

OPTIMIZE

MERCURY QUICKTEST PROFESSIONAL™

.NET ADD-IN

VERSION 9.2

Extensibility Developer's Guide

MERCURY™

BUSINESS TECHNOLOGY OPTIMIZATION

Mercury QuickTest Professional .NET Add-in

Extensibility Developer's Guide

Version 9.2

Document Release Date: February 26, 2007

Mercury QuickTest Professional .NET Add-in Extensibility Developer's Guide, Version 9.2

This document, and the accompanying software and other documentation, is protected by U.S. and international copyright laws, and may be used only in accordance with the accompanying license agreement. Features of the software, and of other products and services of Mercury Interactive Corporation, may be covered by one or more of the following patents: United States: 5,511,185; 5,657,438; 5,701,139; 5,870,559; 5,958,008; 5,974,572; 6,137,782; 6,138,157; 6,144,962; 6,205,122; 6,237,006; 6,341,310; 6,360,332; 6,449,739; 6,470,383; 6,477,483; 6,549,944; 6,560,564; 6,564,342; 6,587,969; 6,631,408; 6,631,411; 6,633,912; 6,694,288; 6,738,813; 6,738,933; 6,754,701; 6,792,460 and 6,810,494. Australia: 763468 and 762554. Other patents pending. All rights reserved.

U.S. GOVERNMENT RESTRICTED RIGHTS. This Software Documentation is a "commercial item" as defined at 48 C.F.R. 2.101 (October 1995). In accordance with 48 C.F.R. 12.212 (October 1995), 48 C.F.R. 27.401 through 27.404 and 52.227-14 (June 1987, as amended) and 48 C.F.R. 227.7201 through 227.7204 (June 1995), and any similar provisions in the supplements to Title 48 of the C.F.R. (the "Federal Acquisition Regulation") of other entities of the U.S. Government, as applicable, all U.S. Government users acquire and may use this Documentation only in accordance with the restricted rights set forth in the license agreement applicable to the Computer Software to which this Documentation relates.

Mercury, Mercury Interactive, the Mercury logo, the Mercury Interactive logo, LoadRunner, WinRunner, SiteScope and TestDirector are trademarks of Mercury Interactive Corporation and may be registered in certain jurisdictions. The absence of a trademark from this list does not constitute a waiver of Mercury's intellectual property rights concerning that trademark.

All other company, brand and product names may be trademarks or registered trademarks of their respective holders. Mercury disclaims any responsibility for specifying which marks are owned by which companies or which organizations.

Mercury provides links to external third-party Web sites to help you find supplemental information. Site content and availability may change without notice. Mercury makes no representations or warranties whatsoever as to site content or availability.

Mercury Interactive Corporation
379 North Whisman Road
Mountain View, CA 94043
Tel: (650) 603-5200
Fax: (650) 603-5300
<http://www.mercury.com>

© 1992 - 2007 Mercury Interactive Corporation, All rights reserved

If you have any comments or suggestions regarding this document, please send them by e-mail to documentation@mercury.com.

Table of Contents

Welcome to This Guide	5
How This Guide Is Organized	6
Who Should Read This Guide	8
QuickTest Professional Online Documentation	8
Additional Online Resources.....	10
Documentation Updates	11
Typographical Conventions.....	12
Chapter 1: Introducing QuickTest Professional .NET Add-in	
Extensibility	13
Understanding .NET Add-in Extensibility	14
Using the .NET Windows Forms Spy	16
Understanding Coding Options: .NET DLL and XML.....	17
Understanding Custom Server Run-Time Contexts	18
Understanding Test Object Mapping.....	20
Chapter 2: Installing the Custom Server C# Project Template	21
Before You Install	22
Running the Installation Program	22
Uninstalling the Project Template	24
Chapter 3: Using a .NET DLL to Extend Support for a Custom	
Control	25
About Using a .NET DLL to Extend Support for a Custom	
Control.....	26
Creating a Custom Server.....	26
Using the XML Configuration Segment	31
Implementing Test Record for a Custom Control Using the	
.NET DLL.....	31
Implementing Test Run for a Custom Control Using the	
.NET DLL.....	36
Running Code under Application Under Test from the	
QuickTest Context	38
API Overview	39

Chapter 4: Using an XML File to Extend Support for a Custom Control	41
About Using an XML File to Extend Support for a Custom Control	42
Understanding the Control Definition XML File	42
Example of a Control Definition XML File	45
Chapter 5: Using Test Object Configuration Files	47
About Using Test Object Configuration Files	47
Guidelines for Implementing Test Object Configuration Files	48
Understanding the Test Object Configuration File	51
Deploying the Test Object Configuration File	53
Chapter 6: Configuring QuickTest to Use the Custom Server	55
About Configuring QuickTest to Use the Custom Server	55
Understanding the QuickTest System Windows Forms Configuration File	56
Chapter 7: Tutorial - Step-by-Step Basic Example	61
Creating a New Custom Server Project	62
Implementing Test Record Logic	65
Implementing Test Run Logic	67
Configuring QuickTest Professional	68
Testing the Custom Server	71
Understanding the TrackBarSrv.cs File	71
Chapter 8: Tutorial - Advanced Example	73
Toolbar Example	73
Understanding the ToolBarSrv.cs File	81
Index	85

Welcome to This Guide

Welcome to QuickTest Professional .NET Add-in Extensibility.

QuickTest Professional .NET Add-in Extensibility enables you to support testing applications using third-party and custom .NET controls that are not supported out-of-the-box by the QuickTest Professional .NET Add-in.

This chapter describes:	On page:
How This Guide Is Organized	6
Who Should Read This Guide	8
QuickTest Professional Online Documentation	8
Additional Online Resources	10
Documentation Updates	11
Typographical Conventions	12

How This Guide Is Organized

This guide explains how to use QuickTest Professional .NET Add-in Extensibility to extend QuickTest support for third-party and custom .NET controls.

This guide should be used together with the *QuickTest Professional .NET Add-in Extensibility API Reference* (provided in online Help format). These documents should also be used in conjunction with the *QuickTest Professional User's Guide*, the *QuickTest Professional .NET Add-in Guide*, and the *QuickTest Professional Object Model Reference*. These guides can be accessed online by choosing **Help > QuickTest Professional Help** from the QuickTest main window.

This guide contains:

- Chapter 1 Introducing QuickTest Professional .NET Add-in Extensibility**
Explains the concepts of extending support to custom .NET controls.
- Chapter 2 Installing the Custom Server C# Project Template**
Explains how to install the QuickTest .NET Add-in Extensibility module and how to configure your QuickTest Professional .NET Add-in project to use extensibility.
- Chapter 3 Using a .NET DLL to Extend Support for a Custom Control**
Explains how to extend support for a custom control using a .NET DLL.
- Chapter 4 Using an XML File to Extend Support for a Custom Control**
Explains how to extend support for a custom control using an XML file.
- Chapter 5 Using Test Object Configuration Files**
Explains how to use test object configuration files to enable additional functionality for custom methods and properties.

Chapter 6 Configuring QuickTest to Use the Custom Server

Explains how to configure QuickTest to use the Custom Server and describes the configuration file format.

Chapter 7 Tutorial - Step-by-Step Basic Example

Provides instructions and leads you step-by-step through the process of creating custom support for a control.

Chapter 8 Tutorial - Advanced Example

Provides instructions for creating custom support for a control that requires more complex support implementation.

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

Business components and scripted components are part of Mercury Business Process Testing, which utilizes a keyword-driven methodology for testing applications. For more information, refer to the *QuickTest Professional User's Guide* and the *QuickTest Professional for Business Process Testing User's Guide*.

Who Should Read This Guide

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

To use this guide, you should be familiar with:

- ▶ Major QuickTest features and functionality
- ▶ QuickTest Professional Object Model
- ▶ QuickTest Professional .NET Add-in
- ▶ XML (basic knowledge)
- ▶ .NET Programming in C#

QuickTest Professional Online Documentation

QuickTest Professional includes the following online documentation:

Readme provides the latest news and information about QuickTest. Choose **Start > Programs > QuickTest Professional > Readme**.

QuickTest Professional Installation Guide explains how to install and set up QuickTest. Choose **Help > Printer-Friendly Documentation > Mercury QuickTest Professional Installation Guide**.

QuickTest Professional Tutorial teaches you basic QuickTest skills and shows you how to design tests for your applications. Choose **Help > QuickTest Professional Tutorial**.

Product Feature Movies provide an overview and step-by-step instructions describing how to use selected QuickTest features. Choose **Help > Product Feature Movies**.

Printer-Friendly Documentation displays the complete documentation set in Adobe portable document format (PDF). Online books can be viewed and printed using Adobe Reader, which can be downloaded from the Adobe Web site (<http://www.adobe.com>). Choose **Help > Printer-Friendly Documentation**.

QuickTest Professional Help includes:

- ▶ **What's New in QuickTest Professional** describes the newest features, enhancements, and supported environments in the latest version of QuickTest.
- ▶ **QuickTest User's Guide** describes how to use QuickTest to test your application.
- ▶ **QuickTest for Business Process Testing User's Guide** provides step-by-step instructions for using QuickTest to create and manage assets for use with Business Process Testing.
- ▶ **QuickTest Object Model** describes QuickTest test objects, lists the methods and properties associated with each object, and provides syntax information and examples for each method and property.
- ▶ **QuickTest Advanced References** contains documentation for the following QuickTest COM and XML references:
 - ▶ **QuickTest Automation** provides syntax, descriptive information, and examples for the automation objects, methods, and properties. It also contains a detailed overview to help you get started writing QuickTest automation scripts. The automation object model assists you in automating test management, by providing objects, methods and properties that enable you to control virtually every QuickTest feature and capability.
 - ▶ **QuickTest Test Results Schema** documents the XML schema that enables you to customize your test results.
 - ▶ **QuickTest Test Object Schema** documents the XML schema that enables you to extend test object support in different environments.
 - ▶ **QuickTest Object Repository Automation** documents the Object Repository automation object model that enables you to manipulate QuickTest object repositories and their contents from outside of QuickTest.
- ▶ **VBScript Reference** contains Microsoft VBScript documentation, including VBScript, Script Runtime, and Windows Script Host.

To access the QuickTest Professional Help, choose **Help > QuickTest Professional Help**. You can also access the QuickTest Professional Help by clicking in selected QuickTest windows and dialog boxes and pressing F1. Additionally, you can view a description, syntax, and examples for a QuickTest test object, method, or property by placing the cursor on it and pressing F1.

Note: Your QuickTest Help may contain additional items relevant to any QuickTest add-ins you have installed. For more information, refer to the relevant add-in documentation.

Additional Online Resources

Knowledge Base uses your default Web browser to open the Mercury Customer Support Web site directly to the Knowledge Base landing page. Choose **Help > Knowledge Base**. The URL for this Web site is <http://support.mercury.com/cgi-bin/portal/CSO/kbBrowse.jsp>.

Customer Support Web Site uses your default Web browser to open the Mercury Customer Support Web site. This site enables you to browse the Mercury Support Knowledge Base and add your own articles. You can also post to and search user discussion forums, submit support requests, download patches and updated documentation, and more. Choose **Help > Customer Support Web Site**. The URL for this Web site is <http://support.mercury.com>.

Send Feedback enables you to send online feedback about **QuickTest Professional** to the product team. Choose **Help > Send Feedback**.

Mercury Home Page uses your default Web browser to access Mercury's Web site. This site provides you with the most up-to-date information on Mercury and its products. This includes new software releases, seminars and trade shows, customer support, educational services, and more. Choose **Help > Mercury Home Page** and click the **Mercury Home** link in the top right corner of the page. The URL for this Web site is <http://www.mercury.com>.

Mercury Best Practices contain guidelines for planning, creating, deploying, and managing a world-class IT environment. Mercury provides three types of best practices: Process Best Practices, Product Best Practices, and People Best Practices. Licensed customers of Mercury software can read and use the Mercury Best Practices available from the Customer Support site, <http://support.mercury.com>.

Documentation Updates

Mercury is continually updating its product documentation with new information. You can download the latest version of this document from the Customer Support Web site (<http://support.mercury.com>).

To download updated documentation:

- 1** If you are not already logged into the Customer Support Web site, enter your login information and click **Login**.
- 2** Click the **Documentation** link.
- 3** Under **Please Select Product**, select **QuickTest Professional**.

Note that if the required product does not appear in the list, you must add it to your customer profile. Click **My Account** to update your profile.

- 4** Click **Retrieve**. The Documentation page opens and lists the documentation available for the current release and for previous releases. If a document was updated recently, **Updated** appears next to the document name.
- 5** Click a document link to download the documentation.

Typographical Conventions

This guide uses the following typographical conventions:

UI Elements and Function Names	This style indicates the names of interface elements on which you perform actions, file names or paths, and other items that require emphasis. For example, “Click the Save button.” It also indicates method or function names. For example, “The wait_window statement has the following parameters:”
<i>Arguments</i>	This style indicates method, property, or function arguments and book titles. For example, “Refer to the <i>Mercury User’s Guide</i> .”
<Replace Value>	Angle brackets enclose a part of a file path or URL address that should be replaced with an actual value. For example, <MyProduct installation folder>\bin .
Example	This style is used for examples and text that is to be typed literally. For example, “Type Hello in the edit box.”
CTRL+C	This style indicates keyboard keys. For example, “Press ENTER.”
[]	Square brackets enclose optional arguments.
{ }	Curly brackets indicate that one of the enclosed values must be assigned to the current argument.
...	In a line of syntax, an ellipsis indicates that more items of the same format may be included. In a programming example, an ellipsis is used to indicate lines of a program that were intentionally omitted.
	A vertical bar indicates that one of the options separated by the bar should be selected.

1

Introducing QuickTest Professional .NET Add-in Extensibility

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

It is possible to record tests on .NET controls that are not supported out-of-the-box by the QuickTest Professional .NET Add-in without using the Extensibility module. However, the recorded steps will reflect the low-level activities passed as Windows messages. By supporting a .NET control with the Extensibility module, this default low-level support is extended so that tests are meaningful, understandable, and easy to modify.

This chapter describes:	On page:
Understanding .NET Add-in Extensibility	14
Using the .NET Windows Forms Spy	16
Understanding Coding Options: .NET DLL and XML	17
Understanding Custom Server Run-Time Contexts	18
Understanding Test Object Mapping	20

Understanding .NET Add-in Extensibility

QuickTest Professional .NET Add-in Extensibility enables you to support third-party and custom .NET controls by extending QuickTest test objects with methods representing the meaningful behaviors of those .NET controls.

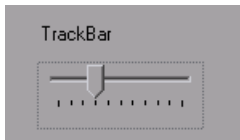
The QuickTest Professional .NET Add-in, without the Extensibility module, supports many .NET controls out-of-the-box. The .NET Add-in provides test objects that supply methods representing these controls' meaningful behaviors.

The Extensibility module enables you to implement this level of support for additional .NET controls. Using the Extensibility module, you extend the .NET Add-in interfaces by overriding existing methods and defining new ones, creating a Custom Server. When the custom control is mapped to an existing QuickTest test object, you have the full functionality of a QuickTest test object, including visibility in IntelliSense and meaningful steps in the test.

Understanding the Concept of Meaningful Behaviors

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

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



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

Action1			
Sample Application			
trackBar1	Drag	50,10	
trackBar1	Drop	32,11	
trackBar1	Drag	34,11	
trackBar1	Drop	51,12	
trackBar1	Drag	50,4	
trackBar1	Drop	23,7	
trackBar1	Click	83,10	
trackBar1	Click	91,11	
Close	Click		

In the Expert View, the recorded test looks like this:

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

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

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

Action1			
Sample Application			
trackBar1	SetValue	5	
trackBar1	SetValue	0	
trackBar1	SetValue	10	
trackBar1	SetValue	6	
Sample Application	Close		

In the Expert View, the recorded test looks like this:

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

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

Using the .NET Windows Forms Spy

The .NET Windows Forms Spy enables you to view details about selected .NET Windows Forms controls and their run-time properties. You can use the .NET Windows Forms Spy to help you develop extensibility for .NET Windows Forms controls.

You can use the .NET Windows Forms Spy when planning .NET Add-in extensibility implementation to create support for custom .NET Windows Forms controls. The .NET Windows Forms Spy assists you in examining .NET Windows Forms controls within your application. It also enables you to see which events cause your application to change (to facilitate record and run extensibility implementation) and how the changes manifest themselves in the control's state.

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

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

For more information on the .NET Windows Forms Spy, refer to the *QuickTest Professional .NET Add-in Guide*.

Understanding Coding Options: .NET DLL and XML

You can implement QuickTest custom support in the following ways:

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

Guidelines for Selecting a Coding Option

Most Custom Servers are implemented as a .NET DLL. This option is generally preferred because development is supported by all the services of the program development environment, such as syntax checking, debugging, and Microsoft IntelliSense. Furthermore, a Custom Server implemented as a .NET DLL can perform part of its Test Record functions in the **QuickTest** context and part in the **Application under test** context. For more information, see “Using a .NET DLL to Extend Support for a Custom Control” on page 25, and refer to the *QuickTest Professional .NET Add-in Extensibility API Reference*.

For information on run-time contexts, see “Understanding Custom Server Run-Time Contexts” on page 18.

The XML implementation is most practical either with relatively simple, well documented controls, or with controls that map well to an existing object but for which you need to replace the **Test Record** implementation, or replace or add a small number of test object **Test Run** methods. It is also useful when a full programming environment is not available because it requires only a text editor.

However, when implementing a custom control with XML, you have none of the support provided by a program development environment. The XML implementation runs only in the **Application under test** context. For more information, see “Using an XML File to Extend Support for a Custom Control” on page 41.

For information on setting the coding option, see “Configuring QuickTest to Use the Custom Server” on page 55.

Understanding Custom Server Run-Time Contexts

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

- **Application under test**
- **QuickTest**

An object created in the **Application under test** context has direct access to the .NET control’s events, methods, and properties. However, it cannot listen to Windows messages.

An object created in the **QuickTest** context can listen to Windows messages. However, it does not have direct access to the .NET control’s events, methods, and properties.

If the Custom Server is implemented as a .NET DLL, an object created in the **QuickTest** context can create Assistant objects that run in the **Application under test** context.

Guidelines for Selecting the Custom Server Run-Time Context

The Custom Server may implement **Test Record**, **Test Run**, or both.

Test Record is the software module used in a session that records the actions performed on the application being tested and the application's resulting behaviors. The recording is then converted to a test. If you plan to create tests using keyword-driven testing, and not by recording steps on an application, you do not need to implement **Test Record**.

Test Run is the software module used to test if the application is performing as required (by running the test and tracking the results). **Test Run** is nearly always implemented in the **Application under test** context. Direct access to the control makes setting values and calling the control's methods straightforward. There is no need to listen to Windows messages during a **Test Run** session, so the **QuickTest** context is not required. However, if your application uses **QuickTest** services more than it uses services of the custom control, it may be more efficient to implement **Test Run** in the **QuickTest** context.

The programming for **Test Record** is generally simpler in the **Application under test** context. However, if it is essential to use Windows messages for recording, you must use the **QuickTest** context.

If the .NET DLL Custom Server must both listen to Windows messages and access control events and properties, use **Assistant classes**. The Custom Server running in the **QuickTest** context can listen to events in the **Application under test** context with **Assistant class** objects that run in the **Application under test** context. These objects also provide direct access to control properties.

For more information, see “Implementing Test Record for a Custom Control Using the .NET DLL” on page 31.

For more information on **Assistant classes**, see “Using a .NET DLL to Extend Support for a Custom Control” on page 25, and refer to the *QuickTest Professional .NET Add-in Extensibility API Reference*.

For more information on setting the context, see “Configuring QuickTest to Use the Custom Server” on page 55.

Understanding Test Object Mapping

All Custom Servers are mapped to a parent QuickTest test object. When the test object is applied to the custom control, the Custom Server extends the parent test object.

When you map your Custom Server to a functionally similar QuickTest test object, you do not need to override those Test Run methods of the parent object that apply without change to your custom object. For example, most controls contain a Click method. If the Click method of the parent object implements the Click method of the custom object adequately, you do not need to override the parent's method.

To cover the Test Run functionality of the custom object that does not exist in the parent, add new methods in your Custom Server. To cover functionality that has the same method name, but a different implementation, override the parent methods. The custom control support consists of the Test Run members of the parent object (or overrides of those members), and new members added by this Custom Server.

Note that mapping is sometimes sufficient without any programming. If the parent QuickTest test object adequately covers a control, it is sufficient to map the control to the QuickTest test object. If the QuickTest test object adequately covers Test Record, but you need to customize Test Run, do not implement Test Record. If you do implement Test Record, the implementation replaces that of the parent object. You must implement all required Test Record functionality.

If you do not specify a mapping, QuickTest maps the custom control to the default generic test object, **SwfObject**.

When you edit a step that references the custom control, Microsoft IntelliSense displays the properties and methods of the custom control in addition to those of the parent QuickTest test object. QuickTest uses test object configuration files to provide IntelliSense for custom methods and properties. For more information, see “Using Test Object Configuration Files” on page 47.

For more information on mapping, see “Configuring QuickTest to Use the Custom Server” on page 55.

2

Installing the Custom Server C# Project Template

This chapter describes how to install the Custom Server C# Project Template for Microsoft Visual Studio .NET or Microsoft Visual Studio 2005.

This installation provides a Custom Server project template and the wizard that runs when the template is selected to create a new project.

The Custom Server template provides a framework of blank code, some sample code, and the QuickTest project references required to build a Custom Server.

The wizard simplifies setting up a Microsoft Visual Studio .NET or Microsoft Visual Studio 2005 project to create a Custom Server .NET DLL using the .NET Add-in Extensibility module. For more information, see “Using a .NET DLL to Extend Support for a Custom Control” on page 25.

This chapter describes:	On page:
Before You Install	22
Running the Installation Program	22
Uninstalling the Project Template	24

Before You Install

Before you install the Custom Server C# Project Template, review the requirements listed below.

- ▶ You must have access to the **InstWizard.msi** file. You can access the **InstWizard.msi** file either from a computer on which the QuickTest Professional .NET Add-in is installed, or from the root folder of the QuickTest Professional .NET Add-in CD-ROM.
- ▶ Microsoft Visual Studio .NET or Microsoft Visual Studio .NET 2005 must be installed on your computer.

Running the Installation Program

The **InstWizard.msi** file is located in the QuickTest Professional .NET Add-in installation, and on the QuickTest Professional .NET Add-in CD-ROM.

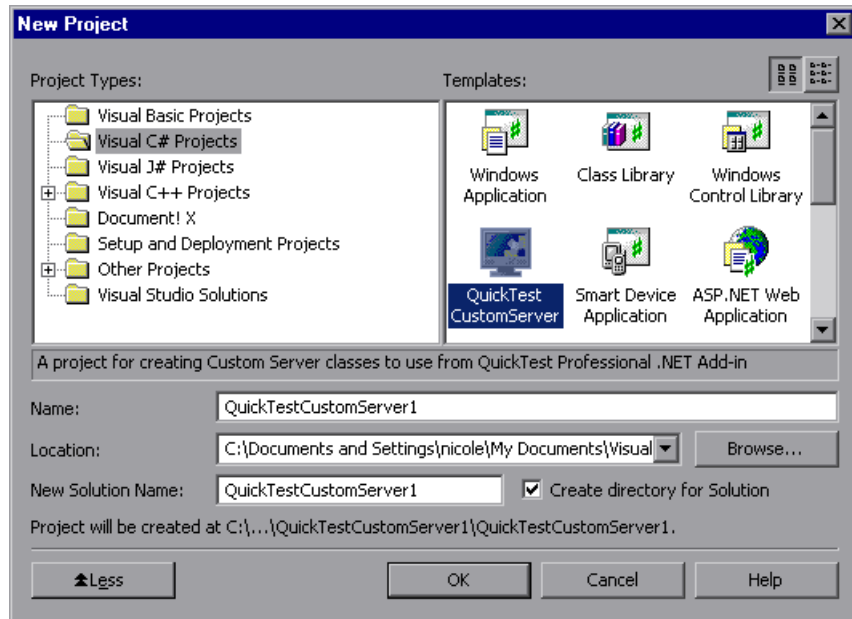
To install the .NET Add-in Custom Server C# Project Template:

- 1** Close all instances of Microsoft Visual Studio.
- 2** Locate the **InstWizard.msi** file. You can find it in one of the following locations:
 - ▶ In the <QuickTest Professional installation path>\bin\Custom folder on a computer on which the QuickTest Professional .NET Add-in is installed.
 - ▶ In the root folder of the QuickTest Professional .NET Add-in CD-ROM.
- 3** Run the installation by double-clicking on the **InstWizard.msi** file. The Custom Server C# Project Template is installed on your computer.

To confirm that the installation was successful:

- 1** Open Microsoft Visual Studio .NET or Microsoft Visual Studio 2005.
- 2** Choose **File > New > Project** to open the New Project dialog box.
- 3** Select **Visual C# Projects** in the **Project Types** list.

- 4 Confirm that the **QuickTest CustomServer** template icon is displayed in the **Templates** pane.



Note: The above dialog box is from Microsoft Visual Studio .NET. The New Project dialog box in Microsoft Visual Studio 2005 differs slightly in appearance.

Uninstalling the Project Template

You can uninstall the Custom Server C# Project Template from the Windows Control Panel.

To uninstall the project template:

- 1** Select **Start > Settings > Control Panel > Add/Remove Programs**. The Add/Remove Programs dialog box opens.
- 2** In the **Add/Remove Programs** list, select **Mercury CustomWizard**.
- 3** Click **Remove**.

3

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

You can support a .NET control by creating a Custom Server implemented as a .NET DLL.

To create a .NET DLL Custom Server you need to know how to program a .NET Assembly. The illustrations and instructions in this chapter assume that you are using Microsoft Visual Studio .NET as your development environment and that the Custom Server C# Project Template is installed. For more information, see “Installing the Custom Server C# Project Template” on page 21.

This chapter describes:	On page:
About Using a .NET DLL to Extend Support for a Custom Control	26
Creating a Custom Server	26
Using the XML Configuration Segment	31
Implementing Test Record for a Custom Control Using the .NET DLL	31
Implementing Test Run for a Custom Control Using the .NET DLL	36
Running Code under Application Under Test from the QuickTest Context	38
API Overview	39

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

You can create a Custom Server to implement high level support for a custom .NET control. The Custom Server is a .NET DLL class library that implements interfaces for **Test Record** and/or **Test Run**, and general utilities. For more information, see “Implementing Test Record for a Custom Control Using the .NET DLL” on page 31, “Implementing Test Run for a Custom Control Using the .NET DLL” on page 36, and “API Overview” on page 39.

After creating the Custom Server, configure QuickTest to use it. For more information, see “Configuring QuickTest to Use the Custom Server” on page 55.

Creating a Custom Server

To create a Custom Server, set up a .NET project in Microsoft Visual Studio .NET, code the support for QuickTest **Test Record** and/or **Test Run**, and edit the configuration file so that QuickTest loads the Custom Server.

Setting up the .NET Project

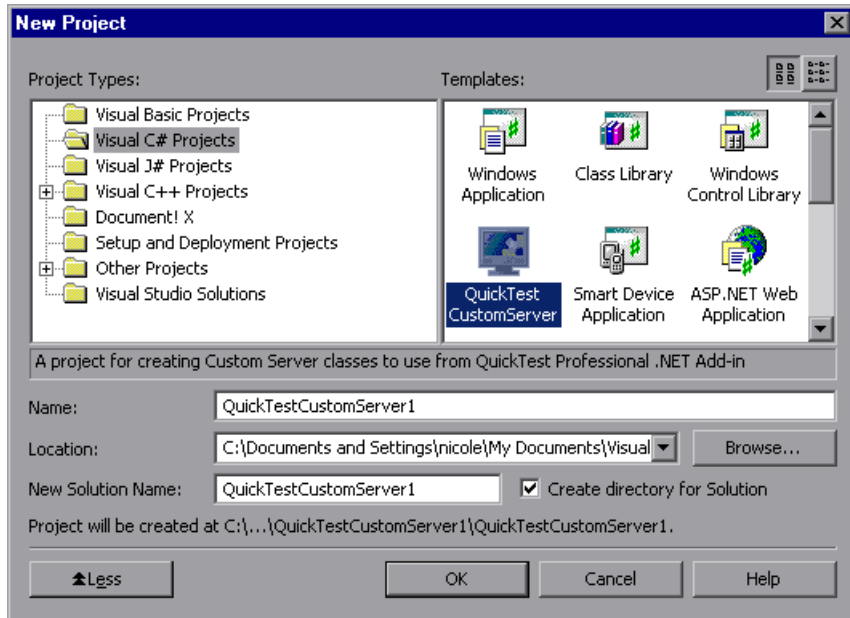
Set up a .NET project in Microsoft Visual Studio .NET using the Custom Server C# Project Template.

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

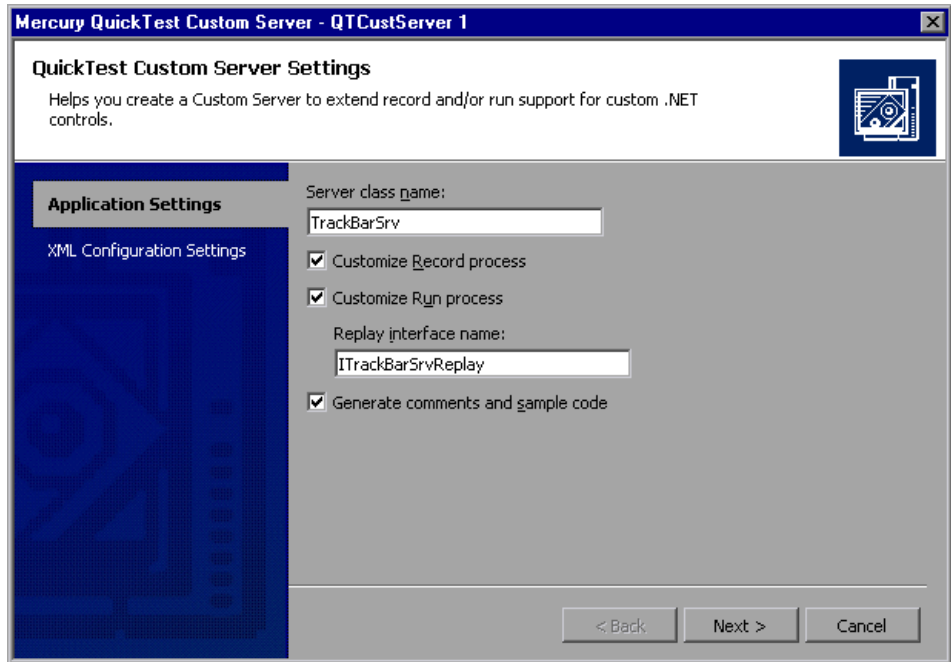
- ▶ Creates an XML file with definitions of the Custom Server that you can copy into the QuickTest configuration file.
- ▶ Creates the project files necessary for the build of the .DLL file.
- ▶ Sets up a C# file with commented code that contains the definitions of methods that you can override when you implement **Test Record** or **Test Run**.
- ▶ Provides sample code that demonstrates some **Test Record** and **Test Run** implementation techniques.

To setup a new .NET project:

- 1** Start Microsoft Visual Studio .NET.
- 2** Choose **File > New > Project** to open the New Project dialog box, or press CTRL + SHIFT + N. The New Project dialog box opens.
- 3** Select **Visual C# Projects** in the **Project Types** list.



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



- 5 Make your selections in the Application Settings page of the wizard.
 - In the **Server class name** box, provide a descriptive name for your custom server class.
 - Select the **Customize Record process** check box if you intend to implement the Test Record process in QuickTest.

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

Do not select this check box if you are going to create the test manually in QuickTest, or if you are going to use the Test Record functions of the parent test object to which this control will be mapped. Note that if you implement Test Record, the implementation replaces that of the parent object. You must implement all required Test Record functionality.

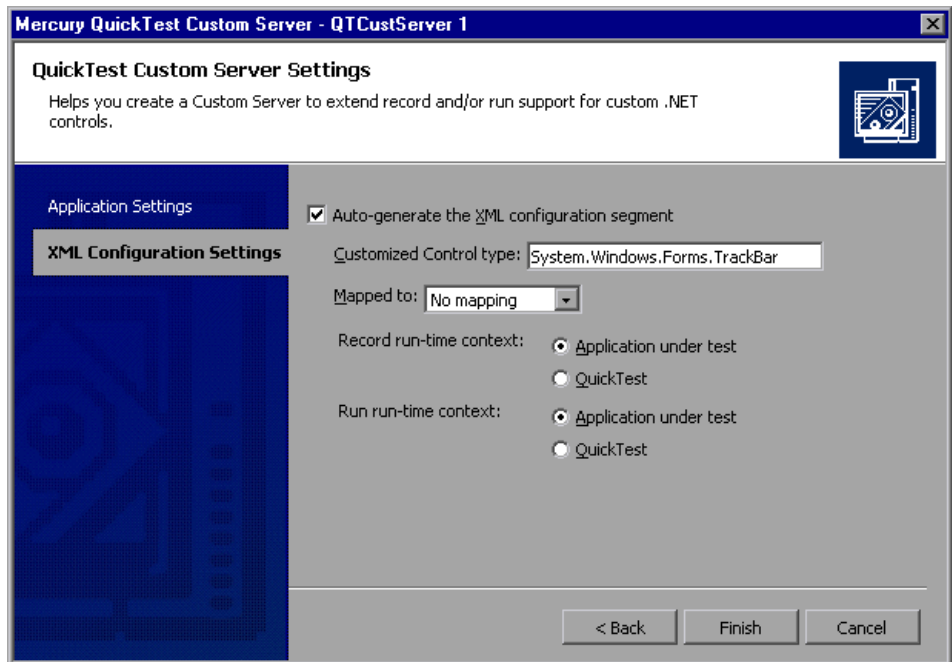
- ▶ Select the **Customize Run process** check box if you intend to implement the Test Run functions for the custom control. Enter a name for the replay interface you will create in the **Replay interface name** box.

If you select the **Customize Run process** check box, the wizard creates a framework of code to implement Test Run support.

Select the **Customize Run process** check box if you are going to override any of the existing test object's methods, or extend the test object with new methods.

- ▶ Select the **Generate comments and sample code** check box if you want the wizard to add comments and samples in the code that it generates.

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



7 Make your selections in the XML Configuration Settings page of the wizard.

- ▶ Select the **Auto-generate the XML configuration segment** check box to instruct the wizard to create the **Configuration.xml** file, containing an XML segment with the configuration information for QuickTest.
- ▶ In the **Customized Control type** box, enter the full type name of the control for which you are creating the Custom Server, including all wrapping namespaces, for example, `System.Windows.Forms.CustomButton`.
- ▶ In the **Mapped to** box, select the test object to which you want to map the Custom Server. If you select **No mapping**, the Custom Server is automatically mapped to the **SwfObject** test object.

For more information, see “Understanding Test Object Mapping” on page 20.

- ▶ Select the run-time context for Test Record and/or Test Run: **Application under test** or **QuickTest**.

For more information, see “Understanding Custom Server Run-Time Contexts” on page 18.

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

When you click **Finish** in the wizard, a **Configuration.xml** file is created and added to the project. When you are ready to use the Custom Server, update and modify the configuration information as required and transfer it to the QuickTest configuration file as described in “Using the XML Configuration Segment” on page 31.

Using the XML Configuration Segment

The XML configuration segment created by the wizard is used when the Custom Server is ready for deployment. Before using it, add the information that was not available when you created the project.

To use the XML configuration segment when configuring QuickTest:

- 1** Edit the **Configuration.xml** file in the project to ensure that the information is correct. Set the **DllName** element value to the location in which you will install the Custom Server. If **Test Record** and/or **Test Run** are to be loaded in different run-time contexts, edit the **Context** value accordingly.
- 2** Copy the entire **<Control>...</Control>** node. Do not include the enclosing **<Controls>** tags.
- 3** Open the QuickTest Professional .NET Add-in configuration file, **<QuickTest Professional>\dat\SwfConfig.xml**. Paste the **Control** node from **Configuration.xml** at the end of the file, before the closing **</Controls>** tag.
- 4** Save the file. If QuickTest was open, you must close and reopen it for the **SwfConfig.xml** changes to take effect.

For more information, see “Configuring QuickTest to Use the Custom Server” on page 55.

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

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

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

To implement **Test Record**, implement the methods in the **IRecord** interface created by the wizard. Add all the functionality required by your application. Your **Test Record** implementation does not inherit from the parent test object to which the custom control is mapped. It replaces the parent object's **Test Record** implementation entirely. Therefore, if you need any of the parent object's functionality, code it explicitly.

Before reading this section, make sure you are familiar with “Understanding Custom Server Run-Time Contexts” on page 18.

For more details about the interfaces, classes, enumerations, and methods in this section, refer to the *QuickTest Professional .NET Add-in Extensibility API Reference*.

This section describes:

- Implementing the **IRecord** Interface
- Writing Test Object Methods to the Test

Implementing the IRecord Interface

To implement the **IRecord** interface, override the call-back methods described in this section and add the details of your implementation in your event handlers or message handler.

Callback Method InitEventListener

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

1 Implement handlers for the control's events.

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

```
public void OnMouseDown(object sender, MouseEventArgs e)
{
    // Get the event.
    if(e.Button != System.Windows.Forms.MouseButtons.Left)
        return;
    /*
    For more complex events, here you would get any
    other information you need from the control.
    */
    // Write the test object method to the test
    RecordFunction("MouseDown",
        RecordingMode.RECORD_SEND_LINE,
        e.X,e.Y);
}
```

For more information, see “Writing Test Object Methods to the Test” on page 36.

2 Add your event handlers in InitEventListener:

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

Note that if Test Record will run in the **Application under test** context, you can use the syntax:

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

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

3 Add a remote event listener.

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

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

When you implement a remote event listener, you must override `EventListenerBase.InitEventListener` and `EventListenerBase.ReleaseEventListener` in addition to overriding these call-back functions in `CustomServerBase`. The use of these `EventListenerBase` call-backs is the same as for the `CustomServerBase` call-backs. For details, refer to the `EventListenerBase` class in the *QuickTest Professional .NET Add-in Extensibility API Reference*.

Note that when you handle events from the **QuickTest** context, the event arguments must be serialized. For details, refer to `CustomServerBase.AddHandler(String, Delegate, Type)` and the `IEventArgsHelper` interface in the *QuickTest Professional .NET Add-in Extensibility API Reference*.

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

Callback Method OnMessage

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

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

For details, refer to CustomServerBase.OnMessage in the *QuickTest Professional .NET Add-in Extensibility API Reference*.

Callback Method GetWndMessageFilter

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

For details, refer to IRecord.GetWndMessageFilter in the *QuickTest Professional .NET Add-in Extensibility API Reference*.

Callback Method ReleaseEventListener

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

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

you must release it:

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

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

Writing Test Object Methods to the Test

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

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

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

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

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

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

- ▶ Sets the values of attributes of the control object
- ▶ Calls a method of the control object
- ▶ Makes mouse and keyboard simulation calls
- ▶ Reports a step outcome to QuickTest

- Reports an error to QuickTest
- Makes calls to another library (to show a message box, write custom log, and so forth)

The custom control is mapped to a parent QuickTest test object. If there is no explicit mapping, it is mapped to **SwfObject**. The test object type that supports the custom control is the new type that consists of the members of the parent object (or overrides of those members), and new members added by this Custom Server.

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

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

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

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

For more details, refer to the *QuickTest Professional .NET Add-in Extensibility API Reference*.

Running Code under Application Under Test from the QuickTest Context

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

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

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

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

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

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

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

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

API Overview

This section provides a quick reference of the most commonly used API calls.

For more details, refer to the *QuickTest Professional .NET Add-in Extensibility API Reference*.

Test Record Methods

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

Test Record Callback Methods

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

Test Run Methods

DragAndDrop, KeyDown, KeyUp, MouseClick, MouseDbClick, MouseDown, MouseMove, MouseUp, PressKey, PressNKeys, SendKeys, SendString	Mouse and keyboard simulation methods.
PrepareForReplay	Prepares the control for an action run.
ReplayReportStep	Writes an event to the test report.
ReplayThrowError	Generates an error message and changes the reported step status.

ShowError	Displays the .NET warning icon.
TestObjectInvokeMethod	Invokes one of the methods exposed by the test object's IDispatch interface.

Cross-Process Methods

AddRemoteEventListener	Creates an EventListener instance in the Application under test process.
CreateRemoteObject	Creates an instance of an Assistant object in the Application under test process.
GetEventArgs (IEventArgs)	Retrieves and deserializes the EventArgs object.
Init (IEventArgsHelper)	Initializes the EventArguments helper class with an EventArgs object.
InvokeAssistant	Invokes a method of a CustomAssistantBase class in the control's thread.
InvokeCustomServer	Invokes the Custom Server's methods running in the QuickTest process from the Application under test process.
SetTargetControl	Attaches to the source control object by the control's window handle.

General Methods

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

4

Using an XML File to Extend Support for a Custom Control

You can extend support for a customized .NET control using an XML file. This enables you to extend support without a program development environment.

This chapter describes:	On page:
About Using an XML File to Extend Support for a Custom Control	42
Understanding the Control Definition XML File	42
Example of a Control Definition XML File	45

About Using an XML File to Extend Support for a Custom Control

You can implement custom control support without programming a .NET DLL by entering the appropriate **Test Record** and **Test Run** instructions in a Control Definition XML file. You can instruct QuickTest Professional to load the instructions by pointing to this control definition file in the QuickTest configuration file, **SwfConfig.xml**.

When using this technique, you do not have the support of the .NET development environment—the object browser and the debugger. However, by enabling the implementation of custom control support without the .NET development environment, this technique enables relatively rapid implementation, even in the field.

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

Understanding the Control Definition XML File

The Control Definition XML file specifies the control events to be captured during recording and to be used to generate steps to be written to the test. These steps are calls to methods of the custom control's test object. The file also specifies the operations QuickTest performs for each method during **Test Run**. You do not always need to enter both a **Record** and a **Run** element.

If the custom object is mapped to a parent test object that implements either all the required **Test Record** methods or all the required **Test Run** methods, you do not need to create the section of the definition file that defines that element.

If you create a **Record** element, the definitions replace the **Test Record** implementation of the parent object entirely. If you create a **Run** element, it inherits the **Test Run** implementation of the parent object and extends it. For more information on test object mapping options, see “Understanding Test Object Mapping” on page 20.

The structure of the Control Definition XML file is:

```

<?xml version="1.0" encoding="UTF-8" ?>
<Customization>
  <Record>
    <Events>
      <!-- There are 1 to n Event elements -->
      <Event name="controlEventName" enabled="true|false">
        <RecordedCommand name="theCommandName">
          <!-- There are 0 to n Parameter elements -->
          <Parameter> param</Parameter>
        </RecordedCommand>
      </Event>
    </Events>
  </Record>
  <Replay>
    <Methods>
      <!-- There are 1 to n Method elements -->
      <Method name="theCommandName">
        <Parameters>
          <!-- There are 0 to n Parameter elements -->
          <Parameter type="theDataType" name="param 1
            name"></Parameter>
        </Parameters>
        <MethodBody>theCommand</MethodBody>
      </Method>
    </Methods>
  </Replay>
</Customization>

```

Control Definition File Elements

- **Customization.** The root element.
- **Record.** Information on the conversion of events to test steps.
- **Events.** Collection of Event elements.
- **Event.** Contains the information needed to convert a specific event to a step in a test. It contains the following attributes:
 - **name.** The name of the control event.
 - **enabled.** Indicates whether recording is active for this event. Can be true or false.
- **RecordedCommand.** Defines the step to be written when the event described in the parent Event element is received. Has the following attribute:
 - **name.** The test object method name to write to the test.
- **Parameter.** Each Parameter element defines a parameter to be written to the test after the name of the RecordedCommand. The parameters are written to the test in the order in which they are defined in the Control Definition XML file.

A Parameter element may contain a single line of text content that will be evaluated and then written to the test. Alternatively, it may contain a short section of code to be run to produce the value to be written. In this case, the **lang** attribute must be specified, and the final value must be assigned to the return value variable, **Parameter**.

You can use the following reserved words in a Parameter element:

- **Sender.** The object that fired the event.
- **EventArgs.** The object that represents EventArgs parameter of the Event Handler.
- **Parameter.** The return value of the code.

You can specify the following optional attribute in a Parameter element:

- **lang.** If the element contains code, the **lang** attribute specifies the programming language. Currently, C# is supported.

- **Replay.** Information on the conversion of test object methods to the activities to be performed during the Test Run session.
- **Methods.** Collection of Method elements.
- **Method.** Defines a method added to the test object interface. It has the following attribute:
 - **name.** The test object method name.
- **Parameters.** Collection of Parameter elements.
- **Parameter.** Each Parameter element contains instructions for reading a command line parameter from the test. The order of Parameter elements must be the same as the order of the command line parameters in the script.

These parameters are used in the MethodBody element to create the method call. Each parameter element has the following attributes:

 - **type.** The data type of the value as it will be used in the MethodBody.
 - **name.** The name by which to refer to the value in the MethodBody.
- **MethodBody.** A series of C# instructions to perform when the test object method is executed.

The reserved word RtObject refers to the run-time object.

Example of a Control Definition XML File

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<Customization>
  <Record>
    <Events>
      <Event name="MouseUp" enabled="true">
        <RecordedCommand name="SetValue">
          <Parameter>
            Sender.Value
          </Parameter>
          <Parameter lang="C#">
            String xy;
            xy = EventArgs.X + ";" + EventArgs.Y;
            Parameter = xy;
          </Parameter>
        </RecordedCommand>
      </Event>
    </Events>
  </Record>
  <Replay>
    <Methods>
      <Method name="SetValue">
        <Parameters>
          <Parameter type="int" name="Value"/>
          <Parameter type="String" name="MousePosition"/>
        </Parameters>
        <MethodBody>
          RtObject.Value = Value;
          System.Windows.Forms.MessageBox.Show(MousePosition,
            "Mouse Position at Record Time");
        </MethodBody>
      </Method>
    </Methods>
  </Replay>
</Customization>
```


5

Using Test Object Configuration Files

You can use test object configuration files to enable additional functionality for custom methods and properties.

This chapter describes:	On page:
About Using Test Object Configuration Files	47
Guidelines for Implementing Test Object Configuration Files	48
Understanding the Test Object Configuration File	51
Deploying the Test Object Configuration File	53

About Using Test Object Configuration Files

Test object configuration files contain definitions of test objects (for example, their identification properties, the test object methods they support, and so forth). You create a test object configuration file according to a specific XML schema, and then place the XML file in a specific location on the QuickTest computer.

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

Each time you open QuickTest, it reads all of the test object configuration files and merges the information for each test object class from the different files into a single test object definition.

Guidelines for Implementing Test Object Configuration Files

Implementation of a test object configuration file is optional. If you choose not to implement the test object configuration file, the test object methods and properties defined in the .NET Custom Server DLL or Control Definition XML files will work as expected, but some additional functionality will be missing. For example, custom methods will not be displayed in IntelliSense, and they will not have tooltips, custom icons, context-sensitive Help, or documentation in the Keyword View Documentation column.

By creating a test object configuration file, you can implement the additional functionality described above. When you add a custom method or property to the test object configuration file, the definition is added to the existing definition of this test object class, affecting all objects of this class.

For example, if you add an identification property, it appears in QuickTest in the list of properties for all objects of this class, but has no value unless it is implemented for the specific control. If you specify that the identification property should be available for use in checkpoints, and you create a checkpoint on this property in a test, the checkpoint will fail if the identification property does not exist in the relevant control.

If you add a test object method, it is displayed in IntelliSense list of test object methods in QuickTest, but if you use the test object method in a test, and it is not supported for the relevant control, a run-time error occurs. For this reason, you should only define custom methods and properties in the test object configuration file if you want them to be available for all test objects of the specified class. This is because after you implement the test object configuration file, custom methods and properties are automatically displayed in IntelliSense for all test objects of the relevant class, even if they are not supported or relevant for a specific test object. Test designers may use a custom method in a test step without realizing that it is not relevant for a specific test object, and then the test run will fail.

Tip: It is recommended that you add a unique prefix to all custom method and property names so that test designers can easily identify the custom methods and properties and use them in test steps only if they know that the custom method or property is supported for the specific test object.

You must also make sure that the information you define in the test object configuration file exactly matches the corresponding information defined in the .NET Custom Server DLL or Control Definition XML files. For example, the test object method names must be exactly the same in both locations. Otherwise, the methods will appear to be available (for example, in IntelliSense) but they will not work, and, if used, the run session will fail.

Note: When you modify a test object configuration file, the changes take effect only after you restart QuickTest.

Understanding How QuickTest Merges Configuration Files

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

You define the priority of each test object configuration file using the **Priority** attribute of the **TypeInfo** element. For more information, see the *QuickTest Test Object Schema Help* (**Help > QuickTest Professional Help > QuickTest Advanced References > QuickTest Test Object Schema**).

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

When multiple test object definitions exist, QuickTest must handle any conflicts that arise. The following sections describe the process QuickTest follows when **ClassInfo**, **ListOfValues**, and **Operation** elements are defined in multiple test object configuration files.

ClassInfo Elements

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

ListOfValues Elements

- ▶ If a conflict arises between **ListOfValues** definitions in different files, the definition in the file with the higher priority overrides (replaces) the information in the file with the lower priority (the definitions are not merged).

- ▶ If a **ListOfValues** definition overrides an existing list, the new list is updated for all arguments of type **Enumeration** that are defined for operations of classes in the same test object configuration file.
- ▶ If a **ListOfValues** is defined in a configuration file with a lower priority than the existing definition, the lower priority definition is ignored.

Operation Elements

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

Understanding the Test Object Configuration File

The test object configuration file contains information on test object classes, methods, and properties and enables additional functionality that cannot be implemented in the .NET Custom Server DLL or Control Definition XML files. The test object configuration file follows a defined XML schema and is used in conjunction with the .NET Custom Server DLL or Control Definition XML files.

A test object configuration file can include:

- ▶ the name of the test object class and its attributes
- ▶ the icon to use for the test object class in the Keyword View, Step Generator, Test Results, and object repository (Optional. If not defined, the default test object icon is used.)
- ▶ the methods and properties for the test object class, including the following information for each method and property:
 - ▶ the arguments, including the argument type and direction
 - ▶ whether the argument is mandatory, and, if not, its default value

- ▶ the description (shown as a tooltip in the Keyword View, Expert 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
- ▶ the test object method that is selected by default in the Keyword View and Step Generator when a step is generated for a test object of this class
- ▶ the identification properties that are available for use in checkpoints
- ▶ the context-sensitive help topic to open when F1 is pressed on a selected object, method, or property in the Keyword View and Expert View, or when the **Operation Help** button is clicked for a specific method or property in the Step Generator

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

```

</TypeInfoInformation>
...
  <ClassInfo Name="SwfObject"
...
    <TypeInfo>
      <Operation Name="Set" PropertyType="Method"
        ExposureLevel="CommonUsed">
        <Description>Set the percentage in the task bar</Description>
        <Documentation><![CDATA[Set the %l %t to <Percent> percent.]]>
          </Documentation>
        <Argument Name="Percent" IsMandatory="true" Direction="In">
          <Type VariantType="Integer"/>
          <Description>The percentage to set in the task bar.
            </Description>
        </Argument>
      </Operation>
    </TypeInfo>
  </ClassInfo>
</TypeInfoInformation>

```

For information on the structure and syntax of the test object configuration file, refer to the *QuickTest Test Object Schema Help* (**Help > QuickTest Professional Help > QuickTest Advanced References > QuickTest Test Object Schema**).

Deploying the Test Object Configuration File

After you create the test object configuration file, you must deploy it by placing it in the correct location, so that the test object definitions are available to QuickTest. Make sure that QuickTest is closed, and then place the test object configuration files in the **<QuickTest installation folder>\dat\Extensibility\DotNet** folder. If the QuickTest Add-in for Quality Center is installed, you must also place the test object configuration files in the **<QuickTest Add-in for Quality Center installation folder>\Dat\Extensibility\DotNet** folder.

Each time you open QuickTest, it reads all of the test object configuration files in this location and merges the information for each test object class from the different files into a single test object definition.

6

Configuring QuickTest to Use the Custom Server

The QuickTest System Windows Forms Configuration File provides QuickTest with all the information necessary to load your Custom Server with the required configuration.

This chapter describes:	On page:
About Configuring QuickTest to Use the Custom Server	55
Understanding the QuickTest System Windows Forms Configuration File	56

About Configuring QuickTest to Use the Custom Server

To instruct QuickTest to load Custom Servers and to define the required configuration for QuickTest, enter the information in the QuickTest System Windows Forms Configuration File. The configuration file, **SwfConfig.xml**, is located in the <**QuickTest Professional installation path**>\dat folder.

Each control is configured in a **Control** node in the file.

Understanding the QuickTest System Windows Forms Configuration File

The structure of the **SwfConfig.xml** file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Controls>
  <Control Type=" " MappedTo="" >
    <CustomRecord>
      <Component>
        <Context> </Context>
        <DIIName></DIIName>
        <TypeName></TypeName>
      </Component>
    </CustomRecord>
    <CustomReplay>
      <Component>
        <Context></Context>
        <DIIName></DIIName>
        <TypeName></TypeName>
      </Component>
    </CustomReplay>
    <Settings>
      <Parameter Name=""> </Parameter>
    </Settings>
  </Control>
</Controls>
```

Configuration File Elements

- ▶ **?xml**. The XML declaration, version="1.0" encoding="UTF-8"?, is required.
- ▶ **Controls**. The root element.
- ▶ **Control**. The information required to support a custom control.

Attributes:

- ▶ **Type.** The custom control's full type including wrapping namespaces, for example, `System.Windows.Forms.CustomButton`.
- ▶ **MappedTo.** Optional. A QuickTest test object class containing behaviors that are similar to those that your Custom Server will inherit, for example, `SwfCheckBox` or `SwfButton`.
- ▶ **Settings.** This element is generally a collection of `Parameter` elements. (For .NET DLL Custom Servers, the element is optional.)

The first use is to pass information for the internal use of your Custom Server. This use is optional. You can use these parameters for any purpose appropriate to your application. You may also use a different structure—you are not bound to a collection of `Parameter` elements. However, if you use a different structure, you must parse it yourself in code, whereas the collection of `Parameter` elements has straightforward support in the API.

The second use is required when extended control support is implemented with XML, and you must use the collection of `Parameter` elements. The full path and name of the XML file containing the implementation of the extended control support is passed in a `Parameter` element where the **Name** attribute is `ConfigPath` and the value of the element is the file path name.

- ▶ **Parameter.** A value to be passed to the Custom Server at run time.
 - ▶ **Name.** The name of the `Parameter`.
- ▶ **CustomRecord.** The information required for Test Record.
- ▶ **CustomReplay.** The information required for Test Run.

The `CustomRecord` and `CustomReplay` nodes both contain a `Component` node. Not all `Component` sub-elements apply to both processes.

- ▶ **Component.** The Custom Server component data.

- ▶ **Context.** The Custom Server run-time context and the coding option.
 - ▶ **AUT.** The run-time context is the **Application under test** process. The support is implemented as a .NET DLL Custom Server.
 - ▶ **QTP.** The run-time context is the **QuickTest** process. The support is implemented as a .NET DLL Custom Server.
 - ▶ **AUT-XML.** The run-time context is the **Application under test** process. The support is implemented in an XML file.
- ▶ **DllName.** The filename of the DLL in which the user's class type is defined. Applies to the .NET DLL coding option only. You can use the full path and file name. Alternatively, if the Custom Server assembly is installed in the global assembly cache (GAC), pass the type name with the standard syntax, for example:

```
myQTCustomServer
```

or

```
myQTCustomServer, Version=1.0.1234.0
```

or

```
myQTCustomServer, Version=1.0.1234.0, Culture="en-US",  
PublicKeyToken=b77a5c561934e089c
```

- ▶ **TypeName.** The name of the type created by the Custom Server, including wrapping namespaces. Applies to the .NET DLL coding option only.

Example of a Configuration XML File

Following is an example of a file that configures QuickTest to recognize two controls.

Support for the **CustomMyListView.CustListView** control is implemented in a .NET DLL Custom Server. **MyListView** is mapped to the SwfListView test object, and runs in the **Application under test** context. The Custom Server is not installed in the GAC.

Support for the **mySmileyControls.SmileyControl2** control is implemented in an XML file.

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

  <Control Type="mySmileyControls.SmileyControl2">
    <Settings>
      <Parameter Name="ConfigPath">d:\Qtp\bin\ConfigSmiley.xml
```

Chapter 6 • Configuring QuickTest to Use the Custom Server

```
    </Parameter>
  </Settings>
  <CustomRecord>
    <Component>
      <Context>AUT-XML</Context>
    </Component>
  </CustomRecord>
  <CustomReplay>
    <Component>
      <Context>AUT-XML</Context>
    </Component>
  </CustomReplay>
</Control>
</Controls>
```

7

Tutorial - Step-by-Step Basic Example

In this tutorial, you will learn how to build a Custom Server for a Microsoft TrackBar control that enables QuickTest Professional to record and run a **SetValue** operation on the control.

This tutorial refers to Visual Studio .NET. However, you can use also Visual Studio 2005 to build the Custom Server as described in this tutorial.

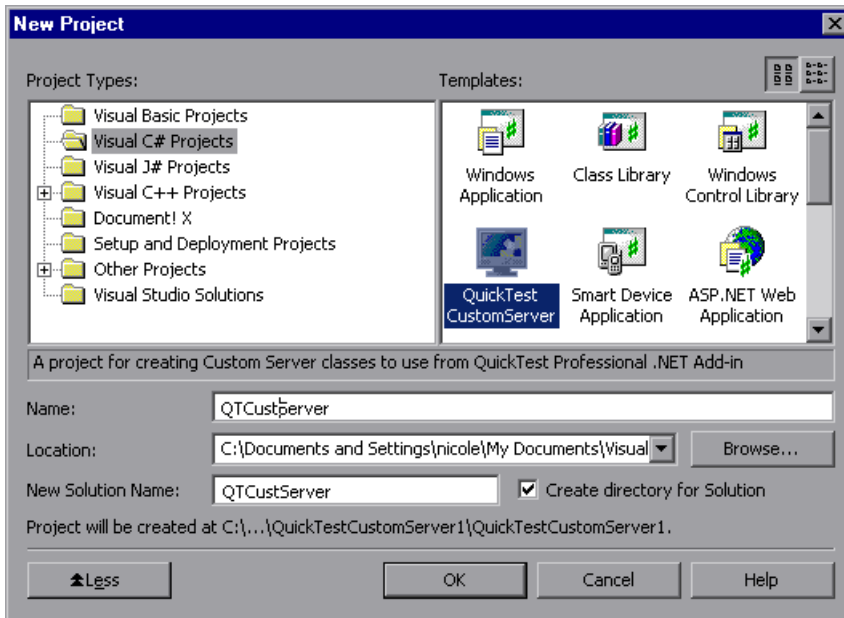
This chapter describes:	On page:
Creating a New Custom Server Project	62
Implementing Test Record Logic	65
Implementing Test Run Logic	67
Configuring QuickTest Professional	68
Testing the Custom Server	71
Understanding the TrackBarSrv.cs File	71

Creating a New Custom Server Project

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

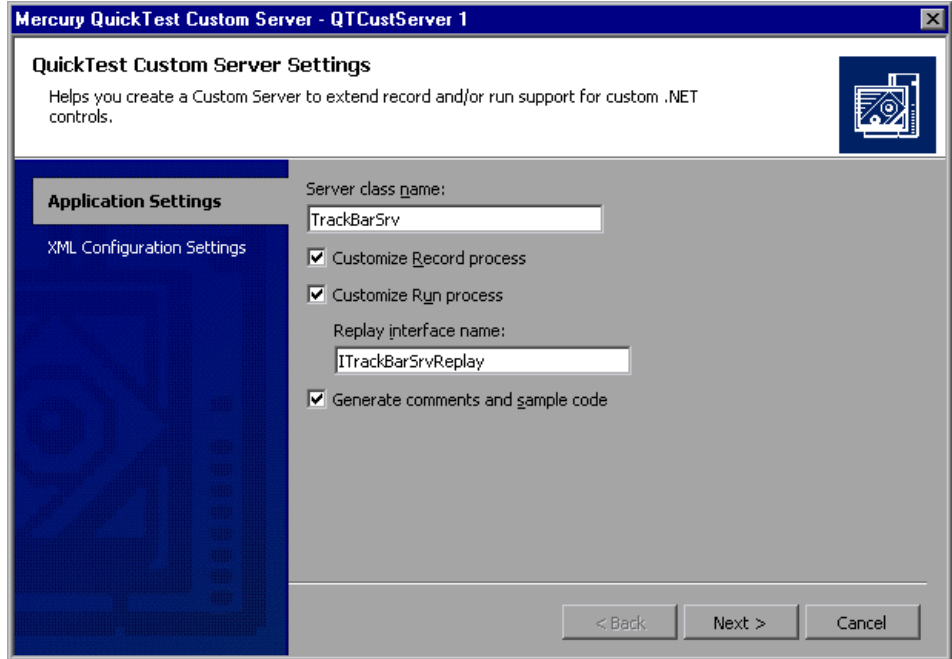
To create a new Custom Server project:

- 1 Open Microsoft Visual Studio .NET.
- 2 Select **File > New > Project**. The New Project dialog box opens.



- 3 Specify the following settings:
 - ▶ Select **Visual C# Projects** in the **Project Types** pane.
 - ▶ Select **QuickTest CustomServer** in the **Templates** pane.
 - ▶ In the **Name** box, specify the project name QTCustServer.
 - ▶ In the **Location** box, specify the location in which to save your project.
 - ▶ Accept the rest of the default settings.

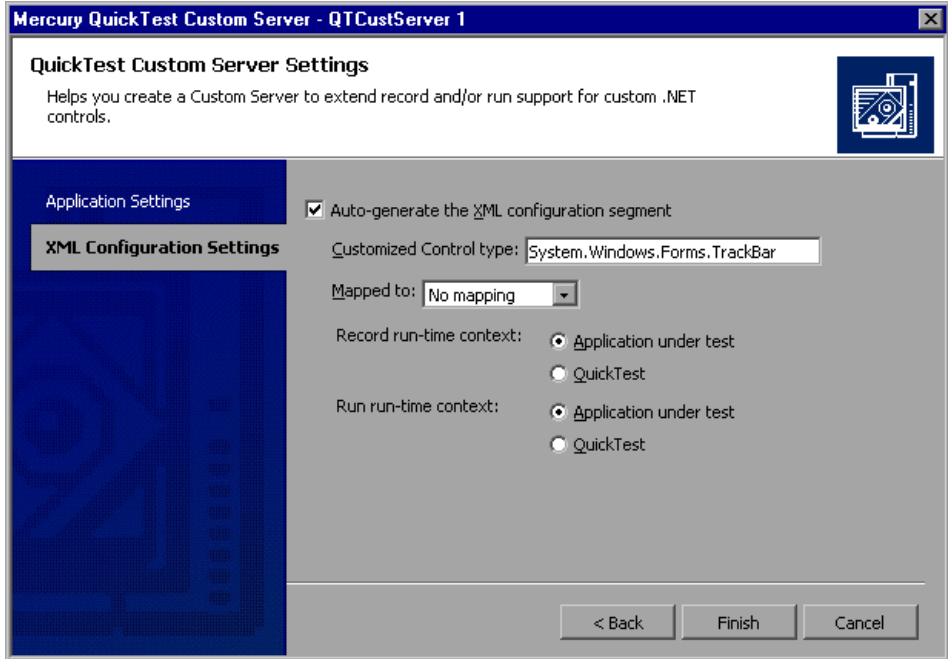
4 Click **OK**. The QuickTest Custom Server Settings wizard opens.



5 In the Application Settings page, specify the following settings:

- In the **Server class name** box, enter TrackBarSrv.
- Select the **Customize Record process** check box.
- Select the **Customize Run process** check box.
- Accept the rest of the default settings.

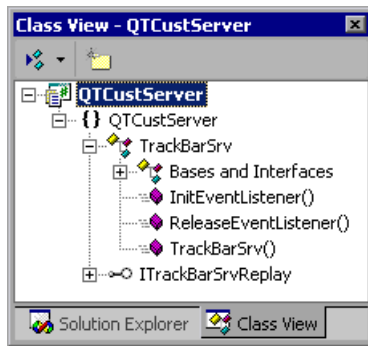
6 Click **Next**. The XML Configuration Settings page opens.



7 In the XML Configuration Settings page, specify the following settings:

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

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



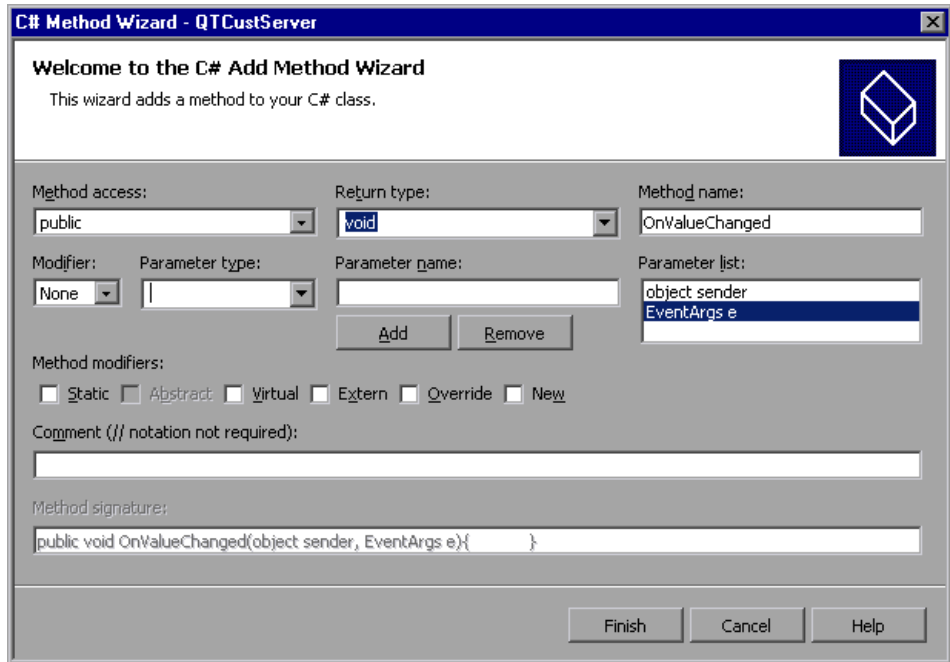
Implementing Test Record Logic

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

To implement the Test Record logic:

- 1 In the Class View window, right-click the **TrackBarSrv** class name and select **Add > Add Method**.

The C# Method Wizard opens.



- 2 Use the C# Method Wizard to add a new method with the following signature:

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

Note: Alternatively, you can add the new method manually without using the C# Method Wizard.

- 3** Add the following implementation to the function you just added:

```
public void OnValueChanged(object sender, EventArgs e)
{
    System.Windows.Forms.TrackBar trackBar =
    (System.Windows.Forms.TrackBar)sender;
    // get the new value
    int newValue = trackBar.Value;
    // Record SetValue command to the test
    RecordFunction("SetValue", RecordingMode.RECORD_SEND_LINE,
    newValue);
}
```

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

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

Implementing Test Run Logic

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

To implement the Test Run logic:

- 1** Add the following method definition to the **ITackBarSrvReplay** interface:

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

- 2 Add the following method implementation to the **TrackBarSrv** class:

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

- 3 Build your project.

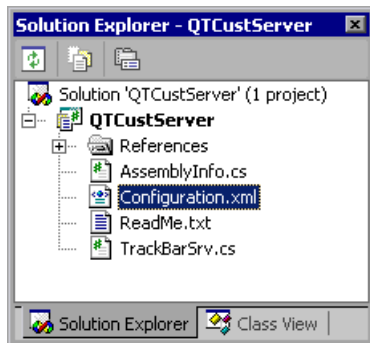
Note: You can see the full source code of the **TrackBarSrv** class in “Understanding the TrackBarSrv.cs File” on page 71.

Configuring QuickTest Professional

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

To configure QuickTest Professional to use the Custom Server:

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



The following content should be displayed:

```
<!-- Merge this XML content into file "<QuickTest Professional>\dat\
SwfConfig.xml". -->
<Control Type="System.Windows.Forms.TrackBar">
  <CustomRecord>
    <Component>
      <Context>AUT</Context>
      <DllName>D:\Projects\QTCustServer\Bin\QTCustServer.dll
      </DllName>
      <TypeName>QTCustServer.TrackBarSrv</TypeName>
    </Component>
  </CustomRecord>
  <CustomReplay>
    <Component>
      <Context>AUT</Context>
      <DllName>D:\Projects\QTCustServer\Bin\QTCustServer.dll
      </DllName>
      <TypeName>QTCustServer.TrackBarSrv</TypeName>
    </Component>
  </CustomReplay>
  <!--<Settings>
    <Parameter Name="sample name">sample value</Parameter>
  </Settings> -->
</Control>
```

- 2** Select the `<Control>...</Control>` segment and select **Edit > Copy** from the menu.
- 3** Open the **SwfConfig.xml** file located in `<QuickTest Professional installation folder>\dat`.

- 4 Paste the `<Control>...</Control>` segment you copied from **Configuration.xml** into **SwfConfig.xml**, under the `<Controls>` tag in **SwfConfig.xml**. After you paste the segment, the **SwfConfig.xml** file should look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Controls>
  <Control Type="System.Windows.Forms.TrackBar">
    <CustomRecord>
      <Component>
        <Context>AUT</Context>
        <DllName>D:\Projects\QTCustServer\Bin\QTCustServer.dll
          </DllName>
        <TypeName>QTCustServer.TrackBarSrv</TypeName>
      </Component>
    </CustomRecord>
    <CustomReplay>
      <Component>
        <Context>AUT</Context>
        <DllName>D:\Projects\QTCustServer\Bin\QTCustServer.dll
          </DllName>
        <TypeName>QTCustServer.TrackBarSrv</TypeName>
      </Component>
    </CustomReplay>
  </Control>
</Controls>
```

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

Testing the Custom Server

You can now verify that QuickTest records and runs tests as expected on the custom `TrackBar` control.

To test the Custom Server:

- 1** Open QuickTest Professional with the .NET Add-in loaded.
- 2** Start recording on a .NET application with a `System.Windows.Forms.TrackBar` control.
- 3** Click the **TrackBar** control. QuickTest should record commands such as:
`SwfWindow("Form1").SwfObject("trackBar1").SetValue 2`
- 4** Run the test. The `TrackBar` control should receive the correct values.

Understanding the `TrackBarSrv.cs` File

Following is the full source code for the `TrackBarSrv` class.

```
using System;
using Mercury.QTP.CustomServer;

namespace QTCustServer
{
    [ReplayInterface]
    public interface ITrackBarSrvReplay
    {
        void SetValue(int newValue);
    }
    public class TrackBarSrv:
        CustomServerBase,
        ITrackBarSrvReplay
    {
        public TrackBarSrv()
        {
        }
    }
}
```

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

public override void ReleaseEventListener()
{
}

public void OnValueChanged(object sender, EventArgs e)
{
    System.Windows.Forms.TrackBar trackBar =
        (System.Windows.Forms.TrackBar)sender;
    int newValue = trackBar.Value;
    RecordFunction("SetValue",
        RecordingMode.RECORD_SEND_LINE,
        newValue);
}

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

8

Tutorial - Advanced Example

In this tutorial, you will learn how to build a Custom Server for controls that require more complex implementation solutions, so that QuickTest Professional can record and run operations on these controls.

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

This chapter describes:	On page:
Toolbar Example	73
Understanding the ToolBarSrv.cs File	81

Toolbar Example

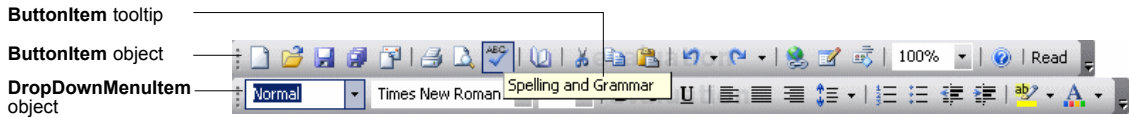
This example demonstrates how to implement .NET extensibility for the Divelements Limited **TD.Sandbar.Toolbar** control.

You can view the full source code of the final **ToolBarSrv.cs** class implementation in “Understanding the ToolBarSrv.cs File” on page 81.

Tip: You can download an evaluation copy of the **TD.Sandbar.Toolbar** control from:

<http://www.divil.co.uk/net/download.aspx?product=2&license=5>

The **Toolbar** control appears as follows:



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

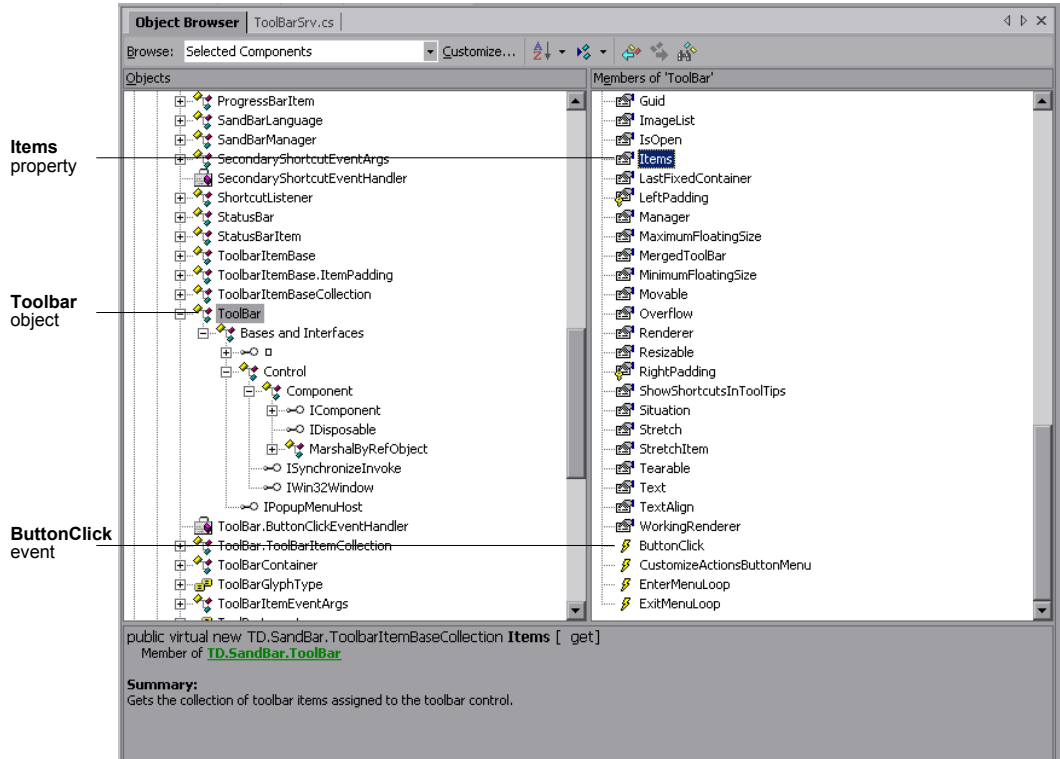
- ▶ **ButtonItem** objects, which represent buttons in the toolbar. **ButtonItem** objects contain images and no text. Each **ButtonItem** object has a unique tooltip.
- ▶ **DropDownMenuItem** objects, which represent drop-down menus in the toolbar.

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

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

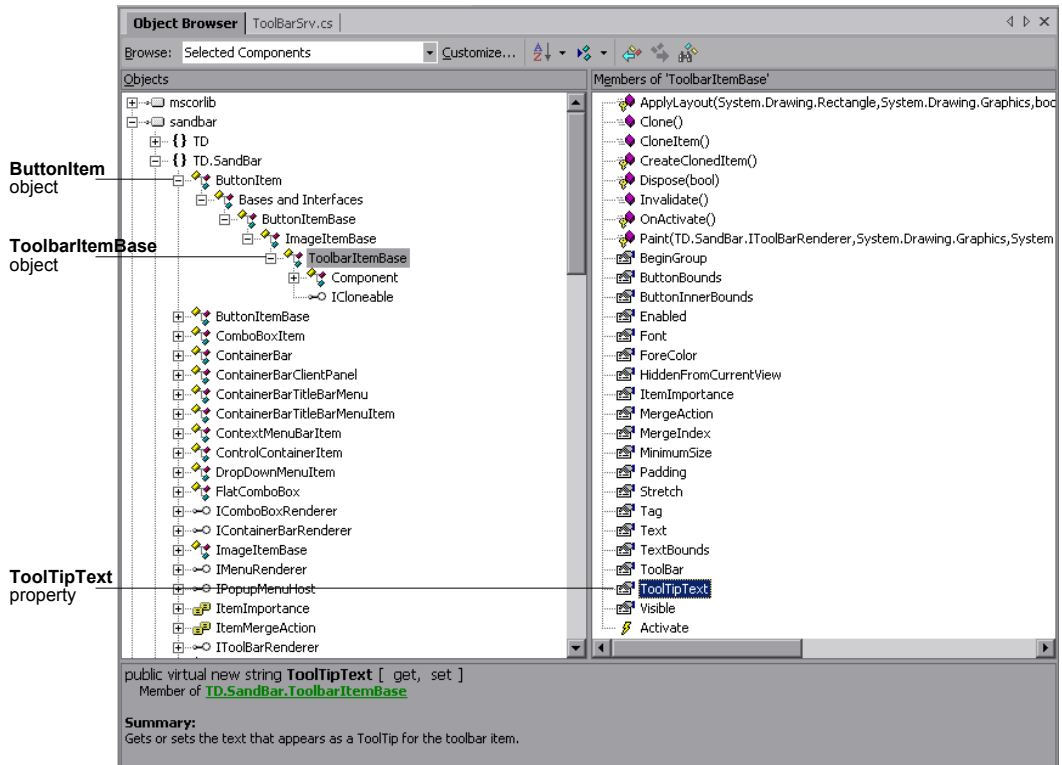
For example, suppose you want to implement a user pressing a button on a custom toolbar. Before doing so, you must understand the toolbar object, its properties, and methods, and understand how you can use them to implement the Custom Server.

Following are the SandBar **ToolBar** object's properties, methods, and events:



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

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



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

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

Issue 1: When recording user actions, when you handle the **ButtonClick** event, how can you recognize which button in the toolbar was actually clicked?

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

The screenshot shows the Visual Studio Object Browser for the file `ToolBarSrv.cs`. The left pane, titled "Objects", displays a tree view of the assembly's types. The `ToolBarItemEventArgs` class is highlighted, and a line points to it from the label "ToolBarItemEventArgs object". Below it, the `EventArgs` class is also visible, with a line pointing to it from the label "EventArgs object". The right pane, titled "Members of 'ToolBarItemEventArgs'", shows the `Item` property. A line points to this property from the label "Item property". At the bottom of the window, the following code is displayed:

```
public TD.SandBar.ToolBarItemBase Item [ get ]
    Member of TD.SandBar.ToolBarItemEventArgs

Summary:
The toolbar item that was activated.
```

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

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

```
#region Record events handlers
private void oControl_ButtonClick(object sender,
TD.SandBar.ToolBarItemEventArgs e)
{
    TD.SandBar.ToolBar oControl = (TD.SandBar.ToolBar)SourceControl;

    //Add a step in the test for the test object with the ClickButton method and the
    tooltip text as an argument
    base.RecordFunction("ClickButton",
        RecordingMode.RECORD_SEND_LINE, e.Item.ToolTipText);
}
#endregion
```

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

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


Issue 2: When running the test (replaying the user's actions), how do you perform a step that contains a **ClickButton** method, but the **ButtonItem** object does not contain a **Click** method or event, and you know only the **ButtonItem** object's tooltip text?

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

The screenshot shows the Visual Studio Object Browser for the file `ToolBarSrv.cs`. The left pane, titled "Objects", displays a tree view of classes and properties. The `ToolBarItemBase` class is selected and highlighted. The right pane, titled "Members of 'ToolBarItemBase'", lists various methods and properties. The `ButtonBounds` property is highlighted in blue. Below the panes, the code signature for the property is shown: `public System.Drawing.Rectangle ButtonBounds [get]`, with a note that it is a member of `ID.SandBar.ToolBarItemBase`. A "Summary" section below the code states: "The boundaries of this toolbar item, in client coordinates." Two lines of text on the left side of the image, "ButtonBounds property" and "ToolBarItemBase object", have arrows pointing to the `ButtonBounds` property in the right pane and the `ToolBarItemBase` class in the left pane, respectively.

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

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

```
#region Replay interface implementation
public void ClickButton(string text)
{
    TD.SandBar.ToolBar oControl = (TD.SandBar.ToolBar)SourceControl;

    //Find the correct item in the toolbar according to its tooltip text.
    for(int i=0; i<oControl.Items.Count; i++)
    {
        //Found the correct ButtonItem
        if(oControl.Items[i].ToolTipText == text)
        {
            //Retrieve the rectangle of the button's boundaries and locate its center
            System.Drawing.Rectangle oRect = oControl.Items[i].ButtonBounds;
            int x = oRect.X + oRect.Width/2;
            int y = oRect.Y + oRect.Height/2;

            //Click the middle of the button item
            base.MouseClick(x, y, MOUSE_BUTTON.LEFT_MOUSE_BUTTON);
            break;
        }
    }

    //Add the step to the report
    base.ReplayReportStep("ClickButton",
        EventStatus.EVENTSTATUS_GENERAL, text);
}
#endregion
```

Understanding the ToolBarSrv.cs File

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

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

namespace ToolBar
{
    [ReplayInterface]
    public interface IToolBarSrvReplay
    {
        void ClickButton(string text);
    }
    /// <summary>
    /// Summary description for ToolBarSrv.
    /// </summary>
    public class ToolBarSrv:
        CustomServerBase,
        IToolBarSrvReplay
    {
        // You shouldn't call Base class methods/properties at the constructor
        // since its services are not initialized yet.
        public ToolBarSrv()
        {
            //
            // TODO: Add constructor logic here
            //
        }
        #region IRecord override Methods
        #region Wizard generated sample code (commented)
        /// <summary>
        /// To change Window messages filter, implement this method.
        /// The default implementation is to get only the control's
        /// Windows messages.
        /// </summary>
```

```

public override WND_MsgFilter GetWndMessageFilter()
{
    return(WND_MsgFilter.WND_MSGS);
}

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

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

/// <summary>
/// At the end of the Record process, this method is called by QuickTest to
/// release all the handlers the user added in the InitEventListener method.
/// Note that handlers added via QuickTest methods are released by
/// the QuickTest infrastructure.
/// </summary>

```

```

public override void ReleaseEventListener()
{
    TD.SandBar.ToolBar oControl = (TD.SandBar.ToolBar)SourceControl;
    oControl.ButtonClick -= new
    TD.SandBar.ToolBar.ButtonClickEventHandler(oControl_ButtonClick);
}
#endregion

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

#region Replay interface implementation
public void ClickButton(string text)
{
    TD.SandBar.ToolBar oControl = (TD.SandBar.ToolBar)SourceControl;
    //Find the correct item in the toolbar according to its tooltip text.
    for(int i=0; i<oControl.Items.Count; i++)
    {
        //Found the correct ButtonItem
        if(oControl.Items[i].ToolTipText == text)
        {
            // Retrieve the rectangle of the button's boundaries and
            // locate its center
            System.Drawing.Rectangle oRect=oControl.Items[i].ButtonBounds;
            int x = oRect.X + oRect.Width/2;
            int y = oRect.Y + oRect.Height/2;
            //Click the middle of the button item
            base.MouseClick(x, y, MOUSE_BUTTON.LEFT_MOUSE_BUTTON);
            break;
        }
    }
}
}

```

```
        //Add the step to the report
        base.ReplayReportStep("ClickButton",
                               EventStatus.EVENTSTATUS_GENERAL, text);
    }
#endregion
}
}
```

Index

A

API overview 39
Application under test run-time context 18,
38
Assistant classes 19, 38

C

C# project template
installing 21
uninstalling 24
coding options 17
Component, Configuration XML tag 57
Configuration XML file 56
example 59
configuring the Custom Server 55
Context, Configuration XML tag 57, 58
Control Definition XML file
example 45
explaining 42
control events 38
Control, Configuration XML tag 56
Controls, Configuration XML tag 56
conventions. *See* typographical conventions
Cross-Process methods 40
Custom Server 14, 16
configuring 55
installing template 21
mapping 20, 30, 36
uninstalling template 24
CustomAssistantBase class 38
Customization, Control Definition XML tag
44
customizing
Test Record 28
Test Run 29
XML configuration 29

CustomRecord, Configuration XML tag 57
CustomReplay, Configuration XML tag 57

D

DllName, Configuration XML tag 58
documentation updates 11

E

enabled, Event Control Definition XML tag
attribute 44
Event, Control Definition XML tag 44
Events, Control Definition XML tag 44
EventsListenerBase class 32

I

installation, requirements 22
installing Custom Server template 21
InstWizard.msi file 22
IRecord interface 31

K

Knowledge Base 10

L

listening to control events 38

M

MappedTo, Control Configuration XML tag
attribute 57
mapping 20, 30, 36
Mercury Best Practices 11
Mercury Customer Support Web site 10

Index

Mercury Home Page 10
Method, Control Definition XML tag 45
MethodBody, Control Definition XML tag 45
methods
 Cross-Process 40
 Test Record 39
 Test Record Callback 39
 Test Run 39
Methods, Control Definition XML tag 45

N

name
 Event Control Definition XML tag attribute 44
 Method Control Definition XML tag attribute 45
 Parameter Control Definition XML tag attribute 45
 RecordedCommand Control Definition XML tag attribute 44
Name, Parameter Configuration XML tag attribute 57
.NET DLL Custom Server
 creating 26
 introduction 25
.NET Windows Forms Spy 16

O

online documentation 8
online resources 10

P

Parameter
 Configuration XML tag 57
 Record element Control Definition XML tag 44
 Run element Control Definition XML tag 45
Parameters, Control Definition XML tag 45

Q

QuickTest Custom Server 22
QuickTest Custom Server Wizard 28
QuickTest run-time context 18
QuickTest, configuring to use Custom Server 55

R

Readme 8
Record
 Control Definition XML tag 44
 customizing 28
 implementing with a .NET DLL 31
RecordedCommand, Control Definition XML tag 44
Replay, Control Definition XML tag 45
ReplayInterface 37
requirements, installation 22
Run
 customizing 29
 implementing with a .NET DLL 36
run-time contexts 18
 Application under test 18, 30
 guidelines 19
 QuickTest 18, 30

S

Send Feedback 10
Settings, Configuration XML tag 57
SwfConfig.xml file 56
SwfObject 37
SwfObject test object 20

T

template
 installing Custom Server 21
 uninstalling Custom Server 24
test object mapping 20, 30, 36
test object methods, writing to the test 36
Test Record
 customizing 28
 implementing with a .NET DLL 31
Test Record Callback methods 39

- Test Record methods 39
- Test Run
 - customizing 29
 - implementing with a .NET DLL 36
- Test Run methods 39
- tutorial 61
- Type, Control Configuration XML tag
 - attribute 57
- type, Parameter Control Definition XML tag
 - attribute 45
- TypeName, Configuration XML tag 58
- typographical conventions 12

U

- uninstalling Custom Server template 24
- updates, documentation 11

X

- XML configuration, customizing 29
- XML Custom Server 41, 47
- XML file, configuration 59

