

Natural for Ajax

Java Custom Controls

Version 9.3.1

February 2025

This document applies to Natural for Ajax Version 9.3.1 and all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Copyright © 2009-2025 Software GmbH, Darmstadt, Germany and/or its subsidiaries and/or its affiliates and/or their licensors.

The name Software AG and all Software GmbH product names are either trademarks or registered trademarks of Software GmbH and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software GmbH and/or its subsidiaries is located at <https://softwareag.com/licenses>.

Use of this software is subject to adherence to Software GmbH's licensing conditions and terms. These terms are part of the product documentation, located at <https://softwareag.com/licenses> and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third-Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software GmbH Products / Copyright and Trademark Notices of Software GmbH Products". These documents are part of the product documentation, located at <https://softwareag.com/licenses> and/or in the root installation directory of the licensed product(s).

Use, reproduction, transfer, publication or disclosure is prohibited except as specifically provided for in your License Agreement with Software GmbH.

Document ID: ONE-NATNJX-CUSTOMCONTROLS-JAVA-931-20250213

Table of Contents

Preface	v
1 About this Documentation	1
Document Conventions	2
Online Information and Support	2
Data Protection	3
2 Overview	5
3 Control Concept	7
Page Generation	8
Tag Handlers and Macro Tag Handlers	9
Library Concept	13
Binding Concept	14
Integrating Controls into the Layout Painter	16
Summary	25
4 Composing New Controls Out of Existing Controls	21
Concept	22
Programmed Macro Control - Example	22
Configured Macro Control - Example	28
5 Creating Macro Controls Out of Existing Controls	31
General Information	32
Creating Macro Controls	33
6 Creating New Controls	41
Concept	42
Example 1	42
Njxdemos Sample Control: NADC:TEXTCONTROL1	44
JavaScript Functions	45
Example 2	47
Njxdemos Sample Control: NADC:TEXTCONTROL3	50
Example 3 (Applet)	51
Summary	56
7 Special Issues	57
Protocol Item	58
Bringing Controls into the Layout Painter	59
Bringing Controls into the Layout Painter - Data Types	60
Array Binding	61
Text ID/Multi Language Controls	61
8 Control Library	63

Preface

This documentation provides information on how to develop your own custom controls with Application Designer. It is organized under the following headings:

Overview	General information about custom controls and when to use them.
Control Concept	Details about the control concept and how to create custom controls.
Composing New Controls Out of Existing Controls	How to create macro controls from existing controls.
Creating Macro Controls Out of Existing Controls	How to create macro controls from existing controls.
Creating New Controls	How to create completely new controls.
Special Issues	Additional advanced topics for control creation.
Control Library	Gives information about a generally available control library in the web.

1

About this Documentation

■ Document Conventions	2
■ Online Information and Support	2
■ Data Protection	3

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Monospace font	Identifies service names and locations in the format <i>folder.subfolder.service</i> , APIs, Java classes, methods, properties.
<i>Italic</i>	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
Monospace font	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the symbol.
[]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Product Documentation

You can find the product documentation on our documentation website at <https://documentation.softwareag.com>.

Product Training

You can find helpful product training material on our Learning Portal at <https://learn.software-ag.com>.

Tech Community

You can collaborate with Software GmbH experts on our Tech Community website at <https://tech-community.softwareag.com>. From here you can, for example:

- Browse through our vast knowledge base.
- Ask questions and find answers in our discussion forums.
- Get the latest Software GmbH news and announcements.
- Explore our communities.
- Go to our public GitHub and Docker repositories at <https://github.com/softwareag> and <https://containers.softwareag.com/products> and discover additional Software GmbH resources.

Product Support

Support for Software GmbH products is provided to licensed customers via our Empower Portal at <https://empower.softwareag.com>. Many services on this portal require that you have an account. If you do not yet have one, you can request it at <https://empower.softwareag.com/register>. Once you have an account, you can, for example:

- Download products, updates and fixes.
- Search the Knowledge Center for technical information and tips.
- Subscribe to early warnings and critical alerts.
- Open and update support incidents.
- Add product feature requests.

Data Protection

Software GmbH products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

2 Overview

This documentation provides information on the Application DesignerNatural for Ajax control concept. It is recommended that you first become familiar with the “normal development” of screens inside Application DesignerNatural for Ajax.

When do you need custom controls? In general there are two cases:

1. You want to combine existing controls to form complex controls with a certain dedicated task. Maybe you want to define an “address area” control which consists of a certain arrangement of fields and labels that form an address. This kind of building controls is called “composing controls” in this documentation. These kinds of controls are called “macro controls” in this documentation - you take what is available and group it into certain units.
2. You want to create new controls - maybe you need some special kind of icon with a certain behavior.

While case 1 does macro controls do not require to deal with JavaScript and HTML, case 2 creating completely new controls requires knowledge of JavaScript and HTML and the use of the JavaScript library functions that are available via the Application DesignerNatural for Ajax framework.

Due to the usage of XML as the layout definition format and due to an open interface for integrating control definitions into the page generation process of this _product, the control concept is a flexible and open framework. Actually, all controls are following the framework - there is no “special way” or “shortcut” that is internally used.

Natural for Ajax supports a flexible and open control framework. The basic concepts of this control framework are XML as the layout definition format and interfaces for integrating control definitions into the page generation process of Natural for Ajax. This framework is not specific to custom controls. All Natural for Ajax controls are using this control framework themselves.

The first concept is the definition of controls, that is, control tags with certain attributes which you can integrate via a tag library concept into layout definitions. The second concept is the binding of the control to server-side adapter propertiesNatural adapter fields and events. Following the

strict Application DesignerNatural for Ajax architecture - that the GUI is a reflection of a “net data”/“model data”, dynamic controls have to transfer their data at runtime to/from adapter propertiesNatural adapter fields. This binding concept is important for new controls and for macro controls:

- On the one hand, you want newly created controls to reference the adapter properties/methods.
- On the other hand, you want to compose controls (for example, an address area) and want to bind them to complex objects (e.g. an address object) on the server side - already providing for a set of data and methods that fit to the control and provide some server side logic.
- When creating new controls, you want to bind the control to corresponding Natural adapter fields and events.
- When creating macro controls (for example, an address area), you sometimes want your control to provide some additional server-side logic. For advanced controls, this logic may go beyond the logic of the single controls which make up the macro control.

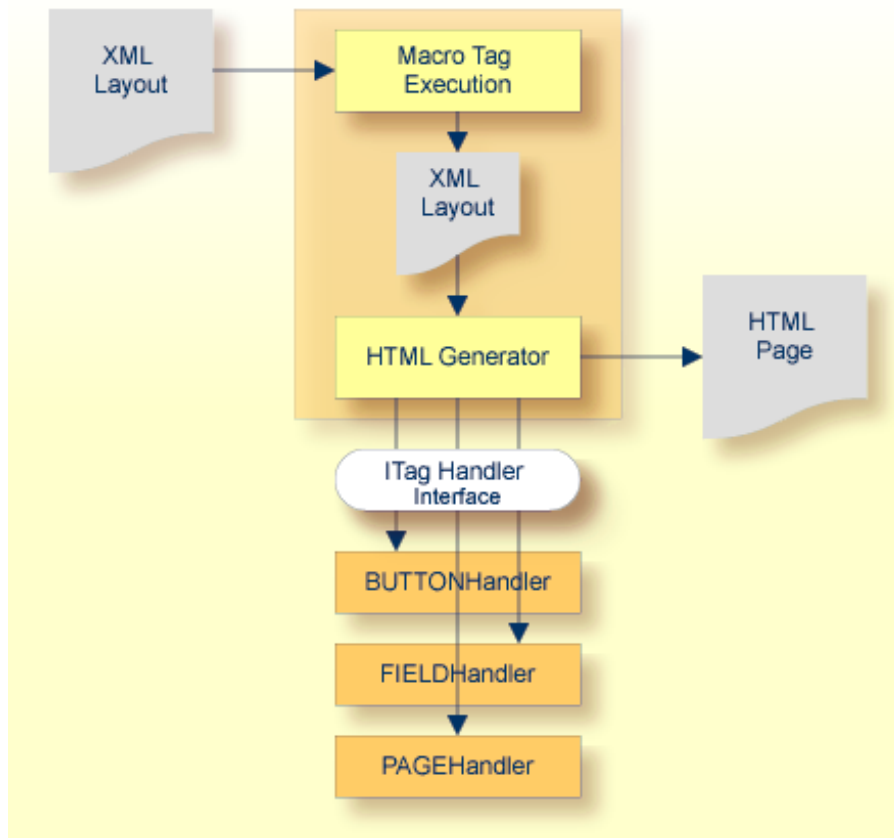
3

Control Concept

■ Page Generation	8
■ Tag Handlers and Macro Tag Handlers	9
■ Library Concept	13
■ Binding Concept	14
■ Integrating Controls into the Layout Painter	16
■ Summary	25

Page Generation

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



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 SAXAn XML 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/JavaScript output itself but which itself produces XML tags. Imagine a control rendering an address input: this control is using existing controls in order to create some defined output area representing an address. The HTML/JavaScript is not produced by the address control directly - the address control internally creates other controls (such as fields or buttons) which themselves produce corresponding HTML/JavaScript code.

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

■ HTML Generation

After having executed the macros, the rendering of HTML/JavaScript is started: for each tag, the renderer creates one object of a tag handler class, that it finds via library definitions and naming conventions. This is done by calling corresponding tag handlers for each tag. A tag handler is a Java class and is applied to the corresponding tag via naming conventions. The HTML generator instantiates objects of a tag handler class for the tags and calls corresponding methods of these tag handlers.

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, can contribute to the generation process. All tag data, including the properties from the layout definition, is passed to the tag handler. In addition, a string with the current HTML/JavaScript is passed and the tag handler can add its control-specific HTML/JavaScript to this HTML/JavaScript string.

A tag handler instance is called at three different points of time by the generator:

- when the tag is starting (for example, the generator finds "<page...>"),
- when the tag is closing (for example, the generator finds "</page>"),
- when the generator creates a defined JavaScript method which is called at runtime in the browser when the page is loaded.

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

Tag Handlers and Macro Tag Handlers

As described above for the HTML generation, control-specific tag handlers are called. The macro execution is either completely done based on XML definitions (see [Creating Macro Controls Out of Existing Controls](#)) or you can also implement a specific macro tag handler. This macro tag handler is then called during the macro execution. In case the macro execution is completely done based on XML definitions, a general macro tag handler is used internally.

Macro tag handlers and tag handlers are Java classes which implement specific interfaces. The corresponding classes are applied to the tags via the following naming convention: for a tag `<mytag>`, the corresponding Java class must have the name "MYTAGHandler".

The following topics describe the tag handler and the macro tag handler interfaces in more detail:

The following topics are covered below:

- [Macro Tag Handlers \(IMacroTagHandler\)](#)
- [Tag Handlers \(ITagHandler\)](#)
- [Call Sequence \(IMacroTagHandler and ITagHandler\)](#)
- [Extensions of IMacroTagHandler and ITagHandler](#)

Macro Tag Handlers (IMacroTagHandler)

The interface `com.softwareag.cis.gui.generate.IMacroTagHandler` contains two methods which represent the different points of time 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);
}
```

Detailed information about the methods can be found inside the Javadoc documentation which is part of your installation.

Tag Handlers (ITagHandler)

The interface `com.softwareag.cis.gui.generate-ITagHandler` contains three methods that represent the different points of time 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);
}

```

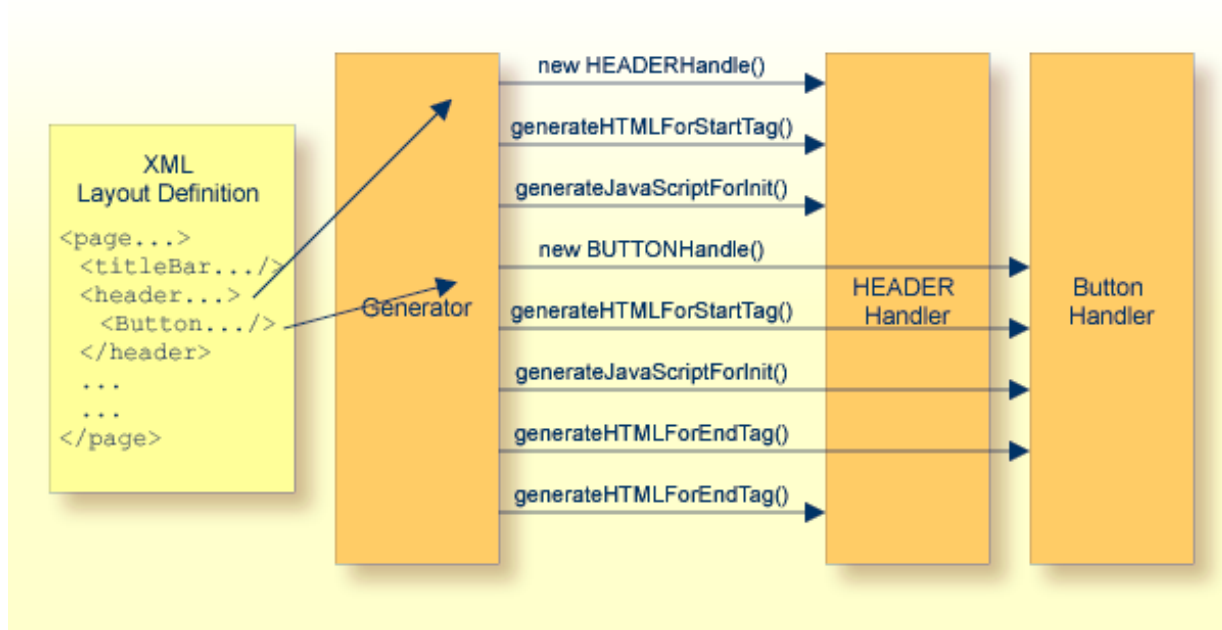
Detailed information about the methods can be found inside the Javadoc documentation which is part of your installation.

Call Sequence (IMacroTagHandler and ITagHandler)

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 a similar way.)

- The generator finds the tag, reads its properties 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` or `generateXMLForStartTag` method correspondingly. It passes the list of properties, the string buffer which represents the HTML/JavaScript or XML string correspondingly and a protocol item in which the tag handler can store further information.
- For tag handlers only: 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` or `generateXMLForEndTag` method correspondingly.

The following image illustrates the call sequence for tag handlers:



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 subtags 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. Details are provided inside the Javadoc documentation of your installation.

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 Application Composer Natural for Ajax tag control from the Natural for Ajax product - 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, for example, `demo:address`. This indicates that an external control library is used. The generator looks into a certain configuration file (`<install_dir>/config/controllibraries.xml`) and finds out the package name which deals with the "demo:" library. There is a central configuration file (`<install_dir>/config/controllibraries.xml`) which contains the external control library names and the corresponding Java package names. Besides this central configuration file, also corresponding configuration files on the user interface level are supported. This is explained later. 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.

When writing your own controls, be sure to use a tag name with your own prefix (such as `test:mycontrol`) and use your own Java package name which must not start with `com.softwareag`. The tag names without prefixes and the Java package `com.softwareag` are reserved for the Natural for Ajax product.

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 Composer Natural for Ajax 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 corresponding `.jar` or `.class` files can be copied either to the central `WEB-INF/lib` or `WEB-INF/classes` directory of the *cisnatural* web application, or they can be copied to the `.appclasses/lib` or `.appclasses/classes` subdirectory of a user interface component.

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.

As an alternative to this central control file, you can have a file `./cisconfig/controllibraries.xml` in your user interface component (for example, `njxdemos/cisconfig/controllibraries.xml`). The format of this file is identical to the central control file `controllibraries.xml`.

Binding Concept

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

- Controls refer to properties and methods.
- For pages implemented with Java, `pProperties` and methods are directly implemented as `set/get` methods or as straight methods inside the adapter class.

As you might already have read in the part *Binding between Page and Adapter* of the *Special Development Topics* (part of the Application Designer documentation), the binding is much more flexible. You can define hierarchical access paths for both methods and properties.



Note: The Application Designer documentation is included in the Natural for Ajax distribution package.

- For pages implemented with Natural, properties and methods are mapped to an internally used XML representation of the data. The Service Data Objects technology (SDO) is used for manip-

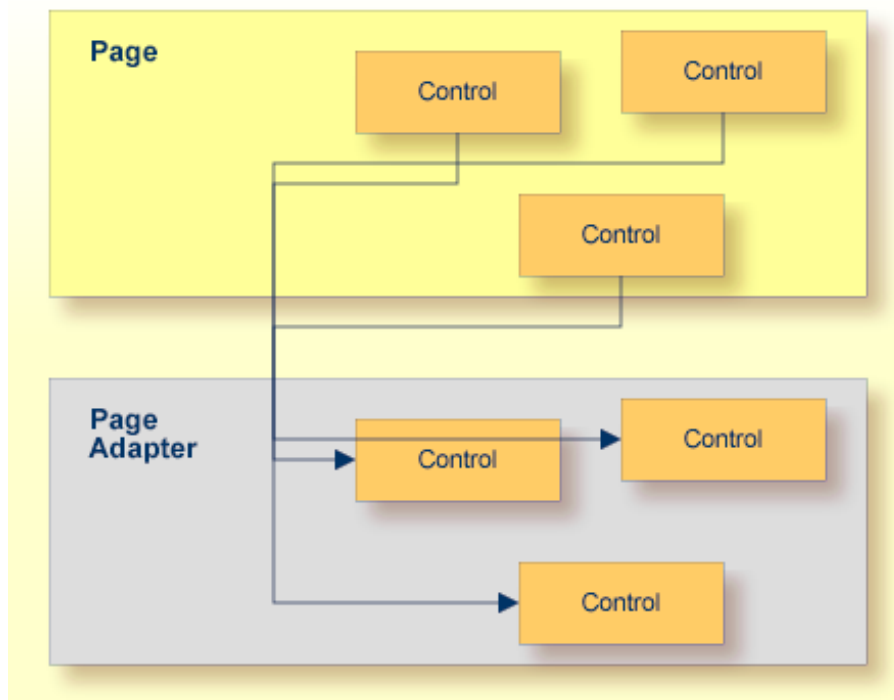
ulating the XML internally (see also http://download.oracle.com/otndocs/jcp/sdo-2_1_1-fr-oth-JSpec/). The XML is then bound to fields and events in the generated Natural adapter.

If you do not delegate the data binding to already existing controls, you need to call specific methods in your handler class which add the corresponding data structures to the SDO. The custom control examples in the Natural for Ajax demos contain corresponding guidelines.

For complex bindings, you sometimes need to implement a corresponding binding class. A binding class is a Java class which extends the class `com.softwareag.cis.adapter.ndo.NDOCustomControlInfoBase`. You sometimes use a binding class if you want to implement some control functionality which is not appropriate to be implemented in the JavaScript layer. There is also a naming convention for these binding classes. For a tag `demo:address`, the name of the binding class would be `ADDRESSInfo`. The custom control examples in the Natural for Ajax demos contain corresponding guidelines.

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`

The files with the name `editor_*.xml` are used to extend the Layout Painter with your own custom controls. These `editor_*.xml` files can either be located centrally in `<webappdir>/cis/config/` or in a subfolder `cisconfig` of a user interface component.

Have a look at the `editor.xml` file: all controls that come with Application DesignerApplication ComposerNatural for Ajax 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.

There is a central file *editor.xml* which defines the central controls that come with the Natural for Ajax framework. For each control, the properties and how the control fits into other controls is defined. In addition, data type definitions to provide value help for the properties are defined 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, that is, 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. Again naming conventions are used: for a control library named "demo", you would define your controls in a file with the name *editor_demo.xml*. This means that you will have one *editor_*.xml* file per control library.

Have a look at 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 property `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>
    </rowarea>
  </taginstance>
</tag>
```

```
<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" ↵
useincodgenerator="true"/>
</protocolitem>
</tag>
<tagsubnodeextension control="pagebody" newsubnode="demo:addressrowarea3"/>
```

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

- It provides one property `addressprop`.
- It contains the macro XML (between `<taginstance>` and `</taginstance>`) for building the control out of existing 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.

Whenever possible we recommend to use the Control Editor to create and edit the `editor_*.xml` files.

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_*.xml` - to define the controls and how they fit into existing controls.
- `IMacroTagHandler` - implementations that transfer XML control definitions into other XML control definitions. Remember: Implementing your own macro tag handler is optional. If a control does not have its own macro tag handler, a general macro tag handler is used internally.
- `ITagHandler` - implementations that transfer XML control definitions into HTML/JavaScript.
- `NDOCustomControlInfoBase` - base class to implement complex data binding or control functionality for execution at runtime.

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

4 Composing New Controls Out of Existing Controls

■ Concept	22
■ Programmed Macro Control - Example	22
■ Configured Macro Control - Example	28

Concept

The concept is quite simple: you provide a macro control that tells how a given XML tag is transferred into an XML string representing the internally used Application DesignerApplication ComposerNatural for Ajax controls.

There are two ways to provide a macro control:

- Either you program a control on your own,
- or you configure the control using an XML definition.

The first way is the most flexible way - you create a piece of code translating the macro XML tag into other controls' XML tags. The second way is the easier way by which you can use a certain XML definition to define macro controls without having to code at all.

Programmed Macro Control - Example

Let us have a look at the following page:

The screenshot shows a web application window titled "Demo: Composition of controls". At the top right of the window is an "Exit" button. Below the title bar, there are two identical "Address" form panels stacked vertically. Each panel has a title bar with the word "Address" and a dropdown arrow. Inside each panel, there are four input fields: "First Name", "Last Name", "Street", and "Town". The "Town" field is split into two parts. Below the input fields in each panel is a "Clear" button.

This page contains two address areas. Now let us look at the corresponding XML layout definition:

```
<page model="com.softwareag.cis.test14.ControlLibraryControlCompositionAdapter">
  <titlebar name="Demo: Composition of controls">
  </titlebar>
  <header withdistance="false">
    <button name="Exit" method="endProcess">
    </button>
  </header>
  <pagebody>
    <demo:addressrowarea2 addressprop="address">
    </demo:addressrowarea2>
    <demo:addressrowarea2 addressprop="addressWife">
    </demo:addressrowarea2>
  </pagebody>
  <statusbar withdistance="false">
  </statusbar>
</page>
```

You see that there is a control `demo:addressrowarea2` which is used two times - once per address area. The control is responsible for arranging all its inner controls. Each tag has an `addressprop` property - we will see later how this property is treated.

Control Handler Code

Let us have a look at the corresponding control handler code:

```
package com.softwareag.cis.demolibrary;

import org.xml.sax.AttributeList;

import com.softwareag.cis.gui.generate.IMacroTagHandler;
import com.softwareag.cis.gui.protocol.Message;
import com.softwareag.cis.gui.protocol.ProtocolItem;

public class ADDRESSROWAREA2Handler
    implements IMacroTagHandler
{
    // -----
    // members
    // -----

    String m_addressprop;

    // -----
    // public usage
    // -----

    /** */
    public void generateXMLForStartTag(String tagName,
```

```

        AttributeList attrlist,
        StringBuffer sb,
        ProtocolItem pi)
{
    readAttributes(attrlist);
    fillProtocol(pi);
    // build XML that consists out of contained controls
    sb.append("<rowarea name='Address'>");
    sb.append("    <itr>");
    sb.append("        <label name='First Name' width='100'/>");
    sb.append("        <field valueprop='"+m_addressprop+".firstName' width='150'/>");
    sb.append("    </itr>");
    sb.append("    <itr>");
    sb.append("        <label name='Last Name' width='100'/>");
    sb.append("        <field valueprop='"+m_addressprop+".lastName' width='150'/>");
    sb.append("    </itr>");
    sb.append("    <vdist height='10'/>");
    sb.append("    <itr>");
    sb.append("        <label name='Street' width='100'/>");
    sb.append("        <field valueprop='"+m_addressprop+".street' width='300'/>");
    sb.append("    </itr>");
    sb.append("    <itr>");
    sb.append("        <label name='Town' width='100'/>");
    sb.append("        <field valueprop='"+m_addressprop+".zipCode' width='50'/>");
    sb.append("        <hdist width='5'/>");
    sb.append("        <field valueprop='"+m_addressprop+".town' width='245'/>");
    sb.append("    </itr>");
    sb.append("    <vdist height='10'/>");
    sb.append("    <itr>");
    sb.append("        <hdist width='100'/>");
    sb.append("        <button name='Clear' ↵
method='"+m_addressprop+".clearAddress'/>");
    sb.append("    </itr>");
    sb.append("</rowarea>");
}

/** */
public void generateXMLForEndTag(String tagName,
                                StringBuffer sb)
{
}

// -----
// private usage
// -----

/** */
private void readAttributes(AttributeList attrlist)
{
    for (int i=0; i<attrlist.getLength(); i++)
    {
        if (attrlist.getName(i).equals("addressprop"))

```

```

        m_addressprop = attrlist.getValue(i);
    }
}

/** */
private void fillProtocol(ProtocolItem pi)
{
    // check
    if (m_addressprop == null)
        pi.addMessage(new Message(Message.TYPE_ERROR, "Attribute ADDRESSPROP is ←
not set"));
    // properties
    pi.addProperty(m_addressprop, "ADDRESSInfo");
    // no further properties to be proposed in code assistant
    pi.suppressFurtherCodegenEntries();
}
}

```

Let us have a look at the building blocks of the code:

- The class has a member `m_addressprop`. This member is filled directly at the beginning of the `generateHTMLForStartTag` method - inside the `readAttributes()` method. The attribute list is walked through and checked for the attribute `addressprop`.
- As the next step, the protocol item is filled. On the one hand, you can put there any information with a certain severity attribute - in the example, an error message is written to protocol if no attribute `addressprop` is defined. On the other hand, you have to tell the protocol item which properties you are accessing from your control.

Pay attention that the properties which are accessed inside the access control are a composition out of the `m_address` value and some fix names which are defined by the control.

- In the processing of the method `generateXMLForStartTag()`, all the XML is created that per control instance creates the container representing an address area.

Control Adapter Code

The control adapter is not required to be written for a control - it is just an option which is extremely useful for structuring you server side code.

In principle, the control definition says that it refers to an address property (`ADDRESSPROP` value). The inner controls take their information out of sub-properties of this control. For example, if the `ADDRESSPROP` value is defined to be "wifeAddress", then the fields and buttons are bound to:

- `wifeAddress.firstName`
- `wifeAddress.lastName`
- `wifeAddress.street`

- wifeAddress.zipCode
- wifeAddress.town
- wifeAddress.clearAddress (method)

You now can provide for a server side control adapter class which provides for all this data. For example, the implementation is:

```
package com.softwareag.cis.demolibrary;

import com.softwareag.cis.server.*;

/**
 * This is the logic-class behind the control ADDRESSROWAREA.
 */
public class ADDRESSInfo
{
    // -----
    // members
    // -----

    String m_firstName;
    String m_lastName;
    String m_street;
    String m_zipCode;
    String m_town;

    // -----
    // public access
    // -----

    public String getFirstName() { return m_firstName; }
    public String getLastName() { return m_lastName; }

    public String getStreet() { return m_street; }
    public String getTown() { return m_town; }

    public String getZipCode() { return m_zipCode; }
    public void setFirstName(String firstName) { m_firstName = firstName; }

    public void setLastName(String lastName) { m_lastName = lastName; }
    public void setStreet(String street) { m_street = street; }

    public void setTown(String town) { m_town = town; }
    public void setZipCode(String zipCode) { m_zipCode = zipCode; }

    public void clearAddress()
    {
        m_firstName = null;
        m_lastName = null;
        m_street = null;
    }
}
```

```

        m_zipCode = null;
        m_town = null;
    }
}

```

A page adapter class can now use this control adapter class and can automatically take over all its contained properties and methods. For example, the page adapter of the page of this example might look like:

```

package com.softwareag.cis.test14;

import com.softwareag.cis.demolibrary.*;
import com.softwareag.cis.server.Model;

public class ControlLibraryControlCompositionAdapter
    extends Model
{
    // -----
    // members
    // -----

    ADDRESSInfo m_address = new ADDRESSInfo();
    ADDRESSInfo m_addressWife = new ADDRESSInfo();

    // -----
    // public access
    // -----

    public ADDRESSInfo getAddress() { return m_address; }
    public ADDRESSInfo getAddressWife() { return m_addressWife; }
}

```

The page adapter just creates two instances of the control adapter `ADDRESSInfo` and publishes them as property `address` and property `wifeAddress`.

Be aware that you can use all Java possibilities on the server side to let the control adapter interact with your page adapter. Maybe you would like to be informed inside the page adapter every time the `clear()` method is invoked? Then just build some eventing functions into the control adapter - and the page adapter can register as event listener to its contained control adapter.

Configured Macro Control - Example

In the previous example, an explicit control handler class was written in order to transfer a short XML statement into a long one. For simple control arrangements without any sophisticated logic, you can do the same by just configuring the control - instead of programming it.

Let us now do the same as done with code in the previous section - this time without coding.

The configuration is done using an editor extension file (e.g. *editor_demo.xml* in the *cis/config* directory). When generating HTML pages, Application DesignerApplication ComposerNatural for Ajax looks into its configuration directory and searches for all *.xml* files starting with "editor_". Each of the files contains configuration information about controls and their usage.

Have a look at the *editor_demo.xml* file and you will see 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"/>
        <field valueprop="$addressprop$.firstName" width="150"/>
      </itr>
      <itr>
        <label name="Last Name" width="100"/>
        <field valueprop="$addressprop$.lastName" width="150"/>
      </itr>
      <vdist height="10"/>
      <itr>
        <label name="Street" width="100"/>
        <field valueprop="$addressprop$.street" width="300"/>
      </itr>
      <itr>
        <label name="Town" width="100"/>
        <field valueprop="$addressprop$.zipCode" width="50"/>
        <hdist width="5"/>
        <field valueprop="$addressprop$.town" width="245"/>
      </itr>
      <vdist height="10"/>
      <itr>
        <hdist width="100"/>
        <button name="Clear" method="$addressprop$.clearAddress"/>
      </itr>
    </rowarea>
  </taginstance>
  <protocolitem>
    <addproperty name="$addressprop$" datatype="ADDRESSInfo" ↵
```

```

useincodegenerator="true"/>
  </protocolitem>
</tag>
<tagsubnodeextension control="pagebody" newsubnode="demo:addressrowarea3"/>

```

The following is defined in this section:

- The new tag ADDRESSROWAREA3 is defined.
- A property addressprop is defined to exist.
- The XML macro is contained that is used for transferring the control into XML. Inside the XML you see that the value of addressprop is referred to by using "\$addressprop\$".
- The new tag is defined to be reachable below the PAGEBODY tag.

The result at the end is the same as produced with the ADDRESSROWAREA2 control of the previous section.

The XML configuration can be either done manually within the XML file or by using the Control Editor.

editor_* File Concept

In the example above, the macro XML definition was part of a file *editor_demo.xml*. If you have a look at the */cis/config* directory of your web application, then you will see some files:

- *editor.xml*
- *editor_demo.xml*
- editor_report.xml*
- editor_pivot.xml*
- and other *editor_** files
- *editorextensions_template.xml*

Each file contains information about controls. When gathering the available controls, Application DesignerApplication ComposerNatural for Ajax reads all *editor_*.xml* files and builds one “big internal” control model.

editor_.xml* files are also mentioned later. Since they hold information on how to arrange controls, they are also used as control files for Application DesignerApplication ComposerNatural for Ajax's Layout Painter. For more details, see the section [Bringing Controls into the Layout Painter](#).



Note: If you are using an old servlet engine of version 2.2 (e.g. Tomcat 3, Websphere 4), there is one additional file to be maintained: *editorextensions.xml*. For detailed information, see the *editorextensions_template.xml* file.

5

Creating Macro Controls Out of Existing Controls

■ General Information	32
■ Creating Macro Controls	33

General Information

There are two types of controls that you can create with Natural for Ajax:

■ Graphical Controls

A graphical control transforms an element of an XML layout definition into HTML/JavaScript code. You can either create completely new controls by writing your own HTML/JavaScript, or you can reuse existing controls and compose the HTML/JavaScript of these controls to new controls. The latter is called “macro control”. The recommendation is to use macro controls if possible and to write your own HTML/JavaScript only if needed.

Example: You can define an address area that comprises several existing controls (such as ITR, LABEL and FIELD). You call the address area "NADC:ADDRESS" where "NADC" is the library prefix. The control is a kind of macro that expands a short XML layout definition, which just contains the NADC:ADDRESS control tag, into a more complex layout definition containing all the single control tags (such as ITR, LABEL and FIELD).

■ Non-visual Controls

A non-visual control adds some data binding to the layout. It allows your Natural program to exchange data with the Natural for Ajax framework and the browser client. The control can decide whether the data is available in the browser or only available in the Natural for Ajax framework of your web application. Non-visual custom controls are usually defined as macro controls from existing non-visual controls.

Examples: You would like to define a specific data structure. This structure should be the same for multiple layouts of your application. To do so, you would define a macro control composed of several XCIDATADEF controls. Or you would like multiple of your layouts to exchange context data. To do so, you could define a macro control composed of XCICONTTEXT controls.

The following two aspects of macro controls are important:

■ Layout Aspect

From the layout aspect, macro controls help to be flexible regarding design changes. Macro controls make sure that a certain graphical arrangement of existing controls is not applied to various page layouts by using copy-and-paste, but by using a proper control definition. When changing the control definition and re-generating the layout definitions that use the control, all changes in the control are automatically propagated to the page layouts.

■ Server-side Aspect

From the server-side aspect, a macro control may have pre-designed server-side Natural data fields and events that can be associated with it. For example, an address control may trigger an event on the Natural server to check the validity of a zip code that the user has specified.

Creating Macro Controls

Creating a macro control consists of the following steps:

- [Defining a New Control Library](#)
- [Defining the Control Attributes and Control Hierarchy \(Subtags, Container\)](#)
- [Defining/Implementing the Rendering of the Control](#)
- [Implementing the Control's Server-Side Processing \(Optional\)](#)
- [Putting Things to Work](#)

The topics below describe a sample control which is used to specify an address.

We recommend that you use all-lowercase letters for prefixes, control names and attributed names. A good example for this is:

nadc:zipcodecity

A bad example would be the following:

NaDc:zipCodeCity

Defining a New Control Library

Each control that is not supplied by Software AG requires a prefix that ensures that controls supplied by different providers can be used within one page. In our example, we use the prefix "nadc" for "Natural for Ajax Demo Controls".

The setup is done in the file `<project>/cisconfig/controllibraries.xml`.

An example for registering the nadc library would be:

```
<library prefix="nadc"
  package="mycontrols.nadc">
</library>
```

The package that is included in the definition is the Java package that contains corresponding tag handlers (optional).



Note: In order to activate new control libraries, you must restart the Tomcat server. For NaturalONE, this means restarting Eclipse.

Defining the Control Attributes and Control Hierarchy (Subtags, Container)

For our `nadc` control library, we create the file `editor_nadc.xml`. This file can be created using the Control Editor.

In the Control Editor, set up the file `editor_nadc.xml`. One file can contain a number of controls. For each control, you specify the following:

The name of the control

In our example, this is `"nadc:address"`. Be sure to add the prefix when entering the name.

The control's attributes

This is the list of attributes that a user of the control must specify when using the control inside a page. In our example, the control `"nadc:address"` has the following attribute:

addressprop

A reference to the runtime property implementation.

Positioning definitions

These specify where the control can be added inside a layout definition. They are used by the layout editor, which only allows controls to be placed in the specified positions.

The positioning definitions include:

Name of the section in the controls palette

The layout editor arranges all controls within the controls palette. This palette is structured into sections. If the name of a section does not yet exist, a new section is created automatically. In our example, the name of the section is `"NJXDemos"`.

Embedding containers

A list of all controls which allow the new control to be positioned inside it. In our example, we decide to position the controls below `"pagebody"`, `"rowarea"`, `"colarea"` and `"splitcell"`.

After maintaining the information in the Control Editor, the content of the file `editor_nadc.xml` is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<controllibrary>
  <editor>
    <tag name="nadc:address">
      <attribute name="addressprop" mandatory="true"/>
      <taginstance>
      </taginstance>
      <protocolitem>
      </protocolitem>
    </tag>
    <tagsubnodeextension control="pagebody" newsubnode="nadc:address"/>
    <tagsubnodeextension control="rowarea" newsubnode="nadc:address"/>
    <tagsubnodeextension control="colarea" newsubnode="nadc:address"/>
    <tagsubnodeextension control="splitcell" newsubnode="nadc:address"/>
    <taggroupsubnodeextension group="NJXDemos" newsubnode="nadc:address"/>
  </editor>
</controllibrary>
```

```
</editor>
</controllibrary>
```

Defining/Implementing the Rendering of the Control

You can define the rendering either in a descriptive way (XML) in the *editor_nadc.xml* file or you can define it by implementing a tag handler. The Natural for Ajax demos contain examples for both options. Here, we will describe how to implement the rendering using a tag handler. The tag handler is a Java class which defines how the control's "short XML" (for example, `<nadc:address addressprop='person' />`) is transformed into an XML layout definition which itself contains standard controls or own controls. The tag handler class must extend the interface `IMacroTagHandler`. For details, see the corresponding Java API documentation. The tag handler has to take care of two main issues:

- defining the rendering of the control;
- defining data bindings (advanced usage).

First, we discuss the sample "nadc:address" control. The tag handler is implemented in the package `com.softwareag.cis.test.customcontrols`; this is the package that was defined with the control library prefix "nadc" in the configuration file *controllibraries.xml*. The name of the tag handler class follows the convention `<controlInUpperCase>Handler`, in our case "ADDRESSHandler".

```
package com.softwareag.cis.test.customcontrols;

import org.xml.sax.AttributeList;

import com.softwareag.cis.gui.generate.IMacroTagHandler;
import com.softwareag.cis.gui.generate.IXSDGenerationHandler;
import com.softwareag.cis.gui.protocol.Message;
import com.softwareag.cis.gui.protocol.ProtocolItem;

public class ADDRESSHandler
    implements IMacroTagHandler
{

    public void generateXMLForStartTag(String tagName,
                                      AttributeList attributes,
                                      StringBuffer xml,
                                      ProtocolItem protocolItem)
    {
        // read attributes
        String ap = attributes.getValue("addressprop");
        // rendering
        xml.append
        (
            "<rowarea name='Address'>" +
            "<itr>" +
            "    <label width='120' name='First/ Last Name' />" +
            "    <field valueprop='"+ap+"'.firstName' width='150' />" +
```

```

        "<hdist width='5'/>" +
        "<field valueprop='"+ap+".lastName' width='150'/>" +
        "</itr>" +
        "<itr>" +
        "<label width='120' name='Street'/>" +
        "<field valueprop='"+ap+".street' width='305'/>" +
        "</itr>" +
        "<itr>" +
        "<label width='120' name='Zip Code/ City'/>" +
        "<field valueprop='"+ap+".zipCode' width='80'/>" +
        "<hdist width='5'/>" +
        "<field valueprop='"+ap+".city' width='220'/>" +
        "<hdist width='10'/>" +
        "<button name='Check' method='"+ap+".onCheck'/>" +
        "</itr>" +
        "</rowarea>"
    );
    IXSDGenerationHandler xga = protocolItem.findXSDGenerationHandler();
    xga.addControlInfoClass(protocolItem, ap, ADDRESSInfo.class);
}

public void generateXMLForEndTag(String arg0, StringBuffer arg1)
{
}
}

```

Here are the major processing steps of the tag handler:

- The attribute `addressprop` is read from the control definition and stored in the variable `ap`.
- The XML layout definition is appended by adding controls such as `ROWAREA` and `FIELD`. Note that inside the rendering definition, property and method references are prefixed with `"ap"` and `"."` (period).
- The `ADDRESS` control implements some advanced binding (optional). As seen later, some part of the control functionality is implemented in a corresponding binding class. This binding class `ADDRESSInfo` is registered as the server-side counterpart of the control at an `IXSDGenerationHandler` interface.

Implementing the Control's Server-Side Processing (Optional)

You can apply binding objects to controls. The association of a control to a binding object is specified in the control's tag handler via the method `addControlInfoClass` of the interface `IXSDGenerationHandler`.

The server binding objects are optional. They may encapsulate functions with the control which are executed for the control on the server side. Example: In our address control, a zip code validity check will be added. This check is automatically executed within the control when the user presses the **Check** button that is part of the control's rendering.

This server-side processing for a control is defined in a class that extends the existing class `NDOCustomControlInfoBase`. See the following code:

```
package com.softwareag.cis.test.customcontrols;

import com.softwareag.cis.adapter.ndo.NDOCustomControlInfoBase;
import commonj.sdo.DataObject;

public class ADDRESSInfo
    extends NDOCustomControlInfoBase
{
    public void onCheck()
    {
        // first execute the control's logic
        DataObject address = getDataObject();
        String zipCode = address.getString("zipCode");
        String city = address.getString("city");
        if ("64297".equals(zipCode) &&
            (city == null || city.length() == 0))
        {
            address.set("city", "Darmstadt");
        }
        // delegate the method to the normal event processing
        super.invokeMethod("onCheck");
    }
}
```

We recommend the following guidelines:

- The class's package should be the same as with the handler class.
- The name of the class is `<controlInUpperCase>Info`.

The class extends the class `NDOCustomControlInfoBase` which is supplied with Natural for Ajax. The base class provides some useful functions:

- It provides the properties for the contained content (that is, "firstName", "lastName", "street", etc.).

- It provides a binding to the SDO to which the control refers. In the address example, the control binds to an "addressprop", all detail properties are defined to be "<valueOfAddressProp>.firstName", "<valueOfAddressProp>.lastName", etc. The method `getDataObject()` returns the SDO object that represents the control.
- It provides a function to map methods to events in the Natural adapter code. If you control a specific execution for a method inside the control (in the example, this is the `onCheck()` method), you can delegate the event to the normal Natural adapter event handling after having handled the control-specific matters.

Putting Things to Work

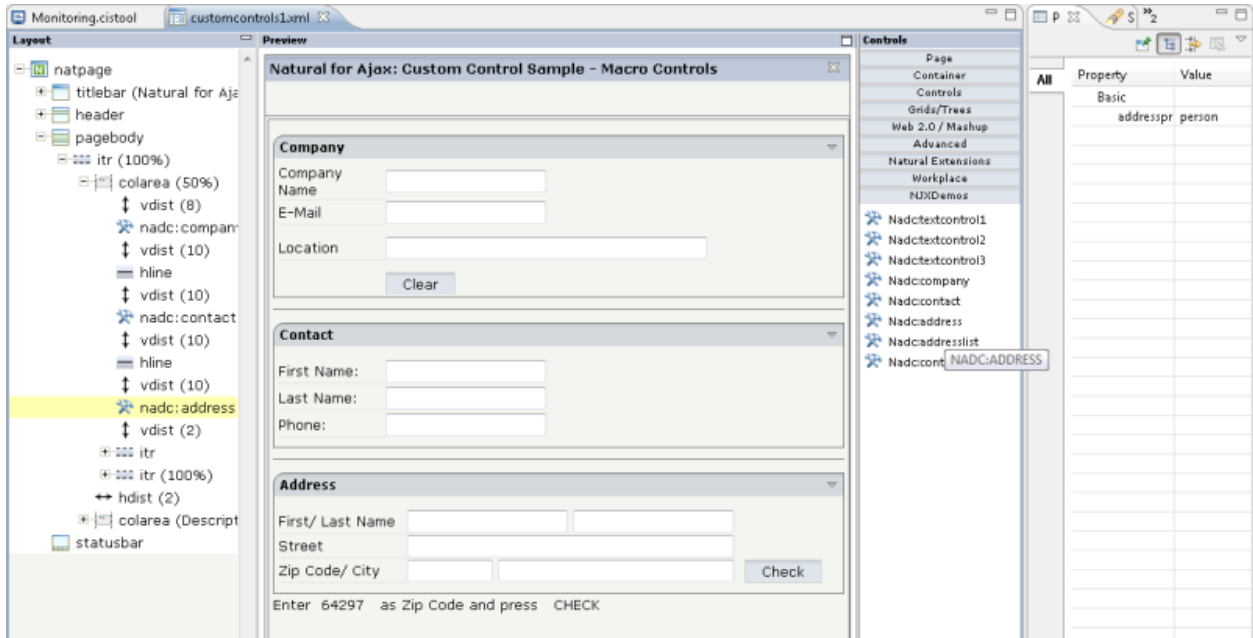
Now we show how to use the control within a page.

When you open the Layout Painter, you can see a new section in the controls palette which has the name "NJXDemos". In this section, you can find the control "nadc:address".

If you define a layout as follows:

```
<natpage>
...
  <pagebody>
    ...
    <nadc:address addressprop="person">
    </nadc:address>
    ...
  </pagebody>
  ...
</natpage>
```

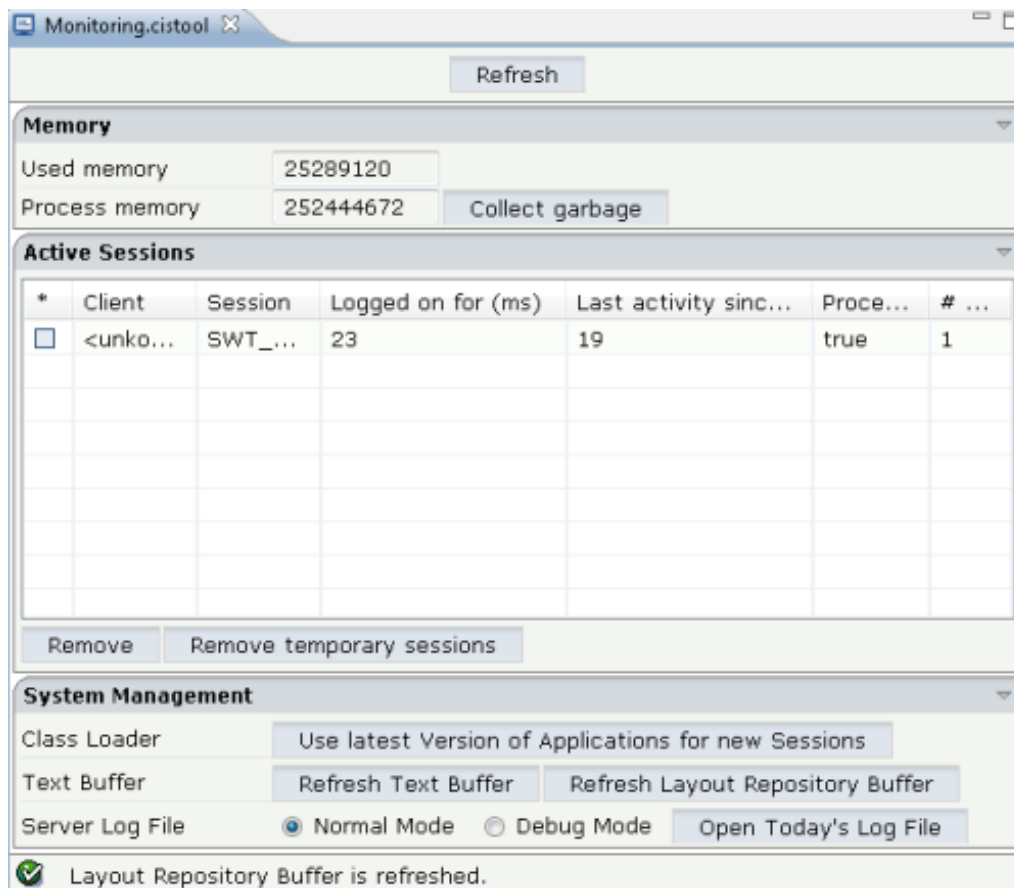
the corresponding page looks as shown below:



When you run a corresponding Natural program with this layout and you then enter "64297" in the field defined by `zipCode` and choose the **Check** button, the name of the corresponding city is "calculated" by the control processing. Note that the event `person.onCheck` is then delegated to the normal Natural adapter event processing. This means that this method can be implemented just as a normal event in the Natural program.



Important: To activate/refresh new/changed controls in existing control libraries, you must choose the **Use latest Version for Applications in new Session, Refresh Text Buffer** and **Refresh Layout Repository Buffer** buttons in the monitoring tool.



6

Creating New Controls

■ Concept	42
■ Example 1	42
■ Njxdemos Sample Control: NADC:TEXTCONTROL1	44
■ JavaScript Functions	45
■ Example 2	47
■ Njxdemos Sample Control: NADC:TEXTCONTROL3	50
■ Example 3 (Applet)	51
■ Summary	56

Concept

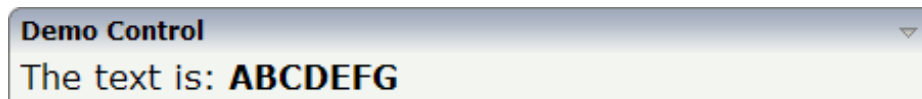
In the previous section, you learned how to compose complex controls create macro controls out of existing controls. You will now learn how to build completely new controls which are not yet part of the Application Designer control set.

The concept of building your own controls is to insert corresponding HTML and JavaScript instructions into the HTML page which is the result of the generation process.

A JavaScript function library is available which can be directly accessed inside the HTML code which is generated. This library contains useful methods for accessing properties and executing methods of the "model" ("net data") behind the page as well as triggering event execution and/or taking part in the execution of events.

Example 1

The first example is a quite simple one: a tag with the name "democontrol" is introduced, which does nothing else than writing a text which is passed via a tag attribute into the generated HTML page:



The corresponding XML layout definition looks as follows:

```
<rowarea name="Demo Control">
  <itr>
    <demo:democontrol text="ABCDEFG">
    </demo:democontrol>
  </itr>
</rowarea>
```

You see that the text which is passed inside the `text` attribute of the `demo:democontrol` tag is displayed inside the control in bold letters.

The Java code of the tag handler of the `demo:democontrol` tag looks as follows:

```

package com.softwareag.cis.demolibrary;

import org.xml.sax.AttributeList;

import com.softwareag.cis.gui.generate.*;
import com.softwareag.cis.gui.protocol.*;

public class DEMOCONTROLHandler implements ITagHandler
{
    // -----
    // members
    // -----

    String m_text;

    // -----
    // public methods
    // -----

    /**
     */
    public void generateHTMLForStartTag(
        int id,
        String tagName,
        AttributeList attrlist,
        ITagHandler[] handlersAbove,
        StringBuffer sb,
        ProtocolItem protocolItem)
    {
        readAttributes(attrlist);
        fillProtocolItem(protocolItem);
        sb.append("\n<!-- DEMOCONTROL begin -->\n");
        sb.append("<td>The text is: <b>" + m_text + "</b></td>\n");
    }

    /**
     */
    public void generateHTMLForEndTag(String tagName, StringBuffer sb)
    {
        sb.append("\n<!-- DEMOCONTROL end -->\n");
    }

    /**
     */
    public void generateJavaScriptForInit(
        int id,
        String tagName,
        StringBuffer sb)
    {
    }

    // -----

```

```
// private methods
// -----

/**
 *
 */
public void readAttributes(AttributeList attrlist)
{
    for (int i=0; i<attrlist.getLength(); i++)
    {
        if (attrlist.getName(i).equals("text"))
            m_text = attrlist.getValue(i);
    }
}

/**
 *
 */
public void fillProtocolItem(ProtocolItem pi)
{
    if (m_text == null)
        pi.addMessage(new Message(Message.TYPE_ERROR,
                                   "Attribute TEXT is not defined"));
}
}
```

In the tag handler, the following steps are processed:

- In the `generateHTMLForStartTag()` method, the attributes which are defined with the tag are read first and the protocol is filled.
- Then, plain HTML information is appended to the HTML string which is passed as a parameter (`StringBuffer sb`). Inside the HTML information, the value of the `text` attribute is dynamically inserted.

This control does not provide for any interactivity; it just writes out a certain value which is defined inside its tag definition.

Njxdemos Sample Control: NADC:TEXTCONTROL1

For the sample layout, see the file *customcontrols3.xml* in the *njxdemos/xml* folder.

NADC:TEXTCONTROL1 is a quite simple example of a new control: It does nothing else than writing text which is passed via a static tag attribute into the generated HTML page:

Pure JavaScript/HTML...

The text is: **Some Static TEXT**

The corresponding XML layout definition looks as follows:

```
<rowarea name="Pure JavaScript/HTML...">
  <vdist height="10"></vdist>
  <itr>
    <nadc:textcontrol1 text="Some Static TEXT">
    </nadc:textcontrol1>
  </itr>
</rowarea>
```

You see that the text which is passed inside the text attribute of the NADC:TEXTCONTROL1 tag is displayed inside the control in bold letters.

For details on the corresponding tag handler, see the description and the source code for the CUSTC3-P.NSP example in the Natural for Ajax demos.

JavaScript Functions

For more interactive controls - for example, which use certain data coming from the server-side adapter - you need to access certain JavaScript functions which are available inside the client. The generated HTML page contains an object named "csciframe". This object provides a certain set of functions for usage from within custom controls.

It is not possible in JavaScript to arrange a set of published functions in some kind of interface in order to only allow users a dedicated access. Therefore, the functions which are allowed to access are listed in this section. You must not use any other functions of the Ajax framework - even if you may see additional functions in the JavaScript sources. Only the following functions belong to the public Java Script library:

Function	Description
setPropertyValue(pn,pv)	Sets a property value inside the adapter. The value is not directly sent to the server but is buffered first in the client. If there is a synchronization event, then the buffer is transferred. pn = name of property pv = value Examples:

Function	Description
	<pre>csciframe.setPropertyValue(companyName,"Software AG"); csciframe.setPropertyValue(address.firstName,"John"); csciframe.setPropertyValue(addresses[2].firstName,"Maria");</pre>
getPropertyValue(pn)	Reads a property value from the adapter (better: the client representation of the adapter). pn = name of property result = string of property value Examples: <pre>var vResult1 = csciframe.getPropertyValue("company"); var vResult2 = csciframe.getPropertyValue("addresses[2].firstName");</pre> Pay attention: the adapter value is always passed back as a string. A boolean value, as a consequence, is returned as "true" string and not as "true" boolean value. Null values of the adapter, that is, where the Java adapter class on the server side passes back "null", are returned as an empty string (""). A JavaScript null value is passed back if the property for which you ask does not exist.
registerListener(me)	Passes a method pointer (me value). The method is called every time when a response of a client request is processed. In other words: every time new data comes from the server or if the model is updated in another way (for example, by flush signals of other controls), then the corresponding methods are called. In the method, you can place a corresponding reaction of your control on new data. The method which you pass must have a parameter <code>model</code> - which is not used anymore, but which has to be defined. Example: <pre>... ... function reactOnNewData(model) { var vResult = csciframe.getPropertyValue("firstName"); alert(vResult); }</pre>

Function	Description
	<pre>csciframe.registerListener(reactOnNewData);</pre>
invokeMethodInModel(mn)	Invokes the calling of a method inside the adapter. As a consequence, the data changes which may have been buffered inside the client are flushed to the server and the method is called. mn = name of adapter method Example: <pre>csciframe.invokeMethodInModel("onSave");</pre>
submitModel(n)	Synchronizes the client with the server. Analogous to the <code>invokeMethodInModel()</code> method from the synchronization point of view - but now without calling an explicit method in the adapter. n = name, must be submit Example: <pre>csciframe.submitModel("submit");</pre>

Example 2

The following example is an extension of the previous example. Whereas in [Example 1](#) the text which is output by the control was defined as an attribute of the tag definition, the text is now dynamically derived from an adapter property.



The XML layout definition is:

```
<rowarea name="Demo Control">
  <itr>
    <label name="Text" width="100">
    </label>
    <field valueprop="text" width="200" flush="screen">
    </field>
  </itr>
  <vdist height="20">
```

```

    </vdist>
    <itr>
        <demo:democontroldyn textprop="text">
        </demo:democontroldyn>
    </itr>
</rowarea>

```

You see that the DEMOCONTROLDYN control references the same adapter property `text` as the FIELD control.

Let us have a look at the tag handler class for the DEMOCONTROLDYN control:

```

package com.softwareag.cis.demolibrary;

import org.xml.sax.AttributeList;

import com.softwareag.cis.gui.generate.*;
import com.softwareag.cis.gui.protocol.*;

public class DEMOCONTROLDYNHandler implements ITagHandler
{
    // -----
    // members
    // -----

    String m_textprop;

    // -----
    // public methods
    // -----

    /**
     */
    public void generateHTMLForStartTag(
        int id,
        String tagName,
        AttributeList attrlist,
        ITagHandler[] handlersAbove,
        StringBuffer sb,
        ProtocolItem protocolItem)
    {
        readAttributes(attrlist);
        fillProtocolItem(protocolItem);
        sb.append("\n<!-- DEMOCONTROL begin -->\n");
        sb.append("<td>The text is: <b>");
        sb.append("<span id='DEMOSPAN'+id+' '></span>");
        sb.append("</b></td>\n");
        sb.append("<script>\n");
        sb.append("function reactOnModelUpdate"+id+"(model)\n");
        sb.append("{\n");
        sb.append("    var vText = csciframe.getPropertyValue('"+m_textprop+"');\n");
    }
}

```

```

        sb.append("    var vSpan = document.getElementById('DEMOSPAN'+id+'');\n");
        sb.append("    vSpan.innerHTML = vText;\n");
        sb.append("}\n");
        sb.append("</script>\n");
    }

    /**
     */
    public void generateHTMLForEndTag(String tagName, StringBuffer sb)
    {
        sb.append("\n<!-- DEMOCONTROL end -->\n");
    }

    /**
     */
    public void generateJavaScriptForInit(
        int id,
        String tagName,
        StringBuffer sb)
    {
        sb.append("csciframe.registerListener(reactOnModelUpdate"+id+"");\n");
    }

    // -----
    // private methods
    // -----

    /**
     */
    public void readAttributes(AttributeList attrlist)
    {
        for (int i=0; i<attrlist.getLength(); i++)
        {
            if (attrlist.getName(i).equals("textprop"))
                m_textprop = attrlist.getValue(i);
        }
    }

    /**
     */
    public void fillProtocolItem(ProtocolItem pi)
    {
        // Messages
        if (m_textprop == null)
            pi.addMessage(new Message(Message.TYPE_ERROR,
                                     "Attribute TEXTPROP is not defined"));

        // Property Usage
        pi.addProperty(m_textprop, "String");
    }
}

```

You see:

- Inside the `generateJavaScript()` method, a JavaScript function is added as a listener to adapter model changes. The function is generated inside the `generateHTMLForStartTag()` method.
- The name of the property is read via the attribute list into the member `m_textprop` - and is dynamically used when calling the JavaScript function `getPropertyValue()`.
- All JavaScript names (e.g. method names, IDs of controls) which are “global” inside the HTML page are suffixed with the control ID which is passed via the `ITagHandler` methods. The reason: if one control is defined multiple times inside a page, then the different methods and IDs are separated by this ID.
- The protocol item is filled with the information about the property which is required by the control. This is necessary at runtime because the Application DesignerNatural for Ajax runtime environment needs to find out which data of an adapter property to send back to the client (see *Binding between Page and Adapter*).

Njxdemos Sample Control: NADC:TEXTCONTROL3

The following example is an extension of the above NADC:TEXTCONTROL1 example. Whereas in the NADC:TEXTCONTROL1 control, the text to be output by the control was defined at design time as a static attribute of the tag definition, the text is now dynamically derived from an adapter property. The adapter sets the final text value during a server roundtrip.



With own Data Binding...

Your name

The text is: **Hello Custom Control User**

The corresponding XML layout definition looks as follows:

```
<rowarea name="With own Data Binding...">
  <vdist height="10"></vdist>
  <itr>
    <label name="Your name" width="100">
      </label>
    <field valueprop="yourname" width="200" flush="server" ↵
flushmethod="onNameChanged">
      </field>
    </itr>
  <vdist height="10"></vdist>
  <itr>
    <nadc:textcontrol3 textprop="hellotext" >
      </nadc:textcontrol3>
```

```

</itr>
<vdist height="10"></vdist>
</rowarea>

```

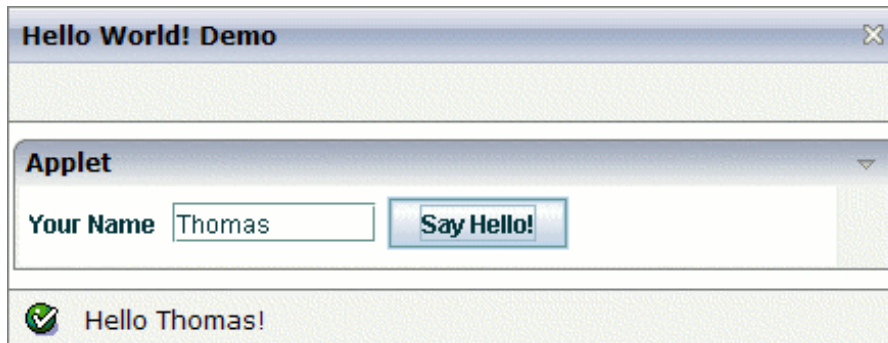
The NADC:TEXTCONTROL3 generates an own adapter property. This results in the generation of a data field in the Natural adapter.

For details on the corresponding tag handler, see the description and the source code for the CUSTC3-P.NSP example in the Natural for Ajax demos.

Example 3 (Applet)

This example shows how to embed Java applets. Unlike [Example 1](#) and [Example 2](#), the generated HTML/JavaScript here just makes the applet aware of the normal data communication between screen and page adapter. The rendering is done in Java.

This example uses a very basic “Say Hello!” applet. It shows one field input and a button. On button click, a message is displayed on the STATUSBAR control.



The XML layout definition is:

```

<page model="SayHelloAdapter">
  <titlebar name="Say Hello! Demo">
  </titlebar>
  <header>
  </header>
  <pagebody>
    <rowarea name="Applet">
      <tr>
        <demo:applet code="SayHelloApplet.class"
                     width="400"
                     height="40"
                     valueprop="yourName"
                     method="onSayHello">
        </demo:applet>
      </tr>
    </rowarea>
  </pagebody>
</page>

```

```
        </rowarea>
    </pagebody>
    <statusbar>
    </statusbar>
</page>
```

The `demo:applet` tag shows the usual applet attributes: `code`, `width` and `height`. With the attribute `valueprop`, the applet's field input is bound to the adapter property `yourName`. The attribute `method` binds the button to adapter method `onSayHello`.

This is the page adapter for this example:

```
import com.softwareag.cis.server.Adapter;

public class SayHelloAdapter extends Adapter
{
    // property >yourName<
    String m_yourName;
    public String getYourName() { return m_yourName; }
    public void setYourName(String value) { m_yourName = value; }

    /** called on button click */
    public void onSayHello()
    {
        outputMessage(MT_SUCCESS, "Hello "+m_yourName+"!");
    }
}
```

The adapter only provides for the property `yourName` and the method `onSayHello`.

The most important thing is the tag handler class. Let us have a look at `APPLETHandler`:

```
package com.softwareag.cis.demolibrary;

import org.xml.sax.*;

import com.softwareag.cis.file.CSVManager;
import com.softwareag.cis.gui.protocol.*;
import com.softwareag.cis.gui.util.*;

public class APPLETHandler implements ITagHandler
{
    // -----
    // members
    // -----

    String m_code;
    String m_codebase = ".";
    String m_width = "100";
    String m_height = "100";
    String m_valueprop;
```

```

String m_method;

// -----
// public usage
// -----

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

    sb.append("\n");
    sb.append("<!-- APPLETT begin -->\n");
    sb.append("<td>\n");
    sb.append("<applet name=\"APPLET"+id+"\" " +
              "codebase=\".\" " +
              "code=\""+m_code+"\" " +
              "width=\""+m_width+"\" " +
              "height=\""+m_height+"\" " +
              "MAYSCRIPT>\n");

    sb.append("    <param name=\"scriptable\" value=\"true\">\n");
    sb.append("    <param name=\"id\" value=\""+id+"\">\n");
    sb.append("    <param name=\"valueprop\" value=\""+m_valueprop+"\">\n");
    sb.append("    <param name=\"method\" value=\""+m_method+"\">\n");
    sb.append("</applet>\n");

    sb.append("<script>\n");
    sb.append("function romu"+id+"(model) {");
    sb.append("    try { document.APPLETT"+id+".reactOnNewData(); } \n");
    sb.append("    catch (exc) { alert('Error occurred when talking to applet!' + exc); } \n");
    sb.append("} \n");
    sb.append("function getPropertyValue"+id+"(propertyName) { return ↵
C.getPropertyValue(propertyName); } \n");
    sb.append("function setPropertyValue"+id+"(propertyName, value) { ↵
C.setPropertyValue(propertyName, value); } \n");
    sb.append("function invokeMethodInModel"+id+"(methodName) { ↵
C.invokeMethodInModel(methodName); } \n");
    sb.append("</script>\n");
}

public void generateHTMLForEndTag(String tagName,
                                  StringBuffer sb)
{
    sb.append("<!-- APPLETT end -->\n");
    sb.append("</td>\n");
}

```

```
public void generateJavaScriptForInit(int id,
                                     String tagName,
                                     StringBuffer sb)
{
    sb.append("C.registerListener(romu"+id+");\n");
}

// -----
// private usage
// -----

/** */
private void readAttributes(AttributeList attrlist)
{
    for (int i=0; i<attrlist.getLength(); i++)
    {
        if (attrlist.getName(i).equals("code")) m_code = attrlist.getValue(i);
        if (attrlist.getName(i).equals("codebase")) m_codebase = ↵
attrlist.getValue(i);
        if (attrlist.getName(i).equals("width")) m_width = attrlist.getValue(i);
        if (attrlist.getName(i).equals("height")) m_height = attrlist.getValue(i);
        if (attrlist.getName(i).equals("valueprop")) m_valueprop = ↵
attrlist.getValue(i);
        if (attrlist.getName(i).equals("method")) m_method = attrlist.getValue(i);
    }
}

/** */
private void fillProtocol(ProtocolItem pi)
{
    // check
    if (m_code == null) pi.addMessage(new Message(Message.TYPE_ERROR, "CODE not ↵
set"));
    if (m_valueprop == null) pi.addMessage(new Message(Message.TYPE_ERROR, ↵
"VALUEPROP not set"));
    if (m_method == null) pi.addMessage(new Message(Message.TYPE_ERROR, "METHOD ↵
not set"));
}
}
```

You see:

- `readAttributes()` reads all attributes from the XML. Their values are saved in member variables.
- `fillProtocol()` checks whether mandatory attributes are set. If not, an error message is shown within the generation protocol.
- `generateHTMLForStartTag()` generates HTML code containing the applet/parameter tags. Some JavaScript functions are added that are available inside the applet coding (`getPropertyValue/setPropertyValue/invokeMethodInModel`).

- `generateJavaScriptForInit()` registers function `romu()` as a listener to model changes. On change, the applet is called by `reactOnNewData`.

The Java applet looks as follows:

```
import netscape.javascript.JSObject;

public class SayHelloApplet extends Applet
{
    private String m_id;
    private String m_valueprop;
    private String m_method;
    private JLabel m_label;
    private JTextField m_fieldJ;
    private JButton m_buttonJ;
    private JSObject m_windowJ = null;

    // -----
    // Data binding
    // -----

    /**
     * Callback method for model change events
     */
    public void reactOnNewData()
    {
        Object[] args = new String[] { m_valueprop };
        Object v = m_windowJ.call("getPropertyValue"+m_id, args);
        if (v == null)
            v = "";
        m_fieldJ.setText((String)v);
    }

    /**
     * button action handler
     */
    private void buttonAction()
    {
        // pass field's value into property
        Object[] args = new String[] { m_valueprop , m_fieldJ.getText() };
        m_windowJ.call("setProperty"+m_id, args);

        // call adapter method
        args = new String[] { m_method };
        m_windowJ.call("invokeMethodInModel"+m_id, args);
    }

    /**
     * Is called on load
     */
    public void init()
```

```
{
    m_windowJ = JSObject.getWindow(this);
    m_id = getParameter("id");
    m_valueprop = getParameter("valueprop");
    m_method = getParameter("method");

    createGUI();
}

// -----
// applet specific methods
// -----

private void createGUI(){..}

public void destroy(){..}

private void cleanUp(){..}
}
```

You see:

- JavaScript methods are called using `JSObject`. It is part of *plugin.jar* of Sun's Java Virtual machine.
- The method `reactOnNewData()` accesses the fresh property value.
- The method `buttonAction()` first sets the user input and then calls the adapter method.

Summary

Writing new controls requires a profound knowledge of HTML and JavaScript. In principle, everything is simple, but there are a couple of pieces which have to be put together in order to form a control properly:

- You have to render the control via HTML.
- You have to manipulate the control via JavaScript - in case you have a dynamic control.
- You have to bind the control to adapter properties/methods.
- You have to pay attention to the fact that all controls are living in the same page - and there must not be any confusion with naming of IDs and method names.
- You have to use the JavaScript initialization for registering your control inside the internal eventing when new page content arrives inside the client.
- You have to properly fill the protocol item.

Some topics have been mentioned here, but have not been fully explained. For more information, see [Special Issues](#).

7

Special Issues

■ Protocol Item	58
■ Bringing Controls into the Layout Painter	59
■ Bringing Controls into the Layout Painter - Data Types	60
■ Array Binding	61
■ Text ID/Multi Language Controls	61

Protocol Item

Inside a tag handler, a protocol item is passed in the called methods. There are some mandatory tasks that you have to do with a protocol item:

- You must tell the protocol item every property you are referencing from your control.

This information is required because only these properties are transferred from the server to the client at runtime which are referenced inside the page.

- You must tell the protocol item every text ID you are referencing from your control.

Again this information is used to send the right text IDs to the client processing.

In case of using macro controls, one macro control is rendered into many normal controls. Each normal control is treated in the way that it generates corresponding HTML/JavaScript and in the way that it itself tells to which properties it binds; that is, each normal control adds its properties/text IDs itself: when your macro control contains some FIELD controls, then each FIELD control will tell during generation the adapter properties to which it binds - there is no necessity for you to re-tell on macro control level.

But: you might tell on macro control level that all the contained adapter properties are not provided via one-by-one implementation but by implementing a server-side class already providing all sub-properties. In this case, you can use the protocol item in the following way: We call these classes "binding classes", see also [Creating Macro Controls Out of Existing Controls](#). To add a binding class ADDRESSInfo, you use the protocol item in the following way:

- Call `addProperty('nameOfProperty', 'serverSideClass')`. For example:

```
addProperty(m_addressprop, 'ADDRESSInfo')
```

- Tell that all property definitions made by internally contained controls are not relevant for implementation by calling the method `suppressFurtherCodeGenEntries()`.

- Call `IXSDGenerationHandler xga = protocolItem.findXSDGenerationHandler();` to get access to an object which implements the `IXSDGenerationHandler` interface. This object is responsible for the generation of the corresponding data bindings for NATPAGE layouts.

- Call the method `addControlInfoClass` and pass your binding class as the third parameter. The second parameter is the name of the complex property to which you apply the binding:

```
xga.addControlInfoClass(protocolItem, myproperty, ADDRESSInfo.class);
```

For more information, see the corresponding Javadoc files of your Natural for Ajax or NaturalONE installation.

Bringing Controls into the Layout Painter

The Layout Painter is configured via a file *editor.xml* inside the `<installdir>/cis/config/` directory. This file contains information about all controls which are available inside the editor. For each control, the list of attributes and the list of possible subnodes is listed.

Have a look at the file - the structure is self-explaining.

With early versions, you had to bring own controls into the *editor.xml* file by editing it accordingly. The disadvantage was that every time Application DesignerApplication ComposerNatural for Ajax changed the *editor.xml* file, you had to reapply your changes. Application DesignerApplication ComposerNatural for Ajax now offers a dynamic way of adding own controls into the logical structure of the *editor.xml*.

Write an *editor_xyz.xml* file and place it into the same directory as *editor.xml*. "xyz" should be the same name as the one you chose as the prefix for your control library. Each *editor_xyz.xml* file holds information about the controls of the *xyz* control library:

- data types of a tag
- name of control tags
- attributes of tags
- subnodes a tag may have
- subnode extensions for existing Application DesignerApplication ComposerNatural for Ajax tags - this means, you define below which Application DesignerApplication ComposerNatural for Ajax controls your new tags should be positioned

The following definition shows the usage of the *editor_xyz.xml* file:

```
<!--
Dynamic extension of editor.xml file.
-->

<controllibrary>
  <editor>

    <!-- datatype TEXT -->
    <datatype name="demo:count">
      <value id="1st" name="First"/>
      <value id="2nd" name="Second"/>
      <value id="3rd" name="Third"/>
    </datatype>

    <!-- control DEMOCONTROL -->
    <tag name="demo:democontrol">
      <attribute name="text" datatype="demo:count"/>
```

```
</tag>
<tagsubnodeextension control="itr" newsubnode="demo:democontrol"/>
<tagsubnodeextension control="tr" newsubnode="demo:democontrol"/>

<!-- control DEMOCONTROLDYN -->
  <tag name="demo:democontroldyn">
    <attribute name="textprop"/>
  </tag>
<tagsubnodeextension control="itr" newsubnode="demo:democontroldyn"/>
<tagsubnodeextension control="tr" newsubnode="demo:democontroldyn"/>

<!-- control ADDRESSROWAREA -->
<tag name="demo:addressrowsarea">
  <attribute name="addressprop"/>
</tag>
<tagsubnodeextension control="pagebody" newsubnode="demo:addressrowarea"/>

</editor>
</controllibrary>
```

Note that the structure of the file directly corresponds to the structure of the original *editor.xml* file. The data is an add-on that is logically added to the information from the *editor.xml* file.

Note also that both new data types and new control tags are named together with their prefix - in order not to mix up with standard Application DesignerApplication ComposerNatural for Ajax controls or with controls of other control library providers.

Bringing Controls into the Layout Painter - Data Types

In a previous section, you learned how to integrate controls into the Layout Painter (see [Integrating Controls into the Layout Painter](#)). In addition to the basic concepts described in that section, you can also apply data-type definitions to your control attributes.

The following example shows how to apply data types to an attribute *editor_nadc.xml* file:

```
<!--
Dynamic extension of editor.xml file.
-->

<controllibrary>
  <editor>

    <!-- datatype TEXT -->
    <datatype name="nadc:count">
      <value id="1st" name="First"/>
      <value id="2nd" name="Second"/>
      <value id="3rd" name="Third"/>
    </datatype>
```

```
<!-- control MYCONTROL -->
<tag name="nadc:mycontrol">
  <attribute name="fixedtext" datatype="nadc:count"/>
</tag>
....
```

Note that both new data types and new control tags are named together with their prefix - in order not to mix them up with standard Application Designer controls or with controls of other control library providers.

Array Binding

As shown in the `CUSTC2-P` example of the Natural for Ajax demos, you can create controls with repeated data structures without having to deal with all the details of array bindings. The control `NADC:ADDRESSLIST` is an example for this. The idea is to have your own custom controls and simply reuse the repeated concepts and array bindings of the framework together with your own custom controls. We recommend to use this approach whenever possible.

If you really need to implement your own repeated data binding, you will find some details for generating and implementing your own array binding in the description of the `CUST2-P` example and the corresponding Natural for Ajax demos.

Text ID/Multi Language Controls

Please contact Software AG in case you create new controls with language-dependent information - and if you want to use the same translation methods as Application Designer does for these controls.

8 Control Library

You have written nice sets of controls? Why not pass these controls to others who might be interested?

Application DesignerApplication ComposerNatural for Ajax will build up a library of control libraries inside the web. If desired, we will check the control library for being conform to the Application DesignerApplication ComposerNatural for Ajax framework - and then publish it within our pages. There will be no publishing without your explicit agreement. Licensing conditions - between you and the users of your control - will be defined by yourself and must be clearly defined before publishing.

Please contact the Application DesignerApplication ComposerNatural for Ajax team at Software AG.

