Flex in Action  
Capítulo 17

# Introducción

Flex es un framework que se utiliza para crear aplicaciones RIA (Rich Internet Application). Las aplicaciones RIAs son aplicaciones completas que corren en un navegador de internet.

Anteriormente la interacción que se tenia con las paginas web era que nosotros hacíamos click en los botones y la pagina cambiaba de forma. A este tipo de llamadas al servidor web se le llaman peticiones Sincrónicas ya que nosotros elegimos cuando enviar información al servidor y luego nos quedamos esperando a que el servidor procese nuestro pedido para luego cambiarnos la página.   
El cambio, las aplicaciones RIA basan su potencial en las peticiones asincrónicas, estas nos permiten continuar con nuestra interacción con la pagina mientras que el servidor procesa la información. Cuando el servidor termina de procesar nuestra información nos avisa que los datos ya se han recibidos y nosotros manejamos estos datos como mejor nos parezca.   
Para poder realizar estas tareas, Flex nos brinda componentes y características tales como Servicios Web, arrastrar y soltar, columnas ordenables, graficas, efectos de animaciones, eventos, estados, etc.

Hoy en día existen muchas herramientas para el desarrollo de aplicaciones RIA, Flex es una de ellas pero también existe AJAX con JavaScript, JavaFX, Microsoft Prestation Foundation, Open Lazlo, etc.

# Lenguaje de programación en Flex

Las aplicaciones Flex se escriben utilizando tres tipos de lenguaje de programación:

* **ActionScript3:** Es un lenguaje orientado a objetos basado en la especificación ECMAS Script 4. Este leguaje incluye las mayoría de los elementos de los lenguajes orientados a objetos como puede ser : definición de clases, estructura de paquetes, herencia de clases, declaración fuerte de variables, etc.
* **MXML:** Es un lenguaje de etiqueta s basado en XML. Se utiliza para definir la aplicación Flex y algunos de sus componentes, la mayoría de los elementos MXML se corresponden con clases AS3 que se pueden encontrar en la biblioteca de clases de Flex.
* **FXG:** Es un nuevo lenguaje basado en XML que permite representar objetos gráficos como etiquetas XML. MXML incluye etiquetas de dibujo de gráficos vectoriales que permiten declarar objetos de bajo nivel en las aplicaciones, FXG nos da más poder, y nos permite utilizar otras etiquetas para gráficos más avanzados y nos permite tratar los ficheros FXG como imágenes.

Dentro de código de la aplicación podemos utilizar MXML y ActionScript de manera intercambiable pero por lo general se utiliza MXML para declarar componentes visuales.  
Es importante destacar que todo lo que se puede hacer con ActionScript, también se puede hacer con MXML. Es el desarrollador el que decide cual utilizar y en el momento, pero MXML tiene una ventaja sobre AS que es que hace más fácil y rápido escribir las aplicaciones.

Presentamos un ejemplo para ilustrar la diferencia de cantidad de línea que se deben utilizar para realizar el mismo input.

|  |  |
| --- | --- |
| **ActionScript** | import mx.events.FlexEvent;  import spark.components.Label;  protected var etiqueta1:Label;  protected function  creationCompleteHandler(event:FlexEvent):Void{  etiqueta1 = new Label();  etiqueta1.text = "Hola ActionScript";  etiqueta1.setStyle("fontSize", 18);  etiqueta1.setStyle("fontWeight", "bold");  this.contentGroup.addElement(etiqueta1);  } |
| **MXML** | <s:Label id="myMXMLText" text="Hola" fontSize="18" fontWeight="bold"/> |

# Flex Orientado a Objetos

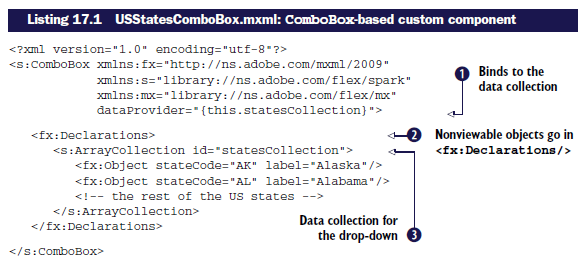
Flex utiliza las técnicas de: Modularidad, encapsulación, herencia, polimorfismo.

* Modularidad: Una aplicación se construye en módulos, así se genera una aplicación robusta en donde los cambios en un módulo no afecta al comportamiento de los otros módulos.  
  Un ejemplo de esto podría ser una aplicación que toma datos de usuarios. Un modulo seria el que presente el formulario y otro modulo en el que se encargue de procesar los datos.
* Encapsulación: Un objeto puede ocultarse de la implementación del resto de la aplicación y podría su funcionalidad. La documentación de los mientras del objeto se conoce como application programming interface (API) de la clase.  
  Biblioteca de Flex:  
    
  - Constants: Propiedades cuyos valores nunca cambian.  
  -Events: Mensajes que el objeto puede enviar al resto de la aplicación para compartir información sobre las acciones del usuario y los datos que quiere compartir.  
  -Methods: Funciones que se pueden llamar para ejecutar ciertas acciones del objeto.  
  -Properties: Datos almacenados dentro del objeto.  
  -Skin Parts: Una parte de componentes Spark que muestran una parte del componente y pueden ser modificados en un skin personalizado.  
  -Skin State: Estado de vista que un componente lanza para display, ocultarse o cambiar en su presentación visual.  
  -Style: Caracteristicas visuales de un objeto que determinan su aparencia.
* Herencia: Capacidad de algunas clase para extender a otras y heredar sus propiedades y métodos.

# Componentes visuales en Flex

Flex incluye un modelo basado en el desarrollo de componentes que podemos usar para desarrollar la aplicación y la interfaz de usuario. Podemos utilizar las componentes visuales incluidos en flex, a su vez podemos extender los componentes para añadir nuevas propiedades y métodos. Además si fuera necesario podemos crear nuestros propios componentes, esto nos brinda una gran ventaja que es la reutilización de código, ya que las nuevas componentes o las componentes personalizadas las podremos utilizar en diferentes aplicaciones.  
Los componentes visuales son muy flexibles y nos ofrecen un gran control sobre su apariencia. Fácilmente le asignamos respuestas a interacciones del usuario, le personalizamos las características visuales como puede ser:

* Tamaño: tamaño que va a tener la componente que utilicemos.
* Estilo: Característica como fuente, tamaño, alineación del texto ect. Son las mismas características que definen con CSS.
* Comportamiento: Cambiar estado visible o audible del componente por medio de la aplicación o por la acción del usuario. (Ejemplo, mover o redimensionar el componente en función de un click del ratón o si el usuario se posiciona sobre el componente)
* Skins.



(Ejemplo de utilización de Combo-Box para la interfaz de usuario.)

<mx:Button id="button1"

label="Haceme Click!"

width="100"

fontSize="12"

click="bottonClick();"/>

(Ejemplo de la personalización que se le se puede hacer a la componente y interacciones con el usuario)

DeveloperFlexComponents  
Capítulo 2-(Components)

**Utilizando Componentes Flex**

Con la llegada de Flex 4 el marco de componentes ahora tiene dos variantes.

1. Flex 2/3 componentes origíneles (Halo)
2. Flex 4 con el conjunto de componentes Spark

Para diferenciar entre Halo y Spark necesitamos utilizar los espacios de nombres como muestra el siguiente ejemplo:

<?xml version="1.0" encoding="utf-8"?>

<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/mx">

<mx:Button id="haloBtn" label="I’m a Halo Button" />

<s:Button id="sparkBtn" label="I’m a Spark Button" />

</s:Application>

Los componentes Halo se declaran utilizando el espacio de nombre MX, mientras que los componentes Spark nuevos usan el espacio de nombre s.

Además, tenga en cuenta que el espacio de nombres por defecto para los elementos que no son componentes específicos, como los bloques de script y etiquetas de estilo, en el marco Flex ahora utiliza un prefijo de espacio de nombres de FX

**Componente árbol genealógico**

Todos los componentes de Flex 4 se dividen en 5 grupos distintos.

Estos cinco grupos son:

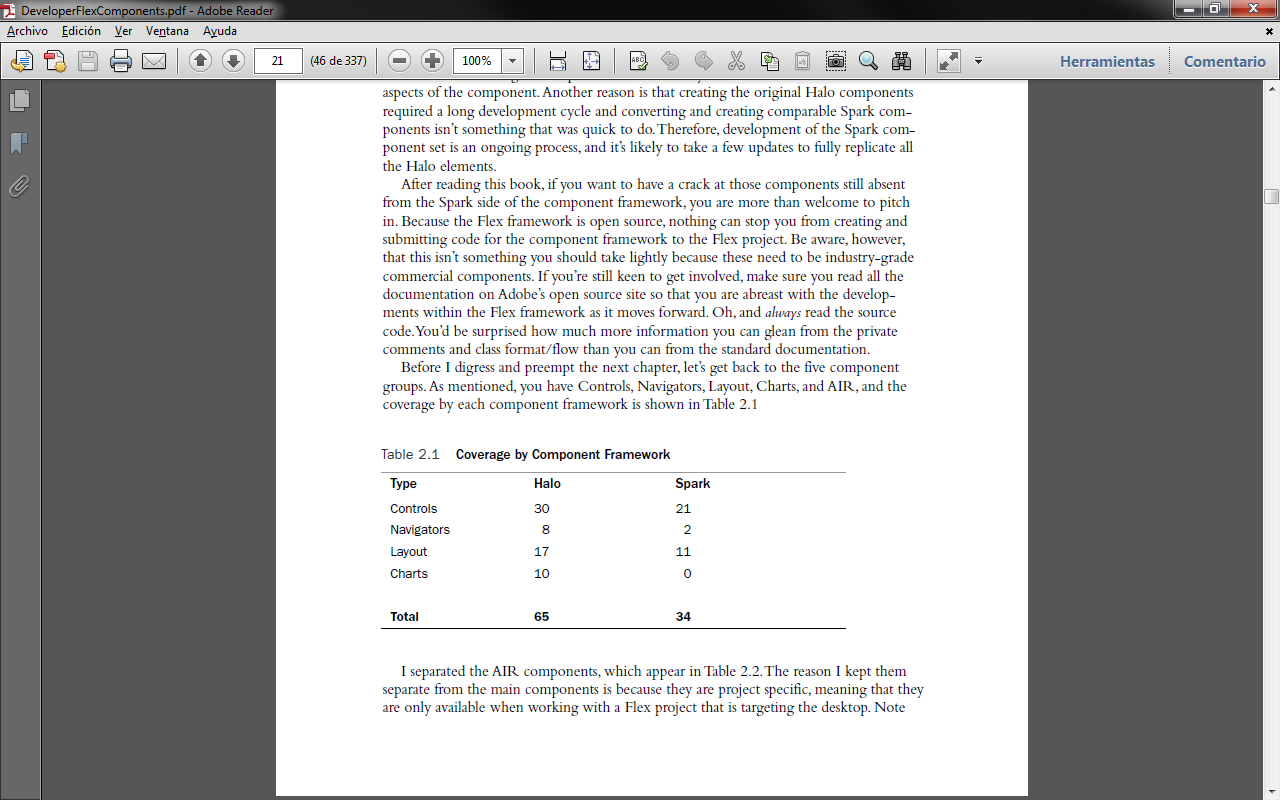
1. Controles
2. Navegadores
3. Diseño
4. Gráficos
5. AIR
6. Componentes Personalizados

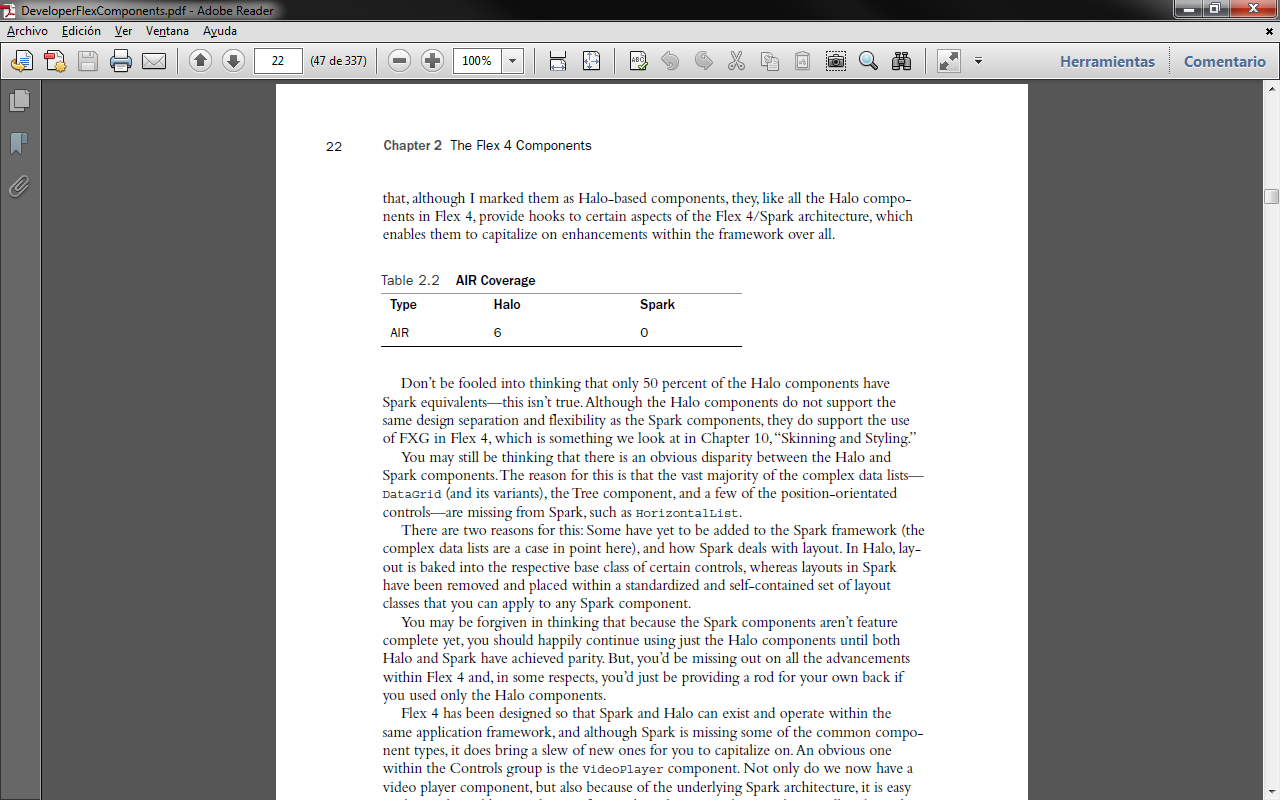
Ellos se componen tanto de Halo como de componentes Spark, con la excepción de Gráficos y los componentes AIR, que todavía se basan principalmente en Halo.

En total, hay 105 componentes, sin embargo, la gran mayoría se superponen, por lo que hay tanto de Halo como de Spark equivalentes del mismo componente.

Ahora, usted puede preguntarse por qué hay duplicados en el marco. Obviamente, una de las razones es que no todos los proyectos de Flex pueden aprovechar el marco completo Flex 4 y Spark está disponible sólo en aplicaciones destinadas a Flash Player 10, para que las personas que mantienen las aplicaciones más antiguas pueden seguir utilizando los componentes de Halo y utilizar Flash Player 9 y el marco Flex 3. Sin embargo, a medida que actualice sus aplicaciones, puede reemplazar poco a poco la mayoría de los componentes de Halo con versiones Spark.

Antes de hacer una digresión y adelantarse al próximo capítulo, vamos a volver a los cinco grupos de componentes. Como se ha mencionado, hay controles, Navegadores, Diseño, Gráficos y AIR, y la cobertura por cada marco de componentes se muestra en la Tabla 2.1





Flex 4 ha sido diseñado para que Spark y Halo puedan existir y operar dentro del mismo marco de aplicación, y aunque le falta un poco a Spark de los tipos de componentes comunes, este trae un montón de otras nuevas para que usted pueda capitalizar.  
Una obvia dentro del grupo de controles es el componente VideoPlayer. No sólo ahora tenemos un componente de reproductor de vídeo, sino también por la arquitectura de Spark subyacente, que facilita la piel, estilo, y diseñar la interfaz sin tener que buscar en el código general. También incorpora el nuevo Open Screen Media Framework (OSMF), que le permite "fácilmente proporcionar componentes utilizados para experiencias de reproducción de alta calidad.

Otra adición a los componentes Spark es el RichText y componentes RichEditableText. Estos incorporan el nuevo Text Layout Framework (TLF), que aprovecha la representación de texto mejorado en Flash Player 10 y, a su vez, permite que estos componentes proporcionen soporte para varios idiomas, texto bidireccional que mantiene la fidelidad de impresión y ofrece la entrada estándar y controles de gestos para la copia, pegar, cortar, etc. Esto no está presente en halo.

Algunos componentes no necesariamente tienen que ser convertidos a la estructura Spark; estos componentes seguirán funcionando felizmente con las versiones de Halo. Sospecho que si liberan las versiones futuras de Flex, lo más probable es que Halo quedará obsoleto, y todo va a estar disponible en el Spark. Sin embargo, hasta ese día, no dude en mezclar y combinar entre Spark y Halo como mejor le parezca.

**Control Components**

Aquí, tenemos las bases para casi todo lo que necesita para la captura y el suministro de interfaces de usuario.

Como se puede ver en la Figura 2.1, hay un montón de ellos, pero lo que todos tienen en común es que son componentes visuales.



Figure 2.1 Control components

**Loaders and Displays**

Los Loaders se basan en el contenido que cargan sobre sí mismos para ofrecer su aspecto visual.

Algunos componentes de control tienen que cargar su contenido antes de que puedas verlos en la aplicación. Por ejemplo, los componentes SWFLoader e Imagen son ambos transparentes a menos que tengan contenido para cargar y mostrar. El componente VideoDisplay (que no debe confundirse con el componente VideoPlayer) es idéntico y no muestra un fondo si no tiene un archivo de vídeo cargado en la misma. La Figura 2.2 muestra la imagen, SWFLoader y componentes VideoDisplay con contenido cargado en ellos.



Figura 2.2 De izquierda a derecha: Una imagen, Una SWFLoader, y un componente VideoDisplay mostrar los datos que han cargado

**Los consumidores de datos (Data Consumers)**

Los controles que responden a esta categoría son los List, Grid y Tree

La lista completa sería la siguiente:

1. List
2. HorizontalList
3. DataGrid
4. AdvancedDataGrid
5. OLAPDataGrid
6. TitleList
7. Tree
8. Combobox
9. DropDownList

Todos ellos pueden consumir y visualizar datos complejos a través de diferentes presentaciones visuales (véase la Figura 2.3)



Figura 2.3 De izquierda a derecha: Los mismos datos mostrados por List, DataGrid, Tree

De todos los componentes mencionados en este apartado, los dos que sobresalen son ComboBox y DropDownList porque visualmente son más similares a los distintos componentes de botón.

Los incluyo aquí porque apoyan datos complejos y técnicamente controles basados ​​en listas, aunque no heredan directamente de cualquiera de las clases ListBase.

También puede preguntarse cuál es la diferencia entre el ComboBox y DropDownList. Bueno, tiene que ver con el área del sistema actual de la componente. En un ComboBox, este es un componente TextInput, por lo que un usuario puede editar el texto que se muestra en su interior, mientras que el DropDownList utiliza un componente Label como su área de mensajes, lo que ofrece una pantalla no editable.

Estos componentes pueden aceptar una amplia selección de formatos de datos para utilizar como un proveedor de datos. Sin embargo, los más utilizados son las **arrays y collections**. Si utiliza una Arrays, sólo tiene que utilizar los valores individuales, los objetos anónimos, o instancias de objetos con tipo. Lo mismo se puede aplicar a las colecciones. En lo que difieren es la forma de acceder a interactuar con los datos, arrays, como usted probablemente sabe, utilizan un índice basado en cero, mientras que las colecciones frente a los elementos. Por lo tanto, el acceso a los mismos datos de una matriz y una colección se ve algo así:

// access data from an Array stored in index 2

private var myObj :Object = myArray[2];

// access the same data when it is stored in a collection

private var myOtherObj :Object = myCollection.getItemAt(2);

Colecciones vienen es diferentes formas, pero las más comunes son ArrayCollections y XMLCollections.

Por defecto, las listas, grid y los árboles aceptan valores que se pueden utilizar como "nombre, pares de valores". Estos pares se construyen de una etiqueta y una propiedad de datos. Si se omite uno de ellos, van a utilizar la otra propiedad de ambos. Esto no lo limita para pasar objetos de datos simples a estos componentes. Pueden aceptar datos complejos y mostrarlo para usted. Si tiene varios campos en un objeto de datos que desea mostrar, se puede emplear itemRenderer.

Lo que hace Renderers verdaderamente especial es su capacidad para permitir la inclusión de componentes adicionales para mejorar la prestación de estos datos. La figura 2.4 muestra dos componentes Tree de lado a lado. Ambos están consumiendo los mismos datos. Sin embargo, el de la derecha se ha mejorado mediante el uso de componentes adicionales (en este caso, un componente de imagen) para mostrar el contenido como imágenes en miniatura. Como puede ver, el de la izquierda sólo muestra un simple icono mientras que el que se ha mejorado muestra la imagen real.



Figura 2.4 Árbol de componentes estándar y una utilizando un itemRenderer para mostrar las miniaturas de las imágenes que representa

**Controles Button**

En que puede venir como una sorpresa al descubrir que la gran mayoría de los componentes de control están basados ​​en el botón, empezando con el componente Button, entonces tenemos la ToggleButton, CheckBox, LinkButton PopUpButton, PopUpMenuButton y RadioButton (ver Figura 2.5).

No termina allí, sin embargo, una gran cantidad de componentes adicionales que incorporan botones para proporcionar la funcionalidad que requiera algún tipo de información visual, esto es incluyendo el conjunto de los componentes basados ​​en listas que acaba de ver y ComboBox y DropDownList.



Figura 2.5

Ahora, mientras que Button, ToggleButton, Checkbox, y LinkButton se pueden dejar en una aplicación y se utiliza como es, PopupButton, PopUpMenuButton y RadioButton, requieren datos adicionales (o en el caso del control RadioButton, un componente RadioGroup) para funcionar como se espera. Esto es debido a que proporcionan un mayor nivel de funcionalidad en comparación con el simple "clic" y "selección" de los tres primeros componentes de botón.

Los componentes de pop-up son similares a ComboBox y DropDownList en funcionalidad, pero ambos PopUpButton y PopUpMenuButton pueden tener su área de visualización de la página principal clic para enviar un evento. PopUpButton permite definir lo que se muestra cuando se hace clic, mientras que PopUpMenuButton incluye un menú como parte de su funcionalidad, pero también se puede hacer clic.

**Steppers, Sliders, Pickers, Choosers**

Todo lo que queda son los componentes especializados que, aunque único, están siendo muy útiles: NumberStepper, Spinner, Sliders, ColorPicker, Date, y barra de desplazamiento (véase la Figura 2.6). Algunos de estos son los elementos específicos que proporcionan un nivel adicional de funcionalidad cuando se requiere una tarea o un proceso único, pero valioso.

Los componentes Date, DateField y DateChooser, ofrecen soluciones fáciles, cuando es necesario establecer criterios basados ​​en el tiempo, como las vacaciones o una fecha de nacimiento. Los Sliders, tanto el VSlider y HSlider, proporcionan un medio por el cual el usuario puede filtrar o ajustar los datos que tiene un valor máximo y el mínimo, como el volumen de un reproductor de vídeo. NumericStepper ofrece el mismo tipo de filtrado, pero a través de valores numéricos puros, que permite a un usuario ajustar con precisión un valor pero aún así permanecer dentro de los límites definidos por la aplicación.



Figure 2.6-De izquierda a derecha.

Spinner, NumericStepper, DateChooser, DateField, VSlider, HSlider, VScrollBar, HScrollBar, and ColorPicker

**Spinner,** aunque un componente independiente en su propio derecho, es los botones arriba / abajo **NumericStepper**, le permite incorporar ya sea dentro de los componentes que requieren la capacidad de subir y bajar dentro de los datos proporcionados.

**ColorPicker,**  proporciona un mecanismo rápido y fácil para seleccionar un valor de color.

**Scrollbar,** por último, pero no menos importante, usted tiene los componentes de la barra de desplazamiento, tanto vertical como horizontal. Aunque estos no son nuevos para Spark, barras de desplazamiento desempeñan un papel más importante en el control de los componentes y el diseño a través del componente Scroller.

**Text Controls**

La capacidad para visualizar y habilitar la inserción de texto (si es texto plano o HTML) en sus aplicaciones es uno bastante importante. Estos son los componentes de texto, comenzando con el componente Label y nuestra forma de con la gran funcionalidad que es la RichTextEditor. En el medio, tenemos el Text, TextInput, TextArea, RichText y componentes RichEditableText.

**Label** es una sola línea de texto no editable componente que es útil para la visualización de partidas simples y entradas.

El componente **Text** es casi idéntico, pero tiene la ventaja adicional de mostrar el contenido de varias líneas. Sin embargo, como con el componente Label, este texto no es editable.

Los componentes **TextInput** y **TextArea** son básicamente equivalentes pueden editar las Label y Text; Hay una diferencia clave entre los componentes de Text y TextArea, sin embargo.

Al componente TextArea se le ha incorporado las barras de desplazamiento, por lo que si el contenido excede sus límites, se muestran automáticamente.

Si el componente necesita una barra de desplazamiento vertical, su inclusión puede obligar a que el componente también se muestre la barra de desplazamiento horizontal.

Usted puede evitar esto si usted no quiere barras de desplazamiento mediante el establecimiento de horizontalScrollPolicy o verticalScrollPolicy en off, como en la Figura 2.7 y el siguiente código para crear demuestra



Figure 2.7 Los componentes de texto que muestran el mismo contenido, la imagen de la derecha que tiene dos barras de desplazamiento desactivadas

Hasta este punto, los componentes de texto que hemos visto son todos disponibles dentro de Halo y Spark, pero dos nuevos componentes sólo están disponibles en el marco de Spark: RichText y RichEditableText.

Diseñado como un campo de texto de usos múltiples, que permiten la visualización de texto sin formato y ricos (HTML), como su nombre indica. Estos ricos componentes de texto utilizan el nuevo Text Layout Framework (TLF), que le da la capacidad tener al texto múltiple estilos y orientaciones (puede mostrar todas las formas de texto bidireccional en el mismo componente a través de TLF), el control de las posiciones de tabulación, el formato y el flujo. El beneficio de esto es que se requiere un enfoque un poco más detallado de los datos de asignación a uno de estos componentes.

El siguiente código muestra un sencillo componente RichText con formatos de texto. Los ricos y los componentes de texto TLF le dan más o menos píxel perfecto control sobre el diseño y el formato.

<?xml version="1.0" encoding="utf-8"?>

<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/mx"

minWidth="955" minHeight="600">

<s:layout>

<s:HorizontalLayout horizontalAlign="center" verticalAlign="middle" />

</s:layout>

<s:BorderContainer>

<s:layout>

<s:HorizontalLayout paddingBottom="5" paddingLeft="5"

paddingRight="5" paddingTop="5" />

</s:layout>

<s:RichText width="300">

<s:content><s:p>As you can see you declare formatting by

prefixing common tags with an <s:span fontFamily="Courier">s</s:span>

in the same manner as you would form normal Spark

elements.</s:p>

<s:br />Strangely enough forgetting the <s:span

fontStyle="italic" fontWeight="bold">'s' prefix</s:span>

is the commonest point of frustration when applying

formatting inline.<s:br />

<s:p><s:span fontWeight="bold">Remember</s:span> if in

doubt always prefix your elements!</s:p>

</s:content>

</s:RichText>

</s:BorderContainer>

</s:Application>

Para mayor claridad, envolví este código en un BorderContainer para que muestre los límites del componente RichText y apliqué un HorizontalLayout a ella sólo para añadir un poco de relleno. RichEditableText componentes son los mismos que los componentes RichText excepto que se puede editar el texto.

****

Figura 2.8 RichTextEditor

**Navigator Components**

Los Navigatos son, como su nombre lo indica, una colección de componentes que permiten al usuario moverse a través de su contenido. Esto puede ser en una forma lineal o no lineal. La mayoría de los componentes del navegador tienen una propiedad denominada dirección que permiten especificar la orientación del modelo real, por lo que es tan fácil tener una TabBar vertical, que es tener una horizontal. La Figura 2.9 muestra los navegantes.



Figure 2.9 Navigator components

Sin embargo, estos componentes no crean los propios botones, como desarrollador, es necesario proporcionar una fuente de datos para ellos, y ellos a su vez crearán el número de botones en función de cada fuente de datos única.

El código siguiente proporciona información suficiente para la Figura 2.10 para crear y diseñar los botones correspondientes. Si desea crear un sistema de navegación con estado para indicar al usuario qué elemento está activo actualmente, se puede utilizar un ToggleButtonBar o una barra de menú.



Figure 2.10 Basic ButtonBar with four buttons

<?xml version="1.0" encoding="utf-8"?>

<!-- Providing a ButtonBar component with a Array of simple data types (Strings)

--/>

<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/mx"

minWidth="1024" minHeight="768"

creationComplete="application1\_creationCompleteHandler (event)">

<fx:Script>

<![CDATA[

import mx.events.FlexEvent;

private var \_dp :Array = ["Item 1",

"Item 2",

"Item 3",

"Item 4"];

protected function

application1\_creationCompleteHandler(event:FlexEvent):void

{

myBtnBar.dataProvider = \_dp;

}

]]>

</fx:Script>

<mx:ButtonBar id="myBtnBar" />

</s:Application>

El **ToggleButtonBar** funciona de forma idéntica a la barra de botones, pero los elementos que muestra se ponen a cambiar. La **MenuBar** es de naturaleza similar a la ToggleButtonBar, pero tiene un estado de rollover que se pueden activar y desactivar. Cuando el usuario pasa el puntero del ratón sobre una entrada, se destaca la entrada. El componente MenuBar también es compatible con la pantalla de menús jerárquicos, los cuales se crean y se muestran sobre la base de la estructura de datos proporcionados (ver Figura 2.11). Como muchos renders de datos, puede hacer que otros componentes dentro del componente de menú para suministrar una funcionalidad mejorada.



Figure 2.11 ToggleButtonBar and MenuBar navigator components

Pasando de los navegadores del estilo del botón, usted tiene la TabBar y TabNavigator. Estos se diferencian entre sí en que el **TabNavigator** y sus contenidos están vinculados entre sí, mientras que el **TabBar** y su contenido pueden ser separados físicamente por otros problemas de diseño, y si te gusta, otros elementos y componentes. Más allá de esta diferencia visual, que tanto la función de una manera casi idénticas.

El siguiente es un navegador pestaña de clases: el **Acordeón**. Este componente le proporciona un control que puede contener varios recipientes vista dentro de un área determinada. Esto es útil cuando se necesita un sistema multivisión y el espacio es bastante reducido.

El último componente del navegador es la **ViewStack**. Este navegador no tiene aspecto visual. Funciona de un modo similar de ver estados (véase el capítulo 5, "Gestión de los Estados" para obtener más información acerca de estados de vista), pero proporciona una implementación más flexible

**Layout Components**

Ahora que ha analizado los componentes de Flex más visuales, vamos a explorar los responsables para el manejo de posicionamiento y el diseño (que se muestra en la Figura 2.12). Como se mencionó al principio de este capítulo, una de las diferencias básicas entre los componentes Spark y Halo.

En Halo la cuestión fundamental es que los componentes de diseño no implementan el proceso como un conjunto de normas comunes, cada uno de ellos lo aplica como mejor les parezca.



Figure 2.12 Layout components

Se tomó la decisión de corregir este déficit y llevar el modelo de componentes de Spark en línea con la estrategia global de diseño de Flex 4.

Todo el control de diseño se desacopla de los componentes y se mudó a un conjunto de alto nivel de las clases que el diseño aplicado de manera consistente a través de toda la aplicación, ya sea para una vista o un solo componente. El que puede definir fácilmente su propio diseños y asignar un layout dinámicamente en tiempo de ejecución, y el controlador de distribución se aplicarán exactamente de la misma manera como los diseños estándar que vienen con Flex 4. Esto es casi imposible de hacer con los componentes de Halo, más allá de establecer la orientación de horizontal al vertical, o absoluta, que tendría que entrar en algunas subclases pesadas.

**Componentes Gráficos**

Los componentes de gráficos son perfectos para hacer frente a la visualización de datos o creación de informes



Figure 2.14 Charting components

**AIR-Specific Components**

Los componentes específicos de AIR son extensiones de las que ya hemos visto, actualizado para permitir la interacción con el sistema operativo del usuario.



Figure 2.15 AIR components

DeveloperFlexComponents  
Capítulo 4 - (Creating Your First Component)

Para crear el componente necesitamos setearle la superclase y como lo estamos creando de cero,Tambien necesitamos indicarle que herede de UIComponent.

import mx.core.UIComponent;

public class MyFirstComponent extends UIComponent

{

public function MyFirstComponent()

{

super();

}

}

Al hacer que el componente herede de UIComponent automaticamente le estamos proporcionando 5 metodos estos metodos son los que diferencian al component de una actionScript Class comun:



**Contructor**: Este metodo se llama de primero en todos los componentes flex, lleva el mismo nombre que la clase que se esta creando. Permite inicializar variables entre otras cosas.

**CreateChildren**: Este metodo se utiliza para agregar hijos al componente, todos los componentes tienen una lista de hijos dentro de si, todos estos hijos son actualizados por otros métodos como updateDisplayList, para que un objeto pueda ser hijo de un componente debe heredar de la clase.

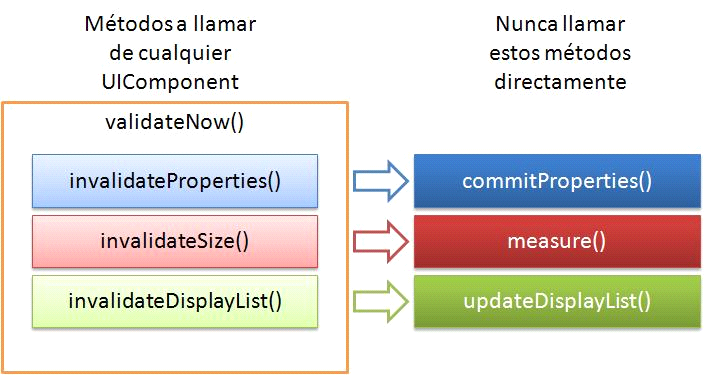
**DisplayObject**: Este metodo solo se ejecuta una vez al inicializarse el componente. Los hijos que se quieran agregar después deben agregarse en otros segmentos de código.

**CommitProperties**: Este metodo se invoca para definir o actualizar las propiedades del componente. Se puede ejecutar varias veces, y siempre se ejecuta la rimera vez justo despues del metodo CreateChildren.

**Measure**: Este metodo se utiliza para definir el tamaño del componente, cada vez que flex no sabe que tamaño tiene el componente entra en este método. Es importante saber que si se define un tamaño explicito (width y height) al componente entonces flex no entrará mas en este método.

**UpdateDisplayList**: Este método se encarga de pintar el componente, en este lugar se ejecutan todas las sentencias de coódigo que tengan que ver con la visualizacion del componente. Tambien se ejecuta muchas veces.

Es importante saber que estos tres últimos métodos mencionados no se pueden invocar directamente, si queremos llamarlos debemos utilizar unas funciones pensadas para eso:



La razón de no llamar a estos métodos directamente es porque el framework de flex debe llamarlos cuando lo considere pertinente, es decir, con los métodos invalidateProperties, invalidateSize e invalidateDisplayList nosotros le decimos a flex que porfavor los llame, a partir de ese momento flex esperará el momento adecuado para llamarlos.

Para mantener guardadas las propiedades de nuestro componente usamos variables privadas y para poder accederlas desde afuera vamos a necesitar tambien crear sus Getters y Setters.  
  
private var \_backgroundColor :uint = 0xff0000;

public function set backgroundColor(val:uint):void

{

\_backgroundColor = val;

}

public function get backgroundColor():uint

{

return \_backgroundColor;

}

Para poder ver los resultados debemos introducir nuestro component en un archivo MXML.

DeveloperFlexComponents  
Capítulo 5 - (Managing States)

**Estados visuales en Flex:**

Los estados visuales permiten variar el contenido y la apariencia de un componente o aplicación, por lo general en respuesta a una acción del usuario. Por ejemplo, lel estado visual de una aplicación podría ser la página de inicio e incluir un logo, una barra lateral, y algunos contenidos de bienvenida. Cuando el usuario hace clic en un botón en la barra lateral, la aplicación cambia dinámicamente su apariencia, es decir, su estado visual, reemplazando el área de contenido principal, con un formulario de orden de compra, pero dejando el logotipo y la barra lateral en su lugar.

Los estados visuales proporcionan al desarrollador Flex una manera fácil de ajustar el aspecto y el comportamiento de una aplicación Flash, por lo general en respuesta a algunas de las acciones del usuario.

Por ejemplo, un botón en su estado por defecto se ve de una manera, cuando está siendo pulsado, el botón tendrá una apariencia ligeramente diferente para indicar la acción que tiene lugar.

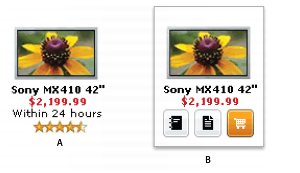
En el uso de estados visuales, se ampliar este concepto a otros componentes o para toda la aplicación.

Ejemplo:

Partes de una aplicación se pueden desactivar hasta que el usuario selecciona una opción en un menú desplegable o también una pantalla de login puede desaparecer una vez que el usuario ha iniciado sesión.

En muchas aplicaciones ricas de Internet, la interfaz cambia en base a las tareas que el usuario esté realizando. Un ejemplo sencillo es una imagen que cambia cuando el usuario pasa el ratón por encima. Ejemplos más complejos incluyen interfaces de usuario cuyo contenido cambiará dependiendo del progreso del usuario a través de una tarea, como cambiar de una vista de exploración por una vista de detalle. Los estados visuales (states views) permiten implementar fácilmente estas aplicaciones.

En su forma más simple, un estado visual define una visión particular de un componente. Por ejemplo, una imagen del producto puede tener dos estados visuales, un estado base con un mínimo de información, y un estado "rico", con enlaces para obtener más información o para añadir el artículo a la cesta de la compra, como muestra la siguiente figura:



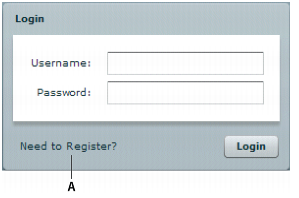
1. estado visual base B. estado visual rico

Para crear un estado visual, se define un estado base y a continuación un conjunto de cambios o redefiniciones (overrides) , que modifican el estado base para definir el nuevo estado visual. Cada estado visual adicional puede modificar el estado base mediante la adición o eliminación de componentes hijos, mediante el establecimiento de los valores de estilo y de propiedad, o definiendo controladores de eventos específicos de cada estado.

Un estado visual no tiene por qué modificar el estado base. Un estado visual puede especificar modificaciones a cualquier otro estado visual. Esto le permite crear un conjunto de estados visuales que comparten funcionalidades comunes, al mismo tiempo que permite la adición de redefiniciones (overrides) específicas para cada estado visual.

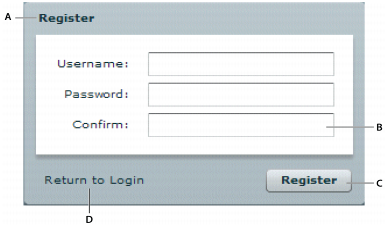
**Definir una interfaz de inicio de sesión utilizando los estados visuales (views states)**

Uno de los usos de los estados visuales es la implementación de un inicio de sesión y formulario de inscripción. En este ejemplo, el estado visual base solicita al usuario que inicie sesión e incluye un control LinkButton que permite registrar el usuario, si es necesario, como muestra la siguiente imagen:



1. LinkButton control

Si el usuario selecciona la necesidad de registrarse, el form cambia el estado visual para mostrar la información de registro, como lo muestra la siguiente imagen:



1. Título modificado del panel contenedor. B. Nuevo elemento del form. C. Etiqueta modificada del Button control. D. Nuevo LinkButton control.

Observe los siguientes cambios en el estado visual base para crear este nuevo estado visual:

- El título del Panel contenedor es “Register”.

- El contenedor del Form tiene un Nuevo control de tipo TextInput para confirmar la password.

- La etiqueta del Button control es “Register”.

El control LinkButton se ha sustituido por un nuevo control LinkButton que permite que el usuario pueda cambiar al estado visual base.

Cuando el usuario hace clic en el vínculo “Return to Login”, el estado visual cambia de nuevo a estado base para mostrar el formulario de inicio de sesión, revirtiendo todos los cambios hechos al haberse cambiado el estado visual de registro.

**Creación de estados visuales:**

Considerar los siguientes puntos cuando se define un estado visual.

- Definir un estado visual en la raíz de una aplicación o en la raíz de un componente personalizado, es decir, como una propiedad dentro de la etiqueta Application o dentro de la etiqueta raíz de un componente MXML.

- Definir estados utilizando la etiqueta <s:states> in MXML.

- Completar la propiedad de estados con un array de uno o más objetos de estado, donde cada objeto estado corresponde a un estado visual.

- Utilice la propiedad name del objeto estado para especificar su identificador.

Ejemplo de código:

<s:Application>

<!-- Define the view states.

The <s:states> tag can also be a child tag of

the root tag of a custom component.

-->

<s:states>

<s:State name="State1"/>

<s:State name="State2"/>

<s:State name="State3"/>

.

.

</s:states>

<!-- Application definition. -->

.

.

</s:Application>

El estado visual predeterminado se define como el primer estado visual en el array <s:states>. Una aplicación utiliza el estado visual predeterminado al cargarse. El nombre del estado visual predeterminado no está reservado, por lo que no se requiere que sea "default".

**Cambiando un estado visual:**

La clase UIComponent define la propiedad currentState que se utiliza para establecer el estado visual actual. Cuando se inicia la aplicación, el valor predeterminado de la propiedad currentState es el nombre del primer estado visual definido por la etiqueta <s:states>.

En el siguiente ejemplo, se utiliza un control Button para establecer la propiedad currentState del objeto Application a "State1" o "State2", el nombre de un estado visual especificado por la etiqueta <s:State>:

<s:Button id="b1" label="Change to State 1" click="currentState='State2';"/>

<s:Button id="b2" label="Change to the default" click="currentState='State1';"/>

El segundo botón en el ejemplo anterior establece el estado actual "State1" para que pueda volver al estado de la vista por defecto de "Estado2".

También puede establecer la propiedad currentState a una cadena vacía para establecerlo en el estado por defecto, como el siguiente ejemplo:

<s:Button id="b2" label="Change to the default" click="currentState='';"/>

Se puede cambiar el estado visual de un componente llamando al método setCurrentState () de la clase UIComponent.

Estableciendo propiedades, estilos y eventos para un estado visual.

Definir los valores de propiedad y estilo específicos de cada estado usando el punto (.). La notación tiene el siguiente formato:

propertyOrStyleName.stateName

Por ejemplo, especificar el valor de la propiedad label de un control Button para el estado visual predeterminado y para un estado visual adicional, como en el siguiente ejemplo:

<s:Button label="Default State" label.State2="New State"/>

**Agregar o eliminar componentes para un estado visual**

Utilizar los atributos MXML includeIn y excludeFrom para especificar el conjunto de estados visuales en los cuales un componente es incluído, donde:

- El atributo includeIn especifica la lista de estados visuales en los que aparece el componente. Este atributo toma una lista separada por comas de nombres de estados visuales, todos los cuales deben haber sido declarados previamente en el array <s:states>.

- El atributo excludeFrom especifica la lista de estados visuales donde se omite el componente. Este atributo toma una lista separada por comas de nombres de estados visuales.

- Los atributos excludeFrom y includeIn son mutuamente excluyentes, por lo que es un error definir ambos en una misma etiqueta MXML.

El ejemplo siguiente utiliza estados visuales para agregar componentes a la aplicación en base al estado actual:

<?xml version="1.0"?>

<!-- states\StatesSimpleIncExc.mxml -->

<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"

xmlns:mx="library://ns.adobe.com/flex/mx"

xmlns:s="library://ns.adobe.com/flex/spark">

<s:layout>

<s:VerticalLayout/>

</s:layout>

<s:states>

<s:State name="default"/>

<s:State name="addCheckBox"/>

<s:State name="addTextInput"/>

<s:State name="addCheckBoxandButton"/>

</s:states>

<s:HGroup >

<!-- Included in the addCheckBox and addCheckBoxandButton view states. -->

<s:CheckBox id="myCB" label="Checkbox"

includeIn="addCheckBox, addCheckBoxandButton"/>

<!-- Included in the addTextInput view state. -->

<s:TextInput id="myTI"

includeIn="addTextInput"/>

<!-- Included in the addCheckBoxandButton view state. -->

<s:Button id="myB"

includeIn="addCheckBoxandButton"/>

<!-- Exclude from addTextInput view state. -->

<s:TextArea text="Exclude from addTextInput"

excludeFrom="addTextInput"/>

</s:HGroup>

<s:HGroup>

<s:Button label="Add CheckBox"

click="currentState='addCheckBox'"/>

<s:Button label="Show Textinput Only"

click="currentState='addTextInput'"/>

<s:Button label="Add CheckBox and Button"

click="currentState='addCheckBoxandButton'"/>

<s:Button label="Default"

click="currentState='default'"/>

</s:HGroup>

</s:Application>

The executing SWF file for the previous example is shown below:

**Controlar el almacenamiento en caché de los objetos creados en un estado visual**

Un cambio sobre un estado visual puede causar la creación de un objeto en Flex. De forma predeterminada, luego que Flex crea el objeto, el mismo es colocado en cache indefinidamente, aún cuando se pasa a otro estado visual.

Utilizar el atributo itemDestructionPolicy con los atributos includeIn excludeFrom para que Flex destruya por completo el objeto al salir de un estado visual, incluyendo la eliminación de la caché. Por lo general es más eficiente permitir que los objetos estén en caché, ya que mejora el rendimiento cuando se cambia de nuevo al estado visual que creó el objeto. Sin embargo, su aplicación podría definir un estado visual que se utiliza muy poco, y no se optaría por asignar memoria para almacenar en caché el objeto.

Se puede utilizar el atributo itemDestructionPolicy en cualquier objeto en MXML que soporte el includeIn y excludeFrom. Los valores posibles para itemDestructionPolicy son never (por defecto) y auto.

El valor never especifica que el objeto se almacena en caché indefinidamente. El valor auto significa que el objeto se destruye al salir de un estado visual donde se encuentra el objeto. En el ejemplo siguiente se establece el atributo de auto:

<s:TextInput includeIn="newTextInput" itemDestructionPolicy="auto"/>

**Utilizando eventos de estados visuales**

Cuando cambia la propiedad currentState de un componente, el objeto estado para los estados que se entran y salen disparan los siguientes eventos:

**enterState**: disparado luego de entran en un estado visual. Disparado por un objeto Estado o por un componente.

**exitState**: se dispara cuando el estado visual está a punto de ser terminado. Disparado por un objeto de estado antes de salir del mismo, y por un componente antes de que salga del estado visual actual. El componente sobre el que se modifica la propiedad currentState para hacer que el cambio de estado dispara los siguientes eventos:

**currentStateChanging**: se dispara cuando el estado visual está a punto de cambiar.

**currentStateChange** Se dispara después de que el estado visual ha completado el cambio.

El ejemplo siguiente utiliza los eventos enterState y exitState para actualizar dos controles TextArea con el nombre del nuevo estado y el del viejo también:}

<?xml version="1.0"?>

<!-- states\StatesSimpleEvent.mxml -->

<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"

xmlns:mx="library://ns.adobe.com/flex/mx"

xmlns:s="library://ns.adobe.com/flex/spark"

height="450">

<s:states>

<!-- Define the new view states. -->

<s:State name="default"

enterState="MyEnterTA.text = 'Enter state: default';"

exitState="MyExitTA.text = 'Exit state: default';"/>

<s:State name="NewButton"

enterState="MyEnterTA.text = 'Enter state: NewButton';"

exitState="MyExitTA.text = 'Exit state: NewButton';"/>

</s:states>

<s:VGroup id="g1">

<s:HGroup>

<s:Button id="b1" label="Click Me"

enabled.NewButton="false"/>

<s:Button id="b2" label="New Button"

includeIn="NewButton"/>

</s:HGroup>

<s:Button label="Change to NewButton state"

click="currentState='NewButton';"/>

<s:Button label="Change to default view state"

click="currentState='default';"/>

<s:TextArea id="MyEnterTA"/>

<s:TextArea id="MyExitTA"/>

</s:VGroup>

</s:Application>

The executing SWF file for the previous example is shown below: