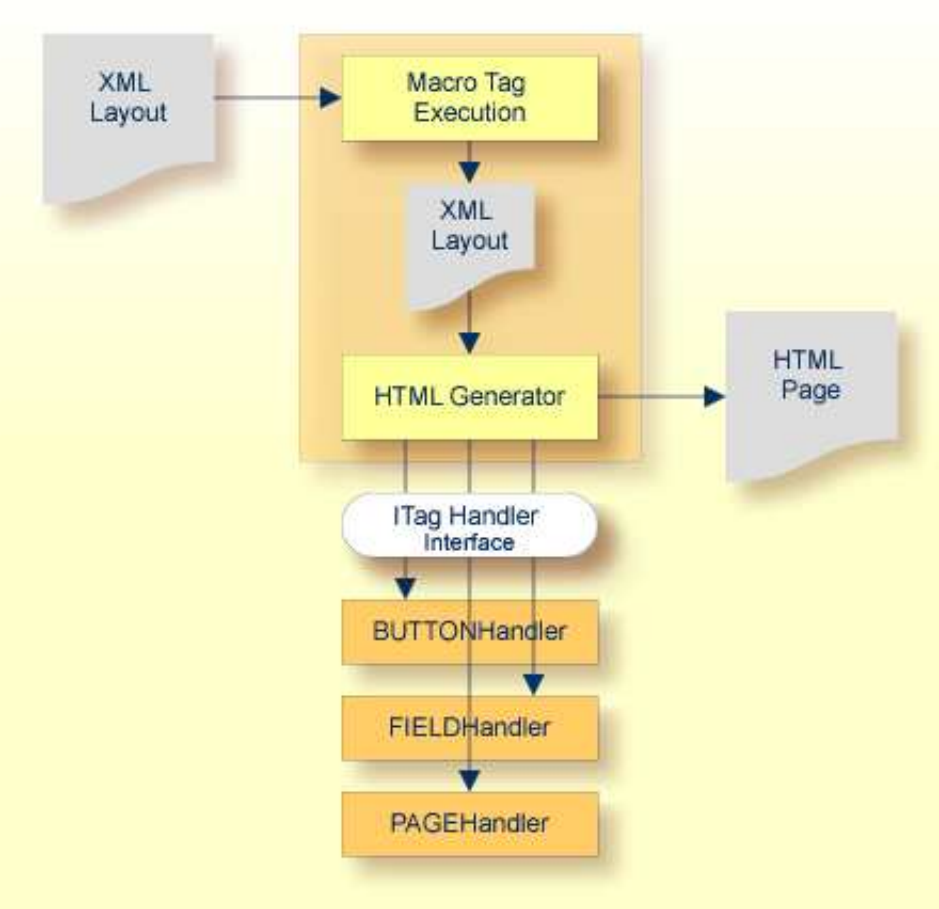
Control Concept

This chapter gives you an overview about the control concept. It covers the following topics:

- Page Generation
- Control Interface
- Library Concept
- Binding Concept
- Integrating Controls into the Layout Painter
- Summary

Page Generation

The page generation is the process of transferring an XML layout definition into an HTML/JavaScript page. It is automatically executed inside Application Designer’s Layout Painter when previewing a layout. It can also be called from outside processing.



A generator program (`com.softwareag.cis.gui.generate.HTMLGenerator`) is receiving a string which contains the XML layout definition. The generator program parses this string with a SAX parser and as a consequence processes the string tag by tag.

The generation of HTML pages is done in two steps:

- **Macro Execution**
  First, each tag of the XML layout is checked if it is a so called "macro tag". A macro tag is a tag which does not produce HTML output itself but which itself produces Application Designer XML tags. Imagine a control rendering an address input: this controls is using Application Designer controls in order to create some defined output area representing an address. The HTML is not produced by the address control directly - the address control internally creates normal Application Designer controls (e.g. fields, buttons, etc.) which themselves produce corresponding HTML code.

  The execution of macro tags is recursively done until no macro tag is contained in the XML layout anymore; i.e. macro tag themselves can internally use macro tags.

- **HTML Generation**
  After having executed the macros, the rendering of HTML is started: for each tag, the renderer creates one object of a tag handler class, that it finds via library definitions and naming conventions.

  Each tag handler is called via a defined interface (`com.softwareag.cis.gui.generate.ITagHandler`) and is invited to take part in the generation process. It gets all the tag data including the attributes from the layout definition and it gets the HTML string "on the right" and is allowed to append own information into this HTML string.

A tag handler instance is called at three different point of times by the generator:

- when the tag is starting (e.g. generator finds "<page…>"),
- when the tag is closing (e.g. generator finds ""),
- when the generator creates one defined JavaScript method which is called at runtime when the page is built up.

It is now the task of the tag handler to create HTML/JavaScript statements at the right point of time.

Control Interface

The following topics are covered below:

- IMacroTagHandler
- ITagHandler
- Call Sequence
- Extensions of IMacroTagHandler and ITagHandler

IMacroTagHandler

The interface `com.softwareag.cis.gui.generate.IMacroTagHandler` contains two methods which represent the different point of times when the generator calls the tag handler during the macro execution phase.

```
package com.softwareag.cis.gui.generate;

import org.xml.sax.AttributeList;
import com.softwareag.cis.gui.protocol.ProtocolItem;

public interface IMacroTagHandler
{
    public void generateXMLForStartTag(String tagName,
                                AttributeList attrlist,
                                StringBuffer sb,
                                ProtocolItem pi);
    public void generateXMlForEndTag(String tagName,
                                     StringBuffer sb);
}
```

A detailed information about the methods can be found inside the Javadoc documentation which is part of your Application Designer installation.

ITagHandler

The interface `com.softwareag.cis.gui.generate-ITagHandler` contains three methods that represent the different point of times when the generator calls a tag handler during the HTML generation phase.

```
package com.softwareag.cis.gui.generate;

import org.xml.sax.AttributeList;
import com.softwareag.cis.gui.protocol.*;

public interface ITagHandler
{
    public void generateHTMLForStartTag(int id,
                                        String tagName,
                                        AttributeList attrlist,
                                        ITagHandler[] handlersAbove,
                                        StringBuffer sb,
                                        ProtocolItem protocolItem);

    public void generateHTMLForEndTag(String tagName,
                                      StringBuffer sb);

    public void generateJavaScriptForInit(int id,
                                          String tagName,
                                          StringBuffer sb);
}
```

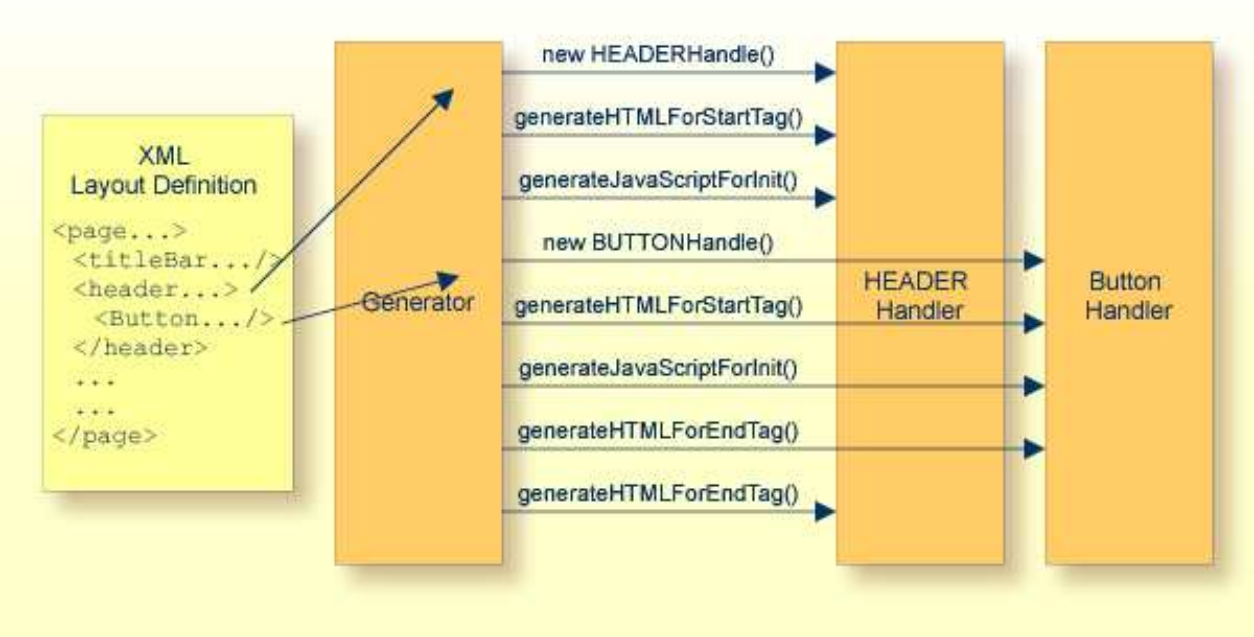
A detailed information about the methods can be found inside the Javadoc documentation which is part of your Application Designer installation.

Call Sequence

A tag is processed by the generator in a certain way that is now described for the HTML generation phase. (The macro execution phase is processed in an analogue way.)

- The generator finds the tag, reads its attributes and assigns an ID. The ID is unique inside one page.
- The generator creates a new instance of the tag handler which is responsible for processing the tag.
- The generator calls the `generateHTMLForStartTag` method. It passes the list of attributes, the string buffer which represents the HTML/JavaScript string and a protocol item in which the tag handler can store further information.
- The generator calls the `generateJavaScriptForInit` method. It passes as main parameter a string representing the method body of the `initialisation` method. You can append JavaScript statements to this string.
- (If the generator finds tags below the current tag, these tags are processed in the same way now.)
- The generator finds the end tag and calls the `generateHTMLForEndTag` method.

The following image illustrates the call sequence:



Be aware of the following:

- There is one instance of a corresponding tag handler per tag. If there are three button definitions inside a layout definition, then during generation there are three instances of the `BUTTONHandler` class.
- There is one instance of a protocol item which is passed as parameter per tag. Each tag has its own protocol item. All the protocol items are collected at generation point of time to form one generation protocol.

Extensions of IMacroTagHandler and ITagHandler

There are certain interfaces which extend the framework for specific situations:

- `com.softwareag.cis.gui.generate.IMacroHandlerWithSubTags` - this is an extension of `IMacroHandler` and provides the possibility to also receive subtags of a tag,
- `com.softwareag.cis.gui.generate.ITagWithSubTagsHandler` - this is an extension of the `ITagHandler` interface and provides the possibility to also receive the sub tags of a tag.
- `com.softwareag.cis.gui.generate.IRepeatCountProvider` and `com.softwareag.cis.gui.generate.IRepeatBehaviour` - these interfaces are responsible for controlling a special management for the REPEAT processing, which you use, for example, inside grids (ROWTABLEAREA2).

You do not need to know anything about these extensions to create your first controls. Documentation is provided inside the Javadoc documentation.

Library Concept

The library concept is responsible for defining the way how the generator finds a tag handler class for a certain tag. There are two situations:

1. The generator finds a tag without a ":" character. This indicates that this is a native Application Designer tag - the according tag handler is found inside the package `com.softwareag.cis.gui.generate`, the class name is created by converting the tag name to upper case and appending "Handler".

   For example, if the generator finds the tag "header", it tries to use a tag handler class `com.softwareag.cis.gui.generate.HEADERHandler`.

2. The generator finds a tag with a ":" character, e.g. `demo:address`. This indicates to the generator that an external control library is used. The generator looks into a certain configuration file (*<installdir>/config/controllibraries.xml*) and finds out the package name which deals with the "demo:" library. After having found the package name, the class name is built in the same way as with standard Application Designer controls.

   For example, if the generator finds the tag `demo:address` and in the configuration file the demo prefix is assigned to the package *com.softwareag.cis.demolibrary*, then the full class name of the tag handler is `com.softwareag.cis.demolibrary.ADDRESSHandler`.

What happens if the generator does not find a valid class for a certain tag? In this case, it just copies the tag of the layout definition inside the generated HTML/JavaScript string. Via this mechanism, it is possible to define, for example, HTML tags inside the layout definition which are just copied into the HTML/JavaScript generation result.

Control Libraries

A control library is a Java library containing `ItagHandler`/`IMacroTagHandler` implementations. The corresponding *.jar* file has to be part of the Application Designer application libraries in order to be found inside the Layout Painter and Layout Manager; i.e. it can be copied, for example, into the *<webappdir>/<projectdir>/appclasses/lib* directory.

The central control file for configuring control libraries in your installation is the file *<webappdir>/cis/config/controllibraries.xml*. An example of the file looks as follows:

```
<controllibraries>
        <library package="com.softwareag.cis.demolibrary"
                 prefix="demo">
        </library>
</controllibraries>
```

Each library is listed with its tag prefix and with the package name in which the generator looks for tag handler classes.

Binding Concept

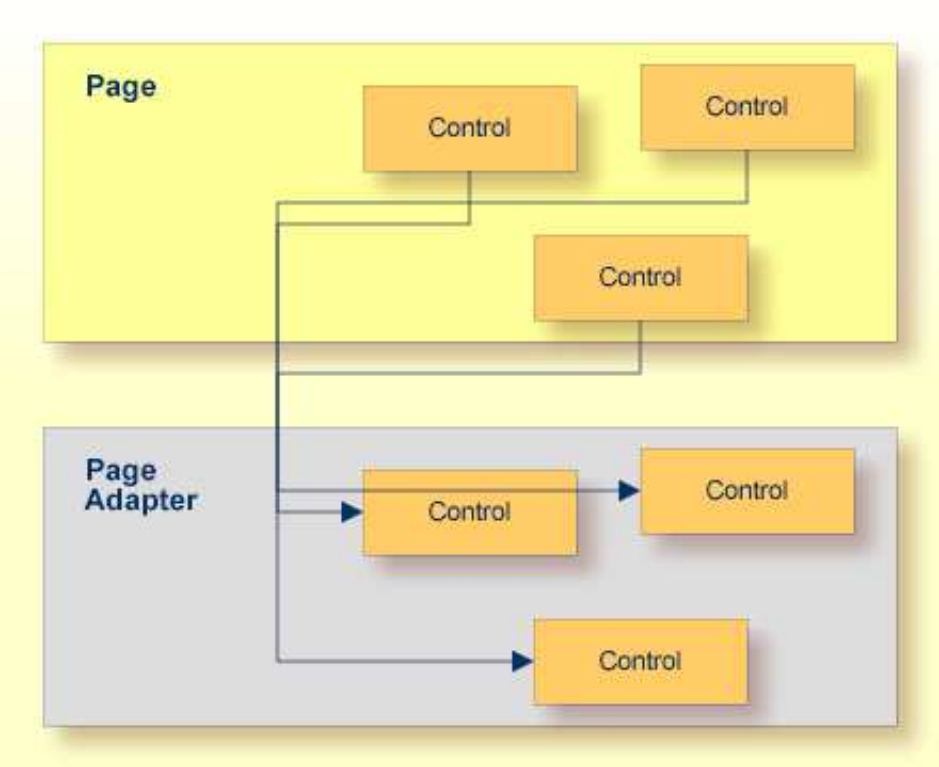
The normal binding concept between page and a corresponding class is:

- Controls refer to properties and methods.
- Properties and methods are directly implemented as `set`/`get` method or as straight methods inside the adapter class.

But: as you might already have read in the part *Binding between Page and Adapter* of the *Special Development Topics*, it is much more flexible. You can define hierarchical access paths for both methods and properties.

For example, you can define a FIELD control which binds to the property `address.street`. As a consequence, the adapter is first asked for an object via a `getAddress()` method. Then the result of this method is asked for `getStreet()`. The same is true for methods: in a BUTTON control, you can define the method `address.clear` - as a consequence, the adapter again is first asked for `getAddress()`, then the method `clear()` is called in the result object.

Why is this important with controls? Well, it is especially important for composing controls: you might want complex controls, e.g. an address control which internally is composed out of 10 FIELD controls, to be represented on the server side by a corresponding server class which matches the property and method requirements of the control. Even more: if you add an additional FIELD control to the address control, then you might not want to update all adapter classes, but just want to update the corresponding server class.



In analogy to the "Adapter", which is the representation of a whole page, the server side classes, which deal with certain controls, are called "Control Adapter" classes.

This all sounds a bit abstract - wait for the control adapter code example. Then you will see how powerful and simple this binding concept is.

Integrating Controls into the Layout Painter

Once having created new controls, you want to use them inside the Layout Painter. The Layout Painter is configured by a set of XML files, all of them located inside *<webappdir>/cis/config/*:

- *editor.xml*
- *editor_*.xml*

Have a look into the *editor.xml* file: all controls that come with Application Designer are listed inside this file. Each control defines the attributes that can be maintained and defines how it fits into other controls. Data type definitions to provide value help for the attributes is defined as well inside this file.

In short: *editor.xml* controls the way in which controls are presented inside the Layout Painter.

When creating new controls, you want to integrate your controls into the Layout Painter, i.e. you want to register them inside *editor.xml* as well. Instead of letting you directly manipulate *editor.xml*, there is an extension concept - in order to keep your definitions untouched by release upgrades. There are some *editor_*.xml* files, each of the files containing the definitions of *editor.xml* for a certain control library.

Have a look into the *editor_demo.xml* file:

```
<!-- DEMO:ADDRESSROWAREA2 -->
  <tag name="demo:addressrowarea2">
    <attribute name="addressprop" mandatory="true"/>
    <protocolitem>
    </protocolitem>
  </tag>
  <tagsubnodeextension control="pagebody" newsubnode="demo:addressrowarea2"/>
```

In this example, a new control `demo:addressrowarea2` is defined:

- It provides one attribute `addressprop`.
- It can be placed into the existing Application Designer control `pagebody`.

Or have a look at the following section:

```
<!-- DEMO:ADDRESSROWAREA3 -->
  <tag name="demo:addressrowarea3">
    <attribute name="addressprop" mandatory="true"/>
    <taginstance>
      <rowarea name="Address">
          <itr>
              <label name="First Name" width="100">
              </label>
              <field valueprop="$addressprop$.firstName" width="150">
              </field>
          </itr>
          <itr>
              <label name="Last Name" width="100">
              </label>
              <field valueprop="$addressprop$.lastName" width="150">
              </field>
          </itr>
          <vdist height="10">
          </vdist>
          <itr>
              <label name="Street" width="100">
              </label>
              <field valueprop="$addressprop$.street" width="300">
              </field>
          </itr>
          <itr>
              <label name="Town" width="100">
              </label>
              <field valueprop="$addressprop$.zipCode" width="50">
              </field>
              <hdist width="5">
              </hdist>
              <field valueprop="$addressprop$.town" width="245">
              </field>
          </itr>
          <vdist height="10">
          </vdist>
          <itr>
              <hdist width="100">
              </hdist>
              <button name="Clear" method="$addressprop$.clearAddress">
              </button>
          </itr>
      </rowarea>
    </taginstance>
    <protocolitem>
```

```
      <addproperty name="$addressprop$" datatype="ADDRESSInfo" useincodegenerator="true"/>
  </protocolitem>
</tag>
<tagsubnodeextension control="pagebody" newsubnode="demo:addressrowarea3"/>
```

The control `demo:addressarea3` has the following features:

- It provides one attribute `addressprop`.
- It contains the macro XML (between `<taginstance>` and `</taginstance>`) for building the control out of existing Application Designer controls.
- It binds to an address property of type `ADDRESSInfo` (between `<protocolitem>` and `</protocolitem>`).
- It can be positioned below the `pagebody` control.

The *editor_*.xml* files should not be maintained by yourself directly. Instead, use the Control Editor to define the file in a comfortable way.

Summary

When defining new controls, there are the following resources:

- *<webapp>/cis/config/controllibraries.xml* - to define control library prefixes and their binding to a certain Java package holding control implementations.
- *<webapp>/cis/config/editor_<yourchoice>.xml* - to define how controls fit into existing controls.
- `ITagHandler` implementations that transfer XML control definitions into HTML/JavaScript.
- `IMacroTagHandler` implementations that transfer XML control definitions into other XML control definitions.

The following section contains examples for building macro controls and new controls.