Chapter 1: Advanced Spark Visual Components in ActionScript

You can create advanced visual components for use in Adobe® Flex® applications. Advanced visual components override methods of the UIComponent or SkinnableComponent base classes.

About creating advanced Spark components

Simple visual components are subclasses of existing Flex components that modify the appearance of the component by using skins or styles, or add new functionality to the component. For example, you add a new event type to a Button control, or modify the default styles or skins of a DataGrid control. For more information, see Simple Visual Components in ActionScript.

In advanced components, you typically perform the following actions:

- Modify the basic functionality or logic of an existing component.
- Create a composite component that encapsulates two or more components within it.
- Create a component by creating a subclass of the SkinnableComponent class.

This topic contains several examples of Spark ActionScript components. For more examples, examine the source code for the Spark components in your Flex installation directory. The spark.components and spark.components.supportClasses packages contain many of the Spark components mentioned in this topic.

Spark component and skin classes

When creating an advanced Spark component in ActionScript, you typically create two classes: the component class and the skin class.

The component class defines the core behavior of the component. This behavior includes defining the events dispatched by the component, the data that the component represents, the skin parts implemented by the skin class, and the view states that the skin class supports.

The skin class manages the visual appearance of the component and create visual subcomponents. The skin class defines the default layout of the component, its default size, the supported view states, graphics, and data representation.

About overriding protected UIComponent methods

All Flex visual components are subclasses of the UIComponent class. Therefore, visual components inherit the methods, properties, events, styles, and effects defined by the UIComponent class.

To create an advanced visual component, you must implement a class constructor. Also, you optionally override one or more of the following protected methods of the UIComponent class:

UIComponent method	Description
`commitProperties()`	Commits any changes to component properties, either to make the changes occur at the same time or to ensure that properties are set in a specific order. For more information, see "Implementing the commitProperties() method" on page 8.
`createChildren()`	Creates any child components of the component. For example, the Halo ComboBox control contains a Halo TextInput control and a Halo Button control as child components. Typically, you do not implement this method in a Spark component because any child components are defined in the skin class. This topic does not describe how to implement the `createChildren()` method. For more information, see the example for creating a Halo component in Implementing the createChildren() method.
`measure()`	Sets the default size and default minimum size of the component. You typically do not have to implement this method for Spark components. The default size of a Spark component is defined by the skin class, and by the children of the skin class. You also set the minimum and maximum sizes of the component in the root tag of the skin class. This topic does not describe how to implement the `measure()` method. For more information, see the example for creating a Halo comonent in Implementing the measure() method.
`updateDisplayList()`	Sizes and positions the children of the component on the screen based on all previous property and style settings, and draws any skins or graphic elements used by the component. The parent container for the component determines the size of the component itself. Typically, you do not have to implement this method for Spark components. A few Spark components, such as spark.components.supportClasses.SkinnableComponent, do implement it. The SkinnableComponent class implements it to pass sizing information to the component's skin class. This topic does not describe how to implement the `updateDisplayList()` method. For more information, see the example for creating a Halo comonent in Implementing the updateDisplayList() method.

Component users do not call these methods directly; Flex calls them as part of the initialization process of creating a component, or when other method calls occur. For more information, see About the component instantiation life cycle.

About overriding SkinnableComponent methods

All Spark visual components are subclasses of the SkinnableComponent class. Therefore, visual components inherit the methods, properties, events, styles, and effects defined by the SkinnableComponent class.

To create an advanced visual Spark component, you optionally override one or more of the following methods of the SkinnableComponent class:

SkinnableComponent method	Description
`attachSkin()detachSkin()`	Called automatically by the `UIComponent.commitProperties()` method when a skin part is added, `attachSkin()`, or a skin part is removed, `detachSkin()`. You can optionally implement these methods to add a specific behavior to a skin. Typically you do not implement these methods.
`partAdded()partRemoved()`	Called automatically to add or remove a skin part. You typically override `partAdded()` to attach event handlers to the skin part, configure the skin part, or perform other actions when a skin part is added. Implement the `partRemoved()` method to remove the even handlers added in `partAdded()`. For more information, see "Implementing the partAdded() and partRemoved() methods" on page 12.
`getCurrentSkinState()`	Called automatically by the `UIComponent.commitProperties()` method to set the view state of the skin class. For more information, see "Implementing the getCurrentSkinState() method" on page 13.

Component users do not call these methods directly; Flex calls them as part of the initialization process of creating a component, or when other method calls occur. For more information, see About the component instantiation life cycle.

About the invalidation methods

During the lifetime of a component, your application might modify the component by changing its size or position, modifying a property that controls its display, or modifying a style or skin property of the component. For example, you might change the font size of the text displayed in a component. As part of changing the font size, the component's size might also change, which requires Flex to update the layout of the application. The layout operation might require Flex to invoke the `commitProperties()`, `measure()`, `getCurrentSkinState()`, and the `updateDisplayList()` methods of your component.

Your application can programmatically change the font size of a component much faster than Flex can update the layout of an application. Therefore, you only want to update the layout after you are sure that you've determined the final value of the font size.

In another scenario, when you set multiple properties of a component, such as the `label` and `icon` properties of a Button control, you want the `commitProperties()`, `measure()`, `getCurrentSkinState()`, and `updateDisplayList()` methods to execute only once, after all properties are set. You do not want these methods to execute when you set the `label` property, and then execute again when you set the `icon` property.

Also, several components might change their font size at the same time. Rather than updating the application layout after each component changes its font size, you want Flex to coordinate the layout operation to eliminate any redundant processing.

Flex uses an invalidation mechanism to synchronize modifications to components. Flex implements the invalidation mechanism as a set of methods that you call to signal that something about the component has changed and requires Flex to call the component's `commitProperties()`, `measure()`, `getCurrentSkinState()`, or `updateDisplayList()` methods.

The following table describes the invalidation methods:

Invalidation method	Description
`invalidateDisplayList()`	Marks a component so that its `updateDisplayList()` method gets called during the next screen update.
`invalidateProperties()`	Marks a component so that its `commitProperties()` method gets called during the next screen update.
`invalidateSize()`	Marks a component so that its `measure()` method gets called during the next screen update.
`invalidateSkinState()`	Marks a component so that its `commitProperties()` method gets called on the next screen update to change the view state of the skin class. The `commitProperties()` method calls the `getCurrentSkinState()` method.

When a component calls an invalidation method, it signals to Flex that the component must be updated. When multiple components call invalidation methods, Flex coordinates updates so that they all occur together during the next screen update.

Typically, component users do not call the invalidation methods directly. Instead, they are called by the component's setter methods, or by any other methods of a component class as necessary. For more information and examples, see Implementing the commitProperties() method.

About the component instantiation life cycle

The component instantiation life cycle describes the sequence of steps that occur when you create a component object from a component class. As part of that life cycle, Flex automatically calls component methods, dispatches events, and makes the component visible.

The following example creates a Button control in ActionScript and adds it to a Group container:

```
// Create a Group container.
var groupContainer:Group = new Group();
// Configure the Group container.

// Create a Button control.
var b:Button = new Button()
// Configure the button control.
b.label = "Submit";
...
// Add the Button control to the Box container.
groupContainer.addElement(b);
```

The following steps show what occurs when you execute the code to create the Button control, and add the control to the container:

1 You call the component's constructor, as the following code shows:

```
// Create a Button control.
var b:Button = new Button()
```

2 You configure the component by setting its properties, as the following code shows:

```
// Configure the button control.
b.label = "Submit";
```

Component setter methods might call the `invalidateProperties()`, `invalidateSize()`, `invalidateSkinState()`, or `invalidateDisplayList()` methods.

3 You call the `addElement()` method to add the component to its parent, as the following code shows:

```
// Add the Button control to the Box container.
gropContainer.addElement(b);
```

Flex then performs the following actions:

4 Sets the `parent` property for the component to reference its parent container.

5 Computes the style settings for the component.

6 Dispatches the `preinitialize` event on the component.

7 Calls the component's `createChildren()` method.

8 Calls the `invalidateProperties()`, `invalidateSize()`, `invalidateSkinSate()`, and `invalidateDisplayList()` methods to trigger calls to the `commitProperties()`, `measure()`, `getCurrentSkinState()`, or `updateDisplayList()` methods during the next `render` event.

The only exception to this rule is that Flex does not call the `measure()` method when the user sets the height and width of the component.

9 Dispatches the `initialize` event on the component. At this time, all of the component's children are initialized, but the component has not been sized or processed for layout. You can use this event to perform additional processing of the component before it is laid out.

10 Dispatches the `childAdd` event on the parent container.

11 Dispatches the `initialize` event on the parent container.

12 During the next `render` event, Flex performs the following actions:

- **a** Calls the component's `commitProperties()` method. The `commitProperties()` method calls the `partAdded()` and `getCurrentSkinState()` methods.
- **b** Calls the component's `measure()` method.
- **c** Calls the component's `updateDisplayList()` method.

13 Flex dispatches additional `render` events if the `commitProperties()`, `measure()`, or `updateDisplayList()` methods call the `invalidateProperties()`, `invalidateSize()`, `invalidateSkinSate()`, or `invalidateDisplayList()` methods.

14 After the last `render` event occurs, Flex performs the following actions:

- **a** Makes the component visible by setting the `visible` property to `true`.
- **b** Dispatches the `creationComplete` event on the component. The component is sized and processed for layout. This event is only dispatched once when the component is created.
- **c** Dispatches the `updateComplete` event on the component. Flex dispatches additional `updateComplete` events whenever the layout, position, size, or other visual characteristic of the component changes and the component is updated for display.

Most of the work for configuring a component occurs when you add the component to a container by using the `addElement()` method. That is because until you add the component to a container, Flex cannot determine its size, set inheriting style properties, or draw it on the screen.

You can also define your application in MXML, as the following example shows:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:Group>
        <s:Button label="Submit"/>
    </s:Group>
</s:Application>
```

The sequence of steps that Flex executes when creating a component in MXML are equivalent to the steps described for ActionScript.

You can remove a component from a container by using the `removeElement()` method. If there are no references to the component, it is eventually deleted from memory by the garbage collection mechanism of Adobe® Flash® Player or Adobe® AIR™.

About the steps for creating a component

When you implement a component, you override component methods, define new properties, dispatch new events, or perform any other customizations required by your application.

To implement your component, follow these general steps:

1 Create the skin class for the component. You typically create the skin class in MXML. For more information on skins, see Creating Spark Skins.

2 Create the component's ActionScript class file.

- **a** Extend one of the base classes, such as SkinnableComponent, or another component class.
- **b** Implement the constructor.

c Implement the UIComponent.createChildren() method. You rarely have to implement this method for Spark components.

d Implement the UIComponent.commitProperties() method.

e Implement the UIComponent.measure() method. You rarely have to implement this method for Spark components.

f Implement the UIComponent.updateDisplayList() method. You rarely have to implement this method for Spark components.

g Implement the SkinnableComponent.partAdded() and SkinnableComponent.partRemoved() methods.

h Implement the SkinnableComponent.getCurrentSkinState() method.

i Add properties, methods, styles, events, and metadata.

3 Deploy the component as an ActionScript file or as a SWC file.

For more information about MXML tag properties and embedding graphic and skin files, see Simple Visual Components in ActionScript.

You do not have to override all component methods to define a new component. You only override the methods required to implement the functionality of your component. If you create a subclass of an existing component, such as Button, you implement only the methods necessary for you to add any new functionality to the component.

About interfaces

Flex uses interfaces to divide the basic functionality of components into discrete elements so that they can be implemented piece by piece. For example, to make your component focusable, it must implement the IFocusable interface; to let it participate in the layout process, it must implement ILayoutClient interface.

To simplify the use of interfaces, the UIComponent class implements all of the interfaces defined in the following table, except for the IFocusManagerComponent interface. However, many subclasses of UIComponent implement the IFocusManagerComponent interface.

Therefore, if you create a subclass of the class or subclass of UIComponent, you do not have to implement these interfaces. But, if you create a component that is not a subclass of UIComponent, and you want to use that component in Flex, you might have to implement one or more of these interfaces.

Note: *Adobe recommends that all of your components extend the UIComponent class or a class that extends UIComponent.*

The following table lists the main interfaces implemented by Flex components:

Interface	Use
IAdvancedStyleClient	Indicates that the component supports the advanced style subsystem.
IAutomationObject	Indicates that a component is an object within the automation object hierarchy.
IChildList	Indicates the number of children in a container.
IConstraintClient	Indicates that the component support layout constraints.
IDeferredInstantiationUIComponent	Indicates that a component or object can effect deferred instantiation.
IFlexDisplayObject	Specifies the interface for skin elements.
IFlexModule	indicates that the component can be used with module factories

Interface	Use
IFocusManagerComponent	Indicates that a component or object is focusable, which means that the components can receive focus from the FocusManager. The UIComponent class does not implement IFocusable because some components are not intended to receive focus.
IID	indicates that the component can have an identifier.
IInvalidating	Indicates that a component or object can use the invalidation mechanism to perform delayed, rather than immediate, property commitment, measurement, and drawing or layout.
ILayoutManagerClient	Indicates that a component or object can participate in the LayoutManager's commit, measure, and update sequence.
IPropertyChangeNotifier	Indicates that a component supports a specialized form of event propagation.
IRepeaterClient	Indicates that a component or object can be used with the Repeater class.
IStateClient	Indicates that the component supports view states.
IToolTipManagerClient	Indicates that a component has a `toolTip` property, and therefore is monitored by the ToolTipManager.
IUIComponent	Defines the basic set of APIs that you must implement in order to be a child of layout containers and lists.
IValidatorListener	Indicates that a component can listen for validation events, and therefore show a validation state, such as a red border and error tooltips.
IVisualElement	Indicates that the component can be laid out and displayed in a Spark application.

Implementing the component

When you create a custom component in ActionScript, you have to override the methods of the UIComponent and SkinnableComponent classes.

Basic component structure

The following example shows the basic structure of a Flex component:

```
package myComponents
{
    public class MyComponent extends SkinnableComponent
    {
        ....
    }
}
```

You must define your ActionScript custom components within a package. The package reflects the directory location of your component within the directory structure of your application.

The class definition of your component must be prefixed by the `public` keyword. A file that contains a class definition can have one, and only one, public class definition, although it can have additional internal class definitions. Place any internal class definitions at the bottom of your source file below the closing curly brace of the package definition.

Implementing the constructor

Your ActionScript class should define a public constructor method for a class that is a subclass of the UIComponent class, or a subclass of any child of the UIComponent class. The constructor has the following characteristics:

- No return type
- Declared public
- No arguments
- Calls the `super()` method to invoke the superclass' constructor

Each class can contain only one constructor method; ActionScript does not support overloaded constructor methods. For more information, see Defining the constructor.

Use the constructor to set the initial values of class properties. For example, you can set default values for properties and styles, or initialize data structures, such as Arrays. You can also set the `skinClass` style to the name of your skin class.

Do not create child display objects in the constructor; you should use it only for setting initial properties of the component. If your component creates child components, create them in the skin class.

Implementing the commitProperties() method

You use the commitProperties() method to coordinate modifications to component properties. Most often, you use it with properties that affect how a component appears on the screen.

Flex schedules a call to the `commitProperties()` method when a call to the `invalidateProperties()` method occurs. The `commitProperties()` method executes during the next `render` event after a call to the `invalidateProperties()` method. When you use the `addElement()` method to add a component to a container, Flex automatically calls the `invalidateProperties()` method.

The typical pattern for defining component properties is to define the properties by using getter and setter methods, as the following example shows:

```
// Define a private variable for the alignText property.
private var _alignText:String = "right";

// Define a flag to indicate when the _alignText property changes.
private var bAlignTextChanged:Boolean = false;

// Define getter and setter methods for the property.
public function get alignText():String {
        return _alignText;
}

public function set alignText(t:String):void {
    _alignText = t;
    bAlignTextChanged = true;

    // Trigger the commitProperties(), measure() method.invalidateProperties();
}

// Implement the commitProperties() method.
override protected function commitProperties():void {
    super.commitProperties();

    // Check whether the flag indicates a change to the alignText property.
    if (bAlignTextChanged) {
        // Reset flag.
        bAlignTextChanged = false;

        // Handle alignment change
        ...

        // Call invalidateDisplayList() to update the disaply.
        invalidateDisplayList();
    }
}
```

As you can see in this example, the setter method modifies the property, calls the `invalidateProperties()` method, and then returns. The setter itself does not perform any calculations based on the new property value. This design lets the setter method return quickly, and leaves any processing of the new value to the `commitProperties()` method.

The `commitProperties()` method in the previous example process the changes to the property, then calls the `invalidateDisplay()` method to cause the component to update its display.

The main advantages of using the `commitProperties()` method are the following:

- To coordinate the modifications of multiple properties so that the modifications occur synchronously.

  For example, you might define multiple properties that control the text displayed by the component, such as the alignment of the text within the component. A change to either the text or the alignment property requires Flex to update the appearance of the component. However, if you modify both the text and the alignment, you want Flex to perform any calculations for sizing or positioning the component once, when the screen updates.

  Therefore, you use the `commitProperties()` method to calculate any values based on the relationship of multiple component properties. By coordinating the property changes in the `commitProperties()` method, you can reduce unnecessary processing overhead.

- To coordinate multiple modifications to the same property.

You do not necessarily want to perform a complex calculation every time a user updates a component property. For example, users modify the `icon` property of the Button control to change the image displayed in the button. Calculating the label position based on the presence or size of an icon can be a computationally expensive operation that you want to perform only when necessary.

To avoid this behavior, you use the `commitProperties()` method to perform the calculations. Flex calls the `commitProperties()` method when it updates the display. That means you perform the calculations once when Flex updates the screen, regardless of the number of times the property changed between screen updates.

The following example shows how you can handle two related properties in the `commitProperties()` method:

```
// Define a private variable for the text property.
private var _text:String = "ModalText";
private var bTextChanged:Boolean = false;

// Define the getter method.
public function get text():String {
        return _text;
}

//Define the setter method to call invalidateProperties()
// when the property changes.
public function set text(t:String):void {
    _text = t;
    bTextChanged = true;
    invalidateProperties();
}

// Define a private variable for the alignText property.
private var _alignText:String = "right";
private var bAlignTextChanged:Boolean = false;

public function get alignText():String {
        return _alignText;
}

public function set alignText(t:String):void {
    _alignText = t;
    bAlignTextChanged = true;
    invalidateProperties();
}

// Implement the commitProperties() method.
override protected function commitProperties():void {
    super.commitProperties();

    // Check whether the flags indicate a change to both properties.
    if (bTextChanged && bAlignTextChanged) {
        // Reset flags.
        bTextChanged = false;
        bAlignTextChanged = false;

        // Handle case where both properties changed.
        invalidateDisplayList();
```

```
    }

    // Check whether the flag indicates a change to the text property.
    if (bTextChanged) {
        // Reset flag.
        bTextChanged = false;

        // Handle text change.
        invalidateDisplayList();
    }

    // Check whether the flag indicates a change to the alignText property.
    if (bAlignTextChanged) {
        // Reset flag.
        bAlignTextChanged = false;

        // Handle alignment change.
        invalidateDisplayList();
    }
}
```

Implementing the updateDisplayList() method

The updateDisplayList() method sizes and positions parts of the component based on all previous property and style settings. The parent container for the component determines the size of the component itself. You rarely have to implement this method for Spark components.

A component does not appear on the screen until its `updateDisplayList()` method gets called. Flex schedules a call to the `updateDisplayList()` method when a call to the `invalidateDisplayList()` method occurs. The `updateDisplayList()` method executes during the next `render` event after a call to the `invalidateDisplayList()` method. When you use the `addElement()` method to add a component to a container, Flex automatically calls the `invalidateDisplayList()` method.

The main uses of the `updateDisplayList()` method are the following:

- To set the size and position of the elements of the component for display.

  Many components are made up of one or more child components, or have properties that control the display of information in the component.

  To size components in the `updateDisplayList()` method, you use the `setActualSize()` method, not the sizing properties, such as `width` and `height`. To position a component, use the `move()` method, not the `x` and `y` properties.

- To draw any visual elements necessary for the component.

  Components support many types of visual elements such as graphics, styles, and borders. Within the `updateDisplayList()` method, you can add these visual elements, use the Flash drawing APIs, and perform additional control over the visual display of your component.

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

```
protected function updateDisplayList(unscaledWidth:Number,
    unscaledHeight:Number):void
```

The properties have the following values:

unscaledWidth Specifies the width of the component, in pixels, in the component's coordinates, regardless of the value of the scaleX property of the component. This is the width of the component as determined by its parent container.

unscaledHeight Specifies the height of the component, in pixels, in the component's coordinates, regardless of the value of the `scaleY` property of the component. This is the height of the component as determined by its parent container.

Scaling occurs in Flash Player or AIR, after `updateDisplayList()` executes. For example, a component with an `unscaledHeight` value of 100, and with a `scaleY` property of 2.0, appears 200 pixels high in Flash Player or AIR.

Implementing the partAdded() and partRemoved() methods

Some components are composed of one or more subcomponents. For example, a NumericStepper component contains a subcomponent for an up button, a down button, and a text area.

The component class is responsible for controlling the behavior of the subcomponents. The skin class is responsible for defining the subcomponents, including the appearance of the component, its subcomponents, and any other visual aspects of the component.

Flex clearly defines the relationship between the component class and the skin class. The component class must do the following:

- Define the skin class or classes that it uses.
- Identify the skin parts that it uses with the `[SkinPart]` metadata tag. Each skin part typically corresponds to a subcomponent of the component class. For more information on using the `[SkinPart]` metadata tag, see SkinPart metadata tag.
- Identify the view states that the component supports with the `[SkinStates]` metadata tag. For more information on the `[SkinState]` metadata tag, see SkinState metadata tag.

The skin class must do the following:

- Specify the component name with the `[HostComponent]` metadata tag. For more information on the `[HostComponent]` metadata tag, see HostComponent metadata tag.
- Declare the view states, and define their appearance.
- Define the appearance of the skin parts. The skin parts must use the same name as the corresponding skin-part property in the component.

Flex calls the `partAdded()` and `partRemoved()` methods automatically when a skin part is created or destroyed. You typically override the `partAdded()` method to attach event handlers to a skin part, configure a skin part, or perform other actions when a skin part is added. You implement the `partRemoved()` method to remove the even handlers added in `partAdded()`.

In the component class, define skin parts as properties. In the following example, the component defines two required skin parts:

```
// Define the skin parts.
[SkinPart(required="true")]
public var modeButton:Button;

[SkinPart(required="true")]
public var textInput:RichEditableText;
```

The first skin part defines a Button control, and the second defines a RichEditableText control. While the skin parts are defined as properties of the component, component users do not directly modify them. The skin class defines their implementation and appearance. For more information on defining skin parts, see Skin parts.

In your implementation of the `partAdded()` method, you determine the skin part that was added, and configure it. Use the property name of the skin part to reference it. In this example, you set properties on the skin part and add event listeners to it:

```
override protected function partAdded(partName:String, instance:Object):void {
    super.partAdded(partName, instance);

    if (instance == textInput) {
        textInput.editable = false;
        textInput.text= _text;
        textInput.addEventListener("change", handleChangeEvent);
    }

    if (instance == modeButton) {
        modeButton.label = "Toggle Editing Mode";
        modeButton.addEventListener("click", handleClickEvent);
    }
}
```

In your implementation of the `partRemoved()` method, you remove the event listeners added by the `partAdded()` method:

```
override protected function partRemoved(partName:String, instance:Object):void {
    super.partRemoved(partName, instance);

    if (instance == textInput) {
        textInput.removeEventListener("change", handleChangeEvent);
    }

    if (instance == modeButton) {
        textInput.removeEventListener("click", handleClickEvent);
    }
}
```

Implementing the getCurrentSkinState() method

A component must identify the view states that its skin supports. Use the `[SkinState]` metadata tag to define the view states in the component class. This tag has the following syntax:

```
[SkinState("stateName")]
```

Specify the metadata before the class definition. For more information on the `[SkinState]` metadata tag, see SkinState metadata tag.

The following example defines two view states for the component named ModaTextStates:

```
[SkinState("normal")]
SkinState("textLeft")]
public class ModalTextStates extends SkinnableComponent
{
    ...
```

The component's skin class then define these view states, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<s:SparkSkin xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="100" minHeight="25">

    <s:states>
        <s:State name="normal" />
        <s:State name="textLeft"/>
    </s:states>
    ...
```

In your component, implement the `getCurrentSkinState()` method to set the view state of the skin class. Flex calls the `getCurrentSkinState()` method automatically from the `commitProperties()` method.

The `getCurrentSkinState()` method takes no arguments, and returns a String identifying the view state of the skin. You can use information in the class to determine the new view state. In the following example, you examine the `_textPlacement` property of the component to determine the view state of the skin:

```
override protected function getCurrentSkinState():String {
    var returnState:String = "normal";

    // Use information in the class to determine the new view state of the skin class.
    if (_textPlacement == "left") {
        returnState = "textLeft";
    }
    return returnState;
}
```

Making components accessible

A growing requirement for web content is that it should be accessible to people who have disabilities. Visually impaired people can use the visual content in Flash applications by using screen reader software, which provides an audio description of the material on the screen.

When you create a component, you can include ActionScript that enables the component and a screen reader for audio communication. When developers use your component to build an application in Flash, they use the Accessibility panel to configure each component instance.

Flash includes the following accessibility features:

- Custom focus navigation
- Custom keyboard shortcuts
- Screen-based documents and the screen authoring environment
- An Accessibility class

To enable accessibility in your component, add the following line to your component's class file:

```
mx.accessibility.ComponentName.enableAccessibility();
```

For example, the following line enables accessibility for the MyButton component:

```
mx.accessibility.MyButton.enableAccessibility();
```

For additional information about accessibility, see Creating Accessible Applications.

Adding version numbers

When releasing components, you can define a version number. This lets developers know whether they should upgrade, and helps with technical support issues. When you set a component's version number, use the static variable `version`, as the following example shows:

```
static var version:String = "1.0.0.42";
```

Note: *Flex does not use or interpret the value of the* `version` *property.*

If you create many components as part of a component package, you can include the version number in an external file. That way, you update the version number in only one place. For example, the following code imports the contents of an external file that stores the version number in one place:

```
include "../myPackage/ComponentVersion.as"
```

The contents of the ComponentVersion.as file are identical to the previous variable declaration, as the following example shows:

```
static var version:String = "1.0.0.42";
```

Best practices when designing a component

Use the following practices when you design a component:

- Keep the file size as small as possible.
- Make your component as reusable as possible by generalizing functionality.
- Use the Border class rather than graphical elements to draw borders around objects.
- Use tag-based skinning.
- Assume an initial state. Because style properties are on the object, you can set initial settings for styles and properties so your initialization code does not have to set them when the object is constructed, unless the user overrides the default state.

Example: Creating a composite Spark component

Composite components are components that contain multiple subcomponents. They might be graphical assets or a combination of graphical assets and component classes. For example, you can create a component that includes as subcomponents a button, rich text field, and a border graphic. Or, you can create a component that includes a text field and a validator.

When you create composite components, you define the subcomponents inside the component's skin class. You must plan the layout of the subcomponents that you are including, and set properties such as default values in the skin class.

Properties of the individual subcomponents are not directly accessible in MXML. For example, if you create a component that extends the SkinnableComponent class and uses a Button and a RichEditableText component as subcomponents, you cannot set the Button control's `label` property in MXML.

Instead, you can define a myLabel property on the custom component that is exposed in MXML. When the user sets the myLabel property, your custom component can set that property on the Button.

Creating the ModalText component

This example component, called ModalText and defined in the file ModalText.as, combines a Button control and a RichEditableText component. You use the Button control to enable or disable text input in the RichEditableText component.

This control has the following attributes:

- By default, you cannot edit the RichEditableText component.
- Click the Button control to toggle editing of the RichEditableText component.
- Use the `ModalText.text` property to programmatically write content to the RichEditableText component.
- Editing the text in the RichEditableText component dispatches the `change` event.
- Use the `text` property as the source for a data binding expression.

The following is an example MXML file that uses the ModalText control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- asAdvancedSpark/SparkMainModalText.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:MyComp="myComponents.*">

    <MyComp:ModalText/>

</s:Application>
```

Component users cannot directly access the properties of the Button and RichEditableText subcomponents. However, they can use descendant selectors to set the styles on the subcomponents, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- asAdvancedSpark/SparkMainModalTextStyled.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:MyComp="myComponents.*">

    <fx:Style>
        @namespace MyComp "myComponents.*";
        @namespace s "library://ns.adobe.com/flex/spark";

        MyComp|ModalText s|Button {
            baseColor: #663366;
            color: #9999CC;
        }
    </fx:Style>

    <MyComp:ModalText/>

</s:Application>
```

In this example, you use descendent selectors to set the set the `baseColor` and `color` styles of the Button subcomponent. For more information, see Using Cascading Style Sheets.

Defining event listeners for composite components

Subcomponents can dispatch events. The main component can either handle the events internally, or propagate the event so that a component user can handle them.

Custom components implement the `partAdded()` method to create children of the component, as the following example shows:

```
override protected function partAdded(partName:String, instance:Object):void {
    super.partAdded(partName, instance);
    if (instance == textInput) {
        textInput.editable = false;
        textInput.text= _text;
        textInput.addEventListener("change", handleChangeEvent);
    }
    if (instance == modeButton) {
        modeButton.label = "Toggle Editing Mode";
        modeButton.addEventListener("click", handleClickEvent);
    }
}
```

The `partAdded()` method contains a call to the `addEventListener()` method to register an event listener for the `change` event generated by the RichEditableText subcomponent, and for the `click` event for the Button subcomponent. These event listeners are defined within the ModalText class, as the following example shows:

```
// Handle events for a change to RichEditableText.text property.
private function handleChangeEvent(eventObj:Event):void {
        dispatchEvent(new Event("change"));
}

// Handle the click event for the Button subcomponent.
private function handleClickEvent(eventObj:Event):void {
        text_mc.editable = !text_mc.editable;
}
```

You can handle an event dispatched by a child of a composite component in the component. In this example, the event listener for the Button subcomponent's `click` event is defined in the class definition to toggle the `editable` property of the RichEditableText subcomponent.

However, if a child component dispatches an event, and you want that opportunity to handle the event outside of the component, you must add logic to your custom component to propagate the event. Notice that the event listener for the `change` event for the RichEditableText subcomponent propagates the event. This lets you handle the event in your application, as the following example shows:

```
<?xml version="1.0"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:MyComp="myComponents.*">

    <fx:Script>
        <![CDATA[

            import flash.events.Event;

            function handleText(eventObj:Event)
            {
                ...
            }
        ]]>
    </fx:Script>

    <MyComp:ModalText change="handleText(event);"/>
</s:Application>
```

Creating the skin class for the ModalText component

The ModalText component defines a Button subcomponent and a Rich TextArea subcomponent. Each of these subcomponents is defined in the ModalText.as file as a required skin part.

The skin class performs the following:

- Uses the `[HostComponent]` metadata tag to specify ModalText as the host component of the skin.
- Defines a single view state named normal.
- Uses the Rect class to draw a border around the RichTextArea subcomponent.
- Uses a Scroller component to add scroll bars to the RichTextArea subcomponent.

The skin class is defined in MXML, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<s:SparkSkin xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="100" minHeight="25">

    <!-- Define ModalText as the host component of the skin. -->
    <fx:Metadata>
        <![CDATA[
            [HostComponent("ModalText")]
        ]]>
    </fx:Metadata>
    <s:states>
        <s:State name="normal"/>
    </s:states>

    <!-- Define the border around the RichEditableText control. -->
    <s:Rect x="{myScroller.x}"
        width="{myScroller.width}"
        height="{myScroller.height}">
```

```
        <s:stroke>
            <s:SolidColorStroke color="0x686868" weight="1"/>
        </s:stroke>
    </s:Rect>

    <!--- Defines the appearance of the Button subcomponent. -->
    <s:Button id="modeButton"
        x="0"
        minHeight="25"/>

    <!--- Defines the appearance of the RichEditableText subcomponent. -->
    <s:Scroller id="myScroller"
        x="{modeButton.width + 6}">
        <s:RichEditableText id="textInput"
            minHeight="25"
            heightInLines="4"
            paddingLeft="4" paddingTop="4"
            paddingRight="4" paddingBottom="4"/>
    </s:Scroller>
</s:SparkSkin>
```

Creating the ModalText component

The following code implements the class definition for the ModalText component. The ModalText component is a composite component that contains a Button and a RichEditableText subcomponent:

```
package myComponents
{
    import flash.events.Event;
    import spark.components.Button;
    import spark.primitives.RichEditableText;
    import spark.components.supportClasses.SkinnableComponent;

    // ModalText dispatches a change event when the text of the
    // RichEditableText subcomponent changes.
    [Event(name="change", type="flash.events.Event")]

    /** a) Extend SkinnableComponent. */
    public class ModalText extends SkinnableComponent
    {
        /** b) Implement the constructor. */
        public function ModalText() {
            super();

            // Set the skin class.
            setStyle("skinClass", ModalTextSkin);
        }

        /** c) Define the skin parts for the Button
        *     and RichEditableText subcomponents. **/
        [SkinPart(required="true")]
        public var modeButton:Button;

        [SkinPart(required="true")]
        public var textInput:RichEditableText;

        /** d) Implement the commitProperties() method to handle the
```

```
*     change to the ModalText.text property.
*     Changes to the ModalText.text property are copied to
*     the  RichEditableText subcomponent. */
override protected function commitProperties():void {
    super.commitProperties();

    if (bTextChanged) {
        bTextChanged = false;
        textInput.text = _text;
        invalidateDisplayList();
    }
}

/** e) Implement the partAdded() method to
*     initialize the Button and RichEditableText subcomponents. */
override protected function partAdded(partName:String, instance:Object):void {
    super.partAdded(partName, instance);

    if (instance == textInput) {
        textInput.editable = false;
        textInput.text= _text;
        textInput.addEventListener("change", handleChangeEvent);
    }

    if (instance == modeButton) {
        modeButton.label = "Toggle Editing Mode";
        modeButton.addEventListener("click", handleClickEvent);
    }
}

/** f) Implement the partRemoved() method to remove the
*     event listeners added by partAdded(). */
override protected function partRemoved(partName:String, instance:Object):void {
    super.partRemoved(partName, instance);

    if (instance == textInput) {
        textInput.removeEventListener("change", handleChangeEvent);
    }

    if (instance == modeButton) {
        textInput.removeEventListener("click", handleClickEvent);
    }
}

/** g) Add methods, properties, and metadata.
*     The general pattern for properties is to specify a
*     private holder variable. */

// Implement the ModalText.text property.
private var _text:String = "ModalText";
private var bTextChanged:Boolean = false;

// Create a getter/setter pair for the text property.
[Bindable]
public function set text(t:String):void {
    _text = t;
    bTextChanged = true;
```

```
            invalidateProperties();
        }

        public function get text():String {
                return textInput.text;
        }

        // Dispatch a change event when the RichEditableText.text
        // property changes.
        private function handleChangeEvent(eventObj:Event):void {
                dispatchEvent(new Event("change"));
        }

        // Handle the Button click event to toggle the
        // editting mode of the RichEditableText subcomponent.
        private function handleClickEvent(eventObj:Event):void {
                textInput.editable = !textInput.editable;
        }
    }
}
```

Creating the ModalTextStates component

The following code example implements the class definition for the ModalTextStates component. The ModalTextStates component modifies the ModalText components shown in the previous section to add view states to the skin class. This control has all of the attributes of the ModalText class, and adds the following attributes:

- Uses the `textPlacement` property of the component to make the RichEditableText subcomponent appear on the right side or the left side of the component.
- Editing the `textPlacement` property of the control dispatches the `placementChanged` event.
- Uses the `textPlacement` property as the source for a data binding expression.
- Setting the `textPlacement` property so that the RichTextArea component appears on the right configures the skin class to use normal view state. Setting it to appear on the left uses the textLeft view state.
- Disabling editing of the RichTextArea component sets the view state of the skin class to normalDisabled or textLeftDisabled.

The following example uses the ModalTextStates component in an application:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- asAdvancedSpark/SparkMainModalTextStates.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:MyComp="myComponents.*">
     <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
      <![CDATA[
        import flash.events.Event;

        private function placementChangedListener(event:Event):void {
          myEvent.text="placementChanged event occurred - textPlacement = "
              + myMT.textPlacement as String;
        }
      ]]>
    </fx:Script>

    <MyComp:ModalTextStates id="myMT"
        textPlacement="left"
        placementChanged="placementChangedListener(event);"/>
    <mx:TextArea id="myEvent" width="50%"/>
    <mx:Label text="Change Placement" />
    <mx:Button label="Set Text Placement Right"
        click="myMT.textPlacement='right';" />
    <mx:Button label="Set Text Placement Left"
        click="myMT.textPlacement='left';" />
</s:Application>
```

This applications sets the initial placement of the RichEditableText subcomponent to left. Use the buttons to switch the placement between right and left. When the placement changes, the event handler for the placementChanged event displays the current placement.

Creating the skin class for the ModalTextStates component

The skin class for the ModalTextStates component, ModalTextStatesSkin.mxml, adds three view states to control the display of the component:

- normal The RichEditableText subcomponent is on the right, and editing is enabled.
- disabled The RichEditableText subcomponent is on the right, and editing is disabled.
- textLeft The RichEditableText subcomponent is on the left, and editing is enabled.
- textLeftDisabled The RichEditableText subcomponent is on the left, and editing is disabled.

Shown below is the skin definition:

```
<?xml version="1.0" encoding="utf-8"?>
<s:SparkSkin xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="100" minHeight="25">

    <fx:Metadata>
        <![CDATA[
            [HostComponent("ModalTextStates")]
        ]]>
    </fx:Metadata>
    <s:states>
        <s:State name="normal" />
        <s:State name="normalDisabled" stateGroups="disabledGroup"/>
        <s:State name="textLeft"/>
        <s:State name="textLeftDisabled" stateGroups="disabledGroup"/>
    </s:states>

    <!-- Define the border around the RichEditableText control. -->
    <s:Rect x="{myScroller.x}"
        width="{myScroller.width}"
        height="{myScroller.height}">
        <s:stroke>
            <s:SolidColorStroke color="0x686868" weight="1"/>
        </s:stroke>
    </s:Rect>

    <!--- Defines the appearance of the Button subcomponent. -->
    <s:Button id="modeButton"
        x="0" x.textLeft="{myScroller.width +6}" x.textLeftDisabled="{myScroller.width +6}"
        minHeight="25" height="100%"/>

    <!--- Defines the appearance of the RichEditableText subcomponent. -->
    <s:Scroller id="myScroller"
        x="{modeButton.width + 6}" x.textLeft="0" x.textLeftDisabled="0">
        <s:RichEditableText id="textInput"
            alpha="1.0" alpha.disabledGroup="0.5"
            minHeight="25"
            heightInLines="4"
            paddingLeft="4" paddingTop="4"
            paddingRight="4" paddingBottom="4"/>
    </s:Scroller>
</s:SparkSkin>
```

Creating the ModalTextStates component

Shown below is the code for the ModalTextStates component. The ModalTextStates class modifies the ModalText class in the following ways:

- Adds the implementation of the `ModalTextStates()` method to set the view state of the skin.
- Adds the implementation of the `textPlacement` property to set the placement of the RichEditableText subcomponent.

```
package myComponents
{
    import flash.events.Event;
    import spark.components.Button;
    import spark.primitives.RichEditableText;
    import spark.components.supportClasses.SkinnableComponent;

    // ModalText dispatches a change event when the text of the
    // RichEditableText subcomponent changes,
    // and a placementChanged event
    // when you change the textPlacement property of ModalText.
    [Event(name="change", type="flash.events.Event")]
    [Event(name="placementChanged", type="flash.events.Event")]

    // Define the skin states implemented by the skin class.
    [SkinState("normal")]
    [SkinState("normalDisabled")]
    [SkinState("textLeft")]
    [SkinState("textLeftDisabled")]

    /*** a) Extend SkinnableComponent. ***/
    public class ModalTextStates extends SkinnableComponent
    {
        /*** b) Implement the constructor. ***/
        public function ModalTextStates() {
            super();

            // Set the skin class.
            setStyle("skinClass", ModalTextStatesSkin);
        }

        /*** C) Define the skin parts. ***/
        [SkinPart(required="true")]
        public var modeButton:Button;

        [SkinPart(required="true")]
        public var textInput:RichEditableText;

        /*** d) Implement the commitProperties() method. ***/
        override protected function commitProperties():void {
            super.commitProperties();

            if (bTextChanged) {
                bTextChanged = false;
                textInput.text = _text;
                invalidateDisplayList();
            }
        }

        /*** e) Implement the partAdded() method. ***/
        override protected function partAdded(partName:String, instance:Object):void {
            super.partAdded(partName, instance);

            if (instance == textInput) {
                textInput.editable = false;
                textInput.text= _text;
                textInput.addEventListener("change", handleChangeEvent);
```

```
    }

    if (instance == modeButton) {
        modeButton.label = "Toggle Editing Mode";
        modeButton.addEventListener("click", handleClickEvent);
    }
}

/*** f) Implement the partRemoved() method. ***/
override protected function partRemoved(partName:String, instance:Object):void {
    super.partRemoved(partName, instance);

    if (instance == textInput) {
        textInput.removeEventListener("change", handleChangeEvent);
    }

    if (instance == modeButton) {
        textInput.removeEventListener("click", handleClickEvent);
    }
}

/*** e) Implement the getCurrentSkinState() method. ***/
override protected function getCurrentSkinState():String {
    var returnState:String = "normal";

    if (textInput.editable == true)
    {
        if (_textPlacement == "right") {
            returnState = "normal";
        }
        else if (_textPlacement == "left") {
            returnState = "textLeft";
        }
    }

    if (textInput.editable == false)
    {
        if (_textPlacement == "right") {
            returnState = "normalDisabled";
        }
        else if (_textPlacement == "left") {
            returnState = "textLeftDisabled";
        }
    }
    return returnState
}

/*** h) Add methods, properties, and metadata. ***/
// The general pattern for properties is to specify a private
// holder variable.

// Define the text property.
private var _text:String = "ModalText";
private var bTextChanged:Boolean = false;

// Then, create a getter/setter pair for the text property.
[Bindable]
```

```
        public function set text(t:String):void {
            _text = t;
            bTextChanged = true;
            invalidateProperties();
        }

        public function get text():String {
            return textInput.text;
        }
        // Define the textPlacement property.
        private var _textPlacement:String = "right";

        // Create a getter/setter pair for the textPlacement property.
        [Bindable]
        public function set textPlacement(p:String):void {
            _textPlacement = p;
            invalidateSkinState();
            dispatchEvent(new Event("placementChanged"));
        }

        public function get textPlacement():String {
            return _textPlacement;
        }

        // Dispatch a change event when the RichEditableText.text
        // property changes.
        private function handleChangeEvent(eventObj:Event):void {
            dispatchEvent(new Event("change"));
        }

        // Handle the Button click event to toggle the
        // editting mode of the RichEditableText subcomponent.
        private function handleClickEvent(eventObj:Event):void {
            textInput.editable = !textInput.editable;
            invalidateSkinState();
        }
    }
}
```