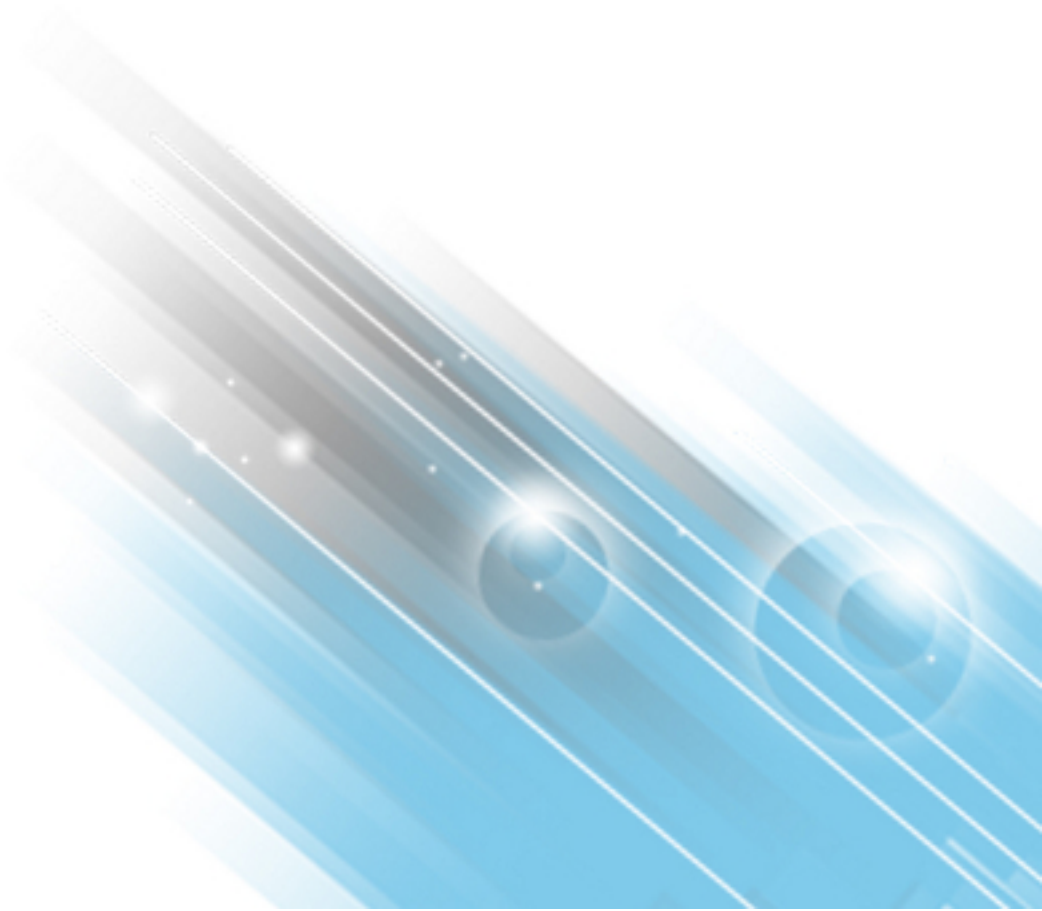
HP UFT WPF and Silverlight Add-in Extensibility

Software Version: 12.50
Windows ® operating systems

Developer Guide



Document Release Date: June 2015
Software Release Date: June 2015

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: https://softwaresupport.hp.com/group/softwaresupport/search-result.

This site requires an HP Passport account. If you do not have one, click the **Create an account** button on the HP Passport Sign in page.

Support

Visit the HP Software Support Online web site at: https://softwaresupport.hp.com

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: https://softwaresupport.hp.com and click **Register**.

To find more information about access levels, go to: https://softwaresupport.hp.com/web/softwaresupport/access-levels.

HP Software Solutions & Integrations and Best Practices

Visit **HP Software Solutions Now** at https://h20230.www2.hp.com/sc/solutions/index.jsp to explore how the products in the HP Software catalog work together, exchange information, and solve business needs.

Visit the **Cross Portfolio Best Practices Library** at https://hpln.hp.com/group/best-practices-hpsw to access a wide variety of best practice documents and materials.

Contents

Welcome to HP UFT WPF and Silverlight Add-in Extensibility 6
About the UFT WPF and Silverlight Add-in Extensibility SDK 6
About the UFT WPF and Silverlight Add-in Extensibility Developer Guide 7
Who Should Read This Guide 8
Additional Online Resources 9

Chapter 1: Developing UFT Support for a Custom WPF or Silverlight Toolkit 10
About Developing WPF or Silverlight Add-in Extensibility Toolkit Support Sets 11
The Test Object Configuration XML File 11
Custom Servers 15
Utility Methods and Properties 16
WPF Add-in Extensibility Sample 16
How to Create Support for a Custom WPF or Silverlight Toolkit 17
How to Add Support for a Custom WPF or Silverlight Control 20
How to Develop a Custom Server 22
WPF/Silverlight Custom Server Setup Dialog Box (in Microsoft Visual Studio) 26
Troubleshooting and Limitations - Developing Support 30

Chapter 2: Tutorial: Create UFT Support for a Custom WPF Control 32
Planning Support for the WPF Calendar Control 34
Setting Up the WPF Add-in Extensibility Project for the WPF Calendar Control 36
Designing the Toolkit Configuration File 39
Designing the Test Object Configuration File 40
Deploying and Testing the Preliminary Toolkit Support Set 42
Design the Basic Custom Server 45
Implement Support for Retrieving Identification Property Values 46
Deploy and Test Your Basic Custom Server and Identification Property Support 47
Implement Support for Running Test Object Operations 48
Deploy and Test Your Support for Test Object Operations 50
Implement Support for Recording 51
Deploy and Test Your Support for Recording 53

Chapter 3: Deploying the Toolkit Support Set 54
About Deploying the Custom Toolkit Support 55
Deploying the Custom Toolkit Support 55
Setting the DevelopmentMode Attribute 57
Modifying Deployed Support 57

Modifying Identification Property Attributes in a Test Object Configuration File 57
Removing Deployed Support 58

Send Us Feedback 60

Welcome to HP UFT WPF and Silverlight Add-in Extensibility

HP UFT WPF and Silverlight Add-in Extensibility is an SDK (Software Development Kit) package that enables you to support testing applications that use third-party and custom WPF or Silverlight controls that are not supported out-of-the-box by the Unified Functional Testing WPF and Silverlight Add-ins.

You must develop support for WPF and Silverlight controls separately, and use different APIs. However, creating Silverlight support is very similar to creating WPF support, therefore both are described together in this guide.

This chapter includes:

- About the UFT WPF and Silverlight Add-in Extensibility SDK 6
- About the UFT WPF and Silverlight Add-in Extensibility Developer Guide 7
- Who Should Read This Guide 8
- Additional Online Resources 9

About the UFT WPF and Silverlight Add-in Extensibility SDK

The UFT WPF and Silverlight Add-in Extensibility SDK installation provides the following:

- APIs that enable you to extend the Unified Functional Testing WPF or Silverlight Add-in to support custom WPF or Silverlight controls.
- WPF and Silverlight Custom Server C# 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).
- A Custom Server Setup dialog box in Visual Studio that enables you to customize a project you are creating based on a Custom Server template.
- The WPF and Silverlight Add-in 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.

  - API References.
  - A Toolkit Configuration 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 WPF Add-in Extensibility support set that extends UFT GUI testing support for the **Microsoft.Windows.Controls.Calendar** custom control.

Accessing UFT WPF and Silverlight 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 WPF and Silverlight Add-in Extensibility Developer Guide

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

This guide assumes you are familiar with UFT functionality, and should be used together with the following documents, provided in the WPF and Silverlight Add-in Extensibility Help (**Start > All Programs > HP Software > HP Unified Functional Testing > Extensibility > Documentation > WPF and Silverlight Add-in Extensibility Help**):

- *API References*
- *Toolkit Configuration Schema Help*
- *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 WPF or Silverlight 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 WPF or Silverlight custom controls.

To use this guide, you should be familiar with:

- Major UFT features and functionality
- The UFT Object Model
- Unified Functional Testing WPF or Silverlight Add-in
- WPF or Silverlight programming in C#
- 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: Developing UFT Support for a Custom WPF or Silverlight Toolkit

Many WPF (Windows Presentation Foundation) and Silverlight customer applications include use of non-Microsoft controls in their UI. UFT represents these controls with the generic WpfObject or SlvObject test object respectively.

In other cases, UFT recognizes a complex control as a set of low-level controls, instead of recognizing the functional significance of the high-level control. For example, UFT might recognize a custom WPF or Silverlight calendar control as several unrelated buttons and text boxes.

In these cases, the tester cannot run any methods containing logic specific to the custom control type. Nor can the tester apply any recording logic specific to this control type.

By creating a toolkit support set using WPF or Silverlight Add-in Extensibility, you define new test object classes to represent these custom controls. The support set gives QA engineers the ability to run, record, learn and spy on custom WPF or Silverlight controls. You must create separate toolkit support sets for WPF and Silverlight controls, and use different APIs.

This chapter includes:

- About Developing WPF or Silverlight Add-in Extensibility Toolkit Support Sets 11
- The Test Object Configuration XML File 11
- Custom Servers 15
- Utility Methods and Properties 16
- WPF Add-in Extensibility Sample 16
- How to Create Support for a Custom WPF or Silverlight Toolkit 17
- How to Add Support for a Custom WPF or Silverlight Control 20
- How to Develop a Custom Server 22
- WPF/Silverlight Custom Server Setup Dialog Box (in Microsoft Visual Studio) 26
- Troubleshooting and Limitations - Developing Support 30

About Developing WPF or Silverlight Add-in Extensibility Toolkit Support Sets

Implement WPF and Silverlight Add-in Extensibility in C# using 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).

- Visual Studio is required only to develop the support set, not to use it.
- To develop Silverlight Add-in Extensibility, you must have the Microsoft Silverlight Tools for Visual Studio installed.
- UFT is required only to run and test your support set, not to develop it.

A toolkit, or an environment, is a set of controls for which you want to provide support in one package.

A toolkit support set consists of:

- **A Test Object Configuration XML File.** In this file, new test object types are defined. For details, see "The Test Object Configuration XML File" below.
- **A toolkit configuration file.** In this file, WPF or Silverlight control types are mapped to test object types (classes) and to the custom servers that implement their record and run logic. For details on the structure and syntax of this file, see the *Toolkit Configuration Schema Help* (available with the WPF and Silverlight Add-in Extensibility Help).
- **.Net DLLs containing the implementation of custom servers.** For details, see "Custom Servers " on page 15.
- **Icon and Help files (Optional).**

  The icon files contain icons used in UFT to represent your test object classes. (Supported file types: **.ico**, **.exe**, **.dll**)

  The Help files are used for context-sensitive Help for your test object classes and their methods and properties. (Supported file type: **.chm**)

The Test Object Configuration XML File

The first stage of developing support for a custom toolkit is to define the test object classes that you want UFT to use to represent your application controls. You define the test object classes in a test object configuration XML file. You need to create a test object class for every type of custom control for which you want to extend or modify UFT support.

In a test object configuration XML, you define the test object classes (for example, the test object methods they support, their identification properties, and so on).

You create a **ClassInfo** element for each test object class that you want to define. In addition, you define the name of the environment or custom toolkit for which the test object classes are intended (in the **PackageName** attribute of the **TypeInformation** element), and the UFT add-in which these test object classes extend (in the **AddinName** attribute of the **TypeInformation** element).

If the relevant add-in is not loaded when UFT opens, UFT does not load the information in this XML. Similarly, if the name of the environment or custom toolkit is displayed in the Add-in Manager dialog box and its check box is not selected, the information in this XML is not loaded.

To ensure the structural correctness of your test object configuration file, you can validate it against the **ClassesDefintions.xsd** file. This file is installed with UFT, in the **<UFT installation folder>\dat** folder. (For backward compatibility reasons, UFT still supports certain XML structures that do not pass validation against this XSD.)

The sections below describe the information that you can include in a test object class definition.

Class Name and Base Class

The name of the new test object class and its attributes, including the base class—the test object class that the new test object class extends. A new test object class extends an existing WPF or SilverlightUFT test object class, directly or indirectly. The base class may be a class delivered with UFT or a class defined using WPF or Silverlight Add-in Extensibility. (A WPF test object class must extend a WPF test object class and a Silverlight test object class must extend a Silverlight test object class.)

By default, the base class is WPFObject or SlvObject.

The test object class name must be unique among all of the environments whose support a UFT user might load simultaneously. For example, when defining a new test object class, do not use names of test object classes from existing UFT add-ins, such as WpfButton, WpfEdit,SlvButton and so on.

Note:

- A test object class inherits the base class' test object operations (methods and properties), generic type, default operation, and icon. Identification properties are not inherited.
- If you create test object classes that extend test object classes defined in another toolkit support set, you create a dependency between the two toolkit support sets. Whenever you select to load the extending toolkit support set in the UFT Add-in Manager, you must also select to load the toolkit support set that it extends.

Generic Type

The generic type for the new test object class, if you want the new test object class to belong to a different generic type than the one to which its base class belongs. (For example, if your new test object class extends WpfObject or SlvObject (whose generic type is **object**), but you would like UFT to group this test object class with the **edit** test object classes.)

Generic types are used when filtering objects (for example, in the Step Generator's Select Object for Step dialog box and when adding multiple test objects to the object repository). Generic types are also

used when creating documentation strings for the Documentation column of the Keyword View (if they are not specifically defined in the test object configuration file).

Test Object Operations

A list of operations for the test object class, including the following information for each operation:

- The arguments, including the argument type (for example, `String` or `Integer`), direction (`In` or `Out`), whether the argument is mandatory, and, if not, its default value.
- The operation description (shown in the Object Spy and as a tooltip in the Keyword View and Step Generator).
- The Documentation string (shown in the **Documentation** column of the Keyword View and in the Step Generator).
- The return value type.
- A context-sensitive Help topic to open when **F1** is pressed for the test object operation in the Keyword View or Editor, or when the **Operation Help** button is clicked for the operation in the Step Generator. The definition includes the Help file path and the relevant Help ID within the file.

Default Operation

The test object operation that is selected by default in the Keyword View and Step Generator when a step is generated for an object of this class.

Identification Properties

A list of identification properties for the test object class. You can also define:

- The identification properties that are used for the object description.
- The identification properties that are used for **smart identification**. (This information is relevant only if smart identification is enabled for the test object class. To enable smart identification, use the Object Identification dialog box in UFT.)
- The identification properties that are available for use in checkpoints and output values.
- The identification properties that are selected by default for checkpoints (in the UFT Checkpoint Properties dialog box).

Icon File

The path of the icon file to use for this test object class. (Optional. If not defined, the base class' icon is used.) The file can be a **.dll**, **.exe**, or **.ico** file.

Help File

A context-sensitive Help topic to open when **F1** is pressed for the test object in the Keyword View or Editor. The definition includes the **.chm** Help file path and the relevant Help ID within the file.

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

Sample Test Object Configuration File

An example of a WPF Add-in Extensibility test object configuration file is shown below. In a Silverlight Add-in Extensibility test object configuration file, the **AddinName** attribute in the **TypeInformation** element needs to be set to `Silverlight`.

In addition, in the toolkit name that you provide in the **PackageName** attribute, you may want to include an indication as to whether this is a WPF or Silverlight toolkit. This is recommended because the Add-in Manager in UFT displays both WPF and Silverlight Add-in Extensibility supported environments as child nodes under the WPF add-in node.

```
<TypeInformation Load="true" AddinName="WPF" PackageName="MyWpfToolkit“>
     <ClassInfo Name="MyWpfButton" BaseClassInfoName="WpfButton"
               ROTypeInfo="false"
               GenericTypeID="button"
               DefaultOperationName="Click"
               FilterLevel="0“>
        <IconInfo
            IconFile="INSTALLDIR\dat\Extensibility\WPF\MyWpfButton_icon.ico"/>
        <TypeInfo>
               <Operation Name="Click“>
                    <Argument Name="X" IsMandatory="false"
                                   DefaultValue="-9999" Direction="In“>
                         <Type VariantType="Integer"/>
                    </Argument>
                    <Argument Name="Y" IsMandatory="false"
                                   DefaultValue="-9999" Direction="In“>
                         <Type VariantType="Integer"/>
                    </Argument>
                    <Argument Name="MouseButton" IsMandatory="false"
                                   DefaultValue="0" Direction="In“>
                         <Type VariantType="Enumeration"
                                   ListOfValuesName="E_ButtonType"/>
                    </Argument>
               </Operation>
         </TypeInfo>
         <IdentificationProperties>
              <IdentificationProperty Name="devname" ForDescription="true"/>
              <IdentificationProperty Name="enabled" ForDescription="false"
                                             ForVerification="true"/>
          </IdentificationProperties>
     </ClassInfo>
</TypeInformation>
```

Custom Servers

For each custom control that you want to support, you develop a custom server class, that derives from the **CustomServerBase** class. The resulting custom server DLL runs in the context of the application and interfaces between UFT and the custom control. At UFT's request, it can retrieve the values of identification properties from the control, perform operations on the control, determine what steps to record in response to user activity on the control and so on. You can compile more than one custom server into a single DLL.

You implement each of these abilities by implementing the relevant interface in the custom server class. For details on the interface methods and their syntax, see the Custom Server API References (available with the WPF and Silverlight Add-in Extensibility Help).

- To support running test object operations, you develop a run interface that contains the methods that run the operations you defined in the test object configuration file. You must tag this interface with the **RunInterfaceAttribute** attribute.

  When developing support for a Silverlight control, you must tag each one of the methods that you design to implement running a test object operation with the Microsoft Silverlight **ScriptableMember** attribute.
- To support retrieving identification property values from the control, you develop a property interface that contains properties that retrieve the values for the identification properties you defined in the test object configuration file. You must tag this interface with the **CustomPropInterfaceAttribute** attribute.

  Note: In the test object configuration file, you must define all identification properties relevant for your test object class. However, the implementation for retrieving the property values is inherited from the base class for any properties that it supports.
- To support table checkpoints and output values, you implement the methods in the **ITableVerify** interface.
- To support recording, you implement the **IRecord** interface.
- To instruct UFT to ignore children of a control (because they are functionally part of the control, and not independent controls themselves), design the **IsKnownPartOf** method in the **IComponentDetector** interface to return true for those child controls.

For task details, see "How to Develop a Custom Server" on page 22.

When you design your custom server, you can use utility methods and properties provided by the WPF or Silverlight Add-in Extensibility API.

Utility Methods and Properties

The **CustomServerBase** class, which your custom server extends, includes a **UtilityObject** property that returns an object that provides utility methods and properties. You can call the following methods and properties in your custom server implementation.

- **AddHandler.** Registers an event handler to use when an event occurs on the control. The handler is added at the beginning of the event handler invocation list for this event.
- Mouse and Keyboard operation simulation methods. Use these methods in methods that perform steps on a control.
- **GetSettingsValue**, **GetSettingsXML.** Retrieve settings defined for this custom server in the toolkit configuration file.
- **Record.** Adds a step to the test and adds a test object to the object repository if it is not already there. Use this method in an event handler that records a step in a test after an event occurs on a control.
- **ReportStepResult.** Adds information about the results of a step to the run results. Use this method in a method that performs a step on a control.
- **ThrowRunError.** Throws an exception based on the specified error and sets the step status to **EventStatus.EVENTSTATUS_FAIL**.
- **ApplicationObject.** This property returns the control object with which the custom server is associated. For example, for a **CheckBoxCustomServer** associated with a check box, this property returns a reference to the check box. You can then use this reference to retrieve information (for example the value of the **IsChecked** property), or to perform some activity on the control (for example set its **IsChecked** property to `true`).

For details, see the **IUtilityObject** interface in the **Mercury.P.WPF.CustomServer** or **Mercury.QTP.Slv.CustomServer** namespace section in the *Custom Server API Reference* (available with the WPF and Silverlight Add-in Extensibility Help).

WPF Add-in Extensibility Sample

When you install the WPF Add-in Extensibility SDK, a custom WPF calendar control and a sample toolkit support set that extends support for this control are installed in the **<WPF Add-in Extensibility SDK installation folder>\samples\WPFExtCalendarSample** folder. You can study this sample to learn more about how to implement WPF and Silverlight Add-in Extensibility. You can also experiment with this sample, testing the control with UFT before and after deploying the sample toolkit support set.

To deploy the sample toolkit support set, place the provided XML and DLL files in the correct locations on the UFT computer, as described in "Developing UFT Support for a Custom WPF or Silverlight Toolkit" on page 10.

The files to deploy are:

- **<WPF Add-in Extensibility SDK installation folder>\samples\WPFExtCalendarSample\Support\QtCalendarSrv\MyWpfToolkit.cfg**
- **<WPF Add-in Extensibility SDK installation folder>\samples\WPFExtCalendarSample\Support\QtCalendarSrv\MyWpfToolkitTestObjects.xml**
- **<WPF Add-in Extensibility SDK installation folder>\samples\WPFExtCalendarSample\Support\QtCalendarSrv\bin\Release\QtCalendarSrv.dll**

How to Create Support for a Custom WPF or Silverlight Toolkit

This task describes how to create, deploy, and test a toolkit support set to extend UFT's support for a set of WPF or Silverlight custom controls.

You must create separate support for WPF and Silverlight controls. However, creating support for Silverlight controls is very similar to creating support for WPF controls, therefore both are described together in this task.

Tip:

- Start by creating a basic toolkit support set with one test object class and minimal functionality changes, and testing that UFT recognizes it correctly. Then gradually add more complex support and more test object classes, and test those as you add them.
- To create your WPF or Silverlight Add-in Extensibility files, use the **UFT WPF CustomServer** or **UFT Silverlight CustomServer** project template that the WPF and Silverlight Add-in Extensibility SDK installs on Visual Studio.

  Using this template helps set up the XML files and the custom server classes that you need to develop in your toolkit support set, simplifying the first three steps in the task described below. For details, see "WPF/Silverlight Custom Server Setup Dialog Box (in Microsoft Visual Studio)" on page 26.

This task includes the following steps:

- "Define new test object classes for UFT to use for your custom controls" on the next page
- "Map the custom controls to test object classes and custom servers" on the next page
- "Design the custom servers that contain the implementation of your support" on page 19
- "Deploy the toolkit support set to UFT" on page 19
- "Test the functionality of the support you developed" on page 19
- "Debug your support - Optional" on page 20

1. ## Define new test object classes for UFT to use for your custom controls

   a. Create a test object configuration XML file named **<custom toolkit name>TestObjects.xml**.

   After you deploy your support, UFT displays this name in the Add-in Manager as a child add-in under the WPF add-in. Therefore, you may want the name to indicate whether this is a WPF or Silverlight toolkit.

   b. In the test object configuration XML file, define new test object classes.

   - When creating support for WPF controls, create test object classes that extend existing UFT WPF test object classes (or other WPF Add-in extensibility test object classes).
   - When creating support for Silverlight controls, create test object classes that extend existing UFT Silverlight test object classes (or other Silverlight Add-in extensibility test object classes).

     For details on the structure and syntax of a test object configuration file, see "The Test Object Configuration XML File" on page 11.

2. ## Map the custom controls to test object classes and custom servers

   In a toolkit configuration XML file, map the custom control types to the test object classes that should represent them in UFT and to the custom servers that contain the implementation of your support. Create a **Control** element for each type of control you want to support.

   Name the file **<custom toolkit name>.cfg**.

   For details on the structure and syntax of a toolkit configuration file, see the *Toolkit Configuration Schema Help* (available with the WPF and Silverlight Add-in Extensibility Help).

   WPF Toolkit Configuration File Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Controls>
    <Control Type="MyCompany.MyDataGrid" MappedTo="MyWpfTable“>
        <CustomServer>
            <Component>
                <DllName>WpfCustomServers.dll</DllName>
                <TypeName>
                   MyCompany.MyWpfDataGridCustServer
                </TypeName>
            </Component>
        </CustomServer>
    </Control>
</Controls>
```

   Silverlight Toolkit Configuration File Example:

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

```
    <Control Type="MyCompany.MyDataGrid" MappedTo="MySlvTable“>
        <CustomServer>
            <Component>
                <DllName>SlvCustomServers.dll</DllName>
                <TypeName>MyCompany.MySlvDataGridCustServer,
                      SlvCustomServers, Version=1.0.0.0,
                      Culture=neutral, PublicKeyToken=null
                </TypeName>
            </Component>
        </CustomServer>
    </Control>
</Controls>
```

3. ### Design the custom servers that contain the implementation of your support

   Design one custom server for each type of control that you want to support.

   For details, see "How to Develop a Custom Server" on page 22.

4. ### Deploy the toolkit support set to UFT

   a. If you have completed the development and are deploying the support set for general use, make sure to set the **DevelopmentMode** attribute of the **TypeInformation** element in the test object configuration file to `false`.

   b. Deploy the toolkit support set by copying the files that you created to the correct locations under the UFT installation folder. For details, see "Developing UFT Support for a Custom WPF or Silverlight Toolkit" on page 10.

5. ### Test the functionality of the support you developed

   a. Open UFT. Ensure that your custom toolkit name is displayed in the Add-in Manager dialog box as a child of the WPF Add-in, and select it. (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.)

   > **Note:** If you are working with Silverlight, you must also select the Silverlight Add-in.

   b. Create and run UFT tests on your custom controls, and verify that UFT interacts with your controls as expected. Make sure that:

   - Controls are represented by the expected test object class in the Object Spy and when learning objects.
   - You can successfully create test objects of the classes you defined, using the Define New Test Object dialog box.
   - You can successfully create test steps using your test object classes in the Keyword View, Editor (using the statement completion functionality), and Step Generator.
   - Operations run correctly. Check the run support exposed by the classes based on CustomServerBase (Click, DblClick, on so on). Verify that operations that are not

implemented by the custom server are supported by the default UFT implementation for the base test object type.

- Table checkpoints and output values function correctly (if relevant).
- Identification property values are retrieved correctly. Check that identification properties not specifically implemented by the custom server are supported by the base class implementation (when relevant).
- Operations are recorded correctly both when record implementation is based on Windows messages and when based on events.

  If you are developing support for a Silverlight control, check that the correct messages are passed to the server, according to the Windows message filter defined by the custom server.
- Your toolkit and its test object classes are properly displayed in the relevant UFT dialog boxes: Object Identification, Available Keywords (for application areas), Define New Test Object, and so on.
- Check that elements of WPF or Silverlight that are not part of your support set have not had their functionality changed by installation of your support set.
- Verify that low level controls that are part of a high level control, as defined in the **IComponentDetector** implementation are not recognized by UFT during Record, Learn, Spy, and so on.

6. **Debug your support - Optional**

   To use Microsoft Visual Studio debugging tools to debug the support that you developed, attach Visual Studio to the application that contains the custom WPF or Silverlight controls, as the custom servers run in the context of this application.

How to Add Support for a Custom WPF or Silverlight Control

This task describes how to add support for a single type of custom WPF or Silverlight control to an existing toolkit support set.

For instructions on creating a toolkit support set, see "How to Create Support for a Custom WPF or Silverlight Toolkit" on page 17.

Tip: To create your WPF or Silverlight Add-in Extensibility files, use the **UFT WPF CustomServer** or **UFT Silverlight CustomServer** project template that the WPF and Silverlight Add-in Extensibility SDK installs on Visual Studio.

Using this template helps set up the XML data and the custom server class that you need to develop to support your custom control, simplifying the first three steps in the task described

below. For details, see "WPF/Silverlight Custom Server Setup Dialog Box (in Microsoft Visual Studio)" on page 26.

If necessary, you can move the XML data and custom server class that you create using the template into an existing toolkit support set. Copy the information from the XML files into the XML files of the existing toolkit support set, and copy the custom server **.cs** file into the existing Visual Studio WPF or Silverlight Add-in Extensibility project.

This task includes the following steps:

- "Define the test object class for UFT to use for your custom control - Optional" below
- "Map the custom control type to the relevant test object class and custom server" below
- "Design the custom servers that contain the implementation for UFT to run" below
- "Deploy and test the toolkit support set on UFT" below

1. ### Define the test object class for UFT to use for your custom control - Optional

   If your toolkit support set does not contain an appropriate test object class, add a **ClassInfo** element to the test object configuration XML file defining a new test object class.

   For details on the structure and syntax of a test object configuration file, see "The Test Object Configuration XML File" on page 11.

2. ### Map the custom control type to the relevant test object class and custom server

   In the toolkit configuration XML file, define a **Control** element for the custom control. Specify the test object class that UFT should use for the control, and the custom server that contains the implementation for supporting this control.

   For details on the structure and syntax of a toolkit configuration file, see the *Toolkit Configuration Schema Help* (available with the WPF and Silverlight Add-in Extensibility Help).

3. ### Design the custom servers that contain the implementation for UFT to run

   For details, see "How to Develop a Custom Server" on the next page.

4. ### Deploy and test the toolkit support set on UFT

   Deploy, test and debug the changes you made as part of the whole toolkit support set.

   For details see, "How to Create Support for a Custom WPF or Silverlight Toolkit" on page 17.

How to Develop a Custom Server

This task describes how to create a custom server that contains the implementation UFT needs to run to interact with the custom control.

This task is part of a higher-level task. For details, see "How to Create Support for a Custom WPF or Silverlight Toolkit" on page 17.

For additional details on the interface methods mentioned in this task, see the Custom Server API References (available with the WPF and Silverlight Add-in Extensibility Help).

This task includes the following steps:

- "Set up the Visual Studio project" below
- "Create the custom server class" on the next page
- "Develop support for test object operations" on the next page
- "Develop support for identification properties" on page 24
- "Develop support for table checkpoints and output values" on page 25
- "Develop support for recording steps" on page 25
- "Prepare your custom server for deployment" on page 26
- "How to Develop a Custom Server" above

1. ## Set up the Visual Studio project

   Do one of the following:

   Set up the project using an extensibility template:

   In Microsoft Visual Studio, create a new project using the **UFT WPF CustomServer** or **UFT Silverlight CustomServer** project template installed with the WPF and Silverlight Add-in Extensibility SDK.

   This sets up the files, references, and classes that you need to develop your custom server, simplifying the remainder of the steps in this task. For details, see "WPF/Silverlight Custom Server Setup Dialog Box (in Microsoft Visual Studio)" on page 26.

   Set up the project manually:

   a. Create a C# project in Visual Studio using the **Visual C# > Windows > Class Library** template or the **Visual C# > Silverlight > Silverlight Class Library** template.

   b. Add references to all necessary .NET framework libraries.

      For example, PresentationCore, PresentationFramework, and WindowsBase (when developing in WPF) or .NET Framework Class Library for Silverlight (when developing in Silverlight).

   c. Add references to the libraries that implement the custom control classes. For example, these may be third party libraries.

d. Add a reference to the DLL file that contains the WPF or Silverlight Add-in Extensibility API. The file is located in the **<WPF and Silverlight Add-in Extensibility installation folder>\SDK\WpfSlv** folder.

   - If you are developing support for a WPF control, add a reference to the **Mercury.QTP.WpfAgent.dll** file.
   - If you are developing support for a Silverlight control, add a reference to the **Mercury.QTP.Slv.CustomServer.dll** file.

     If you develop your toolkit support set on a computer that does not have the extensibility SDK installed, copy the DLL from the computer on which you installed WPF and Silverlight Add-in Extensibility.

e. Add all required references in the `Using` section.

   - To reference the WPF Add-in Extensibility API, add a reference to the **Mercury.QTP.WPF.CustomServer** namespace and not the to **Mercury.WpfAgent**.
   - To reference the Silverlight Add-in Extensibility API, add a reference to the **Mercury.QTP.Slv.CustomServer** namespace.

2. Create the custom server class

Create a custom server class for your custom control, extending the **CustomServerBase** class.

> **Note:** If you used the **UFT WPF/Silverlight CustomServer** template to set up the Visual Studio project, the class declaration is created automatically.

In the next steps, you implement various interfaces in this class, according to the UFT functionality that you want to support.

For example, your custom server class declaration might look like this:

```
public class MyCustomSupport:
       CustomServerBase,
       IMyCustomSupportRun,
       IRecord,
       ITableVerify,
       IMyCustomSupportCustProp,
       IComponentDetector
```

When designing the support, you can call utility methods from the **IUtilityObject** interface implemented by UFT in this base class. For details, see "Utility Methods and Properties" on page 16.

3. Develop support for test object operations

a. To support running new or modified test object operations that you defined for your test object class, implement one Run interface in your custom server, and tag this interface with the **RunInterfaceAttribute** attribute.

> **Note:** If you used the **UFT WPF/Silverlight CustomServer** template to set up the Visual Studio project, and specified that you want to customize running operations, the Run interface definition is created automatically.

b. In the Run interface, design a method for each test object operation that you want to support or override. Each method you design must have the same signature as the test object operation that it implements (as defined in the test object configuration file).

> **Note:** In the test object configuration file, you can define test object operations with optional arguments.
>
> When you develop the custom server methods that support running these test object operations, consider the following:
>
> **For WPF:** You must use the **Optional** attribute from the **System.Runtime.InteropServices** namespace to specify an optional parameter. For example, `void myMethod(int p1, [optional] int p2)`.
>
> **For Silverlight:** UFT does not support the use of C# annotations for optional parameters in these methods. Therefore, instead of designing one instance of a **RunInterface** method with a signature that includes optional parameters, design different instances of the method using the same method name, but different numbers of parameters.

c. If you are developing support for a Silverlight control, you must tag each one of the methods that you design to implement running a test object operation with the **ScriptableMember** attribute. (This is an existing Microsoft Silverlight attribute.)

> **Example:**
>
> If the custom server defines an **ICheckBoxRun** interface with the **Set** method to be used by UFT for running a **Set** operation, tag that interface with **RunInterfaceAttribute**.
>
> If this custom server is designed to support a Silverlight check box, tag the **Set** method with **ScriptableMemberAttribute**.

4. ## Develop support for identification properties

Design your custom server to retrieve identification property values from the control. Do this for any new identification properties you defined for your test object class, or if you want to override the value retrieval implementation inherited from the base class.

a. To support retrieval of identification properties values implement one Custom Properties interface in your custom server and tag this interface with the **CustomPropInterfaceAttribute** attribute.

> **Note:** If you used the **UFT WPF/Silverlight CustomServer** template to set up the Visual Studio project, and specified that you want to customize property retrieval, the interface definition is created automatically.

b. In the Custom Properties interface, define a property for each identification property whose value you want to retrieve from the control. Each property returns the value relevant for the identification property with the same name.

For example, if the custom server defines an **ICheckBoxCustomProp** interface with the **MyIsChecked** property to be used by UFT for retrieving the custom **MyIsChecked** identification property, mark the interface with **CustomPropInterfaceAttribute**.

Customizing the test object name (WPF only): If you implement a **logical_name** identification property, UFT uses its value as the test object name. This enables you to provide a functionally logical name for the test object (for example, the text displayed on the control). Otherwise, UFT generates a default name for the test object.

> **Note:** In the test object configuration file, you must define all identification properties relevant for your test object class. However, the implementation for retrieving the property values is inherited from the base class for any properties that it supports.

5. ## Develop support for table checkpoints and output values

To support table verification and output value retrieval, implement all of the methods in the **ITableVerify** interface in your custom server.

> **Note:** If you used the **UFT WPF/Silverlight CustomServer** template to set up the Visual Studio project, and specified that you want to design support for table checkpoints, a preliminary implementation of this interface is created automatically.

6. ## Specify children of the control that should not be treated as separate controls

To instruct UFT that certain child controls are part of a higher-level control, implement the **IComponentDetector** interface, and design the **IsKnownPartOf** method in the to return true for those child controls.

> **Example:**
>
> A calendar control might be implemented using buttons. To prevent UFT from learning these buttons as separate objects, or recording steps when the user clicks each button, the **IsKnownPartOf** method in the calendar's custom server returns `true` for any button that is part of the calendar.

> **Note:** If you used the **UFT WPF/Silverlight CustomServer** template to set up the Visual Studio project, and specified that you want to customize child object handling, a preliminary implementation of this interface is created automatically.

7. ## Develop support for recording steps

To support recording, implement the **IRecord** interface in your custom server class by overriding the callback methods.

Note: If you used the **UFT WPF/Silverlight CustomServer** template to set up the Visual Studio project, and specified that you want to customize recording, a preliminary implementation of this interface is created automatically.

a. Define and implement the event handlers required by your test object.

b. Define and implement the message handlers required by your test object.

- Implement **OnMessage** to listen directly to Windows messages.
  When developing support for a Silverlight control, the value returned from **OnMessage** indicates whether the custom server handled the message, or whether this message needs to be passed on to other registered event handlers.
- When developing support for a Silverlight control, implement **GetWndMessageFilter** to specify the objects on which to listen to Windows messages. You can listen to messages on the control itself (and the children considered and integral part of it), on the control's children, or on all messages to the application.

  See also the limitation about handling Windows messages in "Troubleshooting and Limitations - Developing Support" on page 30.

c. Implement **RecordInit** and **RecordStop** to register and release your handlers.

8. ### Prepare your custom server for deployment

Compile your custom server to create the DLL.

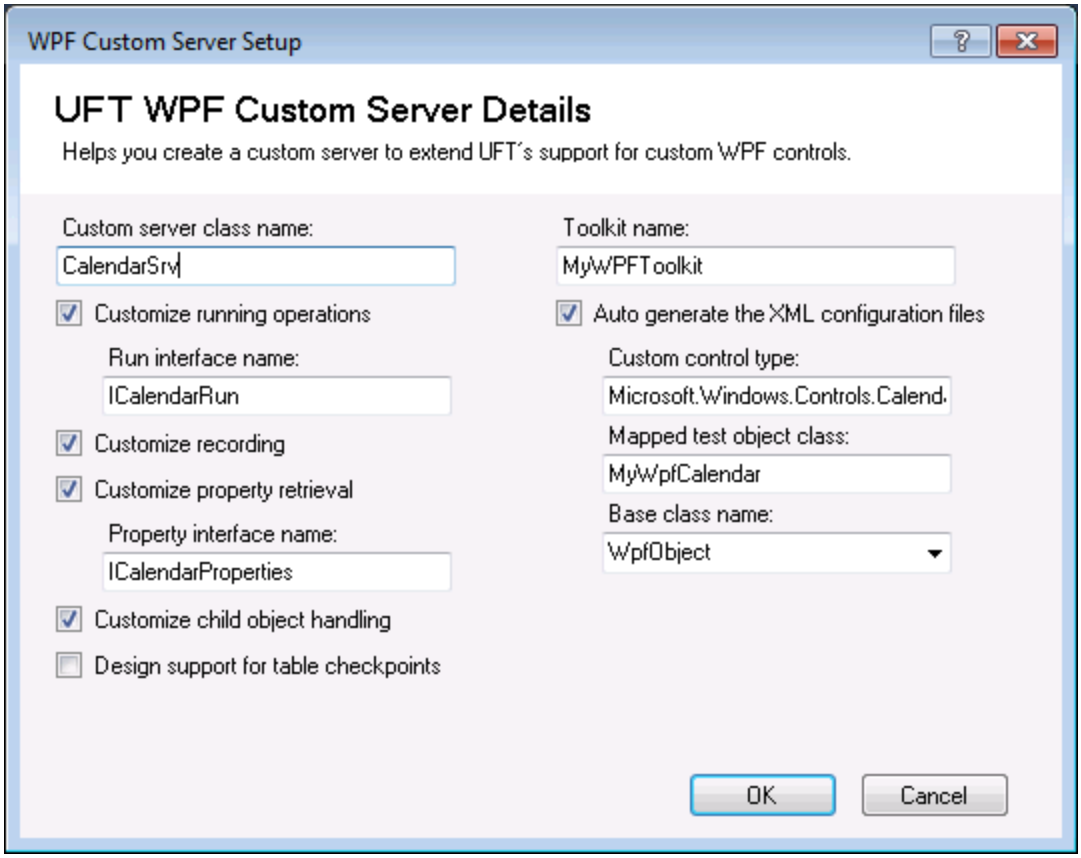
To enable your custom server to work with 32-bit and 64-bit applications, compile the custom server DLL with the **Platform target** option set to **Any CPU**.

WPF/Silverlight Custom Server Setup Dialog Box (in Microsoft Visual Studio)

This dialog box opens when you select the UFT WPF or Silverlight Custom Server template to create a WPF or Silverlight Add-in Extensibility project in Microsoft Visual Studio.

This dialog box enables you to provide specifications that describe the support that you want to create. When the extensibility project is created in Visual Studio, its files are created with the basic content, infrastructure, and references required to create the support you described.

This image displays a WPF Custom Server Setup dialog box. The options on a Silverlight Custom Server Setup dialog box are identical to the ones shown below. Only the dialog box titles and the list of available base classes are different for Silverlight.



To access	1. In Microsoft Visual Studio, select **File > New> Project**. 2. Select one of the following: • **Visual C# Windows** project type and **UFT WPF CustomServer** template • **Visual C# Silverlight** project type and **UFT Silverlight CustomServer** 3. Provide a name and location for your new project, and a name for the solution (or accept the default values provided). 4. Click **OK**. The WPF or Silverlight Custom Server Setup dialog box opens, depending on the template you selected. The project name is also used as the default value for some of the fields in the dialog box.
Important information	To successfully create a project using the **UFT Silverlight CustomServer** template, you must have the Microsoft Silverlight Tools for Visual Studio installed.
Relevant tasks	"How to Create Support for a Custom WPF or Silverlight Toolkit" on page 17
See also	• "Developing UFT Support for a Custom WPF or Silverlight Toolkit" on page 10

	• "Custom Servers " on page 15 • The Custom Server API Reference, Toolkit Configuration Schema Help, and Unified Functional Testing Test Object Schema (available with the WPF and Silverlight Add in Extensibility Help).

User interface elements are described below:

UI Elements	Description
Custom server class name	The name of the custom server class to create in the new project. **Note:** By default, this name is used to create the default values for **Run interface name**, **Property interface name**, and **Mapped test object class**. If you have not changed these values, modifying the **Custom server class name** modifies them as well.
Customize running operations	Indicates that you want to design or override implementation for running test object operations on the control. If you select this option, the Run interface that you specify is defined and tagged with the **RunInterfaceAttribute**. In the custom server class, the interface is implemented with a method stub for an example test object operation. The example test object operation is also added to the project's test object configuration file.
Run interface name	The name of the interface in which you want to implement support for the test object operations. Available only when **Customize running operations** is selected.
Customize recording	Indicates that you want to design support for recording test object operations on the control. If you select this option, the custom server class is defined to implement the **IRecord** interface and includes method stubs for the interface's methods.
Customize property retrieval	Indicates that you want to design or override implementation for retrieving identification property values from the control. If you select this option, the Property interface that you specify is defined and tagged with the **CustomPropInterfaceAttribute**. In the custom server class, the interface is implemented with a method stub for an example identification property. The example identification property is also added to the project's test object configuration file.
Property interface name	The name of the interface in which you want to implement support for property value retrieval. Available only when **Customize property retrieval** is selected.
Customize child object handling	Indicates that you want UFT to treat some of the custom control's child objects as an integral part of the control and not as independent objects. If you select this option, a preliminary implementation of the **IsKnownPartOf** method in the **IComponentDetector** interface is created in the custom server class. Implement the method to return true for the relevant child objects.
Design support for table checkpoints	Indicates that you want the standard checkpoints and output values that UFT creates on your test object to be table checkpoints and output values. If you select this option, a preliminary implementation of the **ItableVerify** interface is created in the custom server class. Implement this interface according to your needs.
Toolkit name	A name for the environment, or set of controls, that you want to support with this toolkit support set.

UI Elements	Description
	You may want the name to indicate whether this is a WPF or Silverlight toolkit. After you deploy your support, UFT displays this name in the Add-in Manager as a child node under the WPF add-in node, enabling the user to specify whether to load support for this environment. In the new project created, the toolkit name is used to create the configuration file names, and entered in the **PackageName** attribute of the **TypeInformation** element in the test object configuration file (if you select **Auto generate the XML configuration files**). It is also used for the root namespace of the project.
Auto generate the XML configuration files	Indicates that in the new project created, a toolkit configuration file and test object configuration file should also be created. The files are created with basic definitions, according to the details you specify in this dialog box. In a toolkit support set, the definitions for all of the controls that you want to support are included in one toolkit configuration file, and one test object configuration file. On the other hand, you design a separate custom server for each control you support. Therefore, if you use the template to create additional custom servers, you might not want to generate additional XML files. Alternatively, you can decide to generate the XML configuration files as well, and then copy the information from these new files to your main configuration files.
Custom control type	The name of the custom control type for which you want to develop support. **For Silverlight:** A full type name, including namespaces. **For WPF:** This can be one of the following: • A full type name, including namespaces. For example: `Infragistics.Controls.Editors.XamComboEditor` • A full type name and an assembly name. For example: `Infragistics.Controls.Editors.XamComboEditor, InfragisticsWPF4.Controls.Editors.XamComboEditor.v12.1` • An assembly qualified name. For example: `Infragistics.Controls.Editors.XamComboEditor, InfragisticsWPF4.Controls.Editors.XamComboEditor .v12.1, Version=12.1.20121.1010, Culture=neutral, PublicKeyToken=7dd5c3163f2cd0cb` **Note:** In most cases, the full type name is sufficient. You can use the more complex name when the same control is included in more than one assembly, for example, when your application includes different versions of the same control.
Mapped test object class	The test object class that you want UFT to use to represent the custom control. In the new project create, a basic definition for this new test object class is created in the test object configuration file (if you select **Auto generate the XML configuration files**).
Base class name	The test object class that your new test object classes extends. Select from the list of built in UFT test object classes, or enter a name of a test object class that you defined in a WPF or Silverlight Add-in Extensibility test object configuration file. (WPF test object classes need to extend other WPF test object classes, and Silverlight test object classes need to extend other Silverlight test object classes.) **Default:** WpfObject or SlvObject

Troubleshooting and Limitations - Developing Support

This section describes troubleshooting and limitations for developing support for custom WPF or Silverlight controls.

- If you define test object operations and the corresponding custom server methods with OUT or IN/OUT parameters, the test object operation does not run correctly.

  Workaround: When designing test object methods to support using WPF or Silverlight Add-in Extensibility custom servers, use only IN parameters. Instead of an OUT or IN/OUT (**ref**) parameter, design the operation to return a value.

- UFT does not handle controls as children of your custom control if they are implemented as pop-up controls. This means that you cannot implement the **IsKnownPartOf** method to instruct UFT to ignore these controls and treat them as integral parts of the custom control.

  UFT will recognize, spy, and learn these controls as independent controls.

  You may still be able to implement recording steps on the custom control in response to events that occur on these controls, by registering event handlers to listen to events on these controls.

- **Relevant for Silverlight only:** When the implementation of a test object method in the custom server includes an operation which blocks until the next step in the test runs, the test run session will not continue to the next step.

  For example: Suppose that clicking a custom button in the application opens a modal dialog box, and your test includes the following steps:

  ```
  MySlvButton.Click
  SlvDialog.Close
  ```

  If the implementation of the **Click** method in the custom server that supports **MySlvButton** calls `(UtilityObject.ApplicationObject as UIElement).Click` and that **Click** method does not return until the modal dialog box is closed, then the test will run the first step and never continue to the second.

  Workaround: Do one of the following:

  - Invoke the blocking statement asynchronously using **BeginInvoke**.
  - Use mouse or keyboard operations to implement the test object method (for example, click the button using `UtilityObject.MouseClick`).

- **Relevant for Silverlight only:** In some cases, when you design your support to receive Windows messages generated for controls other than the custom control you are supporting, some such Windows messages are still not passed to the custom server.

  The reason for this is that 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 generated for 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.

Chapter 2: Tutorial: Create UFT Support for a Custom WPF Control

In this tutorial you manually create support for a WPF Calendar control, learning the basics of creating a WPF Add-in Extensibility toolkit support set. A toolkit, or an environment, is a set of controls for which you want to provide support in one package. In this tutorial, the toolkit is named **MyWpfToolkit**, and contains only the **Microsoft.Windows.Controls.Calendar** control.

To perform this tutorial you must have a supported version of Microsoft Visual Studio installed, in addition to the WPF and Silverlight Add-in Extensibility SDK (which must be installed after Visual Studio).

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).

This tutorial uses the **UFT WPF CustomServer** project template in Visual Studio to set up the files necessary to create the toolkit support set. When you develop your own support, if you want to create your toolkit support set files manually, follow the steps in "How to Create Support for a Custom WPF or Silverlight Toolkit" on page 17.

For details on the classes and interface methods mentioned throughout this tutorial, see the Custom Server API References (available with the WPF and Silverlight Add-in Extensibility Help).

Note: You develop support for a Silverlight control in much the same way as you develop a support for a WPF control. Throughout the tutorial, where modifications would be necessary if this were a Silverlight toolkit support set, the modifications are explained.

The WPF Calendar application is installed in: **<WPF and Silverlight Add-in Extensibility SDK installation folder>\samples\WPFExtCalendarSample\Application**.

The **<WPF and Silverlight Add-in Extensibility SDK installation folder>\samples\WPFExtCalendarSample\Support** folder contains the Microsoft Visual Studio solution and XML files that make up support for this control, similar to the support you create in this tutorial. You can refer to these files while you perform the tutorial.

This tutorial includes:

- Planning Support for the WPF Calendar Control 34
- Setting Up the WPF Add-in Extensibility Project for the WPF Calendar Control 36
- Designing the Toolkit Configuration File 39
- Designing the Test Object Configuration File 40
- Deploying and Testing the Preliminary Toolkit Support Set 42

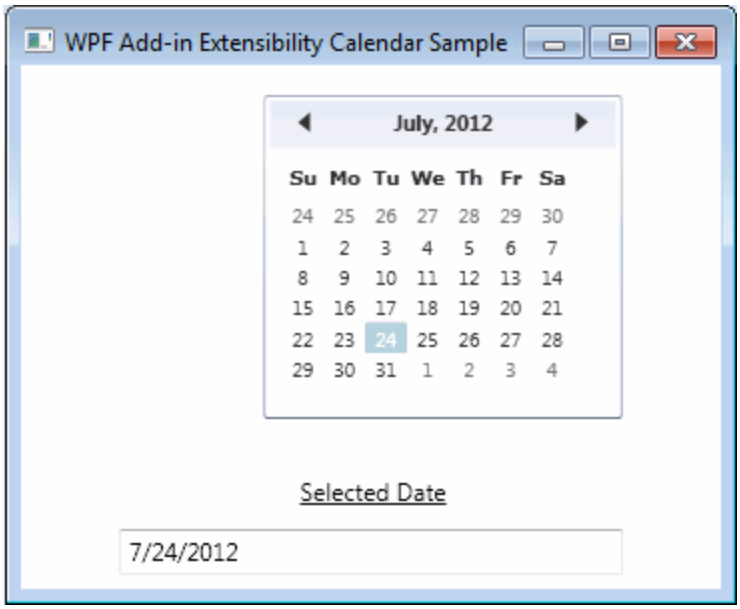
- Design the Basic Custom Server 45
- Implement Support for Retrieving Identification Property Values 46
- Deploy and Test Your Basic Custom Server and Identification Property Support 47
- Implement Support for Running Test Object Operations 48
- Deploy and Test Your Support for Test Object Operations 50
- Implement Support for Recording 51
- Deploy and Test Your Support for Recording 53

Planning Support for the WPF Calendar Control

In this section, you study the behavior of the control that you want to support and the way UFT recognizes it and interacts with it. You then determine what you need to customize in UFT's behavior in order to enable creating test steps that are more meaningful and easier to maintain.

1. **Run the sample WPF Calendar application and study its behavior**

   a. Double-click the **<WPF and Silverlight Add-in Extensibility SDK installation folder>\samples\WPFExtCalendarSample\Application\WpfCalendar.exe** file. The Calendar application opens.

   This application contains 3 controls:

   - A calendar display area with buttons: **Microsoft.Windows.Controls.Calendar**
   - A text label: **System.Windows.Controls.TextBlock**
   - An edit box displaying the selected date: **System.Windows.Controls.TextBox**

   b. Study the application's functionality.

   - You can click the right and left arrows to go the next or previous month.
   - You can select a date in the calendar by clicking on the relevant day of the month.
   - You can view the selected date in the text box.

2. **Use the Object Spy in UFT to see how UFT recognizes the controls in the Calendar application**

   a. Open UFT, and open a GUI test.

b. Select **Record > Record and Run Settings**, and make sure that the selections in the Windows Applications tab enable UFT to record and run tests on the calendar application.

c. Run the Object Spy  and spy on the WPF Calendar.

UFT recognizes the Calendar application as a WpfWindow. Within this window, it recognizes the **Microsoft.Windows.Controls.Calendar** control as a generic WpfObject and the **System.Windows.Controls.TextBox** as a WpfEdit object.

Additionally, within the in the **Microsoft.Windows.Controls.Calendar** control, UFT recognizes the days as independent WpfButtons in the WpfWindow.

UFT ignores other user interface elements contained in the Calendar control, such as the right and left arrows, and the month and year banner.

3. Learn the Calendar control using UFT, adding it to an object repository

a. Open the Object Repository.

b. Click the **Add Objects to Local**  button.

c. Click on an area in the calendar.

d. Add additional objects to the local object repository by clicking the **Add Objects to Local** button and then clicking on different areas of the calendar. For example, click on the days, the text box, the month and year banner, and so on.

UFT learns the application as a WpfWindow. Within this window, it separately learns the calendar display area as a generic WpfObject and the Selected Date box as a WpfEdit object.

The days are learned as independent WpfButtons, but other user interface elements, such as the right and left arrows, and the month and year banner, are not learned at all.

4. Record a test on the Calendar control

a. Click **Record**  .

b. Click on different areas in the calendar.

When you click on the right or left arrows, or on the month and year banner, UFT records a generic click on the WpfObject, specifying the coordinates of the location you clicked.

When you click a day and select it, or when you click in the Selected Date box, nothing is recorded.

When you click in other areas of the application, UFT records a generic click on the WpfWindow.

5. Conclusion: Develop a MyWpfCalendar Test Object Class

- For functional testing purposes, the **Microsoft.Windows.Controls.Calendar** control should be represented in UFT by one MyWpfCalendar test object.
- The buttons within the Calendar control should not be treated as separate controls.
- The Selected Date box is a read-only box, on which no user activity is possible and no steps are

recorded. Therefore, the **System.Windows.Controls.TextBox** control does not have to be supported as part of the MyWpfCalendar test object, and does not need any customization for functional testing.

- The MyWpfCalendar test object class should be based on the existing UFT test object class, WpfObject, and extend its capabilities.
- The MyWpfCalendar test object class should support calendar-related operations.

  In this tutorial you will develop support for the following test object methods: **SetDate** (default operation), **Next**, **Previous**, and a **SelectedDate** test object property.
- User operations performed on the calendar's different user interface elements, should be interpreted and recorded as high-level operations on the calendar as a whole. For example: **SetDate**, **Next**, **Previous**, and so on.
- The MyWpfCalendar test object class should support identification properties relevant for a calendar, such as is_today_highlighted.

Setting Up the WPF Add-in Extensibility Project for the WPF Calendar Control

A WPF Add-in Extensibility support set consists of the following mandatory files:

- **A Test Object Configuration XML File.** In this file, new test object types are defined. For details, see "The Test Object Configuration XML File" on page 11.
- **A toolkit configuration file.** In this file WPF control types are mapped to test object types and to the custom servers implementing their record and run logic. For details of the schema, see the *Toolkit Configuration Schema Help* (available with the WPF and Silverlight Add-in Extensibility Help).
- **.Net DLLs containing the implementation of custom servers.** For details, see "Custom Servers " on page 15.

The WPF and Silverlight Add-in Extensibility SDK installs a project template and a setup dialog box in Microsoft Visual Studio that assist you in setting up the files that you need to create the toolkit support set.

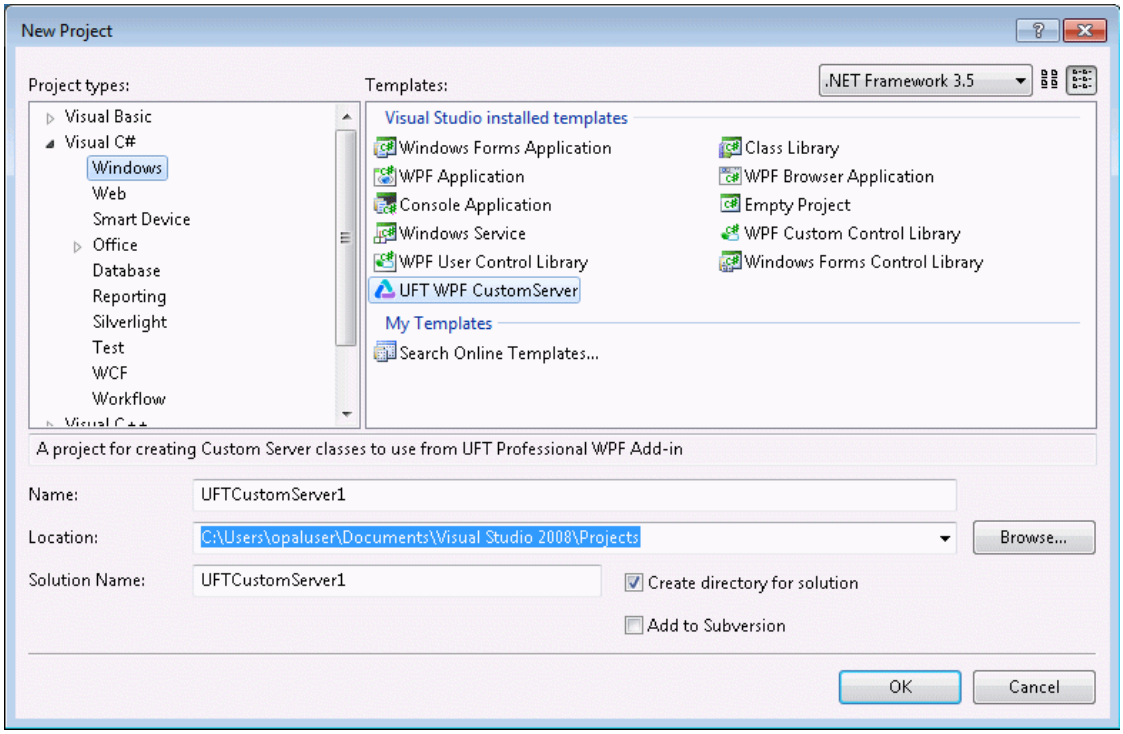
Use the template for each control you want to support. The template sets up both the XML files and the Microsoft Visual Studio solution that you need to create the custom server DLL. When you create support for more than one control in a toolkit, you have to combine the XML content created for each control into one toolkit configuration file and one test object configuration file for the toolkit.

In this tutorial, because you are creating a toolkit support for only one control, you can use the XML files created by the project template, as-is.

Note: The Microsoft Visual Studio dialog box images in this section are taken from Microsoft Visual Studio 2008. If you use a different version, the dialog boxes may differ slightly in appearance.

Create a WPF Add-in Extensibility Project in Microsoft Visual Studio

1. Open Microsoft Visual Studio and click **New Project**. The New Project dialog box opens:



2. Select the **Visual C# Windows** project type and the **UFT WPF CustomServer** template, and click **OK**.

   Note: If you were developing support for a Silverlight control, you would select the **Visual C# Silverlight** project type and the **UFT Silverlight CustomServer** template.

3. Enter `CalendarSrv` as the project **Name**, and click **OK**. The WPF/Silverlight Custom Server Setup dialog box opens.

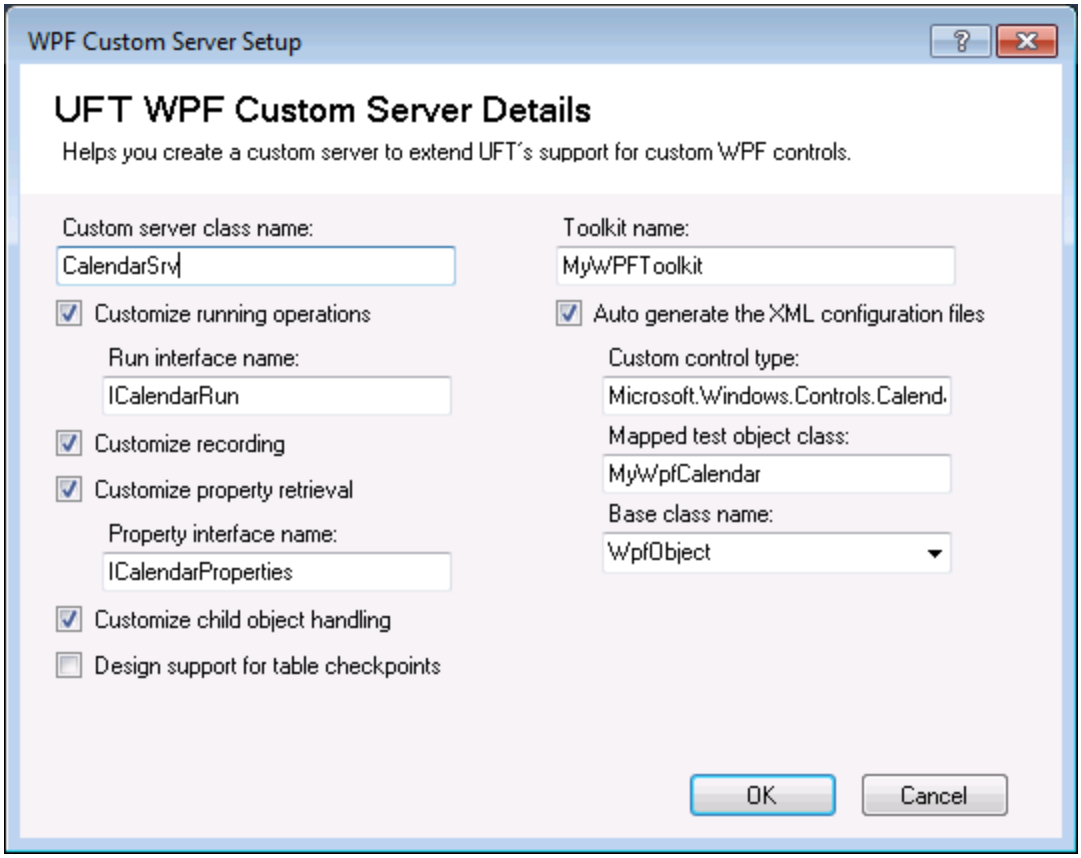
4. In this dialog box, you provide specifications that describe the support that you want to create, and the files required to create this support are created accordingly.

   In this tutorial:

   - You create support for the **MyWpfToolkit** toolkit.
   - Within this toolkit you create support for **Microsoft.Windows.Controls.Calendar** WPF controls.
   - You create a **MyWpfCalendar** test object class, based on the standard UFT**WpfObject** class, to represent the Calendar controls in UFT.
   - You create a **CalendarSrv** custom server class, to provide support for the controls. Within the **CalendarSrv** custom server class, you customize running operations, retrieving properties, recording, and child object handling.

     Specify these details in the WPF Custom Server Setup dialog box, as shown in the image below, selecting also the options for automatically generating XML files, comments, and sample code.

Make sure to enter the **Run interface name** and the **Property interface name** shown below, as this tutorial does not use the default names provided in the dialog box.



If you want more information on this dialog box, see "WPF/Silverlight Custom Server Setup Dialog Box (in Microsoft Visual Studio)" on page 26.

5. Click **OK**.

The CalendarSrv solution is created with the relevant files and references. The solution includes a toolkit configuration file (**MyWpfToolkit.cfg**), a test object configuration file (**MyWpfToolkitTestObjects.xml**), and the C# file for the custom server class (**CalendarSrv.cs**).

It also includes the reference to the **Mercury.QTP.WpfAgent.dll** file, that contains the WPF Add-in Extensibility API.

Note: If you use the **UFT Silverlight CustomServer** template, the solution created includes a reference to the **Mercury.QTP.Slv.CustomServer.dll** file, that contains the Silverlight Add-in Extensibility API.

Designing the Toolkit Configuration File

The name of the toolkit configuration file informs UFT of the new supported environment. After you deploy this file to the correct location on a UFT computer, when UFT opens, it displays the environment in the Add-in Manager, as a child node beneath the WPF Add-in. If you select the check box for this environment, UFT loads the support that you provide for it. (If you are working with Silverlight, you must also select the Silverlight Add-in.)

The configuration file content defines how the controls are supported, which test object classes and custom servers are used for each custom control and so on.

Open the **MyWpfToolkit.cfg** file to see its content.

The **MyWpfToolkit.cfg** file was created automatically based on the specifications you provided. Therefore, it already has all of the necessary content to support the WPF Calendar control is the MyWpfToolkit environment.

```
<?xml version="1.0" encoding="UTF-8"?>
<Controls>
  <Control Type="Microsoft.Windows.Controls.Calendar"
          MappedTo="MyWpfCalendar“>
    <CustomServer>
        <Component>
            <DllName>CalendarSrv.dll</DllName>
            <TypeName>MyWpfToolkit.CalendarSrv</TypeName>
        </Component>
    </CustomServer>
  </Control>
</Controls>
```

The **Control** element's attributes specify that the controls of **TypeMicrosoft.Windows.Controls.Calendar** (the full control type name including namespaces must be specified) is **MappedTo** the test object class **MyWpfCalendar**.

The **CustomServer > Component** element specifies the custom server DLL and type that provides support for this control type.

Note: The **custom server type** must be a full type name including namespaces, and in Silverlight it must include additional information, as described in the Toolkit Configuration Schema Help.

For more information on the elements and attributes in the toolkit configuration file, see the *Toolkit Configuration Schema Help* (available with the WPF and Silverlight Add-in Extensibility Help).

Designing the Test Object Configuration File

You use the test object configuration file to introduce the MyWpfToolkit environment and its test object class to UFT.

1. Open the **MyWpfToolkitTestObjects.xml** file.

   The **MyWpfToolkitTestObjects.xml** file was created with the **AddinName** attribute in the **TypeInformation** element set to `WPF` and the **PackageName** attribute set to `MyWpfToolkit`. This associates the test object configuration file (and the test objects defined in it) with the MyWpfToolkit environment under the WPF Add-in. If, when UFT opens, you do not select the MyWpfToolkit environment, UFT ignores the test object class definitions in this file.

   Note: When developing support for a Silverlight control, the **TypeInformation** element is set to `Silverlight`.

   Based on the information you provided in the WPF Custom Server Setup dialog box, the **ClassInfo** element for the `MyWpfCalendar` test object class was also created, specifying `WpfObject` as its base class. This means that the new `MyWpfCalendar` test object class you define inherits the `WpfObject` methods, generic type, Help file, etc.

2. To extend the test object class and add definitions for the calendar-specific operations and identification properties, replace the comment lines within **MyWpfToolkitTestObjects.xml** so that your test object configuration file contains the following:

```
<TypeInformation AddinName="WPF" PackageName="MyWpfToolkit“ >
    <ClassInfo Name="MyWpfCalendar"
               BaseClassInfoName="WpfObject"
               DefaultOperationName="SetDate“>
        <IdentificationProperties>
            <IdentificationProperty Name="devname"
                                    ForDescription="true" />
            <IdentificationProperty Name="devnamepath"
                                    ForAssistive="true"
                                    AssistivePropertyValue="1"/>
            <IdentificationProperty Name="regexpwndtitle"
                                    ForAssistive="true"
                                    AssistivePropertyValue="2"/>
            <IdentificationProperty Name="x"
                                    ForVerification="true" />
            <IdentificationProperty Name="y"
                                    ForVerification="true" />
            <IdentificationProperty Name="is_today_highlighted"
                                    ForVerification="true"/>
        </IdentificationProperties>
```

```
        <TypeInfo>
            <Operation Name="Next" PropertyType="Method"/>
            <Operation Name="Prev" PropertyType="Method"/>
            <Operation Name="SelectedDate"
                            PropertyType="Property_Get“ >
                <ReturnValueType>
                    <Type VariantType="VT_BSTR"/>
                </ReturnValueType>
            </Operation>
            <Operation Name="SetDate" PropertyType="Method“ >
                <Argument Name="Date"
                          IsMandatory="true"
                          Direction="In“>
                    <Type VariantType="VT_BSTR"/>
                </Argument>
            </Operation>
        </TypeInfo>
    </ClassInfo>
</TypeInformation>
```

You have now defined:

- The **Previous**, **Next**, and **SetDate** test object methods and the **SelectedDate** property, including all relevant parameters, return values, and their types. **SetDate** is the default operation for this test object class.
- The **devname**, **devnamepath**, **regexpwndtitle**, **x**, **y**, and **is_today_highlighted** identification properties.
    - The first 5 properties are supported by the base class, and the implementation for retrieving their values is inherited. However, identification property **definitions** are not automatically inherited, which is why you must define them here.
    - For each identification property, you specified whether it should be included in the test object description, used as an assistive property, or available for verification in checkpoints.

For more information on the elements and attributes in the test object configuration file, see the *HP UFT Test Object Schema Help* (available with the WPF and Silverlight Add-in Extensibility Help).

Deploying and Testing the Preliminary Toolkit Support Set

After defining the **MyWpfCalendar** test object class in the test object configuration file and mapping the Calendar control to this test object class in the toolkit configuration file, you can already test the effect of using the toolkit support set with UFT.

Note: When you develop your own toolkit support set, if you modify attributes of **Identification Property** elements in the test object configuration file, keep the **DevelopmentMode** attribute of the **TypeInformation** element set to `true` during the design stages of the custom toolkit support. Before you deploy the custom toolkit support set for regular use, be sure to remove this attribute (or set it to `false`). This is not required when performing this tutorial lesson. For more information, see Modifying Identification Property Attributes in a Test Object Configuration FileShared File: ModifyTO_IDProp_att.

To deploy the toolkit support set:

1. Copy the **MyWpfToolkitTestObjects.xml** file to **<UFT installation folder>\dat\Extensibility\WPF**.
2. In the **<UFT installation folder>\dat\Extensibility\WPF** folder, create a folder named **MyWpfToolkit**.
3. Copy the **MyWpfToolkit.cfg** file to the **<UFT installation folder>\dat\Extensibility\WPF\MyWpfToolkit** folder.

   Note: If you were developing support for a Silverlight control, you would replace `WPF` in the paths above with `Slv`.

To test the toolkit support set:

1. After you deploy the toolkit support set, open UFT and open a GUI test.

   Note: UFT reads toolkit support files when it opens. Therefore, if UFT is open, you must close UFT and open it again.

   The Add-in Manager dialog box displays **MyWpfToolkit** as a child of the WPF environment in the list of available add-ins. (If the Add-in Manager dialog box does not open, see the *HP Unified Functional Testing Add-ins Guide* for instructions.)
2. Select the check box for **MyWpfToolkit** and click **OK**. UFT opens and loads the support you designed.
3. Use the **Define New Test Object**  button in the Object Repository dialog box to open the Define New Test Object dialog box. The MyWpfToolkit environment is displayed in the **Environment**

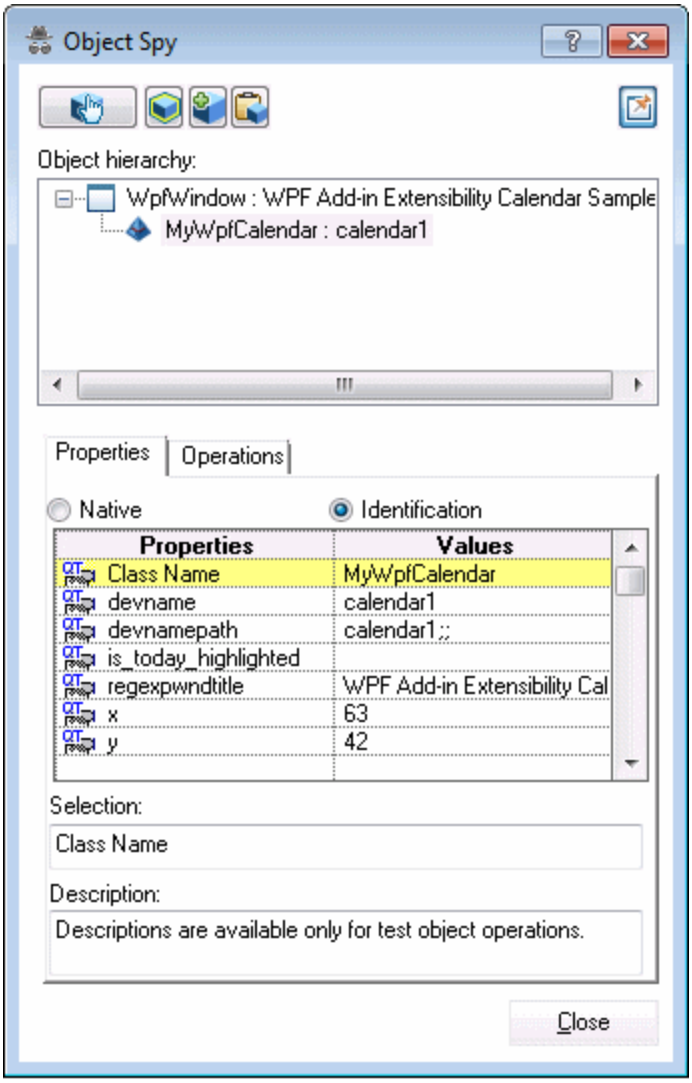
list. When you select the MyWpfToolkit environment from the list, the **MyWpfCalendar** test object class that you defined in the test object configuration file is displayed in the **Class** list.

4. Select **Tools > Object Identification**. In the Object Identification dialog box, when you select the MyWpfToolkit environment in the **Environment** list, the identification property definitions for the **MyWpfCalendar** test object class should match the definitions in the test object configuration file.
5. Run the sample control by opening the **<WPF and Silverlight Add-in Extensibility SDK installation folder>\samples\WPFExtCalendarSample\Application\WpfCalendar.exe** file.

   Note: UFT establishes its connection with an application when the application opens. Therefore, if the Calendar application is open, you must close it and run it again.

6. In UFT, perform the following activities on the Calendar control, to see how UFT recognizes the control. (For more information on working in UFT, see the *HP Unified Functional Testing User Guide*.)
   - Use the Object Spy  to see how UFT recognizes the Calendar control and to view its identification properties and test object operations:
     - The calendar is represented by a **MyWpfCalendar** test object class.
     - The calendar day numbers are still recognized as separate test objects. Later in this tutorial you will customize child object handling to prevent that.
     - The list of test object operations includes all of the operations (methods and properties) inherited from the **WpfObject** base class, as well as all of the operations that you defined in the **MyWpfToolkitTestObjects.xml** test object configuration file.

- The list of identification properties includes all of the properties that you defined in the **MyWpfToolkitTestObjects.xml** test object configuration file.



- The **is_today_highlighted** identification property has no value, because you have not yet implemented its retrieval. For all other identification properties, the value is provided as it would be for a **WpfObject** (because it is the base class).

- In the Editor, type `MyWpfCalendar("MyCalendar").`

  When you type the period, UFT's statement completion feature displays all of the operations available for the **MyWpfCalendar** test object class. This includes operations inherited from **WpfObject**, and ones that you defined in the test object configuration file.

Design the Basic Custom Server

For each custom control that you want to support, you develop a custom server class, that derives from the **CustomServerBase** class. The resulting custom server DLL runs in the context of the application and interfaces between UFT and the custom control.

In this section, you design the CalendarSrv custom server class to support the Calendar control.

1. Open **CalendarSrv.cs**. The basic framework of the class was created based on the specifications that you provided in the WPF Custom Server Setup dialog box.
   - The class inherits from **CustomServerBase**.
   - In the `Using` section, the class includes a reference to the **Mercury.QTP.WPF.CustomServer** namespace in the WPF Add-in Extensibility API.

     Note: If you use the **UFT Silverlight CustomServer** template, the class includes a reference to the **Mercury.QTP.Slv.CustomServer** namespace in the Silverlight Add-in Extensibility API.

   - The class definition includes the list of interfaces it will implement: **IRecord**, **ICalendarRun**, **IComponentDetector**, **ICalendarProperties**.
2. In your project, add a reference to **<WPF and Silverlight Add-in Extensibility SDK installation folder>\samples\WPFExtCalendarSample\Application\WPFToolkit.dll**.

   This enables you to access the methods, properties, and events of **Microsoft.Windows.Controls.Calendar**, by double-clicking the reference node in the Visual Studio Solution Explorer. You need to be familiar with these so that you can design code that interacts with the Calendar control.
3. In **Calendar.cs**, add a `using Microsoft.Windows.Controls;` statement. This enables Microsoft IntelliSense for the **Microsoft.Windows.Controls.Calendar** control type in Visual Studio.
4. In the CalendarSrv class, implement a common helper property that returns a reference to the custom Calendar object. You can use this throughout your custom server code to access the custom control's events, methods, and properties:

```
private Calendar MyCalendar
{
    get
    {
        return UtilityObject.ApplicationObject as Calendar;
    }
}
```

5. You specified, in the WPF Custom Server Setup dialog box, that you want to customize child object handling. Therefore, a preliminary implementation of the **IsKnownPartOf** method in the

IComponentDetector interface was created in the CalendarSrv class.

Modify the **IsKnownPartOf** method to always return true. This means that UFT will treat all child objects within the Calendar as part of the calendar and not as independent objects.

Implement Support for Retrieving Identification Property Values

In this section, you implement the property value retrieval interface in the CalendarSrv class to support retrieving the values of identification properties from the Calendar control.

You specified, in the WPF Custom Server Setup dialog box, that you want to customize property retrieval. Therefore, the **ICalendarProperties** interface that you specified was defined in the **CalendarSrv.cs** file, tagged with the **CustomPropInterface** attribute, and implemented in the **CalendarSrv** class for an example property, **MyCustomProperty**.

1. Locate the **ICalendarProperties** interface definition in the **CalendarSrv.cs** file.

```
[CustomPropInterface()]
    public interface ICalendarProperties
    {
        object MyCustomProperty
        {
            get;
        }
    }
```

2. Replace the example `object MyCustomProperty` with `bool is_today_highlighted` to complete the interface definition.
3. Locate the interface implementation in the CalendarSrv class:

```
public object MyCustomProperty
{
    get
    {
        return null;
    }
}
```

4. Modify the example implementation to retrieve the value for the **is_today_highlighted** identification property:

```
public bool is_today_highlighted
{
```

```
    get
    {
        return MyCalendar.IsTodayHighlighted;
    }
}
```

Deploy and Test Your Basic Custom Server and Identification Property Support

In this section, you deploy the custom server that you developed to support the Calendar control and test its effect on UFT.

1. Build your solution and then deploy the custom server by copying the **CalendarSrv.dll** file to the **<UFT installation folder>\dat\Extensibility\WPF\MyWpfToolkit** folder. You do not need to deploy the XML files because you did not change them.
2. Run the sample control by opening the **<WPF and Silverlight Add-in Extensibility SDK installation folder>\samples\WPFExtCalendarSample\Application\WpfCalendar.exe** file.

   Note: You can use an open instance of UFT because you did not modify configuration files. However, if the Calendar application is open, you must close it and run it again.

3. Use the Object Spy  to see how UFT recognizes the Calendar control and its children, and to view its identification properties:
   - The calendar is represented by a **MyWpfCalendar** test object class.
   - The day numbers within the calendar are considered part of the Calendar control and are not represented by separate WpfButton test objects.
   - The value of the **is_today_highlighted** property is displayed.
   - The Selected Date box remains external to the Calendar control, and is still represented by a separate WpfEdit test object, as planned.
4. Use the **Add Objects to Local**  button in the Object Repository dialog box to learn the Calendar control. A **MyWpfCalendar** test object named **calendar1** is added to the object repository.
5. Start a recording session and create a checkpoint that checks the value of the **is_today_highlighted** property of the **calendar1** test object. Stop the recording session and run the step to verify that the property value is properly retrieved.
6. In the Keyword View, create a test step with the **calendar1** test object. The default **SetDate** operation is selected automatically. Enter a date in the **Argument** column (in the format: `mm/dd/yyyy`).

7. Run the test. Because you have not yet implemented support for running test object methods, a run-time error occurs. In the next section, you implement this support.

Implement Support for Running Test Object Operations

In this section, you implement the Run interface in the CalendarSrv class to support running test object operations on the Calendar control.

You specified, in the WPF Custom Server Setup dialog box, that you want to customize running operations. Therefore, the **ICalendarRun** interface that you specified was defined in the **CalendarSrv.cs** file, tagged with the **RunInterface** attribute and implemented in the **CalendarSrv** class for an example operation, **MyRunMethod**.

1. Locate the **ICalendarRun** interface definition in the **CalendarSrv.cs** file:

```
[RunInterface()]
public interface ICalendarRun
{
    void MyRunMethod();
}
```

2. Replace the example `void MyRunMethod();` with the following lines to complete the interface definition to include all of the operations you want to support:

```
void SetDate(string date);
void Prev();
void Next();
string SelectedDate
{
    get;
}
```

3. Locate the interface implementation in the CalendarSrv class:

```
public void MyRunMethod()
{
}
```

4. Replace the **MyRunMethod()** example with the following implementation of the Calendar-specific methods and property:

```
 public void SetDate(String date)
{
    MyCalendar.SelectedDate = DateTime.Parse(date);
    MyCalendar.DisplayDate = DateTime.Parse(date);
}
```

```
public string SelectedDate
{
   get
   {
     return MyCalendar.SelectedDate.Value.ToShortDateString();
   }
}
```

```
public void Prev()
{
    Button prev = GetDescendantByName(UtilityObject.ApplicationObject,
"PART_PreviousButton") as Button;
    RaiseButtonClickEvent(prev);
}
```

```
public void Next()
{
    Button next = GetDescendantByName(UtilityObject.ApplicationObject,
"PART_NextButton") as Button;
    RaiseButtonClickEvent(next);
}
```

Note: If you were developing support for a Silverlight control, you would tag each one of these methods with the Microsoft Silverlight **ScriptableMember** attribute.

5. Add a `using System.Windows.Media;` statement to the CalendarSrv.cs file and then add the following helper functions:

```
private void RaiseButtonClickEvent(Button button)
{
    if (button != null)
    {
        RoutedEvent e = Button.ClickEvent;
        RoutedEventArgs arg = new RoutedEventArgs();
        arg.RoutedEvent = e;
```

```
        button.RaiseEvent(arg);
    }
}
```

```
private DependencyObject GetDescendantByName(DependencyObject parent, string
name)
{
    if (parent == null)
        return null;
    int count = VisualTreeHelper.GetChildrenCount(parent);
    for (int i = 0; i < count; i++)
    {
        DependencyObject child = VisualTreeHelper.GetChild(parent, i);
        if (child is FrameworkElement)
        {
            if ((child as FrameworkElement).Name == name)
            return child;
        }
        if (child is FrameworkContentElement)
        {
            if ((child as FrameworkContentElement).Name == name)
                return child;
        }
        child = GetDescendantByName(child, name);
        if (child != null)
            return child;
    }
    return null;
}
```

Deploy and Test Your Support for Test Object Operations

In this section, you deploy the custom server again and test the support that you designed for running test object operations.

1. Build your solution and then deploy the custom server by copying the **CalendarSrv.dll** file to the **<UFT installation folder>\dat\Extensibility\WPF\MyWpfToolkit** folder.
2. Run the sample control by opening the **<WPF and Silverlight Add-in Extensibility SDK installation folder>\samples\WPFExtCalendarSample\Application\WpfCalendar.exe** file.
3. Use the **Add Objects to Local**  button in the Object Repository dialog box to learn the Calendar control. A **MyWpfCalendar** test object named **calender1** is added to the object repository.

4. Create test steps with the **calendar1** test object, using each of the test object operations: **SetDate (mm/dd/yyyy)**, **SelectedDate**, **Next**, and **Prev.** (To view the value returned by the SelectedDate property you can use a **msgBox** statement.)
5. Run the test and make sure that the operations are carried out correctly.

Implement Support for Recording

In this section, you implement the **IRecord** interface and write the event and message handling methods to support recording steps on the custom control.

There are three types of steps that need to be recorded for the WPF custom calendar: **Next**, **Prev**, and **SetDate**.

- **Next** and **Prev** are recorded in response to Windows messages, when a user clicks the right and left arrows on the Calendar control. This is handled by the **OnMessage** method.
- **SetDate** is recorded in response to a control event—**SelectedDatesChanged**. This is handled by an event handler that you design and register to handle the relevant control event.

To implement support for recording in your custom server class:

1. In the CalendarSrv class, locate the section for the IRecord interface implementation.

```
public void OnMessage(DependencyObject src, int msg, int wParam, int lParam)
{
}
public void RecordInit()
{
}
public void RecordStop()
{
}
```

2. Declare your event handler and implement **RecordInit** to register it to the control.

```
private EventHandler<SelectionChangedEventArgs> _h;
public void RecordInit()
{
    _h = new EventHandler<SelectionChangedEventArgs>
        (OnSelectedDatesChanged);
    UtilityObject.AddHandler(MyCalendar, "SelectedDatesChanged",
        _h);
}
```

Note: When developing support for a Silverlight control, the **AddHandler** syntax is different. For details, see the **Mercury.QTP.Slv.CustomServer** namespace in the Custom Server API

Reference (available with the WPF and Silverlight Add in Extensibility Help).

3. You do not need to implement **RecordStop** because you registered the event handler using the SDK's **AddHandler** method. This enables UFT to automatically remove the event handler at the end of a recording session.
4. Add the **OnSelectedDatesChanged** event handler implementation:

```
private void OnSelectedDatesChanged(object sender,
System.Windows.Controls.SelectionChangedEventArgs e)
{
    UtilityObject.Record("SetDate",
   RecordingMode.RECORD_SEND_LINE,
MyCalendar.SelectedDate.Value.ToShortDateString());
}
```

   This creates a **SetDate** step with the new date selected by the user during a recording session.
5. Implement the **OnMessage** method as follows, to support recording **Prev** and **Next** operations:

```
public void OnMessage(DependencyObject o, int msg, int wParam, int lParam)
{
    if(o is Button && msg == 0x201) // WM_LBUTTONDOWN
    {
        string name = (o as Button).Name;
        switch (name)
        {
            case "PART_NextButton":
                    base.UtilityObject.Record("Next",
                    RecordingMode.RECORD_SEND_LINE, null);
            break;
            case "PART_PreviousButton":
                    base.UtilityObject.Record("Prev",
                     RecordingMode.RECORD_SEND_LINE, null);
            break;
       }
    }
}
```

   Note: When developing support for a Silverlight control, the **OnMessage** method would return **RECORD_HANDLED**, indicating that the custom server handled this message and it does not have to be passed on to any other event handlers. For more information, see the **Mercury.QTP.Slv.CustomServer** namespace in the Custom Server API Reference (available with the WPF and Silverlight Add in Extensibility Help).
6. **This step is necessary only when developing support for Silverlight controls.**

Implement the **GetWndMessageFilter** method to specify the level of Windows messages to be handled by the custom server.

```
CTL_MsgFilter GetWndMessageFilter()
{
    return CTL_MsgFilter.CTL_MSGS;
}
```

In this tutorial, all children of the control are regarded as part of the control. Therefore, it is sufficient to return `CTL_MSGS`, and handle messages intended only for this control.

If some of the control's children were treated as separate test objects, but you still wanted the control to handle events that occurred on these children, you could implement **GetWndMessageFilter** to return `CHILD_MSGS`.

Deploy and Test Your Support for Recording

You have now completed the design of the support for the WPF Calendar control.

In this section, you deploy the custom server again and test the support that you designed for recording operations on the Calendar control.

1. Build your solution and then deploy the custom server by copying the **CalendarSrv.dll** file to the **<UFT installation folder>\dat\Extensibility\WPF\MyWpfToolkit** folder.
2. Run the sample control by opening the **<WPF and Silverlight Add-in Extensibility SDK installation folder>\samples\WPFExtCalendarSample\Application\WpfCalendar.exe** file.
3. To test that the support you developed for recording is working correctly, start a recording session, select a day in the calendar, click the right arrow at the top of the calendar, and click the left arrow at the top of the calendar. **SetDate**, **Next**, and **Prev** steps should be recorded.

Chapter 3: Deploying the Toolkit Support Set

The final stage of extending UFT support for a custom toolkit is deploying the toolkit support set. This means placing all of the files you created in the correct locations on a computer with UFT installed, enabling UFT to recognize the controls in the toolkit and run tests on them.

While you are developing the toolkit support set, deploying it to UFT enables you to test and debug the support that you create. After the toolkit support set is complete, you can deploy it on any computer with UFT installed, to extend the WPF or Silverlight Add-in.

This chapter includes:

- About Deploying the Custom Toolkit Support 55
- Deploying the Custom Toolkit Support 55
- Modifying Deployed Support 57
- Removing Deployed Support 58

About Deploying the Custom Toolkit Support

From the UFT user's perspective, after you deploy the toolkit support set on a computer on which UFT is installed, the toolkit support set can be used as a UFT add-in.

When UFT opens, it displays the toolkit support set's environment name in the Add-in Manager, as a child node under the WPF Add-in node. Select the check box for your environment to instruct UFT to load support for the environment using the toolkit support set that you developed.

Note: Toolkit support sets that you develop using Silverlight Add-in Extensibility are dependent on the Silverlight Add-in. Therefore, if you select the environment of such a toolkit support set, select the Silverlight Add-in as well.

If support for your environment is loaded:

- UFT recognizes the controls in your environment and can run tests on them.
- UFT displays the name of your environment in all of the dialog boxes that display lists of add-ins or supported environments.
- UFT displays the list of test object classes defined by your toolkit support set in dialog boxes that display the list of test object classes available for each add-in. (For example: Define New Test Object dialog box, Object Identification dialog box.)

Deploying the Custom Toolkit Support

To deploy the toolkit support set that you create, you must place the files in specific locations within the UFT installation folder.

Note: Before you begin, create a folder with the name of your custom toolkit in the **<UFT Installation folder>\dat\Extensibility\WPF** (or **...\Slv**) folder, if one does not already exist.

The following table describes the appropriate location for each of the toolkit support files:

File Name	Location
<Custom Toolkit Name>TestObjects.xml **Note:** This is the recommended file name convention. You can have more than one test object configuration XML file, and name them as you wish.	**When deploying support for WPF:** • `<UFT Installation folder>\dat\ Extensibility\ WPF` • `<UFT Add-in for ALM Installation folder>\dat\Extensibility\ WPF` (Optional. Required only if the folder exists, which means the UFT Add-in for ALM was installed independently from the ALM Add-ins page and not as part of the UFT installation.)

File Name	Location
	When deploying support for Silverlight: • `<UFT Installation folder>\dat\Extensibility\Slv` • `<UFT Add-in for ALM Installation folder>\dat\Extensibility\Slv` (Optional. Required only if the folder exists, which means the UFT Add-in for ALM was installed independently from the ALM Add-ins page and not as part of the UFT installation.)

<Custom Toolkit Name>.cfg	**When deploying support for WPF:**
	`<UFT Installation folder>\dat\Extensibility\WPF\<custom toolkit name>`
	When deploying support for Silverlight:
	`<UFT Installation folder>\dat\Extensibility\Slv\<custom toolkit name>`

Custom Server DLL	The **.dll** file can be located on the computer on which UFT is installed, or in an accessible network location. Specify the location in the DllName element in **<Custom Toolkit Name>.cfg**
Icon files for new test object classes (optional)	The file can be a **.dll**, **.exe**, or **.ico** file, located on the computer on which UFT is installed, or in an accessible network location. Specify the location in `<Custom Toolkit Name>TestObjects.xml`
Help files for the test object classes (optional)	Must be a **.chm** file, located on the computer on which UFT is installed. Specify the location in `<Custom Toolkit Name>TestObjects.xml`

Recommended File Locations

You specify the locations of the custom server DLL, Help, and icon files in the toolkit support set's configuration files. You can specify these locations using relative paths. For more information, see the *Test Object Schema Help* and the *Toolkit Configuration Schema Help* (available with the WPF and Silverlight Add-in Extensibility Help).

The recommended locations for these files are described in the following table:

File Name	Location
Custom Server DLL files	**When deploying support for WPF:** `<UFT Installation folder>\dat\Extensibility\WPF\<custom toolkit name>\CustomServers`

File Name	Location
	When deploying support for Silverlight: `<UFT Installation folder>\dat\Extensibility\Slv\<custom toolkit name>\CustomServers`

Icon files	**When deploying support for WPF:**
	`<UFT Installation folder>\dat\Extensibility\WPF\<custom toolkit name>\Res`
	When deploying support for Silverlight:
	`<UFT Installation folder>\dat\Extensibility\Slv\<custom toolkit name>\Res`

Help files	**When deploying support for WPF:**
	`<UFT Installation folder>\dat\Extensibility\WPF\<custom toolkit name>\Help`
	When deploying support for Silverlight:
	`<UFT Installation folder>\dat\Extensibility\Slv\<custom toolkit name>\Help`

Setting the DevelopmentMode Attribute

- If you modify attributes of **Identification Property** elements in the test object configuration file, keep the **DevelopmentMode** attribute of the **TypeInformation** element set to `true` during the design stages of the custom toolkit support. Before you deploy the custom toolkit support set for regular use, be sure to remove this attribute (or set it to `false`). For more information, see "Modifying Identification Property Attributes in a Test Object Configuration File" below.

Modifying Deployed Support

If you modify a deployed toolkit support set, you must reopen UFT and re-run the WPF or Silverlight application for the changes to take effect.

If you change the identification property definitions that specify the functionalities for which the properties are used in UFT, see "Modifying Identification Property Attributes in a Test Object Configuration File".

Modifying Identification Property Attributes in a Test Object Configuration File

The following attributes of the **Identification Property** element in the test object configuration file specify information that can be modified in UFT (using the Object Identification dialog box):

AssistivePropertyValue, **ForAssistive**, **ForBaseSmartID**, **ForDescription**, **ForOptionalSmartID**, and **OptionalSmartIDPropertyValue**. These attributes determine the lists of identification properties used for different purposes in UFT. For more information, see the *UFT Test Object Schema Help*, available in the UFT WPF and Silverlight Add-in Extensibility Help.

Therefore, by default, UFT reads the values of these attributes from the XML file only once, to prevent overwriting any changes a user makes using the Object Identification dialog box. In this way, UFT provides persistence for the user defined property lists.

If the user clicks the **Reset Test Object** button in the Object Identification dialog box, the attributes' values are reloaded from the XML.

If the XML changed since the last time it was loaded (based on the file's modification date in the system), UFT reads the attributes from the XML. UFT adds identification properties to the relevant lists (and adjusts their order if necessary) according to the values of these attributes, but does not remove any existing identification properties from the lists.

To instruct UFT to completely refresh the identification property lists according to the attributes defined in the XML each time UFT is opened, set the **DevelopmentMode** attribute of the **TypeInformation** element in this test object configuration file to `true`.

Considerations When Modifying Identification Properties Attributes

- If you modify attributes of **Identification Property** elements in the test object configuration file, keep the **DevelopmentMode** attribute of the **TypeInformation** element set to `true` during the design stages of the custom toolkit support. This ensures that UFT uses all of the changes you make to the file.
- Before you deploy the toolkit support set for regular use, be sure to remove the **DevelopmentMode** attribute of the **TypeInformation** element (or set it to `false`). Otherwise, every time UFT opens it will refresh the property lists based on the definitions in the test object configuration file. If UFT users change the property lists using the Object Identification dialog box, their changes will be lost when they reopen UFT.
- Though UFT does not remove existing properties from the property lists when reading a modified test object configuration file (unless the **DevelopmentMode** attribute is set to `true`), it does add properties and adjust the order of the lists based on the definitions in the file. If UFT users removed properties from the lists or modified their order using the Object Identification dialog box, those changes will be lost when a modified file is loaded.

  If you provide the custom toolkit support set to a third party, and you deliver an upgrade that includes a modified test object configuration file, consider informing the UFT users about such potential changes to their identification property lists.

Removing Deployed Support

When opening UFT, the UFT user can use the Add-in Manager to instruct UFT whether to load the support provided for any particular environment or toolkit.

If you want to remove support for a custom toolkit from UFT after it is deployed, you must delete its toolkit configuration file from the custom toolkit's folder under:
UFT Installation folder>\dat\Extensibility\WPF (or ...**\Slv**)

If none of the test object class definitions in a test object configuration file are used to represent any custom controls (meaning they are no longer needed), you can delete the file from: **UFT Installation Folder>\dat\Extensibility\WPF** (or ...**\Slv**) and
UFT Add-in for ALM Installation folder>\dat\Extensibility\WPF (or ...**\Slv)** if relevant.

Send Us Feedback

Can we make this Developer Guide better?

Tell us how: sw-doc@hp.com


Java™
COMPATIBLE