OPTIMIZE

MERCURY QUICKTEST PROFESSIONAL™

JAVA™ ADD-IN

VERSION 9.1

Extensibility Developer's Guide

MERCURY™

BUSINESS TECHNOLOGY OPTIMIZATION

Mercury QuickTest Professional Java™ Add-in

Extensibility Developer's Guide

Version 9.1

Document Release Date: August 1, 2006

MERCURY™

Mercury QuickTest Professional Java Add-in Extensibility Developer’s Guide, Version 9.1

Mercury Interactive Corporation
379 North Whisman Road
Mountain View, CA 94043
Tel: (650) 603-5200
Toll Free: (800) TEST-911
Customer Support: (877) TEST-HLP
Fax: (650) 603-5300

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

QTPJAVAEXTDG9.1/01

Table of Contents

Welcome to This Guide

Welcome to QuickTest Professional Java Add-in Extensibility.

QuickTest Professional Java Add-in Extensibility is an SDK package that enables you to support testing applications that use third-party and custom Java controls that are not supported out-of-the-box by the QuickTest Professional Java Add-in.

The QuickTest Professional Java Add-in Extensibility SDK provides:

- an API that enables you to extend the QuickTest Professional Java Add-in to support custom Java controls.
- a plug-in for the Eclipse Java development environment, which provides wizards and commands that help you create and edit custom toolkit support sets.
- a complete Java Add-in Extensibility documentation set including an API reference, a toolkit configuration file schema Help, and this Developer's Guide.
- a set of sample applications and completed Java Add-in Extensibility projects that extend support for these applications.

This chapter describes:	**On page:**
How This Guide Is Organized	viii
Who Should Read This Guide	ix
QuickTest Professional Online Documentation	x
Additional Online Resources	xii

This chapter describes:	On page:
Documentation Updates	xiii
Typographical Conventions	xiv

How This Guide Is Organized

This guide explains everything you need to know to use QuickTest Professional Java Add-in Extensibility to extend QuickTest support for third-party and custom Java controls.

This guide should be used together with the *QuickTest Professional Java Add-in Extensibility API Reference*, the *QuickTest Test Object Schema*, and the *QuickTest Java Add-in Extensibility Toolkit Configuration Schema* (provided in online Help format). These documents should also be used in conjunction with the *QuickTest Professional User's Guide,* the *QuickTest Professional Java Add-in Guide,* and the *QuickTest Professional Object Model Reference*. All of these guides (including this Developer's Guide) can be accessed online by choosing **Help** > **QuickTest Professional Help** from the QuickTest main window.

This guide contains:

Part I **Working with Java Add-in Extensibility**

Explains how to use QuickTest Professional Java Add-in Extensibility to extend QuickTest support for custom Java controls. This part includes:

➤ Introducing QuickTest Professional Java Add-in Extensibility

➤ Installing the QuickTest Professional Java Add-in Extensibility Software Development Kit

➤ Implementing Custom Toolkit Support

➤ Planning Custom Toolkit Support

➤ Using the QuickTest Java Add-in Extensibility Eclipse Plug-In

Part II Tutorial: Learning to Create Java Custom Toolkit Support

Guides you step-by-step through the process of creating custom support for some sample controls. This part includes:

- Using the QuickTest Java Add-in Extensibility Tutorial
- Learning to Support a Simple Control
- Learning to Support a Custom Static-Text Control
- Learning to Support a Complex Control

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 for Java custom controls.

To use this guide, you should be familiar with:

- Major QuickTest features and functionality
- QuickTest Professional Object Model
- QuickTest Professional Java Add-in
- XML (basic knowledge)
- Java Programming

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** 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.

Choose **Help** > **QuickTest Professional Help**. Online Help is also available from specific QuickTest windows and dialog boxes by clicking in the window and pressing F1. You can also 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 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**. 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. In the Customer Support Web site, click the **Documentation** link.
2. 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.
3. 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.
4. Click a document link to download the documentation.

Typographical Conventions

This guide uses the following typographical conventions:

UI Elements	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."
Arguments	This style indicates method, property, or function arguments and book titles. For example, "Refer to the *Mercury User's Guide*."
<**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."
Function_Name	This style indicates method or function names. For example, "The **wait_window** statement has the following parameters:"
[ ]	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.

Part I

Working with Java Add-in Extensibility

1

Introducing QuickTest Professional Java Add-in Extensibility

Welcome to QuickTest Professional Java Add-in Extensibility.

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

About QuickTest Professional Java Add-in Extensibility

The QuickTest Professional Java Add-in provides built-in support for a number of commonly used Java objects. You use QuickTest Professional Java Add-in Extensibility to extend that support and enable QuickTest to recognize additional Java controls.

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

QuickTest can learn Java controls that are not supported out-of-the-box by the Java Add-in without using Extensibility. However, when QuickTest learns a Java control that is not supported, it recognizes the control as a generic Java test object. This type of Java test object might not have certain characteristics that are specific to the Java control. Therefore, when you try to create test steps with this test object, the available identification properties and test object methods might not be sufficient.

For example, consider a custom control that is a special type of button that QuickTest recognizes as a plain JavaObject. JavaObject test objects do not support simple **Click** operations. The **JavaObject.Click** method requires the coordinates of the click as arguments. To create a test step that clicks this custom control, you would have to calculate the button's location and provide the coordinates for the click.

By creating support for a Java control using Java Add-in Extensibility, you can direct QuickTest to recognize the control as belonging to a specific test object class, and you can specify the behavior of the test object. You can also extend the list of available test object classes that QuickTest is able to recognize. This enables you to create tests that fully support the specific behavior of your custom Java controls.

Identifying the Building Blocks of Java Add-in Extensibility

The sections below describe the main elements that comprise QuickTest object support. These elements are the building blocks of Java Add-in Extensibility. By extending the existing support of one or more of these elements, you can create the support you need to create meaningful and maintainable tests.

Test Object Classes

In QuickTest, every object in an application is represented by a test object of a specific test object class. The Java Add-in maps each supported class to a specific test object class. QuickTest determines which test object class to use according to this mapping.

When QuickTest learns a control of a Java class that is not yet supported (a custom class), it selects the test object class to represent the control based on the class inheritance hierarchy. QuickTest searches for the closest ancestor of the class that is supported, and uses the test object class mapped to this class. For example, if the custom class extends **java.awt.Applet**, QuickTest recognizes the control as a **JavaApplet** test object. If the custom class extends the **java.awt.Canvas**, QuickTest recognizes the control as a **JavaObject** test object.

The icon that is used to represent this type of object in QuickTest, for example in the Keyword View and Object Repository, is also determined by the test object class.

Test Object Names

When QuickTest learns an object, it uses data from the object to generate a name for the test object. A descriptive test object name enables you distinguish between test objects of the same class and makes it easier to identify them in your object repository and in tests.

When QuickTest learns a control of a Java class that is not yet supported and therefore uses a test object class mapped to one of its ancestors, the test object name is based on the rules defined for that test object class. In many cases, this is not the ideal name for the custom control.

Test Object Identification Properties

The test object class that is mapped to the Java class determines the list of identification properties for a test object. It also determines which of these identification properties are used to uniquely identify the object, which identification properties are available for checkpoints (in the Checkpoint Properties dialog box), and which are selected by default for checkpoints. However, the actual values of the identification properties are derived from the definition of the custom class. Therefore, several custom classes that are mapped to the same test object may have different definitions for the same identification property.

Test Object Methods

The test object class that is mapped to the Java class determines the list of test object methods for a test object. However, the actual behavior of the test object method depends on the definition of the specific custom class. This means that the same test object method may operate differently for different custom classes that are mapped to the same test object class.

Recording Events

One way to create QuickTest tests is by recording user operations on the application. When you start a recording session, QuickTest listens for events that occur on objects in the application and registers corresponding test steps. Each Java object class defines which events QuickTest can listen for. The Java Add-in determines what test step to record for each event that occurs.

Deciding When to Use Java Add-in Extensibility

The QuickTest Professional Java Add-in provides a certain level of support for every Java control. Before you extend support for a custom Java control, analyze it from a QuickTest perspective to view the extent of this support and to decide which elements of support you need to modify.

When you analyze the custom control, use the Object Spy, Keyword View, Expert View, and the Record option. Make sure you examine each of the elements described in "Identifying the Building Blocks of Java Add-in Extensibility", above.

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

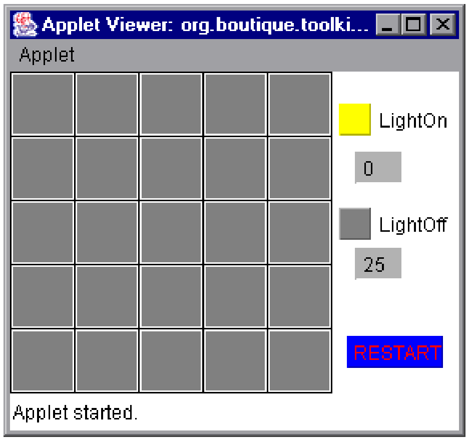
➤ QuickTest might recognize the control using a test object class that does not fit your needs. You can use Java Add-in Extensibility to map the custom class to another existing test object class or to a new test object class that you create.

- The test object class mapped to the control might be satisfactory, but you would like to customize the behavior of certain test object methods or identification properties. You can use Java Add-in Extensibility to override the default implementation of these properties and methods with your own custom implementation.
- You may find that the test object names QuickTest generates for all controls of a certain Java class are identical (except for a unique counter) or that the name used for the control does not clearly indicate the object it represents. You can use Java Add-in Extensibility to modify how QuickTest names test objects for that Java class.
- QuickTest may identify individual sub-controls within your custom control, but not properly identify your main control. For example, if your main custom control is a digital clock with edit boxes containing the hour and minute digits, you might want changes in the time to be recognized as **SetTime** operations on the clock control and not as **Set** operations on the edit boxes. You can use Java Add-in Extensibility to treat a custom control as a **wrapper** object for the controls it contains. QuickTest does not learn the individual controls contained in a wrapper object.
- During a record session, when you perform operations or trigger events on your control, QuickTest may not record a step at all, or it may record steps that are not specific to the control's behavior. Alternatively, QuickTest may record many steps for an event that should be considered a single operation, or it may record a step when no step should be recorded. You can use Java Add-in Extensibility to modify the events to listen for and the test steps to record for specific events.

Analyzing the Default QuickTest Support and Extensibility Options for a Sample Custom Control

The following example illustrates how you can use Java Add-in Extensibility to improve the QuickTest support of a custom control.

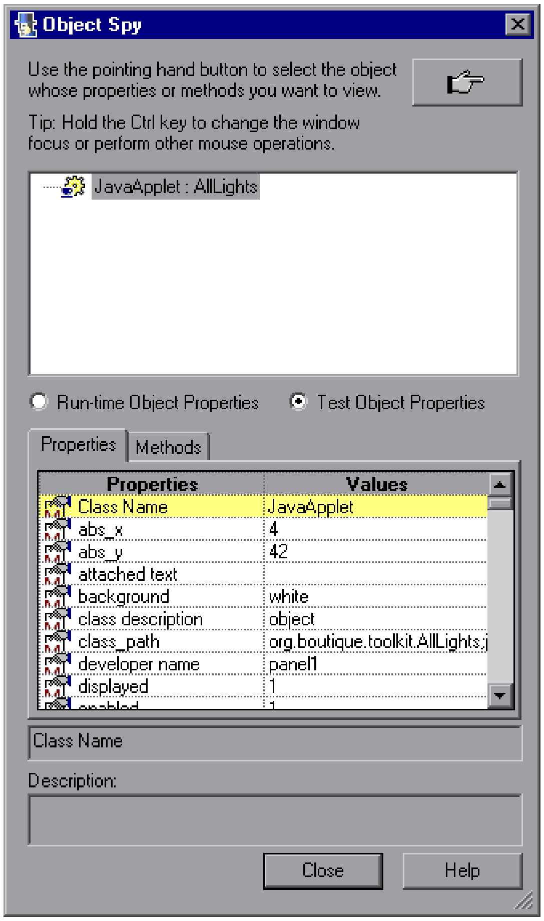
The AllLights control shown below is a game application that is not specifically supported on QuickTest.



This application operates as follows:

- Clicking in the grid area turns different lights on (or off), according to an internal set of rules, and updates the **LightOn** and **LightOff** counters.
- Clicking the **RESTART** button turns off all of the lights. The **LightOn** and **LightOff** counters are updated accordingly.
- Clicking in other areas has no effect.
- The object of the game is to turn on all of the lights, at which point a congratulatory message is displayed.

If you point to this control using the Object Spy, QuickTest recognizes it as a generic JavaApplet named AllLights (the name of the custom class). The icon shown is the standard JavaApplet class icon.


Object Spy

Use the pointing hand button to select the object whose properties or methods you want to view.

Tip: Hold the Ctrl key to change the window focus or perform other mouse operations.

JavaApplet : AllLights

Run-time Object Properties [✓ Test Object Properties]

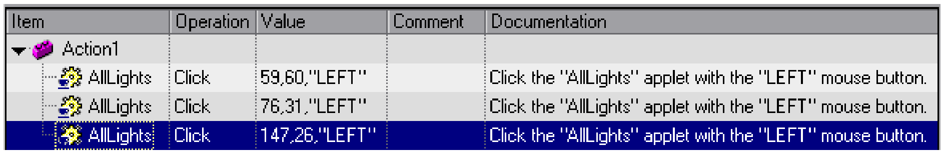
Properties | Methods

Properties	Values
Class Name	JavaApplet
abs_x	4
abs_y	42
attached text	
background	white
class description	object
class_path	org.boutique.toolkit.AllLights;j
developer name	panel1
displayed	1

Class Name

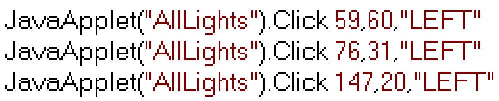
Description:

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



Item	Operation	Value	Comment	Documentation
Action1				
AllLights	Click	59,60,"LEFT"		Click the "AllLights" applet with the "LEFT" mouse button.
AllLights	Click	76,31,"LEFT"		Click the "AllLights" applet with the "LEFT" mouse button.
AllLights	Click	147,26,"LEFT"		Click the "AllLights" applet with the "LEFT" mouse button.

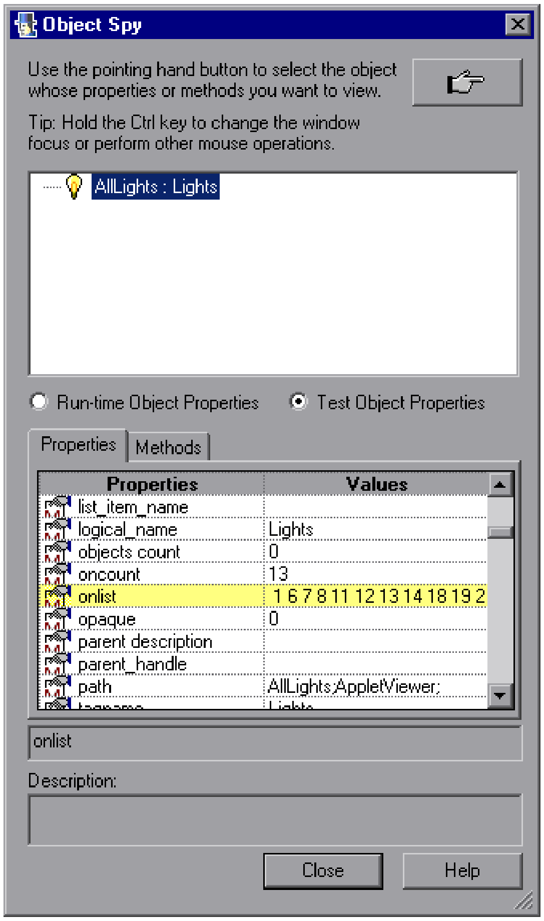
In the Expert View, the recorded test looks like this:



```
JavaApplet("AllLights").Click 59,60,"LEFT"
JavaApplet("AllLights").Click 76,31,"LEFT"
JavaApplet("AllLights").Click 147,20,"LEFT"
```

Note that only generic **Click** steps are recorded, with arguments indicating the low-level recording details (x- and y-coordinates and the mouse button that performed the click). These steps are difficult to understand and modify.

If you use Java Add-in Extensibility to support the AllLights control, the result is more meaningful. QuickTest recognizes the control as an AllLights test object named **Lights** and uses a customized icon. The test object properties include relevant information, such as **oncount** and **onlist**, which provide the total number of all lights that are on at a given moment and their ordinal locations in the grid.



Object Spy

Use the pointing hand button to select the object whose properties or methods you want to view.

Tip: Hold the Ctrl key to change the window focus or perform other mouse operations.

AllLights : Lights

Run-time Object Properties [✓ Test Object Properties]

Properties | Methods

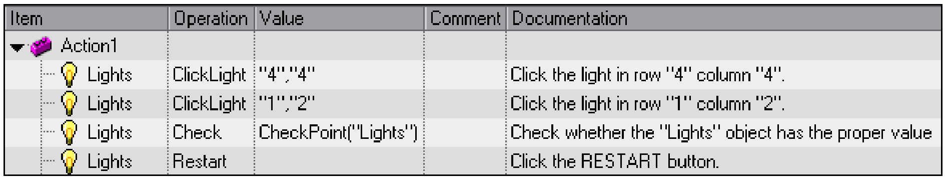
Properties	Values
list_item_name	
logical_name	Lights
objects count	0
oncount	13
onlist	1 6 7 8 11 12 13 14 18 19 2
opaque	0
parent description	
parent_handle	
path	AllLights;AppletViewer;

onlist

Description:

When you are ready to create a test on the control, the **ClickLight** and **Restart** methods are supported. These methods can be recorded or you can select them manually in the Operation column of the Keyword View. You can also create a checkpoint to check the value of identification properties, for example, **gameover** (that indicates whether all lights are on, meaning that you won the game).

In the Keyword View, a test may look like this:



Item	Operation	Value	Comment	Documentation
Action1				
Lights	ClickLight	"4","4"		Click the light in row "4" column "4".
Lights	ClickLight	"1","2"		Click the light in row "1" column "2".
Lights	Check	CheckPoint("Lights")		Check whether the "Lights" object has the proper value
Lights	Restart			Click the RESTART button.

In the Expert View, the test looks like this:

```
AllLights("Lights").ClickLight "4","4"
AllLights("Lights").ClickLight "1","2"
AllLights("Lights").Check CheckPoint("Lights")
AllLights("Lights").Restart
```

This test is easier to understand and modify.

2

Installing the QuickTest Professional Java Add-in Extensibility Software Development Kit

This chapter lists the pre-installation requirements and explains how to install the QuickTest Professional Java Add-in Extensibility SDK.

About Installing the QuickTest Professional Java Add-in Extensibility SDK

The QuickTest Professional Java Add-in Extensibility SDK enables you to design QuickTest support for custom Java controls. The SDK installation includes:

- an API that you can use to create support for custom Java controls
- a plug-in for the Eclipse Java development environment that provides:
    - wizards that guide you through the process of creating custom toolkit support sets

      The Java Add-in Extensibility wizards in Eclipse create all of the required files, classes, and methods. These wizards also provide method stubs for methods that you may need to implement.
    - commands for editing the files after they are created
- a set of sample applications and completed Java Add-in Extensibility projects that extend support for these applications. (The sample applications and their support sets are installed in the **<Java Add-in Extensibility SDK installation folder>\samples** folder.)

Pre-Installation Requirements

Before you install the QuickTest Professional Java Add-in Extensibility SDK:

- Make sure that Eclipse, version 3.1.2, is installed on your computer if you plan to work with the Java Add-in Extensibility Eclipse plug-in. You can download this Eclipse version, free of charge, from http://www.eclipse.org/downloads. (Eclipse 3.1.2 requires Java Development Kit 1.4.2 or later.)

  When you install Eclipse, make sure to note the installation location on your computer. You will need to enter this information when installing the Java Add-in Extensibility SDK.

Note: The Java Add-in Extensibility Eclipse plug-in is required to perform the tutorial described in Part II, "Tutorial: Learning to Create Java Custom Toolkit Support." Additionally, it is recommended to use this plug-in to design at least the skeleton of your toolkit support.

➤ (Optional) Make sure that QuickTest 9.1 with the Java Add-in is installed on the same computer. This enables the Java Add-in Extensibility Eclipse plug-in to interact with QuickTest, enabling you to work more efficiently when debugging and testing your custom toolkit support. For example, if you use the Java Add-in Extensibility Eclipse plug-in on a QuickTest computer, you can deploy the toolkit support to QuickTest for debugging by simply clicking a button.

Note: If you do not install QuickTest and the Java Add-in **before** you install the QuickTest Professional Java Add-in Extensibility SDK, any Java Add-in Extensibility Eclipse plug-in functionality that requires interaction with QuickTest will not be available.

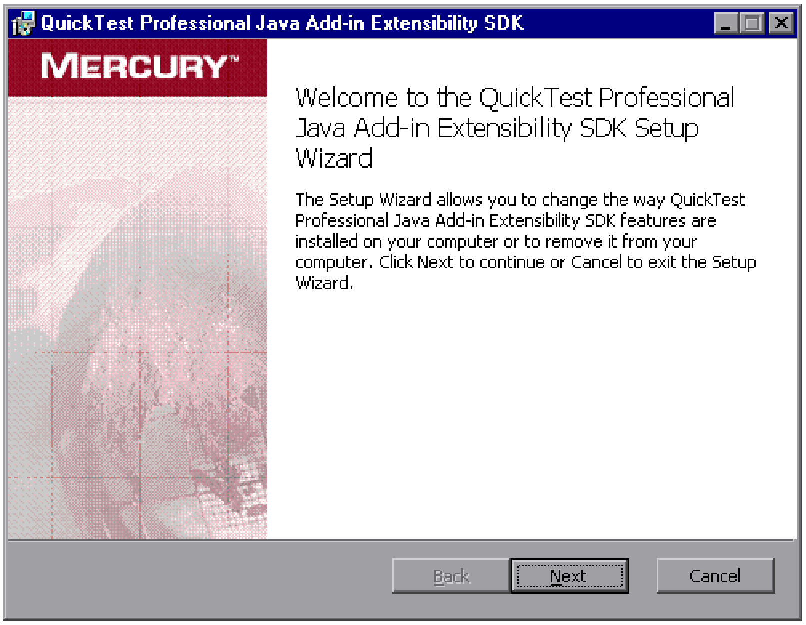
Installing the QuickTest Professional Java Add-in Extensibility SDK

You use the Setup program to install the QuickTest Professional Java Add-in Extensibility SDK on your computer.

To install the QuickTest Professional Java Add-in Extensibility SDK:

1. Close all instances of Eclipse and QuickTest Professional.
2. Insert the QuickTest Professional Java Add-in CD-ROM into the CD-ROM drive and browse to the **Java Add-in Extensibility SDK** folder.

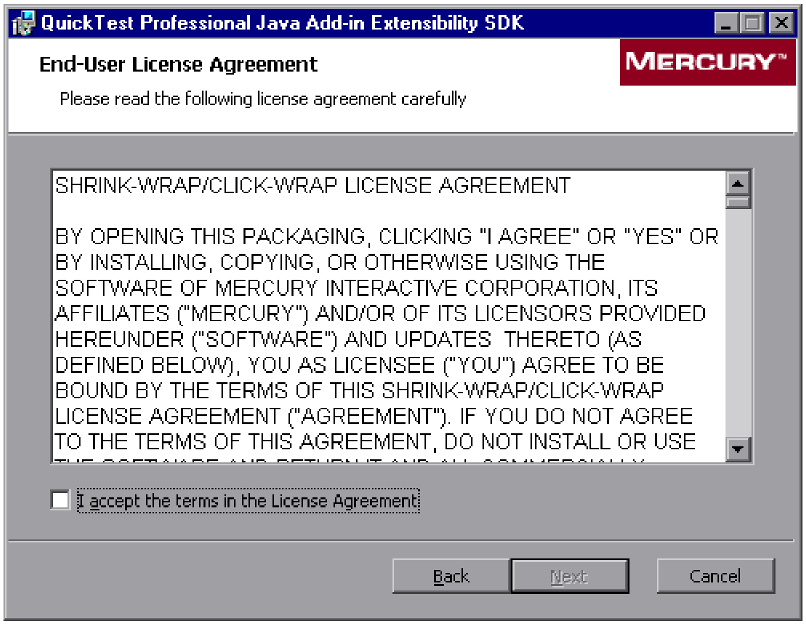
3 Double-click the **setup.msi** file to start the installation.
The Welcome screen of the QuickTest Professional Java Add-in Extensibility SDK Setup Wizard opens.



Welcome to the QuickTest Professional Java Add-in Extensibility SDK Setup Wizard

The Setup Wizard allows you to change the way QuickTest Professional Java Add-in Extensibility SDK features are installed on your computer or to remove it from your computer. Click Next to continue or Cancel to exit the Setup Wizard.

4 Click **Next**. The End-User License Agreement screen opens.


QuickTest Professional Java Add-in Extensibility SDK

End-User License Agreement

Please read the following license agreement carefully

SHRINK-WRAP/CLICK-WRAP LICENSE AGREEMENT

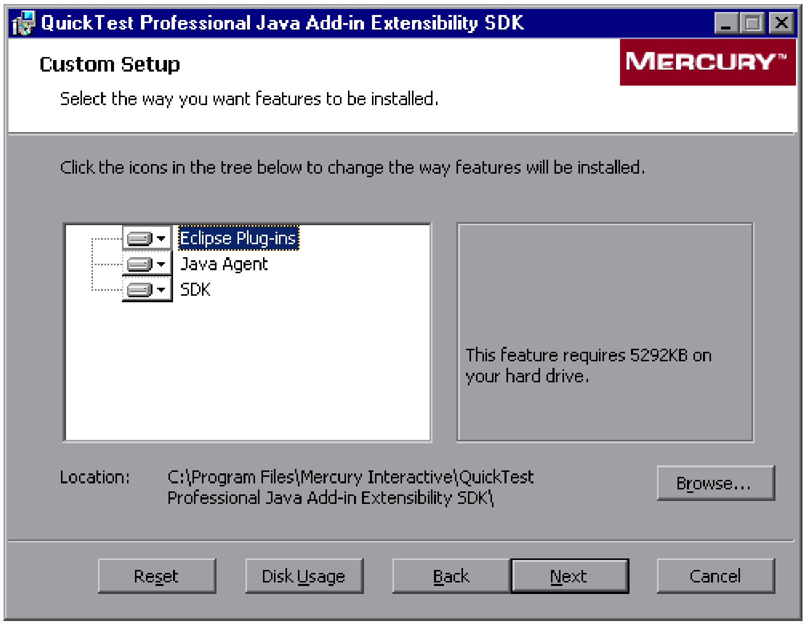
BY OPENING THIS PACKAGING, CLICKING "I AGREE" OR "YES" OR BY INSTALLING, COPYING, OR OTHERWISE USING THE SOFTWARE OF MERCURY INTERACTIVE CORPORATION, ITS AFFILIATES ("MERCURY") AND/OR OF ITS LICENSORS PROVIDED HEREUNDER ("SOFTWARE") AND UPDATES THERETO (AS DEFINED BELOW), YOU AS LICENSEE ("YOU") AGREE TO BE BOUND BY THE TERMS OF THIS SHRINK-WRAP/CLICK-WRAP LICENSE AGREEMENT ("AGREEMENT"). IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DO NOT INSTALL OR USE

☐ I accept the terms in the License Agreement

Note: If the Modify, Repair, or Remove Installation screen opens, the QuickTest Professional Java Add-in Extensibility SDK is already installed on your computer. Before you can install a new version, you must first uninstall the existing one, as described in “Uninstalling the QuickTest Professional Java Add-in Extensibility SDK” on page 22.

Read the license agreement and select the **I accept the terms in the license agreement** check box.

5 Click **Next**. The Custom Setup screen opens.



QuickTest Professional Java Add-in Extensibility SDK

Custom Setup

Select the way you want features to be installed.

Click the icons in the tree below to change the way features will be installed.

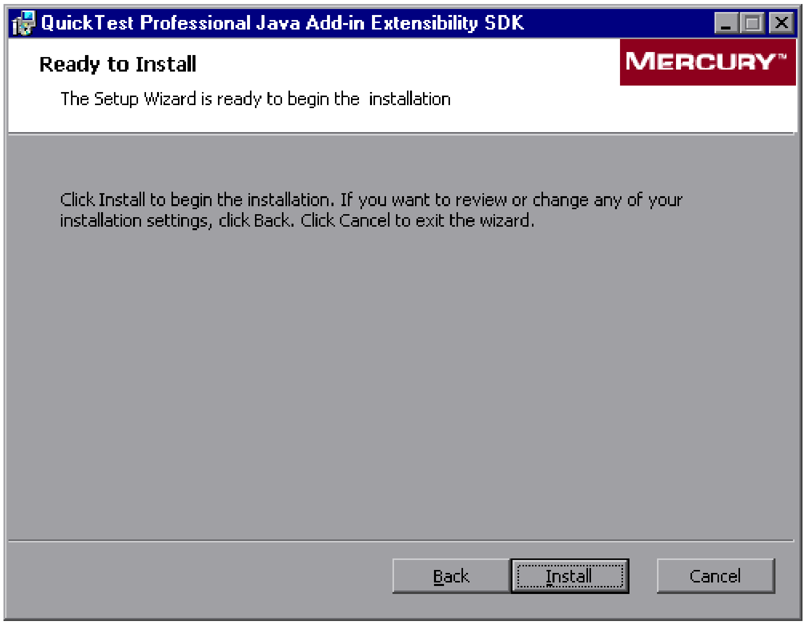
- Eclipse Plug-ins
- Java Agent
- SDK

This feature requires 5292KB on your hard drive.

Location: C:\Program Files\Mercury Interactive\QuickTest Professional Java Add-in Extensibility SDK\

All of the features displayed in the Custom Setup screen are installed automatically during the setup.

6 Click **Next**. The Ready to Install screen opens.

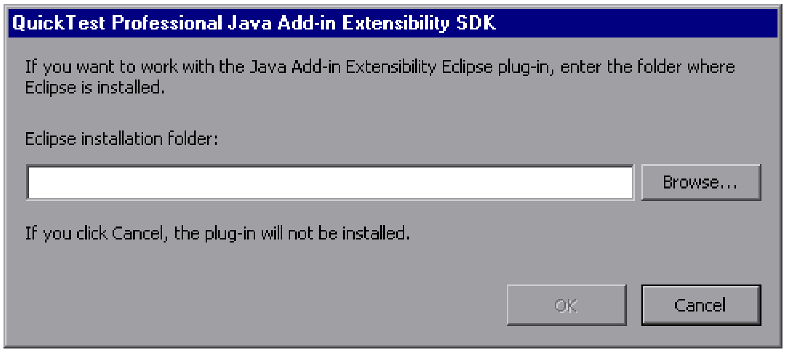


Ready to Install

The Setup Wizard is ready to begin the installation

Click Install to begin the installation. If you want to review or change any of your installation settings, click Back. Click Cancel to exit the wizard.

7 Click **Install**. The Setup program installs the QuickTest Professional Java Add-in Extensibility SDK and displays a dialog box in which you specify the location of the Eclipse installation on your computer.


QuickTest Professional Java Add-in Extensibility SDK

If you want to work with the Java Add-in Extensibility Eclipse plug-in, enter the folder where Eclipse is installed.

Eclipse installation folder:

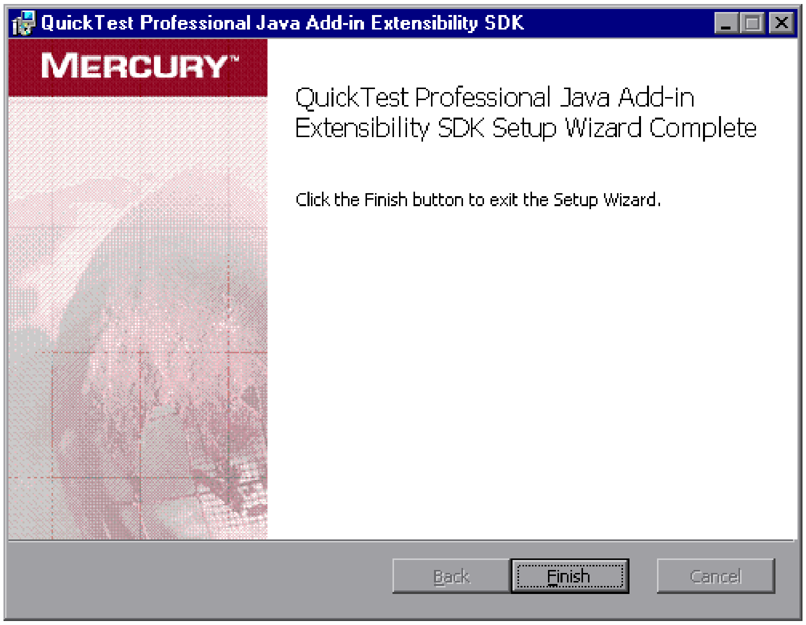
If you click Cancel, the plug-in will not be installed.

The Java Add-in Extensibility Eclipse plug-in is installed on Eclipse according to the location you specify.

Note: You can install the Java Add-in Extensibility Eclipse plug-in on additional Eclipse installations after you finish the QuickTest Professional Java Add-in Extensibility SDK installation process. To do this, browse to the **<QuickTest Professional Java Add-in Extensibility SDL installation folder>\eclipse** folder, and run **deploysdkplugins.exe**. Enter an Eclipse installation folder in the dialog box that opens, and click **OK**.

If you do not plan to use this plug-in, click **Cancel** and proceed to step 8. Otherwise, click **Browse**, navigate to the Eclipse installation folder, and select the root **eclipse** folder. Click **OK**. Then click **OK** to accept the Eclipse installation location.

8 In the final screen, click **Finish** to exit the Setup Wizard.



QuickTest Professional Java Add-in Extensibility SDK Setup Wizard Complete

Click the Finish button to exit the Setup Wizard.

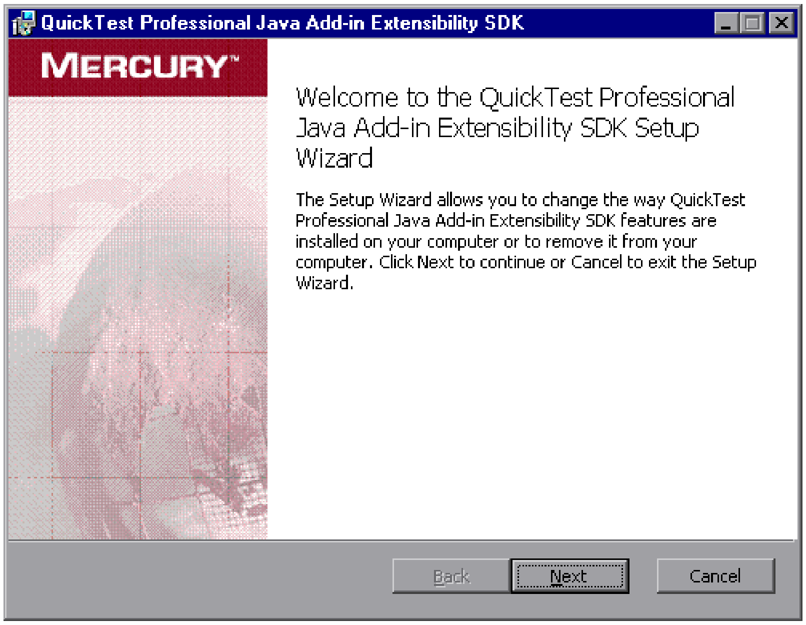
Tip: If you do not see the QuickTest toolbar in Eclipse after the installation, run the command line <Eclipse installation folder>\eclipse -clean on your computer to refresh the Eclipse plug-in configuration, and then re-open Eclipse.

Uninstalling the QuickTest Professional Java Add-in Extensibility SDK

If you no longer need the QuickTest Professional Java Add-in Extensibility SDK, you can remove it from your computer.

To uninstall the QuickTest Professional Java Add-in Extensibility SDK:

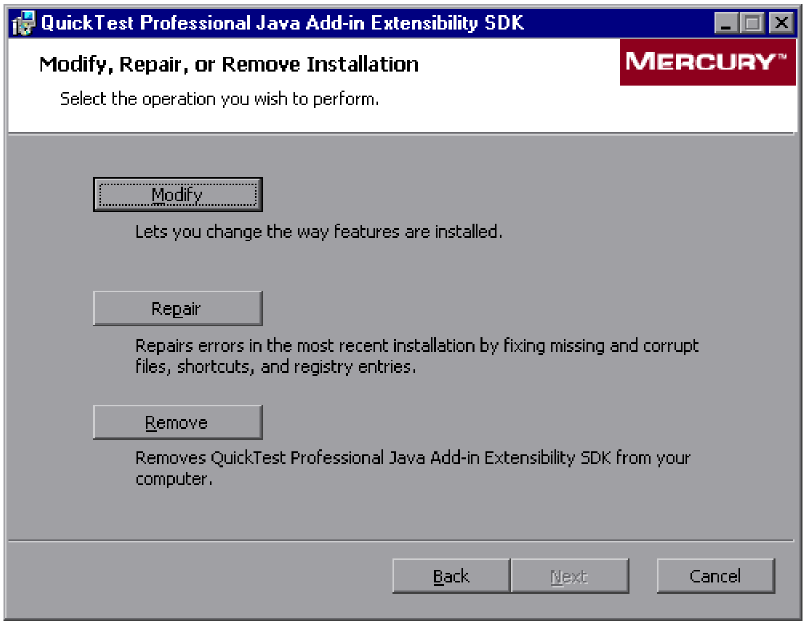
1. Close all instances of Eclipse and QuickTest Professional.
2. Insert the QuickTest Professional Java Add-in CD-ROM into the CD-ROM drive and browse to the **Java Add-in Extensibility SDK** folder.
3. Double-click the **setup.msi** file to run the setup program.
   The Welcome screen of the QuickTest Professional Java Add-in Extensibility SDK Setup Wizard opens.



Welcome to the QuickTest Professional Java Add-in Extensibility SDK Setup Wizard

The Setup Wizard allows you to change the way QuickTest Professional Java Add-in Extensibility SDK features are installed on your computer or to remove it from your computer. Click Next to continue or Cancel to exit the Setup Wizard.

4 Click **Next**. The Modify, Repair, or Remove Installation screen opens.



Modify, Repair, or Remove Installation

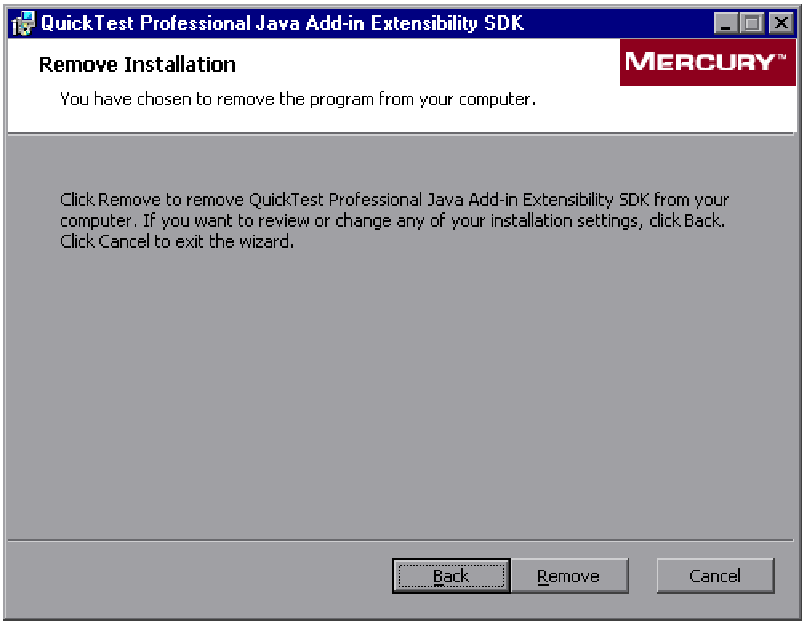
Select the operation you wish to perform.

Lets you change the way features are installed.

Repairs errors in the most recent installation by fixing missing and corrupt files, shortcuts, and registry entries.

Removes QuickTest Professional Java Add-in Extensibility SDK from your computer.

5 Click **Remove**. The Remove Installation screen opens.



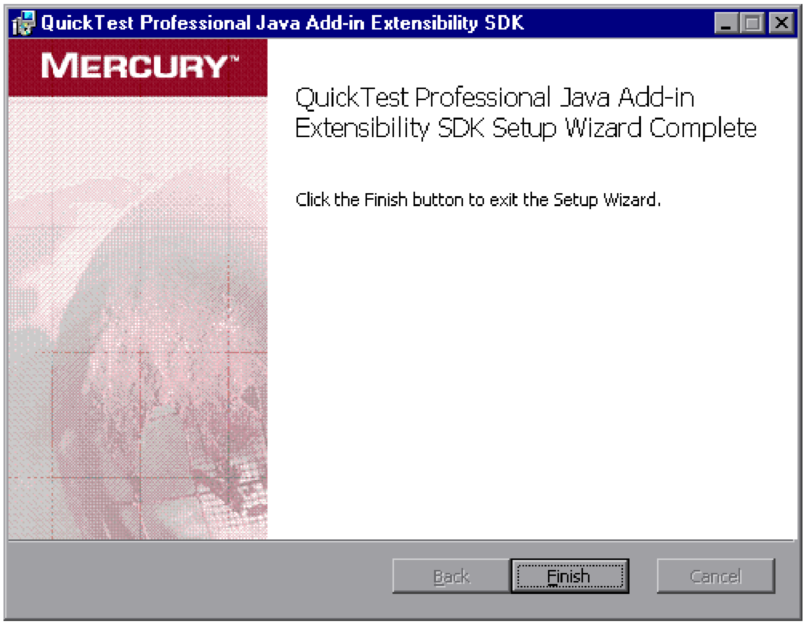
Remove Installation

You have chosen to remove the program from your computer.

Click Remove to remove QuickTest Professional Java Add-in Extensibility SDK from your computer. If you want to review or change any of your installation settings, click Back. Click Cancel to exit the wizard.

6 Click **Remove**. The Setup program removes the QuickTest Professional Java Add-in Extensibility SDK and opens the final Setup Wizard screen.

7 In the final screen, click **Finish** to exit the Setup wizard.



QuickTest Professional Java Add-in Extensibility SDK Setup Wizard Complete

Click the Finish button to exit the Setup Wizard.

Note: When you uninstall the QuickTest Professional Java Add-in Extensibility SDK, the Java Add-in Extensibility Eclipse plug-in is removed from all Eclipse installations.

Tip: If you still see the QuickTest toolbar in Eclipse after uninstalling, run the command line <Eclipse installation folder>\eclipse -clean on your computer to refresh the Eclipse plug-in configuration, and then re-open Eclipse.

3

Implementing Custom Toolkit Support

You implement Java Add-in Extensibility by creating a **custom toolkit support set** for each Java toolkit you want to support. The custom toolkit support set is comprised of Java classes and XML configuration files. The Java classes you create extend existing Java Add-in classes and the support they provide, by overriding their methods and defining new ones.

This chapter explains how to create support for a custom toolkit. It explains what files you have to create for the custom toolkit support set, the structure and content of these files, and where they should be stored.

About Custom Toolkit Support

When you extend QuickTest support of a custom toolkit, you create an API that is based on the existing QuickTest Java Add-in and supplements it. This API, or custom toolkit support set, is composed of Java classes and XML configuration files. It provides an interface between QuickTest and the Java application being tested, enabling QuickTest to identify the Java controls in the application and correctly perform operations on those controls.

This chapter describes the different files, classes, methods, and definitions that you must include in a custom toolkit support set. For more information, refer to the *QuickTest Java Add-in Extensibility API Reference* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility API Reference**).

Note: Before you actually begin to create a custom toolkit support set, you must plan it carefully. For more information, see "Planning Custom Toolkit Support" on page 75.

The QuickTest Professional Java Add-in Extensibility SDK provides a plug-in for the Eclipse Java development environment, which provides wizards that help you create custom toolkit support sets. This plug-in also provides a set of commands that you can use to edit the files after they are created.

When you use the Java Add-in Extensibility wizards to create the custom toolkit support, the wizards create all of the required files, classes, and basic methods. They also provide method stubs for additional methods that you may need to implement.

To gain a better understanding of designing custom toolkit support sets before you begin to design your own, perform the lessons in Part II, "Tutorial: Learning to Create Java Custom Toolkit Support." In these lessons you use the Java Add-in Extensibility wizards in Eclipse to create custom support for sample custom controls.

Even if you do not regularly use Eclipse to develop Java software, it is recommended that you use it for Java Add-in Extensibility, at least for performing the tutorial. It is generally simpler to create the skeleton of the custom toolkit support with the help of the Java Add-in Extensibility wizards than to do it manually. After you have completed this initial stage, you can continue the design of the toolkit support in the development environment of your choice.

For information on setting up Eclipse and the QuickTest Professional Java Add-in Extensibility Eclipse plug-in, as well as using the plug-in, see "Installing the QuickTest Professional Java Add-in Extensibility Software Development Kit" on page 13.

Note: If you choose not use the Java Add-in Extensibility wizards in Eclipse, you can still extend full support for the custom toolkit manually using the information in this chapter.

Introducing Java Add-in Extensibility Terminology

The following terminology, specific to QuickTest Java Add-in Extensibility, is used in this guide:

Native toolkit. A toolkit that implements drawing using native API. Abstract Windows Toolkit (AWT) and Standard Widgets Toolkit (SWT) are native toolkits. Java Foundation Classes (JFC) is not a native toolkit, as it extends AWT.

Basic user interface component. In the AWT toolkit: **java.awt.Component** In the SWT toolkit: **org.eclipse.swt.widgets.Widget**

Custom toolkit. A set of classes, all extending the basic user interface component of the same native toolkit.

Custom class. A Java class that extends **java.awt.Component** or **org.eclipse.swt.widgets.Widget** for which you create QuickTest support.

Custom toolkit support. Extends QuickTest ability to recognize controls in a custom toolkit as test objects, view and check their properties, and run tests on them. (In this guide, custom toolkit support is also referred to as **custom support** or **toolkit support**.)

Preparing to Create Support for a Custom Toolkit

You can extend QuickTest support for any toolkit containing classes that extend **java.awt.Component** or **org.eclipse.swt.widgets.Widget**.

When you create a custom toolkit support set for each custom toolkit, the first step is to determine the set of classes that comprise your custom toolkit. For the purpose of Extensibility, a **custom toolkit** is a set of classes that extend the basic user interface component of the same native toolkit.

This does not prevent you from creating support for a toolkit containing classes that extend **java.awt.Component**, as well as those that extend **org.eclipse.swt.widgets.Widget**. Such a toolkit is simply seen as two separate custom toolkits, and you must create support separately for each set of classes.

Similarly, if you have user interface control classes that extend the basic user interface component of the same native toolkit, and are packaged in separate Java archives or class folders, you can treat them as one custom toolkit. This means you can create a single custom toolkit support set for all those classes.

Within a custom toolkit, you extend QuickTest support for each control (or group of similar controls) separately. You do this by creating **custom support classes** for the different custom control classes in the toolkit. (In this guide, custom support classes are also referred to as **support classes**.)

Before you extend QuickTest support for a custom control make sure you have full access to the control and understand its behavior. You must have an application in which you can view the control in action, and also have access to the class that implements it.

You do not need to modify any of the custom control's sources to support it in QuickTest, but you do need to be familiar with them. Make sure you know which members (fields and methods) you can access externally, the events for which you can listen, and so forth. You use this information when you design the support class. To implement the interface between QuickTest and the custom class, the support class uses custom class members. The support class can only access the members of the custom class that are defined as public.

In addition, you need access to the compiled classes in a Java archive or class folder because you add them to the classpath when compiling the support classes.

Creating a Custom Toolkit Support Set

After you determine the set of custom classes for which you want to extend QuickTest support, you create the custom toolkit support set.

A Java Add-in Extensibility custom toolkit support set comprises the following java classes and XML configuration files:

- One toolkit support class, described on page 35
- One toolkit configuration file, described on page 36
- One or more test object configuration files (if this support set introduces new test object classes or extends existing ones), described on page 38
- Custom support classes (mapped to the custom classes), described on page 43

The Java classes of the custom toolkit support set are packaged in a toolkit root package named **com.mercury.ftjadin.qtsupport.<Custom Toolkit Name>**. Within this package, the custom support classes are stored in a sub-package named **com.mercury.ftjadin.qtsupport.<Custom Toolkit Name>.cs**. The configuration files are stored under the QuickTest installation folder and reference the java packages. For more information, see "Deploying and Running the Custom Toolkit Support" on page 66.

To create a custom toolkit support set:

1. Choose a unique name to represent the custom toolkit.

   You use the custom toolkit name to compose the name of the toolkit support class and its packaging. The name must start with a letter and can contain only alphanumeric characters and underscores.

   Providing unique toolkit names allows a single QuickTest installation to support numerous custom toolkit support sets simultaneously. For this reason, a name such as MyToolkit is not recommended.

2. Create the toolkit root package:
   com.mercury.ftjadin.qtsupport.<Custom Toolkit Name>.

3. Create the toolkit support class in the toolkit root package. Name the class **<Custom Toolkit Name>Support.java**.
   For information on the content of this class, see "Understanding the Toolkit Support Class" on page 35.

4 Create the toolkit configuration file. Name the file: **<Custom Toolkit Name>.xml**.
For information on the content of this file, see "Understanding the Toolkit Configuration File" on page 36.

5 Consider the behavior (fields and methods) of the custom controls, and map the custom controls to a QuickTest test object class. For more information, see "Mapping a Custom Control to a Test Object Class" on page 49.

If you require any new QuickTest test object classes to map to controls in the custom toolkit, create the test object configuration file. Name the file **<Custom Toolkit Name>TestObjects.xml**.

For information on the content of this file and the locations in which to store it, see "Understanding the Test Object Configuration File" on page 38.

Note: In most cases, a custom toolkit support set has only one test object configuration file, named **<Custom Toolkit Name>TestObjects.xml**. However, you could store the definitions for different test object classes in different test object configuration files. You create all of the test object configuration file according to the *QuickTest Test Object Schema Help* (**Help > QuickTest Professional Help > QuickTest Advanced References > QuickTest Test Object Schema**). All of the test object configuration files must be located in the same folders, specified in "Deploying and Running the Custom Toolkit Support" on page 66.

QuickTest loads of all the test object class definitions (from all of the test object configuration files) when it opens, regardless of the custom toolkit for which they were created. This enables you to use the same test object class definitions when supporting different custom toolkits.

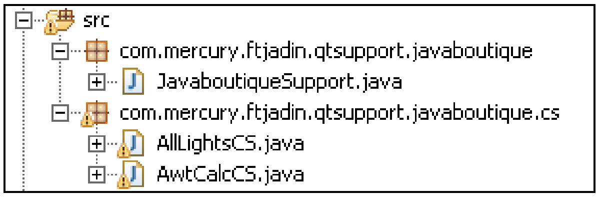
---

6 Create the **com.mercury.ftjadin.qtsupport.<Custom Toolkit Name>.cs** support class sub-package.

7 In the support class sub-package, create the custom support classes for the classes you want to support.

In most cases, you name the custom support class **<Custom Class Name>CS**. If your custom toolkit contains classes from different packages, you might have custom classes with the same name. In this case, you must provide different names for the custom support classes, because they are stored in one package. For information on the content of support classes, see "Understanding Custom Support Classes" on page 43.

The following example illustrates the structure of the java classes in the custom toolkit support set for the custom toolkit named **javaboutique**. Within this toolkit, two custom classes are supported: **AllLights** and **AwtCalc**.


src
com.mercury.ftjadin.qtsupport.javaboutique
JavaboutiqueSupport.java
com.mercury.ftjadin.qtsupport.javaboutique.cs
AllLightsCS.java
AwtCalcCS.java

8 If you develop the custom support using the Java Add-in Extensibility wizard, the wizard defines the required environment variables. If you do not use the wizard, add you must add the following items to the build path (the classpath used by the compiler):

➤ **<Java Add-in Extensibility SDK installation folder>\eclipse\plugins\com.mercury.java.ext.lib_1.0.0\mic.jar**

➤ **<Java Add-in Extensibility SDK installation folder>\eclipse\plugins\com.mercury.java.ext.lib_1.0.0\jacob.jar**

➤ The locations of the compiled custom classes (these locations can be class folders or Java archives)

Note: If, at any time, the custom controls are modified in a way that might affect the support, you should re-compile the support classes, adjusting them if necessary.

Understanding the Toolkit Support Class

Every custom toolkit support set has one toolkit support class that indicates the native toolkit that the custom toolkit extends.

Note: When all of the classes in a custom toolkit extend the basic user interface class of another toolkit (for example **java.awt.Component**) we say the custom toolkit extends that toolkit (in this example: **AWT**).

By extending the custom toolkit support class from the correct native toolkit support set, you ensure that your toolkit inherits all of the necessary utility methods for basic functionality (such as event handling and dispatching).

The QuickTest Professional Java Add-in provides custom toolkit support classes for AWT, SWT, and JFC (Swing). When you create new Java Add-in Extensibility custom toolkit support classes you extend one of these, or the custom toolkit support class of other existing Extensibility custom toolkit support sets.

The inheritance hierarchy of toolkit support classes reflects the hierarchy of the custom toolkits. For example, the JFCSupport class extends the class AWTSupport. A toolkit support class of a toolkit that extends JFC will extend JFCSupport thereby inheriting AWTSupport functionality. No further implementation is required in this class.

For example, this is the toolkit support class for the **Javaboutique** custom toolkit, which extends the **AWT** native toolkit:

```
package com.mercury.ftjadin.qtsupport.javaboutique;
import com.mercury.ftjadin.support.awt.AwtSupport;
public class JavaboutiqueSupport extends AwtSupport {}
```

The following table shows which toolkit support class to extend, if you want to extend the toolkit support classes provided for AWT, SWT, or JFC.

To extend the toolkit support class for:	**Extend:**
AWT	com.mercury.ftjadin.support.awt.AwtSupport
JFC11 (Swing)	com.mercury.ftjadin.support.jfc.JFCSupport
SWT	com.mercury.ftjadin.support.swt.SwtSupport

Understanding the Toolkit Configuration File

Every custom toolkit support set has one toolkit configuration file named **<Custom Toolkit Name>.xml**, which is stored under the QuickTest installation folder. This file provides the information that QuickTest needs to find the classes of the custom toolkit support set.

The toolkit configuration file specifies:

- The location of the toolkit support class
- The location of the compiled support classes (a class folder or Java archive)

  QuickTest adds this location to the Java application classpath when the application runs, enabling the application to find the required support classes.
- The support toolkit description
- A mapping of each custom class to its custom support class

  A single custom support class can be mapped to more than one custom class, but each custom class can be mapped to only one custom support class.

The following example illustrates the configuration file of the **javaboutique** toolkit support, with one supported custom class—**AwtCalc**:

```
<?xml version="1.0" encoding="UTF-8"?>
<Controls
    class="com.mercury.ftjadin.qtsupport.javaboutique.javaboutiqueSupport"
    SupportClasspath="C:\JE\workspace\javaboutiqueSupport\bin"
    description="Javaboutique toolkit support.">
    <Control Type="org.boutique.toolkit.AwtCalc">
        <CustomRecordReplay>
            <ImplementationClass>
                com.mercury.ftjadin.qtsupport.javaboutique.cs.AwtCalcCS
            </ImplementationClass>
        </CustomRecordReplay>
    </Control>
</Controls>
```

For information on the structure and syntax of this file, refer to the *QuickTest Java Add-in Extensibility Toolkit Configuration Schema Help* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility Toolkit Configuration Schema**).

For information on where to store this file, see "Deploying and Running the Custom Toolkit Support" on page 66.

Understanding the Test Object Configuration File

If you map custom controls to new (or modified) test object classes, you must create one or more test object configuration files in the custom toolkit support set. For more information, see "Mapping a Custom Control to a Test Object Class" on page 49.

In the test object configuration files, you define the test object classes (for example, their identification properties, the test object methods they support, and so forth). Each time you run QuickTest, it reads all of the test object configuration files and merges the information for each test object class from the different files into one test object definition. For more information, see "Understanding How QuickTest Merges Test Object Configuration Files" on page 41.

A test object class definition can include:

- the name of the new test object class and its attributes, including the base class—the test object class that the new test object class extends
- the path of the icon file to use for this test object class (Optional. If not defined, the JavaObject icon is used.) The icon file must be in an uncompressed **.ico** format.
- the methods of the new test object, including the following information for each method:
  - the arguments, including the argument type (String or Variant) and direction (In or Out)
  - 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 an object of this class

- the identification properties of the new test object
  - the identification properties that are used for the object description
  - the identification properties that are available for use in checkpoints
  - the identification properties that are selected by default for checkpoints (in QuickTest)

Note: You can also create a definition for an existing test object class in the test object configuration file. This definition is added to the existing definition of this test object class, affecting all objects of this class. It is therefore not recommended to modify existing test object classes in this way.

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 in the specific object's class.

If you add a test object method, it appears in the list of test object methods in QuickTest, but if you use the test object method in a test, and it is not implemented in the specific object's support class, a run-time error occurs.

The following example shows parts of the **Calculator** test object class definition in the **javaboutique** test object configuration file:

```
<ClassInfo BaseClassInfoName="JavaApplet"
    DefaultOperationName="Calculate" Name="Calculator">
    <IconInfo
    IconFile="C:\Program Files\Mercury Interactive\QuickTest Professional Java
        Add-in Extensibility SDK\samples\Javaboutique\Calculator_3D.ico"/>
    <TypeInfo>
        <Operation ExposureLevel="CommonUsed" Name="Calculate"
            PropertyType="Method">
            <Description>Builds the whole calculation process</Description>
            <Documentation><![CDATA[Perform %a1 operation with %a2 and
                %a3 numbers]]></Documentation>
            <Argument Direction="In" IsMandatory="true" Name="operator">
                <Type VariantType="Variant"/>
            </Argument>
            <Argument Direction="In" IsMandatory="true" Name="num1">
                <Type VariantType="Variant"/>
            </Argument>
            <Argument Direction="In" IsMandatory="true" Name="num2">
                <Type VariantType="Variant"/>
            </Argument>
        </Operation>
        ...
    </TypeInfo>
    <Properties>
        <Property ForVerification="true" ForDefaultVerification="true"
            Name="value"/>
        <Property ForVerification="true" Name="objects count"/>
        <Property Name="width"/>
        ...
        <Property ForDescription="true" Name="toolkit class"/>
        ...
    </Properties>
</ClassInfo>
```

This example shows that the **Calculator** test object class extends the **JavaApplet** test object class. It uses the **Calculator_3D.ico** icon file, and its default test object method is **Calculate** (which has three mandatory input parameter of type **Variant**:**operator**, **num1** and **num2**).

The following identification properties are defined for the **Calculator** test object class:

- **value.** Available for checkpoints and selected by default in the Checkpoint Properties dialog box in QuickTest.
- **objects count.** Available for checkpoints but not selected by default.
- **toolkit class.** Used for the test object description but not available for checkpoints.

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

You can practice creating support for a custom control that is mapped to a new test object class in the tutorial lesson "Learning to Support a Complex Control" on page 225.

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

For information on the location in which to store this file, see "Deploying and Running the Custom Toolkit Support" on page 66.

Understanding How QuickTest Merges Test Object Configuration Files

Each time you open QuickTest, it reads all of the test object configuration files located in the <**QuickTest installation folder**>**\dat\Extensibility\Java** 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 **TypeInformation** element. For more information, refer to 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.

The following sections describe the process followed 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 higher priority, 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). In this case, QuickTest goes through all the classes and reattaches the enumeration values for arguments of type **Enumeration**.

- ➤ 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 definition 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 configuration file.
- ➤ If an **Operation** element is defined in a test object configuration file with a higher priority, 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 Custom Support Classes

In a custom toolkit support set, there is a custom support class for each supported custom class. The custom support class provides the actual interface between the custom class methods and the QuickTest capabilities, thus providing the QuickTest Java Add-in Extensibility.

Note: A single custom support class can provide support for more than one custom class. The support class can be mapped (in the toolkit configuration file described on page 36) to more than one custom class. This support class then provides support for the custom classes that are mapped to it, and for their descendants.

The first step in creating the support classes is determining the class inheritance hierarchy. This includes deciding the order in which you create support for classes within the custom toolkit, and determining which existing support class the new support class must extend. For more information, see “Determining the Inheritance Hierarchy for a Support Class,” below.

The second step is deciding what test object class to map to the custom control. For more information, see "Mapping a Custom Control to a Test Object Class" on page 49.

After you make the preliminary decisions regarding hierarchy and test object class, you are ready to write the main part of the QuickTest Java Add-in Extensibility—the custom support class.

Each custom support class determines what test object class is mapped to the custom control it supports and how the identification properties and test object methods are implemented.

The custom support class inherits the methods of its superclass. You can use the super implementation, override the methods, or add new ones, as needed. In support classes, you use the following types of methods:

➤ **Identification property support methods.** Used to support identification properties. For more information, see "Supporting Identification Properties" on page 50.

➤ **Replay methods.** Used to support test object methods. For more information, see "Supporting Test Object Methods" on page 53.

➤ **Event handler methods.** Used to provide support for recording on the custom control. This part of the Extensibility is optional. Even if you do not implement support for recording, you still have full support for the basic QuickTest capabilities on the custom control (for example, learning the object, running tests on it, checking properties and values, and so forth). If the custom class extends SWT, you cannot create support for the QuickTest recording capability. For more information, see "Supporting the Record Option" on page 56.

➤ **Utility methods.** Used to control the Extensibility. These methods do not support the specific functionality of the custom class; they control the interface between QuickTest and the custom application. Different utility methods are used for different purposes.

You can find a list of the available utility methods in the Support Class Summary on page 65. The methods are described in detail, in the following sections: "Supporting the Record Option" on page 56, "Supporting Top-Level Objects" on page 59, and "Supporting Wrapper Controls" on page 59.

When you implement these methods in the custom support class, you can use different methods supplied in the **MicAPI**. For more information, see "Using Methods from MicAPI" on page 66, and refer to the *QuickTest Java Add-in Extensibility API Reference* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility API Reference**).

For a short summary of the types of methods a custom class contains, see "Support Class Summary" on page 65.

Determining the Inheritance Hierarchy for a Support Class

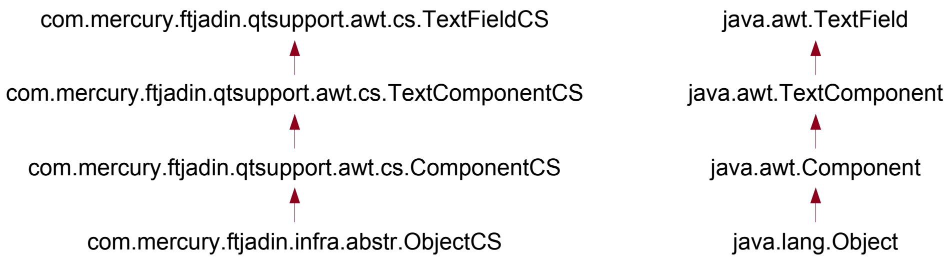
Within the custom toolkit for which you create QuickTest support, you must decide:

- Which custom classes must have matching support classes, and which can be supported by the support classes of their superclasses.
- Which existing support class each new support class extends.
  (This also determines the order in which support classes must be created.)

Understanding the Hierarchy of Support Classes

The hierarchy of the support classes must reflect the hierarchy of the custom classes.

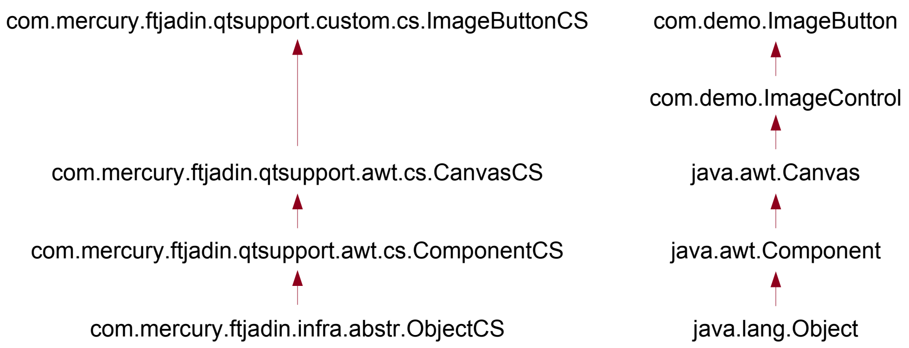
The following example illustrates the hierarchy of the **TextField** class support. The column on the left illustrates the hierarchy of the TextField support class, **TextFieldCS**. The column on the right illustrates the hierarchy of the **TextField** class in the AWT toolkit.


com.mercury.ftjadin.qtsupport.awt.cs.TextFieldCS
com.mercury.ftjadin.qtsupport.awt.cs.TextComponentCS
com.mercury.ftjadin.qtsupport.awt.cs.ComponentCS
com.mercury.ftjadin.infra.abstr.ObjectCS
java.awt.TextField
java.awt.TextComponent
java.awt.Component
java.lang.Object

In this example, a support class exists for every custom class, but this is not mandatory.

When QuickTest learns an object, it can always identify the class name of the object. According to the class, QuickTest determines the inheritance hierarchy of this class. QuickTest then searches the toolkit configuration files for the support class that is mapped to that class. If no support class is found, QuickTest searches for a support class that is mapped to the support class' immediate superclass, and so on, until a matching support class is found. Support classes can be provided by Mercury Interactive or any other vendor. If no other support class is found, AWT objects are supported by the **ComponentCS** class; SWT objects are supported by the **WidgetCS** class.

The following example illustrates the hierarchy of the **ImageButton** class support. The column on the left illustrates the hierarchy of the ImageButton support class, **ImageButtonCS**. The column on the right illustrates the hierarchy of the **ImageButton** class in the AWT toolkit.


com.mercury.ftjadin.qtsupport.custom.cs.ImageButtonCS
com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS
com.mercury.ftjadin.qtsupport.awt.cs.ComponentCS
com.mercury.ftjadin.infra.abstr.ObjectCS
com.demo.ImageButton
com.demo.ImageControl
java.awt.Canvas
java.awt.Component
java.lang.Object

No support class is mapped to the superclass of ImageButton, **ImageControl**. Therefore, the support class for **ImageButton** extends the support class mapped to the higher level—**CanvasCS**.

Determining Which Support Classes to Create

When determining which custom classes require support classes, you must consider the functionality and hierarchy of the custom classes.

If the support provided for a custom class' superclass is sufficient to support this custom class (meaning the custom class has no specific behavior that needs to be specifically supported), there is no need to create a support class for it.

Otherwise, you must create a new support class that extends the superclass' support class and map it to the custom class (in the toolkit configuration file described on page 36). In the new support class you need to implement only those elements of support that are not sufficiently supported by the superclass' support class.

If more than one custom class extends the same superclass, and they share an identification property or test object method that requires the same support, provide this support in a support class for the superclass, and not separately in each class' support class.

Determining Which Classes the New Support Classes Extend

To determine the existing support class that your new support class needs to extend, you examine the hierarchy of the custom class and check which support classes are mapped to its superclasses.

When you use the Java Add-in Extensibility wizards to create the custom toolkit support, the New Custom Support Class wizard determines which class to extend for each support class you create. It displays the custom class hierarchy and informs you which existing support class is the base (superclass) for the new support class. For more information, see "Custom Class Selection Screen" on page 99.

To determine the support class inheritance without the help of the Java Add-in Extensibility wizard:

1 Determine the inheritance hierarchy of the custom class.

2 Search the toolkit configuration files for a support class that is already mapped to a superclass of the custom class.

   You must search the toolkit configuration files that are part of the QuickTest Professional Java Add-in, as well as in those that are part of Extensibility custom toolkit support. These files are located in **<QuickTest Installation Folder> bin\java\classes\builtin** and in **<QuickTest Installation Folder> bin\java\classes\extension**, respectively.

3 Create the support class for the custom class, extending the support class that you found mapped to its closest superclass.

Note: If the closest support class you found is part of the QuickTest Professional Java Add-in, it is located in the **com.mercury.ftjadin.support** package. In this case, instead of extending it directly, you must extend the class with the same name provided in the **com.mercury.ftjadin.qtsupport** package.

The example below uses the **ImageButton** custom control to illustrate the process of determining the hierarchy of a support class.

This is the hierarchy of the **ImageButton** class:

```
java.lang.Object
 └─java.awt.Component
     └─java.awt.Canvas
         └─com.demo.ImageControl
             └─com.demo.ImageButton
```

ImageButton's nearest superclass, **com.demo.ImageControl**, is not mapped to a support class. The next superclass, **java.awt.Canvas** is mapped to **com.mercury.ftjadin.support.awt.cs.CanvasCS**. This is part of the QuickTest Professional Java Add-in, so **ImageButtonCS** will extend the **CanvasCS** class in the **qtsupport** package: **com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS**. This is the **ImageButtonCS** class definition:

```
package com.mercury.ftjadin.qtsupport.imagecontrols.cs;
import com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS;
...
public class ImageButtonCS extends CanvasCS {};
```

Mapping a Custom Control to a Test Object Class

The test object class that is mapped to a custom control determines the identification properties and test object methods that QuickTest uses for the control. The values and behavior of these properties and methods are determined by support methods implemented in the custom control's support class.

You can map the custom control to an existing test object class that has all of the identification properties and test object methods relevant to the custom control. Alternatively, you can create a new test object class definition (in a test object configuration file) and map the custom control to the new test object class.

Each new test object class is based on an existing one, extending its set of identification properties and test object methods. All test object classes extend the JavaObject class. If an existing test object class definition includes some, but not all, of the identification properties and test object methods that you need, create a new test object class that extends it. (It is not recommended to add identification properties and test object methods to an existing test object class because that would affect all of the test objects of this class.)

You map the custom control to a test object class by implementing the **to_class_attr** method in the support class, to return the name of the relevant test object class. If the test object class returned by the inherited **to_class_attr** method is appropriate for the custom control, you do not have to override the **to_class_attr** method in the new support class.

The **to_class_attr** method provides the value for the **Class Name** identification property. When QuickTest learns an object, it finds the support class to use for this object, as described in "Understanding the Hierarchy of Support Classes" on page 45. QuickTest then uses the **Class Name** identification property to determine which test object class is mapped to this control. QuickTest then uses this test object class name to find the test object definition, which can be taken from either an existing QuickTest test object, or from a new test object configuration file that you create.

For more information, see "Understanding the Test Object Configuration File" on page 38.

Supporting Identification Properties

The identification properties of a custom control are defined in the test object class. This can be an existing QuickTest test object class or one you define in a test object configuration file.

Support for the identification properties is provided in the support class by implementing a method with the following signature for each identification property:

```
public String <identification property name>_attr(Object obj)
```

The method name must contain only lower case letters (even if the property name in the test object configuration file contains upper case letters). The **obj** argument is the object that represents the custom control.

Within the method, you return the value of the required property by using the custom class' public members. (Note that the support class can access only those custom class members that are defined as public.)

For example, the **width_attr** method implements support for a **width** identification property:

```
public String width_attr(Object obj) {
    return Integer.toString(((Component) obj).getBounds().width);
}
```

When your support class extends the support class of a functionally similar control, you do not have to implement support for those identification properties that apply without change to the custom control. For example, many controls have a **label** property. If the implemented support of the **label** property adequately supports the custom control, you do not need to override the parent's method.

Note: You might inherit (or create) support methods for identification properties that are not included in the test object class definition. These identification properties are not displayed in QuickTest in the Object Spy or in the Checkpoint Properties dialog box. You can access these identification properties by using the **GetROProperty** method. For more information on the **GetROProperty** method, refer to the *QuickTest Professional Object Model Reference*.

To cover identification properties of the custom control that are not supported by the parent support class, add new methods in your support class. To cover identification properties that have the same name as supported ones, but a different implementation, override the parent methods.

Note: In JavaTree and JavaList test objects, there are identification properties named **tree_content** and **list_content** (respectively) that are used in checkpoints. QuickTest calculates these properties based on the **count** identification property and the **GetItem** test object method, as follows: QuickTest retrieves the **count** identification property, and calls the **GetItem** test object method for each item in the tree or list (from zero to count-1).

If you override the implementation of **count_attr** or **GetItem_replayMethod**, you must make sure that they return the type of information that QuickTest expects. For example, **count_attr** must return a numeric value and **GetItem_replayMethod** must return an item for each index from zero to count-1.

If you map a custom control to the JavaTree or JavaList test object classes, and the custom support class does not inherit the **count_attr** and **GetItem_replayMethod** methods, you must implement them to return the information that QuickTest expects.

Special Identification Property Support Methods

The following basic identification property support methods are commonly used when creating support classes. In Part II, "Tutorial: Learning to Create Java Custom Toolkit Support," you can practice using some of these methods:

- The **to_class_attr** method (described in "Mapping a Custom Control to a Test Object Class" on page 49) supports the **Class Name** identification property. It provides the mapping of the custom control to a test object class, by returning the name of the relevant test object class. QuickTest uses this property to determine which test object class is mapped to the custom control.
- The name of a test object is determined by its **tag** property. All AWT support classes extend ObjectCS. ObjectCS implements the **tag_attr** method to check a set of properties in a specified order and to return the first valid value it finds. A valid value is one that is not empty and does not contain spaces.

  In the ObjectCS class, the **tag_attr** method checks the following properties (in the order in which they are listed):

  - label
  - attached_text (for more details, see below)
  - unqualified custom class (the name of the class without the package name)

  To change the name of a custom control test object, do not override the **tag_attr** method in the support class. Instead, make use of its existing implementation and override the method **label_attr**.
- ObjectCS, which all AWT support classes extend, also implements the **attached_text_attr** method. It searches for adjacent static-text objects close to the custom control and returns their text. This mechanism is useful for controls such as edit boxes and list boxes, which do not have their own descriptive text, but are accompanied by a label.

  You can create support for a custom static-text control to enable QuickTest to use its **label** property as the **attached text** for an adjacent control. For more information, see "New QuickTest Custom Static-Text Support Class Wizard" on page 132.

- The **class_attr** method returns the name of the test object's generic type (**object**, **button**, **edit**, **menu**, **static_text**, and so forth). This is not the specific test object class mapped to the object, but the general type of test object class. If you are creating a support class for a static-text control, you must implement the **class_attr** method to return the string static_text. Otherwise, do not override it.
- The **value_attr** method is not mandatory, but it implements the **value** identification property, which is commonly used to represent the current state of the control. For example, the **value_attr** method may return the name of the currently selected tab in a tab control, the path of the currently selected item in a tree, the currently displayed item in a menu, and so forth. If you are creating a new test object class, and the term **current state** is relevant, implement support for a **value** identification property. If your support class inherits a **value_attr** method, verify that its implementation is correct for the supported control.

Supporting Test Object Methods

The test object methods of a custom control are defined in the test object class. This can be an existing QuickTest test object class or one you define in a test object configuration file.

Support for the test object methods is provided in the support class by implementing a **replay** method with the following signature for each test object method:

```
public Retval <test object method name>_replayMethod(Object obj, <… list of
String arguments>)
```

The **obj** argument is the object that represents the custom control.

Replay methods accept only strings as arguments, and QuickTest passes all arguments to them in a string format. To use the boolean or numeric value of the argument, use **MicAPI.string2int**.

Within the replay method, you carry out the required operation on the custom control by using the custom class public methods or by dispatching low-level events using MicAPI methods. (Note that the support class can access only those custom class methods that are defined as public.) For more information, refer to the *QuickTest Java Add-in Extensibility API Reference*.

For example, **Click_replayMethod** (in the ImageButtonCS class), supports the **Click** test object method on an ImageButton custom control:

```
public Retval Click_replayMethod(Object obj) {
    ImageButton button = (ImageButton) obj;
    MicAPI.mouseClick((Object) button, button.getWidth() / 2,
                                            button.getHeight() / 2);
    Return Retval.OK;
}
```

All replay methods must return a **MicAPI.Retval** value. The **Retval** value always includes a return code, and can also include a string return value. The return code provides information to QuickTest about the success or failure of the test object method. The return value can be retrieved and used in later steps of a QuickTest test.

For example, the EmulatedTextField control is a custom text box. The **GetValue_replayMethod** (in the EmulatedTextFieldCS class) returns the text from this custom text box in addition to the return code **OK**:

```
public Retval GetValue_replayMethod(Object obj) {
    String myvalue;
    EmulatedTextField emt = (EmulatedTextField)obj;
    myvalue = emt.getText();
    return new Retval(RError.E_OK, myvalue);
}
```

For more information on the **MicAPI.Retval** values recognized by QuickTest, refer to the *QuickTest Java Add-in Extensibility API Reference* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility API Reference**).

When your support class extends the support class of a functionally similar control, you do not have to implement support for those test object methods that apply without change to the custom control. For example, many controls have a **Click** test object method. If the implemented support of the **Click** test object method adequately supports the custom control, you do not need to override the parent's method.

To cover test object methods of the custom control that are not supported by the parent support class, add new methods in your support class. To cover test object methods that have the same name as supported ones, but a different implementation, override the parent methods.

Note: In JavaTree and JavaList test objects, there are identification properties named **tree_content** and **list_content** (respectively) that are used in checkpoints. QuickTest calculates these properties based on the **count** identification property and the **GetItem** test object method, as follows: QuickTest retrieves the **count** identification property, and calls the **GetItem** test object method for each item in the tree or list (from zero to count-1).

If you override the implementation of **count_attr** or **GetItem_replayMethod**, you must make sure that they return the type of information that QuickTest expects. For example, **count_attr** must return a numeric value and **GetItem_replayMethod** must return an item for each index from zero to count-1.

If you map a custom control to the JavaTree or JavaList test object classes, and the custom support class does not inherit the **count_attr** and **GetItem_replayMethod** methods, you must implement them to return the information that QuickTest expects.

Supporting the Record Option

You can extend QuickTest support of the recording option only for controls that extend AWT.

If you do not implement support for recording, you still have full support for all of the other QuickTest capabilities on the custom control, for example, learning the object, running tests on it, checking properties and values, and so forth.

To support recording on a custom control, the custom support class must:

- ➤ Implement listeners for the events that you want to trigger recording.
- ➤ Register the listeners on the custom controls when the are created.
- ➤ Send Record events to QuickTest when the relevant events occur.
- ➤ Override low-level recording if you want to record more complex operations. For example, if you want to record a **JavaEdit.Set** operation, you must override the recording of individual keyboard inputs. If you want to record selecting an option in a menu, you must override recording of mouse clicks.

In Part II, "Tutorial: Learning to Create Java Custom Toolkit Support" you can practice creating support for recording on custom controls.

To add support for recording to a custom support class:

1 Include the listeners in the support class signature. For example, the ImageButton support class ImageButtonCS listens for Action events:

```
public class ImageButtonCS extends CanvasCS implements ActionListener {}
```

2 Use a constructor for the support class to generate a list containing all of the listeners that you want to register on the custom control, and the methods used to add and remove these listeners.

You do this by calling the utility method **addSimpleListener** for each listener. This method accepts three arguments of type String: The name of the listener, the name of the registration method, and the name of the method used to remove the listener.

In the example below, the Action listener is listed for registration on ImageButton custom controls:

```
public ImageButtonCS() {
        addSimpleListener("ActionListener", "addActionListener",
                "removeActionListener");
```

The first time QuickTest identifies the custom control, it creates an instance of the support class for this custom control. This instance of the support class is used to support all subsequent controls of this custom class. Whenever a custom class instance is created, the support class registers the required listeners on the object using the registration methods you specified.

3 Override low-level recording (optional):

To override recording of low-level mouse events:

```
protected Object mouseRecordTarget(MouseEvent e) {
    return null;
}
```

To override recording of low-level keyboard events:

```
protected Object keyboardRecordTarget(KeyEvent e) {
    return null;
}
```

4 Implement the relevant event handler methods from the listener interface, to send record messages to QuickTest, using the **MicAPI.record** methods.

For information on how to use **MicAPI.record**, refer to the *QuickTest Java Add-in Extensibility API Reference* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility API Reference**).

For example, the following event handler method is implemented in ImageButtonCS, the support class for ImageButton:

```
public void actionPerformed(ActionEvent e) {
    try {
        if (!isInRecord())
            return;
        MicAPI.record(e.getSource(), "Click");
        } catch(Throwable tr)
            { tr.printStackTrace();
            }
}
```

When an Action event occurs on an ImageButton, QuickTest records a **Click** operation on the ImageButton.

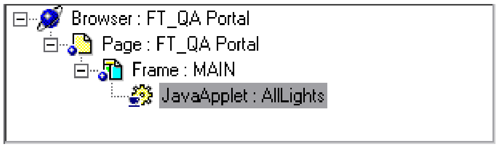
The **try ... catch** block prevents unnecessary activity if this code is reached when the Java application is running while QuickTest is idle. The stack trace is printed to the same log file as other Java Add-in Extensibility log messages, enabling you to determine when this method was called inadvertently. For more information, see "Logging and Debugging the Custom Support Class" on page 71.

For information on recording on wrapper controls, see "Supporting Wrapper Controls," below.

Note: If **MicAPI.record** is called when there is no active QuickTest recording session, nothing happens. If you perform additional calculations or assignments before calling **MicAPI.record**, make sure that you first call **isInRecord** to determine whether a recording session is active. If no recording session is active, you may want to avoid certain operations.

Supporting Top-Level Objects

If you want QuickTest to recognize the custom control as the highest Java object in the test object hierarchy, you need to inform QuickTest that this Java control is a top-level object. You do this by overriding the utility method **isWindow (Object obj)** in the support class to return **true**. In the following example, the JavaApplet **AllLights** is a top-level Java object.

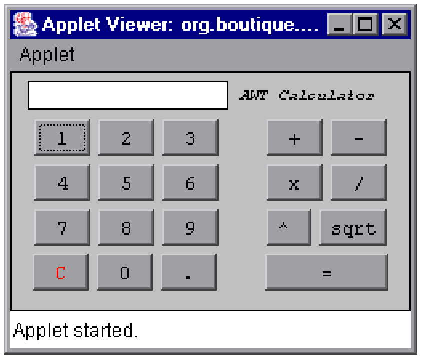


Only a container object can be a top-level object. A container object is one that extends **java.awt.container** if it is AWT-based, or **org.eclipse.swt.widgets.Composite** if it is SWT-based.

If the control is a top-level object only in some situations, you can implement the **isWindow** method to return **true** in some situations and **false** in others. For example, an applet can be a standalone application or an object within a Web browser.

Supporting Wrapper Controls

A wrapper control is a container control that groups the controls within it and represents them as a single control. For example, the AwtCalc calculator control is a wrapper control.



When QuickTest learns a wrapper control, it does not learn the controls within it separately as descendants. If you record a test on a wrapper control, events that occur on the controls within it, are recorded as operations on the wrapper control.

Note: Only AWT-based controls can be supported on QuickTest as wrapper controls. If the custom control is SWT-based, it is always learned with all of its descendants.

For example, the AwtCalc calculator control contains simple buttons for digits and operators. In a recording session on this control, you might want simple click operations to be interpreted as more meaningful calculator-oriented operations. You can use Java Add-in Extensibility to instruct QuickTest to record clicks on digit buttons as **Calculator.SetValue** steps, and clicks on operator buttons as **Calculator.SetOperator** steps.

Understanding How QuickTest Handles Wrapper Controls

Wrapper controls must register themselves as wrappers for the types of controls that they wrap.

Before QuickTest learns a control as a descendant, QuickTest checks if any wrappers are registered for this type of control. If there are registered wrappers, QuickTest searches for the one to which this particular control belongs. QuickTest performs this search by calling the **checkWrappedObject** method of each registered wrapper. If QuickTest finds a relevant wrapper, QuickTest does not learn the descendant control. If no relevant wrapper is found, QuickTest learns the descendant control.

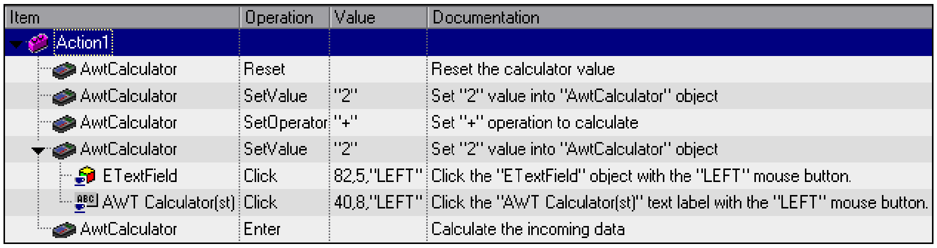
When a control is learned separately (by clicking on the specific control), QuickTest does not check for wrappers.

Similarly, before QuickTest records an operation on a control, QuickTest checks if any wrappers are registered for this type of control. If there are registered wrappers, QuickTest searches for the one to which this particular control belongs. If QuickTest finds a relevant wrapper, QuickTest passes the record message to the wrapper control before adding a step to the test. If no relevant wrapper is found, the operation is recorded as is.

When the wrapper receives a record message (triggered by an operation performed on one of its wrapped objects), it can do one of the following:

- Discard the message to prevent the recording of the operation.
- Modify the message to record a different operation.
- Leave the message as is to record the operation without intervention.

The following section describes how this mechanism is implemented, using the AwtCalc wrapper control as an example. After support for the AwtCalc control is implemented, a test recorded on the control could look like this:



Item	Operation	Value	Documentation
Action1			
AwtCalculator	Reset		Reset the calculator value
AwtCalculator	SetValue	"2"	Set "2" value into "AwtCalculator" object
AwtCalculator	SetOperator	"+"	Set "+" operation to calculate
AwtCalculator	SetValue	"2"	Set "2" value into "AwtCalculator" object
ETextField	Click	82,5,"LEFT"	Click the "ETextField" object with the "LEFT" mouse button.
AWT Calculator(st)	Click	40,8,"LEFT"	Click the "AWT Calculator(st)" text label with the "LEFT" mouse button.
AwtCalculator	Enter		Calculate the incoming data

Implementing Support for Wrapper Controls

If you want to support a wrapper control, you must implement the **com.mercury.ftjadin.infra.abstr.RecordWrapper** interface in MicAPI. This interface includes the following methods:

- **public void registerWrapperInspector()**
- **public Object checkWrappedObject(Object obj)**
- **public RecordMessage wrapperRecordMessage(RecordMessage message, Object wrapper)**
- **public boolean blockWrappedObjectRecord()**

The sections below describe each of these methods in detail.

public void registerWrapperInspector()

This method is used to register as a wrapper for the relevant types of controls.

For example, the AwtCalcCS support class registers itself as a wrapper of **Button** controls:

```
public void registerWrapperInspector() {
    MicAPI.registerWrapperInspector(Button.class, this);
}
```

The AwtCalcCS is registered as a wrapper for **Button** controls only, therefore operations on the **AWT Calculator** label or on the edit box will be recorded without any wrapper intervention. In addition, when the AwtCalc control is learned, the label and edit box are learned as its descendants.

public Object checkWrappedObject(Object obj)

QuickTest calls this method to check whether a specific object belongs to the custom control. The support class implements this method to return the specific wrapper instance if **obj** is wrapped by the custom control. Otherwise, it returns null.

For example, the **checkWrappedObject** method in AwtCalcCS is implemented, as follows:

```
public Object checkWrappedObject(Object obj) {
Component comp = (Component)obj;
if
(comp.getParent().getClass().getName().equals("org.boutique.toolkit.AwtCalc"))
    return comp.getParent();
return null;
}
```

public RecordMessage wrapperRecordMessage(RecordMessage message, Object wrapper)

QuickTest calls this method during a recording session when a wrapped object sends a record message. QuickTest passes the record message to the wrapper control before adding a step to the test.

This method returns one of the following:

- **null**, indicating that this message should be ignored and no step should be recorded
- a modified record message to be sent instead of the original one
- the original record message

For example, in the **wrapperRecordMessage** method in AwtCalcCS, if the operation to record is on a button, the method replaces it with the appropriate operation to record—Reset, Enter, SetOperator or SetValue (with the appropriate parameters). If the operation in the record message in on a label or text field, AwtCalc does not interfere with the recording.

```
public RecordMessage wrapperRecordMessage(RecordMessage message,
Object wrapper) {
    Object subject = message.getSubject();
    if (subject instanceof Button) {
        // Get the label of the button
        String value = ((Button) subject).getLabel().trim();
        String operation;
        // Select what method will be recorded and with what parameters
        if (value.equals("=")) {
            return RecordMessage.getRecordMessageInstance(wrapper,"Enter");
        }
        if (value.equals("C")) {
            return RecordMessage.getRecordMessageInstance(wrapper,"Reset");
        } else {
            if (value.equals("+") || value.equals("-") || value.equals("x")
                    || value.equals("/") || value.equals("^")
                    || value.equals("sqrt"))
                operation = "SetOperator";
            else
                operation = "SetValue";
            }
            String params[] = new String[1];
            params[0] = value;
            RecordMessage res =
                RecordMessage.getRecordMessageInstance(wrapper, operation,
                    params, AgentRecordMode.NORMAL_RECORD);
            return res;
        }
        // AwtCalc does not interfere if the message is not from a button
        return message;
    }
```

boolean blockWrappedObjectRecord()

When this method returns **false**, the controls contained in the wrapper generate record messages in response to events as if they were independent controls. QuickTest then calls **wrapperRecordMessage** to pass the record messages it receives from wrapped controls to the wrapper. The wrapper can then decide whether to discard the message, modify it, or record the operation as is.

When this method returns **true**, it causes all of the controls contained in the wrapper to ignore all events. The wrapped controls do not send any record messages to QuickTest, and **wrapperRecordMessage** is never called.

If **blockWrappedObjectRecord** returns null, and you want the wrapper to record events that occur on the objects it contains, the wrapper itself must register new event listeners on the wrapped objects. Then it must handle the events to generate the appropriate test steps (using **MicAPI.record**) during a recording session.

Support Class Summary

The following table summarizes the types of methods you use in a custom support class. For more information, refer to the *QuickTest Java Add-in Extensibility API Reference* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility API Reference**).

Method Type	Syntax	Common Methods
Identification property methods	public String <identification property name>_attr(Object obj)	to_class_att tag_attr label_attr attached_text_attr class_attr value_attr
Test Object Methods	public Retval <test object method name>_replayMethod(Object obj, <… list of String arguments>)	
Event Handling methods	Dependent on the listener that is being implemented.	Call MicAPI.record from the event handler methods.
Utility methods to use	protected void addSimpleListener(String listenerName, String addMethodName, String removeMethodName) public static final boolean isInRecord()	
Utility methods to override	public boolean isWindow(Object obj) protected Object mouseRecordTarget (MouseEvent e) protected Object keyboardRecordTarget (KeyEvent e) public boolean blockWrappedObjectRecord() public void registerWrapperInspector() public Object checkWrappedObject(Object obj) public RecordMessage wrapperRecordMessage(RecordMessage message, Object wrapper)	

Using Methods from MicAPI

MicAPI contains several sets of methods that you can use in the custom support classes to provide the following types of functionality:

- Dispatching low-level events. These methods include **MouseClick**, **KeyType**, and **postEvent**. These methods are commonly used in replay methods.
- Recording custom control operations on QuickTest. These methods are commonly used in event handler methods.
- Logging messages and errors from the support classes. These methods are used throughout the custom support class, to print log and error messages. For more information, see "Logging and Debugging the Custom Support Class" on page 71.

Deploying and Running the Custom Toolkit Support

The final stage of extending QuickTest support for a custom toolkit is deployment. This means placing all of the files you created in the correct locations, so that the custom toolkit support is available to QuickTest.

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

You can also deploy the toolkit support during the development stages, to test how it affects QuickTest and debug the custom toolkit support set that you are creating.

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

File Name	Location
<**Custom Toolkit Name**>**.xml**	<QuickTest Installation Folder>\bin\java\classes\extension
<**Custom Toolkit Name**>**TestObjects.xml** Optional. Required only if mapping custom classes to new test object classes. **Note:** This file name convention is used by the Java Add-in Extensibility wizard. You can have more than one test object configuration file, and name them as you wish.	• <QuickTest Installation Folder>\Dat\Extensibility\Java • <QuickTest Add-in for Quality Center Installation Folder>\Dat\Extensibility\Java (Optional. Required only if QuickTest Add-in for Quality Center is installed)
<**Custom Toolkit Name**>**Support.class**	All of the Java support classes can be packaged in class folders or Java archives on the computer on which QuickTest is installed, or in an accessible network location. Specify the location in <Custom Toolkit Name>.xml
<**CustomClass**>**CS.class**	
Icon files for new test object classes (optional)	Must be an uncompressed **.ico** format file, located on the computer on which QuickTest is installed, or in an accessible network location. Specify the location in <Custom Toolkit Name>TestObjects.xml

Notes:

If you want to remove support for a custom toolkit from QuickTest after it is deployed, you must delete its toolkit configuration file from:
<QuickTest Installation Folder> bin\java\classes\extension

If none of the test object class definitions in a test object configuration file are mapped to any custom controls (meaning they are no longer needed), you can delete the file from:
<QuickTest Installation Folder>\Dat\Extensibility\Java (and **<QuickTest Add-in for Quality Center Installation Folder>\Dat\Extensibility\Java** if relevant).

Deploying Custom Support During the Development Stage

During the design stages of the custom toolkit support, the support class files can remain in your workspace. You deploy the custom toolkit support by placing the toolkit configuration files (including the test object configuration file) in the correct locations, and by specifying the location of the compiled support classes in the toolkit configuration (XML) file. In addition, if your new test object classes use specific icons, you specify their locations in the test object configuration file.

If you develop custom toolkit support using the QuickTest Java Add-in Extensibility plug-in in Eclipse, and QuickTest is installed on your computer, you deploy toolkit support by clicking the **Deploy Toolkit Support** Eclipse toolbar button, or by choosing **QuickTest** > **Deploy Toolkit Support**. The XML configuration files are copied to the correct QuickTest locations, while the Java class files remain in the Eclipse workspace. (The actual locations of the toolkit support class and the custom support classes are listed in the toolkit configuration file.)

If you do not use the QuickTest Java Add-in Extensibility plug-in in Eclipse, or if QuickTest is installed on another computer, you must perform the deployment manually, according to the information in the table on page 66.

To deploy custom support manually during the development stages:

1. Make sure that the compiled support classes (toolkit support class and custom support classes) are in a location that can be accessed by QuickTest.
2. Update the configuration files with the correct locations of the compiled support classes and icon files (if relevant).
3. Copy the configuration files to the appropriate folders, as described in the table on page 66.

Deploying Custom Support After the Design is Completed

When the custom toolkit support is fully designed, you can deploy it to any computer on which QuickTest is installed.

To deploy custom support after the design is completed:

1. Place the compiled support classes (toolkit support class and custom support classes) in their permanent location. The classes can be in class folders or in a Java archive, in a location that can be accessed by QuickTest.

   In addition, if you have new test object classes using specific icons, place the icon files in a location that can be accessed by QuickTest.
2. Update the toolkit configuration file with the correct location of the compiled support classes.

   If necessary, update the test object configuration file with the correct location of the icon files.
3. Copy the configuration files to the appropriate folders, as described in the table on page 66.

Modifying Deployed Support

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

- If you modify the toolkit configuration file or a test object configuration file, you must deploy the support.
- If you modify a test object configuration file, you must reopen QuickTest after deploying the support.
- Whether you modify the configuration files or only the Java support classes, you must re-run the Java application for the changes to take effect.

Running an Application with Supported Custom Controls

After you deploy the custom toolkit support, you can perform QuickTest operations on an application that contains the supported custom controls to test the effects of the support.

You can run the application in any way you choose. If you run an SWT application from Eclipse, Eclipse overrides the Java library path to add the SWT dll. Therefore, you must add the **jvmhook.dll** path (required by the Java Add-in) to the library path manually.

To add the jvmhook.dll path to the library path:

1. Right-click the application file in the Eclipse Package Explorer. Choose **Run As** > **SWT Application**.
2. In the Eclipse toolbar, choose **Run** > **Run**. The Run dialog box opens.
3. Select the SWT application in the **Configurations** list.
4. Click the Arguments tab.
5. In the VM arguments area, enter:

   `-Djava.library.path=<System Folder>\system32`

   (For example: `-Djava.library.path=c:\WINNT\system32`)
6. Close the application and repeat step 1 to run the application again.

Logging and Debugging the Custom Support Class

When you design your support classes, it is recommended to include writing messages to a log file, to assist in debugging any problems that may arise.

Use the **MicAPI.logLine** method to send messages to the log file. For more information, see *QuickTest Java Add-in Extensibility API Reference* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility API Reference**).

To control the printing of the log messages (to prevent all messages from being printed at all times), you create debug flags in each support class. When you call **MicAPI.logLine**, you provide the appropriate debug flag as the first argument. **MicAPI.logLine** prints the log messages only when the debug flag that you specified is on.

The following example illustrates how to print log messages in a support class:

```
private static final String DEBUG_ALLLIGHTSCS = "DEBUG_ALLLIGHTSCS";

public String light_on_positions_attr(Object obj) {
        AllLights lights = (AllLights) obj;
...
        for (int i = 0; i < 5; i++) {
           for (int j = 0; j < 5; j++) {
            if(lights.isSet(j, i)) {
                MicAPI.logLine(DEBUG_ALLLIGHTSCS, "Light "+i+":"+j+" is set");
...
            }
        }
    }
}
```

In QuickTest, you create a test with the following two lines and run it to control the logging. Within the test, list the flags to turn on and the file to which the messages should be written:

```
javautil.SetAUTVar "sections_to_debug", "DEBUG_ALLLIGHTSCS"
javautil.SetAUTVar "debug_file_name", "C:\JavaExtensibility\Javalog.txt"
```

If you want to turn on more than one flag simultaneously, enter all of the flag strings consecutively in the second argument (separated by spaces), as in the following example:

```
javautil.SetAUTVar "sections_to_debug", "DEBUG_ALLLIGHTSCS
DEBUG_AWTCALC"
```

The messages printed by **MicAPI.logLine**, according to the flags you set, are printed to the specified file when the support class runs. To change the flags controlling the log printing, or to change the file to which they are written, run the QuickTest test again with the appropriate arguments.

Debugging Your Custom Toolkit Support

The Java support classes run in the context of the application you are testing. Therefore, if you want to debug your support classes, you can do so in the same way as you would debug the application itself.

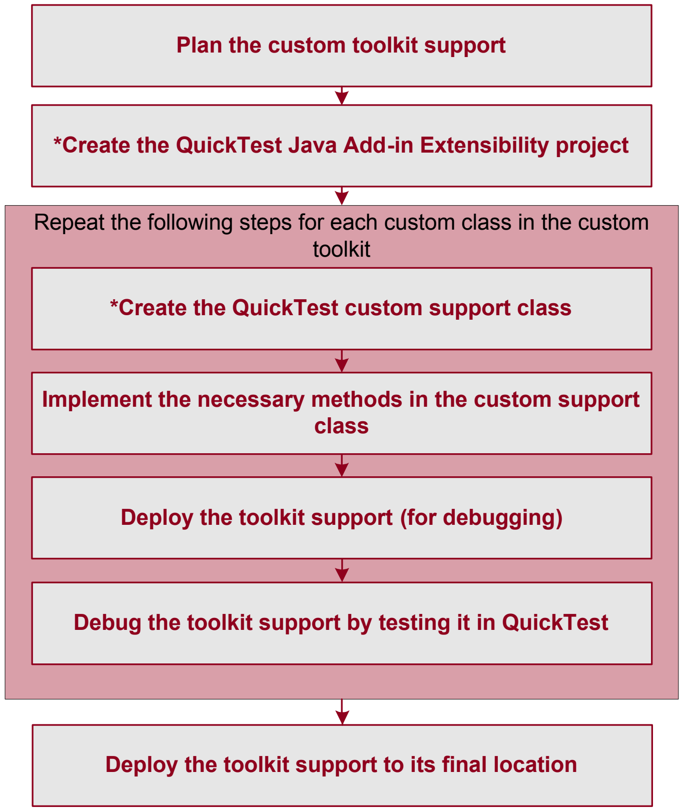
To begin debugging, place breakpoints within the support classes, run the application as though you were debugging it, and perform different QuickTest operations on the application to reach the different parts of the support classes.

If the application code is stored in Eclipse, you can run it in debug mode from Eclipse. (Right-click the application file and choose **Debug As** > **Java Applet** (or **Application**) or **Debug As** > **SWT Application**.)

If the application code is not stored in Eclipse, use remote debugging on the application to debug the support classes. For information about remote debugging, refer to the *Eclipse Help*.

Workflow for Implementing Java Add-in Extensibility

The following workflow summarizes the steps you need to perform to create QuickTest Java Add-in Extensibility support for a custom toolkit, and the order in which you need to perform them. Follow these steps for each custom toolkit you want to support.


Plan the custom toolkit support
*Create the QuickTest Java Add-in Extensibility project
Repeat the following steps for each custom class in the custom toolkit
*Create the QuickTest custom support class
Implement the necessary methods in the custom support class
Deploy the toolkit support (for debugging)
Debug the toolkit support by testing it in QuickTest
Deploy the toolkit support to its final location

* You can use the wizards in the QuickTest Java Add-in Extensibility Eclipse plug-in to create the custom toolkit support project, the custom classes, and all of the required files. Alternatively, if you choose not to use the wizards, you must create the necessary packages and files manually, as described in "Creating a Custom Toolkit Support Set" on page 32. In addition, if you then decide to map custom classes to new test object classes, you must define the new test object classes in a test object configuration file.

4

Planning Custom Toolkit Support

Before you begin to create support for a custom toolkit, you must carefully plan the support. Detailed planning of how you want QuickTest to recognize the custom controls enables you to correctly build the fundamental elements of the custom toolkit support. It is important to plan all of the details before you begin. Making certain changes at a later stage might require intricate manual changes, or even require you to recreate the custom support.

Note: This chapter assumes familiarity with the concepts presented in Chapter 3, "Implementing Custom Toolkit Support."

About Planning Custom Toolkit Support

Creating custom toolkit support is a process that requires detailed planning. To assist you with this, the sections in this chapter include sets of questions related to the implementation of support for your custom toolkit and its controls. When you are ready to create your custom toolkit support, you will be implementing support for it based on the answers you provide to these questions.

The first step is determining general information related to your custom toolkit, after which you will define the specific information related to each custom class you want to support.

Determining the Custom Toolkit Related Information

To plan the details related to the custom toolkit answer the following questions:

- What custom classes are included in the custom toolkit?

  Provide a unique name for the custom toolkit, and list the locations of the custom classes. The locations can be Eclipse projects, Java archive files or class folders.

  For the rules on grouping custom classes into toolkits you can support, see "Preparing to Create Support for a Custom Toolkit" on page 30.

- What native toolkit (or existing supported toolkit) does the custom toolkit extend?

Note: When all of the classes in a custom toolkit extend the basic user interface class of another toolkit (for example **java.awt.Component**) we say the custom toolkit extends that toolkit (in this example: **AWT**).

- In what order do you want to create support for the different classes within the toolkit?

  For information on how to answer this question, see "Determining the Inheritance Hierarchy for a Support Class" on page 45.

Determining the Support Information for Each Custom Class

Before you begin planning the support for a custom class, make sure you have full access to the control and understand its behavior. You must have an application in which you can view the control in action, and also have access to the custom class that implements it.

You do not need to modify any of the custom control's sources to support it in QuickTest, but you do need to be familiar with them. Make sure you know which members (fields and methods) you can access externally, the events for which you can listen, and so forth.

When planning custom support for a specific class, carefully consider how you want QuickTest to recognize controls of this class—what type of test object you want to represent the controls in QuickTest tests, which identification properties and test object methods you want to use, and so forth. The best way to do this is to run the application containing the custom control and to analyze the control from a QuickTest perspective using the Object Spy, the Keyword View, and the Record option. This enables you to see how QuickTest recognizes the control without custom support, and helps you to determine what you want to change.

To view an example of analyzing a custom control using QuickTest, see "Analyzing the Default QuickTest Support and Extensibility Options for a Sample Custom Control" on page 8.

Understanding the Custom Class Support Planning Checklist

When you plan your custom support for a specific class, you must ask yourself a series of questions. These are explained below and are available in an abbreviated, printable checklist on page 80.

Note: Questions 1, 4, and 5 are fundamental to the design of the custom toolkit support. Changing the answers to these questions after creating support may require you to make complex manual changes, or even to recreate the custom support.

---

1 Make sure you select the correct custom class to support:

   a Does the custom class have a superclass for which QuickTest custom support is not yet available?

   b Does the custom control have identification properties or test object methods that require the same QuickTest support as other controls that extend the same superclass?

   If so, consider creating support for the superclass first.

2 Make sure you have access to custom class sources and to an application that runs the custom control on a computer with QuickTest installed.

3 Make sure you have access to the compiled custom class on the computer on which you are programming. The classes can be in class folders, a Java archive, or an Eclipse project.

4 Is there an existing Java test object class which adequately represents the custom control? If so, which one?

5. If not, you need to create a new test object class:
   - **a** Is there an existing Java object class which can be extended to represent the custom control? If so, which one? If not, your new test object class needs to extend the JavaObject class.
   - **b** Do you want QuickTest to use a different icon for the new test object? If so, make sure the icon file is available in an uncompressed **.ico** format.
   - **c** Specify one or more identification properties that can be used to uniquely identify the control (in addition to the test object class and the fully qualified Java class name of the control).
   - **d** Specify the default test object method to be displayed in the Keyword View and Step Generator when a step is generated for an object of this class.
6. Do you want QuickTest to recognize the custom control as a top-level Java test object?
7. Does the custom control contain objects that are significant only in the context of this control (meaning, is it a wrapper)? (For example, a Calculator object is a wrapper for the calculator button objects.)
8. Specify the basis for naming the test object that represents the control.
9. List the identification properties you want to support.

   If you are creating a new test object class, also decide which properties should be selected by default in the Checkpoint Properties dialog box in QuickTest.
10. List the test object methods you want to support. Specify the method argument types and names, and whether it returns a value in addition to the return code.
11. If the custom control is AWT-based, do you want to provide support for creating QuickTest tests by using the Record option?

    If so, list the events you want to record on the custom control during a QuickTest recording session.

Note: You can use the checklist on the following page to mark your answers.

Custom Class Support Planning Checklist

Use this checklist to plan your custom class toolkit support.

☑	**Custom Class Support Planning Checklist**
☐	Does the custom class have a superclass for which QuickTest custom support is not yet available? **Yes /No**
☐	If so, should I first extend support for a control higher in the hierarchy? **Yes /No**
☐	Do I have an application that runs the custom control on a computer with QuickTest installed? **Yes /No**
☐	The sources for this custom control class are located in:
☐	Which existing Java test object matches the custom control?
☐	If none, create a new Java test object class named: • New test object class extends: (Default—JavaObject) • Icon file location (optional): • Identification property for description: • Default test object method:
☐	Should QuickTest recognize the custom control as a top-level Java test object? **Yes /No**
☐	Is the custom control a wrapper? **Yes /No**
☐	Specify the basis for naming the test object:
☐	List the identification properties to support, and mark default checkpoint properties:
☐	List the test object methods to support (include arguments and return values if required):
☐	Provide support for recording? (AWT-based only) **Yes /No**
☐	If so, list the events that should trigger recording:

Where Do You Go from Here?

After you finish planning the custom toolkit support, you create the custom toolkit support set to support the custom toolkit as per your plan. You can create all of the required files, classes, and basic methods using the QuickTest Java Add-in Extensibility wizards in Eclipse. The wizards also provide method stubs for additional methods that you may need to implement. For more information, see "Using the QuickTest Java Add-in Extensibility Eclipse Plug-In" on page 83.

If you choose not use the Java Add-in Extensibility wizard in Eclipse, you can still extend full support for the custom toolkit manually using the information in Chapter 3, "Implementing Custom Toolkit Support."

5

Using the QuickTest Java Add-in Extensibility Eclipse Plug-In

The QuickTest Professional Java Add-in Extensibility SDK includes a plug-in for the Eclipse Java development environment. This plug-in provides wizards that you can use to create custom toolkit support sets and commands for editing the files after they are created.

If you choose not use the Java Add-in Extensibility wizards, you can skip this chapter. In this case, you can extend full support for the custom toolkit manually, as described in "Implementing Custom Toolkit Support" on page 27.

This chapter describes:	On page:
About the QuickTest Java Add-in Extensibility Eclipse Plug-In	84
New QuickTest Java Add-in Extensibility Project Wizard	85
Modifying QuickTest Java Add-in Extensibility Project Properties	96
New QuickTest Custom Support Class Wizard	97
New QuickTest Custom Static-Text Support Class Wizard	132
Working with QuickTest Commands in Eclipse	137

About the QuickTest Java Add-in Extensibility Eclipse Plug-In

When you install the QuickTest Professional Java Add-in Extensibility SDK, the QuickTest Java Add-in Extensibility plug-in is added to Eclipse. This plug-in provides wizards that you can use to create custom toolkit support sets and commands for editing the files after they are created. For information about installing and uninstalling the Java Add-in Extensibility SDK, see "Installing the QuickTest Professional Java Add-in Extensibility Software Development Kit" on page 13.

You can use the wizards supplied by the QuickTest Java Add-in Extensibility plug-in in Eclipse to create and deploy custom toolkit support. The wizards create all of the necessary files, classes, and methods, based on details you specify about the custom classes and the required support. The wizards also provide method stubs for the additional methods you may need to implement.

This chapter assumes that you have read the "Implementing Custom Toolkit Support" chapter of this guide (on page 27), which explains the elements that comprise custom toolkit support and the workflow for creating this support.

When you create support for a custom toolkit, you first use the New QuickTest Java Add-in Extensibility Project Wizard to create an Eclipse project containing the packages and files for the custom toolkit support.

Then you create support classes for the relevant custom classes using the New QuickTest Custom Support Class Wizard (described on page 97). To create a support class for a custom static-text class, you use the New QuickTest Custom Static-Text Support Class Wizard (described on page 132).

After the wizard creates the support class according to your specifications, you must complete the design of the custom support. To do this, you implement the method stubs created by the wizard to match the needs of the custom control.

The QuickTest Java Add-in Extensibility Eclipse plug-in also provides commands that you can use to edit the support you are designing, and to deploy it to QuickTest for debugging. These commands are described in "Working with QuickTest Commands in Eclipse" on page 137.

Note: While you are working with the wizard, do not rename or delete any of the files that the wizard creates. When the wizard performs the commands you specify, it searches for the files according to the names it created. When the custom toolkit support set is complete and you are performing the final deployment, you can rename the configuration files. In the final deployment stage, you can also divide the test object configuration file into more than one file. Place the custom toolkit support set files in the appropriate folders, as specified in "Deploying Custom Support After the Design is Completed" on page 69.

New QuickTest Java Add-in Extensibility Project Wizard

You use the New QuickTest Java Add-in Extensibility Project wizard to create a new project in Eclipse containing the files that comprise the support set for a specific custom toolkit. After you specify the details of the custom toolkit, the wizard creates the necessary toolkit support files.

After you create the New QuickTest Java Add-in Extensibility project, you can create support for each of the custom toolkit classes. To do this, you use the New QuickTest Custom Support Class Wizard, described on page 97 (or the New QuickTest Custom Static-Text Support Class Wizard, described on page 132).

To open the New QuickTest Java Add-in Extensibility Project wizard in Eclipse:

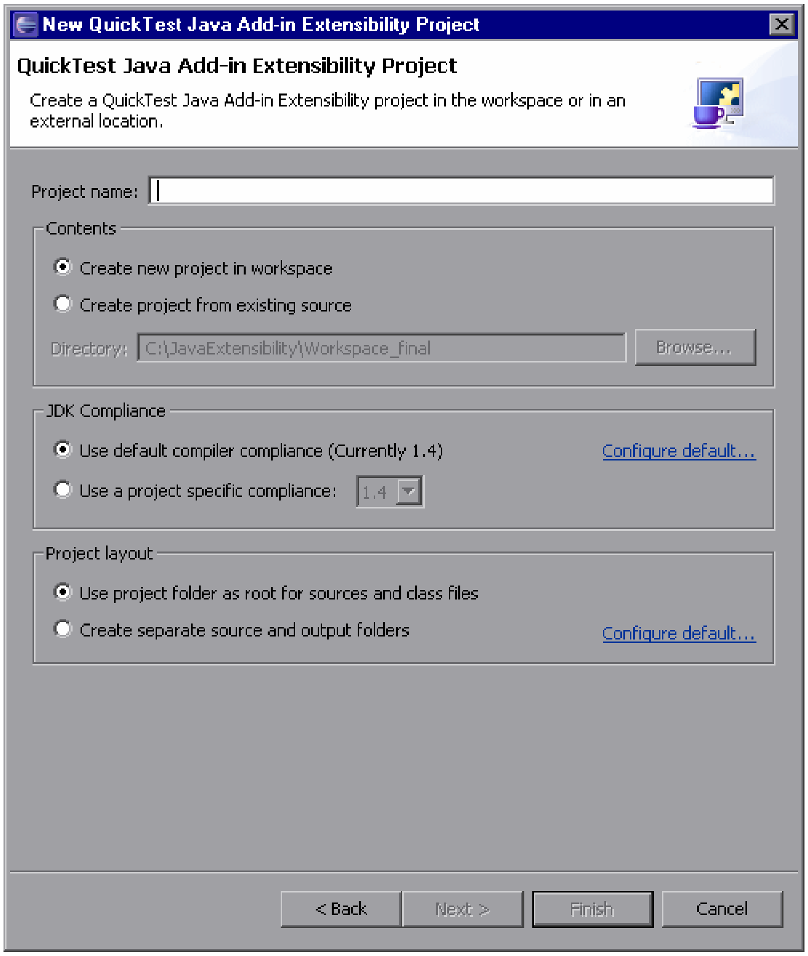
1. Choose **File** > **New** > **Project**. The New Project dialog box opens.
2. Expand the **QuickTest Professional** folder and select **QuickTest Java Add-in Extensibility Project**.

3 Click **Next**. The New QuickTest Java Add-in Extensibility Project Screen opens (described on page 87).

Tip: You can shorten this process by customizing Eclipse to provide **QuickTest Java Add-in Extensibility Project** as an option in the **New** menu. To do this, perform the following: Choose **Window > Customize Perspective**. In the Shortcuts tab in the dialog box that opens, select the **QuickTest Professional** and **QuickTest Java Add-in Extensibility Project** check boxes. Click **OK**.

QuickTest Java Add-in Extensibility Project Screen

In the QuickTest Java Add-in Extensibility Project screen, you create a QuickTest Java Add-in Extensibility project and define the project layout.



New QuickTest Java Add-in Extensibility Project

QuickTest Java Add-in Extensibility Project

Create a QuickTest Java Add-in Extensibility project in the workspace or in an external location.

Project name:

Contents

(●) Create new project in workspace

( ) Create project from existing source

Directory: C:\JavaExtensibility\Workspace_final Browse...

JDK Compliance

(●) Use default compiler compliance (Currently 1.4) Configure default...

( ) Use a project specific compliance: 1.4

Project layout

(●) Use project folder as root for sources and class files

( ) Create separate source and output folders Configure default...

< Back Next > Finish Cancel

Perform the following:

➤ In the **Project name** box, enter a name for the project.

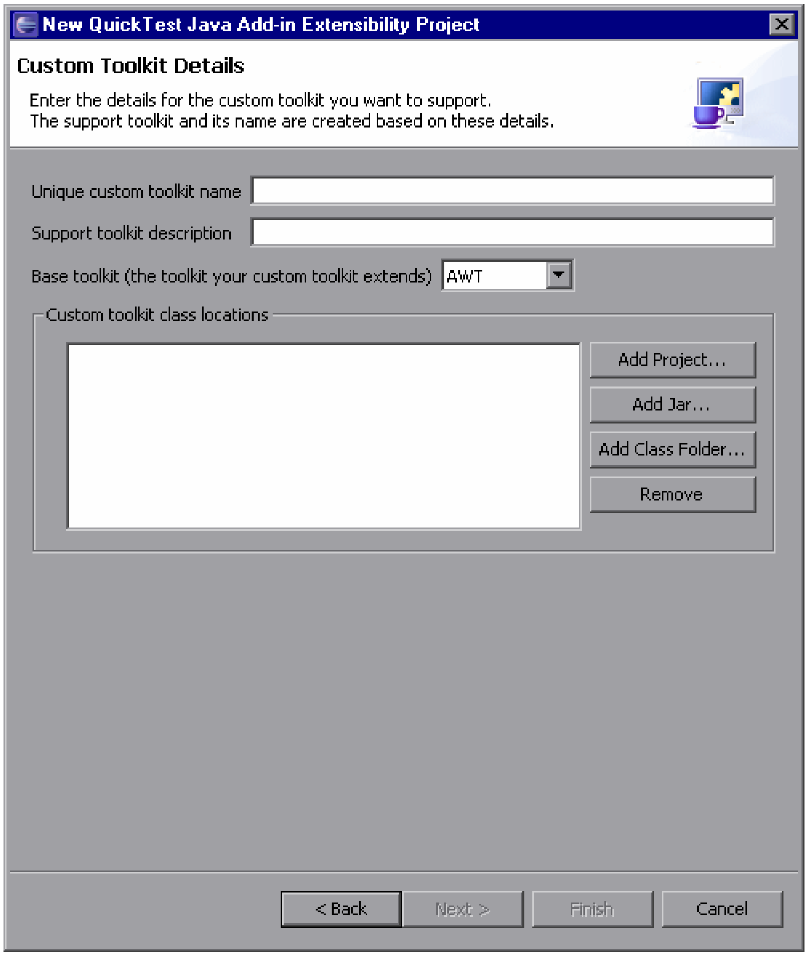
➤ In the **Project Layout** area, select **Create separate source and output folders**.

Click **Next** to continue to the Custom Toolkit Details Screen (described on page 89).

For information on the options available in this Eclipse wizard screen, refer to the *Eclipse Help*.

Custom Toolkit Details Screen

In the Custom Toolkit Details screen, you provide the details of the custom toolkit so that the wizard can generate a corresponding custom toolkit support set.


New QuickTest Java Add-in Extensibility Project

Custom Toolkit Details

Enter the details for the custom toolkit you want to support.
The support toolkit and its name are created based on these details.

Unique custom toolkit name

Support toolkit description

Base toolkit (the toolkit your custom toolkit extends) AWT

Custom toolkit class locations

Specify the following details:

➤ **Unique custom toolkit name.** A name that uniquely represents the custom toolkit for which you are creating support.

The name must begin with an English letter and contain only alphanumeric characters and underscores. When the wizard creates the new toolkit support class, it uses the name you specify for the custom toolkit, and adds the suffix, Support, to this name. For example, if you name the custom toolkit ImageControls, when the wizard creates the toolkit support class, it names it ImageControlsSupport.

Providing unique toolkit names enables a single QuickTest installation to support numerous custom toolkit support sets simultaneously.

Note: You cannot specify the name of a custom toolkit whose support is already deployed to QuickTest. If you want to create a new project using the wizard, and use this project to replace existing custom toolkit support, you must first manually delete the existing support. To do this, browse to **<QuickTest Installation Folder> bin\java\classes\extension**, delete the toolkit configuration file, and then use the **Reload Support Configuration** command described on page 138.

- **Support toolkit description.** A sentence describing the support toolkit. The description is stored in the toolkit configuration file.
- **Base toolkit.** The toolkit that the custom toolkit extends. A toolkit can be considered the base toolkit of a custom toolkit if all of the custom controls in the custom toolkit extend controls in the base toolkit.

  The **Base toolkit** list contains a list of toolkits for which QuickTest support already exists. After you create and deploy support for your own toolkits, they are displayed in the list as well.

  When the wizard creates the new custom toolkit support set, it creates a new toolkit support class. This new toolkit support class extends the toolkit support class of the base toolkit you select. As a result, the new custom toolkit support inherits all of the necessary utility methods for basic functionality (for example, event handling and dispatching) from the base toolkit support.

- **Custom toolkit class locations.** A list of the locations of the custom classes you want to support in this project. You can specify Eclipse projects, **.jar** files, and Java class folders (the file system folders containing the compiled Java classes).

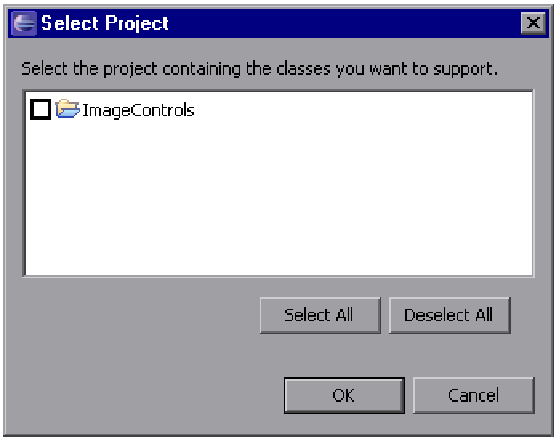
  When the new Java Add-in Extensibility project is built, these locations are added to the project build path.

Note: The Custom Class Selection Screen in the New QuickTest Custom Support Class Wizard (shown on page 99) displays the custom classes from the locations you list in this box. This enables you to select the required custom class when creating a custom support class. (You create custom support classes after the new Java Add-in Extensibility project is built.)

To add custom toolkit class locations to the list:

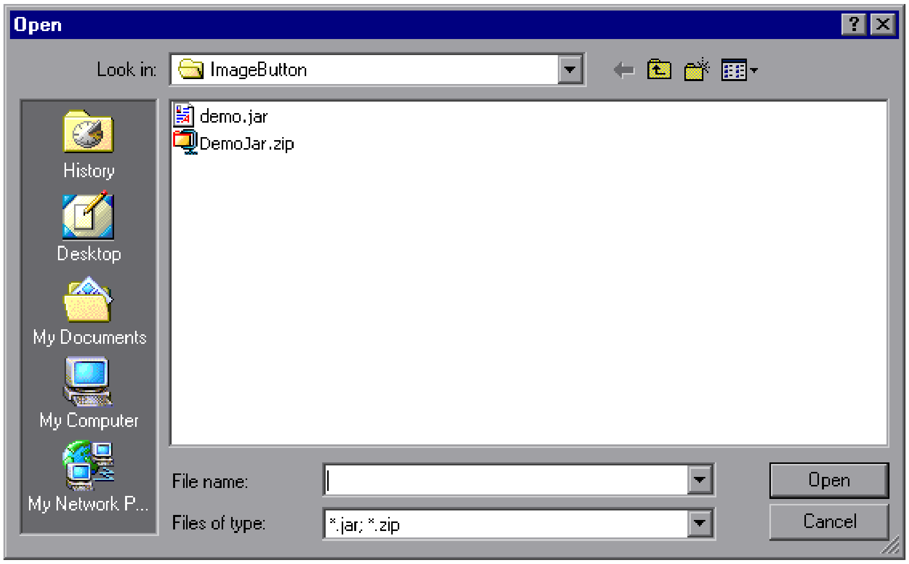
Add the locations of the custom toolkit classes using one or more of the following options:

- Click **Add project** to select an Eclipse project. The Select Project dialog box opens and displays the projects in the current Eclipse workspace.


Select Project
Select the project containing the classes you want to support.
ImageControls
Select All
Deselect All
OK
Cancel

  Select the check box for the appropriate project and click **OK** to add it to the **Custom toolkit class locations** box.

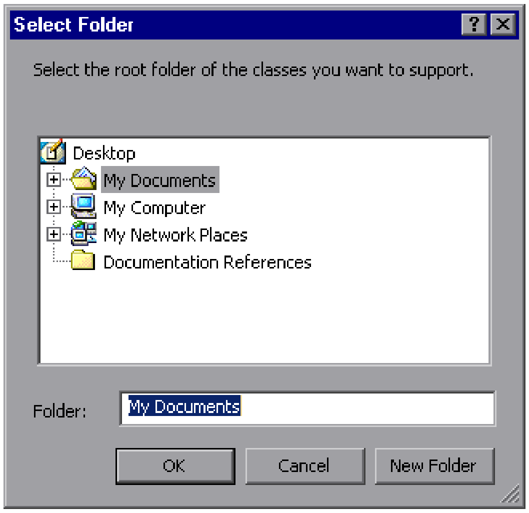
➤ Click **Add Jar** to add a Java archive (**.jar**) file. The Open dialog box opens.



demo.jar
DemoJar.zip
File name:
Files of type: *.jar; *.zip

Browse to the appropriate Java archive file, select it, and click **OK** to add it to the **Custom toolkit class locations** box.

➤ Click **Add Class Folder** to add a class folder. The Select Folder dialog box opens.



Select the root folder of the classes you want to support.

Desktop
My Documents
My Computer
My Network Places
Documentation References

Folder: My Documents

Browse to the appropriate folder, select it, and click **OK** to add it to the **Custom toolkit class locations** box.

Note: Select the root folder that contains the compiled class packages. For example, the file **ImageButton.java** defines the class **com.demo.ImageButton**. When you compile this class and store the result in the **bin** folder, the class file **ImageButton.class** location is: **bin\com\demo\ImageButton.class**. If you want to select the location of this class for the **Custom toolkit class locations**, select the **bin** folder.

To remove custom toolkit class locations from the list:

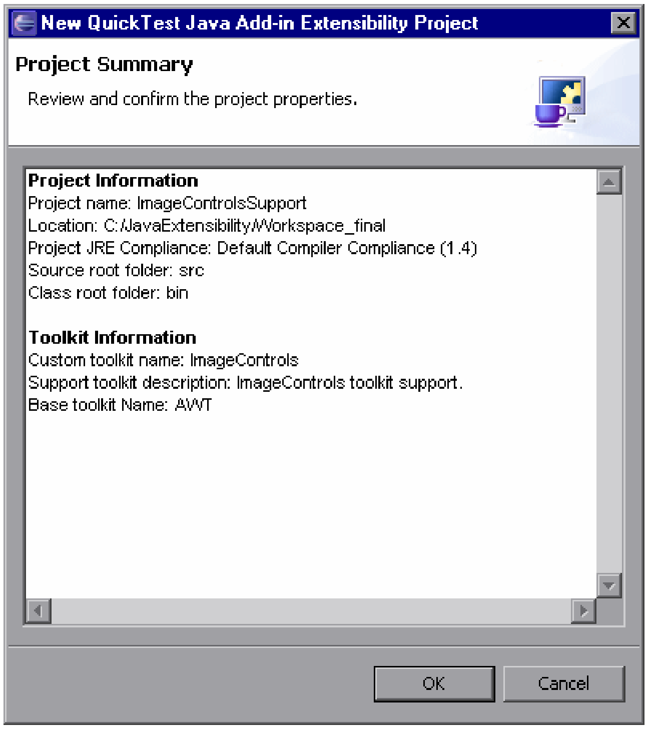
Select the location in the **Custom toolkit class locations** box and click **Remove**.

Note: To add or remove custom class locations in a Java Add-in Extensibility project after it is created, use the Properties dialog box for QuickTest Java Add-in Extensibility projects described on page 96.

After you add the locations of all of the custom classes included in the custom toolkit, click **Finish**. The Project Summary screen opens.

Project Summary Screen

Before the wizard creates the custom toolkit support files, the Project Summary screen summarizes the specifications you provided for the new Java Add-in Extensibility project.


New QuickTest Java Add-in Extensibility Project
Project Summary
Review and confirm the project properties.
OK
Cancel

Project Information
Project name: ImageControlsSupport
Location: C:/JavaExtensibility/Workspace_final
Project JRE Compliance: Default Compiler Compliance (1.4)
Source root folder: src
Class root folder: bin

Toolkit Information
Custom toolkit name: ImageControls
Support toolkit description: ImageControls toolkit support.
Base toolkit Name: AWT

Review the information. If you want to change any of the data, click **Cancel** to return to the Custom Toolkit Details Screen (described on page 89). Use the **Back** and **Next** buttons to open the relevant screens and make the required changes.

If you are satisfied with the definitions, click **OK**. The wizard creates new QuickTest Java Add-in Extensibility project, containing the following items:

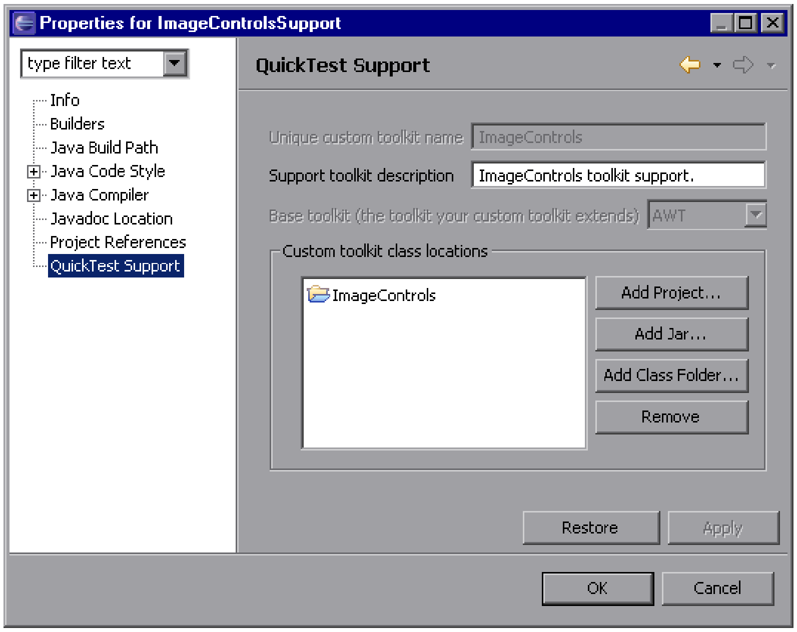
- ➤ The toolkit root package: **com.mercury.ftjadin.qtsupport.<Custom Toolkit Name>** containing:
  - ➤ The toolkit support class in the toolkit root package: **<Custom Toolkit Name>Support.java**

    For information on the content of this class, see "Understanding the Toolkit Support Class" on page 35.
  - ➤ The support class sub-package: **com.mercury.ftjadin.qtsupport.<Custom Toolkit Name>.cs**
- ➤ A folder for configuration files named **Configuration**. It contains:
  - ➤ The **<Custom Toolkit Name>.xml** toolkit configuration file. For information on the content of this file, see "Understanding the Toolkit Configuration File" on page 36.
  - ➤ The **TestObjects** folder for test object configuration files.

Note: If you have more than one Java Run-time Environment (JRE) installed on your computer, and one or more of the custom toolkit class locations you specified were Eclipse projects, make sure that the custom toolkit projects and the new Java Add-in Extensibility project are using the same JRE. If they are not, modify the JRE for one or more of the projects so that all of the projects use the same JRE.

Modifying QuickTest Java Add-in Extensibility Project Properties

In the Eclipse menu bar, choose **Project** > **Properties**. The Properties dialog box opens. In the left pane, select **QuickTest Support** from the list of property types. The **QuickTest Support** properties are displayed in the right pane.


Properties for ImageControlsSupport

QuickTest Support

Unique custom toolkit name: ImageControls

Support toolkit description: ImageControls toolkit support.

Base toolkit (the toolkit your custom toolkit extends): AWT

Custom toolkit class locations: ImageControls

For information on the options in this dialog box, see “Custom Toolkit Details Screen” on page 89.

After the Java Add-in Extensibility project is created, you cannot change the **Unique custom toolkit name** or the **Base toolkit**.

You can change the **Support toolkit description**. You can also add or remove locations in the **Custom toolkit class locations** list. When you modify this list, you must modify the project’s build path accordingly.

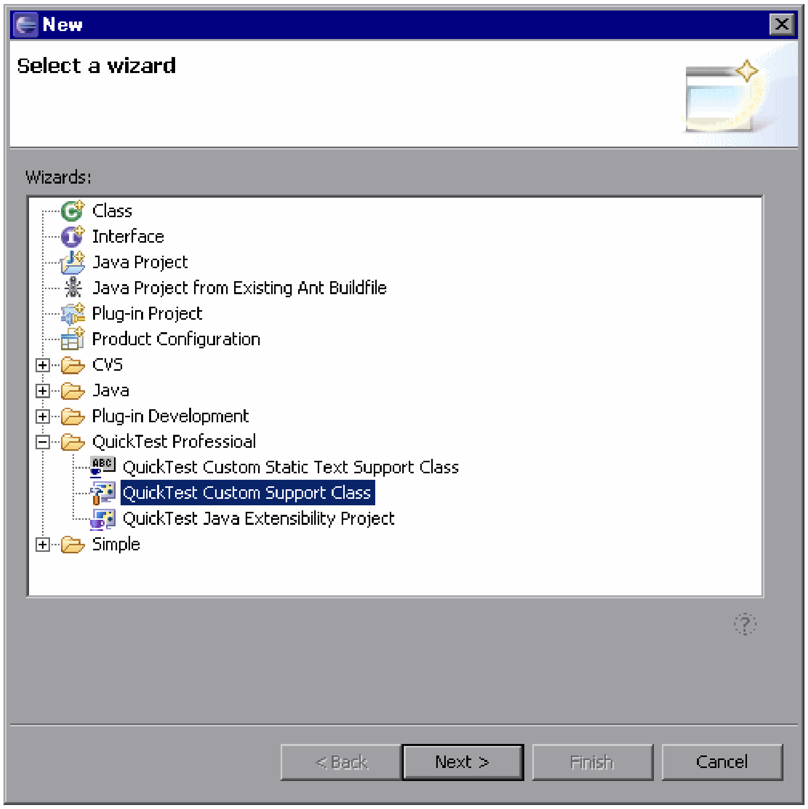
You can click the **Restore** button to restore the settings in this dialog box to the most recently saved values.

New QuickTest Custom Support Class Wizard

You use the New QuickTest Custom Support Class wizard to create each support class within a Java Add-in Extensibility project. After you specify the details of the custom class and the required QuickTest support, the wizard creates the support class and all of the necessary methods, accordingly. The wizard also provides method stubs for any additional methods you need to implement.

To open the New QuickTest Custom Support Class wizard in Eclipse:

1 In the Eclipse Package Explorer tab, select a QuickTest Java Add-in Extensibility project. Then choose **File** > **New** > **Other**. The New dialog box opens.



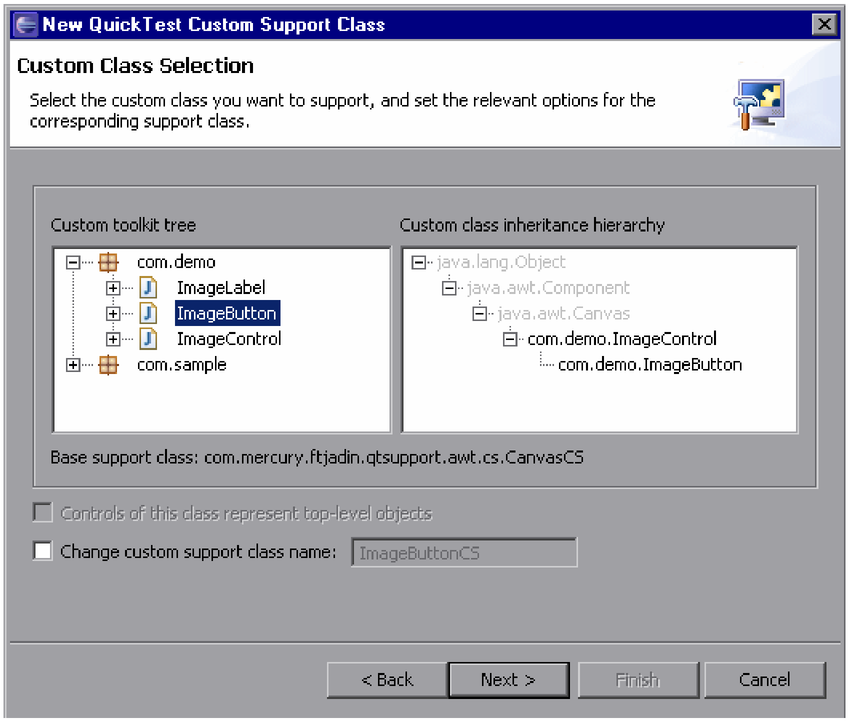
2 Expand the **QuickTest Professional** folder and select **QuickTest Custom Support Class**.

3 Click **Next**. The Custom Class Selection Screen opens.

Tip: You can shorten this process by customizing Eclipse to provide **QuickTest Custom Support Class** as an option in the **New** menu. To do this, perform the following: Choose **Window** > **Customize Perspective**. In the Shortcuts tab in the dialog box that opens, select the **QuickTest Professional** and **QuickTest Custom Support Class** check boxes. Click **OK**.

Custom Class Selection Screen

The Custom Class Selection screen is the first screen in the New QuickTest Custom Support Class wizard. In this screen, you select the custom class you want to support and set the relevant options. The wizard automatically determines which existing support class the new support class must extend, based on the custom class inheritance hierarchy.


New QuickTest Custom Support Class

Custom Class Selection

Select the custom class you want to support, and set the relevant options for the corresponding support class.

Custom toolkit tree

```
com.demo
    ImageLabel
    ImageButton
    ImageControl
com.sample
```

Custom class inheritance hierarchy

```
java.lang.Object
    java.awt.Component
        java.awt.Canvas
            com.demo.ImageControl
                com.demo.ImageButton
```

Base support class: com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS

Controls of this class represent top-level objects

Change custom support class name: ImageButtonCS

The main area of this screen contains the following options:

➤ **Custom toolkit tree.** Displays all of the classes in the custom toolkit that are candidates for support (taken from the custom toolkit class locations you listed in the New QuickTest Java Add-in Extensibility Project wizard). Use the expand (+) and collapse (-) signs to expand and collapse the tree, and to view its packages and classes.

Only classes that fulfill the following conditions are displayed:

- Classes that extend **java.awt.Component** or **org.eclipse.swt.widgets.Widget**.
- Classes for which QuickTest support has not yet been extended. If support for a custom class was previously deployed to QuickTest, or if support for a custom class is being developed in the current Eclipse project, the custom class does not appear in this tree.

Note: If you think a certain class meets all of the requirements above, but it still does not appear in the tree, try to update your environment by using the **Reload Support Configuration** command (described on page 138).

For example, if you delete custom support in an Eclipse Java Add-in Extensibility project to create new support for the same custom control, you must reload the support configuration. This enables the custom class to appear in the **Custom toolkit tree**.

- **Custom class inheritance hierarchy.** Displays the inheritance hierarchy of the class selected in the **Custom toolkit tree**. Gray nodes indicate classes that are not included in this toolkit. Black nodes indicate classes that are part of the custom toolkit.

  You can select the custom class you want to extend in the **Custom toolkit tree** or the **Custom class inheritance hierarchy**. (In the **Custom class inheritance hierarchy** you can select only black nodes, and only classes that do not have QuickTest support.)

- **Base support class.** The support class that the new support class must extend. You cannot modify this information. The wizard selects the support class of the closest ancestor in the hierarchy that has QuickTest support. (If support for a custom class was previously deployed to QuickTest, or if support for a custom class is being developed in the current Eclipse project, the wizard recognizes the custom class as having QuickTest support.)

Note: When QuickTest recognizes a Java object that is not mapped to a specific support class, it uses the support class mapped to the object's closest ancestor. Therefore, the base support class is the class that would provide support for the custom control if it were not mapped to a specific support class. In the new custom support class, you need to implement (or override) only the support that the base support class does not adequately provide.

You can use the information displayed in the **Custom class inheritance hierarchy** and **Base support class** to help you decide whether you should first extend support for another custom class, higher in the hierarchy. Before you decide, consider the following:

➤ Is there a custom class higher in hierarchy that does not have QuickTest support?

➤ If so, does the custom class have elements that need to be supported in a similar manner for more than one of its descendants?

If you answered "yes" to the above, consider creating support for the higher class first. This will enable its support class to be used as the **Base support class**. If the class is displayed as a black node in the hierarchy, you can select it in this screen and create support for it in this session of the wizard. If the class appears as a gray node, it is not part of this toolkit, and you cannot create support for it within the current QuickTest Java Add-in Extensibility project.

If the higher class extends the base toolkit of the current support project, you can add it to the scope of this project by adding it to the custom toolkit. For information on base toolkits, see "Custom Toolkit Details Screen" on page 89. For information on adding a custom class to an existing support project, see "Modifying QuickTest Java Add-in Extensibility Project Properties" on page 96.

Otherwise, if you want to create support for the higher class first and then use its support class as a base support class, you must perform the procedure described below.

To create support for a higher class that is not part of this custom toolkit and use this support as a base support class:

1. Create support for the higher class in another QuickTest Java Add-in Extensibility project.
2. Deploy the support to QuickTest.
3. Reopen the original QuickTest Java Add-in Extensibility project. Choose **QuickTest** > **Reload Support Configuration** or click the **Reload Support Configuration** button.
4. Open the New QuickTest Custom Support Class wizard (described on page 97). The wizard now selects the new support class you created as the **Base support class**.

Note: Selecting the class to support is fundamental to creating a custom support class. If you make changes in later screens and then return to this screen and select a different class, those changes will be discarded.

The bottom of the Custom Class Selection screen contains the following options:

➤ **Controls of this class represent top-level objects.** Enables you to specify that QuickTest may be expected to recognize the control as the highest Java object in the test object hierarchy. For more information see, “Supporting Top-Level Objects” on page 59.

If you select this check box, the wizard implements the **isWindow** method in the new custom support class. This method returns **true**.

This option is available only if the class you selected to support is a container class, meaning that it extends **java.awt.container** or **org.eclipse.swt.widgets.Composite**. The check box is selected by default if the new support class extends one of the following support classes: **ShellCS** (SWT), **WindowCS** (AWT), **AppletCS** (AWT).

- **Change custom support class name.** Enables you to modify the default name the wizard provided for the support class, if needed.

  By default, the name for a support class is **<custom class name>CS**. In most cases, there is no need to change the default name. However, if your custom toolkit contains classes from different packages, you might have more than one custom class with the same name. In this case, you must provide different names for the custom support classes because they are stored in one package.

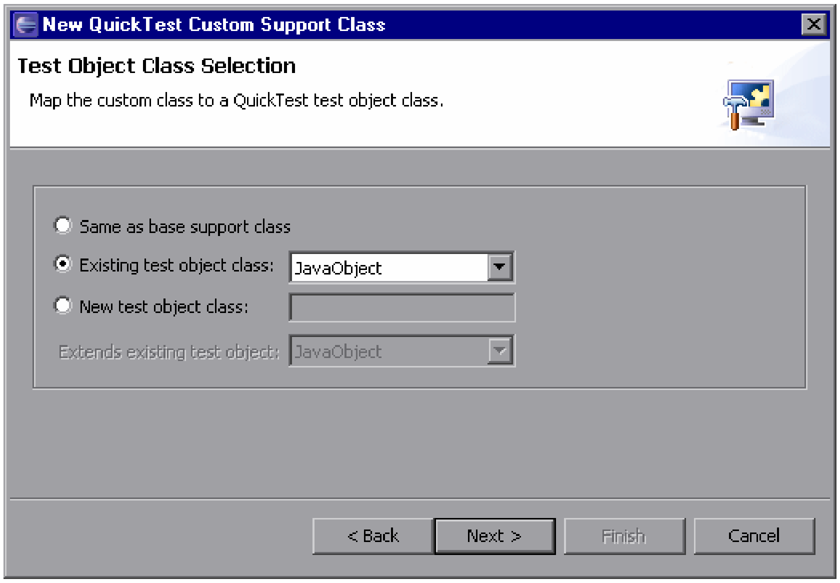
  To modify the custom support class name, select the **Change custom support class name** check box and then enter the new name.

Note: The options in the Custom Class Selection screen are identical to the options available in the Custom Static-Text Class Selection screen in the New QuickTest Custom Static-Text Support Class wizard (described on page 132).

Click **Next** to continue to the Test Object Class Selection Screen, described below.

Test Object Class Selection Screen

In QuickTest tests, the custom class controls are represented by test objects of the selected test object class. In the Test Object Class Selection screen, you map the custom class to a test object class. In the custom support class, the wizard adds a **to_class_attr** property method that is implemented to return the test object class you select in this screen. This enables the support class to inform QuickTest what test object class is mapped to the custom class.


New QuickTest Custom Support Class

Test Object Class Selection

Map the custom class to a QuickTest test object class.

( ) Same as base support class
(•) Existing test object class: JavaObject
( ) New test object class:
Extends existing test object: JavaObject

Select one of the following options:

➤ **Same as base support class.** Maps the custom class to the test object class returned by the **to_class_attr** property method of the base support class. (If you select this option, the wizard does not add a **to_class_attr** method to the new support class that it creates. The new support class inherits the base support class' method.)

In the Custom Class Selection Screen (described on page 99), you determined the base support class, which is the support class that the new support class extends. The custom class supported by the base support class is mapped to a specific test object class. If this test object class is also a logical test object for your custom class, select the **Same as base support class** option.

The following examples illustrate when to select the **Same as base support class** option:

- ➤ You want to support a custom control that is similar to the one supported by the base support class. Controls are considered similar if they have the same set of identification properties and test object methods, but the properties and methods are implemented differently. In this case, the test object class returned by the **to_class_attr** property method of the base support class is appropriate for your custom control.
- ➤ You are creating a support class for other support classes to extend—not to support actual controls. In this case, you can select this option because it is not important which test object class you map to the custom class. To view an example of this type, see "Creating Support for the ImageControl Custom Class" on page 213.

- ➤ **Existing test object class.** Enables you to map the custom class to an existing test object class that is already supported by QuickTest. This list contains all of the Java objects that QuickTest supports. If you define new test object classes for custom support, they are also included in the list.

  If you select this option, you must also select the appropriate existing test object class from the list.

Notes:

If you defined new test object classes in the current Eclipse workspace, they are displayed in this list immediately. Otherwise, new test object classes are displayed in the list only after they are deployed to QuickTest and you reload the configuration (for more information, see "Reload Support Configuration" on page 138).

If you select a test object class that is not defined within your project, its test object class definition must also be deployed to QuickTest for your support to function properly.

Tip: Select this option only if this test object class includes all of the identification properties and test object methods of the custom control. If you need to add additional properties or methods, select **New test object class**.

- **New test object class.** Enables you to map the custom control to a new test object class that you create. Select this option if none of the existing test object classes include all of the identification properties and test object methods of the custom control. Then enter a name for the new test object class. The test object class name must begin with a letter and contain only alphanumeric characters and underscores.

  - **Extends existing test object.** Each new test object class is based on an existing one, extending its set of identification properties and test object methods. All test object classes extend the JavaObject class. You can choose a more specific existing test object class to extend by selecting it from the list.

    The list of existing test objects contains all of the Java objects that QuickTest supports. If you define new test object classes for custom support, they are included in the list as well.

    Note: If you defined new test object classes in the current Eclipse workspace, they are displayed in this list immediately. Otherwise, new test object classes are displayed in the list only after they are deployed to QuickTest and you reload the configuration (for more information, see "Reload Support Configuration" on page 138).
    If you select a test object class that is not defined within your project, its test object class definition must also be deployed to QuickTest for your support to function properly.

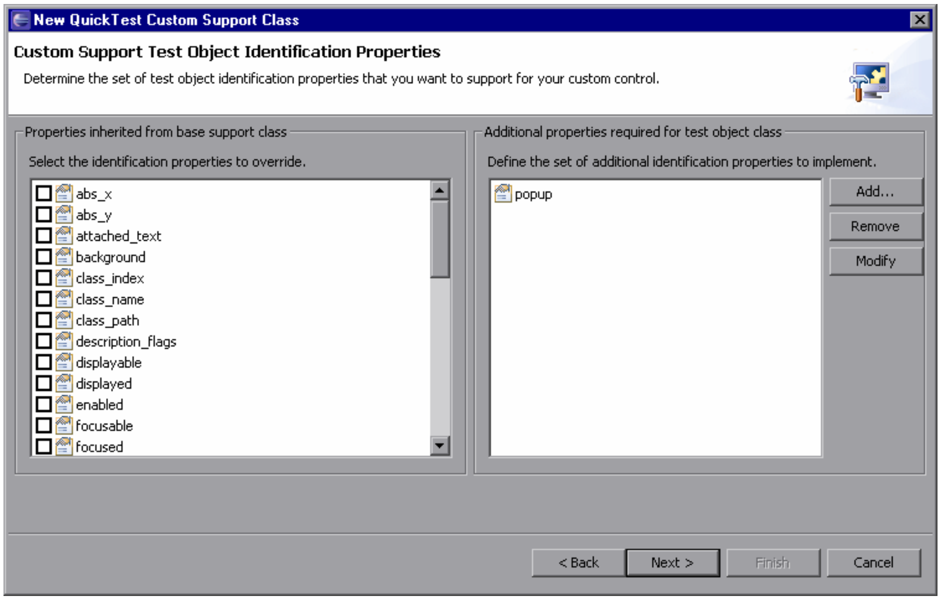
If you select the **New test object class** option, you define additional details about the new test object class in the New Test Object Class Details Screen (described on page 128). The wizard then adds the definition of the new test object class to the test object configuration file. For information on the structure and content of this file, refer to the *QuickTest Test Object Schema Help* (**Help** > **QuickTest Professional Help** > **QuickTest Advanced References** > **QuickTest Test Object Schema**).

Note: Selecting the test object class to map to the custom class is fundamental to creating a custom support class. If you make changes in later screens and then return to this screen and select a different test object class, those changes will be discarded.

Click **Next** to continue to the Custom Support Test Object Identification Properties Screen, described below.

Custom Support Test Object Identification Properties Screen

The Custom Support Test Object Identification Properties screen displays the identification properties supported by the base support class you are extending, as well as additional properties that are defined in the test object class you selected, but are not yet supported. It enables you to select properties whose support you want to implement or override with new functionality. It also enables you to add new properties.



New QuickTest Custom Support Class

Custom Support Test Object Identification Properties

Determine the set of test object identification properties that you want to support for your custom control.

Properties inherited from base support class

Select the identification properties to override.

- [ ] abs_x
- [ ] abs_y
- [ ] attached_text
- [ ] background
- [ ] class_index
- [ ] class_name
- [ ] class_path
- [ ] description_flags
- [ ] displayable
- [ ] displayed
- [ ] enabled
- [ ] focusable
- [ ] focused

Additional properties required for test object class

Define the set of additional identification properties to implement.

popup

Properties Inherited from Base Support Class

The left pane displays all of the identification properties implemented by the base support class. These are the identification properties that will be inherited by the support class you are creating. You can select any identification properties whose support you want to override with a different implementation.

Note: Some of these identification properties are not included in the test object class definition. Therefore, they are not displayed in QuickTest in the Object Spy or in the Checkpoint Properties dialog box. You can access those identification properties by using the **GetROProperty** method. For more information on the **GetROProperty** method, refer to the *QuickTest Professional Object Model Reference*.

When the wizard creates the support class file, it adds a support method stub, named **<identification property name>_attr**, for each of the identification properties you select. The support method stubs return the same values as the support methods in the base support class. You can implement the new support methods to match the needs of your custom control.

Additional Properties Required for Test Object Class

The right pane displays the identification properties that are defined in the test object class you selected, but are not supported by the base support class. You can modify this list using the **Add**, **Remove**, and **Modify** buttons.

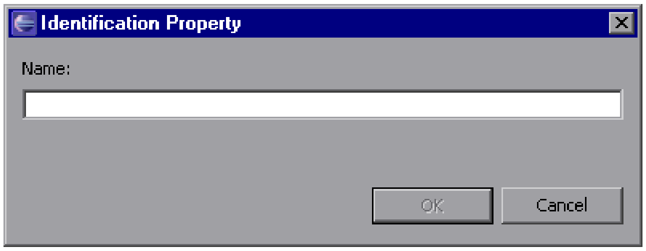
For each of the identification properties in this pane, the wizard adds a support method stub to the support class it creates. The support method stubs return **null** until you implement them to match the needs of your custom control.

If you add identification properties to this list, the wizard adds them to the test object class definition in the test object configuration file. For information on the structure and content of this file, refer to the *QuickTest Test Object Schema Help* (**Help** > **QuickTest Professional Help** > **QuickTest Advanced References** > **QuickTest Test Object Schema**).

Note: If you selected the **Same as base support class** option in the Test Object Class Selection Screen(on page 104), the wizard does not know which test object class is mapped to the custom control. As a result, no identification properties are displayed in the right pane. If you add an identification property, the wizard adds the appropriate support method stub to the support class it creates. However, the identification property is not added to any test object class definition.

To add an identification property:

1 Click **Add**. The Identification Property dialog box opens.



2 Enter a name for the new identification property and click **OK**. (The identification property name must begin with a letter and contain only alphanumeric characters and underscores.)

Note: If you add identification properties to this list, they are added to the test object class definition. This means that the new properties appear in the list of identification properties in QuickTest for all test objects of this class.

Therefore, if you plan to add properties, it is recommended to create a new test object class based on the existing one, instead of using the existing test object class.

To remove an identification property:

1 In the **Additional properties required for test object class** pane, select the property you want to remove.

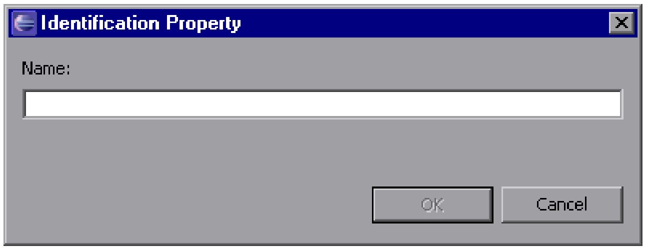
2 Click **Remove**. Then click **Yes** to confirm.

Note: If you remove an identification property from the list, it is no longer supported for this custom class. However, it is still part of the test object class definition. Therefore, although it still appears in the list of identification properties shown in the QuickTest Object Spy, it will have no value.

To modify an identification property:

1 In the **Additional properties required for test object class** pane, select the property you want to rename.

2 Click **Modify**. The Identification Property dialog box opens.



3 Modify the identification property name and click **OK**.

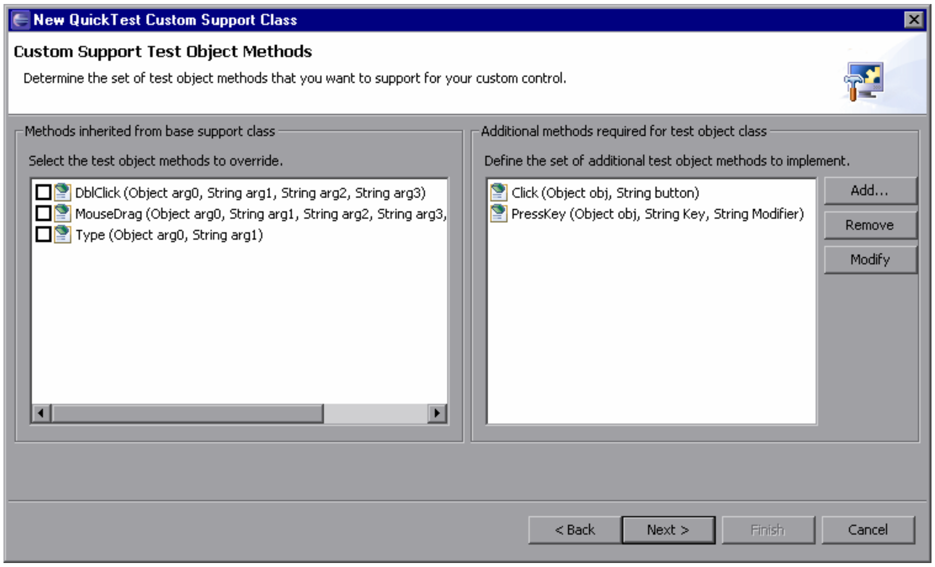
Note: When you remove an identification property from the existing test object class, the property is no longer supported for the custom class, but is still part of the test object class. When you add an identification property, it is added to the test object class definition, and will appear in QuickTest for all test objects of this class. Modifying an identification property is equivalent to removing it and adding a new one.

Tip: To add identification properties after the support class is created, use the **Add Identification Property** button or choose **QuickTest** > **Add Identification Property**.

Click **Next** to continue to the Custom Support Test Object Methods Screen, described below.

Custom Support Test Object Methods Screen

The Custom Support Test Object Methods screen displays the test object methods defined for the test object class you mapped to the custom control. You use this screen to select test object methods whose support you want to implement or override with new functionality and to add new test object methods.


New QuickTest Custom Support Class

Custom Support Test Object Methods

Determine the set of test object methods that you want to support for your custom control.

Methods inherited from base support class

Select the test object methods to override.

- [ ] DblClick (Object arg0, String arg1, String arg2, String arg3)
- [ ] MouseDrag (Object arg0, String arg1, String arg2, String arg3,
- [ ] Type (Object arg0, String arg1)

Additional methods required for test object class

Define the set of additional test object methods to implement.

- Click (Object obj, String button)
- PressKey (Object obj, String Key, String Modifier)

Methods Inherited from Base Support Class

The left pane displays all of the test object methods that are defined for the test object class you selected and are implemented by the base support class. These are the test object methods that will be inherited by the support class you are creating. You select any test object methods whose support you want to override with a different implementation.

When the wizard creates the support class file, it adds a support method stub, named **<test object method name>_replayMethod**, for each test object method you selected. The support method stubs return the same values as the support methods in the base support class. You can implement the new support methods to match the needs of your custom control.

Note: If you selected the **Same as base support class** option in the Test Object Class Selection Screen (on page 104), the wizard does not know which test object class is mapped to the custom control. As a result, no test object methods are displayed in the left pane. After the wizard creates the new support class, you can override any of the replay methods that it inherits from the base support class by adding them to the class manually.

Additional Methods Required for Test Object Class

The right pane displays the test object methods that are defined in the test object class you selected, but are not supported by the base support class.

You can modify the list in this pane using the **Add**, **Remove**, and **Modify** buttons.

The **Add** and **Modify** buttons enable you to add test object methods to this list or to modify the methods that are already in the list. Note that modifying the name of a method is equivalent to removing the method and adding a new one. For more information, see “Understanding the Test Object Method Dialog Box” on page 116.

For each of the test object methods in this pane, the wizard adds support method stubs to the support class it creates. The support method stubs return the error value **Retval.NOT_IMPLEMENTED** until you implement them to match the needs of your custom control.

If you add test object methods to this list, the wizard adds them to the test object class definition in the test object configuration file. For information on the structure and content of this file, refer to the *QuickTest Test Object Schema Help* (**Help** > **QuickTest Professional Help** > **QuickTest Advanced References** > **QuickTest Test Object Schema**).

Notes:

- If you add test object methods to this list, they are added to the existing test object class. This means that the new methods appear in QuickTest for all test objects of this class, regardless of whether or not they are supported for these objects. In a QuickTest test, if you call a test object method for an object, and that method is not supported, a run-time error occurs.
  Therefore, if you plan to add test object methods to support a custom control, it is recommended to create a new test object class based on the existing one, instead of using the existing test object class.
- If you selected the **Same as base support class** option in the Test Object Class Selection Screen (on page 104), the wizard does not know which test object class is mapped to the custom control. As a result, no test object methods are displayed in the right pane. If you add a test object method, the wizard adds the appropriate replay method stub to the support class it creates. However, the test object method is not added to any test object class definition.

To remove a test object method from the list:

1. In the **Additional methods required for test object class** pane, select the test object method you want to remove.
2. Click **Remove**. Then click **Yes** to confirm.

Note: If you remove a test object method from the list, it is no longer supported for this custom class. However, it is still part of the test object class definition. Therefore, it still appears in the list of test object methods in QuickTest.

If you use this test object method on a custom control in QuickTest tests, a run-time error occurs. For example, although a **drop-down-list** control is supported as a **List** test object, if you select the **select_range** test object method for a **drop-down-list** control, and it is not supported, a run-time error occurs.

Tip: To add test object methods after the support class is created, use the **Add Test Object Method** button or choose **QuickTest** > **Add Test Object Method**.

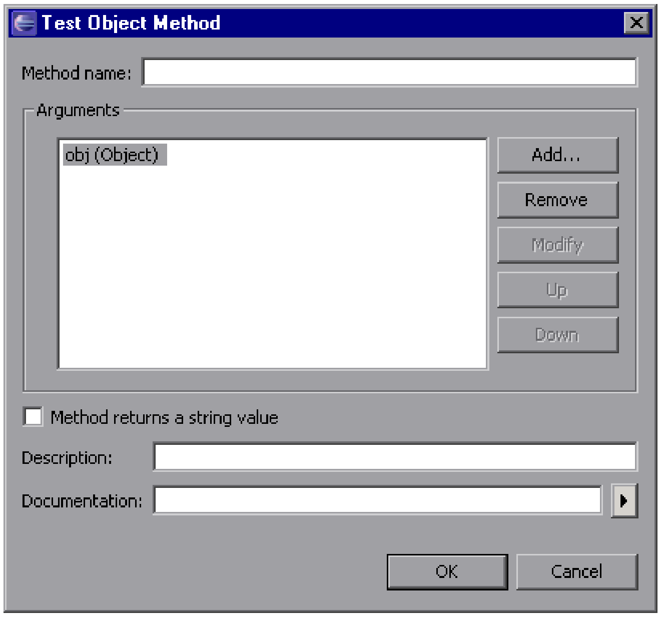
After you specify the custom support test object methods, click **Next**. One of the following screens open:

- If you are creating support for an AWT-based custom control, the Custom Control Recording Support Screen (described on page 121) opens.
- If you are creating support for an SWT-based custom control, and you mapped a new test object class to the custom control, the New Test Object Class Details Screen (described on page 128) opens.
- If neither of the previous conditions is met, the Custom Control Support Class Summary Screen (described on page 130) opens.

Understanding the Test Object Method Dialog Box

When you click **Add** or **Modify** in the Custom Support Test Object Methods Screen (described on page 112), the Test Object Method dialog box opens.

The Test Object Method dialog box enables you to specify details for the test object methods listed in the **Additional methods required for test object class** pane in the Custom Support Test Object Methods screen.



The Test Object Method dialog box contains the following items:

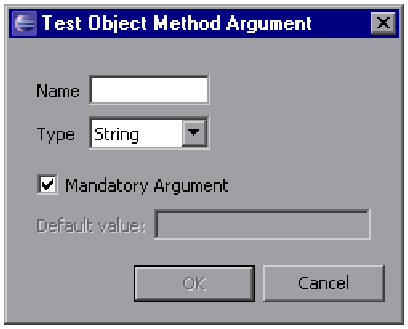
Option	Description
Method name	The name of the test object method as it appears in QuickTest tests. The name should clearly indicate what the test object method does so that a user can select it from the Step Generator or in the Keyword View. Method names cannot contain non-English letters or characters. In addition, method names must begin with a letter and cannot contain spaces or any of the following characters: ! @ # $ % ^ & * ( ) + = [ ] \ { } | ; ' : "" , / < > ? **Notes:** - Do not use the name of a test object method that already exists in the support class. (The Custom Support Test Object Methods Screen lists the test object methods that are already included in the support class.) If you want to override the implementation of an existing test object method, select it in the left pane of the Custom Support Test Object Methods screen (rather than creating a new test object method with the same name). - Modifying the name of a method is equivalent to removing the method and adding a new one.

Option	Description
Arguments	A list of the test object method arguments and their types. Use the following buttons to modify the list: • **Add.** Enables you to add additional arguments to the test object method. For more information, see "Adding or Modifying an Argument for a Test Object Method," below. • **Remove.** Removes the selected argument from the list. • **Modify.** Enables you to modify the arguments of the test object method. For more information, see "Adding or Modifying an Argument for a Test Object Method," below. • **Up.** Moves the selected argument up in the list. • **Down.** Moves the selected argument down in the list. **Notes:** - The first argument of every test object method must be **obj (Object)**. You cannot remove, modify, or move this argument. - You cannot modify the signature of a test object method that belongs to the existing test object class that you selected in the Test Object Class Selection Screen (described on page 104). (This means that in an existing test object method, you cannot add or remove arguments, or change their types.)

Option	**Description**
Method returns a string value	Indicates that this test object method returns a string value in addition to the return code. (The return value can be retrieved and used in later steps of a QuickTest test.) If you select this check box: • the wizard adds the **ReturnValueType** element to the test object method definition that it creates in the test object configuration file. • the method stub that the wizard creates in the new support class, returns the object **Retval ("")**, which includes the return code **OK** and an empty string. When you implement the replay method for this test object method, you can use different types of **Retval**. If everything is ok, return the string value. Otherwise, return only the relevant error code. For more information, refer to the *QuickTest Java Add-in Extensibility API Reference* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility API Reference**).
Description	The tooltip that is displayed when the cursor is positioned over the test object method in the Step Generator, in the Keyword View, and when using IntelliSense.
Documentation	A sentence that describes what the step that includes the test object method actually does. This sentence is displayed in the **Step documentation** box in the Step Generator and in the **Documentation** column of the Keyword View. You can insert arguments in the **Documentation** text by clicking ▸ and selecting the relevant argument. The arguments are then replaced dynamically by the relevant values.

Adding or Modifying an Argument for a Test Object Method

When you click **Add** or **Modify** in the Test Object Method dialog box, the Test Object Method Argument dialog box opens. The Test Object Method Argument dialog box enables you to specify the details for each of the arguments you list in the Test Object Method dialog box.



The Test Object Method Argument dialog box contains the following items:

Option	Description
Name	The name of the argument as it appears in QuickTest tests. The argument name should clearly indicate the value that needs to be entered for the argument. Argument names must contain only alphanumeric characters. In addition, argument names must begin with a letter and cannot contain spaces or any of the following characters: ! @ # $ % ^ & * ( ) + = [ ] \ { } | ; ' : "" , / < > ?

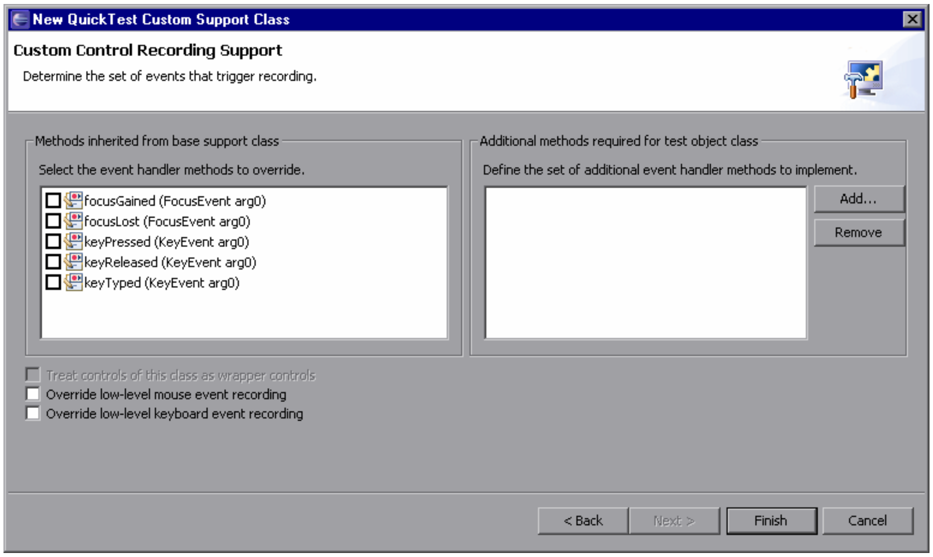
Option	Description
Type	Instructs QuickTest to do one of the following: • Require **String** values for this argument in test steps with this test object method • Allow **Variant** values Even if you define the **Type** as **Variant**, all arguments are passed to the replay methods as strings. In addition, when you record test steps, the arguments are always registered as strings. **Note:** If you want to define a list of possible values for an argument, you must do so manually. In the test object configuration file, define the list of values and change the argument's type to **ListOfValues**. For more information, refer to the *QuickTest Test Object Schema Help* (**Help** > **QuickTest Professional Help** > **QuickTest Advanced References** > **QuickTest Test Object Schema**).
Mandatory Argument	Instructs QuickTest whether or not to require the person writing the test to supply a value for the argument. In the list of arguments, mandatory arguments cannot follow optional arguments.
Default value	If an argument is optional, you can provide a default value that QuickTest uses if no other value is defined. This option is not available for mandatory arguments.

Custom Control Recording Support Screen

Note: The Custom Control Recording Support screen opens only if you are creating a support class for an AWT-based custom class.

To support recording on a custom control, the support class must implement listeners for the events that trigger recording.

The Custom Control Recording Support screen displays the event handler methods implemented by the support class you selected to extend.


New QuickTest Custom Support Class

Custom Control Recording Support

Determine the set of events that trigger recording.

Methods inherited from base support class

Select the event handler methods to override.

- [ ] focusGained (FocusEvent arg0)
- [ ] focusLost (FocusEvent arg0)
- [ ] keyPressed (KeyEvent arg0)
- [ ] keyReleased (KeyEvent arg0)
- [ ] keyTyped (KeyEvent arg0)

Additional methods required for test object class

Define the set of additional event handler methods to implement.

- [ ] Treat controls of this class as wrapper controls
- [ ] Override low-level mouse event recording
- [ ] Override low-level keyboard event recording

The Custom Control Recording Support screen enables you to:

- ➤ select methods whose implementation you want to override with new functionality
- ➤ add new event listeners to implement
- ➤ set recording-related options

For information about how the wizard implements the details you specify in this screen, see "Understanding What the Wizard Adds to the Support Class" on page 125.

Methods Inherited from Base Support Class

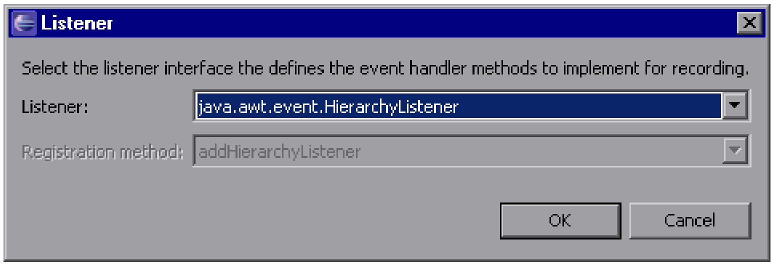
The left pane displays the event handler methods implemented by the base support class. You can select the methods you want to override.

Additional Methods Required for Test Object Class

In the right pane, you specify the listeners you want to add for the new support class. Each listener you select implies a set of event handler methods you can implement.

To add a listener to the list:

1 Click **Add** and select the appropriate listener from the Listener dialog box that opens.



Select the listener interface the defines the event handler methods to implement for recording.

Listener: java.awt.event.HierarchyListener

Registration method: addHierarchyListener

Note: The list contains the listeners that can be registered on the custom control. The wizard compiles this list by identifying listener registration methods in the custom class and its superclasses. The wizard identifies as registration methods, only methods named **add<XXX>Listener** whose first argument extends **java.util.EventListener**. If your custom class uses a registration method that does not comply with this definition, you cannot add the corresponding listener using the wizard. You can implement the required support manually after the wizard creates the new custom support class.

---

2 If the selected listener has more than one registration method, select a method from the **Registration method** list.

3 Click **OK**. The listener you selected and all of the event handler methods it includes are added to the list.

To remove a listener from the list:

Select a listener or one of its event handler methods and click **Remove**.

Custom Control Recording Support Screen Options

The Custom Control Recording Support screen contains the following options:

Option	Description
Treat controls of this class as wrapper controls	Instructs the wizard to implement the **com.mercury.ftjadin.infra.abstr.RecordWrapper** interface in the new support class. If the custom control extends **java.awt.container**, this check box is selected by default. Otherwise, it is not available. For more information, see "Wrapper Implementation in the Support Class" on page 126.
Override low-level mouse event recording	Instructs the wizard to implement the **mouseRecordTarget** method in the new support class so that it returns **null**. This instructs QuickTest not to record low-level mouse events (coordinate-based operations), so you can record more complex operations, such as, selecting an option in a menu.
Override low-level keyboard event recording	Instructs the wizard to implement the **keyboardRecordTarget** method in the new support class, so that it returns **null**. This instructs QuickTest not to record low-level keyboard events, enabling you to record more complex events, such as, setting a value in an edit box.

Note: The options listed in the table above are available only in the wizard (and not in the Eclipse QuickTest commands that you can use to edit a support class after it is created). If you do not select these options when you create the support class, and you want to implement them later, you will have to do so manually.

Tip: To add event handlers after the support class is created, use the **Add Event Handler** button or choose **QuickTest** > **Add Event Handler**.

Understanding What the Wizard Adds to the Support Class

The following sections describe the methods that the wizard adds to the support class it creates, based on the definitions in this screen:

Listener Implementation in the Support Class

In the support class file it creates, the wizard implements the listeners and options you specified, as follows:

- The implemented listener interfaces are added to the support class signature.
- A constructor is added to the support class, listing all of the listeners that need to be registered on the custom control. It also lists the methods used to add and remove the listeners. This is done by calling **addSimpleListener** for each listener.
- A method stub is added to the support class for each of the event handler methods you selected in the left pane. The method stubs call the corresponding event handler methods in the base support class. You can implement the new event handler methods to match the needs of your custom control.

Note: Some of the event handler methods are implemented in existing support classes as final methods, which cannot be overriden. If you select one of these methods in the left pane, the wizard adds an underscore at the beginning of the method name in the method stub that it creates. For example, if you select **focusGained**, **focusLost**, **keyTyped**, **keyPressed**, or **keyReleased**, the wizard creates **_focusGained**, **_focusLost**, **_keyTyped**, **_keyPressed**, or **_keyReleased**, respectively. Each one of the final methods is implemented to call **_<method name>** after performing its basic functionality. Therefore, you can override the **_<method name>** methods to add functionality to these final methods.

- A method stub is added to the support class for each of the event handlers listed in the right pane. You must implement the event handler methods to call **MicAPI.record**. (Each method stub includes a comment to remind you to do this, and a basic skeleton which provides a recommendation for the method's structure.) For more information, see "Supporting the Record Option" on page 56.

Wrapper Implementation in the Support Class

You select the **Treat controls of this class as wrapper controls** check box if you are creating support for a container control that groups the controls within it and represents them as a single control. If you select this check box, the wizard adds the following method stubs to the support class:

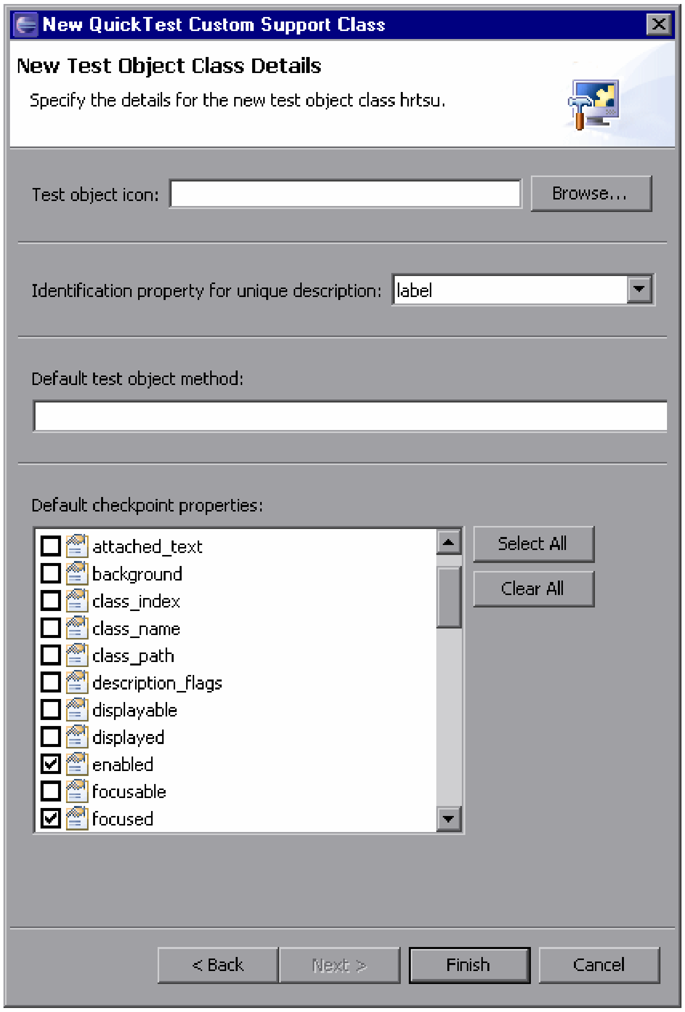
- **blockWrappedObjectRecord.** (Returns **False**.)
- **registerWrapperInspector.** (A comment is added to remind you to implement this method to register this class as a wrapper of specific control types.)
- **checkWrappedObject.** (Returns **null**.)
- **wrapperRecordMessage.** (Returns the record message sent by the wrapped control without performing any intervention.)

You can implement these methods to achieve the required wrapping functionality. For more information, see "Supporting Wrapper Controls" on page 59.

If you mapped a new test object class to the custom control, click **Next** to continue to the New Test Object Class Details Screen (described below). Otherwise, click **Finish** to continue to the Custom Control Support Class Summary Screen (described on page 130).

New Test Object Class Details Screen

If you mapped a new test object class to the custom control, you define additional details about the new test object class in the New Test Object Class Details screen.



New Test Object Class Details

Specify the details for the new test object class hrtsu.

Test object icon:

Identification property for unique description: label

Default test object method:

Default checkpoint properties:

- [ ] attached_text
- [ ] background
- [ ] class_index
- [ ] class_name
- [ ] class_path
- [ ] description_flags
- [ ] displayable
- [ ] displayed
- [x] enabled
- [ ] focusable
- [x] focused

The New Test Object Class Details screen contains the following options:

Option	Description
Test object icon	The path of the icon file to use in the Keyword view for this test object class. The icon file must be in an uncompressed **.ico** format. This is optional. If you do not define an icon file, the JavaObject icon is used.
Identification property for unique description	Specifies the identification property that QuickTest uses to uniquely identify the control (in addition to the toolkit class and index properties). You can select an identification property from the list or leave the property the wizard selected by default.
Default test object method	Specifies the default test object method displayed in the Keyword View and Step Generator when a step is generated for an object of this class. Select a test object method from the list.
Default checkpoint properties	Specifies the identification properties that are selected by default when you create a checkpoint for an object of this class. Select the check boxes for the appropriate properties. Click **Select All** or **Clear All** to select or clear all of the check boxes.

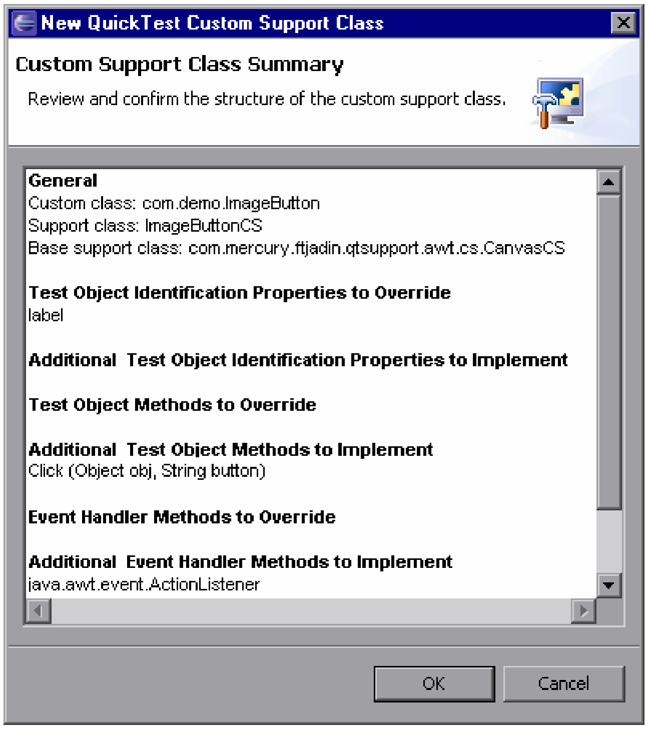
When the wizard creates the new support class, it adds the new test object type to the test object configuration file. The options you specify in the New Test Object Class Details screen are recorded in this file. For information about the structure of this file, refer to the *QuickTest Test Object Schema Help* (**Help** > **QuickTest Professional Help** > **QuickTest Advanced References** > **QuickTest Test Object Schema**).

Note: If you want QuickTest to include additional identification properties in the test object description, you must manually specify this in the test object configuration file. The wizard adds the test object class definition to the test object configuration file. For each property that you want to add to the test object description, find the line that describes it in the file. Between the words Property and Name, add the words ForDescription="true".

Click **Finish**. The Custom Control Support Class Summary Screen opens, as described below.

Custom Control Support Class Summary Screen

Before the wizard creates the custom support class file, the Custom Support Class Summary screen summarizes the specifications you provided for the new support class.


New QuickTest Custom Support Class
Custom Support Class Summary
Review and confirm the structure of the custom support class.
OK
Cancel

General
Custom class: com.demo.ImageButton
Support class: ImageButtonCS
Base support class: com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS

Test Object Identification Properties to Override
label

Additional Test Object Identification Properties to Implement

Test Object Methods to Override

Additional Test Object Methods to Implement
Click (Object obj, String button)

Event Handler Methods to Override

Additional Event Handler Methods to Implement
java.awt.event.ActionListener

If you want to change any of the data, click **Cancel** to return to the previous wizard screen. Use the **Back** and **Next** buttons to open the relevant screens and make the required changes.

If you are satisfied with the definitions, click **OK**. The wizard creates the new support class with all of the required methods, according to your specifications.

In addition, the wizard adds the test object class definition to the test object configuration file if one of the following conditions is met:

- You mapped a new test object class to the custom control.
- You added identification properties or test object methods to an existing test object class.

Note: If the test object configuration file does not exist, the wizard creates it at this time. For information on the structure 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**).

Completing the Custom Class Support

After you finish creating a custom support class (using the New QuickTest Custom Support Class Wizard), you need to perform the following additional steps:

- Save the class. The changes made by the wizard are codependent and need to be saved to prevent discrepancies.

Note: In Eclipse, the new class file is opened and displayed in a tab in the right pane. Until you save the class, an asterisk (*) is displayed in the tab next to the support class file name.

➤ Implement any method stubs that the wizard created in the new custom support class. For more information, see “Understanding the Toolkit Support Class” on page 35.

➤ Deploy the toolkit support to QuickTest to enable the support to be available. For more information, see “Deploying and Running the Custom Toolkit Support” on page 66.

New QuickTest Custom Static-Text Support Class Wizard

You use the New QuickTest Custom Static-Text Support Class wizard to create a support class for a custom static-text class within a Java Add-in Extensibility project. Supporting a static-text class enables QuickTest to use its **label** property as the **attached text** for an adjacent control.

The only thing that you need to specify in this wizard is which custom class you want to support as a static-text class (and the **controls of this class represent top-level objects**, if relevant). The wizard creates the new support class with the methods required for the support of static-text objects. These methods are described in “Custom Static-Text Support Class Summary Screen” on page 135.

After the wizard creates the new support class, you complete its implementation as described in “Completing the Custom Static-Text Class Support” on page 136.

In most cases, it is not necessary to support any additional identification properties or test object methods for a static-text control. However, after the wizard creates the new support class, you can add additional methods to the class, providing support for additional identification properties or test object methods, or for recording. You can add these methods manually, or by using the commands described in “Working with QuickTest Commands in Eclipse” on page 137.

To open the New QuickTest Custom Static-Text Support Class wizard in Eclipse:

1 In the Eclipse Package Explorer tab, select a QuickTest Java Add-in Extensibility project. Then choose **File** > **New** > **Other**. The New dialog box opens.

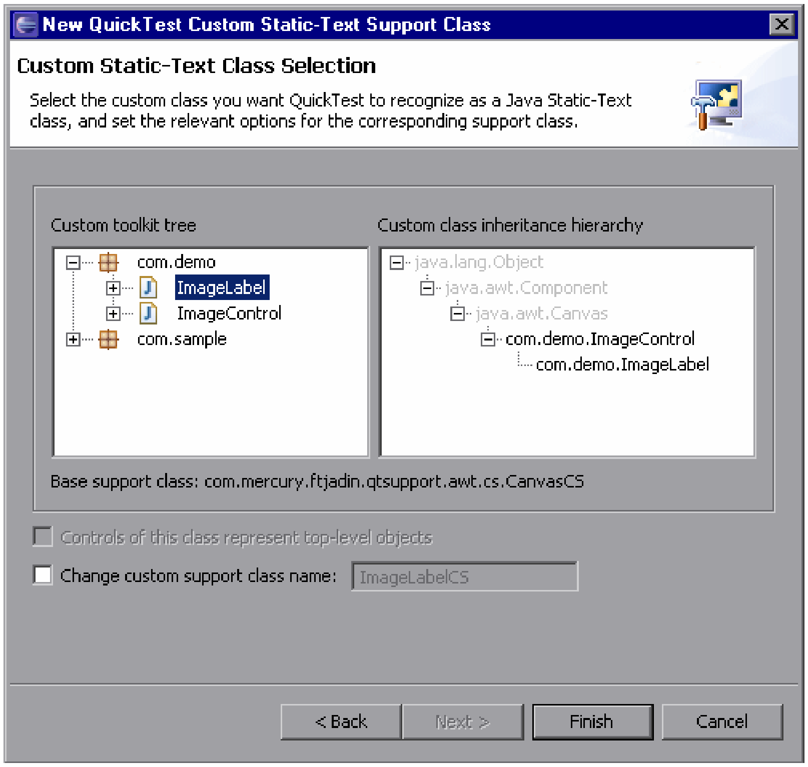
2 Expand the **QuickTest Professional** folder and select **QuickTest Custom Static-Text Support Class**.

3 Click **Next**. The Custom Static-Text Class Selection Screen opens.

Tip: You can shorten this process by customizing Eclipse to provide **QuickTest Custom Static-Text Support Class** as an option in the **New** menu. To do this, perform the following: Choose **Window** > **Customize Perspective**. In the Shortcuts tab in the dialog box that opens, select the **QuickTest Professional** and **QuickTest Custom Static-Text Support Class** check boxes. Click **OK**.

Custom Static-Text Class Selection Screen

The options in the Custom Static-Text Class Selection screen are identical to the options in the Custom Class Selection Screen (described on page 99).


New QuickTest Custom Static-Text Support Class

Custom Static-Text Class Selection

Select the custom class you want QuickTest to recognize as a Java Static-Text class, and set the relevant options for the corresponding support class.

Custom toolkit tree

```
com.demo
    ImageLabel
    ImageControl
com.sample
```

Custom class inheritance hierarchy

```
java.lang.Object
  java.awt.Component
    java.awt.Canvas
      com.demo.ImageControl
        com.demo.ImageLabel
```

Base support class: com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS

☐ Controls of this class represent top-level objects

☐ Change custom support class name: ImageLabelCS

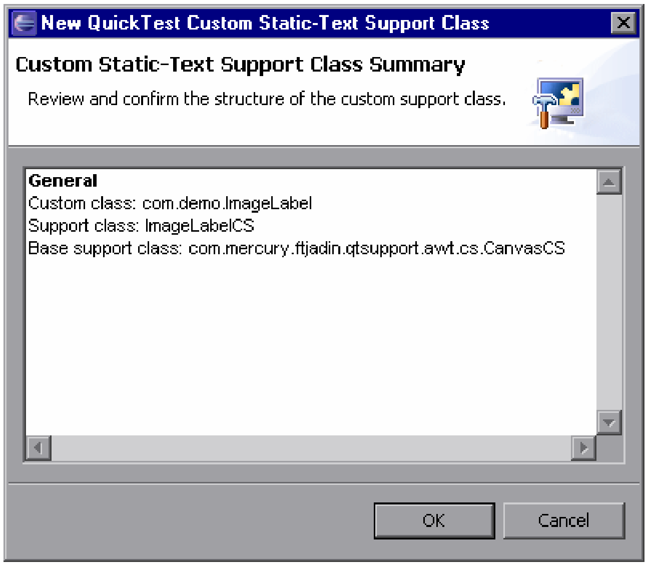
You select the custom class you want QuickTest to recognize as static-text and set the relevant options.

Static-text controls do not normally have any identification properties or test object methods that are relevant for QuickTest tests. Therefore, no additional specifications are required in this wizard.

Click **Finish**. The Custom Static-Text Support Class Summary Screen opens (described below).

Custom Static-Text Support Class Summary Screen

Before the wizard creates the custom support class file, the Custom Static-Text Support Class Summary screen summarizes the specifications you provided for the new support class.



Custom Static-Text Support Class Summary

Review and confirm the structure of the custom support class.

General
Custom class: com.demo.ImageLabel
Support class: ImageLabelCS
Base support class: com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS

If you want to change any of the data, click **Cancel** to return to the Custom Static-Text Class Selection Screen, described above.

If you are satisfied with the definitions, click **OK**. The wizard creates the new support class with the following methods, which are required for the support of static-text objects:

- **class_attr.** Returns the string static_text, enabling QuickTest to recognize objects of this class as static-text controls.
- **label_attr.** Returns the **label** property of the superclass.

When the **label** property for a Java control is empty, QuickTest looks for an adjacent static-text control and uses its **label** property for the test object name. Therefore you may want to implement the **label_attr** method to return the appropriate name, for example, the string displayed by the static-text control.

- **tag_attr.** Returns the **tag** property of the superclass (which returns the **label** property value) with the suffix (st). This method provides the test object name for the static-text control itself, while the **label_attr** method provides the name used for adjacent controls.

  For example, if you implement the **label_attr** method to return MyButton, the **tag_attr** method returns MyButton(st).

  For more information, see "Special Identification Property Support Methods" on page 52.

- **value_attr.** Returns the **label** property.

  The **value** property represents a control's test object state. For static-text controls, the **label** property adequately represents this state.

You can practice creating support for a custom static-text control in the tutorial lesson "Learning to Support a Custom Static-Text Control" on page 197.

Completing the Custom Static-Text Class Support

After you finish creating a custom support class for a custom static-text class (using the New QuickTest Custom Static-Text Support Class Wizard), you need to perform the following additional steps:

- Save the class. The changes made by the wizard are codependent and need to be saved to prevent discrepancies.

Note: In Eclipse, the new class file is opened and displayed in a tab in the right pane. Until you save the class, an asterisk (*****) is displayed in the tab next to the support class file name.

- Implement the **label_attr** method, if needed.
- Deploy the toolkit support to QuickTest to enable the support to be available. For more information, see "Deploying and Running the Custom Toolkit Support" on page 66.

Working with QuickTest Commands in Eclipse

After you install the QuickTest Java Add-in Extensibility SDK, which includes the Java Add-in Extensibility Eclipse Plug-in, a toolbar with the following buttons is added to Eclipse:

Button	Definition	Button	Definition
	Deploy Toolkit Support		Add Identification Property
	Reload Support Configuration		Add Test Object Method
	Delete Custom Support		Add Event Handler

A new **QuickTest** menu is also added to Eclipse, with these same commands. The commands are described in detail in the following sections.

Deploy Toolkit Support

The **Deploy Toolkit Support** command is available when you select a QuickTest Java Add-in Extensibility project (or elements within it) in the Eclipse Package Explorer tab.

Note: The **Deploy Toolkit Support** command is not available if you installed the QuickTest Professional Java Add-in Extensibility SDK before installing QuickTest and the Java Add-in. To solve this problem, uninstall the QuickTest Professional Java Add-in Extensibility SDK and install it again. For more information, see "Installing the QuickTest Professional Java Add-in Extensibility Software Development Kit" on page 13.

You use the **Deploy Toolkit Support** command to deploy the toolkit support during the development stages. To use this command, QuickTest and the QuickTest Java Add-in Extensibility Eclipse Plug-in must be installed on the same computer.

This command copies the toolkit configuration file and the test object configuration file to the appropriate QuickTest folders. In the toolkit configuration file, the location specified for the support classes is the Eclipse workspace. The next time you run the Java application, the support you developed is available on QuickTest. For more information, see “Deploying and Running the Custom Toolkit Support” on page 66.

Notes:

The **Deploy Toolkit Support** command copies only the test object configuration file that is named **<Custom Toolkit Name>TestObjects.xml**. If you create additional test object configuration files you must copy them to the appropriate folders, specified in “Deploying and Running the Custom Toolkit Support” on page 66.

The **Deploy Toolkit Support** command validates the configuration files against their corresponding XSD files, and only deploys them if their format meets the requirements. For information about the structure of the configuration files, refer to the *QuickTest Java Add-in Extensibility Toolkit Configuration Schema Help* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer’s Guide** > **QuickTest Java Add-in Extensibility Toolkit Configuration Schema**) and the *QuickTest Test Object Schema Help* (**Help** > **QuickTest Professional Help** > **QuickTest Advanced References** > **QuickTest Test Object Schema**).

Reload Support Configuration

The **Reload Support Configuration** command is available when you select a QuickTest Java Add-in Extensibility project (or elements within it) in the Eclipse Package Explorer tab.

Note: The **Reload Support Configuration** command is not available if you installed the QuickTest Professional Java Add-in Extensibility SDK before installing QuickTest and the Java Add-in. To solve this problem, uninstall the QuickTest Professional Java Add-in Extensibility SDK and install it again. For more information, see "Installing the QuickTest Professional Java Add-in Extensibility Software Development Kit" on page 13.

The **Reload Support Configuration** command instructs the QuickTest Java Add-in Extensibility Eclipse plug-in to update the plug-in's list of supported Java classes and test object classes by reloading:

- ➤ the selected project's configuration files and support classes
- ➤ all of the toolkit configuration files and test object configuration files from the QuickTest installation folder

The **Reload Support Configuration** command affects the following items in the New QuickTest Custom Support Class wizard:

- ➤ The list of custom classes displayed in the custom toolkit tree in the Custom Class Selection Screen (described on page 99).
- ➤ The wizard's selection of the base support class in the Custom Class Selection Screen (described on page 99).
- ➤ The list of existing test object classes displayed in the Test Object Class Selection Screen (described on page 104).

The following examples demonstrate situations that require reloading the support configuration:

- ➤ You modified the test object configuration file in the QuickTest Java Add-in Extensibility project, adding or removing test object classes. You now want the wizard's list of existing test object methods to reflect these changes.
- ➤ You manually deployed support of a custom toolkit to QuickTest, and you want the wizard to recognize that the classes are supported.

- You manually deleted support for some classes from QuickTest, and you want these classes to be removed from the list of supported classes in the Eclipse plug-in.
- You created a custom toolkit support set (named Support Set A) in one Eclipse project and deployed it. You are now creating a custom toolkit support set (named Support Set B) for another custom toolkit in a different Eclipse project. You want to use a support class from Support Set A as the base support class for a support class in Support Set B.

Delete Custom Support

The **Delete Custom Support** command is available when you select a QuickTest Java Add-in Extensibility custom support class in the Eclipse Package Explorer tab.

Note: The command is available only if this class was created as a QuickTest Java Add-in Extensibility custom support class in this Eclipse workspace.

You use this command to delete support for a specific custom class. The support class is deleted, as well as its listing in the toolkit configuration file. If you created a new test object class for this support class, it is not deleted from the test object configuration file because other support classes can use it.

If you delete the support class using the Eclipse Delete command, instead of the **Delete Custom Support**, you must manually remove the mapping of the custom control to this support class in the toolkit configuration file.

Tip: If you want to delete a support class you have just created, make sure you save the support class first.

After deleting a support class, if you previously deployed support for this custom class to QuickTest, you must re-deploy the toolkit support. This replaces the toolkit configuration file with the updated one, removing the support for this custom class from QuickTest as well.

If you delete a support class that serves as the base support class for another, you must manually change the inheritance of this other class. For example: InheritedCS extends ToDeleteCS that extends BuiltInCS. If you delete ToDeleteCS, you must manually change InheritedCS to extend BuiltInCS.

Note: You cannot remove support of a complete toolkit using the QuickTest Java Add-in Extensibility Eclipse Plug-in commands. To do this you have to manually delete the toolkit configuration files from their locations in the QuickTest folders. For more information, see "Deploying and Running the Custom Toolkit Support" on page 66.

Add Identification Property

The **Add Identification Property** command is available when you select a QuickTest Java Add-in Extensibility custom support class in the Eclipse Package Explorer tab.

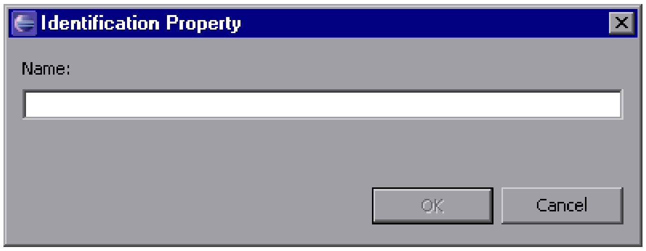
Note: This command is available only if this class was created as a QuickTest Java Add-in Extensibility custom support class in this Eclipse workspace.

You use this command to add an identification property after the support class is created.

You must deploy the toolkit support for the changes to take effect on QuickTest.

To add an identification property:

1 Click the **Add Identification Property** button. The Identification Property dialog box opens.



2 Enter a name for the new identification property and click **OK**.

3 A confirmation box opens notifying you that in addition to adding the new identification property to the support class, the property will also be added to the definition of the test object class mapped to the supported control. This identification property will then appear in the list of identification properties in QuickTest for all test objects of this class.

Click **Yes** if you want to continue. (If you click **No**, the new identification property is discarded.)

A support method stub for the identification property you defined, named **<identification property name>_attr**, is added to the support class. The method stub returns **null** until you implement the method to match the needs of your custom control.

4 Another message box prompts you to select whether you want the new identification property to be selected by default in checkpoints.

After you make your selection, the new identification property is added to the test object class description in the test object configuration file.

If you clicked **Yes**, the **ForDefaultVerification** attribute is added to the identification property definition and set to **true**. Otherwise, the **ForDefaultVerification** is not added. (In both cases, the **ForVerification** attribute is added and set to **true**, so that the new identification property is always available for checkpoints.)

Note: If you add an identification property to the test object class definition, it appears in the list of identification properties in QuickTest for all test objects of this class.

It is for this reason that, if you plan to add properties, you create a new test object class based on the existing one, instead of using the existing test object class.

Tip: If you add an identification property that you want to be part of the unique test object description, you have to manually define this in the test object configuration file. In the row for this identification property, between the words Property and Name add the words ForDescription="true". This adds the **ForDescription** attribute to the **Property** element and sets it to **true**.

For more information, refer to the *QuickTest Test Object Schema Help* (**Help** > **QuickTest Professional Help** > **QuickTest Advanced References** > **QuickTest Test Object Schema**).

Add Test Object Method

The **Add Test Object Method** command is available when you select a QuickTest Java Add-in Extensibility custom support class in the Eclipse Package Explorer tab.

Note: This command is available only if this class was created as a QuickTest Java Add-in Extensibility custom support class in this Eclipse workspace.

You use this command to add a test object method after the support class is created.

You must deploy the toolkit support for the changes to take effect on QuickTest.

To add a test object method:

1. Click the **Add Test Object Method** button. The Test Object Method Dialog box opens.
2. Define the details of the test object method you want to add, and click **OK**. For more information, see "Understanding the Test Object Method Dialog Box" on page 116.
3. A confirmation box opens notifying you that in addition to adding new test object method to the support class, the test object method will also be added to the definition of the test object class mapped to the supported control. The test object method will then appear in QuickTest for all test objects of this class.

   Click **Yes** if you want to continue. (If you click **No**, the new test object method is discarded.)

   A support method stub for the test object method you defined, named **<test object method name>_replayMethod**, is added to the support class. This method stub returns the error value **Retval.NOT_IMPLEMENTED** until you implement it to match the needs of your custom control.

   In addition, the test object method is added to the test object class definition in the test object configuration file. For information on the structure and content of this file, refer to the *QuickTest Test Object Schema Help* (**Help** > **QuickTest Professional Help** > **QuickTest Advanced References** > **QuickTest Test Object Schema**).

Note: If you add a test object method to an existing test object class, the new methods appear in QuickTest for all test objects of this class, regardless of whether or not they are supported for these objects. In a QuickTest test, if you call a test object method for an object, and that method is not supported, a run-time error occurs.

Therefore, if you plan to add test object methods to support a custom control, it is recommended to create a new test object class based on the existing one, instead of using the existing test object class.

Add Event Handler

The **Add Event Handler** command is available when you select an AWT-based QuickTest Java Add-in Extensibility custom support class in the Eclipse Package Explorer tab.

Note: This command is available only if this class was created as a QuickTest Java Add-in Extensibility custom support class in this Eclipse workspace.

You use this command to add an event handler to the support class after it is created.

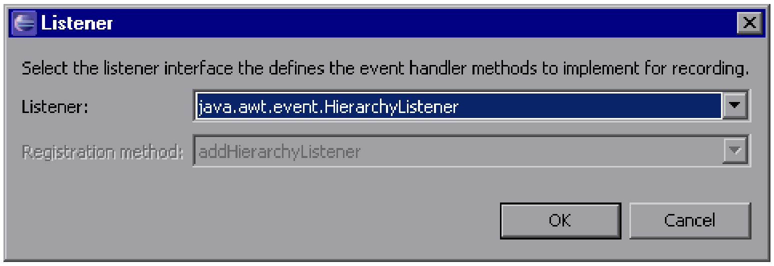
The following options are available in the Custom Control Recording Support wizard screen when you create a new support class:

- **Treat controls of this class as wrapper controls**
- **Override low-level mouse event recording**
- **Override low-level keyboard event recording**

If you did not select them when you created the support class, and you want to implement them, you have to do so manually. For information on how to do this, see "Supporting the Record Option" on page 56.

To add event handler methods:

1 Click the **Add Event Handler** button. The Listener dialog box opens.


Listener

Select the listener interface the defines the event handler methods to implement for recording.

Listener: java.awt.event.HierarchyListener

Registration method: addHierarchyListener

2 Select a listener from the list.

If the selected listener has more than one registration method, select a method from the **Registration method** list.

➤ Click **OK**.

The listener you selected is added to the signature of the support class.

A constructor is added to the support class (or modified, if it already exists) to call the **addSimpleListener** method for the listener you selected. This adds the listener to the list of listeners that need to be registered on the custom control, and specifies the methods used to register and remove it.

Method stubs for all of the event handler methods that comprise the listener you selected are added to the support class. A comment is added to each method stub, reminding you to implement the event handler to call **MicAPI.record** to send a record message to QuickTest. For more information see “Supporting the Record Option” on page 56.

Part II

Tutorial: Learning to Create Java Custom Toolkit Support

6

Using the QuickTest Java Add-in Extensibility Tutorial

The QuickTest Java Add-in Extensibility tutorial comprises lessons that will familiarize you with the process of creating, testing, and deploying custom toolkit support. After completing the tutorial, you can apply the skills you learn to create QuickTest support for your own custom toolkits and controls.

Understanding the Tutorial Lesson Structure

This tutorial is divided into three lessons. Each lesson assumes that you have already performed the previous lessons or have an equivalent level of experience. In each lesson, you learn more about the capabilities and techniques available with QuickTest Java Add-in Extensibility. It is recommended to perform the lessons in order.

In each lesson in this tutorial, you extend QuickTest support for a different custom control, using the QuickTest Java Add-in Extensibility Eclipse plug-in. The custom controls are provided in sample custom toolkits that you can find in the **<Java Add-in Extensibility SDK installation folder>\samples** folder. This folder also contains the custom toolkit support sets required to support these custom controls and additional examples of custom toolkits and their support.

Each lesson in the tutorial explains the Java Add-in Extensibility wizard options that you need to use in that specific lesson. For more information on these wizards, see “Using the QuickTest Java Add-in Extensibility Eclipse Plug-In” on page 83.

Checking Tutorial Prerequisites

To perform the lessons in this tutorial, you must meet the requirements described in this section.

System Requirements

The following minimum system requirements are required to perform the lessons in this tutorial:

Eclipse, version 3.1.2

Eclipse, version 3.1.2 requires Java 1.4.2 or later.

Java Add-in Extensibility SDK

For information on installing Eclipse or the Java Add-in Extensibility SDK see “Installing the QuickTest Professional Java Add-in Extensibility Software Development Kit” on page 13.

QuickTest Professional 9.1 or later, including the Java Add-in

For information on installing QuickTest Professional, refer to the *QuickTest Professional Installation Guide*.

If your QuickTest Professional installation is not on the same computer as Eclipse, you can still perform the lessons in this tutorial. However, when you are instructed to deploy the toolkit support to QuickTest, you must manually transfer the custom support class files and configuration files to the correct folders on the QuickTest computer as described in "Deploying and Running the Custom Toolkit Support" on page 66.

A computer on which support has not yet been implemented for the custom toolkits and controls in this tutorial

If support has already been for the custom toolkits and controls in this tutorial, remove the support as described in "Deploying and Running the Custom Toolkit Support" on page 66 or use another computer.

Knowledge Requirements

The lessons in this tutorial assume you have the background knowledge described below:

Familiarity with major QuickTest features and functionality

You should have a thorough understanding of the following: test objects, object repository, Object Spy, Keyword View, and Expert View. You should also have experience recording, editing, and running tests. For more information, refer to the *QuickTest Professional User's Guide*.

Experience with Java programming

You should be familiar with the concepts related to Java programming (class, package, interface, inheritance, and so on) and know how to write and compile Java classes.

Familiarity with XML

You should be familiar with the concepts of elements and attributes and feel comfortable working with schemas and editing XML files.

A basic understanding of the concepts described in the Implementing Custom Toolkit Support chapter

This tutorial assumes familiarity with the concepts described in "Implementing Custom Toolkit Support" (beginning on page 27).

Learning to Support a Simple Control

The lesson, Learning to Support a Simple Control (on page 155), uses a basic custom Java control named ImageButton to teach you the fundamental elements of custom support. This lesson guides you through the steps required to create a custom toolkit support project containing one custom support class. Through this lesson, you will learn to recognize and understand the files and methods that comprise custom support.

In this lesson, you use two of the wizards provided by the QuickTest Java Add-in Extensibility Eclipse plug-in: the New QuickTest Java Add-in Extensibility Project wizard and the New QuickTest Custom Support Class wizard.

Learning to Support a Static-Text Control

The lesson, Learning to Support a Custom Static-Text Control (on page 197), uses the ImageLabel control to teach you how to support a static-text control. This lesson guides you through the steps required to create a support class for a static-text control in an existing custom toolkit support project. (The ImageLabel control belongs to the same custom toolkit as the ImageButton control that you used in the previous lesson.) Through this lesson, you will learn about the basic methods that are required in a support class for a static-text control.

In this lesson, you use the New QuickTest Custom Static-Text Support Class wizard provided by the QuickTest Java Add-in Extensibility Eclipse plug-in.

Learning to Support a Complex Control

The lesson, Learning to Support a Complex Control (on page 225), uses the custom Java control AllLights to teach you more about custom support. AllLights is a complex control with unique behavior that requires a new test object class definition. The lesson guides you through the steps of creating a custom support class containing new identification properties and test object methods that did not exist in the parent support class. You will also learn to understand the test object configuration file.

7

Learning to Support a Simple Control

In this lesson you create support for the ImageButton control within the ImageControls toolkit. Adding support for the ImageButton control requires only minimal customization, allowing you to learn the basics of creating a custom toolkit support set.

Before you perform this lesson, ensure that you have read the "Implementing Custom Toolkit Support" and "Planning Custom Toolkit Support" chapters in this guide and that you meet the tutorial prerequisites as described in Chapter 6, "Using the QuickTest Java Add-in Extensibility Tutorial."

Preparing for This Lesson

Before you extend QuickTest support for a custom control, you must:

- Make sure you have full access to the control.
- Understand its behavior and what functionality needs to be tested.
- Have an application in which you can see and operate the control.
- Have access to the class that implements it. (Although you do not modify any of the custom control classes when creating your custom support, you reference the compiled classes, and rely on information you can gain from the source files.)

Perform the following steps to create an Eclipse project containing the ImageControls custom toolkit classes and a sample application containing the custom controls.

Note: The sample application is designed to run from the default **<QuickTest Professional Java Add-in Extensibility SDK installation>\samples** folder. If you install the SDK to another location, you need to modify the sample application slightly before you begin this lesson. For information, see "Modifying the Sample Application to Run From Another Location" on page 158.

To create a new Java project with the ImageControls sample in Eclipse:

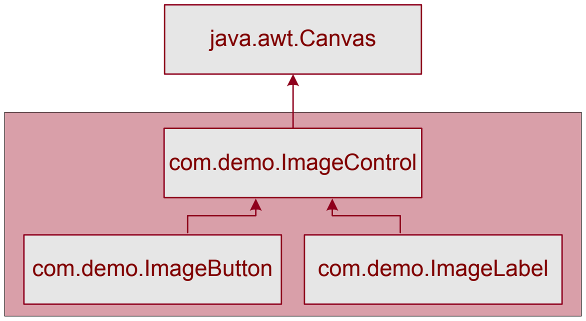
1. Run Eclipse and choose **File** > **New** > **Project**. The New Project dialog box opens.
2. Select **Java Project** and click **Next**. The New Java Project dialog box opens.
3. Enter ImageControls in the **Project name** box.
4. Select the **Create project from existing source** option.
5. Click the **Browse** button and browse to the **<QuickTest Professional Java Add-in Extensibility SDK installation folder>\samples\ImageControls\src** folder. Click **OK** to return to the New Java Project dialog box.

6 Click **Finish**. A new Java project is created with the ImageControls sample source files. The new project, named **ImageControls**, is displayed in the Package Explorer tab.

Expand the **ImageControls** project to view its content. The **ImageControls\src** package folder contains two packages:

- The **com.sample** package contains the sample application: **SampleApp**.
- The **com.demo** package contains three custom controls: **ImageButton**, **ImageControl** and **ImageLabel**.

The following diagram shows the inheritance hierarchy of the classes in the **com.demo** package.



The functionality provided by the classes in this package is as follows:

- **ImageControl.** This class extends the **Canvas** class, and displays an image on the control.
- **ImageLabel.** This class extends the **ImageControl** class, and allows writing additional text over the image displayed on the control.
- **ImageButton.** This class extends the **ImageControl** class, and draws a button-like rectangle around the control. It listens for low-level events on the control, and triggers an **Action** event when the button is clicked.

Modifying the Sample Application to Run From Another Location

If you installed the QuickTest Professional Java Add-in Extensibility SDK to a folder other than the default installation folder, you must modify the sample application before performing this lesson.

After you copy the ImageControls source files into Eclipse, browse to the package **ImageControls\src\com.sample** in Eclipse and open the **SampleApp.java** file.

Locate the code containing the image file paths:

```
C:/Program Files/Mercury Interactive/QuickTest Professional Java Add-in Extensibility SDK/samples/ImageControls/images/mercury.gif
```

```
C:/Program Files/Mercury Interactive/QuickTest Professional Java Add-in Extensibility SDK/samples/ImageControls/images/JavaExt1.gif
```

Replace `C:/Program Files/Mercury Interactive/QuickTest Professional Java Add-in Extensibility SDK` in these paths with the actual installation folder to enable the sample application to function properly.

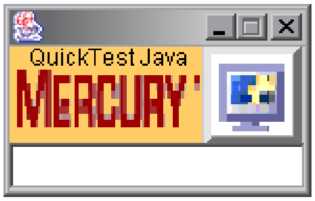
Planning Support for the ImageButton Control

In this section, you analyze the current QuickTest support of the ImageButton control, determine the answers to the questions in the "Understanding the Custom Class Support Planning Checklist" on page 78, and fill in the "Custom Class Support Planning Checklist" on page 163, accordingly.

The best way to do this is to run the application containing the custom control, and analyze it from a QuickTest perspective using the Object Spy, Keyword View, and Record option.

1 Run the SampleApp application and open QuickTest.

In the Eclipse Package Explorer tab, right-click **SampleApp**. Choose **Run As** > **Java Application**. The SampleApp application opens.



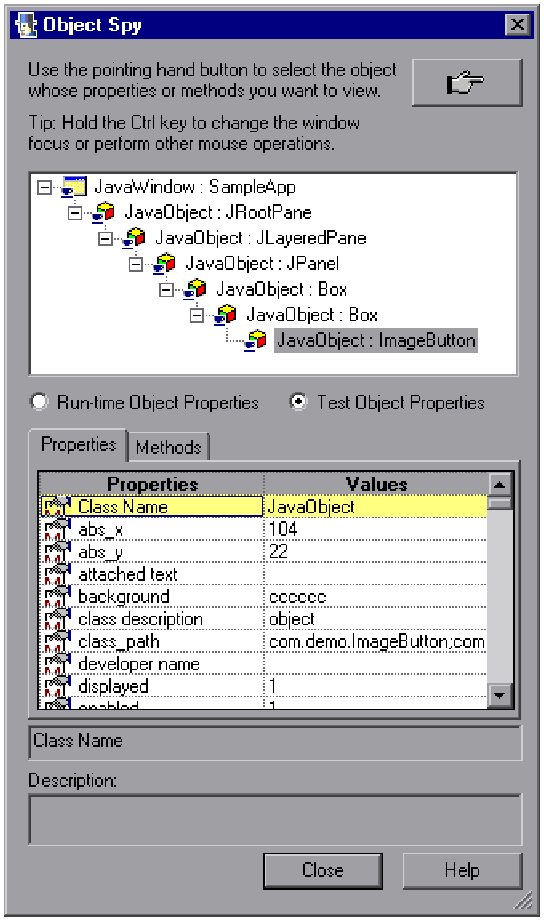
Open QuickTest and load the Java Add-in.

2 Use the Object Spy to view the ImageButton properties.

In QuickTest Choose **Tools** > **Object Spy** or click the **Object Spy** toolbar button to open the Object Spy dialog box. Click the **Properties** tab.

In the Object Spy dialog box, click the pointing hand, then click the button in the SampleApp application.

The ImageButton control is based on a custom class that QuickTest does not recognize. Therefore, it recognizes the button as a generic **JavaObject** named **ImageButton**, and the icon shown is the standard JavaObject class icon.


Object Spy
Use the pointing hand button to select the object whose properties or methods you want to view.
Tip: Hold the Ctrl key to change the window focus or perform other mouse operations.
JavaWindow : SampleApp
JavaObject : JRootPane
JavaObject : JLayeredPane
JavaObject : JPanel
JavaObject : Box
JavaObject : Box
JavaObject : ImageButton
Run-time Object Properties
Test Object Properties
Properties
Methods
Class Name
Description:
Close
Help

Properties	Values
Class Name	JavaObject
abs_x	104
abs_y	22
attached text	
background	cccccc
class description	object
class_path	com.demo.ImageButton;com
developer name	
displayed	1
enabled	1

Close the Object Spy.

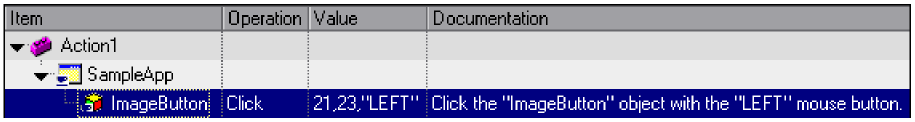
3 Record an operation on the ImageButton control.

In QuickTest choose **Automation** > **Record and Run Settings** to open the Record and Run Settings dialog box. In the Java tab, select **Record and run test on any open Java application**. If the Web Add-in is also loaded, click the Web tab and select **Record and run test on any open browser**. Click **OK**.

Record

Click the **Record** button or choose **Automation** > **Record**. Click the button in the SampleApp application. The counter value in the edit box increases by one.

A new step is added to the test.



Item	Operation	Value	Documentation
Action1			
SampleApp			
ImageButton	Click	21,23,"LEFT"	Click the "ImageButton" object with the "LEFT" mouse button.

Stop

Click the **Stop** button or choose **Automation** > **Stop** to end the recording session.

The **Click** operation on the ImageButton JavaObject is a generic click, with arguments indicating the low-level recording details (x and y coordinates and the mouse button that performed the click).

4 Determine the custom toolkit to which the ImageButton control belongs.

When you extend QuickTest support for a control you always do so in the context of a toolkit. For the purpose of this tutorial, three classes that share the same ancestor, java.awt.Canvas, are grouped to form the custom toolkit named ImageControls: ImageButton, ImageLabel, and their superclass ImageControl.

In this lesson you create support for the ImageControls toolkit, initially supporting only the ImageButton class.

5 Complete the custom class support planning checklist.

You want QuickTest to treat the ImageButton as a special kind of button and you want it to support the operation it performs. Therefore it makes sense to create Extensibility support for this control.

The custom class ImageButton extends another custom class, ImageControl, for which QuickTest also does not provide support. At this point, there does not seem to be any functionality requiring special QuickTest support, which ImageButton shares with other classes that extend ImageControl. Therefore it is sufficient to extend support directly to the ImageButton class.

When fully supported, QuickTest should recognize the ImageButton control as a JavaButton test object. You want JavaButton test objects representing controls of this type to be named according to the name of the image file that the control displays.

The custom support should also include support for the simple Click-on-the-button operation. (Note that in QuickTest, the simple JavaButton **Click** operation has an optional argument that specifies which mouse button performed the click.) The ImageButton custom class listens for low-level mouse events and substitutes them with events that are more relevant to button behavior, in this case an **Action** event. Therefore, to record mouse clicks, the support class must listen for **Action** events.

On the next page you can see the checklist, completed based on the information above.

Custom Class Support Planning Checklist

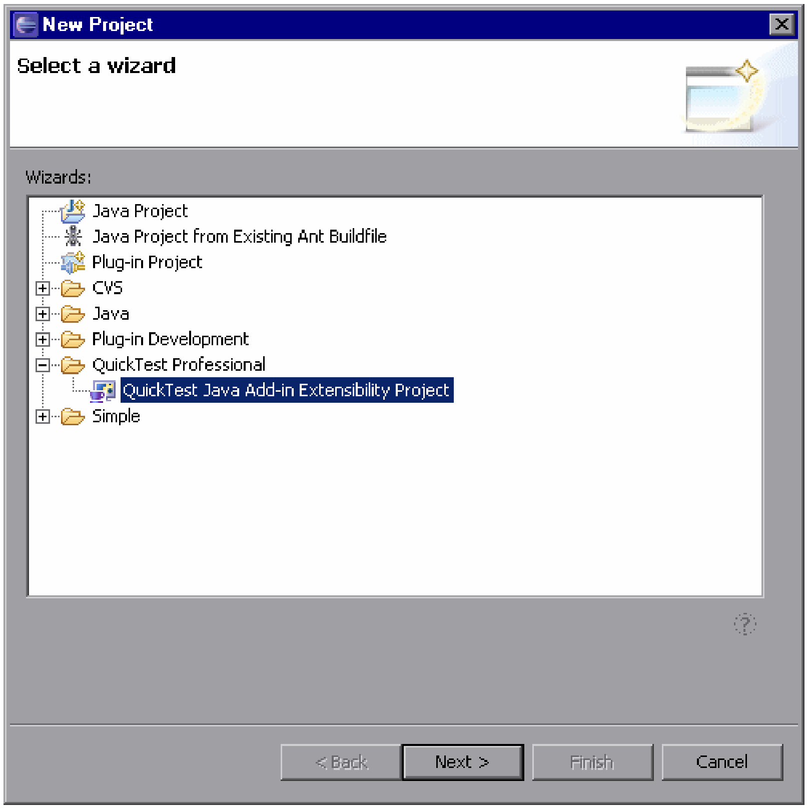
☑	**Custom Class Support Planning Checklist**
☐	Does the custom class have a superclass for which QuickTest custom support is not yet available? **Yes** /~~**No**~~
☐	If so, should I first extend support for a control higher in the hierarchy? ~~**Yes**~~ /**No**
☐	Do I have an application that runs the custom control on a computer with QuickTest installed? **Yes** /~~**No**~~
☐	The sources for this custom control class are located in: an Eclipse project called ImageControls
☐	Which existing Java test object matches the custom control? JavaButton
☐	If none, create a new Java test object class named: N/A • New test object class extends: (Default—JavaObject) • Icon file location (optional): • Identification property for description: • Default test object method:
☐	Is the custom control a top-level object? ~~**Yes**~~ /**No**
☐	Is the custom control a wrapper? ~~**Yes**~~ /**No**
☐	Specify the basis for naming the test object: its image file name
☐	List the identification properties to support, and mark default checkpoint properties: nothing special
☐	List the test object methods to support (include arguments and return values if required): Click (button)
☐	Provide support for recording? (AWT-based only) **Yes** /~~**No**~~
☐	If so, list the events that should trigger recording: ActionEvents

Creating a New QuickTest Java Add-in Extensibility Project

In this section you create a new project for the ImageControls toolkit support. To do this, you use one of the wizards provided by the QuickTest Java Add-in Extensibility plug-in in Eclipse.

1 Open the New QuickTest Java Add-in Extensibility Project wizard.

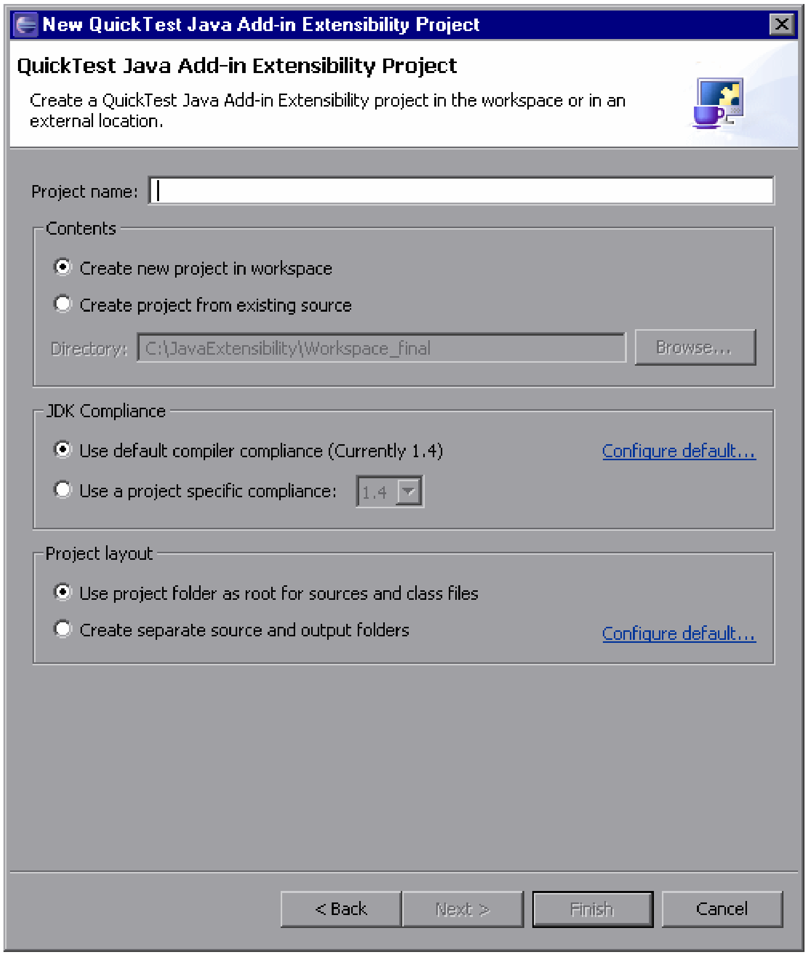
In Eclipse, choose **File** > **New** > **Project**. The New Project dialog box opens. Expand the **QuickTest Professional** folder and select **QuickTest Java Add-in Extensibility Project**.



Click **Next**. The QuickTest Java Add-in Extensibility Project screen opens.

2 Enter the QuickTest Java Add-in Extensibility project details.

In the **Project name** box, enter ImageControlsSupport. Select the **Create separate source and output folders** option. For more information on this dialog box, refer to the *Eclipse Help*.



New QuickTest Java Add-in Extensibility Project

QuickTest Java Add-in Extensibility Project

Create a QuickTest Java Add-in Extensibility project in the workspace or in an external location.

Project name:

Contents

- (●) Create new project in workspace
- ( ) Create project from existing source

Directory: C:\JavaExtensibility\Workspace_final

JDK Compliance

- (●) Use default compiler compliance (Currently 1.4)
- ( ) Use a project specific compliance: 1.4

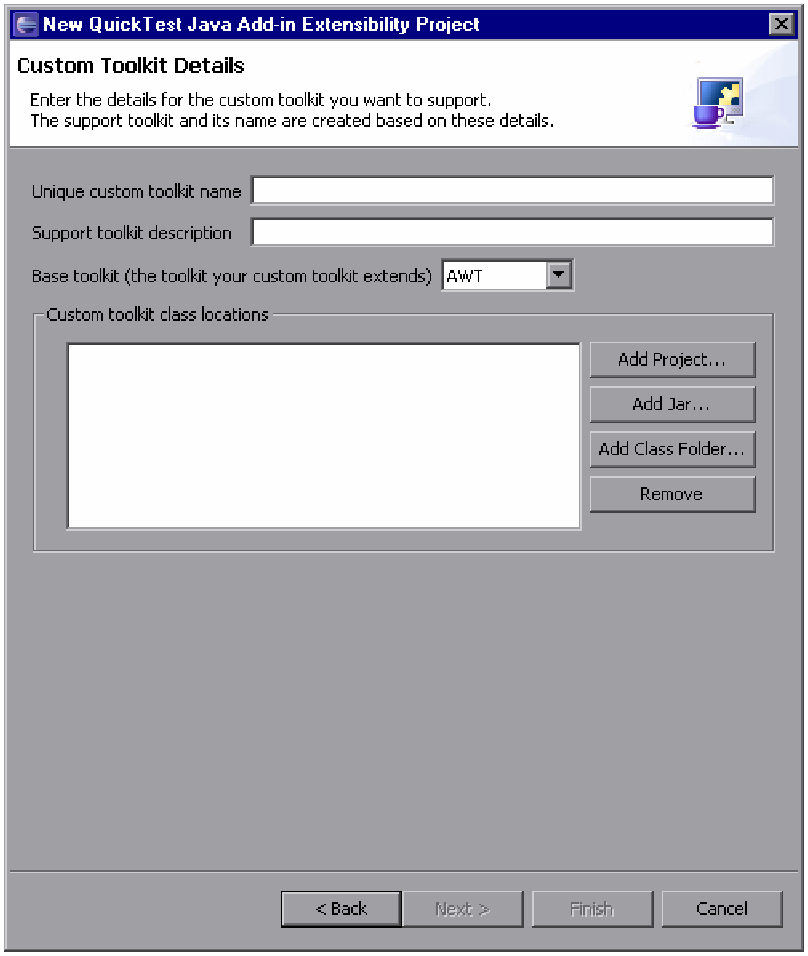
Project layout

- (●) Use project folder as root for sources and class files
- ( ) Create separate source and output folders

Click **Next**. The Custom Toolkit Details screen opens.

3 Enter the custom toolkit details.

In this screen, you provide the details of the ImageControls toolkit so that the wizard can generate a corresponding custom toolkit support set.


New QuickTest Java Add-in Extensibility Project

Custom Toolkit Details

Enter the details for the custom toolkit you want to support.
The support toolkit and its name are created based on these details.

Unique custom toolkit name

Support toolkit description

Base toolkit (the toolkit your custom toolkit extends) AWT

Custom toolkit class locations

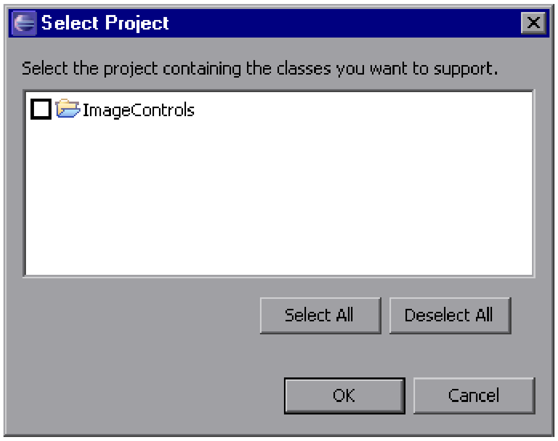
➤ In the **Unique custom toolkit name** you enter a name that uniquely represents the custom toolkit for which you are creating support. The new toolkit support class is given this name plus the suffix-word Support. Providing unique toolkit names allows a single QuickTest installation to support numerous custom toolkit support sets simultaneously.

Enter the name ImageControls.

- In the **Support toolkit description** box enter: ImageControls toolkit support.
- The **Base toolkit** list contains a list of toolkits for which QuickTest support already exists. After you create support for your own toolkits, they are displayed in the list as well.

  The ImageButton custom class extends an **AWT** component, so keep the default selection **AWT** as the **Base toolkit**.
- You must specify the location of the custom classes you want to support in this toolkit. When the new Java Add-in Extensibility project is built, these classes are added to the project build path. You can specify **.jar** files or file system folders for the class location.

  In the **Custom toolkit class locations** area, click **Add project** to select the Eclipse Java project containing the custom classes for the ImageControls toolkit. The Select Project dialog box opens and displays the projects in the current Eclipse workspace.

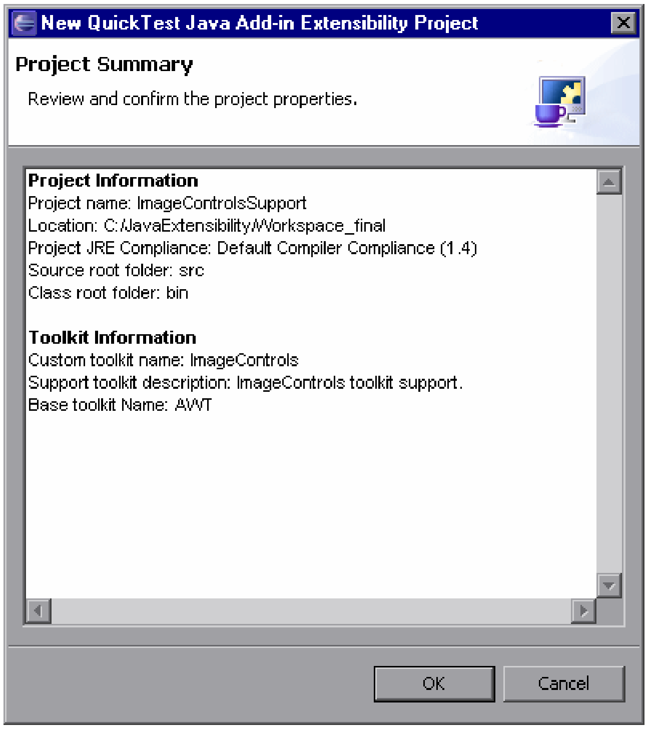
  Select the project containing the classes you want to support.

  ImageControls

  Select the **ImageControls** check box. Click **OK**. The ImageControls project is added in the **Custom toolkit class locations** box.
- Click **Finish**. The Project Summary screen opens.

4 View the Project Summary wizard screen.

Review the details of the project and click **OK**.


New QuickTest Java Add-in Extensibility Project
Project Summary
Review and confirm the project properties.
OK
Cancel

Project Information
Project name: ImageControlsSupport
Location: C:/JavaExtensibility/Workspace_final
Project JRE Compliance: Default Compiler Compliance (1.4)
Source root folder: src
Class root folder: bin

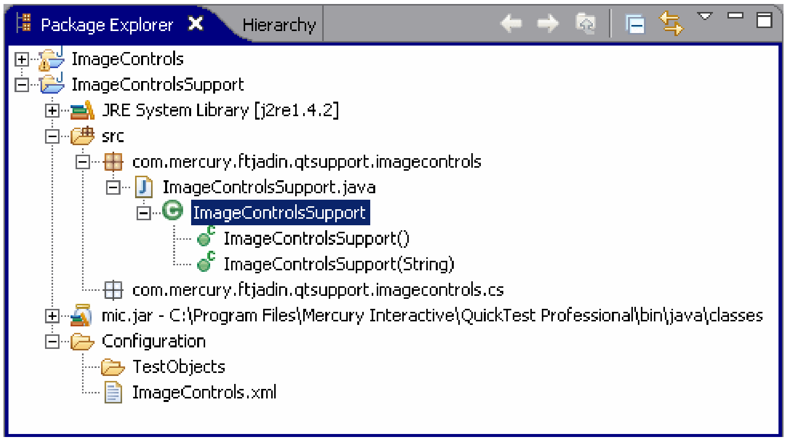
Toolkit Information
Custom toolkit name: ImageControls
Support toolkit description: ImageControls toolkit support.
Base toolkit Name: AWT

Understanding Your New Custom Toolkit Support Set

Your new Java Add-in Extensibility project is displayed in the Package Explorer tab.

Note: If you have more than one JRE installed on your computer, make sure that the ImageControls project and the ImageControlsSupport project are using the same JRE version. If they are not, modify the JRE for one of the projects so that they use the same version.

Expand the **ImageControlsSupport** project to view its content.



```
ImageControls
ImageControlsSupport
    JRE System Library [j2re1.4.2]
    src
        com.mercury.ftjadin.qtsupport.imagecontrols
            ImageControlsSupport.java
                ImageControlsSupport
                    ImageControlsSupport()
                    ImageControlsSupport(String)
        com.mercury.ftjadin.qtsupport.imagecontrols.cs
    mic.jar - C:\Program Files\Mercury Interactive\QuickTest Professional\bin\java\classes
    Configuration
        TestObjects
        ImageControls.xml
```

The **src** folder contains the following packages:

➤ **com.mercury.ftjadin.qtsupport.imagecontrols**

This package contains the new toolkit support class file, **ImageControlsSupport.java**, which defines the new toolkit support class, **ImageControlsSupport**:

```
public class ImageControlsSupport extends AwtSupport {
}
```

The ImageControls toolkit for which you are creating support extends AWT. Therefore, the ImageControls toolkit support class extends the built-in QuickTest **AwtSupport**. No additional implementation is needed in this class.

➤ **com.mercury.ftjadin.qtsupport.imagecontrols.cs**

This package is currently empty. When you create the individual custom control support classes, they are stored is this package.

The **Configuration** folder contains the following items:

- The **TestObjects** folder

  This folder is currently empty. If you create new test object classes to represent the custom controls in your toolkit, a test object configuration file is created in this folder. This is not relevant for this lesson.

- The toolkit configuration file: **ImageControls.xml**

  Open the file to view its content.

```
<Controls
class="com.mercury.ftjadin.qtsupport.imagecontrols.ImageControlsSupport"
SupportClasspath="C:\Documents and Settings\user\workspace1\ImageControl
sSupport\bin" description="ImageControls toolkit support.">
</Controls>
```

At this point, the XML file contains a single **Controls** element that declares the toolkit support class by providing values for the **class**, **SupportClasspath**, and **description** attributes.

When you create the individual custom control support classes, the mapping of each custom control to its support class is added to this configuration file.

Notice that the support class location is currently in your Eclipse workspace. This is appropriate for the development phase of the custom support. When the support is fully implemented and tested, you store the support classes in a more permanent location on a QuickTest computer and update the values in the toolkit configuration file appropriately. For more information, see "Deploying and Running the Custom Toolkit Support" on page 66.

For a complete understanding of the structure of this file, refer to the *QuickTest Java Add-in Extensibility Toolkit Configuration Schema Help* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility Toolkit Configuration Schema**).

Creating a New QuickTest Custom Support Class

In this section you create a custom support class for the ImageButton control, as part of the ImageControls toolkit support. To do this, you use one of the wizards provided by the QuickTest Java Add-in Extensibility plug-in in Eclipse. The details you specify in each wizard screen reflect the decisions you made when planning the custom support. In the subsequent sections you implement the methods that the wizard creates in this class.

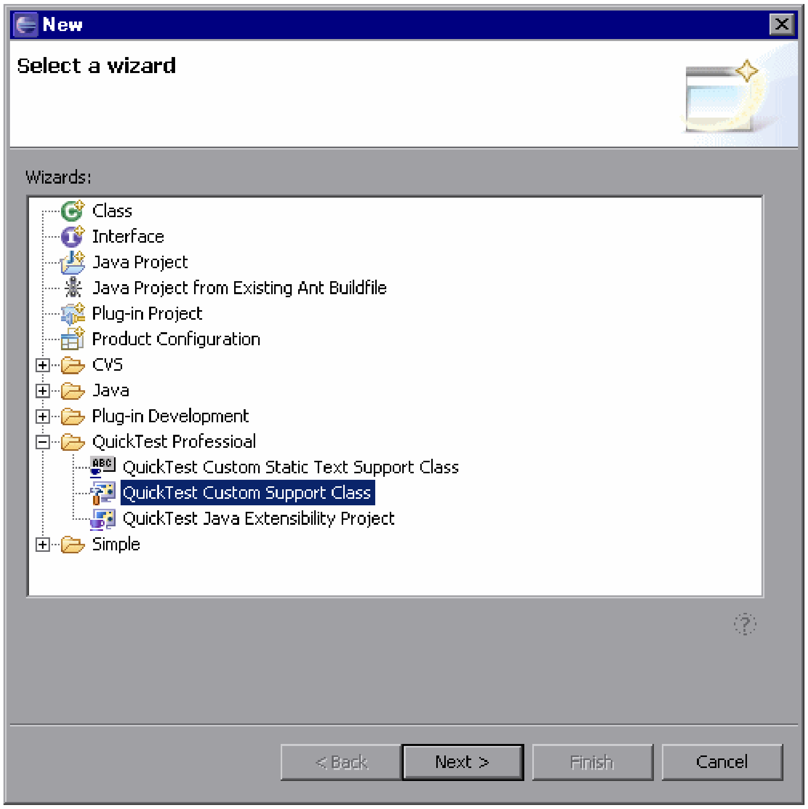
Note: The following sections describe only the options in the wizard screens that are relevant to this lesson. For a complete description of all options available in the wizard screens, see Chapter 5, "Using the QuickTest Java Add-in Extensibility Eclipse Plug-In."

---

1 Open the New QuickTest Custom Support Class wizard.

In the Eclipse Package Explorer tab, select the new QuickTest Java Add-in Extensibility project, **ImageControlsSupport**. Choose **File** > **New** > **Other**. The New dialog box opens.

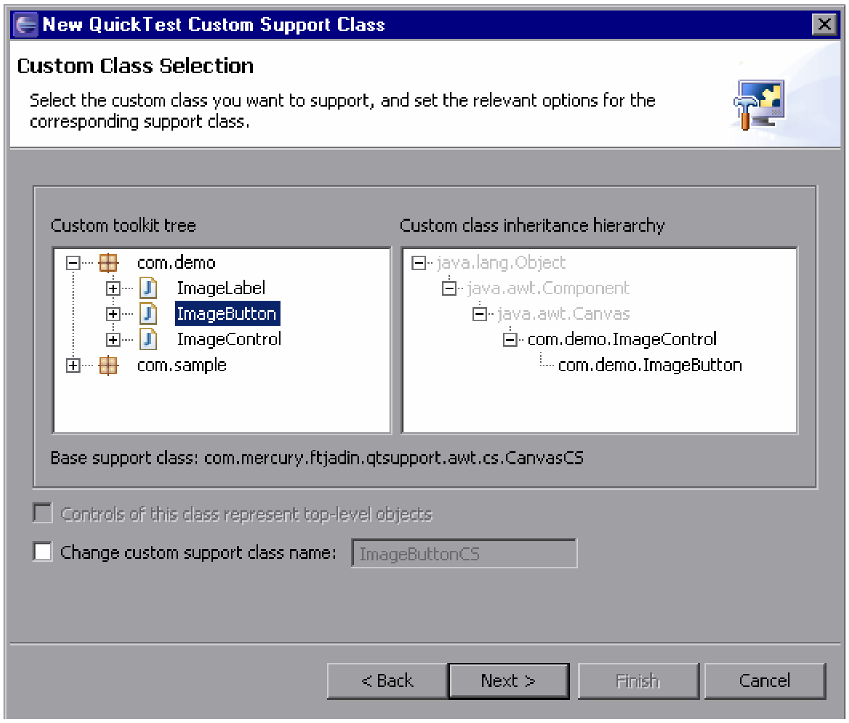
Expand the **QuickTest Professional** folder and select **QuickTest Custom Support Class**.



Click **Next**. The Custom Class Selection screen opens.

2 Select the custom class to support, and set the options for the support class.

Expand the **com.demo** package and select the **ImageButton** class.


New QuickTest Custom Support Class

Custom Class Selection

Select the custom class you want to support, and set the relevant options for the corresponding support class.

Custom toolkit tree

```
com.demo
    ImageLabel
    ImageButton
    ImageControl
com.sample
```

Custom class inheritance hierarchy

```
java.lang.Object
    java.awt.Component
        java.awt.Canvas
            com.demo.ImageControl
                com.demo.ImageButton
```

Base support class: com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS

☐ Controls of this class represent top-level objects

☐ Change custom support class name: ImageButtonCS

In the **Custom toolkit tree** pane, you can see the structure of the ImageControls project, which you selected for the **custom toolkit class location**, in step 3 of “Creating a New QuickTest Java Add-in Extensibility Project” (on page 166). The **com.demo** package contains the ImageControls custom toolkit, with its custom classes, as described in “Preparing for This Lesson” on page 156.

Note: The **com.samples** package is included in the ImageControls sample project, only to provide convenient access for running the sample application. The main content of the ImageControls project is the ImageControls custom toolkit, contained in **com.demo** package.

In the **Custom class inheritance summary pane**, you can see the hierarchy of the **ImageButton** class you have selected. It extends the **ImageControl** class, which is part of the same toolkit, and is therefore shown in black.

The **ImageControl** custom class is not supported, but the **Canvas** class does have a matching support class, provided in the **com.mercury.ftjadin.support.awt.cs** package. Therefore the **Base support class** for the **ImageButton** support class you are creating is **CanvasCS**. This is the class that your new support class extends.

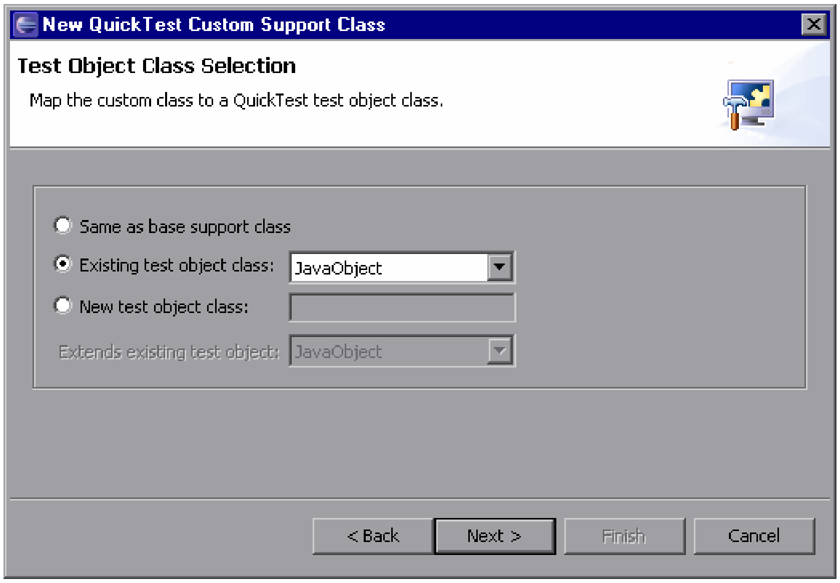
The **Controls of this class represent top-level objects** option is disabled because the ImageButton class is not a container class.

The name for the ImageButton support class is, by default, ImageButtonCS. It is recommended to keep the default name.

Click **Next**. The Test Object Class Selection screen opens.

3 Select a test object class to represent the custom control.

In this screen, you map the custom control to a test object class. In QuickTest tests, the custom class controls are represented by test objects of this test object class. This is the first and most important decision you make when creating a custom support class.


New QuickTest Custom Support Class

Test Object Class Selection

Map the custom class to a QuickTest test object class.

Same as base support class

[✓ Existing test object class:] JavaObject

New test object class:

Extends existing test object: JavaObject

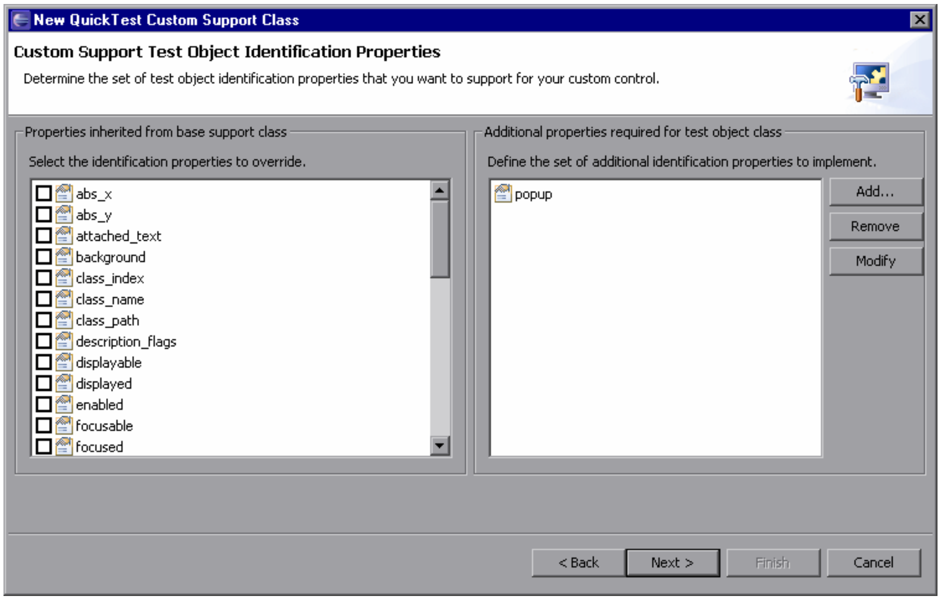
In the previous screen, you determined the support class that the new support class extends. When the test object mapped to the class whose support you are extending is also a logical test object for the custom class, you select **Same as base support class**. The ImageButtonCS class extends CanvasCS, whose test object class does not adequately represent ImageButton controls.

The existing JavaButton test object does answer the needs of your custom support. Select the **Existing test object class** option and select **JavaButton** from the list. The list of existing test objects contains all of the Java objects that QuickTest currently supports. If you define new test objects for custom support, they are included in the list as well.

Click **Next**. The Custom Support Test Object Identification Properties screen opens.

4 Determine the set of test object identification properties to implement in ImageButtonCS.

This screen displays the identification properties supported by the base support class you are extending, as well as additional properties that are defined in the test object class you selected, but are not yet supported. It enables you to select properties whose support you want to implement or override with new functionality and to add new properties.


New QuickTest Custom Support Class

Custom Support Test Object Identification Properties

Determine the set of test object identification properties that you want to support for your custom control.

Properties inherited from base support class

Select the identification properties to override.

- [ ] abs_x
- [ ] abs_y
- [ ] attached_text
- [ ] background
- [ ] class_index
- [ ] class_name
- [ ] class_path
- [ ] description_flags
- [ ] displayable
- [ ] displayed
- [ ] enabled
- [ ] focusable
- [ ] focused

Additional properties required for test object class

Define the set of additional identification properties to implement.

popup

The left pane shows all of the identification properties whose support is implemented by CanvasCS, and therefore inherited by the new ImageButtonCS support class. For most of the properties in this list, the default implementation is sufficient.

➤ Select the **label** property by clicking the check box. After you finish generating the support files using the wizard, you will override the existing support for this property with a custom implementation that matches the needs of the custom control.

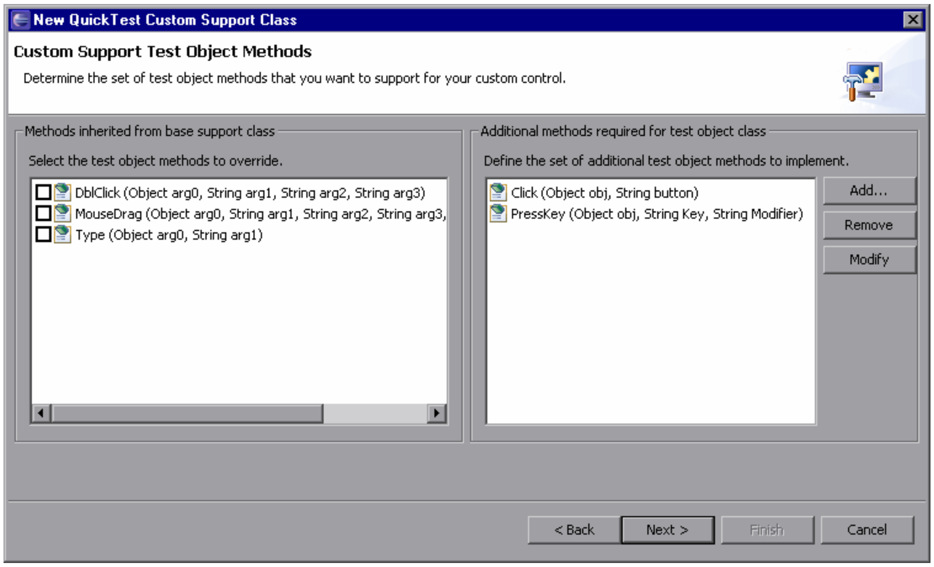
➤ The **popup** identification property is displayed in the right pane because it is a JavaButton property, but it is not supported by CanvasCS. This property is not required for the ImageButton support. Select it, click **Remove**, and then click **Yes** to confirm.

Note: The popup identification property is part of the JavaButton test object identification properties. Removing it from this list means that it is not supported for ImageButton controls. It will still appear in the list of identification properties shown in the QuickTest Object Spy, but will have no value.

➤ Click **Next**. The Custom Support Test Object Methods screen opens.

5 Determine the set of test object methods to implement in ImageButtonCS.

This screen displays the test object methods defined in the test object class you selected. It enables you to select methods whose support you want to implement or override with new functionality and to add new methods.


New QuickTest Custom Support Class

Custom Support Test Object Methods

Determine the set of test object methods that you want to support for your custom control.

Methods inherited from base support class

Select the test object methods to override.

- ☐ DblClick (Object arg0, String arg1, String arg2, String arg3)
- ☐ MouseDrag (Object arg0, String arg1, String arg2, String arg3,
- ☐ Type (Object arg0, String arg1)

Additional methods required for test object class

Define the set of additional test object methods to implement.

- Click (Object obj, String button)
- PressKey (Object obj, String Key, String Modifier)

The left pane shows all of the test object methods (defined in the test object class you selected) whose support is implemented by CanvasCS, and therefore inherited by ImageButtonCS. This existing implementation is sufficient for ImageButton so there is no need to select any methods to override.

In the right pane, you can see the test object methods that are defined for the JavaButton test object class, but are not supported by CanvasCS.

➤ The only test object method that ImageButton requires from this list is the **Click (Object obj, String button)** method. After you finish generating the support files using the wizard, you will implement support for this method.

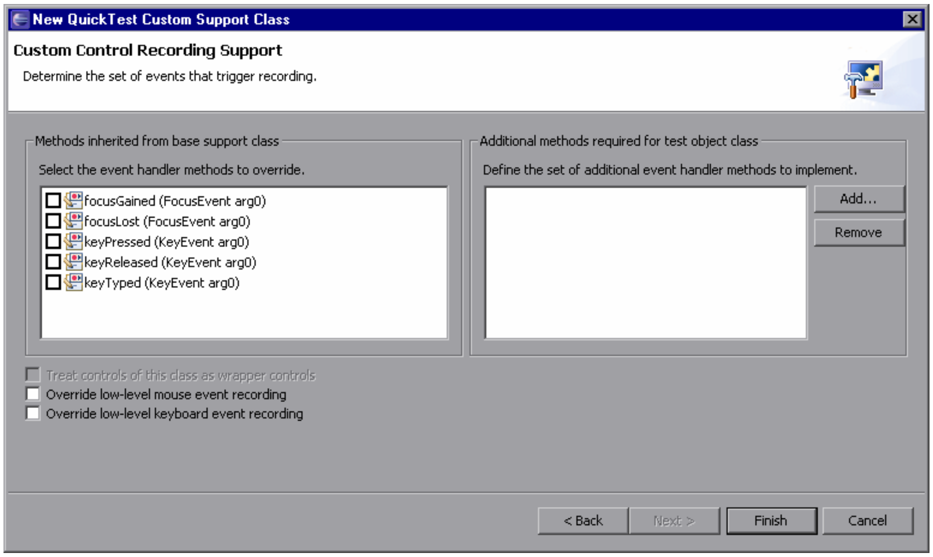
➤ Select the **PressKey** method, click **Remove**, and then click **Yes** to confirm.

Note: The **PressKey** method is one of the JavaButton test object methods. Removing it from this list means that it is not supported for ImageButton controls. It will still appear in the list of test object methods in QuickTest. If you use the **PressKey** method on an ImageButton control in a QuickTest test, a run-time error occurs.

➤ Click **Next**. The Custom Control Recording Support wizard screen opens.

6 Determine the set of events for which to listen, in order to support recording on the ImageButton control.

This screen displays the event listeners implemented by the support class you are extending. It enables you to select event handler methods whose implementation you want to override with new functionality and to add new event listeners to implement.


New QuickTest Custom Support Class

Custom Control Recording Support

Determine the set of events that trigger recording.

Methods inherited from base support class

Select the event handler methods to override.

- [ ] focusGained (FocusEvent arg0)
- [ ] focusLost (FocusEvent arg0)
- [ ] keyPressed (KeyEvent arg0)
- [ ] keyReleased (KeyEvent arg0)
- [ ] keyTyped (KeyEvent arg0)

Additional methods required for test object class

Define the set of additional event handler methods to implement.

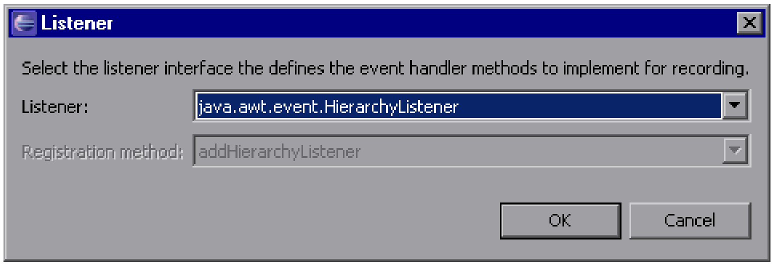
- [ ] Treat controls of this class as wrapper controls
- [ ] Override low-level mouse event recording
- [ ] Override low-level keyboard event recording

In the left pane, you can see the listeners implemented by CanvasCS. You do not have to override any of these for the ImageButtonCS custom support class.

In the right pane, you specify the listeners you want to add for ImageButtonCS. Each listener you select implies a set of event handler methods that the wizard adds to the support class.

➤ Click **Add** to add the **ActionListener**.

The Listener dialog box opens.



Select the listener interface the defines the event handler methods to implement for recording.

Listener: java.awt.event.HierarchyListener

Registration method: addHierarchyListener

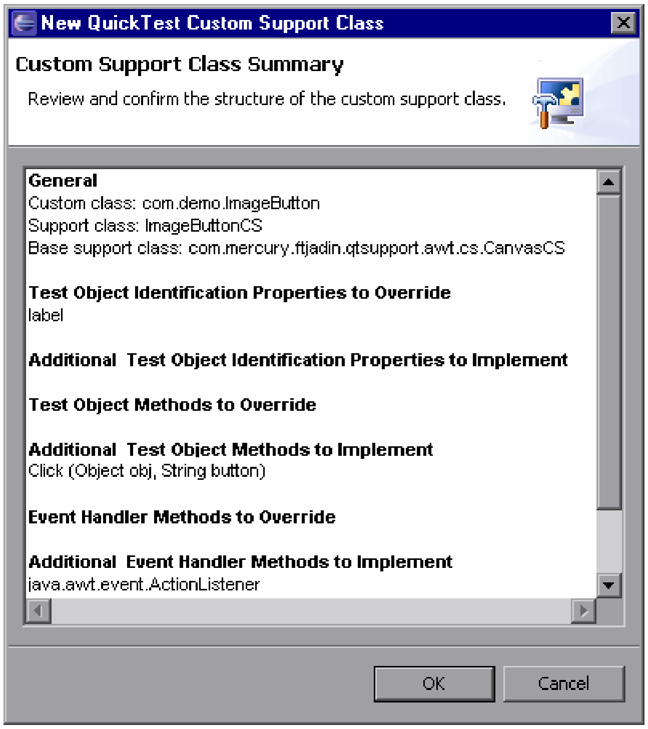
- Select **java.awt.event.ActionListener** from the **Listener** list. If the selected listener had more than one registration method, you would also select a method from the **Registration method** list.
- Click **OK**. The Listener dialog box closes and the **ActionListener**, and all of the event handler methods it includes, are added to the list in the right pane of the wizard screen.

➤ On the Custom Control Recording Support screen, select the **Override low-level mouse event recording** check box to prevent low-level events (coordinate-based operations) from being recorded instead of the events you want to record. For more details on this option, see “Understanding Event Recording Support” on page 184.

➤ Click **Finish**. The Custom Control Support Class Summary screen opens.

7 View the custom control support class summary.

Review the planned content of the custom support class, and click **OK**.



Custom Support Class Summary

Review and confirm the structure of the custom support class.

General
Custom class: com.demo.ImageButton
Support class: ImageButtonCS
Base support class: com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS

Test Object Identification Properties to Override
label

Additional Test Object Identification Properties to Implement

Test Object Methods to Override

Additional Test Object Methods to Implement
Click (Object obj, String button)

Event Handler Methods to Override

Additional Event Handler Methods to Implement
java.awt.event.ActionListener

The following changes are made in the ImageControlsSupport project:

- A new QuickTest custom support class, ImageButtonCS, is created in the **com.mercury.ftjadin.qtsupport.imagecontrols.cs** package. The file is opened and displayed in a tab in the right pane.
- A new **ImageControlsTestObjects.xml** file is created in the **Configuration\TestObjects** folder.
- The **ImageControls.xml** file is modified.

For a detailed explanation of these changes, see “Understanding the New Custom Support,” below.

The asterisk (*****) next to the ImageButtonCS file name (in the ImageButtonCS tab) indicates that it has not been saved. The changes made by the wizard are codependent, and must be saved to prevent discrepancies. Choose **File** > **Save**, or click the **Save** button.

Understanding the New Custom Support

Your new QuickTest Java Add-in Extensibility custom toolkit support set is composed of:

- One toolkit support class: **ImageControlsSupport**, which is created by the wizard when the **ImageControlsSupport** project is created, and not changed.
- One toolkit configuration file: **ImageControls.xml**. This file is created by the wizard when the **ImageControlsSupport** project is created. It is updated with each support class you add for this toolkit.

  The **ImageControls.xml** file is now updated to map the com.demo.ImageButton custom control, to its support class, com.mercury.ftjadin.qtsupport.imagecontrols.cs.ImageButtonCS.

- One test object configuration file: **ImageControlsTestObjects.xml**. Since you did not add any identification properties or test object methods to this the JavaButton test object class, this file does not currently contain any significant information.

  For a complete understanding of the structure of this file, refer to the *QuickTest Test Object Schema Help* (**Help** > **QuickTest Professional Help** > **QuickTest Advanced References** > **QuickTest Test Object Schema**).

- Custom support classes, one per custom class. In this case, you created one custom support class: **ImageButtonCS**.

  The following sections explain the elements that the wizard creates in the **ImageButtonCS** class.

Understanding the Basics of the ImageButtonCS Class

The QuickTest Java Add-in Extensibility wizard creates the custom support class based on the specifications you entered, and registers it in the toolkit support configuration file.

The two most basic characteristics of a support class are:

- the support class it extends
- the test object class mapped to the custom control

Open **ImageButtonCS.java** to review the support class that the wizard created for ImageButton.

The first declaration reflects your selection in the wizard to extend the CanvasCS class:

```
public class ImageButtonCS extends CanvasCS implements ActionListener {
    private static final String DEBUG_IMAGEBUTTONCS =
"DEBUG_IMAGEBUTTONCS";
...
}
```

Note: DEBUG_IMAGEBUTTONCS is defined to control printing log messages. For more information, see “Logging and Debugging the Custom Support Class” on page 71.

The **to_class** property, implemented by the **to_class_attr** method, defines the test object class selected to represent this custom control. QuickTest decides the set of identification properties and test object methods for the test object based on this mapping.

```
    public String to_class_attr(Object obj) {
        return "JavaButton";
    }
```

This implementation is sufficient to provide initial recognition of the custom control in QuickTest.

Understanding Identification Property and Test Object Method Support

Each identification property that can be learned for a particular custom control is represented in the support class, by a method called **<property name>_attr**. Each test object method that can be supported for the control is represented by a method called **<test object method name>_replayMethod**.

When the wizard creates the support class, it inserts stubs for the required methods, according to the identification properties and test object methods that you selected to implement.

The following method stub was added because you selected to override the label identification property, inherited from CanvasCS, in step 4 of "Creating a New QuickTest Custom Support Class" (on page 176):

```
public String label_attr(Object arg0) {
    // TODO Auto-generated method stub
    return super.label_attr(arg0);
}
```

The following method stub was added because you selected to implement the **Click (Object obj)** test object method, in step 5 of "Creating a New QuickTest Custom Support Class" (on page 177):

```
public Retval Click_replayMethod(Object obj, String button) {
    return Retval.NOT_IMPLEMENTED;
}
```

Understanding Event Recording Support

In the ImageButtonCS class, the following elements provide the basis for event recording:

➤ Low-level recording override (enables recording of higher-level events):

```
protected Object mouseRecordTarget(MouseEvent e) {
    return null;
}
```

This method is added because you selected the **Override low-level mouse event recording** check box in step 6 in "Creating a New QuickTest Custom Support Class" (on page 179).

➤ Listing ActionListener for registration on the ImageButton control:

```
public ImageButtonCS() {
        addSimpleListener("ActionListener", "addActionListener",
                                "removeActionListener");
    }
```

This constructor method is added because in step 6 in "Creating a New QuickTest Custom Support Class" (on page 179) you added the **ActionListener** to the list of listeners you want to implement.

The constructor calls the **addSimpleListener** method to add the **ActionListener** to the list of listeners that need to be registered on the custom control.

➤ Action event handler implementation:

```
public void actionPerformed(ActionEvent arg0) {
        try {
            if (!isInRecord())
                return;
            // TODO: Uncomment and edit the call to MicAPI.record
            // MicAPI.record(arg0.getSource(), <Operation>, new
            // String[]{<Parameters>});
            } catch (Throwable th) {
        }
    }
```

The wizard creates this method stub without any actual implementation. You implement it when you get to the step for "Implementing Event Handler Methods to Support Recording" on page 192. The method stub contains the **try..catch** block and the **isInRecord** check, providing a recommendation for this method's structure. For more information, see "Supporting the Record Option" on page 56.

Deploying and Testing the New Custom Toolkit Support

In this part of the lesson, you use the QuickTest **Deploy Toolkit Support** command in Eclipse to deploy the ImageControls toolkit support to QuickTest. Currently only one control in this toolkit, the ImageButton control, is supported. The toolkit support is not yet complete, but you can already test the support created up to this point.

1 Deploy the ImageControls toolkit support to QuickTest.

In the Eclipse Package Explorer tab, select the **ImageControlsSupport** project.

Click the **Deploy Toolkit Support** button, or choose **QuickTest** > **Deploy Toolkit Support**. In the confirmation messages that open, click **Yes** and then **OK.**

The toolkit configuration file and the test object configuration file are copied to the relevant folders in your QuickTest installation folder. The custom support will be available the next time you start the custom application.

For more information on deploying custom toolkit support, see "Deploying and Running the Custom Toolkit Support" on page 66.

2 Test the new custom support.

Repeat steps 1, 2, and 3 in "Planning Support for the ImageButton Control" on page 158, to run the application, view the ImageButton control with the QuickTest Object Spy, and try to record a Click operation on it.

Note: QuickTest establishes its connection with an application when the application opens. Therefore, although you can use an open QuickTest session to test the changes, you must close the SampleApp application, and run it again.

QuickTest recognizes the ImageButton as a JavaButton named ImageButton.

Note: The new support class (ImageButtonCS) inherited some identification properties from the base support class (CanvasCS) that are not included in the JavaButton test object class definition. These properties are displayed in the Custom Support Test Object Identification Properties screen (described on page 176), but they are not displayed in QuickTest in the Object Spy or in the Checkpoint Properties dialog box. You can access these identification properties by using the **GetROProperty** method. For more information on the **GetROProperty** method, refer to the *QuickTest Professional Object Model Reference*.

Because you have overridden the low-level recording, but have not yet implemented the **actionPerformed(ActionEvent arg0)** event handler method, QuickTest currently does not record anything when you click the button.

Changing the Name of the Test Object

In this part of the lesson, you extend QuickTest support of the ImageButton control to recognize its name as per your plan ("Planning Support for the ImageButton Control" on page 158). To do this, you will learn about the special property methods implemented in ObjectCS: **tag_attr** and **attached_text_attr**.

The name of a test object is determined by its **tag** property. All AWT support classes extend ObjectCS. ObjectCS implements the **tag_attr** method to check a set of properties in a specified order, and return the first valid value it finds. A valid value is one that is not empty, and does not contain spaces.

In the **tag_attr** method in the ObjectCS class, the following properties are checked, in the order in which they are listed:

- label
- attached_text (for more detail see below).
- unqualified custom class

The **label** property is implemented in the custom support class with the **label_attr** method. In ImageButtonCS, this method currently returns null, as does its superclass, CanvasCS.

The **attached_text_attr** method is also implemented by ObjectCS. It searches for adjacent static-text objects near the object, and returns their text. This mechanism is useful for controls like edit boxes and list boxes, which do not have their own descriptive text, but are accompanied by a label.

Note: You can teach QuickTest to recognize custom static-text objects using the QuickTest Custom Static-Text Support Class Wizard, which you access from the Eclipse New dialog box. For more information, see "Learning to Support a Custom Static-Text Control" on page 197.

In ImageButton, the **attached_text** property is empty, so QuickTest must use a fallback mechanism. It uses the **unqualified custom class**, which is the name of the class, without the package name. In this case, the custom class: **com.demo.ImageButton** resulted in the name ImageButton for test object.

To change the name of a custom control test object, do not override the **tag_attr** method in the support class. Instead, make use of its existing implementation, and override the method **label_attr**. Override the label_attr method in the ImageButtonCS class

1 Override the label_attr method in the ImageButtonCS class.

In Eclipse, in the **ImageButtonCS.java** file, in the **label_attr** method stub, replace `return super.label_attr(arg0);` with the following code, so that it returns the name of the image file used for the ImageButton (without the full path):

```
ImageButton ib = (ImageButton)arg0;
String res = ib.getImageString();
if(res == null || res.length() == 0)
    return null;
int last = res.lastIndexOf('/');
if(last == -1)
```

```
        return res;
    return res.substring(last+1);
```

Click the **Save** button, or choose **File** > **Save** to save your changes.

Note: You do not have to deploy the toolkit support to QuickTest again because you changed only Java class files and not configuration files.

---

2 Test the new custom support.

Repeat steps 1 and 2 in "Planning Support for the ImageButton Control" on page 158, to run the application and view the ImageButton control with the QuickTest Object Spy.

Note: You can use an open QuickTest session, but you must close the SampleApp application, and run it again, for the changes you made in the custom support to take effect.

QuickTest now recognizes the ImageButton as a JavaButton named **JavaExt1.gif**.

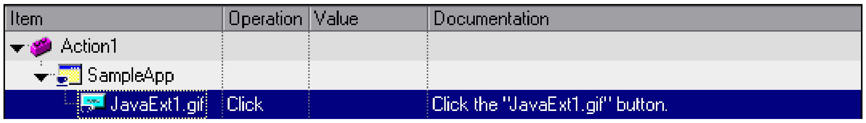
Implementing Support for a Test Object Method

In this section you extend QuickTest support of the ImageButton, to support a Click-the-button test object method. To do this, you must implement the **Click_replayMethod** in the custom support class, to call the appropriate MicAPI function.

1 Test the current functionality of the Click method on an ImageButton.

In QuickTest, create a new test, add the **JavaExt1.gif** button to the object repository, and add a step with this object. For instructions on how to do this, refer to the *QuickTest Professional User's Guide*.

The ImageButton is recognized as a JavaButton item (note the icon used) named **JavaExt1.gif**. The **Click** operation is the default operation for this item, as it is for all JavaButton items.



Item	Operation	Value	Documentation
Action1			
SampleApp			
JavaExt1.gif	Click		Click the "JavaExt1.gif" button.

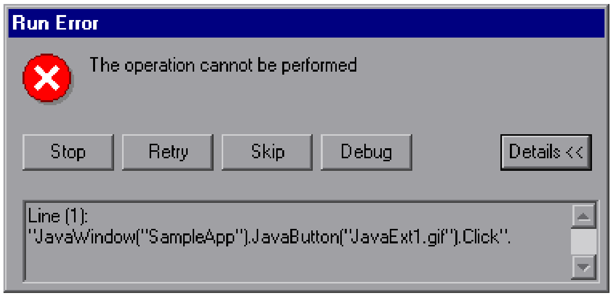
Run

Click **Run** or choose **Automation** > **Run**. The Run dialog box opens.

Select **New run results folder**. Accept the default results folder name.

Click **OK** to close the Run dialog box.

QuickTest runs the test, and an error message is displayed. Click **Details** on the message box. The following information is displayed:



The operation cannot be performed

```
Line (1):
"JavaWindow("SampleApp").JavaButton("JavaExt1.gif").Click".
```

The reason for this error is that in order to run the **Click** operation, the QuickTest calls Click_replayMethod, which is currently implemented in the ImageButtonCS to return the error code NOT_IMPLEMENTED.

Click **Stop**, to stop running the test.

2 Implement the Click_replayMethod method in ImageButtonCS.

In Eclipse, in the **ImageButtonCS.java** file, import com.mercury.ftjadin.custom.MicAPI and replace the **Click_replayMethod** method stub, with the following code:

```
public Retval Click_replayMethod(Object obj, String button) {
        ImageButton ib = (ImageButton) obj;
        MicAPI.mouseClick((Object) ib, ib.getWidth() / 2,
                ib.getHeight() / 2);
        return Retval.OK;
    }
```

Click the **Save** button, or choose **File** > **Save**.

Note: This implementation ignores the button argument. For an implementation that takes this argument into account, you could call a different **MicAPI.mouseClick** method. For more information, refer to the *QuickTest Java Add-in Extensibility API Reference* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility API Reference**).

---

3 Test the new custom support.

Note: You do not have to deploy the toolkit support to QuickTest again because you changed only Java class files and not configuration files.

Close the SampleApp application and run it again.

In QuickTest, run the test you created in step 1 above. The test run completes successfully. As you can see, the click counter in the edit box is increased when the test executes the Click operation.

Implementing Event Handler Methods to Support Recording

Because you planned to support recording on the ImageButton control, you suppressed low-level recording on this object, and registered to listen for Action events on this control.

In this section, you implement the **actionPerformed** listener method, to call **MicAPI.record**, and record the Click operation on the ImageButton object.

1 Implement the actionPerformed listener method to record Click operations.

In Eclipse, in the **ImageButtonCS.java** file, in the **actionPerformed** listener method stub, modify the code to look like this:

```
public void actionPerformed(ActionEvent arg0) {
    try {
        if (!isInRecord())
            return;
        MicAPI.record(arg0.getSource(), "Click");
    } catch (Throwable th) {
        MicAPI.logStackTrace(th);
        }
}
```

The **MicAPI.logStackTrace** method prints a stack trace to the log file containing all of the other Java Add-in Extensibility log messages, and allows you to determine when the **actionPerformed** method was called inadvertently. For more information, see "Logging and Debugging the Custom Support Class" on page 71.

Click the **Save** button, or choose **File** > **Save**.

Note: You do not have to deploy the toolkit support to QuickTest again because you changed only Java class files and not configuration files.

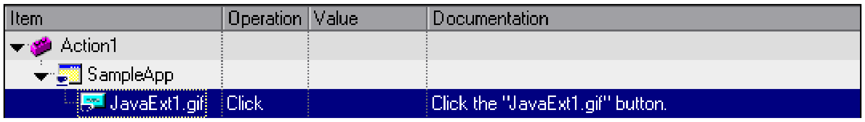
---

2 Test the new custom support.

Close the SampleApp application and run it again.

Create a new test and click the **Record** button or choose **Automation > Record**. If the Record and Run Settings dialog box opens, make sure the **Record and run test on any open Java application** option is selected, and click **OK**. Click the button in the SampleApp application.

A simple Click operation is recorded on the **JavaExt1.gif** JavaButton.



Item	Operation	Value	Documentation
Action1			
SampleApp			
JavaExt1.gif	Click		Click the "JavaExt1.gif" button.

The ImageButton custom control is now fully supported, according to the specifications you decided on when planning your custom support.

Lesson Summary

In this lesson you created support for the ImageButton control, allowing QuickTest to recognize it as a JavaButton test object. You modified the object name, and supported the Click operation.

➤ You learned how to create a toolkit support project, with one custom support class.

➤ You learned to recognize and understand the files that make up the toolkit support.

➤ You learned to use the following identification property support methods:

to_class_attr

tag_attr

label_attr

attached_text_attr

And made use of the following functions:

addSimpleListener

mouseRecordTarget

MicAPI.mouseClick

MicApi.record

Where Do You Go from Here?

For more information on the structure and content of a custom toolkit support set, see "Implementing Custom Toolkit Support" on page 27.

For more information on the toolkit configuration file, refer to the *QuickTest Java Add-in Extensibility Toolkit Configuration Schema Help* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility Toolkit Configuration Schema**).

For more information on the MicAPI methods, refer to the *QuickTest Java Add-in Extensibility API Reference* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility API Reference**).

In the next lesson you learn how to create support for a static-text custom control. Static-text controls normally do not have to support any specific operations; They simply provide a label for adjacent controls. In the support class for a static-text control, simply implementing a set of specific methods provides the necessary support. The New QuickTest Custom Static-Text Support Class Wizard is specifically dedicated to creating custom support for static-text custom controls.

8

Learning to Support a Custom Static-Text Control

In this lesson you create support for the ImageLabel control within the ImageControls toolkit. The ImageLabel control does not have any specific identification properties or test object methods that need to be supported. Its main purpose is to serve as a label. Therefore, you create support for the ImageLabel as a static-text object.

This lesson assumes that you already performed the lesson “Learning to Support a Simple Control” on page 155, in which you created the custom toolkit support set for the custom toolkit ImageControls. In this lesson, you create another support class in the same custom toolkit support set.

This lesson guides you through the following stages:	**On page:**
Preparing for This Lesson	198
Planning Support for the ImageLabel Control	198
Creating the QuickTest Custom Static-Text Support Class	203
Understanding the New Custom Static-Text Support Class	207
Deploying and Testing the New Custom Static-Text Support Class	208
Completing the Support for the Static-Text Control	210
Optimizing the ImageControls Toolkit Support	213
Lesson Summary	223

Preparing for This Lesson

The ImageControls Java project that you created in Eclipse when you prepared for the lesson “Learning to Support a Simple Control” (on page 156), contains the ImageLabel class. The sample application that you ran in that lesson displays the ImageLabel control (to the left of the ImageButton). The purpose of the ImageLabel control in this application is to provide a label for the text box below it, which does not have a label identification property of its own.

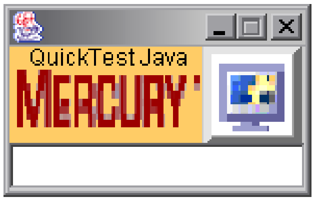
Open Eclipse and locate the **ImageControls** Java project.

Planning Support for the ImageLabel Control

In this section, you analyze the current QuickTest support of the ImageLabel control and the adjacent text box, determine how you want QuickTest to recognize the controls, and fill in the “Custom Class Support Planning Checklist” on page 202, accordingly.

1 Run the SampleApp application and open QuickTest.

In the Eclipse Package Explorer tab, right-click **SampleApp**. Choose **Run As** > **Java Application**. The SampleApp application opens.



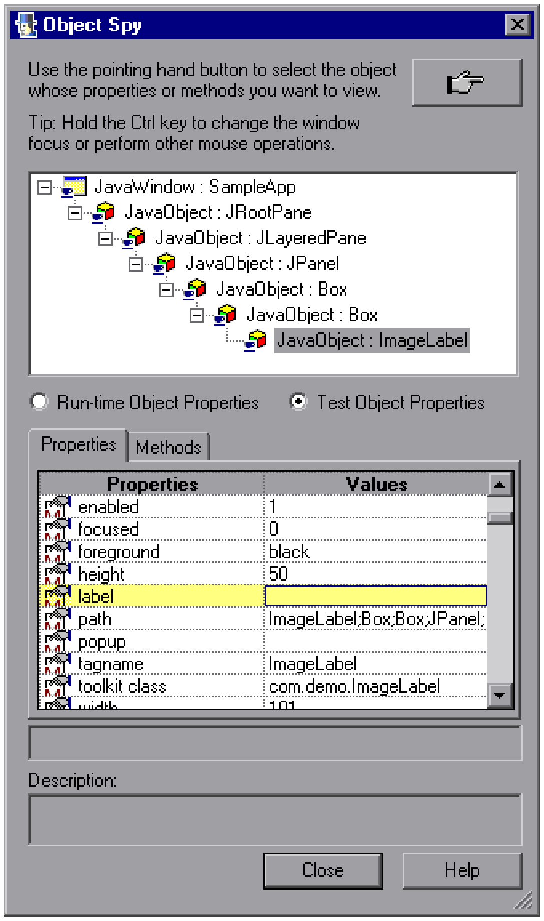
Open QuickTest and load the Java Add-in.

2 Use the Object Spy to view the ImageLabel properties.

In QuickTest Choose **Tools** > **Object Spy** or click the **Object Spy** toolbar button to open the Object Spy dialog box. Click the **Properties** tab.

In the Object Spy dialog box, click the pointing hand, then click the image on the left in the SampleApp application.

The ImageLabel control is based on a custom class that QuickTest does not recognize. Therefore, it recognizes the button as a generic **JavaObject** object named **ImageLabel**, and the icon shown is the standard JavaObject class icon. The **label** identification property is empty.


Object Spy
Use the pointing hand button to select the object whose properties or methods you want to view.
Tip: Hold the Ctrl key to change the window focus or perform other mouse operations.
JavaWindow : SampleApp
JavaObject : JRootPane
JavaObject : JLayeredPane
JavaObject : JPanel
JavaObject : Box
JavaObject : Box
JavaObject : ImageLabel
Run-time Object Properties
Test Object Properties
Properties
Methods
Description:
Close
Help

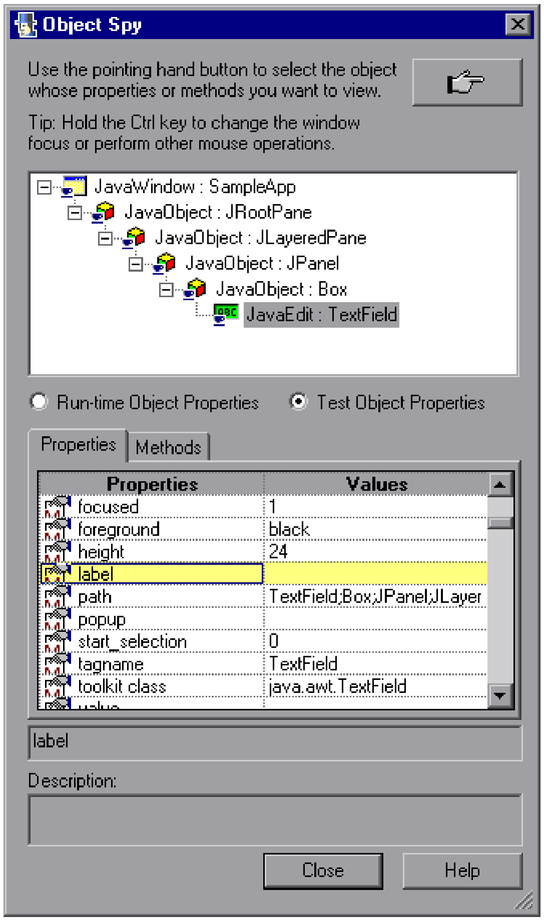
Run-time Object Properties [✓ Test Object Properties]

Properties	Values
enabled	1
focused	0
foreground	black
height	50
label	
path	ImageLabel;Box;Box;JPanel;
popup	
tagname	ImageLabel
toolkit class	com.demo.ImageLabel
width	101

3 Use the Object Spy to view the text box properties.

In the Object Spy dialog box, click the pointing hand, then click the text box in the SampleApp application.

The text box is based on a standard TextField class; therefore QuickTest recognizes it as a **JavaEdit** test object. However, the label identification property is empty and QuickTest does not recognize any adjacent controls as static-text controls. Therefore, the JavaEdit test object is named according to its class name—**TextField**.


Object Spy
Use the pointing hand button to select the object whose properties or methods you want to view.
Tip: Hold the Ctrl key to change the window focus or perform other mouse operations.
Run-time Object Properties
Test Object Properties
Properties
Methods
Description:
Close
Help

```
JavaWindow : SampleApp
  JavaObject : JRootPane
    JavaObject : JLayeredPane
      JavaObject : JPanel
        JavaObject : Box
          JavaEdit : TextField
```

Properties	Values
focused	1
foreground	black
height	24
label	
path	TextField;Box;JPanel;JLayer
popup	
start_selection	0
tagname	TextField
toolkit class	java.awt.TextField

label

Close the Object Spy.

4 Complete the custom class support planning checklist.

The ImageLabel control is a static-text control. You want QuickTest to recognize this fact, and use the ImageLabel’s **label** property as **attached text** for adjacent controls that do not have their own **label** property.

The ImageLabel displays an image file optionally accompanied by additional text. When the control does not display any text, the name of the test object that represents the control can be based on the name of the image file that the control displays.

The ImageLabel itself does not have any additional identification properties or test object methods that need to be identified in QuickTest tests. In addition, there is no need to record any operations on the ImageLabel control.

On the next page you can see the checklist, completed based on the information above.

Custom Class Support Planning Checklist

☑	**Custom Class Support Planning Checklist**
☐	Does the custom class have a superclass for which QuickTest custom support is not yet available? **Yes** / **~~No~~**
☐	If so, should I first extend support for a control higher in the hierarchy? **~~Yes~~** / **No**
☐	Do I have an application that runs the custom control on a computer with QuickTest installed? **Yes** / **~~No~~**
☐	The sources for this custom control class are located in: an Eclipse project called ImageControls
☐	Which existing Java test object matches the custom control? JavaStaticText
☐	If none, create a new Java test object class named: N/A • New test object class extends: (Default—JavaObject) • Icon file location (optional): • Identification property for description: • Default test object method:
☐	Is the custom control a top-level object? **~~Yes~~** / **No**
☐	Is the custom control a wrapper? **~~Yes~~** / **No**
☐	Specify the basis for naming the test object: its text or (if there is no text) its image file name
☐	List the identification properties to support, and mark default checkpoint properties: nothing special
☐	List the test object methods to support (include arguments and return values if required): nothing special
☐	Provide support for recording? (AWT-based only) **~~Yes~~** / **No**
☐	If so, list the events that should trigger recording: N/A

Creating the QuickTest Custom Static-Text Support Class

In the lesson "Learning to Support a Simple Control", you created the ImageControlsSupport QuickTest Java Add-in Extensibility project (as described on page 164). In that project, you created the custom support class for the ImageButton control.

In this section you create another custom support class in the same project to support the ImageLabel control.

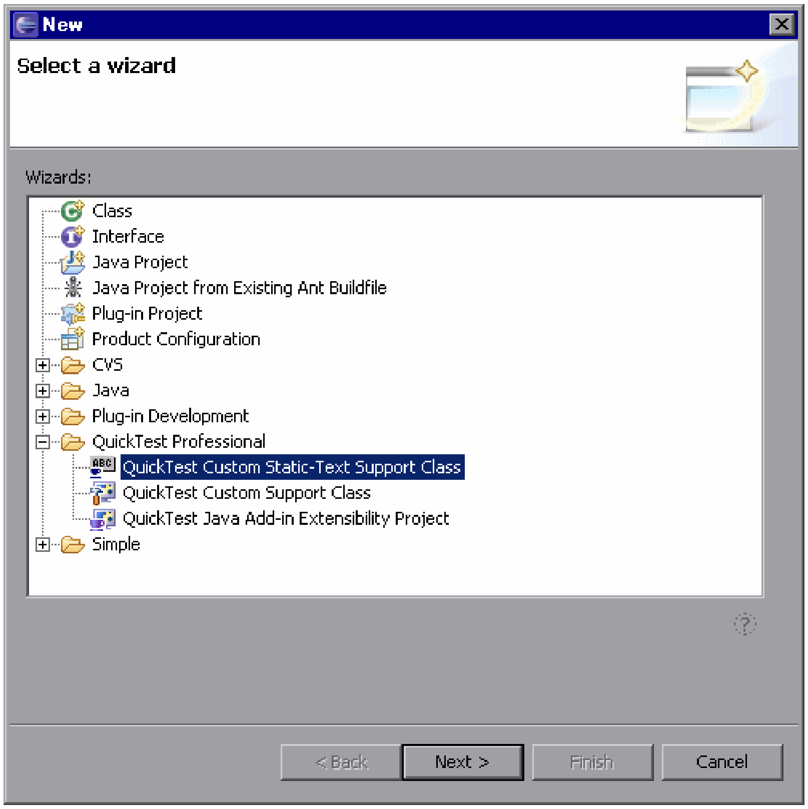
In most cases, static-text controls do not have identification properties or test object methods that need to be identified in QuickTest tests. In addition, there is usually no need to record any operations on a static-text control. Therefore, the QuickTest Java Add-in Extensibility Eclipse plug-in provides a special wizard for creating support classes for static-text controls.

In this wizard, all you have to do is select the ImageLabel class to be supported as a static-text control. The wizard creates the new support class with all the required methods. After the wizard creates the new support class, you modify the methods that the wizard creates to complete the support.

1 Open the New QuickTest Custom Static-Text Support Class wizard.

In the Eclipse Package Explorer tab, select the QuickTest Java Add-in Extensibility project, **ImageControlsSupport**. Choose **File** > **New** > **Other**. The New dialog box opens.

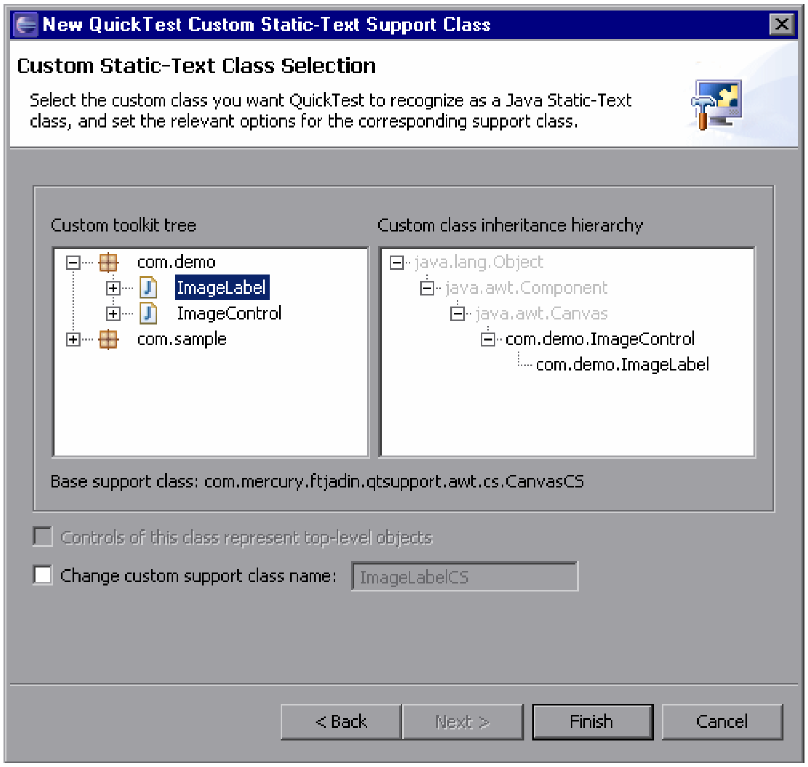
Expand the **QuickTest Professional** folder and select **QuickTest Custom Static-Text Support Class**.



Click **Next**. The Custom Class Selection screen opens.

2 Select the custom class to support, and set the options for the support class.

Expand the **com.demo** package and select the **ImageLabel** class.



Custom Static-Text Class Selection

Select the custom class you want QuickTest to recognize as a Java Static-Text class, and set the relevant options for the corresponding support class.

Custom toolkit tree

```
com.demo
    ImageLabel
    ImageControl
com.sample
```

Custom class inheritance hierarchy

```
java.lang.Object
    java.awt.Component
        java.awt.Canvas
            com.demo.ImageControl
                com.demo.ImageLabel
```

Base support class: com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS

☐ Controls of this class represent top-level objects

☐ Change custom support class name: ImageLabelCS

Since you are creating support for a class in the ImageControls custom toolkit, the **Custom toolkit tree** pane looks similar to the one in the lesson “Learning to Support a Simple Control”, as shown in step 2 of “Creating a New QuickTest Custom Support Class” (on page 173). The **Custom toolkit tree** represents the list of classes that you can select to support. The ImageButton class does not appear in this list because you already created support for it.

In the **Custom class inheritance summary pane**, you can see the hierarchy of the **ImageLabel** class you have selected. It extends the **ImageControl** class, which is part of the same toolkit, and is therefore shown in black.

The **ImageControl** custom class is not supported, but the **Canvas** class does have a matching support class, provided in the **com.mercury.ftjadin.support.awt.cs** package. Therefore the **Base support class** for the **ImageLabel** support class you are creating is **CanvasCS**. This is the class that your new support class extends.

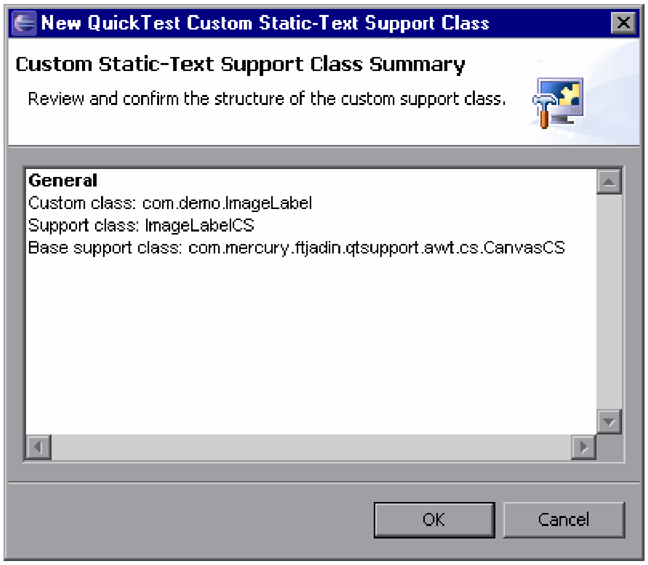
The **Controls of this class represent top-level objects** option is disabled because the ImageLabel class is not a container class.

The name for the ImageLabel support class is, by default, ImageLabelCS. It is recommended to keep the default name.

Click **Finish**. The Custom Static-Text Support Class Summary screen opens.

3 View the custom static-text control support class summary.

Review the planned content of the custom static-text support class, and click **OK**.


New QuickTest Custom Static-Text Support Class
Custom Static-Text Support Class Summary
Review and confirm the structure of the custom support class.
OK
Cancel

General
Custom class: com.demo.ImageLabel
Support class: ImageLabelCS
Base support class: com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS

The following changes are made in the ImageControlsSupport project:

➤ The **ImageControls.xml** file is modified to map the ImageLabel custom class to its support class—ImageLabelCS.

- A new QuickTest custom support class, ImageLabelCS, is created in the **ImageLabelCS.java** file in the **com.mercury.ftjadin.qtsupport.imagecontrols.cs** package. The file is opened and displayed in a tab in the right pane.

  For a detailed explanation of the content of the ImageLabelCS class, see "Understanding the New Custom Static-Text Support Class," below.

  The asterisk (*****) next to the ImageLabelCS file name (in the ImageLabelCS tab) indicates that it has not been saved. The changes made by the wizard are codependent, and must be saved to prevent discrepancies. Choose **File** > **Save**, or click the **Save** button.

Understanding the New Custom Static-Text Support Class

Examine the contents of the new **ImageLabelCS.java** file. The ImageLabelCS custom static-text support class extends CanvasCS.

In the new support class, the wizard created stubs for the following methods:

- **class_attr.** Returns the string static_text.

  This informs QuickTest that the ImageLabel control is a JavaStaticText object. This means that the QuickTest mechanism that searches for attached text can use the ImageLabel's **label** property as **attached text** for adjacent controls.

- **label_attr.** Returns the label property of the superclass (in this case CanvasCS).

  This method defines ImageLabel's **label** identification property. The text in this identification property is used for adjacent controls' **attached text**. The wizard includes a comment in this method stub, reminding you to implement it to return the appropriate text.

- **tag_attr.** This method supports the **tag** property, which represents the name of the static-text test object.

  In the lesson "Learning to Support a Simple Control", in the section "Changing the Name of the Test Object" on page 187, you learned how the **tag** property is implemented. The **tag_attr** method in the support class that the wizard creates returns super.tag_attr(obj) with the added suffix (st). This means that the name for the static-text test object is derived by using the same logic as for regular test objects (label, attached text or unqualified class name), and adding (st) at the end.

➤ **value_attr.** Returns the **label** property.

The **value** property represents a control's test object state. For static-text controls, the **label** property adequately represents this state.

For more information on these special identification properties, see "Special Identification Property Support Methods" on page 52.

Deploying and Testing the New Custom Static-Text Support Class

In this section, you use the QuickTest **Deploy Toolkit Support** command in Eclipse to deploy the ImageControls toolkit support to QuickTest. This adds the ImageLabel support to QuickTest, in addition to the ImageButton control whose support you deployed previously. The ImageLabel support is not yet complete, but you can already test the support created up to this point.

1 Deploy the ImageControls toolkit support to QuickTest.

In the Eclipse Package Explorer tab, select the **ImageControlsSupport** project.

Click the **Deploy Toolkit Support** button, or choose **QuickTest** > **Deploy Toolkit Support**. In the confirmation messages that open, click **Yes** and then **OK.**

The toolkit configuration file and the test object configuration file are copied to the relevant folders in your QuickTest installation folder. The custom support will be available the next time you start the custom application.

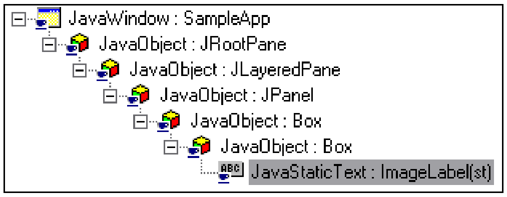
For more information on deploying custom toolkit support, see "Deploying and Running the Custom Toolkit Support" on page 66.

2 Test the new custom support.

Repeat steps 1 and 2 in “Planning Support for the ImageLabel Control” on page 198, to run the application, and view the ImageLabel control and the text box with the QuickTest Object Spy.

Note: QuickTest establishes its connection with an application when the application opens. Therefore, although you can use an open QuickTest session to test the changes, you must close the SampleApp application, and run it again.

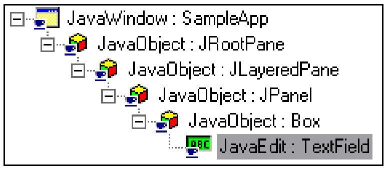
QuickTest recognizes the ImageLabel as a **JavaStaticText** object named **ImageLabel(st)**.



```
JavaWindow : SampleApp
  JavaObject : JRootPane
    JavaObject : JLayeredPane
      JavaObject : JPanel
        JavaObject : Box
          JavaObject : Box
            JavaStaticText : ImageLabel(st)
```

CanvasCS, which ImageLabelCS extends, does not provide support for the **label** identification property. Therefore, ImageLabel’s **label** property is empty (as is its **attached text** property). As a result, the superclass **tag** property returns ImageLabel’s class name, and ImageLabel’s **tag** property is **ImageLabel(st)**.

QuickTest still identifies the text box as a **JavaEdit** test object named **TextField** (its class name) because the **label** property of the adjacent static-text object, **ImageLabel**, is still empty.



```
JavaWindow : SampleApp
  JavaObject : JRootPane
    JavaObject : JLayeredPane
      JavaObject : JPanel
        JavaObject : Box
          JavaEdit : TextField
```

Completing the Support for the Static-Text Control

In this part of the lesson, you implement the **label_attr** method in the ImageLabelCS class to return the name of the image file used for the ImageLabel. This enables QuickTest to use the ImageLabel's **label** property as **attached text** for adjacent controls. In addition, implementing the ImageLabel's **label** property provides the ImageLabel test object with a more specific name.

1 Implement the label_attr method in the ImageLabelCS class.

In Eclipse, in the **ImageLabelCS.java** file, in the **label_attr** method stub, replace return super.label_attr(obj); with the following code:

```
ImageLabel il = (ImageLabel)obj;
String res = il.getText();
if(res != null && res.length() > 0)
    return res;
res = il.getImageString();
if(res == null || res.length() == 0)
    return null;
int last = res.lastIndexOf('/');
if(last == -1)
    return res;
return res.substring(last+1);
```

The label identification property returns the text on the label (if it exists) or the name of the image file used for the ImageLabel (without the full path).

Click the **Save** button, or choose **File** > **Save** to save your changes.

Note: You do not have to deploy the toolkit support to QuickTest again because you changed only Java class files and not configuration files.

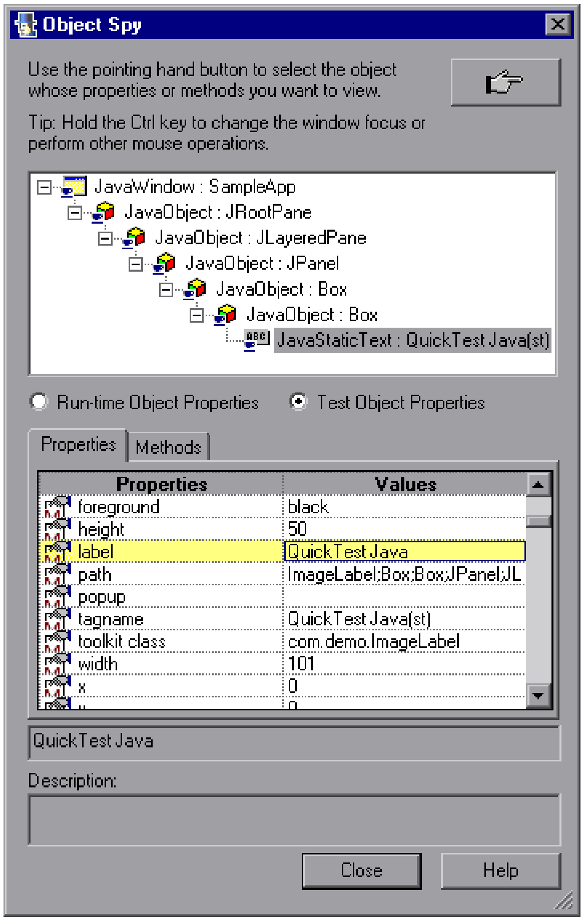
---

2 Test the new custom support.

Repeat steps 1 and 2 in "Planning Support for the ImageLabel Control" on page 198, to run the application and view the ImageLabel control and the text box with the QuickTest Object Spy.

Note: You can use an open QuickTest session, but you must close the SampleApp application, and run it again, for the changes you made in the custom support to take effect.

QuickTest now recognizes the ImageLabel as a **JavaStaticText** test object named **QuickTest Java(st)**, with the **label** property **QuickTest Java**.


Object Spy
Use the pointing hand button to select the object whose properties or methods you want to view.
Tip: Hold the Ctrl key to change the window focus or perform other mouse operations.

JavaWindow : SampleApp
JavaObject : JRootPane
JavaObject : JLayeredPane
JavaObject : JPanel
JavaObject : Box
JavaObject : Box
JavaStaticText : QuickTest Java(st)

Run-time Object Properties [✓ Test Object Properties]

Properties | Methods

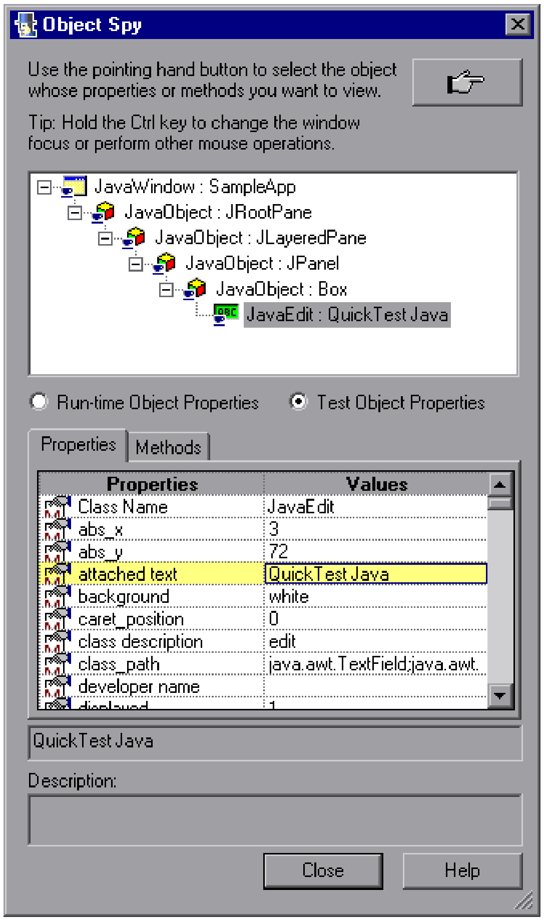
Properties	Values
foreground	black
height	50
label	QuickTest Java
path	ImageLabel;Box;Box;JPanel;JL
popup	
tagname	QuickTest Java(st)
toolkit class	com.demo.ImageLabel
width	101
x	0
y	0

QuickTest Java

Description:

QuickTest now recognizes the text box as a **JavaEdit** test object named **QuickTest Java**. The **label** property of the **JavaEdit** test object is empty. The

ImageLabel's **label** property provides the text for the JavaEdit's **attached text** property, which is used as the test object name.


Object Spy
Use the pointing hand button to select the object whose properties or methods you want to view.
Tip: Hold the Ctrl key to change the window focus or perform other mouse operations.
Run-time Object Properties
Test Object Properties
Properties
Methods
Description:
Close
Help

```
JavaWindow : SampleApp
  JavaObject : JRootPane
    JavaObject : JLayeredPane
      JavaObject : JPanel
        JavaObject : Box
          JavaEdit : QuickTest Java
```

Properties	Values
Class Name	JavaEdit
abs_x	3
abs_y	72
attached text	QuickTest Java
background	white
caret_position	0
class description	edit
class_path	java.awt.TextField;java.awt.
developer name	
displayed	1

QuickTest Java

Note: If you modify the SampleApp application and remove the line imageLb.setText("QuickTest Java");, the ImageLabel will not display any text. QuickTest will then recognize the ImageLabel as a **JavaStaticText** test object named **mercury.gif(st)**, with the **label** property **mercury.gif**. QuickTest will recognize the text box as a **JavaEdit** test object named **mercury.gif**.

The ImageLabel static-text custom control is now fully supported, according to the specifications you determined when planning your custom support. The support for the ImageControls toolkit is now complete. You can find a ready-made example of this support in the **<QuickTest Professional Java Add-in Extensibility SDK installation folder>\samples\ImageControlsSupport** folder.

Optimizing the ImageControls Toolkit Support

Note that the implementation you used for the **label** identification property in the ImageLabel class is very similar to the implementation of the **label** identification property in the ImageButton class. Since both of these classes extend the ImageControl class, it might have been preferable to implement support for the **label** identification property in a support class for the ImageControl (ImageControlCS).

This means that when planning support for the ImageButton and ImageLabel controls, the answer to the second question in the "Custom Class Support Planning Checklist" on page 202 would have been **Yes** (I should first extend support for a control higher in the hierarchy). ImageButtonCS and ImageLabelCS would then extend ImageControlCS, and in ImageLabelCS you would fine-tune the **label** property by overriding the inherited **label_attr** method.

In the following sections you modify the ImageControls toolkit support set to prevent the duplicate implementation of the **label_attr** method. The changes do not affect the functionality of the support. You create the ImageControlCS support class and modify ImageButtonCS and ImageLabelCS to extend ImageControlCS.

Creating Support for the ImageControl Custom Class

In this section, you create a custom support class for the ImageControl class in the ImageControlsSupport project.

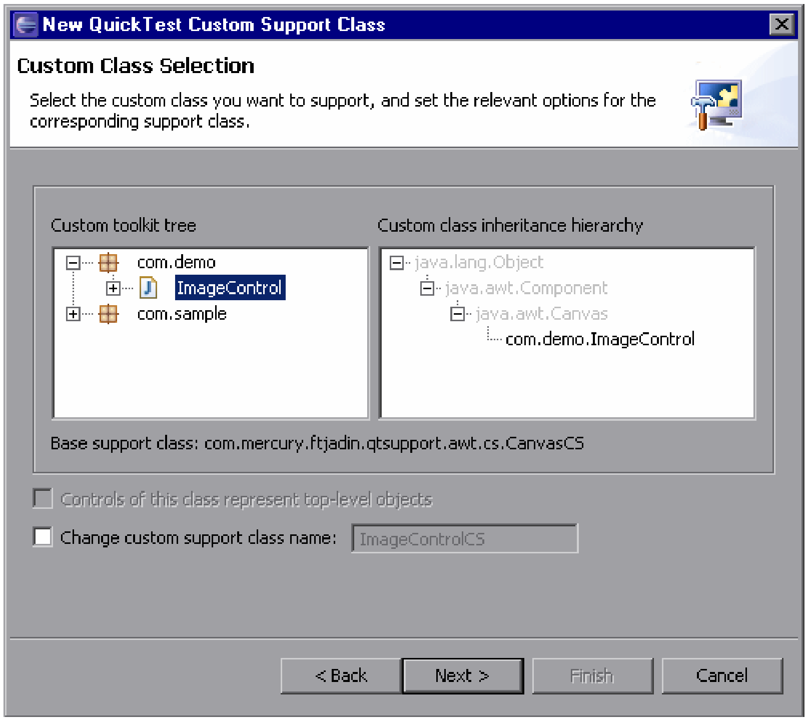
1 Open the New QuickTest Custom Support Class wizard.

In the Eclipse Package Explorer tab, select the new QuickTest Java Add-in Extensibility project, **ImageControlsSupport**. Choose **File** > **New** > **Other**. The New dialog box opens.

Expand the **QuickTest Professional** folder, select **QuickTest Custom Support Class** and click **Next**. The Custom Class Selection screen opens.

2 Select the custom class to support, and set the options for the support class.

➤ Expand the **com.demo** package and select the **ImageControl** class.


New QuickTest Custom Support Class

Custom Class Selection

Select the custom class you want to support, and set the relevant options for the corresponding support class.

Custom toolkit tree

```
⊟ com.demo
   ⊞ ImageControl
⊞ com.sample
```

Custom class inheritance hierarchy

```
⊟ java.lang.Object
   ⊟ java.awt.Component
      ⊟ java.awt.Canvas
         com.demo.ImageControl
```

Base support class: com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS

☐ Controls of this class represent top-level objects

☐ Change custom support class name: ImageControlCS

In the **Custom toolkit tree** pane, you can see that the ImageControl class is the only class in the **com.demo** package that is not yet supported.

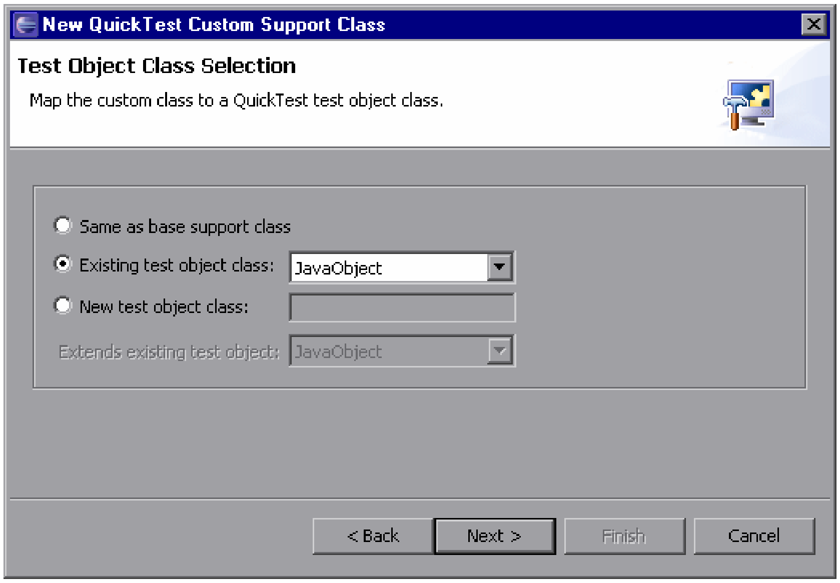
In the **Custom class inheritance summary pane**, you can see the hierarchy of the **ImageControl** class you have selected. The **ImageControl** class extends **java.awt.Canvas**, therefore the **Base support class** for the ImageControl support class you are creating is **CanvasCS**.

Leave the default name, **ImageControlCS**, for the ImageControl support class.

➤ Click **Next**. The Test Object Class Selection screen opens.

3 Select a test object class to represent the custom control.

You are creating the ImageControlCS support class only to use it as a base support class for other support classes, not to support actual controls. Therefore, it is not important to which test object class you map the ImageControl custom class.


New QuickTest Custom Support Class

Test Object Class Selection

Map the custom class to a QuickTest test object class.

Same as base support class [✓ Existing test object class: JavaObject] New test object class:

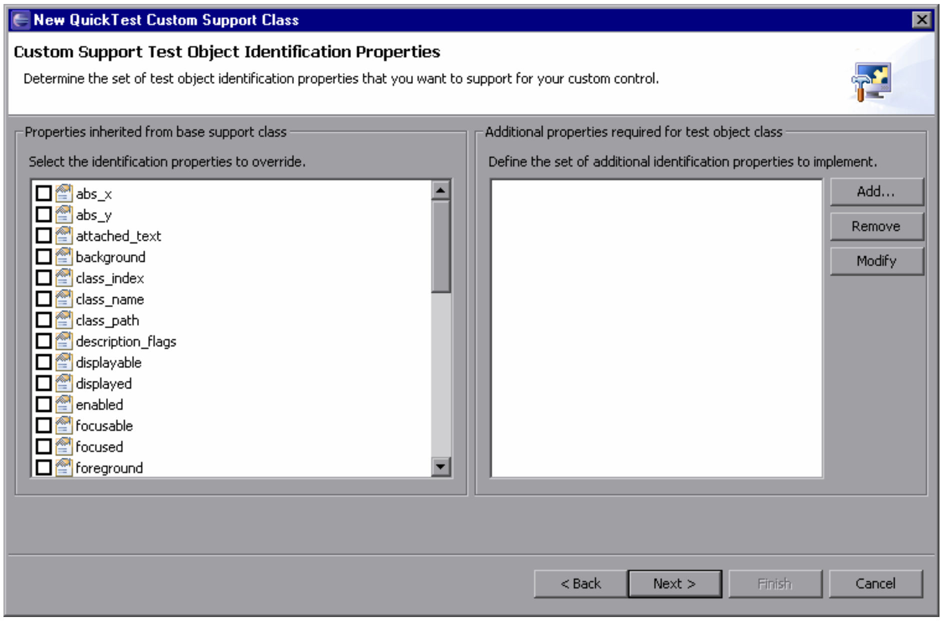
Extends existing test object: JavaObject

➤ Select **Same as base support class**. This maps the ImageControl custom class to whichever test object class is mapped to java.awt.Canvas. No direct mapping takes place. The new support class does not implement a **to_class_attr** method, but inherits it from the base support class.

➤ Click **Next**. The Custom Support Test Object Identification Properties screen opens.

4 Determine the set of test object identification properties to implement in ImageControlCS.

This screen displays the identification properties supported by the base support class you are extending, as well as additional properties that are defined in the test object class you selected, but are not yet supported.


New QuickTest Custom Support Class

Custom Support Test Object Identification Properties

Determine the set of test object identification properties that you want to support for your custom control.

Properties inherited from base support class

Select the identification properties to override.

- ☐ abs_x
- ☐ abs_y
- ☐ attached_text
- ☐ background
- ☐ class_index
- ☐ class_name
- ☐ class_path
- ☐ description_flags
- ☐ displayable
- ☐ displayed
- ☐ enabled
- ☐ focusable
- ☐ focused
- ☐ foreground

Additional properties required for test object class

Define the set of additional identification properties to implement.

The left pane displays all of the identification properties whose support is implemented by CanvasCS, and therefore inherited by the new ImageControlCS support class. It enables you to select properties whose support you want to override with new functionality.

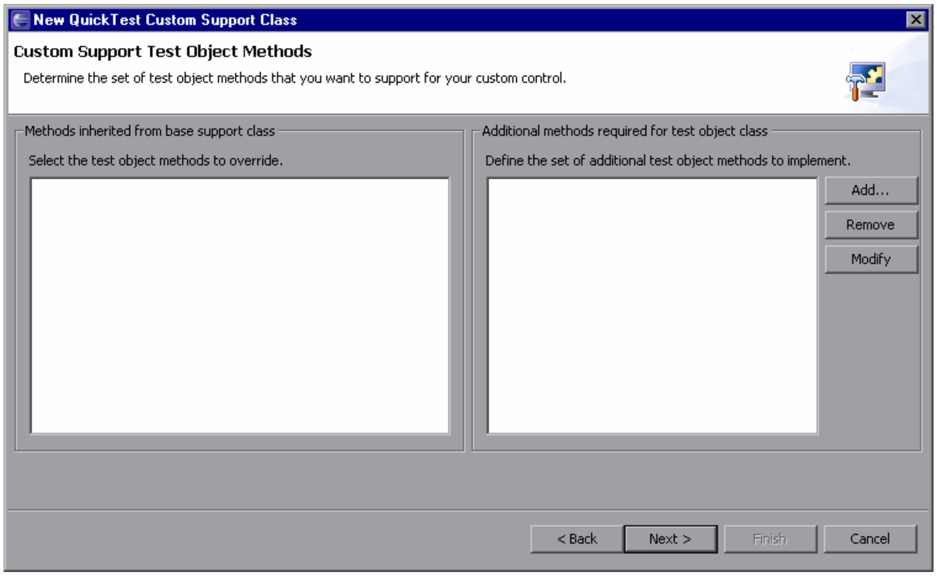
In the Test Object Class Selection screen (on page 215), you did not select a specific test object class. Therefore, the wizard does not know which test object class is mapped to the ImageControl custom control. As a result, no identification properties are displayed in the right pane.

➤ Select the **label** property by clicking its check box. After you finish generating the support files using the wizard, you will override the existing support for this property with a custom implementation that matches the needs of the custom control.

➤ Click **Next**. The Custom Support Test Object Methods screen opens.

5 Determine the set of test object methods to implement in ImageControlCS.

This screen displays the test object methods defined in the test object class you selected.


New QuickTest Custom Support Class
Custom Support Test Object Methods
Add...
Remove
Modify
< Back
Next >
Finish
Cancel

Determine the set of test object methods that you want to support for your custom control.

Methods inherited from base support class

Select the test object methods to override.

Additional methods required for test object class

Define the set of additional test object methods to implement.

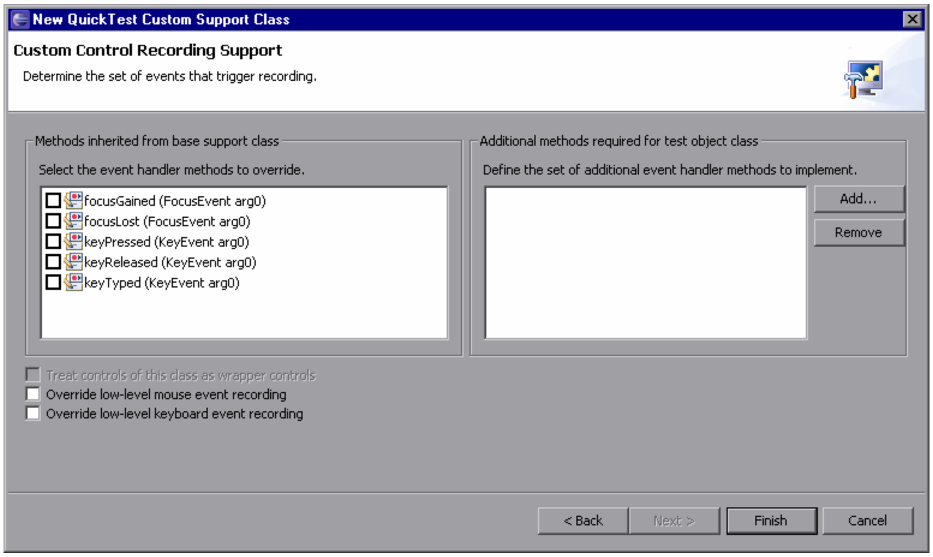
In the Test Object Class Selection screen (on page 215), you did not select a specific test object class. Therefore, the wizard does not know which test object class is mapped to the ImageControl custom control. As a result, no test object methods are displayed in this screen.

The ImageControl custom control does not have any test object methods that need to be supported.

Click **Next**. The Custom Control Recording Support wizard screen opens.

6 Determine the set of events for which to listen, in order to support recording on the ImageControl control.

This screen displays the event listeners implemented by the support class you are extending. It enables you to select event handler methods whose implementation you want to override with new functionality and to add new event listeners to implement.


New QuickTest Custom Support Class

Custom Control Recording Support

Determine the set of events that trigger recording.

Methods inherited from base support class

Select the event handler methods to override.

- [ ] focusGained (FocusEvent arg0)
- [ ] focusLost (FocusEvent arg0)
- [ ] keyPressed (KeyEvent arg0)
- [ ] keyReleased (KeyEvent arg0)
- [ ] keyTyped (KeyEvent arg0)

Additional methods required for test object class

Define the set of additional event handler methods to implement.

- [ ] Treat controls of this class as wrapper controls
- [ ] Override low-level mouse event recording
- [ ] Override low-level keyboard event recording

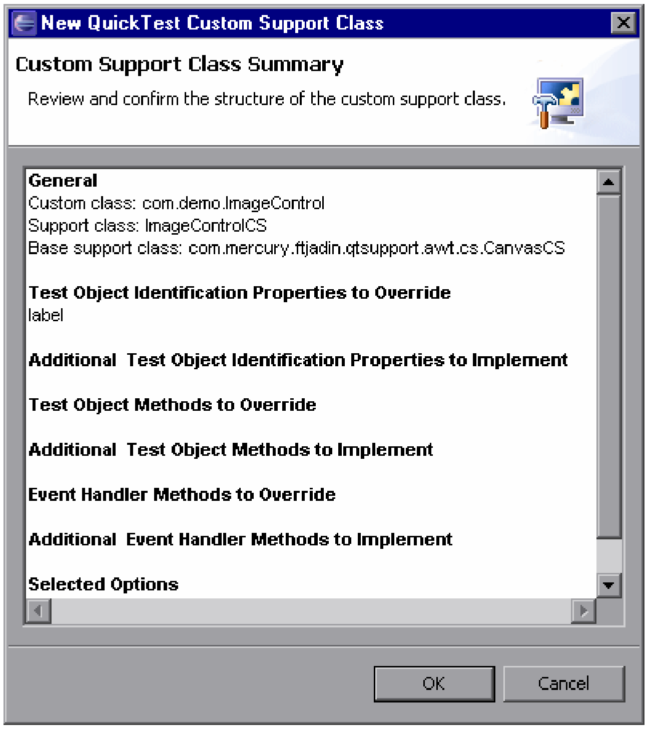
In the left pane, you can see the listeners implemented by CanvasCS. You do not have to override any of these for the ImageControlCS custom support class.

You are creating the ImageControlCS support class only to use it as a base support class for other support classes, not to support actual controls. Therefore, it is not necessary to support recording on ImageControl controls.

Click **Finish**. The Custom Control Support Class Summary screen opens.

7 View the custom control support class summary.

Review the planned content of the custom support class, and click **OK**.


New QuickTest Custom Support Class
Custom Support Class Summary
Review and confirm the structure of the custom support class.
OK
Cancel

General
Custom class: com.demo.ImageControl
Support class: ImageControlCS
Base support class: com.mercury.ftjadin.qtsupport.awt.cs.CanvasCS

Test Object Identification Properties to Override
label

Additional Test Object Identification Properties to Implement

Test Object Methods to Override

Additional Test Object Methods to Implement

Event Handler Methods to Override

Additional Event Handler Methods to Implement

Selected Options

The following changes are made in the ImageControlsSupport project:

- The **ImageControls.xml** file is modified to map the ImageControl custom class to its support class—ImageControlCS.
- A new QuickTest custom support class, ImageControlCS, is created in the **ImageControlCS.java** file in the **com.mercury.ftjadin.qtsupport.imagecontrols.cs** package. The file is opened and displayed in a tab in the right pane.

  The ImageControlCS class extends CanvasCS and contains only one method stub—**label_attr**.

The asterisk (*) next to the ImageControlCS file name (in the ImageControlCS tab) indicates that it has not been saved. The changes made by the wizard are codependent, and must be saved to prevent discrepancies. Choose **File** > **Save**, or click the **Save** button.

8 Implement the label_attr method in the ImageControlCS class.

➤ In Eclipse, in the **ImageControlCS.java** file, in the **label_attr** method stub, replace `return super.label_attr(obj);` with the following code, so that it returns the name of the image file used for the ImageControl (without the full path):

```
ImageControl ic = (ImageControl)arg0;
String res = ic.getImageString();
if(res == null || res.length() == 0)
  return null;
int last = res.lastIndexOf('/');
if(last == -1)
  return res;
return res.substring(last+1);
```

➤ Click the **Save** button, or choose **File** > **Save** to save your changes.

Modifying the ImageControls Toolkit Support Hierarchy

The hierarchy of the support classes must match the hierarchy of the custom classes. Now that the ImageControl class is mapped to the support class ImageControlCS, the support classes for the ImageControl descendants must extend ImageControlCS.

Both ImageButtonCS and ImageLabelCS inherit **label_attr** method. ImageLabelCS needs to override this method to fine-tune its support of the **label** property.

1 Modify the ImageButtonCS class to extend ImageControlCS.

➤ Open the **ImageButtonCS.java** file in the ImageControlsSupport project in Eclipse, and locate the **ImageButtonCS** class signature:

```
public class ImageButtonCS extends CanvasCS implements ActionListener
```

➤ Modify the signature so that ImageButtonCS extends ImageControlCS:

```
public class ImageButtonCS extends ImageControlCS implements
ActionListener
```

➤ Remove the **label_attr** method from the ImageButtonCS class.

➤ Save the **ImageButtonCS.java** file.

2 Modify the ImageLabelCS class to extend ImageControlCS.

➤ In the **ImageLabelCS.java** file, replace public class ImageLabelCS extends CanvasCS with public class ImageLabelCS extends ImageControlCS.

➤ Replace the following lines in the **label_attr** method in the ImageLabelCS class:

```
res = il.getImageString();
if(res == null || res.length() == 0)
  return null;
int last = res.lastIndexOf('/');
if(last == -1)
  return res;
return res.substring(last+1);
```

with:

```
return super.label_attr(obj);
```

➤ Save the changes.

Deploying and Testing the New ImageControls Toolkit Support

When you created the new ImageControlCS support class, the wizard modified the **ImageControls.xml** file to map the ImageControl class to the ImageControlCS support class. Therefore, you must redeploy the ImageControls toolkit support for your changes to take effect.

1 Deploy the ImageControls toolkit support to QuickTest.

In the Eclipse Package Explorer tab, select the **ImageControlsSupport** project.

Click the **Deploy Toolkit Support** button, or choose **QuickTest** > **Deploy Toolkit Support**. In the confirmation messages that open, click **Yes** and then **OK.**

2 Test the modified custom support.

Repeat the procedures in "Planning Support for the ImageButton Control" on page 158 and "Planning Support for the ImageLabel Control" on page 198, to re-run the SampleApp application and ensure that the support for ImageButton and ImageLabel is functioning properly.

Note: You did not change any test object configuration files, therefore you can use an open session of QuickTest.

The changes you made to the custom toolkit support set do not affect the way QuickTest recognizes and tests the ImageLabel and ImageButton controls. However, the support for the **label** identification property for both of these controls is now inherited from the ImageControlCS class. If additional custom classes that extend ImageControl are created, their **label** property will be similarly supported on QuickTest with no additional effort required.

You can find a ready-made example of the improved support for the ImageControls toolkit in the **<QuickTest Professional Java Add-in Extensibility SDK installation folder>\samples\ImageControlsSupportAdvanced** folder.

Lesson Summary

In this lesson you created support for the ImageLabel control, allowing QuickTest to recognize it as a static-text object and use its **label** property as **attached text** for adjacent controls.

You then created support for the ImageControl class to improve the flexibility of the toolkit support, and modified the hierarchy of the ImageControls toolkit support set accordingly.

➤ You learned how to create a support class for a custom static-text control, using the following identification property support methods:

class_attr

tag_attr

label_attr

value_attr

➤ You used the **Same as base support class** option in the Test Object Class Selection screen, and learned about the effects of that selection.

Where Do You Go from Here?

For more information on the identification properties that you used in this lesson, see “Special Identification Property Support Methods” on page 52.

In the next lesson you learn how to create support for a custom control that needs to be mapped to a new test object class. You will define special identification properties and test object methods for the new test object class, and implement support for them.

9

Learning to Support a Complex Control

In this lesson you create support for the AllLights control within the Javaboutique toolkit. This is a complex control, with unique behavior, that requires a new test object class definition.

In the lesson "Learning to Support a Simple Control" on page 155, you learned to create support for a simple custom control. You are now familiar with the basics of Java Add-in Extensibility, therefore this lesson explains only the more advanced information.

Preparing for This Lesson

Before you extend QuickTest support for a custom control, you must have access to its class and an application that runs it.

In this section, you create an Eclipse project containing the Javaboutique custom toolkit classes. The AllLights class can run as an Applet, so there is no need for an additional application containing the custom control.

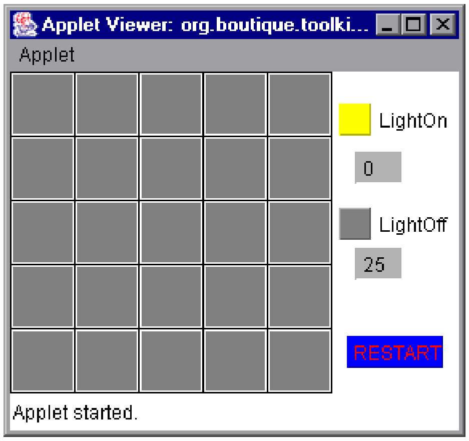
To create a new Java project with the Javaboutique sample in Eclipse:

1. Run Eclipse and choose **File** > **New** > **Project**. The New Project dialog box opens.
2. Select **Java Project** and click **Next**. The New Java Project dialog box opens.
3. Enter Javaboutique in the **Project name** box.
4. Select the **Create project from existing source** option.
5. Click the **Browse** button and browse to the **<QuickTest Professional Java Add-in Extensibility SDK installation folder>\samples\Javaboutique\src** folder. Click **OK** to return to the New Java Project dialog box.
6. Click **Finish**. A new Java project is created with the ImageControls sample source files. The new project, named Javaboutique, is displayed in the Package Explorer tab.

   Expand the **Javaboutique** project to view its content. The **Javaboutique\src** package folder contains the **org.boutique.toolkit** package. This package contains three custom controls: **AllLights**, **AwtCalc** and **ETextField**.

   In this lesson, you create the QuickTest Java Add-in Extensibility project for the Javaboutique custom toolkit and the support class for AllLights. (You can find the completed support for AllLights and for AwtCalc in the **<QuickTest Professional Java Add-in Extensibility SDK installation folder>\samples\JavaboutiqueSupport** folder.

In the Eclipse Package Explorer tab, right-click the **Allights.java** class in the **org.boutique.toolkit** package and choose **Run As > Java Applet**. The AllLights application opens:



Click different locations in the frame to become familiar with the AllLights behavior:

➤ Clicking in different parts of the grid area turns different lights on (or off), according to an internal set of rules, updating the **LightOn** and **LightOff** counters.

➤ Clicking the **RESTART** button turns off all of the lights. The **LightOn** and **LightOff** counters are updated accordingly.

➤ Clicking in other areas has no effect.

➤ The object of the game is to turn on all of the lights, at which point a congratulation message is displayed.

Planning Support for the AllLights Control

In this section, you analyze the current QuickTest support of the AllLights control, determine the answers to the questions in the "Understanding the Custom Class Support Planning Checklist" on page 78, and fill in the "Custom Class Support Planning Checklist" on page 233, accordingly.

The best way to do this is to run the application containing the custom control, and analyze it from a QuickTest perspective using the Object Spy, Keyword View, and Record option.

1 **Run the AllLights application and open QuickTest.**

If the AllLights application is already running, choose **Applet** > **Restart** from the application toolbar so the application looks like the image shown above. Otherwise, right-click **AllLights.Java** in the Eclipse Package Explorer tab, and choose **Run As** > **Java Applet** to run it.

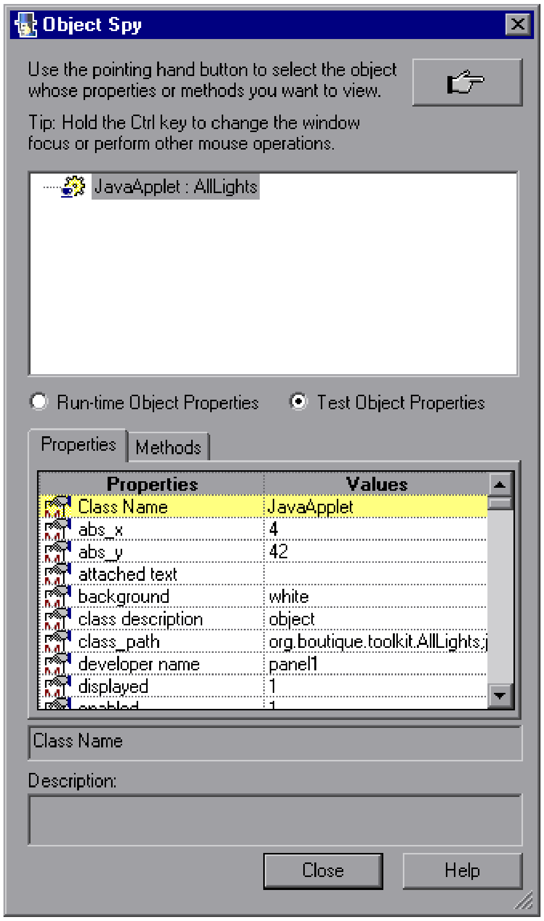
Open QuickTest and load the Java Add-in.

2 **Use the Object Spy to view the AllLights properties and methods.**

In QuickTest Choose **Tools** > **Object Spy** or click the **Object Spy** toolbar button to open the Object Spy dialog box. Click the **Properties** tab.

In the Object Spy dialog box, click the pointing hand, then click the AllLights application.

The AllLights control extends JavaApplet, for which QuickTest support is built in, therefore it recognizes the application as a **JavaApplet**, named **AllLights**. The icon shown is the standard JavaApplet class icon.



Object Spy

Use the pointing hand button to select the object whose properties or methods you want to view.

Tip: Hold the Ctrl key to change the window focus or perform other mouse operations.

JavaApplet : AllLights

Run-time Object Properties [✓ Test Object Properties]

Properties | Methods

Properties	Values
Class Name	JavaApplet
abs_x	4
abs_y	42
attached text	
background	white
class description	object
class_path	org.boutique.toolkit.AllLights;j
developer name	panel1
displayed	1
enabled	1

Class Name

Description:

Close the Object Spy.

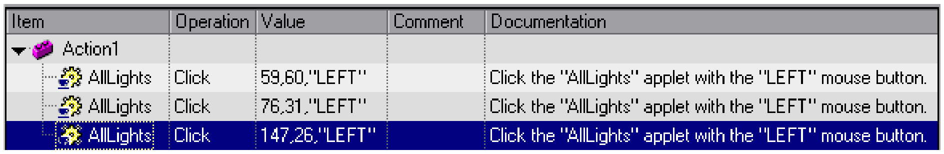
3 Record operations on the AllLights control.

In QuickTest choose **Automation > Record and Run Settings** to open the Record and Run Settings dialog box. In the Java tab, select **Record and run test on any open Java application**. If the Web Add-in is also loaded, click the Web tab and select **Record and run test on any open browser**. Click **OK**.

Record

Click the **Record** button or choose **Automation** > **Record**. Click on different locations in the AllLights application: the grid, the **RESTART** button, and one of the counters.

With each click, a new step is added to the test:



Item	Operation	Value	Comment	Documentation
Action1				
AllLights	Click	59,60,"LEFT"		Click the "AllLights" applet with the "LEFT" mouse button.
AllLights	Click	76,31,"LEFT"		Click the "AllLights" applet with the "LEFT" mouse button.
AllLights	Click	147,26,"LEFT"		Click the "AllLights" applet with the "LEFT" mouse button.

Stop

Click the **Stop** button or choose **Automation** > **Stop** to end the recording session.

The **Click** operation on the AllLights JavaApplet is a generic click, with arguments indicating the low-level recording details (x and y coordinates and the mouse button that performed the click).

4 Determine the custom toolkit to which the AllLights control belongs.

When you extend QuickTest support for a control you always do so in the context of a toolkit. For the purpose of this tutorial, three classes that extend AWT are grouped to form the custom toolkit named Javaboutique: AllLights, AwtCalc, and ETextField.

In this lesson you create support for the Javaboutique toolkit, initially supporting only the AllLights class.

5 Complete the custom class support planning checklist.

This section describes the required support for the AllLights control, and then summarizes the information in the support planning checklist.

➤ Deciding which custom class to support:

The AllLights custom class extends the Applet class, supported on QuickTest by AppletCS.

You want QuickTest to treat the AllLights as a special kind of Applet. You want it to support the special operations it performs, and to recognize its properties. Therefore it makes sense to create Extensibility support for this control.

- Mapping a test object class to the custom control:

  The JavaApplet test object class provides basic support for the AllLights control, but does not support all of the necessary identification properties and test object methods. Therefore you create a new test object class extending JavaApplet, named AllLights and map it to the AllLights custom control.

- Deciding the details for the new test object class:

  The new test object class is represented by the icon file: **<QuickTest Professional Java Add-in Extensibility SDK Installation folder>\samples\Javaboutique\AllLights_icon.ico**

  The identification properties used by default to uniquely define the test object (label, class, and index) are sufficient.

  The default test object method is ClickLight.

  The new identification properties to support are: OnCount, OnList, and GameOver. They should all be selected by default in the QuickTest Checkpoint Properties dialog box.

- AllLights controls are top-level objects, but not wrappers.
- The name of the test object itself should be **Lights**.
- The custom support should include support for the following identification properties:
  - **OnCount.** Specifies the number of lights that are on, at the given moment.
  - **OnList.** Lists the location of the lights that are on, at the given moment. The lights are numbered 0 through 24, starting from the upper left corner and numbering row by row. The list contains the numbers of the lights that are on, each preceded by a space.
  - **GameOver.** A **Yes** or **No** string, indicating whether all lights are on or not.
- The custom support should include support for the following test object methods:

  - **ClickLight.** Simulates clicking a specific light in the grid. This method requires two arguments, **Row** and **Column**, specifying the location of the light to click.
  - **Restart.** Simulates clicking the Restart button.

- Support for recording:

  Override low-level mouse event recording.

  Listen for mouse events. Based on the location of the click, send a record message to record ClickLight or Restart operations.

  On the next page you can see the checklist, completed based on the information above.

Custom Class Support Planning Checklist

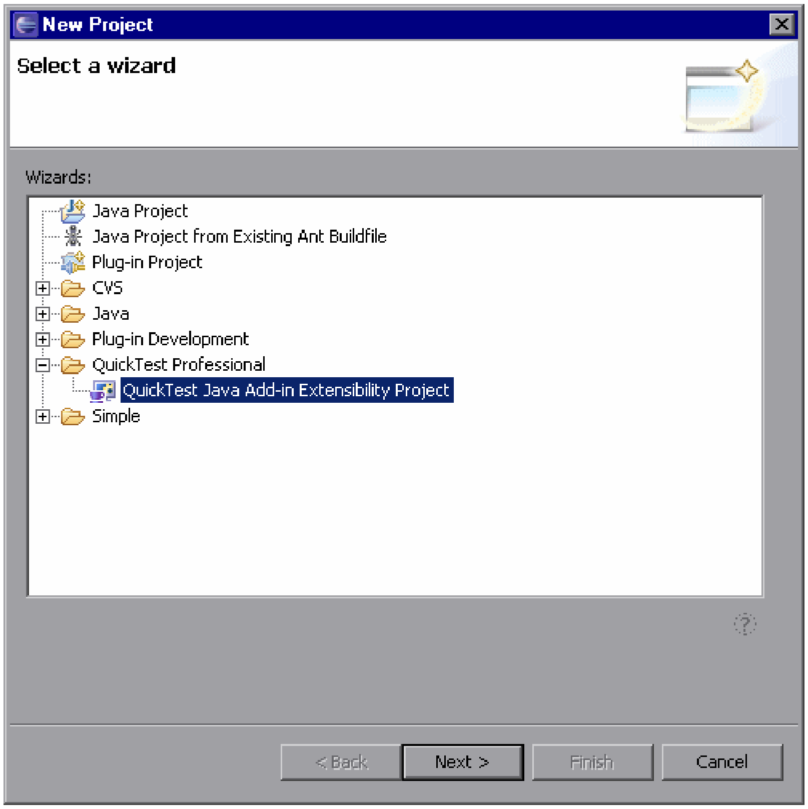
☑	**Custom Class Support Planning Checklist**
☐	Does the custom class have a superclass for which QuickTest custom support is not yet available? **~~Yes~~ /No**
☐	If so, should I first extend support for a control higher in the hierarchy? **N/A**
☐	Do I have an application that runs the custom control on a computer with QuickTest installed? **Yes /~~No~~**
☐	The sources for this custom control class are located in: an Eclipse project called Javaboutique
☐	Which existing Java test object matches the custom control? None
☐	If none, create a new Java test object class named: AllLights • New test object class extends: (Default—JavaObject) JavaApplet • Icon file location (optional): <QuickTest Professional Java Add-in Extensibility SDK Installation folder>\samples\Javaboutique\AllLights_icon.ico • Identification property for description: label • Default test object method: ClickLight
☐	Is the custom control a top-level object? **Yes /~~No~~**
☐	Is the custom control a wrapper? **~~Yes~~ /No**
☐	Specify the basis for naming the test object: Use the name: "Lights"
☐	List the identification properties to support, and mark default checkpoint properties: OnCount, OnList, GameOver (all selected by default in checkpoints)
☐	List the test object methods to support (include arguments and return values if required): ClickLight (Variant Row, Variant Column) Restart (no arguments)
☐	Provide support for recording? (AWT-based only) **Yes /~~No~~**
☐	If so, list the events that should trigger recording: MouseEvents

Creating the QuickTest Java Add-in Extensibility Project

In this section you create a new project for the Javaboutique toolkit support. To do this, you use one of the wizards provided by the QuickTest Java Add-in Extensibility plug-in in Eclipse.

1 Open the New QuickTest Java Add-in Extensibility Project wizard.

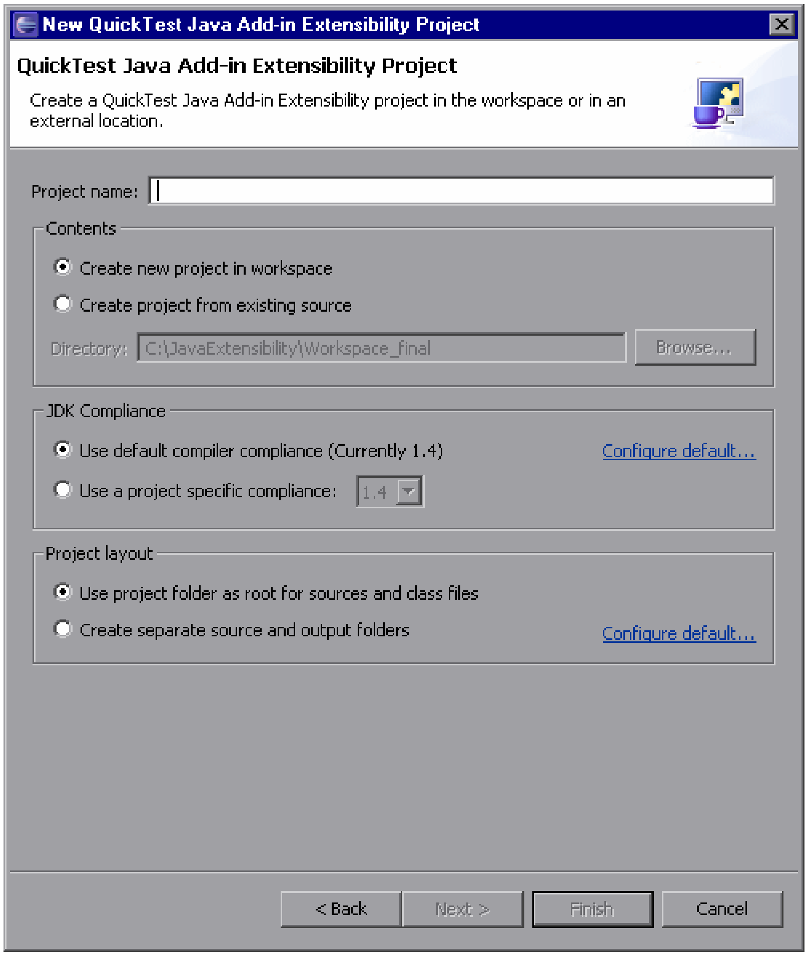
In Eclipse, choose **File** > **New** > **Project**. The New Project dialog box opens. Expand the **QuickTest Professional** folder and select **QuickTest Java Add-in Extensibility Project**.



Click **Next**. The QuickTest Java Add-in Extensibility Project screen opens.

2 Enter the QuickTest Java Add-in Extensibility project details.

In the **Project name** box, enter JavaboutiqueSupport. Select the **Create separate source and output folders** option. For more information on this dialog box, refer to the *Eclipse Help*.



New QuickTest Java Add-in Extensibility Project

QuickTest Java Add-in Extensibility Project

Create a QuickTest Java Add-in Extensibility project in the workspace or in an external location.

Project name:

Contents

(●) Create new project in workspace

( ) Create project from existing source

Directory: C:\JavaExtensibility\Workspace_final

JDK Compliance

(●) Use default compiler compliance (Currently 1.4)

( ) Use a project specific compliance: 1.4

Project layout

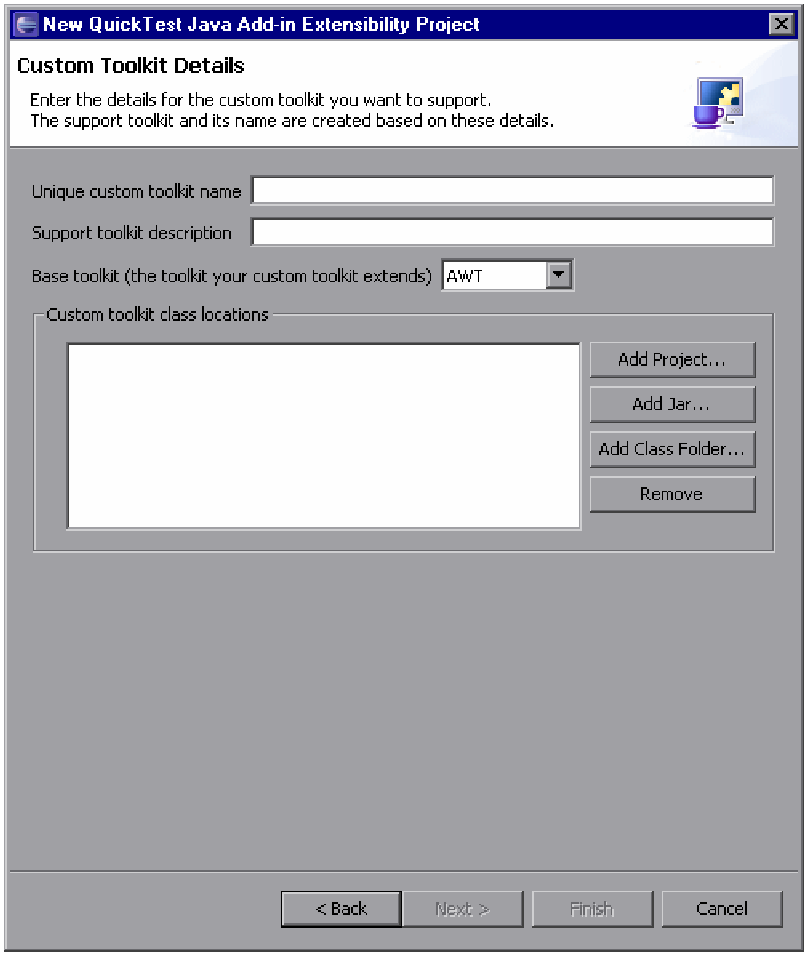
(●) Use project folder as root for sources and class files

( ) Create separate source and output folders

Click **Next**. The Custom Toolkit Details screen opens.

3 Enter the custom toolkit details.

In this screen, you provide the details of the Javaboutique toolkit so that the wizard can generate a corresponding custom toolkit support set.



Enter the details for the custom toolkit you want to support.
The support toolkit and its name are created based on these details.

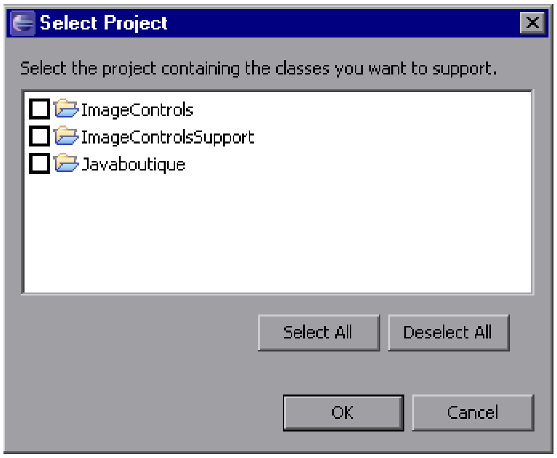
Unique custom toolkit name
Support toolkit description
Base toolkit (the toolkit your custom toolkit extends) AWT
Custom toolkit class locations

In the **Unique custom toolkit name** enter Javaboutique.

In the **Support toolkit description** box enter: Javaboutique toolkit support.

The AllLights custom class extends an AWT component, so keep the default selection AWT as the **Base toolkit**.

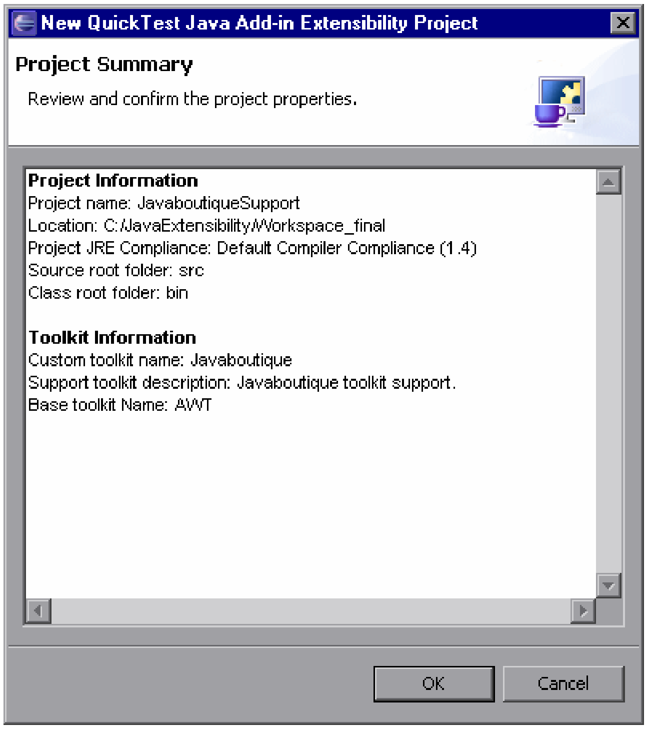
In the **Custom toolkit class locations** area, click **Add project** to select the Eclipse Java project containing the custom classes for the Javaboutique toolkit. The Select Project dialog box opens and displays the projects in the current Eclipse workspace.



Select the **Javaboutique** check box. Click **OK**. The Javaboutique project is added in the **Custom toolkit class locations** box. Click **Finish**. The Project Summary screen opens.

4 View the Project Summary wizard screen.

Review the details of the project and click **OK**.



Project Summary

Review and confirm the project properties.

Project Information
Project name: JavaboutiqueSupport
Location: C:/JavaExtensibility/Workspace_final
Project JRE Compliance: Default Compiler Compliance (1.4)
Source root folder: src
Class root folder: bin

Toolkit Information
Custom toolkit name: Javaboutique
Support toolkit description: Javaboutique toolkit support.
Base toolkit Name: AWT

The New QuickTest Java Add-in Extensibility project JavaboutiqueSupport is created, with the basic packages and files of the custom toolkit support set:

- The package **com.mercury.ftjadin.qtsupport.javaboutique**, containing the new toolkit support class file, **JavaboutiqueSupport.java**.
- The package **com.mercury.ftjadin.qtsupport.javaboutique.cs**
- The **Configuration** folder, containing the **TestObjects** folder and the new toolkit configuration file: **Javaboutique.xml**

Note: If you have more than one JRE installed on your computer, make sure that the Javaboutique project and the JavaboutiqueSupport project are using the same JRE version. If they are not, modify the JRE for one of the projects so that they use the same version.

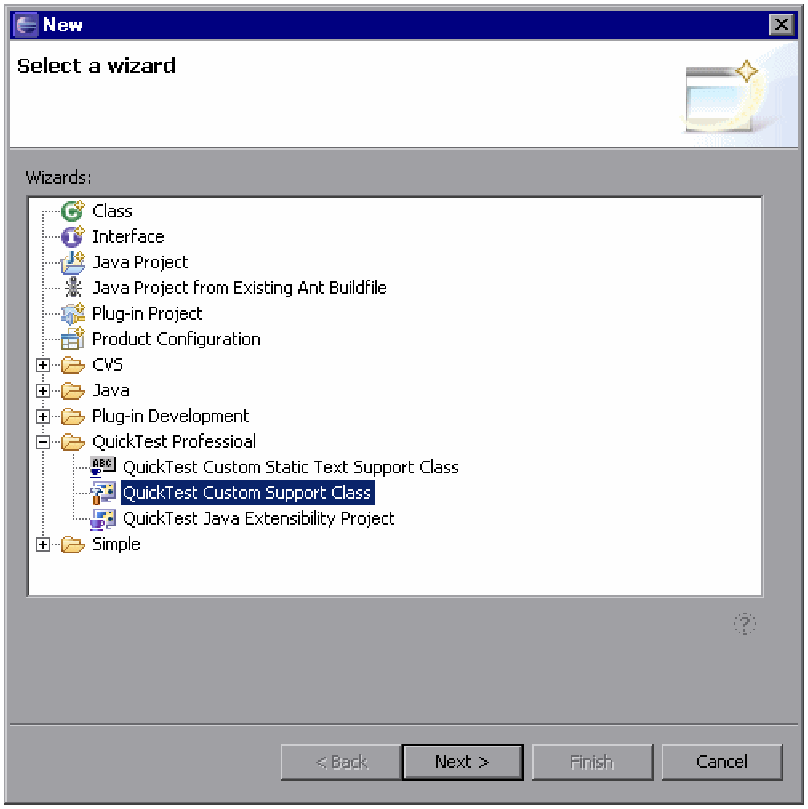
Creating the New QuickTest Custom Support Class

In this section you create a custom support class for the AllLights control, as part of the Javaboutique toolkit support. To do this, you use one of the wizards provided by the QuickTest Java Add-in Extensibility plug-in in Eclipse. The details you specify in each wizard screen reflect the decisions you made when planning the custom support. In the subsequent sections you implement the methods that the wizard creates in this class.

Note: The following sections describe only the options in the wizard screens that are relevant to this lesson. For a complete description of all options available in the wizard screens, see Chapter 5, "Using the QuickTest Java Add-in Extensibility Eclipse Plug-In."

1 Open the New QuickTest Custom Support Class wizard.

In the Eclipse Package Explorer tab, select the new QuickTest Java Add-in Extensibility project, **JavaboutiqueSupport**. Choose **File** > **New** > **Other**. The New dialog box opens.

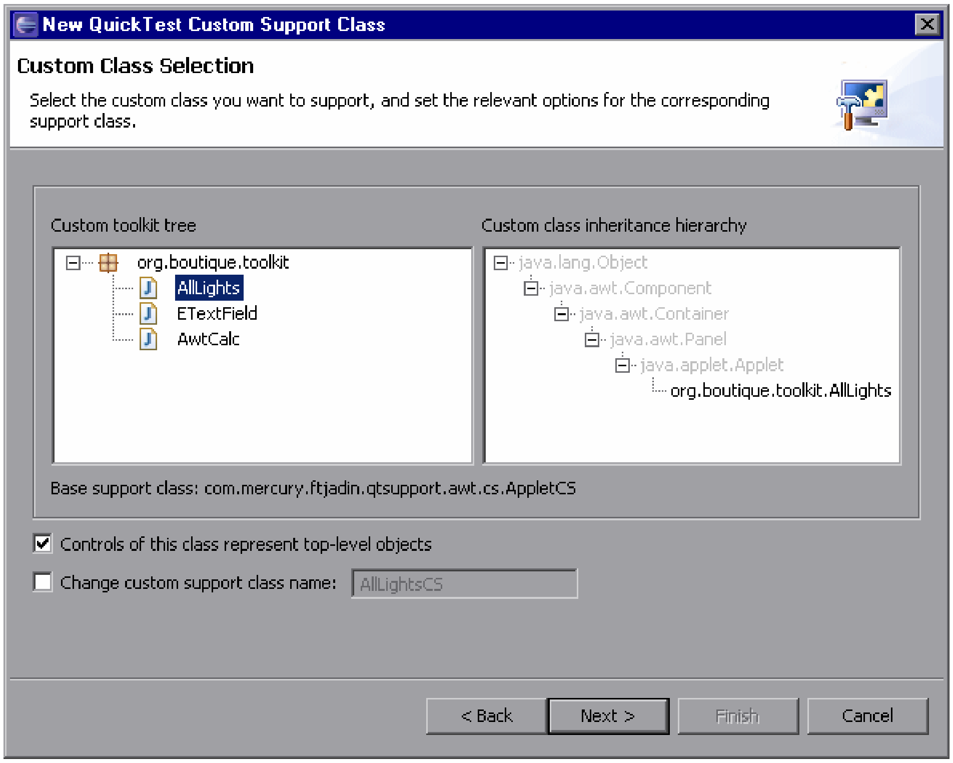


Expand the **QuickTest Professional** folder and select **QuickTest Custom Support Class**.

Click **Next**. The Custom Class Selection screen opens.

2 Select the custom class to support, and set the options for the support class.

Select the **AllLights** class in the **org.boutique.toolkit** package.


New QuickTest Custom Support Class

Custom Class Selection

Select the custom class you want to support, and set the relevant options for the corresponding support class.

Custom toolkit tree

```
org.boutique.toolkit
    AllLights
    ETextField
    AwtCalc
```

Custom class inheritance hierarchy

```
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Panel
        java.applet.Applet
          org.boutique.toolkit.AllLights
```

Base support class: com.mercury.ftjadin.qtsupport.awt.cs.AppletCS

☑ Controls of this class represent top-level objects

☐ Change custom support class name: AllLightsCS

The AllLights custom class extends java.applet.Applet, which is supported on QuickTest. The AllLights support class therefore extends the **Base support class**: **com.mercury.ftjadin.qtsupport.awt.cs.AppletCS**. As a result, the **Controls of this class represent top-level objects** check box is selected by default.

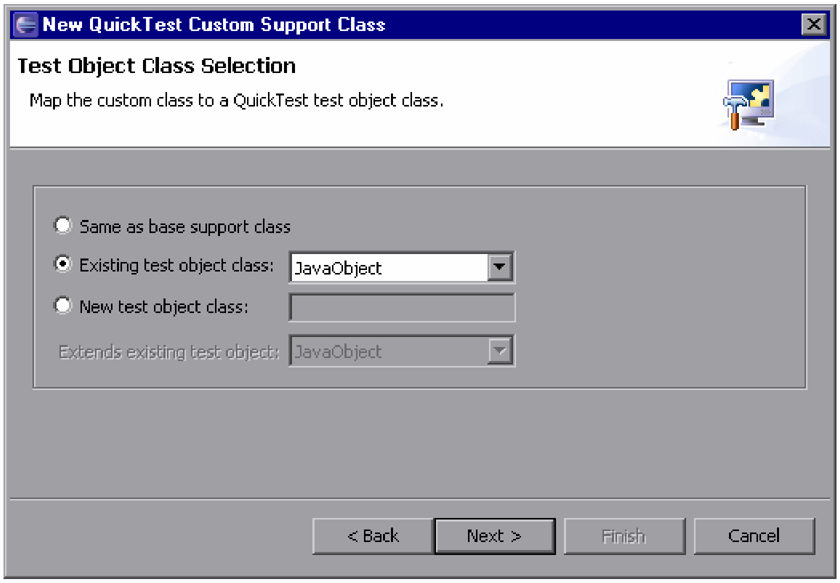
Leave this check box selected, because you want QuickTest to recognize the AllLights controls as the highest Java test objects in the test object hierarchy.

Keep the default custom support class name: AllLightsCS.

Click **Next**. The Test Object Selection screen opens.

3 Select a test object class to represent the custom control.

In this screen, you map the custom control to a test object class. In QuickTest tests, the custom class controls are represented by test objects of this test object class.


New QuickTest Custom Support Class

Test Object Class Selection

Map the custom class to a QuickTest test object class.

( ) Same as base support class
(●) Existing test object class: JavaObject
( ) New test object class:
Extends existing test object: JavaObject

In step 5 of "Planning Support for the AllLights Control", described on page 230, you decided to map the AllLights custom control to a new test object class, AllLights, that extends JavaApplet.

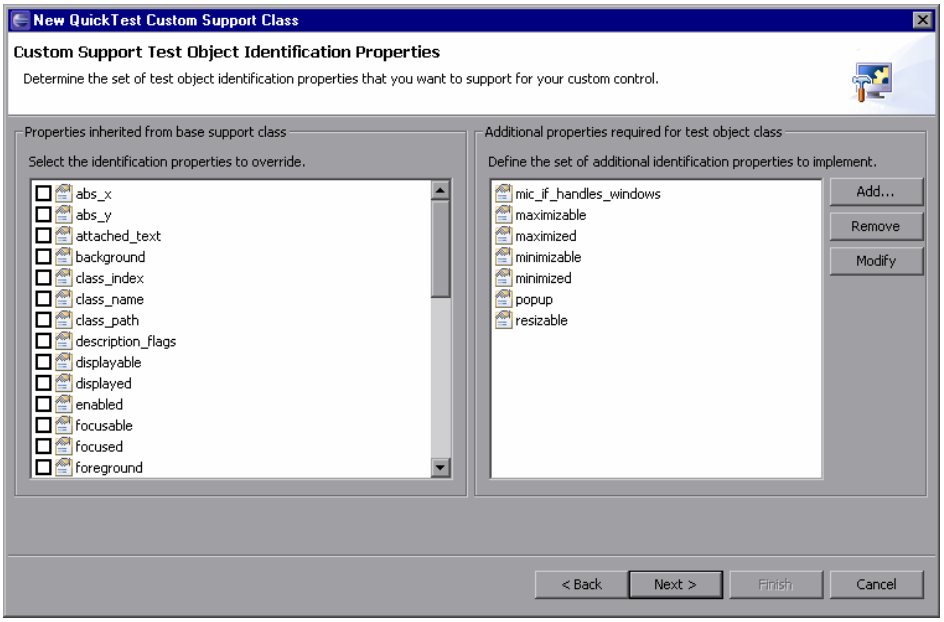
Select the **New test object class** option and enter AllLights as the name for the test object class.

In the **Extends existing test object** list, select **JavaApplet**. This list contains all of the Java objects that QuickTest currently supports. If you define new test objects for custom support, they are included in the list as well.

Click **Next**. The Custom Support Test Object Identification Properties screen opens.

4 Determine the set of test object identification properties to implement in AllLightsCS.

This screen displays the identification properties supported by the base support class you are extending, as well as additional properties that are defined in the test object class you selected, but are not yet supported. It enables you to select properties whose support you want to implement or override with new functionality and to add new properties.



New QuickTest Custom Support Class

Custom Support Test Object Identification Properties

Determine the set of test object identification properties that you want to support for your custom control.

Properties inherited from base support class

Select the identification properties to override.

- ☐ abs_x
- ☐ abs_y
- ☐ attached_text
- ☐ background
- ☐ class_index
- ☐ class_name
- ☐ class_path
- ☐ description_flags
- ☐ displayable
- ☐ displayed
- ☐ enabled
- ☐ focusable
- ☐ focused
- ☐ foreground

Additional properties required for test object class

Define the set of additional identification properties to implement.

- mic_if_handles_windows
- maximizable
- maximized
- minimizable
- minimized
- popup
- resizable

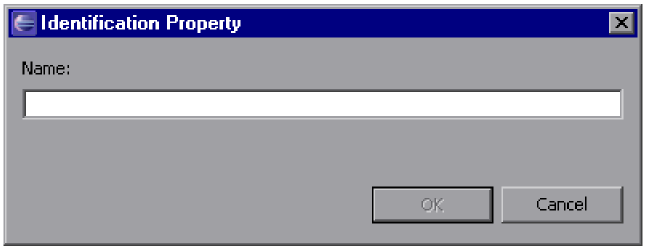
The left pane shows all of the identification properties whose support is implemented by AppletCS, and therefore inherited by the new AllLightsCS support class. For most of the properties in this list, the default implementation is sufficient. Scroll down and select the **label** check box. After you finish generating the support files using the wizard, you will override the existing support for this property with a custom implementation that matches the needs of your custom control.

The identification properties displayed in the right pane are JavaApplet properties that are not supported by AppletCS. These properties are not required for the AllLights support. Select them, click **Remove**, and then click **Yes** to confirm.

In step 5 of "Planning Support for the AllLights Control", described on page 230, you decided to support new identification properties on AllLights test objects. You now add these properties to the list of additional properties required for the test object class.

Note: The identification properties are added to the test object class definition. This means that the new properties appear in the list of identification properties in QuickTest for all test objects of this class. This is the reason you are creating the new AllLights test object class.

➤ Click **Add** in the **Additional properties required for test object class** pane. The Identification Property dialog box opens.



In the **Name** box, enter OnCount. Click **OK** to add the new Identification Property to the list.

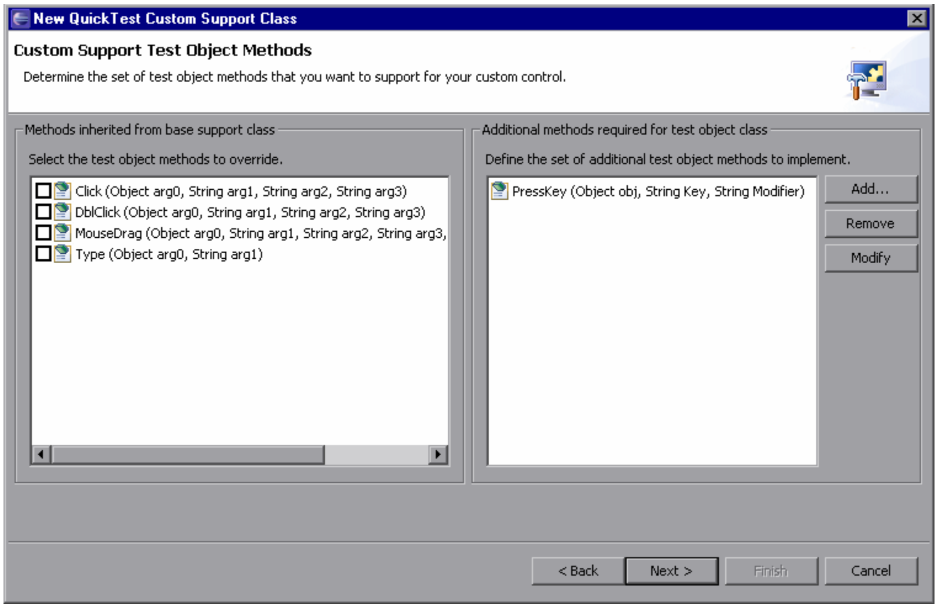
➤ Repeat this procedure to add the properties OnList and GameOver.

After you finish generating the support files using the wizard, you will implement support for these properties.

➤ Click **Next**. The Custom Support Test Object Methods screen opens.

5 Determine the set of test object methods to implement in AllLightsCS.

This screen displays the test object methods defined in the test object class you selected. It enables you to select methods whose support you want to implement or override with new functionality, and to add new methods.


New QuickTest Custom Support Class

Custom Support Test Object Methods

Determine the set of test object methods that you want to support for your custom control.

Methods inherited from base support class

Select the test object methods to override.

- [ ] Click (Object arg0, String arg1, String arg2, String arg3)
- [ ] DblClick (Object arg0, String arg1, String arg2, String arg3)
- [ ] MouseDrag (Object arg0, String arg1, String arg2, String arg3,
- [ ] Type (Object arg0, String arg1)

Additional methods required for test object class

Define the set of additional test object methods to implement.

PressKey (Object obj, String Key, String Modifier)

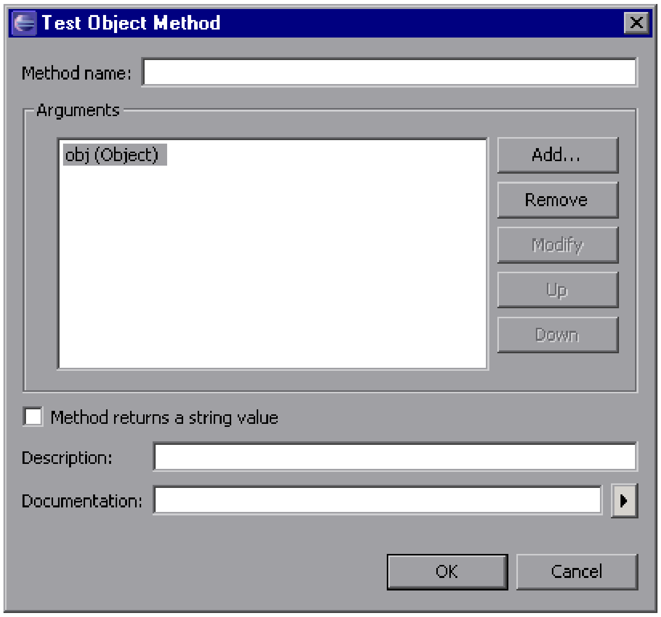
The left pane shows all of the test object methods (defined in the test object class you selected) whose support is implemented by AppletCS, and therefore inherited by AllLightsCS. There is no need to select any methods to override.

The **PressKey** method appears in the right pane, because it is a JavaApplet test object method, but it is not supported by AppletCS. This method is not required for AllLights support. Select it, click **Remove**, and then click **Yes** to confirm.

In step 5 of "Planning Support for the AllLights Control", described on page 230, you decided to support new test object methods on AllLights test objects. You now add these methods to the list of additional test object methods required for the test object class.

Note: The test object methods are added to the existing test object class. This means that the new methods appear in QuickTest for all test objects of this class, regardless of whether or not they are supported for these objects. In a QuickTest test, if you call a test object method for an object, and that method is not supported, a run-time error occurs. This is the reason you are creating the new AllLights test object class.

a Click **Add** in the **Additional test object methods required for test object class** pane. The Test Object Method dialog box opens.

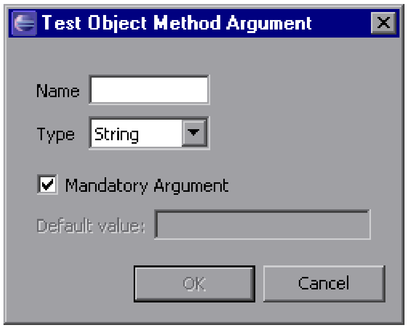


- In the **Method Name** box, enter: Restart. The Restart test object method does not require any arguments other than the mandatory **obj (Object)** that represents the custom control.
- Leave the **Method returns a string value** check box cleared. This method returns only a return code.
- In the **Description** box, enter: Clicks the RESTART button.

- In the **Documentation** box, enter: Click the RESTART button.
- Click **OK** to close the Test Object Method dialog box and add the Restart method to the list.

b Add another test object method by clicking **Add** once again. In the Test Object Method dialog box that opens, perform the following:

- In the **Method Name** box, enter: ClickLight.
- Add the Row and Column arguments to the ClickLight method:
- In the **Arguments** area, click **Add**. The Test Object Method Argument dialog box opens.



Type: String
[✓] Mandatory Argument

  - In the **Name** box, enter: Row.
  - In the **Type** box, select **Variant**. (If you select **String**, then when you add steps in QuickTest tests with the ClickLight method, you have to enclose the row number argument in quotes.)
  - Leave the **Mandatory Argument** check box selected.
  - Click **OK** to close the Test Object Method Argument dialog box and add the Row argument to the list of arguments for the ClickLight test object method.
- Repeat this procedure to add the Column argument to the list.
- Leave the **Method returns a string value** check box cleared.
- In the **Description** box, enter: Clicks the specified light.

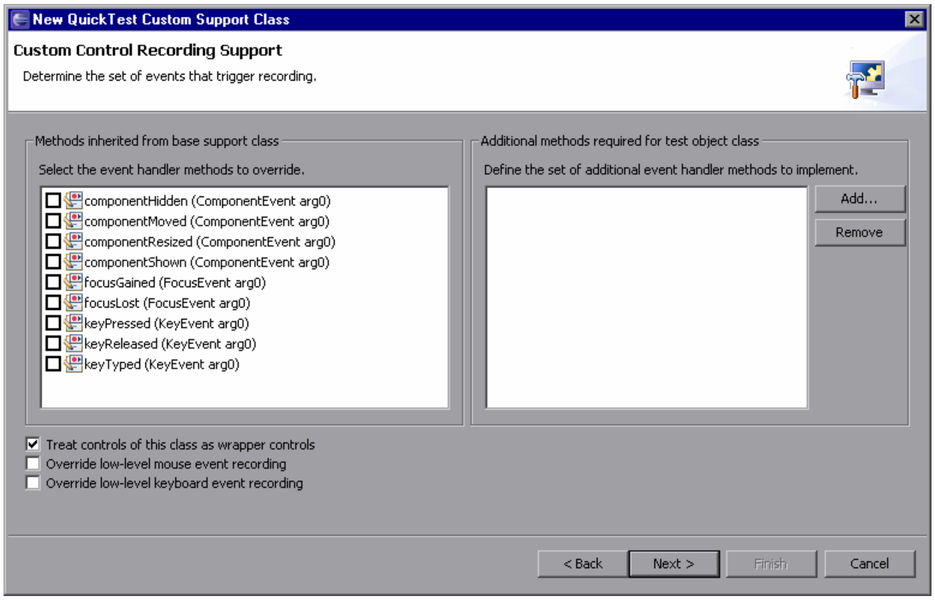
- In the **Documentation** box, enter: Click the light in row <Row> column <Column>. Enter the <Row> and <Column> arguments in the sentence by clicking ▸ and selecting the relevant argument.
- Click **OK** to close the Test Object Method dialog box and add the ClickLight method to the list.

After you finish generating the support files using the wizard, you will implement support for the methods you added.

c Click **Next**. The Custom Control Recording Support wizard screen opens.

6 Determine the set of events for which to listen, in order to support recording on the AllLights control.

This screen displays the event listeners supported by the support class you selected to extend. It enables you to select listeners whose implementation you want to override with new functionality and to add new event listeners to implement.



New QuickTest Custom Support Class

Custom Control Recording Support

Determine the set of events that trigger recording.

Methods inherited from base support class

Select the event handler methods to override.

- ☐ componentHidden (ComponentEvent arg0)
- ☐ componentMoved (ComponentEvent arg0)
- ☐ componentResized (ComponentEvent arg0)
- ☐ componentShown (ComponentEvent arg0)
- ☐ focusGained (FocusEvent arg0)
- ☐ focusLost (FocusEvent arg0)
- ☐ keyPressed (KeyEvent arg0)
- ☐ keyReleased (KeyEvent arg0)
- ☐ keyTyped (KeyEvent arg0)

Additional methods required for test object class

Define the set of additional event handler methods to implement.

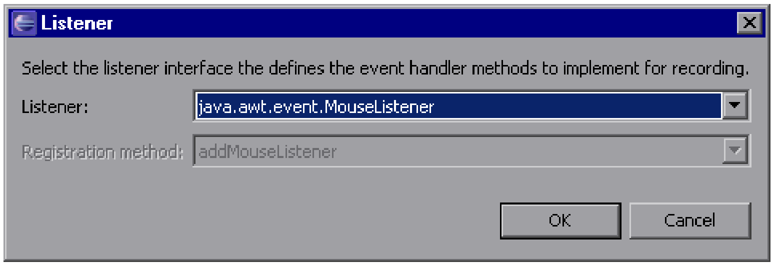
- ☑ Treat controls of this class as wrapper controls
- ☐ Override low-level mouse event recording
- ☐ Override low-level keyboard event recording

In the left pane, you can see the listeners implemented by AppletCS. You do not have to override any of these for the AllLightsCS custom support class.

In the right pane, you specify the listeners you want to add for AllLightsCS. Each listener you select implies a set of event handler methods you add to the custom support class.

Click **Add** to add the **MouseListener**.

The Listener dialog box opens.



Select the listener interface the defines the event handler methods to implement for recording.

Listener: java.awt.event.MouseListener

Registration method: addMouseListener

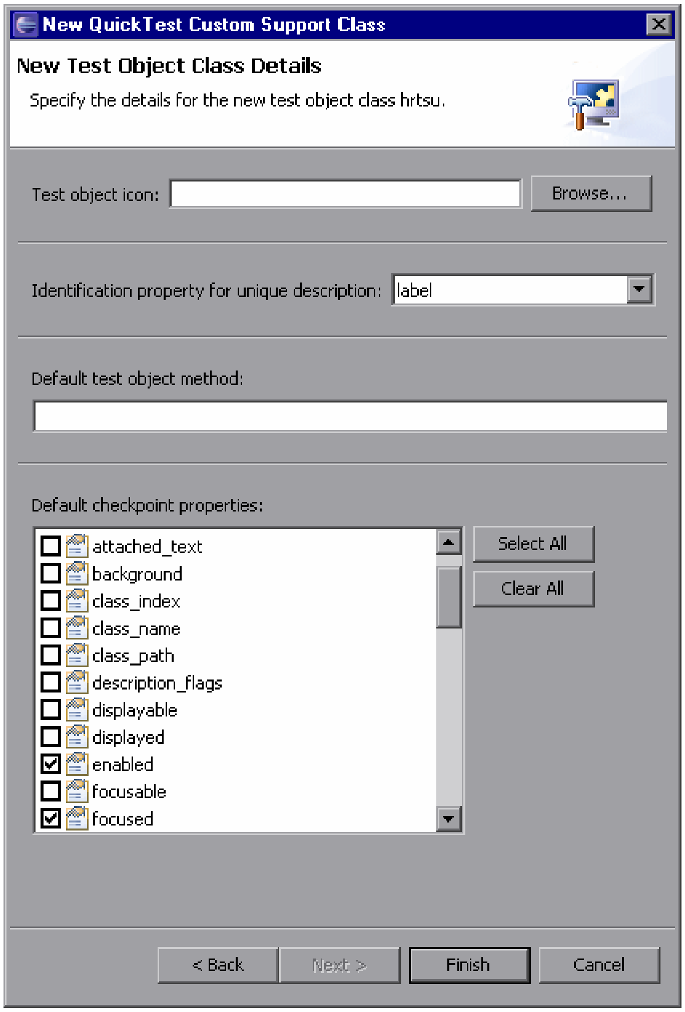
Select **java.awt.event.MouseListener** from the **Listener** list and click **OK**. The Listener dialog box closes and the **MouseListener**, and all of the event handler methods it includes, are added to the list in the right pane of the wizard screen.

On the Custom Control Recording Support screen:

- Clear the **Treat controls of this class as wrapper controls** check box. It is selected, by default, because the AllLights class extends **java.awt.container**.
- Select the **Override low-level mouse event recording** check box to prevent low-level events (coordinate-based operations) from being recorded instead of the events you want to record.
- Click **Next**. The New Test Object Class Details screen opens.

7 Define the details for the new test object class AllLights.

In this screen you define the details of the new test object class you are creating to map to the custom control.


New QuickTest Custom Support Class

New Test Object Class Details

Specify the details for the new test object class hrtsu.

Test object icon:

Identification property for unique description: label

Default test object method:

Default checkpoint properties:

- ☐ attached_text
- ☐ background
- ☐ class_index
- ☐ class_name
- ☐ class_path
- ☐ description_flags
- ☐ displayable
- ☐ displayed
- ☑ enabled
- ☐ focusable
- ☑ focused

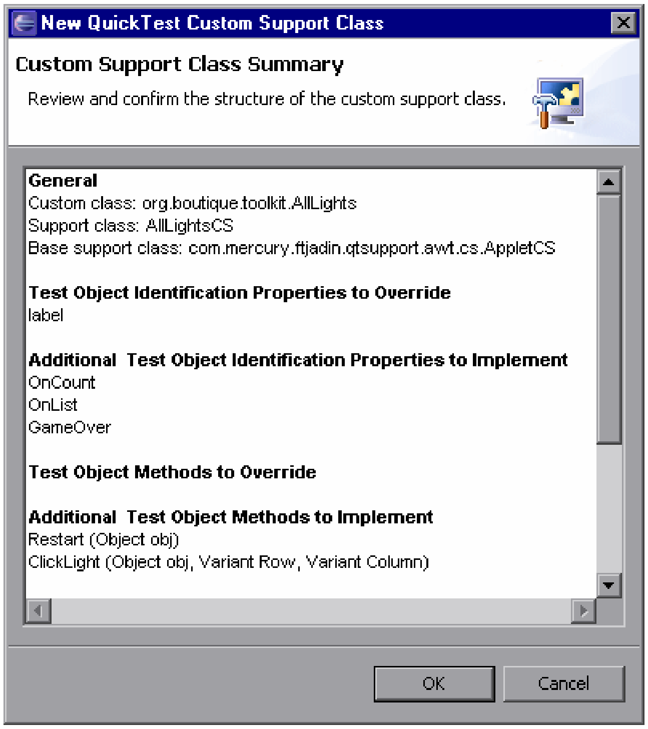
Perform the following:

➤ For the **Test object icon**, click **Browse**, locate the <**QuickTest Professional Java Add-in Extensibility SDK Installation folder**>**samples****Javaboutique** folder, and select the **AllLights_icon.ico** file.

- In the **Identification property for unique description box**, leave the selected **label** property.
- In the **Default test object method** list, select **ClickLight**.
- In the **Default checkpoint properties** box, leave the selected properties and select also the **GameOver**, **OnCount**, and **OnList** check boxes.
- Click **Finish**. The Custom Control Support Class Summary screen opens.

8 View the custom control support class summary.

Review the planned content of the custom support class, and click **OK**.



New QuickTest Custom Support Class

Custom Support Class Summary

Review and confirm the structure of the custom support class.

General
Custom class: org.boutique.toolkit.AllLights
Support class: AllLightsCS
Base support class: com.mercury.ftjadin.qtsupport.awt.cs.AppletCS

Test Object Identification Properties to Override
label

Additional Test Object Identification Properties to Implement
OnCount
OnList
GameOver

Test Object Methods to Override

Additional Test Object Methods to Implement
Restart (Object obj)
ClickLight (Object obj, Variant Row, Variant Column)

The following changes are made in the JavaboutiqueSupport project:

- A new QuickTest custom support class, AllLightsCS, is created in the **com.mercury.ftjadin.qtsupport.Javaboutique.cs** package. The file is opened and displayed in a tab in the right pane.

- A new **JavaboutiqueTestObjects.xml** file is created in the **Configuration\TestObjects** folder.
- The **Javaboutique.xml** file is modified. An element is added to the file, mapping the **AllLights** custom class to the **AllLightCS** support class. For information on the structure of this file, refer to the *QuickTest Java Add-in Extensibility Toolkit Configuration Schema Help* (**Help** > **QuickTest Professional Help** > **Java Add-in Extensibility Developer's Guide** > **QuickTest Java Add-in Extensibility Toolkit Configuration Schema**).

For a detailed explanation of the AllLightsCS class and the **JavaboutiqueTestObjects.xml** file, see "Understanding the New Custom Support Files," below.

The asterisk (*****) next to the **AllLightsCS** file name (in the AllLightsCS tab) indicates that it has not been saved. The changes made by the wizard are codependent, and must be saved to prevent discrepancies. Choose **File** > **Save**, or click the **Save** button.

Understanding the New Custom Support Files

When you completed the process of the New QuickTest Custom Support Class, the wizard registered the new class in the toolkit configuration file, and created the following files:

- **AllLightsCS.java.** This file contains the new AllLightsCS support class.
- **JavaboutiqueTestObject.xml.** This file contains the new test object classes defined for the Javaboutique toolkit support. At this point, there is only one such test object class: AllLights.

The following sections explain the content that the wizards created in these files.

Understanding the AllLightsCS Custom Support Class

After having performed the lesson "Learning to Support a Simple Control" on page 155, you are familiar with the basic elements that the wizard creates in a new custom support class. Examine the contents of the new AllLightsCS.java file, and locate the following methods and declarations:

- The declaration of the **AllLightsCS** support class, which indicates that it extends the **AppletCS** support class and implements the **MouseListener** interface.
- The declaration of the **DEBUG_ALLLIGHTSCS** flag, which can be used to control printing log messages.
- The **AllLightsCS** constructor method, which calls **addSimpleListener** to add **MouseListener** to the list of listeners that need to be registered on the **AllLights** control.
- The **to_class_attr** method, which returns the new test object class name: **AllLights**.
- A method stub for **label_attr** returning **super.label_attr**, which you can replace with a more specific label.
- Method stubs for the **oncount_attr**, **onlist_attr**, and **gameover_attr** methods, which you must implement to support the identification properties you added. Until you do so, these methods return null, because these are new methods that you added and they are not implemented in the superclasses that AllLightsCS extends.

Note: You can use capital letters in the identification property names that you provide in the wizard screen. These names are written as is in the test object configuration file. However, in the names of the support methods for these identification properties, upper case letter are replaced with lower case ones.

- Method stubs for the **Restart_replayMethod** and **ClickLight_replayMethod** methods, which you must implement to support the test object methods you added. Until you do so, these methods return the error code NOT_IMPLEMENTED.

- The **mouseRecordTarget** method, which returns **null** to override recording of low-level mouse events.
- Method stubs for the event handler methods defined by the **MouseListener** interface: **mouseClicked**, **mouseEntered**, **mouseExited**, **mousePressed**, and **mouseReleased**. These method stubs contain comments reminding you to implement them as necessary, calling **MicAPI.record** to send record messages to QuickTest.

The **is_window** method, returning **true**, was added to the **AllLightsCS** support class because you selected the **Controls of this class represent top-level objects** check box, on the Custom Class Selection screen. When learning the test object, QuickTest calls the **is_window** method to determine whether to look for a parent object or view this object as the highest Java object in the hierarchy.

Understanding the Javaboutique Test Object Configuration File

The wizard builds the test object class definition in the test object configuration file based on the details you specify.

Open the new JavaboutiqueTestObject.xml file and examine its contents. For information on the structure of this file, refer to the *QuickTest Test Object Schema Help* (**Help** > **QuickTest Professional Help** > **QuickTest Advanced References** > **QuickTest Test Object Schema**).

Locate the following elements in the test object configuration file:

- The test object class that the new test object class extends:

  ```
  BaseClassInfoName="JavaApplet"
  ```

- The name of the new test object class and its default test object method:

  ```
  DefaultOperationName="ClickLight" Name="AllLights">
  ```

- The location of the icon file:

  ```
  IconFile="<QuickTest Professional Java Add-in Extensibility SDK Installation
  folder>\samples\Javaboutique\AllLights_icon.ico"
  ```

- The definition of the new test object methods you added, and their description, documentation, and arguments (inside the <TypeInfo> element).
- The definition of the identification properties for this test object class (inside the <Properties> element). Note the identification properties marked ForVerification, ForDefaultVerification, and ForDescription.

Deploying and Testing the New Custom Toolkit Support

In this part of the lesson, you use the QuickTest **Deploy Toolkit Support** command in Eclipse to deploy the Javaboutique toolkit support to QuickTest. Currently only one control in this toolkit, the AllLights control, is supported. The toolkit support is not yet complete, but you can already test the support created up to this point.

1 Deploy the Javaboutique toolkit support to QuickTest.

In the Eclipse Package Explorer tab, select the **JavaboutiqueSupport** project.

Click the **Deploy Toolkit Support** button, or choose **QuickTest** > **Deploy Toolkit Support**. In the confirmation messages that open, click **Yes** and then **OK.**

The toolkit configuration file and the test object configuration file are copied to the relevant folders in your QuickTest installation folder. The custom support will be available the next time you start the custom application.

For more information on deploying custom toolkit support, see "Deploying and Running the Custom Toolkit Support" on page 66.

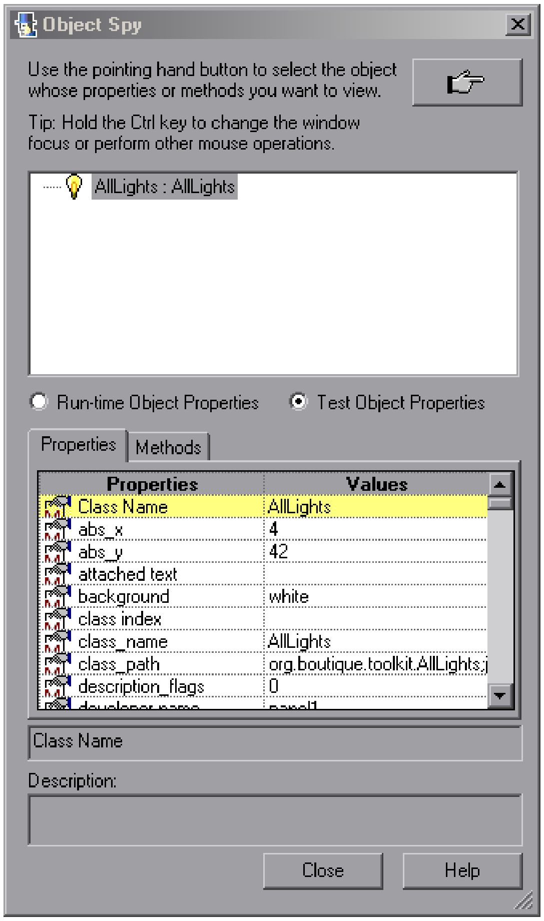
2 Test the new custom support.

Repeat steps 1, 2, and 3 in "Planning Support for the AllLights Control" on page 228, to open QuickTest, run the application, view the AllLights control with the QuickTest Object Spy, and try to record a Click operation on it.

Note: QuickTest reads test object configuration files when it opens. The Javaboutique toolkit support contains a new test object configuration file. Therefore, you must close QuickTest open it again.

QuickTest establishes its connection with an application when the application opens. Therefore, you must close the SampleApp application, and run it again.

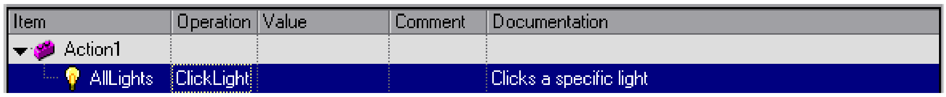
QuickTest recognizes the AllLights control as an AllLights test object (according to the **to_class_attr** method) named AllLights (the name of the custom class). The Object Spy displays the icon you specified in the wizard for this test object class.


Object Spy
Use the pointing hand button to select the object whose properties or methods you want to view.
Tip: Hold the Ctrl key to change the window focus or perform other mouse operations.
AllLights : AllLights
Run-time Object Properties
Test Object Properties
Properties
Methods
Class Name
Description:
Close
Help

Properties	Values
Class Name	AllLights
abs_x	4
abs_y	42
attached text	
background	white
class index	
class_name	AllLights
class_path	org.boutique.toolkit.AllLights;j
description_flags	0

Because you have overridden the low-level recording, but have not yet implemented the **mouseClicked (MouseEvent arg0)** event handler method, QuickTest currently does not record anything when you click in the application frame.

In QuickTest, add the AllLights object to the object repository, and create a test step with this object in the Keyword View.



Item	Operation	Value	Comment	Documentation
Action1				
AllLights	ClickLight			Clicks a specific light

The ClickLight test object method is selected, by default, as the step Operation. If you provide the required arguments for this method and run the test with this step, a run-time error occurs, because the **ClickLight_replayMethod** method returns .NOT_IMPLEMENTED.

Implementing Support for the AllLights Control

In this part of the lesson, you modify the AllLightsCS class to extend QuickTest support of the AllLights control, as per your plan ("Planning Support for the AllLights Control" on page 228).

Open the **AllLightsCS.java** file. In the **label_attr** method, replace the code: return super.label_attr(obj); with the code: return "Lights"; to change the name of the test object. Then perform the following procedures:

- ➤ Implementing Support for New Identification Properties (described on page 258)
- ➤ Implementing Support for New Test Object Methods (described on page 260)
- ➤ Implementing Support for Recording (described on page 261)
- ➤ Testing the Completed Support (described on page 263)

Implementing Support for New Identification Properties

In this section, you implement the methods that support the new identification properties you defined for the AllLights test object class.

Analyze the AllLights custom class to see the properties it supports. Determine which properties you can access from the new support class to provide the relevant identification properties to QuickTest.

Notice the public methods GetcounterOn, which allows you to check how many lights are on at a given time, and isSet, which tells you the status of a particular light.

1 Implement the oncount_attr method.

In the **oncount_attr** method, replace the code return null; with return String.valueOf(((AllLights)obj).GetcounterOn());

This implementation retrieves the counter from the AllLights custom class and returns it to QuickTest.

2 Implement the onlist_attr method.

In the **onlist_attr** method, delete the code return null; and implement the method as follows to scan all of the lights and create a list of all the lights that are on:

```
public String onlist_attr (Object obj) {
    AllLights lights = (AllLights) obj;
    StringBuffer buffer = new StringBuffer();
    for (int i=0; i<5; i++)
        for (int j=0;j<5;j++)
            if (lights.isSet(j,i)) {
                buffer.append (" ");
                buffer.append (i*5+j+1);
                }
    return buffer.toString();
}
```

3 Implement the gameover_attr method.

In the **gameover_attr** method, delete the code return null; and implement the method as follows to return Yes or No depending on whether or not all of the lights are on:

```
public String gameover_attr(Object obj) {
    if (((AllLights) obj).GetcounterOn() == 25)
        return "Yes";
    return "No";
}
```

Choose **File** > **Save** or click the **Save** button to save the **AllLightsCS.java** file.

Implementing Support for New Test Object Methods

In this section, you implement the methods that support the new test object methods you defined for the AllLights test object class.

Analyze the AllLights custom class methods to determine what actions the class performs when a user clicks the Restart button or a light in the grid. You want to simulate these actions when QuickTest runs the test object methods.

1 Implement the Restart_replayMethod method.

When a user clicks within the borders of the **RESTART** button, the AllLights custom class calls **init** and **update(lights.getGraphics())** to initialize and redraw the application. The **Restart_replayMethod** method needs to simulate this behavior by calling the same methods.

To do this, delete the code: return Retval.NOT_IMPLEMENTED; and implement the method as follows:

```
public Retval Restart_replayMethod (Object obj){
    AllLights lights = (AllLights) obj;
    lights.init();
    lights.update(lights.getGraphics());
    return Retval.OK;
}
```

2 Implement the ClickLight_replayMethod method.

The AllLights custom class performs the algorithm of turning lights on or off in response to a click, when it receives a mouseUp event. Therefore, when QuickTest runs the ClickLight_replayMethod, and you want to simulate a click on a specific light, you can simply send the AllLights object a mouseUp event with the appropriate coordinates.

In the method **ClickLight_replayMethod**, delete the code
`return Retval.NOT_IMPLEMENTED;` and implement the method as follows:

```
public Retval ClickLight_replayMethod(Object obj, String Row, String Column) {
    AllLights lights = (AllLights) obj;
    int col_num = Integer.valueOf(Column).intValue();
    int row_num = Integer.valueOf(Row).intValue();
    /* Row and column are 40 pixels wide*/
    Event event = new Event (lights, System.currentTimeMillis(),
        Event.MOUSE_UP, col_num *40, row_num *40, 0, 0);
    lights.mouseUp(event, col_num *40, row_num *40);
    return Retval.OK;
}
```

Note: To support this code, import **java.awt.Event** in **AllLightsCS.java**.

Choose **File** > **Save** or click the **Save** button to save the **AllLightsCS.java** file.

Implementing Support for Recording

Because you planned to support recording on the AllLights control, you suppressed low-level recording on this object, and registered to listen for mouse events on this control.

The only mouse event that you want to trigger recording on the AllLights control is a mouse click. Therefore, in this section, you implement only the **mouseClicked (MouseEvent arg0)** event handler method and leave the other mouse event handler methods empty.

Implement the **mouseClicked** method as follows and save the **AllLightsCS.java** file:

```
public void mouseClicked(MouseEvent arg0) {
    AllLights lights = (AllLights) arg0.getSource();
    int x = arg0.getX();
    int y = arg0.getY();

    try{
        if (!isInRecord())
            return;
        /* If click is within the Restart button borders*/
        if ((x > 210) && (x < 270) && (y > 165) && (y < 185)) {
            MicAPI.logLine(DEBUG_ALLLIGHTSCS, "Record Restart operation");
            MicAPI.record(lights, "Restart");
        }

        /* If click is within the borders of the grid - record a ClickLights*/
        if ((x >= 0) && (x < 200) && (y >= 0) && (y < 200)) {
            MicAPI.logLine(DEBUG_ALLLIGHTSCS, "Record ClickLight
                operation");
            MicAPI.record(lights, "ClickLight", new String[]
                {String.valueOf(x/40), String.valueOf(y/40)});
        }
    } catch (Throwable th) { MicAPI.logStackTrace(th);}
}
```

Note: To support this code, import **com.mercury.ftjadin.custom.MicAPI** in **AllLightsCS.java**.

In this event handler method, you call **MicAPI.record** in different ways. To record the Restart operation you provide only the object and the operation name. To record the ClickLight operation you provide additional arguments as well, specifying the coordinates of the clicked light.

The **isInRecord** method is called avoid carrying out any unnecessary operations if QuickTest is not currently recording.

The **MicAPI.logLine** method prints the message to the log file only when the DEBUG_ALLLIGHTSCS flag is on. For more information, see “Logging and Debugging the Custom Support Class” on page 71.

The **try ... catch** block prevents unnecessary activity if this code is reached when the Java application is running while QuickTest is idle. The **MicAPI.logStackTrace** method prints a stack trace to the same log file as other Java Add-in Extensibility log messages, enabling you to determine when this **mouseClicked** method was called inadvertently.

Testing the Completed Support

In this section you test the Javaboutique toolkit support you have just completed. You do this by analyzing its effect on how QuickTest views the AllLights control.

You do not have to deploy the toolkit support to QuickTest again in order to test it because you changed only Java class files and not configuration files. You can use an open QuickTest session, but you must close the AllLights application, and run it again, for the changes you made in the custom support to take effect.

1 Test the new custom support in the Object Spy.

Close the AllLights application and run it again.

Open QuickTest and load the Java Add-in.

Use the Object Spy to view the AllLights properties and methods. The AllLights test object is now named **Lights**.

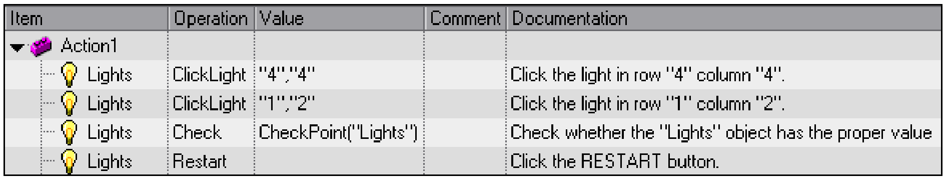
Close the Object Spy.

2 Create and run a QuickTest test on the AllLights control.

Add the AllLights control to the test object repository.

Create a test that clicks in two locations in the grid, checks that the game is not over, and clicks Restart.

The test you create looks something like this:



Item	Operation	Value	Comment	Documentation
Action1				
Lights	ClickLight	"4","4"		Click the light in row "4" column "4".
Lights	ClickLight	"1","2"		Click the light in row "1" column "2".
Lights	Check	CheckPoint("Lights")		Check whether the "Lights" object has the proper value
Lights	Restart			Click the RESTART button.

Note: The ClickLight_replayMethod, does not check the argument values to make sure they are valid. If you provide arguments that are out of bounds (column or row higher than 4) a run-time error will occur.

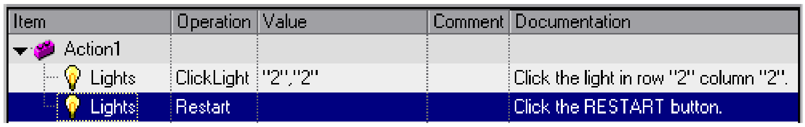
Run the test and see that it operates correctly (if you defined the checkpoint to check only that the game is not over—it succeeds).

3 Record operations on the AllLights control.

In QuickTest, create a new test and choose **Automation** > **Record and Run Settings** to open the Record and Run Settings dialog box. In the Java tab, select **Record and run test on any open Java application**. If the Web Add-in is also loaded, click the Web tab and select **Record and run test on any open browser**. Click **OK**.

Click the **Record** button or choose **Automation** > **Record**. Click on different locations in the AllLights application: the grid, the **RESTART** button, and one of the counters.

When you click in the grid, a ClickLight step is added to the test, with the relevant arguments. When you click the RESTART button, a Restart step is added. When you click anywhere else, no operation is recorded (because you disabled low-level mouse event recording). The recorded test looks something like this:



Item	Operation	Value	Comment	Documentation
Action1				
Lights	ClickLight	"2","2"		Click the light in row "2" column "2".
Lights	Restart			Click the RESTART button.

Click the **Stop** button or choose **Automation** > **Stop** to end the recording session.

The AllLights custom control is now fully supported, according to the specifications you decided on when planning your custom support.

Lesson Summary

In this lesson you created a new test object class, AllLights, defining its identification properties and test object methods. You created support for the AllLights control, allowing QuickTest to recognize it as an AllLights test object.

- ➤ You learned to understand the test object configuration file.
- ➤ You learned to support new identification properties and test object methods in the custom support class.
- ➤ You made use of the **is_window** utility method, and called the **MicAPI.record** method with additional parameters.

Where Do You Go from Here?

Now that you have performed the lessons in this tutorial, you are ready to apply the Java Add-in Extensibility concepts and the skills you learned to creating your own custom toolkit support.

For more information on the structure and content of a custom toolkit support set, see “Implementing Custom Toolkit Support” on page 27.

For more information on the structure and content 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**).

Index