MVC FORMS IN LIVECYCLE ES

Version 1.0
30th November 2007

Statement of Confidentiality

Revision and Related Document Change Log

Document Revisions

Revision Number	Date	Page(s) Affected	Comments	Authors
1.0	30th Nov. 2007	All	Initial version	Herve Dupriez

Related Documents

Revision Number	Date	Document Title	Comments

Table of Contents

1 Introduction 1

1.1 Purpose of this Document 1
1.2 The Paper Document Paradigm 1
1.3 What Makes a Good Form Framework? 2
1.4 MVC Patterns and the XML Form Architecture 3

2 MVC Pattern Applied to XFA forms 5

2.1 The Model-view-Controller Design Pattern 5
2.2 The Model 6
2.2.1 Localization 9
2.2.2 Back or Undo Buttons 10
2.3 The View 11
2.3.1 Defining Views and Panels in the Application Model 13
2.3.2 View Context 15
2.4 The Controller 15
2.4.1 Form Controller Scripts 16
2.4.2 Navigation 17
2.4.3 Initialisation 17
2.4.4 Validation 17
2.4.5 Printing 18
2.4.6 Language Selection 18

3 Using Form Fragments 19

3.1 Benefits of Form Fragments 19
3.2 Fragment Delegation 19
3.2.1 Action Delegation 19
3.2.2 Validation Delegation 20

4 Logging 21

5 Conclusion 22

1 Introduction

1.1 Purpose of this Document

This document describes some design practices that have been used successfully on dynamic XFA PDF forms, and it can certainly be read in that perspective only; however, by writing it we have certainly tried to achieve a little bit more than that.

Best practice documents about form design do not always attempt to address the need for a generic and reusable framework, limiting themselves to a list of recommendations applicable to specific projects, gathered from an expert knowledge of Adobe LiveCycle Designer, and assimilated from and number of practical project experiences. By doing so, they have not really facilitated the emergence of common practices that could be adopted across multiple projects and speed up development.

We believe that with the introduction of Adobe LiveCycle Enterprise Suite, many more organisations will adopt Adobe's vision of the engagement platform, and it then becomes absolutely critical to introduce new form design strategies based on carefully engineered foundations.

By presenting a form design framework centered on the application of well known design patterns (such as the MVC pattern), we are trying to make a step in that direction.

By default we assume that the reader who will investigate and test the recommendations in this document will be using Adobe LiveCycle Designer ES to develop forms, and testing them in Adobe Reader 8. x. Many of these recommendations were implemented with earlier versions of Designer (Designer 7.1 in most cases) and tested in earlier versions of Adobe Reader (7.0.5 or later) and should work as well in these earlier versions.

1.2 The Paper Document Paradigm

The implementation of dynamic forms as PDF documents is a brilliant idea, but it has also caused some confusion in many people's minds since its inception. If PDF documents are the perfect electronic transposition of paper documents, then what would be the expression of a *dynamic form* in the context of paper forms?

First of all let's state the obvious: forms are only thought to be dynamic as a result of the user interacting with the document or the form. If the user does nothing, one should not expect the document or the form to behave dynamically in any way. Secondly if the user interacts with the form, it is only when the *layout* of the form changes that it can be considered dynamic. Since the layout of paper forms never changes – the fields are already positioned on the sheet of paper when people start filling them – one can hardly find any obvious scenario where paper forms would be seen as dynamic.

This is actually not the case with paper documents, which can be thought as dynamic in document *authoring* scenarios such as:

- writing on a blank page,
- underlining text,

- adding comments.
- adding a new page when one has reached the bottom of the current page.

But finding a satisfactory equivalent – in paper forms – for a dynamic, expandable flow of content is quite challenging.

In fact the closest real life scenario to illustrate *"dynamic paper forms"* would require the operation of assembling together two different paper documents:

- a paper form that enable the capture of data;
- a printable, read-only form, that would be a document of record for the data that has been previously captured.

For instance, if you need to apply for an identity card, you would typically fill in a form, and after your identity has been checked successfully, you would be given the identity card. Or let's say you need a car insurance – you would fill in a form and once you form has been processed you would receive a certificate of insurance, another good example of a document of record. Or if you are a student and you apply to a university, if you are admitted and pass your various examinations successfully, after a few years you will be given a diploma. The fact that it can take from a few minutes to a number of years to get from the capture form to the document of record is not the point here – most of us we would generally dream that it would only take a few seconds.

From that perspective, we can conclude that dynamic forms essentially give the promise that a process, initiated by the user filling in a form, will somehow be accelerated, and that the time needed to complete that process will be significantly reduced because the dynamic nature of the form. As all of us have observed this is not always possible.

The absence of an unquestionable parallel between paper forms and dynamic PDF forms has left form designers with a dilemma between:

- dynamic forms that adopt the document authoring paradigm, and
- dynamic forms that are similar to dynamic web pages, desktop applications, or rich-client applications, and where the concept of turning pages has completely disappeared.

The attempt to retain the document authoring paradigm, where form element can flow dynamically over multiple pages, is the most common approach and would be perfectly acceptable, however experience has proven that it can fail when the number of pages is too large, which means that something different is needed to support complex forms. Many form designers have developed great forms using this approach – only to find late in the project, while they were adding the final elements to the form, that the performance has become unacceptable, or that the form file size after saving locally was too big.

In this white paper we are describing a framework that can support the second approach – design dynamic forms that are similar to dynamic web pages – but we are providing at the same time design techniques that will help maintain the benefits of both worlds.

1.3 What Makes a Good Form Framework?

Because designing forms to support enterprise processes is complex and requires specific expertise, because implementing a large numbers of forms requires strong coding practices and the enforcement

of strict quality assurance procedures, enterprises are willing to invest in these strategic areas:

- Form frameworks that rely effectively on reusable form components (such as fragments);
- Form frameworks that can support their most complex form requirements;
- Form frameworks that integrate seamlessly with their backend infrastructure.

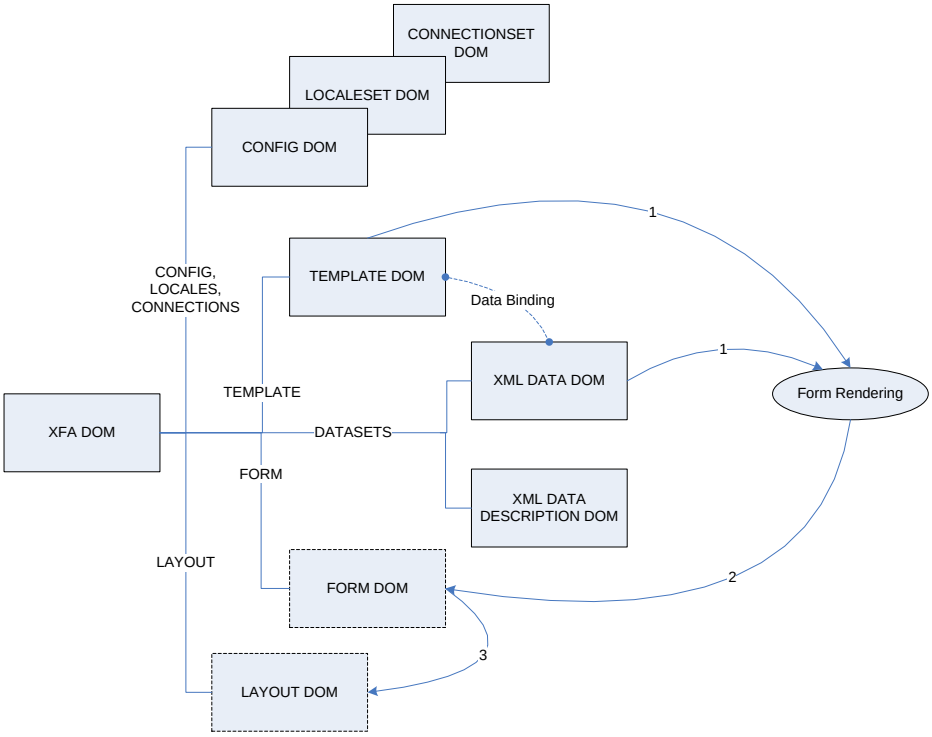
Enterprise will never be ready to invest heavily in form frameworks that would eventually fall short of addressing these objectives, and they been a constant priority for us, when designing the so-called "MVC forms".

1.4 MVC Patterns and the XML Form Architecture

The XML Form Architecture (XFA) cannot be associated with any particular GUI design pattern such as the MVC pattern. As a matter of fact the XFA specification does not include any reference to design patterns and defines itself as "*a template-based grammar and a set of processing rules that allow businesses to build interactive forms*".

Adopting specific GUI design patterns is mainly a concern for the applications that implement XFA, such as Adobe Acrobat, Adobe Reader, or LiveCycle Forms.

The XML Form Architecture describes forms through a main Document Object Model – the XFA DOM, which includes many other DOMs, as illustrated in the following diagram. The process of rendering forms by merging templates with data also introduces additional DOMs such as the Form DOM and the Layout DOM.



The XFA DOM

Although XFA form rendering engines are sometimes described as implementing a MVC pattern, this is not very helpful to the form designer. The job of the form designer can cover simultaneously the model, the view and the controller[1], and the main result of a form design work – the template DOM – can mix together presentation rules, data model rules and user interaction rules.

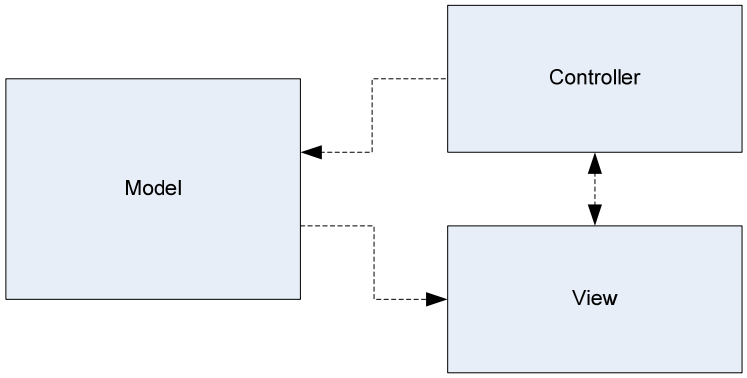
Without any specific GUI design pattern in mind, the form designer will essentially apply a "forms and controls" architecture, where forms are essentially the result of combining together a multitude of reusable controls to define a form layout, and adding additional logic to implement the rules that cannot be implemented in the individual controls. In doing so, the form designer will find the principle of *data binding* immensely helpful to enforce the separation between the Presentation and the Model. Another fundamental principle is the use of *events* (such as init, click, enter exit etc.) allowing individual controls to notify the form whenever they occur, allowing the form to execute some specific code.

[1] However the effort needed to define the model can be significantly reduced when a XML schema is already available to describe the data used in the form.

2 MVC Pattern Applied to XFA forms

2.1 The Model-view-Controller Design Pattern

Many discussions have been going on about the Model-View-Controller (MVC) design pattern; this document will not attempt to give you another definition here. The amount of time spent on this topic shows how hugely successful this design pattern has been since it was first introduced in SmallTalk-80, about three decades ago. It is also very easy to gather a lot of information about it on the Internet, but this large amount of information (sometimes contradictory) also makes it a controversial and confusing topic.



In summary:

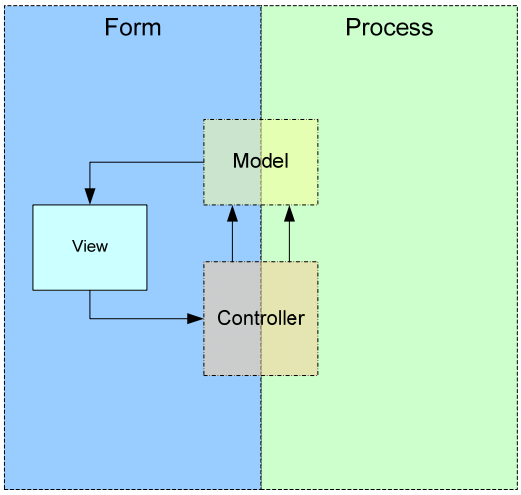
- The view reacts to changes in the model to display up-to-date information about the model; the view also reacts to user input and passes it to the controller.
- The controller receives user input from the view and modifies the model in response to user input.
- The model changes, based on an update from the controller, and notifies the view of the change.

It is generally agreed that MVC patterns are mostly successful for complex solution designs, and that one should not bother introducing a MVC pattern to develop a simple application. In that sense it may look like a heresy to recommend a MVC pattern to form designers, when Adobe has put in their hands a tool – LiveCycle Designer – that can be so easy and intuitive to use, a tool that allows them to develop forms simultaneously in terms of data modeling, presentation and controls. Also in many circumstances forms can be seen as simple documents that enable the user to capture information needed at a specific point in time, as part of a more complex business processes. If a single form becomes too complex, shouldn't that be a signal for us to start thinking about redesigning the entire process around a new user experience?

It is not always possible – front ends in a business process cannot always be simplified – take offline PDF forms as an example, or some complex dashboards developed in Flex.

All of those who have spent a significant amount of time developing forms, and often many forms for the same organization, know how important it is to develop forms in a consistent manner, as part of a more global business process management framework.

The importance of allowing a clear separation of responsibilities between Model, View and Controller becomes indeed critical when your aim is to develop a framework that will tie together forms and processes. Without the appropriate design patterns it is difficult to avoid a form-centric design for form development, and a process-centric design for process development. Using the MVC pattern is a reasonable approach to solve this architectural problem. Obviously forms and processes should share the same model, as it would be very ineffective to require systematic translation from one model to another. It is equally important to use a framework that specifies how the control of a business application can be shared between forms and processes.



2.2 The Model

In simple terms, from the form designer perspective, the Model is the set of form components that manage the form data, and the logic that can be applied to that data, regardless of any GUI element that can be defined to help a user interact with that model. These form components are obviously non visual components of the XML Form Architecture, such as:

- data connections
- scripting objects
- document variables

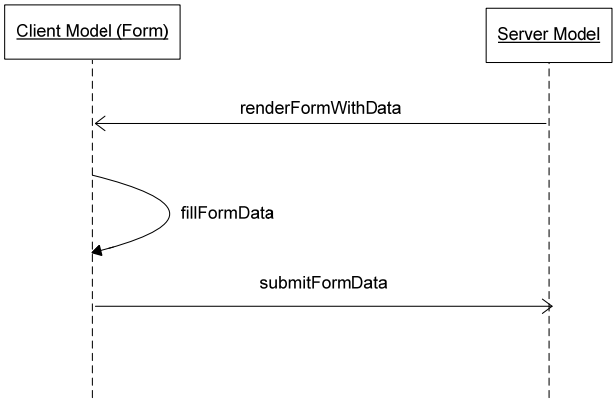
Traditional enterprises applications implement the Model purely on the server side. However in form design there is a need for both a centralized and a decentralized representation of data, and the synchronization of data between the centralized model and the decentralized model is implemented in the XML Form Architecture through *data connections*.

Because a typical requirement for PDF forms is the ability to save the form locally, we can say that the enterprise application must delegate some intelligence of the model to the form. Even in the context of online PDF forms, some sort of model delegation is required because the PDF form typically requires

only two integration points with the enterprise tier:

- when the form is rendered pre-populated with data;
- and when the form is finally submitted;

It means that between these two events, the PDF form must be able to implement business rules that change the data and that belong to the model.



One particularly effective way of managing the model at XFA form level is to use a XML schema representative of the domain model, at all times[2]. XML schemas provide a specification of the data that can be shared between the form client and the server; the more rules can be described through the schema, the easier it will be to share these rules and synchronize them when a change needs to be applied to the model.

In the rest of this document we shall always assume that the model is managed through the XML schema data connection and the XFA data model.

One important point about XML schema binding in XFA is that there can only be one XML schema bound to the form. The other important thing to remember is that when you save a form bound to a schema, only the fields that are bound to the schema will be saved and retained the next time you open that form; if you submit the form to a web application (in a save and retrieve scenario), the same will happen so that when the application will pre-populate the form with the data that was submitted, the user will only get back the form fields that were bound; so obviously you don't want to miss any data binding during development, but more importantly, if there is any important information that the user captures in the form, you need to make sure that this information can be saved, and consequently that there is an element within the schema that can be bound to that information.

But what if the schema for the domain model does not contain all the elements needed to support the form capture? This could be, for instance, the name of the end user that must be displayed in the form.

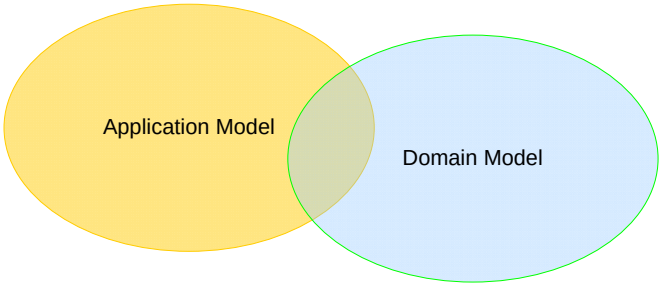
In practice you would typically start from a schema provided by the organization for which you are developing the form. We can call it the domain model schema. That schema may already exist, or it may

[2] Previous versions of MVC forms that were released did not demonstrate this interaction with XML schemas, which generated of lot of complication around the management of data.

be that you will have to develop that schema to support your form development.

If the schema already exists, it may have been developed with many possible applications in mind, because in principle a domain model schema should not be limited to a single application. Often you will find that this schema is larger than needed, and does not necessarily cover exactly the same domain. And invariably you will find that some elements are missing to support the user experience that was defined in the early stages of the project.

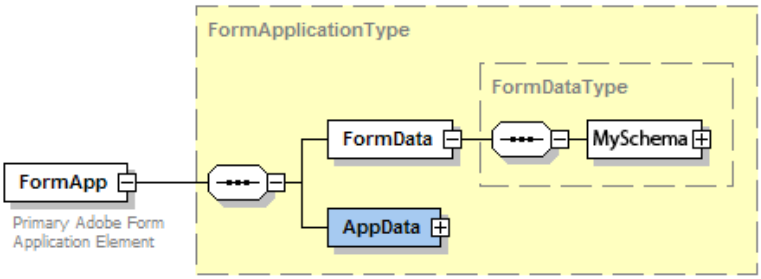
There is a clearly need for an application model, and the XML schema, because it has to be unique in the XML Form Architecture, will have to cover the application model as well as the domain model.



If you plan to develop a schema you will have the opportunity to define it in such a way that it will describe all the data that goes in and out of the form and only that data. You will also have the liberty to define any element that will help manage the form state and its presentation logic in relation with the user experience design. When doing this repeatedly for multiple forms, you may notice some application requirements that should be made generic across multiple forms.

What should you then do?

Through multiple projects we have implemented a design pattern where the domain model schema is wrapped inside an application model schema, we consistently named it "FormApp".



This means that instead of creating a data connection directly to the "business" schema that may already exist, we reference that schema ("MySchema" in the above diagram) inside a "wrapper schema" that also references additional elements needed to maintain the form as an application rather than just a form data container. This second node named *AppData* also enables a consistent way for the form and the process to communicate together as a single application within the LiveCycle ES service container.

It is in the nature of LiveCycle ES form applications to be built from reusable components, and that if we can define a common model for many data elements that are needed across multiple projects, we can

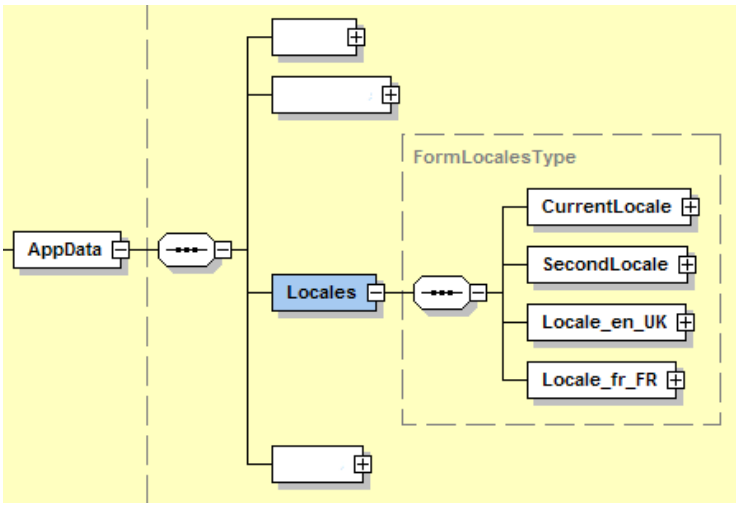
enhance even further reusability and it will then be easier and quicker to develop forms and integrate these forms with a business process.

Let's give a couple of examples to illustrate the benefits of this concept of an Application Domain Schema.

2.2.1 Localization

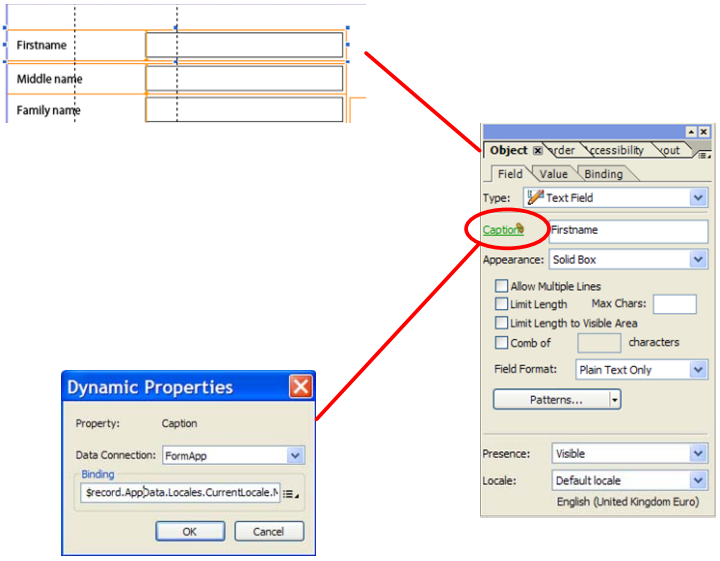
The first example is the form localization framework that we have developed for a project that required a single form to support 7 different languages, and the ability to dynamically switch from one language to another within the form.

Under the *AppData* node we have defined a *Locales* node that will include all the textual language resources that need to be embarked into the form to enable this language switching capability when the form is offline.



In the above example we have only include the resources for English (sub-node *Locale_en_UK* for UK English) and French (sub-node *Locale_fr_FR* for France French), but we could in fact add as many language specific resource nodes as needed.

To add multi-lingual support to the MVC form, the form designer should bind the field captions to the resources defined under the *CurrentLocale* node. The following picture illustrates the binding operation being performed in LiveCycle Designer.



Once the needed language resources are loaded in the form data, language switching – let's say to French – is performed by serializing the *Locale_fr_FR* using the `saveXML()` method, and then deserializing the XML into the *CurrentLocale* node, using this time the `loadXML()` method.

This localization framework can also be extended with *resource capture forms* and *translation forms*. Both are PDF forms intended to facilitate support for additional languages.

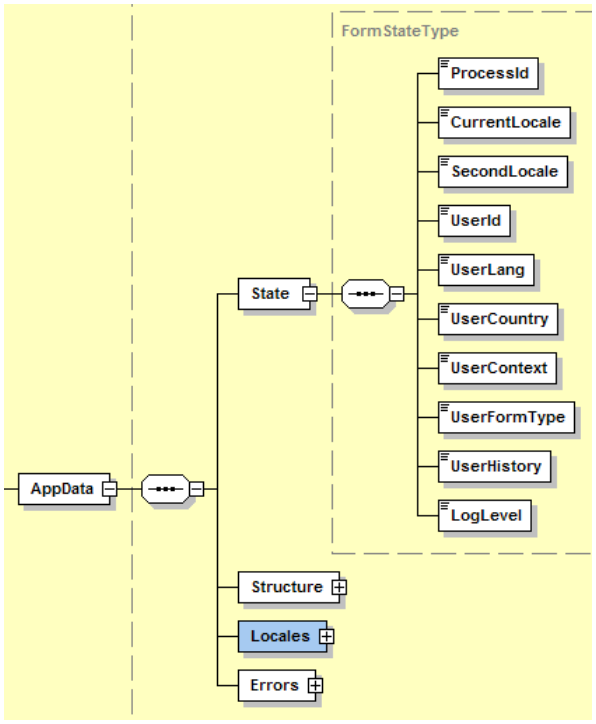
Resource capture forms allow the update of the application schema model with additional message declarations.

Translation forms simplify the translation of messages in a new language by showing all messages side by side in two different languages. This is why we have defined a *SecondLocale* node in the application model schema, in addition to the *currentLocale* node; *SecondLocale* is only used in a form when two languages need to be displayed simultaneously.

2.2.2 Back or Undo Buttons

Another benefit of the MVC pattern is that it provides a fairly straightforward path to implement GUIs with Back or Undo functions. Because the Controller component of the MVC form (that we shall describe in detail later in this document) is systematically invoked to forward the user to the next view, it is easy to store the corresponding actions in a history table.

In our MVC form sample, we are simply storing the name of the previous view in a node names *UserHistory*, under the */FormApp/AppData/State* element, as depicted in the figure below:



Each time the controller is invoked, it will update *UserHistory*. It is then trivial to implement a generic "Back" button – the controller will retrieve the name of the view to display and request the view component to display that view.

This history information can also be used to store the latest view being displayed before closing the form, so that the next time the user is opening the document, this view will be displayed immediately rather than the first or default view.

A more complex logic will be needed to implement an undo function, but fundamentally it will require the same ability to persist history data in the application domain schema.

As you may have noticed we avoided using the term *page*, and used the term *view* instead. So now we need to tell you more about views.

2.3 The View

The view is what is displayed to the user; in the case of a PDF form, that would be a list of pages. With dynamic forms it is extremely easy to hide and show pages, so why show all pages at all times?

We already mentioned that the more pages, the bigger the PDF document will be, and the slower it will be. But even before worrying about performance issues, form designers should be concerned about *User Experience* (performance is after all just a part of the overall user experience).

Simplicity – achieved by showing only what is needed when it is needed – is a key to successful form designs and it should among our top priorities. However we all know too well that simplicity is rarely a priority for paper form design, due to the static nature of paper forms and the usual requirement to fit

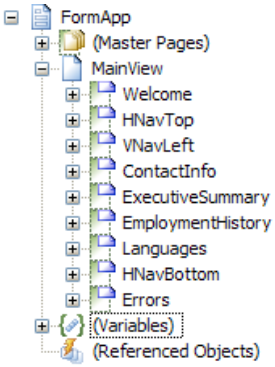
as much information as possible in a minimum number of pages.

The need for simplicity in successful user experiences has led Adobe to add the concept of form guides in LiveCycle Designer ES, where the user is guided through the process of capturing data into the form. Each screen in a form guide is a much simpler view than the complete document, allowing the user to address the task of filling a potentially very complex form by focusing on very simple tasks, one at a time. At any time during the capture process, the user has the ability to display the entire form in a printable format.

In our MVC form design we can create a user experience that is very similar to form guides, where the user is guided through a sequence of *views*, and at any time the user can display the entire form in a printable format. However the fundamental difference between forms guides and MVC forms is that the later is implemented entirely through a single PDF form, whereas form guides rely on a combination of a Flex user interface and a PDF form displayed within an Internet browser window. Although it is not our intent here to compare the benefits of the two approaches, we'll just say that managing a complete form capture experience through a single PDF file that can be reader-enabled and saved locally, is a good and sensible way to address a number of business scenarios[3].

Let's have a closer look at how views can work inside a dynamic PDF form[4].

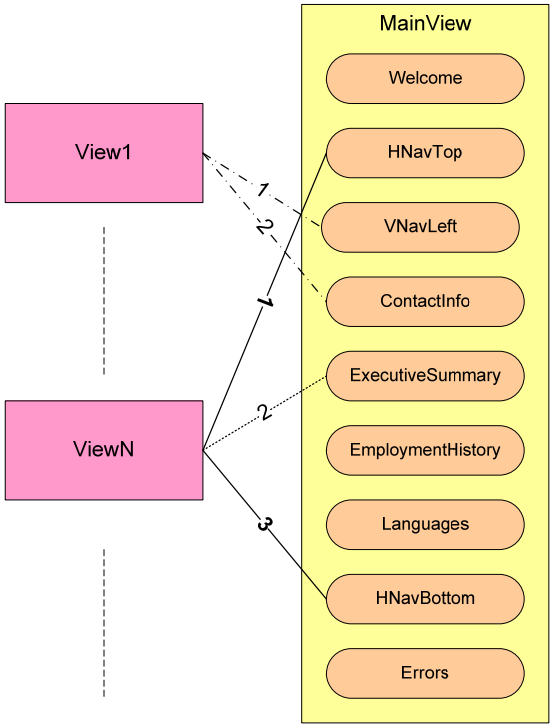
Rather than having multiple *body* pages in our XFA form, we take the approach of defining a single *body* page called "*MainView*". This page must have a flowed content, and it should contain all the sub-forms that may need to be displayed at various points in time.



If we look at the above example, we can see that the *MainView* page contains a number of sub-forms: *Welcome, HNavTop, VNavLeft, ContactInfo, ExecutiveSummary, EmploymentHistory, Languages, HNavBottom, Errors.*

In our MVC form design these sub-forms are called "*panels*". They are not the views themselves, but they are the building blocks for the *views*.

[3] MVC forms can also be directly displayed in LiveCycle Workspace as well as in custom Flex-based workspace applications.
[4] There are two very different designs that can be used to manage views in a MVC form. One that was implemented a couple of years ago relied on the use of the instance manager. The other design described in this document is using the presence attribute of sub-forms.



We can also define the *MainView* object as a container for a library of panels. However, as we shall see later, the order in which panels are included in the *MainView* object really matters, because the panels are part of a flowed content and therefore they cannot be reordered.

In that model a *view* is essentially an ordered list of sub-forms or as we chose to call them, *panels*.

2.3.1 Defining Views and Panels in the Application Model

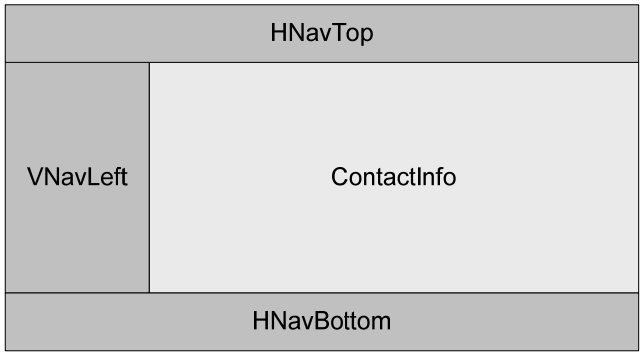
The XML Form Architecture has not this concept of *views* and *panels* that we have introduced. This means that we need to define views and panels in the application model. We have in fact two options to implement them in the application model:

1. defining views and panels via *scripting*;
2. *modeling* views and panels in the application model schema.

There are advantages to both options, and this is why we have actually implemented both of them.

Managing view and panels via *scripting* will allow the dynamic initialization of views through the initialize event script of the form. This allows a self-registration mechanism of the views – let's take an example to clarify this technique.

In our MVC form sample form we have a "*ContactInfo*" panel, and we need to display that panel with a banner (*HNavTop*), a left navigation (*VNavLeft*) pane and a footer (*HNavBottom*). See picture below.



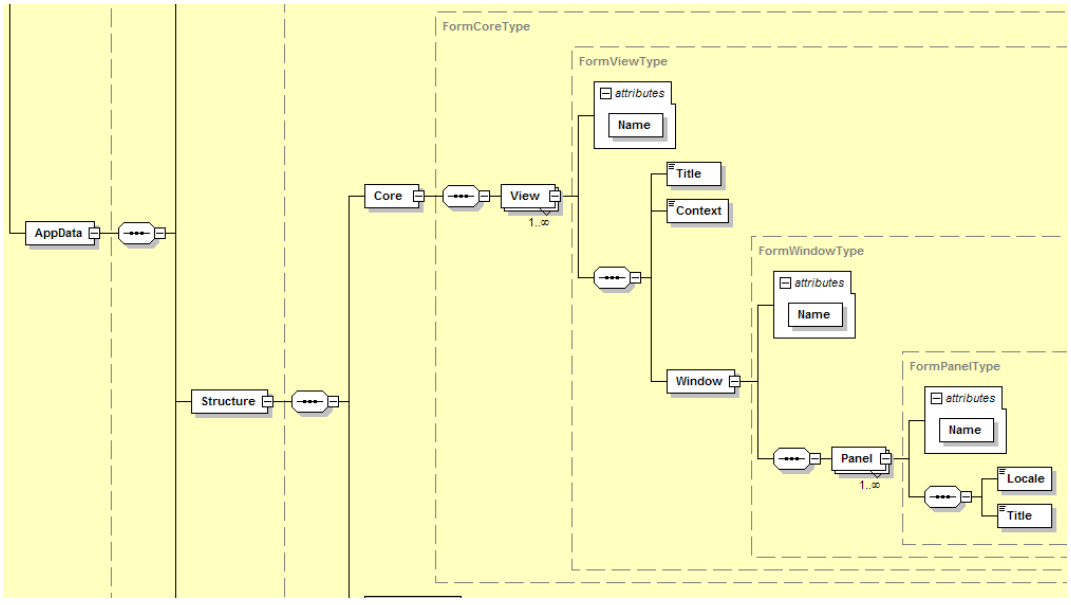
In order to reference this view and make it available to the user, we add the following code to the Initialize event of the *ContactInfo* panel sub-form:

```
scoController.createView("ContactInfoView", "Contact Information", "input" );

scoController.addPanelToView("ContactInfoView", "HNavTop");
scoController.addPanelToView("ContactInfoView", "VNavLeft");
scoController.addPanelToView("ContactInfoView", "ContactInfo");
scoController.addPanelToView("ContactInfoView", "HNavBottom");
```

Creating views by scripting allows a very flexible and dynamic definition of what will be displayed to the user. The view can even be changed at "runtime" as a result of a specific sequence operations performed by the user.

Now let's have a look at the persistence of views and panels through the application model schema. The schema is as follows:



Having views defined persistently through the application schema model will allow scenarios such as:

- Defining the *views* and *panels* externally in a LiveCycle ES process (such as a custom *RenderForm*

orchestration); this would allow the same form to be displayed contextually with a different content or presentation based on the business logic implemented in the process.

- The *Structure* node of the application model schema can also be used as an input specification for the assembly of form fragments.

2.3.2 View Context

A view is always defined with a specific display context in mind. There are two main contexts present by default in the MVC form design:

- input
- output

It is possible to define additional contexts. For instance if the form needs to display help pages separately from the input, it will make sense to define a "help" context. You can also consider that when a user opens the form for the very first time, some specific introduction pages should be displayed – it is then appropriate to define a "welcome" context, so that once the user has been through this introduction, he can be conducted to the input mode, and the next time the user opens the form, the introduction pages will no longer show up.

2.4 The Controller

The controller is really the core element in our effort to apply the MVC pattern to XFA forms, and also the part that has led to different flavours of designs and implementations.

First of all let's remember that there will never be a single controller object in any XFA form design. As we mentioned in the introduction, the XFA Form Architecture encourages a form design made of a multitude of individual form controls; each control is characterized not only by visual properties and its location on the form, but also by its events scripts and data binding. This means that each individual form control has its own controller.

That being said, there is a need to federate all these form controls under the form itself, and without a controller of its own, the form would be merely a container for these form controls, and the logic that bind these controls together would be scattered across the form template without a clear and consistent organisation. As forms grow in complexity, this approach is likely to cause big maintenance issues.

The idea of a form controller defined as a scripting object came originally as an attempt to centralize the form navigation logic, in opposition to traditional form design where this logic was implemented in various form controls (such as next / previous button). The point was that the responsibility to guide the user through the form should not be left to individual form controls, but that there should be a central object with the form to manage this navigation. This was obviously the right thing to do in the case of wizard-based forms, where additional code is needed in order to hide all pages except one.

As the idea of a form controller was gaining some credit, it became important to define the responsibilities of the form controller. In our current MVC form design, the form controller has the following responsibilities:

- Form initialization

- Form navigation
- Form validation
- Form submission
- Form printing
- Interaction between the form and its attachments
- Selection of the context in which the form is displayed (e.g. input versus output)
- Selection of the language in which the form is displayed (if the form is multi-lingual)

This list could be extended, but this should be done very carefully because the form controller should not become a burden for the form designer, and as we mentioned earlier, there will always be a fair amount of control logic that must be left to the individual form controls.

In fact, with the fundamental role played by form fragments in effective form design, we have come to a point where the main form controller will cooperate with "fragment controller", as described later in this document.

2.4.1 Form Controller Scripts

The form controller is implemented using a number of scripting objects:

scoController: this scripting object should ideally never be modified by the form developer, any change would undermine the reusability of form fragments across multiple forms. We have designed this scripting object in such a way that there is in principle no need to change it. *scoController* also includes support for a number of global actions such as

- *Print*: will switch to the output context and print the form.
- *Back*: will tale the user to the view that was displayed before the current view.
- *ShowView*: will display the specified view.
- *First*: will display the first view in the current context.
- *Last*: will display the last view in the current context.
- *Next*: will display the view that comes just after the current one.
- *Previous*: will display the view that comes just before the current one.
- *Edit*: will take the user back to input mode (switch display context to "input").
- *Preview*: will take the user to output mode (switch display context to "output").
- *SetLocale*: will re-display the form using another locale.

- *Submit*: will submit the form.

scoConfig: this scripting object is responsible for the initialization of the form. There is a single method init() that should be invoked on the init event of the form. The init() method is where you check that the version of Adobe Reader being used is supported, this is where you define which view should be displayed when the form is opened. This script can be customized by the form designer.

scoAction: this scripting object is where you can define your own custom global actions for the form. Although the *scoController* already defines some global actions, you can override them here, and you can also define completely new global actions.

scoSubmit: this scripting object is where you can define the submission logic for your form.

2.4.2 Navigation

A basic task of the form controller is to listen to user commands that require the display of another view of the form. The form controller script includes a `showView()` method that will display a view by showing all the panels contained in that view, and by hiding everything else.

2.4.3 Initialisation

The form controller will initialize the form in accordance with all the form state properties stored in the application data (*AppData*). One of these properties is the view that should be displayed when the form is opened.

Practically this means that the `showView()` method (see 2.4.2) will be called during form initialization.

The form designer should ideally take advantage of this in the following ways:

- Before saving the form template in Adobe LiveCycle Designer, hide all the form fragments; this will dramatically reduce the size of the PDF. In business processes where large forms are initially emailed or downloaded, this simple tip can make a big difference.

- If a logic is implemented to check the version of Adobe Reader used to open the form, set all fragments as hidden by default in LiveCycle Designer, except for a single fragment that can be used to display an error page to the end user.

2.4.4 Validation

XFA incorporates an event-based model for form validation. However this model does not always fit well with user requirements, for instance in the following circumstances:

- When forms are required to perform validation on a page by page basis; adding validation patterns and scripts on field validation event is not appropriate in this scenario.

- When forms are submitted both as a way of temporary saving data to the server, and as the final submission of the completed form; in this case validation should be disabled to allow temporary saving of incomplete (and consequently invalid) data.

- Another very common requirement is to display validation errors on the page rather than using popup windows; the way the XFA validation model is implemented it can only report errors using popup windows.

MVC models often include built-in support for validation because the Controller is obviously a very convenient place to invoke validation rules. As one can expect, MVC form designs facilitate additional validation scenarios such as those listed above.

In our MVC form design we are can reuse scripted validation techniques which are very common in form development, such as the definition of an array of mandatory fields. At the same time we try to maintain the validation logic as close as possible to the objects that are subject to validation rules. In order to do so we attach a validation function to each fragment, and we let each form fragment define its own list of mandatory fields.

2.4.5 Printing

Printing may seem very trivial, but in MVC form design it becomes a bit more advanced because the form must suddenly be re-rendered *in full*, as opposed to the capture mode where the form is rendered *partially* – one capture view at a time. In addition, some form fragments may have been designed to be used for both data capture and printing, and when that is the case, the presentation of these fragments will typically change between capture and output.

This is the main reason why we have introduced the concept of view contexts, with two default contexts: input and output.

In our MVC form sample we have also incorporated a naming convention that facilitates the design of form fragments that can support both capture and output. This is how it works:

- Sub-forms prefixed "C_" are used only for capture; as a result they will be hidden when the form is rendered in output mode.
- Sub-forms prefixed "O_" are used only for output; as a result they will be hidden when the form is rendered in capture mode.
- Sub-forms without any of the above prefixes are used for both capture and output; as a result they will be visible at all times.
- The above rules are applied recursively.

2.4.6 Language Selection

Another simple naming convention can be used to define sub-forms which are language specific. This is done by suffixing sub-forms with the language code. When rendering a multi-lingual form in a specific language, the language suffix will be used to filter the sub-forms that should be visible and those that should be hidden.

3 Using Form Fragments

3.1 Benefits of Form Fragments

Form fragments provide an important benefit to the design of MVC forms.

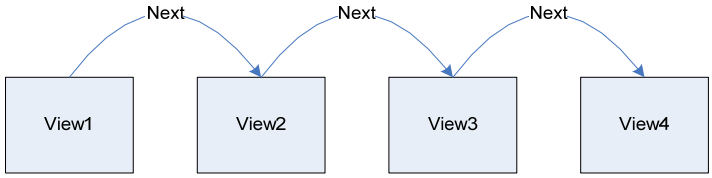
First of all, just like any other form, fragments allow the development of forms using a component based design. Componentization enables multiple developers to work simultaneously on the same project with minimal interference. Each fragment can be better managed and versioned in a source code control system, there is the ability to unit-test each form fragment, and most importantly, form fragments can be reused easily.

Reusability requires that fragments are stored together in the same folder (typically `./fragments` or `../fragments` in order to keep base templates and fragments linked to each other via a relative path)[5].

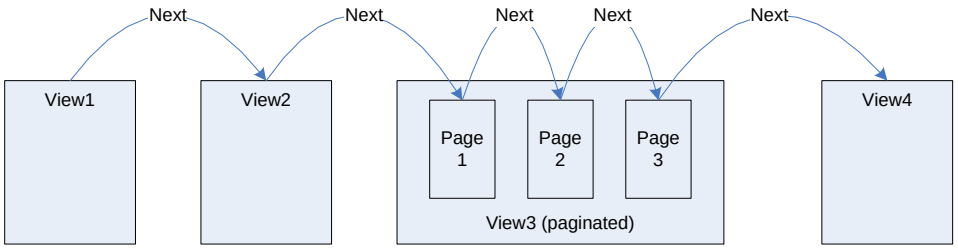
3.2 Fragment Delegation

3.2.1 Action Delegation

The handling of actions can be delegated to the fragment e.g. when the fragment itself is composed of multiple view states. Next/Previous buttons would normally call the form controller to define the next view.



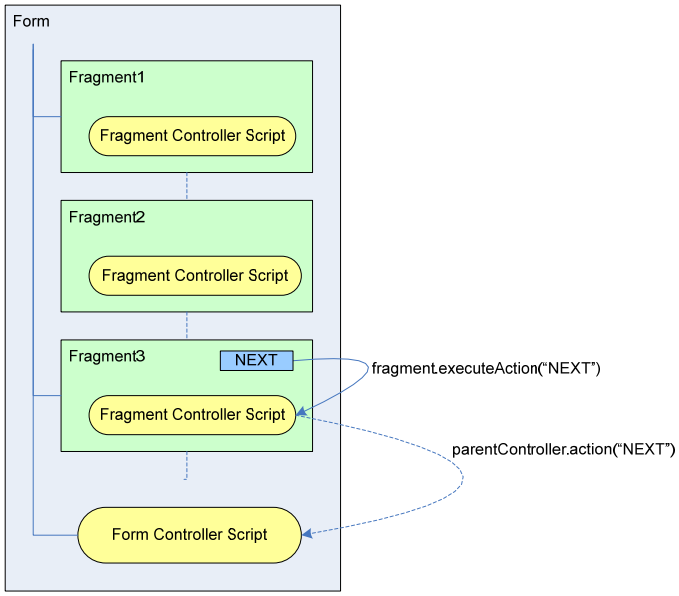
However a fragment that includes pagination logic needs to take control in order to define what the next page should be in a paginated list of items. In this case control must be delegated to the fragment.



In order to implement this delegation, the fragment should always invoke its own controller first, and if

[5] In our Eclipse reference project for MVC form development, we are using ../fragments because it allows us to have multiple form template folders accessing the same set of reusable fragments.

the requested action is not implemented within the fragment, it should then delegate the execution of the action to the parent controller.



3.2.2 Validation Delegation

Validation can be delegated to fragments; each fragment has its own validation logic, ideally it should not be centralized. At the same time what to do with validation results is the responsibility of the controller.

4 Logging

Logging in PDF form can be quite useful during form development, in the case of PDF form design, the ability to print messages on the Acrobat console is often used as a tracing mechanism. What we have implemented in our latest MVC form designs is a *scoLogger* scripting object that includes the following features:

- Logging categorization
- Log level that can be changed during execution.

Log levels can be defined in a 0 to 4 scale:

- 0: no log
- 1: error
- 2: debug
- 3: info
- 4: trace

To define a new logging category, simply use:

```
scoLogger.setLogLevel(category, level)
```

As soon as a logging category is set at a value greater than zero, the Acrobat console will automatically show up.

Those who have been using the Acrobat console for debugging have noticed that the Acrobat console can only hold a limited amount of text, and that Acrobat or Reader become very slow when the amount of text is important. To alleviate this problem the logger has a *maxLog* parameter which can be set to the maximum number of lines that will be written to the console before it is cleared up.

Usage of logging is really the responsibility of the form designer. If you don't need to use console logging – that's fine, simply set all log levels to 0 in the `scoLogger.init()` function.

If you still want to log messages somewhere but don't want to use the Acrobat console for that purpose, that's fine too, simply modify the code of the function `scoLogger.anyLog()`. All the other logging functions use `anyLog()` to log messages. A good example would be to send all logging messages to a multi-line text field in the form. You could also define an element in the application model schema that would contain all the messages logged during execution.

There is one logging category defined be default in our MVC form sample, and that category is named "Controller". It is also a good practice to define a logging category specifically for each complex form fragment.

5 Conclusion

We hope that the reader will have taken some useful information from this document. Maybe you will consider using this MVC form design framework in your future projects, but even if you don't and you are nevertheless engaged in some complex form development project, we hope that you will take some of the ideas described here for your own advantage and that you will be more successful as a result.

If you are interested to know more on this topic, please contact Adobe Consulting.