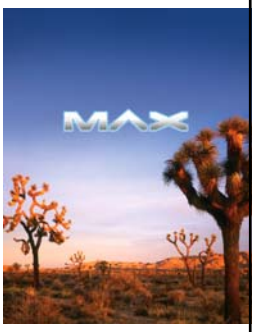
MAX 2006
Beyond Boundaries

Flex Best Practices: Applying Design Patterns and Architecture

Joe Berkovitz

Chief Architect, Allurent Inc.



Background

- 25+ years developing user-facing software
- Started in scientific data visualization and manipulation
- Educational applications for K-6
- ATG: the web application explosion
- contributor to JSF
- currently Chief Architect at Allurent, Inc.

Overview

- Why use architecture and patterns?
- What we want from a "well-built" Flex application
- A specific example: ReviewTube
- A useful division of labor in the client
- Dealing with asynchronous communication
- Some guidelines and handy ideas
- Focus on principles rather than recipes

What Is Software Architecture?

- Ideas that give “mental traction” on building stuff
- Broad solutions to large problems
- Themes that shape solutions to many small problems
- Breaking up a complex system into simpler ones with…
  - responsibilities
  - relationships
  - interactions
- Generating and characterizing diverse approaches
- Comparing their concrete strengths and weaknesses

What Are Design Patterns?

- Design Patterns are “mini-architectures”
- Unitary and easily named (Command, Session Façade…)
- Applicable to small, recurring problem spaces
- Templates of responsibilities, relationships and interactions
- Complex problem spaces sometimes map to a composition of patterns

What's a Framework?

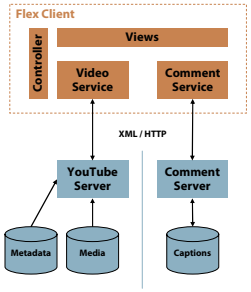
- A framework instantiates an architecture and set of patterns
- Concrete software package with classes, APIs, etc. (e.g. Flex, Cairngorm, JSF)
- Further nails down scope and nature of solution
- Specific responsibilities, relationships and interactions
- Implies set of possibilities and constraints
- What does it give you?

Flex Applications: Basic Goals

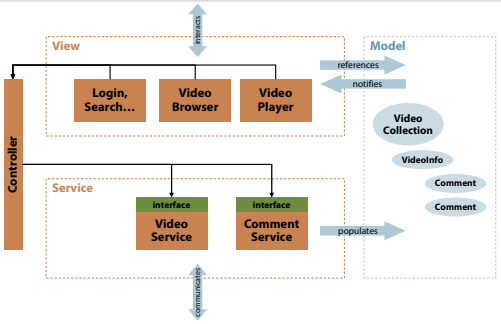
- Isolate state, user interaction, operations, communication
- Cope with unpredictable evolution of UI design
- Parallelize design and coding
- Minimize and simplify server communication
- Respond instantly to user actions, provide good feedback
- Handle cross-cutting concerns:
  - testability
  - logging
  - security
  - error handling

Sample Flex Application

- ReviewTube: a mashup between YouTube.com (a popular video publishing site) and a custom Comment Server
- adds caption display synchronized with cue points in YouTube videos
- YouTube provides metadata and media
- custom Ruby on Rails server provides add-on caption data (and mirrors relevant metadata)

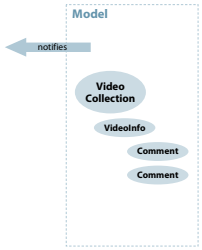


Flex Client Architecture: Models, Views, Controllers, Services

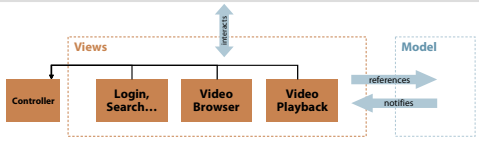


The Model: Representing State

- Encapsulates all client state as classes, collections, properties
- Exposes all state changes as event notifications
  - (Note: bindings imply automatic events)
- Does not refer to any non-Model application component
- Not necessarily identical to server representation: event notification often drives design.



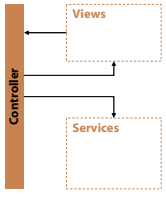
The View: Supporting Presentation and Interaction



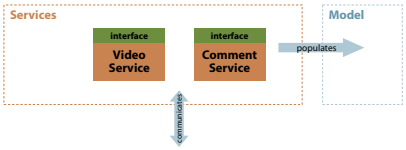
- Encapsulates all user interface design
- Presents application state exposed by Model objects
- Responds to user gestures
- couples user gestures to Controller operations
- Responds to Model change notifications by updating itself
- typically MXML plus fraction of AS

The Controller: Exposing Operations, Invoking Services

- Encapsulates implementation of all user-initiated operations in the application
- exposes operations, model objects to Views
- behind the scenes, invokes Services
- manages indirect view-to-view relationships
- acts as Façade for:
  - miscellaneous application logic
  - access to model objects
  - progress and error reporting
  - user confirmation and validation
  - security and authentication



The Service Layer: Remote Operations and Logic

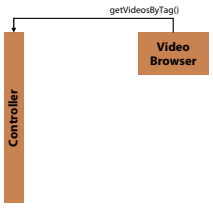


- Encapsulates all remote operations and logic outside the application
- Constructs and populates Model objects with remote data
- Logical place for stubs early in development or as test harness
- Caching and other performance enhancements can be easily added

Interactions: Model, View, Controller, Service

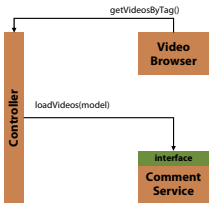
User initiates gesture, searching for videos by tag

Interactions: Model, View, Controller, Service



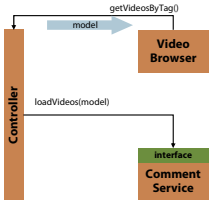
View asks Controller to get the videos

Interactions: Model, View, Controller, Service



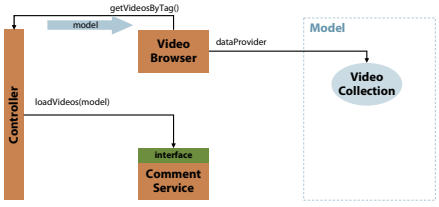
Controller uses Service to load videos into a model

Interactions: Model, View, Controller, Service



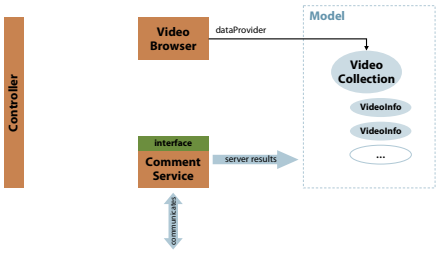
Controller returns model (still empty) to View

Interactions: Model, View, Controller, Service



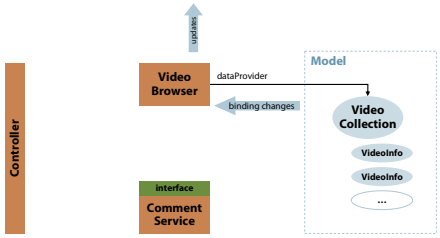
View binds Model as its dataProvider, listening for change events

Interactions: Model, View, Controller, Service



Service processes server response, populates model

Interactions: Model, View, Controller, Service



Model emits event notifications, triggering View updates

MVCS In Summary

- Model: semantic data only
- View: interaction and presentation only
- Services: communication only
- Controller: handles the relationships between all of the above as well as cross-cutting concerns like status displays, etc.

Global Access to Components: AS superclass

Components.as:

```
public class Components extends UIComponent {
    // expose components as abstract classes or interfaces
    public var controller:Controller;
    public var mediaService:IMediaService;
    public var commentService:ICommentService;
    public var session:Session;

    public function Components() {
        super();
        _instance = this;
        dispatchEvent(new Event("instanceChanged"));
    }

    [Bindable("instanceChanged")]
    public static function get instance():Components {
        return _instance;
    }
}
```


Global Access to Components: MXML subclass

ReviewTubeComponents.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- define concrete component classes and properties -->
<Components>
    <YouTubeMediaService id="mediaService"
        urlPrefix="http://www.youtube.com/"
        developerId="719eMsd32SD"/>

    <CommentService id="commentService"
        urlPrefix="http://localhost:3000/flex/"/>

    <controller:Controller id="controller"
        mediaService="{mediaService}"
        commentService="{commentService}"/>

    <model:Session id="session"/>
</Components>
```


Top Level of Application

ReviewTube.mxml:

```
<mx:Application>
    <mx:Style source="ReviewTube.css"/>
    <services:ReviewTubeComponents id="components"/>
    <view:TopLevelView/>
</mx:Application>
```

Sample View Code

YouTubeTagSearchForm.mxml:

```
<mx:HBox>
  <mx:Script> <![CDATA[
     private function searchByTag(tag:String):void {
        Components.instance.controller.searchByTag(tag);
     }
  ]]></mx:Script>

  <mx:Label text="Search by tag:"/>
  <mx:TextInput id="tagInput"
                enter="searchByTag(tagInput.text)"/>
</mx:HBox>
```


Sample View Code (2)

LoggedInHeader.mxml:

```
<mx:HBox>
  <mx:Script> <![CDATA[
     private function logout():void {
        Components.instance.controller.logout();
     }
  ]]></mx:Script>

  <mx:Label styleName="userWelcomeLabel"
    text="Hi,{Components.instance.session.user.username}!"/>

  <mx:Button label="Sign Out" click="logout()"/>
</mx:HBox>
```


Approaches to Model Implementation

- Value Object Classes:
  - vehicle for uniform data representation in client
  - All scalar properties Bindable
  - All collections implement ICollectionView
- Data only, no behavior allowed!
- Think "backward" from both UI design and server-side schemas to determine makeup of data holder objects:
  - Make data binding convenient
  - Make it easy to tell if you've fetched something from the server
  - Minimize rearrangement of server data structures
- 100% mapping to server side is not always desirable

Sample Model Class

ReviewInfo.as:

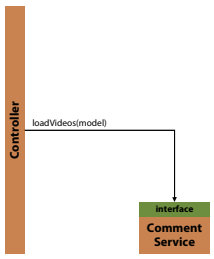
```
[Bindable]
public class ReviewInfo extends ValueObject
{
    public var id:int = -1;
    public var providerId:String;
    public var commentCount:int;
    public var comments:ArrayCollection
                              = new ArrayCollection();
    public var modificationDate:Date;
}
```


The Asynchronous Tango: Using The Command Pattern

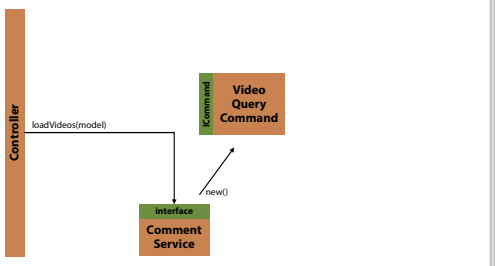
- Many operations are subject to a common handling in the Controller:
  - initiation with some kind of delayed completion
  - on-the-fly construction and population of Model objects
  - possible need for status, progress, error reporting
- Command pattern: Controller works with Command objects which expose a uniform interface for completion, status, error handling, etc.
- Allows Controller to define optimistic or pessimistic handling of operations
- Factory pattern: Service can act as factory for Commands returned to caller
- Service properties (e.g. host URL, protocol, etc.) apply to all its Commands
- Composite pattern: Commands can be composed into a higher-level CompoundCommand that executes its children in sequence

(Note: "Command" here can mean "command to remote server", not just "user interface command")

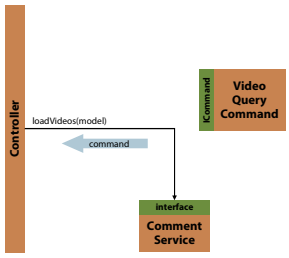
Interactions: Controller, Service, Command



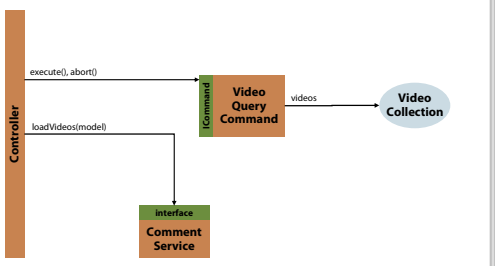
Interactions: Controller, Service, Command



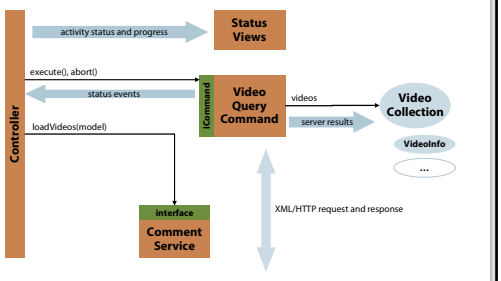
Interactions: Controller, Service, Command



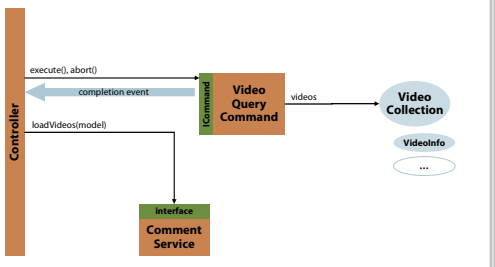
Interactions: Controller, Service, Command



Interactions: Controller, Service, Command



Interactions: Controller, Service, Command



Sample Controller Code

from Controller.as:

```
public function getAllVideos():ICollectionView {
  var dataProvider:ArrayCollection = new ArrayCollection();
  performCommand(commentService.loadAllVideos(dataProvider));
  return dataProvider;
}

public function logout():void {
  performCommand(commentService.logout());
}

public function performCommand(c:Command):void {
  showStatusText(c.displayName);
  c.addEventListener(Event.COMPLETE, commandComplete);
  c.addEventListener(ErrorEvent.ERROR, commandFailed);
}
```

Sample Service Code

from CommentService.as:

```
public function loadAllVideos
            (dataProvider:ArrayCollection):Command {
  return VideoQueryCommand.allVideos(this, dataProvider);
}
```

from VideoQueryCommand.as (extends XmlHttpCommand):

```
public var videos:ArrayCollection;

public static function allVideos  // static factory method
            (s:CommentService, dp:ArrayCollection): void {
   var c:VideoQueryCommand =
       new VideoQueryCommand(service.urlPrefix + "/all");
   c.videos = dataProvider;
   return c;
}
```


Grain Size: Remote Communication

- Common mistake: many fine-grained calls such as in a Data Access Object
- Can allow rogue requests or buggy applications to create inconsistent server state
- Inherently less efficient
- Rule of thumb: remote operation <= UI commit grain size, but not too much less

Approaches to View Implementation: The Fundamentals

- Set child properties to communicate "inward"
- Dispatch events, set Bindable properties to communicate "outward"
- Never assume that properties will be set in a well-defined order
- Components shouldn't know what container they go in
- Containers shouldn't know the structure of their child components
  - MXML id-based properties are always public, which creates temptation!
- Refactor ruthlessly to encapsulate and reuse components

Approaches to View Implementation (2)

- Factoring components via composition: more powerful than inheritance

```
<mx:Canvas>

  <mx:metadata>
    [Event(name="increment",type="flash.events.Event")]
  </mx:metadata>

  [Bindable]
  public var value:int = 0;

  <mx:Label text="{value}"/>
  <mx:Button label="Increment"
         click="value++;dispatchEvent(new Event('increment'))"/>

</mx:Canvas>
```


Approaches to View Implementation (3)

- Factoring components via inheritance
  - common pattern: AS base (view logic), MXML subclass (layout and styling)
  - superclass `var x` overridden by subclass `<component id="x"/>` pattern
  - "Childless" MXML superclass can customize superclass properties and be used as a component itself in some other MXML file:

```
<mx:Button prop1="value1" prop2="value2" …/>
```


Approaches to View Implementation (4)

- Miscellaneous guidelines
  - safely expose child property by binding to public property of parent

```
<mx:Canvas>
  [Bindable]
  public var text:int = 0;

  <mx:TextInput id="input" text="{text}"/>
  <mx:Label text="Word count: {countWords(text)}"/>
  <mx:Binding source="input.text" destination="text"/>

</mx:Canvas>
```

  - All event handling functions should be private
  - Use component names with meaningful suffixes, e.g. `priceLabel`
  - Isolate style/skin knowledge: always prefer `styleName` to explicit styles

Approaches to View Implementation (5)

- Beyond straight variable bindings
  - use Bindable getter/setter functions to add "side effects" for setting properties
    - example: requesting load of optional data needed by some view
  - bind to a function of a Bindable value to "decorate" its value and still be change-dependent
    - example: complex programmatic formatting of properties
- "view-dependent" state is often useful and may live outside the model (e.g. sorting and filter criteria)

The Cairngorm Framework

- It's well conceived and executed
- Many people like it
- It handles many of the issues discussed here
- Some differences: usage of Command, Responder, Locator patterns
- Semi-aligned with concepts in J2EE Core Patterns
- Strong reliance on event broadcasting for user actions

Avoiding The Dark Side

- Preconceived vs. emergent approaches
- Force-fitting problems to patterns:
  "If I have a hammer, this must be a nail"
- Using patterns needlessly:
  "That looks like a nail, I must need a hammer"
- Satisfy key needs, not possible needs
- Abstraction costs time: use it at obvious "hinge points"

Many Paths to Glory

- There's always more than one way to do something
- Business and politics do influence technical approaches, like it or not
- Patterns and practices in the end are a tool to meet one's real-world needs

Resources

- ReviewTube sample application
  can be downloaded from:

http://joeberkovitz.com/max2006/

Better by Adobe™