvParam")
'Prints the User-defined MyEnvParam Environment variable to the console
```

**How it works...**

Similar to the workings of the `Scripting.Dictionary` object, by accessing an existing key, the `Environment` object returns its paired value.

**See also**

User-defined Environment variables can be stored in an XML file and dynamically loaded during the runtime session. Refer to the *Using global variables (Environment)* recipe of Chapter 8, *Utility and Reserved Objects*.

**Reading values from an INI file**

Files with the extension `.ini` are the legacy of the old Windows versions (16 bit). In the past, they were extensively used—and still are to some extent—ubiquitously to store the settings for applications. Nowadays, it is common practice to store settings in the registry. Though textual, such files have a very well-defined structure; there are sections and key-value pairs. A section starts with a label enclosed in square brackets: `[section-name]` and a key-value is implemented as `<variable name>=<value>`. Such a structure could be useful, for instance, if we wanted to keep the settings organized by environments or by user profiles within an `.ini` file.

In this recipe, you will also see an example of how to use the `Extern` reserved object to define references to methods in external DLLs, such as those of the Win32API. These methods can then be loaded and executed during runtime. A more elaborate description is available in the *Drawing a rectangle on the screen with Win32 API methods (Extern)* recipe of Chapter 8, *Utility and Reserved Objects*.

**For More Information:**

[www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook](http://www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook)
Getting ready

To complete this recipe, we need to use the global Extern object, which with proper use provides the UFT with access to the methods of an external Dynamic Link Library (DLL). We will define a variable and assign it a reference to the global Extern object (this is done to avoid persistence, as Extern is a reserved object not released from memory until UFT closes):

```vba
Dim oExtern
set oExtern = Extern
```

Then, we will declare the method or methods we wish to call from the relevant Win32API. In this case, the method is GetPrivateProfileString, which retrieves the value of a given key within a specific section:

```vba
oExtern.Declare micInteger,"GetPrivateProfileString", 
"kernel32.dll","GetPrivateProfileStringA", _
micString, micString, micString, 
micString+micByRef, micInteger, micString
```

How to do it...

After defining the connection to the DLL with its returned value and arguments, we can retrieve the value of any key within a given section. In the following example, the ConfigFileVersion key specified in the file wrls.ini is located in the UFT/bin folder. In the end, the Extern object reference is destroyed at the end of the run:

```vba
call oExtern.GetPrivateProfileString("ProgramInformation", 
"ConfigFileVersion", ", RetVal, 255, "C:\Program 
Files\HP\Unified Functional Testing\bin\wrls_ins.ini")
print  RetVal

set oExtern = nothing
```

The output to the console in this case was the string 1.05.

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
Using a configuration file to manage test environments

As shown in the previous recipe, it is possible to read variable values from an .ini file. We will show how to define several environments within such a file and load the input data for the current environment during runtime.

Getting ready

Follow the same steps stated in the Getting ready section of the Reading values from an INI file recipe.

How to do it...

Create a new file with the name QA-env-settings.ini. Enter the following entries to create three sets of parameters corresponding to three test environments QA1, QA2, and QA3:

```plaintext
[QA1]
InputDataSrc= "RegressionData1.xls"
Username   = "user1"
URL        = "http://www.url1.com"
Description = "Data for QA1 environment"

[QA2]
InputDataSrc= "RegressionData2.xls"
Username   = "user2"
URL        = "http://www.url2.com"
Description = "Data for QA2 environment"

[QA3]
InputDataSrc = "RegressionData3.xls"
Username   = "user3"
URL        = "http://www.url3.com"
Description = "Data for QA3 environment"
```

In our test, we will load the input data based on the value of the Environment variable QA_ENV, which will take one of the following environments: QA1, QA2, or QA3. Before running the test, ensure that the variable exists, and provide the value for the required testing environment (see the Storing data in the Environment object recipe). Therefore, our code in Action1 will look like the following code snippet:

```vbnet
Dim sDataSourcePath, sURL, sUsername

oExtern.GetPrivateProfileString(Environment("QA_ENV"), _
```
"InputDataSrc", "", sDataSourcePath, 255, _
"QA_env_settings.ini")

oExtern.GetPrivateProfileString(Environment("QA_ENV"), _
"InputDataSrc", "", sURL, 255, "QA_env_settings.ini")

oExtern.GetPrivateProfileString(Environment("QA_ENV"), _
"InputDataSrc", "", sUsername, 255, "QA_env_settings.ini")

DataTable.Import(sDataSourcePath)

How it works...

We retrieve the value of the QA_ENV Environment variable, and accordingly load the values of the variables in the corresponding section in the .ini file. The value of the InputDataSrc key within the corresponding section is then retrieved (note that the parameter is passed by reference and filled by the target method) and is used to import the Excel file (as you can see in the Importing an Excel file to a test recipe) that contains the input data for the given testing environment.

Using a global dictionary for fast shared data access

Using a DataTable is a generally good practice because spreadsheet data is easy to create, visualize, and maintain. This is because MS Excel lies behind the DataTable, which is, as mentioned before, a wrapper to the Excel COM object. Other advantages of using the DataTable include its full integration with the test and action iterations mechanism and with the results report, in which one can visualize each iteration, along with the input data.

This is all good for the retrieval of input data that is prepared during design time. However, using the DataTable for sharing between actions has two main drawbacks during runtime:

- Repeated writes and reads may hinder performance when it comes to a large number of iterations and a large number of parameters, as is quite often the case with many information systems.
- Sharing data with GlobalSheet is very difficult to implement. For example, suppose we need to store the CustomerID given by the system upon customer creation. In GlobalSheet, it will be stored at the current row. Though we may set the exact row using the DataTable method, that is, SetCurrentRow (<rownumber>), it is still a question of how to ensure that at a later stage, an action that needs a CustomerID would know the correct row number.

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
An alternative to sharing data among actions would be to use the UFT’s built-in Output and Input parameters. However, Input parameters are good only to pass data from an action to its nested (called) actions, and Output parameters are good only to pass data to other sibling actions (that is, those which are at the same hierarchical level). Hence, they do not enable the flexibility one may need when testing complex systems and are cumbersome to manage.

A better approach is to have the data that must be shared and stored in the Dictionary object of a global scope. A Dictionary object is actually a hash table with a capacity to store values of different types, such as strings, numbers, Booleans, arrays, and references to objects (including other nested Dictionary objects, which is a powerful, yet very advanced technique that is out of scope here). Each value is stored with a unique key by which it can be accessed later.

**Getting ready**

In UFT, create a new function library by navigating to **File | New | Function Library** (or use the key shortcut **Alt + Shift + N**) and save it as **UFT_Globals.vbs**. It is recommended to save it in a folder, which would be shared later by all tests.

Navigate to **File | Settings** and attach the function library to the test.

**How to do it...**

As any public variable declared in a function library attached to a test can be accessed by any action, we will define a global variable and two functions to initialize `initGlobalDictionary` and dispose `disposeGlobalDictionary`:

```vbscript
Dim GlobalDictionary

Function initGlobalDictionary()
    If not (lcase(typename(GlobalDictionary)) = "dictionary") Then
        Set GlobalDictionary = CreateObject("Scripting.Dictionary")
    End If
End Function

Function disposeGlobalDictionary()
    Set GlobalDictionary = nothing
End Function
```

For More Information:
[www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook](http://www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook)
Chapter 1

The initGlobalDictionary() function will check if the public variable GlobalDictionary was not initialized earlier, and then set it with a reference to a new instance of a Dictionary object, as mentioned in the previous code. The disposeGlobalDictionary() function is given for the sake of completeness, as in any case, memory is released when the test stops. However, we may wish to empty the GlobalDictionary variable in certain cases, so it is recommended to include this function as well.

Now, in Action1 (or whichever action runs first in our test), we will write the following code:

```vbscript
If CInt(Environment("TestIteration")) = 1 and CInt(Environment("ActionIteration")) = 1 Then
    call initGlobalDictionary()
End If
```

The previous code will ensure that the GlobalDictionary variable is instantiated only once at the beginning of the run session. If we need a new instance for every test iteration, then we just need to change the code to the following lines of code, so that we get a new instance only at the start of the first Action1 iteration:

```vbscript
If CInt(Environment("ActionIteration")) = 1 Then
    call initGlobalDictionary()
End If
```

With our test set up this way, we can now use this global object to share data as in the following example. Create a new Action2 DataTable and make it run after Action1 (at the end of the test). Now, write the following code in Action1:

```vbscript
GlobalDictionary.Add "CustomerID", "123456789"
Print Environment("ActionName") & ": " & GlobalDictionary("CustomerID")
```

In Action2, write the following code:

```vbscript
Print Environment("ActionName") & ": " & GlobalDictionary("CustomerID")
```

It is strongly recommended to remove a key from the dictionary when it is no longer required:

```vbscript
GlobalDictionary.Remove "CustomerID"
```

Alternatively, to remove all keys from the dictionary altogether at the end of a test iteration or at the beginning of a test iteration greater than the first, use the following line of code:

```vbscript
GlobalDictionary.RemoveAll
```

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
Data-driven Tests

As mentioned earlier, keys must be unique and if we use the same keys in each test iteration, it would cause a runtime error with the first key found to exist in the dictionary. Another way, as mentioned earlier, is to call the `disposeGlobalDictionary` at the end of each test iteration and the `initializeGlobalDictionary()` method at the start.

**How it works...**

When you run this test, in `Action1`, it first creates a new `Dictionary` instance and assigns a reference to the public variable `GlobalDictionary`. Then, it adds a new key `CustomerID` with the value `123456789`, and prints the action name from the Environment built-in runtime variables ("Action1") and the value, by referring to the `CustomerID` key we just added. Then, it executes `Action2`, where it again prints in the same manner as in `Action1`. However, as the `ActionName` Environment variable is dynamic, it prints "Action2". This is to prove that `Action2` actually has access to the key and value added in `Action1`. The output of this test is as shown in the screenshot:

![Output screenshot](image)

See also

Refer to the *Using a global dictionary for fast shared code access* recipe.

For More Information:

[www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook](www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook)
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.

The remaining implementation deserves special attention. In the Globals.vbs function library that we attached to the test, we will add the following pieces of code:

```vbs
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
```

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
If err.number <> 0 Then
    reporter.ReportEvent micFail, "GetInstance", "Class " & cls & " is not defined (error #" & err.number & ")"
    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:

For More Information:
www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook
**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:

```vbnet
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:

```vbnet
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*.
Where to buy this book

You can buy Advanced UFT 12 for Test Engineers Cookbook from the Packt Publishing website: https://www.packtpub.com/application-development/advanced-qtp-12-test-engineers-cookbook.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.