

Casabac GUI Server

Control Developer's Guide

Release 2.5
2003-11-15

VERSIONS & CHANGES	3
OVERVIEW.....	4
CONTROL CONCEPT.....	5
PAGE GENERATION	5
CONTROL INTERFACE	6
<i>ITagHandler</i>	6
<i>Call Sequence</i>	6
<i>Extensions of ITagHandler</i>	7
LIBRARY CONCEPT	7
<i>Control Libraries</i>	8
BINDING CONCEPT	8
<i>Referencing Adapter Properties and Methods</i>	8
COMPOSING NEW CONTROLS OUT OF EXISTING CONTROLS	10
CONCEPT	10
EXAMPLE	10
<i>Control Handler Code</i>	11
<i>Control Adapter Code</i>	14
CONTROLS WITH REPEAT STRUCTURES	15
SUMMARY	15
CREATING NEW CONTROLS.....	17
CONCEPT	17
EXAMPLE 1	17
JAVASCRIPT FUNCTIONS	18
EXAMPLE 2	20
SUMMARY	22
SPECIAL – BUT IMPORTANT! - ISSUES.....	23
PROTOCOL ITEM	23
BRINGING CONTROLS INTO THE LAYOUT PAINTER	23
TEXT ID/ MULTILANGUAGE CONTROLS	24
CONTROL LIBRARY – LIBRARY	25

Versions & Changes

2002-09-12 – First Version

2003-06-14 – Release 2.0

- No content changes

2003-11-02

- If creating new "composed controls" you so far had to create ids for the contained controls on your own – now you can obtain the ids from a Java API (`com.casabac.gui.protocol.ProtocolItem.generatePageControlIndex()`).
- You are now able to define "editor.xml" extensions. This means you can add your own controls by an "editor_XXX.xml" file – where "XXX" represents the prefix name of your library. The content of your add-on file will be dynamically mixed with the original editor.xml file content.

Overview

This documentation provides information about the Casabac control concept. Please first become familiar with the "normal development" of screens inside the Casabac GUI Server - the documentation "Developer's Guide" contains the corresponding information.

When do you need own 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 out of a certain arrangement of fields and labels, which form an address. This kind of building controls is called "Composing Controls" in this documentation - you take what is available and group it into certain units.
- (2) You want to create really new controls - maybe you need some special kind of icon with a certain behaviour.

While case (1) does not require from you to deal with JavaScript and HTML, case (2) requires a profound knowledge of JavaScript and HTML and the use of the JavaScript library functions which are available via the Casabac framework.

Due to the usage of XML as layout definition format and due to an open interface, how to integrate your control definitions into the page generation process of the Casabac GUI Server, the Casabac control concept is a flexible and open framework. Actually, all Casabac controls are themselves following the framework - there is no "special way" or "shortcut", which is internally used.

One part of the concept is the definition of controls, i.e. control-tags with certain attributes, which you can integrate via a tag library concept into layout definitions. The other part is the binding to server side adapter properties. Following the "strict" Casabac architecture - that the GUI is a reflection of a "net-data"/ "model-data", dynamic controls have to determine their data at runtime from adapter properties. This binding concept is important:

- On the one hand you want newly created controls to reference to adapter properties/ methods.
- On the other hand you want to compose controls (e.g. an "address area") and want to bind them to complex objects (e.g. an "address object") on server side - already providing for a set of data and methods.

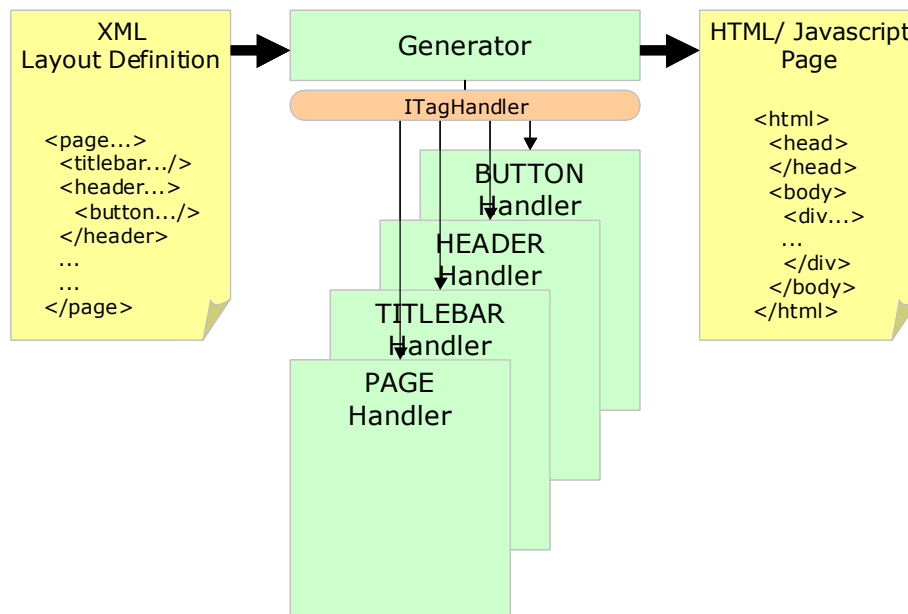
Please read the part "Binding between Page and Adapter" of the "Developer's Guide" to get detailed information, especially about the hierarchical name binding ("access path").

Control Concept

This chapter gives you an overview about the control concept. You will not be able to create own controls immediately after reading this chapter, but you might have enough background information in order to understand the examples in the following chapters.

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 Casabac Layout Painter – when previewing a layout. It also can be called from outside processing.



A generator-program (`com.casabac.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 consequences processes the string tag by tag. For each tag it creates one object of a tag handler class, which it finds via library definitions and naming conventions.

Each tag handler is called via a defined interface (`com.casabac.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 "</page>")
- 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

ITagHandler

The interface "com.casabac.gui.generate-ITagHandler" contains three methods, which represent the different point of times when the generator calls a tag handler.

```
package com.casabac.gui.generate;

import org.xml.sax.AttributeList;
import com.casabac.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);
}
```

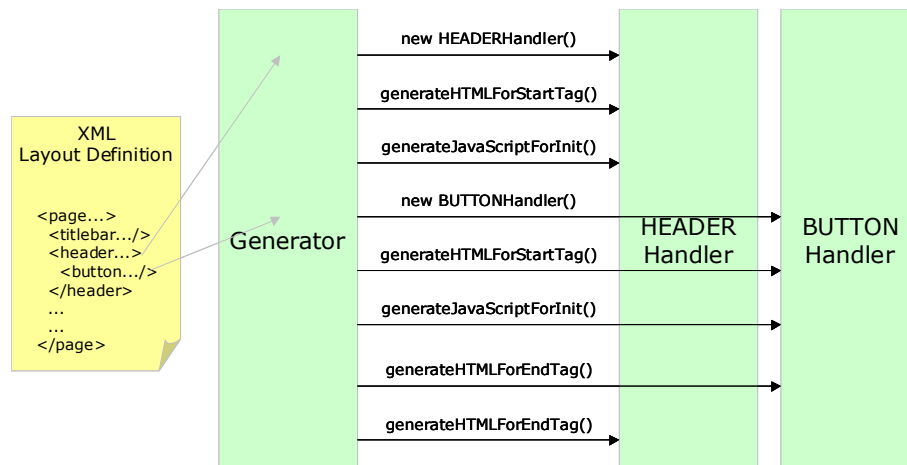
A detailed information about the methods can be found inside the JavaDoc-documentation, which is part of your Casabac installation.

Call Sequence

A tag is processed by the generator in the following 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 then 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:



Please be aware of:

- 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 ITagHandler

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

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

You do not require any know how about these extensions to create your first controls. Documentation is provided for 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 Casabac tag – the according tag handler is found inside the package `"com.casabac.gui.generate"`, the class name is created by converting the tag name to upper case and appending "Handler".
E.g. if the generator finds the tag "header" it tries to use a tag handler class `"com.casabac.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 build in the same way as with standard Casabac controls.
E.g. if the generator finds the tag "demo:address" and in the configuration file the demo-prefix is

assigned to the package "com.casabac.demolibrary" then the full class name of the tag handler is "com.casabac.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 e.g. 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-implementations. The corresponding ".jar"-file has to be part of the Casabac application libraries in order to be found inside the Layout Painter and Layout Manager. I.e. it can be e.g. copied into the <installdir>/appclasses/lib-directory.

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

```
<controllibraries>
  <library package="com.casabac.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

Referencing Adapter Properties and Methods

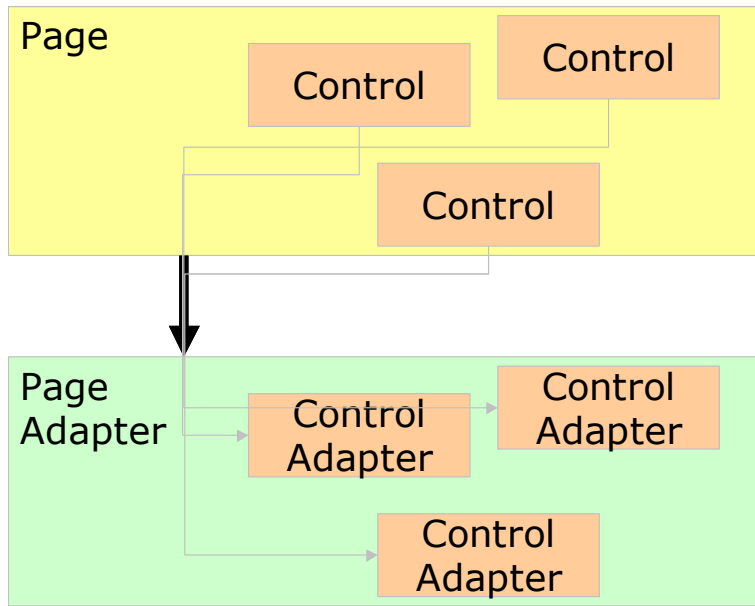
The "normal" binding concept between page and a corresponding class is:

- 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 inside the chapter "Binding between Page and Adapter" inside the "Developer's Guide" it is much more flexible. You can define hierarchical access paths both for methods and for properties.

You can for example define a FIELD control, which binds to the property "address.street". As consequence the adapter first is 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 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 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 class, 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 sound a bit abstract – please wait for the first example, then you will see how powerful and simple this binding concept is.

Composing new Controls out of existing Controls

Concept

The concept behind is quite simple: you write a control handler which itself calls other control handlers – the ones of existing controls. For a control handler it is not relevant if the control is called from the generator itself or from some interim instance.

In other words: your “composed control”-handler acts as a “mini-generator” to its contained controls.

Example

Let's have a look on the following page:

This page contains two address areas. Now let's look on the corresponding XML layout definition:

```
<page model="com.casabac.test14.ControlLibraryControlCompositionAdapter">
  <titlebar name="14 Demo: Composition of controls">
  </titlebar>
  <header withdistance="false">
    <button name="Exit" method="endProcess">
    </button>
  </header>
  <pagebody>
    <demo:addressrowarea addressprop="address">
    </demo:addressrowarea>
    <demo:addressrowarea addressprop="addresswife">

```

```

        </demo:addressrowarea>
    </pagebody>
    <statusbar withdistance="false">
    </statusbar>
</page>

```

You see that there is a control "demo:addressrowarea", which is used two times – once per address area. The control is responsible for arranging all its inner controls. Each tag has an ADDRESSPROP-attribute – we will see later on the way, this property is treated.

Control Handler Code

Let us have a look on the corresponding control handler, but: do not get confused: it seems to be a lot of code, but it's always the same...

```

package com.casabac.demolibrary;
import org.xml.sax.AttributeList;
import com.casabac.gui.generate.*;
import com.casabac.gui.protocol.*;

/**
 * Demo for control, which is composed out of different other controls
 *
 * <rowarea name="Address">
 *   <itr>
 *     <label name="First Name" width="100"/>
 *     <field valueprop="ADDRESS.firstName" width="150"/>
 *   </itr>
 *   <itr>
 *     <label name="Last Name" width="100"/>
 *     <field valueprop="ADDRESS.lastName" width="150"/>
 *   </itr>
 *   <vdist height="10"/>
 *   <itr>
 *     <label name="Street" width="100"/>
 *     <field valueprop="ADDRESS.street" width="300"/>
 *   </itr>
 *   <itr>
 *     <label name="Town" width="100"/>
 *     <field valueprop="ADDRESS.zipcode" width="50"/>
 *     <hdist width="5"/>
 *     <field valueprop="ADDRESS.town" width="245"/>
 *   </itr>
 *   <vdist height="10"/>
 *   <itr>
 *     <hdist width="100"/>
 *     <button name="Clear" method="clearAddress"/>
 *   </itr>
 * </rowarea>
 */
public class ADDRESSROWAREAHandler
    implements ITagHandler
{
    // -----
    // members
    // -----

    String m_addressprop;

    StringBuffer m_javascriptInit = new StringBuffer();

    ROWAREAHandler m_rowareaHandler = new ROWAREAHandler();
    ITRHandler m_itrHandler1 = new ITRHandler();
    LABELHandler m_labelHandlerFN = new LABELHandler();
    FIELDHandler m_fieldHandlerFN = new FIELDHandler();
    ITRHandler m_itrHandler2 = new ITRHandler();
    LABELHandler m_labelHandlerLN = new LABELHandler();
    FIELDHandler m_fieldHandlerLN = new FIELDHandler();
    VDISTHandler m_vdistHandler = new VDISTHandler();
    ITRHandler m_itrHandler3 = new ITRHandler();
    LABELHandler m_labelHandlerST = new LABELHandler();
    FIELDHandler m_fieldHandlerST = new FIELDHandler();
    ITRHandler m_itrHandler4 = new ITRHandler();
    LABELHandler m_labelHandlerTO = new LABELHandler();
    FIELDHandler m_fieldHandlerZI = new FIELDHandler();
    HDISTHandler m_hdistHandler = new HDISTHandler();
    FIELDHandler m_fieldHandlerTO = new FIELDHandler();
    VDISTHandler m_vdistHandler2 = new VDISTHandler();
    ITRHandler m_itrHandler5 = new ITRHandler();
    HDISTHandler m_hdistHandler2 = new HDISTHandler();
    BUTTONHandler m_buttonHandler = new BUTTONHandler();

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

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

```

```

fillProtocol(protocolItem);

SimpleAttributeList sal;
int controlId;

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
sal.addAttribute("name", "Address");
m_rowareaHandler.generateHTMLForStartTag(controlId, "rowarea", sal, handlersAbove, sb, protocolItem);
m_rowareaHandler.generateJavaScriptForInit(controlId, "rowarea", m_javascriptInit);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
m_itrHandler1.generateHTMLForStartTag(controlId, "itr", sal, handlersAbove, sb, protocolItem);
m_itrHandler1.generateJavaScriptForInit(controlId, "itr", m_javascriptInit);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
sal.addAttribute("name", "First Name");
sal.addAttribute("width", "100");
m_labelHandlerFN.generateHTMLForStartTag(controlId, "label", sal, handlersAbove, sb, protocolItem);
m_labelHandlerFN.generateJavaScriptForInit(controlId, "label", m_javascriptInit);
m_labelHandlerFN.generateHTMLForEndTag("label", sb);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
sal.addAttribute("valueprop", m_addressprop+.firstName");
sal.addAttribute("width", "150");
m_fieldHandlerFN.generateHTMLForStartTag(controlId, "field", sal, handlersAbove, sb, protocolItem);
m_fieldHandlerFN.generateJavaScriptForInit(controlId, "field", m_javascriptInit);
m_fieldHandlerFN.generateHTMLForEndTag("field", sb);

m_itrHandler1.generateHTMLForEndTag("itr", sb);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
m_itrHandler2.generateHTMLForStartTag(controlId, "itr", sal, handlersAbove, sb, protocolItem);
m_itrHandler2.generateJavaScriptForInit(controlId, "itr", m_javascriptInit);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
sal.addAttribute("name", "Last Name");
sal.addAttribute("width", "100");
m_labelHandlerLN.generateHTMLForStartTag(controlId, "label", sal, handlersAbove, sb, protocolItem);
m_labelHandlerLN.generateJavaScriptForInit(controlId, "label", m_javascriptInit);
m_labelHandlerLN.generateHTMLForEndTag("label", sb);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
sal.addAttribute("valueprop", m_addressprop+.lastName");
sal.addAttribute("width", "150");
m_fieldHandlerLN.generateHTMLForStartTag(controlId, "field", sal, handlersAbove, sb, protocolItem);
m_fieldHandlerLN.generateJavaScriptForInit(controlId, "field", m_javascriptInit);
m_fieldHandlerLN.generateHTMLForEndTag("field", sb);

m_itrHandler2.generateHTMLForEndTag("itr", sb);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
sal.addAttribute("height", "10");
m_vdistHandler.generateHTMLForStartTag(controlId, "vdist", sal, handlersAbove, sb, protocolItem);
m_vdistHandler.generateJavaScriptForInit(controlId, "vdist", m_javascriptInit);
m_vdistHandler.generateHTMLForEndTag("vdist", sb);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
m_itrHandler3.generateHTMLForStartTag(controlId, "itr", sal, handlersAbove, sb, protocolItem);
m_itrHandler3.generateJavaScriptForInit(controlId, "itr", m_javascriptInit);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
sal.addAttribute("name", "Street");
sal.addAttribute("width", "100");
m_labelHandlerST.generateHTMLForStartTag(controlId, "label", sal, handlersAbove, sb, protocolItem);
m_labelHandlerST.generateJavaScriptForInit(controlId, "label", m_javascriptInit);
m_labelHandlerST.generateHTMLForEndTag("label", sb);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
sal.addAttribute("valueprop", m_addressprop+.street");
sal.addAttribute("width", "300");
m_fieldHandlerST.generateHTMLForStartTag(controlId, "field", sal, handlersAbove, sb, protocolItem);
m_fieldHandlerST.generateJavaScriptForInit(controlId, "field", m_javascriptInit);
m_fieldHandlerST.generateHTMLForEndTag("field", sb);

m_itrHandler3.generateHTMLForEndTag("itr", sb);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
m_itrHandler4.generateHTMLForStartTag(controlId, "itr", sal, handlersAbove, sb, protocolItem);
m_itrHandler4.generateJavaScriptForInit(controlId, "itr", m_javascriptInit);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
sal.addAttribute("name", "Town");
sal.addAttribute("width", "100");
m_labelHandlerTO.generateHTMLForStartTag(controlId, "label", sal, handlersAbove, sb, protocolItem);
m_labelHandlerTO.generateJavaScriptForInit(controlId, "label", m_javascriptInit);
m_labelHandlerTO.generateHTMLForEndTag("label", sb);

controlId = protocolItem.generatePageControlIndex();
sal = new SimpleAttributeList();
sal.addAttribute("valueprop", m_addressprop+.zipCode");
sal.addAttribute("width", "50");
m_fieldHandlerZI.generateHTMLForStartTag(controlId, "field", sal, handlersAbove, sb, protocolItem);
m_fieldHandlerZI.generateJavaScriptForInit(controlId, "field", m_javascriptInit);
m_fieldHandlerZI.generateHTMLForEndTag("field", sb);

```

```

        controlId = protocolItem.generatePageControlIndex();
        sal = new SimpleAttributeList();
        sal.addAttribute("width", "5");
        m_hdistHandler.generateHTMLForStartTag(controlId, "hdist", sal, handlersAbove, sb, protocolItem);
        m_hdistHandler.generateJavaScriptForInit(controlId, "hdist", m_javascriptInit);
        m_hdistHandler.generateHTMLForEndTag("hdist", sb);

        controlId = protocolItem.generatePageControlIndex();
        sal = new SimpleAttributeList();
        sal.addAttribute("valueprop", m_addressprop + ".town");
        sal.addAttribute("width", "245");
        m_fieldHandlerTO.generateHTMLForStartTag(controlId, "field", sal, handlersAbove, sb, protocolItem);
        m_fieldHandlerTO.generateJavaScriptForInit(controlId, "field", m_javascriptInit);
        m_fieldHandlerTO.generateHTMLForEndTag("field", sb);

        m_itrHandler4.generateHTMLForEndTag("itr", sb);

        controlId = protocolItem.generatePageControlIndex();
        sal = new SimpleAttributeList();
        sal.addAttribute("height", "10");
        m_vdistHandler2.generateHTMLForStartTag(controlId, "vdist", sal, handlersAbove, sb, protocolItem);
        m_vdistHandler2.generateJavaScriptForInit(controlId, "vdist", m_javascriptInit);
        m_vdistHandler2.generateHTMLForEndTag("vdist", sb);

        controlId = protocolItem.generatePageControlIndex();
        sal = new SimpleAttributeList();
        m_itrHandler5.generateHTMLForStartTag(controlId, "itr", sal, handlersAbove, sb, protocolItem);
        m_itrHandler5.generateJavaScriptForInit(controlId, "itr", m_javascriptInit);

        controlId = protocolItem.generatePageControlIndex();
        sal = new SimpleAttributeList();
        sal.addAttribute("width", "100");
        m_hdistHandler2.generateHTMLForStartTag(controlId, "hdist", sal, handlersAbove, sb, protocolItem);
        m_hdistHandler2.generateJavaScriptForInit(controlId, "hdist", m_javascriptInit);
        m_hdistHandler2.generateHTMLForEndTag("hdist", sb);

        controlId = protocolItem.generatePageControlIndex();
        sal = new SimpleAttributeList();
        sal.addAttribute("name", "Clear");
        sal.addAttribute("method", m_addressprop + ".clearAddress");
        m_buttonHandler.generateHTMLForStartTag(controlId, "button", sal, handlersAbove, sb, protocolItem);
        m_buttonHandler.generateJavaScriptForInit(controlId, "button", m_javascriptInit);
        m_buttonHandler.generateHTMLForEndTag("button", sb);

        m_itrHandler5.generateHTMLForEndTag("itr", sb);
    }
    m_rowareaHandler.generateHTMLForEndTag("rowarea", sb);
}

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

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

// -----
// 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 + ".firstName", "boolean");
    pi.addProperty(m_addressprop + ".lastName", "String");
    pi.addProperty(m_addressprop + ".street", "String");
    pi.addProperty(m_addressprop + ".zipCode", "String");
    pi.addProperty(m_addressprop + ".town", "String");
}
}
}

```

Let's have a look on the "building blocks" of the code:

- The class has a member "m_addressprop". This member is filled directly at the beginning of the method "generateHTMLForStartTag"-method – inside the "readAttributes()" -method. The attribute list is walked through and checked for the attribute "addressprop".
- As 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.

Please pay attention to 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.

- For each control tag which is part of the composed control one own tag handler instance is created. In the processing of the method "generateHTMLForStartTag()" these control handlers are called in a sequence which represents their sequence which normally would occur inside a normal page generation. Please pay attention that the JavaScript initialisation info (calling of method "generateJavaScriptForInit") is appended to an internal string buffer "m_javascriptInit" – and is passed later on in one step to the generator's processing. Every control, which has a certain binding to adapter properties and methods, does so by always referencing a fix field/ method inside the "base property", which is passed via the "m_addressprop"-value. The consequence is: if the user, who uses this address-control, defines the attribute ADDRESSPROP to be "wifeAddress", then the fields bind to the properties "wifeAddress.firstName", "wifeAddress.lastName" etc.
- You see that each control gets a unique id. The id is generated by the method generateControlIndex() that is available in the ProtocolItem object.

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 principal 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. E.g. 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. E.g. the implementation is:

```
package com.casabac.demolibrary;
import com.casabac.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 now can use this control adapter class and can automatically “take over” all its contained properties and methods. E.g. the page adapter of the page of this example might look like:

```

package com.casabac.test14;

import com.casabac.demolibrary.*;
import com.casabac.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”.

Please be aware of that you can use “all Java possibilities” on server side to let the control adapter interact with your page adapter. Maybe you want to get 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.

Controls with REPEAT Structures

Please contact Casabac if you want to create composed controls with embedded REPEAT structures – as used in ROWTABLEAREA2. There are certain issues to be recognized there – that are not yet currently documented in this documentation.

Summary

In general, “everything” is quite simple:

- You can define controls, which serve as kind of “Macro” for creating other controls.
- You have a control handler class, in which you can control the generation of contained controls in a very fine granular way.

- You can bind properties and methods in a hierarchical way.

It's the combination of all aspects, which make the powerful result:

- You can build complex controls and define the binding in a way that the controls bind to predefined Java classes (control adapter classes), which contain the data and behaviour of the control.

The concept of building composed controls as consequence builds a strong modularisation aspect of the Casabac GUI Server – both on client side and on server side.

Creating new Controls

In the previous chapter you learned how to compose complex controls out of existing controls. This chapter now will tell you how to build completely new controls – which are not part of the Casabac control set yet.

Concept

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

There is a JavaScript function library available, which can be directly accessed inside the HTML code, which is generated. This library contains very useful method for accessing properties and executing methods of the "model" ("net-data"), which behind the page.

Example 1

The first example is a quite simple one: a tag with the name "democontrol" is introduced, which does nothing else then 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="ABCDEFGF">
    </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.casabac.demolibrary;
import org.xml.sax.AttributeList;
import com.casabac.gui.generate.*;
import com.casabac.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 first the attributes which are defined with the tag are read and the protocol is filled.
- Then plain HTML information is appended to the HTML string, which is passed as 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.

JavaScript Functions

For more interactive control – which e.g. use certain data coming from the server side adapter – you need to access certain JavaScript functions which are available inside the client. Inside the HTML page, which is generated, there is an object "csciframe" available, which provides for a certain set of functions to use.

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. As consequence these functions, which you are allowed to access, are listed inside this chapter. You must only stay with these functions – even if you may see additional ones inside the JavaScript sources of the Casabac GUI Server.

Function	Description
setProperty(pn,pv)	<p>Sets a property value inside the adapter. The values 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.</p> <p>pn = name of property pv = value</p> <p>Examples: <code>csciframe.setProperty("companyName","Casabac");</code> <code>csciframe.setProperty("address.firstName","John");</code> <code>csciframe.setProperty("addresses[2].firstName","Maria");</code></p>
getProperty(pn)	<p>Reads a property value from the adapter (better: the client representation of the adapter).</p> <p>pn = name of property result = string of property value</p> <p>Examples: <code>var vResult1 = csciframe.getProperty("company");</code> <code>var vResult2 = csciframe.getProperty("addresses[2].firstName");</code></p> <p>Please pay attention to that the adapter value is always passed back as string. A boolean value as consequence is returned as "true"-string and not as "true"-boolean value.</p> <p>Null values of the adapter, i.e. where the Java adapter class on server side passes back "null", are returned as empty String (""). A JavaScript-null value is passed back if the property, which you ask for, does not exist.</p>
registerListener(me)	<p>Passes 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 (e.g. 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.</p> <p>The method, which you pass, must have a parameter "model" – which is not used anymore, but which has to be defined.</p> <p>Example:</p> <pre> function reactOnNewData(model) { var vResult = csciframe.getProperty("firstName"); alert(vResult); } csciframe.registerListener(reactOnNewData); </pre>

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

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 attribute of the tag definition, now the text is 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 to the same adapter property "text" as the FIELD-control.

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

```
package com.casabac.demolibrary;
import org.xml.sax.AttributeList;
import com.casabac.gui.generate.*;
import com.casabac.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 listener to adapter model changes. The functions is generated inside the generateHTMLForStartTag()-method.
- The name of the property is read via the attribute list into the member "m_textprop" – and 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 suffixes 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, which property is required by the control. This is necessary at runtime, because the Casabac runtime environment needs to find out, which data of an adapter property to send back to the client (see chapter "Binding between Page and Adapter" inside the "Developer's Guide").

Summary

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

- 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 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 initialisation for registering your control inside the "internal eventing", when new page contents arrives inside the client.
- You have to properly fill the protocol item.

...and: There are still some issues which are not mentioned yet – but which you might be interested when writing certain controls. But: the concept is covered inside this chapter – and after having built your first controls, there is no complex item more to be added.

Special – but important! - Issues

Protocol Item

Inside a tag handler you receive a protocol item. 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 run time, 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.

Bringing Controls into the Layout Painter

The Layout Painter is configured via a file "editor.xml" inside the "<installdir>/casabac/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 sub-nodes is listed.

Please have a look into the file – the structure is self-explaining.

Up to release 2.5 you had to bring own controls into the editor.xml file by editing it accordingly. The disadvantage was that every time Casabac changed the editor.xml file you had to reapply your changes. From release 2.5 on there is 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 prefix for your control library. Each "editor_xyz.xml" file holds information about the controls of the xyz-control library:

- datatypes of a tag
- name of control tags
- attributes of tags
- subnodes a tag may have
- subnode extensions for existing Casabac tags – this means you define below which Casabac 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"/>
  </editor>
</controllibrary>
```

```
<!-- 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>
```

Please 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.

Please also note that both new data types and new control tags are name together with their prefix – in order not to mix up with standard Casabac controls or with controls of other control library providers.

Text Id/ Multilanguage Controls

Please contact Casabac in case you create new controls with language dependent information – and if you want to use the same translation methods as Casabac does for these controls.

Control Library – Library

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

Casabac will build up a library of control libraries inside the web. If desired, we will check the control library for being conform to the Casabac 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 info@casabac.com!