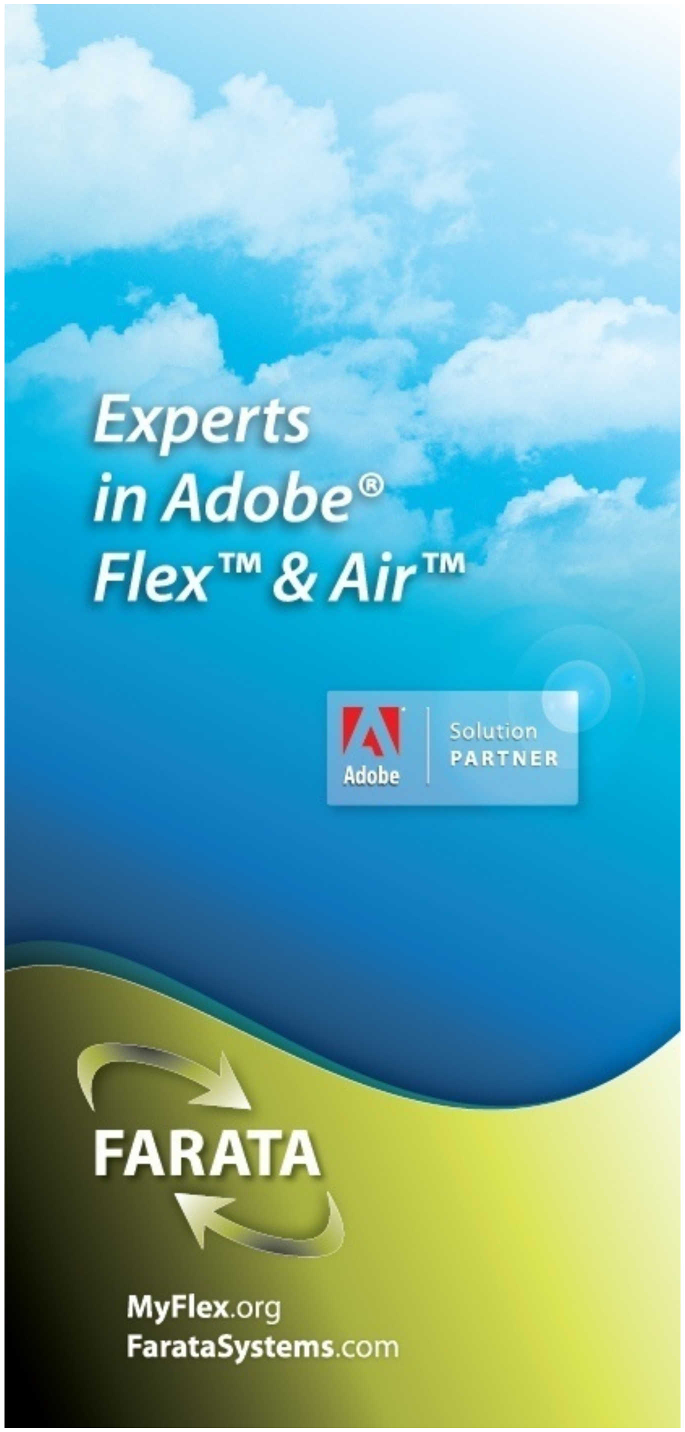
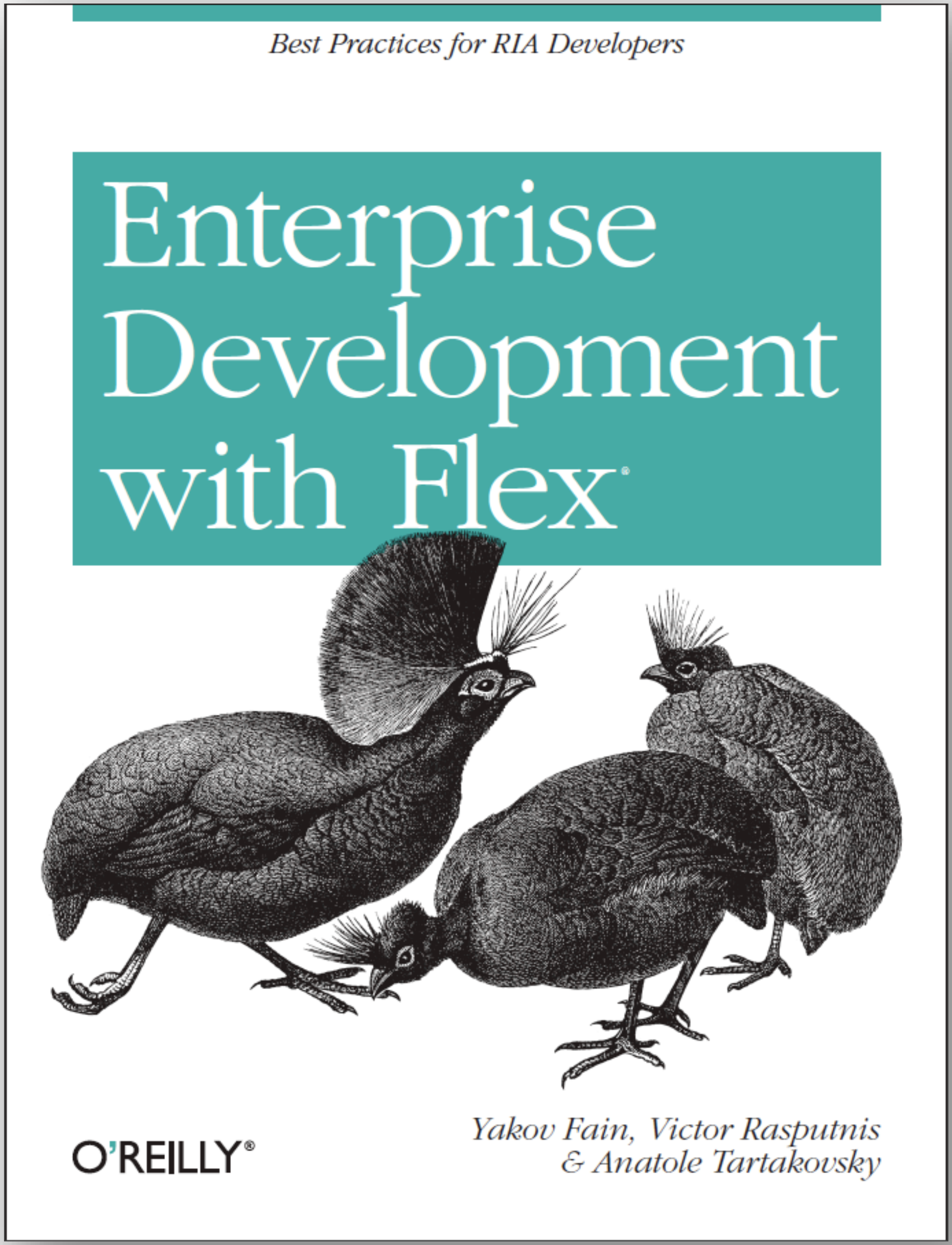


Flex Modularization

By Yakov Fain


Best Practices for RIA Developers

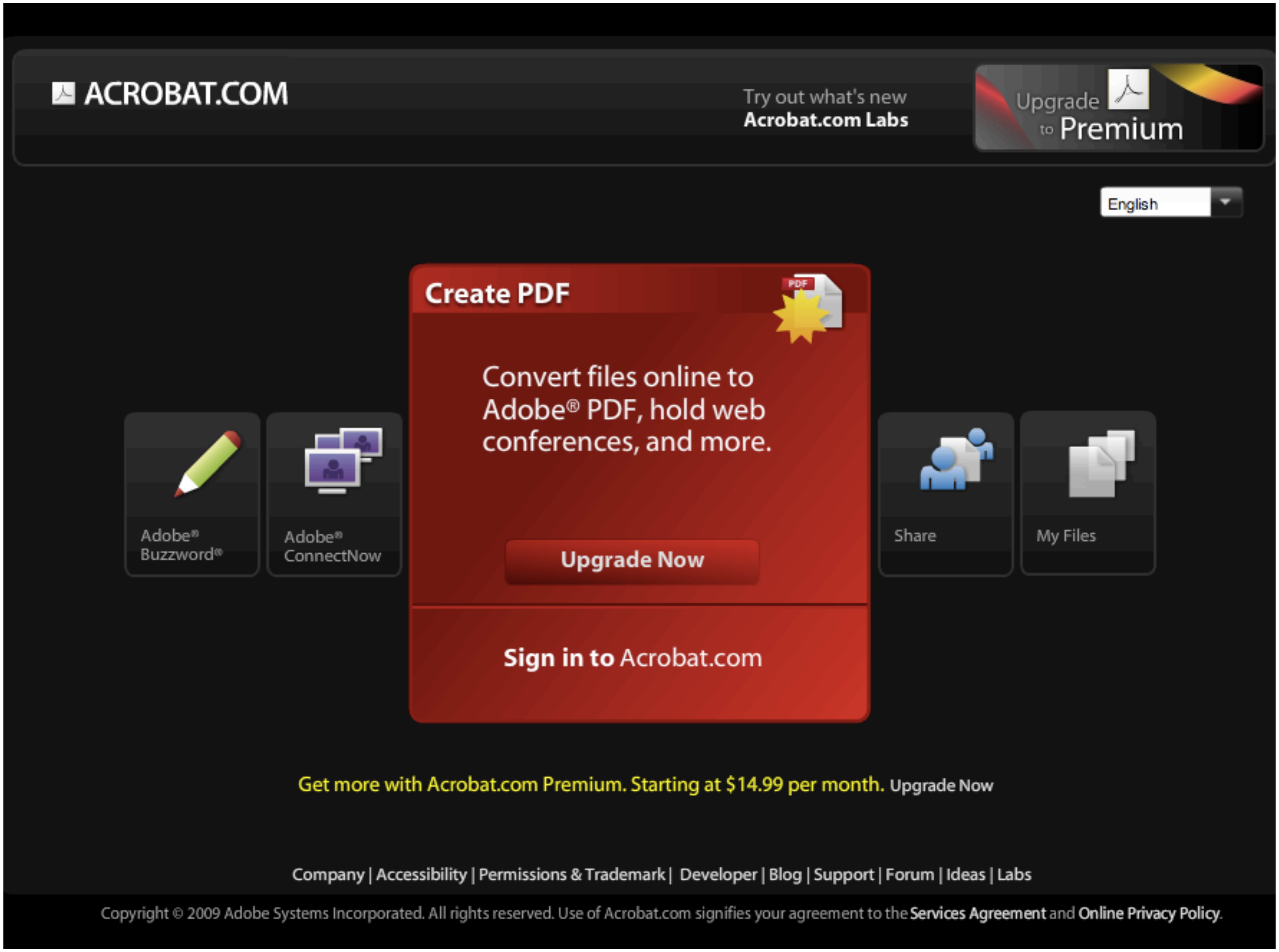
Enterprise Development with Flex

Yakov Fain, Victor Rasputnis & Anatole Tartakovsky

O'REILLY

You can find detailed coverage of Flex modularization in chapter 7 of this book

Perceived Performance and the 1st screen



Convert files online to Adobe® PDF, hold web conferences, and more.

Get more with Acrobat.com Premium. Starting at $14.99 per month. Upgrade Now

Copyright © 2009 Adobe Systems Incorporated. All rights reserved. Use of Acrobat.com signifies your agreement to the Services Agreement and Online Privacy Policy.

Flex Applications, Modules, Libraries

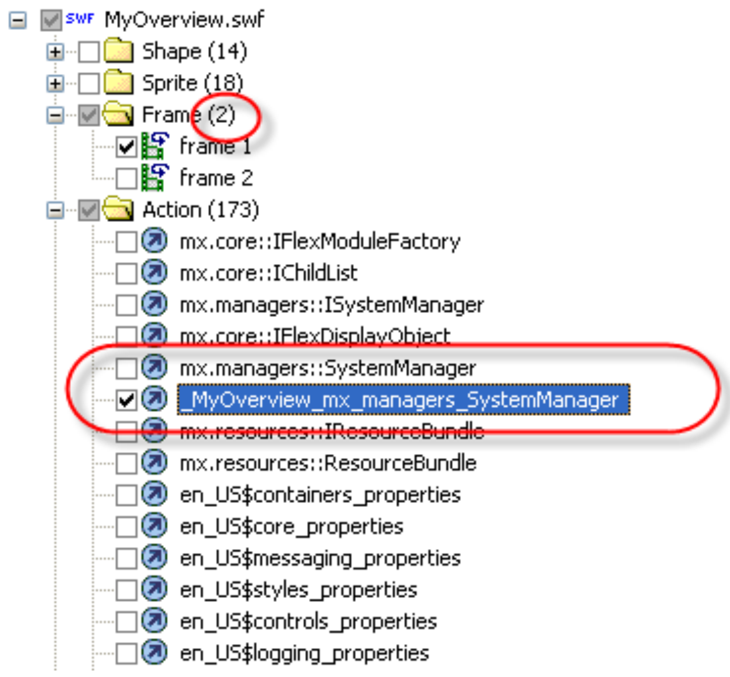
- Applications and Modules have two frames
- Modules must be explicitly loaded by ModuleLoader or ModuleManager
- Libraries are not loaded by application code – they are linked
- The main SystemManager class is located in the first frame of an application or module's SWF
- The second frame contains all the rest - Flex framework classes, user application classes, embedded assets (images, fonts, etc)
- The library SWF has only one frame with a bunch of classes, without SystemManager

An SWC (similarly to a JAR) is an archive file that contains library.swf and catalog.xml files. The SWF must be extracted on deployment.

Peeking inside Application & Module

- Application and Modules are two frame SWFs
- Frame 1 or 2 in a Flex Aplication is controlled by a SystemManager class – the entry point to your application

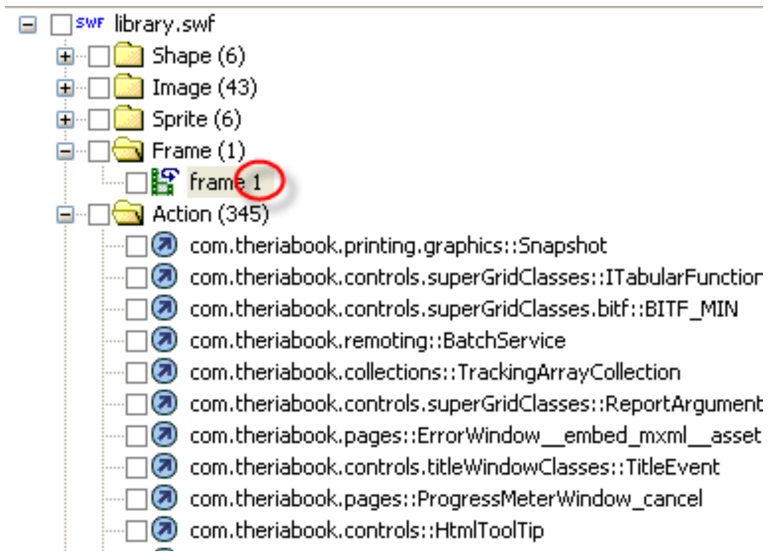
Sothink SWF Decompiler →



```
MyOverview.swf
  Shape (14)
  Sprite (18)
  Frame (2)
    frame 1
    frame 2
  Action (173)
    mx.core::IFlexModuleFactory
    mx.core::IChildList
    mx.managers::ISystemManager
    mx.core::IFlexDisplayObject
    mx.managers::SystemManager
    _MyOverview_mx_managers_SystemManager
    mx.resources::IResourceBundle
    mx.resources::ResourceBundle
    en_US$containers_properties
    en_US$core_properties
    en_US$messaging_properties
    en_US$styles_properties
    en_US$controls_properties
    en_US$logging_properties
```

Peeking inside a Library SWF

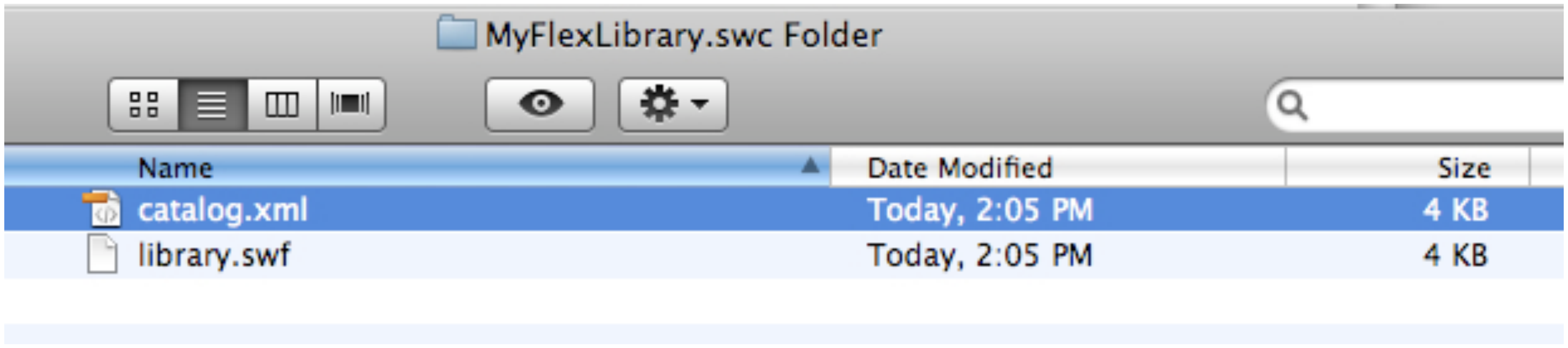
Library SWFs do not have the initialization frame



```
library.swf
    Shape (6)
    Image (43)
    Sprite (6)
    Frame (1)
        frame 1
    Action (345)
        com.theriabook.printing.graphics::Snapshot
        com.theriabook.controls.superGridClasses::ITabularFunctior
        com.theriabook.controls.superGridClasses.bitf::BITF_MIN
        com.theriabook.remoting::BatchService
        com.theriabook.collections::TrackingArrayCollection
        com.theriabook.controls.superGridClasses::ReportArgument
        com.theriabook.pages::ErrorWindow__embed_mxml__asset
        com.theriabook.controls.titleWindowClasses::TitleEvent
        com.theriabook.pages::ProgressMeterWindow_cancel
        com.theriabook.controls::HtmlToolTip
```

How to create an SWC file?

- Create the Flex Library project in Flash Builder
- Compile the code with the compc compiler



Name	Date Modified	Size
catalog.xml	Today, 2:05 PM	4 KB
library.swf	Today, 2:05 PM	4 KB

Let’s consider MyFlexApplication.swf application that needs to use Some classes from the library MyFlexLibrary.swc.

What are your linkage choices?

Library Linkage Types

Merge-in: Only those entities from library.swf that are explicitly referenced in MyFlexApplication.swf get embedded directly into the application's SWF.

External: No entities from library.swf are included in the body of the MyFlexApplication.swf. **Assumption:** by the time MyFlexApplication needs to create instances of classes from the library.swf the definitions for these classes will be already loaded into the currentDomain.

RSL: The entities contained in library.swf won't be included in the body of the MyFlexApplication.swf. But as opposed to *External*, all definitions originally contained in the library.swf will be loaded into the main application domain during the application startup. RSLs are preloaded by the SystemManager

How much of an application code must arrive from the server if you have the following swf and swc?

MyFlexApplication .swf weighs 500Kb

MyFlexLibrary.swc weighs 100Kb

Case 1: Merged Into Code:

500Kb+n, where n<=100Kb

Only those library classes that are mentioned in the app get merged in

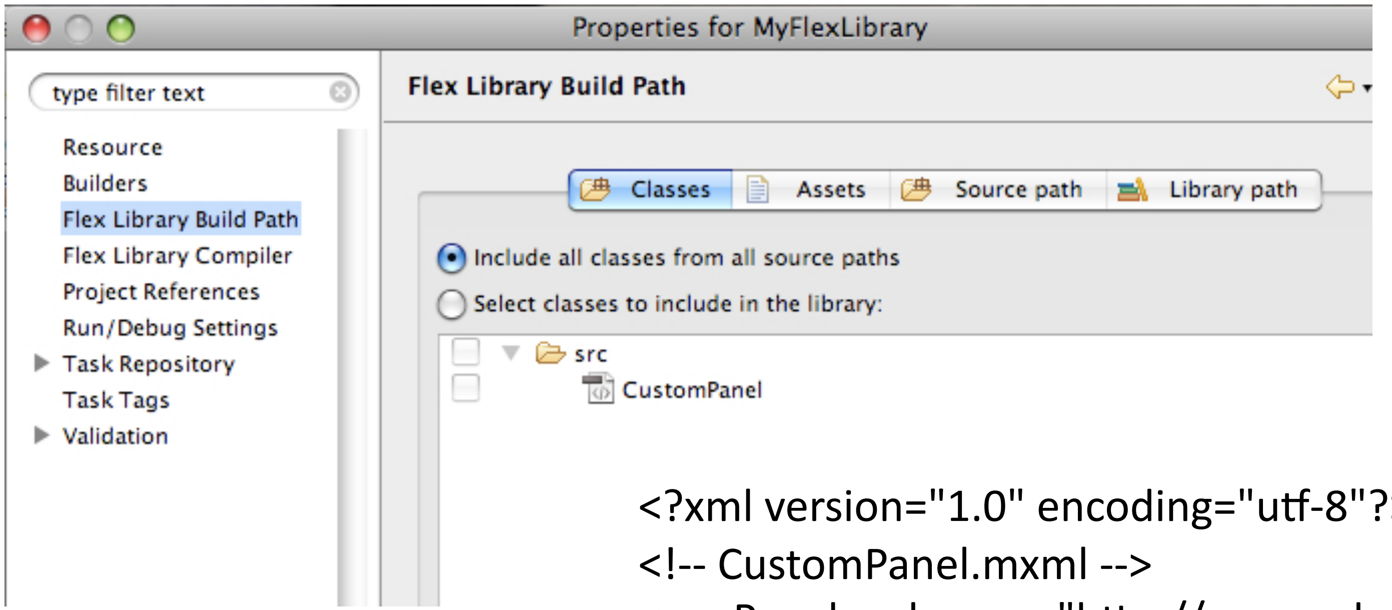
Case 2: RSL:

500Kb+100Kb=600Kb

Do you think I'm stupid or something to use RSL?

Quiz: How many SWFs get deployed in each case?

CustomPanel in MyFlexLibrary



[✓ Include all classes from all source paths]

Select classes to include in the library:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- CustomPanel.mxml -->
<mx:Panel xmlns:mx="http://www.adobe.com/2006/mxml"
title="'Custom' Panel #{instanceNumber}"
width="300" height="150"
creationComplete="instanceNumber=++count;"
>
<mx:Script>
public static var count:int;
      [Bindable]
       private var instanceNumber:int;
   </mx:Script>
</mx:Panel>
```

RSL Caveat

- Currently, RSL’s don’t allow dynamic linking of your application classes

- Library project yields two code artifacts:
  1. *library.swf* inside the SWC file,
  2. Mixin to the application or module that includes the library a.k.a. bootstrap code

- The bootstrap code initializes (library) classes that are statically referenced by the application or a module. It **does not** provide this service for classes that you want to reference dynamically.

- UsingLibrary

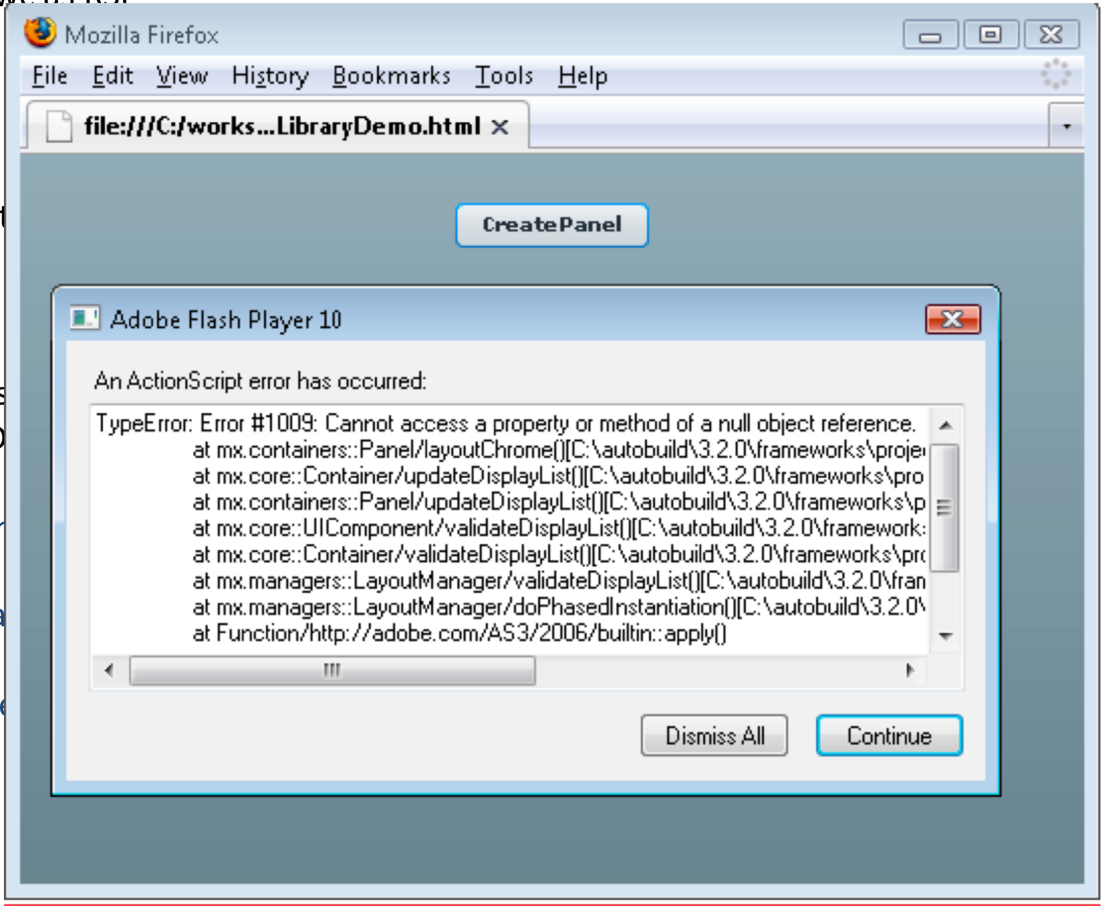
An Illustration of RSL Caveat

Not knowing about Panel, Flex compiler omits creation and use of _ControlBarStyle and _PanelStyle mixins. This leads to uninitialized *titleBackgroundSkin* in Panel.as and, ultimately – to a reference error inside Panel::layoutChrome() method

```
// Compiler-generated SystemManager for the LibraryDemo
package { …
        public class _LibraryDemo_mx_managers_SystemManager extends
        mx.managers.SystemManager   implements IFlexModuleFactory {
         . . .
        override    public function info():Object {
                return {
           . . .
           mainClassName: "LibraryDemo",
           mixins: [ "_LibraryDemo_FlexInit", "_richTextEditorTextAreaStyleStyle",
        ~~"_ControlBarStyle",~~ "_alertButtonStyleStyle", "_textAreaVScrollBarStyleStyle", "_headerDateTextStyle",
        "_globalStyle", "_todayStyleStyle", "_windowStylesStyle", "_ApplicationStyle", "_ToolTipStyle",
        "_CursorManagerStyle", "_opaquePanelStyle", "_errorTipStyle", "_dateFieldPopupStyle", "_dataGridStylesStyle",
        "_popUpMenuStyle", "_headerDragProxyStyleStyle", "_activeTabStyleStyle",~~"_PanelStyle",~~ "_ContainerStyle",
        "_windowStatusStyle", "_ScrollBarStyle", "_swatchPanelTextFieldStyle", "_textAreaHScrollBarStyleStyle",
        "_plainStyle", "_activeButtonStyleStyle", "_advancedDataGridStylesStyle", "_comboDropdownStyle",
        "_ButtonStyle", "_weekDayStyleStyle", "_linkButtonStyleStyle" ],
                    rsls: [{url: "ComponentLibrary.swf", size: -1}]
                }
      }
}}
```

Not truly dynamic way of loading of objects from RSL

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx=http://www.adobe.com/2006/mxml layout="vertical" >

     <!–
            1. Declare ComponentLibrary.swc as RSL
            2. import mx.containers.Panel;
             var someDummyPanel:Panel;
      -->

  <mx:Button label="CreatePanel" click="creat
     <mx:Script>
          <![CDATA[

     //      import mx.containers.Panel; var s
               private var displayObject:D

               private function createCom

                    var clazz : Class = loa
                    displayObject = Disp
                    addChild(displayObj
               }
          ]]>

     </mx:Script>
</mx:Application>
```


Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/works...LibraryDemo.html
CreatePanel
Adobe Flash Player 10
Dismiss All
Continue

An ActionScript error has occurred:

```
TypeError: Error #1009: Cannot access a property or method of a null object reference.
	at mx.containers::Panel/layoutChrome()[C:\autobuild\3.2.0\frameworks\proje
	at mx.core::Container/updateDisplayList()[C:\autobuild\3.2.0\frameworks\pro
	at mx.containers::Panel/updateDisplayList()[C:\autobuild\3.2.0\frameworks\p
	at mx.core::UIComponent/validateDisplayList()[C:\autobuild\3.2.0\framework:
	at mx.core::Container/validateDisplayList()[C:\autobuild\3.2.0\frameworks\pr
	at mx.managers::LayoutManager/validateDisplayList()[C:\autobuild\3.2.0\fran
	at mx.managers::LayoutManager/doPhasedInstantiation()[C:\autobuild\3.2.0\
	at Function/http://adobe.com/AS3/2006/builtin::apply()
```

Truly dynamic way of loading of objects from RSL

Replace SWF produced by *compc* compiler (or build of the Flex Library Project) with your own compiled by *mxmlc*:

1. Turn off “AutoExtract” linking option. Library’s SWF will not be extracted on each build of the application SWF

2. Add to the project a descendant of the *mx.core.SimpleApplication* that statically references all library classes. Wrap it with MXML to force compiler into generating mix-ins. Compile the project with Ant **with *mxmlc* compiler** – as an application. Now you have library that takes care of itself.

3. Drop this autonomous library SWF at the deployment location instead of the autoExtracted one.

We call this technique building **self-initialized or bootstrap libraries**

Bootstrapping the RSL

```
package {
import mx.core.SimpleApplication;
public class ComponentLibrary_Application extends SimpleApplication {
  import com.farata.samples.CustomPanel; CustomPanel; //forced link
  public function ComponentLibrary_Application() {
       trace(“Library initialized"); // Place you custom init code here
 }}}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ComponentLibrary_Bootstrap.mxml
     By virtue of  MXML we force compiler to generate all mix-ins required by
     the   library  classes  (for the generated bootstrap class
-->
<ComponentLibrary_Application xmlns="*" />
```

Loaders are tools for partitioning

- A SWF of the main application can load classes located in other SWFs packaged as:
  - modules
  - libraries
  - Applications
- The class flash.display.Loader loads
  - Images
  - Styles
  - Modules
  - Applications
- App. partitioning = dynamic class loading

Loading an Image

- Dynamic Loading:

```
<mx:Image source="assets/logo.png"/>
```

- Embedding

```
<mx:Image source="@Embed('assets/logo.png')"/>
```

- Explicit Reference: Embedding with reuse

```
 <mx:Script><![CDATA[
   [Embed(source="assets/farata_logo.png")]
   [Bindable] private var logoClass:Class; //←BitmapAsset
]]></mx:Script>

<mx:Image source="{logoClass}"/>
<mx:Button icon="{logoClass}"/>
```

Dynamic Image Loading

```
// Loading the image (transfer the code: URLLoader into a ByteArray)

[Bindable] private var imageData:ByteArray;
private function loadImage():void {
    var urlRequest:URLRequest =  new URLRequest(IMAGE_URL);
    var urlLoader:URLLoader = new URLLoader();
    urlLoader.dataFormat = DataFormat.BINARY;
    urlLoader.addEventListener("complete",onComplete);
    urlLoader.load(urlRequest);
}
private function onComplete(event:Event):void{
    var urlLoader:URLLoader = event.target as URLLoader;
    imageData = urlLoader.data as ByteArray;
}
. . .
// Creation of the Adding the image to the stage
 <mx:Button label="Load Image" click="loadImage()" />
<mx:Image id="image" source="{imageData}"/>
```

ByteCode vs. Class Definitions

- Ultimate subjects of the dynamic loading are class definitions, be that definitions of assets or components
- Transfer of the byte code and actual creation of class definitions are two separate actions
  - Transfer: URLLoader
  - Creation of class definition & adding to stage: Image

- ModuleLoaders:ImageDemo

Dynamic Styles

```
/* styles.css */
Application {
    background-image:Embed("assets/background.png") ;
    background-size:"100%" ;
}
. . . .
controlBarPanelStyle {
    border-style: none ;
    fillColors: #4867a2, #4f75bf ;
    border-skin: ClassReference("border.SimpleGradientBorder");
}
```

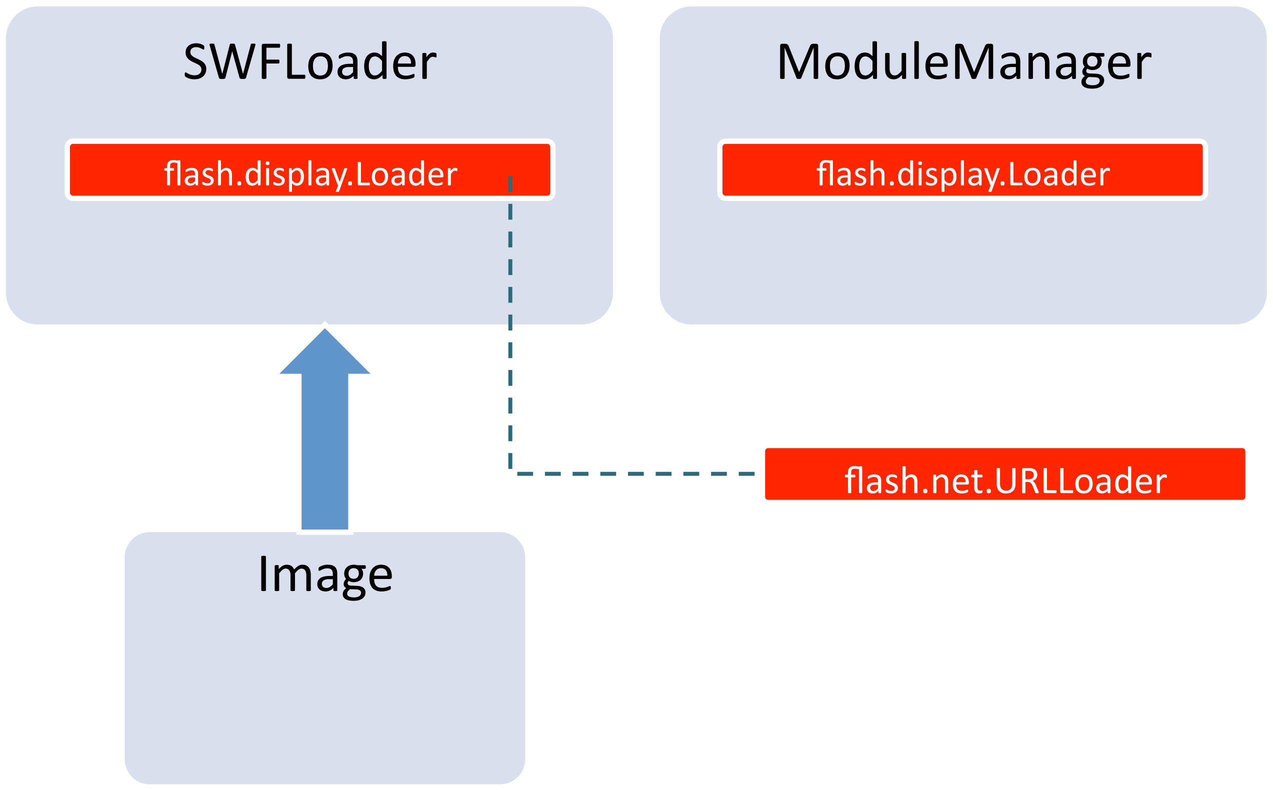
Compile styles.css into styles.swf and load the SWF

Dynamic Styles: Meet the Managers

```
import mx.modules.IModuleInfo;
import mx.modules.ModuleManager;

private function toggleStyles():void {
  var moduleInfo:IModuleInfo =ModuleManager.getModule
      ('styles.swf');

  if (moduleInfo.loaded) {
    StyleManager.unloadStyleDeclarations('styles.swf');
  } else {
    StyleManager.loadStyleDeclarations('styles.swf');
  }
}
```

ModuleLoaders: RuntimeStylesDemo

All Roads Lead To Loader



All work is done by Loader optionally assisted by URLLoader

You can load modules with

- ModuleLoader

- ModuleManager

Modules: ModuleLoader

In case of ModuleLoader, all heavy lifting is done by flash.display.Loader:

```
<mx:Button label="Load Module"
   click="moduleLoader.loadModule('SimpleModule.swf')" />
<mx:Button label="Unload Module“
   click="moduleLoader.unloadModule()“
/>
<mx:ModuleLoader id="moduleLoader"/>
```

Preloading Modules: ModuleManager:

- Provides a singleton type of modules’ registry
- Separates module loading from instantiation
- Guarantees that module bytes are downloaded only once

```
 private var moduleInfoRef:Object = {}; // trick GC to keep moduleInfoRef alive

private function loadModule():void {
 var info:IModuleInfo  = ModuleManager.getModule(moduleUrl);
 info.addEventListener(ModuleEvent.READY, onModuleReady ) ;
 moduleInfoRef[moduleUrl] = info;
 info.load();

private function onModuleReady(event:ModuleEvent):void {
    moduleInfoRef[event.module.url]=null;
}
```

Module Manager: Instantiation

```
private function createModuleInstance(moduleUrl:String, parent:UIComponent=null):Module {
        var module:Module;
        var info:IModuleInfo  = ModuleManager.getModule(moduleUrl);
        var flexModuleFactory:IFlexModuleFactory = info.factory;
        if (flexModuleFactory != null) {
          module = flexModuleFactory.create() as Module;
          if (parent) {
                    parent.addChild(module);  // in Flex 4: addElement(module);
          }
        }
        return module;
}
```

ModuleManagerDemo

Flex 4: StyleManager

- StyleManager is not a singleton any longer. You may even unload the module now.
- A module or a sub-application has its own StyleManager
- Style properties are inherited and merged
- To get a local StyleManager and load styles:

```
private var mod1StyleManager:IStyleManager2 = null;
mod1StyleManager=StyleManager.getStyleManager(
                                        this.moduleFactory);
eventDispatcher = mod1StyleManager.loadStyleDeclarations
("mod1Styles.swf");
```

Dynamic Styles

```
/* styles.css */

@namespace s "library://ns.adobe.com/flex/spark";
@namespace mx "library://ns.adobe.com/flex/halo";  // halo is mx now

s|Application {
    skin-class:ClassReference("skins.ApplicationSkin");
}
s|Button.leftArrow {
    skin-class:ClassReference("skins.LeftArrowButtonSkin");
}
.controlBarPanelStyle {
    skin-class: ClassReference("skins.GradientRectSkin");
}
```

Dynamic Styles: Meet the Managers

```
private const STYLE_MODULE_URL:String = '/Modules/styles.swf';
private function toggleStyles():void {
    var localStyleManager:IStyleManager2 = StyleManager.getStyleManager(
        this.systemManager
    );
    try {
        var moduleInfo:IModuleInfo = ModuleManager.getModule
        (STYLE_MODULE_URL);
    } catch (e:Error) {trace("Check that " + STYLE_MODULE_URL + " exists");}

    if (moduleInfo.loaded) {
            localStyleManager.unloadStyleDeclarations(STYLE_MODULE_URL);
    } else {
            localStyleManager.loadStyleDeclarations(STYLE_MODULE_URL);
    }
}
```

RuntimeStylesDemo

Module Manager: Instantiation

```
private function createModuleInstance(moduleUrl:String,
                                      parent:UIComponent=null):Module {
     var module:Module;
     var info:IModuleInfo  = ModuleManager.getModule(moduleUrl);
     var flexModuleFactory:IFlexModuleFactory = info.factory;

     if (flexModuleFactory != null) {
       module = flexModuleFactory.create() as Module;
       if (parent) {
            parent.addChild(module);  // Flex 4: addChild();
       }
     }
     return module;
}
```

ModuleManager Demo

Communicating With Modules

- Bad practice: direct name reference
- Slightly better: code to interfaces

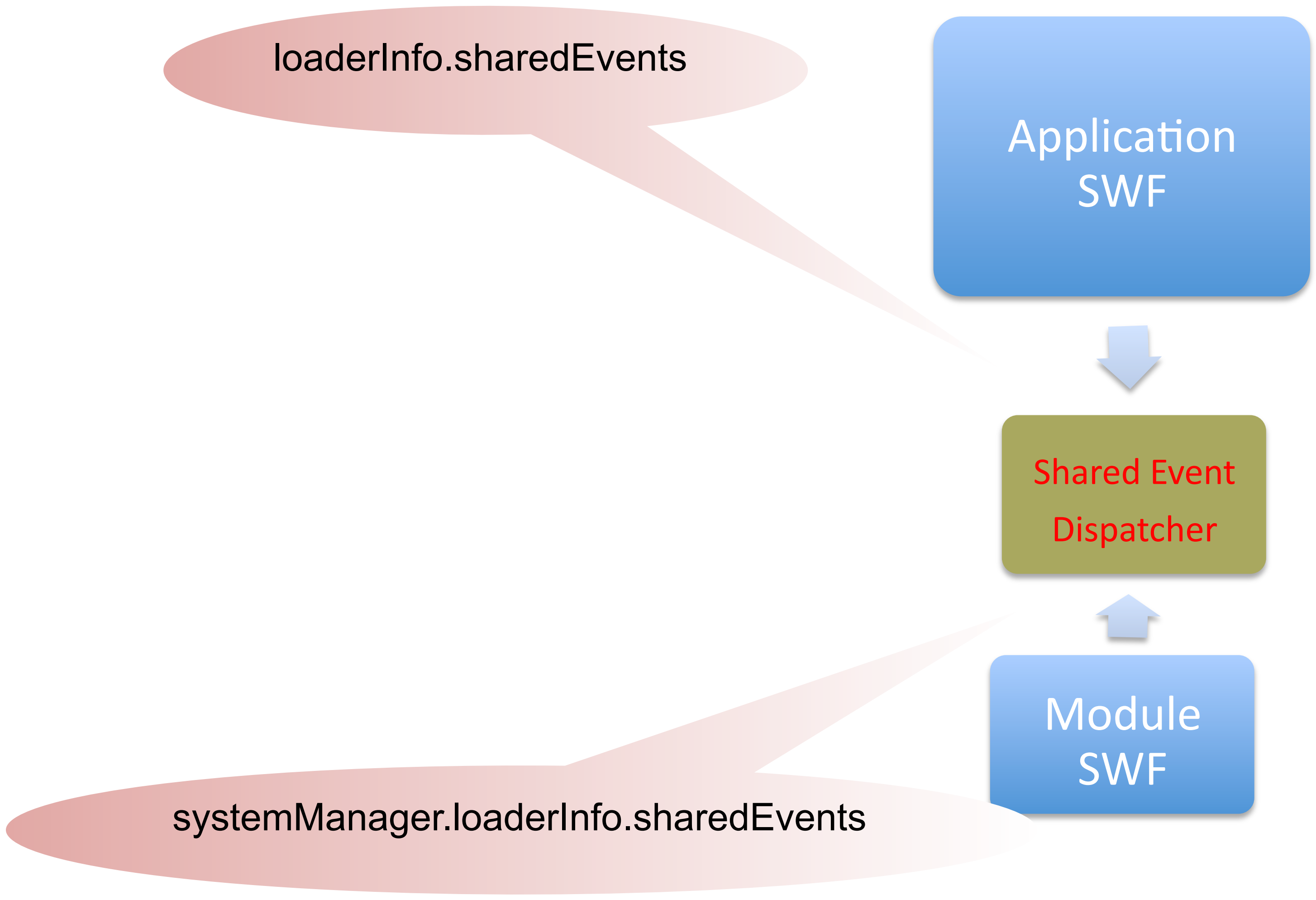
But any direct referencing might cause memory leaks

<u>ReferenceCommunicationDemo</u>

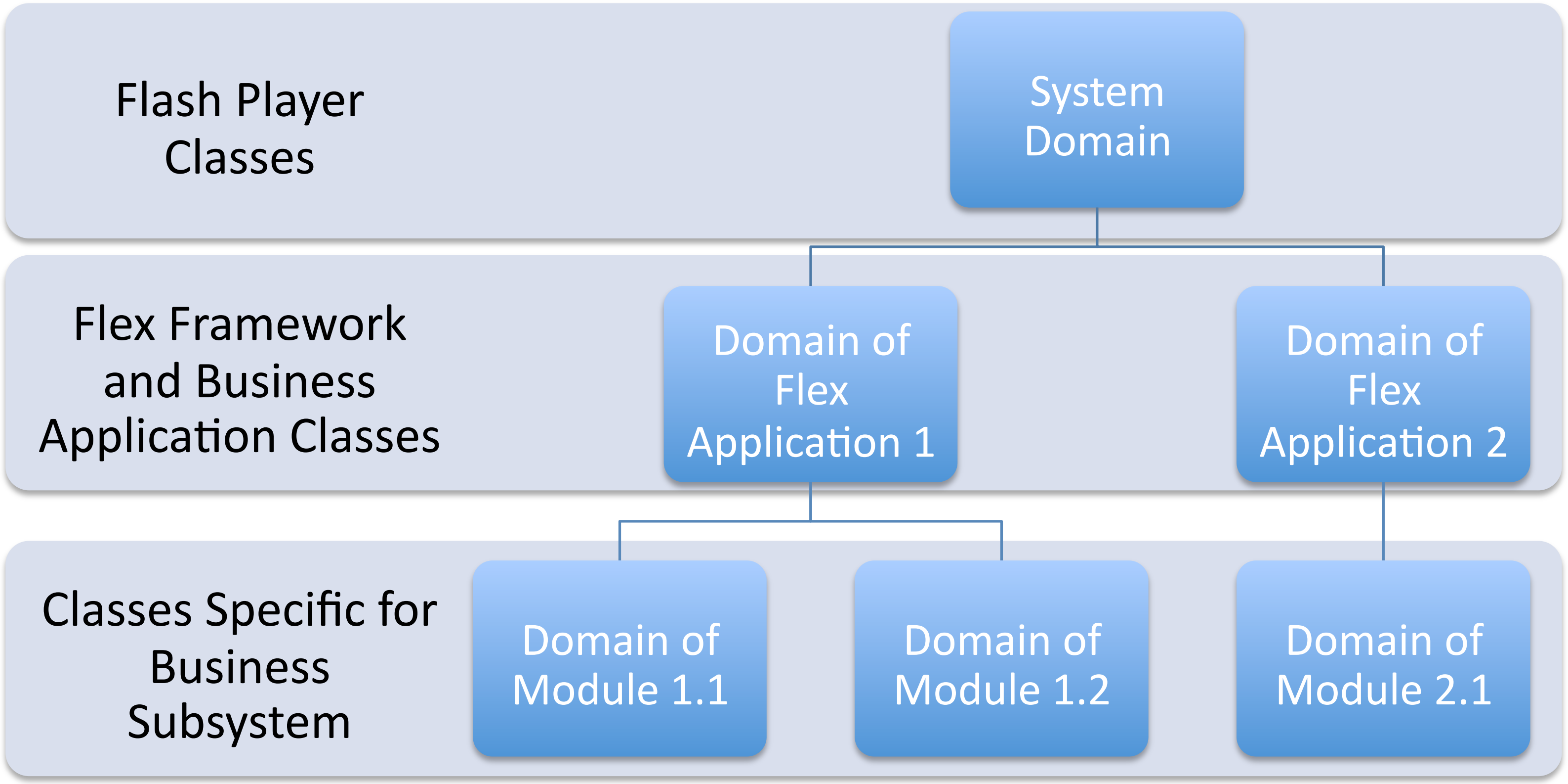
- Best: exchange events via ***loaderInfo.sharedEvents***

<u>EventCommunicationDemo</u>

Shared Event Dispatcher

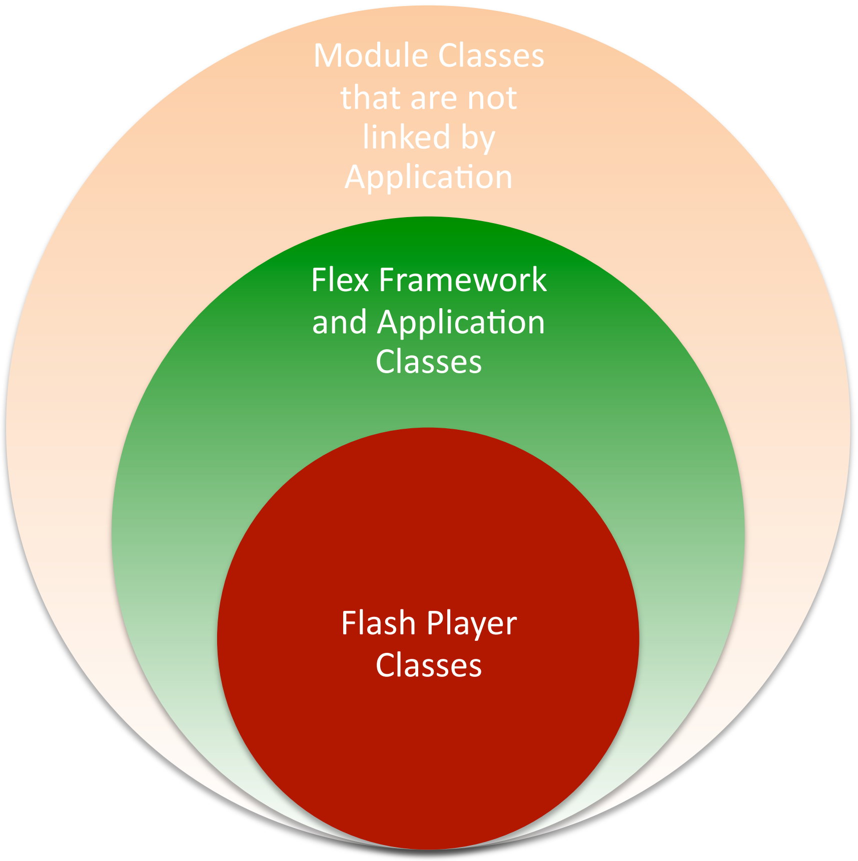


A Tree of Application Domains


Flash Player Classes
System Domain
Flex Framework and Business Application Classes
Domain of Flex Application 1
Domain of Flex Application 2
Classes Specific for Business Subsystem
Domain of Module 1.1
Domain of Module 1.2
Domain of Module 2.1

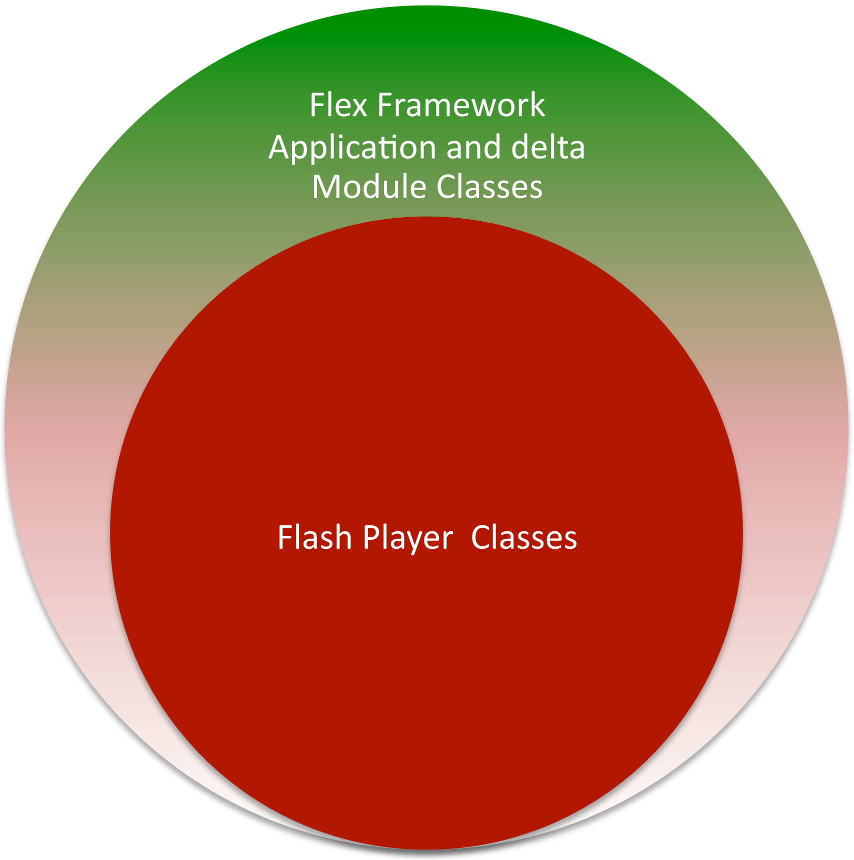
- Classes get loaded into Application Domains
- Child domain classes “see” parent’s classes; re-load of the same class is not possible (unless via unload)

Loading Modules into Child Domain (Default)


Module Classes that are not linked by Application
Flex Framework and Application Classes
Flash Player Classes

```
<mx:ModuleLoader applicationDomain=”{
      new ApplicationDomain(
            ApplicationDomain.currentDomain
      )
 }"  url=”ModuleToLoad.swf"/>
```

Loading Modules into Same Domain



```
<mx:ModuleLoader applicationDomain=”{
    ApplicationDomain.currentDomain
}"  url=”ModuleToLoad.swf"/>
```

ModuleDomainDemo

- Application will have access to the module classes when module is explicitly loaded in the application’s domain
- Dynamic class instantiation:

```
try {
	var clazz:Class = loaderInfo.applicationDomain.getDefinition
							("CustomGrid") as Class;
} catch (error:ReferenceError) {
	Alert.show ("Definition of 'CustomGrid' class can not be found
in the current domain of the application ","Class Not Found Error");
	return;
}

dg  = DataGrid(new clazz());
```

Modules Recommendations

- Use Modules to package
  - Subsystems of the enterprise applications that you would be able to recompile from sources for every release
- Applications load modules explicitly either in child application domain (default) or in the same application domain
- Load modules into the same domain to achieve the seamless integration (can’t be unloaded)
- Manage memory by focusing on unloading arrays and images that take real space, rather then on unloading modules.
- Beware: [RemoteObject] make modules unloadable

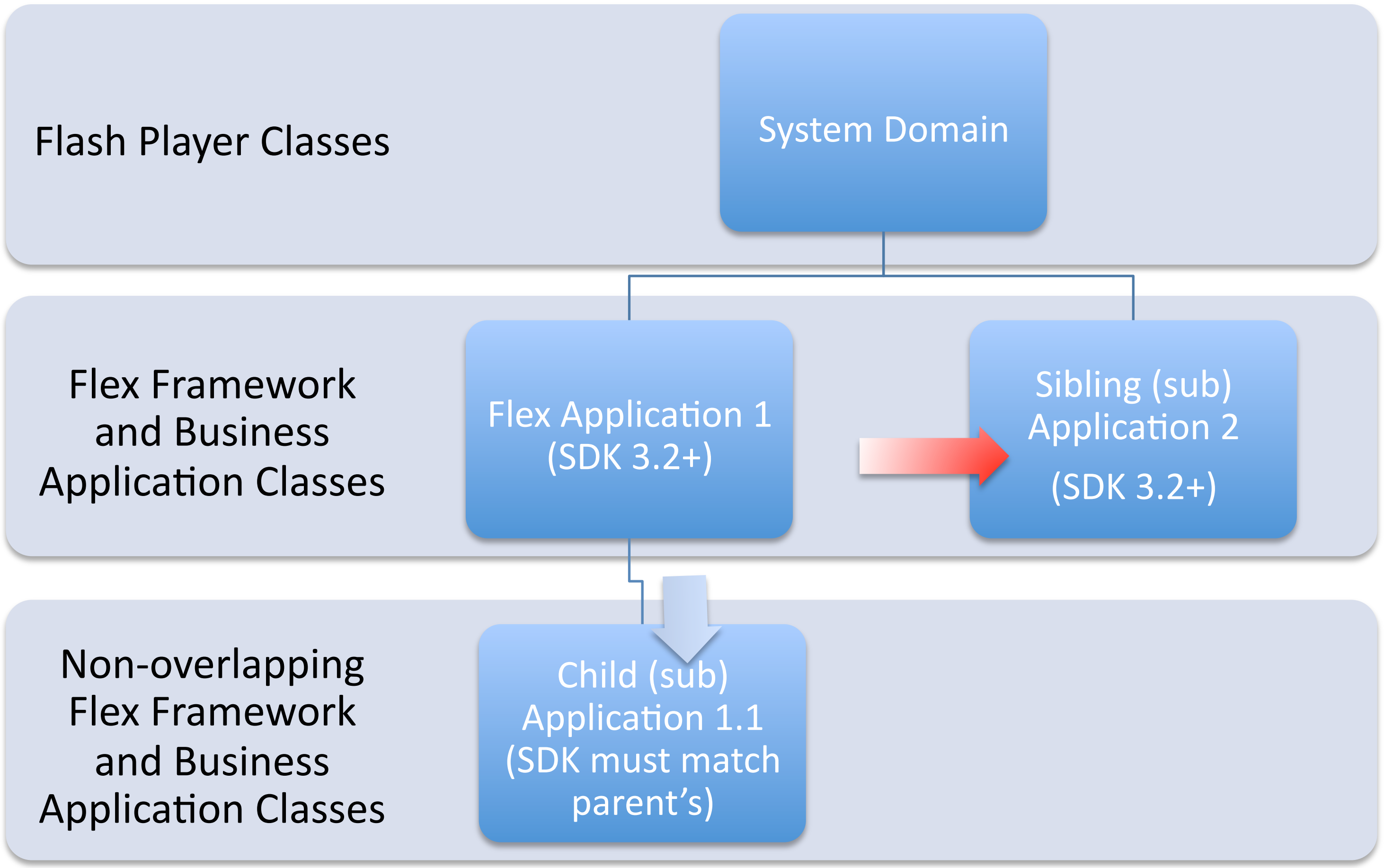
Modules Recommendations (cont.)

1. Modules are notorious for rewriting the global application settings, so
   - Compile module without *–services* option to avoid rewrite of the channels and destinations
   - **In Flex3** reuse the *same* style definition file (good) or compile and load *style modules (*best) to avoid rewriting the global CSS styles by each module. Style clashes are fixed in **Flex4** with **per module** StyleManager

2. Do not restrain yourself by loading modules into the child domain. If your ultimate goal is class isolation, you are better off with sub-applications
3. Manage memory by focusing on **unloading data and images** that take more space, then module’s code.

4. Provide a test harness for local debugging and unit test of the module in the isolation from the main Flex project

Loading Application as Siblings



Flash Player Classes
System Domain
Flex Framework and Business Application Classes
Flex Application 1 (SDK 3.2+)
Sibling (sub) Application 2 (SDK 3.2+)
Non-overlapping Flex Framework and Business Application Classes
Child (sub) Application 1.1 (SDK must match parent's)

```
<mx:SWFLoader                                id="swfLoader”
loadForCompatibility="true" /> or
swfLoader.loaderContext= new LoaderContext( false,
    new ApplicationDomain(null)
);
```

Loading Scenarios
Inside the Same Web Domain

Loader Context Syntax	SWFLoader Syntax	Scenarios

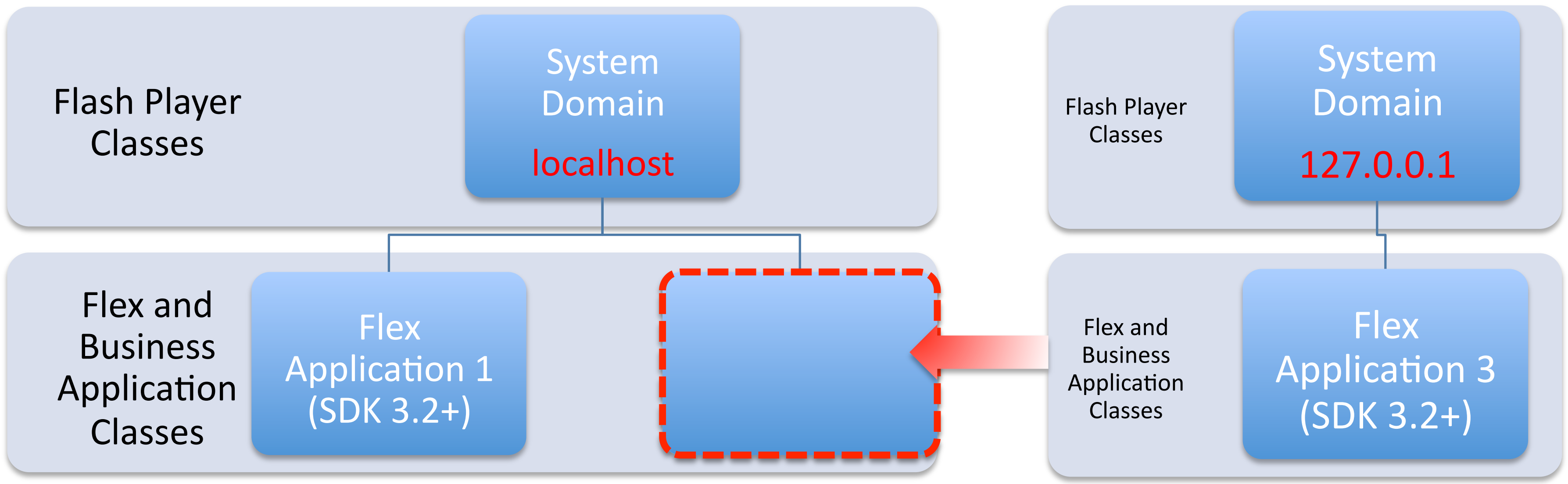
`swfLoader.loaderContext=new`
`LoaderContext( false,`
`new ApplicationDomain(null)`
`);`
	`<mx:SWFLoader`
	`id="swfLoader"`
	`loadForCompatibility="true"`
	`/>`
		SS*
		D_ifferent
		D_omain (sibling)
		
		Multiversioning

`swfLoader.loaderContext=new`
`LoaderContext( false,`
`new ApplicationDomain(`
`ApplicationDomain.currentDomain`
`)`
`);`
	`<mx:SWFLoader`
	`id="swfLoader"`
	`/>`
		SS*
		C_hild
		D_omain
		
		The default one

`swfLoader.loaderContext=new`
`LoaderContext( false,`
`ApplicationDomain.currentDomain`
`);`
	Not applicable	**SS***
		S_same
		D_omain
		
		Possible, dubious

SameSandboxChildDomainDemo - default portlet loading

Loading Application Across Web Domains



Flash Player Classes
System Domain
localhost
Flex and Business Application Classes
Flex Application 1 (SDK 3.2+)
Flash Player Classes
System Domain
127.0.0.1
Flex and Business Application Classes
Flex Application 3 (SDK 3.2+)

```
<mx:SWFLoader                              id="swfLoader"
loadForCompatibility="true"
trustContent="true" /> or
swfLoader.loaderContext= new LoaderContext( false,
                         new     ApplicationDomain(null),
SecurityDomain.currentDomain
);
```

Applications summary

- Use Applications to package
  - Subsystems of the enterprise applications that you will not be able or might not want to recompile from sources for every release: different versions of the base libraries, different versions of the Flex framework, etc.

- Loading of the (sub) applications is an explicit task of the application code

- Applications from a different net domain are automatically loaded into the sibling application domain

- Applications from the same net domain are by default loaded into the child application domain. You may want to load them into a sibling domain for complete version separation

Contact info and useful links

Email: yfain@faratasystems.com

Flex Blog: flexblog.faratasystems.com

Web site: www.faratasystems.com

Code samples: http://faratasystems.com/entflex_sc/chapter7/chapter7.zip