HP UFT Java Add-in Extensibility

Software Version: 12.00
Windows ® operating systems

Developer Guide

Legal Notices

Warranty

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

The information contained herein is subject to change without notice.

Restricted Rights Legend

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

Copyright Notice

Trademark Notices

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

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

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

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

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

Documentation Updates

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

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

To check for recent updates or to verify that you are using the most recent edition of a document, go to:
https://softwaresupport.hp.com/group/softwaresupport/search-result.

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

Support

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

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

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

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

Most of the support areas require that you register as an HP Passport user and sign in. Many also require a support contract. To register for an HP Passport ID, go to: https://softwaresupport.hp.com and click **Register**.

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

HP Software Solutions & Integrations and Best Practices

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

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

Contents

Welcome to HP UFT Java Add-in Extensibility

HP UFT Java Add-in Extensibility is an SDK (Software Development Kit) 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 UFT Java Add-in.

This chapter includes:

About the UFT Java Add-in Extensibility SDK

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

- An API that enables you to extend the UFT 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.

- The Java Add-in Extensibility Help, which includes the following:

  - A developer guide, including a step-by-step tutorial in which you develop support for a sample custom control.

  - An API Reference.

- A Toolkit Configuration Schema Help.

- The UFT Test Object Schema Help.

The Help is available from **Start > All Programs > HP Software > HP Unified Functional Testing > Extensibility > Documentation**

- A printer-friendly Adobe portable document format (PDF) version of the developer guide (available from **Start > All Programs > HP Software > HP Unified Functional Testing > Extensibility > Documentation** and in the **<Unified Functional Testing installation>\help\Extensibility** folder).

- A set of sample applications and completed Java Add-in Extensibility projects that extend support for these applications.

Accessing UFT Java Add-in Extensibility in Windows 8 Operating Systems

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

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

- **Non-program files.** You can access documentation from the **Apps** screen.

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

About the UFT Java Add-in Extensibility Developer Guide

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

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

- *API References*

- *Toolkit Configuration Schema Help*

- *Test Object Schema Help*

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

- *HP Unified Functional Testing User Guide*

- The Java section of the *HP Unified Functional Testing Add-ins Guide*

- *HP UFT Object Model Reference for GUI Testing*

Note:

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

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

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

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

- **AND**, **OR**, **NEAR**, and **NOT** logical operators. Available from the arrow next to the search box.

- **Search previous results.** Available from the bottom of the **Search** tab.

- **Match similar words.** Available from the bottom of the **Search** tab.

- **Search titles only.** Available from the bottom of the **Search** tab.

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

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

Who Should Read This Guide

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

To use this guide, you should be familiar with:

- Major UFT features and functionality

- The UFT Object Model

- UFT Java Add-in

- Java programming

- XML (basic knowledge)

Additional Online Resources

The following additional online resources are available:

Resource	Description
Troubleshooting & Knowledge Base	The Troubleshooting page on the HP Software Support Web site where you can search the Self-solve knowledge base. The URL for this Web site is http://h20230.www2.hp.com/troubleshooting.jsp.

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

Part 1: Working with Java Add-in Extensibility

Chapter 1: Introducing UFT Java Add-in Extensibility

Welcome to UFT Java Add-in Extensibility.

UFT 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 UFT Java Add-in.

This chapter includes:

About UFT Java Add-in Extensibility

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

When UFT 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 UFT.

UFT can learn Java controls that are not supported out-of-the-box by the Java Add-in without using Extensibility. However, when UFT 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 UFT 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 UFT 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 UFT 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 UFT 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 UFT, 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. UFT determines which test object class to use according to this mapping.

When UFT 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. UFT 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**, UFT recognizes the control as a **JavaApplet** test object. If the custom class extends the **java.awt.Canvas**, UFT recognizes the control as a **JavaObject** test object.

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

Test Object Names

When UFT 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 UFT 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 support 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 UFT GUI tests is by recording user operations on the application. When you start a recording session, UFT listens for events that occur on objects in the application and registers corresponding test steps. Each Java object class defines which events UFT 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 UFT 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 UFT perspective to view the extent of this support and to decide which elements of support you need to modify.

When you analyze the custom control, use the Object Spy, Keyword View, Editor, and the Record option. Make sure you examine each of the elements described in "Identifying the Building Blocks of Java Add-in Extensibility" on page 16.

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:

- UFT 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 UFT 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 UFT names test objects for that Java class.

- UFT 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. UFT 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, UFT may not record a step at all, or it may record steps that are not specific to the control's behavior. Alternatively, UFT 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 UFT Support and Extensibility Options for a Sample Custom Control

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

The AllLights control shown below is a game application that is not specifically supported on UFT.
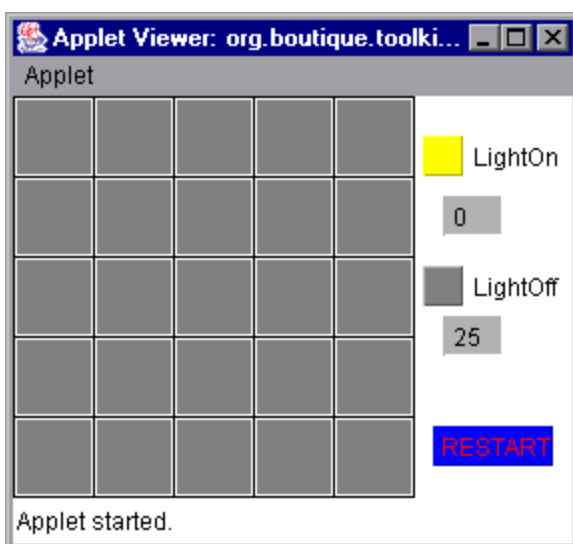
This application operates as follows:

- 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, UFT recognizes it as a generic
JavaApplet named AllLights (the name of the custom class). The icon shown is the
standard JavaApplet class icon.
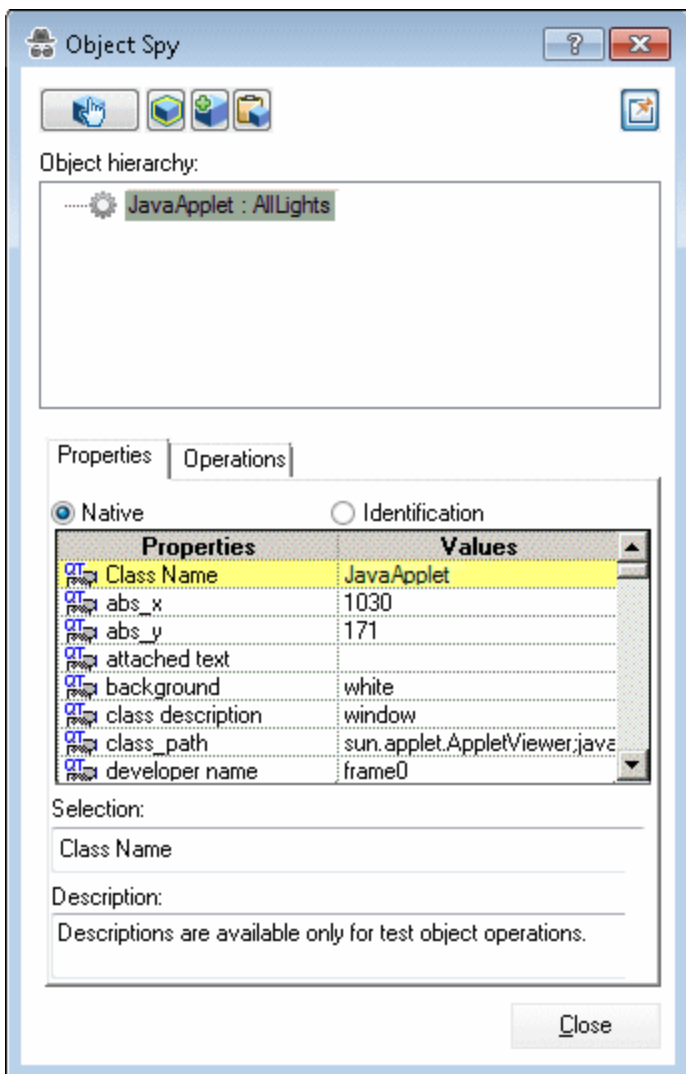
If you record on the AllLights control without implementing support for it, the Keyword
View looks like this:

In the Editor, 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. UFT recognizes the control as an AllLights test object named **Lights** and uses a customized icon. The identification 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.
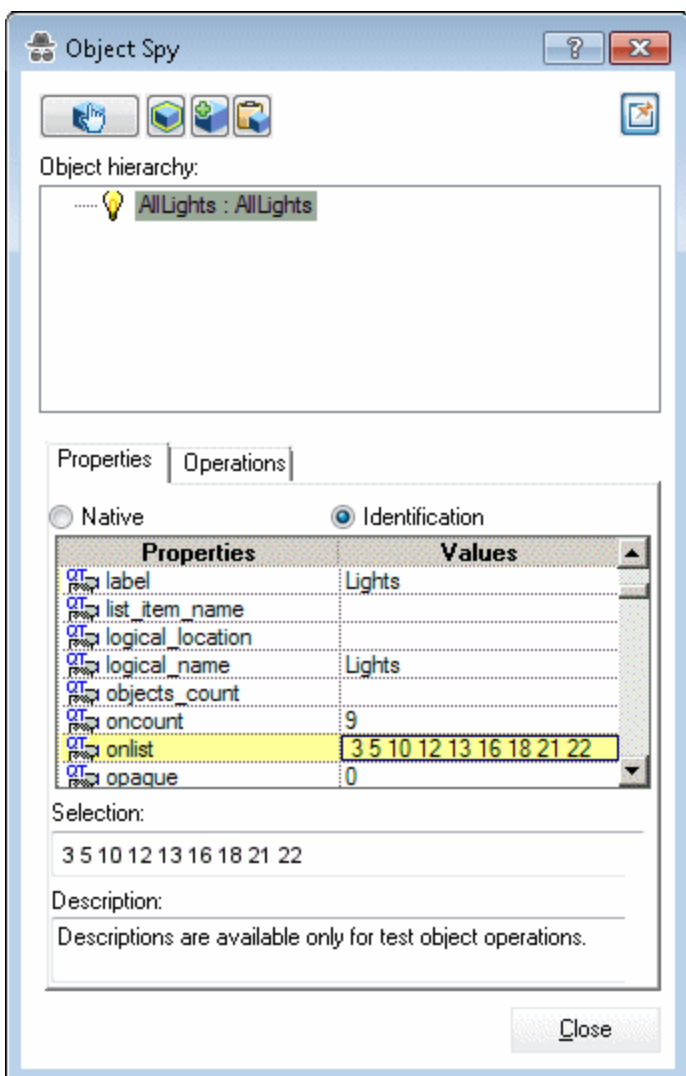
When you are ready to create a test on the control, the **ClickLight** and **Restart** methods are supported. These methods can be recorded or you can select them manually in the **Operation** column of the Keyword View. You can also create a checkpoint to check the value of identification properties, 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 Editor, 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.

Chapter 2: Installing the HP UFT Java Add-in Extensibility Software Development Kit

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

This chapter includes:

About Installing the HP UFT Java Add-in Extensibility SDK

The HP UFT Java Add-in Extensibility SDK enables you to design UFT 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 IDE (integrated development environment) for Java developers 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.)

Using the UFT Java Add-in Extensibility Samples

You can use the samples provided as part of the Java Add-in Extensibility SDK to learn more about designing Java Add-in Extensibility support sets.

The samples assume that the SDK is installed in the **%ProgramFiles%\HP\Unified Functional Testing** folder, on a 32-bit operating system.

If this is not the case, you need to make the following adjustments in the sample toolkit support sets before you can use them on UFT.

If your SDK is not installed in **%ProgramFiles%\HP\Unified Functional Testing**:

1. For each sample, in the toolkit configuration XML file located in the **Configuration** folder (in the **<Java Add-in Extensibility SDK installation folder>\samples\<SampleName>Support** folder), update the **SupportClassPath** property with the current UFT installation path.

2. For each sample, in the **.classpath** file located in the **<Java Add-in Extensibility SDK installation folder>\samples\<SampleName>Support** folder, update all relevant file paths according to the current UFT installation path.

   If your SDK is installed on a 64-bit operating system, make the following additional change for each sample: In the **.classpath** file, modify the path to the **mic.jar** file from **C:/Program Files/HP/Unified Functional Testing/bin/java/classes/mic.jar** to **<Java Add-in Extensibility SDK installation folder>/bin/java/classes64/mic.jar**.

Pre-Installation Requirements

Before you install the UFT Java Add-in Extensibility SDK, do the following:

- Make sure that the Eclipse IDE for Java developers is installed on your computer if you plan to work with the Java Add-in Extensibility Eclipse plug-in. You can download the Eclipse IDE, free of charge, from http://www.eclipse.org/downloads. For a list of supported Eclipse versions, see the *HP Unified Functional Testing Product Availability Matrix*, available from the UFT help folder or the HP Support Matrix page (requires an HP passport).

  When you install the Eclipse IDE, make sure to note the installation location on your computer. You 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 "Tutorial: Learning to Create Java Custom Toolkit Support" on page 157. Additionally, it is recommended to use this plug-in to design at least the skeleton of your toolkit support.

- (Optional) Make sure that UFT with the Java Add-in is installed on the same computer. This enables the Java Add-in Extensibility Eclipse plug-in to interact with UFT, 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 UFT computer, you can deploy the toolkit support to UFT for debugging by simply clicking a button.

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

Installing the HP UFT Java Add-in Extensibility SDK

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

To install the UFT Java Add-in Extensibility SDK:

1. Close all instances of Eclipse and UFT.

2. Insert the Unified Functional Testing DVD into the CD-ROM/DVD drive. The Unified Functional Testing Setup window opens. (If the window does not open, browse to the DVD and double-click **setup.exe** from the root folder.)

3. Click **Add-in Extensibility and Web 2.0 Toolkits**. The Unified Functional Testing Add-in Extensibility and Web 2.0 Toolkit Support screen opens.

4. Click **HP UFT Java Add-in Extensibility SDK Setup**.

   The Welcome screen of the HP UFT Java Add-in Extensibility SDK Setup Wizard opens.

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

   Note: If the Modify, Repair, or Remove Installation screen opens, the SDK is already installed on your computer. Before you can install a new version, you must first uninstall the existing one, as described in "Uninstalling the HP UFT Java Add-in Extensibility SDK" on page 32.

   Read the license agreement and select **I accept the terms in the License Agreement**.

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

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

- This screen displays the location in which the UFT Java Add-in Extensibility SDK will be installed.

  If you install the UFT Java Add-in Extensibility SDK on a computer on which UFT is installed, the UFT installation folder is selected by default.

  You can select a different location for the installation by clicking **Browse**, choosing a folder, and then clicking **OK**.

- If you click **Disk Usage**, a window opens displaying the amount of free disk space on your computer and the amount required for this installation. The space required for the installation includes space required for the UFT Java Add-in Extensibility SDK files and folders (on the disk that you select for this installation) and additional space required on the system disk (the disk on which the operation system is installed), which is used only during the installation process.

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

8. Click **Install**. The Setup program installs the UFT Java Add-in Extensibility SDK and displays a dialog box in which you specify the location of the Eclipse installation on your computer.
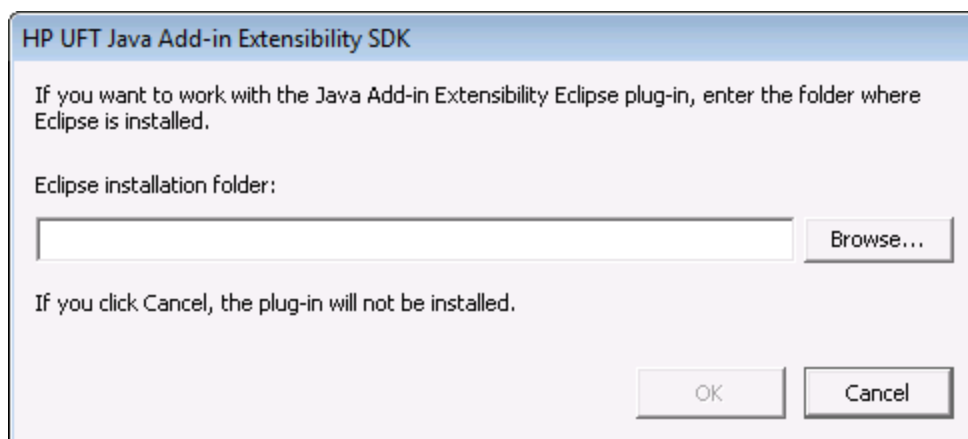
The Java Add-in Extensibility Eclipse plug-in is installed on Eclipse according to the location you specify.

> **Note:** You can install the Java Add-in Extensibility Eclipse plug-in on additional Eclipse installations after you finish the UFT Java Add-in Extensibility SDK installation process. To do this, browse to the **<UFT Java Add-in Extensibility SDK 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 the next step. 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.

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

Click **Finish** to exit the Setup Wizard.

> **Tip:** If you do not see the UFT menu or 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 reopen Eclipse.

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

Uninstalling the HP UFT Java Add-in Extensibility SDK

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

Considerations when uninstalling the SDK:

- When you uninstall the HP UFT Java Add-in Extensibility SDK, the Java Add-in Extensibility Eclipse plug-in is removed from all Eclipse installations.

  If you still see the UFT menu or 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 reopen Eclipse.

- If you use the setup program to uninstall the SDK, you must use the same version of the setup program as you used for the original installation.

- You must be logged on with Administrator privileges to uninstall the UFT Java Add-in Extensibility SDK.

To uninstall the HP UFT Java Add-in Extensibility SDK:

1. Close all instances of Eclipse and UFT.

2. Insert the Unified Functional Testing DVD into the CD-ROM/DVD drive. The Unified Functional Testing Setup window opens. (If the window does not open, browse to the DVD and double-click **setup.exe** from the root folder.)

3. Click **Add-in Extensibility and Web 2.0 Toolkits**. The Unified Functional Testing Add-in Extensibility and Web 2.0 Toolkit Support screen opens.

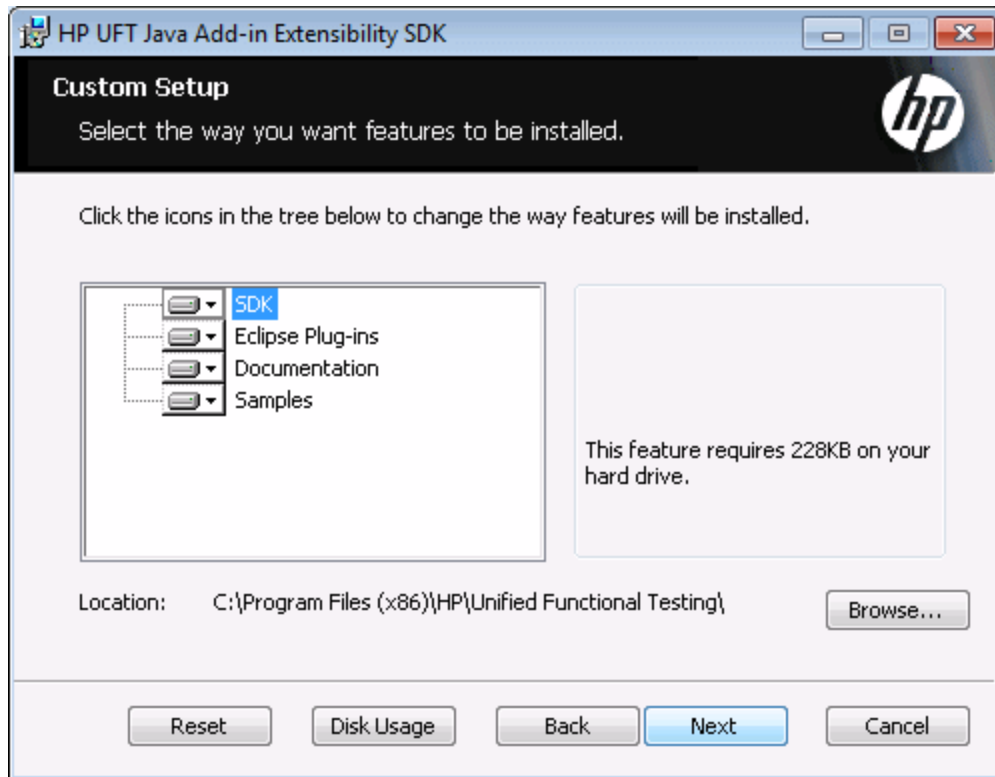4. Click **HP UFT Java Add-in Extensibility SDK Setup**. The Welcome screen of the HP UFT Java Add-in Extensibility SDK Setup Wizard opens.

    > **Note:** If you have a previous version of the SDK installed, the HP QuickTest Professional Java Add-in Extensibility SDK Setup wizard opens. Using this wizard, follow the instructions below to uninstall the old SDK version.

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

6. Follow the instructions in the wizard to remove the HP UFT Java Add-in Extensibility SDK.

Troubleshooting and Limitations – Java Add–in Extensibility Eclipse Plug-in

This section describes troubleshooting and limitations when working with the UFT Java Add-in Extensibility.

When the Java Add-in Extensibility plug-in is installed on Eclipse 3.3, using the software update options in the Eclipse Help menu may fail.

Workaround: Save the **eclipse\features\com.mercury.qtjext.PluginFeature_ 1.0.0\feature.xml** file in UTF-8 format instead of ANSI format.

Chapter 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.

This chapter includes:

About Custom Toolkit Support

When you extend UFT support of a custom toolkit, you create an API that is based on the existing UFT 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 UFT and the Java application being tested, enabling UFT 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, see the *UFT Java Add-in Extensibility API Reference* (available with the Java Add-in Extensibility**SDK Help**).

Before you begin to create a custom toolkit support set, you must plan it carefully. For more information, see "Planning Custom Toolkit Support" on page 90.

The UFT 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 "Tutorial: Learning to Create Java Custom Toolkit Support" on page 157. 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 UFT Java Add-in Extensibility Eclipse plug-in, as well as using the plug-in, see "Installing the HP UFT Java Add-in Extensibility Software Development Kit" on page 24.

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

Introducing Java Add-in Extensibility Terminology

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

- **Basic user interface component:**

  - In the AWT toolkit: **java.awt.Component**

  - In the SWT toolkit: **org.eclipse.swt.widgets.Widget**

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

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

- **Custom toolkit support.** Extends UFT 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**.)

- **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.

Preparing to Create Support for a Custom Toolkit

You can extend UFT 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 UFT 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 UFT 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 UFT, 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 UFT 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 UFT 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 41.

- One toolkit configuration file, described on page 42.

- One or more test object configuration classes (if this support set introduces new test object classes or extends existing ones), described on page 44.

- Custom support classes (mapped to the custom classes), described on page 53.

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 UFT installation folder and reference the java packages. For more information, see "Deploying and Running the Custom Toolkit Support" on page 77.

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.

    After you develop the support and deploy it to UFT, UFT displays the custom toolkit name in all of the dialog boxes that display lists of add-ins or supported environments. For example, when UFT opens, it displays the custom toolkit name

as a child of the Java Add-in in the Add-in Manager dialog box and the UFT user can specify whether to load support for that toolkit.

Providing unique toolkit names allows a single UFT 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 41.

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 42.

   > **Note:** The custom toolkit name that UFT displays (in the Add-in Manager and other dialog boxes) is derived from this file name.

5. Consider the behavior (fields and methods) of the custom controls, and map the custom controls to a UFT test object class. For more information, see "Mapping a Custom Control to a Test Object Class" on page 58.

   If you require any new UFT 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 44.

   > **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 *HP UFT Test Object Schema Help* (available with the Java Add-in Extensibility**SDK**Help). 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 77.

When UFT opens, the UFT user can select (in the Add-in Manager dialog box) the environments or custom toolkits for which to load support. UFT then loads the test object class definitions (from the test object configuration files) for all custom Java toolkits whose support is loaded. This enables you to use the same test object class definitions when supporting different custom Java 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 53.

   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**:

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, you must add the following items to the build path (the classpath used by the compiler):

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

- **<Java Add-in Extensibility SDK installation folder>\bin\Java\sdk\ 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:**
> - The build path must also include the locations of all parent classes of the custom classes. Add these locations manually to the build path if any custom classes are not derived directly from SWT, AWT, or JFC (Swing) and the parent classes are not located in the same location as the custom classes.
>
> - If, at any time, the custom controls are modified in a way that might affect the support, you must recompile the support classes, adjusting them if necessary.

Understanding the Toolkit Support Class

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

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 UFT 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 UFT installation folder. This file provides the information that UFT 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)

  UFT 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

  UFT displays the description in the Add-in Manager when a user selects the custom toolkit's name in the list of available add-ins. If you are developing this toolkit support set for distribution, include a **Provided by** clause in this description, specifying the relevant person or company.

- 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
<?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>
```

You can validate your toolkit configuration file against:
<UFT installation folder>\bin\java\sdk\eclipse\plugins
com.mercury.qtjext.plugin.QTJavaExt_1.0.0\ToolkitSchema.xsd

For information on the structure and syntax of the toolkit configuration file, see the *UFT Java Add-in Extensibility Toolkit Configuration Schema Help* (available with the Java Add-in Extensibility**SDK Help**).

For information on where to store the toolkit configuration file, see "Deploying and Running the Custom Toolkit Support" on page 77.

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 58.

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

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

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

  If you add test object methods to existing test object classes, you might add a prefix to the method name that indicates the toolkit support for which you added the method (for example, **CustomButtonClick**, **CustomEditSet**). This enables test designers to easily identify the custom methods and use them in test steps only if they know that the custom method is supported for the specific object.

- If you add an identification property, it appears in UFT in the list of properties for all test objects of this class, but has no value unless it is implemented for the specific supported object.

In the test object configuration XML file, you create a **ClassInfo** element for each test object class that you want to define. In addition, you define the name of the environment or custom toolkit for which the test object classes are intended (in the

PackageName attribute of the **TypeInformation** element), and the UFT add-in which these test object classes extend (in the **AddinName** attribute of the **TypeInformation** element).

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

For more information, see "How UFT Loads the Test Object Configuration XML " on page 50.

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

Class Name and Base Class

The name of the test object class and its attributes, including the base class—the test object class that the new test object class extends (relevant only when defining new test object classes). A new test object class extends an existing JavaUFT test object class, directly or indirectly. The base class may be a class delivered with UFT or a class defined using Java Add-in Extensibility.

By default, the base class is JavaObject.

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

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

> the UFT Add-in Manager, you must also select to load the toolkit support set
> that it extends.

Generic Type

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

Generic types are used when filtering objects (for example, in the Step Generator's
Select Object for Step dialog box and when adding multiple test objects to the object
repository). Generic types are also used when creating documentation strings for the
Documentation column of the Keyword View (if they are not specifically defined in the
test object configuration file).

Test Object Operations

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

- The arguments, including the argument type (for example, `String` or `Integer`), direction (`In` or `Out`), whether the argument is mandatory, and, if not, its default value.

- The operation description (shown in the Object Spy and as a tooltip in the Keyword View and Step Generator).

- The Documentation string (shown in the **Documentation** column of the Keyword View and in the Step Generator).

- The return value type.

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

Default Operation

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

Identification Properties

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

- The identification properties that are used for the object description.

- The identification properties that are used for **smart identification**. (This information is relevant only if smart identification is enabled for the test object class. To enable smart identification, use the Object Identification dialog box in UFT.)

- The identification properties that are available for use in checkpoints and output values.

- The identification properties that are selected by default for checkpoints (in the UFT Checkpoint Properties dialog box).

Icon File

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

Help File

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

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

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 236.

You can validate your test object configuration file against:
**<UFT installation folder>\bin\java\sdk\eclipse\plugins\
com.mercury.qtjext.plugin.QTJavaExt_1.0.0\ClassesDefintions.xsd**

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

For information on the location in which to store the test object configuration file, see .

Test Object Configuration File Example

The following example shows parts of the test object configuration file that defines the **Calculator** test object class definition for the **javaboutique** custom toolkit:

```
<TypeInformation Load="true" PackageName="javaboutique"
                      AddinName="Java">
    <ClassInfo BaseClassInfoName="JavaApplet"
        DefaultOperationName="Calculate" Name="Calculator">
        <IconInfo
        IconFile="C:\Program Files\HP\Unified Functional
            Testing\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>
        <IdentificationProperties>
            <IdentificationProperty ForVerification="true"
                ForDefaultVerification="true "Name="value"/>
            <IdentificationProperty ForVerification="true"
 Name="objects count"/>
            <IdentificationProperty Name="width"/>
            <IdentificationProperty ForDescription="true"
 Name="toolkit class"/>
                ...
        </IdentificationProperties>
    </ClassInfo>
</TypeInformation>
```

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 **PackageName** attribute in the **TypeInformation** element indicates that the **Calculator** test object class is created for the **javaboutique** toolkit support.

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 UFT.

- **objects count.** Available for checkpoints but not selected by default.

- **toolkit class.** Used for the test object description but not available for checkpoints.

How UFT Loads the Test Object Configuration XML

Each time you run UFT, 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 class definition. For more information, see "Understanding How UFT Merges Test Object Configuration Files" on the next page.

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

Identification dialog box): **AssistivePropertyValue**, **ForAssistive**, **ForBaseSmartID**, **ForDescription**, **ForOptionalSmartID**, and **OptionalSmartIDPropertyValue**. These attributes determine the lists of identification properties used for different purposes in UFT.

Therefore, by default, UFT reads the values of these attributes from the XML only once, to prevent overwriting any changes a user makes using the Object Identification dialog box. In this way, UFT provides persistence for the user defined property lists. For more information, see "Modifying Identification Property Attributes in a Test Object Configuration File" on page 83.

Understanding How UFT Merges Test Object Configuration Files

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

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

- The **Load** attribute of the **TypeInformation** element is set to `false`.

- The environment relevant to the test object configuration file is displayed in the Add-in Manager dialog box, and the UFT user selects not to load the environment.

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

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

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

ClassInfo Elements

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

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

ListOfValues Elements

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

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

- If a **ListOfValues** is defined in a configuration file with a lower priority than the existing definition, the lower priority definition is ignored.

Operation Elements

- **Operation** element definitions are either added, ignored, or overridden, depending on the priority of the test object configuration file.

- If an **Operation** element is defined in a test object configuration file with a priority higher than the existing definition, the operation is added to the existing definition for the class. If a conflict arises between **Operation** definitions in different files, the definition in the file with the higher priority overrides (replaces) the definition with

the lower priority (the definitions are not merged).

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

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 UFT capabilities, thus providing the UFT Java Add-in Extensibility.

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 42) 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" on page 55.

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 58.

After you make the preliminary decisions regarding hierarchy and test object class, you are ready to write the main part of the UFT 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 59.

- **Replay methods.** Used to support test object methods. For more information, see "Supporting Test Object Methods" on page 63.

- **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 UFT 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 UFT recording capability. For more information, see "Supporting the Record Option" on page 65.

- **Utility methods.** Used to control the Extensibility. These methods do not support the specific functionality of the custom class; they control the interface between UFT 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 75. The methods are described in detail, in the following sections: "Supporting the Record Option", " Supporting Top-Level Objects" on page 68, and "Supporting Wrapper Controls" on page 69.

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 76 and the *UFT Java Add-in Extensibility API Reference* (available with the Java Add-in Extensibility **SDK**Help).

For a short summary of the types of methods a custom class contains, see "Support Class Summary" on page 75.

Determining the Inheritance Hierarchy for a Support Class

Within the custom toolkit for which you create UFT 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:
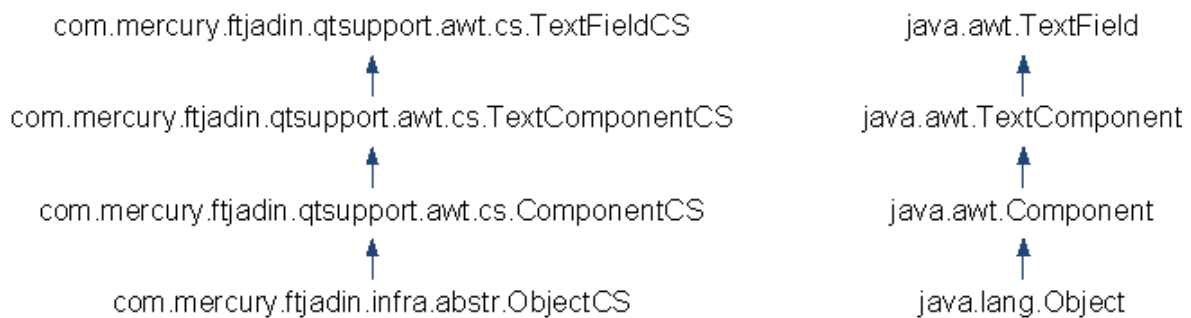
In this example, a support class exists for every custom class, but this is not mandatory.

When UFT learns an object, it can always identify the class name of the object. According to the class, UFT determines the inheritance hierarchy of this class. UFT then searches the toolkit configuration files for the support class that is mapped to that class. If no support class is found, UFT 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 HP 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:
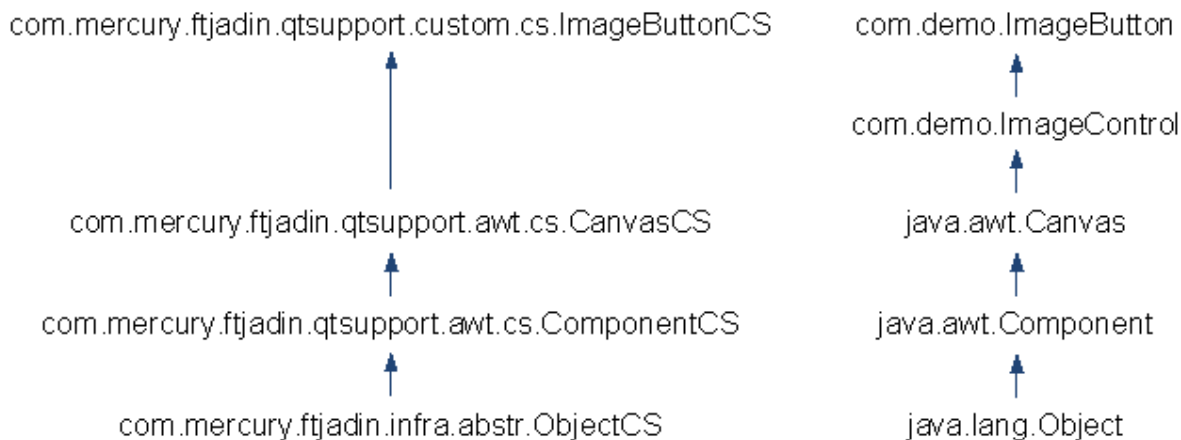
No support class is mapped to the superclass of ImageButton, **ImageControl**. Therefore, the support class for **ImageButton** extends the support class mapped to the higher level—**CanvasCS**.

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). 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 112.

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 UFT Java Add-in, as well as in those that are part of Extensibility custom toolkit support. These files are located in **<UFT Installation Folder> bin\java\classes\builtin** and in **<UFT 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 UFT 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 UFT 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 {};
```

Note: If you design your support classes to extend support classes from another toolkit support set, you create a dependency between the two toolkit support sets. Whenever you select to load the extending toolkit support set in the UFT Add-in Manager, you must also select to load the toolkit support set that it extends.

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 UFT 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 UFT learns an object, it finds the support class to use for this object, as described in "Understanding the Hierarchy of Support Classes" on page 55. UFT then uses the **Class Name** identification property to determine which test object class is mapped to this control. UFT then uses this test object class name to find the test object definition, which can be taken from either an existing UFT 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 44.

Supporting Identification Properties

The identification properties of a custom control are defined in the test object class. This can be an existing UFT 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 lowercase letters (even if the property name in the test object configuration file contains uppercase 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's 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.

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 UFT 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, see the *HP UFT Object Model Reference for GUI Testing*.

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

Reserved Identification Properties

UFT uses a number of identification properties internally and expects them to have a specific implementation:

- UFT supports the following identification properties for every test object class and uses these properties to retrieve specific information about the object. Do not override UFT's implementation of these identification properties in your toolkit support set: **index** (or **class_index**), **class** (or **class_name**), **to_class**, **toolkit_class**.

- In JavaTree and JavaList test objects, there are identification properties named **tree_content** and **list_content** (respectively) that are used in checkpoints. UFT calculates these properties based on the **count** identification property and the **GetItem** test object method, as follows: UFT 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 UFT 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 UFT expects.

Common Identification Property Support Methods

The following basic identification property support methods are commonly used when creating support classes. In "Tutorial: Learning to Create Java Custom Toolkit Support" on page 157, 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 58) 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. UFT 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 UFT to use its **label** property as the **attached text** for an adjacent control. For more information, see "New UFT Custom Static-Text Support Class Wizard" on page 142.

- 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 UFT 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 UFT 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, see the *UFT Java Add-in Extensibility API Reference* (available with the Java Add-in Extensibility SDK Help).

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 UFT about the success or failure of the test object method. The return value can be retrieved and used in later steps of a UFT GUI test.

For example, the `GetItem_replayMethod` in the SearchBoxCS class (that supports the SearchBox custom control) returns the name of a specified item in addition to the return code **OK**:

```
public Retval GetItem_replayMethod(Object obj, String Index) {
    SearchBox sb = (SearchBox) obj;
    int indexint;
    String item;
    indexint = MicAPI.string2int(Index);
    if (indexint == MicAPI.BAD_STRING) {
        return Retval.ILLEGAL_PARAMETER;
    }
    if (indexint < 0 || indexint > sb.getItemCount() - 1) {
        return Retval.OUT_OF_RANGE;}
    item = sb.getItem(indexint);
    return new Retval(RError.E_OK, item);
}
```

For more information on the **MicAPI.Retval** values recognized by UFT, see the *UFT Java Add-in Extensibility API Reference* (available with the Java Add-in Extensibility SDK Help).

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

Do not override the implementation of fundamental UFT methods, such as:
CheckProperty, **FireEvent**, **GetRoProperty**, **GetTOProperty**, **SetTOProperty**, and
WaitProperty.

> **Note:** When supporting JavaTree and JavaList test objects, you must make sure
> that the **count_attr** and **GetItem_replayMethod** methods return the type of
> information that UFT expects. For more information, see "Reserved Identification
> Properties" on page 61.

Supporting the Record Option

You can extend UFT 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 UFT 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 UFT 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 "Tutorial: Learning to Create Java Custom Toolkit Support" on page 157, 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 UFT 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 UFT, using the **MicAPI.record** methods.

For information on how to use **MicAPI.record**, see the *UFT Java Add-in Extensibility API Reference* (available with the Java Add-in Extensibility SDK Help).

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, UFT 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 UFT 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 86.

For information on recording on wrapper controls, see "Supporting Wrapper Controls" on page 69.

Note: If **MicAPI.record** is called when there is no active UFT 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 UFT to recognize the custom control as the highest Java object in the test object hierarchy, you need to inform UFT 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.
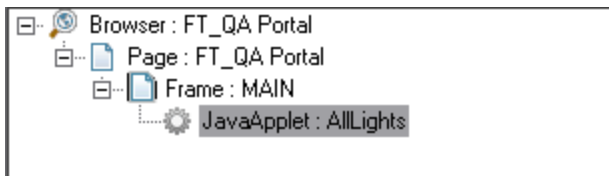
Only a container object can be a top-level object. A container object is one that extends **java.awt.container** if it is AWT-based, or **org.eclipse.swt.widgets.Composite** if it is SWT-based.

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. An example of wrapper control is the AwtCalc calculator control.
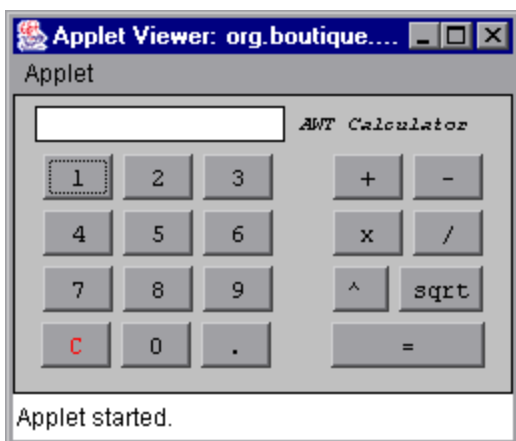
When UFT learns a wrapper control, it does not learn the controls within it separately as descendants. If you record a test on a wrapper control, events that occur on the controls within it are recorded as operations on the wrapper control.

> **Note:** Only AWT-based controls can be supported on UFT 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 UFT to record clicks on digit buttons as **Calculator.SetValue** steps, and clicks on operator buttons as **Calculator.SetOperator** steps.

Understanding How UFT Handles Wrapper Controls

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

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

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

Similarly, before UFT records an operation on a control, UFT checks if any wrappers are registered for this type of control. If there are registered wrappers, UFT searches for the one to which this particular control belongs. If UFT finds a relevant wrapper, UFT 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()

The **registerWrapperInspector** 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**:

```
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)

UFT calls the **checkWrappedObject** 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)

UFT calls the **wrapperRecordMessage** method during a recording session when a wrapped object sends a record message. UFT 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 is 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
```

```java
        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 the **blockWrappedObjectRecord** method returns **false**, the controls contained in the wrapper generate record messages in response to events as if they were independent controls. UFT 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 the **blockWrappedObjectRecord** 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 UFT, 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, see the *UFT Java Add-in Extensibility API Reference* (available with the Java Add-in Extensibility SDK Help).

Method Type	Syntax	Common Methods

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

Method Type	Syntax	Common Methods

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 UFT. 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 86.

To use the methods provided in MicAPI, add an `import com.mercury.ftjadin.custom.MicAPI;` statement in your code. For details on these methods, see the *UFT Java Add-in Extensibility API Reference* (available with the Java Add-in Extensibility SDK Help).

Deploying and Running the Custom Toolkit Support

The final stage of extending UFT 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 UFT.

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

About Deploying the Custom Toolkit Support

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

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

> **Note:** Only applications that are opened after loading or unloading support for the custom toolkit are affected.

If you do not load the support for your custom toolkit, the code that you designed in your toolkit support set does not run.

If you load support for your custom toolkit:

- UFT recognizes the controls in your custom toolkit and can run test steps on them.

- UFT displays the name of your custom toolkit in the **Environment** list in all of the dialog boxes that display lists of add-ins or supported environments.

- UFT displays the list of test object classes defined by your toolkit support set in dialog boxes that display the list of test object classes available for each add-in or environment. (For example: Define New Test Object dialog box, Object Identification dialog box.)

> **Note:** Test object classes defined in a toolkit support set that was developed using a Java Add-in Extensibility SDK version earlier than 10.00 are displayed in the UFT dialog boxes as Java test object classes. To cause UFT to display these test object classes under the correct environment name, change the **PackageName** attribute in the test object configuration file to the name of the custom toolkit, as it appears in the Add-in Manager. Additionally, if an **index** identification property is implemented for any test object classes in the toolkit support set, remove this implementation to enable the use of the **Generate Scripts** button in the Object Identification dialog box.

Deploying the Custom Toolkit Support

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

File Name	Location
<Custom Toolkit Name>.xml	**<UFT Installation Folder>\bin\java\classes\extension**

File Name	Location
<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.	• **<UFT Installation Folder>\Dat\Extensibility\Java** • **<Unified Functional Testing Add-in for ALM Installation Folder>\Dat\ Extensibility\Java** (Optional. Required only if Unified Functional Testing Add-in for ALM is installed.)
<Custom Toolkit Name>Support.class **<CustomClass>CS.class**	All of the compiled Java support classes can be packaged in class folders or Java archives on the computer on which UFT is installed, or in an accessible network location. Specify the location in `<Custom Toolkit Name>.xml`.
Icon files for new test object classes (optional)	The file can be a **.dll** or **.ico** file, located on the computer on which UFT is installed, or in an accessible network location. Specify the location in **<Custom Toolkit Name>TestObjects.xml**.

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, specify their locations in the test object configuration file.

> **Note:** Compile the support classes before deploying and check for compilation errors, to avoid run-time failure.

If you modify attributes of **Identification Property** elements in the test object configuration file, it is recommended to keep the **DevelopmentMode** attribute of the **TypeInformation** element set to `true` during the design stages of the custom toolkit support. For more information, see "Modifying Identification Property Attributes in a Test Object Configuration File" on page 83.

If you develop custom toolkit support using the UFT Java Add-in Extensibility plug-in in Eclipse, and UFT is installed on your computer, you deploy toolkit support by clicking the **Deploy Toolkit Support** Eclipse toolbar button, or by choosing **UFT > Deploy Toolkit Support**. The XML configuration files are copied to the correct UFT 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.) For details on deploying support using the Eclipse plug-in, see "Deploy Toolkit Support " on page 148.

If you do not use the UFT Java Add-in Extensibility plug-in in Eclipse, or if UFT is installed on another computer, you must perform the deployment manually, according to the information in the table on page 77.

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 UFT.

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 77.

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 UFT is installed.

If you set the **DevelopmentMode** attribute of the **TypeInformation** element in the test object configuration file to `true` while developing the custom toolkit support, make sure to remove this attribute (or set it to `false`) before deploying the custom support for regular use. For more information, see "Modifying Identification Property Attributes in a Test Object Configuration File" on page 83.

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 UFT.

   In addition, if you have new test object classes using specific icons, place the icon files in a location that can be accessed by UFT.

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 77.

Running an Application with Supported Custom Controls

After you deploy the custom toolkit support, you can perform UFT 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 using a version earlier than 3.3, 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 (when working with Eclipse versions earlier than 3.3):

1. Right-click the application file in the Eclipse Package Explorer. Select **Run As > SWT Application**.

2. In the Eclipse toolbar, select **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 run the application again. (Right-click the application file in the Eclipse Package Explorer and select **Run As > SWT Application)**.

Modifying Deployed Support

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

- If you modify the 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 UFT 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.

Modifying Identification Property Attributes in a Test Object Configuration File

The following attributes of the **Identification Property** element in the test object configuration file specify information that can be modified in UFT (using the Object Identification dialog box): **AssistivePropertyValue**, **ForAssistive**, **ForBaseSmartID**, **ForDescription**, **ForOptionalSmartID**, and **OptionalSmartIDPropertyValue**. These attributes determine the lists of identification properties used for different purposes in UFT. For more information, see the *UFT Test Object Schema Help*, available in the UFT Java Add-in Extensibility Help.

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

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

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

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

Considerations When Modifying Identification Properties Attributes

- If you modify attributes of **Identification Property** elements in the test object configuration file, keep the **DevelopmentMode** attribute of the **TypeInformation** element set to `true` during the design stages of the custom toolkit support. This ensures that UFT uses all of the changes you make to the file.

- Before you deploy the toolkit support set for regular use, be sure to remove the **DevelopmentMode** attribute of the **TypeInformation** element (or set it to `false`). Otherwise, every time UFT opens it will refresh the property lists based on the definitions in the test object configuration file. If UFT users change the property lists using the Object Identification dialog box, their changes will be lost when they reopen UFT.

- Though UFT does not remove existing properties from the property lists when reading a modified test object configuration file (unless the **DevelopmentMode** attribute is set to `true`), it does add properties and adjust the order of the lists based on the definitions in the file. If UFT users removed properties from the lists or modified their order using the Object Identification dialog box, those changes will be lost when a modified file is loaded.

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

Removing Deployed Support

When opening UFT, the UFT user can use the Add-in Manager to instruct UFT whether to load the support provided for any particular toolkit. If the support for your custom toolkit is not loaded, the code that you designed in your toolkit support set does not run, and the test object classes that you defined in the test object configuration file are not available in UFT.

- If you want to remove support for a custom toolkit from UFT after it is deployed, you must delete its toolkit configuration file from: **<UFT 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:
  <UFT Installation Folder>\Dat\Extensibility\Java (and **<Unified Functional Testing Add-in for ALM Installation Folder>\Dat\Extensibility\Java** if relevant).

- If you want to remove only parts of the custom toolkit support that you created, consider the following:

  - To remove support for a specific custom class, delete its custom support class, and remove the references to this support class from the toolkit configuration file.

    Before you delete a custom support class, make sure that no other custom support classes extend it.

  - To remove a new test object class that you defined, remove its definition from the test object configuration file.

    Before you remove the definition of a test object class, make sure that no custom classes are mapped to this test object class and that no other test object classes extend it.

  - To remove support for test object methods or identification properties that you added, remove the relevant support methods from your custom support class.

    Removing support for test object methods or identification properties from the support class does not remove them from the test object class definition. They are available in UFT when editing tests but are not supported for this custom class.

  - To remove your custom support for test object methods or identification properties whose implementation you overrode, remove the relevant support methods from your custom support class.

  - To remove test object methods or identification properties from the test object class definition, remove them from the test object configuration file.

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 *UFT Java Add-in Extensibility API Reference* (available with the Java Add-in Extensibility SDK Help).

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 UFT, 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 UFT GUI 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 UFT 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 select **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 on remote debugging, see the *Eclipse Help*.

Workflow for Implementing Java Add-in Extensibility

The following workflow summarizes the steps you need to perform to create UFT 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:
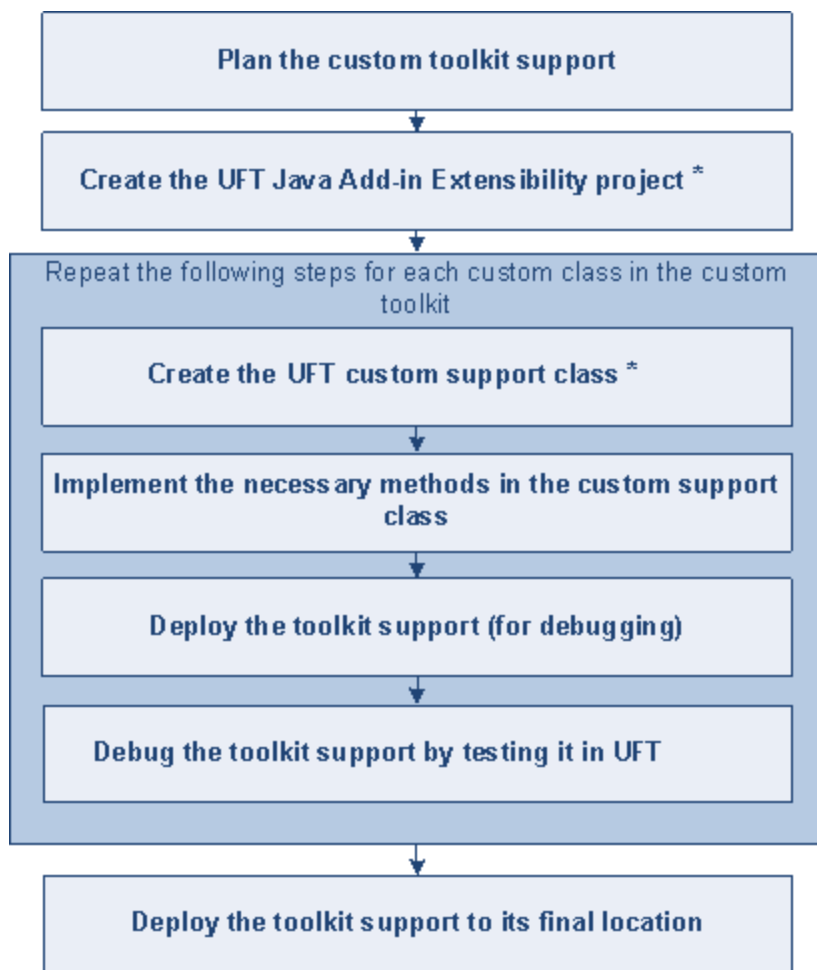
* You can use the wizards in the UFT Java Add-in Extensibility Eclipse plug-in to create the custom toolkit support project, the custom classes, and all of the required files. Alternatively, if you choose not to use the wizards, you must create the necessary packages and files manually, as described in "Creating a Custom Toolkit Support Set" on

page 38. 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.

Chapter 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 UFT 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 "Implementing Custom Toolkit Support" on page 34.

This chapter includes:

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 is the name of the custom toolkit?

  Provide a unique name for the custom toolkit. After you develop the support and deploy it to UFT, UFT displays the custom toolkit name in all of the dialog boxes that display lists of add-ins or supported environments. For example, when UFT opens, it displays the custom toolkit name as a child of the Java Add-in in the Add-in Manager dialog box and the UFT user can specify whether to load support for that toolkit.

- What custom classes are included in the custom toolkit?

  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 37.

- 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 55.

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 UFT, 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 UFT to recognize controls of this class—what type of test object you want to represent the controls in UFT GUI 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 UFT perspective using the Object Spy, the Keyword View, and the Record option. This enables you to see how UFT recognizes the control without custom support, and helps you to determine what you want to change.

To view an example of analyzing a custom control using UFT, see "Analyzing the Default UFT Support and Extensibility Options for a Sample Custom Control" on page 20.

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 95.

> **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 UFT custom support is not yet available?

    b. Does the custom control have identification properties or test object methods that require the same UFT 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 UFT 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.

      > **Note:** If you create test object classes that extend test object classes defined in another toolkit support set, you create a dependency between the two toolkit support sets. Whenever you select to load the extending toolkit support set in the UFT Add-in Manager, you must also select to load the toolkit support set that it extends.

   b. Do you want UFT 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 UFT 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 UFT.

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

UFT GUI tests by using the Record option?

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

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 UFT 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 UFT 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 UFT 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:

☑ Custom Class Support Planning Checklist
❑ 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 UFT 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 UFT Java Add-in Extensibility Eclipse Plug-In" on page 97.

If you choose not to use the Java Add-in Extensibility wizard in Eclipse, you can still extend full support for the custom toolkit manually using the information in "Implementing Custom Toolkit Support" on page 34.

Chapter 5: Using the UFT Java Add-in Extensibility Eclipse Plug-In

The UFT 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 34.

This chapter includes:

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

When you install the UFT Java Add-in Extensibility SDK, the UFT 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 on installing and uninstalling the Java Add-in Extensibility SDK, see "Installing the HP UFT Java Add-in Extensibility Software Development Kit" on page 24.

You can use the wizards supplied by the UFT 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 34), 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 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 Custom Support Class Wizard (described on page 110). To create a support class for a custom static-text class, you use the New Custom Static Text Support Class Wizard (described on page 142).

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 UFT 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 UFT for debugging. These commands are described in "Working with UFT Commands in Eclipse" on page 147.

> **Caution:** 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 81.

New UFT Java Add-in Extensibility Project Wizard

You use the New UFT 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 UFT Java Add-in Extensibility project, you can create support for each of the custom toolkit classes. To do this, you use the New Custom Support Class Wizard, described on page 110 (or the New Custom Static Text Support Class Wizard, described on page 142).

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

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

2. Expand the **Unified Functional Testing** folder and select **UFT Java Add-in Extensibility Project**.

3. Click **Next**. The New Project Screen opens (described on page 101).

   > **Tip:** You can shorten this process by customizing Eclipse to provide **UFT Java Add-in Extensibility Project** as an option in the **New** menu. To do this, perform

the following: Select **Window > Customize Perspective**. In the Shortcuts tab in the dialog box that opens, select the **Unified Functional Testing** and **UFT Java Add-in Extensibility Project** check boxes. Click **OK**.

UFT Java Add-in Extensibility Project Screen

In the UFT Java Add-in Extensibility Project screen, you can create a UFT Java Add-in Extensibility project and define the project layout. The details on this screen may vary, depending on the version of Eclipse that you are using.
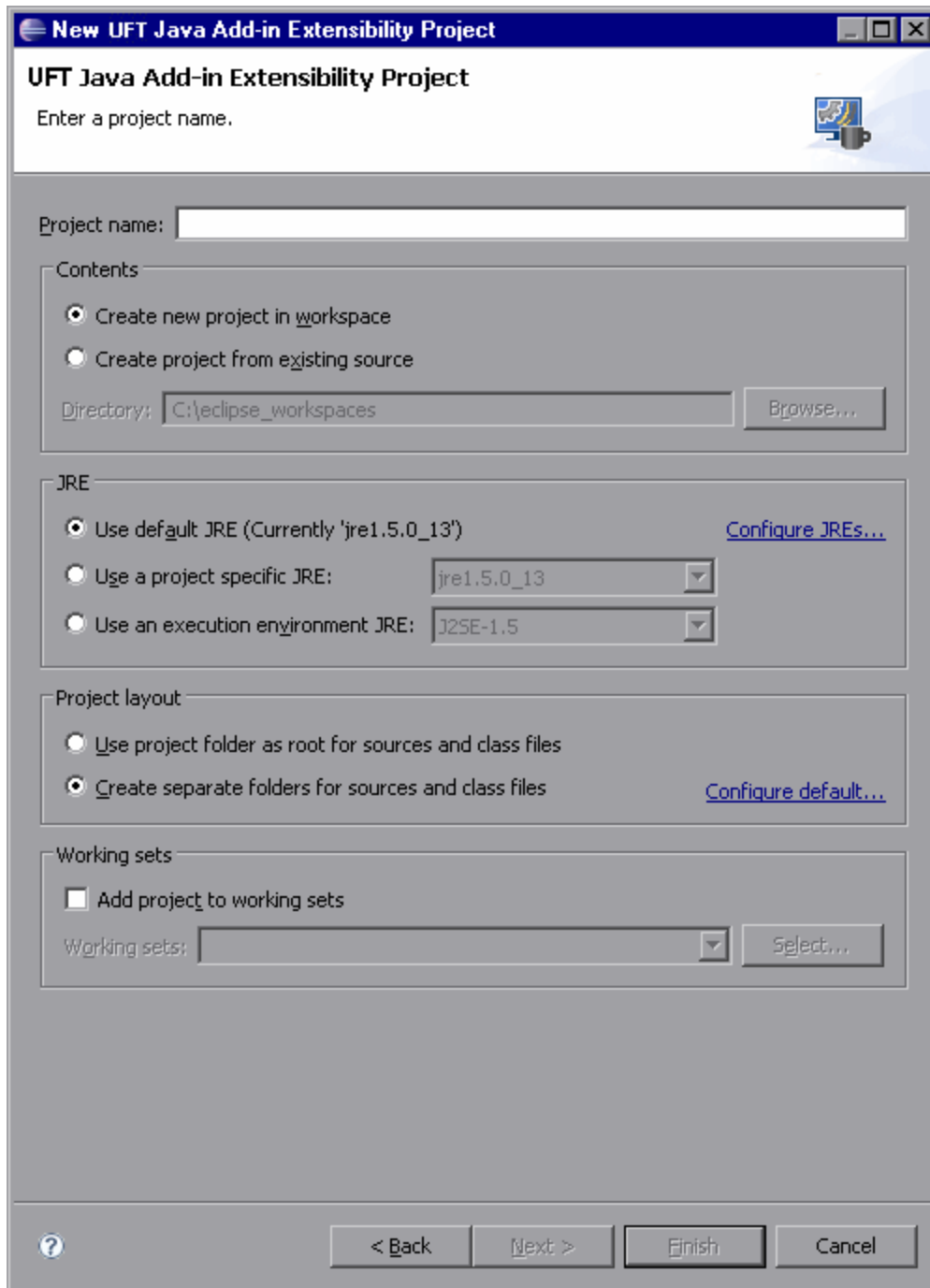
Perform the following:

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

2. In the **Project Layout** area, select **Create separate folders for sources and class files**. (In earlier Eclipse versions this option is named **Create separate source and output folders**.)

3. Click **Next** to continue to the Custom Toolkit Details Screen (described on page 103).

For information on the options available in this Eclipse wizard screen, see 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. When you click **Finish** the Project Summary screen described on page 108 opens.
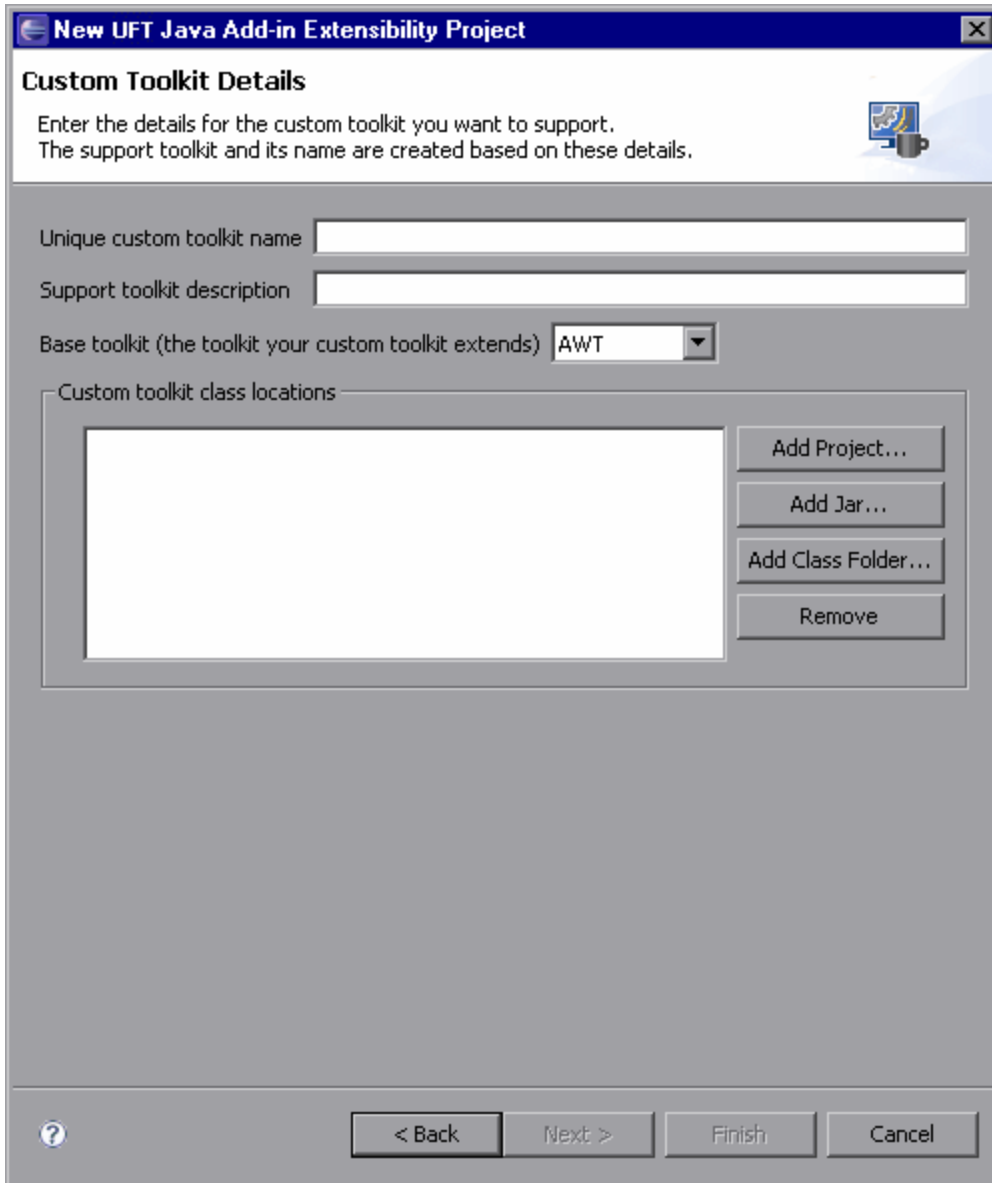
In this wizard screen you specify the following details:

- **Unique custom toolkit name.** A name that uniquely represents the custom toolkit for which you are creating support. UFT displays this name in all of the dialog boxes that display lists of add-ins or supported environments. Providing unique toolkit names enables a single UFT installation to support numerous custom toolkit support sets simultaneously.

  The name must begin with an English letter and contain only alphanumeric characters and underscores.

  The wizard uses this name when it creates the new toolkit support set. For example:

  - The toolkit support class is named `<custom toolkit name>Support`.

  - The toolkit configuration file is named `<custom toolkit name>.xml`. (The custom toolkit name that UFT displays in the Add-in Manager and other dialog boxes is derived from the name of this file.)

  - If the wizard creates a test object configuration file, it enters the custom toolkit name in the **PackageName** attribute of the **TypeInformation** element. This enables UFT to associate the new test object classes to the correct custom toolkit.

    You cannot specify the name of a custom toolkit whose support is already deployed to UFT. 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 **<UFT Installation Folder> bin\java\classes\extension**, delete the toolkit configuration file, and then use the **Reload Support Configuration** command described on page 149.

- **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 UFT 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.

The build path must also include the locations of all parent classes of the custom classes. Add these locations manually to the build path in your project if any custom classes are not derived directly from SWT, AWT, or JFC (Swing) and the parent classes are not located in the same location as the custom classes.

- The "Custom Class Selection Screen" on page 112 in the "New UFT Custom Support Class Wizard" on page 110 (shown on page 112) 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 or remove custom class locations in a Java Add-in Extensibility project after it is created, use the Properties dialog box for UFT Java Add-in Extensibility projects described on page 109.

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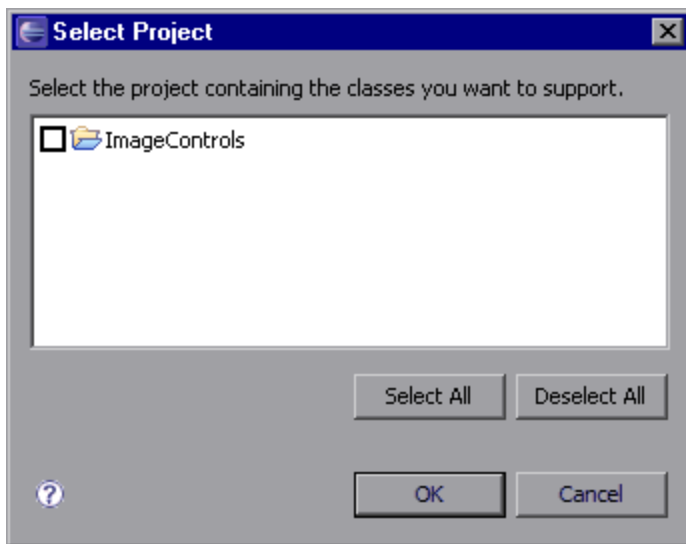
Select the check box for the appropriate project and click **OK** to add it to the **Custom toolkit class locations** box.

- Click **Add Jar** to add a Java archive (**.jar**) file. The Open dialog box opens.
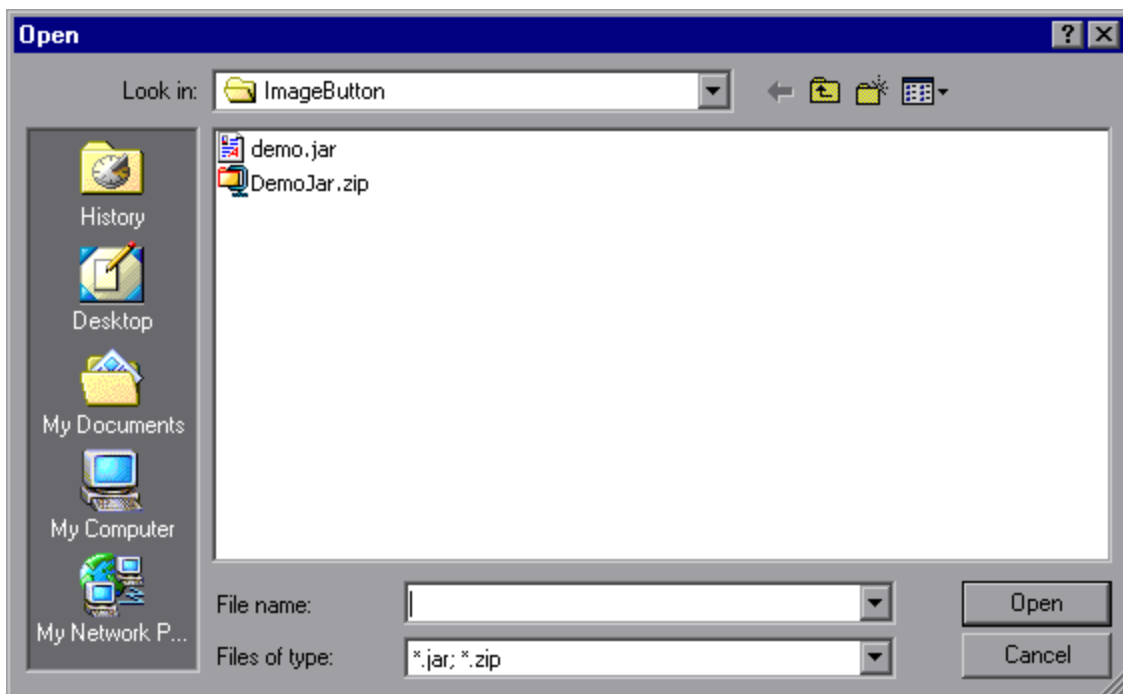
Browse to the appropriate Java archive file, select it, and click **OK** to add it to the **Custom toolkit class locations** box.

- Click **Add Class Folder** to add a class folder. The Select Folder dialog box opens.
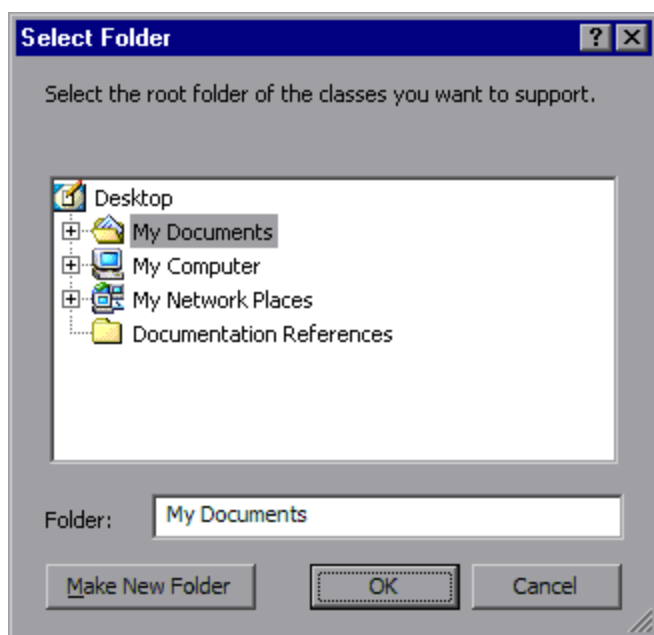
Browse to the appropriate folder, select it, and click **OK** to add it to the **Custom toolkit class locations** box.

> **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**.

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.
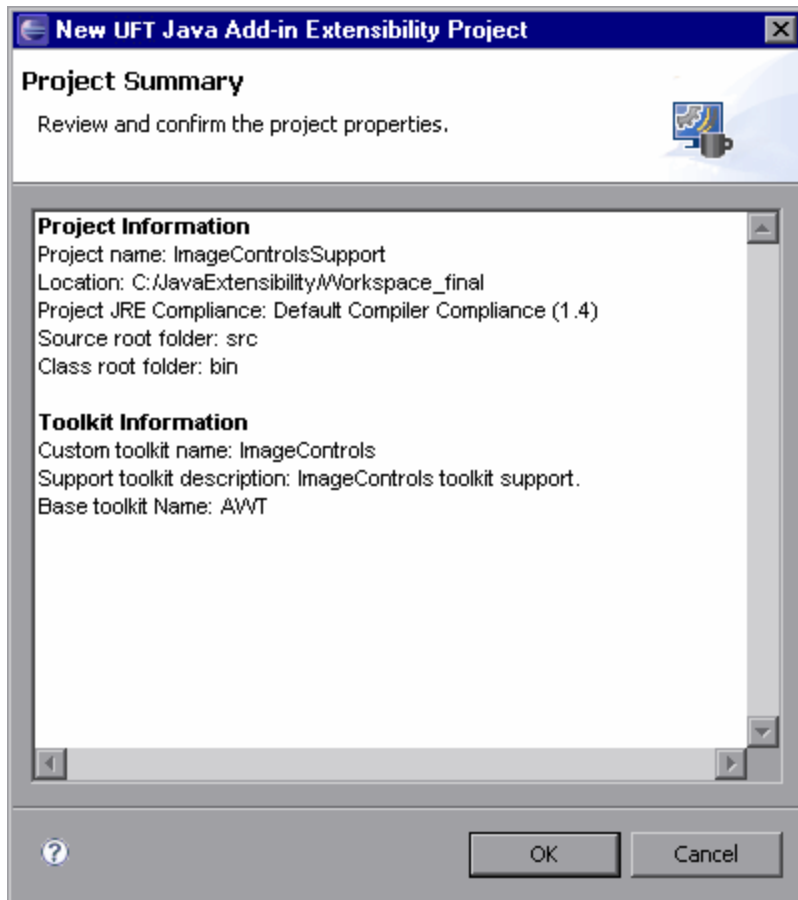
Review the information. If you want to change any of the data, click **Cancel** to return to the "Custom Toolkit Details Screen" on page 103 (described on page 103). Use the **Back** and **Next** buttons to open the relevant screens and make the required changes.

If you are satisfied with the definitions, click **OK**. The wizard creates new UFT 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 41.

  - 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 42.

  - 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 UFT Java Add-in Extensibility Project Properties

In the Eclipse menu bar, select **Project > Properties**. The Properties dialog box opens. In the left pane, select **UFT Support** from the list of property types. (The items in this list may vary, depending on the version of Eclipse that you are using.) The **UFT Support** properties are displayed in the right pane.
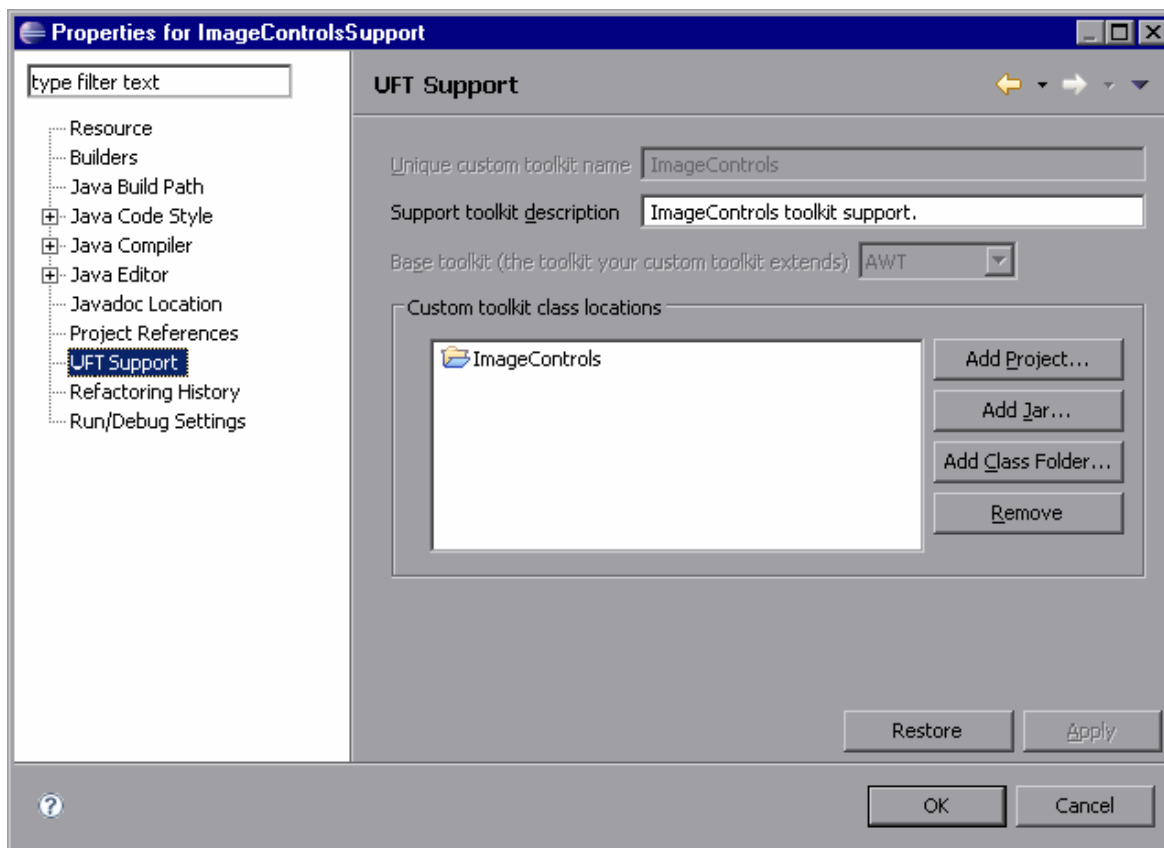
For information on the options in this dialog box, see "Custom Toolkit Details Screen" on page 103.

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 UFT Custom Support Class Wizard

You use the New UFT 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 UFT 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 UFT Custom Support Class wizard in Eclipse:

1. In the Eclipse Package Explorer tab, select a UFT Java Add-in Extensibility project.
   Then select **File > New > Other**. The New dialog box opens.
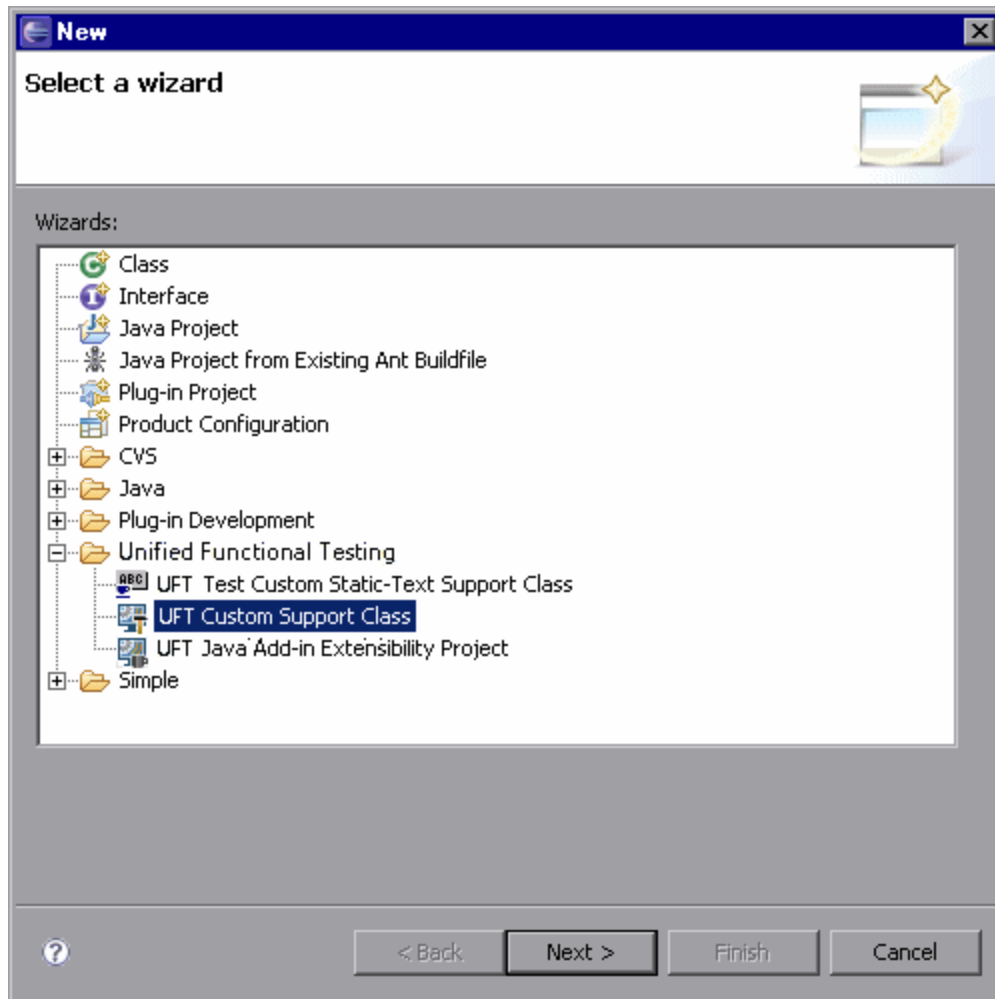
2. Expand the **Unified Functional Testing** folder and select **UFT Custom Support
   Class**.

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

> **Tip:** You can shorten this process by customizing Eclipse to provide **UFT**

Custom Support Class as an option in the **New** menu. To do this, perform the following: Select **Window > Customize Perspective**. In the Shortcuts tab in the dialog box that opens, select the **Unified Functional Testing** and **UFT Custom Support Class** check boxes. Click **OK**.

Custom Class Selection Screen

The Custom Class Selection screen is the first screen in the New UFT 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.

When you click **Next**, the Test Object Class Selection Screen described on page 117 opens.

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.
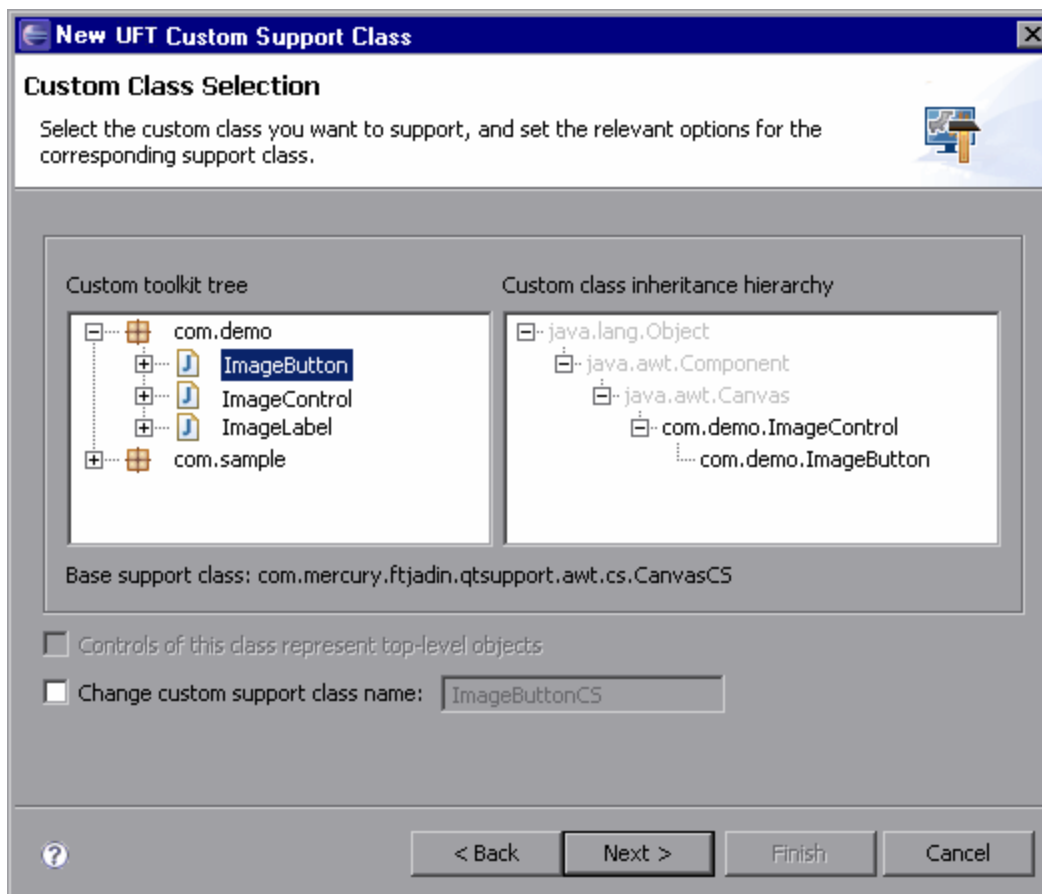
The main area of this screen contains the following options:

- **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 UFT 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 UFT support has not yet been extended. If support for a custom class was previously deployed to UFT, 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 149).
>
> 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 UFT 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 UFT support. (If support for a custom class was previously deployed to UFT, or if support for a custom class is being developed in the current Eclipse project, the wizard recognizes the custom class as having UFT support.)

  When UFT 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 UFT 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 UFT 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 103. For information on adding a custom class to an existing support project, see " Modifying UFT Java Add-in Extensibility Project Properties" on page 109.

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 on page 116.

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 UFT 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 68.

  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 UFT Custom Static-Text Support Class Wizard" on page 142 (described on page 142).

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 UFT Java Add-in Extensibility project.

2. Deploy the support to UFT.

3. Reopen the original UFT Java Add-in Extensibility project. Select **UFT > Reload Support Configuration** or click the **Reload Support Configuration** button.

4. Open the "New UFT Custom Support Class Wizard" on page 110 (described on page 110). The wizard now selects the new support class you created as the **Base support class**.

Test Object Class Selection Screen

In the Test Object Class Selection screen, you map the custom class to a test object class. In UFT GUI tests, the custom class controls are represented by test objects of the selected 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 UFT what test object class is mapped to the custom class.

When you click **Next**, the Custom Support Test Object Identification Properties Screen described on page 120 opens.

> **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.
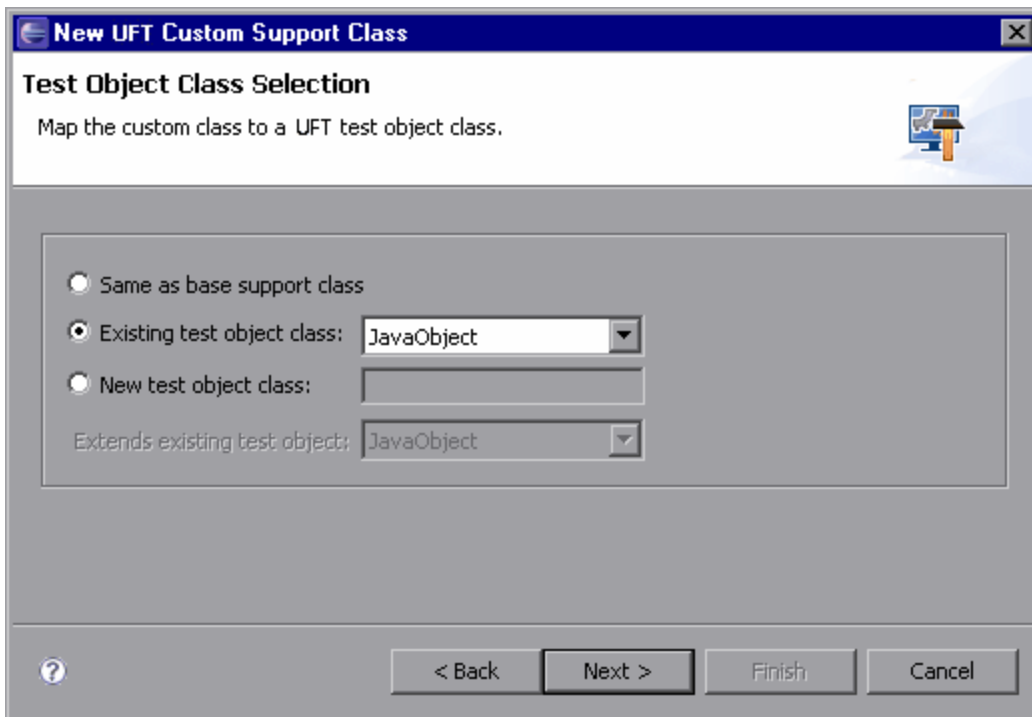
In this wizard screen you select one of the following options:

- **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 112), 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 224.

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

  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 UFT and you reload the configuration (for
  more information, see "Reload Support Configuration" on page 149).

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

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

  > **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.

  If you select this option, the **Extends existing test object** option is enabled.

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. This list is identical to the list of existing test object classes provided for the **Existing test object class** option.

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 137). 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, see the *HP UFT Test Object Schema Help*(available with the Java Add-in Extensibility SDK Help).

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.

When you click **Next**, the Custom Support Test Object Methods Screen described on page 124 opens:
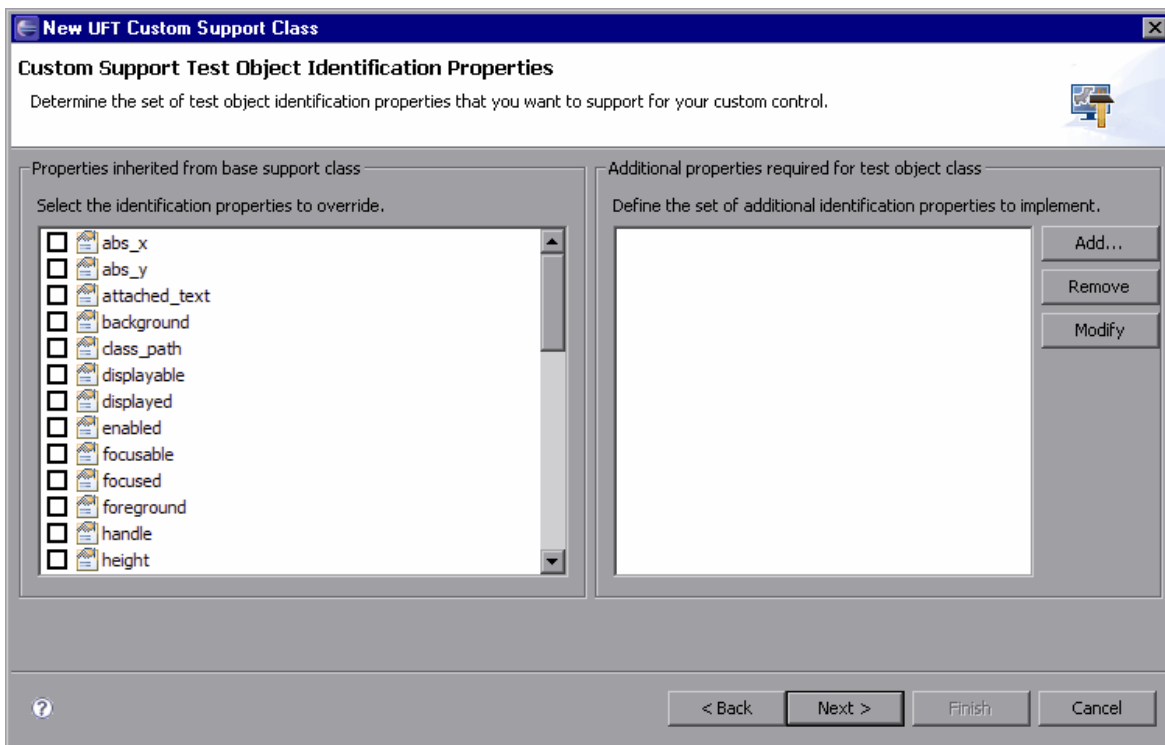
Properties Inherited from Base Support Class

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 UFT 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, see the *HP UFT Object Model Reference for GUI Testing*.

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

> **Note:** If you selected the **Same as base support class** option in the Test Object
> Class Selection Screen (on page 117), 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.

Considerations for Adding and Removing Properties from the List

- 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 UFT 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.

- 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 UFT Object Spy, it will have no value.

- Modifying an identification property is equivalent to removing it and adding a new one.

Managing the List of Identification Properties

The procedures below describe how to add, remove, and modify identification properties in the list of additional properties required for the test object class.

To add an identification property:

1. In the **Additional properties required for test object class** pane, click **Add**. The Identification Property dialog box opens.
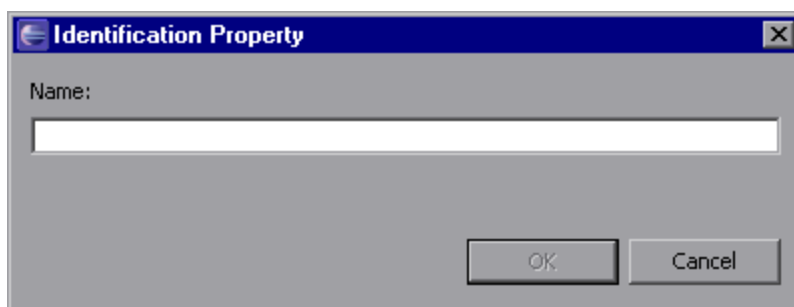
2. Enter a name for the new identification property and click **OK**. (The identification property name must begin with a letter and contain only alphanumeric characters and underscores.)

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.

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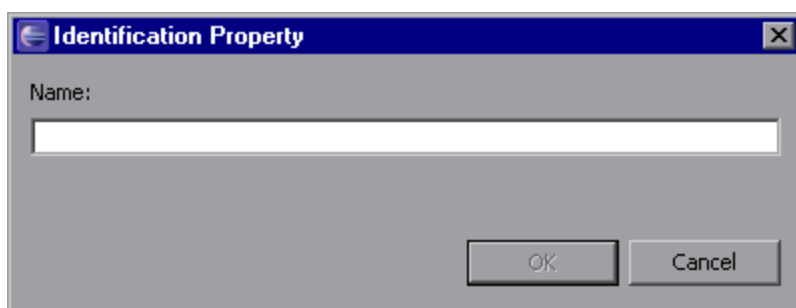
3. Modify the identification property name and click **OK**.

> **Tip:** To add identification properties after the support class is created, use the **Add Identification Property** ▣ button or select **UFT > Add Identification Property** in Eclipse.

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.

When you 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 132) 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 137) opens.

- If neither of the previous conditions is met, the Custom Control Support Class Summary Screen (described on page 140) opens.
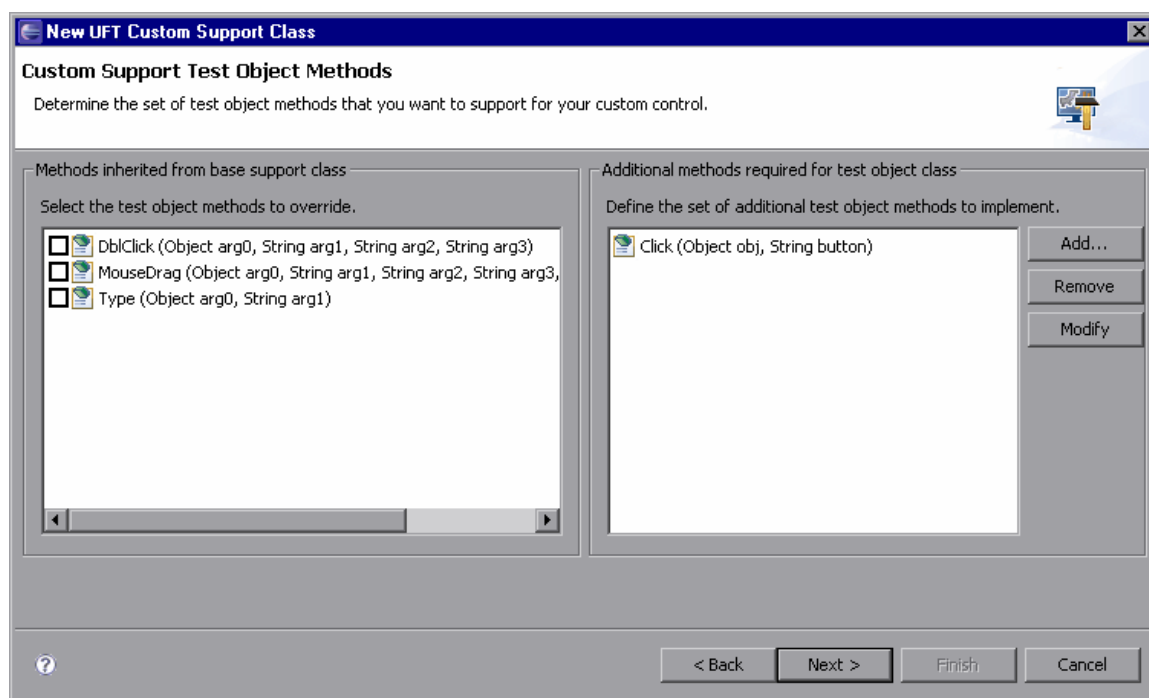
Methods Inherited from Base Support Class

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 117), 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.

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 the next page.

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

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

> **Note:** If you selected the **Same as base support class** option in the "Test Object Class Selection Screen" (on page 117), 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.

Considerations for Adding and Removing Test Object Methods to the List

- 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 UFT for all test objects of this class, regardless of whether or not they are supported for these objects. In a UFT GUI 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 remove a test object method from this 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 UFT.

  If you use this test object method on a custom control in UFTGUI 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.

Understanding the Test Object Method Dialog Box

When you click **Add** or **Modify** in the Custom Support Test Object Methods Screen"Custom Support Test Object Methods Screen" (described on page 124), 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.
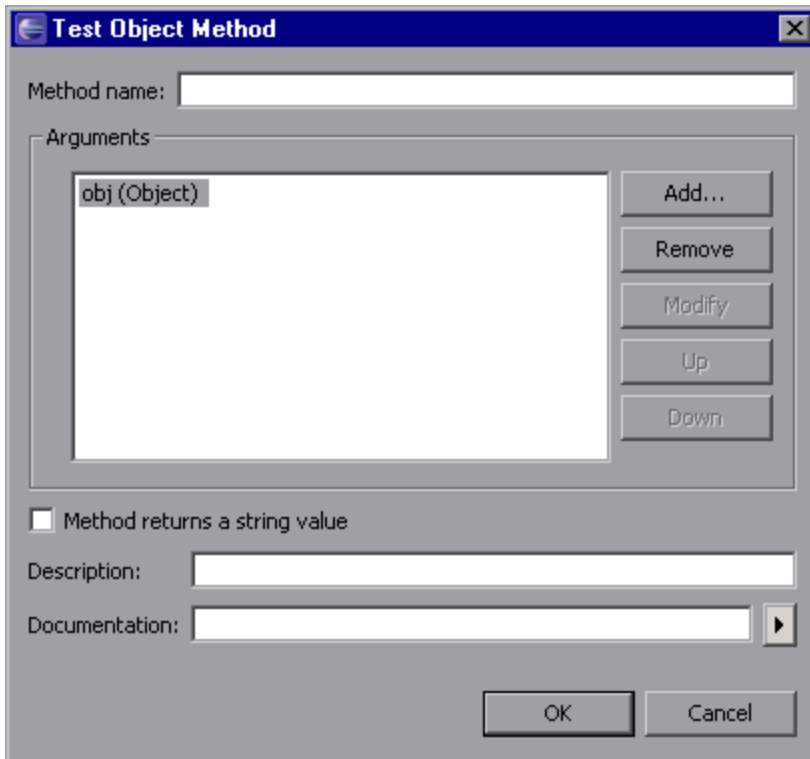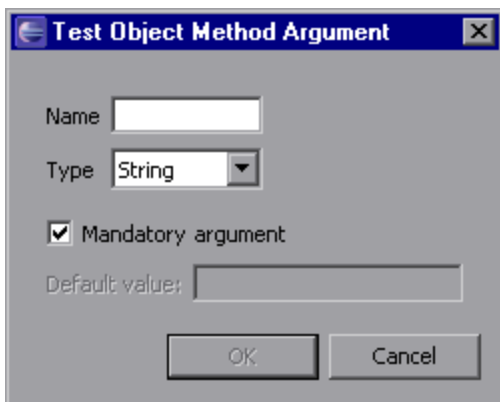
The Test Object Method dialog box contains the following items:

Option	Description
Method name	The name of the test object method as it appears in UFT GUI 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: ! @ # $ % ^ & * ( ) + = [ ] \ { } | ; ': " , / < > ? **Note:** • Modifying the name of a method is equivalent to removing the method and adding a new one.
Arguments	A list of the test object method arguments and their types. Use the following buttons to modify the list: • **Remove.** Removes the selected argument from the list. • **Up.** Moves the selected argument up in the list. • **Down.** Moves the selected argument down in the list. **Note:** • The first argument of every test object method must be **obj (Object)**. You cannot remove, modify, or move this argument.

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 UFT GUI 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 the method succeeds, return **OK** and an appropriate string value. Otherwise, return only the relevant error code. For more information, see the *UFT Java Add-in Extensibility API Reference* (available with the Java Add-in Extensibility SDK Help).
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 the statement completion functionality in the Editor.
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 UFT GUI 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 UFT 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, see the *HP UFT Test Object Schema Help* (available with the Java Add-in Extensibility SDK Help).
Mandatory argument	Instructs UFT whether 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 UFT 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 does not open if you are creating a support class for an SWT-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.

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 on how the wizard implements the details you specify in this screen, see "Understanding What the Wizard Adds to the Support Class" on page 136.

After you complete the recording-related support information:

- 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 on page 137).

- Otherwise, click **Finish** to continue to the "Custom Control Support Class Summary Screen" (described on page 140).
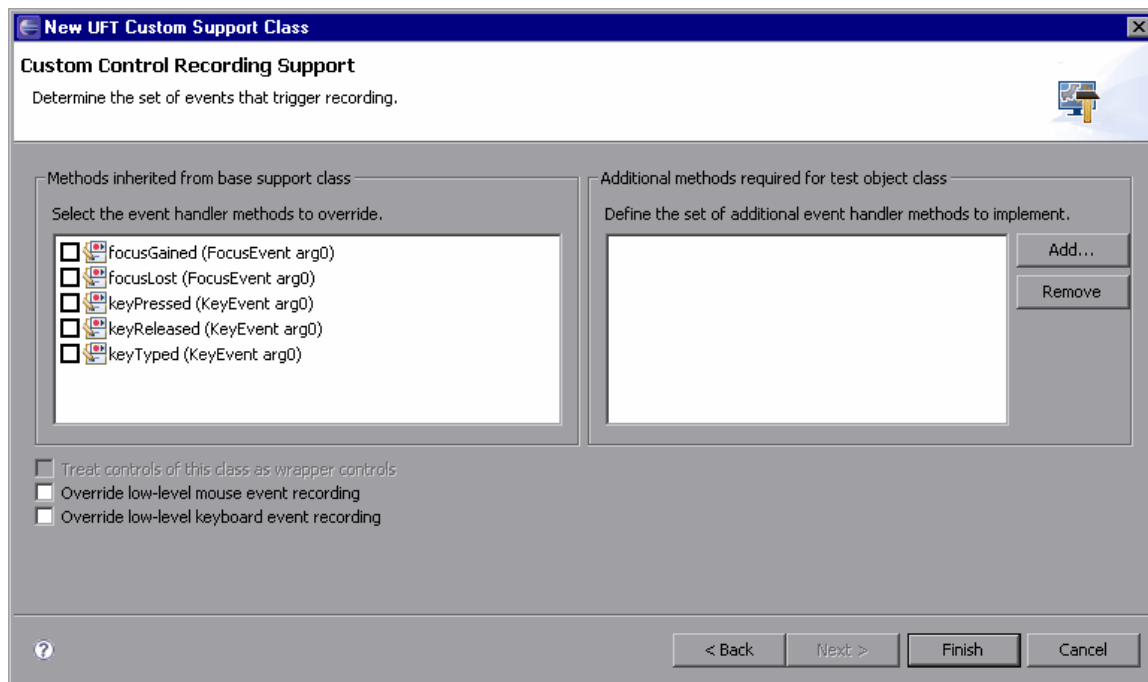
Methods Inherited from Base Support Class

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.
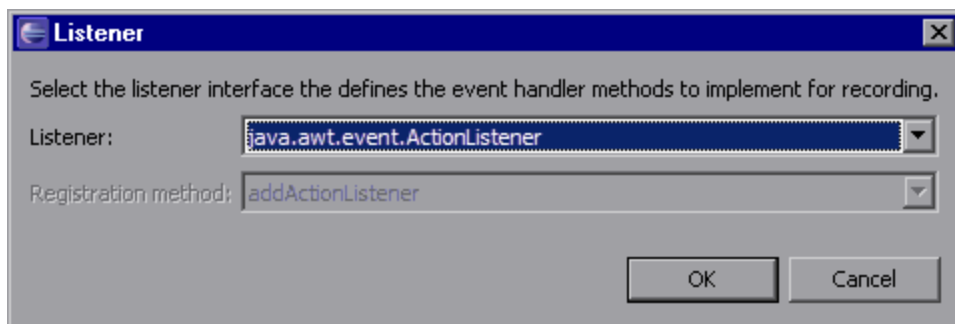
   The list contains the listeners that can be registered on the custom control. The wizard compiles this list by identifying listener registration methods in the custom class and its superclasses. The wizard identifies as registration methods, only methods named **add<XXX>Listener** whose first argument 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**.

> **Tip:** To add event handlers after the support class is created, use the **Add Event Handler** ⌨ button or select **UFT > Add Event Handler** in Eclipse.

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 137.
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 UFT 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 UFT not to record low-level keyboard events, enabling you to record more complex events, such as, setting a value in an edit box.

The options listed in the table above are available only in the wizard (and not in the Eclipse UFT 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.

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 Custom Control Recording Support 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.

  Some of the event handler methods are implemented in existing support classes as final methods, which cannot be overridden. 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 65.

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 69.

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.

When you click **Finish**, the "Custom Control Support Class Summary Screen" described on page 140 opens.
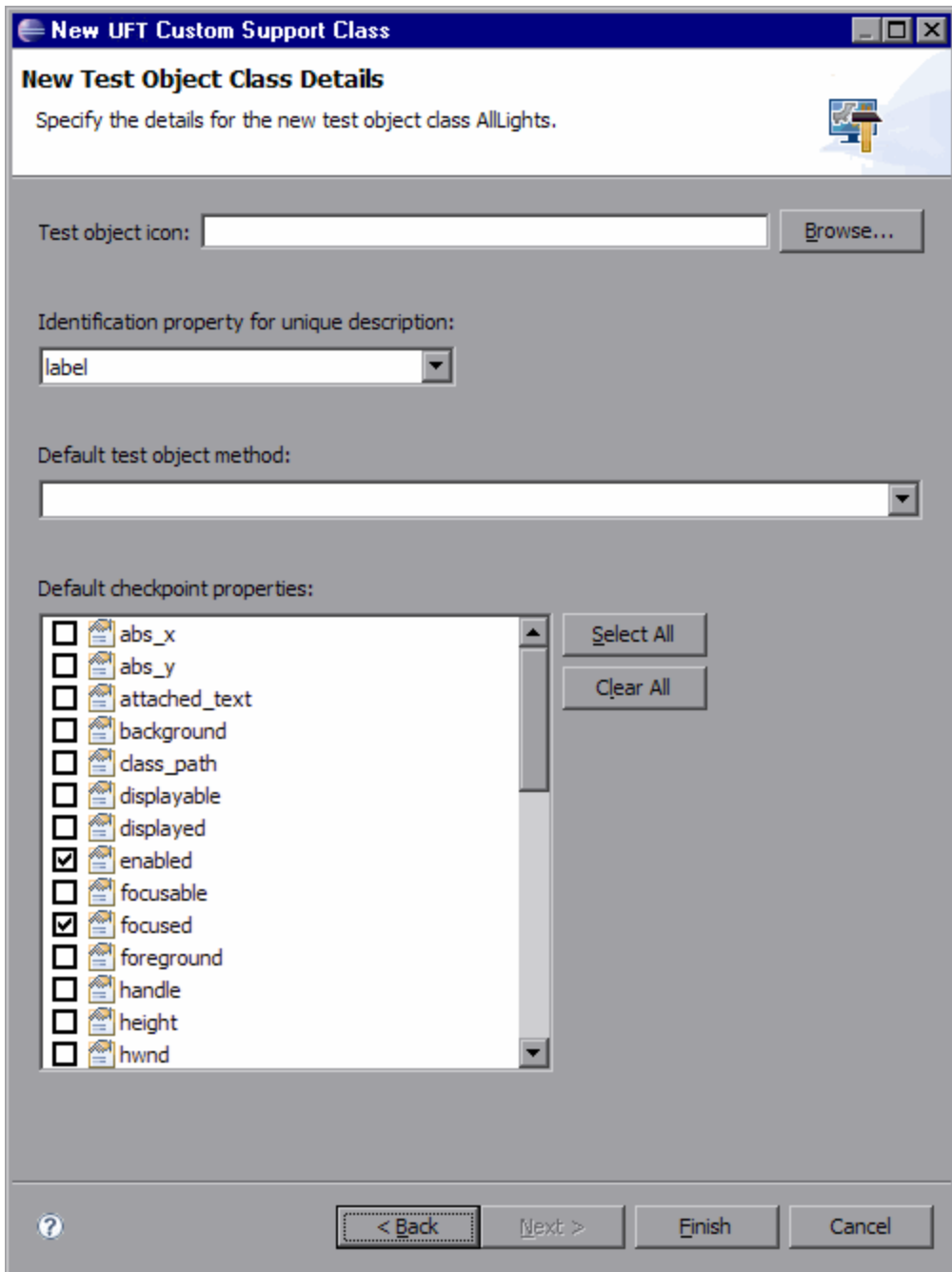
The New Test Object Class Details screen contains the following options:

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 UFT 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 on the structure of this file, see the *HP UFT Test Object Schema Help* (available with the Java Add-in Extensibility SDK Help).

If you want UFT 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"`.

The list of identification properties in the test object description can be modified in UFT using the Object Identification dialog box. Therefore, by default, UFT reads this information from the test object configuration file only once, to prevent overwriting any changes a user makes in UFT. For information on how to ensure UFT reads modifications you make to the `ForDescription` attribute, see "Modifying Identification Property Attributes in a Test Object Configuration File" on page 83.

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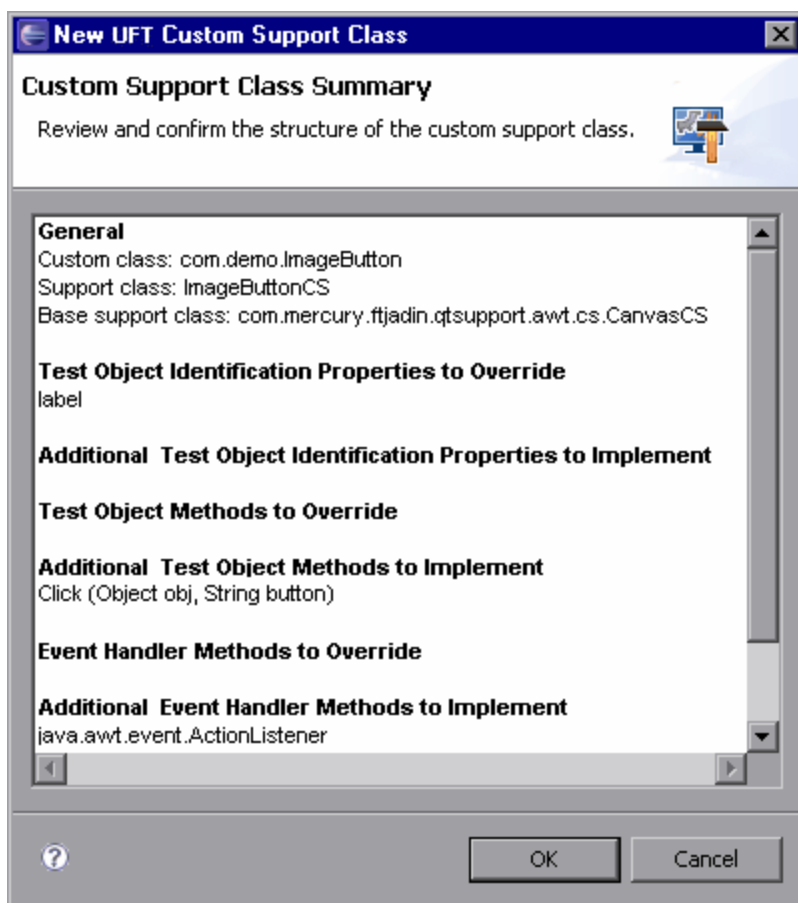
If you want to change any of the data, click **Cancel** to return to the previous wizard screen. Use the **Back** and **Next** buttons to open the relevant screens and make the required changes.

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

Completing the Custom Class Support

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

- Save the class.

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

- 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 41.

  If you added new test object methods or identification properties to the test object class, the wizard adds them to the test object class definition in the test object configuration file.

  If you remove (or do not implement) the support methods that the wizard created in the support class, the test object methods or identification properties remain part of the test object class definition. They are available in UFT when editing tests but are not supported for this custom class.

- Deploy the toolkit support to UFT to enable the support to be available. For more information, see "Deploying and Running the Custom Toolkit Support" on page 77.

New UFT Custom Static-Text Support Class Wizard

You use the New UFT 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 UFT 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 145.

After the wizard creates the new support class, you complete its implementation as described in "Completing the Custom Static-Text Class Support" on page 147.

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 UFT Commands in Eclipse" on page 147.

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

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

2. Expand the **Unified Functional Testing** folder and select **UFT 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 **UFT Custom Static-Text Support Class** as an option in the **New** menu. To do this, perform the following: Select **Window > Customize Perspective**. In the Shortcuts tab in the dialog box that opens, select the **Unified Functional Testing** and **UFT 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 112). When you click **Finish**, the "Custom Static-Text Support Class Summary Screen" (described on page 145) opens.
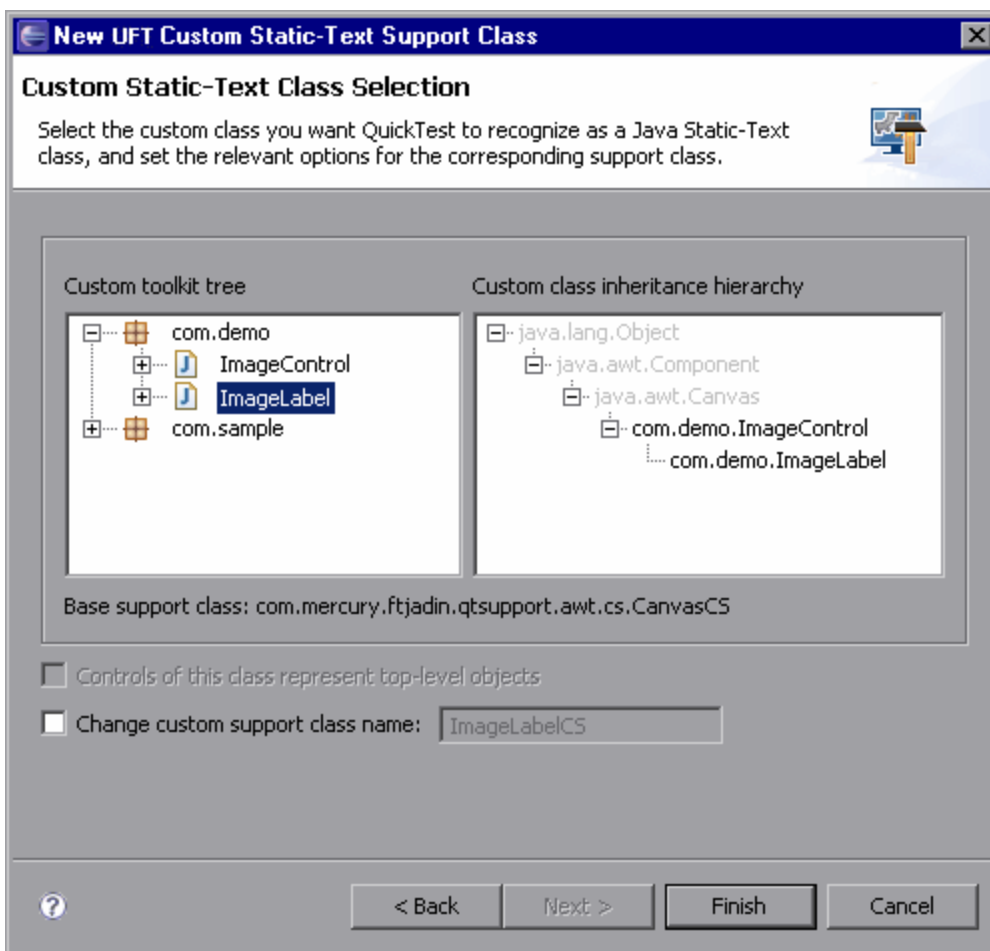
Select the custom class you want UFT to recognize as static-text and set the relevant options.

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

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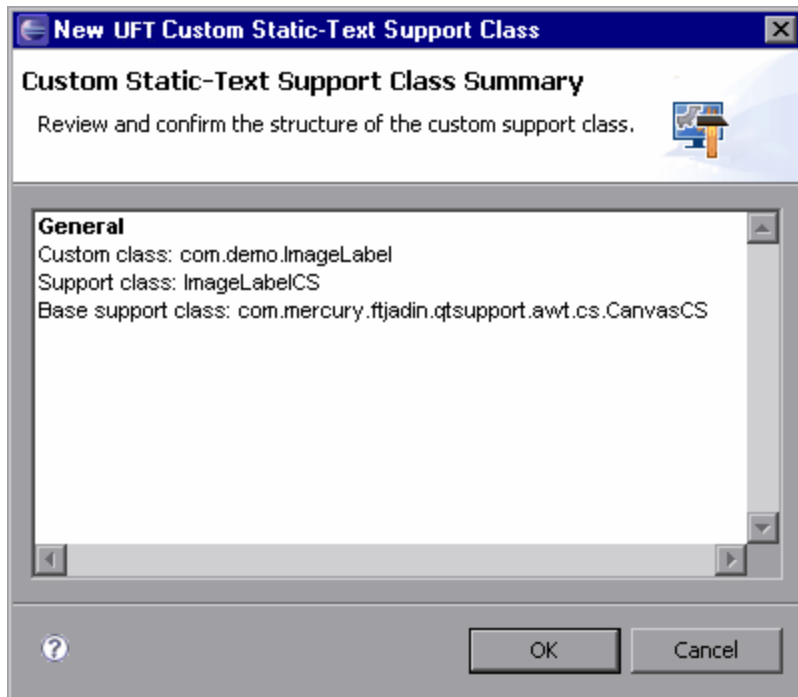
If you want to change any of the data, click **Cancel** to return to the "Custom Static-Text Class Selection Screen", described above.

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 UFT 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, UFT 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 "Common Identification Property Support Methods" on page 61.

- **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 205.

Completing the Custom Static-Text Class Support

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

- Save the class.

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

- Implement the **label_attr** method, if needed.

- Deploy the toolkit support to UFT to enable the support to be available. For more information, see "Deploying and Running the Custom Toolkit Support" on page 77.

Working with UFT Commands in Eclipse

After you install the UFT 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 **UFT** 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 in Eclipse when you select a UFT 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 UFT Java Add-in Extensibility SDK before installing UFT and the Java Add-in. To solve this problem, uninstall the UFT Java Add-in Extensibility SDK and install it again. For more information, see "Installing the HP UFT Java Add-in Extensibility Software Development Kit" on page 24.

You use the **Deploy Toolkit Support** command to deploy the toolkit support during the development stages. To use this command, UFT and the UFT 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 UFT folders. In the toolkit configuration file, the location specified for the compiled support classes is the Eclipse workspace. The next time you run the Java application, the support you developed is available on UFT. For more information, see "Deploying and Running the Custom Toolkit Support" on page 77.

> **Note:** The deploy command compiles the Java classes before deploying, but does not validate compilation results. Save the support classes before deploying and check for compilation errors, to avoid run-time failure.

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 77.

The **Deploy Toolkit Support** command validates the configuration files against their corresponding XSD files, and only deploys them if their format meets the requirements (or you specify that you want to deploy in spite of the displayed discrepancies). For information on the structure of the configuration files, see the *UFT Java Add-in*

Extensibility Toolkit Configuration Schema Help and the *HP UFT Test Object Schema Help* (both available with the Java Add-in Extensibility SDK Help).

The toolkit configuration file is validated against:
<UFT installation folder>\bin\java\sdk\eclipse\plugins\ com.mercury.qtjext.plugin.QTJavaExt_1.0.0\ToolkitSchema.xsd

The test object configuration file is validated against:
<UFT installation folder>\bin\java\sdk\eclipse\plugins\ com.mercury.qtjext.plugin.QTJavaExt_1.0.0\ClassesDefintions.xsd

Reload Support Configuration

The **Reload Support Configuration** [icon] command is available in Eclipse when you select a UFT 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 UFT Java Add-in Extensibility SDK before installing UFT and the Java Add-in. To solve this problem, uninstall the UFT Java Add-in Extensibility SDK and install it again. For more information, see "Installing the HP UFT Java Add-in Extensibility Software Development Kit" on page 24.

The **Reload Support Configuration** command instructs the UFT 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 UFT installation folder

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

- The list of custom classes displayed in the custom toolkit tree in the "Custom Class Selection Screen" (described on page 112).

- The wizard's selection of the base support class in the "Custom Class Selection Screen" (described on page 112).

- The list of existing test object classes displayed in the "Test Object Class Selection Screen" (described on page 117).

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

- You modified the test object configuration file in the UFT 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 UFT, and you want the wizard to recognize that the classes are supported.

- You manually deleted support for some classes from UFT, 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 in Eclipse when you select a UFT 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 UFT 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 UFT, 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 UFT 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.

You cannot remove support of a complete toolkit using the UFT 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 UFT folders. For more information, see "Deploying and Running the Custom Toolkit Support" on page 77.

Add Identification Property

The **Add Identification Property** command is available in Eclipse when you select a UFT 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 UFT 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 UFT.

If you add an identification property to the test object class definition, it appears in the list of identification properties in UFT 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 modifying an existing test object class.

To add an identification property:

1. Click the **Add Identification Property** ![icon] button in the UFT toolbar in Eclipse. The Identification Property dialog box opens.
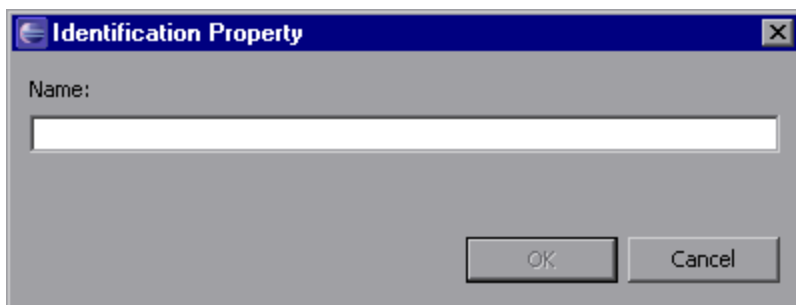
2. Enter a name for the new identification property and click **OK**.

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 UFT 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 definition 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.)

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

Add Test Object Method

The **Add Test Object Method** command is available in Eclipse when you select a UFT 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 UFT 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 UFT.

If you add a test object method to an existing test object class, the new methods appear in UFT for all test objects of this class, regardless of whether or not they are supported for these objects. In a UFT GUI 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 modifying an existing test object class.

To add a test object method:

1. Click the **Add Test Object Method** button in the UFT toolbar in Eclipse. 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 127.

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

Add Event Handler

The **Add Event Handler** 📇 command is available in Eclipse when you select an AWT-based UFT 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 UFT 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 65.

To add event handler methods:

1. Click the **Add Event Handler** 📇 button in the UFT toolbar in Eclipse. The Listener dialog box opens:
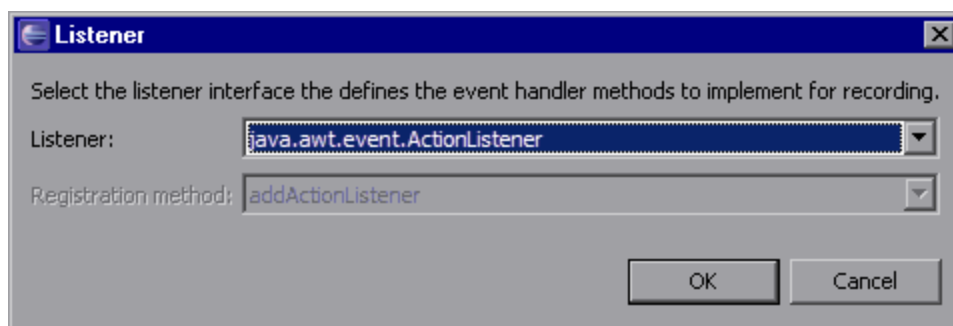
2. Select a listener from the list.

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

3. 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 UFT. For more information, see "Supporting the Record Option" on page 65.

Part 2: Tutorial: Learning to Create Java Custom Toolkit Support

Chapter 5: Using the UFT Java Add-in Extensibility Tutorial

The UFT 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 UFT support for your own custom toolkits and controls.

This chapter includes:

Understanding the Tutorial Lesson Structure

This tutorial is divided into 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 UFT Java Add-in Extensibility. It is recommended to perform the lessons in order.

In each lesson in this tutorial, you extend UFT support for a different custom control, using the UFT 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. If you deploy the sample custom toolkit support sets manually, you must compile the Java classes before deploying.

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 UFT Java Add-in Extensibility Eclipse Plug-In" on page 97.

About Learning to Support a Simple Control

The lesson, "Learning to Support a Simple Control" on page 163, 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 UFT Java Add-in Extensibility Eclipse plug-in: the New UFT Java Add-in Extensibility Project wizard and the New UFT Custom Support Class wizard.

About Learning to Support a Custom Static-Text Control

The lesson, "Learning to Support a Custom Static-Text Control" on page 205, 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 UFT Custom Static-Text Support Class wizard provided by the UFT Java Add-in Extensibility Eclipse plug-in.

About Learning to Support a Complex Control

The lesson, "Learning to Support a Complex Control" on page 236, 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.

Checking Tutorial Prerequisites

Before you begin to perform the lessons in this tutorial, make sure that the requirements described in this section are met.

System Requirements

You must have the following items installed on a computer on which support has not yet been implemented for the custom toolkits and controls in this tutorial. If such support has already been implemented, remove the support as described in "Deploying and Running the Custom Toolkit Support" on page 77.

Eclipse

For a list of supported Eclipse versions, see the *HP Unified Functional Testing Product Availability Matrix*, available from the UFT help folder or the HP Support Matrix page

(requires an HP passport).

Java Add-in Extensibility SDK

For information on installing Eclipse or the Java Add-in Extensibility SDK see "Installing the HP UFT Java Add-in Extensibility Software Development Kit" on page 24.

Unified Functional Testing, including the Java Add-in

For information on installing Unified Functional Testing (UFT), see the *HP Unified Functional Testing Installation Guide*.

If your UFT 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 UFT, you must manually transfer the custom support class files and configuration files to the correct folders on the UFT computer as described in "Deploying and Running the Custom Toolkit Support" on page 77.

Knowledge Requirements

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

Familiarity with major UFT features and functionality

You should have a thorough understanding of the following: test objects, object repository, Object Spy, Keyword View, and Editor. You should also have experience recording, editing, and running tests. For more information, see the *HP Unified Functional Testing User 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 34).

Chapter 6: 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 "Using the UFT Java Add-in Extensibility Tutorial" on page 158.

This lesson guides you through the following stages:

Preparing for This Lesson

Before you extend UFT 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 procedure 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 **<UFT 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 .

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

1. Run Eclipse and select **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 **<UFT 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.

> **Note:** The steps for creating a new project in Eclipse may vary, depending on the Eclipse version that you use.

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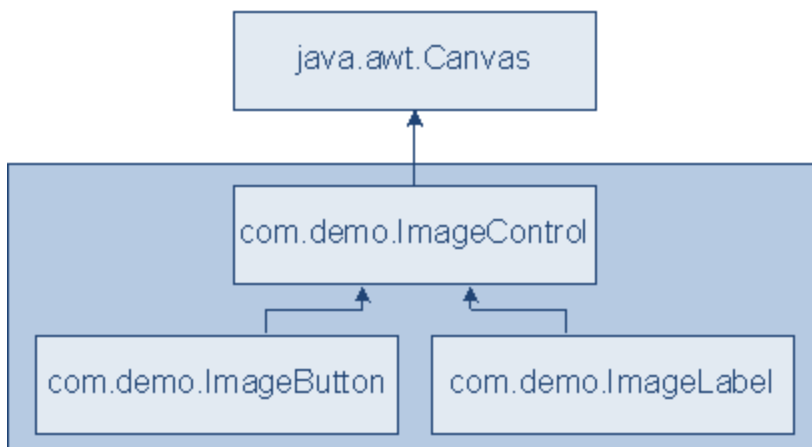
The functionality provided by the classes in this package is as follows:

- **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 the UFT Java Add-in Extensibility SDK is installed under a folder other than **C:/Program Files/HP/Unified Functional Testing**, you must modify the sample application before performing this lesson.

To modify the sample application:

1. 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.

2. Locate the code containing the image file paths:

```
C:/Program Files/HP/Unified Functional
Testing/samples/ImageControls/images/
mercury.gif
C:/Program Files/HP/Unified Functional
Testing/samples/ImageControls/images/
JavaExt1.gif
```

3. Replace `C:/Program Files/HP/Unified Functional Testing` 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 UFT support of the ImageButton control, determine the answers to the questions in the "Understanding the Custom Class Support Planning Checklist" on page 93, and fill in the "Custom Class Support Planning Checklist " on page 171, accordingly.

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

1. **Run the SampleApp application and open UFT.**

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

   b. Open UFT and load the Java Add-in.

2. **Use the Object Spy to view the ImageButton properties.**

   a. In UFT, open a GUI test and select **Tools > Object Spy** or click the **Object Spy** toolbar button to open the Object Spy dialog box. Click the **Properties** tab.

   b. 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 UFT does not recognize. Therefore, it recognizes the button as a generic **JavaObject** named **ImageButton**, and the icon shown is the standard JavaObject class icon.
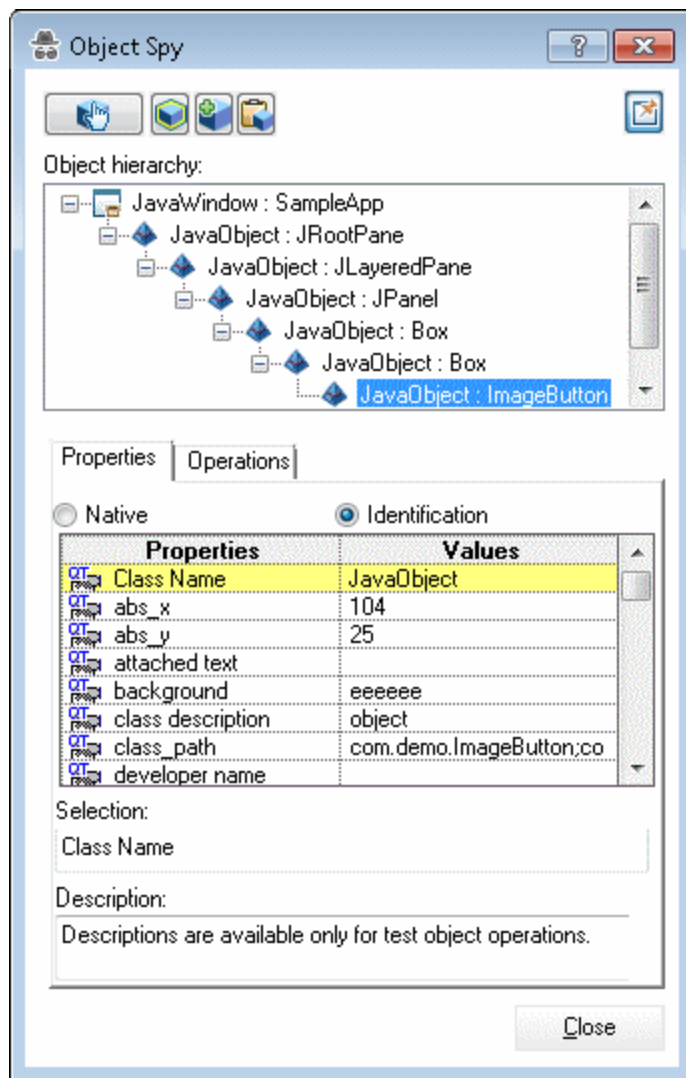
c. Close the Object Spy.

3. **Record an operation on the ImageButton control.**

a. In UFT select **Record > 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**.

b. Click the **Record** button or select **Record > 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	30,26,"LEFT"	Click the "ImageButton" object with the "LEFT" mouse button.

c. Click the **Stop** button or select **Record > Stop** to end the recording session.

The **Click** operation recorded 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 UFT 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 UFT 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 UFT also does not provide support. At this point, there does not seem to be any functionality requiring special UFT 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, UFT 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 UFT, 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 UFT custom support is not yet available? 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 UFT installed? Yes
☑	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? No
☑	Is the custom control a wrapper? 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

☑	**Custom Class Support Planning Checklist**
☑	If so, list the events that should trigger recording: `ActionEvents`

Creating a New UFT 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 UFT Java Add-in Extensibility plug-in in Eclipse.

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

   a. In Eclipse, select **File > New > Project**. The New Project dialog box opens. Expand the **Unified Functional Testing** folder and select **UFT Java Add-in Extensibility Project**.
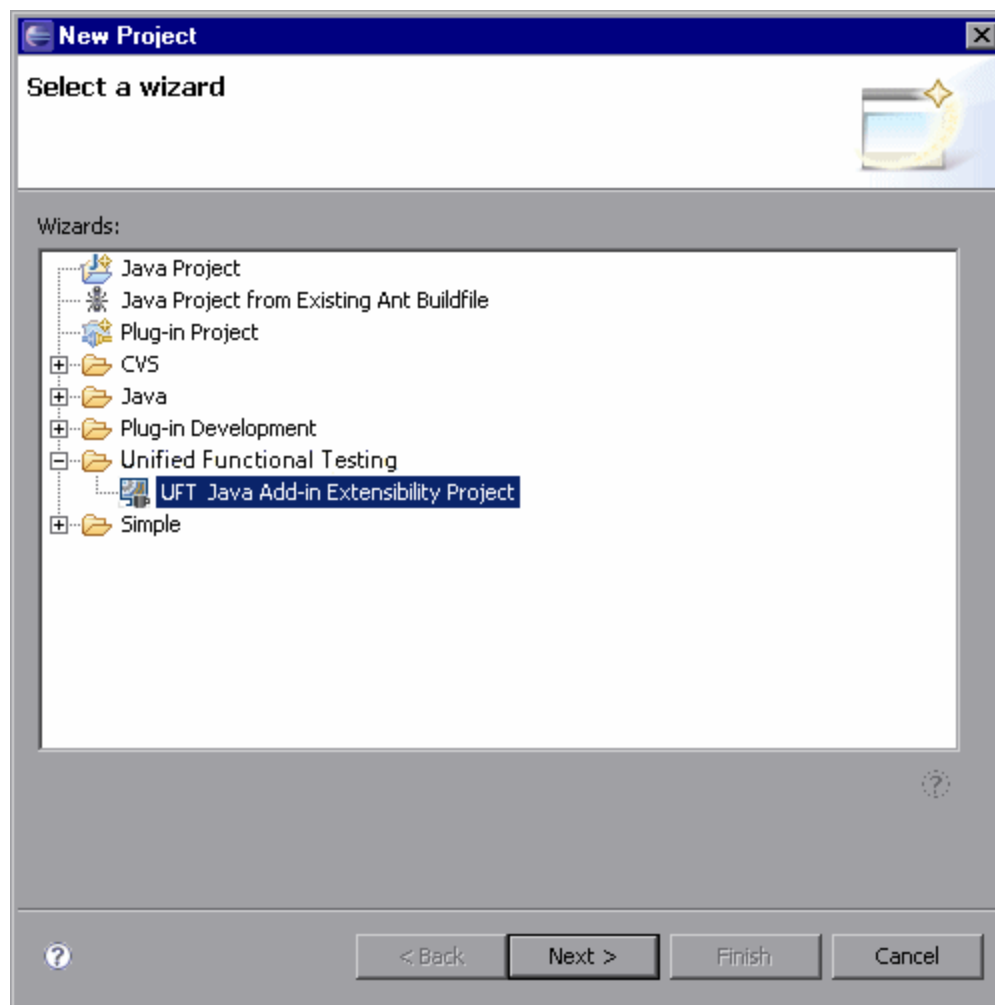
b. Click **Next**. The UFT Java Add-in Extensibility Project screen opens. The details on this screen may vary, depending on the version of Eclipse that you are using.

2. **Enter the UFT Java Add-in Extensibility project details.**

a. In the **Project name** box, enter `ImageControlsSupport`. Select **Create separate folders for sources and class files**. (In earlier Eclipse versions this option is named **Create separate source and output folders**.) For more information on this dialog box, see the *Eclipse Help*.
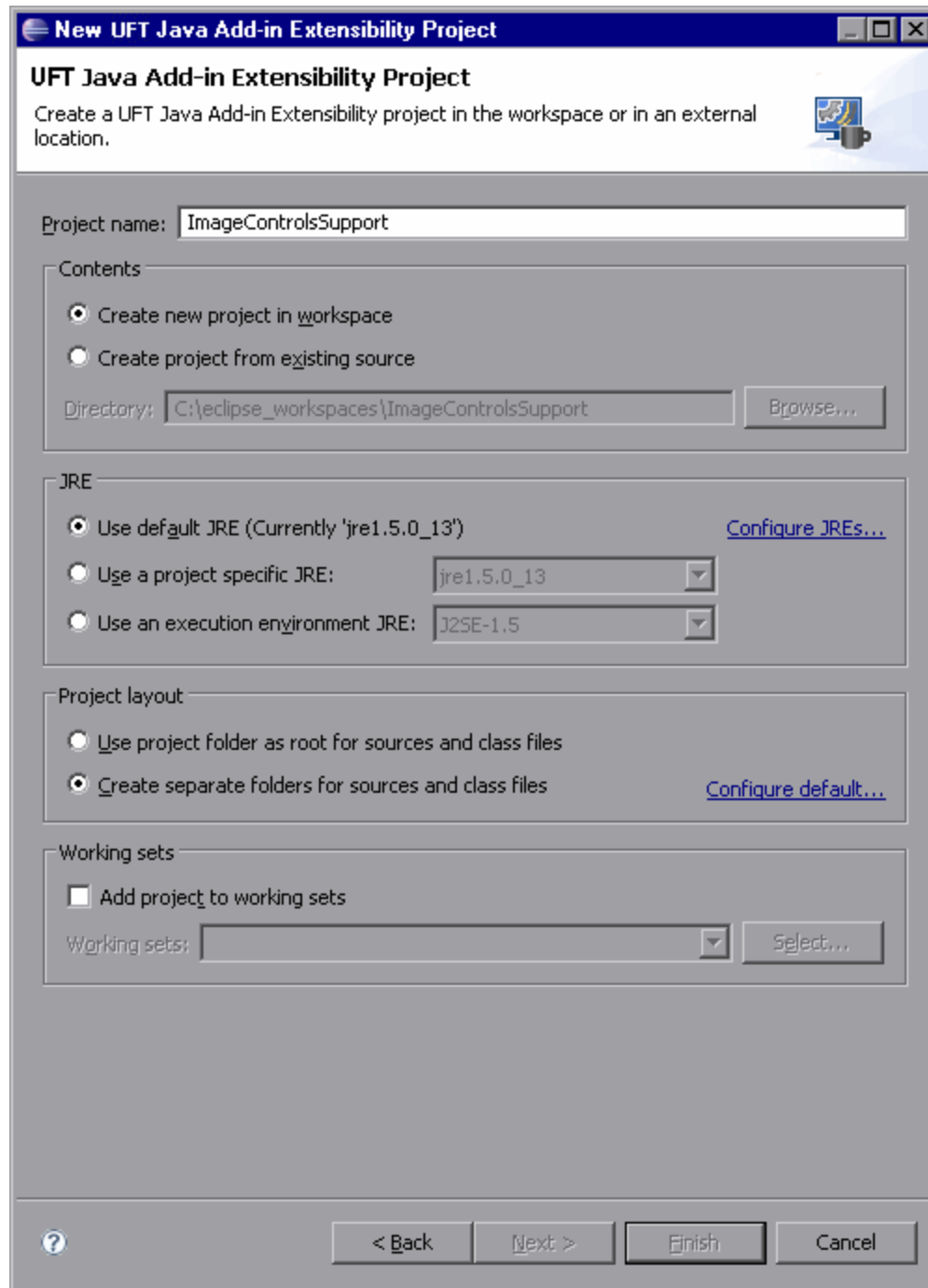
    b. Click **Next**. The Custom Toolkit Details screen opens.

  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.
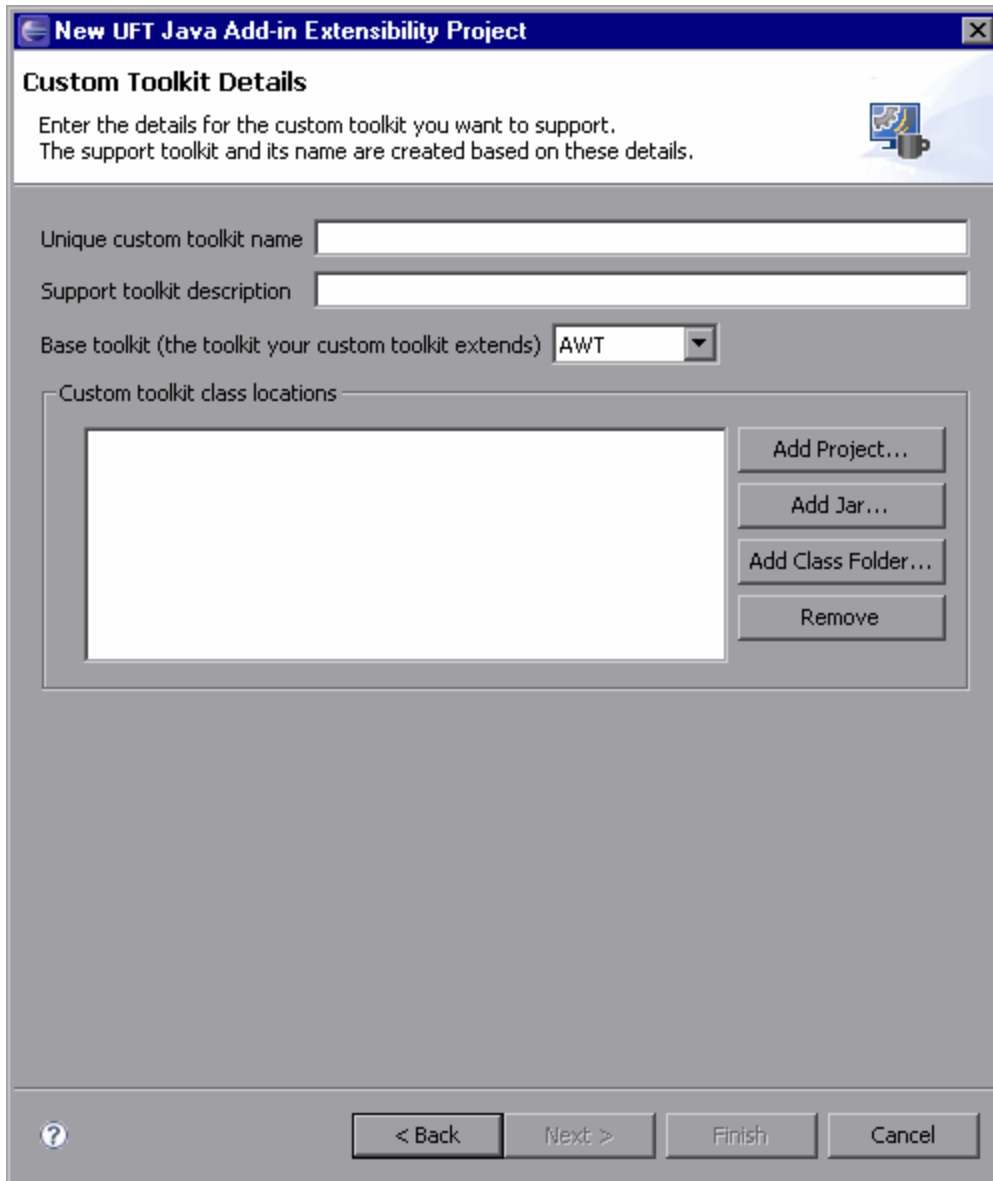
a. Define the following information:

○ In the **Unique custom toolkit name** box, 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 UFT installation to support numerous custom toolkit support sets simultaneously.

After you develop the support and deploy it to UFT, UFT displays the custom toolkit name in all of the dialog boxes that display lists of add-ins or supported environments.

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 UFT 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.
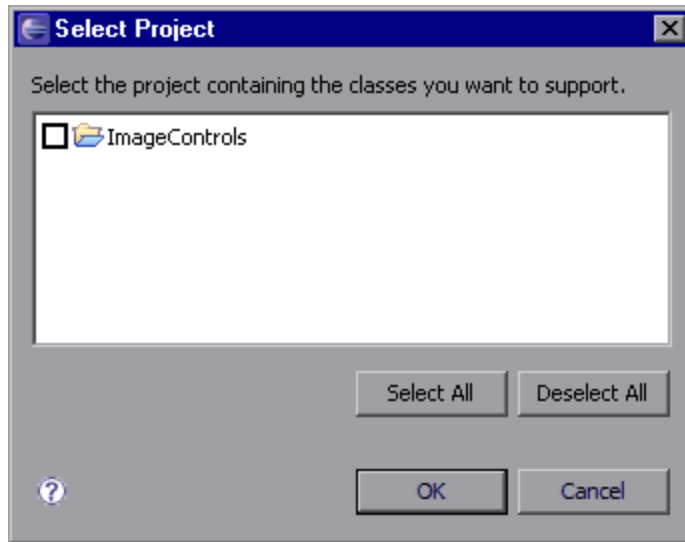
b. Select the **ImageControls** check box. Click **OK**. The ImageControls project is added in the **Custom toolkit class locations** box.

c. Click **Finish**. The Project Summary screen opens.

4. **View the Project Summary wizard screen.**

Review the details of the project and click **OK**.
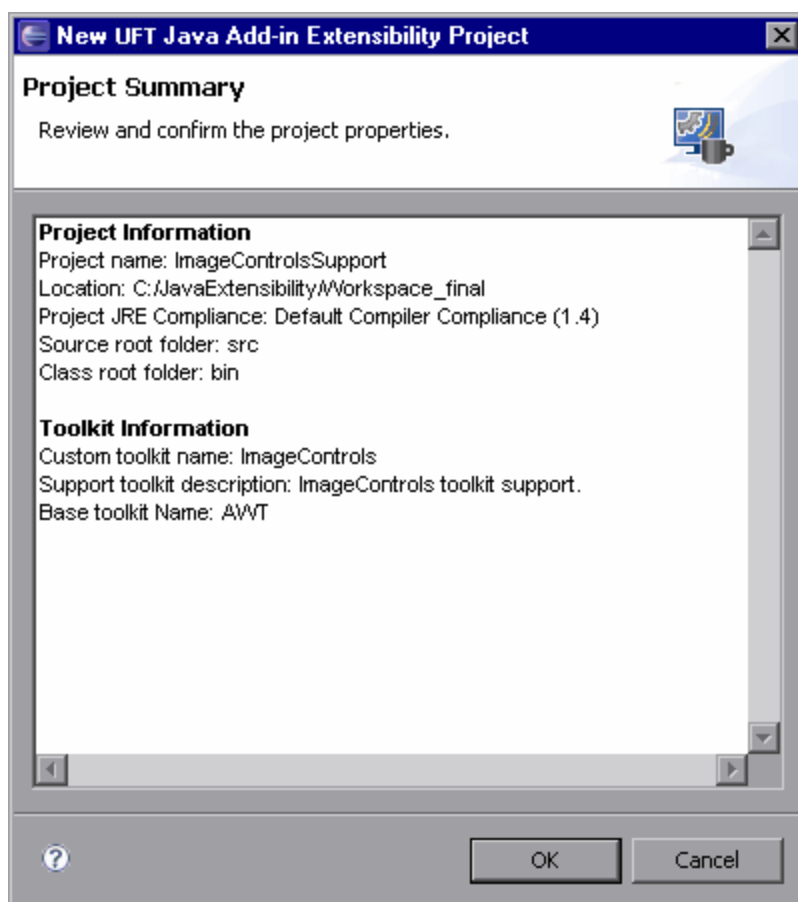
The wizard creates a new Java Add-in Extensibility project named ImageControlsSupport, containing the basic files required for custom toolkit support.

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.
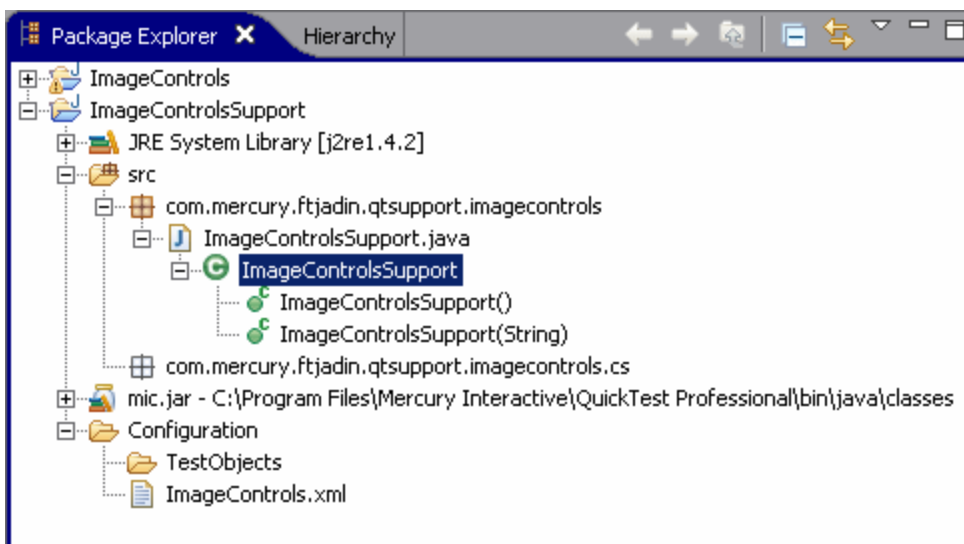
The **src** folder contains the following packages:

- **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 UFT**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 in 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:\JavaExtensibility\Workspace_final\
  ImageControlsSupport\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 UFT computer and update the values in the toolkit configuration file appropriately. For more information, see "Deploying and Running the Custom Toolkit Support" on page 77.

  For a complete understanding of the structure of this file, see the *UFT Java Add-in ExtensibilityToolkit Configuration Schema Help*(available with the Java Add-in Extensibility SDK Help).
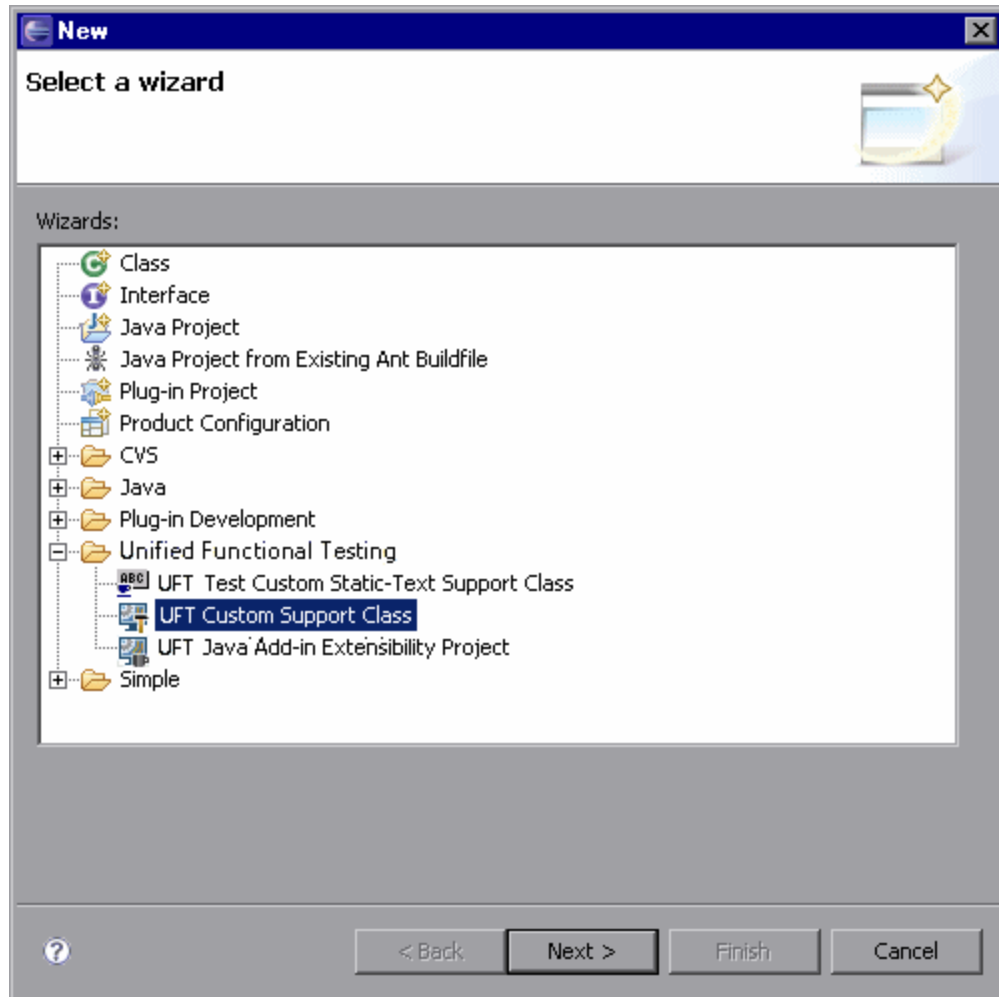
Creating a New UFT Custom Support Class

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 UFT 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 .
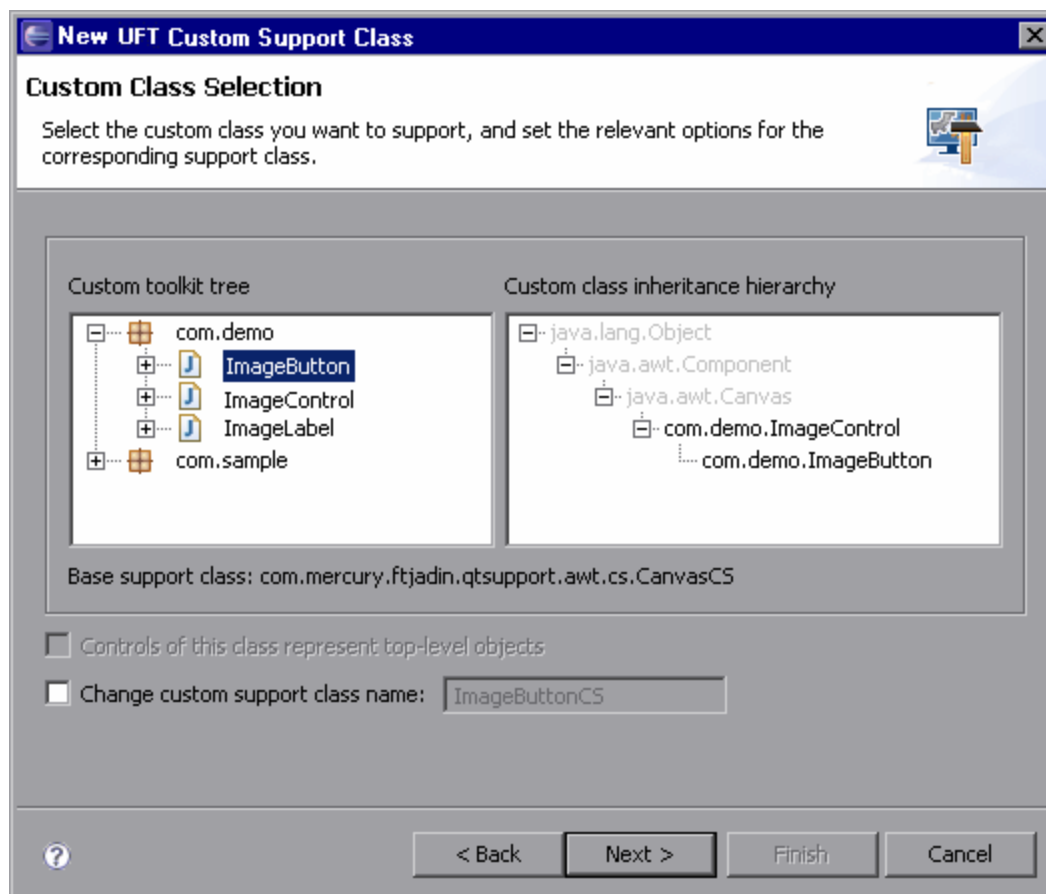
1. **Open the New UFT Custom Support Class wizard.**

    a. In the Eclipse Package Explorer tab, select the new UFT Java Add-in Extensibility project, **ImageControlsSupport**. Select **File > New > Other**. The New dialog box opens.

b. Expand the **Unified Functional Testing** folder and select **UFT Custom Support Class**.

c. Click **Next**. The Custom Class Selection screen opens.

2. **Select the custom class to support, and set the options for the support class.**

a. Expand the **com.demo** package and select the **ImageButton** class.

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 "Creating a New UFT Java Add-in Extensibility Project" on page 172. The **com.demo** package contains the ImageControls custom toolkit, with its custom classes, as described in "Preparing for This Lesson" on page 164.

> **Note:** The **com.sample** 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 hierarchy** 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.

b. 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 UFT 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.
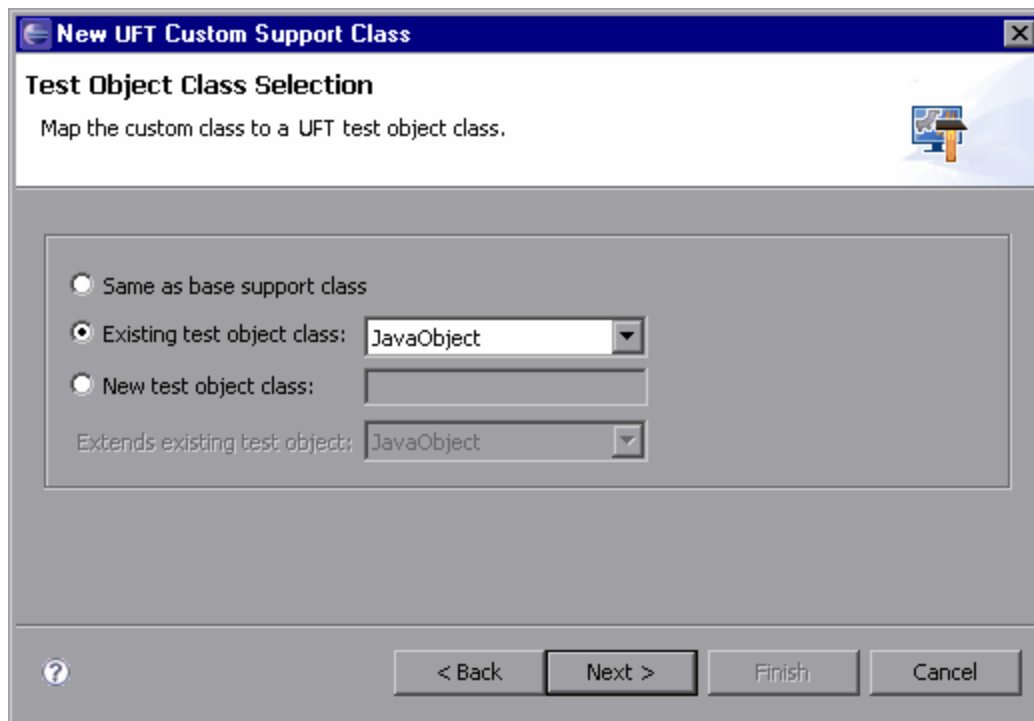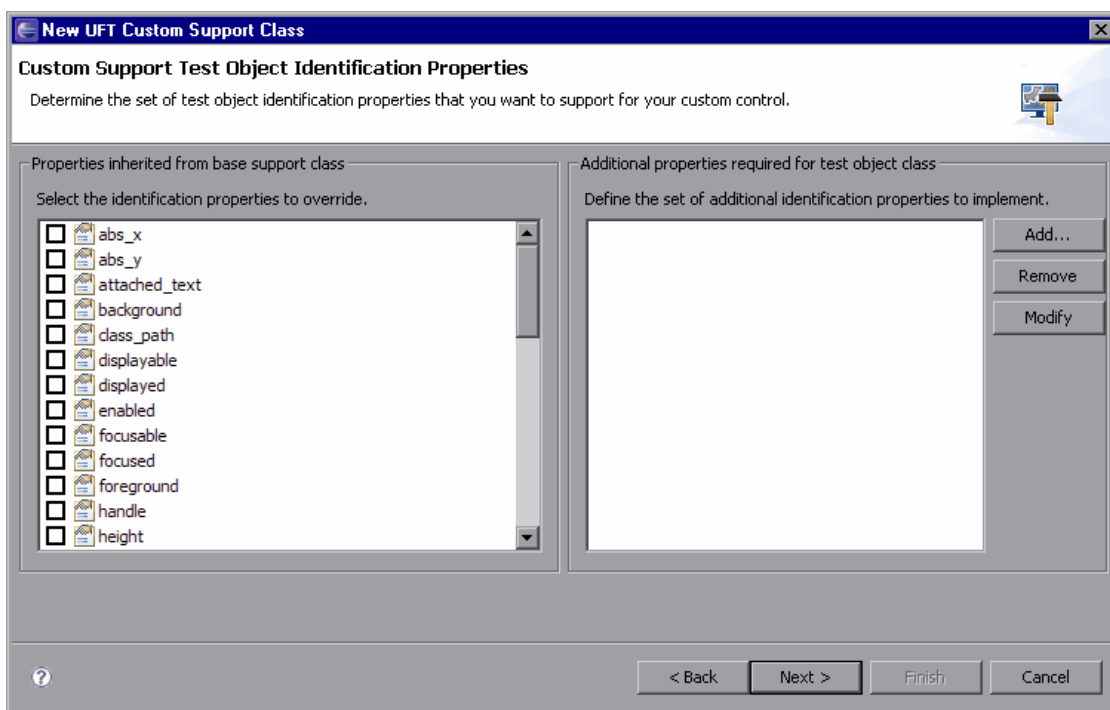

   In the previous screen, you determined the support class that the new support class extends. When the test object mapped to the class whose support you are extending is also a logical test object for the custom class, you select **Same as base support class**. The ImageButtonCS class extends CanvasCS, whose test object class does not adequately represent ImageButton controls.

   The existing JavaButton test object does answer the needs of your custom support.

   a. 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 UFT currently supports. If you define new test objects for custom support, they are included in the list as well.

b. 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.

   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.
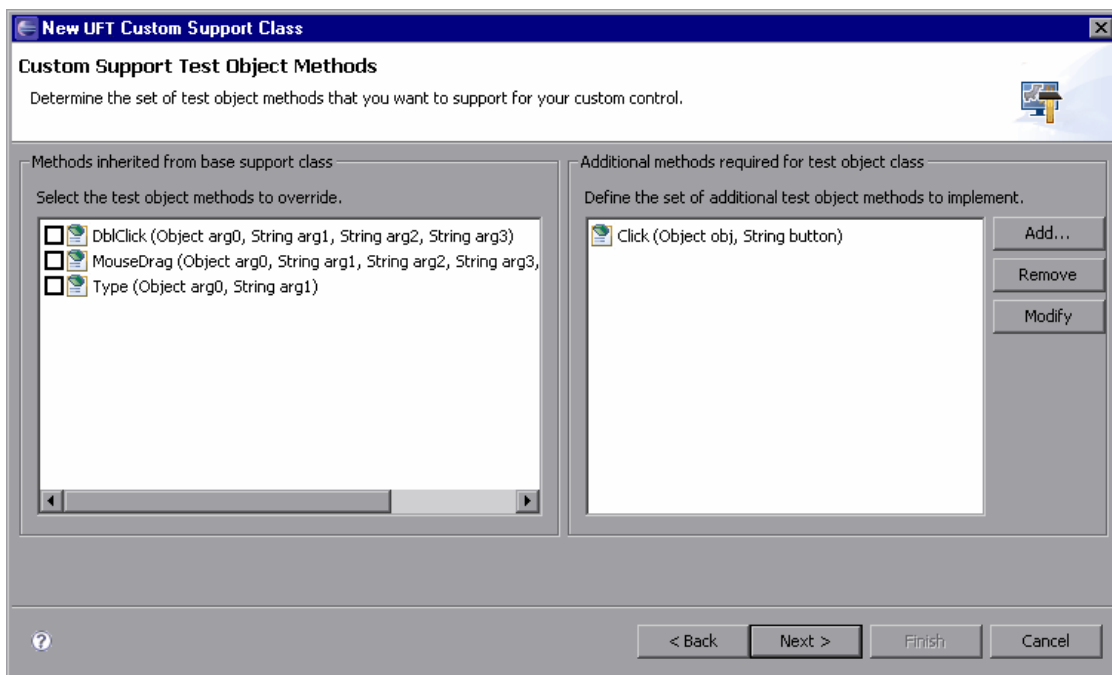
   a. Select the **label** property by clicking the check box. After you finish generating the support files using the wizard, you will override the existing support for this property with a custom implementation that matches the needs of the custom

control.

b. 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.

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.

a. Note that there is only one such method—**Click(Object obj, String button)**. After you finish generating the support files using the wizard, you will

implement the ImageButton support for this method.

b. Click **Next**. The Custom Control Recording Support wizard screen opens.

6. **Determine the set of events for which to listen, 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.
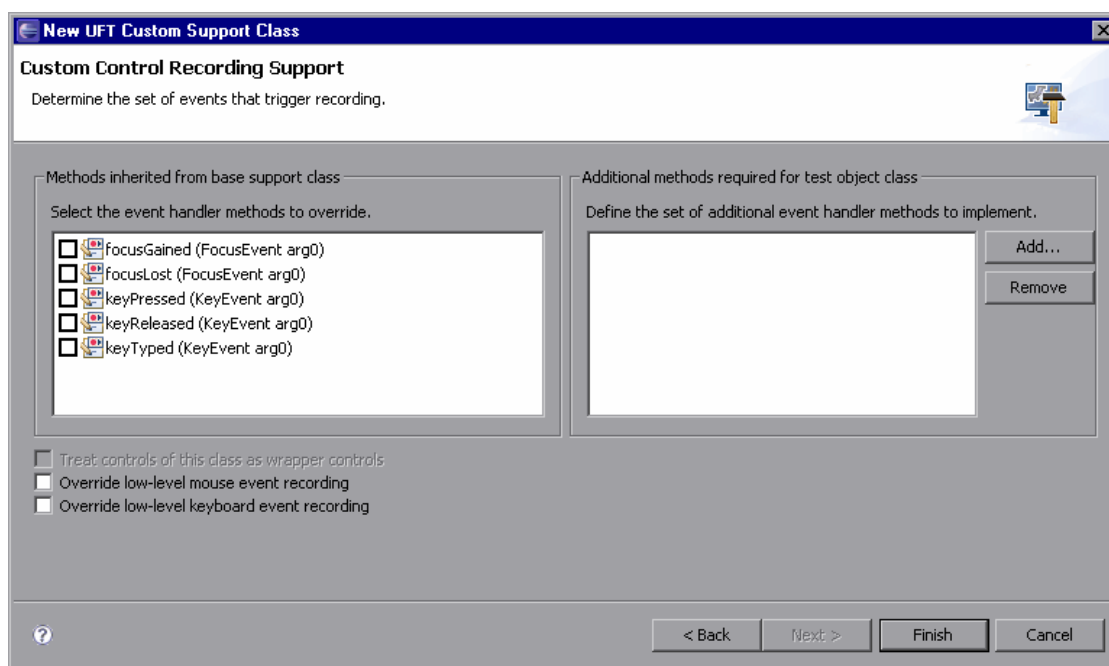

In the left pane, you can see the listeners implemented by CanvasCS. You do not have to override any of these for the ImageButtonCS custom support class.

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.

Perform the following:

a. Click **Add** to add the **ActionListener**.

The Listener dialog box opens.
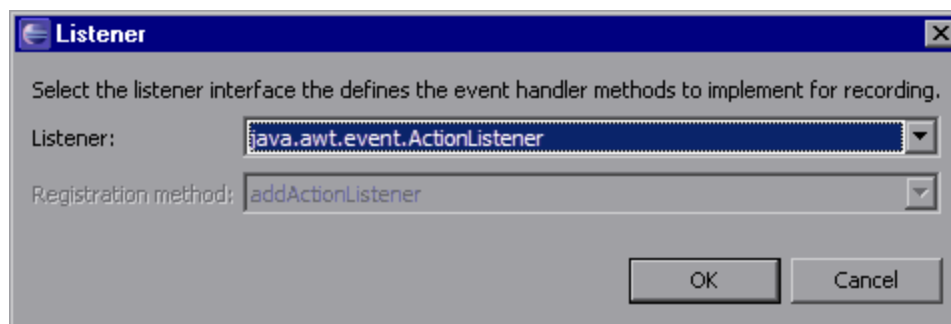
b. If it is not already selected, select **java.awt.event.ActionListener** from the **Listener** list. If the selected listener had more than one registration method, you would also select a method from the **Registration method** list.

c. 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.

d. 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 194.

e. 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**.
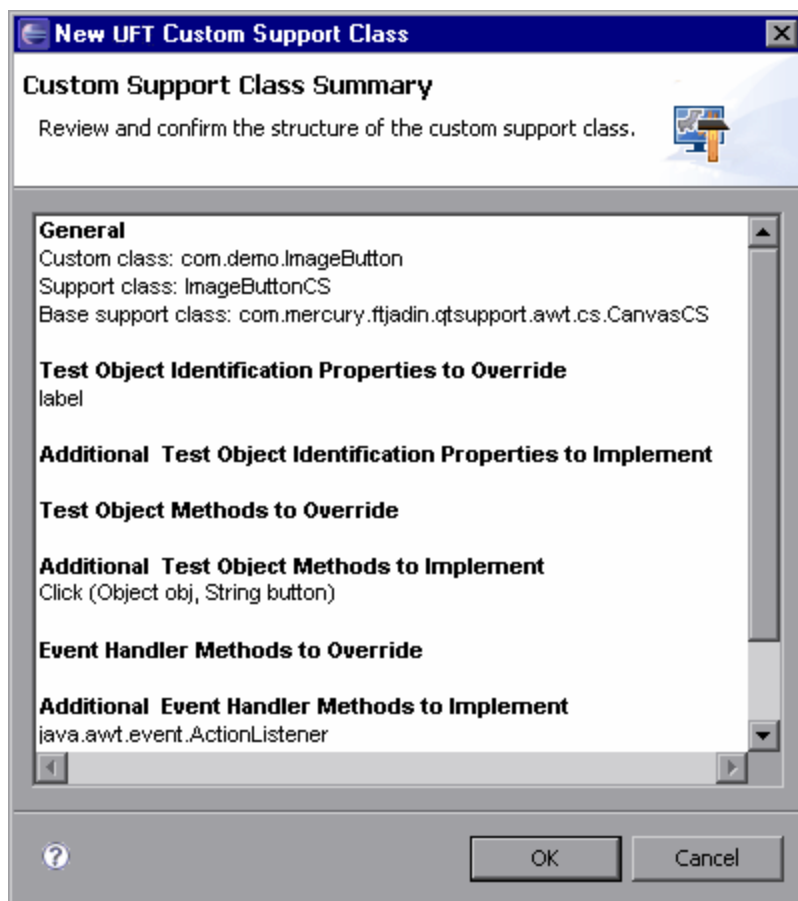
The following changes are made in the ImageControlsSupport project:

- A new UFT 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
.

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. Select **File > Save**, or click the **Save** button.

Understanding the New Custom Support

Your new UFT 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, see the *HP UFT Test Object Schema Help*(available with the Java Add-in Extensibility**SDK**Help).

- 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 UFT 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 86.

The **to_class** property, implemented by the **to_class_attr** method, defines the test object class selected to represent this custom control. UFT 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 UFT.

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 "Creating a New UFT Custom Support Class" on page 181:

```
public String label_attr(Object arg0) {
    return super.label_attr(arg0);
}
```

The following method stub was added because you selected to implement the **Click (Object obj)** test object method, in "Creating a New UFT Custom Support Class" on page 181:

```
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 "Creating a New UFT Custom Support Class" on page 181.

- Listing ActionListener for registration on the ImageButton control:

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

This constructor method is added because in "Creating a New UFT Custom Support Class" on page 181, 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 202. 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 65.

Deploying and Testing the New Custom Toolkit Support

In this part of the lesson, you use the UFT**Deploy Toolkit Support** command in Eclipse to deploy the ImageControls toolkit support to UFT. 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 UFT.**

   In the Eclipse Package Explorer tab, select the **ImageControlsSupport** project.

   Click the **Deploy Toolkit Support** button, or select
   UFT > 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 UFT installation folder. The custom support will be available the next time you open UFT and start the custom application.

For more information on deploying custom toolkit support, see "Deploying and Running the Custom Toolkit Support" on page 77.

2. **Open UFT and load the Java Add-in and the custom toolkit support.**

   Open UFT. The Add-in Manager dialog box displays **ImageControls** as a child of the **Java** Add-in in the list of available add-ins. (If the Add-in Manager dialog box does not open, see the *HP Unified Functional Testing Add-ins Guide* for instructions.)

   Select the check box for **ImageControls** and click **OK**. UFT opens and loads the support you designed.

3. **Run the SampleApp application.**

   UFT establishes its connection with an application when the application opens. Therefore, if the SampleApp application is open, you must close and re-run it.

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

4. **Test the new custom support.**

   View the ImageButton control with the UFT Object Spy and try to record a **Click** operation on it. For details, see the steps in "Planning Support for the ImageButton Control" on page 167.

- UFT recognizes the ImageButton as a JavaButton named ImageButton.

- 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 186), but they are not displayed in UFT 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, see the *HP UFT Object Model Reference for GUI Testing*.

- Because you have overridden the low-level recording, but have not yet implemented the **actionPerformed(ActionEvent arg0)** event handler method, UFT 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 UFT support of the ImageButton control to recognize its name as per your plan ("Planning Support for the ImageButton Control" on page 167). 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 UFT to recognize custom static-text objects using the UFT 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 205.

In ImageButton, the **attached_text** property is empty, so UFT 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**.

1. **Override the label_attr method in the ImageButtonCS class.**

   a. 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);
```

b. Click the **Save** button, or select **File > Save** to save your changes.

> **Note:** You do not have to deploy the toolkit support to UFT again because you changed only Java class files and not configuration files.

2. **Test the new custom support.**

Run the application and view the ImageButton control with the UFT Object Spy, as described in .

> **Note:** You can use an open UFT session (running with the ImageControls custom toolkit support loaded), but you must close the SampleApp application, and run it again, for the changes you made in the custom support to take effect.

UFT now recognizes the ImageButton as a JavaButton named **JavaExt1.gif**.

Implementing Support for a Test Object Method

In this section you extend UFT 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.

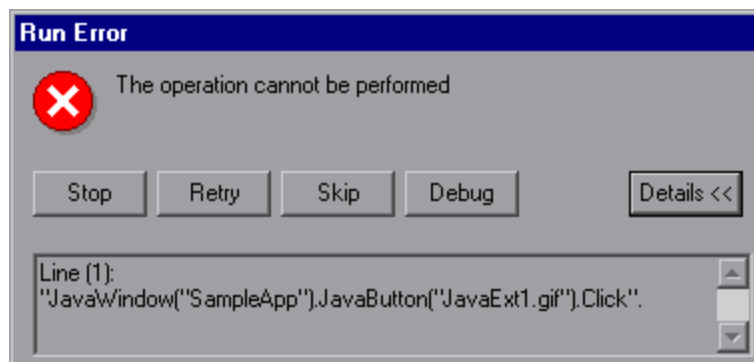1. **Test the current functionality of the Click method on an ImageButton.**

a. In UFT, create a new GUI test, add the **JavaExt1.gif** button to the object repository, and add a step with this object. For instructions on how to do this, see the *HP Unified Functional Testing User 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.

b. Click the **Run** button or select **Run** > **Run**. The Run dialog box opens.

c. Select **New run results folder**. Accept the default results folder name.

d. Click **OK** to close the Run dialog box.

UFT runs the test, and an error message is displayed. Click **Details** on the message box. The following information is displayed:

```
Run Error
  ❌   The operation cannot be performed

    Stop     Retry     Skip     Debug          Details <<

  Line (1):
  "JavaWindow("SampleApp").JavaButton("JavaExt1.gif").Click".
```

The reason for this error is that to run the **Click** operation, the UFT calls Click_ replayMethod, which is currently implemented in the ImageButtonCS to return the error code `NOT_IMPLEMENTED`.

e. Click **Stop**, to stop running the test.

2. **Implement the Click_replayMethod method in ImageButtonCS.**

a. 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;
    }
```

Note: When the wizard created the ImageButtonCS.java file, it automatically added the `importcom.mercury.ftjadin.custom.MicAPI`, required to support this code.

b. Click the **Save** button, or select **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, see the *UFT Java Add-in Extensibility API Reference* (available with the Java Add-in Extensibility SDK Help).

3. **Test the new custom support.**

Note: You do not have to deploy the toolkit support to UFT again because you changed only Java class files and not configuration files.

a. Close the SampleApp application and run it again.

b. In UFT, run the test you created 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.**

   a. 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 86.

b. Click the **Save** button, or select **File > Save**.

> **Note:** You do not have to deploy the toolkit support to UFT again because you changed only Java class files and not configuration files.

2. **Test the new custom support.**

   a. Close the SampleApp application and run it again.

   b. Create a new GUI test and click the **Record** button or select **Record > 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 UFT 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**

- You 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 34.

For more information on the toolkit configuration file, see the *UFT Java Add-in Extensibility Toolkit Configuration Schema Help* (available with the Java Add-in Extensibility SDK Help).

For more information on the MicAPI methods, see the *UFT Java Add-in Extensibility API Reference* (available with the Java Add-in Extensibility SDK Help).

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 UFT Custom Static-Text Support Class Wizard" is specifically dedicated to creating custom support for static-text custom controls.

Chapter 7: 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 163, 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:

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 163, 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 UFT support of the ImageLabel control and the adjacent text box, determine how you want UFT to recognize the controls, and fill in the "Custom Class Support Planning Checklist" on page 211, accordingly.

1. **Open UFT and load the Java Add-in and the custom toolkit support.**

   a. Open UFT. The Add-in Manager dialog box displays **ImageControls** (for which you created support in the previous lesson) as a child of the **Java** Add-in in the list of available add-ins. (If the Add-in Manager dialog box does not open, see the *HP Unified Functional Testing Add-ins Guide* for instructions.)

   b. Make sure that the check boxes for both **Java** and **ImageControls** are selected, and click **OK**.
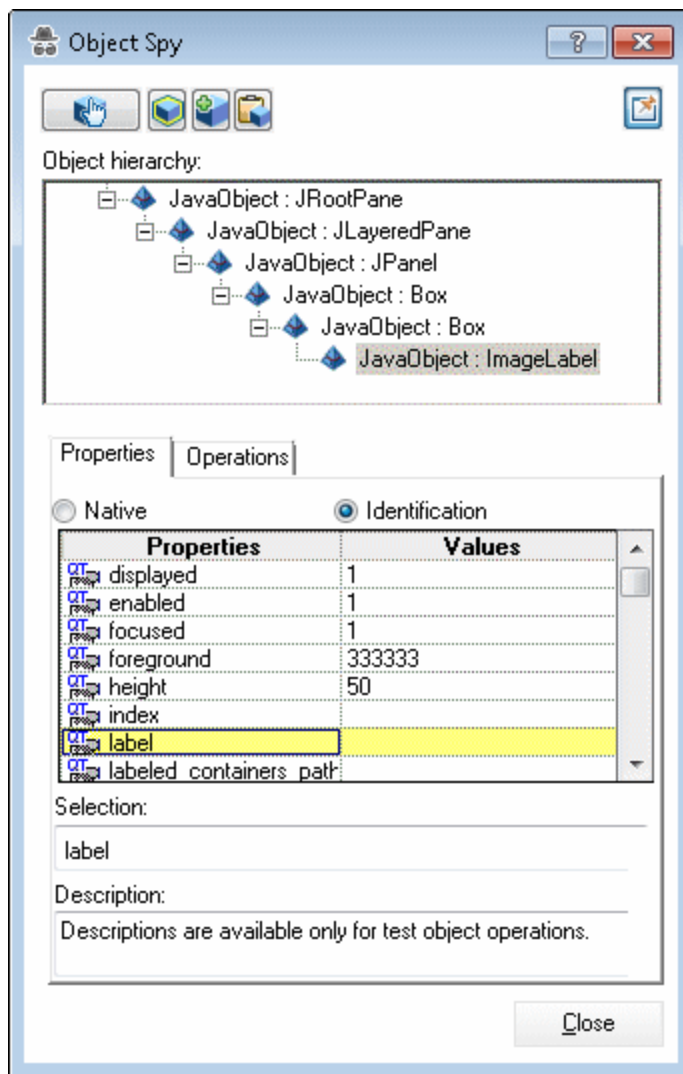
2. **Run the SampleApp application.**

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

3. **Use the Object Spy to view the ImageLabel properties.**

    a. In UFT, open a GUI test and select **Tools > Object Spy** or click the **Object Spy**
        toolbar button to open the Object Spy dialog box. Click the **Properties** tab.

    b. 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 UFT 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:


4. **Use the Object Spy to view the text box properties.**

    a. 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 UFT recognizes it as a **JavaEdit** test object. However, the **label** identification property is empty and UFT does not recognize any adjacent controls as static-text controls. Therefore, the JavaEdit test object is named according to its class name— **TextField**:
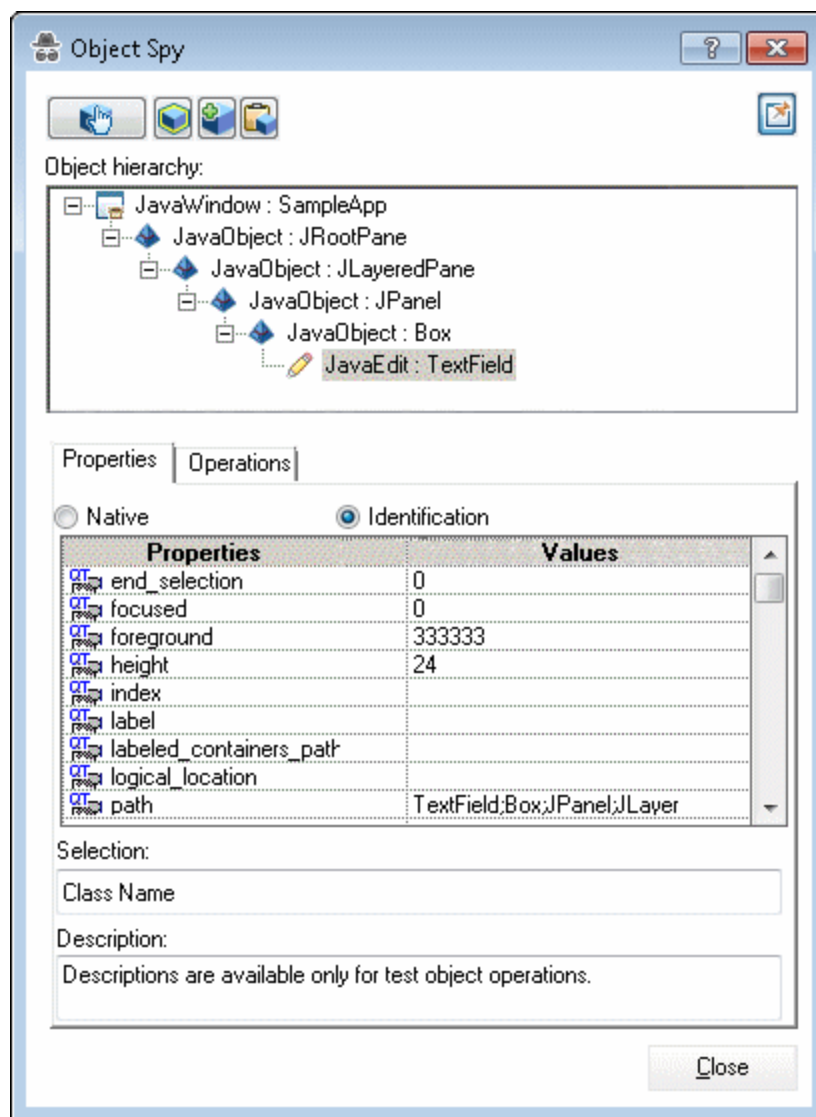
b. Close the Object Spy.

5. **Complete the custom class support planning checklist.**

   The ImageLabel control is a static-text control. You want UFT 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 UFTGUI 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 UFT custom support is not yet available? 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 UFT installed? Yes
☑	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? No
☑	Is the custom control a wrapper? 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`

☑	**Custom Class Support Planning Checklist**
☑	List the test object methods to support (include arguments and return values if required): `nothing special`
☑	Provide support for recording? (AWT-based only) No
☑	If so, list the events that should trigger recording: N/A

Creating the UFT Custom Static-Text Support Class

In the lesson "Learning to Support a Simple Control", you created the ImageControlsSupport UFT Java Add-in Extensibility project (as described on page 172). 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 UFT tests. In addition, there is usually no need to record any operations on a static-text control. Therefore, the UFT 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 UFT Custom Static-Text Support Class wizard.**

   a. In the Eclipse Package Explorer tab, select the UFT Java Add-in Extensibility project, **ImageControlsSupport**. Select **File > New > Other**. The New dialog box
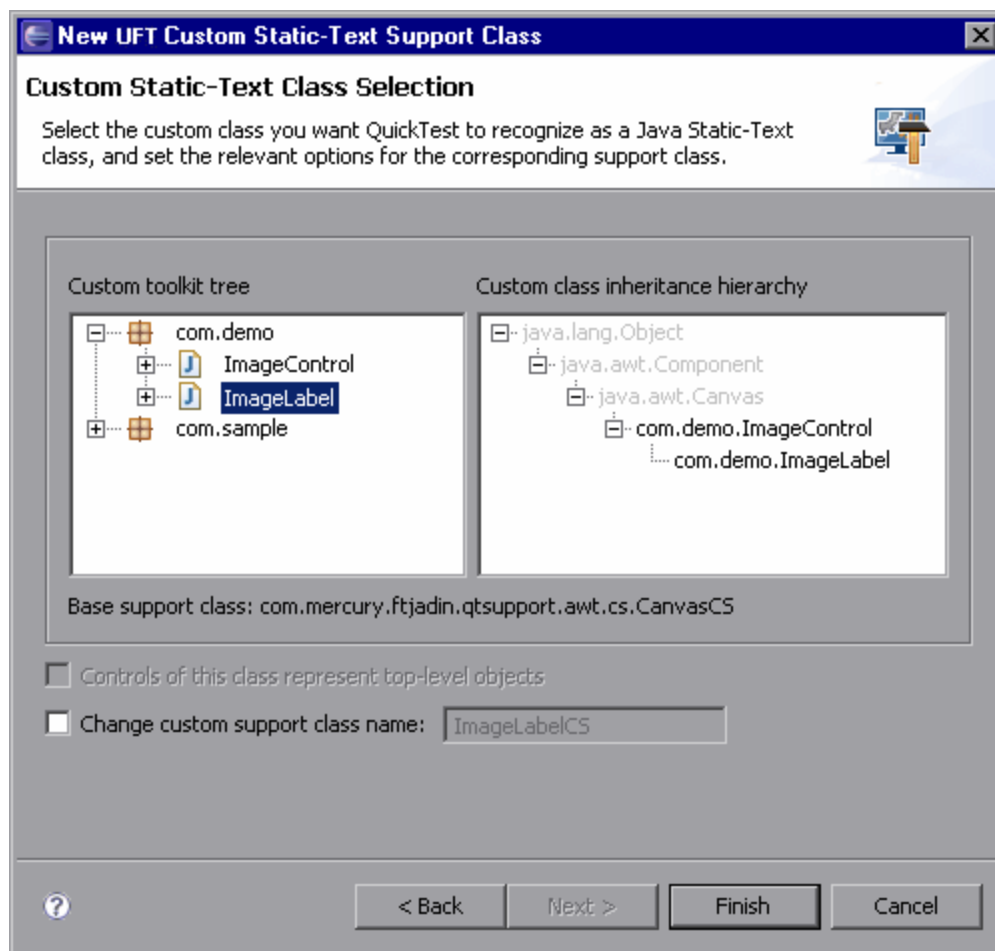
opens.

b. Expand the **Unified Functional Testing** folder and select **UFT Custom Static-Text Support Class**.
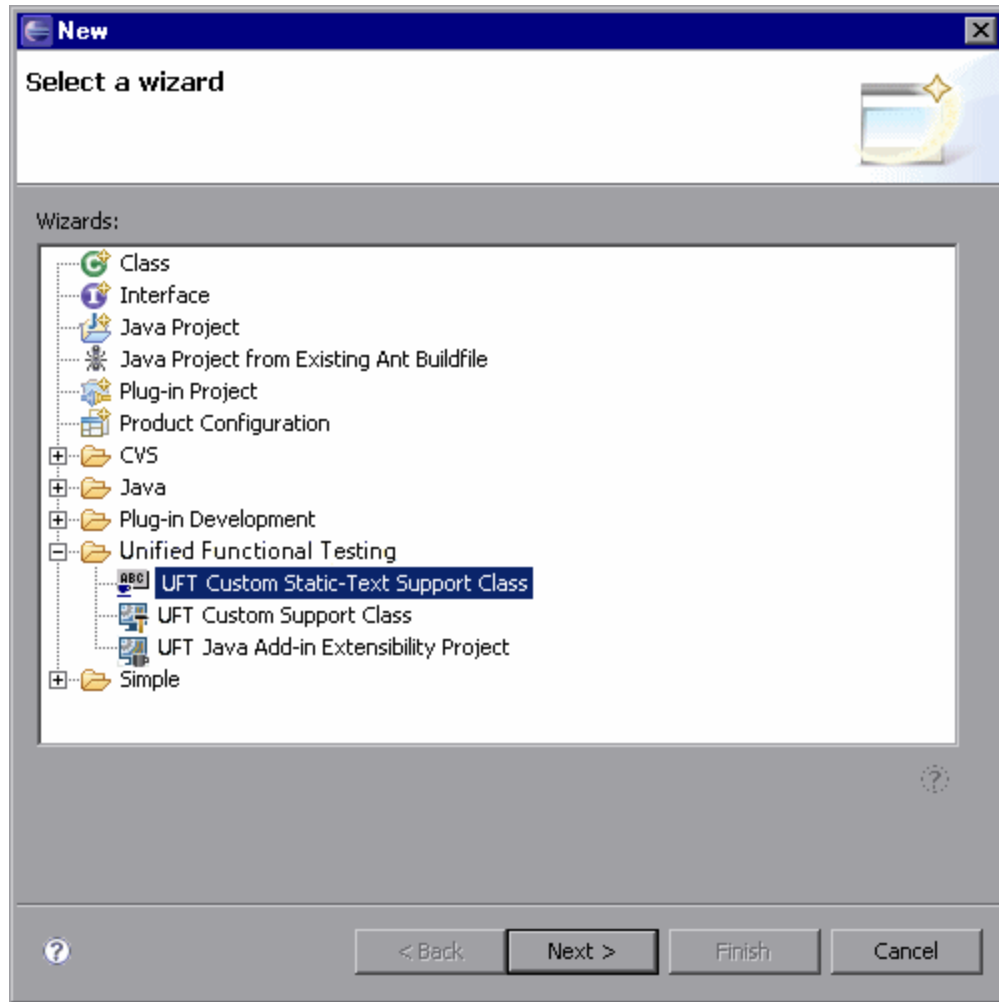
c. Click **Next**. The Custom Class Selection screen opens.

2. **Select the custom class to support, and set the options for the support class.**

   a. Expand the **com.demo** package and select the **ImageLabel** class.

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 "Creating a New UFT Custom Support Class". 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 hierarchy** 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.

b. 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**.
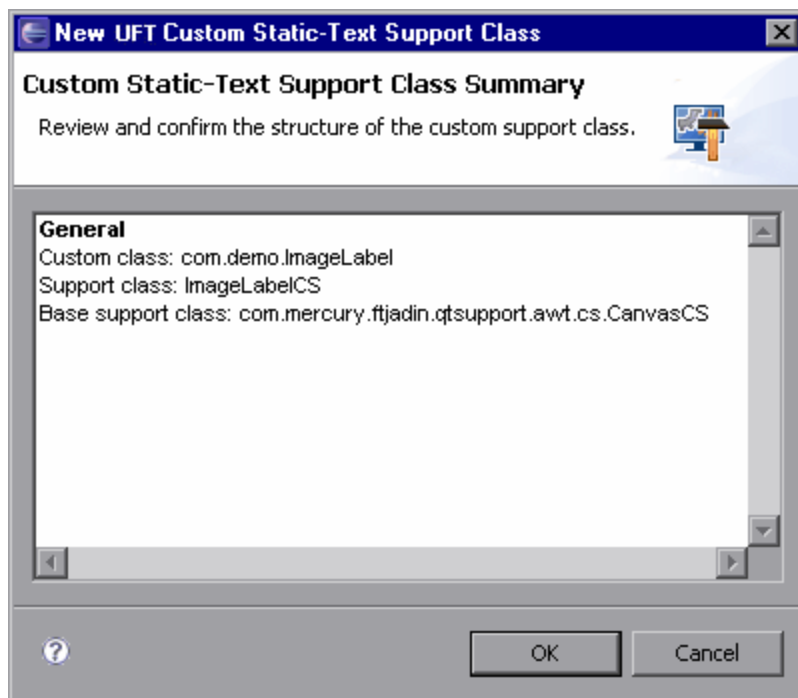
The following changes are made in the ImageControlsSupport project:

- The **ImageControls.xml** file is modified to map the ImageLabel custom class to its support class—ImageLabelCS.

- A new UFT 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. Select **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 UFT that the ImageLabel control is a JavaStaticText object. This means that the UFT 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", 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 "Common Identification Property Support Methods" on page 61.

Deploying and Testing the New Custom Static-Text Support Class

In this section, you use the UFT **Deploy Toolkit Support** command in Eclipse to deploy the ImageControls toolkit support to UFT. This adds the ImageLabel support to UFT, 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 UFT.**

   In the Eclipse Package Explorer tab, select the **ImageControlsSupport** project.

   Click the **Deploy Toolkit Support** button, or select **UFT > 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 UFT installation folder. The custom support will be available the next time you start the custom application. (The test object configuration file was not modified. Therefore, it is not necessary to reopen UFT.)

   For more information on deploying custom toolkit support, see "Deploying and Running the Custom Toolkit Support" on page 77.

2. **Test the new custom support.**

   Run the application and view the ImageLabel control and text box with the UFT Object Spy, as described in "Planning Support for the ImageLabel Control" on page 206.

   > **Note:** UFT establishes its connection with an application when the application opens. Therefore, although you can use an open UFT session (running with the ImageControls toolkit support loaded) to test the changes, you must close the SampleApp application, and run it again.

   UFT recognizes the ImageLabel as a **JavaStaticText** object named **ImageLabel(st)**.
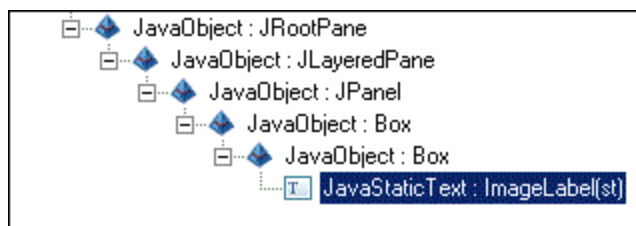
CanvasCS, which ImageLabelCS extends, does not provide support for the **label** identification property. Therefore, ImageLabel's **label** property is empty (as is its **attached text** property). As a result, the superclass **tag** property returns ImageLabel's class name, and ImageLabel's **tag** property is **ImageLabel(st)**.

UFT 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.
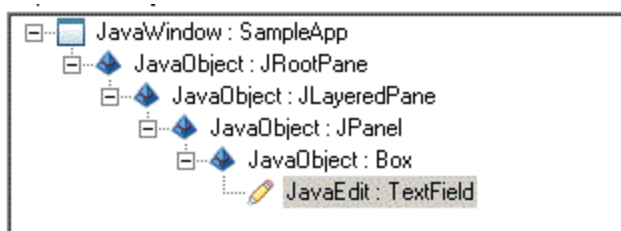
Completing the Support for the Static-Text Control

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 UFT 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.**

   a. 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).

b. Click the **Save** button, or select **File > Save** to save your changes.

> **Note:** You do not have to deploy the toolkit support to UFT again because you changed only Java class files and not configuration files.

2. **Test the new custom support.**

Run the application and view the ImageLabel control and the text box with the UFT Object Spy, as described in "Planning Support for the ImageLabel Control" on page 206.

> **Note:** You did not modify the test object configuration file. Therefore, you can use an open UFT session (running with the ImageControls custom toolkit support loaded). However, you must close the SampleApp application and run it again, for the changes you made in the custom support to take effect.

UFT now recognizes the ImageLabel as a **JavaStaticText** test object named **UFT Java(st)**, with the **label** property **UFT Java**.
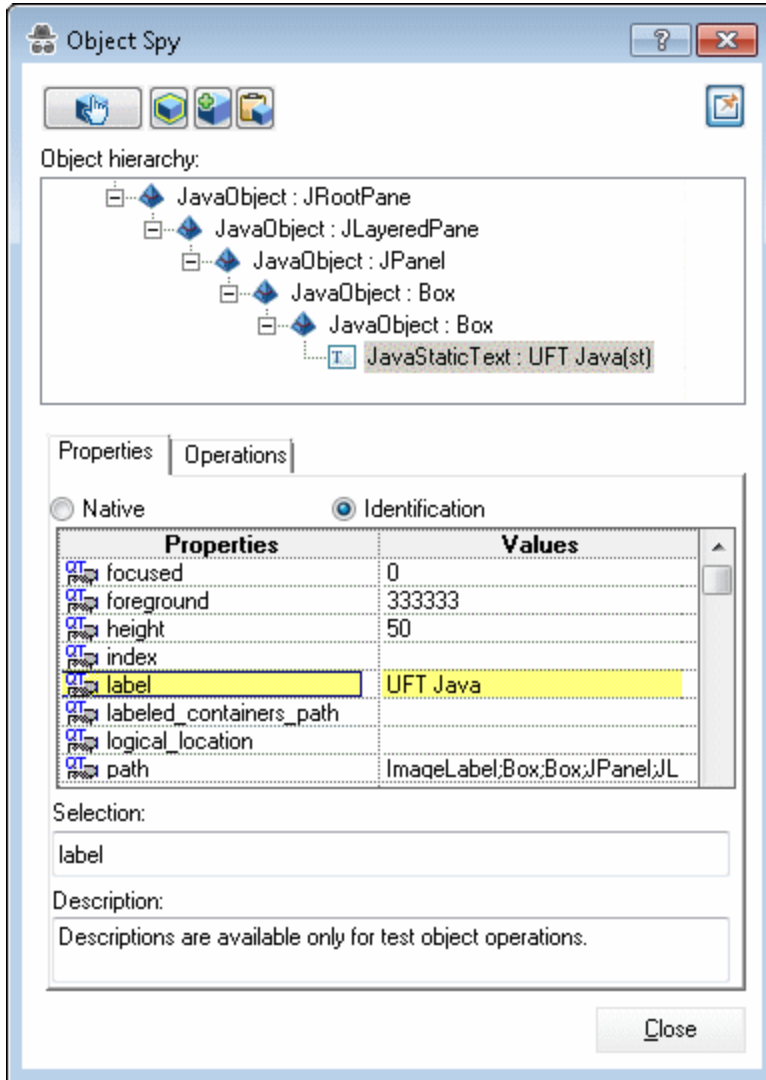
UFT now recognizes the text box as a **JavaEdit** test object named **UFT Java**. The **label** property of the **JavaEdit** test object is empty. The ImageLabel's **label** property provides the text for the JavaEdit's **attached text** property, which is used as the test object name:
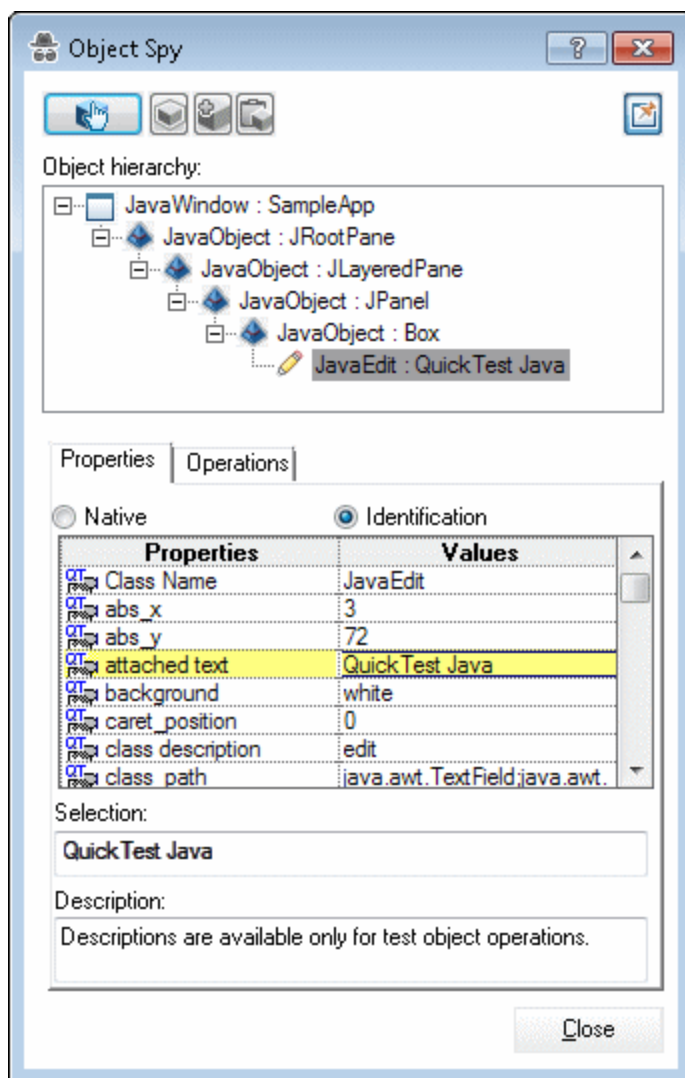
Note: If you modify the SampleApp application and remove the line `imageLb.setText("QuickTest Java");`, the ImageLabel will not display any text. UFT will then recognize the ImageLabel as a **JavaStaticText** test object named **mercury.gif(st)**, with the **label** property **mercury.gif**. UFT will 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
<UFT Java Add-in Extensibility SDK installation folder>\samples\ImageControlsSupport folder. (If you deploy this example manually, you must compile the Java classes before deploying.)

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 211 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 UFT Custom Support Class wizard.**

    a. In the Eclipse Package Explorer tab, select the new UFT Java Add-in Extensibility project, **ImageControlsSupport**. Select **File > New > Other**. The New dialog box opens.

    b. Expand the **Unified Functional Testing** folder, select **UFT 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.**

    a. Expand the **com.demo** package and select the **ImageControl** class:
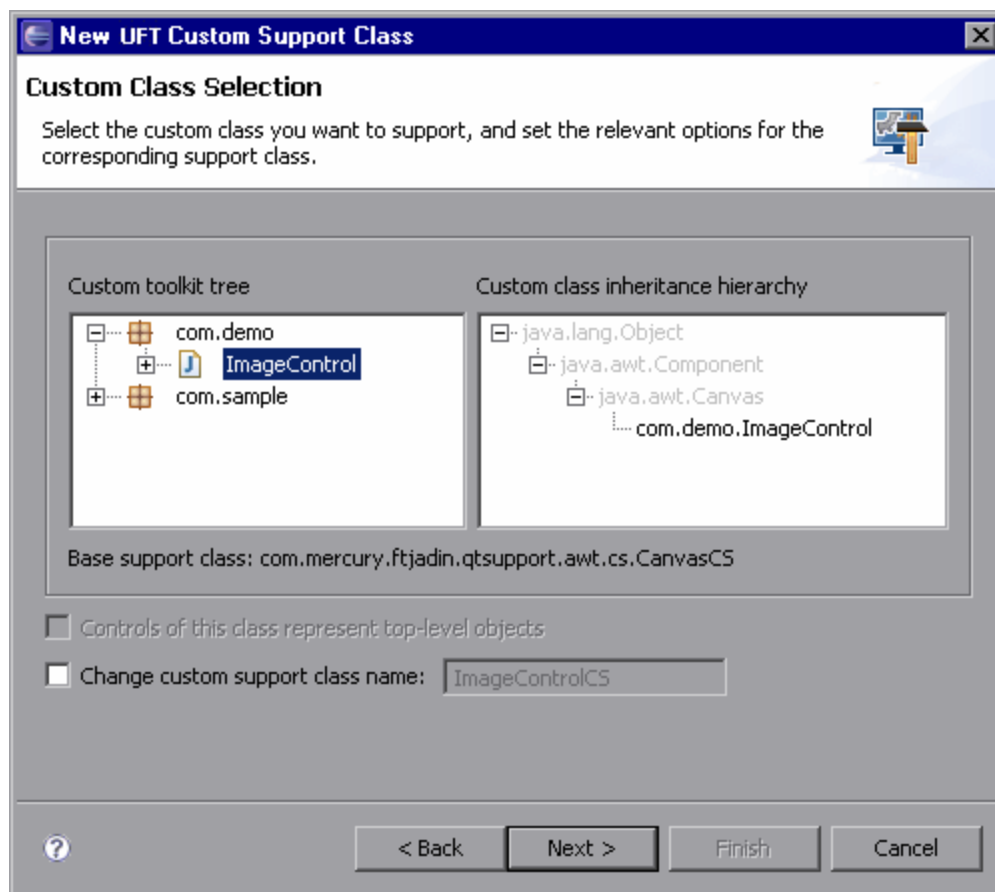
    In the **Custom toolkit tree** pane, you can see that the ImageControl class is the only class in the **com.demo** package that is not yet supported.

    In the **Custom class inheritance hierarchy** 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.

    b. Click **Next**. The Test Object Class Selection screen opens.

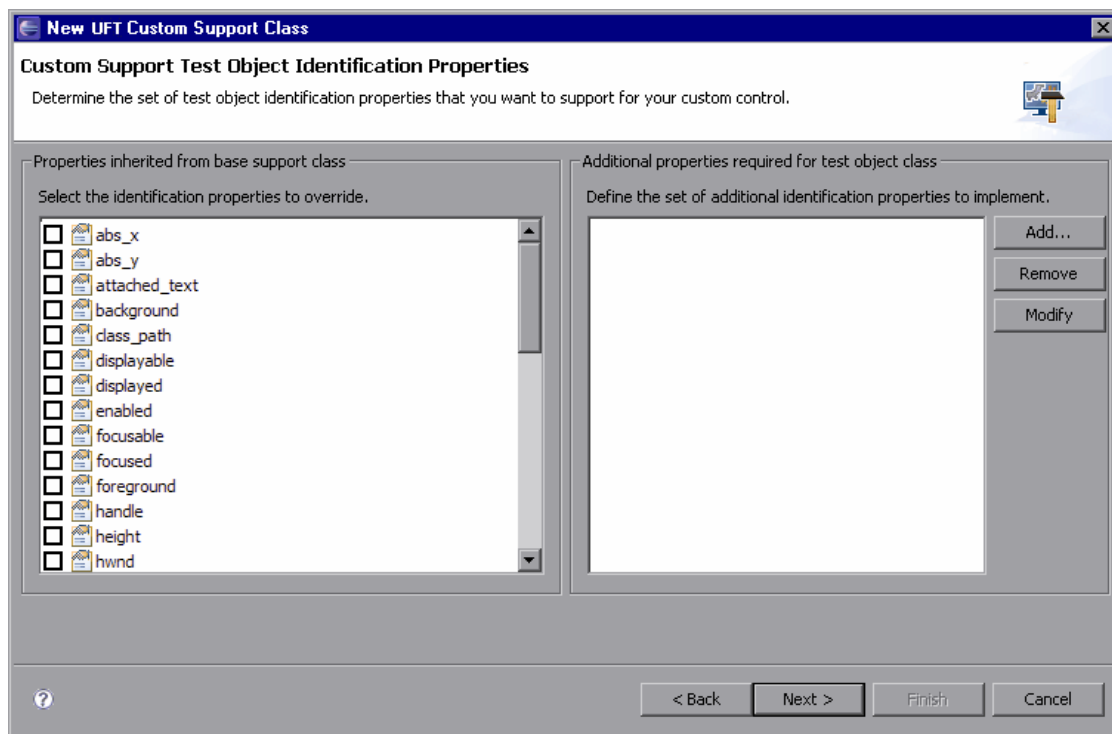3. **Select a test object class to represent the custom control.**

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.
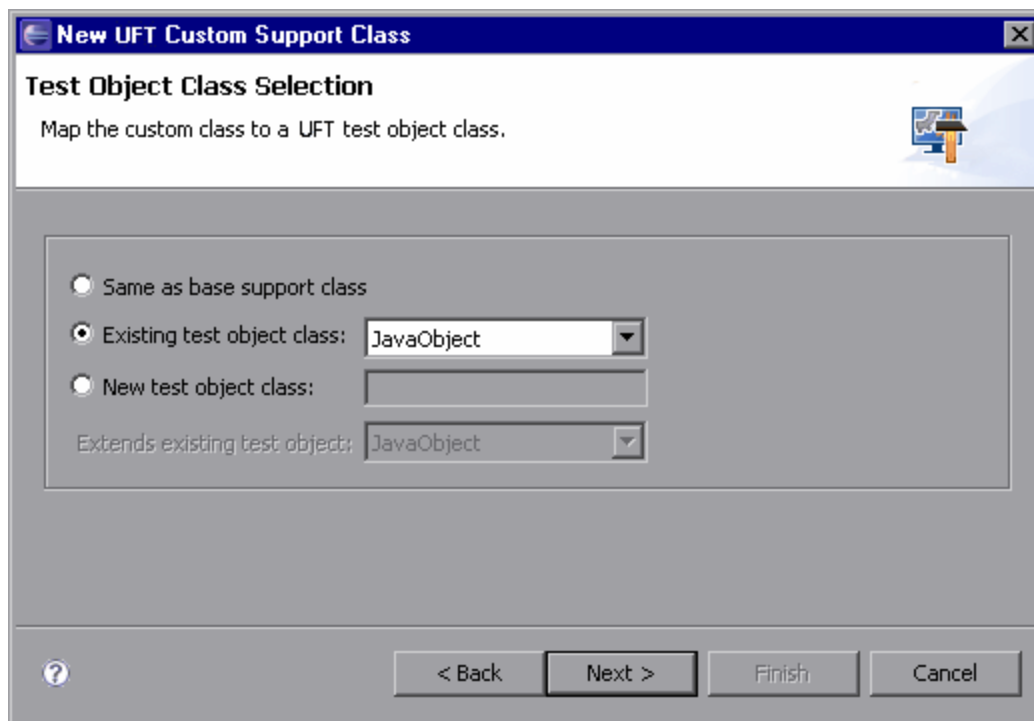
Perform the following:

a. 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.

b. 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.

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 226), 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.
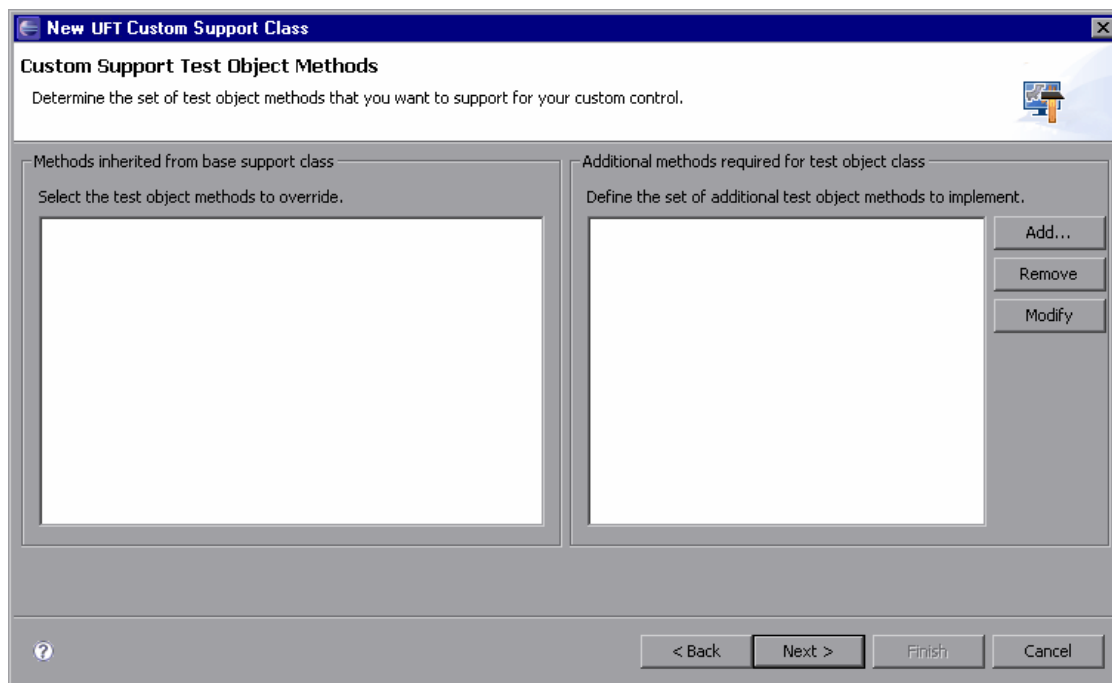
a. Select the **label** property by clicking its check box. After you finish generating the support files using the wizard, you will override the existing support for this property with a custom implementation that matches the needs of the custom

control.

    b. 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.


In the Test Object Class Selection screen (on page 226), 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.

    a. Consider that the ImageControl custom control does not have any test object methods that need to be supported.

    b. Click **Next**. The Custom Control Recording Support wizard screen opens.

6. **Determine the set of events for which to listen, 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.
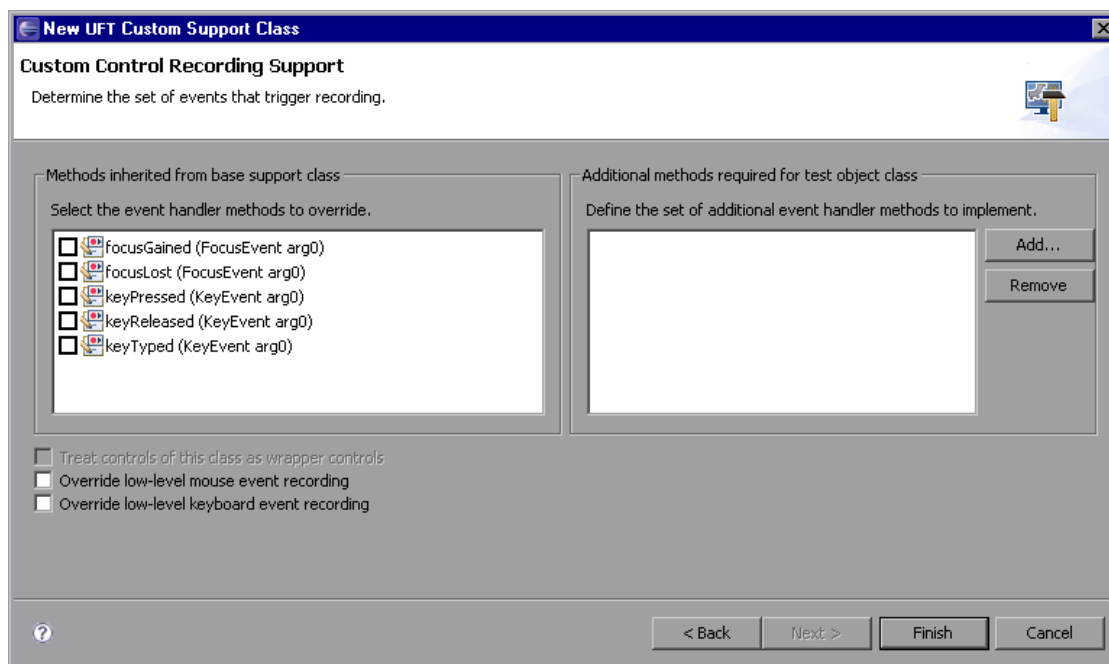

   In the left pane, you can see the listeners implemented by CanvasCS. You do not have to override any of these for the ImageControlCS custom support class.

   a. Consider that 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.

   b. 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**.
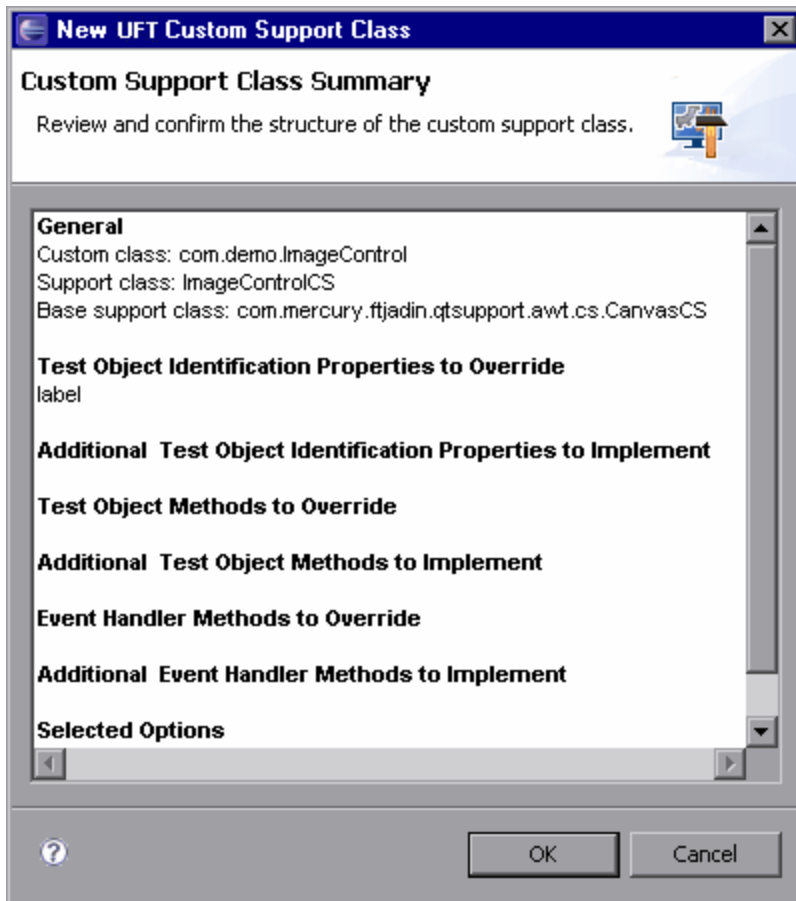
The following changes are made in the ImageControlsSupport project:

- The **ImageControls.xml** file is modified to map the ImageControl custom class to its support class—ImageControlCS.

- A new UFT 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. Select **File > Save**, or click the **Save** button.

8. **Implement the label_attr method in the ImageControlCS class.**

   a. 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);
   ```

   b. Click the **Save** button, or select **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.**

   a. Open the **ImageButtonCS.java** file in the ImageControlsSupport project in Eclipse, and locate the **ImageButtonCS** class signature:

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

   b. Modify the signature so that ImageButtonCS extends ImageControlCS:

      ```
      public class ImageButtonCS extends ImageControlCS implements
      ActionListener
      ```

   c. Remove the **label_attr** method from the ImageButtonCS class.

   d. Save the **ImageButtonCS.java** file.

2. **Modify the ImageLabelCS class to extend ImageControlCS.**

   a. In the **ImageLabelCS.java** file, replace `public class ImageLabelCS extends CanvasCS` with `public class ImageLabelCS extends ImageControlCS`.

   b. Replace the following lines in the **label_attr** method in the ImageLabelCS class:

      ```
      ImageLabel il = (ImageLabel)obj;
      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);
```

c. 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 UFT.**

    In the Eclipse Package Explorer tab, select the **ImageControlsSupport** project.

    Click the **Deploy Toolkit Support** button, or select
    UFT > 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 167 and "Planning Support for the ImageLabel Control" on page 206, to re-run the SampleApp application and to 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 UFT (running with the ImageControls custom toolkit

support loaded).

The changes you made to the custom toolkit support set do not affect the way UFT 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 UFT with no additional effort required.

You can find a ready-made example of the improved support for the ImageControls toolkit in the
<UFT Java Add-in Extensibility SDK installation folder>\samples\ImageControlsSupportAdvanced folder. (If you deploy this example manually, you must compile the Java classes before deploying.)

Lesson Summary

In this lesson you created support for the ImageLabel control, allowing UFT 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 "Common Identification Property Support Methods" on page 61.

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.

Chapter 8: 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 163, 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.

This lesson guides you through the following stages:

Preparing for This Lesson

Before you extend UFT 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 select **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 **<UFT 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 Javaboutique sample source files. The new project, named Javaboutique, is displayed in the Package Explorer tab.

   > **Note:** The steps for creating a new project in Eclipse may vary, depending on the Eclipse version that you use.

After you create the **Javaboutique** project, expand the 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 UFT Java Add-in Extensibility project for the Javaboutique custom toolkit and the support class for AllLights.

You can find a ready-made example of the support for AllLights and for AwtCalc in the **<UFT Java Add-in Extensibility SDK installation folder>\samples\JavaboutiqueSupport** folder. (If you deploy this example manually, you must compile the Java classes before deploying.)

Run the AllLights application to become familiar with the behavior of the AllLights control:

In the Eclipse Package Explorer tab, right-click the **Alllights.java** class in the **org.boutique.toolkit** package and select **Run As > Java Applet**. The AllLights application opens:
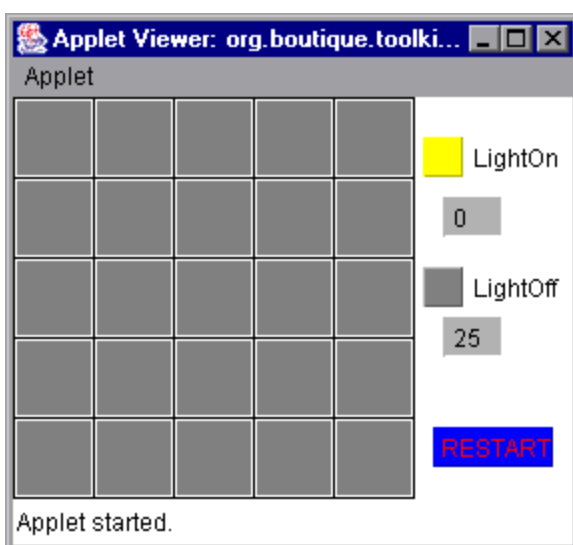
Click different locations in the frame:

- 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 UFT support of the AllLights control, determine the answers to the questions in the "Understanding the Custom Class Support Planning Checklist" on page 93, and fill in the "Custom Class Support Planning Checklist " on page 244, accordingly.

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

1. **Run the AllLights application and open UFT.**

   a. If the AllLights application is already running, select **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 select **Run As > Java Applet** to run it.

   b. Open UFT and load the Java Add-in.

2. **Use the Object Spy to view the AllLights properties and methods.**

   a. In UFT, open a GUI test and select **Tools > Object Spy** or click the **Object Spy** toolbar button to open the Object Spy dialog box. Click the **Properties** tab.

   b. In the Object Spy dialog box, click the pointing hand , then click the AllLights application.

The AllLights control extends JavaApplet, for which UFT support is built in, therefore it recognizes the application as a **JavaApplet**, named **AllLights**. The icon shown is the standard JavaApplet class icon:
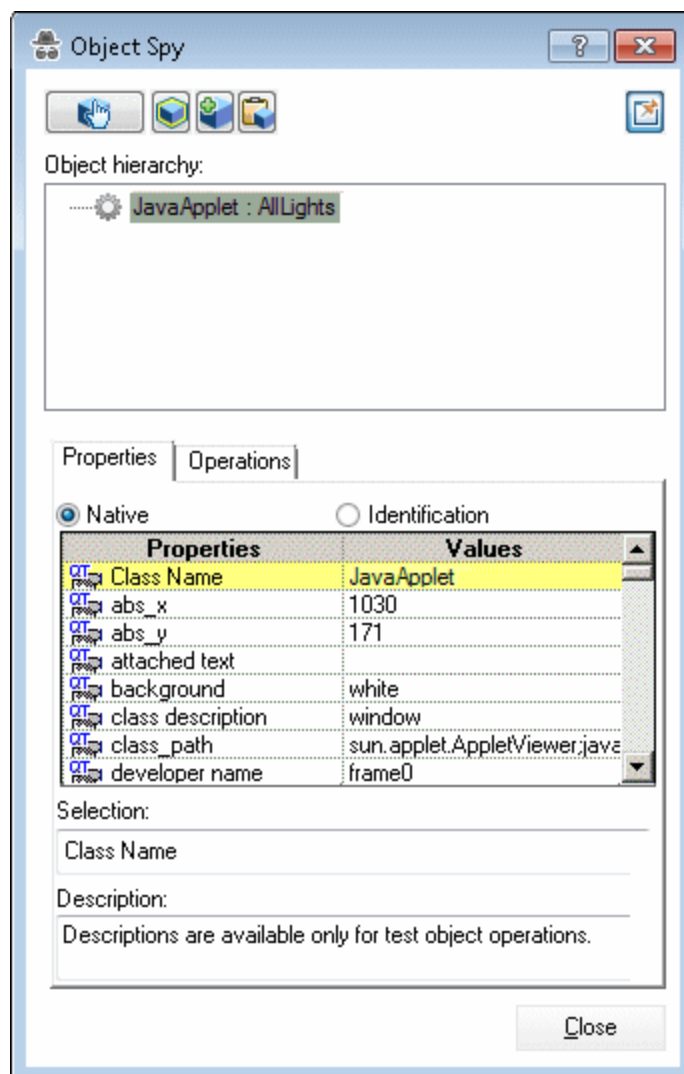
c. Close the Object Spy.

3. **Record operations on the AllLights control.**

a. In UFT, select **Record > 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**.

b. Click the **Record** button or select **Record > 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	Documentation
▼ 🧩 Action1			
▼ ☐ AllLights			
⚙ AllLights	Click	142,144,"LEFT"	Click the "AllLights" applet with the "LEFT" mouse button.
⚙ AllLights	Click	16,188,"LEFT"	Click the "AllLights" applet with the "LEFT" mouse button.
⚙ AllLights	Click	211,35,"LEFT"	Click the "AllLights" applet with the "LEFT" mouse button.

c. Click the **Stop** button or select **Record > 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 UFT 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.**

In this step you plan the required support for the AllLights control and summarize the information in the support planning checklist.

a. Decide which custom class to support:

The AllLights custom class extends the Applet class, supported on UFT by AppletCS.

You want UFT 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.

b. Map 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.

c. Decide the details for the new test object class:

- The new test object class is represented by the icon file:
  <UFT 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 UFT Checkpoint Properties dialog box.

d. Determine whether the control is a top-level object or a wrapper, and decide how to name the test objects:

- AllLights controls are top-level objects, but not wrappers.

- The name of the test object itself should be **Lights**.

e. Determine the identification properties that need to be supported:

- **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.

f. Determine the test object methods that need to be supported:

- **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.

g. Determine how to support 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 UFT custom support is not yet available? 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 UFT installed? Yes
☑	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): `<UFT 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
☑	Is the custom control a wrapper? 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)`

☑	**Custom Class Support Planning Checklist**

☑	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
☑	If so, list the events that should trigger recording: `MouseEvents`

Creating the UFT Java Add-in Extensibility Project

In this section you create a new project for the Javaboutique toolkit support. Do this using one of the wizards provided by the UFT Java Add-in Extensibility plug-in in Eclipse:

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

   a. In Eclipse, select **File > New > Project**. The New Project dialog box opens. Expand the **Unified Functional Testing** folder and select **UFT Java Add-in Extensibility Project**.
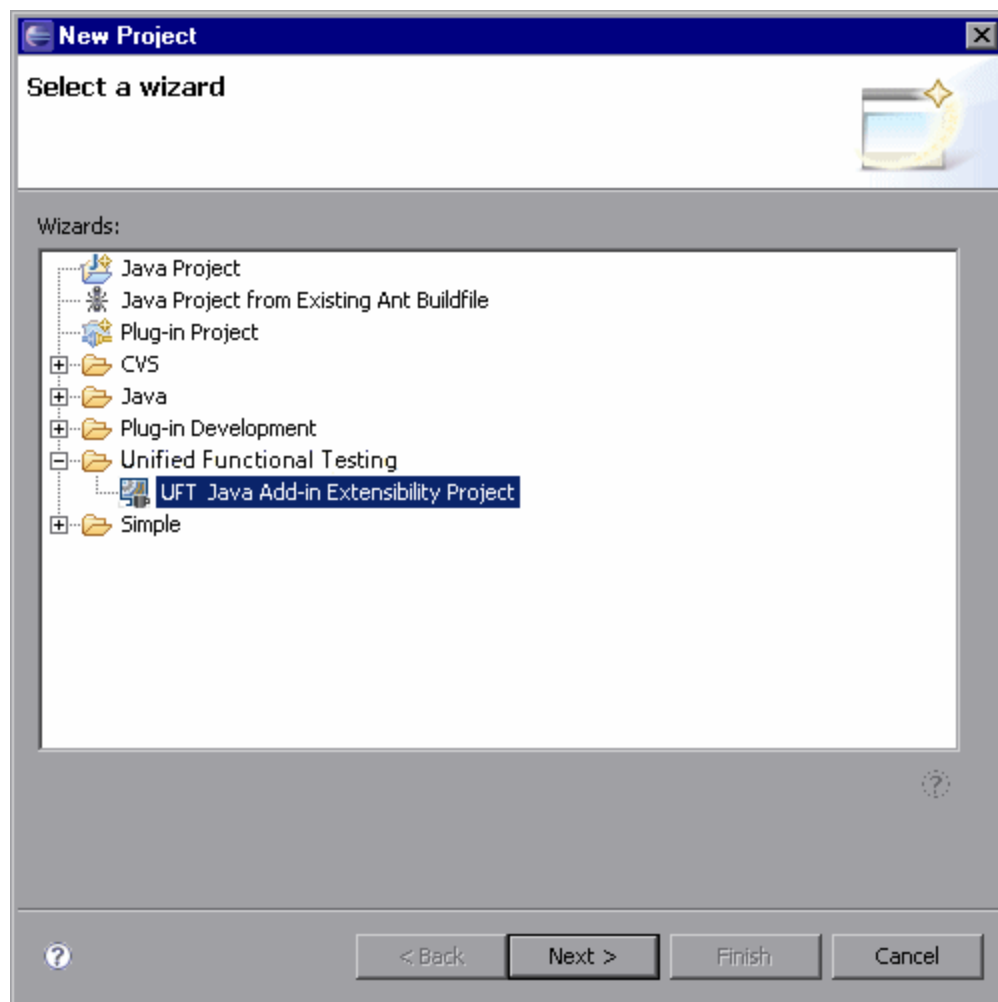
b. Click **Next**. The UFT Java Add-in Extensibility Project screen opens. The details on this screen may vary, depending on the version of Eclipse that you are using.

2. **Enter the UFT Java Add-in Extensibility project details.**

a. In the **Project name** box, enter `JavaboutiqueSupport`. Select **Create separate folders for sources and class files**. (In earlier Eclipse versions this option is named **Create separate source and output folders**.) For more information on this dialog box, see the *Eclipse Help*.
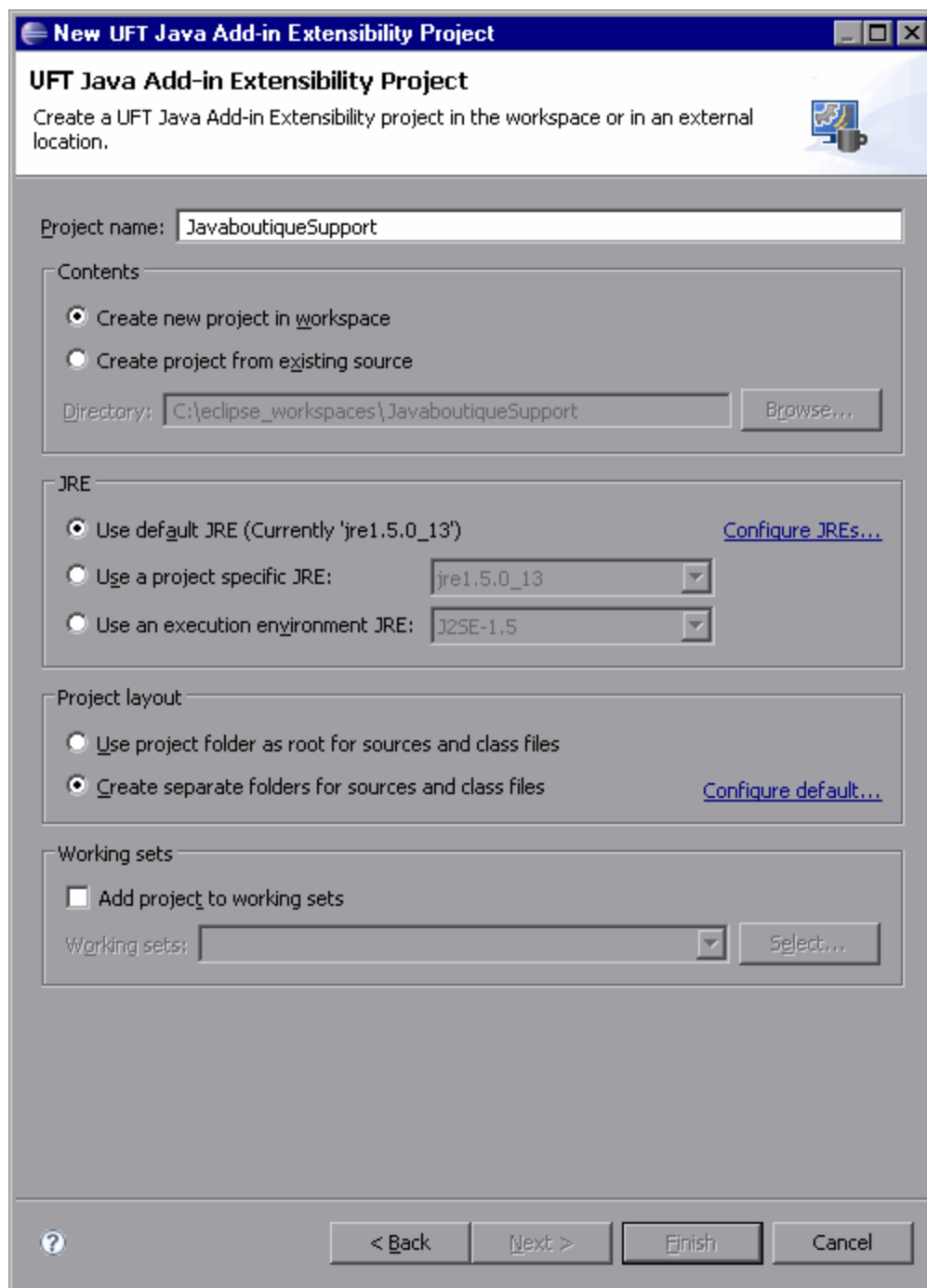
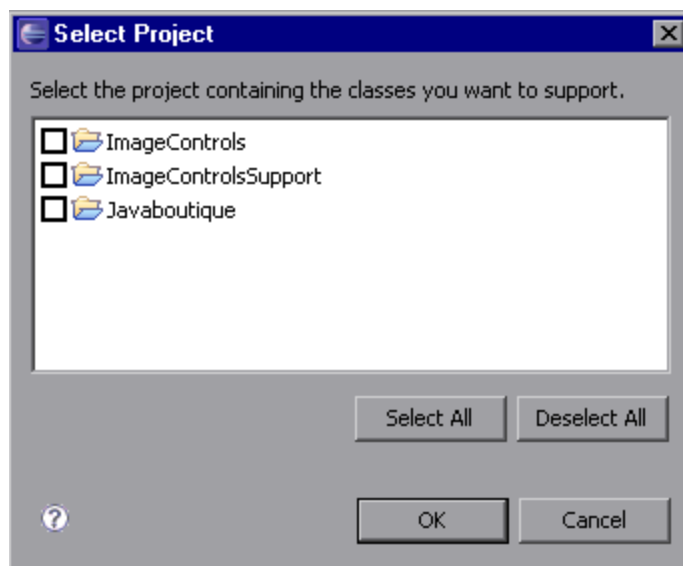b. Click **Next**. The Custom Toolkit Details screen opens.

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.
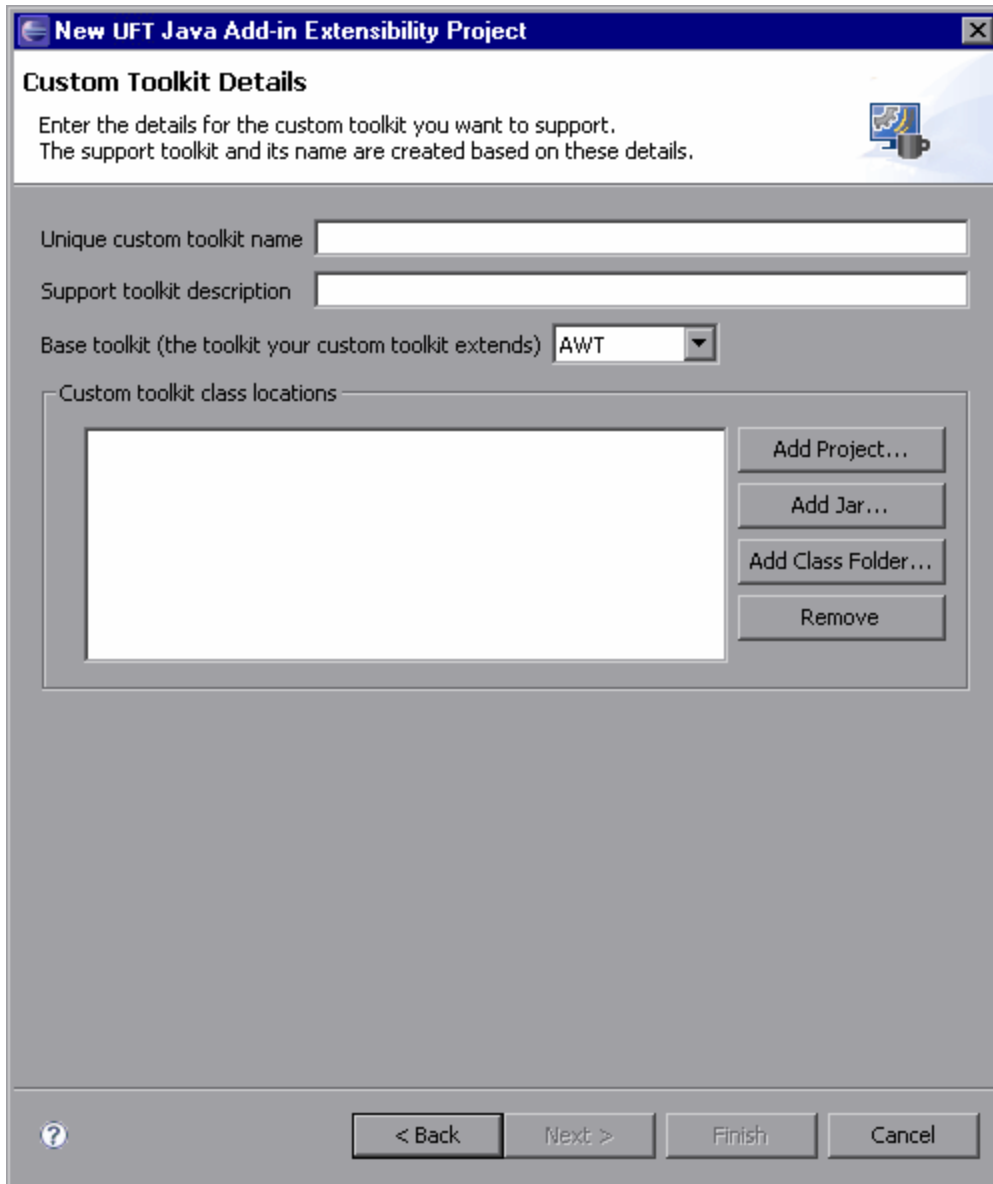
Enter the following details:

a. In the **Unique custom toolkit name** enter `Javaboutique`.

b. In the **Support toolkit description** box enter: `Javaboutique toolkit support`.

c. The AllLights custom class extends an AWT component, so keep the default selection `AWT` as the **Base toolkit**.

d. 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:

e. 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**.
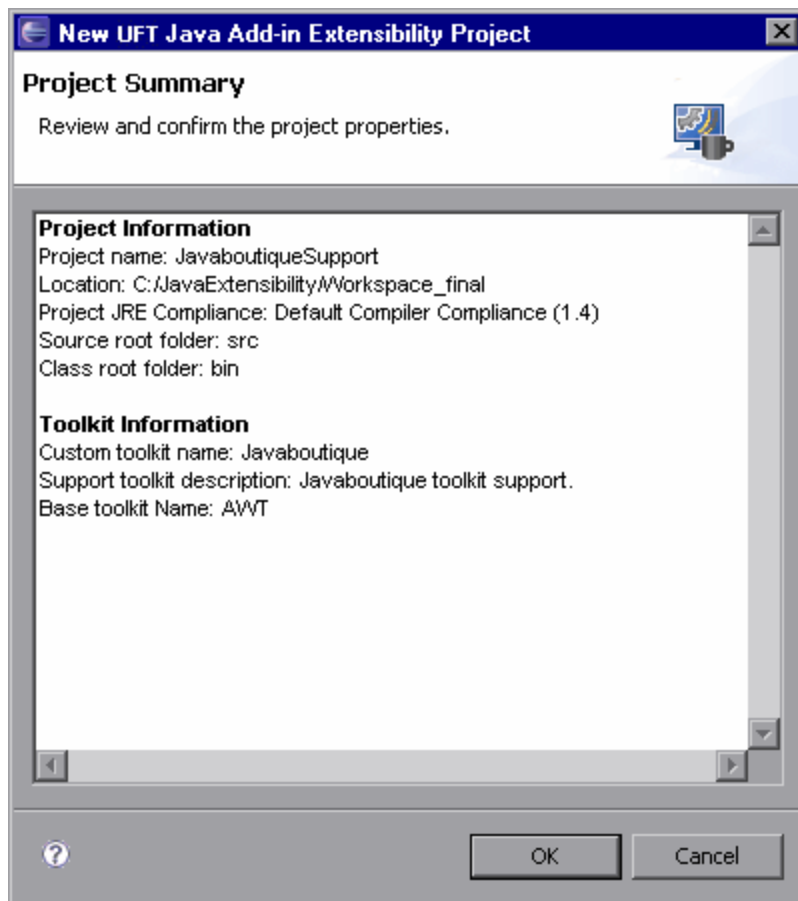
   The New UFT Java Add-in Extensibility project JavaboutiqueSupport is created, with the basic packages and files of the custom toolkit support set:

   - 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 UFT 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 UFT 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 "Using the UFT Java Add-in Extensibility Eclipse Plug-In" on page 97.

1. **Open the New UFT Custom Support Class wizard.**

   a. In the Eclipse Package Explorer tab, select the new UFT Java Add-in Extensibility project, **JavaboutiqueSupport**. Select **File > New > Other**. The New dialog box opens.
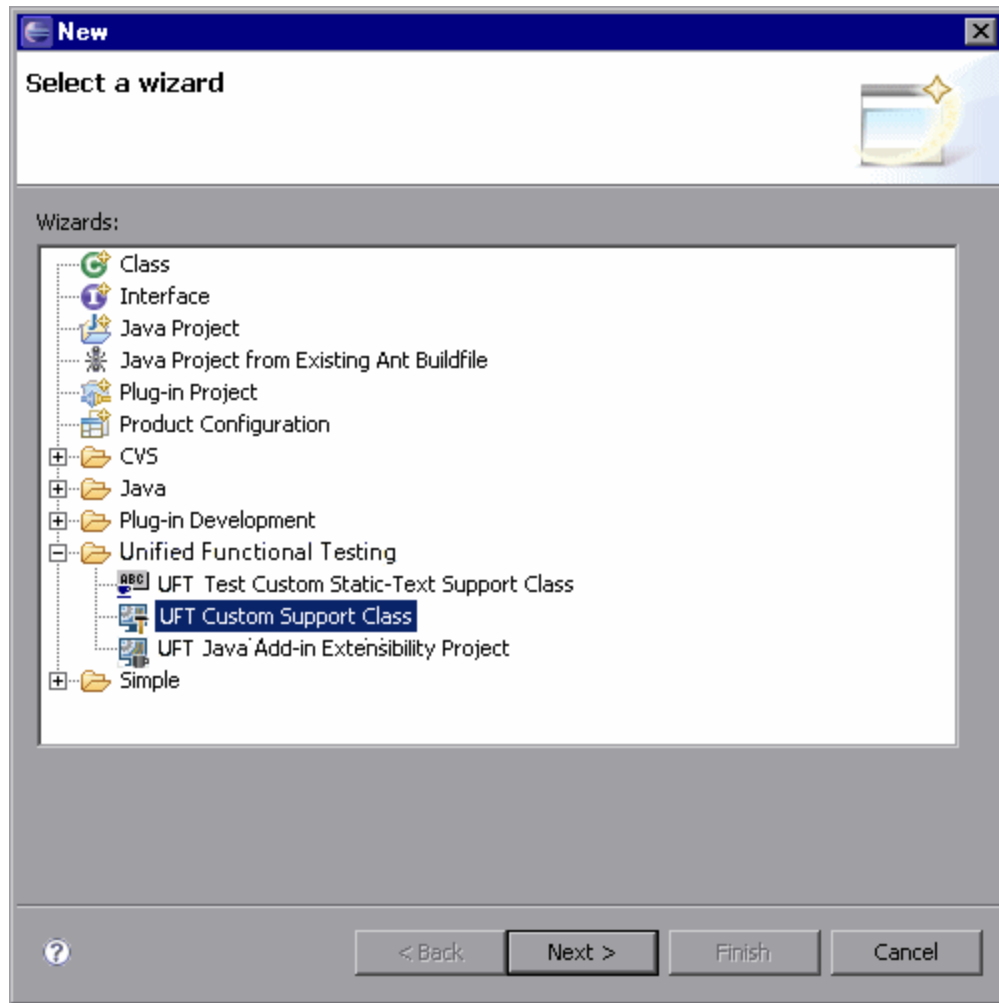
b. Expand the **Unified Functional Testing** folder and select **UFT Custom Support Class**.

c. Click **Next**. The Custom Class Selection screen opens.

2. **Select the custom class to support, and set the options for the support class.**

    a. Select the **AllLights** class in the **org.boutique.toolkit** package.
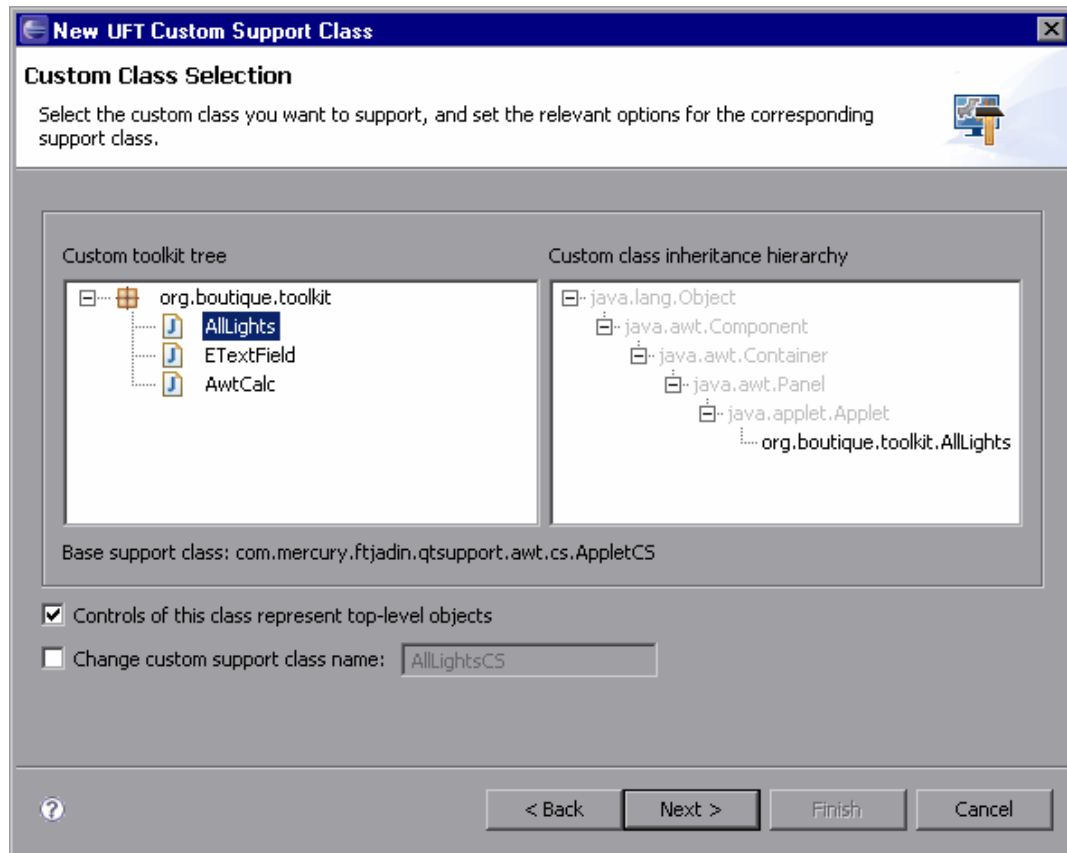
        The AllLights custom class extends java.applet.Applet, which is supported on UFT. The AllLights support class therefore extends the **Base support class**: **com.mercury.ftjadin.qtsupport.awt.cs.AppletCS**. As a result, the **Controls of this class represent top-level objects** check box is selected by default.

    b. Leave this check box selected, because you want UFT to recognize the AllLights controls as the highest Java test objects in the test object hierarchy.

    c. Keep the default custom support class name: `AllLightsCS`.

    d. 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 UFT GUI tests,
   the custom class controls are represented by test objects of this test object class.
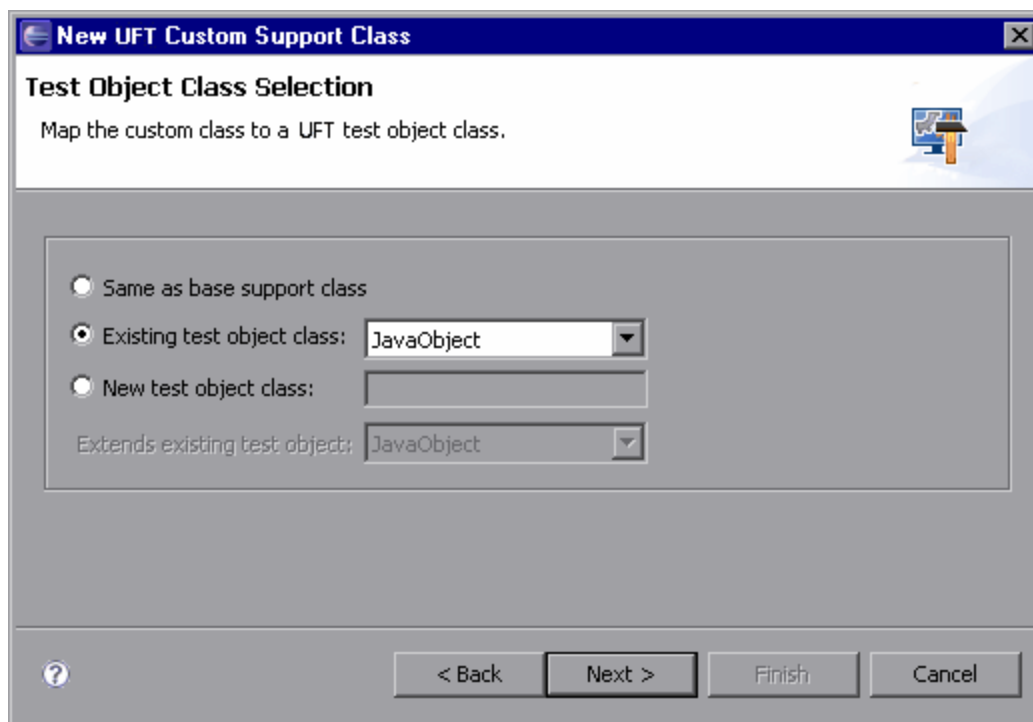
   In "Planning Support for the AllLights Control" on page 239, you decided to map the
   AllLights custom control to a new test object class, AllLights, that extends
   JavaApplet.

   a. Select the **New test object class** option and enter `AllLights` as the name for
      the test object class.

   b. In the **Extends existing test object** list, select **JavaApplet**. This list contains all
      of the Java objects that UFT currently supports. If you define new test objects
      for custom support, they are included in the list as well.

   c. 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.
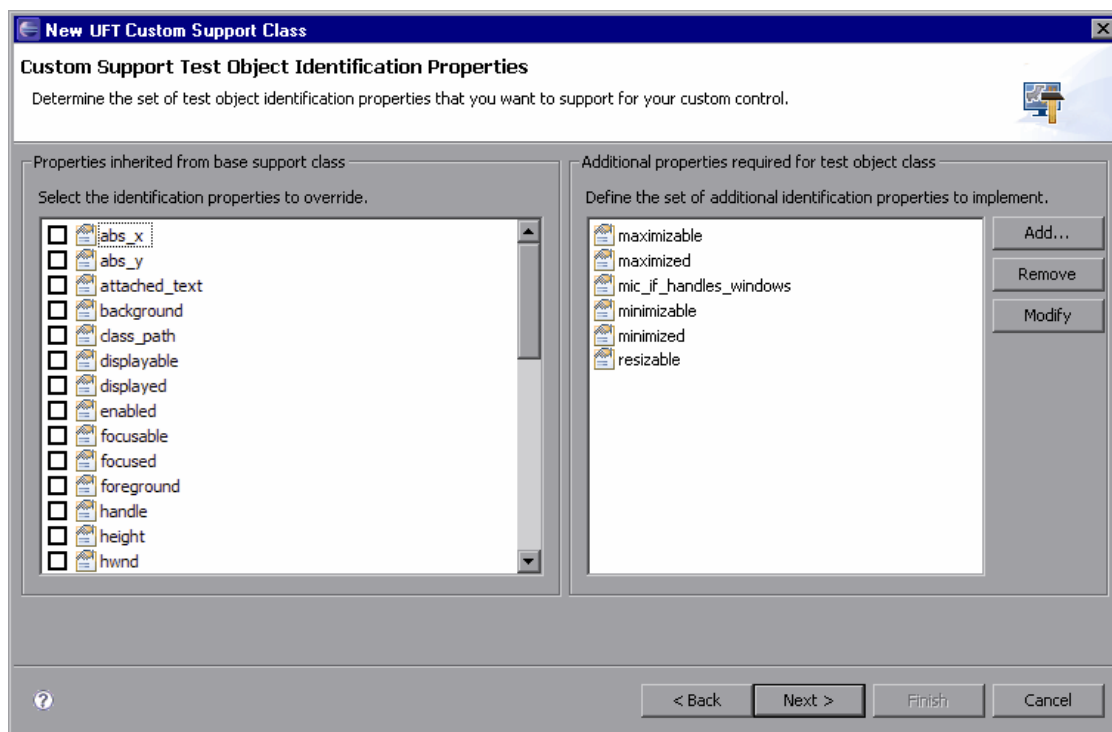
a. The left pane shows all of the identification properties whose support is implemented by AppletCS, and therefore inherited by the new AllLightsCS support class. For most of the properties in this list, the default implementation is sufficient.

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.

b. 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.
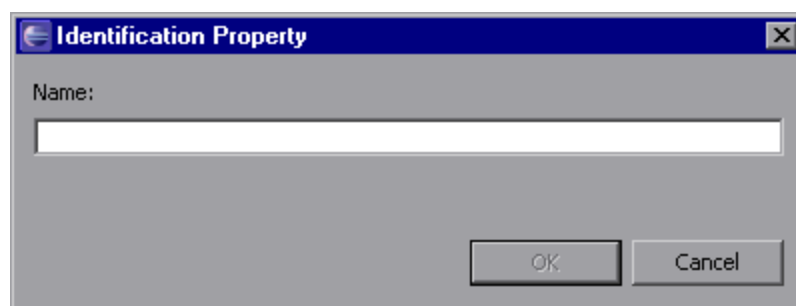
These identification properties are part of the AllLights test object class definition that is created based on the JavaApplet test object class. Removing the properties from this list means that they are not supported for AllLights controls. They will still appear in the list of identification properties shown in the UFT Object Spy, but will have no value.

c. In "Planning Support for the AllLights Control" on page 239, you decided to support new identification properties on AllLights test objects. In the next step you add these properties to the list of additional properties required for the test object class. After you finish generating the support files using the wizard, you will implement support for these properties.

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 UFT for all test objects of this class. This is the reason you are creating the new AllLights test object class.

5. **Add the new test object identification properties you want to implement in AllLightsCS.**

a. Click **Add** in the **Additional properties required for test object class** pane. The Identification Property dialog box opens.


b. In the **Name** box, enter `OnCount`. Click **OK** to add the new Identification Property to the list.

    c. Repeat this procedure to add the properties OnList and GameOver.

    d. Click **Next**. The Custom Support Test Object Methods screen opens.

6. **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.
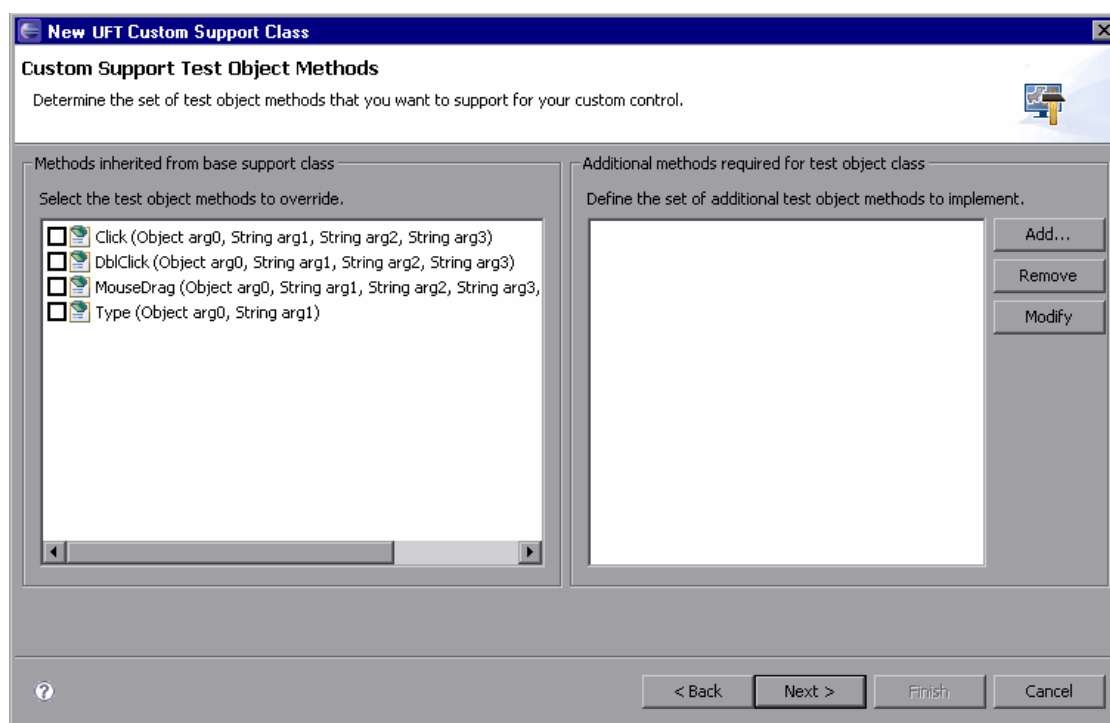
The left pane shows all of the test object methods (defined in the test object class you selected) whose support is implemented by AppletCS, and therefore inherited by AllLightsCS. There is no need to select any methods to override.

The right pane displays the test object methods that are defined for the AllLights test object class, but are not supported by AppletCS. There are no such methods currently defined.

In "Planning Support for the AllLights Control" on page 239, you decided to support new test object methods on AllLights test objects. You now need to add these

methods to the list of additional test object methods required for the test object class. After you finish generating the support files using the wizard, you will implement support for the methods you add.

The test object methods are added to the existing test object class. This means that the new methods appear in UFT for all test objects of this class, regardless of whether or not they are supported for these objects. In a UFT GUI 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.
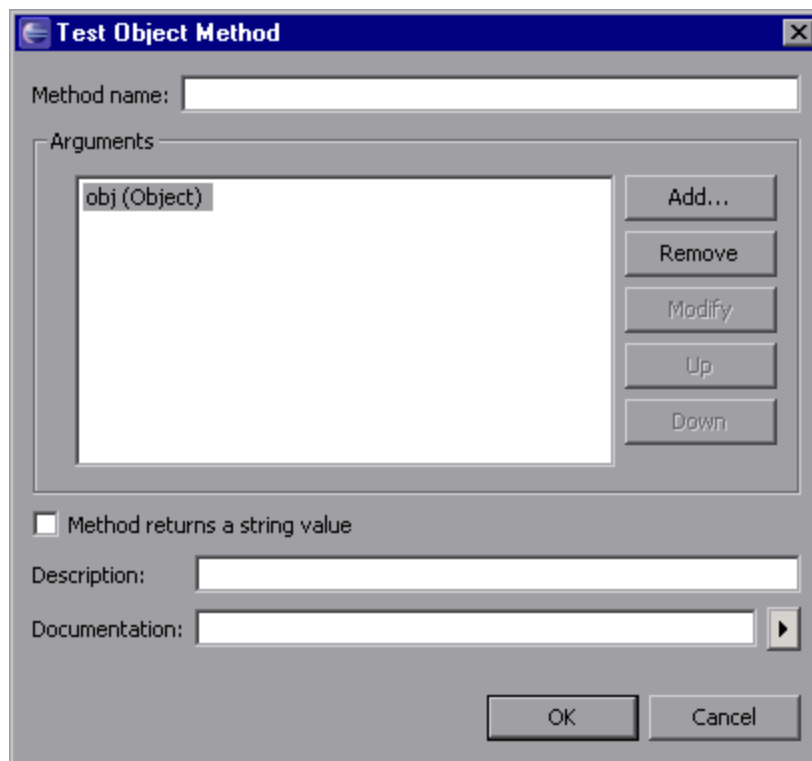
○ In the **Method Name** box, enter: `Restart.` The Restart test object method does not require any arguments other than the mandatory **obj (Object)** that represents the custom control.

○ 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.
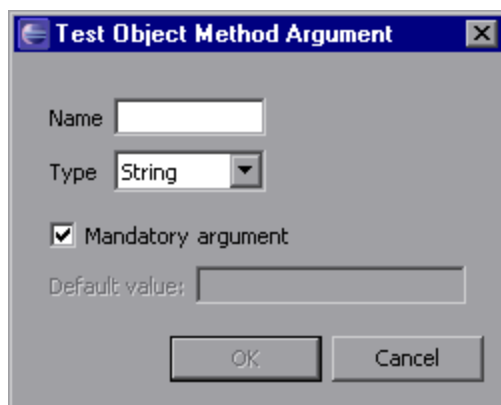
In the **Name** box, enter: `Row`.

In the **Type** box, select **Variant**. (If you select **String**, then when you add steps in UFTGUI 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. The final text in the **Documentation** box will be: `Click the light in row %a1 column %a2.`

- ○ Click **OK** to close the Test Object Method dialog box and add the ClickLight method to the list.

c. Click **Next**. The Custom Control Recording Support wizard screen opens.

7. **Determine the set of events for which to listen, 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:
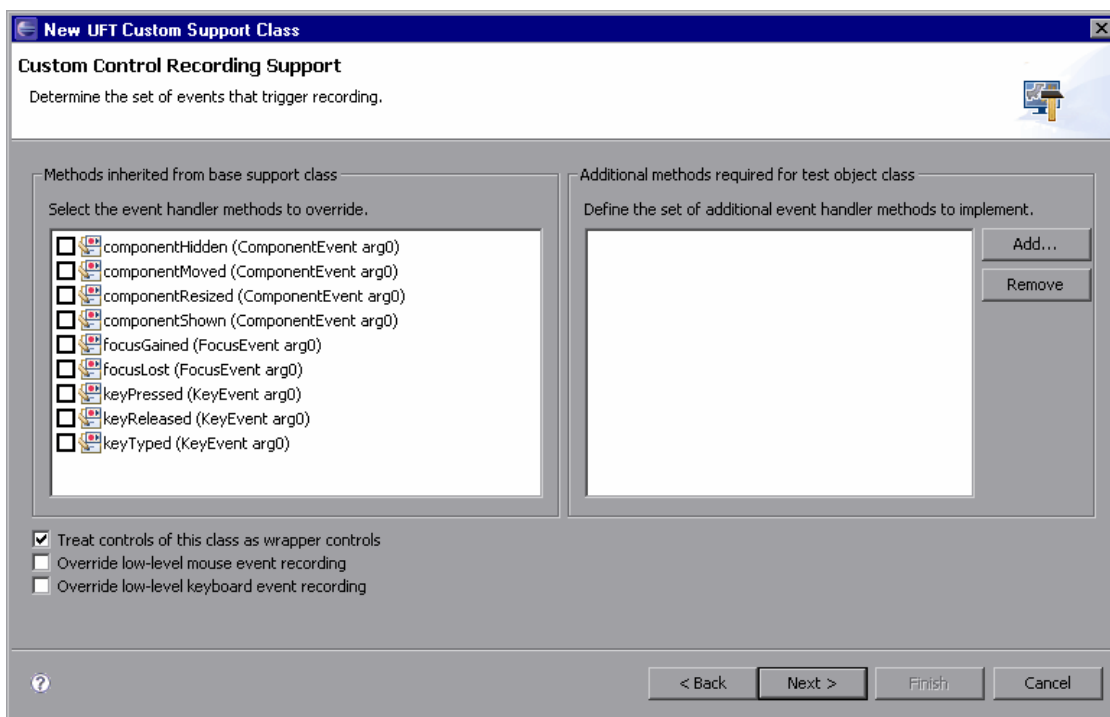
In the left pane, you can see the listeners implemented by AppletCS. You do not have to override any of these for the AllLightsCS custom support class.

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.

a. Click **Add** to add the **MouseListener**.

The Listener dialog box opens:
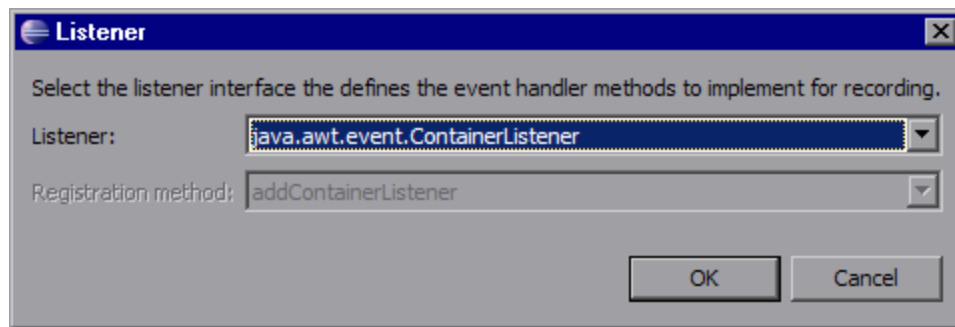
Select **java.awt.event.MouseListener** from the **Listener** list and click **OK**. The Listener dialog box closes and the **MouseListener**, and all of the event handler methods it includes, are added to the list in the right pane of the wizard screen.

b. 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.

c. Click **Next**. The New Test Object Class Details screen opens.

8. **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.
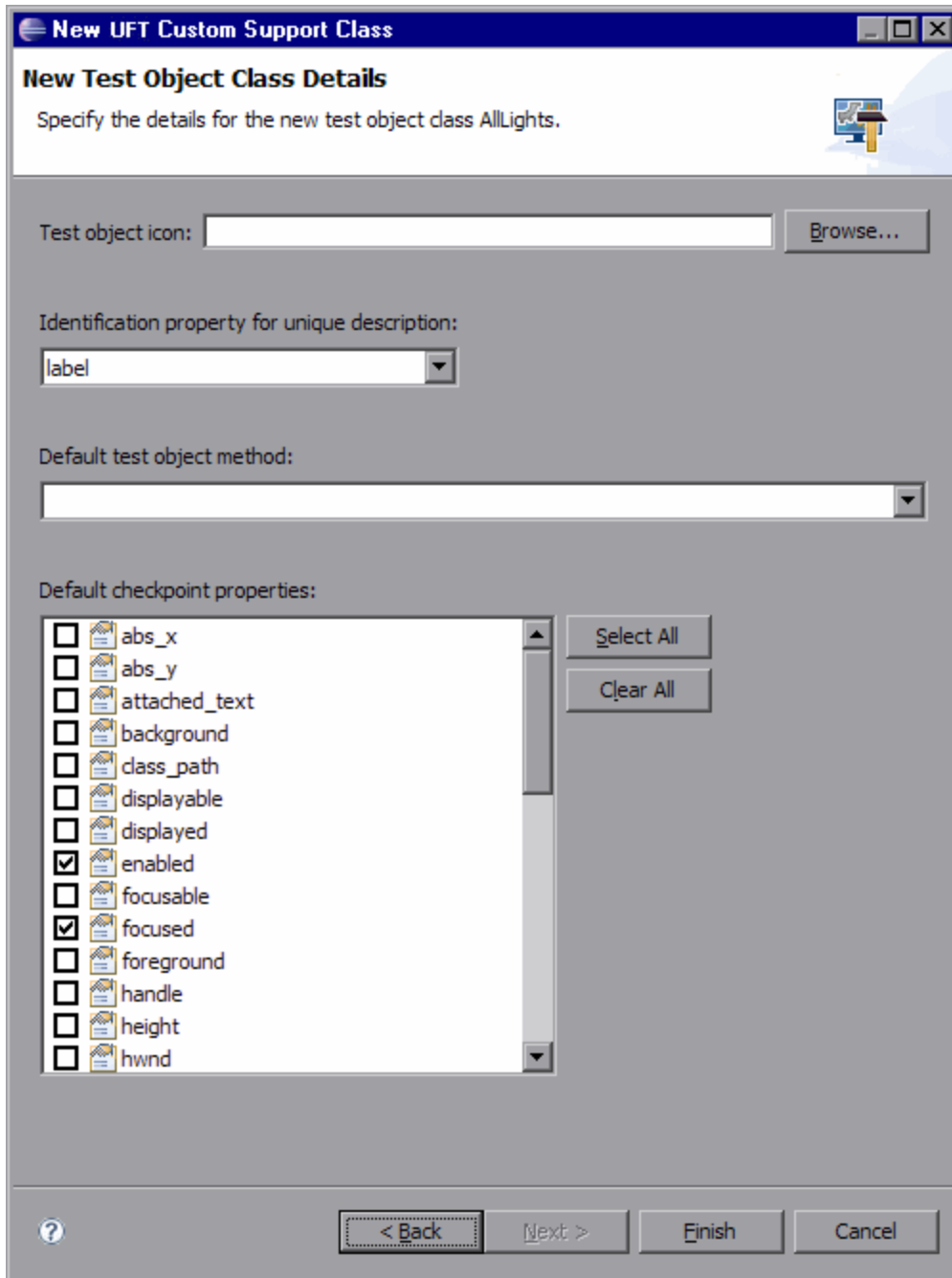
Perform the following:

a. For the **Test object icon**, click **Browse**, locate the **<UFT Java Add-in Extensibility SDK Installation folder>\samples\Javaboutique** folder, and select the **AllLights_icon.ico** file.

b. In the **Identification property for unique description box**, leave the selected **label** property.

> **Note:** If you wanted to include additional identification properties in the test object description, you would have to manually specify this in the test object configuration file. For more information, see "New Test Object Class Details Screen" on page 137.

c. In the **Default test object method** list, select **ClickLight**.

d. In the **Default checkpoint properties** box, leave the selected properties and select also the **GameOver**, **OnCount**, and **OnList** check boxes.

e. Click **Finish**. The Custom Control Support Class Summary screen opens.

9. **View the custom control support class summary.**

   Review the planned content of the custom support class, and click **OK**.
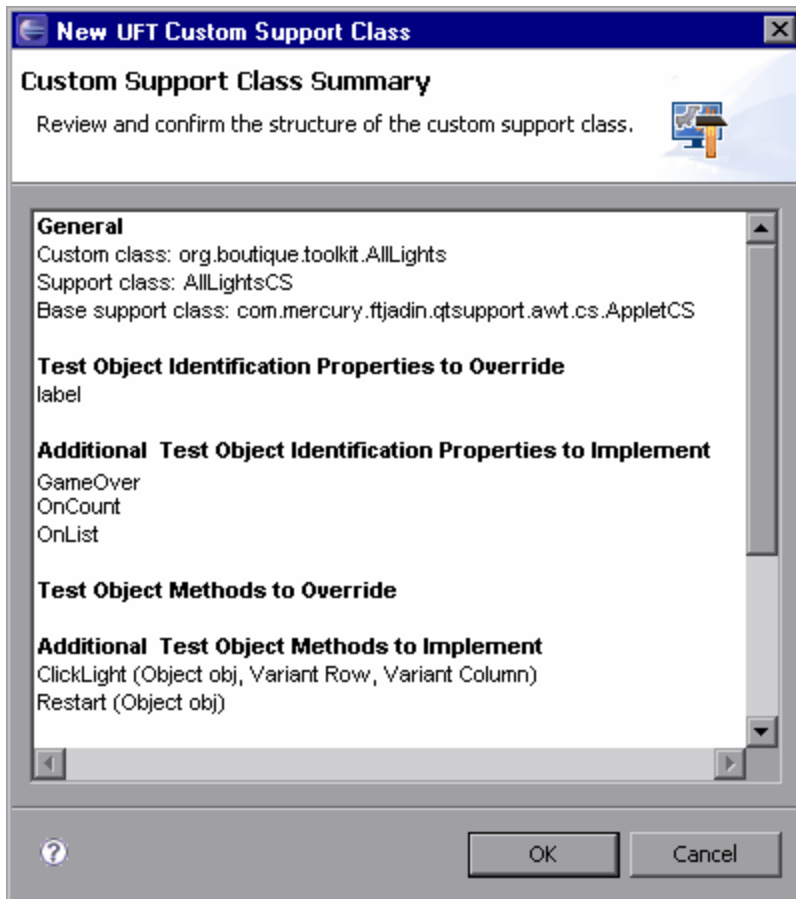
The following changes are made in the JavaboutiqueSupport project:

■ A new UFT 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, see the *UFT Java Add-in ExtensibilityToolkit
Configuration Schema Help*(available with the Java Add-in Extensibility SDK
Help).

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. Select **File > Save**, or click the **Save**
button.

Understanding the New Custom Support Files

When you completed the process of the New UFT 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 163, 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 uppercase 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, the wizard replaces uppercase letters with lowercase ones. In UFT, the identification property names are displayed in lowercase letters only.

- 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 UFT.

- The **isWindow** 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, UFT calls the **isWindow** 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, see the *UFT Test Object Schema Help* (available with the Java Add-in Extensibility SDK Help).

Locate the following elements in the test object configuration file:

- The names of the custom toolkit and the add-in to which the test object classes in this file belong (in the **TypeInformation** element):

```
PackageName="Javaboutique" AddinName="Java"
```

- The test object class that the new test object class extends (in the **ClassInfo** element):

```
BaseClassInfoName="JavaApplet"
```

- The name of the new test object class and its default test object method (in the **ClassInfo** element):

```
DefaultOperationName="ClickLight" Name="AllLights"
```

- The location of the icon file (in the **IconInfo** element):

```
IconFile="<UFT 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 (in the `<TypeInfo>` element).

- The definition of the identification properties for this test object class (in the `<IdentificationProperties>` 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 UFT **Deploy Toolkit Support** command in Eclipse to deploy the Javaboutique toolkit support to UFT. 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 UFT.**

    a. In the Eclipse Package Explorer tab, select the JavaboutiqueSupport project.

    b. Click the **Deploy Toolkit Support** button, or select
        UFT > 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 UFT installation folder. The custom support will be
        available the next time you open UFT and start the custom application.

        For more information on deploying custom toolkit support, see "Deploying and
        Running the Custom Toolkit Support" on page 77.

2. **Open UFT and load the Java Add-in and the custom toolkit support.**

    a. Open UFT. The Add-in Manager dialog box displays Javaboutique as a child of
        the **Java** Add-in in the list of available add-ins. (If the Add-in Manager dialog box
        does not open, see the *HP Unified Functional Testing Add-ins Guide* for
        instructions.)

    b. Select the check box for Javaboutique and click **OK**. UFT opens and loads the
        support you designed.

3. **Test the new custom support.**

    Repeat the steps in "Planning Support for the AllLights Control" on page 239 to
    perform the following:

    ■ Run the application. (UFT establishes its connection with an application when the
        application opens. Therefore, you must close the SampleApp application, and run
        it again.)

    ■ View the AllLights control with the UFT Object Spy.

    ■ Try to record a **Click** operation on the AllLights control.

UFT 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:
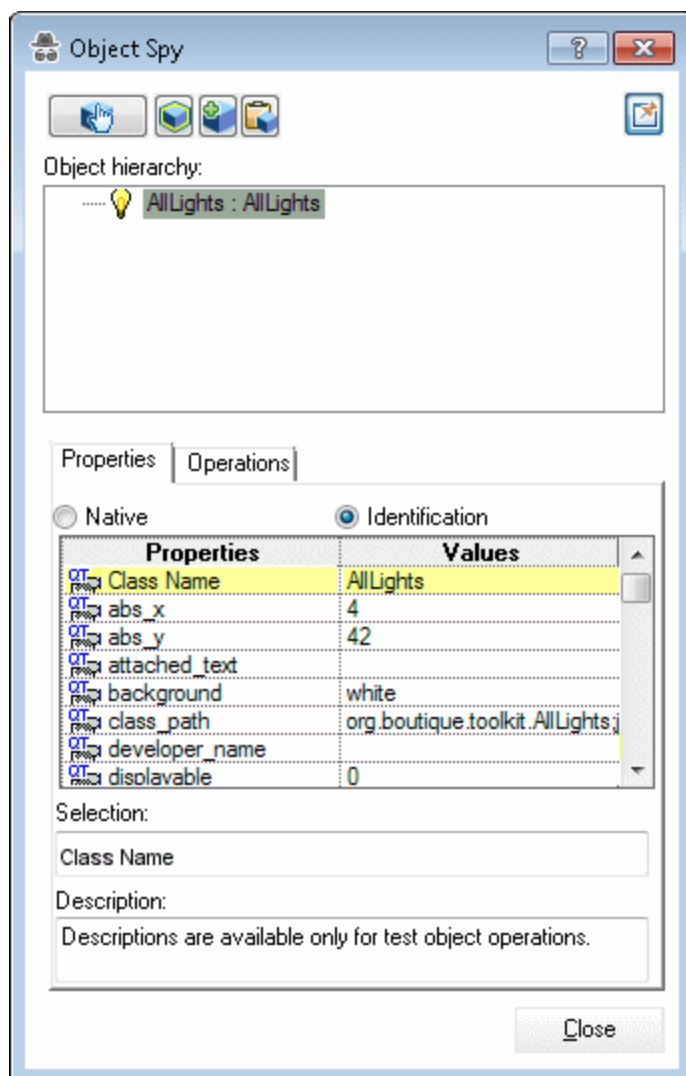
Because you have overridden the low-level recording, but have not yet
implemented the **mouseClicked (MouseEvent arg0)** event handler method, UFT
currently does not record anything when you click in the application frame.

In UFT, 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 UFT support of the AllLights control, as per your plan ("Planning Support for the AllLights Control" on page 239).

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 272)

- "Implementing Support for New Test Object Methods" (described on page 274)

- "Implementing Support for Recording" (described on page 275)

- " Testing the Completed Support" (described on page 277)

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 UFT.

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 UFT.

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";
   ```

```
}
```

Select **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 UFT 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 UFT 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**.

Select **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
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(y/40), String.valueOf(x/40)});
        }
    } catch (Throwable th) { MicAPI.logStackTrace(th);}
}
```

Note: When the wizard created the **AllLightsCS.java** file, it automatically added the `importcom.mercury.ftjadin.custom.MicAPI`, required to support this code.

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 UFT 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 86.

The **try ... catch** block prevents unnecessary activity if this code is reached when the Java application is running while UFT 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 UFT views the AllLights control.

You do not have to deploy the toolkit support to UFT again to test it because you changed only Java class files and not configuration files. You can use an open UFT session (running with the Javaboutique toolkit support loaded), 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.**

   a. Close the AllLights application and run it again.

   b. Open UFT and load the Java Add-in and the Javaboutique toolkit support.

   c. Open a GUI test and use the Object Spy to view the AllLights properties and methods. The AllLights test object is now named **Lights**.

   d. Close the Object Spy.

2. **Create and run a UFT test on the AllLights control.**

   a. Add the AllLights control to the test object repository.

   b. 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.

   c. 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.**

   a. In UFT, create a new GUI test and select **Record > 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**.

   b. Click the **Record** button or select **Record > 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.

   c. Click the **Stop** button or select **Record > 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 UFT 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 **isWindow** 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 34.

For more information on the structure and content of the test object configuration file, see the *HP UFT Test Object Schema Help* (available with the Java Add-in Extensibility SDK Help).

Send Us Feedback

Can we make this Developer Guide better?

Tell us how: sw-doc@hp.com