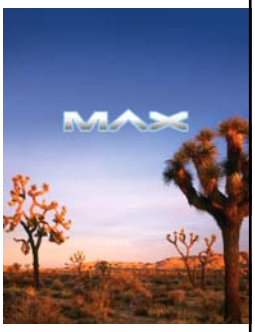
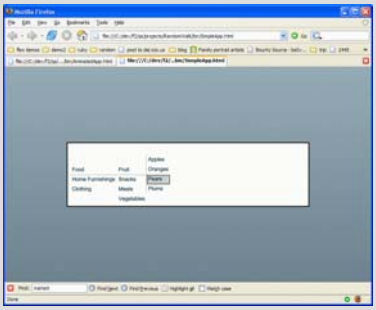
MAX 2006
Beyond Boundaries

Alex Harui
Flex Framework Engineer
Building Custom Components
Adobe Systems Incorporated



A Sample Component: Random Walk



The Component Lifecycle

- From birth to death, a Component goes through a defined set of steps:
  - Construction
  - Configuration
  - Attachment
  - Initialization
  - Invalidation
  - Validation
  - Interaction
  - Detachment
  - Garbage Collection
- Building your component is the process of implementing this lifecycle…

The Component Lifecycle

- Implementing the instantiation portion of the lifecycle boils down to these methods:
  - Constructor()
  - createChildren()
  - commitProperties()
  - measure()
  - updateDisplayList()

Construction
Configuration
Attachment
Initialization
Invalidation
Validation
Interaction
Detachment
Garbage Collection

Choose a Base Class

- UIComponent:
  - Base class for all component and containers
  - Gateway to key flex functionality: styles, Containers, invalidation, etc.
  - Best choice for most components
- Container (and derivatives):
  - Only use if *your* customers will think of your component as a container
  - Allows developers to specify children in MXML (but there are other ways)
  - Scrolling, clipping, layout, and chrome management for free
- Other 'Leaf' Components
  - Good for minor enhancements and guaranteeing type compatibility
  - Major functionality changes run the risk of 'dangling properties'
  - Consider using aggregation instead

Example Code:

```
package
{
…
  public class RandomWalk extends UIComponent
  {
  …
  }
…
}
```

The Component Lifecycle - Construction

- MXML-able components must not have constructor parameters
- Call super()…or the compiler will do it for you.
- Good place to attach your own event handlers
- Try to avoid creating children here…for best performance
- Components are instantiated like this:

```
<local:RandomWalk />
```

which basically resolves to:

```
var instance:RandomWalk = new RandomWalk();
```


Example Code:

```
public function RandomWalk()
{
  super();
  this.addEventListener(MouseEvent.CLICK,
                        clickHandler);
}
```


The Component Lifecycle - Configuration

- MXML assigns properties before components are attached or initialized (avoids duplicate code execution).
- Your properties (get,set functions) need to expect that sub-components haven't been created yet.
- Avoid creating performance bottlenecks: make set functions fast, defer work until validation.
- Attributes set like this:

```
<local:RandomWalk width="600" labelText="hello />
```

basically resolve to:

```
instance.width = 600;
instance.labelText = "hello";
```


Example Code:

- Use the following pattern in your setters:

```
public function set labelText(value:String):void
{
    // store new value in temporary variables
    _labelText = value;
    // BAD _label.text = labelText;
    // label sub-component may not be created yet
    // set a flag and actually apply the new value
    // in commitProperties();
    _labelTextChanged = true;
    // request that commitProperties be called later
    invalidateProperties();
}
```


The Component Lifecycle - Attachment

- A newly constructed component is not attached to the application and therefore will not be displayed
- Most component initialization is deferred until it gets attached to a parent
- Styles may not be initialized until attachment
- Parent.addChild (or addChildAt) calls initialize() method to trigger next phase:

```
parentComponent.addChild(instance);
```

results in call to:

```
instance.initialize();
```


The Component Lifecycle - Initialization

- The default initialize() method does the following:
  1. Dispatch 'preinitialize' event
  2. Calls createChildren method to add sub-components
  3. Dispatch 'initialize' event – component and sub-components are now created
  4. Invalidates the component.
- Later:
  1. First validation pass occurs
  2. 'creationComplete' event is fired – component is fully committed, measured, and updated.

The Component Lifecycle – Initialization

- Override createChildren to create and attach your component's sub-pieces.
  - Creating children here streamlines startup performance
  - Follow the same pattern MXML uses: create, configure, attach.
  - Flex components give subclasses first-crack at defining subcomponents.
  - Don't forget to call super.createChildren();
  - Defer creating dynamic and data-driven components to commitProperties();

Example Code:

```
protected var commitButton:UIComponent;

override protected function createChildren():void
{
  // Flex components check to see if a subclass has
  // overridden what the sub-component is.
  if (commitButton == null)
  {
      commitButton = new Button();
      Button(commitButton).label = "OK";
  }

  addChild(commitButton);
  commitButton.addEventListener(MouseEvent.CLICK,
                                commitHandler);

  super.createChildren();
}
```


The Component Lifecycle - Initialization

- 3 Important Rules for adding children:

1. Containers ***must*** contain only UIComponents
2. UIComponents ***must*** go inside other UIComponents.
3. UIComponents can contain anything (Sprites, Shapes, MovieClips, Video, etc).

The Component Lifecycle - Invalidation

- Flex imposes a *deferred validation* model on the underlying Flash API
- Aggregate changes, defer work until the last possible moment
- avoid creating performance traps for your customers
- Three main invalidation functions:
  - invalidateProperties() for deferred calculation, child management
  - invalidateSize() for changes to the measured size of a component
  - invalidateDisplayList() for changes to the appearance of a component

The Component Lifecycle - Invalidation

5 Rules of Thumb for invalidation:

1. Change values immediately (get what you set)
2. Dispatch events immediately
3. Defer side-effects and calculations to commitProperties()
4. Defer positioning and sizing of sub-components and all drawing to updateDisplayList()
5. Be suspicious of Rules of Thumb

The Component Lifecycle – Validation

- CommitProperties()
  - Invoked by the framework immediately before measurement and layout
  - Use it to calculate and commit the effects of changes to properties and underlying data
  - Avoid extra work: Use flags to filter what work needs to be done
  - Proper place to destroy and create subcomponents based on changes to properties or underlying data.

Example Code:

```
override protected function commitProperties():void
{
   if (labelChanged)
   {
        labelChanged = false;
        label.text = _labelText;
   }
   …
   // if committing properties could change the computed size
   // of the component, call invalidateSize()
   invalidateSize();
   // if committing properties could change the visual appearance
   // of the component, call invalidateDisplayList()
   invalidateDisplayList();
}
```


The Component Lifecycle – Validation

- measure()
  - Invoked by the framework when a component's invalidateSize() is called
  - Components calculate their 'natural' size based on content and layout rules.
  - Implicitly invoked when component children change size.
  - Don't count on it: Framework optimizes away unnecessary calls to measure.
  - Quick Tip: start by explicitly sizing your component, and implement this later.

Example Code:

```
override protected function measure():void
{
   // figure out what it takes to set
   // measuredWidth, measuredHeight
   // measuredMinWidth, measuredMinHeight
   var textMetrics:TextLineMetrics = measureText(labelText);
   measuredWidth = measuredMinWidth = textMetrics.width;
   measuredHeight = measuredMinHeight = textMetrics.height;
}
```


The Component Lifecycle – Validation

- updateDisplayList()
  - Invoked by the framework when a component's invalidateDisplayList() is called
  - The 'right' place to do all of your drawing and layout.

Example Code:

```
override protected function updateDisplayList(
                                    unscaledWidth:Number,
                                    unscaledHeight:Number):void
{
   // position the children
   label.move(0, 0);
   label.setActualSize(unscaledWidth, unscaledHeight);

   // draw anything that needs drawing
   graphics.clear();
   graphics.moveTo(…);
   …
}
```


The Component Lifecycle - Interaction

- Flex is an *Event Driven* Interaction Model
- System is based on the W3C DOM Event model (same model the browsers use).
- Events consist of:
  - *Name*: A unique (per target) name identifying the type of event
  - *Target*: the object that dispatched the event
  - *Event*: An Object containing additional information relevant to the event
  - *Handler*: the function invoked when the event occurs.

Construction
Configuration
Attachment
Initialization
Invalidation
Validation
Interaction
Detachment
Garbage Collection

The Component Lifecycle - Interaction

You'll deal with events in two ways:

1. Handling Input Events. In the constructor we did this:

```
addEventListener(MouseEvent.CLICK, clickHandler);
```

The clickHandler alters the state of the component

```
private function clickHandler(event:MouseEvent):void
{
  // draw the component in an expanded state
  expandItem();
}
```

2. Dispatching Custom Events
   - Flex's event system is extensible – you can define the events you need to make your component useful.

The Component Lifecycle – Interaction

Dispatching Custom Events:

1. Pick a name
   - Choose something descriptive. If the name is already in use, try to make sure your event has the same basic meaning.
   - Flex defines event constants to make AS coders lives easier. You should too.
   - Our naming convention: events that signify something is about to happen are gerunds (itemOpening, stateChanging). Events that signify something has happened are present tense verbs (click, itemRollOver).
2. Define An Event class
   - Define your own class, at least to own your event constant. Add additional fields if there is data your developers might find relevant.
   - It's OK to reuse the same event class if it seems to make sense.

The Component Lifecycle – Interaction

Dispatching Custom Events (continued):

3. Decide if it should bubble.
   - The answer is almost guaranteed to be no.
4. Declare your intent to dispatch the event
   - Class level Metadata lets the compiler (and future doc tools) know that your component dispatches this event, and what Event class it uses.

```
[Event(name="itemClick", type="RandomWalkEvent")]
public class RandomWalk extends UIComponent
{ ...
```

5. Do it.

```
dispatchEvent(new RandomWalkEvent(RandomWalkEvent.ITEM_CLICK,
                                  node));
```


The Component Lifecycle - Detachment

- Components can be added and removed as well as re-parented
- Things to note:
  - Components removed from parents don't get validation calls
  - Re-parenting is not as expensive as initialization, but does have a cost
  - Hiding is usually better than removing if the component will become visible again later
  - You should remove before adding when re-parenting
  - Only removed components get garbage collected

Construction
Configuration
Attachment
Initialization
Invalidation
Validation
Interaction
Detachment
Garbage Collection

The Component Lifecycle – Garbage Collection

- Components removed from the display list will be garbage collected automatically once there are no other references to them
  - References found by following references starting from the application (and static classes) count
  - References from children back to parents don't count
- Common causes of memory leaks:
  - Event Listeners (especially on data). Listening to parent's events actually means the parent has a reference to a child's method.
  - Maps and Arrays
- Flash provides weak references in these cases
  - Improper implementation results in "random" exceptions

Construction
Configuration
Attachment
Initialization
Invalidation
Validation
Interaction
Detachment
Garbage Collection

Making Your Custom Component Customizable

So far, we've focused on making the component work.

But to re-use this component in other contexts, you may want to allow for customization. We call this "generalizing".

Three important concepts for generalizing your component:

- ***SKINNING!***
- ***STYLING!***
- ***TEMPLATING!***

Generalizing Your Component

Roughly Speaking…

- Use Properties to generalize the behavior and data
- Use Skinning and Styling to generalize the look
- Use *Templating* to generalize the sub-components/children , if any.

- We sometimes refer to templating as 'custom containers,' or 'control composition.'

Two different mechanisms for Templating…

Generalizing Your Component: Templating

1. Instance properties
   - Properties typed as UIComponent can be set in MXML like any other property.
   - Re-parenting allows you to parent these UIComponents as sub-components.
   - Allows you to define complex components with configurable parts

```
public function set thumbnailView(value:UIComponent)
{
      _thumbnailView = value;
      addChild(thumbnailView);
}
```


Generalizing Your Component: Templating

2. Item Renderers (Factories)
   - Factories are used to generate multiple child components
   - Data driven components use them to generate *renderers* for the data
   - Allows you to separate management of the data from displaying the data.

   Quick Tips:
   - Type your item renderers as IFactory
   - Use the IDataRenderer interface to pass your data to the instances
   - If you have additional data to pass, define a custom interface and test to see if it is supported first.

Generalizing Your Component: Styles

- Styles:
  - can return strings, numbers and even classes to instantiate as children
  - can inherit, properties cannot
  - cannot participate in databinding, properties can
  - take longer to compute.
- Add Style metadata to the class definition

```
[Style(name="verticalGap", type="Number", inherit="no")]
```

- Call getStyle() in your code

```
var result:Number = getStyle("verticalGap");
```


Generalizing Your Component: Skinning

- Many of the visual aspects of a component are complex graphics known as "skins".
  - Buttons
  - Scrollbar arrows
- A skin is really an instance of a class that has Actionscript that draws the visual, or an instance of a class that displays a bitmap.
- Get the class to instantiate by using getStyle() to get a custom style property.

```
var highlightClass:Class = getStyle("itemHighlightSkin");
if (highlightClass != null)
{
       _highlight = new highlightClass();
       addChild(DisplayObject(_highlight));
}
```


Generalizing Your Component: Binding

- Databinding is there to eliminate boilerplate data routing code

```
<mx:Button enabled="{randomWalk.selectedItem != null}" />
```

- Any property can be the destination of a binding, but the source needs special support

- Good rule of thumb: If you think someone *might* want to bind to it…make it bindable.

How do I make a property bindable?

Generalizing Your Component: Binding

1. Add [Bindable] to your class:

```
[Bindable]
public class RandomWalk extends UIComponent { …
```

- Makes all public vars bindable
- Convenience feature for value objects.

2. Add [Bindable] to your property

```
[Bindable]
public var selectedItem:Object;

[Bindable]
public function get selectedItem():Object { …
```

- Wraps the variable or property in an autogenerated get/set
- Good for simple properties.

Generalizing Your Component: Binding

3. Roll your own event based bindings:

```
[Bindable(event="selectedItemChange")]
public function get selectedItem():Object
{
        …
        dispatchEvent(new Event("selectedItemChange"));
```

- Works well for read only and derived properties.

Generalizing Your Component: API Design

Your API defines your MXML schema

- Specifically:
  - ClassName -> XML Tags Name
  - Package -> XML Namespace
  - Properties -> XML Attributes
  - Complex Properties -> Child Tags
- When you design a Component API, you're designing a mini-schema, as well.

Generalizing Your Component: API Design

- Choose Properties over Methods
  - Properties can be set from MXML
- Avoid write-once properties
  - Anything that can be set in MXML can be bound to.
- Use value objects for complex and multi-valued properties
  - MXML makes object graphs simple

```
<DataGrid>
  <columns>
    <DataGridColumn columnName="revenue" dataField="revYr" />
    <DataGridColumn columnName="profit" dataField="profYr" />
  </columns>
</DataGrid>
```

- Use different names for styles, events, and properties.

Generalizing Your Component: Metadata

- MXML uses AS3 Metadata to provide hints to the compiler and tool
- Metadata improves the support MXMLC and FlexBuilder can give to your customers.
- Control the contents of an array property:

```
[ArrayElementType("Number")]
public function set thresholds(value:Array):void {}
```

- Control the values of an string property:

```
[Inspectable(enumeration="vertical,horizontal")]
public function set direction(value:String):void {}
```

- Group your properties in FlexBuilder's Property Inspector:

```
[Inspectable(category="Data")]
public function set filterData(value:Boolean):void {}
```


Better by Adobe™