Guidelines for Dynamically Assembling Customized Forms and Documents

Adobe® LiveCycle® Designer 11

March 2013

Legal Notices

Contents

About this document

This Adobe® LiveCycle® Designer 11 document, Dynamically Assembling Customized Forms and Documents, provides a flexible and powerful way to combine forms and fragments.

The intent of this document is to guide users to successfully assemble customized forms and documents. Advanced form developers can take advantage of features beyond those described here.

Who should read this document?

This document is intended for individuals who are responsible for dynamically assembling customized forms and documents.

Additional information

The resources in this table can help you learn more about Adobe® LiveCycle® ES4.

For information about	See
Creating forms using Designer	Designer Help
Using the Assembler service	Adobe LiveCycle ES4 Services Reference Overview
Document Description XML (DDX) language	Assembler Service and DDX Reference
Using Adobe LiveCycle Output 11	Adobe LiveCycle ES4 Services Reference Overview
Editing XDC files	XDC Editor Help
Using Document Builder to create DDX files	Creating Assembly Descriptors Using Workbench

Introduction

Dynamically assembling PDF forms offers total flexibility in customizing XDP files to a specific need. First, you can author and manage related forms and fragments using Designer. Then, you dynamically assemble the forms and fragments together before they are rendered into a PDF form or document. Using this technique, complex logic can be applied to match the rendered form or document specifically to the usage required.

PDF forms can sometimes be complicated to use, for example, when a single form has to cater to every possible client. With a little information about the client, dynamic assembling makes it possible to effectively build a form to match that client's needs exactly. The result is a simplified filling process where the actual form is smaller and more efficient.

Similarly, documents often have common attributes. For example, a loan agreement for one state requires a certain waiver page to be included as a part of a group of pages in a loan agreement document. However, another state requires a different waiver page. In addition, certain clauses and sections on different pages may vary based on criteria. For example, the state the where the client lives, marital status, dependents, and loan options chosen. In both cases, a set of related forms and fragments can be assembled into a unique XDP file that is customized for a specific client.

Multiple forms can be concatenated into a larger form. As well, each XDP file can contain named insertion points that are used to place any fragment or group of fragments within the form. Through these simple techniques, forms and fragments can be assembled into a single XDP document and then used to render highly customized PDF forms and documents.

How it works

The example in this section provides a basic understanding of how dynamically assembling forms and fragments works. It takes you through the steps for assembling a simple patient clinic visit form. These steps are the building blocks for more involved implementations, upon which complex logic can be applied to specifically match the rendered form or document to the usage required:

1. In Designer, create a form design that contains insertion point placeholders for the fragments that are inserted into the form. Save the form design in XDP format.
2. In Designer, create one or more fragments and save them in XDP format. When the XDP files are dynamically assembled into a single XDP file, the fragments appear in place of the insertion point placeholder.
3. Create a Document Description XML (DDX) file indentifying the XDP files (forms and fragments) to be dynamically assembled into a single XDP file. The resulting XDP can then be rendered as a PDF form, a PDF, or print format document.
4. Submit the job to the Assembler service. (See LiveCycle Services.)

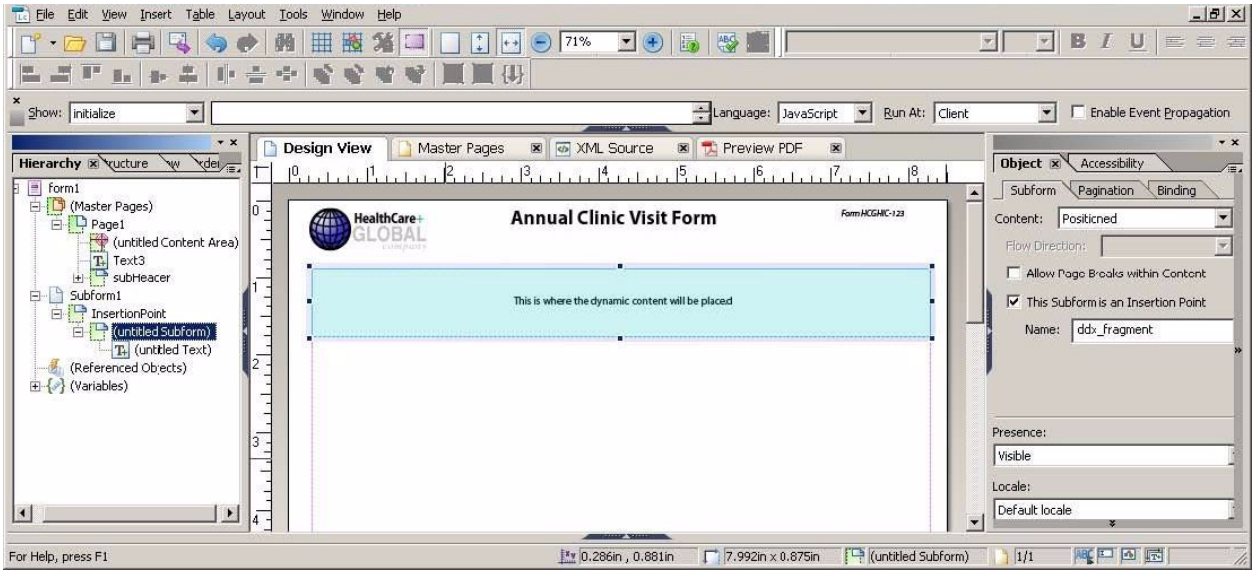
Create a form design

For this example, a simple form design has been created using Designer, and then saved as an XDP file. The form design's master page contains a logo, title, form id, and approval notes. Only one Insertion Point object has been added to the page.

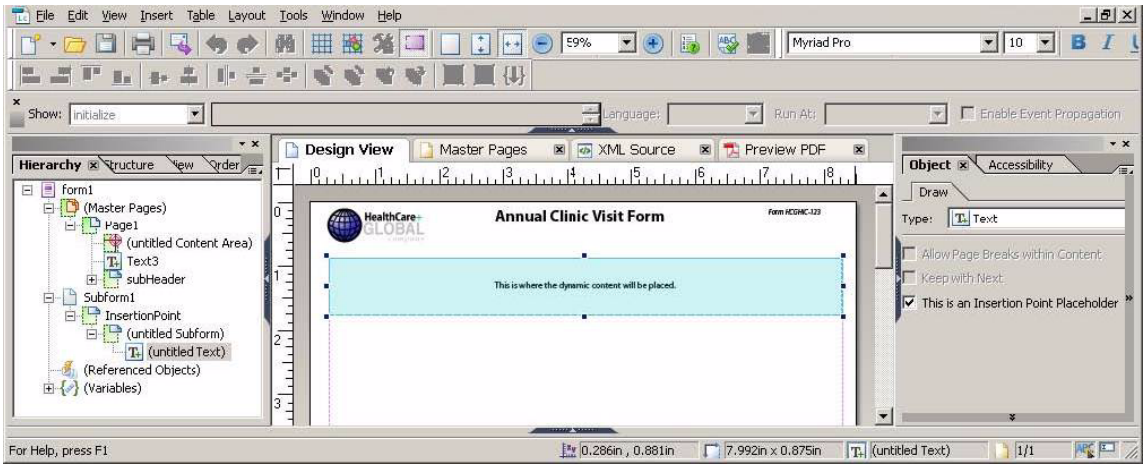
The Insertion Point object, named *InsertionPoint* by default, is a subform that acts as a placeholder for a fragment. The fragment is inserted into the form when the form is assembled on a server by using the Assembler service. (See Designer Help.)

Inside the InsertionPoint subform, there is a subform that contains an insertion point placeholder (a text object that contains temporary placeholder content). Notice that when the subform is selected in the Hierarchy palette, as shown in the following figure, the Subform tab shows that This Subform Is An Insertion Point is selected. For this example, the insertion point name is ddx_fragment. For the examples that follow, ddx_fragment is used in the DDX file to designate an insertion point.

Note: *Although giving subforms meaningful names is recommended, the subform names are not used to locate insertion points when the XDP files are dynamically assembled.*



When the insertion point text object is selected in the Hierarchy palette, you can see that This Is An Insertion Point Placeholder is selected in Draw tab. Form designers type placeholder content to assist them in developing the form. In this example, the placeholder content reads, "This is where the dynamic content will be placed." This content is automatically removed when a fragment is inserted when the XDP files are dynamically assembled into a single XDP file.



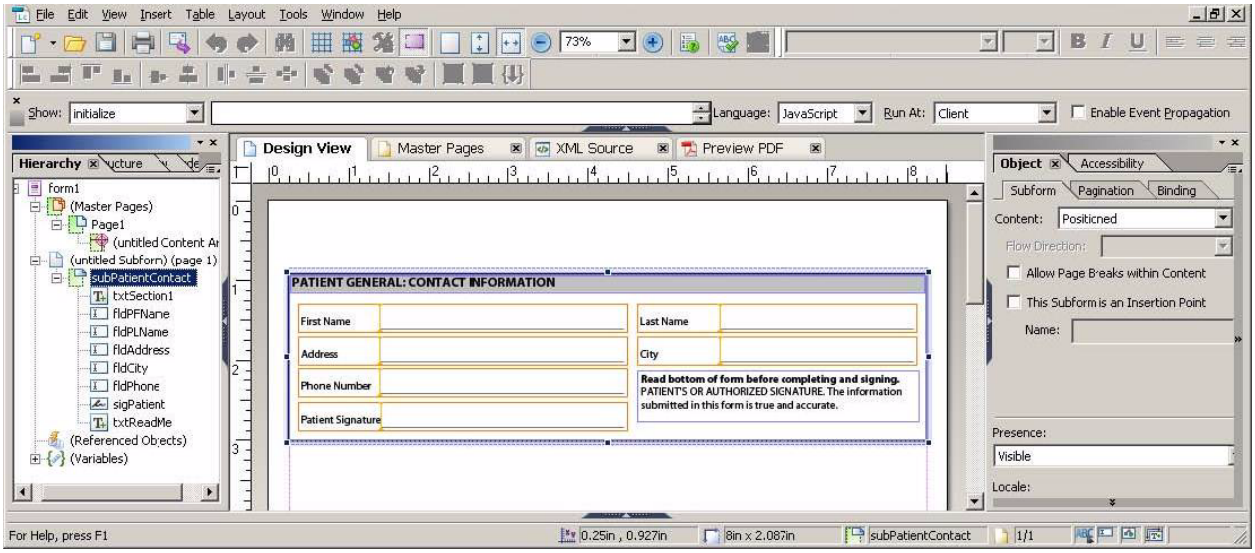
Create one or more fragments

Fragments are a reusable part of a form that can be inserted into multiple custom forms and documents. For example, a fragment can be a logo or a block of address fields.

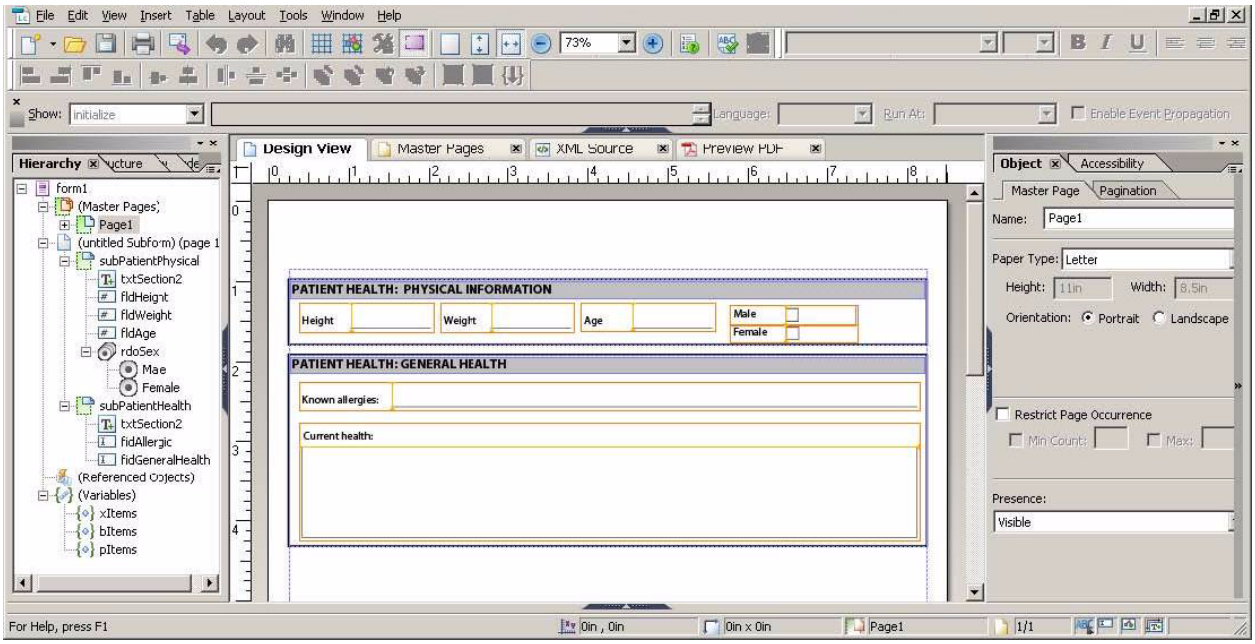
You can replace the selected objects in the current form design with a reference to a fragment. (See Using Fragments in Designer Help.)

For this example, three fragments are used:

- The file named wp_simple_contact.xdp, contains a single fragment, “subPatient Contact.”



- A second file named wp_simple_patient.xdp, contains two fragments, “subPatientPhysical” and “subPatientHealth”.



Create a Document Description XML (DDX) file

The Assembler service uses the DDX language to define the desired resulting document. For this example, the DDX file identifies the wp_simple_template_flowed.xdp file as the XDP source. The three fragments are defined in XDPContent and are inserted at the insertion point named “ddx_fragment” when the XDP is dynamically assembled.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ************************************************************************
 *
 *  Simple Dynamic Assembly with a single XDP with one insertion point and
```

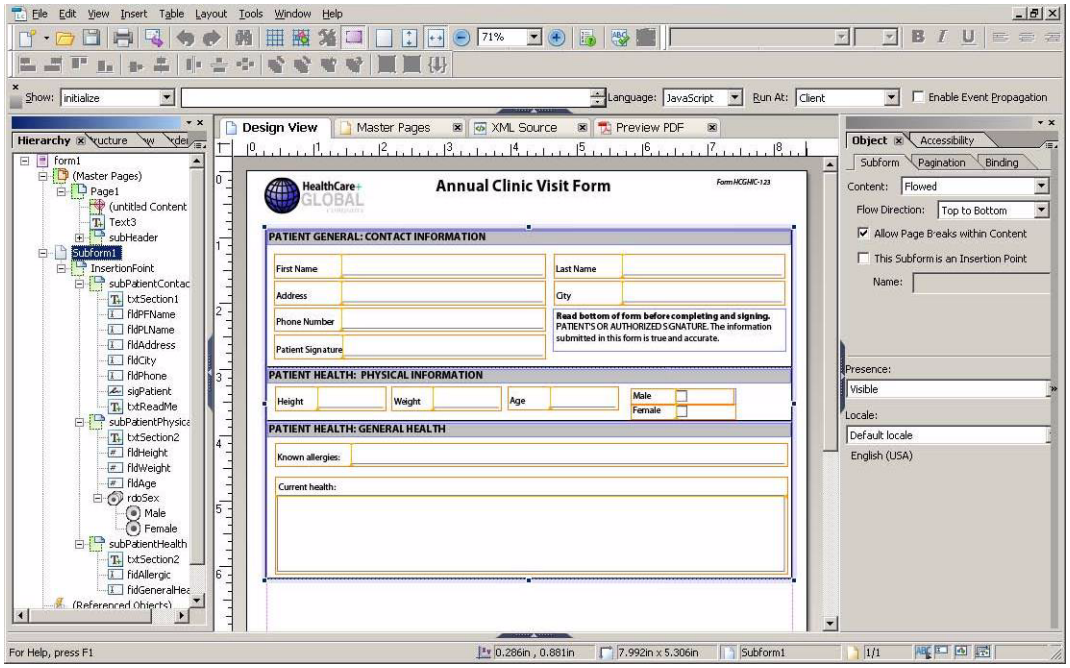
```
 *     multiple fragment flowed into that point.
 *
 * ************************************************************************ -->
<DDX xmlns="http://ns.adobe.com/DDX/1.0/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ns.adobe.com/DDX/1.0/
   http://spg.corp.adobe.com/pub/doc-o-matic/fibonacci/vrg/schemas/pdfm/ddx.xsd">

   <XDP result="wp_simple_result.xdp">
      <XDP source="wp_simple_template_flowed.xdp">
      <XDPContent insertionPoint="ddx_fragment" source="wp_simple_contact.xdp" fragment="subPatientContact"
      />
      <XDPContent insertionPoint="ddx_fragment" source="wp_simple_patient.xdp" fragment="subPatientPhysical"
      />
      <XDPContent insertionPoint="ddx_fragment" source="wp_simple_patient.xdp" fragment="subPatientHealth" />
      </XDP>
   </XDP>
</DDX>
```

Submit the job to the Assembler service

The form design saved as an XDP file, fragments, and DDX file are passed to the Assembler service to create the dynamically assembled XDP file.

Note: *Each fragment is inserted at the insertion point in the order in which they appear in the DDX file.*

You can view the resulting XDP file in Designer:



Dynamic Assembly Guidelines

The following are guidelines and considerations to follow when setting up forms and documents for dynamic assembly.

XDP file guidelines

This section describes guidelines and considerations to follow when creating the XDP files that you want dynamically assembled into a single XDP file.

Note: *Designer manages a UUID and timestamp as part of the XDP files it creates. Document Services takes advantage of these identifiers for caching. If an XDP file is hand edited without changing the UUID and timestamp, unexpected results could occur due to the caching. Therefore, avoid editing XDP files outside Designer. A new UUID and timestamp are generated for dynamically assembled custom forms and documents.*

Insertion point guidelines

It is recommended that you use a single insertion point for a single location in the XDP file. You can then define the insertion point in the DDX file to include multiple fragments. Similarly, use multiple insertion points in the XDP file whenever there is content that separates the insertion points.

There are several ways to insert an insertion point using Designer (See Creating an insertion point in Designer Help.):

- Insert an insertion point object
- Insert an insertion point into an existing subform
- Define an existing subform as an insertion point

The recommended method for creating an insertion point is to insert an insertion point object. The insertion point object automatically generates a flowed insertion point that is designed to maintain fragment position, while changing size to fit the fragment content. For more information, see "Positioning insertion points" on page 6.

You can also choose to define an existing subform as an insertion point. You can use a subform with a positioned layout if the fragment is a fixed size. For example, if you have a custom logo that is required to be a specified size. You can also use the insertion point object in this case.

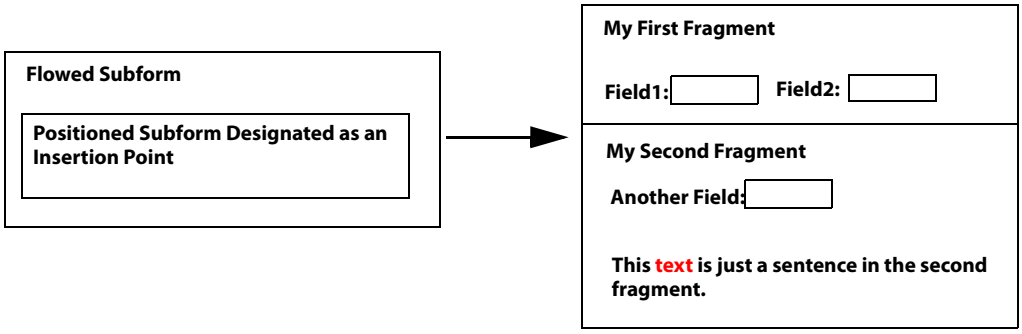
Naming insertion points

It is recommended that you give the insertion point a meaningful and unique name, unless you are inserting the same fragment in multiple insertion points. For example, if two insertion points are named "myInsertionPoint", fragments that are designated to be placed in "myInsertionPoint" are placed in both locations.

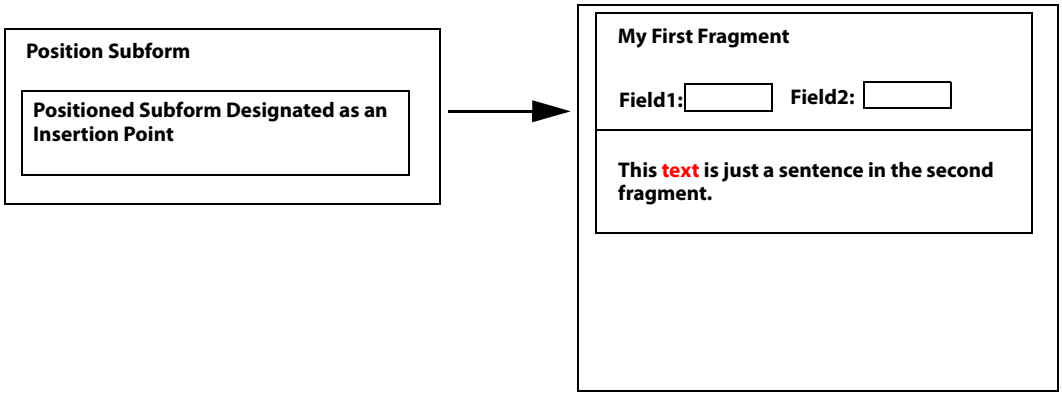
Positioning insertion points

As you work with insertion points, consider whether the insertion point subform manages fixed or flowed content. (See About Subforms in Designer Help.)

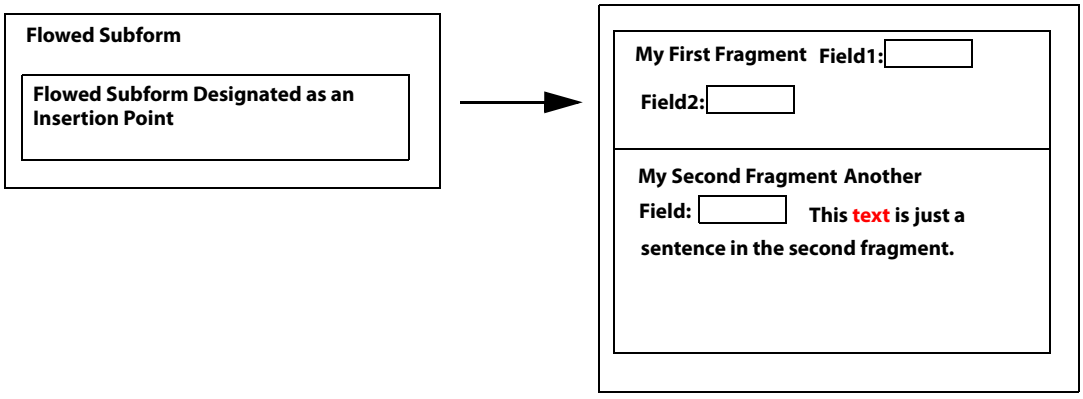
Inside the InsertionPoint subform, which flows content, there is a subform that positions content. The nested subforms allow the content inserted at the insertion point to retain its original relative position. Each added section of content flows down the page. For example, in the illustration below, both fragments, My First Fragment and My Second Fragment, maintain their original positions and flow the content.



If the InsertionPoint subform does not flow content, the fragments overlap as they are placed in the insertion point. In the following example, the My First Fragment appears but it covers up most of My Second Fragment:



If both the InsertionPoint subform and the subform inside flow content, the overall fragments flow and the content of each fragment reflows.



Sizing insertion points

It is recommended that you use the Auto-fit options in the Layout tab for the height of subforms designated as insertion points. Fragments inherit the height and width of the subform designated as the insertion point when placed into the form design. When you select the Auto-fit options, the inserted fragment uses its own size. By default, the fragment content overflows the space if the designated size is too small. Therefore, overlapping occurs if the height is a fixed value and the fragment is larger that this height. Conversely, gaps occur if the fragment is smaller than the fixed value.

The Auto-fit option is selected for the height of the insertion point object subforms, allowing the inserted fragment to maintain its own height. Moving an insertion point object after placing it in the form design can disable the Auto-fit option, causing the height to be set. Ensure that the Auto-fit options in the Layout tab are selected after you move an insertion point.

Placeholder content

You can designate the content within an Insertion Point subform as placeholder content. As shown in "How it works" on page 2, placeholder content is automatically inserted when you add the insertion point object to the form design. You can customize the content to indicate the purpose of the insertion point in your form. For example, you can provide a sample of the content that is inserted into the form or document to help the form designer visualize the final document.

The placeholder content is removed when the form or document is dynamically assembled. By default, the insertion points are not retained and therefore, the placeholder content is also removed regardless of whether the insertion point is used or not. If the DDX file specifies that insertion points are retained, then the placeholder content also remains for any unused insertion points.

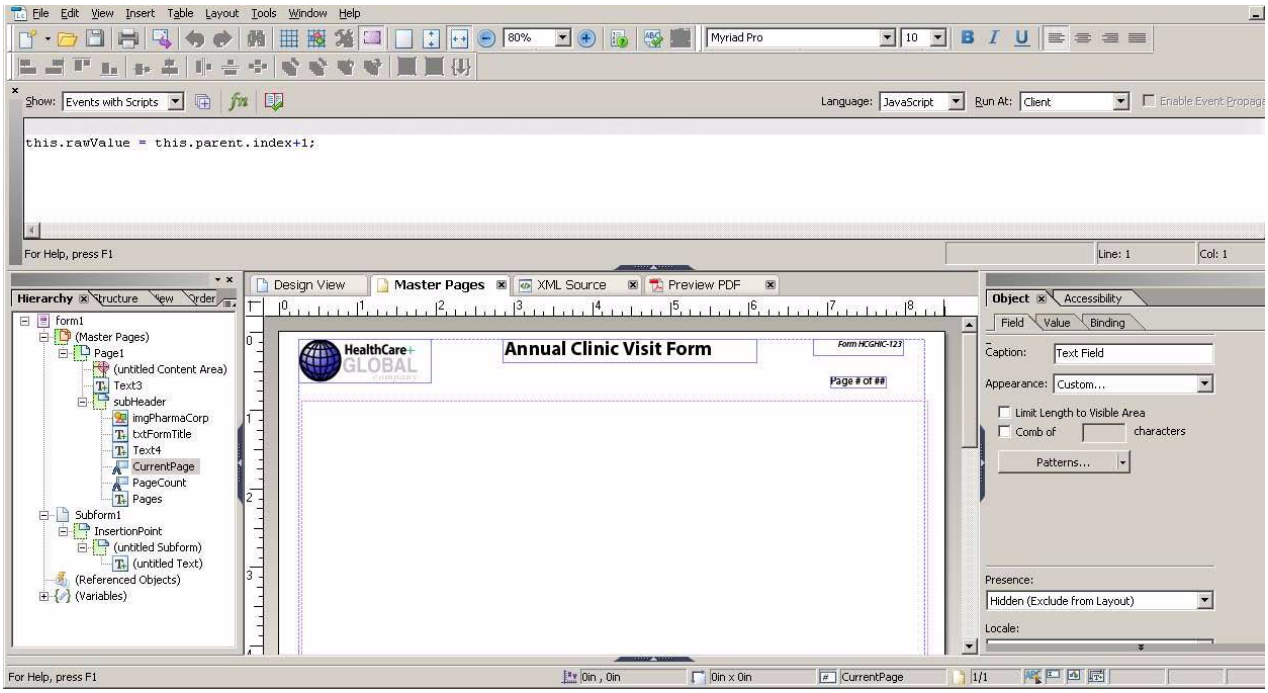
Master page guidelines

In Designer, you use master pages to add objects that appear in the same position throughout the form design. Master pages are useful for defining size and position of content areas, adding page numbering, and including headers and footers. You can add an insertion point in the master page for fragments that hold these elements. However, because master pages are only displayed as the content layout is put into them, do not add an insertion point for content.

When two complete forms are assembled into a single form, the master pages are retained for both documents. When the content begins for the second form, a page break is forced and the master pages from the second form are used.

The page size of each of the master pages is retained when the two documents are assembled. However, the page count is not reset when the content begins for the second form. To reset page numbering when the master pages from the second form are used, edit the page number scripts in Designer.

You use the page number objects in the Designer Custom palette to insert page numbers into the master page. Then, using the Script Editor, edit the object's script to set the raw value of the `CurrentPage` from `xfa.layout.page(this)` to `this.parent.index+1`. Also change the raw value of the total `PageCount` from `xfa.layout.pageCount()` to `this.parent.all.length`. (See Scripting in Designer Help).



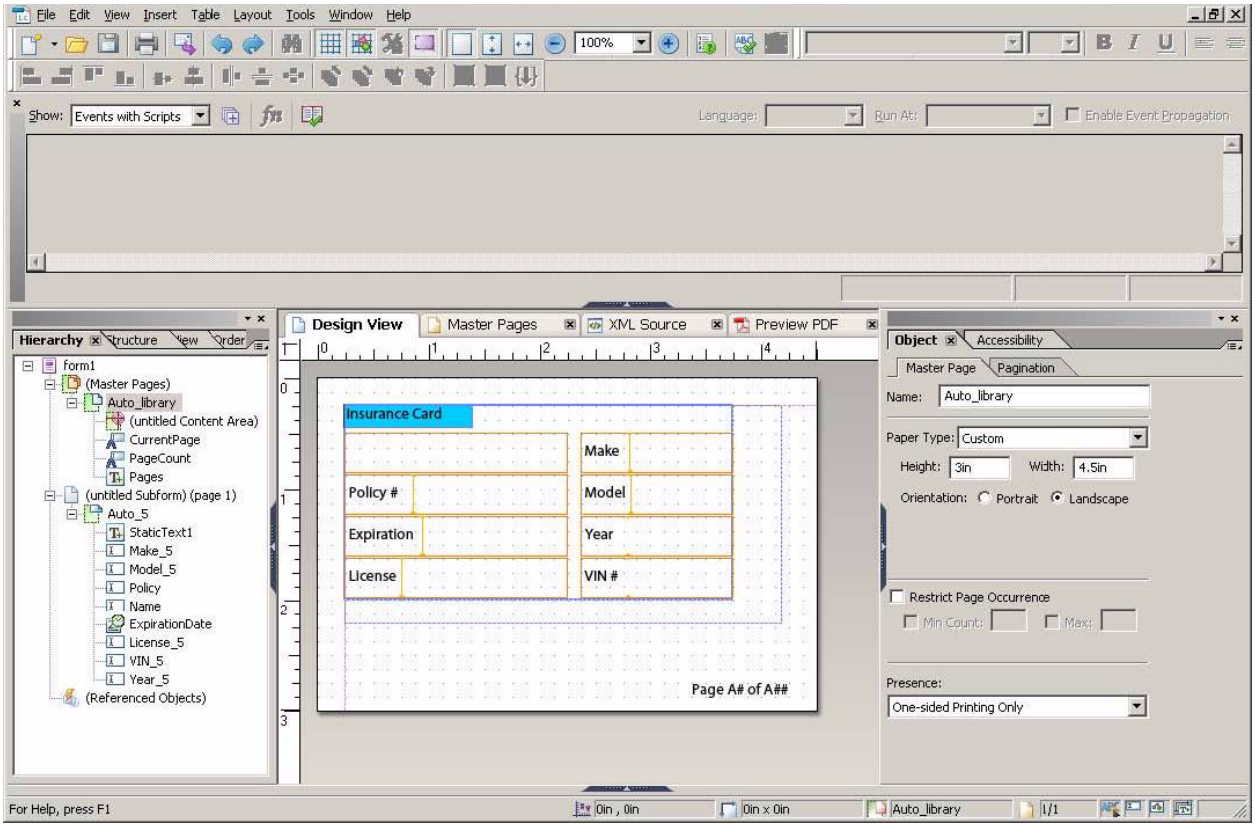
Example of modifying page numbers and size

This example demonstrates the different outcomes when assembling XDP files with master pages that use different page sizes and numbering schemes.

There are two XDP files:

- The wp_simple_template_pageNbrs.xdp file, in which the page size has not been changed and the page numbering starts on page 1
- The wp_simple_small_page.xdp file, which has a smaller page size

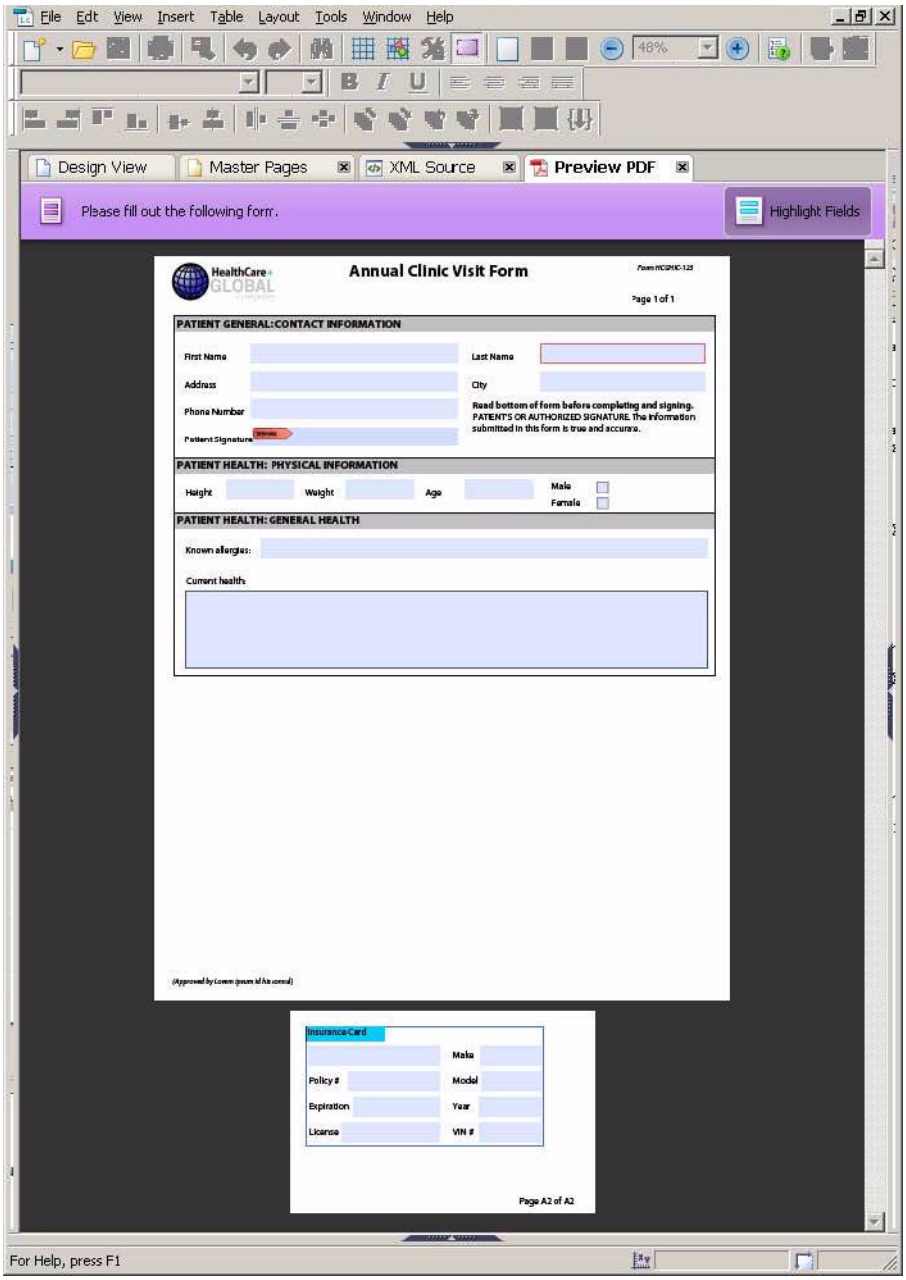
The scripts are not edited and therefore the page numbering is not reset when assembling the two forms. The page label has been modified to prefix an **A** to help identify the label used.



The wp_simple_template_flowed.xdp and wp_simple_small_page.xdp files are defined using the following DDX snippet.

```
<XDP result="wp_master_page_continuous_result.xdp">
   <XDP source="wp_simple_template_pageNbrs.xdp">
   <XDPContent insertionPoint="ddx_fragment" source="wp_simple_contact.xdp" fragment="subPatientContact" />
   <XDPContent insertionPoint="ddx_fragment" source="wp_simple_patient.xdp" fragment="subPatientPhysical" />
   <XDPContent insertionPoint="ddx_fragment" source="wp_simple_patient.xdp" fragment="subPatientHealth" />
</XDP>
<XDP source="wp_simple_small_page.xdp"/>
</XDP>
```

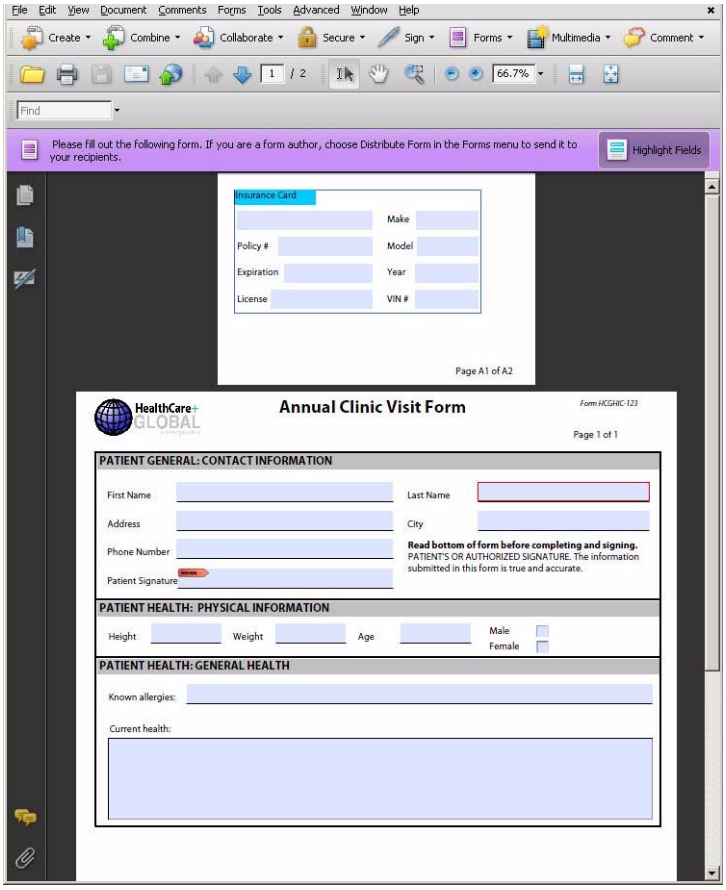
When the XDP files are assembled, the resulting XDP file starts at page 1 and the pages from the second wp_simple_small_page form start at page **A2**. Also, notice that the label reads Page 1 of 1. Looking at the script changes that were made earlier to the wp_simple_template_pageNbrs.xdp file, notice that the modified script specifies that the count be calculated based on the number of pages using that master page rather than the entire document.



The following DDX snippet results in wp_simple_template_pageNbrs.xdp and wp_simple_small_page.xdp assembled in the opposite order. Recall that the wp_simple_template_pageNbrs.xdp file has been edited so that the scripts for the page numbering start at page 1, even though it is now the second page in the result.

```
<XDP result="wp_master_page_discontinuous_result.xdp">
   <XDP source="wp_simple_small_page.xdp"/>
   <XDP source="wp_simple_template_pageNbrs.xdp">
      <XDPContent insertionPoint="ddx_fragment" source="wp_simple_contact.xdp" fragment="subPatientContact"/>
      <XDPContent insertionPoint="ddx_fragment" source="wp_simple_patient.xdp"
      fragment="subPatientPhysical"/>
      <XDPContent insertionPoint="ddx_fragment" source="wp_simple_patient.xdp" fragment="subPatientHealth"/>
   </XDP>
</XDP>
```

The result shows wp_simple_small_page starting at page 1 as expected. Notice that the page is labeled: A1 of A2. The reason is that the wp_simple_small_page script is using the default document page count. However, the appended pages from wp_simple_template_pageNbrs.xdp also start at page 1. The reason is that the scripts were edited so that the page number would start at 1 in the master pages. It also shows Page 1 of 1 because the script was modified to show the master page count rather than the full document count.



You can also edit the script so that the starting page number is a number other than 1 for an appended document. You specify the starting page number by adding that number to the script. In the following example, the script has been edited to start number with 5:

```
form1.#pageSet[0].Page1.subHeader.CurrentPage::ready:layout - (JavaScript, client)
this.rawValue = this.parent.index+5;
```

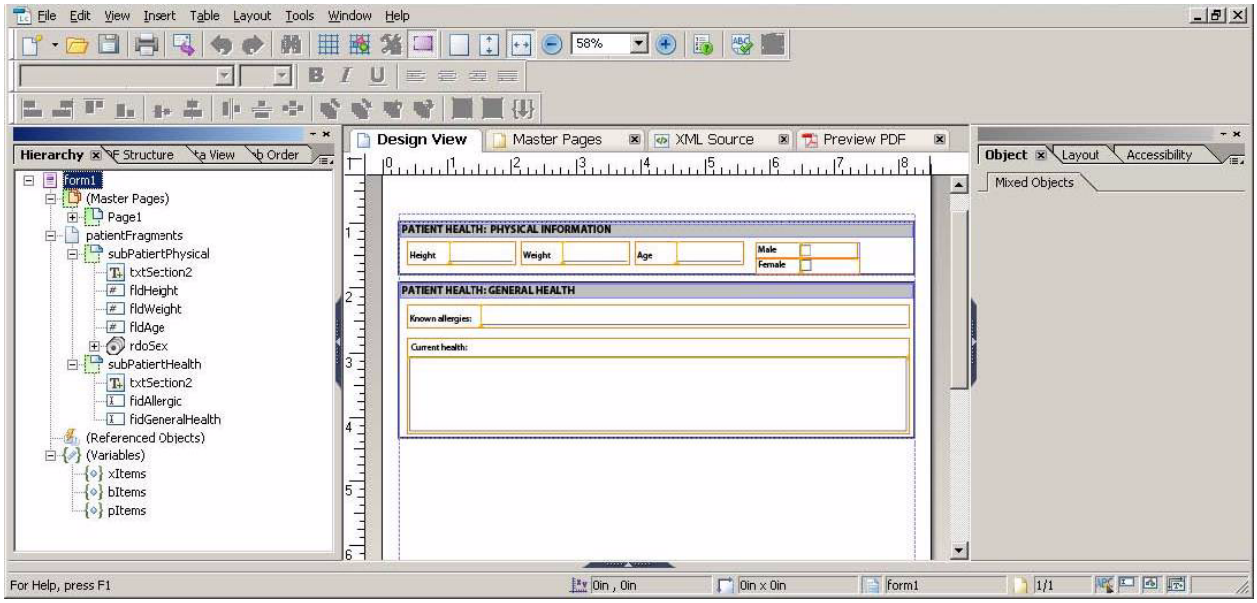
The best practice is to set up page numbering scripts consistently in all of the documents. To set up page numbering scripts consistently, use continuous labeling and do not change the scripts in any of the XDP files to be assembled. The result is a continuous page numbering scheme with the total count always being the full number of pages in the document. Otherwise, edit all of the script consistently. In the last example, if both forms had the same script modification, then the assembled form would have shown Page A1 of A1 and Page 1 of 1, instead of Page A1 of A2, where there is no A2.

Fragment guidelines

Although you can generate fragments using Designer, it is not necessary. The fragment is defined by designating the name of a subform to pull from any form created using Designer.

Assign fragments unique meaningful names unless you know that you want the same fragment name as in the sourceMatch selections described in "DDX Capabilities for Dynamically Assembling Forms and Fragments" on page 21.

In the following Designer form design, the Hierarchy palette shows three subforms: patientFragments, subPatientPhysical, and subPatientHealth. Any of these subforms can be used as a fragment when dynamically assembling a single XDP file. If the patientFragments fragment is used, then the content of both subPatientPhysical and subPatientHealth is inserted. In addition, the lower-level fragment, subPatientPhysical, can also be used to only insert a portion of the wp_simple_patient.xdp.



Using fragments created in Designer

You create a fragment using Designer by selecting the objects to include in the fragment, and then selecting Edit > Fragments > Create Fragment. (See Using Fragments in Designer Help.)Designer separates the content into its own file (saved in XDP format) to be used as a fragment. This fragment file does contain its own master pages; therefore, the fragment name must be specified for the XDPContent element. If the fragment name is not specified, both the fragment content and the master pages for the fragment are inserted at the insertion point.

Floating Field Guidelines

If a text object in a fragment contains a floating field reference, that reference must be within the fragment.

Manifest Guidelines

The manifest object is used in a several contexts: paper forms barcodes, Acrobat signatures, and data signatures. Any objects referenced from these manifests (Collections) must be within the fragment.

Tab Order Guidelines

Any explicit tab order definitions in the fragment must be self-contained. That is, no object inside the fragment can specify a tab position outside the fragment.

Locale Guidelines

For XDP files having an explicitly set locale, any inserted fragment must be consistent with that locale or must explicitly declare its own locale.

Script Guidelines

XDP files can include scripts. Here are some things to consider:

- Any script used by the fragment must be self-contained.
- Any referenced script objects must also exist in the same document.
- Any script in an XDP file that performs document-wide operations (for example, traverses objects in the form), must be generic and robust enough to work after fragments are inserted into the document.

Form Data Guidelines

Document Services does not provide any merging of data descriptions during assembly. The only data description that is retained is that of the base document. The base document is either the first document of the assembly or the source XDP file with the attribute `baseDocument="true"`. Only one document can be the base document. If more than one is specified, only the first instance where `baseDocument="true"` is used.

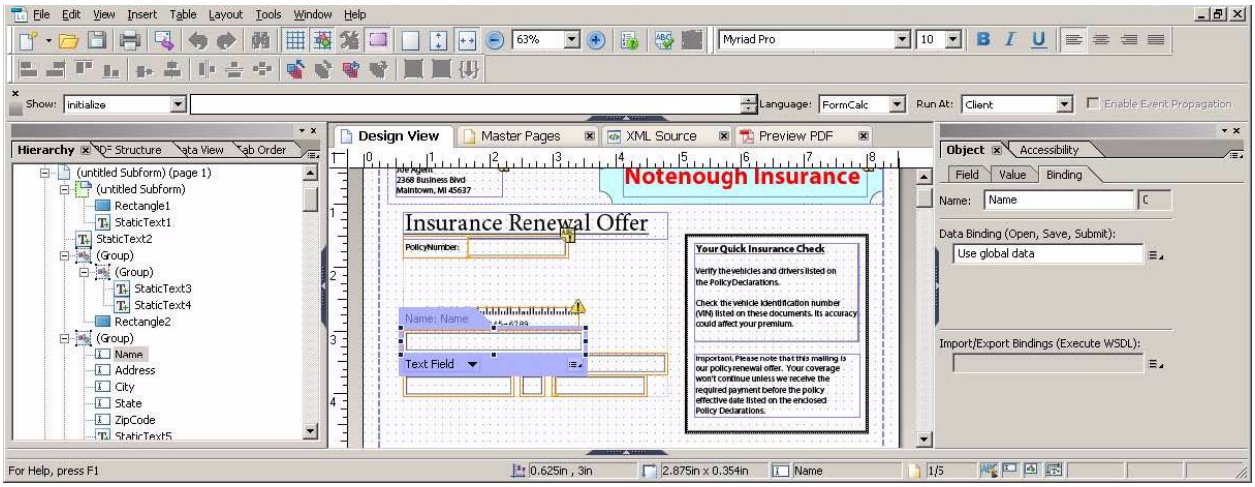
Although only the data description from the base document is retained, it is still possible to bind data for fragment fields. You can use global data binding on the form fields or provide the complete schema for the base document and fragments in the base document. For more information, see "Data binding using a schema" on page 16.

Using name and global data binding

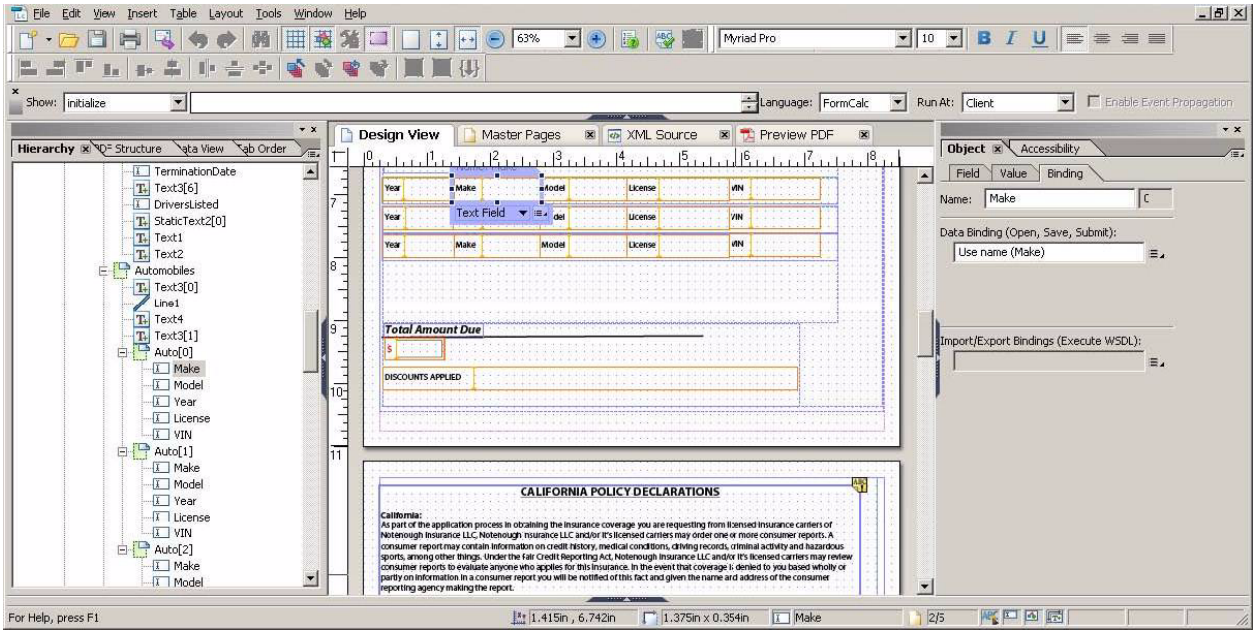
If the schema is expected to change or is unknown, set the binding type to Use Name or Use Global Data to make the best name match of the fields and data elements.

In Designer, set the binding type to Use Name to bind the XML data to the field based on a name match. The first name match binds that field to that data element from the data XML. The same XML data element cannot be used again for a match.

Use global data binding when the field that maps to the same data element is displayed multiple times. In the example below, the Binding tab in the Object palette shows that Use Global Data is selected for Name (displayed in the Name text box). The first match in the XML data for Name displays in every instance of the fields named Name. Global data binding is useful, for example, when the name of the client is displayed multiple times in the form.



For the following example, the XML data file contains multiple vehicle records. In the Automobiles subform, the binding type is set to Use Name (Automobiles). The Auto subform binding is set to Use Name (Auto). The Auto subform is repeated throughout the form, allowing each subform to take the next section of Auto data from the data XML.



The following is an abbreviated section of the data XML. By setting the binding type to Use Name (Make), the first instance of Make is "Dodge", the second is "Infiniti", and the third is "Tesla".

```
<Automobiles>
   <Auto id="1" Reference="Auto_1">
      <Year>1998</Year>
```

```
      <Make>Dodge</Make>
      <Model>Ram Pickup</Model>
   </Auto>
   <Auto id="2" Reference="Auto_2">
      <Year>2003</Year>
      <Make>Infiniti</Make>
      Model>I30t</Model>
   </Auto>
   <Auto id="3" Reference="Auto_3">
      <Year>2008</Year>
      <Make>Tesla</Make>
      <Model>Roadster</Model>
   </Auto>
</Automobiles>
```

The Use Name and Use Global Data binding types provide simple connections between the form fields and the data. Yet, they can also create situations that are difficult the debug. For example, if you add a new subform called X between Automobile and Auto and the Use Name binding type is selected, then Auto and all the children bindings will fail. To resolve the problem, select No Data Binding in the X subform.

You can use explicit data binding to avoid unintentional data and form field matching in data binding. However, modifying the data schema requires that you also modify the explicit binding. Explicit data binding requires that you bind the exact schema to the field. For example, if $.Automobiles.Auto[0].Make is entered into the Data Binding Object property for the Make field, then the Make from the first Auto element under Automobiles is bound in the data XML.

Note: *The data element can be bound to multiple form fields by specifying "$record.Automobiles.Auto[0].Make" in any form field that requires that value.*

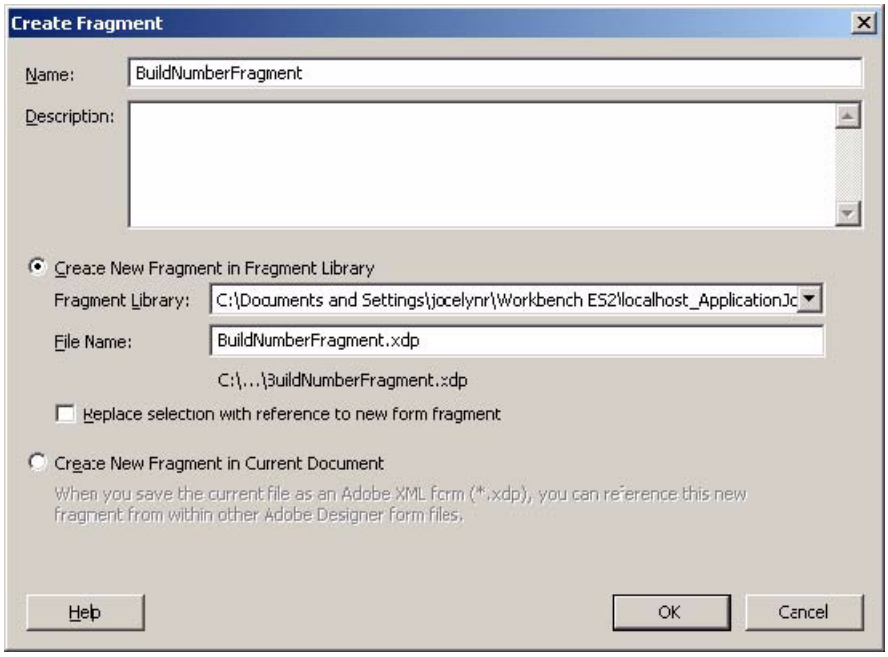
For more advanced binding of multiple records, use dynamic forms.

Data binding using a schema

In the case where the schema is known, the source XDP file can be created from the schema. Use Workbench to create an XDP file from the full schema. The data model is only retained on the source XDP file. It is important that the document includes the full schema for all fragments and appended XDP files, in addition to its own form fields.

When you create a fragment, you can either select an existing subform or select one or more objects in the form design. If you select objects that are not in a subform, the objects are wrapped in a subform when the fragment is created. Also, clear the Replace Selection With Reference To New Form Fragment checkbox in the Create Fragment dialog box. (See Using Fragments in Designer Help.) After the fragment has been added, designate the subform as an insertion point and either delete its contents or designate this content as placeholder content.

Through these steps, a source XDP file and a fragment have been created that can be referred to in the DDX file for assembly. When the fragment is inserted, the form field automatically binds to the schema from the XDP file.



Form Printer Directives

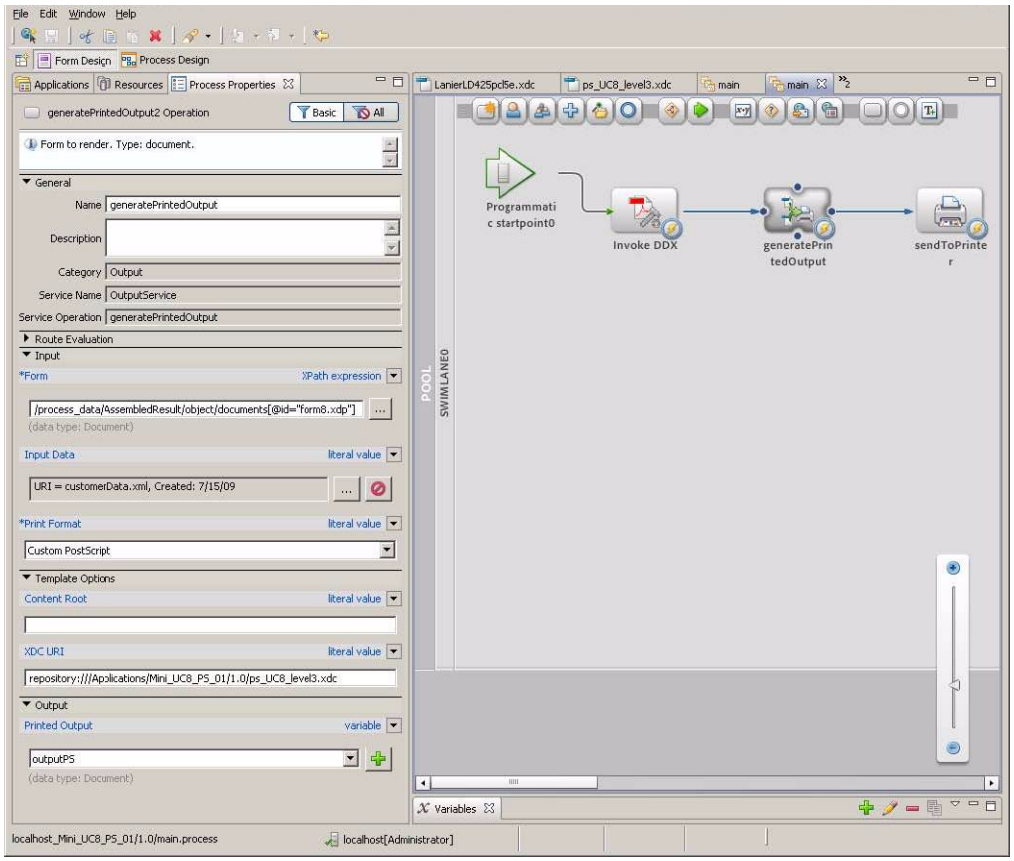
You can use the Output service to send a dynamically assembled XDP file directly to a printer.

To send the output directly to the correct printer and switch paper trays as required, you configure and map the media type in the XDC file that is associated with a printer to the selected paper type in Designer. To set up the printing features for the form, do the following tasks:

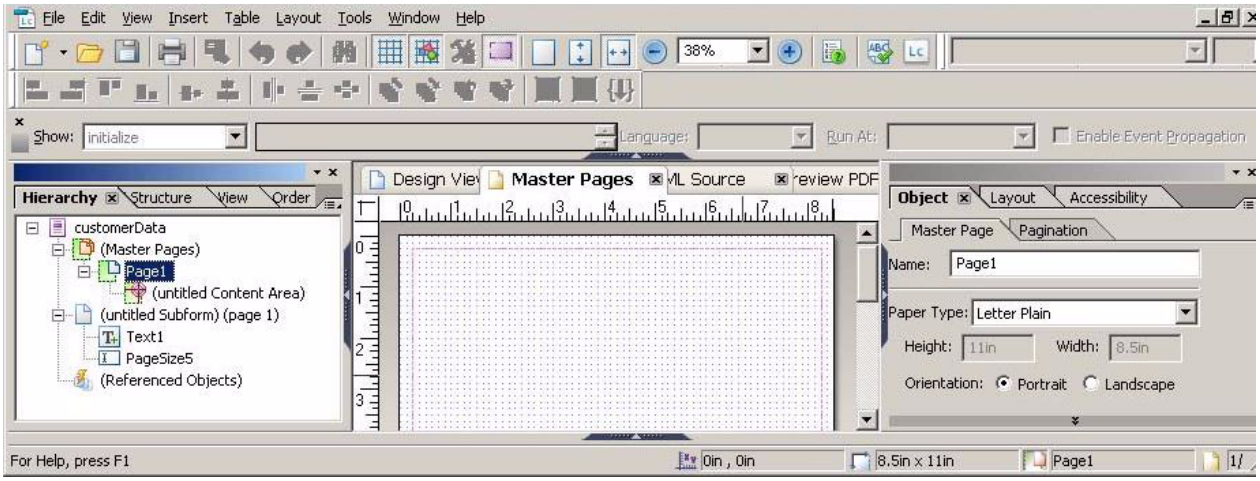
- In XDC Editor, create a custom XDC file that maps the selected medium to an input tray number for the printer. (See “Creating Device Profiles” in Workbench Help)
- In Designer, select the paper type (media type) for each XDP file

The following orchestration calls DDX to assemble the forms, sends the resulting form to generatePrintedOutput, which in turn, sends the form to the printer.

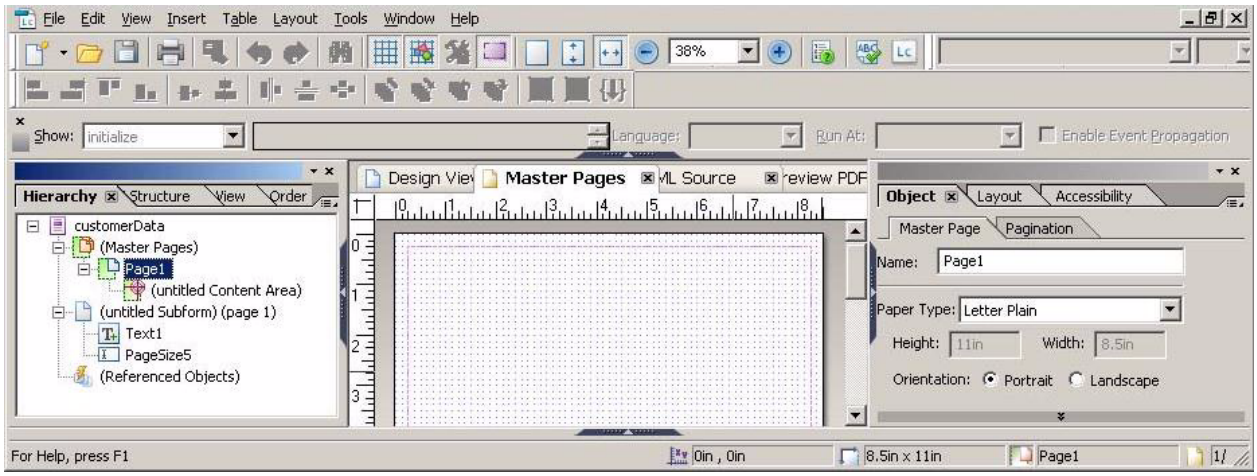
In the Process Properties tab for the generatePrintedOutput call, notice that a custom XDC file is selected in the repository. This XDC file is used to configure the form for printing.



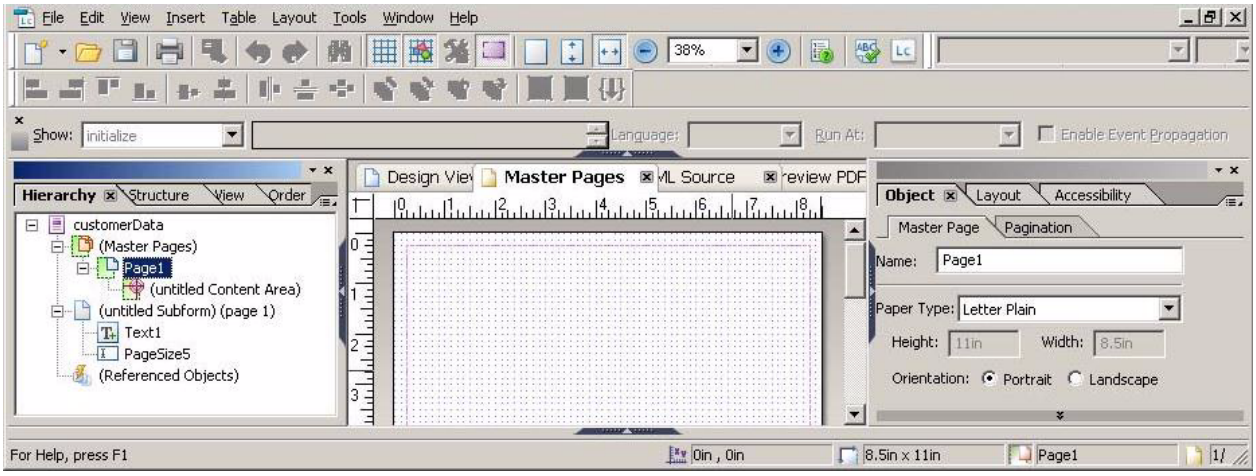
The custom XDC file maps the named medium to an input tray number for the printer. You define the XDC file to match the associated printer's capabilities and input trays. (See “Creating Device Profiles” in Workbench Help).



In Designer, you select the paper type (media type) in the Paper Type field in the Master Page tab of the Object palette. To achieve paper tray switching, you can select a different paper type for each XDP file before you submit the job to the Assembler service. In the following template8a.xdp file, Letter Letterhead is selected as the paper type for the Page1 master page:



In the following template8b.xdp file, Letter Plain is selected as the paper type for the Page1 master page:



When the template8a.xdp and template8b.xdp files are dynamically assembled, each file's content remains associated with its original master page. As the resulting document moves through the orchestration, the XDC file maps the paper type to a printer tray. The document is sent to the printer. After the last content page of the template8a.xdp file is printed, the paper tray switches to correctly print the content of the template8b.xdp file, as defined by its paper type.

DDX Capabilities for Dynamically Assembling Forms and Fragments

DDX encompasses several mechanisms that enable you to dynamically assemble forms and fragments. This section focuses exclusively on those mechanisms in DDX. (See Assembler Service and DDX Reference.)

Note: *DDX files start with the DDX element. However, for the purposes of this chapter, the examples are abbreviated to focus on DDX capabilities for dynamically assembling forms and fragments.*

In the following example, the master.xdp file includes two fragments:

- personal information (piInfoLib), based on a state
- automobile information (vehiclesLibs), based on the specific automobile.

```
<XDP result="finalform.xdp">
   <XDP source="master.xdp">
      <XDPContent insertionPoint="piInfo_goes_here"
          source="application:///UC_10_005/1.0/PolicyForms/Fragments/piInfoLib.xdp"
          fragment="inputmap:///stateName" required="false"/>
      <XDPContent insertionPoint="vehicles_list_goes_here"
          source="application:///UC_10_005/1.0/PolicyForms/Fragments/vehiclesLib.xdp"
          fragment="inputmap:///auto1Reference"/>
   </XDP>
</XDP>
```

You can store and manage assets such as forms, fragments, images, and XML schemas in the application space. The application space is a simple solution because it manages the assets in an application within the repository.

You use the application URL in the DDX file to reference the processes and assets. The application URL takes the format `application:///`, followed by application name, the version, and then the full path to the asset. For example:

`application:///UC_10_005/1.0/PolicyForms/Fragments/vehiclesLib.xdp".`

Selecting variable fragments

There are several methods available for dynamically selecting variable fragments that compose the form in the DDX file.

For complex cases, a DDX can be programmatically generated and passed to the Assembler service with all of the required section logic in the program. The methods described in this section are techniques for simple section methods within Workbench.

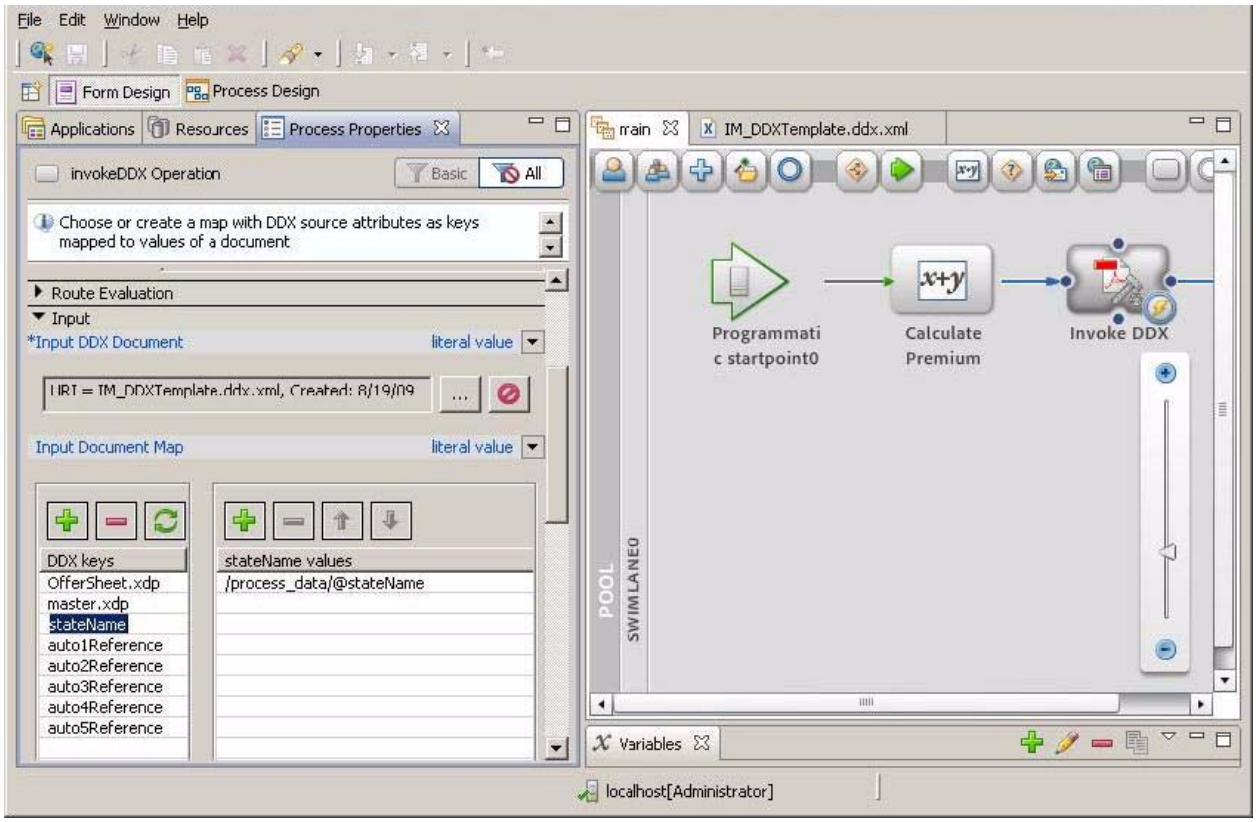
The recommended best practice is to use the input map.

Input map variables

You can specify input map variables in place of string and file references. The Assembler service InvokeDDX process uses an input map to specify the input for DDX processing. The DDX file references the input map names, which are assigned to values through the input map. The input map values can be set to literals or process variables. The input map URL starts with the `inputmap:///`, followed by a name. For example, using XDPContent, you can define a fragment to point to an input map variable using the input map URL:

```
<XDPContent insertionPoint="piInfo_goes_here"
   source="application:///UC_10_005/1.0/PolicyForms/Fragments/piInfoLib.xdp"
   fragment="inputmap:///stateName" required="false"/>
```

Then, in the Process Properties panel for the InvokeDDX process, the `inputmap` variable can be set to a literal value or a process variable. The following example shows the input map variable of `stateName` being set to the process variable of `stateName, "/process_data/@StateName"`.



Process variables

Process variables defined in Workbench can also be referenced from the DDX file using the process URL ("process:///"). In the previous input map example, the process variable URL could have been set directly in the DDX file:

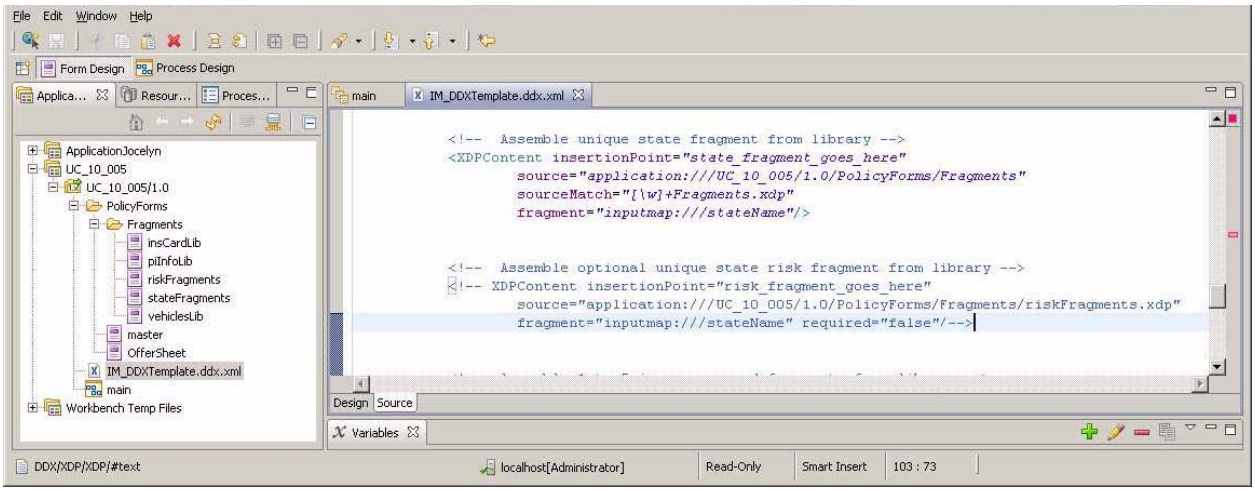
```
<XDPContent insertionPoint="piInfo_goes_here"
   source="application:///UC_10_005/1.0/PolicyForms/Fragments/piInfoLib.xdp"
   fragment="process:///process_data/@stateName" required="false"/>
```

However, using the process variable directly requires naming process variables during DDX design time and remembering to define these variables at the time of process definition. It also ties the DDX file to a specific process. By using the input map, the process variable definitions can be made during the process creation and mapped to the required input map variables.

sourceMatch attribute

You can also select files using the `sourceMatch` attribute. The `sourceMatch` attribute selects a file from a set of files based on matching criteria, which can be variable. This approach is useful when the fragments are organized into different files with the same fragment name. For example, the riskFragments.xdp and stateFragements.xdp files both contain fragments named by the state, i.e. "FL". You can use one insertion point to insert all of the same named fragments from the files that match the `sourceMatch`.

In the example below, both stateFragments.xdp and riskFragments.xdp files match `sourceMatch="[\w]+Fragments.xdp`. Therefore, the fragment name mapped to stateName is from each of these fragment forms and inserted at the insertion point. Because the default sort order is ascending based on the filename, the riskFragment.xdp appears first.



By setting the attribute `required="false"` on XDP elements, the inclusion of forms and fragments can be controlled. If `required="false"`, then forms and fragments that are not available, are not included with no errors.

Aggregation of XDPContent

By default, insertion points can be reused at the same scope, but not at the global scope. In the following example, the XDPContent labeled "local scope" is contained in the `XDP source="master.xdp` element. Because the XDPContent is local to the master.xdp file, the content is only applied to the insertion points that match in the master.xdp file.

On the other hand, because `fragment="anyState"` is a sibling to the `XDP source="master.xdp` element, the fragment is applied globally. The result is the stitching of its siblings. That is, the global XDPContent looks for insertion points named "state_fragment_goes_here" in the OfferSheet.xdp and master.xdp files. It also looks for any fragments inserted into master.xdp from the XDP content labeled "local scope".

For simplicity, the following example defines only one insertion point named state_fragment_goes_here in the master.xdp file.

```
<XDP result="finalform.xdp" aggregateXDPContent="false">
   <XDP source="OfferSheet.xdp">
   <XDP source="master.xdp">
   <!--  local scope -->
   <XDPContent insertionPoint="state_fragment_goes_here"
      source="application:///UC_10_005/1.0/PolicyForms/Fragments"
      sourceMatch="[\w]+Fragments.xdp"
      fragment="inputmap:///stateName"
      required=="false"/>
   </XDP>
   <!--  global scope -->
   <XDPContent insertionPoint="state_fragment_goes_here"
      source="application:///UC_10_005/1.0/PolicyForms/GenericFragments.xdp"
      fragment="anyState"/>
</XDP>
```

In this case, the DDX file has specified `aggregateXDPContent="false"` (which is also the default). If `sourceMatch` under "state_fragment_goes_here" has a match, only the XDPContent matched in the local scope is inserted into the document. Because aggregation is not allowed, the global scope XDPContent is ignored. However, if `sourceMatch` does not match any files, then the insertion point remains available for the global scope XDPContent.

In the previous example, there are typically state specific fragments that must be added to the form. Yet, there are some states that do not have state specific fragments. If no specific fragments for the state are found, the global scope inserts a generic anyState fragment, allowing customization with assurance that the generic is used when no customization is found.

For a situation where you always want the global scope but there are additional fragments at the local scope based on variables, use `aggregateXDPContent="true"`. Using `aggregateXDPContent="true"` allows the insertion point to be used at both the local and global scope until it is removed.

Generating an XDP or PDF result

Dynamically assembling forms and fragments can result in an assembled XDP file or a directly rendered PDF form. Ultimately, the XDP file must be rendered into a form or document before distributing to users. There are two different methods to facilitate rendering of the XDP result into a PDF form. First, the DDX file can specify a PDF result by wrapping the XDP elements inside the PDF result element. When this DDX is processed, the XDP elements are rendered to create a PDF result. Second, an orchestration can pass the XDP result to the Forms or Output services for rendering for finer control on render options.

The following a snippet of DDX show how to specify rendering during DDX processing by wrapping the XDP assembly into a PDF result. To separate the dynamic assembly of the XDP files and fragments from PDF assembly, all XDP elements must be enclosed in a plain XDP element.

```
<PDF result="finalform.xdp">
   <XDP>
      <XDP source="master.xdp"/>
      <XDPContent insertionPoint="state_fragment_goes_here"
          source="application:///UC_10_005/1.0/PolicyForms/GenericFragments.xdp"
          fragment="anyState"/>
   </XDP>
</PDF>
```

Processing the plain XDP element is the same as creating an intermediate result and passing it in as a PDF. Therefore, the DDX could also have been written as follows to generate the same result.

```
<XDP result="finalform.xdp">
   <XDP source="master.xdp"/>
   <XDPContent insertionPoint="state_fragment_goes_here"
      source="application:///UC_10_005/1.0/PolicyForms/GenericFragments.xdp"
      fragment="anyState"/>
</XDP>
```

```
<PDF result="finalform.pdf">
   <PDF source="finalform.xdp">
</PDF>
```

By using DDX to render the XDP, all Assembler features allowed on PDF files with XFA streams are also allowed on plain XDP elements. This most notable limitation is the allowance of only one XFA stream in assembled PDF files. This existing limitation does not allow each PDF file to have XFA streams to be assembled without removing the XFA. Therefore, an XDP assembly that contains an XFA stream cannot also be assembled with a PDF file having an XFA stream without removing the XFA.

To dynamically assemble more the one form with XFA streams, use the NoXFA element to remove the XFA stream from the resulting PDF document. The NoXFA element is useful when the data is applied to the XDP result before removing the XFA Stream. Use the flatten attribute to indicate whether the data be flattened into content or remain in non-XFA form fields.

```
<PDF result="finalform.pdf">
   <PDF source="anotherXFAForm.xdp"/>
   <XDP>
      <XDP source="master.xdp"/>
      <XDPContent insertionPoint="state_fragment_goes_here"
          source="application:///UC_10_005/1.0/PolicyForms/GenericFragments.xdp"
          fragment="anyState"/>
      <XFAData source="customerData.xml"/>
   </XDP>
   <NoXFA flatten="false"/>
</PDF>
```

To use an orchestration for rendering, use an XDP result to assemble forms and fragments. Then, pass the resulting XDP to the Output or Forms services process in the orchestration for rendering.

```
<XDP result="finalform.xdp">
   <XDP source="master.xdp"/>
   <XDPContent insertionPoint="state_fragment_goes_here"
      source="application:///UC_10_005/1.0/PolicyForms/GenericFragments.xdp"
      fragment="anyState"/>
</XDP>
```

You can specify how the Assembler service resolves the image references in the source XDP documents. References can be either absolute or relative. You can choose to have all the images embedded in the resultant so that it contains no relative or absolute references. You define this by setting the value of the `resolveAssets` tag, which can be none, all, relative, or absolute.

You can specify the value of the `resolveAssets` attribute either in the XDP source tag or in the parent XDP result tag. If the attribute is specified for the XDP result tag, its value will be inherited by all the XDP source elements which are children of the XDP result. However, explicitly specifying the attribute for a source element overrides the setting of the result element for that source document alone.

```
<XDP result="result.xdp" resolveAssets="all">
    <XDP source="input1.xdp" >
        <XDPContent source="fragment.xdp" insertionPoint="MyInsertionPoint" fragment="myFragment"/>
    </XDP>
    <XDP source="input2.xdp" />
</XDP>
```