## Livecycle XFA Form design using the ‘core’ fragment

## Background

The core fragment was originally developed by some contractors for the development of a very large form for the Natural Health Products Directorate (NHP). The reasoning behind this fragment was to abstract the UI layer from the business logic of the form. It follows the View Model-Controller design pattern.

As such, there is some latent code in this fragment that is specific to NHP. It should be noted that the developers that built that this framework.

## Framework Advantages and Disadvantages

## Basic Form Script Structure

1. A subform called script (hidden, flowed subform, parent is the root node)
2. Core fragment as a child of the script subform.
3. Form Specific script subform as a child of the script subform. (For HPFB we consistently name the subform HCeForm. This makes code sharing easier).
4. Subforms: config, control, model, nodeDefinitions, and resources (spelling case? specific) under the form specific script.

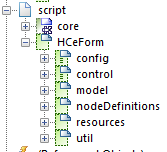


Figure 1: Form script structure

## Config

The config subform can consist of one or more script objects with key/ value pairs defining constants that are used throughout the form. This facilitates update of a constant, if needed. To retrieve a configuration values use the configuration manager and call the via the key:

var cConfigManager =script.core.util.ConfigurationManager

var FRENCH= cConfigManager.get("FRENCH");

## Control

This contains all the controllers used in the form. The controllers are where the business logic is implemented. Each controller is responsible for a specific area on the form, and any node definitions related to the specific area should only be defined in that controller. You can have any number of controllers, defined by their own script object. Child subforms can be used to further organize the controllers if necessary.

## Model

If required, the model subform contains script objects that are used to define the model of the form. In general, this is not used as the forms are not that complex as to need to model any data. Generally, most of the logic is in the controllers.

## Delegates

Delegates are defined under a child subform of Model. Delegates are used to apply validation logic against a field or set of fields. Validation failures are then rolled up into a single interface. Delegates are useful as they allow for reuse of validation code.

//TODO: add more information

### How to reuse Delegates

After creating a generic delegate, it can be reused across form elements by doing the following:

1. Create a script object for the form element you want to validate.
2. Add a createInstance() function.
3. In the createInstance() function, create the delegate.
4. Add error text resource that is specific to the form element.
5. Return the delegate instance

For example:

var cLog = script.core.util.Log;

var cEmailFormatDelegate= script.HCeForm.model.delegate.EmailFormatDelegate;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Gets a new Delegate.

\* @return the Delegate instance

\*/

function createInstance()

{

var oDelegate= cEmailFormatDelegate.createInstance();

oDelegate.setErrorText("error\_contactEmail");

return oDelegate;

}

## Node Definitions

Node definitions are the key data structure used to separate the business logic form. Node definitions break out every element of the pdf form, and are addressed by a unique identifier (developer defined). Node definitions are used to define localized caption text, delegates, toolTips and dropList values.

For example a typical node is defined as:

Unique name (alias) for lookup🡪 chkUseChange: {

Field name in the form🡪 somExpression: "chkUseChange",

parent node alias🡪 parentNodeKey: "sfFormulationType",

label key (from resource file)🡪 labelKey: "lbCheckUse",

toolTipKey (from resource file)🡪 toolTipKey: "ttChangeUse"

},

The only node that is unique in it’s definition is the root node as it has no parent. It is defined as the following:

root:{

somExpression: "xfa.form.PATENT\_LIST",

},

### Node Definitions tools

Given that creating node definitions is tedious there is a macro developed for LiveCycle Designer that will automatically generate base node definitions automatically.

#### Installation

In the installation folder of LiveCycle Designer (C:\Program Files (x86)\Adobe\Adobe LiveCycle Designer ES3) create a macros folder if it doesn’t already exist. Create a folder for the nodes definition macro, unzip into the folder

#### Limitations/Bugs

1. You cannot generate node definitions for master pages ☹
2. When generating node definitions for radio buttons, the radio buttons parents are incorrect. This must be manually corrected.

#### Usage

1. In the designer, select the UI node that you want to generate node definitions from (because of limitation 1, it cannot be the root node).
2. Open the macro. Ignore the flash dialog (i.e. use the defaults).
3. Click OK.
4. Notepad will open with the list of node definitions. Copy the information into a script object

## Resources and localization

Resources were designed with multiple localizations in mind. Resource script objects are created in pairs, with the same base name, separated by \_en, and \_fr to identify the language (more languages could be added if needed). Each resource Key- value pair is duplicated in the localized languages. Localization can be organized with multiple script object pairs.

TODO:util

## Controller Structure using the core fragment

Every controller is defined in a separate script object. The controllers are used to encapsulate a specific set of common business functionality. Each controller has the following structure:

1. Declarations/imports/constants
2. Action declarations
3. Form Node declarations
4. Form Node Initialization
5. Action event handler
6. Business Logic (Public/private)

## Declarations/imports/constants

This is for all the common elements the controller will be using, for example:

var cLog = script.core.util.Log;

var SCRIPT\_OBJECT\_NAME= "HumanReportHandler";

var cFormHandler= script.ir.control.FormHandler;

var NO= cConfigManager.get("NO");

## Action Declarations

These are just local controller constants for uniquely defining an action:

var SELECT\_SYSTEM\_ACTION= "selectSystem";

## Form Node declarations

Form Node declarations are an alias to the adobe node. This implementation abstracts the view from the business logic. Changes to the layout only require a change to the node definitions. A form node is the equivalent to a “class” of the node instance.

var HUMAN\_ROOT\_SF\_FN= cFormNodeManager.getFormNodeForKey("page\_2");

## Event Handlers

In an object event the general implementation would be:

var cActionEvent = script.core.event.ActionEvent;

var cLog = script.core.util.Log;

var cMainHandler = script.ir.control.FormHandler; //points to the controller

try

{

// set up event object

var actionName = cMainHandler.LANGUAGE\_CHANGE;

var anActionEvent = cActionEvent.createInstance(actionName, this, this.rawValue); //must pass an object, value is optional

// deploy event to controller

cMainHandler.actionPerformed(anActionEvent);

}

catch(exception)

{

cLog.warn("Problem with pageLanguage::change", exception);

}

### Example: In a controller the action event handler format

/\*

\* Performs the called actions

\* @param anEvent- the event object containing the action and Source

\*

\*/

function actionPerformed(anEvent)

{

if(!isHostVersionSupported()) return;

switch(anEvent.getActionCommand())

{

case INITIALIZE\_FORM\_ACTION:

\_updateFormLanguage(anEvent.getSource(),anEvent.getValue());

initialize();

break;

case LANGUAGE\_CHANGE:

\_updateFormLanguage();

break;

default:

throw cException.createInstance(SCRIPT\_OBJECT\_NAME

+ ".actionPerformed() - unrecognized event '"

+ anEvent.getActionCommand() + "'.", "UnsupportedAction");

## Code Examples-TODO

Droplist, detecting a change

In the enter event:

var cSystem = script.core.System;

cSystem.setCurrentField(this);

In the exit event

var cActionEvent = script.core.event.ActionEvent;

var cLog = script.core.util.Log;

var cGeneralHandler = script.ir.control.GeneralHandler;

var cSystem = script.core.System;

try

{

if(cSystem.hasCurrentFieldValueChanged(this)){

// set up event object

var actionName = cGeneralHandler.UPDATE\_LOCATION\_COUNTRY;

var anActionEvent = cActionEvent.createInstance(actionName, this);

// deploy event to controller

cGeneralHandler.actionPerformed(anActionEvent);

}

}

catch(exception)

{

cLog.warn("Problem with Location country::exit", exception);

}