

Understanding the Flex 3 Component and Framework Lifecycle

An in-depth look at understanding and building better Adobe Flex® and Adobe AIR™ applications and components

By James Polanco and Aaron Pedersen

Table of Contents

Table of Contents	2
Introduction	4
How To Read This Paper	5
A Brief History of Flex	5
It's All About the Frames	5
<i>The Elastic Racetrack</i>	6
<i>Managing the Racetrack</i>	7
The Life Stages of a Flex Application	8
An Introduction Into the Component Lifecycle	8
<i>Component Phases: Overview</i>	8
<i>Component Phases: Construction (birth)</i>	9
<i>Component Phases: Addition (birth)</i>	10
<i>Component Phases: Initialization (birth)</i>	11
<i>Component Phases: Invalidation (growth / maturity)</i>	12
<i>The Invalidation and Validation Cycle</i>	12
<i>The Three Types Of Invalidation</i>	15
<i>Component Phases: Validation (growth maturity)</i>	16
<i>Component Phases: Update (maturity)</i>	18
<i>Component Phases: Removal (death)</i>	18
A Flex Application Is Born	19
Flex Application Phases: Construction	20
<i>The Dark Art of the Flex Compiler</i>	20
<i>At the Top Sits the SystemManager</i>	21
Flex Application Phases: Initialization	23
<i>Managing Externalization</i>	23
Flex Application Phases: Preloading	24
Flex Application Phases: Child Creation	25
Flex Application Phase: Child Display.....	26
Flex Application Phase: Destruction.....	27
Flex Component Development Best Practices	27
Using Construction	27
<i>JIT and the Constructor</i>	28
Using Initialization	29
<i>If You Must Override Initialize</i>	30
The Invalidation-Validation Cycle and Methods	30
<i>Separating the Phases</i>	30
<i>Using Dirty Flags</i>	32
<i>Implementing Validation Methods</i>	32
Using And Accessing Styles	34
<i>Applying Styles</i>	34

Conclusion 36
The Future of Flex Components 36

Introduction

The Adobe Flex Framework SDK has an aura of voodoo around it that is partially created by the power of a well-designed semi-black box system and the fact that in most cases we, as developers, don't have the time or energy to really dive into the unknown during our project cycles. Technically, the Flex Framework is not a black box, you can read and view all of the source¹, but due to the complexity of the code and how it's designed we tend to treat the framework as input in, functionality out. Most developers, the authors included, tend to learn Flex on the job, finding new tricks and techniques via implementation, experimentation, research, fellow developers, blogs, lists, and when all else fails; documentation.

Adobe has done a pretty amazing job of documenting the Flex Framework, which has been broken down into two main categories: User Guides and API (ASDoc) documentation. Yet, even with their extensive documentation there is a large gap between the User Guides and the API documentation. The User Guides cover a wide range of topics, from getting started to relatively complex functionality, but stop at the more nitty-gritty level functionality such as deep dives into Style propagation techniques, metadata do's and don'ts and of course the Flex Framework and Component lifecycle. There are high-level overviews but beyond that it's up to the user to figure it out.

We can infer a lot of functionality and order from the API documentation, sometimes the commenter guides us, other times they tease us with a bit of information but its our task to track down and determine the overarching ecosystem. Often, the API docs assume you know what you looking for and only explain exactly what the method does, not when or why you should use it. Because of this gap between macro-focused user guides and the micro-focused API, optimized development becomes a dark art that requires developers' years of experience through trial and error to tease out and define best practices. Augment this with the fact that Flex is only five or so years old, which has evolved dramatically since its initial incarnation, we are still very much in the technologies infancy.

Yet we, as 3rd party developers, are not the only one's left on the outside. Often Adobe Engineers do not have full insight into all the Framework's minuet details. James attended Deepa Subramaniam's presentation on Flex Components at the Adobe MAX 2008 conference and Deepa commented about the Flex Lifecycle. Her presentation was half dedicated to the lifecycle, and even then half of that half was focused on the current Flex 3 lifecycle and the rest was dedicated to Flex 4's changes. During her in-depth, yet still high-level, overview she mentioned that recently one of the Flex Architects finally sat her down and walked her through the

¹ This does not include base Flash Player components (Sprite, MovieClip, etc.), which are stored inside the `playerglobal.swc` file. The current implementation of SWC prevents access to the source code, therefore parts of the Framework's implementation hierarchy is inaccessible to developers.

entire lifecycle. Her comment to the audience was that she wished she had known this information sooner because it helped her become a better Flex engineer.

Our point is that Flex is still voodoo for a lot of us, even some of the most highly regarded engineers. It's a complex system designed to be very powerful and, excuse the pun, flexible. Yet, this flexibility creates many possibilities for extension and implementation. Many of those possibilities are not the best way to develop and even have the potential to be detrimental to overall performance, stability and future scalability of the application.

The goal of this paper is to try and shine some light onto the entire lifecycle so that we, as a community of developers, can create better applications and components within Flex. One word of caution, much of the following information is inferred from reading the source code, some of its well documented, some of its not. If you see something that isn't quite right or maybe could be done in a better way, feel free to contact us at DevelopmentArc (info@developmentarc.com) so that we can append/update this document to make it as technically correct as possible.

How To Read This Paper

Throughout this document we refer to a lot of the Flex Framework code, but for brevity's sake, we do not always show the code we are referring to. When reading the paper we recommend that you have Flex Builder open, or access to the Flex 3 Framework source, to access the referenced code so that you can follow along as we discuss what the code is doing and why it is doing it.

If you do follow along in code, please be aware that we often skip over functionality or specific details of the code so that we may focus on the current topic at hand. This is to prevent us from diverging too far from the current topic just to explain every little nuance in the code. This is not to say that what the code is doing is not important, but often the code is handling special use cases, preventing potential issues or handling cases that occur later on in the lifecycle, which we may not have been discussed yet.

A Brief History of Flex

Before we jump head first into the minutia of the Flex Framework and how we can leverage it, we should step back and look at the overall big picture of Flex and why we use it. Most of the readers of this paper probably have experience with Flex, you may even have a LOT of experience, but to understand some of the decisions the Flex team made, we need to look at the history of Flex and the Flash Player. This is important because many of the paths the Flex Framework take are dependent upon the lower levels of the Flash Player.

It's All About the Frames

Fundamentally, Flex is Flash. At DevelopmentArc, when we sit down to teach new developers, or clients, about Flex we have to keep re-iterating this point. At the end of the day Flex generates a SWF file just like Adobe Flash Professional does. This

means that Flex is bound to the same set of rules and realities of any Flash SWF. The one fundamental rule that all SWF files must conform to is that Flash is frame-based.

For those unfamiliar with the history of Flash, the initial goal of the player was to create animation. Traditional animation is cell based, you draw an image on the cell replace it with a new cell that has a slightly modified image and at a certain speed the changes look like motion. So, in place of cells, Flash uses frames to create the same effect for animations. Frames still exist and in fact all logic and screen rendering is bound to the frame change and the defined frame rate.

The frame rate tells Flash how many frames per second it should try to process; by default Flex is set to 24 frames per second. This number does not guarantee that Flash will execute 24 frames per second but it does guarantee that it won't do more than 24 frames per second. The real-time frame rate (how many frames per second Flash actually executes) is dependent on many different things, a few being: the complexity of code being executed, the amount of content that has changed causing it to be re-rendered on screen, and of course the performance ability of the machine hosting the Flash Player.

The Elastic Racetrack

Ted Patrick wrote an excellent post about the Flash frame execution order and what he has dubbed "The Elastic Racetrack"². It is an older article, targeted at Flash Player 7, but what he talks about is fundamentally true today with Flash Player 10.

The main aspect to take away from his post is that the Flash frame is split into two phases: the code execution phase and the screen re-draw/render phase. The code execution is a linear, non-threaded pass that has to execute all code requested for the current frame. During this pass, data is processed, calculations are made and any modifications to the Display List (DOM) are applied. Once all the code has been executed, and the display list has been updated, then the Flash Player renders the changes to the screen.

With the release of Flash Player 9 and ActionScript 3, the player team created a new ActionScript Virtual Machine (AVM2) to support the new language and feature set. They also re-designed the Flash DOM to support easier development within the new player. Sean Christmann, from EffectiveUI, took Ted Patrick's Elastic Racetrack concept and updated it to reflect the changes in the new virtual machine³.

² Read Ted Patrick's post at:

<http://www.onflex.org/ted/2005/07/flash-player-mental-model-elastic.php>

³ Read Sean Christmann's update to the Elastic Racetrack at:

<http://www.craftymind.com/2008/04/18/updated-elastic-racetrack-for-flash-9-and-avm2/>

Sean has augmented the racetrack by bringing in the concept of what he calls “The Marshal”. The core difference between the old virtual machine (AVM1) and AVM2 is that AVM1 worked solely on a split frame environment. The AVM2’s Marshal

is responsible for carving out time slices for the Flash Player to operate on. Its important to clarify up front that these time slices are not the same thing as the framerate compiled into a swf ... the player ultimately synthesizes a framerate from these slices

These slices are broken into multiple steps: player event dispatching, user code execution, pre-render event dispatching, user code execution (post pre-render), and finally player rendering. Sean explores how these steps are processed, when they are executed and how frame rate can modify which step(s) are executed and when.

We highly recommend reading both articles before continuing on with this document because we will consistently refer to frames and how the Flex Framework leverages this system. ⁴

Managing the Racetrack

The code execution and display rendering of content can take a lot of time to complete, and the more complex the code and/or the display changes are, the longer the frame may take to render. One of the most common undesirable developer experiences with the elastic racetrack is animation performance lagging as large amounts of data is being processed.

For example, on a client project DevelopmentArc created a loading animation that was using Flex’s Tween system. Running by itself the loading animation looked great, but as soon as we made our call to load server data, the animation became extremely choppy. The cause of this performance issue was that we were asking the player to parse a large amount of XML data coming back from the server and, at the same time, attempting to programmatically make an animation move across the screen.

What we had done was overload the players code side of the racetrack (frame), therefore causing the player to spend too much time calculating per cycle and not enough time rendering. This meant that the player couldn’t meet the frame rate required to display a smooth animation because most of the time was dedicated to calculation. Ted’s final comment from the racetrack post says it all:

As it has been said many times before, "Just wait a frame!"

This leads us back to the Flex Framework. Due to the shear complexity of the Flex Framework, imagine how it would perform it if had to handle nested layout

⁴ For simplicity we will consider a frame and slice to be interchangeable, even though on a technical level this may not always be true.

calculations (HBox, VBox), animation (Effects, Transitions), data loading (Remote Object, HTTPService), user interaction (mouse, keyboard), and your custom code all in a single pass, every frame. It would bring the player to its knees and Flex would be totally unusable.

Much of what the Framework does is “wait a frame” for us. Its architecture is designed in such a way to create phases and steps that are optimized to how the Flash player works, and does so in such a way that you can also take advantage of this if you adopt and tap into the provided (yet often unexplained) API hooks.

The Life Stages of a Flex Application

With any biological Lifecycle we typically have four stages: birth, growth, maturity and finally death. Just like the biological Lifecycle, the Flex framework and its components follow a very similar path. We will first examine the birthing process; how Components are created, how they grow and mature and finally what happens when its time to remove them from our application.

After reviewing components we will look at the Flex Application Framework’s lifecycle. We will review how the Flex framework starts up, ties into the Flash Player, and creates its Application and other children components. We will look at what happens when the application grows and matures as the user interacts with it. Finally we will look at the death of the application.

Understanding all these elements and the order they occur will help you design and build better components and potentially, applications. Once we have fully examined these stages we will then explore how to take advantage of this knowledge when creating new functionality for your application.

An Introduction Into the Component Lifecycle

Before we start dealing with how a Flex Application starts up and creates its children we should spend a bit of time looking at the general component lifecycle. Its important to understand the component lifecycle first because the Flex application lifecycle is an extension of the component lifecycle, with some important modifications to help performance and other required application level functionality.

Component Phases: Overview

A Flex component goes through seven distinct phases that we can compartmentalize into the four life stages. As the component goes through the seven main phases of its lifecycle, some phases are repeated often, others only occur once. The seven phases are construction, addition, initialization, invalidation, validation, updating and removal.

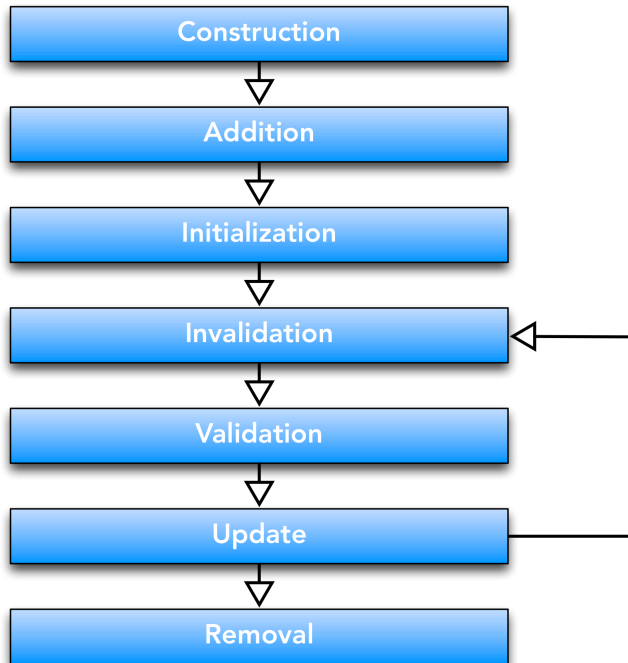


Figure 1 - The Phases of the Component Lifecycle

These are not the official Adobe terms, but we have organized the stages into categories to help define what the component is doing and when. Our seven stages correspond to the four life stages as follows. The Birth stage is made up of Construction, Addition, and Initialization. The growth and maturity phase are made up of Invalidation, Validation and Update. The death state is defined by the Removal phase. Let's look at the first phase: Construction.

Component Phases: Construction (birth)

The Construction phase is the very first phase of a component and is defined when we call the Component's constructor. The Adobe Flex documentation uses `Button` as their example component, so let's follow in their footsteps.

```
var myButton:Button = new Button(); // this is our construction phase
```

Pretty obvious, right? What isn't obvious about this phase is that very little actually happens. So what does happen during the Construction phase?

If we start at the `Button`'s constructor and walk up the `super()` chain, we first go to `UIComponent`, then to `FlexSprite`, then to `Sprite` (where we can no longer see the source code since its part of *playerglobal.swc*). `FlexSprite` is high enough up the inheritance chain to get a good overall picture of the construction process, so we will start there for this example.

Looking at `FlexSprite`'s constructor all it does is define the `name` property with a unique string value.⁵ Since that is all that happens in the `FlexSprite` constructor we can step down to `UIComponent` constructor. `UIComponent` does a bit more during construction, but not a lot. It sets up the focus management, starts tracking its own add and remove events (for delegation reasons), sets up listening for focus events, keyboard events, defines its reference to the Resource Manager (used for localization) and then stores a private version of the components current width and height that may have been defined on the super Class. Again, very little is done. No layout, no styles, no children creation, etc.

Finally, we get back to the `Button` Class. All `Button` does is register to the `MouseEvent`s to handle user interaction. That's it, that's all.

For more details about using the constructor in your own component development, we discuss best practices and recommendations in the section Using Construction, later in this document. Now that we are constructed, what starts the next phase?

Component Phases: Addition (birth)

The Addition phase occurs when you add the component to a parent component:

```
this.addChild(myButton); // the addition phase begins
```

Figure 2 - Add Child Example

The process of adding the component to the parent executes a huge amount of functionality on the parent and this is where the lifecycle really kicks into high gear. Let's look at `UIComponent`'s `addChild()` method and how it defines the Addition phase.

The `addChild()` method is broken down into three sub-method calls: `addingChild()`, `$addChild()` and `childAdded()`. The pattern of breaking phases down into sub-steps is a common one within the Flex Framework. By breaking complex processes into smaller sections helps enable better control over the entire process. We will delve more into this ability later on in the following lifecycle sections and also in the Flex Component Development Best Practices section.

When the `addChild()` method is called, it first executes the `addingChild()` method⁶. The `addingChild()` method does a lot of the heavy lifting for us, such as setting the

⁵ When you launch a Flex app and set a breakpoint to view the current display stack in the Flex debugger, you will see components referenced as "Button12" or "ComboBox4" in the Flex component hierarchy. This is the `name` value of the component. The `FlexSprite`'s constructor is where the `name` of the component is defined. Flex uses a utility called `NameUtil` that takes the class name and appends a unique counter value depending upon how many instances of the class name have been previously created. "Button12" would be the 12th button created in the application.

⁶ In `addChild()`, the method first checks to see if this addition is a re-parenting of the component, if so it removes the component from the previous parent first before attempting to add the child to the new parent. In our example, the `Button` was just created so no previous parent exists. The method sets the child's index position and then executes the `addingChild()` method.

child's parent reference, setting the child's document (parent `Application`) reference, defines the `LayoutManager` (this is very important for later phases), defines which fonts are available and starts the style propagation and style management for the component⁷.

Once `addChild()` is complete, the `addChild()` method calls `$addChild()`, which is a Flash Player level method that actually adds the component to the display list, enabling the player to draw the components graphical content to the screen during the render phase of the Flash Frame⁸.

The final sub-method called is `childAdded()` which determines if the child component has been initialized yet, and if not the method calls the `initialize()` method on the child, completing the Addition phase and starting the Initialization phase.

Component Phases: Initialization (birth)

The initialization phase is responsible for creating children of the component and preparing for the first Invalidation and Validation phase cycle. The `initialize()` method, which was called by the parent's `childAdded()`, is broken down into four sub-methods: `createChildren()`, `childrenCreated()`, `initializeAccessibility()` and `initializationComplete()`.

When the `initialize()` method is called, the `UIComponent` first dispatches the `FlexEvent.PREINITIALIZE` event and then calls `createChildren()` enabling any extended component to create and then add child components to itself⁹.

For example, the `Button` component overrides `UIComponent`'s `createChildren()` method to generate the `TextField` instance that renders the `Button`'s label text, sets the `styleName` of the `TextField` to the `Button` instance and then adds the `TextField` as a child of itself by calling `addChild()`.¹⁰

⁷ Once that the component has been added to the parent, a lot of valuable data is available to us as developers. An interesting aspect to note is that this is the first time that all the style inheritance information is defined and all the CSS declarations are applied. For more information details about style management in the component Lifecycle, refer to the section [Using And Accessing Styles](#) later in this document.

⁸ Recall that all the logic we are currently executing is happening during the execution phase of the Flash frame

⁹ By default, `UIComponent`'s `createChildren()` does nothing in the method body.

¹⁰ Its important to consider the implications of `createChildren()` and how it can start off a complex chain of code execution (and child component phases) that eventually creates your entire application's layout. Looking at the simple `Button` example, we see first the `TextField` is constructed and then is added to the `Button` as a child. This means that the `TextField` now begins it's own lifecycle. Construction has already occurred and by adding the element to the `Button`, the `TextField` next enters its Addition and then Initialization phase. If the `TextField` has children (in this case it does not) it would then construct its children adding them to itself and the chain continues on, etc. etc.

After `createChildren()` is complete, the `childrenCreated()` method is called. This method is responsible for enabling the first Invalidation-Validation cycle, which we will discuss in the next section.

The last two methods called by `initialize()` are `initializeAccessibility()` and `initializationComplete()`. The method `initializeAccessibility()` configures the component to use the Flash Player's accessibility system, if enabled. The method `initializationComplete()` is the last method called and is used to define the `processedDescriptors` setter method which dispatches the `FlexEvent.INITIALIZED` event.

Although initialization has been completed, there is still work to be done before the component is ready for use. For more information about how to use initialization, refer to the section [Using Initialization](#) later on in this document.

Component Phases: Invalidation (growth / maturity)

Invalidation is the first phase of the component lifecycle that gets repeated throughout the entire life of the component and is also the first phase of growth and maturity. After the `createChildren()` method is called, the `UIComponent`'s `childrenCreated()` method is called. In this method, three sub-methods are called¹¹ which begin the invalidation phase: `invalidateProperties()`, `invalidateSize()` and `invalidateDisplayList()`.

The Invalidation and Validation Cycle

We can't really look too deeply into Invalidation without talking about how it interacts and defines the following Validation phase.

¹¹ As you can see, the Flex Framework is once again breaking down a phase into sub-steps to enable finer control. This process will be examined in greater depth as we move along.

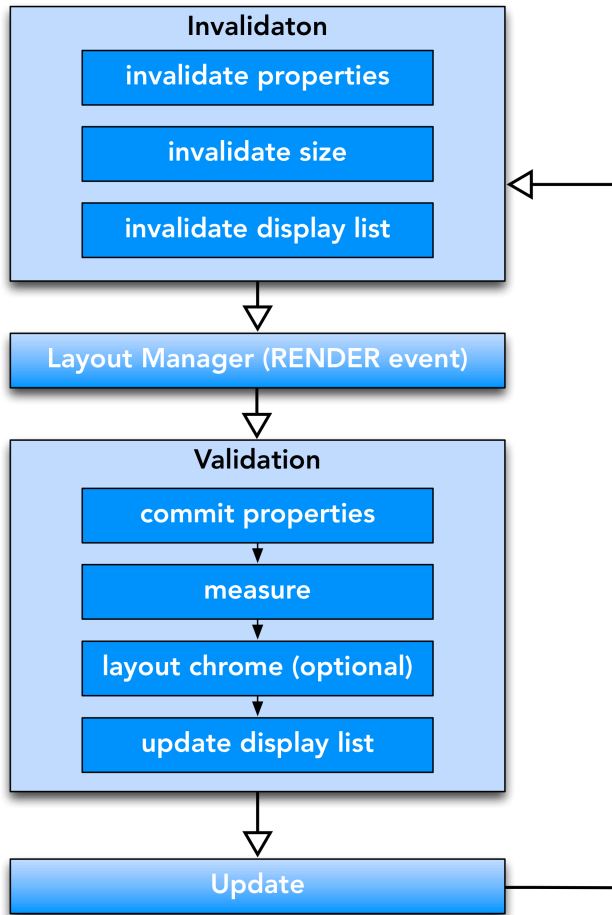


Figure 3 - Invalidation / Validation Cycle

Earlier in this paper we talked about how the Flex Framework does the “wait a frame” for us to manage and mitigate performance within our applications. The Invalidation-Validation cycle is the first place in the component lifecycle that really demonstrates the power the Flex Framework provides us.

The Invalidation-Validation cycle provides a decoupling process. One that separates changing of values (invalidating, i.e. making the old values no long valid) from processing values (validating, i.e. applying the new values). By decoupling this process we get a lot of added benefits, such as postponing processor heavy computation until the last possible code execution stage (just before the player draws to the screen) and preventing unnecessary code from being executed over and over. Let’s look at an example.

```
myButton.width = 20;  
myButton.width = 25;
```

Figure 4 - Setting Width Multiple Times Example

In the code above, let’s imagine that calling the `width` setter did all the calculations to update the component’s size right away; this change would set the components

own width requiring the component to update its layout, set its children's `width` causing them to update their layout and then inform its parent that the `width` change, possibly spawning even more calls to other `width` affected properties and calculations.

Now, on the next line we set `width` again requiring all the changes to be applied a second time. The execution time has now doubled and if the set `width` occurs even more times, the execution time can jump dramatically causing our Flash frame to extend for no good reason. What we would prefer is for all those operations to occur only once and after the last execution of a set `width`.

The Invalidation-Validation cycle solves this problem by using a flag system and delaying the execution of the required update code until later. Let's look at the `width` setter for `UIComponent` (this is partial code):

```
public function set width(value:Number):void {
    if(explicitWidth != value) {
        explicitWidth = value;
        invalidateSize();
    }
    // other code follows...
```

Figure 5 - Width Code Example

The method first checks to see if the value changed, if not the method ignores the new value. If the value has changed, it stores the new value and the calls `invalidateSize()`. Let's look at what the `invalidateSize()` method does.

```
public function invalidateSize():void {
    if (!invalidateSizeFlag) {
        invalidateSizeFlag = true;

        if (parent && UIComponentGlobals.layoutManager)
            UIComponentGlobals.layoutManager.invalidateSize(this);
    }
}
```

Figure 6 - Invalidate Size Example

The `invalidateSize()` method is a simple, but powerful method. It first checks to see if `invalidateSize()` has been called previously using the `invalidateSizeFlag` value¹². The first time this method is called (either during birth or post-validation), the `invalidateSizeFlag` is false and is set to true, the method then checks if we

¹² The `invalidateSizeFlag` should be considered an internal flag, one that we should never set directly. Later on in the Flex Component Development Best Practices section we will demonstrate how to create our own property flags to track the current Invalidation-Validation property states and these should not be confused with the internal `UIComponent` flags.

have a parent and then registers itself with the `LayoutManager` instance¹³. If we have called `invalidateSize()` previously, and have not run a validation phase yet, then we don't need to worry about registering with the `LayoutManager` because we have already registered for the size to be validated during the next Validation pass. We can now call `setWidth` on our `Button` as many times as we want before the next validation pass and performance will not be hindered.

Now that our component is registered with the `LayoutManager`, some pretty ingenious code is executed. What that `LayoutManager` does, is it uses the `callLater()` method on a hidden `UIComponent` instance to execute the validation phase once the next `Event.RENDER` event is dispatched from the stage. If you recall from Sean Christmann's article, the `RENDER` event is dispatched to allow user code to be executed just before the Flash Player draws the display stack to screen. By running the validation phase after the `RENDER` event guarantees us that this is the last possible code to be executed before the screen is rendered¹⁴.

The `LayoutManager` acts as a queuing system, tracking all the components that register invalidation changes before the next `RENDER` pass. When the `LayoutManager` gets the `RENDER` event, it checks to see what components have registered with it during the invalidation phase and then begins executing the validation phase on those objects. If the `LayoutManager` is in the process of executing validation on a set of components and new components register with the `LayoutManager`, these new components will be queued for the next `RENDER` pass. This helps prevent an endless cycle of updates during a single pass.

The process of Invalidation and Validation is a cycle because any time a property changes it invalidates the component; the validation process must then be executed again to put the component back into a valid state.

The Three Types Of Invalidation

Going back to our `Button` example, during the first Invalidation phase (called during the Initialization phase) we call the three invalidation methods:

`invalidateProperties()`, `invalidateSize()`, and `invalidateDisplayList()`.

¹³ The reason the code checks for a parent first, is to determine if the component is currently on the display stack. If the component does not have a parent then there is no reason to run these calculations because we will not be rendered on screen.

¹⁴ If we return to Sean Christmann's blog post about the AVM2 Marshal frame system the benefit of executing the validation on the render pass becomes more apparent. Sean's "Flash frames synthesized from AVM2 slices" diagram clearly shows that the user action is executed twice for every invalidate action at 25 frames per second. In the Flex Framework the invalidate action diagrammed by Sean corresponds to the Flex validation phase. If the Flex Framework did not wait for the `RENDER` event then the validation code would be executed twice before the content was actually rendered, potentially causing performance issues and undue calculations triggered by the first user action and then invalidated again by the second user action. In the 25 frames example we would only want the second user action's changes to be applied during the validation phase.

It is important to know what is affected during the Invalidation phase when we call these methods. The `invalidateProperties()` method is called whenever a property changes that is required before the component is re-measured and laid out (bear with us, this will be clearer when we get to Validation). The `invalidateSize()` method is called when something changes (property or other action) that requires the component to re-calculate its size or its children's size. Finally, the `invalidateDisplayList()` method is called whenever something changes that requires the components display list to be updated/managed (adding/removing of children, skin updates, drawing API, etc) .

Component Phases: Validation (growth maturity)

If we change a property, such as data, it may not require the component to change its size or update its display list, yet we still need to execute some calculation on the data when it changes. By breaking invalidation into three methods we can also break validation into three corresponding methods with a special fourth method called `layoutChrome()` which we will explain in a bit. The four methods of validation are: `commitProperties()`, `measure()`, `layoutChrome()`, and `updateDisplayList()`.

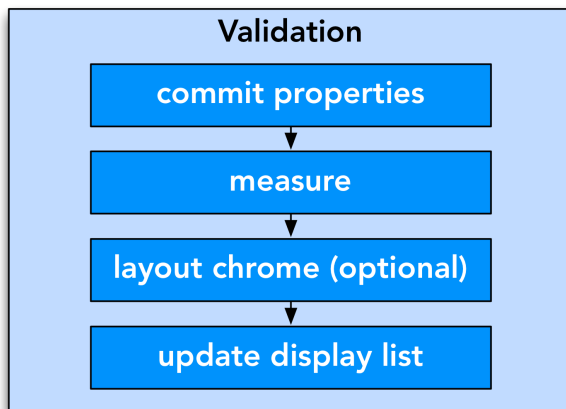


Figure 7 - The Validation Stages

Unlike Invalidation¹⁵, Validation has a pre-defined order in the way the methods get executed: `commitProperties()` first, `measure()` second, `layoutChrome()` third (if enabled), and finally `updateDisplayList()`. Validation has this linear arrangement because the order that we validate (apply settings) is very important.

For example, when we use `invalidateProperties()` to state that one or more of our properties have changed, we need to make sure to validate the properties

¹⁵ The Invalidation methods don't have a call order because you execute them only when you need to invalidate some aspect of your component. Since multiple aspects of the component can change at any time there cannot, nor should there be, a required call order for invalidation.

before we start laying out our component, because those new property values may be required during the layout stage.

Each invalidation method has a corresponding validation method:

`invalidateProperties()` corresponds to `commitProperties()`, `invalidateSize()` corresponds to `measure()` and `invalidateDisplayList()` corresponds to `updateDisplayList()`. As we invalidate the component, the `LayoutManager` tracks which invalidation methods have been called on the component, and only calls the corresponding validation methods during the validation phase.

For example, let's imagine that our component has a property called `data`, if the value does not affect the size or the display of the component, then we would only want to call the `invalidateProperties()` method during Invalidation. This means that when the next `RENDER` event is dispatched, the `LayoutManager` will check the component, see that only properties are invalid and then call `commitProperties()`.¹⁶

Validation also has a fourth method, and as you can see in Figure 7 - The Validation Stages, called `layoutChrome()`. The `layoutChrome()` method is not defined by `UIComponent`, but by the base `Container` Class. When creating a container, you will often want to create a border or padding around the children content, which should be executed within this method. `layoutChrome()` is executed after `measure()` so that the children's dimensions are known and the "chrome" can be applied.

Now that we have defined the Invalidation-Validation cycle lets look back to our `Button` component creation process. If you recall, during the Initialization phase the children are created (`TextField` for the label) and then the `childrenCreated()` method calls the three Invalidation methods.

At this point, we now have a parent and access to the `LayoutManager`, so the invalidation methods will register our new `Button` with the `LayoutManager` for all three invalidation methods and wait until our next `RENDER` event to process the changes. All three methods are called because this is the first Validation pass of the component's lifecycle.

After the `LayoutManager` has executed all the required validation methods¹⁷ on the component, the `LayoutManager` checks to see if the component has been marked as

¹⁶ When considering validation order, the order the methods are called are important only when more than one invalidate method has been called during the invalidation pass. If the `invalidateDisplayList()` method is the only invalidation method called, then only `updateDisplayList()` will be called during the validation pass.

¹⁷ Technical Note: The defined validation methods are all protected methods, which are intended to be overridden by developers to provide support for custom component creation. These methods are not the methods called directly by the `LayoutManager`. The `LayoutManager` accesses the public functions `validateProperties()`, `validateSize()`, and `validateDisplayList()`. These methods double check to make sure the component is truly invalidated and handle other checks before executing the methods described above. We focused on the protected methods because this is most often how you create custom components and also how the Adobe documentation explains the validation methods. For

initialized or not, if not the `LayoutManager` marks the component initialized. At this point our component is considered initialized and updated. The `LayoutManager` then has the component dispatch the `FlexEvent.UPDATE_COMPLETE` event.

Component Phases: Update (maturity)

The update phase is defined as any time a component is invalidated and then goes through the validation cycle. Once the component is validated the `LayoutManager` will once again make the component dispatch the `UPDATE_COMPLETE` event¹⁸. This cycle repeats itself over and over until the component is removed from its parent. Its important to note that the Update phase is where a component spends the most time during its lifecycle. As the application is being interacted with by the user the components are constantly being Updated and changed. For more details about how to manage the Update phase refer to the Validation Cycle and Methods section below.

Component Phases: Removal (death)

The last phase of a component is the removal phase. This phase occurs when the component is no longer parented. In most cases, this occurs when `removeChild()` is called on the parent component, passing in the component to remove as a reference. The reason we say “most cases” is because when the parent of the component, or further up the parent hierarchy, is removed from the display stack the component may enter the Removal phase, depending upon how the components are referenced in the application. We explain more about this in a bit.

When `removeChild()` is applied to a child, such as:

```
this.removeChild(myButton);
```

Figure 8 - Remove Child Example

The parent component’s `removeChild()` method calls three methods, just like `addChild()`.¹⁹ The first method called is `removingChild()` which by default (at the `UIComponent` level) does nothing. This can be overridden to perform functionality before the component is removed from the display stack.

The next method called is `$removeChild()` which is a Flash Player level method that does the actual removal of the component from the display stack and then checks the components references in memory. If there are no references to the component

best practices on utilizing each of the validation methods in custom components see the Validation Cycle and Methods section below.

¹⁸ The `LayoutManager` also dispatches an `UPDATE_COMPLETE` event when all of the registered components have completed their Validation phase.

¹⁹ Its important to note that our references to functionality, such as `removeChild()` are defined at the `UIComponent` level. It’s quite possible that overriding these methods later down the inheritance stack can change the defined functionality, order, etc. It’s not recommended to make such radical changes, but it may occur if deemed appropriate.

(such as other object pointing the component, event listeners with strong reference set, etc.) the component is then marked for garbage collection.

Finally, the `childRemoved()` method is called removing references to the components `parent` and `document` properties. This enables the component to be re-parented if required and also prevents the component from entering the Invalidation-Validation cycle.

At this point, the component is in its final phase. If there are no other references to the component then it will be garbage collected and deleted from memory. If the component is garbage collected, then all its children are also garbage collected. This leads back to our earlier comment about how removal may not occur directly via `removeChild()`.

If the parent, grandparent or any parent up the hierarchy is removed and there are no references to any of the components (children or otherwise) in the removed component hierarchy, then all the components will be garbage collected.

For example, we have an `HBox` with our `Button` as a child. If we remove the `HBox` from the display stack then the `HBox` enters the Removal Phase. During this process the `HBox` does not remove any of its children nor do any of its children enter the Removal phase. Yet, if the `HBox` and our `Button` do not have any references to them (outside of the parent/child relationship) then both components will be garbage collected when the time comes.²⁰

A Flex Application Is Born

Now that we have a better understanding of the Component Lifecycle, we can begin looking at the Flex Application Lifecycle. The Flex Application Lifecycle is similar to the Component Lifecycle (stages, phases, etc.), but has some very important additions and modifications to provide support for managing a complete application. We have broken down the Application lifecycle into the following phases: Construction, Initialization, Preloading, Child Creation, Child Display, and Destruction.

²⁰ For more details on the AVM2 Garbage Collection (GC) process we highly recommend reviewing Grant Skinner's excellent articles on how GC is processed and executed in the Flash Player. Understanding the overall GC process will help clarify the parent/child relationship for the Removal phase and also can be applied to Best Practices when developing custom components.

http://www.gskinner.com/blog/archives/2006/06/as3_resource_ma.html

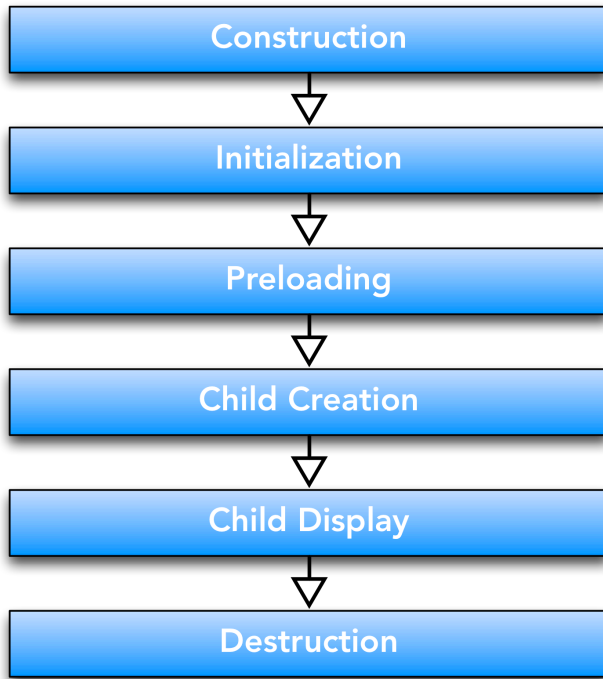


Figure 9 - Flex Application Lifecycle

Flex Application Phases: Construction

The Construction Phase of a Flex Application is a little more in-depth than the component Construction phase because it begins with the loading of the compiled SWF before any Classes are actually constructed.

When a SWF is loaded into the Player the first frame of the SWF is prepared and then executed. One of the powerful features of Flash is that not all of the application SWF has to be loaded before the Player starts to execute. The Player is responsible for streaming in the content and begins execution as the frames are loaded. This ability enables SWF applications of larger size to start playing and showing content to the user while the rest of the application SWF content loads.

The Dark Art of the Flex Compiler

From our current research, the first piece of the Flex Framework code that is created and executed is the `SystemManager`. How the `SystemManager` is linked into the SWF and defined, as the first Class to be created and executed, is not clearly described in available documentation. We do know the Flex Compiler defines this linkage when it is generating the output SWF by defining the application's `SystemManager` class as the root `MovieClip` of the SWF, which tells the Flash Player to construct the `SystemManager` instance on load.

Unfortunately, the actual mechanics of the Flex Compiler and how it creates the initial frame stack is a bit out of scope for this paper. Hopefully, as we do further research over time, we can update this section and explain in excruciating detail

exactly how the compiler builds the call stack for the first frame. Until then, we will just make note of compiler “magic” and try to explain it as best we can. So for now, bear with us and we will start with the creation of the `SystemManager`.

At the Top Sits the SystemManager

Every Flex application (Browser-based or AIR) has a `SystemManager` instance that is responsible for managing all the displayed content in the application window. The application window, as described by the `SystemManager` ASDoc comments, is:

...an area where the visuals of the application are displayed. It may be a window in the operating system or an area within the browser.

This means that there is one `SystemManager` for every window.²¹ When the `SystemManager` is created, it first checks to see if it is the first `SystemManager` in the current window hierarchy. For this paper we will assume that our `SystemManager` is the first `SystemManager` to be created in our windowed application, and we will not explore how child `SystemManagers` or sub-applications are managed.²²

The way the `SystemManager` determines it is the default (or root) `SystemManager` for the current application window, is by checking to see if the `stage` property is defined at construction.

A stage is the base object that all displayable objects hang off of, including the `SystemManager`. Just like in the *Highlander*, there can only be one stage per windowed application. The first `DisplayObject` hung off of the stage is the root object and the `root` property of the `DisplayObject` is set to a reference of itself. All other displayable objects are hung off the root object and their `root` property is a reference to the root `DisplayObject`. Examine Figure 10 - The Windowed Application Hierarchy below to see how this structure is organized.

²¹ AIR applications have the ability to display multiple windows at the same time. This means that there will be one `SystemManager` for every window being displayed. When developing multi-windowed applications in AIR it is important to understand this concept so that you can manage your application correctly.

²² In certain situations it is possible to have multiple `SystemManagers` in the same window instance. This usually occurs when a Flex Application loads a child Flex Application via a Loader. To prevent control conflicts, there can only be one root `SystemManager` that is responsible for the window. During construction, the `SystemManager` checks to see if it is the default manager or if there is a parent `SystemManager` already. If there is a parent `SystemManager`, then the newly constructed instance delegates most of its responsibilities to the root manager.

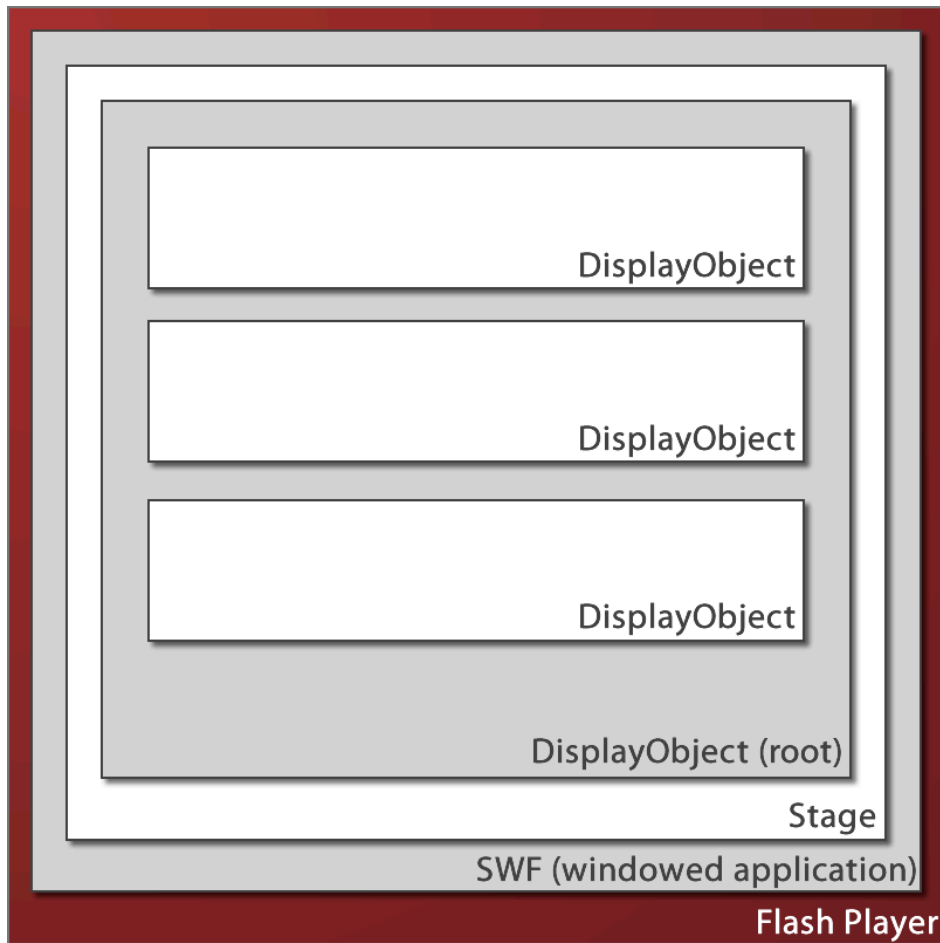


Figure 10 - The Windowed Application Hierarchy

When a `DisplayObject` is added to the stage (or children of the root `DisplayObject`) the `stage` property is defined as a reference to the windowed application's stage. Only the root `DisplayObject` has the `stage` property set at construction, otherwise the `stage` property is set when the object is added to the display stack (`addChild()`, `Loader`, etc). So when the `SystemManager`, which extends `MovieClip` -> `DisplayObject`, is constructed it checks to see if the `stage` property is defined, and if so it knows that it must be the root `SystemManager` because the stage was defined at construction.

Once the `SystemManager` determines it's the root, it then has the ability to access its own `loaderInfo` property. The `loaderInfo` property contains an instance of the `LoaderInfo` Class which allows access to information such as the source SWF `url`, parameters that have been passed to the SWF, domain information, content, bytes loaded, total bytes, etc.

In typical Flash/Flex development we usually access the `loaderInfo` details of a `SWFLoader` or other `Loader` based Class when we are explicitly loading content within our application. We use the `loaderInfo` object as a way to get progress

information as bytes are being loaded to give feedback to our users or to know when the target content is loaded into our application. In the case of the `SystemManager`, the SWF isn't being loaded as a child of our application; it is being loaded by the Flash Player itself. The root `DisplayObject` is given reference to the Flash Player's `loaderInfo` therefore we are essentially able to watch our own SWF load (very meta, isn't it?).

One of the features of the `LoaderInfo` Class is that it dispatches an `Event.INIT` event, when all the properties and methods associated with the loaded SWF are accessible, the constructors of the first frames child objects are constructed and when all the ActionScript in the first frame has been executed. This is important, because the `SystemManager` registers to its own `loaderInfo`, listening for when the `Event.INIT` is dispatched. When this event is dispatched, the `SystemManager` knows that all its important information required for initialization and other configuration details are now loaded and available.

Flex Application Phases: Initialization

Once the `SystemManager`'s `Event.INIT` is dispatched, the `SystemManager` begins the Initialization phase of our Application. The `SystemManager` first does some more parenting checks to see if it's the root or not²³, registers for the `Event.ENTER_FRAME` event²⁴ (which is dispatched at the beginning of every new Flash frame), and then calls it's own `initialization()` method.

Inside the `initialization()` method, the `SystemManager` sets its own `width` and `height` to the encapsulating widowed application's size and then creates an instance of the `Preloader` Class. Before we can start creating our Application, we need to load any required external libraries and resource bundles to make our application function.

Managing Externalization

One of the powerful features of Flex 3 is the ability to externalize functionality in Remote Shared Libraries (RSLs) and localized content in Resource Bundles.²⁵ There are many reasons why we may want to externalize content; such as download size, sharing resources, targeted languages, etc.

One of the technical challenges with creating externalized content is to make sure that it is all available when the application requires it. The loading of required

²³ If the `SystemManager` is not the root manager then this method is responsible for setting up communication paths with the root `SystemManager` and sandbox constraints. This only occurs when Flex Applications load Flex Applications.

²⁴ It's important to note that the `ENTER_FRAME` event will not be dispatched right away. The constructor of the `SystemManager` calls `stop()` to make sure that the next frame is not entered until other configuration requirements are met, we will talk more about this a little bit later.

²⁵ For more information about RSLs and Resource Bundles we recommend reading the *Adobe Flex 3 Help* to learn how you can create custom external content and leverage it in your applications.

external content is delegated to the `SystemManager`, which uses a `Preloader` to load the content in and display feedback to the user as it loads.

Once the `SystemManager` creates the `Preloader`, it registers to the `Preloader`'s `FlexEvent.INIT_PROGRESS` and `FlexEvent.PRELOADER_DONE` events so that the `SystemManager` can begin the Application Class Construction-Initialization phase once all of the required external data has been loaded. The `Preloader` instance is then added as a child²⁶ of the `SystemManager`, but the data loading is not begun yet.

The `SystemManager` first needs to build a list of required external assets before the `Preloader` can begin loading content. The first list of external content defined for the `Preloader` is the list of RSLs required for the application.²⁷ The list of RSLs is provided by using the `info()` method²⁸ which is populated by compiler magic.

Once the list of RSLs has been defined, the `SystemManager` creates an instance of the `ResourceManager` enabling the `Preloader` to have access to the compiled resource bundle. A default Resource Bundle is always compiled into the application so that the `Preloader` can display localized text to the user or if the system runs into any errors, the errors may be displayed to the user in their language. If the system did not have the resource bundle compiled in at this time, it would not be possible to show the user localized text since the `Preloader` has not yet loaded the external bundles.

The `SystemManager` sets up the base `StyleManager` so that the `Preloader` can use it and then the `SystemManager` parses any locale chains set via the flash vars on the SWF to determine which locale to use for the application. Next, the `SystemManager` builds a list of external resource bundles to load via their defined URLs in the flash vars, determines if a custom display class has been defined for the `Preloader` (more compiler magic via `info()`) and then kicks off the `Preloader` passing in the list of external assets (RSLs, resource bundles) to load and the display class used to render the `Preloader`.

Flex Application Phases: Preloading

The `Preloader` takes the external assets lists (RSLs, SWZ, Resource Bundles), creates an instance of the `RSLListLoader` and begins loading in the content. As the RSLs and Resource Bundles are loaded, progress events are dispatched to the UI to

²⁶ By adding the `Preloader` as a child of the `SystemManager`, this begins the `Preloader`'s Component Lifecycle.

²⁷ This list of RSLs will include the Flex Framework library (SWZ) if the developer has enabled this at compile time.

²⁸ The `info()` method is an interesting beast. If you look at the code in the `SystemManager` you will see that `info()` just returns `{}`. Yet, the code consistently makes references such as `info()["rsls"]` which should not return a value according to the default code. We have not confirmed this with the development team (or by researching the compiler) but we believe the compiler overrides the `info()` method during compilation to provide values/settings required by the application. This feature enables dynamic settings to be available to the application without having to access an external source.

inform the `Preloader` when to update the display progress bar. Once the `Preloader` loads all the RSLs, it dispatches the `FlexEvent.INIT_PROGRESS` event informing the `SystemManager` that it has loaded in all the required external assets.

The `Preloader` is responsible for handling two main tasks, first loading in the RSLs and Resources bundles (as we just discussed) and it is also responsible for giving feedback to the user as the Flex Application goes through the Child Creation and Child Display phases. As we get deeper into the process, we will comment on when the `Preloader` updates and when it has completed its tasks.

Now that the RSLs and bundles are loaded, causing the `Preloader` to dispatch `INIT_PROGRESS`, the `SystemManager` receives the event and advances one frame via the `nextFrame()` method. During the Initialize phase the `SystemManager` registered to itself for the `ENTER_FRAME` event which triggers the `docFrameHandler()` method. When we call `nextFrame()`, the `SystemManager` (which extends `MovieClip`) steps a frame and dispatches an `ENTER_FRAME` event.

The `docFrameHandler()` method begins defining many of the Singleton classes throughout the application, such as the `BrowserManager`, `HistoryManager`, `CursorManager`, `LayoutManager`, etc. Once the Singletons are defined, the loaded compiled resource bundles are then installed into the `ResourceManager` instance for the application, making their content available to the entire system.

Flex Application Phases: Child Creation

Up to this point, we have been getting our external data loaded and the system prepared so that we can begin creating our root `Application` class. All this information has to be available to us because it defines the framework and implementations required to run our application. Now that we have everything loaded and in place we can start building out the top-level `Application` Class. To start creating the top-level `Application`, the `docFrameHandler()` method calls the `initializeTopLevelWindow()` method on the `SystemManager`, which initiates the creation process.

First, the `SystemManager` registers to the `MouseEvent.MOUSE_DOWN` for the window to handle focus management and registers to the stage's `Event.RESIZE` event so that when the user changes the size of the window the `SystemManager` can track it and pass it on to its children. Once this registration is complete the `Application` instance is created.

The class that is created for the root `Application` is based on what was defined as the base `Application` during compilation. Its possible that you extended `Application` in your project or in AIR it's a `WindowedApplication` instance. The `SystemManager` uses a factory method called `create()` which references `info()["mainClassName"]` to determine the type of Class to create (even more compiler magic).

Now that we have an instance of our `Application` class created, the `SystemManager` registers to the instance's `FlexEvent.CREATION_COMPLETE` event, defines the `url` and the parameters on the `Application` instance and then updates the `width` and `height` of both the `Application` and the `SystemManager`.²⁹

After the size has been updated on the `Application`, the `SystemManager` registers the `Application` instance with the `Preloader`. The registration of the `Application` to the `Preloader`, enables the `Preloader` to register to the different Lifecycle validation events ("`validatePropertiesComplete`", "`validateSizeComplete`", `CREATION_COMPLETE`, etc.). By registering to these events the `Preloader` can provide the user more feedback as the `Application`'s Component Growth Phases progress.

Flex Application Phase: Child Display

Since our `Application` Class extends `UIComponent` it must follow the Component Lifecycle. At this point, the `Application` has now completed its Construction phase but will not begin the Addition Phase until it is added to its parent, the `SystemManager`.

If the `SystemManager` was to just call `addChild()` now and go through the three child addition steps (`addChild()`, `$addChild()` and `childAdded()`) the Flash Player would begin rendering the `Application` content to screen. Yet, the process of creating all the `Application`'s children and running all the potential initialization code required to get the `Application` finalized takes a lot of time and frame cycles.

Much of this labor would require screen re-draws that may or may not be meaningful until the `Application` has completed its Initialization Phase and first Invalidation-Validation Cycle. To prevent this unnecessary re-drawing the `SystemManager` only calls `addChild()` and `childAdded()` to start the birth and growth phases of the children components.

After the `Application` dispatches the `CREATION_COMPLETE` event³⁰ the `SystemManager` sends the `Application` through one more invalidation pass. The `Preloader` is also listening to the `Application`'s `CREATION_COMPLETE` which dispatches a `PRELOADER_DONE` event to the `SystemManager`.

The `SystemManager` removes the `Preloader` at this point, since it has completed its UI task. Once the `Preloader` is removed the `SystemManager` adds the `Application` instance using `super.addChild()` and then both has the `Application` and itself dispatch the `FlexEvent.APPLICATION_COMPLETE` event.

Now that the `Application` is up and running the `SystemManager` relegates itself back to tracking system mouse events, focus management, dialog management and other

²⁹ The reason that the `SystemManager`'s `width` and `height` are also updated is that between initial creation and now, the user may have resized the application window. If you recall, the `SystemManager` just registered to the `RESIZE` event of the stage, so previous resizes may have been missed.

³⁰ This event means the `Application` has progressed through its entire birth and growth lifecycle.

SystemManager duties. The Application enters its Update phase and the system continues like this until the user closes or quits the application.

Flex Application Phase: Destruction

The Destruction phase of a Flex Application is truly dependent on the how the application is being executed. In the browser, the destruction phase is quick and unannounced to the application meaning that you have no control of what the app does during this time. This also means that the exact order of operation is really hard to tease out. We will cut to the chase and just say that when the user closes the browser, it kills the SWF instance and there is not much you can do about it, at least not purely in ActionScript³¹.

In AIR applications, you have much more fine control with the `WindowedApplication` Class. AIR provided multiple handlers to the closing events dispatched by the application so that you can elegantly handle exiting of the application or even prevent the application from being closed. Once again, the exact order of operation is unknown because `NativeWindow` and `NativeApplication` are both included in the `playerglobal.swc` and therefore we do not have access to the code to allow us an in-depth analysis.

Flex Component Development Best Practices

Now that we have fully explored the Flex Component and Application Lifecycles, we can use this knowledge to begin defining some development best practices. We will examine different features and how we can leverage (or avoid) them to make our applications and components work to their best ability.

Using Construction

One of the misconceptions of the Flex development process is that using the constructor to configure your application is a good approach. Fundamentally, the idea of setting up your properties in the constructor makes sense, yet after studying the Lifecycle, the potential pitfalls of relying on the constructor becomes more apparent.

One example of a potentially hazardous decision is setting style properties during the construction of your component. Technically this would work, the `setStyle()`

³¹ There are some interesting tricks you can do with JavaScript and the Browser to try and give your application a heads up. One approach, proposed by Andrei Lonescu, is to have JavaScript listen for the `onbeforeunload()` event of the browser window. At that point, you have the JavaScript display a dialog to allow the user to chose if they want to quit or not and at the same time tell the app via `External Interface` that the browser may be closing. We have not tested this process, so we cannot state this is a best practice for handling this kind of situation. But it is really cool idea!

<http://www.flexer.info/2008/02/25/browser-window-close-event-and-flex-applications/>

method does store the value properly and when the Addition phase begins the style values are be applied. The problem with this approach is that this style could be inherited up the chain, may not be a valid style, could be overwritten later in the `createChildren()`, etc and this could cause potential unintended issues down the road.

Another issue is that if the developer does not fully understand the Lifecycle, he or she may decide to try and access a property that does not exist yet. Many properties, especially children properties don't exist until much later in the Lifecycle. Sometimes this error becomes obvious the first time the code is executed, in others it might not appear until a specific order of operation occurs.

JIT and the Constructor

Another reason construction should not be used for most of the configuration code is because the executed code within the constructor of a Class is always interpreted by the Flash Player. This means that the Just-In-Time (JIT) compilation process will never be applied to store the constructor code as pre-compiled processor code for future use. Because the constructor will never benefit from JIT, if you must perform calculations at Construction, you should move this logic to a separate method.

```
public class MyCustomClass extends DisplayObject
{
    public function MyCustomClass() {
        this.construct();
    }

    public function construct():void {
        this.addEventListener(FlexEvent.PREINITIALIZE, preinit)
        this.addEventListener(FlexEvent.INITIALIZE, init);
    }
}
```

Figure 11 - Using Methods in Constructors

By moving this code to a separate method you are enabling the Flash Player the ability to apply the JIT process to your calculations if it deems it as benefiting from this process. Unfortunately, you cannot guarantee that the JIT will be applied to your method, but by moving it out of the constructor you have the potential to leverage the performance boost that JIT can provide.³²

³² The Flash Player AVM2 uses a trace approach to JIT. This means that all code is interpreted by the AVM and during this process the AVM tracks statistics on execution until the number of repeated operations reaches a specified threshold. Once the threshold is reached, the next time the code is executed the JIT "traces" the path and then marks it for compilation. The compiled path is then saved for reuse and when the path is requested again the compiled version is used.

For a great explanation of how a tracing JIT works read Mike Pall's explanation for the Lua JIT @ <http://article.gmane.org/gmane.comp.lang.lua.general/44781>

Overall, the Construction Phase is really just prepping the component to be used and very little of the instance is configured or available during this time. We recommend that you perform as little work or calculations as possible during this time, and try to move setup code to later phases of the component.

Using Initialization

Similar to Construction, the Initialize Phase is one of the most overused and probably abused phases by Flex developers. For us (the authors), one of the most eye opening experiences of researching this paper was the realization of our improper use of the `initialize()` method. To be perfectly honest, as developers, we were notorious for using the `initialize()` method as a place to do a lot of configuration and setup, especially when using the Code Behind Pattern³³.

The reason we adopted `initialize()` as the desired override for setting properties is that this was the first apparent location that the children have been created (meaning they are available to us), the styles can be set/accessed and most of the component is ready to be used. For many of us, finding the best location to perform these kinds of setup was the first real grey area of the Flex documentation.

To determine the best location for this kind of configuration, we had to find it through experimentation and a bit of on-the-fly research. Through this, we discovered the Initialization Phase, which at the time felt like a logical place to do this kind of setup. Once found, we stopped digging deeper because it got the job done. Unfortunately, this solution didn't always work. In most cases, it was safe for what we were building but every once and a while a strange race condition would appear, or some issue would appear that could only be teased out during integration testing.

Now that we have fully researched the Flex Lifecycle, our general recommendation is that you don't override `initialize()` at all. If you look at most of the Flex UI components, they never override the `initialize()` method, and for good reason. The problem with `initialize()` is that when you inject code via override you may very well be in the middle of a implementation hierarchy and what you expect to be configured may not be ready just yet.

The Flex component architecture has split the Initialize phase into multiple sub-steps and there are better methods available for us to override. The best one to

³³ The Code Behind Pattern (CBN) is a Flex design pattern that enables to developers to separate layout from logic by creating a "backing" ActionScript Class that the MXML layout extends from. This enables developers to keep layout solely in MXML and all programmatic logic in the ActionScript Class. The CBN is one of the more divisive Flex patterns in the community. For every pro-CBN developer there is an anti-CBN developer. At DevelopmentArc, we are strong proponents for using the pattern and have found it beneficial to the growth and maintenance of applications. For more information about the CBN, we recommend reading James' write up at his blog *Vivisecting Media*.

<http://blog.vivisectingmedia.com/2008/04/the-flex-code-behind-pattern/>

override is dependent on what you are trying to do. If you want to access children after they are created then you will probably want to override `childrenCreated()`.

If you want to wait for all the configuration to be complete and to make sure initialize is done, override the `initializeComplete()` method. This method is called last once all the other initialize sub-methods are done. By overriding `initialize()` you can actually break this order and are potentially making updates post-`initializeComplete()` which should always be the last method called.

If You Must Override Initialize...

There may be a situation, although highly doubtful, where you just have to override the `initialize()` method. If this is the case, then here are a few important notes.

First, you need to decide if your code should be called before or after `super.initialize()`. When you call `super`, and it reaches the `UIComponent` level the `initialize()` method dispatches the `PREINITIALIZE` event, informing the system that children are about to be created. This event should always be dispatched before starting to create the children for the components.

Next, the method calls `createChildren()`, followed by `childrenCreated()` and finally `initializeComplete()`. If your code is called before `super` you need to be aware that the children up and down the inheritance chain do not exist yet. If your code follows `super` then you are executing code post-`initializeComplete()` which is breaking the Flex Component defined architecture call order.

As you can probably tell, overriding `initialize()` is a lose-lose situation in 99.9% of the cases. So, once again, we recommend that you don't do it and choose another location to apply your configuration.

The Invalidation-Validation Cycle and Methods

Understanding the Invalidation-Validation Cycle is probably the most important process for developing Flex components and applications. This cycle is where most of the work is done and occurs over and over again through the lifecycle of the application. In this section, we are going to examine some techniques to leverage this cycle and look at benefits we gain by applying these techniques.

Separating the Phases

If you recall, the `Event.RENDER` event breaks the Invalidation-Validation cycle into two distinct phases. The reason this is done is to enable the ability to only call the validation code when it is required, preventing us from executing code unnecessarily.

We are going to use an example for this section to demonstrate how to write code that takes advantage of the Invalidation-Validation phase. First, we will look at an incorrect way of writing the code and then examine how we can modify it to fully leverage the cycle.

In our example, we have a component that needs a property called `dataValue`, which is an `Array of Numbers`. When our `dataValue` property changes we first need to loop over the `Array` and calculate the total value. Once the total value is updated, we need to draw the values, as a chart, to the screen using the drawing API.

Without knowing about the Invalidation-Validation cycle the developer may write code that looks something like this:

```
[Bindable] public var total:Number;
private var __dataValue:Array;

public function get dataValue():Array {
    return __dataValue;
}

public function set dataValue(value:Array):void {
    __dataValue = value;
    // calculate the total value
    var len:int = value.length;
    total = 0; // reset total
    for(var i:uint=0; i < len; i++) {
        total += Number(value[i]);
    }

    // draw the data to screen
    drawChart();
}
}
```

Figure 12 - Un-optimized Code

Fundamentally, the code in Figure 12 - Un-optimized Code is okay, but we execute this potentially heavy process every time the value changes. Looking back at the AVM2 Marshal in The Elastic Racetrack section, our `dataValue` may change multiple times before our next screen render occurs.

This means that we are both updating and calculating the `total` each time it changes, which also kicks off binding events because we are changing the `total`'s value. We are also calling our `drawChart()` method which uses the drawing API to update the UI display. Since we may not have executed a screen draw between each value change, we are needlessly using the drawing API and using unneeded processor cycles.

So how do we solve the issues raised in Figure 12 - Un-optimized Code? We need to do two things, first prepare for the fact that the `dataValue` may change multiple times before we are ready to render to screen. Second, we need to move our calculations and drawing out of the setter method and into the Validation methods so that they are executed only when the values have changed.

Using Dirty Flags

The first step in solving this process is introducing the concept of dirty flags. A dirty flag is a Boolean variable that is marked true when a value has changed. We can then check the state of the flag, and if it is true then we know the corresponding value has changed and is considered “dirty”. Let’s look at how we would change the code to implement the dirty flags:

```
[Bindable] public var total:Number;
private var __dataValue:Array;
private var _dataValueDirty:Boolean = false;

public function get dataValue():Array {
    return __dataValue;
}

public function set dataValue(value:Array):void {
    if(__dataValue != value) {
        __dataValue = value;
        _dataValueDirty = true;
        invalidateProperties();
        invalidateDisplayList();
    }
}
```

Figure 13 - Using Dirty Flags Example

In Figure 13 - Using Dirty Flags Example, we have added a new variable called `_dataValueDirty`. This Boolean property is our dirty flag. When our `dataValue` changes, we now first check to make sure that the value has changed, next we set our private property to the new value, we mark our flag as true (or dirty it) and then call our invalidate methods, stating that both our properties and display list need to be validated on the next pass.

If you remember, back in The Invalidation and Validation Cycle section of this paper we examined how the invalidate methods mark the component to be validated during the next validation pass by the `LayoutManager`. By setting our dirty flag and then calling these methods we have created our separation between the Invalidation and Validation of the `dataValue` property of our component. This means that our `dataValue` setter method can be called as many times as needed during the Invalidation cycle and no code will be executed until our next Validation pass. This also guarantees that our drawing API calls will only occur once, right before the content is drawn to screen.

Implementing Validation Methods

Now that we have invalidated our component, how do we apply our calculations and draw our UI when the next Validation Phase occurs? Looking back at Component Phases: Validation (growth maturity) we do this by overriding the `commitProperties()` and `updateDisplayList()` methods:

```
private var _chartDirty:Boolean = false;

override protected function commitProperties():void {
    super.commitProperties();

    if(_dataValueDirty) {
        _dataValueDirty = false;
        // calculate the total
        var len:int = __dataValue.length;
        var _total:Number = 0;
        for(var i:uint=0; i < len; i++) {
            _total += Number(__dataValue[i]);
        }
        // set the total
        total = _total;
        _chartDirty= true;
    }
}

override protected function updateDisplayList(
    unscaledWidth:Number,
    unscaledHeight:Number):void {

    super.updateDisplayList(unscaledWidth, unscaledHeight);

    if(_chartDirty) {
        _chartDirty= false;
        // draw the data
        drawChart();
    }
}
```

Figure 14 - Overriding the Validation Methods Example

The validation methods are always called in an explicit order: `commitProperties()`, `measure()`, `layoutChrome()`, and finally `updateDisplayList()`. As we have previously mentioned, this order is important because calculations and properties set in the `commitProperties()` may affect the size or the display list.

Looking at Figure 14 - Overriding the Validation Methods Example, its possible that we may need the total value to be drawn or updated in the display list, so it should be calculated first. This is why we use the invalidate methods. Because we need to calculate a total and also draw our chart we need to make sure that both `invalidateProperites()` and `invalidateDisplayList()` are called by the setter in Figure 13 - Using Dirty Flags Example so that `commitProperties()` and then `updateDisplayList()` are called.

When `commitProperties()` is called, we check to see if the `_dataValueDirty` flag has been set to true. We want to make this check because if the value has not

changed there is no reason to run our calculations and re-draw our UI.³⁴ If our value has changed then we calculate the total value, set it and then mark the chart as dirty so that when `updateDisplayList()` is called we can re-draw the chart.

Once `commitProperties()` is complete the `updateDisplayList()` method is called by the `LayoutManager` and we check to see if our chart flag has changed. If the value has changed, we then draw the chart to the UI.

At first glance, it may seem that we have added a lot of code to do such a simple process, but by breaking out the code into their respective Phases we save ourselves a lot of potential heartache and guarantee much better performance right out of the box.

Using And Accessing Styles

The last best practice we are going to examine is accessing and setting styles in a Flex component. This will be a brief examination of styles because the Style system in Flex deserves its own paper due to the nature and complexity of the provided functionality. In this section we will simply be looking at where and when you should access the properties.

Setting styles can be done at almost any time because the `setStyle()` method is smart enough to store values, determine if it inheriting, etc. Yet, just because you can set a style anywhere, does not mean that it's a good practice.

When configuring your default styles for a component, it may be better to do this post-Addition so that you can check to see if the style has been set previously, before defining a default property. One recommended location would be in the `childrenCreated()` method. This allows you to access defined styles and set styles on your now created children.

Accessing styles is a different issue. Styles are not defined until `addChild()` is called on the `UIComponent`. This method is responsible for looking up applied styles via Themes and inheritance. If you try to use `getStyle()` before then you will have very unpredictable results. This also means that your component must have a parent before you can properly start accessing styles that have been defined.

Applying Styles

It's important to note that styles should not be applied (i.e. rendered) until the Validation Phase. When developing custom content, children, drawing UI (such as our example in The Invalidation-Validation Cycle and Methods) we should wait until the `updateDisplayList()` method is called so that we can properly access and then render the applied styles to our UI.

³⁴ Remember, our `commitProperties()` method may be called by many other property/value changes from either our code or the Framework code. Therefore, we always want to determine what has changed during the last Invalidation Phase before executing any logic.

To help in this process the Flex Component architecture has provided a method called `styleChanged()` which we can override to flag our style changes for our Validation Phase.

Continuing with our `dataValue` component example, lets say that our component exposes a style called `chartLineColor` which is used by the `drawChart()` method to determine the line style color.

```
[Style(name="chartLineColor",type="uint",format="Color",inherit="no")]
```

Figure 15 - Style Metadata Example

We want to make sure that when the style changes that our chart is re-drawn and the new color is applied. We do this by overriding the `styleChanged()` method:

```
override public function styleChanged(styleProp:String):void {
    super.styleChanged(styleProp);
    switch(styleProp) {
        case "chartLineColor":
            _chartDirty = true;
            invalidateDisplayList();
        break;
    }
}
```

Figure 16 - Style Changed Example

When ever a style property changes, the `styleChanged()` method is called³⁵ passing in the name of the style that changed. In our example code, we look for the `chartLineColor` style and if that has changed, we mark the chart as dirty and then invalidate the display list. When the next Validation Pass occurs, our chart will be marked dirty and the `drawChart()` method will be called³⁶.

This is just a basic example of how we can continue leveraging the Invalidation-Validation phases during our application development. There are many ways we can use this phase to modify the look and feel of our components.

³⁵ It's important to mention that `styleChanged()` is called when a component's style property is directly set via MXML during the birth phase of the component. This method is not called for any properties that are set via inheritance or Themes. When implementing styles in your component, you should make sure to get and apply all the styles for the component during the first Validation pass and not rely on the style changed method to flag your styles to determine which to apply.

³⁶ We will assume that the `drawChart()` method looks up the associated styles and applies them.

Conclusion

We hope this paper has been educational and maybe even an eye-opening experience for you about that Flex Component and Application Lifecycle. As mentioned before, this is our first review and analysis of the lifecycle from a code perspective and our best practices have been defined from both experience and research. If you have questions or comments about this document please contact us at info@developmentarc.com because we would greatly appreciate your feedback.

The Future of Flex Components

As of this writing, the Flex 4 Framework is currently under development and the Adobe team is making massive improvements and changes to the overall Flex Framework architecture. Some of you may question the validity of this document due to the upcoming release of Flex 4. We have begun analysis of the Flex 4 component architecture and much of what is written here will still be valid and will not change with the new release.

For example, the new `SkinnableComponent` class extends from the existing `UIComponent`, the `SystemManager` has been slightly updated but still appears to use the same load order, and Halo (Flex 3) components can and will be used inside many Flex 4 applications. We intend to write a follow-up document that examines the differences between Flex 3 and 4 and how it will affect any information in this document. We want to thank you for taking the time (and effort) to read this paper and hopefully it has revealed some of the inner workings of Flex.