HP UFT .NET Add-in Extensibility

For the Windows ® operating systems

Software Version: 12.00

Developer Guide

Document Release Date: March 2014

Software Release Date: July 2014

Legal Notices

Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

Trademark Notices

Adobe® and Acrobat® are trademarks of Adobe Systems Incorporated.

Google™ and Google Maps™ are trademarks of Google Inc.

Intel® and Pentium® are trademarks of Intel Corporation in the U.S. and other countries.

Microsoft®, Windows®, Windows® XP, and Windows Vista ® are U.S. registered trademarks of Microsoft Corporation.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

Documentation Updates

The title page of this document contains the following identifying information:

- Software Version number, which indicates the software version.
- Document Release Date, which changes each time the document is updated.
- Software Release Date, which indicates the release date of this version of the software.

To check for recent updates or to verify that you are using the most recent edition of a document, go to: **http://h20230.www2.hp.com/selfsolve/manuals**

This site requires that you register for an HP Passport and sign in. To register for an HP Passport ID, go to: **http://h20229.www2.hp.com/passport-registration.html**

Or click the **New users - please register** link on the HP Passport login page.

You will also receive updated or new editions if you subscribe to the appropriate product support service. Contact your HP sales representative for details.

Support

Visit the HP Software Support Online web site at: **http://www.hp.com/go/hpsoftwaresupport**

This web site provides contact information and details about the products, services, and support that HP Software offers.

HP Software online support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valued support customer, you can benefit by using the support web site to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Download software patches
- Manage support contracts
- Look up HP support contacts
- Review information about available services
- Enter into discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and sign in. Many also require a support contract. To register for an HP Passport ID, go to:

http://h20229.www2.hp.com/passport-registration.html

To find more information about access levels, go to:

http://h20230.www2.hp.com/new_access_levels.jsp

HP Software Solutions Now accesses the HPSW Solution and Integration Portal Web site. This site enables you to explore HP Product Solutions to meet your business needs, includes a full list of Integrations between HP Products, as well as a listing of ITIL Processes. The URL for this Web site is **http://h20230.www2.hp.com/sc/solutions/index.jsp**

Contents

Welcome to HP UFT .NET Add-in Extensibility

HP UFT .NET Add-in Extensibility is an SDK (Software Development Kit) package that enables you to support testing applications that use third-party and custom .NET Windows Forms controls that are not supported out-of-the-box by the UFT .NET Add-in.

This chapter includes:

About the UFT .NET Add-in Extensibility SDK

The UFT .NET Add-in Extensibility SDK installation provides the following:

- An API that enables you to extend the UFT .NET Add-in to support custom .NET Windows Forms controls.
- Custom Server C# and Visual Basic project templates for Microsoft Visual Studio.

  Each Custom Server template provides a framework of blank code, some sample code, and the UFT project references required to build a custom server.

  Note: For a list of supported Microsoft Visual Studio versions, see the *HP Unified Functional Testing Product Availability Matrix*, available from the UFT help folder or the HP Support Matrix page (requires an HP passport).

- The wizard that runs when the Custom Server template is selected to create a new project. The wizard simplifies setting up a Microsoft Visual Studio project to create a Custom Server .NET DLL using .NET Add-in Extensibility. For more information, see Using a .NET DLL to Extend Support for a Custom Control.
- The .NET Add-in Windows Forms Extensibility Help, which includes the following:
  - A developer guide, including a step-by-step tutorial in which you develop support for a sample custom control.
  - An API Reference.
  - The .NET Add-in Extensibility Configuration Schema Help.
  - The .NET Add-in Extensibility Control Definition Schema Help.
  - The UFT Test Object Schema Help.

  The Help is available from **Start > All Programs > HP Software > HP Unified Functional Testing > Extensibility > Documentation**
- A printer-friendly Adobe portable document format (PDF) version of the developer guide (available from **Start > All Programs > HP Software > HP Unified Functional Testing > Extensibility > Documentation** and in the **<Unified Functional Testing installation>\help\Extensibility** folder).
- A sample .NET Add-in Extensibility support set that extends UFT GUI testing support for the SandBar toolbar custom control.

Accessing UFT .NET Add-in Extensibility in Windows 8 Operating Systems

UFT files that were accessible from the **Start** menu in previous versions of Windows are accessible in Windows 8 from the **Start** screen or the **Apps** screen.

- **Applications (.exe files).** You can access UFT applications in Windows 8 directly from the **Start** screen. For example, to start UFT, double-click the **HP Unified Functional Testing** shortcut.
- **Non-program files.** You can access documentation from the **Apps** screen.

> **Note:** As in previous versions of Windows, you can access context sensitive help in UFT by pressing **F1**, and access complete documentation and external links from the **Help** menu.

About the UFT .NET Add-in Extensibility Developer Guide

This guide explains how to set up UFT .NET Add-in Extensibility and use it to extend UFT GUI testing support for third-party and custom .NET Windows Forms controls.

This guide assumes you are familiar with UFT functionality and should be used together with the following sections of the .NET Add-in Extensibility online Help (**Start > All Programs > HP Software > HP Unified Functional Testing > Extensibility > Documentation > .NET Add-in Windows Forms Extensibility Help**):

- *UFT .NET Add-in Extensibility API Reference*
- *UFT .NET Add-in Extensibility Systems Forms Configuration Schema Help*
- *UFT .NET Add-in Extensibility Control Definition Schema Help*
- *HP UFT Test Object Schema Help*

These documents should also be used in conjunction with the following UFT documentation, available with the UFT installation (**Help > HP Unified Functional Testing Help** from the UFT main window):

- *HP Unified Functional Testing User Guide*
- The .NET section of the *HP Unified Functional Testing Add-ins Guide*
- *HP UFT Object Model Reference for GUI Testing*

Note:

The information, examples, and screen captures in this guide focus specifically on working with UFT GUI tests. However, much of the information in this guide applies equally to business components.

Business components are part of HP Business Process Testing. For more information, see the *HP Unified Functional Testing User Guide* and the *HP Business Process Testing User Guide*.

When working in Windows 8, access UFT documentation and other files from the **Apps** screen.

To enable you to search this guide more effectively for specific topics or keywords, use the following options:

- **AND**, **OR**, **NEAR**, and **NOT** logical operators. Available from the arrow next to the search box.
- **Search previous results.** Available from the bottom of the **Search** tab.
- **Match similar words.** Available from the bottom of the **Search** tab.
- **Search titles only.** Available from the bottom of the **Search** tab.

Tip: When you open a Help page from the search results, the string for which you searched may be included in a collapsed section. If you cannot find the string on the page, expand all the drop-down sections and then use Ctrl-F to search for the string.

To check for recent updates, or to verify that you are using the most recent edition of a document, go to the HP Software Product Manuals Web site (http://h20230.www2.hp.com/selfsolve/manuals).

Who Should Read This Guide

This guide is intended for programmers, QA engineers, systems analysts, system designers, and technical managers who want to extend UFT GUI testing support for .NET Windows Forms custom controls.

To use this guide, you should be familiar with:

- Major UFT features and functionality
- The UFT Object Model
- UFT .NET Add-in
- .NET programming in C# or Visual Basic
- XML (basic knowledge)

Additional Online Resources

The following additional online resources are available:

Resource	Description
Troubleshooting & Knowledge Base	The Troubleshooting page on the HP Software Support Web site where you can search the Self-solve knowledge base. The URL for this Web site is http://h20230.www2.hp.com/troubleshooting.jsp.
HP Software Support	The HP Software Support Web site. This site enables you to browse the Self-solve knowledge base. You can also post to and search user discussion forums, submit support requests, download patches and updated documentation, and more. The URL for this Web site www.hp.com/go/hpsoftwaresupport. • Most of the support areas require that you register as an HP Passport user and sign in. Many also require a support contract. • To find more information about access levels, go to: http://h20230.www2.hp.com/new_access_levels.jsp • To register for an HP Passport user ID, go to: http://h20229.www2.hp.com/passport-registration.html
HP Software Web site	The HP Software Web site. This site provides you with the most up-to-date information on HP Software products. This includes new software releases, seminars and trade shows, customer support, and more. The URL for this Web site is www.hp.com/go/software

Chapter 1: Introducing UFT .NET Add-in Extensibility

UFT .NET Add-in Extensibility enables you to provide high-level support for third-party and custom .NET Windows Forms controls that are not supported out-of-the-box by the UFT .NET Add-in.

This chapter includes:

About UFT .NET Add-in Extensibility

The UFT .NET Add-in provides support for a number of commonly used .NET Windows Forms controls. UFT .NET Add-in Extensibility enables you to support third-party and custom .NET Windows Forms controls that are not supported out-of-the-box by the .NET Add-in.

When UFT learns an object in an application, it recognizes the object as belonging to a specific test object class. This determines the identification properties and test object methods of the test object that represents the application's object in UFT.

Without extensibility, .NET Windows Forms controls that are not supported out-of-the-box are represented in UFT GUI tests by a generic **SwfObject** test object. This generic test object might be missing characteristics that are specific to the .NET Windows Forms control you are testing. Therefore, when you try to create test steps with this test object, the available test object methods might not be sufficient. In addition, when you record a test on controls that are not supported, the recorded steps reflect the low-level activities passed as Windows messages, rather than the meaningful behavior of the controls.

Using UFT .NET Add-in Extensibility, you can teach UFT to recognize custom .NET Windows Forms controls more specifically. When a custom control is mapped to an existing UFT test object, you have the full functionality of a UFT test object, including visibility when using the UFT statement completion feature and the ability to create more meaningful steps in the test.

Note: If UFT recognizes a .NET control out-of-the-box, and uses a .NET add-in test object other than SwfObject to represent it, then you cannot map this control to any other test object type.

The behavior of the existing test object methods might not be appropriate for the custom control. You can modify the behavior of existing test object methods, or extend UFT test objects with new methods that represent the meaningful behaviors of the control.

You develop a Custom Server that extends the .NET Add-in interfaces that run methods on the controls in the application. The Custom Server can override existing methods or define new ones.

Deciding When to Use .NET Add-in Extensibility

The UFT .NET Add-in provides a certain level of support for most .NET Windows Forms controls. Before you extend support for a custom .NET Windows Forms control, analyze it from a UFT perspective to view the extent of this support and to decide which elements of support you need to modify.

When you analyze the custom .NET Windows Forms control, use the .NET Windows Forms Spy, Keyword View, Editor, and the Record option. Make sure you examine each of the elements described in "Recognizing Which Elements of UFT Support Can Be Customized" on the next page.

If you are not satisfied with the existing object identification or behavior, your .NET Windows Forms control is a candidate for .NET Add-in Extensibility, as illustrated in the following situations:

- UFT recognizes your control as a generic SwfObject, but a different test object class exists with more appropriate behavior for your control. You can use .NET Add-in Extensibility to map the control to this test object class.
- UFT might recognize the control using a test object that does not fit your needs. You can use .NET Add-in Extensibility to instruct UFT to change the functionality of the test object by modifying its methods.
- UFT might identify individual sub-controls within your custom control, but not properly identify your main control. For example, if your main custom control is a digital clock with edit boxes containing the hour and minute digits, you might want changes in the time to be recognized as **SetTime** operations on the clock control and not as **Set** operations on the edit boxes. You can use .NET Add-in Extensibility to set a message filter to process messages from child controls, and record operations on the main control in response to events that occur on the controls it contains.

Recognizing Which Elements of UFT Support Can Be Customized

The following elements comprise UFT GUI testing support. By extending the existing support of one or more of these elements, you can develop the support you need to create meaningful and maintainable tests.

Test Object Classes

In UFT, every object in an application is represented by a test object of a specific test object class. The test object class determines the list of identification properties and test object methods available in UFT for this test object. You might want to instruct UFT to use a different test object class to represent your control.

Test Object Methods

The test object class used to represent the .NET Windows Forms control determines the list of test object methods for a test object. However, the same test object method might operate differently for different .NET Windows Forms controls represented by test objects from the same test object class. This happens because depending on the specific type of .NET Windows Forms control, UFT may have to perform the test object method differently.

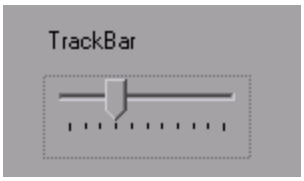
Recording Events

One way to create UFT GUI tests is by recording user operations on the application. When you start a recording session, UFT listens for events that occur on objects in the application and registers corresponding test steps. The test object class and Custom Server used to represent a .NET Windows Forms control determines which events UFT can listen for on the .NET Windows Forms control and what test step to record for each event that occurs.

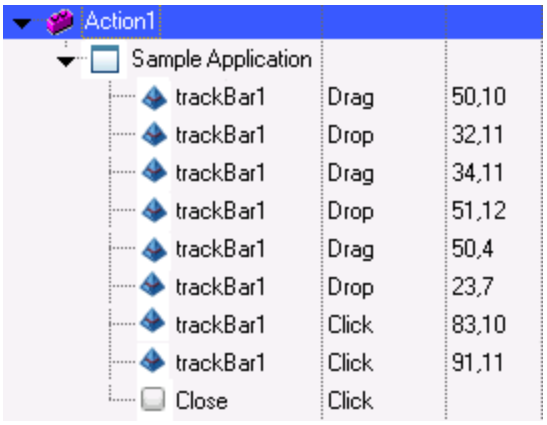
Example: Customizing Recording of an Event's Meaningful Behaviors

A control's meaningful behavior is the behavior that you want to test. For example, when you click a button in a radio button group in your application, you are interested in the value of the selection, not in the **Click** event and the coordinates of the click. The meaningful behavior of the radio button group is the change in the selection.

If you record a test or business component on a custom control without extending support for the control, you record the low-level behaviors of the control. For example, the TrackBar control in the sample .NET application shown below is a control that does not have a corresponding UFT test object.



If you record on the TrackBar without implementing support for the control, the Keyword View looks like this:

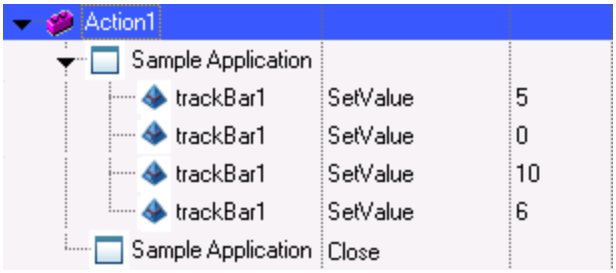


In the Editor, the recorded test looks like this:

```
SwfWindow("Sample Application").SwfObject("trackBar1").Drag 50,10
SwfWindow("Sample Application").SwfObject("trackBar1").Drop 32,11
SwfWindow("Sample Application").SwfObject("trackBar1").Drag 34,11
SwfWindow("Sample Application").SwfObject("trackBar1").Drop 51,12
SwfWindow("Sample Application").SwfObject("trackBar1").Drag 50,4
SwfWindow("Sample Application").SwfObject("trackBar1").Drop 23,7
SwfWindow("Sample Application").SwfObject("trackBar1").Click 83,10
SwfWindow("Sample Application").SwfObject("trackBar1").Click 91,11
SwfWindow("Sample Application").SwfButton("Close").Click
```

Note that the **Drag**, **Drop**, and **Click** methods—the low-level actions of the TrackBar control—are recorded at specific coordinates in the control display. These steps are difficult to understand and modify.

If you use .NET Add-in Extensibility to support the TrackBar control, the result is more meaningful. Below is the Keyword View of a test recorded on the TrackBar with a Custom Server that implements a customized **SetValue** method.



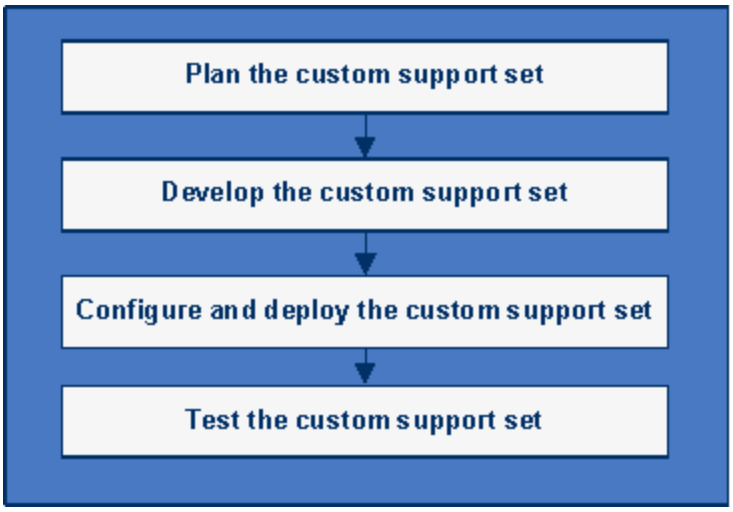
In the Editor, the recorded test looks like this:

```
SwfWindow("Sample Application").SwfObject("trackBar1").SetValue 5
SwfWindow("Sample Application").SwfObject("trackBar1").SetValue 0
SwfWindow("Sample Application").SwfObject("trackBar1").SetValue 10
SwfWindow("Sample Application").SwfObject("trackBar1").SetValue 6
SwfWindow("Sample Application").Close
```

UFT is now recording a **SetValue** operation reflecting the new slider position, instead of the low-level **Drag**, **Drop**, and **Click** operations recorded without the customized test object. You can understand and modify this test more easily.

Understanding How to Implement .NET Add-in Extensibility

You implement .NET Add-in Extensibility support for a set of custom controls by developing a .NET Add-in Extensibility support set. Developing a .NET Add-in Extensibility support set consists of the following stages, each of which is described below.



Planning the .NET Add-in Extensibility Support Set

Detailed planning of how you want UFT to recognize the custom controls enables you to correctly build the fundamental elements of the .NET Add-in Extensibility support set. Generally, to plan the support set, you:

- Determine the .NET Windows Forms controls for which you need to customize support.
- Plan the test object model by determining which test objects and operations you want to support based on the controls and business processes you need to test.
- Plan the most appropriate way for implementing the support.

For more information, see "Planning Your Support Set" on page 28.

Developing the .NET Add-in Extensibility Support Set

To develop a .NET Add-in Extensibility support set, you must:

- Define the test object model.
- Create Custom Servers.
- Map the custom controls to the relevant test object classes.

These activities are described in detail in the following sections:

Define The Test Object Model

Introduce the test object model that you want UFT to use to test your applications and controls. The test object model is a list of the test object classes that represent custom controls in your environment, and their test object methods.

You define the test object model in a test object configuration XML file. For more information, see "Describing the Test Object Model " on page 38.

Create Custom Servers

Create a Custom Server (DLLs or control definition XML file) to handle each custom control. In the Custom Server, you can modify:

- What steps are recorded during a recording session.
- The implementation of test object methods.
- Support for table checkpoints and output values.

The Custom Server mediates between UFT and the .NET application. During a recording session, the Custom Server listens to events and maps the user activities to meaningful test object methods. During a test run, the Custom Server performs the test object methods on the .NET Windows Forms control.

Custom Server Coding Options

The Custom Server can be implemented in one of the following coding options:

- .NET DLL
- XML, based on a schema (which UFT then uses to create a .NET DLL Custom Server behind the scenes)

For more information, see:

- "Using a .NET DLL to Extend Support for a Custom Control" on page 45
- "Using XML Files to Extend Support for a Custom Control" on page 63

Custom Server Run-time Contexts

Classes supplied by a Custom Server may be instantiated in the following software processes (run-time contexts):

- **Application under test** context: An object created in the context of the application you are testing has direct access to the .NET Windows Forms control's events, methods, and properties. However, it cannot listen to Windows messages.
- **UFT** context: An object created in the UFT context can listen to Windows messages. However, it does not have direct access to the .NET Windows Forms control's events, methods, and properties.

If the Custom Server is implemented as a .NET DLL, an object created under UFT can create assistant classes that run under the application you are testing.

For more details on run-time contexts, see "Selecting the Custom Server Run-Time Context Depending on the Test Function" on page 32.

For more information on assistant classes, see "Using a .NET DLL to Extend Support for a Custom Control" on page 45 and see the UFT .NET Add-in Extensibility API Reference.

Map the Custom Controls to the Relevant Test Objects

Map test objects using the .NET Add-in Extensibility configuration file (**SwfConfig.xml**). This file is located in the **<UFT installation path>\dat** folder and contains:

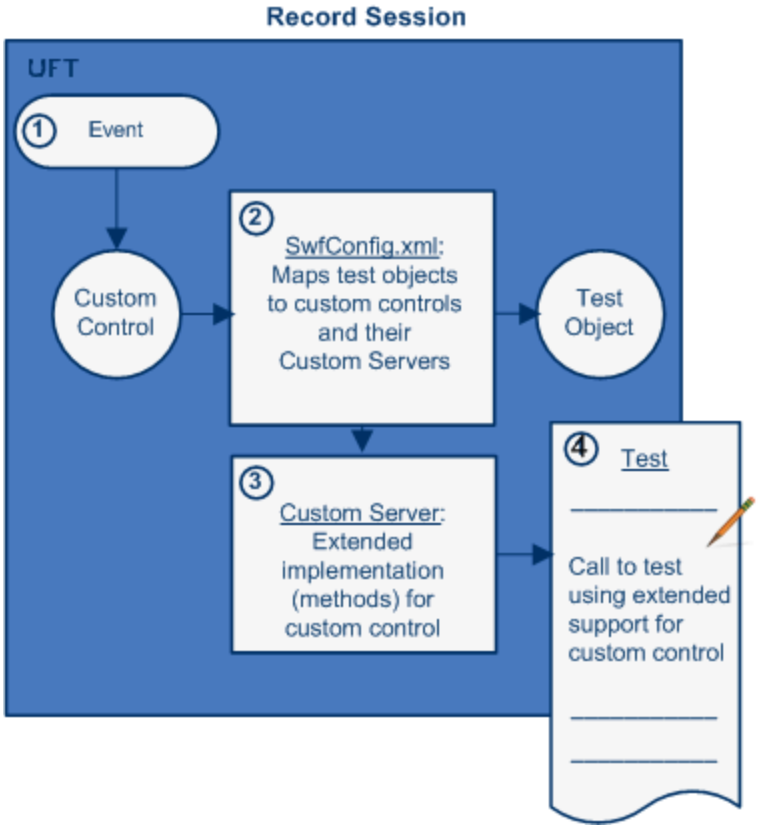
- The mapping of the custom controls to their corresponding test objects.
- The mapping to corresponding Custom Servers. This mapping provides the full functionality to UFT test objects.

For more information, see "Mapping Custom Controls to Test Object Classes" on page 44.

The illustrations below demonstrate how .NET Add-in Extensibility maps custom controls to their test objects and Custom Servers during recording sessions and run sessions.

How UFT Maps Custom Controls to Test Object Classes During Recording

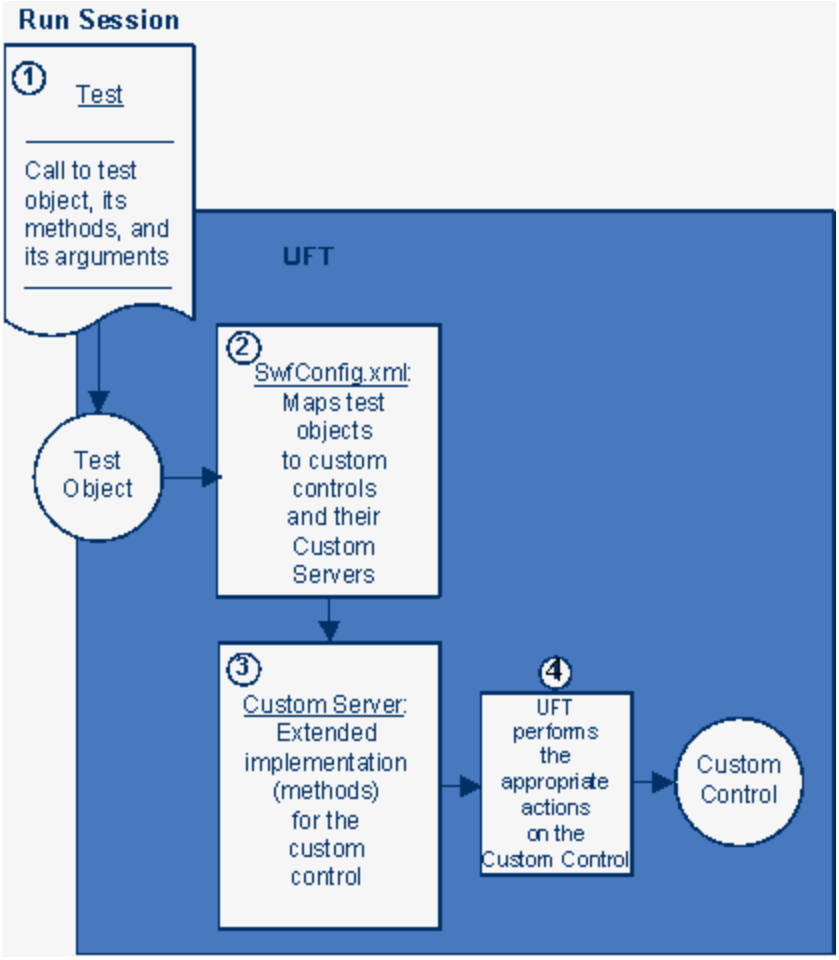
The following illustration and table explain how UFT maps custom controls to their test objects, locates the corresponding extended implementation for the custom control, and records an appropriate test step when recording.



Step	Description
1	An event occurs on a type of control that UFT does not recognize, or for which recording implementation is customized.
2	UFT checks the **Type** attribute of the **Control** elements in the **SwfConfig.xml** file to locate information for this type of custom control. UFT then checks the **MappedTo** attribute, to find the test object class mapped to this type of control. If no specific test object class is specified, **SwfObject** is used.
3	UFT checks the **DLLName** element in the **SwfConfig.xml** file to locate the Custom Server containing implementation for this type of custom control, and communicates with the Custom Server.
4	The Custom Server instructs UFT what step to add to the test in response to the event that occurred.

How UFT Maps Custom Controls to Custom Servers When Running a Test

The following illustration and table explain how UFT maps custom controls to their test objects, locates the corresponding extended implementation for the custom control, and performs the appropriate operations on a custom control when running a test.



Step	Description
1	A test runs. This test includes a test object representing a custom control whose implementation has been customized.
2	UFT locates the **Control** element in the **SwfConfig.xml** file that contains information for the custom control mapped to this test object.
3	UFT checks the **DLLName** element in the **SwfConfig.xml** file to locate the Custom Server containing implementation for the custom control.
4	UFT runs the test using the correct implementation for the test object operation as defined by the implementation of the custom control.

Deploying the .NET Add-in Extensibility Support Set

To deploy your .NET Add-in Extensibility support set and enable UFT to support your controls, copy the files you created to specific locations within the UFT installation folder.

For more information, see "Configuring and Deploying the Support Set" on page 68.

Testing the .NET Add-in Extensibility Support Set

After you have created the .NET Add-in Extensibility support for your controls, test your .NET Add-in Extensibility support set.

You can learn how to develop a .NET Add-in Extensibility support set hands-on, by performing the lessons in "Learning to Create Support for a Simple Custom .NET Windows Forms Control" on page 78 and "Learning to Create Support for a Complex Custom .NET Windows Forms Control" on page 88.

Chapter 2: Installing the HP UFT .NET Add-in Extensibility SDK

This chapter describes the installation process for the HP UFT .NET Add-in Extensibility SDK.

For a list of items that the HP UFT .NET Add-in Extensibility SDK installation provides, see "About the UFT .NET Add-in Extensibility SDK " on page 7.

This chapter includes:

Before You Install

Before you install the HP UFT .NET Add-in Extensibility SDK, review the following requirements:

- You must have access to the Unified Functional Testing installation DVD.
- A supported version of Microsoft Visual Studio must be installed on your computer.

> **Note:** For a list of supported Microsoft Visual Studio versions, see the *HP Unified Functional Testing Product Availability Matrix*, available from the UFT help folder or the HP Support Matrix page (requires an HP passport).

Installing the HP UFT .NET Add-in Extensibility SDK

Use the HP Unified Functional Testing Setup program to install the HP UFT .NET Add-in Extensibility SDK on your computer.

> **Note:** You must be logged on with Administrator privileges to install the UFT .NET Add-in Extensibility SDK.

To install the HP UFT .NET Add-in Extensibility SDK:

1. Close all instances of Microsoft Visual Studio.
2. Insert the Unified Functional Testing DVD into the CD-ROM/DVD drive. The Unified Functional Testing Setup window opens. (If the window does not open, browse to the DVD and double-click **setup.exe** from the root folder.)
3. Click **Add-in Extensibility and Web 2.0 Toolkits**. The Unified Functional Testing Add-in Extensibility and Web 2.0 Toolkit Support screen opens.
4. Click **UFT .NET Add-in Extensibility SDK Setup**. The UFT .NET Add-in Extensibility SDK Setup wizard opens.

   > **Note:** If the wizard screen that enables you to select whether to repair or remove the SDK installation opens, the UFT .NET Add-in Extensibility SDK is already installed on your computer. Before you can install a new version, you must first uninstall the existing one, as described in "Uninstalling the HP UFT .NET Add-in Extensibility SDK" on page 27.

5. Follow the instructions in the wizard to complete the installation.

6. In the final screen of the Setup wizard, if you select the **Show Readme** check box, the UFT .NET Add-in Extensibility Readme file opens after you click **Close**. The Readme file contains the latest technical and troubleshooting information. To open the Readme file at another time, select **Start > All Programs > HP Software > HP Unified Functional Testing > Extensibility > Documentation > .NET Add-in Extensibility Readme**.

   Note: When working in Windows 8, access UFT documentation and other files from the **Apps** screen.

7. Click **Close** to exit the Setup wizard.

8. If you use a non-English edition of Visual Studio, do the following to apply the installed **UFT CustomServer** project templates to your Visual Studio edition:

   Note: The following instructions apply to Visual Studio 2008 installed on a 32-bit operating system. The folder and file names are slightly different if you are working with Visual Studio 2010, or on a 64-bit operating system.

   a. Copy the **QuickTestCustomServerVB.zip** file from: **%ProgramFiles%\Microsoft Visual Studio 9.0\Common7\IDE\ProjectTemplates\VisualBasic\Windows\1033** (English language setting folder) to the folder relevant to the language you use (for example, use 1036 for French).

   b. Run the **PostCustomVizard.exe** program from the **%ProgramFiles%\Microsoft Visual Studio 9.0\Common7\IDE** folder.

   c. Repeat this process for the C# template, copying the **QuickTestCustomServer.zip** file from: **%ProgramFiles%\Microsoft Visual Studio 9.0\Common7\IDE\ProjectTemplates\CSharp\Windows\1033**

To confirm that the installation was successful:

Note: The Microsoft Visual Studio dialog box illustration and the instructions in this procedure refer to Microsoft Visual Studio 2008. If you use a different Microsoft Visual Studio version, the dialog box may differ slightly in appearance and the **UFT CustomServer** template may be located in a slightly different node in the tree.

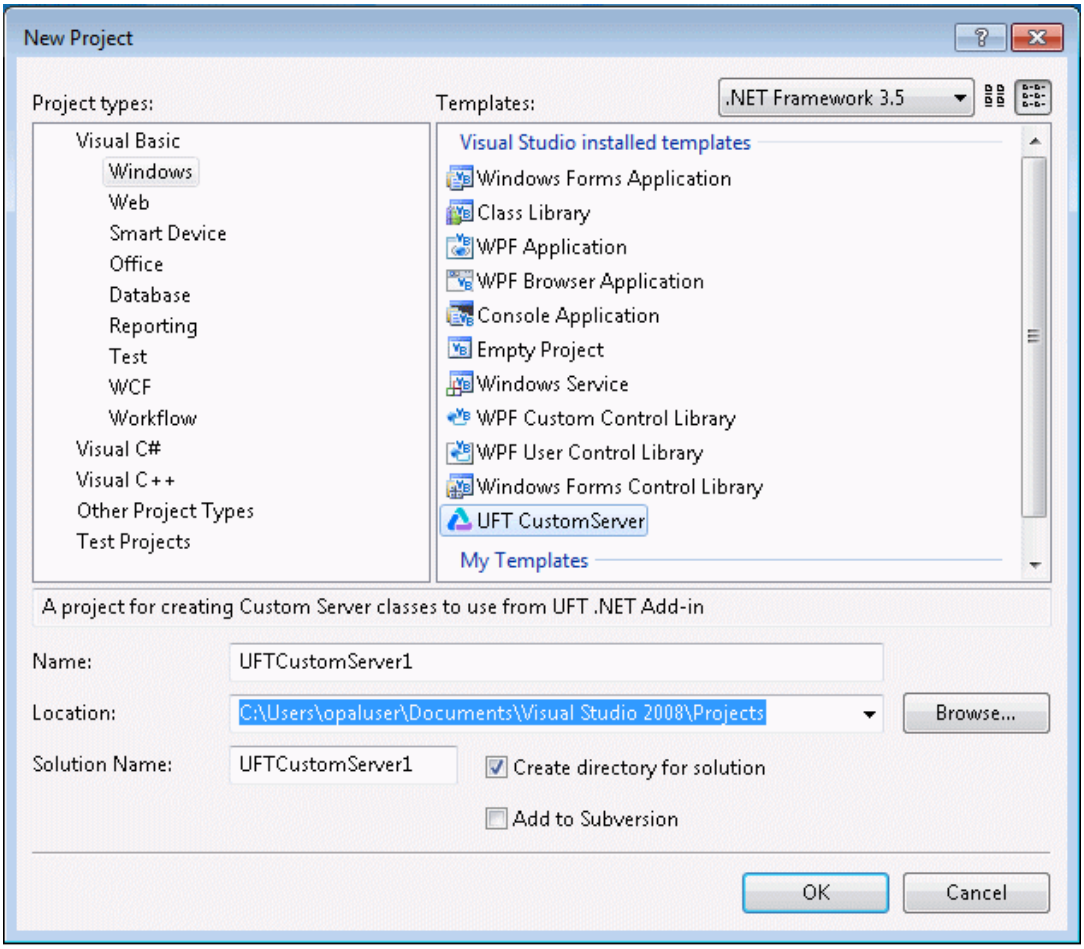
1. Open a supported version of Microsoft Visual Studio.

   For a list of supported Microsoft Visual Studio versions, see the *HP Unified Functional Testing Product Availability Matrix*, available from the UFT help folder or the HP Support Matrix page (requires an HP passport).

2. Select **File > New > Project** to open the New Project dialog box.

3. Select the **Visual Basic > Windows** node in the **Project types** tree.

4. Confirm that the **UFT CustomServer** template icon is displayed in the **Templates** pane:



5. Select the **Visual C# > Windows** node in the **Project types** tree.

6. Confirm that the **UFT CustomServer** template icon is displayed in the **Templates** pane.

> **Note:** If you upgrade to a new version of Microsoft Visual Studio, you must uninstall and reinstall the .NET Add-in Extensibility SDK to be able to access the **UFTCustomServer** template.

Repairing the HP UFT .NET Add-in Extensibility SDK Installation

You can use the Unified Functional Testing Setup program to repair an existing HP UFT .NET Add-in SDK installation by replacing any missing or damaged files from your previous installation.

Note:

- You must use the same version of the setup program as you used for the original installation.
- You must be logged on with Administrator privileges to repair the installation.
- If User Account Control (UAC) is available for your operating system, UAC must be turned off while you repair the installation.

To repair the HP UFT .NET Add-in Extensibility SDK installation:

1. Insert the Unified Functional Testing DVD into the CD-ROM/DVD drive. The Unified Functional Testing Setup window opens. (If the window does not open, browse to the DVD and double-click **setup.exe** from the root folder.)
2. Click **Add-in Extensibility and Web 2.0 Toolkits**. The Unified Functional Testing Add-in Extensibility and Web 2.0 Toolkit Support screen opens.
3. Click **UFT .NET Add-in Extensibility SDK Setup**. The .NET Add-in Extensibility SDK Setup wizard opens, enabling you to select whether to repair or remove the SDK installation.
4. Select **Repair** and click **Finish**. The setup program replaces the UFT .NET Add-in Extensibility SDK files and opens the Installation Complete screen.
5. In the Installation Complete screen, click **Close** to exit the Setup wizard.

Uninstalling the HP UFT .NET Add-in Extensibility SDK

You can uninstall the HP UFT .NET Add-in SDK by using **Add/Remove Programs** as you would for other installed programs. Alternatively, you can use the Unified Functional Testing Setup program.

Note:

- You must use the same version of the setup program as you used for the original installation.
- You must be logged on with Administrator privileges to uninstall the UFT .NET Add-in Extensibility SDK.

To uninstall the HP UFT .NET Add-in Extensibility SDK:

1. Insert the Unified Functional Testing DVD into the CD-ROM/DVD drive. The Unified Functional Testing Setup window opens. (If the window does not open, browse to the DVD and double-click **setup.exe** from the root folder.)
2. Click **Add-in Extensibility and Web 2.0 Toolkits**. The Unified Functional Testing Add-in Extensibility and Web 2.0 Toolkit Support screen opens.
3. Click **UFT .NET Add-in Extensibility SDK Setup**. The .NET Add-in Extensibility SDK Setup wizard opens, enabling you to select whether to repair or remove the SDK.
4. Select **Remove** and click **Finish**. The setup program removes the UFT .NET Add-in Extensibility SDK and opens the Installation Complete screen.
5. In the Installation Complete screen, click **Close** to exit the Setup wizard.

Chapter 3: Planning Your Support Set

Before you begin to create support for custom controls, you must carefully plan the support. Detailed planning of how you want UFT to recognize the custom controls enables you to correctly build the fundamental elements of the .NET Add-in Extensibility support set.

This chapter includes:

About Planning UFT GUI Testing Support for Your .NET Add-in Extensibility Controls

Extending the UFT .NET Add-in's support to recognize custom .NET Windows Forms controls is a process that requires detailed planning. To assist you with this, the sections in this chapter include sets of questions related to the implementation of support for your custom controls. When you create your .NET Add-in Extensibility support set, you implement it based on the answers you provide to these questions.

Determining Information Related to Your Custom Controls

Decide which controls this support set will support.

Before you begin planning support for custom .NET Windows Forms controls, make sure you have full access to the controls and understand their behavior.

You must have an application in which you can view the controls in action.

You must also be able to view the source that implements them. You do not need to modify any of a custom control's sources to support it in UFT, but you do need to be familiar with them.

When planning custom support for a specific type of control, carefully consider how you want UFT to recognize controls of this type—what type of test object you want to represent the controls in UFT GUI tests, which test object methods you want to use, and so on. Make these decisions based on the business processes that might be tested using this type of control and operations that users are expected to perform on these controls:

- Make sure you know the methods the control supports, what properties it has, the events for which you can listen, and so on.
- Identify existing test object classes whose functionality is similar to that of the custom .NET Windows Forms controls.
- Decide what methods need to be written or modified for supporting the controls.

Analyzing the Custom Controls

You can run an application containing the custom control and analyze the control from a UFT perspective using the .NET Windows Forms Spy, the Keyword View, and the Record option. This enables you to see how UFT recognizes the control without custom support, and helps you to determine what you want to change.

Using the .NET Windows Forms Spy

You can use the .NET Windows Forms Spy to help you develop extensibility for .NET Windows Forms controls. The .NET Windows Forms Spy enables you to:

- View details about selected .NET Windows Forms controls and their run-time object properties.
- See which events cause your application to change (to facilitate record and run extensibility implementation) and how the changes manifest themselves in the control's state.

You access the .NET Windows Forms Spy by choosing **Tools > .NET Windows Forms Spy** in the main UFT window.

Note: To spy on a .NET Windows Forms application, make sure that the application is running with Full Trust. If the application is not defined to run with Full Trust, you cannot spy on the application's .NET Windows Forms controls with the .NET Windows Forms Spy. For information on defining trust levels for .NET applications, see the relevant Microsoft documentation.

For more information on the .NET Windows Forms Spy, see the *HP Unified Functional Testing Add-ins Guide*.

When you plan the support for a specific control, you must ask yourself a series of questions. You can find a list of these questions in "Using the .NET Add-in Extensibility Planning Checklist" on page 35. When you are familiar with the questions and you are designing your own custom support classes, you can use the abbreviated, printable checklist on page 36.

Selecting the Coding Option for Implementing the Custom Servers

You can implement custom support for custom .NET Windows Forms controls in the following ways:

- **.NET DLL.** Extends support for the control using a .NET Assembly.
- **XML.** Extends support for the control using an XML file, based on a schema.

.NET DLL: Full Program Development Environment

Most Custom Servers are implemented as a .NET DLL. This option is generally preferred because:

- Development is supported by all the services of the program development environment, such as syntax checking, debugging, and Microsoft IntelliSense.
- If table checkpoint and output value support is needed, this support is available only when implementing the Custom Server as a .NET DLL.
- A Custom Server implemented as a .NET DLL can perform part of its Test Record functions in the UFT context and part in the context of the application being tested. For more information, see "Using a .NET DLL to Extend Support for a Custom Control" on page 45, and the *UFT .NET Add-in Extensibility API Reference* (available in the UFT .NET Add-in Extensibility online Help.)

For information on run-time contexts, see "Selecting the Custom Server Run-Time Context Depending on the Test Function" on the next page.

XML Implementation

There are circumstances when it is most practical to implement Custom Servers using the XML coding method. These circumstances include:

- When the controls are relatively simple and well documented.
- When the controls map well to an existing object, but you need to replace the implementation during a recording session (Test Record), or replace or add a small number of test object methods during a run session (Test Run).
- When a full programming environment is not available–implementation using XML Custom Servers requires only a text editor.

However, when implementing a custom control with XML:

- You have none of the support provided by a program development environment.
- The XML implementation includes C# programming commands, and runs only in the **Application under test** context.

For more information, see "Using XML Files to Extend Support for a Custom Control" on page 63.

Selecting the Custom Server Run-Time Context Depending on the Test Function

Each Custom Server may implement the following test functions for each control:

- Test Record
- Test Run
- Table Verification (to support checkpoints and output values)
- A combination of these test functions

Run-time contexts include:

- **Application under test**: The context of the application which is being tested.
- **UFT**: The UFT context.

The following table provides guidelines for determining which test function you can implement for each run-time context.

Need / Task	Test Record	Test Run	Table Verification	Run-Time Context	Explanation
Create tasks using keyword-driven testing (and not by recording steps on an application)	Not applicable	Yes	Only for .NET DLL Custom Servers	Either **Application under test** or **UFT**	The Test Record test function records the actions performed on the application being tested and the application's resulting behaviors. The recording is then converted to a test. If you plan to create GUI tests using keyword-driven testing, and not by recording steps on an application, you do not need to implement the Test Record function.

Need / Task	Test Record	Test Run	Table Verification	Run-Time Context	Explanation
Implement the Custom Server in the **Application under test** context	Optional	Optional (usually)	Only for .NET DLL Custom Servers	**Application under test**	The Test Run function tests if the application is performing as required by running the test and tracking the results. Test Run is nearly always implemented in the **Application under test** context.
Listen to Microsoft Windows messages	Yes	Only with assistant classes	Only for .NET DLL Custom Servers	**UFT**	If the .NET DLL Custom Server must both listen to Windows messages and access control events and properties, use assistant classes. The Custom Server running in the **UFT** context can listen to events in the **Application under test** context with assistant class objects that run in the **Application under test** context. These objects also provide direct access to control properties.
Implement table checkpoints and output values on custom grid controls	Optional	Optional	Only for .NET DLL Custom Servers	Either **Application under test** or **UFT**	You can implement support for table checkpoints and output values on custom grid controls, regardless of the context in which your .NET DLL runs.
Your application uses UFT services more than it uses services of the custom control	Yes, but possibly less efficient	Possibly more efficient	Possibly more efficient	**UFT** is preferred	There is no need to listen to Windows messages during a Test Run session, so the **UFT** context is not required. However, if your application uses **UFT** services more than it uses services of the custom control, it may be more efficient to implement Test Run in the **UFT** context.

Analyzing Custom Controls and Mapping Them to Test Objects

When you develop .NET Add-in Extensibility, you map custom .NET Windows Forms controls to existing UFT .NET Add-in test object classes and to Custom Servers that you develop.

The first mapping determines the test object class that UFT uses to represent the custom control. The second specifies the Custom Server to use. The Custom Server extends the functionality of the test object that is used for the control to match the control's functionality.

If UFT recognizes a .NET control out-of-the-box, and uses a .NET add-in test object other than SwfObject to represent it (for example SwfEdit or SwfList), then you cannot map this control to any other test object type. However, you can still map it to a Custom Server and extend the test object functionality.

Mapping Custom Controls to Test Objects

Map the custom controls to test objects by using the **MappedTo** attribute in the UFT .NET Add-in Extensibility's System Windows Forms configuration file (**SwfConfig.xml**). Map each custom control to a UFT test object class containing behaviors that are similar to those required to support your control.

If you do not specify a mapping, UFT maps the custom control to the default generic test object, **SwfObject**. For more information on **SwfConfig.xml**, see "Understanding How to Configure UFT Windows Forms Extensibility" on page 69.

Note: Mapping is sometimes sufficient without any programming. If the existing UFT test object adequately covers a control, it is sufficient to map the control to the UFT test object.

Mapping Custom Controls to Custom Servers

When you map your control to a functionally similar UFT test object, then, in your Custom Server, you do not need to override test object methods that apply without change to your custom control. For example, most controls contain a **Click** method. If the **Click** method of the existing test object implements the **Click** method of the custom control adequately, you do not need to override the existing object's method.

To cover the Test Run functionality of the custom object that does not exist in the existing object, add new methods in your Custom Server. To cover functionality that has the same method name, but a different implementation, override the existing object's methods.

If the UFT test object adequately covers Test Record, but you need to customize Test Run, do not implement Test Record. If you do implement Test Record, the implementation replaces that of the existing object. You must implement all required Test Record functionality.

In UFT, when you edit a step with the test object that you customized to support the custom control, the statement completion feature displays the custom properties and methods that you defined for the test object, in addition to those that exist in UFT. UFT uses test object configuration files to provide the list of custom test object methods and properties.

Using the .NET Add-in Extensibility Planning Checklist

When you plan the support for a specific type of control, you must ask yourself a series of questions. These are explained below and are available in an abbreviated, printable checklist on page 36.

1. Make sure you have access to an application that runs the custom control on a computer with UFT installed.

2. Choose a .NET Windows Forms test object class to represent the custom control. (UFT uses **SwfObject** by default)

3. Does the test object class you selected have to be customized?

   a. Specify any test object methods that you want to add to the test object definition. Specify the method argument types and names, and whether the method returns a value in addition to the return code.

      When you design the .NET Add-in Extensibility support set, you specify this information in the test object configuration file.

   b. Specify any test object methods whose behavior you want to modify or override.

      When you design the .NET Add-in Extensibility Custom Server, you will need to implement any new test object methods that you add, or any test object methods whose existing behavior you want to override.

4. Should test objects of this class be displayed in the .NET Windows Forms Spy? (By default they are.)

5. Are you going to provide support for recording? If so, list the events that should trigger recording.

6. If you are creating support for a table control, decide whether you want to provide support for table checkpoints and output values on this control.

.NET Add-in Extensibility Planning Checklist

Use this checklist to plan the support for your custom control:

☑	Custom Control Support Planning Checklist	Specify in Test Object Config. file?	Specify in .NET Add-in Extensibility configuration file?	Specify in Custom Server?
❑	The sources for this custom control are located in:			
❑	Specify the .NET test object class to map to the control: (Default—**SwfObject**)			
❑	Specify the test object methods you want to add or modify (if required, include arguments, and return values):			
❑	Display test objects of this class in the .NET Windows Forms Spy? **Yes** (default)**/No**			
❑	Provide support for recording? **Yes/No** If so, list the events that should trigger recording:			
❑	Provide support for table checkpoints and output values? **Yes/No**			

Where Do You Go from Here?

After you finish planning the custom control support, you create the .NET Add-in Extensibility support set."Developing Your Support Set" on page 37 explains how to develop the .NET Add-in Extensibility support set.

Chapter 4: Developing Your Support Set

This chapter explains how to develop extended support for custom .NET Windows Forms controls. It explains which files you have to create for a .NET Add-in Extensibility support set, the structure and content of these files, and how to develop them to support the different UFT capabilities for your environment.

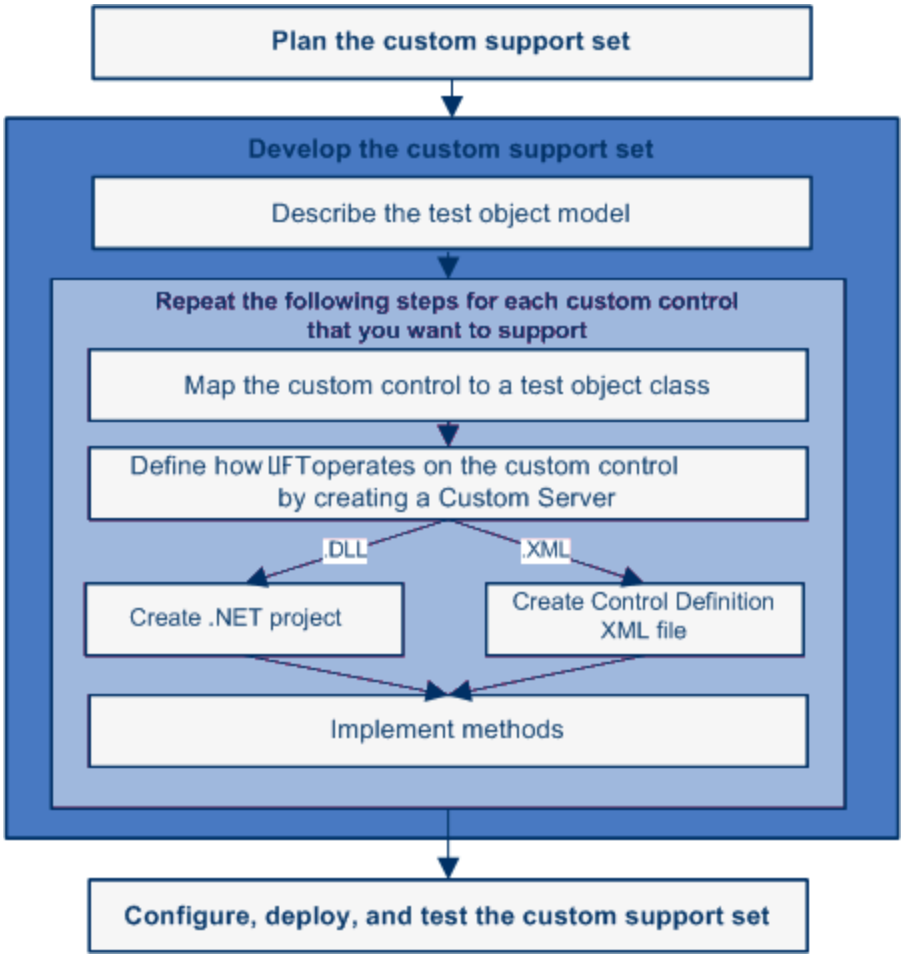
Note: Before you actually begin to create a support set, you must plan it carefully. For more information, see "Planning Your Support Set" on page 28.

For information on where the .NET Add-in Extensibility support set files should be stored to activate the support you design, see "Configuring and Deploying the Support Set" on page 68.

This chapter includes:

Understanding the Development Workflow

Implementing the .NET Add-in Extensibility support set consists of the following stages. The workflow for developing the support set is described in the following sections.



Describing the Test Object Model

The first stage of developing support for custom controls is to introduce the test object model that you want UFT to use to test your applications and controls. The test object model is a list of the test object classes that represent custom controls in your environment and the syntax of the test object methods that support the custom controls.

You define the test object model in a test object configuration file according to a specific XML schema. For details about how to create test object configuration files, see "Creating Test Object Configuration Files" on the next page.

Benefits of Describing Test Object Models

Implementation of a test object configuration file is optional. If you choose not to implement the test object configuration file, the test object methods defined in the .NET Custom Server DLL or control definition files will work as expected, but the functionality listed below will be missing.

Describing your custom test object methods in a test object configuration file enables the following functionality when editing GUI tests in UFT:

- A list of available custom test object methods in the **Operations** column in the Keyword view and when using the statement completion feature in the Editor.
- A test object method selected by default in the Keyword View and Step Generator when a step is generated for a test object of this class.
- Documentation for the custom test object methods in the **Documentation** column in the Keyword view.
- Icons and context-sensitive Help (only for new test object methods added to a test object class).

Creating Test Object Configuration Files

The following steps describe how to create test object configuration files.

To create test object configuration files:

1. Create a copy of the **<UFT installation folder>\dat\Extensibility\DotNet\ DotNetCustomServerMethods.xml** file to create a new test object configuration file in the same folder. (Do not modify the original file.)
2. Edit the new test object configuration file, modifying any test object classes whose behavior you want to modify. Delete any test object classes that you do not modify.
3. Save and close the test object configuration file.

For more information, see:

- "Understanding the Contents of the Test Object Configuration File" on the next page
- "Modifying an Existing Test Object Class" on page 41
- " Make Sure that Test Object Configuration File Information Matches Custom Server Information" on page 41
- " Implementing More Than One Test Object Configuration File" on page 42
- "Example of a Test Object Configuration File" on page 43
- UFT .NET Add-in Extensibility API Reference

Understanding the Contents of the Test Object Configuration File

A test object configuration file can include the following:

- The name of the test object class and its attributes.
- The name of the custom control for which this test object class definition is relevant.
- The methods for the test object class, including the following information for each method:
  - The arguments, including the argument type and direction.
  - Whether the argument is mandatory, and, if not, its default value.
  - The description (shown as a tooltip in the Keyword View, Editor, and Step Generator).
  - The documentation string (shown in the **Documentation** column of the Keyword View and in the Step Generator).
  - A context-sensitive Help topic to open when F1 is pressed for the test object method in the Keyword View or Editor, or when the **Operation Help** button is clicked for the method in the Step Generator. The definition includes the Help file path and the relevant Help ID within the file. (Relevant only for new test object methods added to the test object class.)
  - The return value type.
- The test object method that is selected by default in the Keyword View and Step Generator when a step is generated for a test object of this class.

The following example shows parts of the **SwfObject** test object class definition in a test object configuration file. The example shows that the **SwfObject** is extended by adding a **MyCustomButtonSet** method. The method has one argument (**Percent**, which defines the percentage to set in the control), and it also has a documentation string that appears in the Keyword View:

```
</TypeInformation>
...
    <ClassInfo BaseClassInfoName="SwfObject" Name="MyCompany.MyButton">
...
        <TypeInfo>
            <Operation Name="MyCustomButtonSet"
              PropertyType="Method" ExposureLevel="CommonUsed">
                <Description>Set the percentage in the task
bar</Description>
                <Documentation><![CDATA[Set the %l %t to <Percent>
percent.]]></Documentation>
```

```
                    <Argument Name="Percent" IsMandatory="true"
Direction="In">
                        <Type VariantType="Integer"/>
                        <Description>The percentage to set in the task
bar.</Description>
                    </Argument>
                </Operation>
            </TypeInfo>
        </ClassInfo>
</TypeInformation>
```

For information on the structure and syntax of a test object configuration file, see the *HP UFT Test Object Schema Help* (available with the UFT .NET Add-in Extensibility online help).

Modifying an Existing Test Object Class

Identify a test object class that provides partial support for your control, but needs some modification, for example, additional or modified test object methods.

You can then extend the functionality of this test object by defining and implementing additional test object methods. In addition, you can override existing test object methods by providing an alternate implementation for them. You define the new or changed methods in the test object configuration file, and design their implementation using Custom Servers.

Adding Test Object Methods to an Existing Test Object Class

When you create a test object class definition in the test object configuration file, you specify the custom control for which this definition is relevant. (In the **ClassInfo** element, you specify the test object class in the **BaseClassInfoName** attribute, and the name of the custom class in the **Name** attribute.)

If you then add a custom test object method to the definition of this test object class, this method is available in UFT only for test objects that represent custom controls of the type you specified.

For example, if you added a **Set** method to the SwfEditor test object class when used for **MyCompany.MyButton** controls, then the method is displayed in the statement completion list of test object methods in UFT only for objects that represent such controls. When SwfEditor test objects are used for other types of controls, this method will not be available.

Make Sure that Test Object Configuration File Information Matches Custom Server Information

Make sure that the information you define in the test object configuration file exactly matches the corresponding information defined in the .NET Custom Server DLL or control definition files. For example, the test object method names must be exactly the same in both locations. Otherwise, the methods will appear to be available (for example, when using the statement completion feature) but

they will not work, and, if used, the run session will fail. In addition, the custom control name specified in the test object configuration file must be the same as the name specified in the .NET Add-in Extensibility configuration file.

Implementing More Than One Test Object Configuration File

You can choose to implement one or multiple test object configuration files (or none, if not needed). For example, you can define custom methods for one test object class in one test object configuration file, and custom methods for another test object in a different test object configuration file. You can also choose to define a group of custom methods for a test object class in one test object configuration file, and another group of custom methods for the same test object class in a different test object configuration file.

Each time you open UFT, it reads all of the test object configuration files and merges the information for each test object class from the different files into a single test object class definition. This enables you to use the same test object class definitions when supporting different custom toolkits.

Understanding How UFT Merges Test Object Configuration Files

Each time you open UFT, it reads all of the test object configuration files located in the **<UFT installation folder>\dat\Extensibility\<UFT add-in name>** folders. UFT then merges the information for each test object class from the different files into a single test object class definition, according to the priority of each test object configuration file.

UFT ignores the definitions in a test object configuration file in the following situations:

- The **Load** attribute of the **TypeInformation** element is set to `false`.
- The environment relevant to the test object configuration file is displayed in the Add-in Manager dialog box, and the UFT user selects not to load the environment.

Define the priority of each test object configuration file using the **Priority** attribute of the **TypeInformation** element.

If the priority of a test object configuration file is higher than the existing class definitions, it overrides any existing test object class definitions, including built-in UFT information. For this reason, be aware of any built-in functionality that will be overridden before you change the priority of a test object configuration file.

When multiple test object class definitions exist, UFT must handle any conflicts that arise. The following sections describe the process UFT follows when **ClassInfo**, **ListOfValues**, and **Operation** elements are defined in multiple test object configuration files. All of the **IdentificationProperty** elements for a specific test object class must be defined in only one test object configuration file.

ClassInfo Elements

- If a **ClassInfo** element is defined in a test object configuration file with a priority higher than the existing definition, the information is appended to any existing definition. If a conflict arises between **ClassInfo** definitions in different files, the definition in the file with the higher priority overrides (replaces) the information in the file with the lower priority.
- If a **ClassInfo** element is defined in a test object configuration file with a priority that is equal to or lower than the existing definition, the differing information is appended to the existing definition. If a conflict arises between **ClassInfo** definitions in different files, the definition in the file with the lower priority is ignored.

ListOfValues Elements

- If a conflict arises between **ListOfValues** definitions in different files, the definition in the file with the higher priority overrides (replaces) the information in the file with the lower priority (the definitions are not merged).
- If a **ListOfValues** definition overrides an existing list, the new list is updated for all arguments of type **Enumeration** that are defined for operations of classes in the same test object configuration file.
- If a **ListOfValues** is defined in a configuration file with a lower priority than the existing definition, the lower priority definition is ignored.

Operation Elements

- **Operation** element definitions are either added, ignored, or overridden, depending on the priority of the test object configuration file.
- If an **Operation** element is defined in a test object configuration file with a priority higher than the existing definition, the operation is added to the existing definition for the class. If a conflict arises between **Operation** definitions in different files, the definition in the file with the higher priority overrides (replaces) the definition with the lower priority (the definitions are not merged).

For more information, see the *HP UFT Test Object Schema Help* (available with the .NET Add-in Extensibility SDK Help).

Example of a Test Object Configuration File

The following example shows the definition of the **ToolStrip** test object:

```
<ClassInfo Name="System.Windows.Forms.ToolStrip"
aseClassInfoName="SwfToolBar" FilterLevel="1">
    <TypeInfo>
        <Operation Name="Select" PropertyType="Method
ExposureLevel="CommonUsed" SortLevel="-1">
            <Description>Selects a menu item from a SwfToolBar dropdown menu.
```

```
            </Description>
            <Argument Name="Item" Direction="In" IsMandatory="true">
                <Type VariantType="VT_BSTR"/>
            </Argument>
        </Operation>
        <Operation Name="IsItemEnabled" PropertyType="Method"
ExposureLevel="Expert" SortLevel="-1">
            <Description>Indicates whether the toolbar item is
enabled.</Description>
            <Argument Name="Item" Direction="In" IsMandatory="true">
                <Type VariantType="VT_BSTR"/>
            </Argument>
            <ReturnValueType><Type VariantType="VT_BOOL"/></ReturnValueType>
        </Operation>
        <Operation Name="ItemExists" PropertyType="Method"
ExposureLevel="Expert" SortLevel="-1">
            <Description>Indicates whether the specified toolbar item
exists.</Description>
            <Argument Name="Item" Direction="In" IsMandatory="true">
                <Type VariantType="VT_BSTR"/>
            </Argument>
            <ReturnValueType> <Type VariantType="VT_BOOL"/></ReturnValueType>
        </Operation>
    </TypeInfo>
</ClassInfo>
```

This example shows that the **ToolStrip** test object class extends the **SwfToolBar** test object class. The default test object method for the **ToolStrip** test object class is **Select** (which has one mandatory input parameter: **Item**).

Mapping Custom Controls to Test Object Classes

The mapping of custom controls to test object classes is defined in the .NET Add-in Extensibility configuration file, **SwfConfig.xml**, in the **<UFT installation path>\dat** folder. This XML file describes which test object class represents each custom control and where UFT can locate the information it needs to interact with each control. For more information on mapping, see "Configuring UFT to Use the Custom Server" on page 69.

Defining How UFT Operates on the Custom Controls

After enabling UFT to recognize the custom controls, you must provide support for running test object methods. If you try to run a test with steps that run custom test object methods before providing implementation for these methods, the test fails and a run-time error occurs.

Custom Servers contain the implementation for how UFT interacts with the custom controls. Custom Servers can be .DLL files or .XML files (which UFT converts to .DLL files "behind the scenes" when necessary). For instructions on deciding when it is appropriate to use each method, see "Planning Your Support Set" on page 28.

- Most implementations are developed using DLL files. For more information, see "Using a .NET DLL to Extend Support for a Custom Control" below.
- Simpler implementations can be developed using the XML files, by creating a Control Definition file for each custom control. For more information, see "Using XML Files to Extend Support for a Custom Control" on page 63.

After creating the Custom Server, configure UFT to use it. For more information, see "Configuring UFT to Use the Custom Server" on page 69.

Using a .NET DLL to Extend Support for a Custom Control

You can support a .NET Windows Forms control by creating a Custom Server implemented as a .NET DLL. Set up a .NET project in Microsoft Visual Studio as a .NET DLL class library that implements the interfaces for a combination of:

- Test Record (IRecord interface)
- Test Run (Replay interface)
- Table verification (supports checkpoints and output values)

> **Note:** The IRecord interface is provided in the UFT .NET Add-in Extensibility SDK. When running the UFT Custom Server Settings wizard to create a .NET DLL Custom Server, the wizard provides code that implements the IRecord interface to get you started.
>
> The SDK does not provide interfaces for replay and table verification. You must implement these.
>
> For details, see "Implementing the IRecord Interface" on page 52 and the *UFT .NET Add-in Extensibility API Reference* (available in the UFT .NET Add-in Extensibility online Help.)

To create a .NET DLL Custom Server you need to know how to program a .NET Assembly. The illustrations and instructions in this section assume that you are using Microsoft Visual Studio 2008 as your development environment and that the Custom Server Project Templates are installed. For more information, see "Installing the HP UFT .NET Add-in Extensibility SDK" on page 22.

Considerations for Working with Custom Server DLLs

- The Custom Server DLL that you design is loaded into the 32-bit UFT application, and into the application you are testing. Therefore, to enable your support to work with 64-bit applications, you must build your custom server DLLs with the **Platform target** option set to **Any CPU**.

- Applications running under .NET Framework version 1.1 cannot use DLLs that were created using Visual Studio 2005 or above. Therefore you cannot use a Custom Server that you implemented as a .NET DLL using Visual Studio 2005 or above when you run the application you are testing under .NET Framework version 1.1.
- UFT loads the Custom Server when you open a test. Therefore, if you implement your Custom Server as a .NET DLL, any changes you make to the DLL after the Custom Server is loaded take effect only the next time you open a test.

Designing the Custom Server DLL

Implementing your Custom Server as a .NET DLL involves the following tasks:

- "Setting up the .NET Project" (described on page 46)
- "Implementing Test Record for a Custom Control Using a .NET DLL" (described on page 52)
- " Implementing Test Run for a Custom Control Using the .NET DLL" (described on page 56)
- "Implementing Support for Table Checkpoints and Output Values in the .NET DLL Custom Server" (described on page 56)
- "Running Code under Application Under Test from the UFT Context" (described on page 61)

Setting up the .NET Project

Set up a .NET project in Microsoft Visual Studio using the Custom Server C# Project Template or the Custom Server Visual Basic Project Template. (This template is installed automatically during the installation, as described in "Installing the HP UFT .NET Add-in Extensibility SDK" on page 22).

When you set up the .NET project, the Custom Server Project template does the following:

- Creates the project files necessary for the build of the .DLL file.
- Sets up a C# or Visual Basic file (depending on which template you selected) with commented code that contains the definitions of methods that you can override when you implement Test Record or Test Run.
- Provides sample code that demonstrates some Test Record and Test Run implementation techniques.
- Creates an XML file segment with definitions for the Custom Server that you can copy into the .NET Add-in Extensibility configuration file (**SwfConfig.xml**).

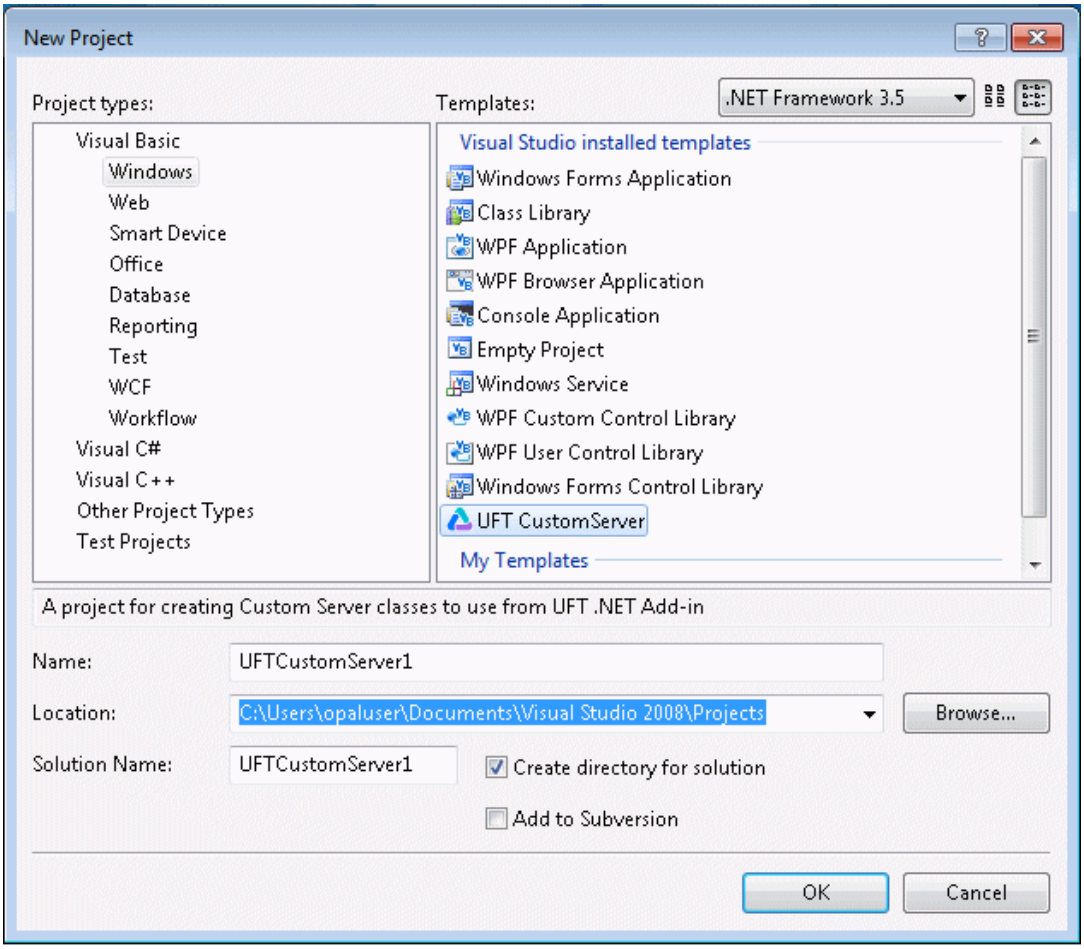
To set up a new .NET project:

Caution: To use the Custom Server Project template to create a .NET project, you must have either administrator privileges or full read and write access to the following folder and all of its sub-folders: **<Microsoft Visual Studio installation folder>\VC#\VC#Wizards**

1. Start Microsoft Visual Studio.

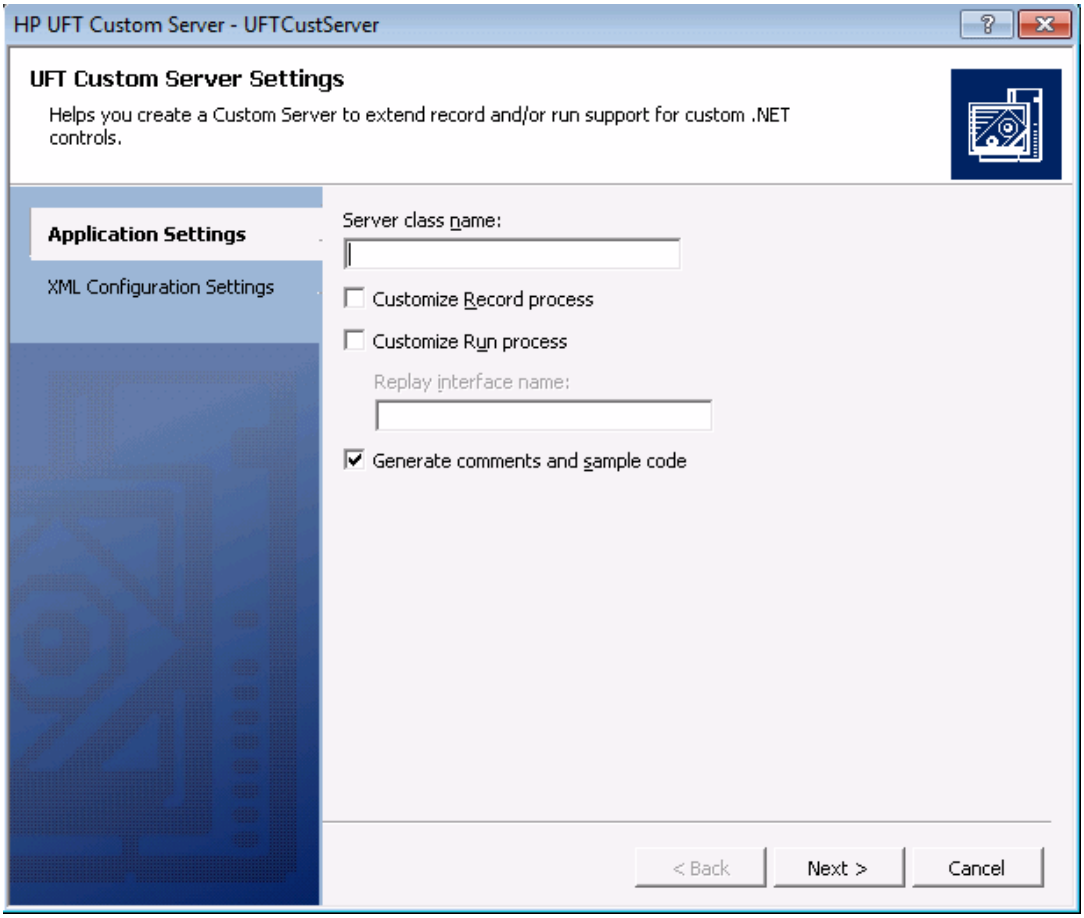
2. Select **File > New > Project** to open the New Project dialog box, or press CTRL + SHIFT + N. The New Project dialog box opens.

3. Select the **Visual C# > Windows** or **Visual Basic > Windows** node in the **Project types** tree.



Note: In Microsoft Visual Studio versions other than 2008, the dialog box may differ slightly in appearance and the **UFT CustomServer** template may be located in a slightly different node in the tree.

4. Select the **UFT CustomServer** template in the Templates pane. Enter the name of your new project and the location in which you want to save the project. Click **OK**. The UFT Custom Server Settings wizard opens.



5. Determine whether your Custom Server will extend Test Record support, Test Run support, or both, by making selections in the Application Settings page of the wizard.

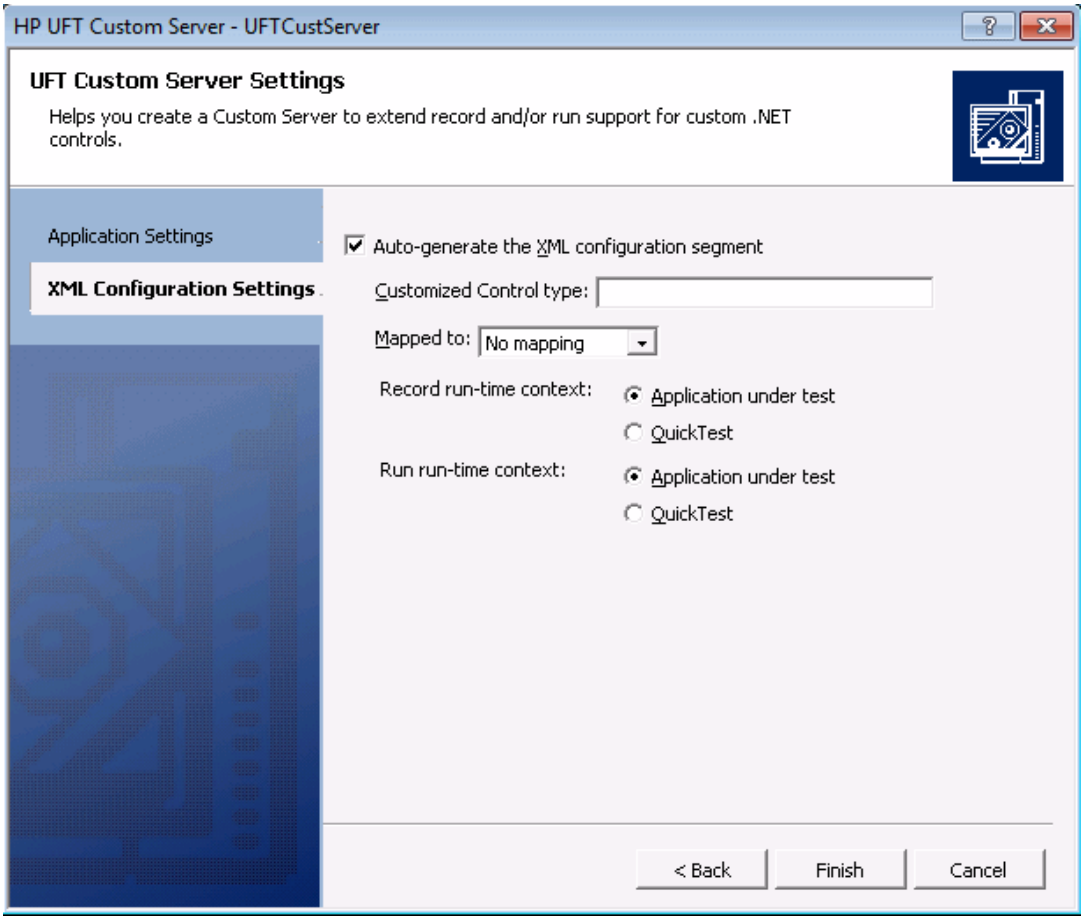
   - In the **Server class name** box, provide a descriptive name for your Custom Server class.
   - Select the **Customize Record process** check box if you intend to implement the Test Record process in UFT.

     If you select the **Customize Record process** check box, the wizard creates a framework of code for the implementation of recording steps.

     Do not select this check box if you are going to create the test manually in UFT, or if you are going to use the Test Record functions of the existing test object to which this control will be mapped. Your Test Record implementation does not inherit from the existing test object to which the custom control is mapped. It replaces the existing object's Test Record implementation entirely. Therefore, if you need any of the existing object's functionality, code it explicitly.
   - Select the **Customize Run process** check box if you intend to implement Test Run functions for the custom control (meaning, if you are going to override any of the existing test object's methods, or extend the test object with new methods). Enter a name for the replay interface you will create in the **Replay interface name** box.

     If you select the **Customize Run process** check box, the wizard creates a framework of code to implement Test Run support.
   - Select the **Generate comments and sample code** check box if you want the wizard to add comments and samples in the code that it generates.

6. Click **Next**. The XML Configuration Settings page of the wizard opens:



7. Using the XML Configuration Settings page of the wizard, you can generate a segment of XML code that can be copied into the .NET Add-in Extensibility configuration file (**SwfConfig.xml**). This file maps the custom control to the test object, and provides UFT with the location of the test object's Custom Server. (If you choose not to generate the XML configuration segment, you can manually edit the .NET Add-in Extensibility configuration file later.) For instructions on copying this segment into the **SwfConfig.xml** file, see "Copying Configuration Information Generated by the UFT Custom Server Settings Wizard" on page 71.

   - Select the **Auto-generate the XML configuration segment** check box to instruct the wizard to create the configuration segment, which is saved in the **Configuration.xml** file.
   - In the **Customized Control type** box, enter the full type name of the control for which you are creating the Custom Server, including all wrapping namespaces, for example, `System.Windows.Forms.CustCheckBox`.

     Note: If you want to specify a control type that is included in more than one assembly you can include the name of the assembly, or other information that will fully qualify the type. For example, you could enter values similar to these:
     `- System.Windows.Forms.CustCheckBox, System2.Windows.Forms.v8.5`
     `-System.Windows.Forms.CustCheckBox, System2.Windows.Forms.v8.5, Version=8.5.20072.1093, Culture=neutral, PublicKeyToken=8aa4d5436b5ad4cd`
     This can be useful, for example, if you have different versions of the control in your application.
   - In the **Mapped to** box, select the test object to which you want to map the Custom Server. If you select **No mapping**, the Custom Server is automatically mapped to the **SwfObject** test object.

     For more information, see "Map the Custom Controls to the Relevant Test Objects " on page 18.
   - Select the run-time context for Test Record and/or Test Run: the context of the application that is being tested (**Application under test**) or the context of UFT (**QuickTest**).

     For more information, see "Create Custom Servers" on page 17.

8. Click **Finish**. The wizard closes and the new project opens, ready for coding.

   When you click **Finish** in the wizard, the **Configuration.xml** segment file is created and added to the project. Update and modify the configuration segment file as required. For more information about using the segment file, see "Copying Configuration Information Generated by the UFT Custom Server Settings Wizard" on page 71.)

Implementing Test Record for a Custom Control Using a .NET DLL

Recording a business component or test on a control means listening to the activity of that control, translating that activity into test object method calls, and writing the method calls to the test. Listening to the activities on the control is done by listening to control events, by hooking Windows messages, or both.

Note: If you plan to create GUI tests using keyword-driven testing, and not by recording steps on an application, you do not need to implement Test Record.

Write the code for Test Record by implementing the methods in the code segment created by the wizard based on the **IRecord** interface (provided with the UFT .NET Add-in Extensibility SDK). Your Test Record implementation does not inherit from the existing test object to which the custom control is mapped. It replaces the existing object's Test Record implementation entirely. Therefore, if you need any of the existing object's functionality, code it explicitly.

Before reading this section, make sure you are familiar with "Create Custom Servers" on page 17.

This section describes:

For more information on the interfaces, classes, enumerations, and methods in this section, see the *UFT .NET Add-in Extensibility API Reference* (available in the UFT .NET Add-in Extensibility online Help.)

Implementing the IRecord Interface

To implement the **IRecord** interface, override the callback methods described below and add the details of your implementation in your event handlers or message handler.

The examples provided below for each callback method are written in C#.

InitEventListener Callback Method

CustomServerBase. InitEventListener is called by UFT when your Custom Server is loaded. Add your event and message handlers using this method.

To add event and message handlers:

1. Implement handlers for the control's events.

   A typical handler captures the event and writes a method to the test. This is an example of a simple event handler:

   ```
   public void OnMouseDown(object sender, MouseEventArgs e)
   {
       // If a button other than the left was clicked, do nothing.
       if(e.Button != System.Windows.Forms.MouseButtons.Left)
       return;
       /*
       For more complex events, here you would get any
       other information you need from the control.
       */
       // Write the test object method to the test
       RecordFunction("MouseDown",
       RecordingMode.RECORD_SEND_LINE,
       e.X,e.Y);
   }
   ```

   For more information, see "Implementing Test Record for a Custom Control Using a .NET DLL" on the previous page.

2. Add your event handlers in **InitEventListener**:

   ```
   public override void InitEventListener()
   {
       .....
       // Adding OnMouseDown handler.
       Delegate  e = new MouseEventHandler(this.OnMouseDown);
       AddHandler("MouseDown", e);
       .....
   }
   ```

   Note that if the Test Record implementation will run in the context of the application being tested, you can use the following syntax:

   ```
   SourceControl.MouseDown += e;
   ```

   If you use this syntax, you must release the handler in **ReleaseEventListener**.

3. Add a remote event listener.

If your Custom Server will run in the UFT context, use a remote event listener to handle events. Implement a remote listener of type **EventListenerBase** that handles the events, and add a call to AddRemoteEventListener in method **InitEventListener**.

```
public class EventsListenerAssist : EventsListenerBase
{
    // class implementation.
}
public override void InitEventListener()
{
    ...
    AddRemoteEventListener(typeof(EventsListenerAssist));
    ...
}
```

When you implement a remote event listener, you must override EventListenerBase.InitEventListener and **EventListenerBase.ReleaseEventListener** in addition to overriding these callback functions in **CustomServerBase**. The use of these **EventListenerBase** callbacks is the same as for the **CustomServerBase** callbacks. For details, see the **EventsListenerBase** class in the *UFT .NET Add-in Extensibility API Reference*.

When you handle events from the UFT context, the event arguments must be serialized. For details, see CustomServerBase.AddHandler(String, Delegate, Type) and the **IEventArgsHelper** interface in the *UFT .NET Add-in Extensibility API Reference*.

To avoid the complications of remote event listeners, run your event handlers in the **Application under test** context, as described above.

OnMessage Callback Method

OnMessage is called on any window message hooked by UFT. If Test Record will run in the UFT context and message handling is required, implement the message handling in this method.

If Test Record will run in the **Application under test** context, do not override this function.

For details, see **CustomServerBase.OnMessage** in the *UFT .NET Add-in Extensibility API Reference*.

GetWndMessageFilter Callback Method

If Test Record will run in the UFT context and listen to windows messages, override this method to inform UFT whether the Custom Server will handle only messages intended for the specific custom object, or whether it will handle messages from child objects, as well.

For details, see **IRecord.GetWndMessageFilter** in the *UFT .NET Add-in ExtensibilityAPI Reference*.

See also:"Troubleshooting and Limitations - Running the Support You Designed" on page 66.

ReleaseEventListener Callback Method

UFT calls this method at the end of the recording session. In **ReleaseEventListener**, unsubscribe from all the events to which the Custom Server was listening. For example, if you subscribe to **OnClick** in **InitEventListener** with this syntax,

```
SourceControl.Click += new EventHandler(this.OnClick);
```

you must release it:

```
public override void ReleaseEventListener()
{
    ....
    SourceControl.Click -= new EventHandler(this.OnClick);
    ....
}
```

However, if you subscribe to the event with the **AddHandler** method, UFT unsubscribes automatically.

Writing Test Object Methods to the Test

When information about activities of the control is received, whether in the form of events, Windows messages, or a combination of both, this information must be processed as appropriate for the application and a step must be written as a test object method call.

To write a test step, use the **RecordFunction** method of the **CustomServerBase** class or the **EventsListenerBase**, as appropriate.

Sometimes, it is impossible to know how an activity should be processed until the next activity occurs. Therefore, there is a mechanism for storing a step and deciding in the subsequent call to **RecordFunction** whether to write it to the test. For details, see **RecordingMode Enumeration** in the *UFT .NET Add-in ExtensibilityAPI Reference*.

To determine the argument values for the test object method call, it may be necessary to retrieve information from the control that is not available in the event arguments or Windows message. If the Custom Server Test Record implementation is running in the context of the application being tested, use the **SourceControl** property of the **CustomServerBase** class to obtain direct access to the public members of the control. If the control is not thread-safe, use the **ControlGetProperty** method to retrieve control state information.

Implementing Test Run for a Custom Control Using the .NET DLL

Defining test object methods for Test Run means specifying the actions to perform on the custom control when the method is run in a step. Typically, the implementation of a test object method performs several of the following actions:

- Sets the values of attributes of the custom control
- Calls a method of the custom control
- Makes mouse and keyboard simulation calls
- Reports a step outcome to UFT
- Reports an error to UFT
- Makes calls to another library (to show a message box, write custom log, and so on)

Define custom Test Run methods if you are overriding existing methods of the existing test object, or if you are extending the existing test object by adding new methods.

Ensure that all test object methods recorded are implemented in Test Run, either by the existing test object or by this Custom Server.

To define custom Test Run methods, define an interface and instruct UFT to identify it as the Test Run interface by applying the **ReplayInterface** attribute to it. Only one replay interface can be implemented in a Custom Server. If your interface defines methods with the same names as existing methods of the existing object, the interface methods override the test object implementation. Any method name that is different from existing object's method name is added as a new method.

Start a test object method implementation with a call to **PrepareForReplay**, specify the activities to perform, and end with a call to **ReplayReportStep** and/or **ReplayThrowError**.

For more information, see the *UFT .NET Add-in Extensibility API Reference* (available in the UFT .NET Add-in Extensibility).

Implementing Support for Table Checkpoints and Output Values in the .NET DLL Custom Server

By adding table checkpoints to a test, UFT users can check the content and properties of tables displayed in their application. By adding table output value steps to a test, you can retrieve values from a table, store them, and then use them as input at a different stage in the run session.

With .NET Add-in Extensibility, you can enable UFT to support table checkpoints and output values for custom table (grid) controls.

To implement table checkpoint and output value support, add a verification class in your Custom Server that inherits from the **VerificationServerBase** class and override the necessary methods

(for more information, see below). In the .NET Add-in Extensibility configuration file, map each custom table control to an **SwfTable** test object, and to the verification class in the relevant Custom Server. For information on the syntax of the verification class methods, see the *UFT .NET Add-in Extensibility API Reference* (available with the .NET Add-in Extensibility SDK online Help).

Note: When creating a Custom Server using the UFT Custom Server Settings wizard, the source code created by the wizard does not include commented code for table checkpoint and output value support. Add the implementation manually.

To implement support for table checkpoints and output values on custom table objects:

1. Map the custom table control to the **SwfTable** test object class. This instructs UFT to use an **SwfTable** test object to represent the custom table control in GUI tests or components.

   In the .NET Add-in Extensibility configuration file, **<UFT Installation folder>\dat\SwfConfig.xml**, create a **Control** element with a **Type** attribute set to the name of the custom table control, and the **MappedTo** attribute set to **SwfTable**.

   For more information on the **SwfConfig.xml** file, see "Understanding How to Configure UFT Windows Forms Extensibility" on page 69 and the .NET Add-in Extensibility Configuration Schema Help (available with the .NET Add-in Extensibility SDK online Help).

2. Specify table verification configuration information for the Custom Server of this custom table control.

   In the same **SwfConfig.xml** file, define a **CustomVerify** element. In this element, specify:

   - The run-time context, which for this element must always be **AUT**.
   - The name of the Custom Server (DLL) that contains the implementation of table checkpoint and output value support for this control.
   - The type name for the verification class within the Custom Server (DLL) including wrapping namespaces.

   A sample of the **CustomVerify** element is provided below:

```
<Control Type="System.Windows.Forms.DataGridView"  MappedTo="SwfTable">
        <CustomRecord>
        ...
        ...
        </CustomRecord>
        <CustomReplay>
        ...
        ...
        </CustomReplay>
        <CustomVerify>
```

```
            <Context>AUT</Context>
            <DllName>C:\MyProducts\Bin\\VfySrv.dll</DllName>
            <TypeName>VfySrv.DataGridCPSrv</TypeName>
        </CustomVerify>
        <Settings>
</Control>
```

3. In the verification class, override the following protected methods so that UFT receives what it requires when supporting table checkpoints and output values.

   - **GetTableData**

     UFT calls this method to retrieve table data from the specified range of rows and returns the data as an array of objects.

     When working with a table checkpoint or output value, UFT calls the **GetTableRowRange** method before this method so that the first and last rows in the data range of the table are known to the **GetTableData** method.

   - **GetTableRowRange**

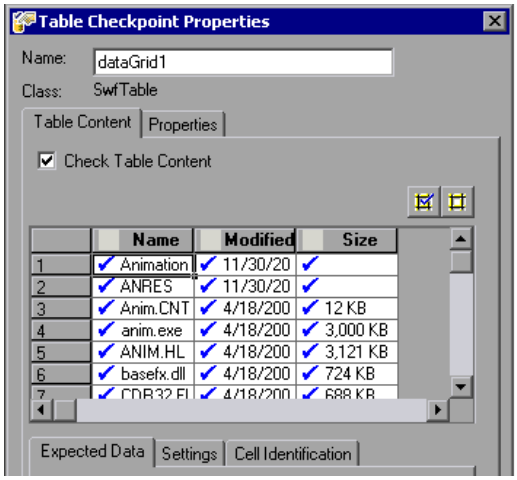
     UFT calls this method to retrieve the number and range of rows in the table that will be included in the checkpoint or output value.

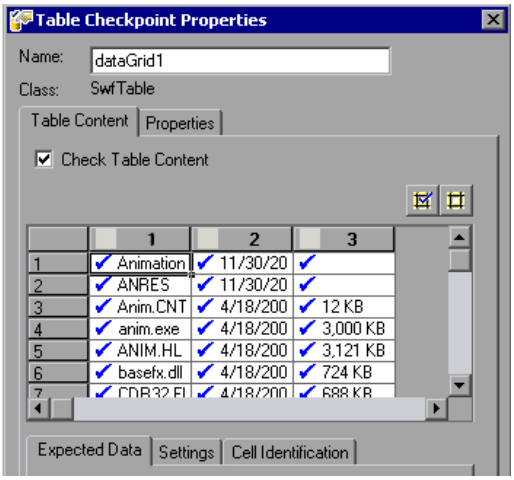
     When working with a table checkpoint or output value, UFT calls this method before the **GetTableData** method. The **GetTableRowRange** method initializes the values of the first and last rows in the data range of the table, which the **GetTableData** method uses as input.

   - **GetTableColNames**

     UFT calls this method to retrieve the column names as an array of strings. UFT displays these column names in the Table Checkpoint Properties and Table Output Value Properties dialog boxes. If this method is not implemented, numbers appear instead of column names in these dialog boxes.

The images below shows what the Table Checkpoint Properties dialog box looks like with and without GetTableColNames implementation:





The following sample (written in C#) demonstrates implementation of the GetTableData, GetTableColNames, and GetTableRowRange methods.

```
using System;
using System.Collections.Generic;
using System.Text;
using Mercury.QTP.CustomServer;
using System.Windows.Forms;
namespace VfySrv
{
   public class DataGridCPSrv : VerificationServerBase
    {
    /// GetTableData() is called by UFT to retrieve the data in a table.
    /// The following base class properties are used:
    ///    SourceControl - Reference to the grid (table) object
    ///    FirstRow - The (zero-based) row number of the start of
    ///               the checkpoint or output value
    ///    LastRow - The (zero-based) row number of the end of
    ///              the checkpoint or output value
    /// Returns a two-dimensional array of objects.
    protected override object[,] GetTableData()
     {
       DataGridView GridView = (DataGridView)(base.SourceControl);
       int TotalRows = GridView.Rows.Count;
       int TotalColumns = GridView.Columns.Count;
       int FirstRowN = base.FirstRow;
       int LastRowN = base.LastRow;
       TotalRows = LastRowN - FirstRown + 1;
       object[,] Data = new object[TotalRows, TotalColumns];
```

```
 DataGridViewRowCollection Rows = GridView.Rows;
   for (int i = FirstRowN; i <= LastRowN; i++)
    {
     DataGridViewRow Row = Rows[i];
     DataGridViewCellCollection Cells = Row.Cells;
     for (int k = 0; k < TotalColumns; k++)
      {
       Data[i - FirstRown, k] = Cells[k].Value;
      }
    }
   return Data;
}

/// GetTableColNames is called by UFT to
/// retrieve the column names of the table.
/// Returns an array of column names.
protected override string[] GetTableColNames()
 {
   DataGridView GridView = (DataGridView)(this.SourceControl);
   int TotalColumns = GridView.Columns.Count;
   string[] ColNames = new string[TotalColumns];
   for (int i = 0; i < TotalColumns; i++)
    {
      ColNames[i] = GridView.Columns[i].HeaderText;
    }
   return ColNames;
 }

/// GetTableRowRange is called by UFT to
/// obtain the number of rows in the table.
protected override void GetTableRowRange
          (out int FirstVisible, out int LastVisible, out int Total)
 {
   DataGridView GridView = (DataGridView)(this.SourceControl);
   DataGridViewRowCollection Rows = GridView.Rows;
   FirstVisible = -1;
   LastVisible = Rows.Count -1;
   for (int i = 0; i < Rows.Count; i++)
    {
      if (Rows[i].Visible == false)
        continue;
      FirstVisible = i;
      break;
    }
   for (int i = FirstVisible + 1; i < Rows.Count; i++)
    {
      if (Rows[i].Visible)
         continue;
```

```
            LastVisible = i;
            break;
          }
         FirstVisible++;
         LastVisible++;
         Total = GridView.Rows.Count;
       }
    }
}
```

Running Code under Application Under Test from the UFT Context

When the Custom Server is running in the **UFT** context, there is no direct access to the control, which is in a different run-time process. To access the control directly, run part of the code in the **Application under test** context. This is done using assistant classes.

To launch code from the UFT context that will run under the **Application under test** context, implement an assistant class that inherits from **CustomAssistantBase**. To create an instance of an assistant class, call **CreateRemoteObject**. Before using the object, attach it to the control with **SetTargetControl**.

After **SetTargetControl** is called, you can call methods of the assistant in one of the following ways:

- If the method can run in any thread of the **Application under test** process, read and set control values and call control methods with the simple **obj.Member** syntax:

```
int i = oMyAssistant.Add(1,2);
```

- If the method must run in the control's thread, use the **InvokeAssistant** method:

```
int i = (int)InvokeAssistant(oMyAssistant, "Add", 1, 2);
```

Tip: You can use the EventListenerBase, which is an assistant class that supports listening to control events.

Reviewing Commonly-used API Calls

This section provides a quick reference of the most commonly used API calls. Review this information before starting to implement methods.

These methods are in CustomServerBase except where indicated.

For more information, see the *UFT .NET Add-in Extensibility API Reference* (available in the UFT .NET Add-in Extensibility online Help.)

Test Record Methods

AddHandler	Adds an event handler as the first handler of the event.
RecordFunction	Records a step in the test.

Test Record Callback Methods

GetWndMessageFilter	Called by UFT to set the Windows message filter.
InitEventListener	Called by UFT to load event handlers and start listening for events.
OnMessage	Called when UFT hooks the window message.
ReleaseEventListener	Stops listening for events.

Test Run Methods

DragAndDrop, KeyDown, KeyUp, MouseClick, MouseDblClick, MouseDown, MouseMove, MouseUp, PressKey, PressNKeys, SendKeys, SendString	Mouse and keyboard simulation methods.
PrepareForReplay	Prepares the control for an action run.
ReplayReportStep	Writes an event to the test report.
ReplayThrowError	Generates an error message and changes the reported step status.
ShowError	Displays the .NET warning icon.
TestObjectInvokeMethod	Invokes one of the methods exposed by the test object's IDispatch interface.

Cross-Process Methods

AddRemoteEventListener	Creates an **EventListener** instance in the **Application under test** process.
CreateRemoteObject	Creates an instance of an assistant object in the **Application under test** process.
GetEventArgs (IEventArgsHelper)	Retrieves and deserializes the **EventArgs** object.

Init (IEventArgsHelper)	Initializes the **EventArguments** helper class with an **EventArgs** object.
InvokeAssistant	Invokes a method of a **CustomAssistantBase** class in the control's thread.
InvokeCustomServer (EventsListenerBase)	Invokes the Custom Server's methods running in the **UFT** process from the **Application under test** process.
SetTargetControl (CustomAssistantBase)	Attaches to the source custom control by the control's window handle.

General Methods

ControlGetProperty	Retrieves a property of a control that is not thread-safe.
ControlInvokeMethod	Invokes a method of a control that is not thread-safe.
ControlSetProperty	Sets a property of a control that is not thread-safe.
GetSettingsValue	Gets a parameter value from the settings of this control in the configuration file.
GetSettingsXML	Returns the settings of this control as entered in the configuration file.

Table Checkpoint and Output Value Support Methods

GetTableData (VerificationServerBase)	Called by UFT to retrieve the data in a table.
GetTableRowRange (VerificationServerBase)	Called by UFT to retrieve the first and last rows of the table.
GetTableColNames (VerificationServerBase)	Called by UFT to retrieve the names of the table columns.

Using XML Files to Extend Support for a Custom Control

You can implement custom control support without programming a .NET DLL by entering the appropriate Test Record and Test Run instructions for that custom control in a control definition file. (Create a separate control definition file for each control you want to customize.) You can instruct UFT to load the custom control implementation instructions by specifying each control definition file in the .NET Add-in Extensibility configuration file, **SwfConfig.xml**.

Note: When extending support using an XML file, UFT generates an ad hoc .NET DLL for you based on the XML file. This ad hoc .NET DLL becomes the custom server for the control.

When using this technique, you do not have the support of the .NET development environment—the object browser and the debugger— or the ability to create table checkpoints or output values.

However, by enabling the implementation of custom control support without the .NET development environment, this technique enables relatively rapid implementation, even in the field.

This feature is most practical either with relatively simple, well documented controls, or with controls that map well to an existing object but for which you need to replace the Test Record definitions, or replace or add a small number of test object Test Run methods.

Understanding Control Definition Files

The control definition file can contain a **Record** element in which you define the customized recording for the control and a **Replay** element in which you define the customized test object methods.

- The **Record** element specifies the control events for which you want UFT to add steps to the test (or component) during a recording session. The steps are calls to test object methods of the custom control's test object.
- The **Replay** element specifies the operations that UFT should perform on the control for each test object method during a run session.

You do not always need to enter both a **Record** and a **Replay** element:

- If the Test Record implementation for the custom test object should be different than the one defined for the existing test object, create a **Record** element in the control definition file for the custom control.
- Similarly, if the Test Run implementation for the custom test object should be different than the one defined for the existing test object, create a **Replay** element in the control definition file for the custom control.

If you create a **Record** element, the definitions replace the Test Record implementation of the existing test object entirely. If you create a **Replay** element, it inherits the Test Run implementation of the existing object and extends it. For more information on test object mapping options, see "Map the Custom Controls to the Relevant Test Objects " on page 18.

For information on the elements in a control definition XML file, see the .NET Add-in Extensibility Control Definition Schema Help (available with the .NET Add-in Extensibility SDK online Help).

An Example of a Control Definition File

The following example shows the handling of an object whose value changes at each **MouseUp** event. The value is in the **Value** property of the object. The **MouseUp** event handler has **Button, Clicks, Delta, X,** and **Y** event arguments.

The **Record** element describes the conversion of the **MouseUp** event to a **SetValue** command. The **Replay** element defines the **SetValue** command as setting the value of the object to the recorded `Value` and displaying the position of the mouse pointer for debugging purposes:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<Customization>
    <Record>
        <Events>
            <Event name="MouseUp" enabled="true">
                <RecordedCommand name="SetValue">
                    <Parameter>
                        Sender.Value
                    </Parameter>
                    <Parameter lang="C#">
                        String xy;
                        xy = EventArgs.X + ";" + EventArgs.Y;
                        Parameter = xy;
                    </Parameter>
                </RecordedCommand>
            </Event>
        </Events>
    </Record>
    <Replay>
        <Methods>
            <Method name="SetValue">
                <Parameters>
                    <Parameter type="int" name="Value"/>
                    <Parameter type="String" name="MousePosition"/>
                </Parameters>
                <MethodBody>
                    RtObject.Value = Value;
                    System.Windows.Forms.MessageBox.Show(MousePosition,
"Mouse Position at Record Time");
                </MethodBody>
            </Method>
        </Methods>
    </Replay>
</Customization>
```

Using the .NET Add-in Extensibility Samples

The .NET Add-in Extensibility SDK provides a sample support set to help you learn about .NET Add-in Extensibility. The toolkit support set files are installed in the **<UFT .NET Add-in Extensibility SDK installation folder>\samples\WinFormsExtSample** folder. You can study the content of these files to gain a better understanding of how to develop your own toolkit support sets.

The sample support set extends UFT support for the SandBar custom .NET Windows Forms control. The custom server provided in this sample is similar to the one you create in "Learning to Create Support for a Complex Custom .NET Windows Forms Control" on page 88.

The **SandbarSample.sln** solution file located in the **WinFormsExtSample** folder includes a configuration file and a fully implemented custom server that supports the SandBar control. The

SandBarCustomServer implementation is provided in C# and in Visual Basic, in separate projects within the solution (**SandbarCustomServer** and **VBSandbarCustomServer**). In addition, the SandbarSample solution includes a sample .NET Windows Forms application that uses the SandBar toolbar control (**SandbarTestApp**).

To learn how extensibility can affect UFT's interaction with custom controls, create and run a UFT GUI test on the sample application before and after deploying the sample toolkit support set to UFT.

Considerations for Working with the SandBar Support Sample

- To open the SandbarSample solution, use Microsoft Visual Studio 2005 or later.
- Before you build the SandbarSample solution, ensure that the following items are installed on your computer:
  - The UFT .NET Add-in Extensibility SDK
  - SandBar for .NET 2.0/3.x (can be downloaded from http://www.divil.co.uk/net/download.aspx?product=2&license=5)
- After successfully building the SandbarSample solution, deploy the C# or Visual Basic custom server it creates as described in "Configuring and Deploying the Support Set" on page 68.
- Before you update the **SwfConfig.xml** file according to the information in **Configuration.xml**, consider the following: The **Configuration.xml** file in the SandbarSample solution is set up to use the DLL generated by the C# project and located in **<UFT .NET Add-in Extensibility installation folder>\samples\WinFormsExtSample\Bin**.
  - To use **VBCustomSandBarSrv.dll**, replace all appearances of `SandbarCustomServer` in with `VBCustomSandBarSrv.dll`.
  - If your DLL file is located in a different location, update the path in the **DllName** element accordingly.

Troubleshooting and Limitations - Running the Support You Designed

This section describes troubleshooting and limitations for developing your support set.

The custom server is not receiving some Windows messages

During a recording session, the custom server mapped to your custom control is only created after some operation takes place on the custom control itself.

If you design the **GetWndMessageFilter** method to specify that your custom server will handle messages that occur on other controls, such messages can only be handled after the custom server is created.

Therefore, for example, you may have to click on the custom control before the custom server can receive and process messages on other controls in the application.

Depending on how you implement support for recording on your custom control, you might want to provide instructions regarding this issue to the UFT users who use your support set.

A General Run Error occurs while running the test in UFT

When using the .NET Add-in Extensibility API with Microsoft .NET Framework 1.1, a **General Run Error** may occur while running your test. This is caused by an **Execution Engine Exception** error in the application under test (AUT).

Workaround: Install Service Pack 1 (or later) for Microsoft .NET Framework 1.1.

A run-time error occurs while running the test in UFT

When using an XML-based Custom Server, if you have more than one version of Microsoft .NET Framework installed, a run-time error might occur during the run session. The error message in the log file indicates that the configuration file contains a compilation error. This is because assemblies compiled with Microsoft .NET Framework version 2.0 and later are not recognized by earlier versions of Microsoft .NET Framework.

Workaround: Perform one of the following:

- **Solution 1:** In the Registry, in the following key **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework**add the following DWORD Value `"OnlyUseLatestCLR"=dword:00000001`
- **Solution 2:** If the .NET application you are testing has a configuration file, add the following information to the file:

```
<configuration>
      <startup>
            <supportedRuntime version="v2.0.50727"/>
      </startup>
</configuration>
```

The configuration file must be named **<executable_name>.exe.Config** and be located in the same folder as the executable of the .NET application you are testing.

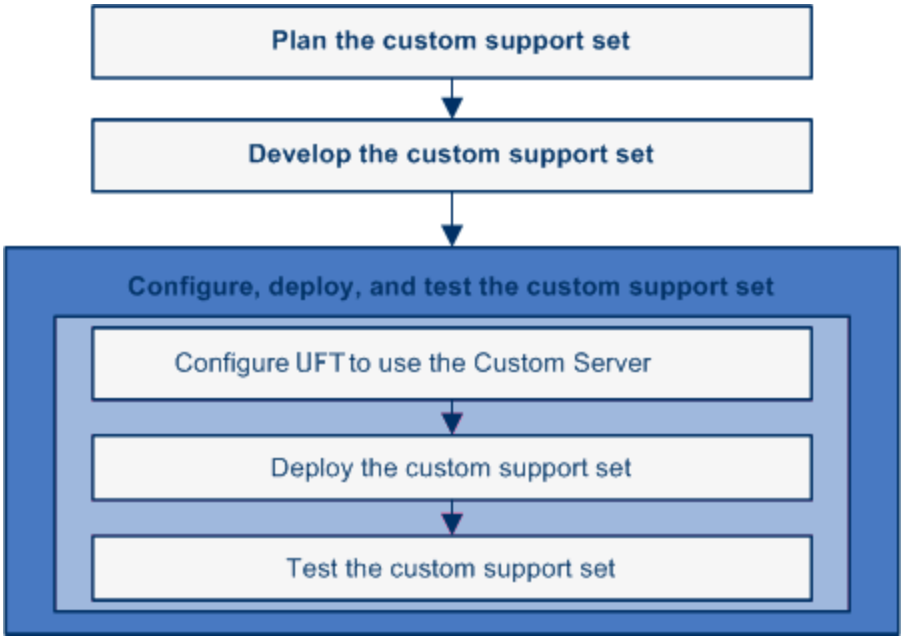
Chapter 5: Configuring and Deploying the Support Set

After developing the implementation for your Custom Server, the UFT .NET Add-in Extensibility Support Set is ready for configuration and deployment.

This chapter includes:

Understanding the Deployment Workflow

The workflow for deploying a .NET Add-in Extensibility support set consists of the stages shown in the highlighted area of the image. These stages are described in detail in the sections below.



Configuring UFT to Use the Custom Server

The .NET Add-in Extensibility configuration file (**SwfConfig.xml**) provides UFT with the configuration information it needs to load your Custom Servers.

Understanding How to Configure UFT Windows Forms Extensibility

To instruct UFT to load Custom Servers according to the appropriate configuration, enter the information in the .NET Add-in Extensibility configuration file. This file, **SwfConfig.xml**, is located in the **<UFT installation folder>\dat** folder.

Enter configuration information into the **SwfConfig.xml** file in one of the following ways:

- Manually edit the file using any text editor.
- Copy information from **configuration.xml** files generated by the UFT Custom Server Settings wizard.

  For more information about the wizard, see "Using a .NET DLL to Extend Support for a Custom Control" on page 45.

  For instructions on how to copy information from **configuration.xml** files, see "Copying Configuration Information Generated by the UFT Custom Server Settings Wizard" on the next page.

When configuring UFT Windows Forms extensibility, define elements according to the coding option you selected for implementing your Custom Server:

- "When Using a .NET DLL Custom Server" below
- "When Using an XML Custom Server" on the next page

When Using a .NET DLL Custom Server

In the **SwfConfig.xml** file, for each custom .NET control that you will implement using a .NET DLL Custom Server, you can define:

- A **MappedTo** attribute, if you want the custom control to correspond to a test object other than the default generic test object **SwfObject**.
- A **CustomRecord** element if you want to customize recording on the control.
- A **CustomReplay** element if you want to customize how test steps are run on a custom control.
- A **CustomVerify** element if you want to add table checkpoint and output value support for custom table controls.
- A **Settings** element, in which you can use the **Parameter** element to pass values to the Custom Server at run-time.

When Using an XML Custom Server

In the **SwfConfig.xml** file, for each custom .NET control that you will implement using an XML Custom Server, you define:

- A **MappedTo** attribute, if you want the custom control to correspond to a test object other than the default test grid object **SwfTable**.
- The **Context** attribute of a **CustomRecord** element if you want to customize recording on the control.
- The **Context** attribute of a **CustomReplay** element if you want to customize how test steps are run on a custom control.
- A **Settings** element, in which you can use the **Parameter** element to pass values to the Custom Server at run-time.

> **Note:** UFT loads the Custom Server when you open a test. Therefore, if you implement your Custom Server as a .NET DLL, any changes you make to the DLL after the Custom Server is loaded take effect only the next time you open a test.

For information on the elements in the .NET Add-in Extensibility configuration file (**SwfConfig.xml**), see the .NET Add-in Extensibility Configuration Schema Help (available with the .NET Add-in Extensibility SDK online Help).

Copying Configuration Information Generated by the UFT Custom Server Settings Wizard

When running the UFT Custom Server Settings wizard to create a Custom Server, the wizard creates an XML configuration segment. The wizard outputs this segment to help you enter the configuration information in the .NET Add-in Extensibility configuration file.

To incorporate the contents of the XML configuration segment before deploying the Custom Server:

1. Edit the **Configuration.xml** segment file in the project to ensure that the information is correct. Set the **DllName** element value to the location to which you will deploy the Custom Server. If **Test Record** and/or **Test Run** are to be loaded in different run-time contexts, edit the **Context** value accordingly.
2. Copy the entire **<Control>...</Control>** node. Do not include the enclosing **<Controls>** tags.
3. Open the .NET Add-in Extensibility configuration file, **<UFT installation folder>\dat\SwfConfig.xml**. Paste the **Control** node from **Configuration.xml** at the end of the file, before the closing **</Controls>** tag.

4. Save the file. If UFT was open, you must close and reopen it for the **SwfConfig.xml** changes to take effect.

> **Note:** You can validate the configuration file you design against the **<UFT installation folder>\dat\SwfConfig.xsd** file.

Example of a .NET Add-in Extensibility Configuration File

Following is an example of a file that configures UFT to recognize the following controls:

- Support for the **MyCompany.WinControls.MyListView** control is implemented in the **CustomMyListView.CustListView** .NET DLL Custom Server. The Custom Server is not installed in the GAC, so the DLL name is specified as a path and file name (and is not passed as a type name according to GAC standard syntax).

  MyListView is mapped to the **SwfListView** test object, and runs in the context of the application being tested.

- Support for the **mySmileyControls.SmileyControl2** control is implemented in an XML file. Therefore, the path and file name for the Control Definition file that contains its implementation is passed to UFT during run-time using the **Parameter** element.

  The **SmileyControl2 control** is not explicitly mapped to any test object in the **SwfConfig.xml** file, so UFT maps it to the default generic test object, **SwfObject**.

- Customized record and run support for the **System.Windows.Forms.DataGridView** control is implemented in a .NET DLL Custom Server called **CustomMyTable.dll**. Table checkpoint and output value support for the **System.Windows.Forms.DataGridView** control is implemented in a .NET DLL Custom Server called **VfySrv.dll**.

  DataGridView must be mapped to the **SwfTable** test object (according to the restrictions imposed by the **TableElement** complex type element in the schema), and, because the customized support includes table checkpoints and output values, must run in the context of the application being tested.

```
<?xml version="1.0" encoding="UTF-8"?>
<Controls>
       <Control Type="MyCompany.WinControls.MyListView" MappedTo="SwfListView" >
              <CustomRecord>
                     <Component>
                            <Context>AUT</Context>
                            <DllName>C:\MyProducts\Bin\CustomMyList View.dll</DllName>
                            <TypeName>CustomMyListView.CustListView</TypeName>
                     </Component>
              </CustomRecord>
              <CustomReplay>
                     <Component>
                            <Context>AUT</Context>
                            <DllName>C:\MyProducts\Bin\CustomMyList View.dll</DllName>
```

```
                        <TypeName>CustomMyListView.CustListView</TypeName>
                </Component>
        </CustomReplay>
        <Settings>
                <Parameter Name="sample name">sample value</Parameter>
                <Parameter Name="ConfigPath">C:\Program Files\HP\Unified Functional
Testing\dat\Extensibility\dotNET\MyContrSIM.xml</Parameter>
        </Settings>
</Control>

<Control Type="mySmileyControls.SmileyControl2">
        <Settings>
                <Parameter Name="ConfigPath">d:\UFT\bin\ConfigSmiley.xml
                </Parameter>
        </Settings>
        <CustomRecord>
                <Component>
                        <Context>AUT-XML</Context>
                </Component>
        </CustomRecord>
        <CustomReplay>
                <Component>
                        <Context>AUT-XML</Context>
                </Component>
        </CustomReplay>
</Control>

<Control Type="System.Windows.Forms.DataGridView"
                MappedTo="SwfTable">
        <CustomRecord>
                <Component>
                        <Context>QTP</Context>
                        <DllName>C:\MyProducts\Bin\CustomMyTable.dll</DllName>
                        <TypeName>CustomMyTable.CustTableView</TypeName>
                </Component>
        </CustomRecord>
        <CustomReplay>
                <Component>
                        <Context>AUT-XML</Context>
                </Component>
        </CustomReplay>
        <CustomVerify>
                 <Context>AUT</Context>
                 <DllName>C:\MyProducts\Bin\VfySrv.dll</DllName>
                 <TypeName>VfySrv.DataGridCPSrv</TypeName>
         </CustomVerify>
        <Settings>
                <Parameter Name="sample name">sample value</Parameter>
        </Settings>
</Control>

</Controls>
```

Deploying the Custom Support Set

The next stage of extending UFT GUI testing support for custom controls is deployment. This means placing all files you created in the correct locations, so that the custom support is available to UFT.

After you deploy the custom support, if you run an application that contains the custom controls and perform UFT operations on the application, you can see the effects of the support you designed.

Placing Files in the Correct Locations

To deploy the support set that you create, place the files in the locations described in the following table. Make sure that UFT is closed before placing the files in their appropriate locations.

File Name	Location
SwfConfig.xml	**<UFT installation path>\dat**
<Test Object Configuration File Name>.xml **Note:** You can have more than one test object configuration file (if any), and name them as you wish.	• **<UFT installation path>\dat\Extensibility\DotNet** • **<UFT Add-in for ALM Installation Path>\dat\Extensibility\DotNet** (Optional. Required only if **UFT** Add-in for **ALM** is installed)
<Control Definition File Name>.xml> **Note:** The Control Definition file is used when creating a Custom Server using the XML coding option. You can have more than one control definition file (one for each custom control).	• **<UFT installation path>\dat\Extensibility\DotNet** • **<UFT Add-in for ALM Installation Path>\dat\Extensibility\DotNet** (Optional. Required only if UFT Add-in for ALM is installed)
<Custom Server File Name>.dll **Note:** This type of Custom Server is used when creating a Custom Server using the .NET DLL coding option. You can have more than one custom server for each custom control.	Specify the location of your compiled Custom Servers (DLLs) in the **SwfConfig.xml** file.

Modifying Deployed Support

If you modify a support set that was previously deployed to UFT, the actions you must perform depend on the type of change you make, as follows:

- If you modify the .NET Add-in Extensibility configuration file or a test object configuration file, you must deploy the support.
- If you modify a test object configuration file, you must reopen UFT and open a GUI test after deploying the support.

Removing Deployed Support

To remove support for a custom control from UFT after it is deployed, you must delete the corresponding section in the **SwfConfig.xml file** from **<UFT installation path>\dat** and remove the corresponding test object configuration file from **<UFT installation path>\dat\Extensibility\DotNet**.

If you remove support for a new test object method that you added in a test object configuration file, you should remove the method definition (or the whole file, if appropriate) so that UFT users do not create test steps that call that method. Modify or remove the test object configuration file in: **<UFT Installation Path>\Dat\Extensibility\DotNet** (and **<UFT Add-in for ALM Installation Path>\Dat\Extensibility\DotNet** if relevant).

Testing the Custom Support Set

We recommend that you test the custom support using an incremental approach. First, test the basic functionality of the support set. Then, test its implementation.

- "Testing Basic Functionality of the Support Set" below
- " Testing Implementation" on page 77

Testing Basic Functionality of the Support Set

After you define a basic .NET Windows Forms configuration file enabling UFT to identify which test object classes to use for the different controls, and (optionally) define your test object model in the test object configuration file, you can test the existing functionality of the support set. To do this, you deploy the support set and test how UFT interacts with the controls in your environment.

To test your support set after defining the test object classes and mapping them to custom .NET Windows Forms controls:

1. In the test object configuration file, set the **TypeInformation\DevelopmentMode** attribute to **true**, to ensure that UFT reads all of the test object class information from the file each time it opens. When you complete the development of the support set, make sure to set this attribute to **false**.

2. Deploy the support set on a UFT computer by copying the files of the support set to the correct locations in the UFT installation folder, as described in "Placing Files in the Correct Locations" on page 74.

3. Open UFT, load the .NET Add-in, and open a GUI test. (If the Add-in Manager dialog box does not open when you open UFT, see the *HP Unified Functional Testing Add-ins Guide* for instructions.)

4. Open an application with your custom controls.

5. Based on the mapping definitions you created, UFT can already recognize and learn your controls.

   Use the **Add Objects to Local**  button in the Object Repository dialog box to learn your controls.

6. If you created a test object configuration file, you can already see its effect on UFT:

   a. If you added a test object method to a test object class, you can view it using the Object Spy  .

   b. You can create test steps that use the test object method that you added. (If you have not yet implemented the custom server that supports this test object method, running a such a test step will cause a run-time error.)

      In the Keyword View:

      Create a test step with a test object from a class that you modified.

      - If you added a test object method to a test object class, the method appears in the list of available operations in the **Operation** column.
      - After you choose an operation, the **Value** cell is partitioned according to the number of arguments of the selected operation, and if you defined possible values for the operation (in the **ListOfValues** element), they are displayed in a list.
      - The descriptions and documentation strings you defined for the test object methods are displayed in tooltips and in the **Documentation** column, respectively.

      In the Editor:

      Create a test step with a test object from a class that you modified. The statement completion feature displays all of the operations available for the test object, and possible input values for these operations, if relevant, based on the definitions in the test object configuration file.

In the Step Generator:

Create a test step with a test object from a class that you modified. The operations that you defined in the test object configuration file are displayed in the **Operation** list, and the descriptions you defined for the operations are displayed as tooltips.

Note: For more information on working with these options in UFT, see the *HP Unified Functional Testing User Guide*.

Testing Implementation

After you complete additional stages of developing support for your environment, you can deploy the support set again and test additional areas of interaction between UFT and your controls (for example, running and recording GUI tests).

To test your support set after developing support for additional UFT functionality:

1. Follow the steps in "Testing Basic Functionality of the Support Set" on page 75 to deploy the support set, open UFT, load the support and run an application with controls from your environment.

2. Depending on the UFT functionality for which you are developing support, perform the relevant UFT operations on the application to test that support. For example, run a test on the application, record test steps on the application and so on.

Chapter 6: Learning to Create Support for a Simple Custom .NET Windows Forms Control

In this tutorial, you will learn how to build a Custom Server for a Microsoft TrackBar control that enables UFT to record and run a **SetValue** operation on the control. You will implement the Custom Server in C#. A Custom Server can be similarly implemented in Visual Basic.

This tutorial refers to Microsoft Visual Studio 2008. However, you can use other supported versions of Visual Studio to build the Custom Server as described in this tutorial.

> **Note:** The Microsoft Visual Studio dialog box images and the instructions in this chapter refer to Microsoft Visual Studio 2008. If you use a different Microsoft Visual Studio version, the dialog boxes may differ slightly in appearance and the **UFT CustomServer** template may be located in a slightly different node in the tree.

This chapter includes:

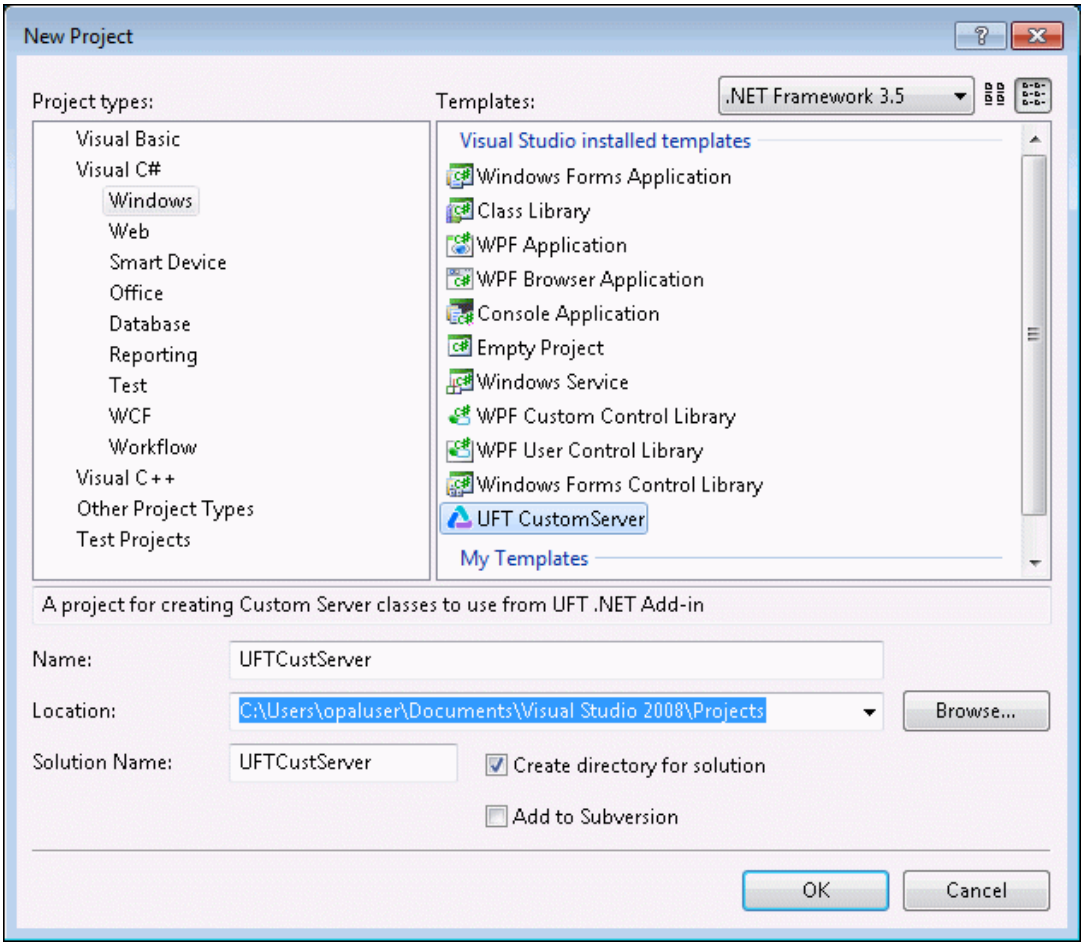
Developing a New Support Set

The first step in creating support for a custom control is to create a new Custom Server project. This project will create support for the TrackBar control.

To create a new Custom Server project:

1. Open Microsoft Visual Studio.

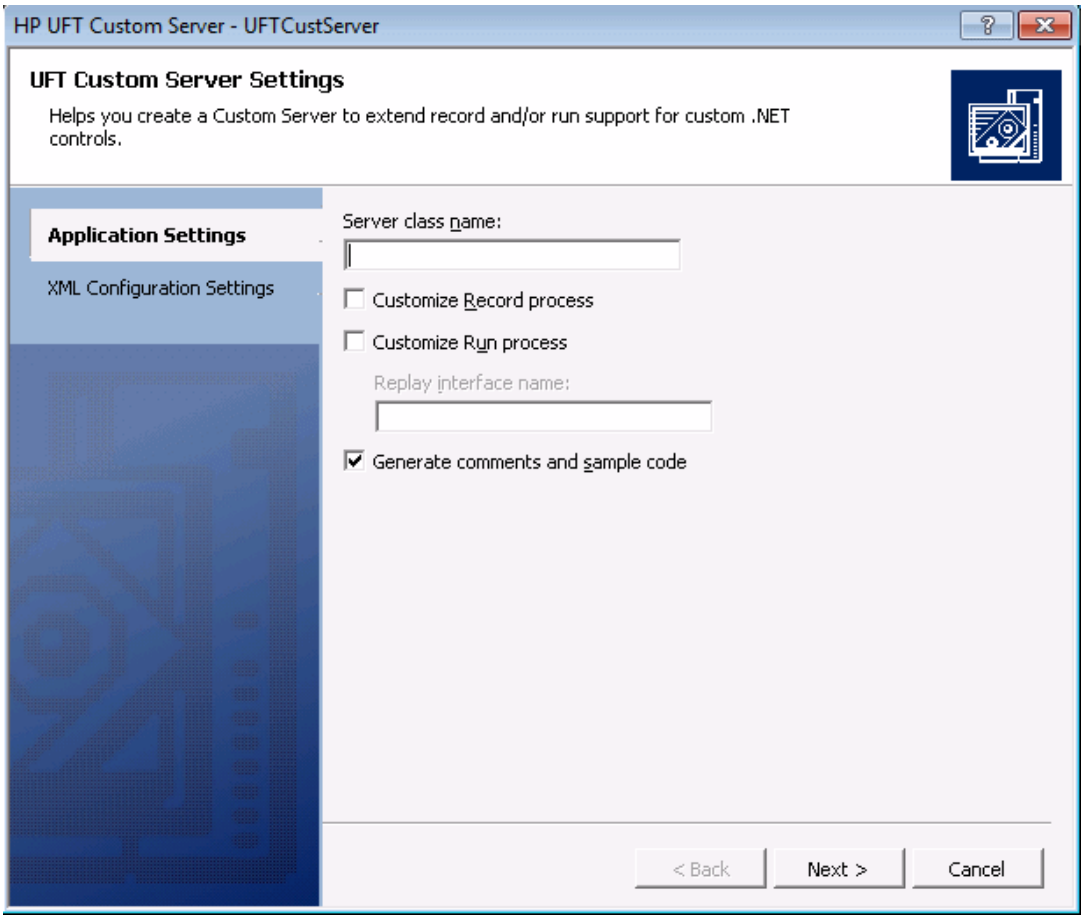
2. Select **File > New > Project**. The New Project dialog box opens.



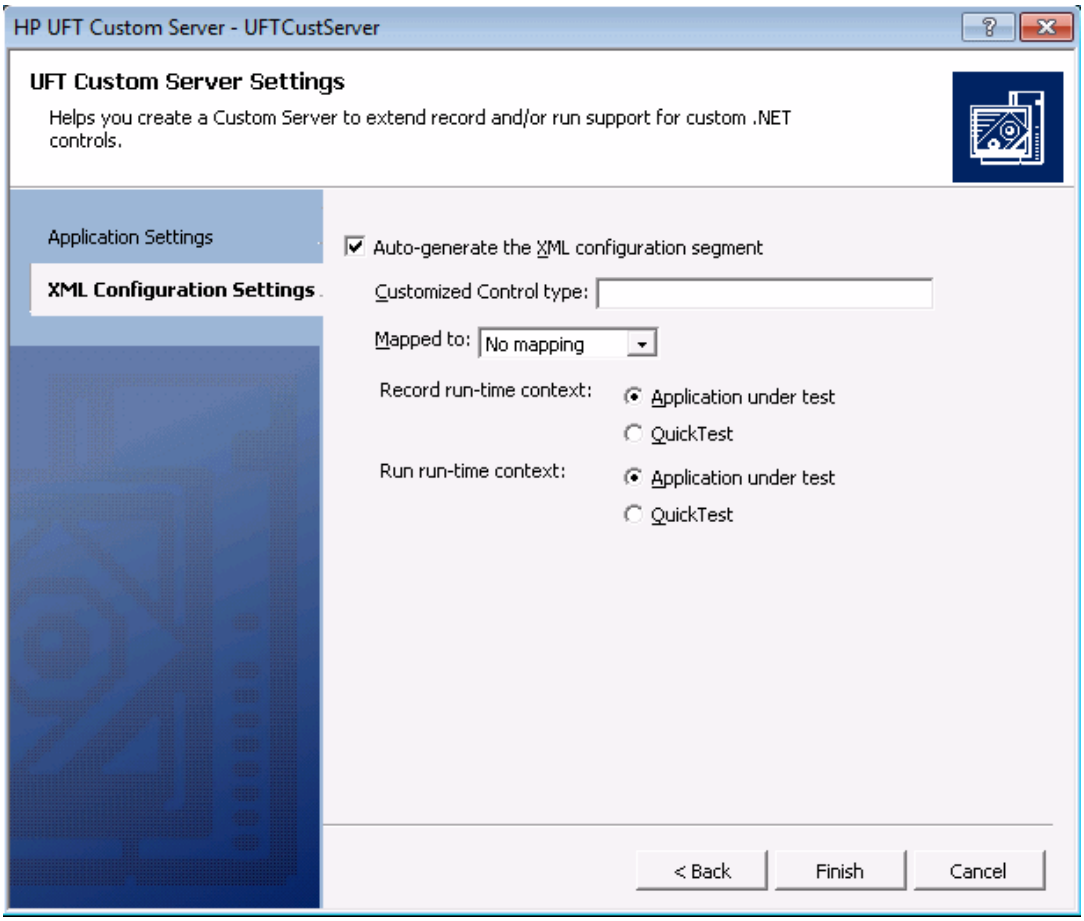
3. Specify the following settings:

   - Select the **Visual C# > Windows** node in the **Project types** tree. (In Microsoft Visual Studio versions other than 2008, the **UFT CustomServer** template may be located in a slightly different node in the tree.)
   - Select **UFT CustomServer** in the **Templates** pane.
   - In the **Name** box, specify the project name `UFTCustServer`.
   - Accept the rest of the default settings.

4. Click **OK**. The UFT Custom Server Settings wizard opens.

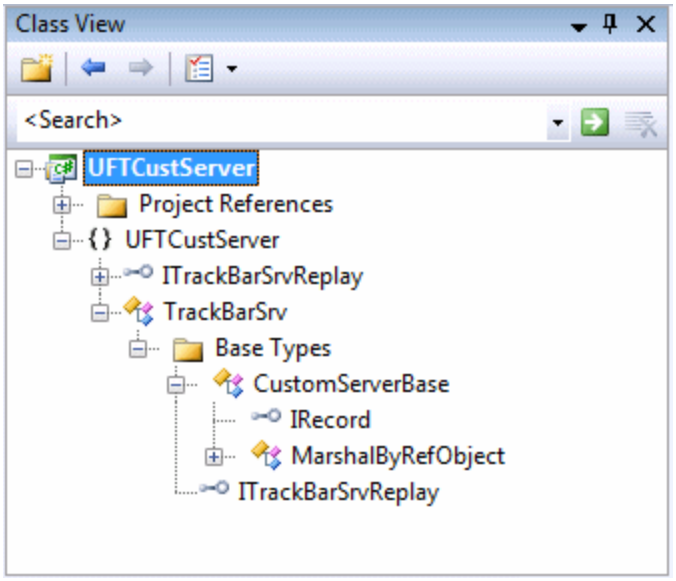


5. In the Application Settings page, specify the following settings:
   - In the **Server class name** box, enter `TrackBarSrv`.
   - Select the **Customize Record process** check box.
   - Select the **Customize Run process** check box.
   - Accept the rest of the default settings.
6. Click **Next**. The XML Configuration Settings page opens.



7. In the XML Configuration Settings page, specify the following settings:
   - Make sure the **Auto-generate the XML configuration segment** check box is selected.
   - In the **Customized Control type** box, enter `System.Windows.Forms.TrackBar`.
   - Accept the rest of the default settings.

8. Click **Finish**. In the Class View window, you can see that the wizard created a **TrackBarSrv** class derived from the **CustomServerBase** class and **ITrackBarSrvReplay** interface.



Implementing Test Record Logic

You will now implement the logic that records a **SetValue(X)** command when a **ValueChanged** event occurs, using an event handler function.

To implement the Test Record logic:

1. In the **TrackBarSrv** class, locate an appropriate place to add a new method, **OnValueChanged**. For example, you might want to add it after other event handler methods, such as **OnMessage**, in the **IRecord override Methods** region.

2. Add the new method with the following signature to the **TrackBarSrv** class:

```
public void OnValueChanged(object sender, EventArgs e) { }
```

> **Note:** You can add the new method manually or use the wizard that Visual Studio provides for adding methods and functions to a class.

3. Add the following implementation to the function you just added (if copying and pasting, remove the redundant line breaks):

```
public void OnValueChanged(object sender, EventArgs e)
{
 System.Windows.Forms.TrackBar trackBar = (System.Windows.Forms.TrackBar)
```

```
sender;
 // get the new value
 int newValue = trackBar.Value;
 // Record SetValue command to the test
 RecordFunction("SetValue", RecordingMode.RECORD_SEND_LINE, newValue);
}
```

4. Register the OnValueChanged event handler for the ValueChanged event, by adding the following code to the InitEventListener method:

```
public override void InitEventListener()
{
  Delegate e = new System.EventHandler(this.OnValueChanged);
  AddHandler("ValueChanged", e);
}
```

Implementing Test Run Logic

You will now implement a **SetValue** method for the test Test Run.

To implement the Test Run logic:

1. Add the following method definition to the **ITrackBarSrvReplay** interface:

```
[ReplayInterface]
public interface ITrackBarSrvReplay
{
   void SetValue(int newValue);
}
```

2. Add the following method implementation to the **TrackBarSrv** class in the **Replay interface implementation** region (if copying and pasting, remove the redundant line breaks):

```
public void SetValue(int newValue)
{
 System.Windows.Forms.TrackBar trackBar = (System.Windows.Forms.TrackBar)
SourceControl;
 trackBar.Value = newValue;
}
```

3. Build your project.

Checking the TrackBarSrv.cs File

Following is the full source code for the TrackBarSrv class. Check that the contents of your **TrackBarSrv.cs** file is similar to the one illustrated below.

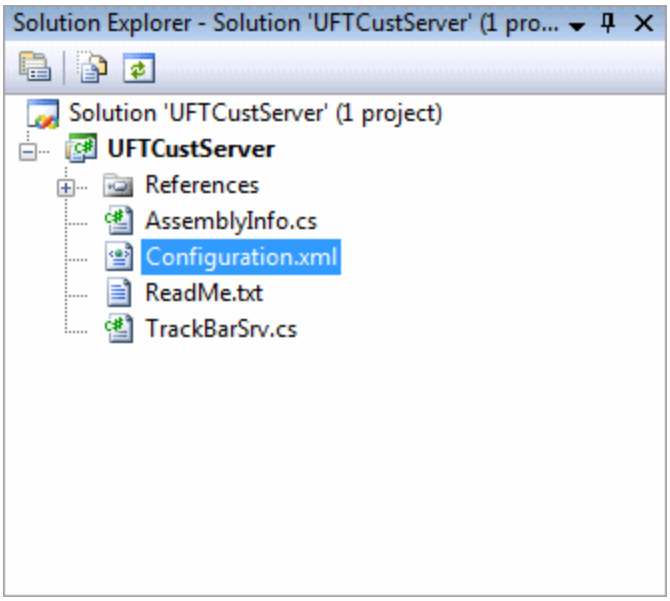
```
using System;
using Mercury.QTP.CustomServer;
namespace UFTCustServer
{
    [ReplayInterface]
    public interface ITrackBarSrvReplay
    {
        void SetValue(int newValue);
    }
    public class TrackBarSrv:
        CustomServerBase,
        ITrackBarSrvReplay
    {
        public TrackBarSrv()
        {
        }
        public override void InitEventListener()
        {
            Delegate  e = new System.EventHandler(this.OnValueChanged);
            AddHandler("ValueChanged", e);
        }
        public override void ReleaseEventListener()
        {
        }
        public void OnValueChanged(object sender, EventArgs e)
        {
            System.Windows.Forms.TrackBar trackBar =
                            (System.Windows.Forms.TrackBar)sender;
            int newValue = trackBar.Value;
            RecordFunction("SetValue",
                            RecordingMode.RECORD_SEND_LINE,
                            newValue);
        }
        public void SetValue(int newValue)
        {
            System.Windows.Forms.TrackBar trackBar =
                            (System.Windows.Forms.TrackBar)SourceControl;
            trackBar.Value = newValue;
        }
    }
}
```

Configuring and Deploying the Support Set

Now that you created the UFT Custom Server, you need to configure UFT to use this Custom Server when recording and running GUI tests on the **TrackBar** control.

To configure UFT to use the Custom Server:

1. In the Solution Explorer window, double-click the **Configuration.XML** file.



The following content should be displayed:

```
<!-- Merge this XML content into file "<UFT installation
folder>\dat\SwfConfig.xml". -->
<Control Type="System.Windows.Forms.TrackBar">
    <CustomRecord>
        <Component>
            <Context>AUT</Context>

            <DllName>D:\Projects\UFTCustServer\Bin\UFTCustServer.dll</Dll
Name>
            <TypeName>UFTCustServer.TrackBarSrv</TypeName>
        </Component>
    </CustomRecord>
    <CustomReplay>
        <Component>
            <Context>AUT</Context>
```

```
            <DllName>D:\Projects\UFTCustServer\Bin\UFTCustServer.dll</Dll
Name>
            <TypeName>UFTCustServer.TrackBarSrv</TypeName>
        </Component>
    </CustomReplay>
    <!--<Settings>
         <Parameter Name="sample name">sample value</Parameter>
    </Settings> -->
</Control>
```

2. Select the **<Control>...</Control>** segment and select **Edit > Copy** from the menu.
3. Open the **SwfConfig.xml** file located in **<UFT installation folder>\dat**.
4. Paste the **<Control>...</Control>** segment you copied from **Configuration.xml** into **SwfConfig.xml**, under the **<Controls>** tag in **SwfConfig.xml**. After you paste the segment, the **SwfConfig.xml** file should look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Controls>
    <Control Type="System.Windows.Forms.TrackBar">
        <CustomRecord>
            <Component>
                <Context>AUT</Context>

                <DllName>D:\Projects\UFTCustServer\Bin\UFTCustServer.dll<
/DllName>
                <TypeName>UFTCustServer.TrackBarSrv</TypeName>
            </Component>
        </CustomRecord>
        <CustomReplay>
            <Component>
                <Context>AUT</Context>

                <DllName>D:\Projects\UFTCustServer\Bin\UFTCustServer.dll<
/DllName>
                <TypeName>UFTCustServer.TrackBarSrv</TypeName>
            </Component>
        </CustomReplay>
    </Control>
</Controls>
```

5. Make sure that the **<DllName>** elements contain the correct path to your Custom Server DLL.
6. Save the **SwfConfig.xml** file.

Testing the Support Set

You can now verify that UFT records and runs GUI tests as expected on the custom TrackBar control by testing the Custom Server.

To test the Custom Server:

1. Open UFT with the .NET Add-in loaded, and open a GUI test.
2. Start recording on a .NET application with a **System.Windows.Forms.TrackBar** control.
3. Click the **TrackBar** control. UFT should record commands such as:

```
SwfWindow("Form1").SwfObject("trackBar1").SetValue 2
```

4. Run the test. The TrackBar control should receive the correct values.

Chapter 7: Learning to Create Support for a Complex Custom .NET Windows Forms Control

In this tutorial, you will learn how to build a Custom Server for controls that require more complex implementation solutions, so that UFT can record and run operations on these controls. You will implement the Custom Server in C#. A Custom Server can be similarly implemented in Visual Basic.

The explanations in this chapter assume that you are familiar with .NET Add-in Extensibility concepts and already know how to implement a Custom Server.

This chapter includes:

SandBar Toolbar Example

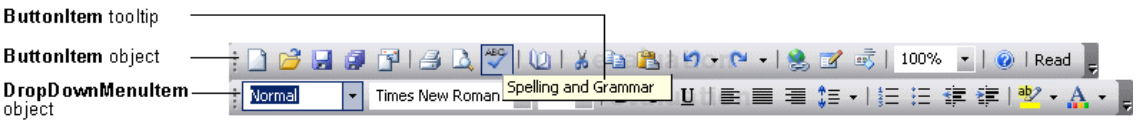
This example demonstrates how to implement .NET Add-in Extensibility for the Divelements Limited **TD.SandBar.Toolbar** control.

You can view the full source code of the final **ToolBarSrv.cs** class implementation in "Understanding the ToolBarSrv.cs File" on page 94.

A complete support set for the SandBar control, implemented both in C# and in Visual Basic, is located in **<UFT .NET Add-in Extensibility SDK installation folder>\samples\WinFormsExtSample**. You can use the files in this sample as an additional reference when performing this tutorial. For more information, see "Using the .NET Add-in Extensibility Samples" on page 65.

Tip: You can download an evaluation copy of the **TD.SandBar.Toolbar** control from: http://www.divil.co.uk/net/download.aspx?product=2&license=5.

The **Toolbar** control appears as follows:



The **Toolbar** control is comprised of a variety of objects, such as:

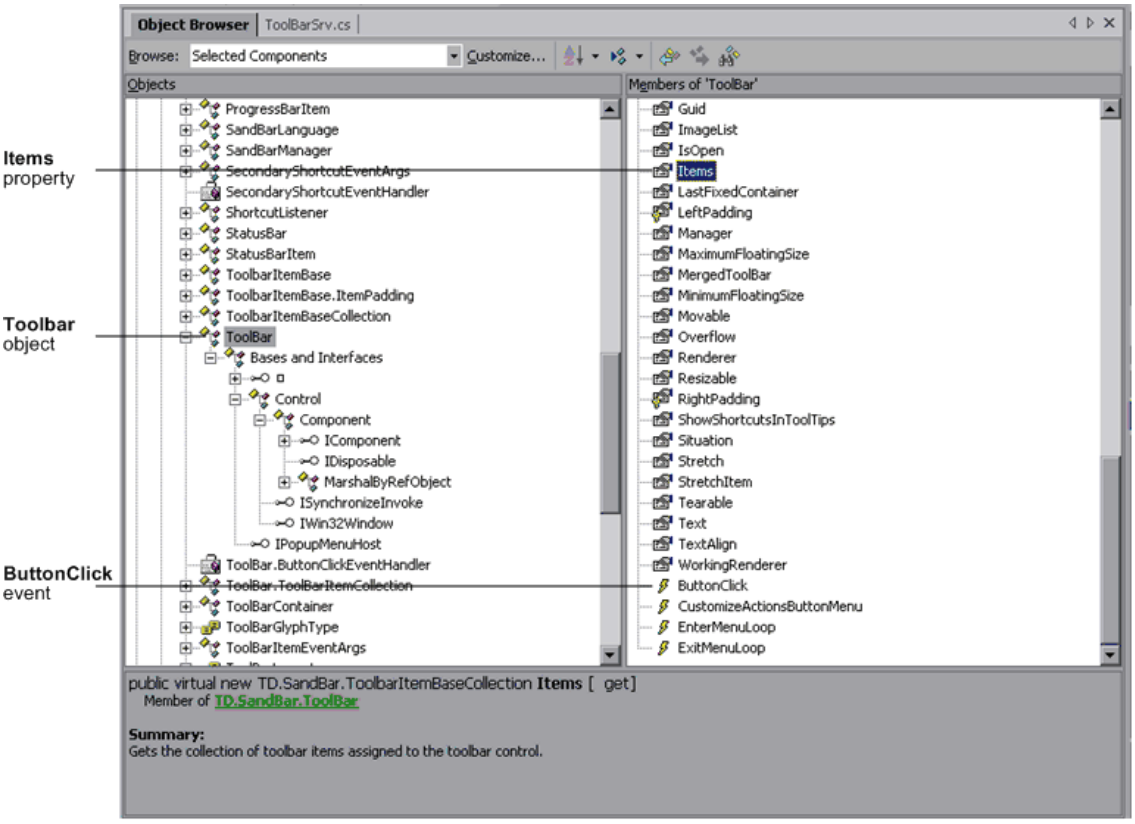
- **ButtonItem** objects, which represent buttons in the toolbar. **ButtonItem** objects contain images and no text. Each **ButtonItem** object has a unique tooltip.
- **DropDownMenuItem** objects, which represent drop-down menus in the toolbar.

Both the **ButtonItem** object and the **DropDownMenuItem** object are derived from the **ToolbarItemBase** object.

When you implement a Custom Server for a custom control, you want UFT to support recording and running the user's actions on the custom controls. When recording the test, your Custom Server listens to the control's events and handles the events to perform certain actions to add steps to the UFT GUI test. When running the test, you simulate (replay) the same actions the user performed on that control.

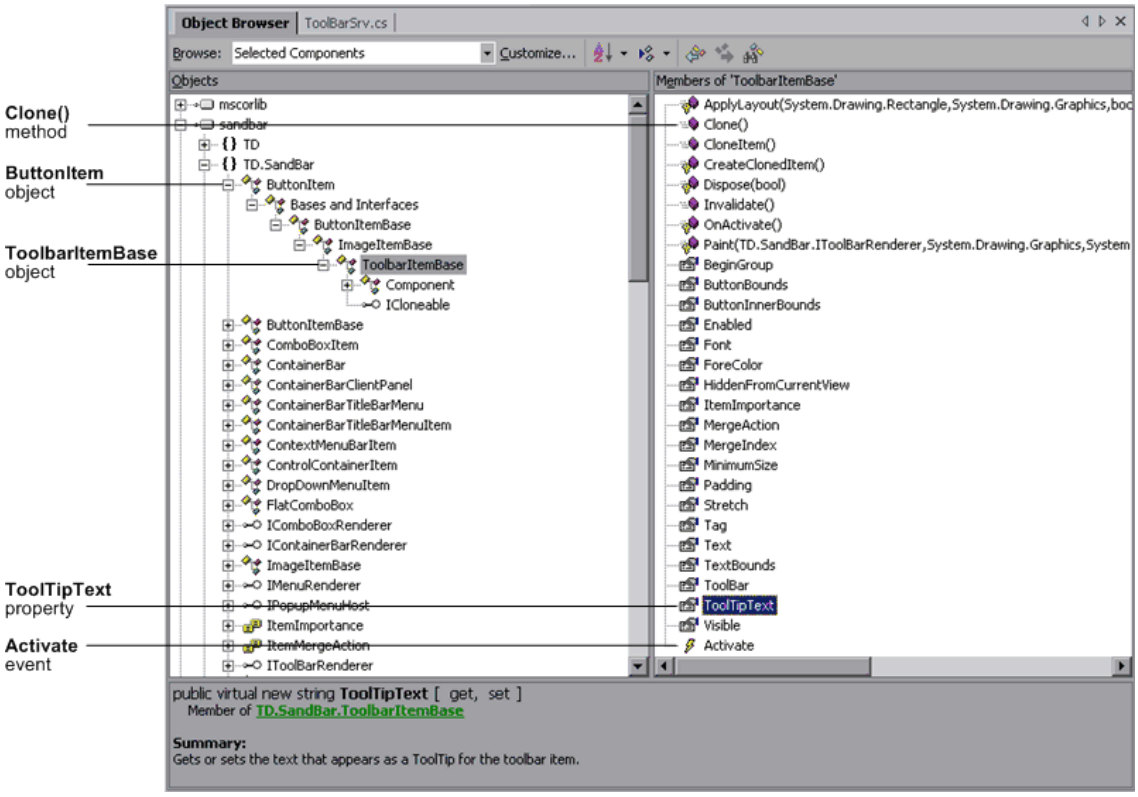
For example, suppose you want to implement a user pressing a button on a custom toolbar. Before doing so, you must understand the toolbar control, its properties and methods, and understand how you can use them to implement the Custom Server.

Following are some of the SandBar **ToolBar** object's properties and events (methods are not visible in this image) as displayed in the Object Browser in Visual Studio:



As you can see in the image above, the **ToolBar** object has a property called **Items** that retrieves the collection of **ToolbarItemBase** objects assigned to the **ToolBar** control. You can also see that the **ToolBar** control has an event called **ButtonClick**. Your Custom Server can listen to the **ButtonClick** event to know when a button in the toolbar is clicked. However, this event does not indicate which specific button in the toolbar is clicked.

Now expand the **ButtonItem** object and review its properties, methods, and events:

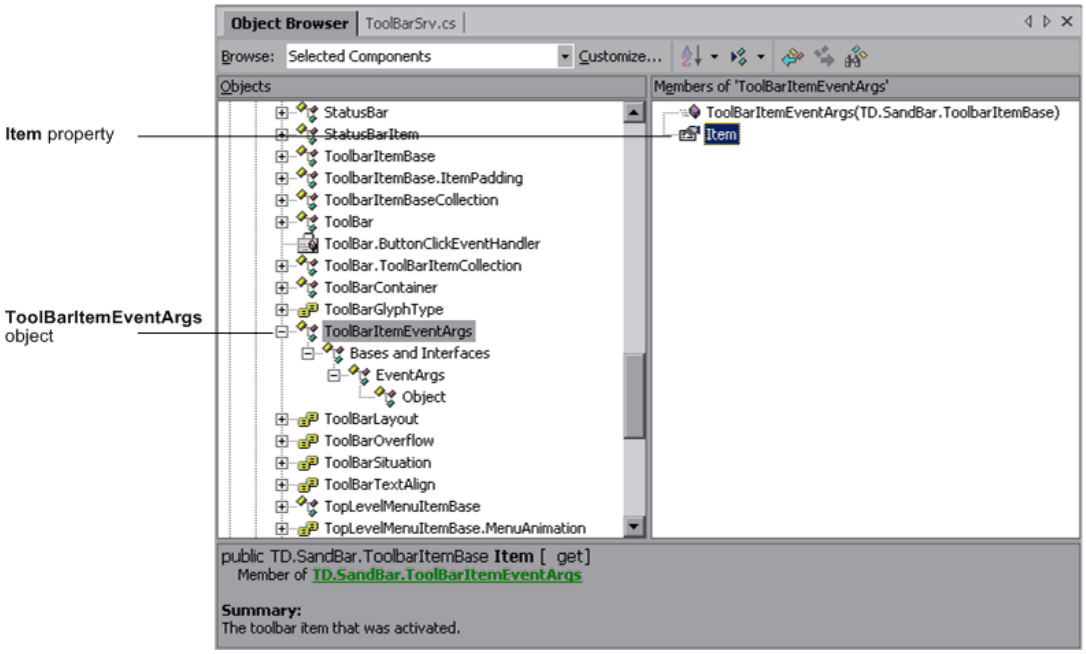


As shown in the image above, the **ButtonItem** object is derived from the **ToolbarItemBase** object. You can see that the **ToolbarItemBase** object contains a **ToolTipText** property, but does not contain a **Click** event or method.

When you look at the custom toolbar object, the following possible implementation issues arise:

1. **When handling a ButtonClick event during recording, how can you tell which button in the toolbar was clicked?**

Solution: All of the **ToolBar** object's events are **ToolBarItemEventArgs** events that are derived from the **EventArgs** object:



The **Item** property indicates which toolbar item (button) raised the event. You can use that toolbar item's unique **ToolTipText** property to recognize which button was clicked and add that to the UFT GUI test.

To do this, enter the following code in the **Record events handlers** section of the **ToolBarSrv.cs** file:

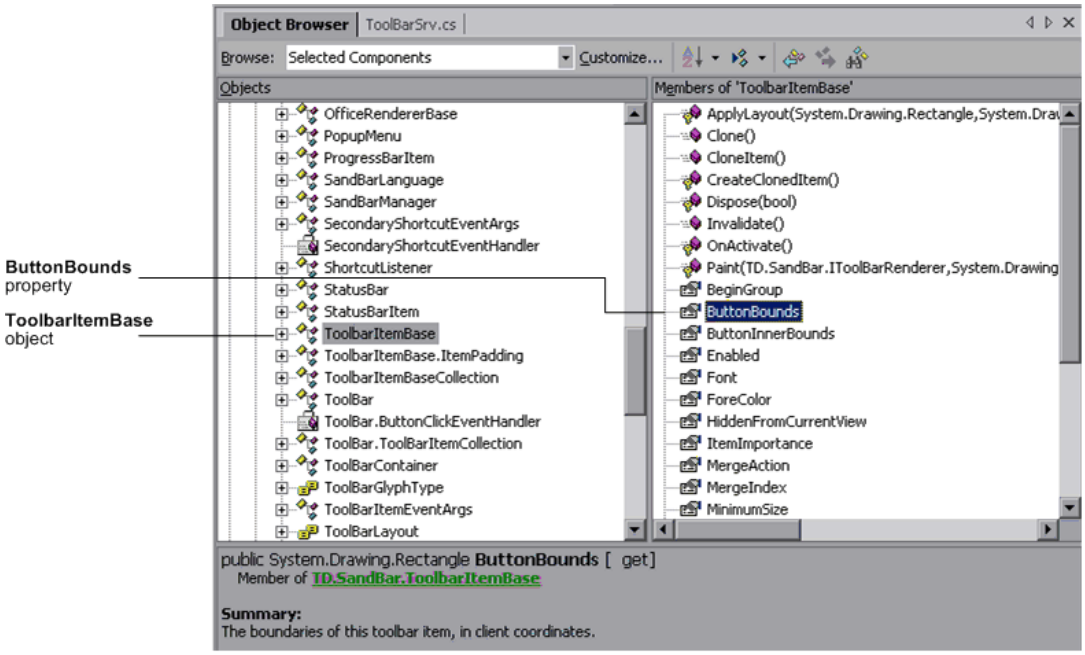
```
#region Record events handlers
private void oControl_ButtonClick(object sender,
TD.SandBar.ToolBarItemEventArgs e)
{
    TD.SandBar.ToolBar  oControl = (TD.SandBar.ToolBar)SourceControl;
    // Add a step in the test for the test object with the
    // ClickButton method and the tooltip text as an argument
    base.RecordFunction("ClickButton", RecordingMode.RECORD_SEND_LINE,
e.Item.ToolTipText);
}
#endregion
```

Now, each time you record a click on a button in the toolbar, a step is added to the test for the toolbar test object with the **ClickButton** method and the tooltip text of the button as its argument. For example:

```
SwfToolbar("MySandBar").ClickButton "Spelling and Grammar"
```

2. **When running a test, how do you perform a ClickButton method, when the ButtonItem object does not contain a Click method or event, and you know only the ButtonItem object's tooltip text?**

   Solution: The **ToolbarItemBase** object has a property called **ButtonBounds**:



You can loop through all of the **ToolbarItemBase** objects until you find a **ToolbarItemBase** objects that has the same tooltip text as the **ButtonItem** object, find that **ToolbarItemBase** object's rectangle boundaries, calculate the middle of its boundaries, and click that point.

To do this, enter the following code in the **Replay interface implementation** section of the **ToolBarSrv.cs** file:

```
#region Replay interface implementation
public void  ClickButton(string  text)
{
TD.SandBar.ToolBar  oControl = (TD.SandBar.ToolBar)SourceControl;
    //Find the correct item in the toolbar according to
    // its tooltip text.
```

```
    for(int i=0; i<oControl.Items.Count; i++)
    {
        //Found the correct ButtonItem
        if(oControl.Items[i].ToolTipText == text)
        {
            //Retrieve the rectangle of the button's boundaries
            // and locate its center
            System.Drawing.Rectangle oRect = oControl.Items
[i].ButtonBounds;
            int x = oRect.X + oRect.Width/2;
            int y = oRect.Y + oRect.Height/2;
            //Click the middle of the button item
            base.MouseClick(x, y, MOUSE_BUTTON.LEFT_MOUSE_BUTTON);
            break;
        }
    }
    //Add the step to the report
    base.ReplayReportStep("ClickButton", EventStatus.EVENTSTATUS_GENERAL,
text);
}
#endregion
```

Understanding the ToolBarSrv.cs File

Following is the full source code for the **ToolBarSrv.cs** class, used to implement UFT record and run support for the **TD.SandBar.Toolbar** control:

```
using System;
using Mercury.QTP.CustomServer;
//using TD.SandBar;

namespace ToolBar
{
       [ReplayInterface]
       public interface IToolBarSrvReplay
       {
              void  ClickButton(string  text);
       }
       /// <summary>
       /// Summary description for ToolBarSrv.
       /// </summary>
       public class ToolBarSrv:
              CustomServerBase,
              IToolBarSrvReplay
       {
              // You shouldn't call Base class methods/properties at the constructor
              // since its services are not initialized yet.
              public ToolBarSrv()
              {
```

```
        //
        // TODO: Add constructor logic here
        //
}
#region IRecord override Methods
#region Wizard generated sample code (commented)
/// <summary>
/// To change Window messages filter, implement this method.
/// The default implementation is to get only the control's
/// Windows messages.
/// </summary>
public override WND_MsgFilter GetWndMessageFilter()
{
        return(WND_MsgFilter.WND_MSGS);
}

/*
/// <summary>
/// To catch Windows messages, you should implement this method.
/// Note that this method is called only if the CustomServer is running
/// under UFT process.
/// </summary>
public override RecordStatus OnMessage(ref Message tMsg)
{
        // TODO:  Add OnMessage implementation.
        return RecordStatus.RECORD_HANDLED;
}
*/
#endregion

/// <summary>
/// If you are extending the Record process, you should add your event
/// handlers to listen to the control's events.
/// </summary>
public override void InitEventListener()
{
           TD.SandBar.ToolBar  oControl = (TD.SandBar.ToolBar)SourceControl;
           oControl.ButtonClick += new
           TD.SandBar.ToolBar.ButtonClickEventHandler(oControl_ButtonClick);
           //AddHandler("ButtonClick", new
           //TD.SandBar.ToolBar.ButtonClickEventHandler(oControl_ButtonClick));
}

/// <summary>
/// At the end of the Record process, this method is called by UFT to
/// release all the handlers the user added in the InitEventListener method.
/// Note that handlers added via UFT methods are released by
/// the UFT infrastructure.
/// </summary>
public override void ReleaseEventListener()
{
        TD.SandBar.ToolBar  oControl = (TD.SandBar.ToolBar)SourceControl;
        oControl.ButtonClick -= new
        TD.SandBar.ToolBar.ButtonClickEventHandler(oControl_ButtonClick);
}
#endregion

#region Record events handlers
```

```
        private void oControl_ButtonClick(object sender,
                                            TD.SandBar.ToolBarItemEventArgs e)
        {
                TD.SandBar.ToolBar  oControl = (TD.SandBar.ToolBar)SourceControl;
                // Add a step in the test for the test object with the ClickButton method
                // and the tooltip text as an argument
                base.RecordFunction("ClickButton",
                                RecordingMode.RECORD_SEND_LINE, e.Item.ToolTipText);
        }
        #endregion
        #region Replay interface implementation
        public void  ClickButton(string text)
        {
        TD.SandBar.ToolBar  oControl = (TD.SandBar.ToolBar)SourceControl;
        //Find the correct item in the toolbar according to its tooltip text.
        for(int i=0; i<oControl.Items.Count; i++)
        {
                //Found the correct ButtonItem
                if(oControl.Items[i].ToolTipText == text)
                {
                // Retrieve the rectangle of the button's boundaries and
                // locate its center
                System.Drawing.Rectangle oRect=oControl.Items[i].ButtonBounds;
                int x = oRect.X + oRect.Width/2;
                int y = oRect.Y + oRect.Height/2;
                //Click the middle of the button item
                base.MouseClick(x, y, MOUSE_BUTTON.LEFT_MOUSE_BUTTON);
                break;
                }
        }
        //Add the step to the report
        base.ReplayReportStep("ClickButton",
                                        EventStatus.EVENTSTATUS_GENERAL, text);
        }
        #endregion
    }
}
```

We appreciate your feedback!

If you have comments about this document, you can contact the documentation team by email. If an email client is configured on this system, click the link above and an email window opens with the following information in the subject line:

Feedback on Developer Guide (UFT .NET Add-in Extensibility 12.00)

Just add your feedback to the email and click send.

If no email client is available, copy the information above to a new message in a web mail client, and send your feedback to sw-doc@hp.com.


Java™
COMPATIBLE