



Adobe **Flex™ 2**
Using the Automated Testing API



© 2007 Adobe Systems Incorporated. All rights reserved.

Using the Automated Testing API

If this guide is distributed with software that includes an end-user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end-user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Flex, Flex Builder and Flash Player are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. ActiveX and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Linux is a registered trademark of Linus Torvalds. Solaris is a registered trademark or trademark of Sun Microsystems, Inc. in the United States and other countries. All other trademarks are the property of their respective owners. Quicktest and Mercury Quicktest Professional are registered trademarks or trademarks of Mercury Interactive Corporation or its wholly-owned subsidiaries, Freshwater Software, Inc. and Mercury Interactive (Israel) Ltd. in the United States and/or other countries.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>). Macromedia Flash 8 video is powered by On2 TrueMotion video technology. © 1992-2005 On2 Technologies, Inc. All Rights Reserved. <http://www.on2.com>. This product includes software developed by the OpenSymphony Group (<http://www.opensymphony.com/>). Portions licensed from Nellymoser (www.nellymoser.com). Portions utilize Microsoft Windows Media Technologies. Copyright (c) 1999-2002 Microsoft Corporation. All Rights Reserved. Includes DVD creation technology used under license from Sonic Solutions. Copyright 1996-2005 Sonic Solutions. All Rights Reserved. This Product includes code licensed from RSA Data Security. Portions copyright Right Hemisphere, Inc. This product includes software developed by the OpenSymphony Group (<http://www.opensymphony.com/>).

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA

Notice to U.S. government end users. The software and documentation are “Commercial Items,” as that term is defined at 48 C.F.R. §2.101, consisting of “Commercial Computer Software” and “Commercial Computer Software Documentation,” as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

Chapter 1: Creating Applications for Testing	3
About the Flex Automation Package	3
Tasks and techniques for testable applications overview	5
Compiling applications for testing	6
Using run-time loading	7
Testing applications that load external libraries	9
Creating testable applications	10
Providing meaningful identification of objects	10
Avoiding duplication of objects	11
Coding containers	11
Writing the wrapper	13
Understanding the automation framework	14
About the automation interfaces	14
Automated testing workflow with QTP	15
Instrumenting events	18
Instrumenting existing events	18
Instrumenting custom components	22
Creating a delegate class	23
Using the class definitions file	25
Setting the automationName property	28
Instrumenting composite components	31
Example: Instrumenting the RandomWalk custom component for QTP	
33	
Instrumenting the RandomWalk custom component	33
Instrumenting RandomWalk events	35
Preparing RandomWalk for playback	37
Linking the delegate to an application	39
Adjusting event recording	39
Adding checkpoints	42
Chapter 2: Creating Custom Agents	43
Introduction	43
Metrics	44
Automated testing	45
Co-browsing	45

About the automation APIs	45
About the SystemManager class	47
About the AutomationManager class	48
About the Automation class	48
About the delegate classes	49
About the agent	50
Understanding the automation flow	51
Creating agents	53
Creating a recording agent	54
Using the CustomAdapter class	54
Using LiveCycle Data Services	58
Accessing user session data	59
Creating a replaying agent	62
Using the AutoQuick example	62
About the AutoQuick example	66

Creating Applications for Testing

You can create applications and components that can be tested with automated testing tools such as Mercury QuickTest Professional (QTP). This topic includes information intended for Flex developers who write applications that are then tested by Quality Control (QC) professionals who use these testing tools. For information on installing and running the Flex plug-in with QTP, QC professionals should see *Testing Flex Applications with Mercury QuickTest Professional*.

Contents

About the Flex Automation Package	3
Tasks and techniques for testable applications overview	5
Compiling applications for testing	6
Creating testable applications	10
Writing the wrapper	13
Understanding the automation framework	14
Instrumenting events	18
Instrumenting custom components	22
Instrumenting composite components	31
Example: Instrumenting the RandomWalk custom component for QTP	33

About the Flex Automation Package

The Flex Automation Package provides developers with the ability to create Flex applications that use the Automation API. You can use this API to create automation agents or to ensure that your applications are ready for testing. In addition, the Flex Automation Package includes support for Mercury QuickTest Professional (QTP) automation tool.

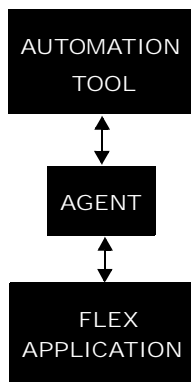
The Flex Automation Package includes the following:

- Automation libraries — The automation.swc and automation_agent.swc libraries are the implementations of the delegates for the Flex framework components. The automation_agent.swc file and its associated resource bundle are the generic agent mechanism. An agent such as QTP builds on top of these libraries.
- QTP files — The QTP files let QTP and Flex applications communicate directly. You can only use these files if you also have QTP. These files include the QTP plug-in, a QTP demonstration video, a QTP-specific environment XML file, and the QTP-specific library, qtp.swc. For more information, see *Testing Flex Applications with Mercury QuickTest Professional*.
- Samples — The samples included with the Flex Automation Package include the CustomAgent and AutoQuick samples. The CustomAgent sample shows a simple agent that records metrics information for Flex applications. The AutoQuick example includes a component that lets you record and play back interactions with a Flex application. For information about creating automation agents, see Chapter 2, “Creating Custom Agents,” on page 43.
- Uninstaller — The uninstaller uninstalls the Flex Automation Package.

When working with the automation API, you should understand the following terms:

- *automation agent* (or, simply, *agent*) — An agent facilitates communication between a Flex application and an automation tool. The Flex Automation Package includes a plugin that acts as an agent between your Flex applications and the QTP testing tool.
- *automation tool* — Automation tools are applications that use the Flex application data that is derived through the agent. These tools include QTP, Omniture, and Segue. In some cases.

The following illustration shows the relationship between a Flex application, an agent, and an automation tool.



Tasks and techniques for testable applications overview

Flex developers should review the information about tasks and techniques for creating testable applications, and then update their Flex applications accordingly. QC testing professionals who use QTP should use the documentation provided in the separate book, *Testing Flex Applications with Mercury QuickTest Professional*. That document is available for download with the Flex plug-in for QTP.

Use the following general steps to create a testable application:

1. Review the guidelines for creating testable applications. For more information, see “Creating testable applications” on page 10.
2. Build a testable application or prepare the application to use automation at run time.
 - To build a testable application, you include automation libraries at compile time. Compile the application’s SWF file with the automation.swc and automation_agent.swc files specified in the compiler’s `include-libraries` option. If your application uses charts, you must also add the automation_charts.swc file. If you are using the QTP plug-in, add the qtp.swc file. For information on the compilation process, see “Compiling applications for testing” on page 6.
 - To use automation at run time, you create a wrapper SWF file that is compiled with the automation libraries. In this wrapper SWF file, you use the SWFLoader to load the SWF file that you plan to test only at run time. For more information, see “Using run-time loading” on page 7.
3. Create an HTML wrapper with proper object naming. For more information, see “Writing the wrapper” on page 13.
4. Prepare customized components for testing. If you have custom components that extend UIComponent, make them testable. For more information, see “Instrumenting custom components” on page 22.
5. Deploy the application’s assets to a web server. Assets can include the SWF file; HTML wrapper; external assets such as theme files, graphics, and video files; module SWF files; and run-time shared libraries (RSLs). The QC professional must be able to access the main application. For more information about QTP, see *Testing Flex Applications with Mercury QuickTest Professional*.

Compiling applications for testing

You must precompile applications that you plan to test. The functional testing classes are embedded in the application at compile time, and the application has no external dependencies for automated testing at run time.

When you embed functional testing classes in your application SWF file at compile time, you increase the size of the SWF file. If the size of the SWF file is not important, you can use the same SWF file for functional testing and deployment. If the size of the SWF file is important, you typically generate two SWF files: one with functional testing classes embedded and one without.

When you precompile the Flex application for testing, you must reference the `automation.swc` and `automation_agent.swc` files in your `include-libraries` compiler option. If your application uses charts, you must also add the `automation_charts.swc` file. You might also be required to add automation tool-specific SWC files; for example, for QTP, you must also add the `qtp.swc` file to your application's library path. If you are using the AutoQuick example, you must add the `AutoQuick.swc` file.

When you create the final release version of your Flex application, you recompile the application without the references to these SWC files. For more information about using the automation SWC files, see the Flex Automation Release Notes.

If you do not deploy your application to a server, but instead request it by using the file protocol or run it from within Adobe Flex Builder, you must put the SWF file into the local-trusted sandbox. This requires configuration information that is separate from the SWF file and the wrapper. For more information that is specific to QTP, see *Testing Flex Applications with Mercury QuickTest Professional*.

To include the `automation.swc` and `automation_agent.swc` libraries, you can add them to the compiler's configuration file or as a command-line option. For the Adobe Flex 2 SDK, the configuration file is located at `flex_install_dir/frameworks/flex-config.xml`. For Adobe Flex Data Services, this file is located at `flex_install_dir/flex/WEB-INF/flex/flex-config.xml` file. Add the following code to the configuration file:

```
<include-libraries>
  ...
  <library>/libs/automation.swc</library>
  <library>/libs/automation_agent.swc</library>
</include-libraries>
```

You can also specify the location of the `automation.swc` and `automation_agent.swc` files when you use the command-line compiler with the `include-libraries` compiler option. The following example adds `automation.swc` and `automation_agent.swc` files to the application:

```
mxm1c -include-libraries+=./frameworks/libs/automation.swc;./frameworks/
  libs/automation_agent.swc MyApp.mxml
```

If your application uses charts, you must also add the `automation_charts.swc` file to the `include-libraries` compiler option.

Explicitly setting the `include-libraries` option on the command line overwrites, rather than appends, the existing libraries. As a result, if you add the `automation.swc` and `automation_agent.swc` files by using the `include-libraries` option on the command line, ensure that you use the `+=` operator. This appends rather than overwrites the existing libraries that are included.

To add automated testing support to a Flex Builder project, you also add the `automation.swc` and `automation_agent.swc` files to the `include-libraries` compiler option.

To add the SWC files to the include-libraries compiler option in Flex Builder:

1. In Flex Builder, select Project > Properties. The Properties dialog box appears.
2. Select Flex Compiler.
3. In the Additional compiler arguments field, enter the following command, and click OK.

```
-include-libraries "Flex SDK 2\frameworks\libs\automation.swc" "Flex SDK
  2\frameworks\libs\automation_agent.swc"
```

In Flex Builder, the `include-libraries` compiler option is relative to the Flex Builder installation directory; the default location of this directory on Windows is `C:\Program Files\Adobe\Flex Builder 2\`.

If your application uses charts, you must also add the `automation_charts.swc` file.

Using run-time loading

You can load support for automated testing at run time. This lets you test SWF files that are compiled without automated testing support. To do this, you create a SWF file that *does* load automated testing. In that new SWF file, you use the `SWFLoader` class to load the other SWF file. The result is that you can test the target SWF file in a testing tool such as QTP, even though the SWF file was not compiled with automated testing support.

To use run-time loading of automated testing support:

1. Create an MXML file with following code:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
```

```
<mx:SWFLoader source="filename.swf" width="100%" height="100%"/>
</mx:Application>
```

2. Replace *filename.swf* with the name of your application SWF file that you plan to test. The loaded SWF file does not have to be compiled with automated testing support. It could be any application SWF file that was compiled with Adobe Flex 2.0.1.
3. Save this file as `atTemplate.mxml`.
4. Compile the `atTemplate.mxml` file and generate an HTML wrapper for it. Ensure that you include automated testing support.
5. Instruct the QC professional to record tests by requesting the `atTemplate.html` file.

You are not required to hard code the name of the SWF file to test. Instead, you can pass it dynamically by using a query string parameter. In the following example, you pass the relative path of the SWF file that you plan to load and test by using the `automationswful` parameter:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- at/RunTimeLoader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
creationComplete="actionScriptFunction()">
  <mx:Script>
    <![CDATA[
      import flash.external.*;
      public function init():void {
        myLoader.addEventListener(IOErrorEvent.IO_ERROR,
          ioErrorHandler);
      }
      private function ioErrorHandler(event:IOErrorEvent):void {
        trace("ioErrorHandler: " + event);
      }
      public function actionScriptFunction():void {
        init();
        myLoader.source =
          Application.application.parameters.automationswful;
      }
    ]]>
  </mx:Script>
  <mx:SWFLoader id="myLoader" width="100%" height="100%"/>
</mx:Application>
```

The following example shows a URL that you could use to request the application:

```
http://localhost/automation.html?automationswful=../applications/myapp.swf
```

To use this example, you must convert the query string parameters to a `flashVars` variable in the HTML wrapper. The Adobe Flex Data Services server does this automatically for you if you request an MXML file; however, you can also do convert the query string parameters with any scripting language such as JSP, ASP.Net, or client-side JavaScript.

The following sample HTML wrapper uses JavaScript to convert the `automationswfurl` query string parameter to a `flashVars` variable:

```
<!-- saved from url=(0014)about:internet -->
<html lang="en"><head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>Runtime Loading Sample</title>
</head>
<body scroll="no" top="0" left="0" >
  <script language='javascript' charset='utf-8'>
    function getQueryVariable(variable) {
      var atquery = window.location.search.substring(1);
      var atstr = atquery.split("&");
      for (var i=0;i<atstr.length;i++) {
        var atvar = atstr[i].split("=");
        if (atvar[0] == variable) {
          return atvar[1];
        }
      }
    }

    document.write('<object classid="clsid:D27CDB6E-AE6D-11cf-96B8
-444553540000" id="automationExample" width="90%" height="90%
codebase="http://fpdownload.macromedia.com/get/flashplayer/current
/swflash.cab">');
    document.write('<param name="movie" value="automation.swf"/>');
    document.write('<param name="quality" value="high"/>');
    document.write('<param name="bgcolor" value="#869ca7"/>');
    document.write('<param name="allowScriptAccess" value="sameDomain"/>');
    document.write('<param name="flashvars" value="automationswfurl=
'+getQueryVariable("automationswfurl")+'"/>');
    document.write('</object>');
  </script>
</body>
</html>
```

For more information, see "Communicating with the Wrapper" in *Flex 2 Developer's Guide*.

Testing applications that load external libraries

Applications that load other SWF file libraries require a special setting for automated testing to function properly. A library that is loaded at run time (including run-time shared libraries (RSLs)) must be loaded into the `ApplicationDomain` of the loading application. If the SWF file used in the application is loaded in a different application domain, automated testing record and playback will not function properly.

The following example shows a library that is loaded into the same `ApplicationDomain`:

```
import flash.display.*;
```

```
import flash.net.URLRequest;
import flash.system.ApplicationDomain;
import flash.system.LoaderContext;

var loader:Loader = new Loader();

var urlReq:URLRequest = new URLRequest("RuntimeClasses.swf");
var context:LoaderContext = new LoaderContext();
context.applicationDomain = ApplicationDomain.currentDomain;
loader.load(request, context);
```

Creating testable applications

As a Flex developer, there are some techniques that you can employ to make Flex applications as “test friendly” as possible. One of the most important tasks that you can perform is to make sure that objects are identifiable in the testing tool’s scripts. This means that you should set the value of the `id` property for all controls that are tested, and ensure that you use a meaningful string for that `id` property. If you can use unique IDs for each control, the testing scripts are more readable.

Providing meaningful identification of objects

When working with testing tools such as QTP, a QC professional only sees the visual representation of objects in your application. A QC professional generally does not have access to the underlying code. When a QC professional records a script, it’s very helpful to see IDs that help the tester identify the object clearly. You should take some time to understand how testing tools interpret Flex applications and determine what names to use for the test objects in the test scripts.

In most cases, testing tools use a visual cue, such as the label of a Button control, to identify the control in the script. Sometimes, however, testing tools use the Flex `id` property of an MXML tag to identify an object in the test script; if there is no value for the `id` property, testing tools use other properties, such as the `childIndex` property.

You should give all testable MXML components an ID to ensure that the test script has a unique identifier to use when referring to that Flex control. You should also try to make these identifiers as human-readable as possible to make it easier for a QC professional to identify that object in the testing script. For example, set the `id` property of a Panel container inside a TabNavigator to `submit_panel` rather than `panel1` or `p1`.

In some cases, agents do not use the `id` property, but it is a good practice to include it to avoid naming collisions or confusion. For more information about how QTP identifies Flex objects, see *Testing Flex Applications with Mercury QuickTest Professional*.

You should set the value of the `automationName` property for all objects that are part of the application's test. The value of this property appears in the testing scripts. Providing a meaningful name makes it easier for QC professionals to identify that object. For more information about using the `automationName` property, see “Setting the `automationName` property” on page 28.

Avoiding duplication of objects

Automation agents rely on the fact that some properties of object instances should not be changed during run time. If you change the Flex component property that is used by the agent as the object name at run time unexpected results can occur.

For example, if you create a Button control without an `automationName` property, and you do not initially set the value of its `label` property, and then later set the value of the `label` property, the agent might get confused. This is because agents often use the value of the `label` property of Button controls to identify an object in its object repository if the `automationName` property is not set. If you later set the value of the `label` property, or change the value of an existing `label` while the QC professional is recording a test, an automation tool such as QTP will create a new object in its repository instead of using the existing reference.

As a result, you should try to understand what properties are used to identify objects in the agent, and try to avoid changing those properties at run time. You should set unique, human-readable `id` or `automationName` properties for all objects that are included in the recorded script.

Coding containers

Containers are different from other kinds of controls because they are used both to record user interactions (such as when a user moves to the next pane in an Accordion container) and to provide unique locations for controls in the testing scripts.

Adding and removing containers from the automation hierarchy

In general, the automated testing feature reduces the amount of detail about nested containers in its scripts. It removes containers that have no impact on the results of the test or on the identification of the controls from the script. This applies to containers that are used exclusively for layout, such as the HBox, VBox and Canvas containers, except when they are being used in multiple-view navigator containers such as the ViewStack, TabNavigator or Accordion containers. In these cases, they are added to the automation hierarchy to provide navigation.

Many composite components use containers, such as Canvas or VBox, to organize their children. These containers do not have any visible impact on the application. So, you usually exclude these containers from being tested because there is no user interaction and no visual need for their operations to be recordable. By excluding a container from being tested, it does not clutter the test scripts and make them harder to read.

To exclude a container from being recorded (but not exclude its children), set the container's `showInAutomationHierarchy` property to `false`. This property is defined by the `UIComponent` class, so all containers that subclass `UIComponent` have this property. Children of containers that are not visible in the hierarchy appear as children of the next highest visible parent.

The default value of the `showInAutomationHierarchy` property depends on the type of container. For containers such as `Panel`, `Accordion`, `Application`, `DividedBox`, and `Form`, the default value is `true`; for other containers, such as `Canvas`, `HBox`, `VBox`, and `FormItem`, the default value is `false`.

The following example forces the VBox containers to be included in the test script's hierarchy:

```
<?xml version="1.0"?>
<!-- at/NestedButton.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Panel title="ComboBox Control Example">
    <mx:HBox id="hb">
      <mx:VBox id="vb1" showInAutomationHierarchy="true">
        <mx:Canvas id="c1">
          <mx:Button id="b1"
            automationName="Nested Button 1"
            label="Click Me"
          />
        </mx:Canvas>
      </mx:VBox>
      <mx:VBox id="vb2" showInAutomationHierarchy="true">
        <mx:Canvas id="c2">
          <mx:Button id="b2"
            automationName="Nested Button 2"
            label="Click Me 2"
          />
        </mx:Canvas>
      </mx:VBox>
    </mx:HBox>
  </mx:Panel>
</mx:Application>
```

Working with multiview containers

You should avoid using the same label on multiple tabs in multiview containers, such as TabNavigator and Accordion containers. Although it is possible to use the same labels, this is generally not an acceptable UI design practice and can cause problems with control identification in your testing environment. QTP, for example, uses the `label` properties to identify those views to testers. When two labels are the same, QTP uses different strategies to uniquely identify the tabs, which can result in a confusing name list.

Also, dynamically adding children to multiview containers can cause delays that might confuse the testing tool. You should try to avoid this.

Writing the wrapper

In most cases, the testing tool requests a file from a web server that embeds the Flex application. This file, known as the *wrapper*, is often written in HTML, but can also be a JSP, ASP, or other file that browsers interpret. You can request the SWF file directly in the testing tool by using the file protocol, but then you must ensure that the SWF file is trusted.

If you are using Adobe Flex Data Services or Flex Builder, you can generate a wrapper automatically. If you are using the Flex Software Development Kit (SDK), you can use the wrapper templates in the *flex_sdk/html_templates* directory to create a wrapper for your application.

When using a wrapper, your wrapper's `<object>` tag must have an `id` attribute, and the value of the `id` attribute can not contain any periods or hyphens. The convention is to set the `id` to match the name of the root MXML file in the application.

When you use Flex Builder to generate a wrapper, the value of the `id` attribute is the name of the root application file. You do not have to make any changes to this attribute.

When you generate a wrapper with the Flex Data Services server, the object tag's `id` attribute is valid. The following example shows the default object tag for a file named `MainApp.swf`:

```
<object id='MainApp' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
  codebase='http://fpdownload.macromedia.com/get/flashplayer/current/
  swflash.cab' height='600' width='600'>
```

TIP

You are not required to change the value of the name in the `<embed>` tag because `<embed>` is used by Netscape-based browsers that do not support the testing feature. The `<object>` tag is used by Microsoft Internet Explorer.

Ensure that you check that the object tag's `id` attribute is the same in the `<script>` and the `<noscript>` blocks of the wrapper.

Understanding the automation framework

This section describes the automation interfaces and shows the flow of the automation framework as you initialize, record, and play back an automatable event.

About the automation interfaces

The Flex class hierarchy includes the following interfaces in the `mx.automation.*` package that enable automation:

Interface	Description
<code>IAutomationClass</code>	Defines the interface for a component class descriptor.
<code>IAutomationEnvironment</code>	Provides information about the objects and properties of automatable components needed for communicating with agents.

Interface	Description
IAutomationEventDescriptor	Defines the interface for an event descriptor.
IAutomationManager	Defines the interface expected from an AutomationManager by the automation module.
IAutomationMethodDescriptor	Defines the interface for a method descriptor.
IAutomationObject	Defines the interface for a delegate object implementing automation for a component.
IAutomationObjectHelper	Provides helper methods for the IAutomationObject interface.
IAutomationPropertyDescriptor	Describes a property of a test object as well as properties of an event object.

About the IAutomationObjectHelper

The IAutomationObjectHelper interface helps the components accomplish the following tasks:

- Replay mouse and keyboard events; the helper generates proper sequence of player level mouse and key events.
- Generate AutomationIDPart for a child: AutomationIDPart would be requested by the Automation for representing a component instance to agents.
- Find a child matching a AutomationIDPart: Automation would request the component to locate a child matching the AutomationIDPart supplied by an agent to it.
- Avoid synchronization issues: Agents invoke methods on Automation requesting operations on components in a sequence. Components may not be ready all the time to perform operations.

For example, an agent can invoke `comboBox.Open`, `comboBox.select "Item1"` operations in a sequence. Because it takes time for the drop-down list to open and initialize, it is not possible to run the select operation immediately. You can place a wait request during the open operation execution. The wait request should provide a function for automation, which can be invoked to check the ComboBox control's readiness before invoking the next operation.

Automated testing workflow with QTP

This section describes the workflow for testing with the QTP automation tool.

Before you automate custom components, you might find it helpful to see the order of events during which Flex's automation framework initializes, records, and plays backs events with QTP. You should keep in mind that the QTP adapter class's implementation is only one way to use the automation API for automated testing.

The following sections show the steps involved.

Automated testing initialization

1. The user launches the Flex application. Automation initialization code associates component delegate classes with component classes. Component delegate classes implement the `IAutomationObject` interface.
2. `AutomationManager` is a mixin. Its instance is created in the mixin `init()` method.
3. The `SystemManager` initializes the application. Component instances and their corresponding delegate instances are created. Delegate instances add event listeners for events of interest.
4. `QTPAgent` class is a mixin. In its `init()` method, it registers itself for `SystemManager.APPLICATION_COMPLETE` event. On receiving the event, it creates a `QTPAdapter` object.
5. `QTPAdapter` sets up the `ExternalInterface` function map. `QTPAdapter` loads the QTP Plugin DLLs by creating the ActiveX object to communicate with QTP.
6. The `QTPAdapter` requests the XML environment information from the plugin and passes it to the `AutomationManager`.
7. The XML information is stored in a chain of `AutomationClass`, `AutomationMethodDescriptor`, and `AutomationPropertyDescriptor` objects.

Automated testing recording

1. The user clicks the Record button in QTP.
2. QTP calls the `QTPAdapter.beginRecording()` method. `QTPAdapter` adds a listener for `AutomationRecordEvent.RECORD` from the `AutomationManager`.
3. The `QTPAdapter` notifies `AutomationManager` about this by calling the `beginRecording()` method. The `AutomationManager` adds a listener for the `AutomationRecordEvent.RECORD` event from the `SystemManager`.
4. The user interacts with the application. In this example, suppose the user clicks a Button control.
5. The `ButtonDelegate.clickEventHandler()` method dispatches an `AutomationRecordEvent` event with the `click` event and `Button` instance as properties.

6. The `AutomationManager.record` event handler determines which properties of the `click` event to store, based on the XML environment information. It converts the values into proper type or format. It dispatches the `record` event.
7. The `QTPAdapter` event handler receives the event. It calls the `AutomationManager.createID()` method to create the `AutomationID` object of the button. This object provides a structure for object identification.

The `AutomationID` structure is an array of `AutomationIDParts`. An `AutomationIDPart` is created by using `IAutomationObject`. (The `UIComponent.id`, `automationName`, `automationValue`, `childIndex`, and `label` properties of the `Button` control are read and stored in the object. The `label` property is used because the XML information specifies that this property can be used for identification for the `Button`.)
8. The `QTPAdapter` uses the `AutomationManager.getParent()` method to get the logical parent of `Button`. The `AutomationIDPart` objects of parent controls are collected at each level up to the application level.
9. All these `AutomationIDParts` are made part of an `AutomationID` object.
10. The `QTPAdapter` sends the information in a call to `QTP`.
11. At this point, `QTP` might call the `AutomationManager.getProperties()` method to get property values of `Button`. The property type information and codec that should be used to modify the value format are gotten from the `AutomationPropertyDescriptor`.
12. User stops recording. This is propagated by a call to the `QTPAdapter.endRecording()` method.

Automated testing playback

1. The user clicks the `Playback` button in `QTP`.
2. The `QTPAdapter.findObject()` method is called to determine whether the object on which the event has to be played back can be found. The `AutomationID` object is built from the XML data received. The `AutomationManager.resolveIDToSingleObject()` method is invoked to see if `QTP` can find one unique object matching the `AutomationID`. The `AutomationManager.getChildren()` method is invoked from application level to find the child object. The `IAutomationObject.numAutomationChildren` property and the `IAutomationObject.getAutomationChildAt()` method are used to navigate the application.
3. The `AutomationManager.isSynchronized()` and `AutomationManager.isVisible()` methods ensure that the object is fully initialized and is visible so that it can receive the event.

4. QTP invokes the `QTPAdapter.run()` method to play back the event. The `AutomationManager.replayAutomatableEvent()` method is called to replay the event.
5. The `AutomationMethodDescriptor` for the `click` event on the `Button` is used to copy the property values (if any).
6. The `AutomationManager.replayAutomatableEvent()` method invokes the `IAutomationObject.replayAutomatableEvent()` method on the delegate class. The delegate uses the `IAutomationObjectHelper.replayMouseEvent()` method (or one of the other replay methods, such as `replayKeyboardEvent()`) to play back the event.
7. If there are check points recorded in QTP, the `AutomationManager.getProperties()` method is invoked to verify the values.

Instrumenting events

When you extend Flex components that are already instrumented, you do not have to change anything to ensure that those components' events can be recorded by a testing tool. For example, if you extend a `Button` class, the class still dispatches the automation events when the `Button` is clicked, unless you override the `Button` control's default event dispatching behavior.

Automation events (sometimes known in automation tools such as QTP as *operations*) are not the same as Flex events. Flex must dispatch an automation event as a separate action. Flex dispatches them at the same time as Flex events, and uses the same event classes, but you must decide whether to make a Flex event visible to the automation tool.

Not all events on a control are instrumented. You can instrument additional events by using the instructions in “Instrumenting existing events” on page 18.

If you change the instrumentation of a component, you must edit that component's entry in the `TEAFlex.xml` file. This is described in “Using the class definitions file” on page 25.

Instrumenting existing events

Events have different levels of relevance for the QC professional. For example, a QC professional is generally interested in recording and playing back a `click` event on a `Button` control. The QC professional is not generally interested in recording all the events that occur when a user clicks the `Button`, such as the `mouseover`, `mousedown`, `mouseup`, and `mouseout` events. For this reason, when a tester clicks on a `Button` control with the mouse, testing tools only record and play back the `click` event for the `Button` control and not the other lower-level events.

There are some circumstances where you would want to record events that are normally ignored by the testing tool. But the testing tool's object model only records events that represent the end-user's gesture (such as a click or a drag and drop). This makes a script more readable and it also makes the script robust enough so that it does not fail if you change the application slightly. So, you should carefully consider whether you add a new event to be tested or you can rely on events in the existing object model.

You can see a list of events that the QTP automation tool can record for each Flex component in the *QTP Object Type Information* document. The Button control, for example, supports the following operations:

- ChangeFocus
- Click
- MouseMove
- SetFocus
- Type

All of these events except for `MouseMove` are automatically recorded by QTP by default. The QC professional must explicitly add the `MouseMove` event to their QTP script for QTP to play back the event.

However, you can alter the behavior your application so that this event is recorded by the testing tool. To add a new event to be tested, you override the `replayAutomatableEvent()` method of the `IAutomationObject` interface. Because `UIComponent` implements this interface, all subclasses of `UIComponent` (which include all visible Flex controls) can override this method. To override the `replayAutomatableEvent()` method, you create a custom class, and override the method in that class.

The `replayAutomatableEvent()` method has the following signature:

```
public function replayAutomatableEvent(event:Event):Boolean
```

The `event` argument is the `Event` object that is being dispatched. In general, you pass the `Event` object that triggered the event. Where possible, you pass the specific event, such as a `MouseEvent`, rather than the generic `Event` object.

The following example shows a custom Button control that overrides the `replayAutomatableEvent()` method. This method checks for the `mouseMove` event and calls the `replayMouseEvent()` method if it finds that event. Otherwise, it calls its superclass's `replayAutomatableEvent()` method.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- at/CustomButton.mxml -->
<mx:Button xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import flash.events.Event;
            import flash.events.MouseEvent;
            import mx.automation.Automation;
            import mx.automation.IAutomationObjectHelper;

            override public function
                replayAutomatableEvent(event:Event):Boolean {

                trace('in replayAutomatableEvent()');

                var help:IAutomationObjectHelper =
                    Automation.automationObjectHelper;

                if (event is MouseEvent &&
                    event.type == MouseEvent.MOUSE_MOVE) {
                    return help.replayMouseEvent(this, MouseEvent(event));
                } else {
                    return super.replayAutomatableEvent(event);
                }
            }
        ]]>
    </mx:Script>
</mx:Button>
```

In the application, you call the `AutomationManager`'s `recordAutomatableEvent()` method when the user moves the mouse over the button. The following application uses this custom class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- at/ButtonApp.mx.xml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:ns1="*"
initialize="doInit()">
  <mx:Script>
    <![CDATA[
      import mx.automation.*;

      public function doInit():void {
        b1.addEventListener(MouseEvent.CLICK,
          dispatchLowLevelEvent);
      }

      public function dispatchLowLevelEvent(e:MouseEvent):void {
        var help:IAutomationManager = Automation.automationManager;
        help.recordAutomatableEvent(b1,e,false);
      }
    ]]>
  </mx:Script>

  <ns1:CustomButton id="b1"
    tooltip="Mouse moved over"
    label="CustomButton"
  />

</mx:Application>
```

If the event is not one that is currently recordable, therefore, you also must define the new event for the agent. For QTP, the class definitions are in the `TEAFlex.xml` file. Automation tools can use files like this to define the events, properties, and arguments for each class of test object. For more information, see “Instrumenting events” on page 18. For example, if you wanted to add support for the `mouseover` event, you would add the following to the `FlexButton`'s entry in the `TEAFlex.xml` file:

```
<Operation Name="MouseOver" PropertyType="Method"
  ExposureLevel="CommonUsed">
  <Implementation Class="flash.events::MouseEvent" Type="mouseover"/>
  <Argument Name="keyModifier" IsMandatory="false" DefaultValue="0">
    <Type VariantType="Enumeration"
      ListOfValuesName="FlexKeyModifierValues"
      Codec="keyModifier"
    />
    <Description>Occurs when the user moves mouse over the
      component</Description>
  </Argument>
</Operation>
```

In the preceding example, however, the `mouseMove` event is already in the `FlexButton` control's entry in that file, so no editing is necessary. The difference now is that the QC professional does not have to explicitly add the event to their script. After you compile this application and deploy the new `TEAFlex.xml` file to the QTP testing environment, QTP records the `mouseMove` event for all of the `CustomButton` objects.

For more information on using a class definitions file, see “Using the class definitions file” on page 25.

Instrumenting custom components

The process of creating a custom component that supports automated testing is called instrumentation. Flex framework components are instrumented by attaching a delegate class to each component at run time. The *delegate class* defines the methods and properties required to perform instrumentation.

If you extend an existing component that is instrumented, such as a `Button` control, you inherit its parent's instrumentation, and are not required to do anything else to make that component testable. If you create a component that inherits from `UIComponent`, you must instrument that class in one of the following ways:

- Create a delegate class that implements the required interfaces.
- Add testing-related code to the component.

You usually instrument components by creating delegate classes. You can also instrument components by adding automation code inside the components, but this is not a recommended practice. It creates tighter coupling between automated testing code and component code, and it forces the automated testing code to be included in a production SWF file.

In both methods of instrumenting a component, you must specify any new events to the agent. For QTP, you must add your new component's information to a class definitions XML file so that QTP recognizes that component. For more information about this file, see “Using the class definitions file” on page 25.

Consider the following additional factors when you instrument custom components:

- **Composition.** When instrumenting components, you must consider whether the component is a simple component or a composite component. Composite components are components made up of several other components. For example, a `TitleWindow` that contains form elements is a composite component.

- Container hierarchy. You should understand how containers are viewed in the automation hierarchy so that the QC professional can easily test the components. Also, you should be aware that you can manipulate the hierarchy to better suit your application by setting some automation-related properties.
- Automation names. Custom components sometimes have ambiguous or unclear default automation names. The ambiguous name makes it more difficult in automation tools to determine what component the a script is referring to. Component authors can manually set the value of the `automationName` property for all components except item renderers. For item renderers, use the `automationValue`.

Creating a delegate class

To instrument custom components with a delegate, you must do the following:

- Create a delegate class that implements the required interfaces. In most cases, you extend the `UIComponentAutomationImpl` class. You can instrument any component that implements `IUIComponent`.
- Register the delegate class with the `AutomationManager`.
- Define the component in a class definitions XML file.

The delegate class is a separate class that is not embedded in the component code. This helps to reduce the component class size and also keeps automated testing code out of the final production SWF file. All Flex controls have their own delegate classes. These classes are in the `mx.automation.delegates.*` package. The class names follow a pattern of *ClassnameAutomationImpl*. For example, the delegate class for a Button control is `mx.automation.delegates.controls.ButtonAutomationImpl`.

To instrument with a delegate class:

1. Create a delegate class.
2. Mark the delegate class as a mixin by using the `[Mixin]` metadata keyword.
3. Register the delegate with the `AutomationManager` by calling the `AutomationManager.registerDelegateClass()` method in the `init()` method. The following code is a simple example:

```
[Mixin]
public class MyCompDelegate {
    public static init(root:DisplayObject):void {
        // Pass the component and delegate class information.
        AutomationManager.registerDelegateClass(MyComp, MyCompDelegate);
    }
}
```

You pass the custom class and the delegate class to the `registerDelegateClass()` method.

4. Add the following code to your delegate class:
 - a. Override the getter for the `automationName` property and define its value. This is the name of the object as it usually appears in automation tools such as QTP. If you are defining an item renderer, use the `automationValue` property instead.
 - b. Override the getter for the `automationValue` property and define its value. This is the value of the object in automation tools such as QTP.
 - c. In the constructor, add event listeners for events that the automation tool records.
 - d. Override the `replayAutomatableEvent()` method. The `AutomationManager` calls this method for replaying events. In this method, return whether the replay was successful. You can use methods of the helper classes to replay common events. For examples of delegates, see the source code for the Flex controls in the `mx.automation.delegates.*` packages.

5. Link the delegate class with the application SWF file in one of these ways:

- Add the following `includes` compiler option and link in the delegate class:

```
mxmhc -includes MyCompDelegate -- FlexApp.mxml
```
- Build a SWC file for the delegate class by using the `compc` component compiler:

```
compc -source-path+=. -include-classes MyCompDelegate -output MyComp.swc
```

Then include this SWC file with your Flex application by using the following `include-libraries` compiler option:

```
mxmhc -include-libraries MyComp.SWC -- FlexApp.mxml
```

This approach is useful if you have many components and delegate classes and want to include them as a single file.

6. After you compile your Flex application with the new delegate class, you must define the new interaction for the agent and the automation tool. For QTP, you must add the new component to QTP's custom class definition XML file. For more information, see "Using the class definitions file" on page 25.

For an example that shows how to instrument a custom component, see "Example: Instrumenting the `RandomWalk` custom component for QTP" on page 33.

Using the class definitions file

The class definitions file contains information about all instrumented Flex components. This file provides information about the components to the automation agent, including what events can be recorded and played back, the name of the component, and the properties that can be tested.

An example of a class definitions file is the TEAFlex.xml file, which is specific to the QTP automation tool. This file is included in the Flex Automation Package. Another example is the FlexEnv.xml file included in the AutoQuick example. These files are very similar, but represent only one way that class definitions can be represented to the agent and automation tool.

The TEAFlex.xml file is located in the “*QTP_plugin_install\Flex 2 Plug-in for Mercury QuickTest Pro*” directory. QTP recognizes any file in that directory that matches the pattern TEAFlex*.xml, where * can be any string. This directory also contains a TEAFlexCustom.xml file that you can use as a starting point for adding custom component definitions.

The TEAFlex.xml and FlexEnv.xml class definitions files describe instrumented components with the following basic structure:

```
<TypeInfo>
  <ClassInfo>
    <Description/>
    <Implementation/>
    <TypeInfo>
      <Operation/>
      ...
    </TypeInfo>
    <Properties>
      <Property/>
      ...
    </Properties>
  </ClassInfo>
</TypeInfo>
```

The top level tag is `<TypeInfo>`. You define a new class that uses the `<ClassInfo>` tag, which is a child tag of the `<TypeInfo>` tag. The `<ClassInfo>` tag has child tags that further define the instrumented classes. The following table describes these tags:

Tag	Description
ClassInfo	Defines the class that is instrumented, for example, <code>FlexButton</code> . This is the name that the automation tools use for the <code>Button</code> control. Attributes of this tag include <code>Name</code> , <code>GenericTypeID</code> , <code>Extends</code> , and <code>SupportsTabularData</code> .
Description	Defines the text that appears in the automation tool to define the component. This is not implemented in the <code>AutoQuick</code> example, but is implemented for <code>QTP</code> .
Implementation	Defines the class name, as it is known by the <code>Flex</code> compiler, for example, <code>Button</code> or <code>MyComponent</code> .
TypeInfo	Defines events for this class. Each event is defined in an <code><Operation></code> child tag, which has two child tags: <ul style="list-style-type: none"> • The <code><Implementation></code> child tag associates the operation with the actual event. • Each operation can also define properties of the event object by using an <code><Argument></code> child tag.
Properties	Defines properties of the class. Each property is defined in a <code><Property></code> child tag. Inside this tag, you define the property's type, name, and description. For each <code>Property</code> , if the <code>ForDescription</code> attribute is <code>true</code> , the property is used to uniquely identify a component instance in the automation tool; for example, the <code>label</code> property of a <code>Button</code> control. <code>QTP</code> lists this property as part of the object in <code>QTP</code> object repository. If the <code>ForVerification</code> attribute is <code>true</code> , the property is visible in the properties dialog box in <code>QTP</code> . If the <code>ForDefaultVerification</code> tag is <code>true</code> , the property appears selected by default in the dialog box in <code>QTP</code> . This results in verification of the property value in the checkpoint.

The following example adds a new component, `MyComponent`, to the class definition file. This component has one instrumented event, `click`:

```
<TypeInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="ClassesDefinitions.xsd"
  Priority="0" PackageName="TEA" Load="true" id="Flex" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ClassInfo Name="MyComponent" GenericTypeID="mycomponent"
    Extends="FlexObject" SupportsTabularData="false">
    <Description>FlexMyComponent</Description>
    <Implementation Class="MyComponent"/>
  </TypeInfo>
```

```

    <Operation Name="Select" PropertyType="Method"
      ExposureLevel="CommonUsed">
      <Implementation Class="myComponentClasses::MyComponentEvent"
        Type="click"/>
    </Operation>
  </TypeInfo>
  <Properties>
    <Property Name="automationClassName" ForDescription="true">
      <Type VariantType="String"/>
      <Description>This is MyComponent.</Description>
    </Property>
    <Property Name="automationName" ForDescription="true">
      <Type VariantType="String"/>
      <Description>The name used by tools to id an object.</Description>
    </Property>
    <Property Name="className" ForDescription="true">
      <Type VariantType="String"/>
      <Description>To be written.</Description>
    </Property>
    <Property Name="id" ForDescription="true" ForVerification="true">
      <Type VariantType="String"/>
      <Description>Developer-assigned ID.</Description>
    </Property>
    <Property Name="index" ForDescription="true">
      <Type VariantType="String"/>
      <Description>The index relative to its parent.</Description>
    </Property>
  </Properties>
</ClassInfo>
...
</TypeInfoInformation>

```

You can edit the class definitions file to add a new recordable event to an existing component. To do this, you insert a new `<Operation>` in the control's `<TypeInfo>` block. This includes the implementation class of the event, and any arguments that the event might take.

The following example adds a new event, `MouseOver`, with several arguments to the `Button` control:

```

<TypeInfo>
  <Operation ExposureLevel="CommonUsed" Name="MouseOver"
    PropertyType="Method">
    <Implementation Class="flash.events::MouseEvent" Type="mouseOver"/>
    <Argument Name="inputType" IsMandatory="false"
      DefaultValue="mouse">
      <Type VariantType="String"/>
    </Argument>
    <Argument Name="shiftKey" IsMandatory="false" DefaultValue="false">
      <Type VariantType="Boolean"/>
    </Argument>
    <Argument Name="ctrlKey" IsMandatory="false" DefaultValue="false">

```

```

        <Type VariantType="Boolean"/>
    </Argument>
    <Argument Name="altKey" IsMandatory="false" DefaultValue="false">
        <Type VariantType="Boolean"/>
    </Argument>
</Operation>
</TypeInfo>

```

When you finish editing the class definitions file, you must distribute the new file to the automation tool users. For QTP, users must copy this file manually to the “*QTP_plugin_install\Flex 2 Plug-in for Mercury QuickTest Pro*” directory. When you replace the class definitions file in the QTP environment, you must restart QTP.

Setting the automationName property

The `automationName` property defines the name of a component as it appears in testing scripts. The default value of this property varies depending on the type of component. For example, a `Button` control’s `automationName` is the label of the `Button` control. Sometimes, the `automationName` is the same as the control’s `id` property, but this is not always the case.

For some components, Flex sets the value of the `automationName` property to a recognizable attribute of that component. This helps QC professionals recognize that component in their scripts. Because they do not usually have access to the underlying source code of the application, having a control’s visible property define that control can be useful. For example, a `Button` labeled “Process Form Now” appears in the testing scripts as `FlexButton("Process Form Now")`.

If you implement a new component, or derive from an existing component, you might want to override the default value of the `automationName` property. For example, `UIComponent` sets the value of the `automationName` to the component’s `id` property by default, but some components use their own methods of setting its value.

For example, in the Flex Store sample application, containers are used to create the product thumbnails. A container’s default `automationName` (it is the same as the container’s `id` property) would not be very useful because it is programmatically generated. So in Flex Store, the custom component that generates a product thumbnail explicitly sets the `automationName` to the product name to make testing the application easier.

The following example from the `CatalogPanel.mxml` custom component sets the value of the `automationName` property to the name of the item as it appears in the catalog. This is much more recognizable than the default automation name.

```
thumbs[i].automationName = catalog[i].name;
```

The following example sets the `automationName` property of the `ComboBox` control to “Credit Card List”; rather than using the `id` property, the testing tool typically uses “Credit Card List” to identify the `ComboBox` in its scripts:

```
<?xml version="1.0"?>
<!-- at/SimpleComboBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      [Bindable]
      public var cards: Array = [
        {label:"Visa", data:1},
        {label:"MasterCard", data:2},
        {label:"American Express", data:3}
      ];

      [Bindable]
      public var selectedItem:Object;
    ]]>
  </mx:Script>
  <mx:Panel title="ComboBox Control Example">
    <mx:ComboBox id="cb1" dataProvider="{cards}"
      width="150"
      close="selectedItem=ComboBox(event.target).selectedItem"
      automationName="Credit Card List"
    />

    <mx:VBox width="250">
      <mx:Text
        width="200"
        color="blue"
        text="Select a type of credit card."
      />
      <mx:Label text="You selected: {selectedItem.label}"/>
      <mx:Label text="Data: {selectedItem.data}"/>
    </mx:VBox>
  </mx:Panel>
</mx:Application>
```

If you do not set the value of the `automationName` property, the name of an object in a testing tool is sometimes a property that can change while the application runs. If you set the value of the `automationName` property, testing scripts use that value rather than the default value. For example, by default, QTP uses a `Button` control’s `label` property as the name of the `Button` in the script. If the label changes, the script can break. You can prevent this from happening by explicitly setting the value of the `automationName` property.

Buttons that have no label, but have an icon, are recorded by their index number. In this case, you should ensure that you set the `automationName` property to something meaningful so that the QC professional can recognize the Button in the script. This might not be necessary if you set the `toolTip` property of the Button because QTP uses that value if there is no label.

After the value of the `automationName` property is set, you should never change the value during the component's life cycle.

For item renderers, use the `automationValue` property rather than the `automationName` property. You do this by overriding the `createAutomationIDPart()` method and returning a new value that you assign to the `automationName` property, as the following example shows:

```
<mx:List xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      import mx.automation.IAutomationObject;
      override public function
        createAutomationIDPart(item:IAutomationObject):Object {
        var id:Object = super.createAutomationIDPart(item);
        id["automationName"] = id["automationIndex"];
        return id;
      }
    ]]>
  </mx:Script>
</mx:List>
```

This technique works for any container or list-like control to add index values to their children. There is no method for a child to specify an index for itself.

Instrumenting composite components

Composite components are custom components made up of two or more components. A common composite component is a form that contains several text fields, labels, and buttons. Composite components can be MXML files or ActionScript classes.

By default, you can record operations on all instrumented child controls of a container. If you have a Button control inside a custom TitleWindow container, the QA professional can record actions on that Button control just like on any Button control. You can, however, create a composite component in which some of the child controls are instrumented and some are not. To prevent the operations of a child component from being recorded, you override the following methods:

- `numAutomationChildren` getter
- `getAutomationChildAt()`

The `numAutomationChildren` property is a read-only property that stores the number of automatable children that a container has. This property is available on all containers that delegate implementation classes. To exclude some children from being automated, you return a number that is less than the total number of children.

The `getAutomatedChildAt()` method returns the child at the specified index. When you override this method, you return null for the unwanted child at the specified index, but return the other children as you normally would.

The following custom composite component is written in ActionScript. It consists of a VBox container with three buttons (OK, Cancel, and Help). You cannot record the operations of the Help button. You can record the operations of the other Button controls, OK and Cancel. The following example sets the values of the OK and Cancel buttons' `automationName` properties. This makes those button controls easier to recognize in the automated testing tool's scripts.

```
// MyVbox.as
package { // Empty package
    import mx.core.UIComponent;
    import mx.containers.VBox;
    import mx.controls.Button;
    import mx.automation.IAutomationObject;
    import mx.automation.delegates.containers.BoxAutomationImpl;

    public class MyVBox extends VBox {
        public var btnOk : Button;
        public var btnHelp : Button;
        public var btnCancel : Button;

        public function MyVBox():void { // Constructor
        }

        override protected function createChildren():void {
            super.createChildren();

            btnOk = new Button();
            btnOk.label = "OK";
            btnOk.automationName = "OK_custom_form";
            addChild(btnOk);

            btnCancel = new Button();
            btnCancel.label = "Cancel";
            btnCancel.automationName = "Cancel_custom_form";
            addChild(btnCancel);

            btnHelp = new Button();
            btnHelp.label = "Help";
            btnHelp.showInAutomationHierarchy = false;
            addChild(btnHelp);
        }

        override public function get numAutomationChildren():int {
            return 2; //instead of 3
        }

        override public function
            getAutomationChildAt(index:int):IAutomationObject {
            switch(index) {
```

```

        case 0:
            return btnOk;
        case 1:
            return btnCancel;
    }
    return null;
}
} // Class
} // Package

```

To make this solution more portable, you could create a custom Button control and add a property that determines whether a Button should be testable. You could then set the value of this property based on the Button instance (for example, `btnHelp.useInAutomation = false`), and check against it in the overridden `getAutomationChildAt()` method, before returning null or the button instance.

Example: Instrumenting the RandomWalk custom component for QTP

The RandomWalk component is an example of a complex custom component. It has custom events and custom itemRenderers. Showing how it is instrumented can be helpful in instrumenting your custom components.

This section describes how to instrument the RandomWalk custom component so that interaction with it can be recorded and played back using the QTP automation tool.

The source code used in this section can be found at the following location:

http://www.adobe.com/go/flex_automation_randomwalk_apps

Instrumenting the RandomWalk custom component

The first task when instrumenting a custom component is to create a delegate and add the new component to the class definitions XML file.

To instrument the RandomWalk custom component:

1. Create a RandomWalkDelegate class that extends `UIComponentAutomationImpl`, similar to RandomWalk extending `UIComponent`. `UIComponentAutomationImpl` implements the `IAutomationObject` interface.

2. Mark the delegate class as a mixin with the `[Mixin]` metadata tag; for example:

```
package {  
  
    ...  
  
    [Mixin]  
    public class RandomWalkDelegate extends UIComponentAutomationImpl {  
    }  
}
```

This results in a call to the static `init()` method in the class when the SWF file loads.

3. Add a public static `init()` method to the delegate class, and add the following code to it:

```
public static init(root:DisplayObject):void {  
    Automation.registerDelegateClass(RandomWalk, RandomWalkDelegate);  
}
```

4. Add a constructor that takes a `RandomWalk` object as parameter. Add a call to the super constructor and pass the object as the argument:

```
private var walker:RandomWalk  
public function RandomWalkDelegate(randomWalk:RandomWalk) {  
    super(randomWalk);  
    walker = randomWalk;  
}
```

5. Update the `TEAFlexCustom.xml` file so that QTP recognizes the `RandomWalk` component. Add the text between the `<TypeInfo>` root tags. The `TEAFlexCustom.xml` file is located in the “*QTP_plugin_install\Flex 2 Plug-in for Mercury QuickTest Pro*” directory. For more information about the `TEAFlexCustom.xml` file, see “Using the class definitions file” on page 25.

```
<TypeInfo xsi:noNamespaceSchemaLocation="ClassesDefintions.xsd"  
    Priority="0" PackageName="TEA" Load="true" id="Flex" xmlns:xsi="http://  
    /www.w3.org/2001/XMLSchema-instance">  
    ...  
    <ClassInfo Name="FlexRandomWalk" GenericTypeID="randomwalk"  
        Extends="FlexObject" SupportsTabularData="false">  
        <Description>FlexRandomWalk</Description>  
        <Implementation Class="RandomWalk"/>  
        <TypeInfo>  
        </TypeInfo>  
        <Properties>  
            <Property Name="automationClassName" ForDescription="true">  
                <Type VariantType="String"/>  
                <Description>To be written.</Description>  
            </Property>  
            <Property Name="automationName" ForDescription="true">  
                <Type VariantType="String"/>  
                <Description>The name used by the automation system to
```

```

        identify an object.</Description>
    </Property>
    <Property Name="className" ForDescription="true">
        <Type VariantType="String"/>
        <Description>To be written.</Description>
    </Property>
    <Property Name="id" ForDescription="true" ForVerification="true">
        <Type VariantType="String"/>
        <Description>Developer-assigned ID.</Description>
    </Property>
    <Property Name="automationIndex" ForDescription="true">
        <Type VariantType="String"/>
        <Description>The object's index relative to its parent.
    </Description>
    </Property>
</Properties>
</ClassInfo>
</TypeInformation>

```

This defines the name of the `FlexRandomWalk` component and its implementing class. It specifies the `automationClassName`, `automationName`, `className`, `id` and `automationIndex` properties as available for identifying a `RandomWalk` instance in the Flex application. This is possible because `RandomWalk` derives from `UIComponent`, and these properties are defined on that parent class.

If your component has a property that you can use to differentiate between component instances, you can also add that property. For example, the `label` property of a `Button` control, though not unique when the whole application is considered, can be assumed to be unique within a container; therefore, you can use it as an identification property.

Instrumenting `RandomWalk` events

The next step in instrumenting a custom component is to identify the important events that must be recorded by QTP. The `RandomWalk` component dispatches a `RandomWalkEvent.ITEM_CLICK` event. Because this event indicates user navigation, it is important and must be recorded.

To instrument the `ITEM_CLICK` event:

1. Add an event listener for the event in the `RandomWalkDelegate` constructor:

```

randomWalk.addEventListener(RandomWalkEvent.ITEM_CLICK,
    itemClickListener)

```

2. Identify the event in the TEAFlexCustom.xml file so that QTP recognizes the event. Add the following text in the <TypeInfo> tag in the TEAFlexCustom.xml file:

```
<Operation Name="Select" PropertyType="Method"
  ExposureLevel="CommonUsed">
  <Implementation Class="randomWalkClasses::RandomWalkEvent"
    Type="itemClick"/>
</Operation>
```

Adding this block to the TEAFlexCustom.xml file names the event as Select and indicates that it is tied to the RandomWalkEvent class, which is available in randomWalkClasses namespace. It also defines the event of type itemClick.

When using the RandomWalk component, users click on items. The component records the item label so that QC professionals can easily recognize their action in the QTP script.

The RandomWalkEvent class has only a single property, item, that stores the XML node information. In the RandomWalk component's implementation, the RandomWalkRenderer item renderer is used to display the data on the screen. It is derived from the Label control, which is already instrumented. The Label control returns the label text as its automationName, which is what you want to record.

To record the label text:

1. Add a new property, itemRenderer, to the RandomWalkEvent class.
2. Add the code to initialize this property for the event before dispatching the event; for example:

```
var rEvent:RandomWalkEvent = new
  RandomWalkEvent(RandomWalkEvent.ITEM_CLICK,node);
rEvent.itemRenderer = child as Label;
dispatchEvent(rEvent);
```

3. Add the new itemRenderer property as an argument to the Select operation in the TEAFlexCustom.xml file:

```
<Operation Name="Select" PropertyType="Method"
  ExposureLevel="CommonUsed">
  <Implementation Class="randomWalkClasses::RandomWalkEvent"
    Type="itemClick"/>
  <Argument Name="itemRenderer" IsMandatory="true" >
    <Type VariantType="String" Codec="automationObject"/>
    <Description>User-clicked item.</Description>
  </Argument>
</Operation>
```

This code block specifies the type of argument as `String` and the codec as `automationObject`. The `Codec` property is an optional attribute of the `<Type>` tag in the `<Argument>` and `<Property>` tags. It defines a class that converts `ActionScript` types to agent-specific types.

4. In the event handler, call the `recordAutomatableEvent()` method with `event` as the parameter, as the following example shows:

```
private function itemClickHandler(event:RandomWalkEvent):void {
    recordAutomatableEvent(event);
}
```

In some cases, the component does not dispatch any events or the event that was dispatched does not have the information that is required during the recording or playing back of the test script. In these circumstances, you must create an event class with the required properties. In the component, you create an instance of the event and pass it as a parameter to the `recordAutomatableEvent()` method.

For example, the `ListItemSelected` event has been added in automation code and is used by the `List` automation delegate to record and play back select operations for list items.

To locate the item renderer object, Automation requires some help. Copy the standard implementation for the following methods:

```
override public function
    createAutomationIDPart(child:IAutomationObject):Object {
    var help:IAutomationObjectHelper = Automation.automationObjectHelper;
    return help.helpCreateIDPart(this, child);
}

override public function resolveAutomationIDPart(part:Object):Array {
    var help:IAutomationObjectHelper = Automation.automationObjectHelper;
    return help.helpResolveIDPart(this, part as AutomationIDPart);
}
```

Preparing RandomWalk for playback

Flex components display the data in a data provider after processing and formatting the data. Playback code requires a way to trace back the visual data to the data provider and the component displaying that data. For example, from the visual label of an item in the `List`, playback code should be able to find the item renderer that shows the element so that the item's click can be played back.

When playing back a `RandomWalkEvent`, you must identify the item renderer with the `automationName` given. Because the `RandomWalk` component has many item renderers that are children which are derived from `UIComponent` subclasses, you must identify them using the `automationName` property.

The `IAutomationObject` interface already has APIs to support this. The `UIComponentAutomationImpl` interface provides default implementations for some methods, but you must override some methods to return information specific to the `RandomWalk` component.

To prepare the `RandomWalk` component for playback:

1. Instruct QTP how many renderers are being used by the instance of the `RandomWalk` component. `RandomWalk` uses an Array of Arrays for all the renderers. Add the following code in `RandomWalkDelegate` to find the total number of instances:

```
override public function get numAutomationChildren():int {
    var numChildren:int = 0;
    var renderers:Array = walker.getItemRenderers();
    for (var i:int = 0;i< renderers.length;i++) {
        numChildren += renderers[i].length;
    }
    return numChildren;
}
```

2. Access the `itemRenderers` property in the delegate; however, this property is private. So, you must add the following accessor method to the `RandomWalk` component:

```
public function getItemRenderers():Array {
    return _renderers;
}
```

Alternatively, you can make the property public or change its namespace.

3. QTP requests each child renderer. Add the following code to determine the exact renderer and return it:

```
override public function getAutomationChildAt(index:int):
IAutomationObject {
    var numChildren:int = 0;
    var renderers:Array = walker.getItemRenderers();
    for(var i:int = 0; i < renderers.length; i++) {
        if(index >= numChildren) {
            if(i+1 < renderers.length && (numChildren + renderers[i].length)
                <= index) {
                numChildren += renderers[i].length;
                continue;
            }
            var subIndex:int = index - numChildren;
            var instances:Array = renderers[i];
            return (instances[subIndex] as IAutomationObject);
        }
    }
    return null;
}
```

Linking the delegate to an application

There are two options to link the delegate class with the application SWF file. You can use the `includes` compiler option and link to the delegate as follows:

```
mxm1c -includes RandomWalkDelegate FlexApp.mxml
```

You can also build a SWC file for the delegate class. You then include the SWC file with the Flex application by using the `include-libraries` compiler option, as the following code shows:

```
mxm1c -include-libraries RandomWalkAT.SWC -- FlexApp.mxml
```

This approach is useful if you have many components and many delegate classes.

Adjusting event recording

You can compile and record any application that uses a `RandomWalk` component. While recording, you might notice that the `FlexLabel().Click` operation is recorded in addition to the `Select` operation. Generally, you do not want to record both operations for each user interaction.

In the following example, you stop the `AutomationRecordEvent.RECORD` event from being recorded. Because it is a bubbling event, you can listen to the event from the children, and prevent the event from being recorded.

To prevent an event from being recorded:

1. Add an event handler in the constructor of the delegate, as the following example shows:

```
obj.addEventListener(AutomationRecordEvent.RECORD, labelRecordHandler);
```

2. Prevent the recording by calling the `preventDefault()` method or by stopping the propagation of the event:

```
public function labelRecordHandler(event:AutomationRecordEvent):void {  
    // if the event is not from the owning component reject it.  
    if (event.replayableEvent.target != uiComponent)  
        //event.preventDefault(); can also be used.  
        event.stopImmediatePropagation();  
}
```

The `RandomWalkDelegate` class must handle the playback of the `RandomWalkEvent` event only. Any other event must be handled by the super class implementation.

3. Override the `replayAutomatableEvent()` method and handle the `RandomWalkEvent` event:

```
override public function replayAutomatableEvent(event:Event):Boolean {  
    if (event is RandomWalkEvent) {  
    }  
}
```

```
    return super.replayAutomatableEvent(event);  
}
```

4. (Optional) To replay the `RandomWalkEvent`, you must replay a click on the item renderer. To use the `replayClick()` method to play back a click on the item renderer, use the following code:

```
override public function replayAutomatableEvent(event:Event):Boolean {
    var help:IAutomationObjectHelper = Automation.automationObjectHelper;
    if (event is RandomWalkEvent) {
        var rEvent:RandomWalkEvent = event as RandomWalkEvent
        help.replayClick(rEvent.itemRenderer);
        return true;
    } else
        return super.replayAutomatableEvent(event);
}
```

For playing back an event, the `Automation.automationObjectHelper` class provides some helper methods, the following table describes:

Method	Description
<code>replayClick()</code>	Dispatches the <code>mouseDown</code> , <code>mouseClick</code> and <code>mouseUp</code> events on a <code>UIComponent</code> .
<code>replayMouseEvent()</code>	Dispatches a particular mouse event on a <code>UIComponent</code> .
<code>replayKeyDownKeyUp()</code>	Dispatches a keyboard event with <code>keyCode</code> and key modifiers specified.
<code>replayKeyboardEvent()</code>	Dispatches a particular keyboard event on a <code>UIComponent</code> .

For more information, see the documentation for the `IAutomationObjectHelper` class in the *Adobe Flex 2 Language Reference*.

5. Record and play back your application.
6. To ensure that the view is updated, add the following code after the call to the `replayClick()` method:

```
override public function replayAutomatableEvent(event:Event):Boolean {
    var help:IAutomationObjectHelper = Automation.automationObjectHelper;
    if (event is RandomWalkEvent) {
        var rEvent:RandomWalkEvent = event as RandomWalkEvent
        help.replayClick(rEvent.itemRenderer);
        (uiComponent as IInvalidating).validateNow();
        return true;
    } else
        return super.replayAutomatableEvent(event);
}
```

Adjustments like this might be required in the delegate to adjust the behavior of the component during playback because QTP does not wait for actions to be completed. The same can also be achieved by adding `Wait` statements in the QTP script.

You should now be able to compile, record, and play back any application with the `RandomWalk` component.

Adding checkpoints

To add checkpoints on public properties of the component, you add a small description of the property to the component description in the custom class definitions XML file (`TEAFlexCustom.xml`). You also add a getter for those public properties.

For the `openChildrenCount` property of the `RandomWalk` component to appear in checkpoints, add the following element as a child of the `<Properties>` tag:

```
<Property Name="openChildrenCount" ForVerification="true"
  ForDefaultVerification="true">
  <Type VariantType="Integer"/>
  <Description>Number of children open currently.</Description>
</Property>
```

When you use a checkpoint operation with a `RandomWalk` component, QTP displays the `openChildrenCount` property in the Checkpoint dialog box.

Also, add the following getter method to the `RandomWalk` class:

```
public function get openChildrenCount():int {
    return numAutomationChildren;
}
```

The `numAutomationChildren` property is inherited from the `UIComponent` class.

Contents

Introduction	43
About the automation APIs	45
Understanding the automation flow	51
Creating agents.....	53
Creating a recording agent.....	54
Creating a replaying agent	62

Introduction

The automation API in Flex can be used for many tasks, including metrics, automated testing, and co-browsing. This section describes some of these tasks in more detail.

To use the automation API, you should understand the following terminology:

- *agent* — An agent facilitates the communication between the Flex application and the automation tool. The way in which the communication happens depends on the automation tool, but it can include using the ExternalInterface API to access the browser's DOM or ActiveX plug-ins. Agents are typically implemented in ActionScript and packaged as a SWC file.
- *automation tool* — An automation tool records, and sometimes plays back, interaction with Flex applications. An automation tool requires an agent to provide communication between it and the Flex application. Automation tools include Mercury QTP and Segue.

The following are required if you want to use the automation framework with Flex applications:

- LiveCycle Data Services license key.

- Automation installer. This installer includes the `automation_agent.swc` and `automation_agent_rb.swc` files. After running the installer, you should copy the `automation_agent.swc` file to your `libs` directory and the `automation_agent_rb.swc` file to your `locale/en_US` directory.

In addition to these requirements, you should download the `CustomAgent.zip` and `AutoQuick.zip` files. The `CustomAgent.zip` file includes a set of classes that define a custom automation agent that records user interaction with a Flex application. The interaction is then written out to a database. The `AutoQuick.zip` file records user interaction and then plays it back. You can download these examples from the following location:

http://www.adobe.com/go/flex_automation_agent_apps

The sample files also include an XML file that the custom agents use to define its environment.

In addition to these samples, you can also download and run a testing-enabled version of the FlexStore application from http://www.adobe.com/go/flex_flexstore_automation.

The Automation Framework defines a single API. The API has two parts:

- Component API — Components need to implement this API to support Automation features. Developers can choose to put the code either in the main component itself or in a mixin class. Mixin classes would implement this API for Flex components.
- Agent API — Agents like accessibility, automated testing (and the rest) would use this API to communicate with the component API.

Component developers need to implement the component API for their component once and then their component will be ready to converse with any agent. Agent developers need to implement the agent API for their specific feature or tool and it will be able to work with any Flex application. For more information, see “About the automation APIs” on page 45.

Metrics

You might want to analyse how your online applications are being used. By gathering metrics information, you can answer the following questions:

- What product views are most popular?
- When do users abandon the checkout process?
- What is a typical path through my application?
- How many product views does a user look at during their session?

Automated testing

Maintaining the quality of a large software application is difficult. Verifying lots of functionality in any individual build can take a QA engineer many hours or even days, and much of the work between builds is repetitive. To alleviate this, automated testing tools have been created that can use applications and verify behavior without human intervention. Major application environments such as Java and .NET all have testing tool support from vendors such as Mercury and Segue.

Using the Flex automation API, you can:

- Record and replay events in a separate tool
- Manage communication between components and agents

Co-browsing

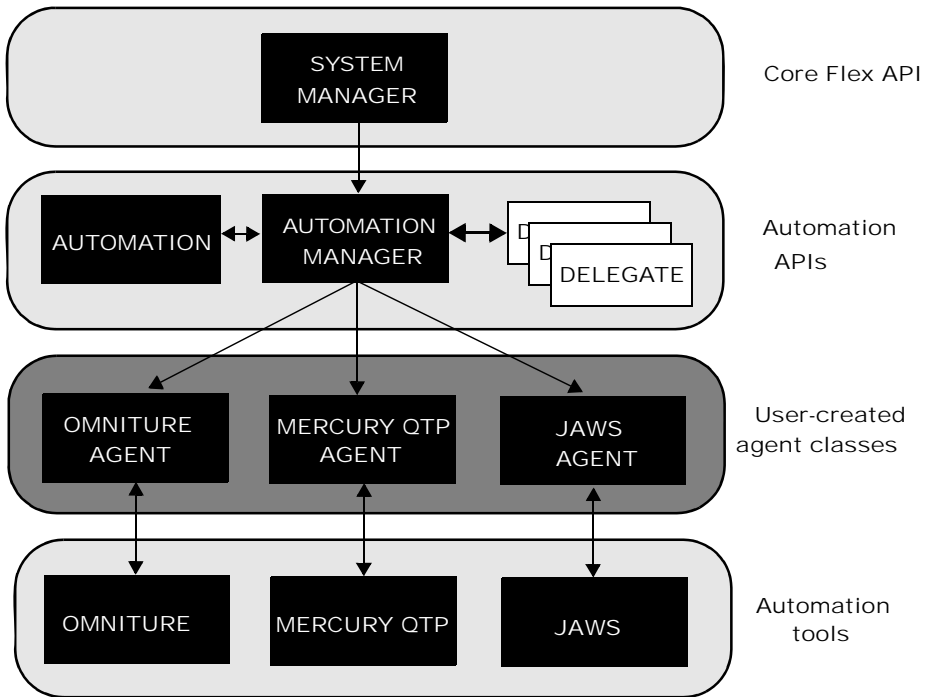
You might want to run the same application at different locations and view the application at the same time. By using the automation API, you can ensure that the applications are synchronized as users navigate through the them. User interaction at any one location can be played at other locations and other users can see the action in real-time.

About the automation APIs

There are four main parts that enable the automation framework in Flex:

- Flex API
- Automation API
- Agent class (also known as an adapter)
- Automation tools

The following illustration shows the relationship between these parts:



About the SystemManager class

The `SystemManager` is one of the highest level classes in a Flex application. Every Flex application has a `SystemManager`. It parents all displayable objects within the application, such as the main `mx.core.Application` instance and all popups, tooltips, cursors, and so on.

The `SystemManager` calls the `init()` method on all mixins. This is responsible for creating the `AutomationManager` as well as the `Agent` and the `delegate` classes. The `SystemManager` is also responsible for adding the `Application` object to the `Player's` stage (root).

About the AutomationManager class

The `AutomationManager` class is a singleton that extends the `EventDispatcher` class. It implements the `IAutomationManager`, `IAutomationObjectHelper`, and `IAutomationMouseSimulator` interfaces.

The `AutomationManager` is a mixin, so its `init()` method is called by the `SystemManager` when the application initializes. In the `init()` method, the `AutomationManager` class adds an event listener for the `Event.ADDED` event. This event is dispatched by the `Player` whenever a display object is added to the display list. When that happens, the `Player` calls the `childAddedHandler()` method:

```
root.addEventListener(Event.ADDED, childAddedHandler, false, 0, true);
```

In the `childAddedHandler()` method, the `AutomationManager` class:

- Creates a new instance of a delegate for each display object.
- Adds the display object to a delegate class map. This maps the display object to a delegate instance; for example:

```
Automation.delegateClassMap[componentClass] =  
    Automation.delegateClassMap[className];
```

The delegate class map is a property of the `Automation` class.

- Ensures that all children of the new display object are added to the class map as well.

In the `recordAutomatableEvent()` method, the `AutomationManager`:

- Creates an `AutomationRecordEvent.RECORD` event.
- Dispatches the `RECORD` events. The agent listens for these events.

The `recordAutomatableEvent()` method is called by the delegates.

About the `Automation` class

During application initialization, the `Automation` class is created. This is a static class that maintains a map of its delegate class to its component class.

This class does the following for recording events:

- Provides access to the `AutomationManager`.
- Creates the `delegateClassMap` as a static property.

For example, there is a `MyButton` class that extends the `Button` class but does not have its own delegate class (`MyButton` may not be adding any new functionality to be recorded or played back). When the `AutomationManager` encounters an instance of `MyButton` class, it checks with the `Automation` class for a corresponding delegate class. When it fails to find one, it uses the `getSuperClassName()` method to get the super class of `MyButton`, which is `Button`. The `AutomationManager` then tries to find the delegate for this `Button` class. At that point, the `AutomationManager` adds a new entry into the delegate-component class for `MyButton`, associating it with the `ButtonDelegateAutomationImpl` class so that the next time, the `AutomationManager` can find this mapping without searching the inheritance hierarchy.

About the delegate classes

The delegate classes provide automation hooks to the Flex framework and charting components. The delegate classes are in the `mx.automation.delegates` package, and they extend the `UIComponentAutomationImpl` class. The delegates are named *ControlNameAutomationImpl*. For example, the Button control's delegate class is `ButtonAutomationImpl`.

The delegate classes register themselves with their associated Automation class by providing the component class and their own class as input. The `AutomationManager` uses the Automation class-to-delegate class map to create a delegate instance that corresponds to a component instance in the `childAddedHandler()` method.

The delegate classes are mixins, so their `init()` method is called by the `SystemManager`. The `init()` method of the delegate classes:

- Calls the `registerDelegateClass()` method of the Automation class. This method maps the class to an automation component class; for example:

```
var className:String = getQualifiedClassName(compClass);
delegateClassMap[className] = delegateClass;
```

- Adds event listeners for the mouse and keyboard events; for example:

```
obj.addEventListener(KeyboardEvent.KEY_UP, btnKeyUpHandler, false,
    EventPriority.DEFAULT+1, true);
obj.addEventListener(MouseEvent.CLICK, clickHandler, false,
    EventPriority.DEFAULT+1, true);
```

These event handlers call the `AutomationManager` class's `recordAutomatableEvent()` method which, in turn, dispatches the `AutomationRecordEvent.RECORD` events that the automation agent listens for.

All core framework and charting classes have delegate classes already created. You are not required to create any delegate classes unless you have custom components that dispatch events that you want to automate. In this case, you must create a custom delegate class for inclusion in your Flex application. For more information, see “Example: Instrumenting the `RandomWalk` custom component for QTP” on page 33.

About the agent

The agent facilitates communication between the Flex application and automation tools such as QTP and Segue.

When recording, the agent class is typically responsible for implementing a persistence mechanism in the automation process. It gets information about events, user sessions, and application properties and typically writes them out to a database, log file, LocalConnection, or some other persistent storage method.

When you create a custom agent, you compile it and its supporting classes into a SWC file. You then add that SWC file to your Flex application by using the `include-libraries` command-line compiler option.

The custom agent class must be a mixin, which means its `init()` method is called by the `SystemManager`. The `init()` method of the agent:

- Defines a handler for the RECORD events.
- Defines the environment. The environment indicates what components and their methods, properties, and events can be recorded with the automation API.

The sample custom agent class uses an XML file that contains Flex component API information. The agent loads it with a `URLRequest`:

```
var myXMLURL:URLRequest = new URLRequest("AutomationGenericEnv.xml");
myLoader = new URLLoader(myXMLURL);
automationManager.automationEnvironment = new CustomEnvironment(new
    XML(source));
```

In this example, the source is an XML file that defines the Flex metadata (or environment information). This metadata includes the events and properties of the Flex components.

Note that representing events as XML is agent-specific. The general-purpose automation API does not require it, but the XML file makes it easy to adjust the granularity of the events that are recorded.

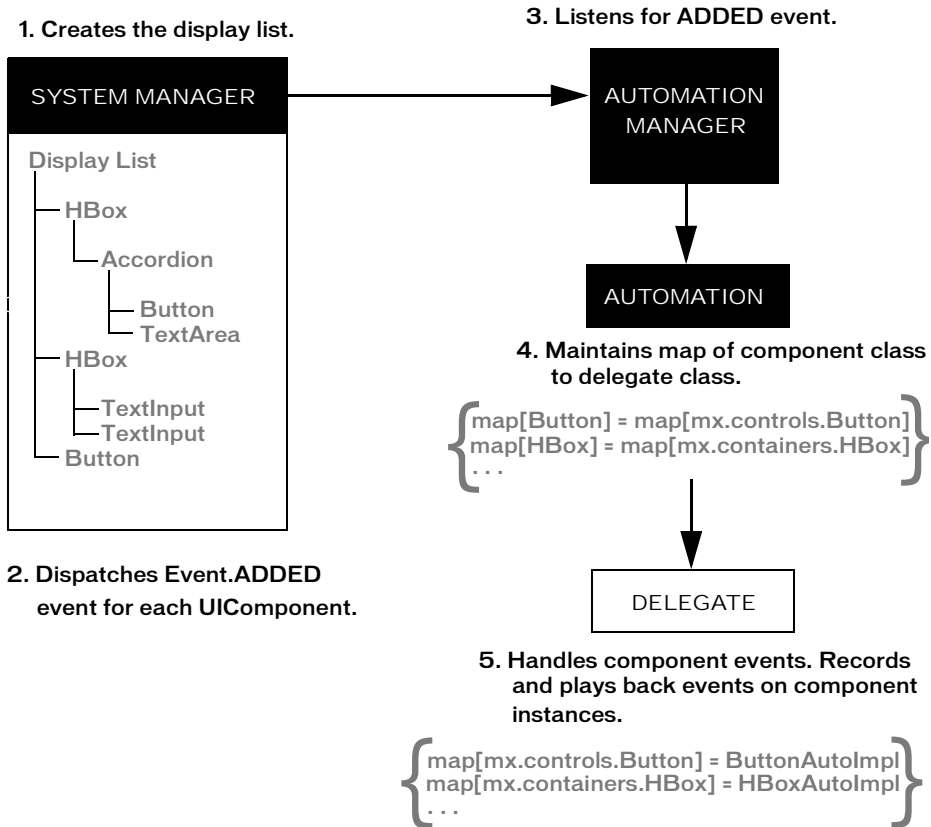
You are not required to create an instance of your adapter in the `init()` method. You can also create this instance in the `APPLICATION_COMPLETE` event handler if your agent requires that the application be initialized before it is instantiated.

Most agents require a set of classes that handle the way in which the agent persists automation information and defines the environment. For more information, see “Creating a recording agent” on page 54.

Understanding the automation flow

When the application initializes, the `AutomationManager` is created. In its `init()` method, it adds a listener for `Event.ADDED` events.

The following illustration shows the order of events when the application initializes and the AutomationManager constructs the delegate map.

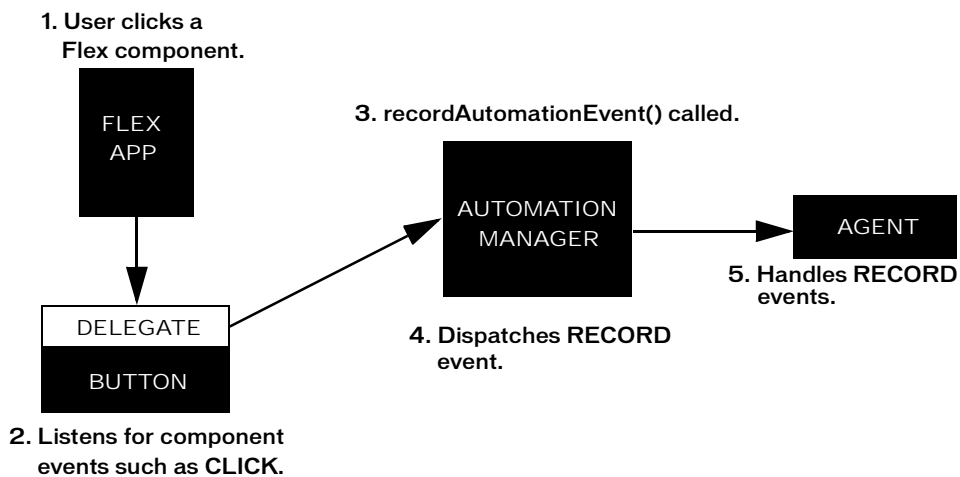


1. The SystemManager creates the display list. This is a tree of visible objects that make up your application.
2. Each time a new component is added, either at the root of the display list or as a child of another member of the display list, the SystemManager dispatches an Event.ADDED event.
3. The AutomationManager listens for the ADDED event. In its ADDED event handler, it calls methods on the Automation class. It then instantiates the delegate for that class.
4. The Automation class maps each component in the display list to its full class name.
5. When it is created, the delegate class adds a reference to its instance in the delegate class map. The delegate class then handles events during record and play back sequences.

The delegate is now considered *registered* with the component. It adds event listeners for the component's events and will call the AutomationManager when the component triggers those events.

After the components in the display list are instantiated and mapped to instances of their delegate classes, the AutomationManager is ready to listen for events and forward them to the agent for processing.

The following illustration shows the flow of operation when a user performs an action that is a recordable event. In this case, the user clicks a Button control in the application.



1. The user clicks the Button control in the Flex application. The SystemManager dispatches a MouseEvent.CLICK event.
2. The ButtonAutomationImpl class, the Button control's automation delegate, listens for click events. In the delegate's click handler, the delegate calls the AutomationManager's recordAutomationEvent() method. (It is likely that the button also defines a click handler to respond to the user action, but that is not shown.)
3. The AutomationManager's recordAutomationEvent() method dispatches an AutomationRecordEvent.RECORD event. In that event, the replayableEvent property points to the original click event.

4. The custom agent class listens for RECORD events. When it receives the RECORD event, it uses the `replayableEvent` property to access the properties of the original event. This is where the agent would record the event properties in a database, log the event properties, or get information about the user before recording them.

Creating agents

You create an agent as a SWC file and link it into the Flex application using the `include-libraries` compiler option. You can link multiple agents in any number of SWC files to the same Flex application. However, to use multiple agents at the same time, you must use the same environment configuration files for all agents.

The general process for creating a custom agent is as follows:

- Mark the agent class as a mixin; this triggers a call to a static `init()` method from the `SystemManager` on application start-up.
- Get a reference to the `AutomationManager`.
- Add event listeners for the `APPLICATION_COMPLETE` and `RECORD` events.
- Load the environment information. Environment information is Flex metadata that describes the objects and operations of the Flex application. It can be an XML file, as the samples use, or be in some other data format.
- Define a static `init()` method that creates an instance of the agent
- Define a method that handles `RECORD` events. In that method, you can access:
 - Automation details such as the automation name
 - User information such as the `FlexSession` object (through a `RemoteObject`)
 - The event that triggered the `RECORD` event
 - The target object and its properties that triggered the `RECORD` event

The record handler in the agent gets an `AutomationRecordEvent` whose target is the object on which recording happened. The `automationManager.createID()` method converts the object to a string that can be recorded on the screen. Some tools may require the entire automation hierarchy which needs to be generated in this method.

You also use the custom agent class to enable and disable recording of automation events.

For more information on creating a custom agent, see “Creating a recording agent” on page 54.

Creating a recording agent

This section describes how to use the classes in the CustomAgent.zip file to create a custom agent that records metrics data as a user interacts with an application. For information about creating an agent that records and plays back user interactions, see “Creating a replaying agent” on page 62.

The custom agent in the CustomAgent.zip file is the CustomAdapter class. The CustomAdapter class calls an HTTPService that writes event data out to a database.

Supporting classes for the CustomAdapter class include the following:

- CustomEnvironment class that implements IAutomationEnvironment
- CustomAutomationClass that implements IAutomationClass
- CustomAutomationEventDescriptor class
- CustomAutomationMethodDescriptor class
- CustomAutomationPropertyDescriptor class

In addition to these classes, the ZIP file also includes the environment XML file, and a utility class that parses that file.

This section also describes how to access a RemoteObject from the custom agent that provides user session information. To simplify the example, this example will write the data out to a trace log rather than write the data to a database.

Using the CustomAdapter class

This section provides step-by-step instructions on how to use the CustomAdapter class and its supporting files to create a Flex application that records automation events.

You cannot use this custom agent with agents that use different environment information, such as the QTP agent included in the Flex Automation Package. As a result, if you use this agent, do not use QTP to record scripts.

Set up your flex installation

Before you can create an application that uses the automation API to record automatable events, you must do the following:

1. Ensure that you have an LiveCycle Data Services serial number and that you have added it to the license.properties file. If you use charts in your example application, ensure that you have a charting serial number as well.

2. Extract all the files in the CustomAgent.zip file to a directory. For example, c:/myfiles/flex2/agent. This file contains the agent and helper classes in the custom.* package. It also includes mySQL and PHP code that you can use to connect your Flex application to a database.
3. Save the AutomationGenericEnv.xml file at the top level directory where you will put the Flex application. In this case, store it in c:/myfiles/flex2/agent.
4. Run the automation installer. The installer includes the automation_agent.swc and automation_agent_rb.swc files, among other files.
 - a. Copy the automation_agent.swc file to the frameworks/libs directory.
 - b. Copy the automation_agent_rb.swc to your frameworks/locale/en_US directory.

Once you have set up your environment, you can edit the CustomAgent to record an interaction with a Flex application.

Edit the CustomAdapter class

The CustomAdapter class handles the RECORD events in the recordHandler() method. The RECORD event has the following properties:

- name — The agent's name of the event that triggered the call to the agent; for example, Click. This is the name as it is defined in the agent's environment.
- replayableEvent — The event that triggered the call to the agent.
- automationObject — The control that triggered the replayableEvent; for example, if the user clicked a Button, this property contains a reference to the Button control. You can access the automationObject by casting it to an IAutomationObject; you can then access properties of that object, such as a Button's label property, by using the AutomationManager's getProperties() method:

```
var obj:IAutomationObject = event.automationObject;
var label:String = automationManager.getProperties(obj, ["label"])[0];
```

- args — Additional information about the event, such as the text entered in a TextInput control or whether the Alt key was held down during a mouse click. You can view all the arguments by using code similar to the following:

```
for (var i:int = 0; i<event.args.length; i++) {
    trace("event.args[" + i + "]: " + event.args[i]);
}
```

The default version of the CustomAdapter's recordHandler() method contains code that writes event information to a database. This section describes how to simplify that method so that you can run the example without configuring a database. After having run through the steps in this section, you can go back and create a database and revert the CustomAdapter class's recordHandler() method.

Open the CustomAdapter.as file and make the following changes:

1. Edit the recordHandler() method. Comment out the service access and replaced it with trace statements; for example:

```
trace("automation name:" + obj.automationName);
trace("event name:" + event.name);
trace("replayable event:" + event.replayableEvent);
trace("event target:" + event.target);
// Arguments to be sent are '#' separated.
var arguments:String = event.args.join("#");
trace("args:" + arguments);

// Show all event args
for (var i:int = 0; i<event.args.length; i++) {
    trace("event.args[" + i + "]: " + event.args[i]);
}
trace("-----");
```

You can later add the database support by uncommenting the service methods.

2. Edit the constructor. Add a trace statement such as the following:

```
trace("in CustomAdapter constructor")
```

This lets you be sure that the custom agent is being instantiated when the application starts up.

3. Compile the custom agent's SWC file. To do this, you use the compc utility and include the classes in the custom package, as the following example shows:

```
compc -source-path+=c:/myfiles/flex2/agent
      -include-classes
        custom.CustomAdapter custom.CustomAutomationClass
        custom.CustomAutomationEventDescriptor
        custom.CustomAutomationMethodDescriptor
        custom.CustomAutomationPropertyDescriptor
        custom.CustomEnvironment
        custom.utilities.EnvXMLParser
      -library-path+=c:/home/dev/depot/flex/sdk/frameworks/libs
      -output c:/myfiles/flex2/agent/CustomAgent.swc
```

This creates the CustomAgent.swc file in the c:/myfiles/flex2/agent directory.

Create and run the Flex application

When you use automation with Flex, you must create a Flex application and supporting wrapper files. If you want to use LiveCycle Data Services to request the MXML file, you can skip this section and use the instructions in “Using LiveCycle Data Services” on page 58.

1. Create Main.xml and store it in the same directory as the CustomAgent.swc file (for example, c:/myfiles/flex2/agent). The following is a sample MXML file:

```
<?xml version="1.0"?>
<!-- agent/Main.xml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="100%"
width="100%">
  <mx:Script><![CDATA[
    private function changeLabel(newLabel:String):void {
      b1.label = newLabel;
    }
  ]]></mx:Script>
  <mx:TextInput id="t1" text="" />
  <mx:Button id="b1" label="Change Label"
click="changeLabel(t1.text)" />
</mx:Application>
```

2. Compile the Flex application. You must include the CustomAgent.swc file as a library. You must also include the automation.swc and automation_agent.swc files, as the following example shows:

```
mxmlc
-include-libraries+=c:/myfiles/flex2/agent/CustomAgent.swc;
  c:/home/dev/depot/flex/sdk/frameworks/libs/automation.swc;
  c:/home/dev/depot/flex/sdk/frameworks/libs/automation_agent.swc
c:/myfiles/flex2/agent/Main.xml
```

This creates Main.swf in the c:/myfiles/flex2/agent directory.

3. Create an HTML wrapper. You must request the SWF file through a wrapper otherwise file security errors will prevent you from loading the environment XML file. The following example wrapper loads the mysource.js file:

```
<html><body>
<script src="mysource.js"></script>
</body></html>
```

4. Create a JavaScript file that defines the <OBJECT> and <EMBED> tags that embed your Flex application's SWF file. For example:

```
document.write("<object id='tempId' classid='clsid:D27CDB6E-AE6D-11cf-
96B8-444553540000' height='100%' width='100%'>");
document.write("<param name='src' value='Main.swf' />");
document.write("<embed name='Main' src='Main.swf' height='100%'
width='100%' />");
document.write("</object>");
```

5. Copy the application SWF, wrapper, JavaScript, and environment XML files to a web server. The web server's directory now contains:
 - Main.swf
 - index-simple.html

- mysource.js
- AutomationGenericEnv.xml

The XML file is loaded at run time, so it must be accessible by the SWF file on the web server.

6. Request the html wrapper from the web server; for example:

```
http://localhost:8100/flex/agent/index-simple.html
```

7. Run the application and change the name of the label to "42". The trace log should contain something like the following:

```
in CustomAdapter constructor
automation name:til
event name:SelectText
event target:[object AutomationManager]
args:0#0
event.args[0]: 0
event.args[1]: 0
-----
automation name:til
event name:Input
event target:[object AutomationManager]
args:42
event.args[0]: 42
-----
automation name:Change Label
event name:Click
event target:[object AutomationManager]
args:
-----
```

Using LiveCycle Data Services

This section describes using LiveCycle Data Services to compile your MXML file. In this case, you do not need to create a wrapper; instead, you request the MXML file directly. The server's built-in web-tier compiler generates one for you.

1. Run the automation installer (available as a separate download). Otherwise, you do not have automation_agent.swc or the related resource bundle SWC file.
2. Copy the automation_agent.swc file to the /WEB-INF/flex/libs directory.
3. Copy the automation_agent_rb.swc to the /WEB-INF/flex/locale/en_US directory.
4. Copy the CustomAgent.swc file to the /WEB-INF/flex/user_classes directory.
5. Edit the flex-config.xml file and add the automation SWC files to the <include-libraries> entry, as the following example shows:

```
<include-libraries>
```

```
<library>user_classes/CustomAgent.swc</library>
<library>../libs/automation.swc</library>
<library>../libs/automation_agent.swc</library>
</include-libraries>
```

6. Copy the AutomationGenericEnv.mxml file to the same directory as Main.mxml.
7. Restart your LiveCycle Data Services server instance.
8. Create an MXML file as described in “Create and run the Flex application” on page 57.
9. Request the MXML file and interact with the application as described in “Create and run the Flex application” on page 57.

Accessing user session data

To access user session data, you must have some server-side code that can access the FlexContext object. This section describes how to create a POJO class that can be called as a RemoteObject. This section assumes that you are using LiveCycle Data Services as the back end.

1. Create the Java class that will be your RemoteObject.

2. Import `flex.messaging.*` in that class. Use the various APIs, such as the servlet API, to define methods that return the info you want to record about the user session. The following is an example class called `UsefulRemoteObject`:

```
package myR0Package;

import flex.messaging.*;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class UsefulRemoteObject {

    public HttpServletRequest request;
    public FlexSession session;

    public UsefulRemoteObject() {
        request = FlexContext.getHttpRequest();
        session = FlexContext.getFlexSession();
    }

    public String getSessionId() throws Exception {
        String s = new String();
        s = (String) session.getId();
        return s;
    }

    public String getHeader(String h) throws Exception {
        String s = new String();
        s = (String) request.getHeader(h);
        return h + "=" + s;
    }
}
```

For details on using the `FlexContext`, `FlexSession`, and other APIs, see the `LiveCycle Data Services` public APIs at http://www.adobe.com/go/lcds_javadoc.

For details on using the `HttpServletRequest` API, see the `JavaDocs` at <http://java.sun.com/javase/5/docs/api/>.

3. Compile your Java class by adding the `flex-messaging.jar` file to your classpath (in addition to the `servlet.jar` file); for example:

```
javac -classpath
c:/home/dev/depot/flex/sdk/modules/profiler/lib/servlet.jar;
c:/lcds/jrun4/servers/default/flex/WEB-INF/lib/flex-messaging.jar c:/
myfiles/flex2/agent/myR0Package/UsefulRemoteObject.java
```

This example also adds the `Servlet.jar` file because the example uses the `HttpServletRequest` class. As an alternative if you are using JRun, you can use the `jrunit.jar` file in `/jrun4/lib/jrunit.jar`.

4. Copy the new class file (`UsefulRemoteObject.class`) to the `/WEB-INF/classes/packagename` directory.
5. Edit the `/WEB-INF/flex/remoting-config.xml` file to add a new destination; for example:

```
<destination id="myRODestination">
  <properties>
    <source>myROPackage.UsefulRemoteObject</source>
  </properties>
</destination>
```

6. Add a call to the `RemoteObject` in the `CustomAdapter` class's `recordHandler()` method:

```
import mx.rpc.remoting.RemoteObject;
import mx.rpc.events.*;
import mx.controls.Alert;

private var ro:RemoteObject;
...
/* Access the methods of the RemoteObject */

// Instantiate RemoteObject and set the destination.
ro = new RemoteObject();
ro.destination = "myRODestination";

// Add event handlers for the RO's methods and faults.
ro.getSessionId.addEventListener("result", getResultHandler);
ro.getHeader.addEventListener("result", getResultHandler);
ro.addEventListener("fault", faultHandler);

// Call the RO's methods.
ro.getSessionId();
ro.getHeader("User-Agent");
```

7. Add methods to the `CustomAdapter` class that handle the results of the `RemoteObject`'s methods and faults:

```
public function faultHandler(event:FaultEvent):void {
    Alert.show(event.fault.faultString, 'Error');
}
private function getResultHandler(event:ResultEvent):void {
    trace("RO RESULT: " + event.result);
}
```

8. After changing the CustomAdapter class, you must recompile the CustomAgent.swc file using the instructions in “Edit the CustomAdapter class” on page 55. If you are using LiveCycle Data Services, be sure to and copy the new SWC file to the /WEB-INF/flex/user_classes directory before requesting the MXML file again.

When you run this example and request Main.mxml, the trace log should contain results similar to the following:

```
RO RESULT: 52306a8196a62d01e691
RO RESULT: User-Agent=Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
SV1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
```

You cannot access request query string parameters from a RemoteObject. They would have to be set on the URL for the channel endpoint in order for them to be available in the RemoteObject class. The URL that LiveCycle Data Services uses to load the SWF is not the same as the URL that LiveCycle Data Services uses to contact the channel endpoint to send data to and from a RemoteObject destination.

Creating a replaying agent

This section describes how to create a custom agent that records and replays user interaction with a Flex application. This process is considerably more complex than simply recording user interaction for the purpose of metrics logging.

You cannot use this custom agent with agents that use different environment information, such as the QTP agent included in the Flex Automation Package. As a result, if you use this agent, do not use QTP to record and play back scripts.

Using the AutoQuick example

This section relies on the AutoQuick example available as a separate download (AutoQuick.zip file) at http://www.adobe.com/go/flex_automation_agent_apps.

Before you write a custom replaying agent, you should try the AutoQuick example. This should show you what type of information an agent must record so that it can play back the user interaction.

The AutoQuick example shows you how you can specify codecs in the XML file and create a map of codecs that you can use for value translation.

To use the AutoQuick example:

1. Download and expand the AutoQuick.zip file. This file is available as a separate download from http://www.adobe.com/go/flex_automation_agent_apps.

2. Build the AutoQuick.swc file using the instructions in “Building the AutoQuick.swc file” on page 67.
3. Copy the AutoQuick.swc file to a location that is accessible by your application’s compiler.
4. Include the AutoQuick.swc library, in addition to the automation.swc and automation_agent.swc files in your Flex application.

If you are using Flex Builder, add the AutoQuick.swc file by selecting Project > Properties > Flex Build Path > Library Path. Ensure that the Merged Into Code option is enabled.

If you are using the command-line compiler, add an entry in your flex-config.xml file that points to this SWC file; for example:

```
<include-libraries>
  <library>c:/myfiles/flex2/agent/simpler_replay/AutoQuick.swc</
  library>
  <library>c:/home/dev/depot/flex/frameworks/libs/automation.swc</
  library>
  <library>c:/home/dev/depot/flex/frameworks/libs/
  automation_agent.swc</library>
</include-libraries>
```

Or on the command line, you can add the files by using the include-libraries option; for example:

```
mxm1c -C:\home\dev\depot\flex\sdk\bin>mxm1c -include-libraries+=c:/
myfiles/flex2/agent/simpler_replay/AutoQuick.swc;c:/home/dev/depot/
flex/sdk/frameworks/libs/automation.swc;c:/home/dev/depot/flex/sdk/
frameworks/libs/automation_agent.swc c:/myfiles/flex2/agent/myreplay/
ReplayExample.mxml
```

Be sure to use the += operator to add the SWC file; this ensures that you do not replace other SWC files in your library path, but rather, append these automation SWC files.

5. Create a sample application that allows for some kind of user interaction. For example, you could create a simple application that has just a Button control:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      private function clickHandler():void {
        b1.label = "clicked";
      }
    ]]>
  </mx:Script>
  <mx:Button id="b1" label="click me" click="clickHandler()"/>
</mx:Application>
```

6. If you have not already done so, create a wrapper for the sample application. Flex Builder creates a wrapper for you. If you use the command-line compiler, you must create one. You can use the following sample wrapper:

```
<html lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  <title>ReplayExample</title>
</head>
<body>
  <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
    id="ReplayExample" width="100%" height="100%"
    codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/
    swflash.cab">
    <param name="movie" value="ReplayExample.swf" />
    <param name="quality" value="high" />
    <param name="allowScriptAccess" value="sameDomain" />
    <embed src="ReplayExample.swf" quality="high"
      width="100%" height="100%" name="applicationReplayExample"
      quality="high"
      allowScriptAccess="sameDomain"
      type="application/x-shockwave-flash"
      pluginspage="http://www.macromedia.com/go/getflashplayer">
    </embed>
  </object>
</body>
</html>
```

7. Copy the wrapper and application SWF file to your web server. You must use a web server
8. Copy the AutoQuickEnv.xml file from the AutoQuick.zip file to the same directory on your web server as your sample application's SWF and wrapper files.

Now that you have created an application that uses the AutoQuick toolbar, you can record and play back events with it.

To record with the AutoQuick toolbar:

1. Request the application SWF file. For example:
`http://localhost:8100/replay/default.htm`
2. The application opens with a Button control and the Automation popup window.
3. Click the Record button on the Automation popup.
4. Click the Button that you created in your application. The Button's label should change from "click me" to "clicked".
5. Click the Stop button on the Automation popup. The TextArea of the Automation popup now contains the recorded interaction with the Flex application.

6. Copy the text from the Automation popup to a text editor. It should look like the following:

```
<Records>
  <Step
    id="automationClassName{FlexApplication
      string}automationName{ReplayExample string}label{
      string}automationIndex{index:-1 string}id{ReplayExample string}
      className{ReplayExample string}|automationClassName{FlexButton
      string}automationName{click%20me string}label{click%20me
      string}automationIndex{index:0 string}id{b1 string}className
      {mx.controls.Button string}"
    method="Click">
  <Args value="" />
  </Step>
</Records>
```

The format of the script is to have a `<Records>` tag wrap the entire script. Each event within the script is represented by a `<Step>` tag. Within the step tag can be `<Args>` child tags that describe arguments to the event. For example, if the event is a `MouseEvent.CLICK` event, the args will define what keys were pressed at the time of the event.

The `id` attribute of the `<Step>` tag contains the entire control hierarchy for each target of an event. Each object in the `id` is separated by a pipe (`|`), with the top-most object in the hierarchy occurring first. For example, in an application that has a `TextInput` control inside an `HBox` container, you would have an `id` structure similar to the following:

```
Application | HBox | TextInput
```

Each object in the hierarchy is defined by a set of properties using the following syntax:

```
property_name{value type}
```

The `ReplayExample` sample shows the following information about the application:

```
automationClassName{FlexApplication string}
automationName{ReplayExample string}
label{string}
automationIndex{index:-1 string}
id{ReplayExample string}
className{ReplayExample string}
```

It shows the following information about the `Button`, including the label of the `Button`:

```
automationClassName{FlexButton string}
automationName{click%20me string}
label{click%20me string}
automationIndex{index:0 string}
id{b1 string}
className{mx.controls.Button string}
```

The object definitions contain values for all the attributes that are defined in the `AutoQuickEnv.xml` file for objects of that type. This ensures that the object can be uniquely identified within the script. This entry in the `id` hierarchy identifies the `Button` as an object who's `automationClassName` is `FlexButton`, `automationName` is its label, `label` is "click me", `automationIndex` is 0, `id` is `b1`, and canonical classname is `mx.controls.Button`.

The `method` attribute of the `<Step>` tag defines the event that was recorded.

To play back the recorded session with the AutoQuick toolbar:

1. Restart your Flex application by either refreshing the browser or closing and reopening the browser.
2. Copy the recorded script from your text editor into the Automation popup's `TextArea`.
3. Click the Play button. The `Button`'s label should change from "click me" to "clicked".

About the AutoQuick example

The AutoQuick example uses a variety of assets, including:

- `AQToolBar.mxml` — Defines the floating window that provides record, stop, and play functions. It calls the methods of the helper classes.
- `AQAdapter.as` — Defines the agent class for the AutoQuick example; this class is a mixin, which triggers a call to its static `init()` method from the `SystemManager` when the application start-up.
- `AutoQuickEnv.xml` — Defines what components and their methods, properties, and events can be recorded with the automation API. This is similar to the `AutomationGenericEnv.xml` file that is used by the `CustomAgent` example.
- `codec.*` class files — The classes in the `codec` package are used to convert `ActionScript` types to agent-specific types; you can think of these classes as providing serialization for the `ActionScript` types. For example, the AutoQuick example uses the `DatePropertyDecode` class to convert `Date` objects to an encoded form. The serialized object can then be loaded by the agent at run time.

Customizing the AutoQuick example

Two classes that you will most likely customize are the `AQToolBar.mxml` file and the `AQAdapter.as` file.

Adding functionality to the AQToolBar.xml file is just like editing any other custom Flex component. You can rebrand it, resize it, or change any of its properties to make it work within your application framework. For example, you could add an HTTPService that calls a PHP page that logs the results of the recording. This would allow you to create a web-based, reusable event recording tool, similar to the CustomAgent example.

You can also customize the AQAdapter class; for example, you could add information to what is recorded. The <Step> and <Args> data is defined in the recordHandler() method of this class. You can edit this class to add additional properties. For example, the <Args> values are currently represented as numeric values. If the user holds the Shift key down while clicking a Button control, the agent records <Args value="2"/>. You could add functionality to the class that includes human-readable values such as <Args value="ShiftKey"/>.

Once you have customized the AutoQuick example's source files, you must then rebuild the SWC file. For more information, see "Building the AutoQuick.swc file" on page 67.

Building the AutoQuick.swc file

The AutoQuick.zip file contains all the classes you need to build the AutoQuick.swc file. When you first unzip the file or if you make changes to the source files, you will want to build your own SWC file.

To build your own AutoQuick.swc file from the AutoQuick source files, compile the custom SWC file using the following compc command:

```
C:\home\dev\depot\flex\sdk\bin>compc
-source-path+=c:/myfiles/flex2/agent/replay/AutoQuick
-output c:/myfiles/flex2/agent/replay/AutoQuick.swc
-include-classes AQAdapter AQAutomationClass AQEnvironment
AQEventDescriptor AQMethodDescriptor AQPropertyDescriptor
IAQAutomationClass IAQCodecHelper IAQMethodDescriptor
IAQPropertyDescriptor IAAutomationPropertyCodec
codec.ArrayPropertyCodec codec.AssetPropertyCodec
codec.AutomationObjectPropertyCodec codec.ColorPropertyCodec
codec.DatePropertyCodec codec.DateRangePropertyCodec
codec.DateScrollDetailPropertyCodec codec.DefaultPropertyCodec
codec.HitDataCodec codec.KeyCodePropertyCodec
codec.KeyModifierPropertyCodec codec.ListDataObjectCodec
codec.RendererPropertyCodec codec.ScrollDetailPropertyCodec
codec.ScrollDirectionPropertyCodec codec.TabObjectCodec
codec.TriggerEventPropertyCodec
```

The exact file paths may be different on your machine.