Agenda

- Introductions
- Quick Flex Overview
- Flex 3 component model
- Break
- Flash Catalyst
- Gumbo (Flex 4)
- Q&A
Your Speakers

- **Matt Chotin**
  - Sr. Product Manager, Flex SDK
  - With Adobe/Macromedia since 2003, working on Flex 5+ years
  - Java engineering background

- **Chet Haase**
  - Author of Filthy Rich Clients with Romain Guy
  - Sr. Computer Scientist, Flex SDK
  - Focus on graphics and animation
FLEX OVERVIEW
Understanding Flex

- **Flex SDK**
  - 2 languages
    - MXML
    - ActionScript 3
  - Compilers
  - Rich Component Library
  - Debuggers

*Flex SDK*

**Flex Class Library**

**Debuggers**
Understanding Flex

- **Flex SDK**
  - 2 languages
    - MXML
    - ActionScript 3
  - Compilers
  - Rich Component Library
  - Debuggers

- **Flex Builder IDE**
  - Eclipse Plugin or turn-key install
  - Accelerates Design and Development
  - Design view and code view
How Flex Works in the Browser

Flex SDK
- MXML
- ActionScript
- Flex Class Library
- Debuggers

Flex Builder IDE

```xml
<?xml version="1.0" encoding="utf-8"?>
    xmlns="*" layout="absolute"
    creationComplete="initApp()">

<mx:Script>
    private function initApp():void {
        hs.send();
    }
</mx:Script>

<mx:HTTPService id="hs" url="data/catalog.xml" />

<mx:Panel layout="vertical" top="10" left="10"
    right="10" bottom="10">
    <mx:TileList itemRenderer="Thumb" width="100%" height="100%"
        dataProvider="(hs.lastResult.catalog.product)" />
</mx:Panel>
</mx:Application>
```
How Flex Works in the Browser

Flex SDK
- MXML
- ActionScript
- Flex Class Library
- Debuggers

Flex Builder IDE

Compile

Web Server

Browser
- Flash Player

SWF
How Flex Works in the Browser

Flex SDK
- MXML
- ActionScript
- Flex Class Library
- Debuggers

Flex Builder IDE

Compile

.Webs.ch

Browser
- Flash Player

Web Server
- SOAP
- HTTP/S
- AMF/S
- RTMP/S

Existing Applications & Infrastructure

XML/HTTP
- REST
- SOAP Web Services

J2EE Application Server

LC Data Services
How Flex Works on the Desktop

Flex SDK
- MXML
- ActionScript
- Flex Class Library
- Debuggers

Flex Builder IDE

```xml
<?xml version="1.0" encoding="utf-8"?>
    xmlns="*" layout="absolute"
    creationComplete="initApp()">
    <mx:Script>
        private function initApp():void {
            hs.send();
        }
    </mx:Script>
    <mx:HTTPService id="hs" url="data/catalog.xml" />
    <mx:Panel layout="vertical" top="10" left="10"
        right="10" bottom="10">
        <mx:TileList itemRenderer="Thumb" width="100%" height="100%"
            dataProvider="(hs.lastResult.catalog.product)" />
        </mx:Panel>
    </mx:Application>
```
How Flex Works on the Desktop

Compile & Package

Desktop

Files
SQLite
Notifications
Clipboard

AIR Client Runtime

Web Server

SOAP
HTTP/S
AMF/S
RTMP/S

XML/HTTP
REST
SOAP Web Services

Existing Applications & Infrastructure

Web Server

LC Data Services

J2EE Application Server

Flex SDK

MXML
ActionScript

Flex Class Library

Debuggers

Flex Builder IDE

Compile & Package

SWF

Copyright 2008 Adobe Systems Incorporated. All rights reserved.
Flex Component Lifecycle

- A mechanism the framework uses to create, manage and destroy components
- A mechanism that makes the most of the player rendering model.
Vocab Lesson

Flex Component Lifecycle

- A mechanism the framework uses to create, manage and destroy components
- A mechanism that makes the most of the player rendering model.

Halo: Component architecture used in Flex 3 and earlier versions.
Vocab Lesson

Flex Component Lifecycle
- A mechanism the framework uses to create, manage and destroy components
- A mechanism that makes the most of the player rendering model.

Halo: Component architecture used in Flex 3 and earlier versions.

Gumbo/Spark
- Gumbo is the codename for the next version of Flex and Flex Builder
- Spark is the component and skinning architecture in Gumbo.
- Spark is built on top of Halo.
Vocab Lesson

Flex Component Lifecycle

- A mechanism the framework uses to create, manage and destroy components
- A mechanism that makes the most of the player rendering model.

Halo: Component architecture used in Flex 3 and earlier versions.

Gumbo/Spark

- Gumbo is the codename for the next version of Flex and Flex Builder
- Spark is the component and skinning architecture in Gumbo.
  - Spark is built on top of Halo.

Parts: Internal parts of a component that are composited together to create a whole component.

- Composition is the name of the game.
Flex Component Lifecycle

- A mechanism the framework uses to create, manage and destroy components
- A mechanism that makes the most of the player rendering model.

Halo: Component architecture used in Flex 3 and earlier versions.

Gumbo/Spark

- Gumbo is the codename for the next version of Flex and Flex Builder
- Spark is the component and skinning architecture in Gumbo.
- **Spark is built on top of Halo.**

Parts: Internal parts of a component that are composited together to create a whole component.

- Composition is the name of the game.

**Lets write a component!**
Defining Patterns in Halo

- Invalidation/Validation Model
  - Methodology to aggregate changes and defer work until an optimal later time
Halo Component Architecture Patterns

Defining Patterns in Halo

- Invalidation/Validation Model
  - Methodology to **aggregate** changes and **defer** work until an optimal later time
- Event Driven Interaction Model
  - Inform the component if something is about to or has already occurred
Defining Patterns in Halo

- Invalidation/Validation Model
  - Methodology to **aggregate** changes and **defer** work until an optimal later time

- Event Driven Interaction Model
  - Inform the component if something is about to or has already occurred

- Composition
  - Parameterization of a component’s appearance or content.
  - Most often occurs through factories and item renderers.
3 Phase Lifecycle

1. Initialization
Halo Component Lifecycle – Broken Down

3 Phase Lifecycle

1. Initialization
   - Construction
3 Phase Lifecycle

1. Initialization
   - Construction
   - Configuration
3 Phase Lifecycle

1. Initialization
   - Construction
   - Configuration
   - Attachment
3 Phase Lifecycle

1. Initialization
   - Construction
   - Configuration
   - Attachment
   - Initialization
3 Phase Lifecycle

1. Initialization
   - Construction
   - Configuration
   - Attachment
   - Initialization

2. Updating
   - Component responds to changes by using the Invalidation/Validation Model
3 Phase Lifecycle

1. Initialization
   - Construction
   - Configuration
   - Attachment
   - Initialization

2. Updating
   - Component responds to changes by using the Invalidation/Validation Model

3. Destruction
   - Out of sight, out of mind
   - Detachment
   - Garbage collection
Lifecycle Phase 1: Initialization

**Construction**
- Configuration
- Attachment
- Initialization

**Invalidation**
- Validation

**Detachment**
- Garbage Collection
Lifecycle Phase 1: Initialization

Construction: Component begins its lifecycle

- Construction
  - Configuration
  - Attachment
  - Initialization
- Invalidation
- Validation
- Detachment
- Garbage Collection
Lifecycle Phase 1: Initialization

Construction: Component begins its lifecycle

- Construction
  - Configuration
  - Attachment
  - Initialization
- Invalidation
- Validation
- Detachment
- Garbage Collection
Lifecycle Phase 1: Initialization

Construction: Component begins its lifecycle

- Decide on the right base-class
- Component is instantiated, through the `new` operator in ActionScript or in markup
Construction: Component begins its lifecycle

- Decide on the right base-class
- Component is instantiated, through the `new` operator in ActionScript or in markup
- Constructor **must** have zero required arguments
- Constructor can add event listeners, hard code initialization properties of super classes
- Minimal work should occur here.
Lifecycle Phase 1: Initialization

- Construction
- Configuration
  - Attachment
  - Initialization
- Invalidation
- Validation
- Detachment
- Garbage Collection
Lifecycle Phase 1: Initialization

Configuration: Component properties are set internally to be processed later
Lifecycle Phase 1: Initialization

Configuration: Component properties are set internally to be processed later

- Property values are assigned before parts are attached or initialized to avoid duplicate code execution.
- Properties must expect that parts haven’t been created yet.
Lifecycle Phase 1: Initialization

- Construction
- Configuration
- **Attachment**
  - Initialization
  - Invalidation
  - Validation
- Detachment
- Garbage Collection
Lifecycle Phase 1: Initialization

- Construction
- Configuration
- **Attachment**
- Initialization
- Invalidation
- Validation
- Detachment
- Garbage Collection

Attachment: Addition to the display list
Attachment: Addition to the display list

- What is the display list?
Lifecycle Phase 1: Initialization

Attachment: Addition to the display list

- What is the display list?
- Component is added to the display list through an `addChild()` call by its parent.
Attachment: Addition to the display list

- What is the display list?
- Component is added to the display list through an `addChild()` call by its parent.
- Without attachment, component lifecycle will stall.
Lifecycle Phase 1: Initialization

- Construction
- Configuration
- Attachment
  - **Initialization**
  - Invalidation
  - Validation
- Detachment
- Garbage Collection
Lifecycle Phase 1: Initialization

- Initialization
  - Full invalidation/validation cycle is invoked (we will come back to this)
Lifecycle Phase 1: Initialization

Initialization

- Full invalidation/validation cycle is invoked (we will come back to this)

- 5 main lifecycle actions occur at this step:
  - preinitialize event is dispatched
  - createChildren() is called
  - initialize event is dispatched
  - First full invalidation/validation pass occurs
  - creationComplete event is dispatched
Lifecycle Phase 1: Initialization

- Construction
- Configuration
- Attachment
- **Initialization**
- Invalidation
- Validation
- Detachment
- Garbage Collection
Lifecycle Phase 1: Initialization

createChildren() - The attachment workhorse

- Ideal place for adding children that are required throughout the lifetime of the component
- Dynamic or data-driven parts which should be added in commitProperties()
Lifecycle Phase 1: Initialization

createChildren() - The attachment workhorse

- Ideal place for adding children that are required throughout the lifetime of the component
- Dynamic or data-driven parts which should be added in commitProperties()
- Check to make sure the children have not been instantiated already
- Follow the same pattern Flex uses: construct, configure, attach.
Lifecycle Phase 1: Initialization

**createChildren()** - The attachment workhorse

- Ideal place for adding children that are required throughout the lifetime of the component
- Dynamic or data-driven parts which should be added in `commitProperties()`
- Check to make sure the children have not been instantiated already
- Follow the same pattern Flex uses: construct, configure, attach.

**Halo Rules:**

- UIComponents can contain anything (Sprites, Shapes, MovieClips, Video, etc).
- UIComponents *must* go inside other UIComponents
- Containers *must* contain only UIComponents
Lifecycle Phase 1: Initialization

- Construction
- Configuration
- Attachment
- Initialization
- Invalidation
- Validation
- Detachment
- Garbage Collection
First full invalidation/validation pass occurs here

- Invalidation is captured by 3 methods:
  - `invalidateProperties()`
  - `invalidateSize()`
  - `invalidateDisplayList()`

- Validation is captured by 3 methods:
  - `commitProperties()`
  - `measure()`
  - `updateDisplayList()`
Lifecycle Phase 1: Initialization

First full invalidation/validation pass occurs here

- Invalidation is captured by 3 methods:
  - `invalidateProperties()`
  - `invalidateSize()`
  - `invalidateDisplayList()`

- Validation is captured by 3 methods:
  - `commitProperties()`
  - `measure()`
  - `updateDisplayList()`
Lifecycle Phase 1: Initialization

First full invalidation/validation pass occurs here

- Invalidation is captured by 3 methods:
  - invalidateProperties()
  - invalidateSize()
  - invalidateDisplayList()

- Validation is captured by 3 methods:
  - commitProperties()
  - measure()
  - updateDisplayList()

Phase 1 Done!
Lifecycle Phase 1: Initialization

First full invalidation/validation pass occurs here

- Invalidation is captured by 3 methods:
  - invalidateProperties()
  - invalidateSize()
  - invalidateDisplayList()

- Validation is captured by 3 methods:
  - commitProperties()
  - measure()
  - updateDisplayList()

Phase 1 Done!
Phase 2: Updating

- Construction
- Configuration
- Attachment
- Initialization
- **Invalidation**
- **Validation**
- Detachment
- Garbage Collection
Phase 2: Updating

Our component has been created!

It’s a living, breathing entity, and now it needs to know how to update.
Phase 2: Updating

Our component has been created!
It’s a living, breathing entity, and now it needs to know how to update.

Updates occur
- When a user interacts with a component
- When methods or properties are invoked/set

A component should use Flex’s Invalidation/Validation Model to respond to changes.
Our component has been created!

It’s a living, breathing entity, and now it needs to know how to update.

Updates occur

- When a user interacts with a component
- When methods or properties are invoked/set

A component should use Flex’s Invalidation/Validation Model to respond to changes.
Images courtesy of Sean Christmann
The Elastic Racetrack

Traditional Flash Player Elastic Racetrack

Images courtesy of Sean Christmann
The Elastic Racetrack

Traditional Flash Player Elastic Racetrack

Images courtesy of Sean Christmann
The Elastic Racetrack

Traditional Flash Player Elastic Racetrack

Flex component lifecycle is built atop this frame model

Invalidation/Validation takes advantage of the elastic racetrack to get work done in an efficient manner.

Images courtesy of Sean Christmann
Deferred Validation Model

Validation occurs right before Rendering

Code Execution

Rendering

1 frame
Deferred Validation Model

- Waiting for update request
- Update Requested
- Invalidation
- Validation

Validation occurs right before Rendering

Diagram: 1 frame, Code Execution, Rendering
Deferred Validation Model: An Optimization

Invalidation/validation model is split into 3 phases:
Deferred Validation Model: An Optimization

Invalidation/validation model is split into 3 phases:

- Update component properties

\[\text{invalidateProperties()} \quad \rightarrow \quad \text{commitProperties()}\]
Deferred Validation Model: An Optimization

Invalidation/validation model is split into 3 phases:

- Update component properties

  - invalidateProperties()
  - commitProperties()

  - invalidateSize()
  - measure()
Deferred Validation Model: An Optimization

Invalidation/validation model is split into 3 phases:

- Update component properties
  - invalidateProperties()
  - commitProperties()

- Update sizing & measurement information
  - invalidateSize()
  - measure()

- Update drawing and positioning
  - invalidateDisplayList()
  - updateDisplayList()
commitProperties()

- Construction
- Configuration
- Attachment
- Initialization
- Invalidation
- Validation
- Detachment
- Garbage Collection
commitProperties()

Property management phase of validation

- **Purpose**: Commit values typically set using a property setter
- Invoked by the framework before measurement and layout.
Property management phase of validation

- **Purpose**: Commit values typically set using a property setter
- Invoked by the framework before measurement and layout.
- There is a definite pattern that should be followed in order to avoid extra work.
  - Dirty flags and storage variables
Property management phase of validation

- **Purpose**: Commit values typically set using a property setter
- Invoked by the framework before measurement and layout.
- There is a definite pattern that should be followed in order to avoid extra work.
  - Dirty flags and storage variables
- This is the place to add/remove children **not** required through the life of the entire component (as opposed to `createChildren()`)
measure()
Sizing phase of validation

- **Purpose**: Component can calculate its ‘natural’ size based on content and layout rules.
Sizing phase of validation

- **Purpose**: Component can calculate its ‘natural’ size based on content and layout rules.

- Implicitly invoked when component children change size. (Don’t call `measure()` on your children).
measure()

Sizing phase of validation

- **Purpose**: Component can calculate its ‘natural’ size based on content and layout rules.

- Implicitly invoked when component children change size. (Don’t call `measure()` on your children).

- Measurement occurs from the bottom up.

```xml
<mx:Application>
  <mx:HBox>
    <mx:Button />
  </mx:HBox>
</mx:Application>
```

- Don’t count on it: Framework optimizes away unnecessary calls to `measure()`.

The `measure()` function calculates and sets four properties:

- `measuredWidth, measuredHeight`
- `measuredMinWidth, measuredMinHeight`
measure()

Sizing phase of validation

- **Purpose**: Component can calculate its ‘natural’ size based on content and layout rules.

- Implicitly invoked when component children change size. (Don’t call `measure()` on your children).

- Measurement occurs from the bottom up.

```xml
<mx:Application>
  <mx:HBox>
    <mx:Button />
  </mx:HBox>
</mx:Application>
```

- Don’t count on it: Framework optimizes away unnecessary calls to `measure()`.

The `measure()` function calculates and sets four properties:

- `measuredWidth`, `measuredHeight`
- `measuredMinWidth`, `measuredMinHeight`
updateDisplayList()
updateDisplayList()

Layout and Drawing Phase of Invalidation

- **Purpose**: Lay out component contents and perform any drawing
updateDisplayList()

Layout and Drawing Phase of Invalidation

- **Purpose**: Lay out component contents and perform any drawing
- Layout and positioning occurs from the top down, given:

  ```xml
  <mx:Application>
      <mx:HBox>
          <mx:Button />
      </mx:HBox>
  </mx:Application>
  ```

- In `updateDisplayList()` you must set position and size of each child
  - If the child is a UIComponent, size it with `setActualSize()` and position it with `move()`
  - If the child is **not** a UIComponent, set the `x`, `y`, `width` and `height` properties.

- Good place to use Flash Player Drawing API
Style support is built into UIComponent
Styles

- Style support is built into UIComponent
- Add Style metadata so the style can be set inline as an attribute in MXML

```plaintext
/**
 * Text color of the label as the user presses it.
 * @default 0x000000
 */

[Style(name="textSelectedColor", type="uint", format="Color", inherit="yes")]
```
Styles

- Style support is built into UIComponent
- Add Style metadata so the style can be set inline as an attribute in MXML

```xml
/**
 * Text color of the label as the user presses it.
 *
 * @default 0x000000
 */
[Style(name="textSelectedColor", type="uint", format="Color", inherit="yes")]
```
Styles

- Style support is built into UIComponent
- Add Style metadata so the style can be set inline as an attribute in MXML

```xml
/**
 * Text color of the label as the user presses it.
 * @default 0x000000
 */
[Style(name="textSelectedColor", type="uint", format="Color", inherit="yes")]
```
Styles

- Style support is built into UIComponent
- Add Style metadata so the style can be set inline as an attribute in MXML

```xml
/**
 * Text color of the label as the user presses it.
 * @default 0x000000
 */
[Style(name="textSelectedColor", type="uint", format="Color", inherit="yes")]
```

- Set default initial value in user’s application or defaults.css
Styles

- Style support is built into UIComponent
- Add Style metadata so the style can be set inline as an attribute in MXML

```java
/**
 * Text color of the label as the user presses it.
 */
@default 0x000000

[Style(name="textSelectedColor", type="uint", format="Color", inherit="yes")]
```
- Set default initial value in user’s application or defaults.css
- Override `styleChanged()` to codify how the component should react when a style has changed.
  - Check which style has changed and call the right invalidation method
Styles

- Style support is built into UIComponent
- Add Style metadata so the style can be set inline as an attribute in MXML

```xml
/**
 * Text color of the label as the user presses it.
 * @default 0x000000
 */
[Style(name="textSelectedColor", type="uint", format="Color", inherit="yes")]
```

- Set default initial value in user’s application or defaults.css
- Override `styleChanged()` to codify how the component should react when a style has changed.
  - Check which style has changed and call the right invalidation method

If it has more to do with presentation it should be a style.
If you have a constant related to presentation, it should be a style.
Events in Flex

- Components dispatch and listen to events
Events in Flex

- Components dispatch and listen to events
- Add Event metadata to allow event to be set inline as an attribute in MXML

```xml
/**
 * Dispatched when the video is loaded and ready to play.
 *
 * @eventType mx.events.VideoEvent.READY
 */
[Event(name="ready", type="mx.events.VideoEvent")]
```
Events in Flex

- Components dispatch and listen to events
- Add Event metadata to allow event to be set inline as an attribute in MXML

```*/

* Dispatched when the video is loaded and ready to play.
* @eventType mx.events.VideoEvent.REY
*/

[Event(name="ready", type="mx.events.VideoEvent")]
```
Events in Flex

- Components dispatch and listen to events
- Add Event metadata to allow event to be set inline as an attribute in MXML

```
/**
 * Dispatched when the video is loaded and ready to play.
 * @eventType mx.events.VideoEvent.REady
 */
[Event(name="ready", type="mx.events.VideoEvent")]
```

- Use custom Event classes
  - Define static constants to enable compile-time checking of event types.
  - If you dispatch the same event as a base class, you must use the same event class.
Events in Flex

- Components dispatch and listen to events
- Add Event metadata to allow event to be set inline as an attribute in MXML

```
/**
 * Dispatched when the video is loaded and ready to play.
 */
@eventType mx.events.VideoEvent.READY

[Event(name="ready", type="mx.events.VideoEvent")]
```

- Use custom Event classes
  - Define static constants to enable compile-time checking of event types.
  - If you dispatch the same event as a base class, you must use the same event class.
Events in Flex

- Components dispatch and listen to events
- Add Event metadata to allow event to be set inline as an attribute in MXML

```xml
/**
 * Dispatched when the video is loaded and ready to play.
 */
@eventType mx.events.VideoEvent.READY
[Event(name="ready", type="mx.events.VideoEvent")]
```

- Use custom Event classes
  - Define static constants to enable compile-time checking of event types.
  - If you dispatch the same event as a base class, you must use the same event class.

Tip: Chose descriptive event names.
Phase 3: Destruction

- Construction
- Configuration
- Attachment
- Initialization
- Invalidation
- Validation
- Detachment
- Garbage Collection
Phase 3: Destruction

Destroying the component

- Detachment: Remove the component from the display list
Phase 3: Destruction

Destroying the component

- Detachment: Remove the component from the display list
- Components do not get validated or drawn when off the display list
- Once off the display list:
  - You can re-parent the component, and it will be brought back to life.
  - Re-parenting is cheaper then re-instantiating a component
Phase 3: Destruction

Destroying the component

- Detachment: Remove the component from the display list
- Components do not get validated or drawn when off the display list
- Once off the display list:
  - You can re-parent the component, and it will be brought back to life.
  - Re-parenting is cheaper then re-instantiating a component
- Garbage Collection
  - No active references can be tough
  - Common culprits include event listeners, dictionaries and timers.
DESIGNERS CAN HELP
How Designers and Developers work together Today

- Limited roles
  Designers create the visual design
  Developers add interactivity and logic

- Linear process from designer to developer
  Designers have one shot to get the visual assets right before throwing them over the wall

- Defining interactivity is challenging for designers
  Interaction design is out of the reach of designers today since it requires coding

Interaction design and the “soul” of the project typically lies with the Developer
How Designers and Developers work together Today

Designer

Visual Comp and Assets/Vignettes

Developer

Interaction and Motion definition
Component selection and structure
Application logic
Database design and data access
Testing and optimization
Integrating late breaking design revisions
Deployment
Examples of RIA design approaches today:

- Styling
- Skinning
- Interaction Design
Programatic Styling:

Tweaking properties - layout, button size, font color, background

Designer’s involvement minimal to none

“Soul” of the project lies with the developer
Programatic Styling: “It’s an app”
Graphical Skinning

Developer typically builds the app first then throws it over to the wall to a designer

“Here - make this pretty for me”

Component “Skins” designed in Photoshop, Illustrator, Fireworks...
Skinning: "It’s a good looking app"

Very linear process

Interaction defined mostly by developer

“Soul” of the project still lies with the developer
Skinning: “It’s a good looking app”

Very linear process

Interaction defined mostly by developer

“Soul” of the project still lies with the developer
Interaction Design - What makes a site/experience Great?

● Interaction is a design discipline - not a development task!
  How the user interface responds
  How objects appear and disappear
  Guiding a user through the whole experience with natural motion and feedback

● Why is interaction design important?
  Improves usability
  Increases customer satisfaction
  Creates a positive emotional response
  Increases usage and loyalty
The Interaction Design Process
- How a great site is created today

- Wireframe diagram
- User studies

Wireframe example courtesy of Adaptive Path and Soundflavor
The Interaction Design Process
- How a great site is created today

- Visual Layout / Static Comp
- Client pitch
- User feedback
The Interaction Design Process
- How a great site is created today

- Click-through / Prototype using timeline/animation tools
- User studies - Client sign-off
The Interaction Design Process
- How a great site is created today

- Throw all this stuff at a developer
- Developer tries to recreate the designer’s intentions using components
- Developer cuts up designs - extracting bits and pieces of the visuals
- Developer takes a stab at programatically defining interactions and behavior
The Interaction Design Process
- How a great site is created today

Developer
The Interaction Design Process - How a great site is created today

- Developer generates a proposed solution
- Hopefully there is iteration and refinement after this point - but it’s a painful process
- Designers ask developers to make late breaking visual changes. (fold in new visuals, enhance interactions, behaviors, motion, etc)
Challenges with current workflow

- Need to start over at each stage
  - Wireframes done in drawing tools or flowchart tools - Thrown Away
  - Visual (static) comps done in image editors - Thrown Away
  - Prototypes created in animation / timeline tools - Thrown Away
  - Developer tries to recreate designer’s intentions through code.

- Not the best experience
  - Designers are limited to the visual design
  - Interactions are defined in development tools (code)
  - Designers typically don’t get to provide much input on the interaction design

- Limited iterative design
  - Not easy to iterate though the creation process
  - Not easy to update designs after development stage begins
Where we want to go...

Let designers design the entire interface, not just component skins.

Enable designers and developers to collaborate on the creation of rich interactive experiences.
Adobe Flash Catalyst

A professional design tool for creating next-generation user interfaces and rich interactive experiences
GUMBO
Introducing Gumbo

Designer/Developer Workflow

- Spark Components and Architecture
- Graphic Primitives, Text, and FXG
- Revamped States and Transitions
- Advanced Layout

Framework Evolution

- FTE/TLF based Text
- 2-way binding
- Advanced CSS

Developer Productivity

- Compiler Performance
- Data Centric Tooling
- Client side data management
Catalyst – Components as Art

- Catalyst helps designers easily convert artwork into components
Code

Data

Logic
Designer/Developer

Code

- Data
- Logic

Markup

- Visuals
- States and Animation
Designer/Developer

Code
- Data
- Logic
- Behavior

Markup
- Visuals
- States and Animation
Designer/Developer

**Code**
- Data
- Logic
- Behavior

No assumptions about appearance

**Markup**
- Visuals
- States and Animation
- Layout
No assumptions about appearance
No assumptions about appearance

No need to dig into code
Halo Components

Code

Component
Logic
Layout
Data
Visuals
code

Markup

styles
Spark Architecture

Code

FxComponent
(extends UIComponent)

Markup
Spark Architecture

Code

FxComponent (extends UIComponent)

(inheritance)

FxComponent

Data and Behavior
No assumptions about appearance

Markup
Spark Architecture

Code

FxComponent (extends UIComponent)

(inheritance)

FxComponent

Data and Behavior
No assumptions about appearance

Markup

Skin
Spark Architecture

**Code**

FxComponent (extends UIComponent)

Data and Behavior
No assumptions about appearance

**Markup**

Skin

(inheritance)

Appearance
- Graphics
- Layout
- Animation
- Parts
- states
Spark Architecture

Code

FxComponent (extends UIComponent)

(inheritance)

FxComponent

Data and Behavior

No assumptions about appearance

Markup

Skin

(inheritance)

Skin

Appearance

- Graphics
- Layout
- Animation
- Parts
- states

Copyright 2008 Adobe Systems Incorporated. All rights reserved.
Spark Architecture

Data and Behavior
No assumptions about appearance

FxComponent
(extends UIComponent)

Designer/Developer Contract
(CSS)

Appearance
- Graphics
- Layout
- Animation
- Parts
- states

Skin

(inheritance)
## Designer/Developer Contract

### Defined by the Component

- **Data**
  ```
  [Bindable]
  public var title:String
  ```

### Defined by the Skin

- ```
  text="{fxComponent.title}"
  ```
  ```
  <FxButton id="upButton" />
  ```
  ```
  <State name="up" />
  ```
**Designer/Developer Contract**

- **Data**

  - [Bindable]
    ```java
    public var title:String
    ```

  - [SkinPart]
    ```java
    public var upButton:FxButton;
    ```

- **Defined by the Skin**

  - `text="{fxComponent.title}"`

  - `<FxButton id="upButton" />`

  - `<State name="up" />`
### Designer/Developer Contract

<table>
<thead>
<tr>
<th>Defined by the Component</th>
<th>Defined by the Skin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong></td>
<td></td>
</tr>
<tr>
<td>[Bindable]</td>
<td></td>
</tr>
<tr>
<td>public var title:String</td>
<td>text=&quot;{fxComponent.title}&quot;</td>
</tr>
<tr>
<td>[SkinPart]</td>
<td>&lt;FxButton id=&quot;upButton&quot; / &gt;</td>
</tr>
<tr>
<td>public var</td>
<td>&lt;State name=&quot;up&quot; /&gt;</td>
</tr>
<tr>
<td>upButton:FxButton;</td>
<td></td>
</tr>
</tbody>
</table>
**Designer/Developer Contract**

- **Data**
  
  ```
  [Bindable]
  public var title:String
  
  [SkinPart]
  public var upButton:FxButton;
  ```

- **Defined by the Component**

- **Defined by the Skin**

  ```
  text="\{fxComponent.title\}"

  <FxButton id="upButton" />

  <State name="up" />
  ```
### Designer/Developer Contract

#### Defined by the Component

**Data**

- `Bindable`

  ```
  public var title:String
  ```

**Parts**

- `SkinPart`

  ```
  public var 
      upButton: FxButton;
  ```

#### Defined by the Skin

- `text="{fxComponent.title}"`

  ```
  <FxButton id="upButton" />
  ```

- `<State name="up" />
  ```
### Designer/Developer Contract

<table>
<thead>
<tr>
<th>Defined by the Component</th>
<th>Defined by the Skin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong></td>
<td></td>
</tr>
<tr>
<td>[Bindable]</td>
<td>text=&quot;{fxComponent.title}&quot;</td>
</tr>
<tr>
<td>public var title:String</td>
<td></td>
</tr>
<tr>
<td><strong>Parts</strong></td>
<td>&lt;FxButton id=&quot;upButton&quot; /&gt;</td>
</tr>
<tr>
<td>[SkinPart]</td>
<td>&lt;State name=&quot;up&quot; /&gt;</td>
</tr>
<tr>
<td>public var</td>
<td></td>
</tr>
<tr>
<td>upButton:FxButton;</td>
<td></td>
</tr>
</tbody>
</table>
**Defined by the Component**

- **Data**
  
  ```
  [Bindable]
  public var title:String
  ```

- **Parts**
  
  ```
  [SkinPart]
  public var
      upButton: FxButton;
  ```

**Defined by the Skin**

- `text="{fxComponent.title}"`
- `<FxButton id="upButton" />`
- `<State name="up" />`
## Designer/Developer Contract

<table>
<thead>
<tr>
<th>Defined by the Component</th>
<th>Defined by the Skin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong></td>
<td></td>
</tr>
<tr>
<td>[Bindable]</td>
<td></td>
</tr>
<tr>
<td><code>public var title:String</code></td>
<td><code>text=&quot;{fxComponent.title}&quot;</code></td>
</tr>
<tr>
<td><strong>Parts</strong></td>
<td></td>
</tr>
<tr>
<td>[SkinPart]</td>
<td></td>
</tr>
<tr>
<td><code>public var upButton:FxButton;</code></td>
<td><code>&lt;/FxButton id=&quot;upButton&quot; /&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;/State name=&quot;up&quot; /&gt;</code></td>
</tr>
</tbody>
</table>
Designer/Developer Contract

Defined by the Component

- **Data**
  - [Bindable]
  - public var title:String

- **Parts**
  - [SkinPart]
  - public var upButton:FxButton;

- **States**
  - [SkinStates("up","over")]
  - public class Button {

Defined by the Skin

- text="{fxComponent.title}"

- <FxButton id="upButton" />

- <State name="up" />
Fx

Diagram of Fx components:

- FxComponent
  - FxBUTTON
  - Fx(H/V)Scrollbar
  - Fx(H/V)Slider
  - Fx NumericStepper
  - FxRadioButton
  - FxRadioGroup
  - FxSpinner
  - FxList
  - FxTextArea
  - FxTextInput

- FxScrollBar
- FxSlider
- FxApplication
- FxScroller
- FxTextinput
- FxTextArea
- FxTextInput
- FxList

(some classes not shown for clarity)
Compatibility is a priority

Our goal:
- Flex 3 applications continue to work seamlessly in Gumbo
- Spark components can be incrementally added to existing applications
- Halo based components can be dropped into new Spark applications

(some classes not shown for clarity)
Design in MXML?
Design in MXML?

...But find this instead

```xml
<Canvas>
  <Label text="name:" />
  <Button />
  <DataGrid />
</Canvas>
```
Design in MXML?

- Designer comes to Flex looking for this:
  ```xml
  <Canvas>
    <Label text="name:" />
    <Button />
    <DataGrid />
  </Canvas>
  ```

- Designers need freedom to design

- Mxml needs support for:
  - Primitives graphics.
  - Flexible, Expressive Layout
  - Rich animation and States
Design in MXML?

- Designer comes to Flex looking for this:
  ```xml
  <Canvas>
    <Label text="name:" />
    <Button />
    <DataGrid />
  </Canvas>
  ```

- Designers need freedom to design

- Mxml needs support for:
  - Primitives graphics.
  - Flexible, Expressive Layout
  - Rich animation and States

Flex UI Framework
Core Media Primitives
Flash / AIR runtime
<Graphic>
  <Path data="..
  
  <fill>
    <LinearGradient angle="90">..</LinearGradient>
  </fill>
  </Path>
  <Path blendMode="screen" data="...
  
  <fill>
    <LinearGradient angle="45">...
    </LinearGradient>
  </fill>
  </Path>
  <GraphicText text="MXML Graphics">
    <filters>
      <Glow color="#00FF00" strength="3" />
    </filters>
  </GraphicText>
</Graphic>
MXML For Graphics

- **MXML Graphics library providing rich primitive**
  - Simple Shape primitives
    (Rectangles, rounded rects, ellipses, circles)
  - Complex Paths
    (Linear, Quadratic, and Bezier curve segments)
  - Full range of fills and strokes
    (solid, transparent, bitmap, linear and radial gradients)
  - Masking, filters, blend modes, and more.
    (blur, glow, dropshadow, screen, multiply…)
  - Color and 2D transformations
    (rotate, scale, tint, brighten…)

```
<Graphic>
  <Path data=" ..... ">
    <fill>
      <LinearGradient angle="90">....</LinearGradient>
    </fill>
  </Path>
  <Path blendMode="screen" data="...">
    <fill>
      <LinearGradient angle="45">...
    </fill>
  </Path>
<GraphicText text="MXXML Graphics">
  <filters>
    <Glow color="#00FF00" strength="3" /> 
  </filters>
```

- **MXML Graphics library providing rich primitive**
  - **Simple Shape primitives**
    (Rectangles, rounded rects, ellipses, circles)
  - **Complex Paths**
    (Linear, Quadratic, and Bezier curve segments)
  - **Full range of fills and strokes**
    (solid, transparent, bitmap, linear and radial gradients)
  - **Masking, filters, blend modes, and more.**
    (blur, glow, dropshadow, screen, multiply…)
  - **Color and 2D transformations**
    (rotate, scale, tint, brighten…)
  - **Integrated text, bitmaps**

```xml
<Graphic>
  <Path data=".....">
    <fill>
      <LinearGradient angle="90">....
      </LinearGradient>
    </fill>
  </Path>
  <Path blendMode="screen" data="...">
    <fill>
      <LinearGradient angle="45">...
      </LinearGradient>
    </fill>
  </Path>
  <GraphicText text="MXML Graphics">
    <filters>
      <Glow color="#00FF00" strength="3" />
    </filters>
  </GraphicText>
</Graphic>
```
GraphicElement

Rect

Ellipse

Path

Line

BitmapGraphic
**1<sup>st</sup> class citizens**
- Anything that can be set can be changed
- Easy to animate with effects, states, transitions, code
- Freely mix and match with Fx components

**GraphicElement**
- Focused on performance
- optimized for fast rendering, low overhead
- One DisplayObject shared by many GraphicElements
FXG: graphics format based on MXML

- Supported by Design tools that know nothing about actionscript
- Closely matches the flash player rendering model.
- Static – no binding or layout, event handlers, script, or styling (means it can be optimized by the compiler)
- Supported by Illustrator, Photoshop, and Fireworks CS4
- Compiled by MXMLC``
Enhanced Flash Text
Enhanced Flash Text

- FTE: New low level text engine (player 10)
- TLF: New text layout library built on top of FTE.
Enhanced Flash Text

- **FTE:** New low level text engine (player 10)
- **TLF:** New text layout library built on top of FTE.

**Benefits:**
- Soft hyphens
- Baseline control (e.g., superscripts and subscripts)
- Right, center, and decimal tabs
- Vertical justification
- Multiple columns
- Ligatures, capitalization styles, digit styles
- *Integrated Rendering of device fonts* (a.k.a. *I can fade and rotate my text*)
- *Bi-Directional text*
Gumbo Text

- Gumbo Introduces 3 new ‘Primitive’Text components:
  - TextBox (Label)
  - TextGraphic (Text)
  - TextView (editable TextField)

- 2 new skinnable text Fx components (based on the primitives):
  - FxTextInput (TextInput)
  - FxTextArea (TextArea)
<states>
  <State name="login">
    <SetProperty target="{goBtn}" name="label" value="…” />
  </State>
  <State name="register">
    <SetProperty target="{goBtn}" name="label" value="…” />
    <AddChild target="grp">
      <FxCheckbox label="Agree to terms" />
    </AddChild>
  </State>
</states>

<Group id="grp">
  <TextBox text="username:" />
  <FxTextInput />
  <TextBox text="password:" />
  <FxTextInput />
  <FxButton label="new user?" />
  <FxButton id="goBtn"
</Group>
States

- Foundation feature of Spark
- But they’re too hard to use
  - Verbose ‘mx:add, mx:set’ syntax destroys readability of MXML
  - Obtuse state hierarchy difficult to manage, difficult to factor, difficult to tool for
  - Explicit object/binding based syntax prevents compiler optimization

```
<states>
  <State name="login">
    <SetProperty target="{goBtn}" name="label" value="…"/>
  </State>
  <State name="register">
    <SetProperty target="{goBtn}" name="label" value="…"/>
    <AddChild target="grp">
      <FxCheckbox label="Agree to terms"/>
    </AddChild>
  </State>
</states>

<Group id="grp">
  <TextBox text="username:" />
  <FxTextInput />
  <TextBox text="password:" />
  <FxTextInput />
  <FxButton label="new user?" />
  <FxButton id="goBtn"
</Group>
```
States, version 2

- Declare your states with a ‘State’ tag
- Describe ‘alternate views’ of your markup
- Change values, bindings, event handlers
- Include and exclude components as easily as setting visibility
- Unscoped entries specify the ‘default’ for all states

```xml
<states>
  <State name="login"/>
  <State name="register"/>
</states>

<Group>
  <TextBox text="username:" />
  <FxTextInput />
  <TextBox text="password:" />
  <FxTextInput />
  <FxButton label="new user?" />
  <FxCheckbox includeIn="register" label="agree to terms" />
  <FxButton label="log in" label.register="sign up" />
</Group>
```
new language features aimed at productivity and usability

Incompatible changes, but committed to compatibility.

Result: MXML 2009

- New language features only available in 2009
- One language namespace per file, but can mix and match per application
- Gumbo and Halo live side by side in 2009
- Gumbo is only intended to be used in 2009 – won’t generally work in 2006

xmlns="http://ns.adobe.com/mxml/2009"
Changes in 2009
Changes in 2009

- `<Declarations />`
  - A place to stick your stuff
  - Anything that’s not a visual child of a component belongs here.
  - Services, Data Models, singletons…anything that doesn’t show up in your UI ends up here
  - Previously ‘stuff’ could just go at the top of the file

- New States syntax
- New Components
- A couple of other minor details…

```xml
<FxApplication>
  <Declarations>
    <HTTPService ... />
  </Declarations>
  <FxTextInput ... />
  ...
</FxApplication>
```
Gumbo Animation

- Animating arbitrary objects
- Animating arbitrary types
- API improvements
  - Animation object
  - API changes
  - New functionality
- More built-in Effects
- More intelligent Effects
Gumbo Animation: Animating Arbitrary Objects

- Flex 3 effects animate targets in various ways
  - Move calls target.move(x, y)
  - Resize sets target.width/target.height
  - Rotate sets target.rotation and called target.move(x, y)
  - Difficult to generalize effects for different target types

- Gumbo effects animate named properties on targets
  - FxMove sets target[“x”], target[“y”]
  - FxResize sets target[“width”], target[“height”]
  - FxRotate sets target[“rotation”], target[“x”], target[“y”]
  - FxAnimate superclass can animate any named property/style
    - To move ‘button’ from (0,0) to (100, 200):

      ```xml
      <FxAnimate id="mover" target="{button}">
        <PropertyValuesHolder property="x" values="{[0, 100]}"/>
        <PropertyValuesHolder property="y" values="{[0, 200]}"/>
      </FxAnimate>
      ```
Gumbo Animation: Animating Arbitrary Types

- Flex 3 effects only understand **Number**
  - Works for most situations (x, y, width, height, rotation, alpha, etc.)
  - But not for all (colors, arbitrary objects)
- Gumbo effects work with arbitrary types instead
  - Default is still Number
  - Effects can supply their own type interpolator
    - FxAnimateColor supplies a ColorInterpolator
  - Developers can supply their own interpolator to the effects as well

```java
public class FooInterpolator {
    public function interpolate(start:*, end:*, fraction:Number):void {
        var val:Foo;
        // Code to interpolate Foo type between start and end, given
        // the elapsed fraction
        return val;
    }
}
```
Gumbo Animation: API Improvements: Animation

- **Animation** is the new **Tween**
- Low-level timing engine that Gumbo effects use
- Can also use Animation directly
- Takes animation attributes, start/end values, updates listeners with animated values

```javascript
// Moves object foo from x=0 to x=100 over a half-second
var anim:Animation = new Animation(0, 100, 500);
anim.addEventListener(AnimationEvent.ANIMATION_UPDATE, updater);
anim.play();

private function updater(event:AnimationEvent):void {
    foo.x = event.value;
}
```
Gumbo Animation: API Changes (since Tween)

- No more auto-play on construction
  - Can pre-construct and call play() when needed
- Only event listeners
  - No more callback handlers and implicit Tween updateHandler/endHandler functions
- Events for start/end/update/repeat
- New IInterpolator and IEaser interfaces
  - Arbitrary types, parameterized easing, easy API for custom implementations

```javascript
anim.easer = new Sine(.7); // Sine, ease in for first 70%
anim.easer = new Power(.4, 3); // Cubic, ease-in for first 40%
anim.easer = new Power(1, 4); // Quartic, ease-in over entirety

public class ReverseEase implements IEaser {
    public function ease(fraction:Number):Number {
        return (1 - fraction);
    }
}
```
Gumbo Animation: API Functionality

- Repetition, including new ‘Reverse’ behavior
- Single timing pulse for all timing events
  - Easier to synchronize multiple effects with delays

```javascript
// 'bouncing' behavior
var anim:Animation = new Animation(0, 300, 1000);
anim.easer = new Power(1, 3); // ease in all the way
anim.repeatBehavior = Animation.REVERSE; // reverse every time
anim.repeatCount = 0; // forever
anim.play();

// animation sequencing
var anim:Animation = new Animation(100, 200, 500);
var anim1:Animation = new Animation(200, 300, 1000);
// anim1 will start in same timing pulse as anim ends
anim1.startDelay = 500;
anim.play();
anim1.play();
```
Gumbo Animation: New Effects

- **FxAnimateColor**
  - auto-tween a color between two colors over time

- **FxAnimateFilter**
  - animate a Pixel Bender shader or other filter over time

- **FxAnimateRotate3D**
  - animate 3D transform properties

- **CallAction**
  - call arbitrary function, useful for sequencing with other effects

- **SetAction**
  - replaces SetPropertyAction/SetStyleAction

- **Pause**
  - now waits for timeout or arbitrary events

- (and we're not done yet...)
Gumbo Animation: More Intelligent Effects

- **Fade-without-Remove**

- In Flex 3, had to use RemoveAction to fade objects in and out of existence:

  ```xml
  <Transition fromState="A" toState="B">
    <Sequence target="{button}" />
    <Fade/>
    <RemoveAction/>
  </Sequence>
</Transition>

- Gumbo takes care of this for you:

  ```xml
  <Transition fromState="A" toState="B">
    <Fade target="{button}" />
  </Transition>
  ```
Gumbo Effects: The Helpful Diagram

Application Code

Gumbo Effect

Animation

Timer

Target

- target, duration
- start/endValue
- property value
Gumbo Effects: The Helpful Diagram

Application Code -> Transition

[optional params]

Gumbo Effect

Animation

Timer

property value -> Target
Animation Demos

- **Coolerizer**
  - Arbitrary type interpolation
  - New effects
  - New Animation capabilities
  - Nifty colors

- **EasingTest**
  - Various animation controls
  - New easing functionality

- **DynamicMenus**
  - Reversing transitions
Containment and Layout in MXML

- mx.core.Container
  - Based on DisplayObject containment, can’t support GraphicElements
  - Too much rarely used functionality (scrollbarPolicy="off", anyone?)
  - Rigid layout rules don’t give design flexibility
  - No support for low level flash features like transforms and 3D effects.
Containment

- **Group**
  - Basis of the containment model in Gumbo
  - Supports both UIComponents and GraphicElements
  - Like Container, but with all the rarely used functionality stripped away
    - Containment
    - Layout (through delegation)
    - Transforms, masks, blendModes, clipping, and other flash player goodness
Containment

- Group
  - Basis of the containment model in Gumbo

- DataGroup
  - The core guts of a list component
    - A data-driven group with itemRenderers
  - What would happen if married a list to a container
  - ‘Repeater’ done right
Containment

- **Group**
  - Basis of the containment model in Gumbo

- **DataGroup**
  - The core guts of a list component
    - A data-driven group with itemRenderers
  - What would happen if married a list to a container
  - ‘Repeater’ done right

In Spark, Everything you see on screen eventually lives inside a Group
All layout in Spark extends LayoutBase

- Plugs into Group and LayoutGroup
- Basis of layout for containers, lists, skins.
- Separate objects guarantee clean interface, easy extension, and consistent behavior
- Handles both UIComponents and GraphicElements
- Full transform support
- 3D effect support
LIST AS EXAMPLE
VideoWidget → FxVideoWidget
Defining Patterns in Halo

- Invalidation/Validation Model
- Event Driven Interaction Model
- Composition
Defining Patterns in Halo

- Invalidation/Validation Model
- Event Driven Interaction Model
- Composition

Defining Patterns in Gumbo

- Clean separation of component logic from its visuals
Component Architecture Patterns Review

Defining Patterns in Halo

- Invalidation/Validation Model
- Event Driven Interaction Model
- Composition

Defining Patterns in Gumbo

- Clean separation of component logic from its visuals
  - Spark Skinning Architecture
Component Architecture Patterns Review

Defining Patterns in Halo

- Invalidation/Validation Model
- Event Driven Interaction Model
- Composition

Defining Patterns in Gumbo

- Clean separation of component logic from its visuals
  - Spark Skinning Architecture
- Component functionality can be compositied together to build up or pare down.
- Designer/Developer contract maintained by data, parts and states
  - Component makes no assumptions about appearance
Component Architecture Patterns Review

Defining Patterns in Halo

- Invalidation/Validation Model
- Event Driven Interaction Model
- Composition

Defining Patterns in Gumbo

- Clean separation of component logic from its visuals
  - Spark Skinning Architecture
- Component functionality can be composited together to build up or pare down.
- Designer/Developer contract maintained by data, parts and states
  - Component makes no assumptions about appearance
  - Skin doesn’t require digging into code
Time to Gumbo-cise!
Time to Gumbo-cise!

Step 1: Sit down and think

- What is the core essence of the component?
Time to Gumbo-cise!

Step 1: Sit down and think
- What is the core essence of the component?

Step 2: Identify skin parts
- Which are required, which are optional?
Time to Gumbo-cise!

Step 1: Sit down and think
- What is the core essence of the component?

Step 2: Identify skin parts
- Which are required, which are optional?

Step 3: Add in Spark-specific code
- Chose a proper base-class
- Implement `partAdded()`, `partRemoved()`
- Decide on the proper set of states
- Author a skin that includes skin parts.
  - Skins are associated with components through CSS
  - Skins use MXML Graphics
Time to Gumbo-cise!

Step 1: Sit down and think

- What is the core essence of the component?

Step 2: Identify skin parts

- Which are required, which are optional?

Step 3: Add in Spark-specific code

- Chose a proper base-class
  - Implement `partAdded()`, `partRemoved()`
- Decide on the proper set of states
- Author a skin that includes skin parts.
  - Skins are associated with components through CSS
  - Skins use MXML Graphics

Step 4: Yank out Halo-specific code

- Rip out `createChildren()`, `measure()` and `updateDisplayList()` *
Beyond Designer/Developer

PRODUCTIVITY AND FRAMEWORK EVOLUTION
Productivity Enhancements
Productivity Enhancements

- 2-way binding
  - `<FxTextInput text="@{data.firstName}"/>

- Enhanced CSS support
  - Existing Flex support is simple:
Productivity Enhancements

- 2-way binding
  - `<FxTextInput text="@{data.firstName}" />

- Enhanced CSS support
  - Existing Flex support is simple:
    - Type selectors: `Button {}`
    - Class selectors: `.myButton {}
  - Gumbo adds:
    - Multiple class selectors `<Button styleName="default subtle" />`
    - Id selectors `#okButton {}`
Productivity Enhancements

- 2-way binding
  - `<FxTextInput text="@{data.firstName}" />

- Enhanced CSS support
  - Existing Flex support is simple:
    - Type selectors: `Button {}`
    - Class selectors: `.myButton {}`
  - Gumbo adds:
    - Multiple class selectors `<Button styleName="default subtle"/>
    - Id selectors `#okButton {}`
    - Descendent selectors `#detailsPane Button {}`
Productivity Enhancements

- **2-way binding**
  - `<FxTextInput text="@{data.firstName}"/>

- **Enhanced CSS support**
  - Existing Flex support is simple:
    - Type selectors: `Button {}`
    - Class selectors: `.myButton {}`
  - Gumbo adds:
    - Multiple class selectors `<Button styleName="default subtle"/>
    - Id selectors `#okButton {}`
    - Descendent selectors `#detailsPane Button {}`
    - Any combination of the above: `VBox.dark Button#okButton {}`
Compiler performance

- Many, many man hours spent on improvement compiler speed in Gumbo
- Some examples:

<table>
<thead>
<tr>
<th>Representative Optimizations</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct AST Generation</td>
<td>Medium</td>
</tr>
<tr>
<td>Turned off POSTLINK when debug is turned ON</td>
<td>HIGH</td>
</tr>
<tr>
<td>Asc InputBuffer/OffsetInputBuffer/Scanner simplified</td>
<td>Medium</td>
</tr>
<tr>
<td>Asc front end performance improvements &amp; bug fixes</td>
<td>Low</td>
</tr>
<tr>
<td>Changes in the compiler for switching StringBuffer to StringBuilder</td>
<td>Low</td>
</tr>
</tbody>
</table>

- These are representative improvements – your mileage will definitely vary.
- Large apps doing incremental debug compiles can expect roughly 2X speedup.
- We’ve seen as much as 5X for very large apps!
- Optimization work is ongoing.
Data-centric Development

- Key Gumbo Theme: Improve your productivity working with services and data
  - Tooling support: reduce the amount of boiler-plate code you write to get data from a service down onto the screen.
  - Client Data Management: Flex data management features for your own RPC based services
More Information

- Tuesday afternoon, Filthy Rich Clients: Beyond Java
- Wednesday Quickie at 13:35, Flex and JSF
- Wednesday 14:00, Future of Rich Internet Applications
- Friday 09:30, ActionScript for Java Developers

- [http://tv.adobe.com](http://tv.adobe.com)
- [http://opensource.adobe.com/flex](http://opensource.adobe.com/flex)
- [http://flex.org](http://flex.org)
THE END