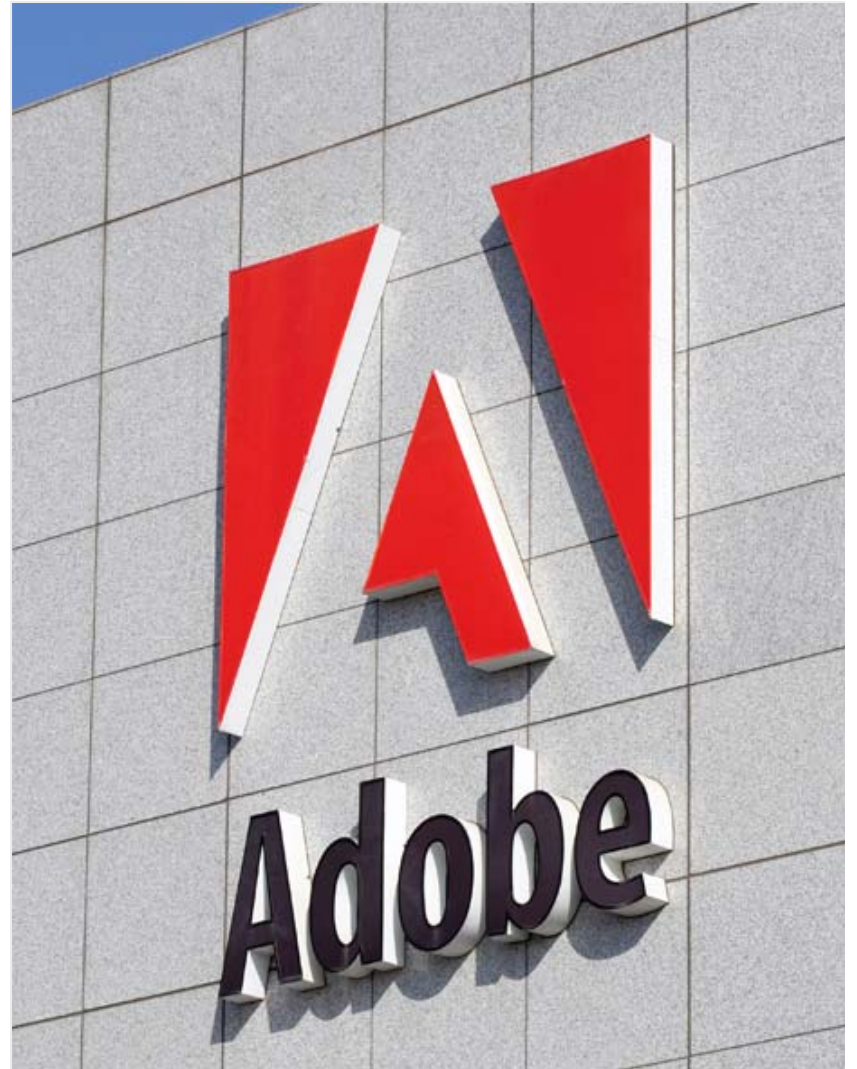


Building a Flex Component

Ely Greenfield

Adobe Systems



How do you build a Flex Component?

1. Design it

- Problems
- Ideas
- Specs
- APIs

2. Prototype it

- Initialization
- Layout and Rendering
- Interaction
- Animation

3. Generalize it

- Skinning
- Styling
- Templating
- Localization

4. Polish it

- Accessibility
- Automation
- Binding

How do you build a Flex Component?

1. Design it

- Problems
- Ideas
- Specs
- APIs

2. Prototype it

- **Initialization**
- **Layout and Rendering**
- **Interaction**
- Animation

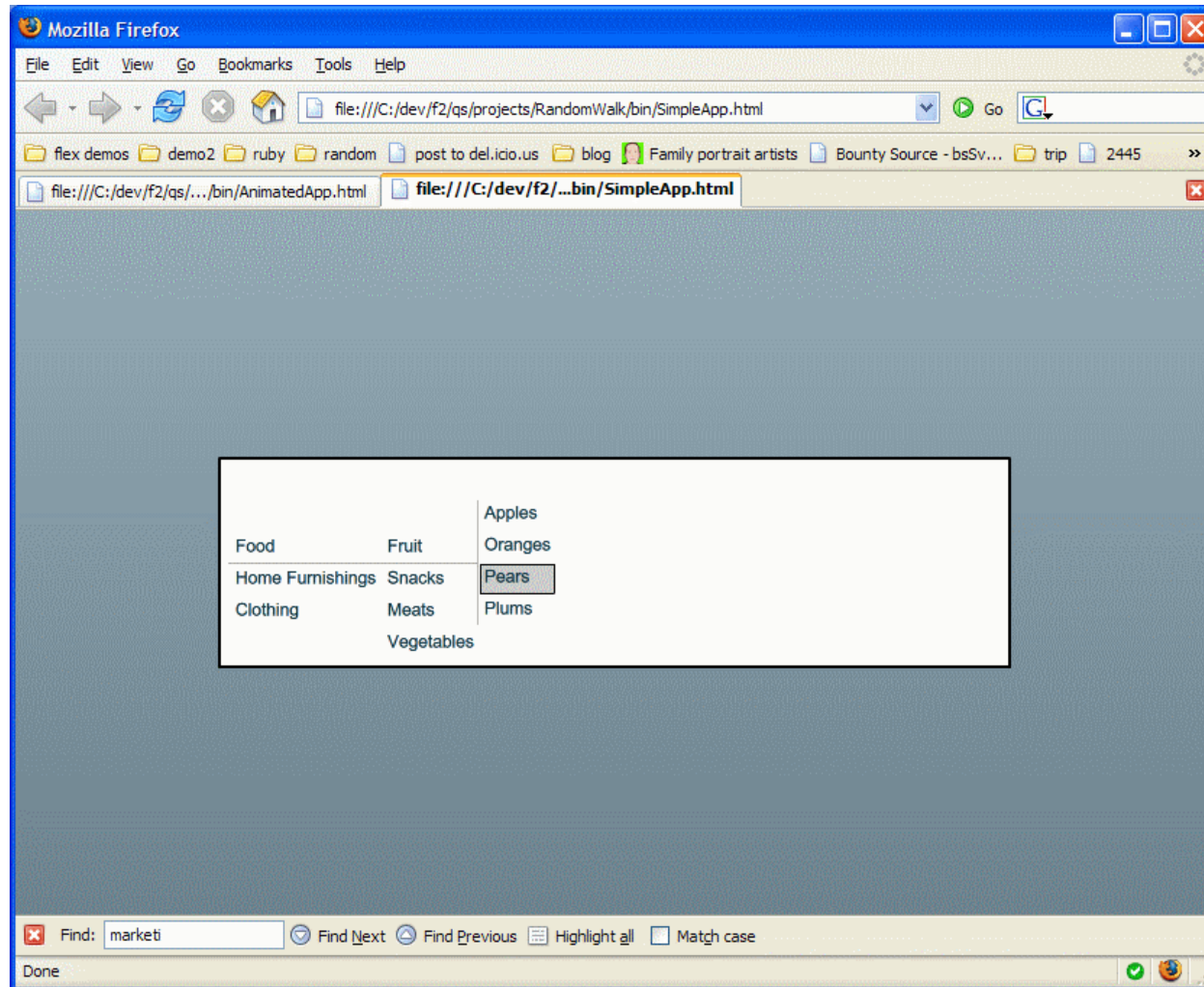
3. Generalize it

- Skinning
- Styling
- **Templating**
- Localization

4. Polish it

- Accessibility
- Automation
- **Binding**

A Sample Component: Random Walk



A Sample Component: Random Walk

So What
did we Add?

The screenshot shows a Mozilla Firefox browser window displaying a web application. The address bar shows the file path: file:///C:/dev/f2/qs/projects/RandomWalk/bin/AnimatedApp.html. The browser's menu bar includes File, Edit, View, Go, Bookmarks, Tools, and Help. The address bar also shows a search engine icon and a 'Go' button. The browser's status bar at the bottom shows 'Done' and a search bar with the text 'marketi'. The main content area displays a list of items under the heading 'Food'. The items are organized into columns: Fruit, Snacks, and Fritos. The 'Snacks' column is highlighted in yellow. The items listed are: Fruit (Home Furnishings, Clothing), Snacks (Meats, Vegetables), Fritos (Snickers, Triscuts, Peanuts, Oreos, Gummy Bears, Pringles, Chips Ahoy, Mint Melties), and Mini Standard King. The following callout boxes point to specific features: 'Back Button Support' points to the back button in the browser's address bar; 'Graphic Skins' points to the 'Food' heading; 'Stylable/Programmatic Skins' points to the 'Gummy Bears' item; 'Focus/Keyboard Support' points to the 'Gummy Bears' item; 'Animation' points to the 'Gummy Bears' item; 'Custom Events Binding' points to the 'Mini Standard King' item; and 'Measurement' points to the 'Mini Standard King' item.

Back Button Support

Graphic Skins

Custom Events Binding

Measurement

Stylable/Programmatic Skins

Animation

Focus/Keyboard Support

Building a prototype

The Flex Component Lifecycle

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 prototype is the process of implementing this lifecycle...

The Component Lifecycle

Construction
Configuration
Attachment
Initialization
Invalidation
Validation
Interaction
Detachment
Garbage Collection

- Implementing the lifecycle boils down to these methods:
 - Constructor()
 - createChildren()
 - commitProperties()
 - measure()
 - updateDisplayList()
 - Custom events

See if you can spot them during tonight's broadcast!

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

Invalidation

Validation

Interaction

Detachment

Garbage Collection

- MXML-able components must have zero arg constructors
- 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

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

or

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

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

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

Invalidation

Validation

Interaction

Detachment

Garbage Collection

- 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.
- Follow this pattern when creating sub-components.

```
...
instance.dataProvider = xmlDataSet;
instance.width = 600;
instance.height = 200;
instance.labelText = "hello";
...
```

```
public function set
labelText(value:String):void
{
    _labelText = value;
    //BAD _label.text = labelText
    _labelTextDirty = true;
    invalidateProperties();
}
```

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

Invalidation

Validation

Interaction

Detachment

Garbage Collection

- Most component initialization is deferred until it gets attached to a parent
- Styles may not be initialized until its ancestors get rooted to the displayList
- `Parent.addChild(At)` calls `initialize()` method to trigger next phase...you can call this explicitly if necessary

...

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

...

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

Invalidation

Validation

Interaction

Detachment

Garbage Collection

- initialization happens in multiple sub-phases:
 1. 'preinitialize' event is dispatched
 2. createChildren method is called, adds sub-components
 3. 'initialize' event is called – component is fully created
 4. First validation pass occurs
 5. 'creationComplete' event is fired – component is fully committed, measured, and updated.

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

Invalidation

Validation

Interaction

Detachment

Garbage Collection

- **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()`;

```
protected var commitButton:UIComponent;
override protected function createChildren():void
{
    if (commitButton == null)
    {
        commitButton = new Button();
        Button(commitButton).label = "OK";
    }
    addChild(commitButton);
    commitButton.addEventListener(MouseEvent.CLICK, commitHandler);
    super.createChildren();
}
```

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

Invalidation

Validation

Interaction

Detachment

Garbage Collection

- 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

Construction

Configuration

Attachment

Initialization

Invalidation

Validation

Interaction

Detachment

Garbage Collection

- 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

Rules of Thumb:

1. Change values immediately
2. Dispatch events immediately
3. Defer Side-effects and calculations to `commitProperties()`
4. Defer rendering to `updateDisplayList()`
5. Be suspicious of rules of Thumb

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

invalidation

Validation

Interaction

Detachment

Garbage Collection

- **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.

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

invalidation

Validation

Interaction

Detachment

Garbage Collection

- **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.

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

invalidation

Validation

Interaction

Detachment

Garbage Collection

- **updateDisplayList()**
 - Invoked by the framework when a component's `invalidateDisplayList()` is called
 - The 'right' place to do all of your drawing and layout.
 - You'll hear more about this later.

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

invalidation

Validation

Interaction

Detachment

Garbage Collection

- 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.

You'll deal with events in two ways:

1. Handling Events

- Registering, removing, capture, bubble – More on this tomorrow.

2. Dispatching Events

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

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

invalidation

Validation

Interaction

Detachment

Garbage Collection

How to dispatch your own event

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 happening 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

Construction

Configuration

Attachment

Initialization

invalidation

Validation

Interaction

Detachment

Garbage Collection

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 let's the compiler (and future doc tools) know that your component dispatches this event, and what Event class it uses.

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

5. Do it.

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

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

invalidation

Validation

Interaction

Detachment

Garbage Collection

A Few Other thoughts on Events:

- If you dispatch the same event as a base class, you must use the same event class.
- Remember that events will bubble up from your sub-components. If you don't want that to happen, you need to explicitly stop them from propogating

The Component Lifecycle

Construction
Configuration
Attachment
Initialization
invalidation
Validation
Interaction

Detachment

Garbage Collection

- The Flash Display List is putty in your hands
- Components can be added, removed, reparented, etc.
- Things to note:
 - Components off the display list don't get validation calls
 - Reparenting is not as expensive as initialization, but does have a cost
 - For volatile sub-components, consider hiding instead of removing

```
bookmarkDock.addChild( browsePanel.selectedProductView );
```

The Component Lifecycle

Construction

Configuration

Attachment

Initialization

invalidation

Validation

Interaction

Detachment

Garbage Collection

- Components removed from the display list will be garbage collected automatically
- Beware the dark side...manage your references carefully!
- Common causes of memory leaks:
 - Event Listeners (especially on data)
 - Mapping Dictionaries
- Flash provides weak references in these cases
- Quick Tip: Remember, if you use a weak reference, make sure the object/function/etc. is being referenced by someone else.

```
myObject.addEventListener("change",  
    function () { ... }, false,0,true); // Uh Oh!
```

The Component Lifecycle

- Here's your prototyping checklist:
 - Constructor()
 - createChildren()
 - commitProperties()
 - measure()
 - updateDisplayList()
 - Custom events

Generalize it

Extending the reach of your component

Generalizing your component

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

But Flex's mission is *functionality* combined with *versatility* and *expressiveness*.

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 content.
-
- 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.
- Reparenting allows you to embed passed values into your own display tree.
- 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.

Polish it

Putting a professional touch on it

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.

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.

Design it

Designing your API

(we got through the other stuff pretty fast)

Choosing a base class

What Base Class should you extend?

- **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, 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

Designing your API

Remember, 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.

Designing your API

A Few Examples:

- 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" width="30" dataField="revYr" />  
    <DataGridColumn columnName="profit" width="30" dataField="profYr" />  
  </columns>  
</DataGrid>
```

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

Designing your API

- 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 a 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 {}
```

And More...

How do you build a Flex Component?

1. Design it

- Problems
- Ideas
- Specs
- APIs

2. Prototype it

- Initialization
- Layout and Rendering
- Interaction
- Animation

3. Generalize it

- Skinning
- Styling
- Templating
- Localization

4. Polish it

- Accessibility
- Automation
- Binding

Thanks