Chapter 34: Importing Flash CS3 Assets into Flex

Adobe Flash CS3 Professional creates SWF files compatible with Adobe Flash Player 9. Adobe Flex applications also support Flash Player 9, which means that you can import assets from Flash CS3 Professional to use in your Adobe Flex applications.

You can create Flex controls, containers, skins, and other assets in Flash CS3 Professional, and then import those assets into your Flex application. Before you can create Flex components in Flash CS3, you must install the Flex Component Kit for Flash CS3.

Contents

Installing the Adobe Flex Component Kit for Flash CS3

This section describes the system requirements and installation instructions for the Flex Component Kit for Flash CS3.

System requirements

Your system must meet the following requirements to install the Flex Component Kit for Flash CS3:

- Flex 3
- Flash CS3 Professional
- Adobe Extension Manager. You can download the Extension Manager from http://www.adobe.com/exchange/em_download/.

Installation procedure

The Flex Component Kit for Flash CS3 consists of the FlexComponentKit.mxp file.

Install the Flex Component Kit for Flash CS3

1. Install Flex.
2. Change the directory to *install_dir*/frameworks/flash-integration.
3. Double-click the FlexComponentKit.mxp file to install the necessary files in Flash CS3.

This file installs the SWC file that contains the classes necessary to create assets compatible with Flex, and adds the Convert Symbol to Flex Component and Convert Symbol to Flex Container options to the Flash CS3 Command menu. For more information, see "Importing dynamic SWF 9 assets" on page 944.

About importing Flash CS3 assets into Flex

Adobe Flash CS3 Professional lets you create many types of assets for use in a Flex application. You can use Flash CS3 to create simple assets such as icons and logos, you can create more complex assets for use as Flex skins, you can create Flex components, and you can create Flex containers.

The way you import these assets depends on the type of asset and where you use the asset in your Flex application. You typically import two types of SWF 9 assets:

Static assets Assets used for simple artwork or skins that do not contain any ActionScript 3.0 code. Import them in the same way that you import SWF 8 assets. For more information, see "Importing static assets" on page 943.

Dynamic assets Assets that correspond to Flex components and skins, or assets that contain ActionScript 3.0 code. These components are designed to work with Flex features such as view states and transitions, skinning, and tool tips. To use dynamic assets in a Flex application, export the assets from Flash CS3 to a SWC file, and then link the SWC file to your Flex application. For more information, see "Importing dynamic SWF 9 assets" on page 944.

Importing static assets

The "Embedding Assets" topic in the *Flex Developer's Guide* describes how to import SWF 8 files into a Flex application. You use the same process to import static SWF 9 assets as you do with SWF 8 assets, where static assets are simple artwork or skins that do not contain any ActionScript 3.0 code.

The following example shows how to import static assets created in Flash CS3:

```
<?xml version="1.0"?>
<!-- embedSWF9/EmbedSimpleSWF9Movie..mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="500" >

    <mx:Script>
       <![CDATA[

         // Embed the SWF file and the symbols from the SWF file.
         [Embed(source="../assets/circlesquare.swf")]
         [Bindable]
         public var LogoCls:Class;

         // Embed the BlueSquare symbol.
         [Embed(source="../assets/circlesquare.swf", symbol="BlueSquare")]
         [Bindable]
         public var LogoClsBlueSquare:Class;

         // Embed the GreenCircle symbol.
         [Embed(source="../assets/circlesquare.swf", symbol="GreenCircle")]
         [Bindable]
         public var LogoClsGreenCircle:Class;
       ]]>
    </mx:Script>

     <!-- Load the SWF file by using the Image control. -->
     <mx:Image source="{LogoCls}"/>
```

```
    <!-- Load the SWF file as the icon for a Button control. -->
    <mx:Button icon="{LogoCls}" height="100" width="200"/>

    <!-- Load the BlueSquare symbol as the icon for a Button control. -->
    <mx:Button icon="{LogoClsBlueSquare}" height="100" width="100"/>

    <!-- Load the GreenCircle symbol as the icon for a Button control. -->
    <mx:Button icon="{LogoClsGreenCircle}" height="100" width="100"/>

    <!-- Use the SWF symbols to skin a Button control. -->
    <mx:Button id="b1" label="Click Me"
       upSkin="@Embed(source='../assets/circlesquare.swf',
           symbol='BlueSquare')"
       overSkin="@Embed(source='../assets/circlesquare.swf',
           symbol='GreenCircle')"
       downSkin="@Embed(source='../assets/circlesquare.swf',
           symbol='BlueSquare')"/>
</mx:Application>
```

Importing dynamic SWF 9 assets

Dynamic SWF 9 assets are Flex components that you create in Flash CS3, and then import into your Flex application. Because these assets are Flex components, you can use them in the same way as you would a component shipped with Flex, or in the same way that you use a custom Flex component that you created in MXML or ActionScript.

To create a Flash component, you define a symbol in your FLA file, and configure the movie clip symbol as a subclass of either of the following classes:

mx.flash.UIMovieClip For assets used as skins or as Flex controls

mx.flash.ContainerMovieClip For assets used as Flex containers

For example, you define a FLA file with four movie clip symbols that represent the following four skins of the Button class:

- Button disabled skin
- Button down skin
- Button over skin
- Button up skin

Each movie clip symbol corresponds to a different subclass of the mx.flash.UIMovieClip class. When you publish your FLA file as a SWC file, the SWC file contains a class definition for each symbol. You can then reference the classes from your Flex application.

Use the following process to import dynamic SWF 9 assets is:

1 Install the Flex Component Kit for Flash CS3. For more information, see "Installing the Adobe Flex Component Kit for Flash CS3" on page 942.

2 Create movie clip symbols for your dynamic assets in the FLA file. For more information, see "Creating a Flash component for a Flex Button skin" on page 945.

3 For a skin or control, run Commands > Convert Symbol to Flex Component to convert your symbol to a subclass of the UIMovieClip class and to configure the Flash CS3 publishing settings for use with Flex. For more information, see "Actions performed by the Convert Symbol to Flex Component command" on page 947.

For a container, run Commands > Convert Symbol to Flex Container to convert your symbol to a subclass of the ContainerMovieClip class and to configure the Flash CS3 publishing settings for use with Flex. For more information, see "Creating a Flash container component" on page 959.

4 Publish your FLA file as a SWC file.

5 Reference the class name of your symbols in your Flex application as you would for any class. For more information, see "Compiling a Flex application that uses a Flash component" on page 948.

6 Include the SWC file in the `library-path` option when you compile your Flex application. For more information, see "Compiling a Flex application that uses a Flash component" on page 948.

Class hierarchy of UIMovieClip

Components created in Flash for use in Flex are subclasses of the mx.flash.UIMovieClip class, and UIMovieClip is a subclass of the flash.display.MovieClip class. The following code shows the definition of the UIMovieClip class:

```
class UIMovieClip extends MovieClip implements IDeferredInstantiationUIComponent,
    IToolTipManagerClient, IStateClient, IFocusManagerComponent, IConstraintClient
```

UIMovieClip implements the interfaces necessary for a Flash component to be used like any other Flex component. Therefore, you can use a subclass of UIMovieClip as a child of a Flex container or as a skin, and it can respond to events, define view states and transitions, and work with effects in the same way as any Flex component.

The mx.flash.ContainerMovieClip is a subclass of the UIMovieClip class. When you create a container in Flash, you create the container as a subclass of the ContainerMovieClip class.

When you install the Flex Component Kit for Flash CS3, you also install the *install_dir*\frameworks\projects\flash_integration\libs\FlexComponentBase.swc file. This SWC file contains all of the information necessary to use the UIMovieClip class in Flash.

For more information on the UIMovieClip and ContainerMovieClip classes, see *Adobe Flex Language Reference*.

About SWC files

You deploy your Flash components in a SWC file. A *SWC* file is an archive file for Flex components and other assets. SWC files contain a SWF file and a catalog.xml file. SWC files make it easy to exchange components and other assets among Flex developers. You exchange only a single file, rather than the MXML or ActionScript files and images and other resource files.

The SWF file in a SWC file is compiled, which means that the code is loaded efficiently and it is hidden from casual view. Also, compiling a component as a SWC file can make namespace allocation an easier process.

For more information on SWC files, see *Building and Deploying Flex Applications*.

Creating a Flash component for a Flex Button skin

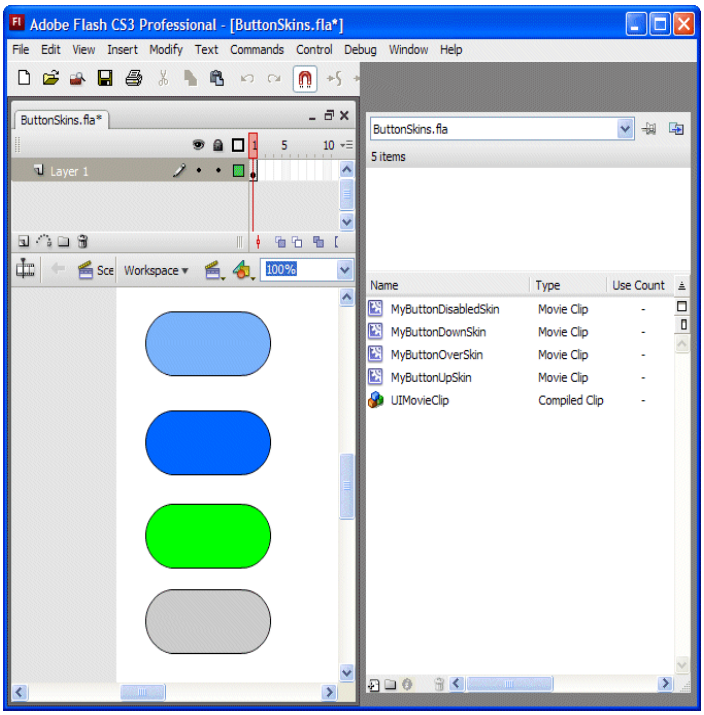
To create a component in Flash, create a symbol in your FLA file that corresponds to the component. Then, use the Convert Symbol to Flex Component command to convert the symbol into a Flex component. This section contains an example that creates four Flash components to use as skins for the Flex Button control. For an example that creates a container in Flash, see "Creating a Flash container component" on page 959.

In the following example, you define four movie clip symbols that correspond to the following four skins of the Button class:

- Button disabled skin: MyButtonDisabledSkin symbol
- Button down skin: MyButtonDownSkin symbol

- Button over skin: MyButtonOverSkin symbol
- Button up skin: MyButtonUpSkin symbol

The following image shows these movie clip symbol definitions for the skins in Flash CS3 after running the Convert Symbol to Flex Component command:



Create a symbol in Flash

1 Ensure that you installed the Flex Component Kit for Flash CS3. For installation instructions, see "Installing the Adobe Flex Component Kit for Flash CS3" on page 942.

2 Flash CS3, select File > New to open the New Document dialog box.

3 Select Flash File (ActionScript 3.0) as the file type.

4 Save the file as ButtonSkins.fla.

5 Create four movie clip symbols, one for each Button skin.

- When you create a symbol, Flash prompts you to specify the symbol name. By default, the symbol name becomes the class name of your Flex component. Also, in Flex Builder, your component's class name is displayed in code hints.

  However, the Convert Symbol to Flex Component command removes all nonalphanumeric characters (including spaces) from the class name and, if the first character is a number, it prefixes the class name with an underscore. Therefore, it is recommended that you use only uppercase and lowercase letters in the symbol name, and do not use spaces.

- Set the registration point to the upper-left corner of the symbol.

If you have Flash content that requires a different registration point, you can wrap the symbol in another symbol with an upper-left registration point. For more information, see "Positioning a Flash component in a Flex container" on page 949.

6 Select the movie clip symbols in the Library panel.

7 Select Commands > Convert Symbol to Flex Component.

This command converts the symbol into a subclass of the UIMovieClip class, and configures the publishing settings of your FLA file for use with Flex. For more information, see "Actions performed by the Convert Symbol to Flex Component command" on page 947.

The Convert Symbol to Flex Component prompts you to set the frame rate of the FLA file to match the default Flex frame rate, which is 24 frames per second. Although you can set the frame rate to a different value, setting it to less than 24 frames per second can make the Flex application appear sluggishness.

After you run this command, a FlexComponentBase compiled clip appears in the Library panel of your Flash application.

8 Select File > Publish to create the SWC file.

The output SWC file is named ButtonSkins.swc.

Actions performed by the Convert Symbol to Flex Component command

The Convert Symbol to Flex Component command performs the following actions to convert a Flash movie clip symbol into a Flex component:

- Removes all nonalphanumeric characters (including spaces) from the class name and, if the first character is a number, it prefixes the class name with an underscore. Therefore, it is recommend that you use only uppercase and lowercase letters in the symbol name, and do not use spaces.
- Sets the base class of the symbol to the mx.flash.UIMovieClip class. In Flash, right-click on the symbol name in the Library panel to open the Linkage Properties dialog box to see this setting.
- Configures the following properties for the symbol:
  - Sets the Export for ActionScript option
  - Sets the Export in first frame option
  - Sets the Class name to be the same as the symbol name
- Configures the following publishing settings for the FLA file:
  - Sets the Version to Flash Player 9
  - Sets the ActionScript version to ActionScript 3.0
  - Selects the Permit Debugging option to let you debug your component in Flex
  - Selects the Export SWC option

  You can see these settings by using the File > Publishing Settings, which opens the Publishing Settings dialog box, and then selecting the Flash tab.
- Optionally sets the Frame Rate of the FLA file to match the default Flex frame rate, which is 24 frames per second. Although you can set the Flex frame rate to a different value, setting it to lees than 24 frames per second can make the application appear sluggishness.

Note: *Instead of using the Convert Symbol to Flex Component command, you can manually configure a symbol to be used as a Flex component by making all of the settings described in this section. However, before you can reference the UIMovieClip class, you must open the Components panel and drag the FlexComponentBase component into the Library panel.*

Compiling a Flex application that uses a Flash component

After you create the SWC file that contains your Flash components, you can use the components in your Flex application. For example, the following application uses the four skin components created in the previous section in a Flex application:

```
<?xml version="1.0"?>
<!-- skins/EmbedSWF9ButtonSkins.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="500"
    xmlns:myComps="*">

    <mx:Button label="Click Me"
        upSkin="MyButtonUpSkin"
        overSkin="MyButtonOverSkin"
        downSkin="MyButtonDownSkin"
        disabledSkin="MyButtonDisabledSkin"/>

    <myComps:MyButtonUpSkin/>
    <myComps:MyButtonOverSkin/>
    <myComps:MyButtonDownSkin/>
    <myComps:MyButtonDisabledSkin/>

</mx:Application>
```

In this example, you use the Flash components to skin the Button control and use the components as stand-alone Flex components. To use a Flash component as a stand-alone Flex component, the main application file must include a namespace definition. The namespace tells Flex where to look for the file that implements the component. By default, Flash creates the class that corresponds to your Flash component in the default package, corresponding to a namespace of "*".

If you define a package name for your Flash component, you must specify the namespace accordingly. For example, suppose you define the package name for your Flash component as the following example shows:

```
package flashComponents {
    // Component definition
}
```

You specify the namespace as "flashComponents.*".

Note: *As a best practice, keep all the files associated with your Flash component in a different directory than the MXML files that reference the Flash component. This is particularly true when you define custom classes as part of creating your Flash components. If the Flash classes and MXML files are in the same directory, the MXML compiler uses the class file directly rather than using the component from the SWC file.*

Compile your application in Flex Builder

1. Open the Project Properties dialog box.
2. Select Flex Build Path, and then select the Library path tab.
3. Click the Add SWC button.
4. In the Add SWC dialog box, click the Browse button.
5. Select your SWC file. From the example in the section "Creating a Flash component for a Flex Button skin" on page 945, the SWC file is named ButtonSkins.swc.

The component now appears in Flex Builder code hints.

6 Add the XML namespace for the component; for example:

```
xmlns:myComps="*"
```

7 Compile your application.

Compile your application by using the Flex mxmlc compiler

1 Include the SWC file in the compilation by using the `library-path` option, as the following example shows:

```
mxmlc --strict=true --show-actionscript-warnings=true --use-network=true --library-
path+=../SWF9SWC --file-specs EmbedSWF9ButtonSkins.mxml
```

In this example, the SWC file is in the SWF9SWC directory. From the example in the section "Creating a Flash component for a Flex Button skin" on page 945, the SWC file is named ButtonSkins.swc.

Using Flash components in a Flex application

You use components that you create in Flash anywhere in a Flex application that you use any other type of Flex component. Therefore, the components can be children of a container, can dispatch and respond to events, work with effects, and perform just about any other action that a standard Flex component can perform.

Accessing public properties in MXML

All public properties defined in your Flash component are accessible in MXML by using MXML tag properties. You can define public properties by using variables, or by using setter and getter methods. For more information, see "Creating Simple Visual Components in ActionScript" on page 75 in *Creating and Extending Flex Components*.

Using a Zoom effect with a Flash component

Flash components do not support the Zoom effect. As a work around, put the Flash component inside a Flex container and apply the Zoom effect to the Flex container.

Positioning a Flash component in a Flex container

Flex containers position child components by setting the x and y coordinates of the component's upper-left corner within the layout area of container. For components created in Flash, containers uses the registration point of the component to position it. Therefore, you should set the registration point of your Flash component to its upper-left corner. If you have Flash component that requires a different registration point, wrap the component in a new symbol with a registration point in the upper-left corner.

Setting focus

For a Flash component, the individual Flash symbols in the component can receive focus if they set the `tabEnabled` property to `true`. Focus is indicated by the standard yellow highlighting of Flash Player. To control the appearance of the focus highlighting, handle the `focusIn` and `focusOut` events for the associated Flash symbol.

In a Flash container, only the Flex components in the content area of the Flash container can receive focus. The Flex components use the standard Flex highlighting to display focus. However, Flash symbols in the Flash container cannot receive focus. For more information on creating a Flash container, see "Creating a Flash container component" on page 959.

Adding custom events

The UIMovieClip class supports many of the same events that you use with any Flex component, such as `mouseOver`, `move`, `initialize`, and others. You can also add your own custom events to a Flash component for use in your Flex application. To add custom events, create an ActionScript file for your class definition, as the following example shows:

```
package {

    // EventBlueSquare.as

    import mx.flash.UIMovieClip;
    import flash.events.Event;

    [Event(name="blueSquareSelected", type="flash.events.Event")]

    public class EventBlueSquare extends UIMovieClip
    {
        public function EventBlueSquare() {
            super();
            addEventListener('mouseDown', dispatchMyEvent);
        }

        protected function dispatchMyEvent(event:Event):void {
            dispatchEvent(new Event("blueSquareSelected"));
        }
    }
}
```

After you create this file, you must create a symbol named EventBlueSquare in your FLA file, and then use the Convert Symbol to Flex Component command to convert that symbol to a Flex component.

You can use your custom event from a Flex application, as the following example shows:

```
<?xml version="1.0"?>
<!-- embedSWF9/EmbedSWF9Event.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:myComps="*">

   <mx:Script>
        <![CDATA[

            public function handleBlueSquareSelected():void {
                myTA.text='blueSquareSelected event occured.';
            }
        ]]>
    </mx:Script>

    <myComps:EventBlueSquare id="myBS"
        blueSquareSelected="handleBlueSquareSelected();"/>

    <mx:TextArea id="myTA"/>
</mx:Application>
```

For more information on defining custom events, see "Creating Custom Events" on page 21 in *Creating and Extending Flex Components*.

Adding tool tips to Flash components

The UIMovieClip class implements the IToolTipManagerClient interface, which includes the `toolTip` property. If you set the `toolTip` property in your Flash component, the tool tip appears in your Flex application when the mouse moves over the component.

For example, to add a tool tip to the EventBlueSquare component that is defined in the section "Adding custom events" on page 950, you modify the constructor to set the `toolTip` property, as the following example shows:

```
public function EventBlueSquare() {
    super();
    addEventListener('mouseDown', dispatchMyEvent);
    toolTip = 'blue square component';
}
```

You can also set the tool tip in MXML when you use the component in a Flex application, as the following example shows:

```
<?xml version="1.0"?>
<!-- embedSWF9/EmbedSWF9ToolTip.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:myComps="*">

    <myComps:EventBlueSquare id="myBS"
        toolTip="MXML Tooltip"/>
</mx:Application>
```

For more information on tool tips, see "Using ToolTips" on page 767 in *Flex Developer's Guide*.

Controlling the size of a Flash component

By default, the measured size of a Flash component at run time matches its actual size. Any run-time changes to the component's size are recognized by Flex, and Flex updates the layout of your application with the component's new size.

You should be aware of some important resizing considerations:

- Typically, the Flash `width` and `height` properties determine the size of the component in Flex.
- Shape tweens are only resized on key frames.
- Masked content is reported at its full size, not its visual size.

There are several situations when you might have to add functionality to your Flash component to control its size:

1 The component resizes itself at run time because of an animation, tween, or other reason. You can add functionality to the Flash component to prevent Flex from having to update the component's size. For more information, see "Adding a bounding box to a Flash component" on page 951.

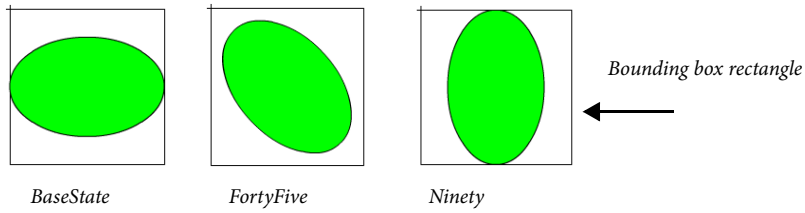
2 Flex resizes the component at run time and you require the component to perform an action based on its new size. For example, you might have a component that can modify its appearance as it gets larger or smaller. For more information, see "Overriding the setActualSize() method" on page 952.

Adding a bounding box to a Flash component

You might add animations or other effects to your Flash component that change its size at run time. By default, Flex sets the size of a Flash component at run time to its actual size. Therefore, when your component changes size, Flex must update the layout of your application; otherwise, the new size of your component might cause it to overlap other components.

One way to control the size of a Flash component is to add a bounding box to the component. The bounding box is a MovieClip instance that Flex uses to size the Flash component at run time. Within the area of the bounding box, the Flash component can perform animations or modify its visual characteristics. But, as long as the area of the bounding box does not change, Flex does not resize the component.

The following image shows a Flash component named StatesGreenOval with three view states:



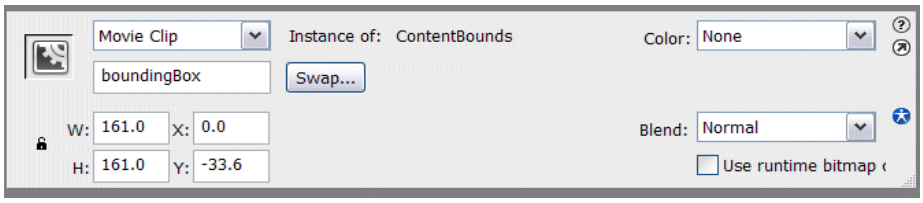
The rectangle surrounding the green oval is the movie clip symbol that corresponds to the bounding box. As the green oval rotates, it stays within the area defined by the bounding box, so Flex does not attempt to resize it.

The UIMovieClip class defines a property named `boundingBoxName` that specifies the bounding box MovieClip for the Flash component. The default value of the `boundingBoxName` property is `"boundingBox"`. Therefore, you typically specify `"boundingBox"` as the instance name of the bounding box MovieClip. If you use a different instance name for the MovieClip, ensure that you set the `boundingBoxName` property to that name.

The Convert Symbol to Flex Component command sets the `visible` property to `false` for the MovieClip that represents the bounding box so that it does not appear in your Flex application.

Add a bounding box to a Flash component

1. In Flash CS3, add a new layer to the symbol definition of your component.
2. Use the Rectangle tool to draw a bounding box rectangle.
3. Set the size of the rectangle.
4. Convert the rectangle to a symbol of type MovieClip.
5. In the Property Inspector for the movie clip, specify `boundingBox` for the instance name of the MovieClip, as the following image shows:



Overriding the setActualSize() method

To make your component able to recognize when Flex resizes it at run time, you override the component's `setActualSize()` method. Flex calls the component's `setActualSize()` method at run time, passing to it the height and width that Flex has allocated for the component.

In the following example, you define a Flash component named DotPattern that lays out a grid of instances of the YellowDot MovieClip. The number of rows and columns of YellowDot instances in the component is determined by the run-time size of the DotPattern component.

```
package {
    import mx.flash.UIMovieClip;

    public dynamic class DotPattern extends UIMovieClip
    {
        private const SIZE:Number = 36; // Size of column/row
        private const GAP:Number = 4;

        public function DotPattern():void
        {
        }

        // Set the default width of the component, in pixels,
        // to be wide enough for 2 columns plus the gap between them.
        override public function get measuredWidth():Number
        {
            return SIZE + GAP + SIZE;
        }

        // Set the default height of the component, in pixels,
        // to be tall enough for 2 columns plus the gap between them.
        override public function get measuredHeight():Number
        {
            return SIZE + GAP + SIZE;
        }

        // Set the number of visble dots at run time based on the size
        // Flex passes to the component at run time.
        override public function
            setActualSize(newWidth:Number, newHeight:Number):void
        {
            if (newWidth != _width || newHeight != _height)
            {
                _width = newWidth;
                _height = newHeight;

                // Clear out our existing children.
                for (var i:int = 0; i < numChildren; i++)
                {
                    removeChildAt(0);
                }

                // Figure out how many rows/columns to create.
                var numRows:int = Math.floor((newHeight) / (SIZE + GAP));
                var numCols:int = Math.floor((newWidth) / (SIZE + GAP));

                // Create and place children
                for (var row:int = 0; row < numRows; row++)
                {
                    for (var col:int = 0; col < numCols; col++)
                    {
                        var dot:YellowDot = new YellowDot();

                        addChild(dot);
                        dot.x = col * (SIZE + GAP);
                        dot.y = row * (SIZE + GAP);
                    }
                }
            }
        }
    }
```

```
}
```

Use the DotPattern component in a Flex application, as the following example shows:

```
<?xml version="1.0"?>
<!-- embedSWF9/EmbedSWF9SetActualSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100%" height="100%"
    xmlns:myComps="*">

    <myComps:DotPattern width="25%" height="25%"/>

    <myComps:DotPattern width="75%" height="75%"/>
</mx:Application>
```

In this example, you use percent values to size the two instances of the DotPattern component. As you resize the browser window at run time, the number of yellow dots displayed by the component changes as the component changes size.

Adding view states and transitions to Flash components

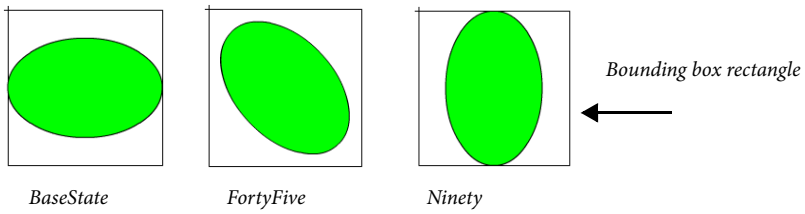
View states let you change appearance of an application, typically in response to a user action. Transitions define how a change of view state looks as it occurs on the screen. You can create Flash components that support view states and transitions for use in your Flex application.

Adding view states

Each view state of a Flash component corresponds to a specific key frame and frame label. For example, to create a component with three view states, you define three key frames and add a frame label to each key frame.

Note: *Ensure that you add the key frames and frame labels to the symbol definition in Flash, not to the scene.*

The following image shows a Flash component named StatesGreenOval with three view states:



In the Flex application, you change the view state of the component by setting its `currentState` property to the frame label, as the following example shows:

```
<?xml version="1.0"?>
<!-- embedSWF9/EmbedSWF9ViewStates.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:myComps="*">

    <myComps:StatesGreenOval id="myOval"/>

    <mx:Button label="BaseState" click="myOval.currentState='BaseState';"/>
```

```
    <mx:Button label="FortyFive" click="myOval.currentState='FortyFive';"/>
    <mx:Button label="Ninety" click="myOval.currentState='Ninety';"/>
</mx:Application>
```

Adding transitions

A Flash component can define an animation that plays during a view state change, where the animation corresponds to a Flex transition. For example, if the change of view state alters the color, size, or other visual aspect of the component, the transition animates that change.

To define an animation in a Flash component that corresponds to a Flex transition, you create two key frames and associated frame labels. The first key frame marks the beginning of the animation, and the second key frame marks the end. The frame labels must adhere to the following naming convention:

- Beginning key frame: *currentViewState*-*destinationViewState*:`start`
- Ending key frame: *currentViewState*-*destinationViewState*:`end`

For example, if the name of the current view state is BaseState, and the name of the destination view state is Ninety, the frame labels for the associated key frames of the transition must be:

```
BaseState-Ninety:start
```

```
BaseState-Ninety:end
```

When you change the view state of the Flash component from BaseState to Ninety, Flex automatically looks for frame labels that define the beginning and end of the transition and, if found, plays the associated animation. When you switch back from the view state Ninety to BaseState, Flex automatically looks for frame labels in the form:

```
Ninety-BaseState:start
```

```
Ninety-BaseState:end
```

If found, Flex automatically plays the animation. If these labels are not found, Flex plays the animation for the BaseState to Ninety transition in reverse.

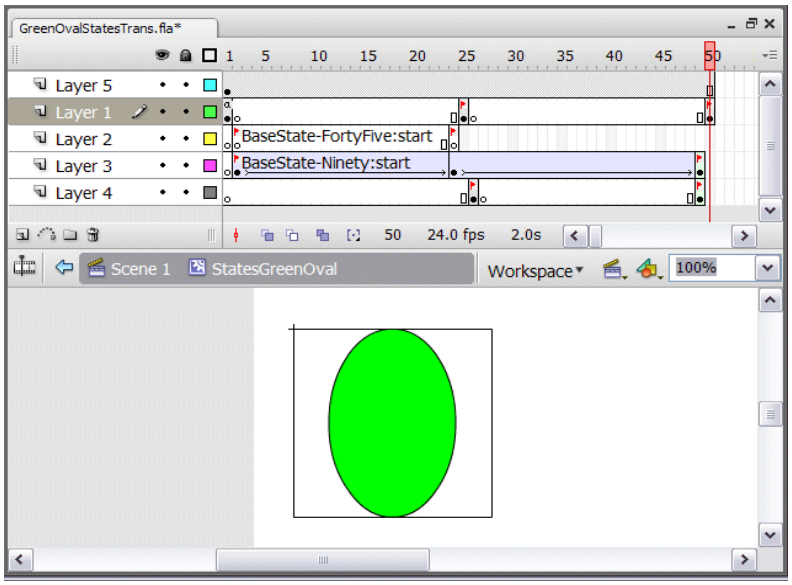
You can use the wildcard symbol, "*", when specifying the frame label, where the wildcard symbol corresponds to any view state. For example, the frame label `BaseState-*:start` specifies a transition from the BaseState view state to any other view state. The frame label `*-BaseState:start` specifies a transition from any view state to BaseState.

Flex searches for frame labels that match the view state change in the following order:

- *currentViewState*-*destinationViewState* (no wildcards)
- *destinationViewState*- *currentViewState* (reversed)
- `*`-*destinationViewState*
- *destinationViewState*-`*` (reversed)
- *currentViewState*-`*`
- `*`-*currentViewState* (reversed)
- `*-*`

This list shows that Flex searches first for frame labels with no wildcards, both forward and reverse, before checking for frame labels that use wildcards. Then, Flex searches for a frame label that uses a wildcard for the current view state, both forward and reverse, followed by a search for a frame label that uses a wildcard for the destination view state. Lastly, Flex searches for a frame label that uses wildcards for both the current and destination view states.

The following image shows a green oval in Flash CS3:



In this example, the layers define the following information:

Layer 1 Specifies three key frames at frames 1, 25, and 50 that define the view states for the green oval. The oval is either at an angle of 0, 45, or 90 degrees. The image shows the 90 degree rotation.

Layer 2 Defines the following two key frames and frame labels for the transition from the BaseState view state to the FortyFive view state:

`BaseState-FortyFive:start`

`BaseState-FortyFive:end` (not shown in the image)

Layer 3 Defines two key frames, frame labels, and a motion tween for the transition from the BaseState view state to the Ninety view state. The motion tween defines the animation that plays when you change view states. The frame labels are:

`BaseState-Ninety:start`

`BaseState-Ninety:end` (not shown in the image)

Layer 4 Defines two key frames and frame labels for the transition from the FortyFive view state to the Ninety view state:

`FortyFive-Ninety:start` (not shown in the image)

`FortyFive-Ninety:end` (not shown in the image)

The following Flex application uses the transition when changing view state:

```
<?xml version="1.0"?>
<!-- embedSWF9/EmbedSWF9ViewStatesTrans.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:myComps="*">

    <mx:Button label="BaseState" click="myOval.currentState='BaseState';"/>
    <mx:Button label="FortyFive" click="myOval.currentState='FortyFive';"/>
```

```
    <mx:Button label="Ninety" click="myOval.currentState='Ninety';"/>

    <myComps:StatesGreenOval id="myOval" width="161" height="161"/>

</mx:Application>
```

Using tweens in a transition

The previous example uses a motion tween to define the animation used by all the transitions for the component. By default, Flex plays the tween when it loads the Flash component. To prevent Flex from playing the tween when the application loads, add the `stop()` statement to the Actions panel of Frame 1 of Layer 1.

Using Flash components as skins

You can use Flash components as Flex skins by creating a different Flash component for each skin, or by creating a stateful Flash component that you can use for multiple skins.

Creating a Flash component for each skin

To use a Flash component as a Flex skin, create a movie clip symbol as a subclass of UIMovieClip, publish it in a SWC file with other Flash components, and then use the class name as the skin of a Flex component. This procedure is very much like using a programmatic skin, as described in "Creating Skins" on page 659.

For an example that uses a different Flash component for each skin of the Flex Button control, see "Compiling a Flex application that uses a Flash component" on page 948.

Creating stateful skins

Many Flex components, such as Button, Slider, and NumericStepper, support stateful skins. A stateful skin uses view states to specify the skins for the different states of the component. For more information on stateful skins, see "Creating stateful skins" on page 691.

To create a Flash component that implements stateful skins, you define view states in the Flash component for the different states of the component. Because a Flash components can have multiple view states, you can create a single component that defines multiple skins.

For example, a Button control has eight possible states, and eight associated skin properties, as the following table shows:

Skin property	State name	Default skin class
`downSkin`	down	mx.skins.halo.ButtonSkin
`overSkin`	over	mx.skins.halo.ButtonSkin
`upSkin`	up	mx.skins.halo.ButtonSkin
`disabledSkin`	disabled	mx.skins.halo.ButtonSkin
`selectedDisabledSkin`	selectedDisabled	mx.skins.halo.ButtonSkin

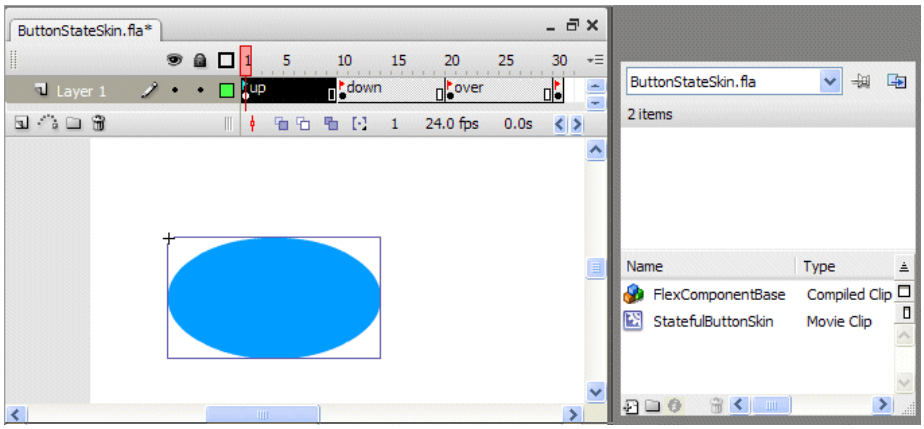
Skin property	State name	Default skin class
`selectedDownSkin`	selectedDown	mx.skins.halo.ButtonSkin
`selectedOverSkin`	selectedOver	mx.skins.halo.ButtonSkin
`selectedUpSkin`	selectedUp	mx.skins.halo.ButtonSkin

You can create a Flash component that defines a stateful skin for one, two, or more states of the Button control.

Each view state of a Flash component corresponds to a specific key frame and frame label. For example, to create a component with four view states, you define four key frames and add a frame label to each key frame. For more information on creating view states in a Flash component, see "Adding view states and transitions to Flash components" on page 954.

Note: *The frame label of the view state in the Flash component must match the state name of the Flex component. The state name is typically the name of the corresponding skin property, without the word "skin". For example, in the previous table for the Button control, the skin property* `downSkin` *defines the skin for the down state of the Button control.*

The following image shows a Flash component named StatefulButtonSkin with four view states; up, down, over, and disabled, that correspond to four states of the Button control:



The view states each define a different color for the Button control, in the same way as did the skins in the section "Creating a Flash component for a Flex Button skin" on page 945.

After publishing the SWC file that contains the definition of the StatefulButtonSkin component, use the component in your Flex application, as the following example shows:

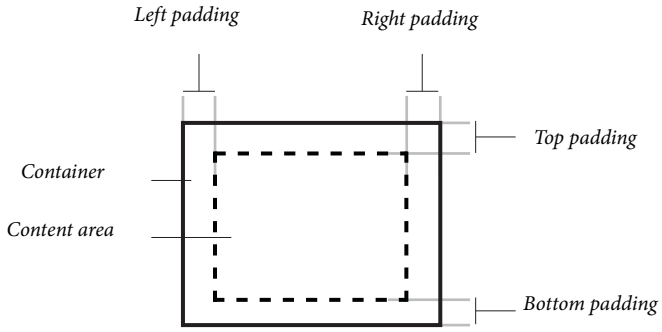
```
<?xml version="1.0"?>
<!-- skins/EmbedSWF9StatefulButtonSkins.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Button label="Click Me"
        skin="StatefulButtonSkin"/>
</mx:Application>
```

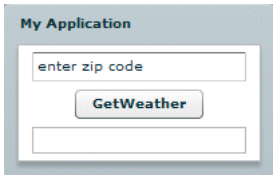
Creating a Flash container component

A Flex *container* defines a rectangular region of the drawing surface of Adobe Flash Player. Within a container, you define the components, both controls and containers, that you want to appear within the container. Components defined within a container are called *children* of the container.

The rectangular region of a container encloses its *content area*, the area that contains its child components. The size of the region around the content area is defined by characteristics of the container, such as the padding and the width of the container border. The following image shows a container and its content area, padding, and borders:



The primary use of a container is to arrange its children, where the children are either controls or other containers. The following image shows a simple Panel container that has three child components:

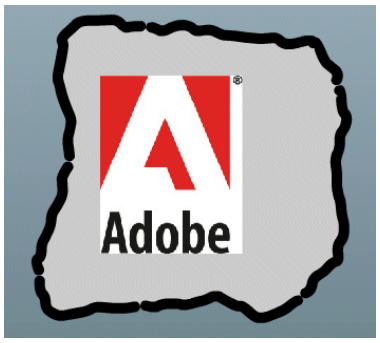


You can create Flex container in Flash CS3, which lets you define the appearance of the container in Flash, including the characteristics of the container padding and boundaries, and then use the container in Flex to add Flex components to the container's content area.

Create a Flex container in Flash CS3 much in the same way that you create a Flex component. First, define a movie clip symbol for the container, then use the Convert Symbol to Flex Container command to convert the symbol to a subclass of the mx.flash.ContainerMovieClip class. You then add the FlexContentHolder symbol to your container to define the container's content area.

Creating a simple Flash container

In this section, you create a simple Flash container, named MyPictureFrameContainer, that defines a picture frame for an image loaded by the Flex Image control. The following image shows this container in a Flex application.



The following example, named PictureFrame.mxml, uses the MyPictureFrameContainer container:

```
<?xml version="1.0"?>
<!-- embedSWF9/PictureFrame.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:myComps="*">

    <myComps:MyPictureFrameContainer>
        <mx:HBox
            width="100%" height="100%">
            <mx:Image
                source="../assets/logowithtext.jpg"
            width="80%" height="80%"/>
        </mx:HBox>
    </myComps:MyPictureFrameContainer>
</mx:Application>
```

Create the MyPictureFrameContainer container

1 Create a FLA file named PictureFrameContainer.fla.

2 Create a movie clip symbol in the FLA file named MyPictureFrameContainer with the outline of your picture frame.

3 Select the MyPictureFrameContainer symbol in the Library panel.

4 Select Commands > Convert Symbol to Flex Container.

The Convert Symbol to Flex Container command performs the same actions as does the Convert Symbol to Flex Component command, except that it sets the base class of the symbol to mx.flash.ContainerMovieClip instead of to mx.flash.UIMovieClip. For more information, see "Actions performed by the Convert Symbol to Flex Component command" on page 947.

The Convert Symbol to Flex Container command also adds a new symbol named FlexContentHolder to the Library. This symbol defines the content area of the container in which you can place child Flex components.

5 Double click on the MyPictureFrameContainer symbol to make sure that it is selected.

6 Drag the FlexContentHolder symbol to the stage to place it within the boundaries of the MyPictureFrameContainer symbol.

7 Position and size the FlexContentHolder symbol to define where to place the Flex components.

8 Publish the PictureFrameContainer.fla file as a SWC file named PictureFrameContainer.swc.

9 Compile PictureFrame.mxml, as described in the section "Compiling a Flex application that uses a Flash component" on page 948.

Adding a child to the ContainerMovieClip.content property

Typically, to add a child to a container in ActionScript, you use the `Container.addChild()` or `Container.addChildAt()` method. However, to add a child to the `ContainerMovieClip.content` property of a Flash container, you do not use the `addChild()` or `addChildAt()` method. Instead, you assign the child directly to the `content` property.

The following example modifies the example in the section "Creating a simple Flash container" on page 960 to add the Image control and HBox container to the MyPictureFrameContainer in ActionScript. Notice in this example that you assign the HBox container to the `content` property:

```
<?xml version="1.0"?>
<!-- embedSWF9/PictureFrameContainer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:myComps="*">

    <mx:Script>
        <![CDATA[
            import mx.containers.HBox;
            import mx.controls.Image;

            private function init():void {
                // Define the Image control.
                var image1:Image = new Image();
                image1.source = "../assets/logowithtext.jpg";
                image1.percentWidth = 80;
                image1.percentHeight = 80;

                // Define the HBox container.
                var hb1:HBox = new HBox();
                hb1.percentWidth = 100;
                hb1.percentHeight = 100;
                hb1.addChild(image1);

                // Assign the HBox container to the
                // ContainerMovieClip.content property.
                pFrame.content = hb1;
            }
        ]]>
    </mx:Script>

    <myComps:MyPictureFrameContainer id="pFrame"
        initialize="init();"/>
</mx:Application>
```

Considerations when creating a Flash container

When you include Flex content in Flash component, consider the following issues:

- The base class of a Flash container is ContainerMovieClip.
- A Flash container can have only a single Flex child. However, this child can be a Flex container, which lets you add additional children.

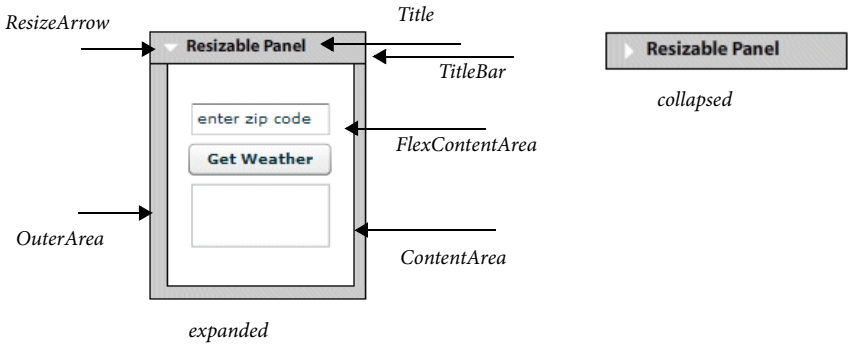
- The `ContainerMovieClip.content` property is the default property of the ContainerMovieClip class. Therefore, when using a Flash container in MXML, you can omit the `ContainerMovieClip.content` property.

  However, if you create an MXML component based on the Flash container, you must specify the `ContainerMovieClip.content` property. For more information on defining MXML components and using the default class property, see "Creating Simple MXML Components" on page 53 in *Creating and Extending Flex Components*.

- If your Flash container modifies the visual characteristics of the Flex components contained in it, such as the `alpha` property, you must embed the fonts used by the Flex components.

Creating an advanced Flash container

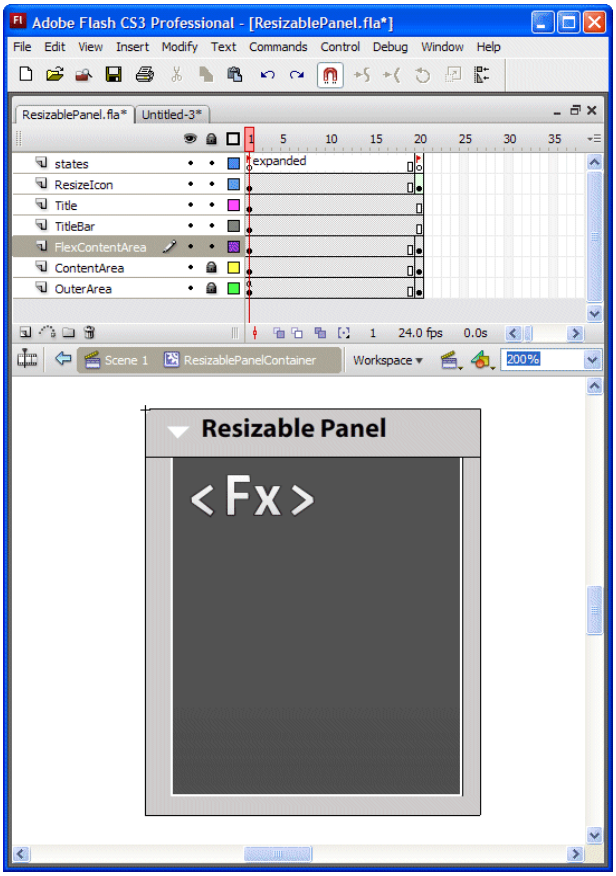
In this example, you create a container in Flash that lets the user expand and collapse the container in response to a mouse click. The following image shows this container:



The following symbols define the parts of this container:

Symbol name	Description
OuterArea	Defines the background color of the container.
ContentArea	Defines the white background for the area in which you place Flex controls.
FlexContentArea	Defines the area for the Flex components. This area is sized to match the size of the ContentArea.
TitleBar	Defines the area of the container in its collapsed state.
Title	Defines the title text.
ResizeArrow	Defines the icon use to collapse the container. The arrow points down when the container is expanded, and to the right when it is collapsed. Click the arrow to minimize the container, and click it again to restore it to its original size.

The following image shows this container in Flash CS3:



To add the animation to the container, you use view states. The states layer defines the following two view states:

expanded The container displays in its full, expanded size, and the resize arrow points down.

collapsed The OuterArea, ContentArea, and FlexContentArea collapse to the size of the TitleBar, hiding all of the Flex components in the FlexContentArea. The resize arrow points to the right.

Note: *You should never remove the FlexContentHolder symbol from a frame of your Flash container. To hide the FlexContentHolder symbol, set its* `height` *and* `width` *properties to 0, or set its* `alpha` *property to 0.*

To handle the state change in response to a click on the arrow, you write the following class file for the ResizablePanelContainer class. This class defines an event handler for the ResizeArrow that changes the current view state of the container in response to a mouse click:

```
package {
    import flash.display.MovieClip;
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import mx.flash.ContainerMovieClip;

    public class ResizablePanelContainer extends ContainerMovieClip {

        // Constructor
        public function ResizablePanelContainer()
```

```
        {
            ResizeArrow.addEventListener(MouseEvent.CLICK, arrowHandler);
            ResizeArrow.useHandCursor = false;
        }

        // Event handler to change view state.
        private function arrowHandler(event:MouseEvent):void
        {
            currentState = currentState ==
                "expanded" ? "collapsed" : "expanded";
        }
    }
}
```

You can animate the change in view state by adding a transition or a tween to the container. For more information on using view states, transitions, and tweens in a Flash component, see "Adding view states and transitions to Flash components" on page 954.