Flex 2 Component Development

London Flash Platform user group January 25th 2007
Mike Jones – FlashGen.Com
http://blog.flashgen.com

Table of Contents

Overview

While component development in Flex isn't hard to grasp if you aren't developing components on a regular basis it can be a trifle slower as most things are until you are fully aware of the steps.

MXML or Actionscript 3.0

When you decide you need to develop a component for the Flex 2 framework what language should you use?

To be honest it doesn't really matter. MXML is generally faster for simple UI components, but as complexity and functionality grow it is easier to utilize Actionscript to do the brunt of the main processing and keep the MXML for managing the presentation layer. That said it can come down to a personal development style that dictates which is favored. For example experienced XML developers may prefer to harness the MXML syntax, while traditional Actionscript developers will, generally prefer to code that way.

When developing components for the Flex framework just relying on MXML will only get you so far. Then again it is likely that if you have developed anything in Flex you'll have already realized this.

In the end both have benefits that should dissuade you from using favoring one over the other.

Component comparison between MXML and AS 3.0

MXML

```
<?xml version="1.0" encoding="utf-8"?>
<mx:ComboBox xmlns:mx="http://www.adobe.com/2006/mxml">
     <mx:dataProvider>
          <mx:String>Chef Ipsum</mx:String>
          <mx:String>Video Test</mx:String>
     </mx:dataProvider>
</mx:ComboBox>
```

ActionScript 3.0

```
package com.flashgen.components.controls {
     import mx.controls.ComboBox;

     public class FGComboBoxAS extends ComboBox
     {
          private var _data  :Array = ["Chef Ipsum", "Video Test"];


          // Define the constructor
          public function FGComboBoxAS()
          {
               super();
               applyData(_data);
          }

          protected function applyData(elements:Array):void
          {
               this.dataProvider = elements;
```

```
            }
        }
}
```

The Actionscript 3.0 component methods

MXML doesn't really have any 'core' methods - as all of the visual components are derived from UIComponent is some fashion. However as we will see later it does have a few methods of it's own that we can capitalize on.

createChildren()

This is basically the same as the AS 2.0 v2 Component framework method of the same name. In a nutshell this is where you would instantiate your child assets and add them to the display list (if required) for display on initialization.

measure()

Again this is similar in nature to size(). Providing the default width and height of the components. It also provides the default minimum size this particular component can be. So no more design view flaws where a user can make the component smaller than is functionally usable.

updateDisplayList()

Like measure(), this is pretty similar to the old draw() method. However it does have the ability to do sizing of any child elements as well as the positioning within the component operational area. It also deals with the application of skins and graphics as required by the component. This is not the same as layoutChrome() though.

layoutChrome()

This method deals with the application of any border area of a component. You will generally only use this when subclassing from Container components / classes or creating a custom component that is a type of container or holder.

commitChanges()

This method deals with the application of property changes within the component itself - either by internal or external intervention. You can either get all changes to invoke at the same time or define a specific order.

Picking the right package

Deciding what you want to base your component off of or create a composition with can be a subjective. For the most part you'll probably extend the more specialized components to add in specific functionality. I personally tend to start off with a blank canvas or extend from UIComponent. I would suggest you treat each case based on their merits and requirements.

The Public API

Another facet component developers need to take in to consideration is that like most developer roles, their code may not be used or implemented by just themselves. However unlike standard class development a component developer can provide a greater scope of accessible channels to aid other developers when integrating their components.

Exposing public the Public API through Binding

Binding properties and methods is achieved via the Binding meta tag. It can be as concise as [Binding], or you can opt to include an event type so that the binding can signal Flex that it has been altered, this then allows Flex to update any other components that are sharing that data. Note that if you omit an event type Flex automatically creates one called 'propertyChange'.

```
[Bindable] //is equivalent to
[Bindable(event="propertyChange")]

[Bindable(event="FGChangeEvent")]
```

Adding properties to the Design view with Inspectable

Inspectable hasn't altered much from AS2.0 it can be defined as simply as [Inspectable] or you can provide a specific data type and or a default range of values if the component is a List, Array etc. The addition of more options is the only real programmatic alteration.

```
[Inspectable] // In it's most succinct form
var isBindable          :Boolean;


[Inspectable(defaultValue=true, verbose=1,
category="Other")]
var isBindable          :Boolean;
```

Values that can be added to the Inspectable tag are:

category - Groups the property into a specific subcategory in the Property inspector of the Flex Builder IDE.
defaultValue - Default value for the inspectable property. This property is required if used in a getter or setter function.
enumeration - Specifies a comma-delimited list of legal values for the property.
environment - Specifies which inspectable properties should not be allowed (none), which are used only for Flex Builder (Flash), and which are used only by Flex and not Flex Builder (MXML).
format - Indicates a value with a file path.
listOffset - Default index in a List value.
name - Specifies the display name for the property; for example, "maxWidth". If not specified, use the property's name, such as _maxWidth.

type - Specifies the type specifier. If omitted, use the property's type. The following values are valid:

- Array
- Boolean
- Color
- Font Name
- List
- Number
- Object
- String

Dispatching events with Event

The Event tag allows you to define what type of event is broadcast when the specified property or method is accessed.

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

If you wish to develop your own events you should extend the Event class in AS 3.0 and provide the values you wish to accompany your new event object. When an event is dispatched it has the following attributes:

type - the type of event i.e.click
target - This is added by DispatchEvent() and is a reference to the element that dispatched the event.
currentTarget - This is different to target, as it details the component instance that is actively processing the event.
eventPhase - this is what phase the event is in, cature, target, bubbling
bubbles - defines whether the event can bubble or not
cancelable - defines whether the event can be cancelled.

Think about storing your events as constants or even, if pertinent, as static properties. This allows for a more flexible approach by loosely associating the values to the dispatchers. This way the actual value could be changed without worry of breaking the rest of the application it is functioning within.

Inherit, composite or customize

This is generally based on what you are trying to achieve or the requirements you have. The likelihood is that most of the time you will be developing components that fall in to the first two options. While it could be argued that 'customize' encompasses both, it is here more to illustrate the development of components, that don't inherit from another UIComponent derived class or have a composed makeup.

Overriding

Overriding comes with a class-based language. In the case of components it depends on what you are adding from a functional standpoint, that will dictate what you need to override. Either way once you have overridden one of the base methods you need to be able to notify your component that things need updating. To this end you can rely on one of three core invalidation methods.

invalidateDisplayList()

This is similar in nature to redraw() from AS 2.0. This marks the component so that it's updateDisplayList() and layoutChrome() methods are called on the next screen update.

invalidateProperty()

In some respects this operates like AS 2.0's change event. If our component has received a data change or its properties have been updated via a specific mechanism we can invoke this method to make sure that commitChanges() gets called during the next screen update.

invalidateSize()

This particular method is obvious compared to the other two, it marks the component so that it's measure() method is called in the next screen update.

IMXMLObject

MXML doesn't have a constructor so it is difficult to invoke a ground-up custom MXML component without some mechanism to act as the constructor. Step up to the plate IMXMLObject

IMXMLObject is an interface that allows MXML components to react to their invocation by the compiler. As MXML utilizes an events-based model to trigger common events like **preinitialize**, **initialize**, and **creationComplete** we can use this model to notify the application of our components status. Unfortunately these events are all inherited from UIComponent so we don't have access to them from our bespoke components.

Now there are a couple of things to know about IMXMLObject and the Flex Builder IDE. Firstly if you chose New > MXML Component from the project options you cannot create a custom component from either IMXMLObject or good old Object. I'm not sure if this an oversight or a bug. I'm going to report it as an inconsistency / bug. Secondly if you switch from code view to design view you will get an error.

The Template component

Template components. Most might think I'm talking about some form of file that has the stub code already included. You couldn't be further from the truth.

The Template component is a special type of control. It allows you to create a component and leave parts of it abstract in their requirement for additional components. For example if you were to extend a Panel components and add a header and footer to it you could explicitly define that these areas could only allow the instantiation of certain controls.

UIComponent

As this can be as broad as making use of UIComponent or as rigid as enforcing that only Button components (and their sub components) can be instantiated.

IDeferredInstance

The template component comes in two variants but they are pretty much identical in their capabilities. The UIComponent variant we have already touched upon. The other is known as an IDeferredInstance Template.

The key differences between UIComponent and IDeferredInstance template components is down to the implementation. IDeferredInstance uses a cast to UIComponent to resolve the components that are defined at author time.

Template Component Example (UIComponent)

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.adobe.com/2006/mxml" width="400"
height="300" creationComplete="init();">
     <mx:Script>
          <![CDATA[
               import mx.containers.HBox;
               import mx.controls.Button;
               import mx.core.UIComponent;

               private var _navContainer          :HBox;
               private var _cntHolder      :HBox;
               public var content          :UIComponent;

               [ArrayElementType("mx.core.UIComponent")]
               public var navigationElements    :Array;

               [ArrayElementType("mx.controls.Button")]
               public var controlElements       :Array;

               private function init():void {
                    this._navContainer = new HBox();
                    this._cntHolder = new HBox();
                    this._cntHolder.autoLayout = true;

                    for (var i :int = 0; i <
navigationElements.length; i++){

     _navContainer.addChild(navigationElements[i]);
                    }

                    for (var j :int = 0; j < controlElements.length;
j++) {
                         _cntHolder.addChild(controlElements[j]);
                    }

                    this.addChild(_navContainer);
                    this.addChild(content);
                    this.addChild(_cntHolder);
                       }
          ]]>
     </mx:Script>
</mx:Panel>
```

Implementing a Template component

Open the following example files and export the application MXML file that came with them. Notice how the two classes are created and where they differ in their operation. (For more information see the comments in the files)

Distribution

Once you have created your components you will more than likely wish to allow other developers to use them. This is where the real finesse comes – Any good developer can create a component. However to be a great component developer you need to consider the end-uder (i.e. another developer).

Assume no prior knowledge

This isn't in relation to coding. This is more to do with providing the right level of information and support material (that you as the developer of the control or controls) feel happy with providing. Remember the likelihood is that you won't be the one actually using your controls.

Just because it makes sense to you...
Don't assume that because you know of certain methods and parameters that are publicly available, that they will be obvious to an end user. Generally, the goal here is to aid development and provide a mechanism to add specific functionality without impacting on time or cost.

Keep it clear
When providing the access points to your component give them transparent names, but don't get too verbose. No-one likes having to type in methods or properties along the lines of 'beginPlaybackOfClip()' when 'playClip()' would suffice - but still convey enough information to the end-user tasked with implementing your controls.

Try to provide some form of context
While we may all develop using the same language, don't assume that we are all on the same wave-length. Take our previous example method 'playClip()'. Taken out of context it provides information that many of us may assume relates to some form of process by which we can play a clip of some sort. However without context we cannot be certain what that may be. Likewise poor context can cause confusion. Imagine if you found the same method, 'playClip()' on a component that dealt with loading and displaying an image, weird, but it could happen.

Packaging your components

Once you're finally ready to let your components out into the wild you have a few options. You can just release the MXML files / Actionscript classes with some documentation and samples in a zip or you can release the components as an SWC (or Swick as some refer to them). The latter approach is my preferred method as it allows for greater control over distribution and deployment.

I'd still place the SWC in a zip for distribution as I can put samples, docs etc in the package as well.

Creating an SWC

So to create an SWC in Flex Builder you need to create a new Flex Library project, once you have one of these define your source folder and place all the classes and files within it that you wish to package within the SWC. You can refine what is included, or excluded, via the properties dialog of the project.

Now a couple of things to note are that if you have automatic build enabled, as most do, the library will immediately try and make a SWC - this generally leads to an entry in the Problem panel as you may not have actually added any source at that time.

Also if you have a 'pathing' issue the SWC generation will fail, but you won't get an error message to either the Console or the Problem panel. This can be a bit frustrating and may be symptomatic of the Mac beta I'm using (I doubt it though).

To create an SWC simple place your classes as I already mentioned in the source folder for the project the SWC will pop out in the 'bin' directory. If you need to rebuild the SWC I find the quickest way is just open one of the files in the source folder and save it - this trips the automatic build and generates a new SWC.

Unlike the commandline compiler you don't have to supply a manifest file if you don't want to. In general all a manifest file allows you to do is define what components should be added to the SWC. In this way you can shield or omit certain components from an SWC if you wish to.

Live Previews

If you wonder where the live previews are created for the Design view, well this is it. The SWC creates an SWF of the file and this is loaded into the Design view to aid in visual layout. To see this in action you need to add your SWC to a Flex project and switch to the Design view – your components will always appear in the component folder called "Custom".

Importing and testing in a project

Once you have a SWC you can import it into a Flex project and test it. To do this open the projects properties dialog and select the **Flex Build Path >Library Path**. On the right you'll see a selection of options; the two that interest us are 'Add SWC' and 'Add SWC Folder'. Depending on how many SWC's you have created you may want to consolidate them into a single location and add the path to the SWC folder. If you want to use specific components on a project you may just opt to use the 'Add SWC' option and add the individual controls needed.

Once the components are added, select your main MXML file and open it, if not already open. You will find them in the Component panel in the Design view of Flex Builder.

A few things to note here - at this point in time you cannot define the directory that the components appear in. This is due to the fact that in Flex Builder 2.x you cannot define a different location within the component panel like you can in Flash 8. Also you may have noticed that they all have a generic icon. This issue we can deal with.

Add a bit of polish

If you are going to make a suite of components for release you will probably want to provide a visual indicator in the Component panel so that a developer can quickly locate the component he needs beyond naming alone. So to add that final bit of polish add a custom icon to your component.

These can be either a GIF, PNG or JPG, I prefer to use PNG's as you can create cleaner looking icons with an alpha than you could with GIF's alone.

Adding a custom icon to your component
Adding a custom icon is pretty easy all you need to do is include this line off meta inside your main component class file:

Something to watch out for here though, if your icon isn't pathed correctly then your Flex Library won't create a new version of your SWC. The big kicker on this is that it won't throw an error either…So if you don't see your SWC being created comment out the icon meta and see if that fixes it. If it does check that your icon file pathing is correct.