Chapter No. 3
"Automated Tests"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO.3 "Automated Tests"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Satheesh Kumar N holds a Bachelor’s degree in Computer Science engineering and has around 17 years of experience in managing the software development life cycle, developing live projects, and program management. He started his career by developing software applications using Borland software products. He worked for multiple organizations in India, the UAE, and the US. His main domain expertise is in retail and he is currently working in Bangalore as a Program Delivery Manager for the top retailer in UK. He is currently handling five agile scrum teams for delivering the website features. His experience also includes implementation and customization of Microsoft Dynamics for an automobile sales company in UAE. He works with the latest Microsoft technologies and has published many articles on LINQ and other features of .NET. He is a certified PMP (Project Management Professional).


I would like to thank my wife for helping me in co-authoring and supporting me in all the ways to complete this book. I would also like to thank my family members and friends for their continuous support in my career and success.

For More Information:
Subashni S holds a Bachelor’s Degree in Computer Science engineering and has around 15 years of experience in software development and testing life cycle, project, and program management. She is a certified PMP (Project Management Professional), CSTM (Certified Software Test Manager), and ITIL V3 Foundation certified. She started her career as a DBA in Oracle 8i technology, and later developed many software applications using Borland software products for a multinational company based in Chennai, and then moved to Bangalore. She is presently working for a multinational company, in the area of Project Management for developing and testing projects. She is also currently working for one of the top multinational companies headquartered at Austin, Texas.

She has also authored *Software Testing using Visual Studio Team System 2008* and *Software Testing using Visual Studio 2010* for Packt Publishing.

I would like to thank my husband for helping me in co-authoring and supporting me in all the ways to complete this book. I would also like to thank my other family members and friends for their continuous support in my career and success.
Software Testing using Visual Studio 2012

The Microsoft Visual Studio 2012 suite contains several features to support the needs of developers, testers, architects, and managers to simplify the development process. Visual Studio 2012 provides different editions of the product such as Professional, Premium, and Ultimate with different set of tools and features. Visual Studio 2012 is tightly integrated with Team Foundation Server, a central repository and configuration management system that provides version control, process guidance and templates, automated builds, automated tests, bug tracking, work item tracking, reporting, and support of the Lab Center and Test Center configurations. The Microsoft Test Manager 2012 is a standalone tool used to organize Test Plans, Manage test cases, and executing manual test cases.

Software Testing using Visual Studio 2012 helps software developers to get familiarized with the Visual Studio tools and techniques to create automated unit tests, and to use automated user interface testing, code analysis and profiling to find out more about the performance and quality of the code. Testers benefit from learning more about the usage of Testing tools, test case management techniques, working with Test Results, and using Test Center and Lab center. This book also covers different types of testing such as web performance test, load test, executing the manual test cases, recording user actions, re-running tests using recording, test case execution, capturing defects, and linking defects with requirements. Testers also get a high level overview on using Lab Center for creating virtual environments for testing multiple users and multiple location scenarios.

Visual Studio 2012 provides user interface tools such as Test Explorer, Test Results, and Test Configuration to create, execute, and maintain the tests and Test Results in integration with Team Foundation Server. This book provides detailed information on all of the tools used for testing the application during the development and testing phases of the project life cycle.

What This Book Covers

Chapter 1, Visual Studio 2012 Test Types, provides an overview of different types of testing which helps testing the software applications through different phases of software development. This chapter also introduces the tools and techniques in Visual Studio 2012 for different testing types, Microsoft Test Manager 2012, and its features.

Chapter 2, Test Plan, Test Suite, and Manual Testing, explains the steps involved in creating and managing the Test Plan, Test cases and Test Suite using Test Center in Test Manager. This chapter also explains how to create manual tests by recording the user actions and running the test with data inputs. Sharing the test recording across multiple tests is also covered in this chapter.

For More Information:  
Chapter 3, Automated Tests, provides a step-by-step approach to creating Coded UI test from user action recordings. It also explains the steps to execute the coded UI test through data source and adding validation and custom rules to the test.

Chapter 4, Unit Testing, explains the detailed steps involved in creating unit test classes and methods for the code. Different type of assert methods and parameters for testing the code, passing set of data from a data source and testing the code also explained in detail. The mocking framework used for isolating the code and testing it with the help of Shims and Stubs is also explained in detail.

Chapter 5, Web Performance Test, explains the basic way of web testing by recording the user actions and creating a test out of it. Running the test using a data source, adding parameters to the web tests, adding validation and extraction rules, adding looping and branching mechanism to the recorded tests, and here configuring the settings required for the Test Runs are some of the features explained as part of this chapter.

Chapter 6, Advanced Web Testing, explains the way of generating code out of the recorded web tests explained in Chapter 5, Web Performance Test using the Generate Code option. This is very much useful for customizing the test through the code, adding additional logic to the test, adding custom validation and extraction rules.

Chapter 7, Load Testing, helps in simulating various numbers of users, network bandwidths, combination of different web browsers, and different configurations. In the case of web applications it is always necessary to test the stability and performance of the application under huge data load and concurrent users. This chapter explains the steps involved in simulating the real world scenario by using Controllers and Agents. The details of analyzing and exporting the load Test Results are also explained in this chapter.

Chapter 8, Ordered and Generic Tests, explains the way of testing the existing third party tool or service which can also be run using the command line. Visual Studio 2012 provides a feature called ordered test to group all or some of these tests and then execute the tests in the same order. The main advantage of creating the ordered test is to execute multiple tests in an order based on the dependencies. Generic tests are just like any other tests except that it is used for testing an existing third party tool or service.

Chapter 9, Managing and Configuring Tests, explains the details of the test settings file and the tools used for managing tests. The configuration includes deployment details, setup and cleaning scripts, collecting data diagnostics information, unit test and web test settings.

Chapter 10, The Command Line, explains the command line tools such as VSTest. Console, MSTest, and TCM used for running the test with different options, then collecting the output and publishing the results. Each of these commands is used for specific purposes including backwards compatibility.

For More Information:  
Chapter 11, *Working with Test Results*, explains the process of running the tests and publishing the Test Results to the Team Project. Also covered in detail is to integrate the tests as part of Team Foundation Server builds, Build reports and Test Results, Creating work items from Test Results, and publishing the Test Results.

Chapter 12, *Exploratory Testing and Reporting*, explains the details of testing which happens without any test cases and scripts and by only exploring the application manually. This chapter also explains the details of accessing the Test Results and publishing Test Results and reporting the same in a specific format. Accessing different types of testing reports and creating new test reports are also explained in this chapter.

Chapter 13, *Test and Lab Center*, is useful for creating and organizing Test Plans and test cases. Test plans can be associated to the requirements using Test Center. The Lab Center helps in creating and configuring different virtual/physical environments for the Test Runs, Test Settings such as defining the roles and configuring the data and diagnostics information for the selected roles, configuring the Test Controllers required for the test, and configuring the test library to store the environment information.

For More Information:  
Automated Tests

Automated test is a form of testing to record the manual testing steps and then re-runs the recorded steps without performing the entire test manually again. The other type is to write scripts using programming language to automate the service level testing. Automating user interface (UI) testing was the biggest challenge but nowadays a few tools provide the flexibility to record user actions and create a script out of it. It is made simpler in Microsoft Visual Studio 2012. These tests are also called coded UI tests. The existing manual tests, test cases, and the action recordings of the user interface tests are re-used for generating the automated tests and the code files in the managed code (C# or VB.NET).

The UI controls can be added to the coded UI test and then the properties and values of controls can be verified using the Coded UI Test Builder feature. To conduct the same test multiple times but with different sets of data, the coded UI test can be made as a data-driven test by adding a data source to the test. The test would then be called for each row of data in the data source.

The coded UI test can be run directly from Visual Studio or Microsoft Test Manager and can be linked to the requirements to determine the number of automated tests for each requirement and also to gather the Test Results for the requirement.

For More Information:
Coded UI tests from action recordings

Action recording is a very useful feature for recording user actions and then creating test scripts out of it. The test scripts can be customized or used as is to play back the test instead of repeating the same test manually. Recording of actions is done using the Test Runner. The details of creating and recording the user actions are covered as part of Chapter 2, Test Plan, Test Suite, and Manual Testing which talks about Test Plans and manual testing. This section explains the details of creating a coded UI test from an existing action recording. The following image shows the successful completion of action recording for the manual test.
Follow these steps to create the coded UI test:

1. In Visual Studio, select the Test Project from the solution explorer, if the Test Project already exists in the solution and then add a new test to the project using the context menu.
   
   Otherwise select Project from the main menu and then select Add Coded UI Test from the list of options available.

   2. The selected Coded UI Test option will add the code file to the Test Project. The code file will contain only the class with the CodedUITest attribute, a test method named CodedUITestMethod1 with the attribute TestMethod and a test context. All these methods are empty as the code for the test is not yet generated.

For More Information:

3. After selecting the coded UI test menu, there are two options available for creating the test. One is to Record actions, edit UI map or add assertions, which is like starting everything from the beginning. The second option is to Use an existing action recording for the manual test.

Choose the second option to use the existing action recording which was recorded as part of manual testing in Chapter 2, Test Plan, Test Suite, and Manual Testing. After choosing the Use an existing action recording, select the work item using the Work Items Picker screen which is displayed. Use the filter options to filter the manual test case for which the action recording is available.

For More Information:
4. Select the test case from this window will generate the code based on the action recording.
Automated Tests

Additional files such as UIMap.uitest, UIMap.cs, and UIMap.Designer.cs are created at the time of code generation. The main method CodedUITestMethod1() in the CodeUITest1.cs file contains calls for the methods created for each action while recording the user actions. The corresponding method definition is created in the UIMap.Designer.cs file by the Coded UI Test Builder itself. The code below contains the action methods generated under CodedUITestMethod1():

This is the sample code for the action method in the class.

```csharp
public void CodedUITestMethod1()
{
    // To generate code for this test, select
    "Generate Code for Coded UI Test" from the shortcut menu
    and select one of the menu items.
    // For more information on generated code,
    see http://go.microsoft.com/fwlink/?LinkId=179463
    this.UIMap.BrowsetotheEmployeeDetailsApplication();
    this.UIMap.OpentheInsertEmployeeDetailsPage();
    this.UIMap.BrowsetotheEmployeeDetailsApplication();
    First_NameLast_NameMiddle_NameParams.UICountryEditText =
        TestContext.DataRow["Country"].ToString();
    this.UIMap.BrowsetotheEmployeeDetailsApplication();
    First_NameLast_NameMiddle_NameParams.UIStateEditText =
        TestContext.DataRow["State"].ToString();
    this.UIMap.BrowsetotheEmployeeDetailsApplication();
    First_NameLast_NameMiddle_NameParams.UICityEditText =
        TestContext.DataRow["City"].ToString();
    this.UIMap.BrowsetotheEmployeeDetailsApplication();
    First_NameLast_NameMiddle_NameParams.UIGenderEditText =
        TestContext.DataRow["Gender"].ToString();
    this.UIMap.BrowsetotheEmployeeDetailsApplication();
    First_NameLast_NameMiddle_NameParams.UIOccupationEditText =
        TestContext.DataRow["Occupation"].ToString();
    this.UIMap.BrowsetotheEmployeeDetailsApplication();
    First_NameLast_NameMiddle_NameParams.UIDepartmentEditText =
        TestContext.DataRow["Department"].ToString();
    this.UIMap.BrowsetotheEmployeeDetailsApplication();
    First_NameLast_NameMiddle_NameParams.UIFirst_NameEditText =
        TestContext.DataRow["First_Name"].ToString();
```

For More Information:
this.UIMap.PhoneCountryStateCityGenderOccupationDepartment
First_NameLast_NameMiddle_Name();
this.UIMap.Submitthedetails();
}

The coded UI test code generation creates several files and adds them to the Test Project. We will now see the details of each file that gets generated.

**Files generated for coded UI test**

While creating the coded UI test, the Test Builder generates multiple files to map the user interface, test methods, parameters, and assertions for all tests.

**CodedUITest1.cs**

The name of this file is generated based on the name of the test that is created. This file can be modified any time. This file contains one public class with the name CodedUITest1, with the CodedUITest attribute added to the class so that this class can be recognized as a test class. The name CodedUITest1 is the default name chosen by the system. If this file already exists, the system increments the number associated with the name and then creates the file with the new name.

The class also contains two default properties, TestContext and UIMap.

The following screenshot shows the default properties:

```
    /// <summary>
    /// Gets or sets the test context which provides
    /// information about and functionality for the current test run.
    /// </summary>
    public TestContext TestContext;
    private TestContext testContextInstance;
    
    public UIMap UIMap;
    private UIMap map;
```

For More Information:
Automated Tests

There are two additional methods which are commented out by default. A region titled **Additional test attributes** contains these two optional methods, as shown in the following screenshot:

```csharp
/// Region Additional test attributes
#else
</region>

The `MyTestInitialize()` method is called once before any other test methods during the Test Run. This is useful for initializing the tests and is identified as the initializer using the attribute `TestInitialize`. Similarly the method `MyTestCleanup()` method is called once after all the tests have been called, and this method is identified using the attribute `TestCleanup()`.

**UIMap.Designer.cs**

The Coded UI Test Builder automatically creates the code in this file when a test is created. The file gets updated whenever the test is modified. This file contains a UIMap class which has the attribute `GeneratedCode`. All classes in this file are auto generated codes and every class has the attribute `GeneratedCode` associated with it. The UIMap class contains the definition of all the methods that were identified during recording. Following are some of the methods captured during recording:

```csharp
public void BrowsetotheEmployeeDetailsApplication()
public void OpentheInsertEmployeeDetailsPage()
public void Submitthedetails()
```

For More Information:
The definition of each method follows a defined structure. The structure contains a summary of the method, a region at the top defining the variables, and then the definitions of the method calls and properties. The following code shows the definition for one of the method calls:

```csharp
/// <summary>
/// BrowsetotheEmployeeDetailsApplication – Shared Steps 14 –
/// Use 'BrowsetotheEmployeeDetailsApplicationParams' to pass
/// parameters into this method.
/// </summary>
public void BrowsetotheEmployeeDetailsApplication()
{
    #region Variable Declarations
    HtmlHyperlink uIEmployeeHyperlink =
        this.UIBlankPageWindowsInteWindow.
            UIEmployeeMaintenanceDocument.UIEmployeeHyperlink;
    #endregion

    // Go to web page 'http://localhost:3062/' using new browser
    // instance
    this.UIBlankPageWindowsInteWindow.LaunchUrl(new System.Uri
        (this. BrowsetotheEmployeeDetailsApplicationParams.
            UIBlankPageWindowsInteWindowUrl));

    // Click 'Employee' link
    Mouse.Click(uIEmployeeHyperlink, new Point(17, 9));
}
```

**UIMap.cs**

This file contains the partial UIMap class but does not contain any properties or methods initially. However, custom code can be included in the UIMap class to customize the existing functionality or add new functionality.
Automated Tests

UiMap.uitest

This is an XML file which represents the structure of the coded UI test recording. These include the actions, properties, and methods of the classes. The UiMap.Designer.cs file contains definitions of all the methods that are generated by the coded UI Builder. As the UiMap test files are generated by the Test Builder, it is not advisable to edit the files directly, but rather to use the UiMap editor to work with the methods. Every time there is a change to the recording or to the controls in the recording the file is regenerated and overwrites the custom code, which is the reason why we do not modify the generated code. The following screenshot shows the editor with the list of recorded actions and the corresponding UI controls. The Click 'Employee' link is the action which corresponds to the UIEmployeeHyperLink.

For More Information:
The editor contains options to delete a method, rename the method, set properties, split into a new method, move code to the UIMap.cs file, insert delays, and locate controls. The following screenshot shows the properties window for one of the user actions - in this case, to go to the web page:

For More Information:  
Automated Tests

The other main functionality in the editor is to move the code to the UIMap.cs file. Initially the UIMap.cs file would be an empty class without any implementation. If there is any customization required, it is not advisable to directly edit in the designer.cs file, but the method can be moved to the UIMap.cs file and then the customization can be done. Choosing the Move code to UIMap.cs option provides a warning saying the method will be removed from the UIMap.uitest file and moved to UIMap.cs and you will not be able to edit the method using coded UI test.

For More Information:
Choosing the **Move code to UIMap.cs** option provides a warning saying the method will be removed from `UIMap.uitest` and moved to the `UIMap.cs` file and you will not be able to edit the method using coded UI test, as shown in the following screenshot:

Once you confirm the code move, the method is removed from `UIMap.uitest` and copied to `UIMap.cs` and is ready for customization.
Data-driven coded UI test

The coded UI test that was created previously is for a given set of data captured during test recording. Later on, the test may be required not only for one set of data but for different sets of data and for multiple times. To achieve this, we parameterize each field to get data from a data source during testing. Each row of data in the data source is an iteration of coded UI test. When generating methods or assertions for the coded UI test, all constants in the recorded methods are parameterized into parameter classes. In the previous code example, there is a `BrowsetotheEmployeeDetailsApplication` method as shown in the following code:

```csharp
/// <summary>
/// BrowsetotheEmployeeDetailsApplication – Shared Steps 14 – 
/// Use 'BrowsetotheEmployeeDetailsApplicationParams' to pass 
/// parameters into this method. 
/// </summary>
public void BrowsetotheEmployeeDetailsApplication()
{
    #region Variable Declarations
    HtmlHyperlink uIEmployeeHyperlink =
        this.UIBlankPageWindowsInteWindow.
            UIEmployeeMaintenanceDocument.UIEmployeeHyperlink;
    #endregion

    // Go to web page 'http://localhost:3062/' using new
    browser instance
    this.UIBlankPageWindowsInteWindow.LaunchUrl(new System.Uri
        (this. BrowsetotheEmployeeDetailsApplicationParams.
            UIBlankPageWindowsInteWindowUrl));

    // Click 'Employee' link
    Mouse.Click(uIEmployeeHyperlink, new Point(17, 9));
}
```

For the above method the Coded UI Test Builder creates the class as shown in the following code and adds fields to the class for every constant value used while recording.

```csharp
public HtmlHyperlink UIEmployeeHyperlink
{
    get
    {
        if ((this.mUIEmployeeHyperlink == null))
```
Chapter 3

{ 
    this.mUIEmployeeHyperlink = new HtmlHyperlink(this);
    #region Search Criteria
    this.mUIEmployeeHyperlink.SearchProperties
        [HtmlHyperlink.PropertyNames.Id] = "ContentPlaceHolder1_Menu1_HyperLink1_2";
    this.mUIEmployeeHyperlink.SearchProperties
        [HtmlHyperlink.PropertyNames.Name] = null;
    this.mUIEmployeeHyperlink.SearchProperties
        [HtmlHyperlink.PropertyNames.Target] = null;
    this.mUIEmployeeHyperlink.SearchProperties
        [HtmlHyperlink.PropertyNames.InnerText] = "Employee";
    this.mUIEmployeeHyperlink.FilterProperties
    this.mUIEmployeeHyperlink.FilterProperties
        [HtmlHyperlink.PropertyNames.Title] = null;
    this.mUIEmployeeHyperlink.FilterProperties
    this.mUIEmployeeHyperlink.FilterProperties
        [HtmlHyperlink.PropertyNames.Class] = null;
    this.mUIEmployeeHyperlink.FilterProperties
        [HtmlHyperlink.PropertyNames.ControlDefinition] = "id=ContentPlaceHolder1_Menu1_HyperLink1_";
    this.mUIEmployeeHyperlink.FilterProperties
        [HtmlHyperlink.PropertyNames.TagInstance] = "5";
    this.mUIEmployeeHyperlink.WindowTitles.Add
        ("Employee Maintenance");
    #endregion
    return this.mUIEmployeeHyperlink;
}

#endregion

#region Fields
private HtmlHyperlink mUIEmployeeHyperlink;
#endregion
Automated Tests

Now the required test and test files are created. Let us create a data source in the form of a .csv file and use it for the coded UI test. The sample data in the .csv file is shown in the following screenshot:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>First_Name</td>
<td>Last_Name</td>
<td>Middle_Name</td>
<td>Department</td>
<td>Occupation</td>
<td>Gender</td>
<td>City</td>
<td>State</td>
<td>Country</td>
<td>Phone</td>
</tr>
<tr>
<td>2</td>
<td>Satheesh</td>
<td>Kumar</td>
<td>N</td>
<td>IT</td>
<td>Architect</td>
<td>Male</td>
<td>Bangalore</td>
<td>Karnataka</td>
<td>India</td>
<td>1112223334</td>
</tr>
<tr>
<td>3</td>
<td>Subashni</td>
<td>S</td>
<td>S</td>
<td>IT</td>
<td>Manager</td>
<td>Female</td>
<td>Bangalore</td>
<td>Karnataka</td>
<td>India</td>
<td>1112223555</td>
</tr>
<tr>
<td>4</td>
<td>Subha</td>
<td>S</td>
<td></td>
<td>IT</td>
<td>Sr Manager</td>
<td>Female</td>
<td>Bangalore</td>
<td>Karnataka</td>
<td>India</td>
<td>1112223355</td>
</tr>
</tbody>
</table>

Select the test method from the class file and insert the data source attribute directly in the code immediately above the method. The earlier Version of Visual Studio has the data source wizard to associate the data source to the test method. But VS 2012 does not have the wizard. Just add the data source attribute and start using the data columns and rows as follows:

```csharp
```

Save the changes to the CodedUITest1.cs file. Now right-click the coded UI test in the code editor and choose Run Unit Test. After the test is run, the overall Test Result for all iterations is displayed in the Test Results window.

If the CodedUITest1 test is run now, the test would run for each row in the data source. As there are three rows in the data source the test would run three times. Even if one of these tests fails, the entire Test Result would fail.

What is shown in the preceding screenshot is only the CSV data source but there are multiple other data sources, such as XML file, Microsoft Excel, test cases in Team Foundation Server, and SQL Express.

Adding controls and validation to coded UI test

Sometimes some kind of validation is required for UI controls. For example, some of the controls in the UI should not be null. Use the Coded UI Test Builder to generate code for the validation method that uses an assertion for a UI Control. Add the UI control to the existing UI map file and generate the code to the existing coded UI test file.

For More Information:
Open the Test Project and the coded UI test file, which is named `CodedUITest1.cs`. In the code file place the cursor on `CodedUITestMethod1()`, right-click and select the option **Generate code for Coded UI Test** and then choose **Use Coded UI Test Builder**.

The Coded UI Test Builder opens with menu options for adding controls and validations. Open the application and then open the UI page for which we need to add the validation logic. Keeping the UI open, simply drag-and-drop the crosshair from the Test Builder to the control on the UI. The other option is to select the control then keep the mouse pointer on the UI Control while pressing the Windows logo key + I to select the control at the mouse pointer.

---

For More Information:

After selecting a control, the **Add Assertions** screen opens for the selected control. The window displays all the properties of the selected control. Select the property of the control to be validated and then click on the **Add Assertion** option. The following image shows the window to select the assertion type, add the **Comparison Value** for the validation, and to provide the **Message on Assertion Failure**.

Just for testing purposes, let's add an **IsNotNull** assertion type for the first name UI control for the **Text** property of the control. Click on **Ok** to add the assertion to the test. Keep adding assertions for all validations required for the controls. Once all required assertions are added, click on the **Generate code** option in the Test Builder and provide name for the assert methods that were added. This option automatically creates the code for assertions and adds the method definition and method calls to the corresponding files.

For More Information:
All assertion method definitions are added to the UIMap.Designer.cs file and the method is called from the main method CodedUITestMethod1() in the CodedUITest1.cs file. The following code shows the code for the assertion generated in the designer file:

```csharp
/// <summary>
// This is the assert method for validating the null value for employee first name
/// </summary>
public void AssertMethod2()
{
    #region Variable Declarations
    HtmlEdit uIFirst_NameEdit =
        this.UIEmployeeWindowsInterWindow.UIEmployeeDocument1.
        UIFirst_NameEdit;
    #endregion

    // Verify that the 'Text' property of 'First_Name'
    // text box is not equal to 'null'
    Assert.IsNotNull(uIFirst_NameEdit.Text,
        "The Employee First Name cannot be Null");
}
```

For More Information:
Now to test the assertion functionality, open the data source CSV file and empty the values for the first name of all the employees in the file. Navigate to CodedUITest1 from the Test Explorer window and run the test. As there are three rows in the data source, there will be three iterations of the test run, and all three will fail because the first name is null in all rows. The test is checking for a not null value in the first name field. So the entire test would fail because of the tests failure.

Now the required coded UI testing is successful. One difficulty in this type of code generation is the code maintenance. As we know that all the assertion codes and the validation methods are added to the UIMap class, there is chance of this class file growing to a larger size, if we keep on adding the controls and methods. To avoid this situation, multiple UIMap files can be generated.

The application can be grouped into modules or logical subsets, and each UIMap file mapped to one particular logical subset of the application. This logical grouping also helps the tester to work on an individual module without affecting other areas of the application. To create the logical grouping, first create a folder under Test Project. Then select the folder and create a new item of type Coded UI Test Map from the available templates as follows:

For More Information:
Click on Add after providing a name for the new Map file. The Coded UI Test Builder window will now appear after minimizing the Visual Studio window. Using the Test Builder, keep recording the actions and creating the validations for the UI controls. Make sure of adding controls and validation specific to the module for which the map file is created. Generate the code using the option in the Test Builder after completing the recording. You can see the new .uitest and designer.cs files added to the test under the new folder. The following image shows the new UIMap files created under the new folder:

For any mapping that we create there are certain best practices to follow for easy maintenance and successful Test Results, such as:

- Do not modify the UIMap.Designer.cs file as it is meant only for the Test Builder to modify.
- Always use Coded UI Test Builder to create all assertions and limit the recording to few user actions.
- Use meaningful names for the UIMap files and the assertion methods to easily identify and maintain the code and tests.
- Always re-record the user actions after any changes to the user interface.
Summary
This chapter provides information on the new features added to the coded UI test in Visual Studio 2012. The new version has the new editor to edit the UIMap.uitest files, along with a few features to move the code and split the code to the map files. The Test Builder is a very handy tool for selecting the controls, add assertion methods, and generating code. This chapter also explained how to maintain multiple Map files under different folders. The samples also explain having a data source to automate the same test with different sets of data, without re-running the test manually for each set of data.

The next chapter explains the details of testing the smallest piece of testable code isolated from the remaining code. This is called the unit testing type which is normally conducted by the developer to test the code independently.

For More Information:
Where to buy this book


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.