Natural for Ajax

Java Pages Development

Version 9.3.4

September 2026

ADABAS & NATURAL

This document applies to Natural for Ajax Version 9.3.4 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 © 2007-2026 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: AT-NJX-DEVELOPMENT-JAVA-934-20260922

Table of Contents

Java Pages Development vii
1 About this Documentation 1
Document Conventions 2
Online Information and Support 2
Data Protection 3
I Binding between Page and Adapter 5
2 Phases of Adapter Processing 7
SET/INVOKE/GET Phase - The Default Phases 8
INIT Phase when Adapter is Constructed 9
DESTROY Phase when Adapter is Deregistered 10
3 Types of Class Binding 11
Direct Class Binding 12
Generic Class Binding 13
4 Types of Property Binding 15
5 Java Bean Property Binding 17
Class Binding 18
Method Binding 18
Property Binding 19
Access Path Restrictions 22
6 Dynamic Access Property Binding 25
Interface IDynamicAccess 26
Example 26
7 XML Property Binding 31
8 Getting Information about Access Paths 33
9 Exception Management Inside an Adapter Object 37
Normal Exceptions are to be Handled by the Application 38
Errors and Runtime Exceptions - The Default Behavior 39
Interrupting the Application Designer Request Processing - AdapterNotAvailableError 40
Errors and Runtime Exceptions - The Special Behavior 41
10 Additional Interfaces 43
Extending the Set of Simple Data Types 44
Avoid the Getting of Certain Simple Data Type Properties 45
Exchanging Objects by Converter Objects 46
II Details on Session Management 47
11 HTTP Sessions - Application Designer Sessions 49
12 Application Designer Session - Application Designer Subsessions 51
13 Application Designer Subsession - Application Designer Adapter Objects 53
14 How Things Start 55
Starting an Application Designer Session 56
Starting Additional Application Designer Subsessions 56
15 How Things End 59
End of an Application Designer Session 60

End of an Application Designer Subsession ... 60
End of an Application Designer Adapter ... 60
16 Workplace Management ... 61
17 Saving Context Data ... 63
Different Levels of Context ... 64
Accessing the Context ... 64
Typical Usage Scenarios ... 65
18 Session IDs ... 67
III Becoming a Member of the Startup Process ... 69
19 Overview ... 71
20 Startup Class ... 73
21 Registration ... 75
IV Adapting the Look & Feel ... 77
22 Introduction ... 79
23 Style Sheet File ... 81
24 Writing a New Style Sheet File ... 83
25 Selecting the Right Style Sheet ... 85
26 Dynamic Selection of the Style Sheet File ... 87
What You Can Do ... 88
Example ... 88
27 Static Selection of the Style Sheet File ... 91
V Multi-Language Management in Java Applications ... 93
28 Multi-Language Management in Java Applications ... 95
Defining the Language at Runtime ... 96
Dealing with Literals inside Your Adapter ... 96
VI Online Help Management ... 99
29 Basics ... 101
Supported Controls ... 102
Way from Control to Online Help Page ... 102
Content of HTML Page ... 104
Where to Put the HTML Help Files ... 105
HELPICON Properties ... 105
30 Customizing the Online Help Pop-up ... 109
Creating a Project-Specific Pop-up ... 111
Runtime Behavior ... 111
31 Other URL Rules? ... 113
32 Other Types of F1-Online Help? ... 115
VII Appendices ... 117
33 Appendix A - Call Sequence for Adapter ... 119
Normal Call Sequence ... 120
Call Sequence when a Subsession is Destroyed ... 121
Call Sequence when a Session is Destroyed ... 122
Error/ Runtime Exceptions ... 122
Pay Attention when Overwriting ... 122
34 Appendix B - Usage of Methods Inherited from the Adapter Class ... 123

Access to Lookup Session Context 124
Access to Application Designer Session Context 125
Access to other Adapters 125
Error Output 125
Page Navigation 126
Opening of Pop-up Dialogs 126
Frame Communication 126
Closing of a Page 127
Multi Language Management 127
35 Appendix C - Data Types to be Used by Adapter Properties 129
Supported Data Types 130
Data Types for Managing Date and Time 130
36 Appendix D - Class Loader Concepts 131
Design Time - Runtime 132
Class Loader Hierarchy 132
Preparing for Runtime 135
37 Appendix E - StartCISPage Servlet 137
Normal Calling of a Page 138
Appending Application Parameters 138
Controlling the Session Life Cycle 138
Controlling the Session ID 139
Setting Default Parameters 139
Mixing Parameters 140
Setting Parameters with the HTTP Method POST 140

Java Pages Development

Binding between Page and Adapter	Transferring data between the page which runs inside the browser and the adapter object
Details on Session Management	In-depth details about session management.
Becoming a Member of the Startup Process	Initialise an initialise this application, for example, by setting up some database connectionapplication, for example, by setting up a database connection
Adapting the Look & Feel	How to modify the default rendering with the help of style sheets
Multi-Language Management in Java Applications	Java-specific customizations of multi-language management
Appendices	Additional Java-specific concepts, strategies, and approaches

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 *`folder.subfolder.service`*, APIs, Java classes, methods, properties.
`Italic`	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.softwareag.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://hub.docker.com/publishers/softwareag** 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 AG 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.

I Binding between Page and Adapter

One of the basic concepts of the Application Designer environment is to provide a simple mechanism for transferring data between the page which runs inside the browser and the adapter object which runs inside Application Designer. The page renders content, while the adapter provides content.

Between the page and the adapter, there is a binding which is defined during development time:

- A page is bound to an adapter class by the PAGE tag.
- Controls are bound to properties and methods of the adapter class.

At runtime, this binding definition is used inside Application Designer for accessing the adapter objects to pull and push data.

The information provided in this part is organized under the following headings:

Phases of Adapter Processing
Class Binding
Types of Property Binding
Java Bean Property Binding
Dynamic Access Property Binding
XML Property Binding
Getting Information about Access Paths
Exception Management Inside an Adapter Object
Additional Interfaces

2 Phases of Adapter Processing

- SET/INVOKE/GET Phase - The Default Phases 8
- INIT Phase when Adapter is Constructed 9
- DESTROY Phase when Adapter is Deregistered 10

SET/INVOKE/GET Phase - The Default Phases

An adapter object is the logical representation of a page. The page runs inside the GUI client, the adapter runs inside the Java server.

The user changes information on the page (e.g. inputs some values into field controls) and operates some functions (e.g. chooses a button). Every time a function is invoked, a request is initiated from the client. The request contains all data that was changed on the client, and it contains the command (e.g. the method to be called; sometimes there is no explicit command but the request just is a "data transfer request", e.g. when having defined `FLUSH="server"` with a FIELD control).

The request is processed in three phases:

- (activate)
- SET phase
- INVOKE phase
- GET phase
- (passivate)

During the SET phase, Application Designer passes all changed property values into the property representations of the server side adapter object. In the following INVOKE phase, the method is called that is associated with the request. In the GET phase, Application Designer checks all property values if they have changed. If a change happened (i.e. during the INVOKE phase, some property values were changed), then the changes are communicated back as response to the client.

The SET and GET phases have dedicated methods which are called inside the adapter in order to signalize the start and end of the phases:

- **SET phase:**

```
reactOnDataTransferStart();
set...();
set...();
set...();
reactOnDataTransferEnd();
```

- **GET phase:**

```
reactOnDataCollectionStart();
get...();
get...();
get...();
reactOnDataCollectionEnd();
```

You can use the methods for diverse purposes:

- `reactOnDataTransferStart()`
  You may want to initialize certain internal data that needs to be initialized for each request processing.
- `reactOnDataTransferEnd()`
  You may want to check which properties actually have changed and which application checks have to be invoked.
- `reactOnDataCollectionStart()`
  You may want to build up some interim objects for complex data structures that allow a faster response for the following get calls.
- `reactOnDataCollectionEnd()`
  You may want to set certain data to initial values - in case they were changed during request processing.

INIT Phase when Adapter is Constructed

The INIT phase is processed only once per adapter instance - at the point of time when it is constructed.

The INIT phase is internally processed in two steps:

- Creation of the adapter object via the `new` operator (without any parameters).
- `init()`

Application Designer first creates an instance of an adapter object and then calls the `init()` method of the object.

Important: Many functions inside your adapter class (extended from Application Designer's `Adapter` class) are only available after having constructed the object. Application Designer first creates the instance of the object and then internally registers the object inside its internal data structures (session management, etc.). The `init()` method is called after the internal registration. All functions that require the adapter object to be correctly registered will fail when being called inside the constructor, but will not fail if called in the `init()` method.

Best practice: use the `init()` method as constructor for an adapter object.

DESTROY Phase when Adapter is Deregistered

The DESTROY phase is processed only once per adapter instance - when Application Designer has internally deregistered the adapter instance:

The DESTROY phase consists of one method that is called inside the adapter:

- `destroy()`

Application Designer's session management deregisters all adapters when a subsession or session is closed. All associated adapters are internally deregistered so that they are available for garbage collection. In addition, each instance receives a destroy signal so that it can internally pass back resources that it used.

3 Types of Class Binding

- Direct Class Binding 12
- Generic Class Binding 13

For each page, there must be one adapter class. The name of the adapter class is given inside the PAGE tag of a page by the corresponding `model` property. Several page definitions can point to one adapter class, for example, when you have several page variants to display the same logical content.

The Application Designer runtime can create an adapter instance of a page in two ways:

1. **Direct Class Binding**
   The Application Designer runtime directly tries to create an adapter instance from the name in the `model` property of the PAGE tag of the page.
2. **Generic Class Binding**
   If the direct class binding does not succeed, the Application Designer runtime tries to create a generic adapter instance that has to be made accessible by the application.

Option 1 is the typical one. Option 2 is a solution if you require an adapter to be used very generically inside your framework.

Direct Class Binding

The following page definition forces the Application Designer runtime to look for a `Test1Adapter` class inside the package `com.softwareag.cis.test`:

```
<page model="com.softwareag.cis.test.Test1Adapter">
    ...
    ...
    ...
</page>
```

The class itself is derived typically from the class `com.softwareag.cis.server.Adapter`:

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

import com.softwareag.cis.server.Adapter;

public class Test1Adapter extends Adapter
{
    // ------------------------------------------------------------------------
    // constructor - without parameters
    // ------------------------------------------------------------------------

    public Test1Adapter()
    {
    }

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

```
    // ---------------------------------------------------------------------

    /** The init message is called when an object is created and all
     *  runtime aspects are correctly set inside the adapter. */
    public void init()
    {
        ...
        ...
    }
}
```

The default constructor is required (without any parameters).

Generic Class Binding

If the runtime does not find the class, it tries to find a generic one. The name of the generic class is created in the following way:

- The package name is taken from the `model` property of the PAGE tag of your page definition.
- The class name is "GenericAdapter".

Example: if you bind a page to the class `com.softwareag.cis.test.Test2Adapter` and the runtime system cannot locate the class `Test2Adapter`, the system tries to load the class `GenericAdapter` in the same package as the class that could not be found:

```
<page model="com.softwareag.cis.test.Test2Adapter">
    ...
    ...
    ...
</page>
```

The generic adapter is just a normal adapter which typically supports the dynamic binding of properties (see below).

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

import com.softwareag.cis.server.Adapter;

public class GenericAdapter extends Adapter
{
    /** */
    public void init()
    {
        System.out.println("My original class name is " + this.m_modelName);
    }
}
```

Each adapter object can access the `m_modelName` member. This member is set after the initialisation of the object. It holds the original adapter name to which the page refers.

4 Types of Property Binding

There are different types of binding techniques that are provided:

- **Java Bean binding** - the adapter provides set/get methods.
- **Dynamic access binding** - the adapter provides the implementation of a generic interface to access its data.
- **XML access binding** - the property values are kept within XML files, together with the page layout.

5 Java Bean Property Binding

- Class Binding 18
- Method Binding 18
- Property Binding 19
- Access Path Restrictions 22

Class Binding

The page binding is defined in the PAGE tag of your page definition. The PAGE tag points to a class supporting the interface `com.softwareag.cis.server.IModel`. There is a class `com.softwareag.cis.server.Adapter` which implements this interface - which should be used to build adapter classes as subclasses of `Adapter`.

Example:

```
<page model="com.softwareag.cis.demo.DemoAdapter" ...>
  ...
  ...
  ...
</page>
```

The above definition points to a class which looks as follows:

```
package com.softwareag.cis.demo;

import com.softwareag.cis.server.*;

public class DemoAdapter
    extends Adapter
{
    // constructor - either no constructor or a constructor
    // without any parameters
    public DemoAdapter()
    {
    }
    ...
    ...
}
```

Note that the adapter class has at least a default constructor (without any parameters).

Method Binding

Controls triggering a method inside the adapter are bound to a method name of the adapter. The method implementation itself must be a method without any parameters.

Example:

```
<button name="Save" method="doSave" ...>
</button>
```

The above button definition points to a method inside the adapter class which looks as follows:

```
public void doSave()
{
    ...
    ...
}
```

Property Binding

Controls presenting or manipulating data of the adapter are bound to properties of the adapter. There is a flexible concept available that makes it possible for you to use the properties as described below.

- Simple Properties which are Provided Directly by the Adapter
- Simple Properties which are Provided by Embedded Objects of the Adapter
- Array Properties which are Provided Directly by the Adapter
- Array Properties which are Provided by Embedded Objects of the Adapter

Simple Properties which are Provided Directly by the Adapter

This is the easiest way of binding: the property name which you specify in the definition of the control is provided directly by the adapter object - by a corresponding set and get method. It depends on the control whether you have to provide both set and get methods or just one of them.

Following the Java Bean conventions, the first character of the property name is written as a capital letter inside the corresponding set or get method.

The get method must return a value which is either a simple data type or a “simple” object. A list of supported return values is shown in *Appendix C - Data Types to be Used by Adapter Properties*. The set method must offer one parameter to update its value at runtime. The parameter type must either be a simple data type or one of the classes that are listed in appendix C.

Example:

```
<field valueprop="name" ...></field>
<field valueprop="age" ...></field>
<field valueprop="weight" ...></field>
<field valueprop="birthday" ...></field>
```

The above field definitions are bound to the following set/get methods:

```
public void setName(String value) { ... }
public String getName() { ... }

public void setAge(int value) { ... }
public String getAge() { ... }

public void setWeight(float value) { ... }
public float getWeight() { ... }

public void setBirthday(Cdate value) { ... }
public Cdate getBirthday() { ... }
```

The correct property name starts with a lowercase letter, because the first letter is always converted to lowercase. Example:

```
<field valueprop="cAPITAL" ...></field>
```

The above field definition is bound to the following set/get method:

```
public void setCAPITAL(String value) { ... }
public String getCAPITAL() { ... }
```

Simple Properties which are Provided by Embedded Objects of the Adapter

Properties can also be provided by an embedded object of the adapter. The embedded object itself must be accessible by a corresponding get method.

Example:

```
<field valueprop="address.street"></field>
```

The above field definition points to a value which is provided in the following way:

```
public class XYZAdapter
    extends com.softwareag.cis.server.Adapter
{
    // access in the adapter to the address object
    public Address getAddress() { ... }
}

public class Address
{
```

```
    public String getStreet() { ... }
    public void setStreet(String value) { ... }
}
```

You can build any chaining of properties you desire.

As shown in the example, embedded objects need not be adapter objects. Only the root object is required to be an adapter.

Array Properties which are Provided Directly by the Adapter

You can use array properties and can access them directly within your binding definitions. An array property always returns an array of objects, each object providing either simple properties or array properties. The type of the object array is not relevant for the Application Designer runtime. If you just return "Object[]" as a result of the method, this is sufficient.

Example:

```
<field valueprop="addresses[0].street" ...></field>
```

The above field definition points to a property which is implemented in the following way:

```
public class XYZAdapter
    extends com.softwareag.cis.server.Adapter
{
    // access in the adapter to the address object
    public Address[] getAddresses() { ... }
}

public class Address
{
    public String getStreet() { ... }
    public void setStreet(String value) { ... }
}
```

Note that the name used inside the control definition for binding (`addresses[0].street` in the our example) can either be entered manually or is implicitly created by some controls. Example: in a TEXTGRID control, specify an array property for the entire control and a simple property inside the COLUMN definition:

```
<textgrid arrayprop="addresses" ...>
    <column property="street" ...></column>
    <column property="city" ...></column>
</textgrid>
```

The TEXTGRID control itself uses these definitions to ask for the properties `addresses[0].street`, `addresses[0].city`, `addresses[1].street`, `addresses[1].city` etc. at runtime.

Note that it is not possible to access an array of simple objects directly. It is not possible to define a field in the following way

```
<field valueprop="streets[0]"></field>
```

having a method:

```
public String[] getStreets() { ... }
```

You always have to go through an array of objects where each element itself provides access to simple properties.

Array Properties which are Provided by Embedded Objects of the Adapter

You can use any combination of *Simple Properties which are Provided by Embedded Objects of the Adapter* and *Array Properties which are Provided Directly by the Adapter*.

Example: define access to array properties in the following way:

```
<field valueprop="person.addresses[0].street" ...></field>

<textgrid arrayprop="person.addresses" ...>
    <column property="street" ...></column>
    <column property="city" ...></column>
</textgrid>
```

Access Path Restrictions

At runtime, Application Designer transfers the data from the adapter to the client. For accessing the data, it uses the following strategy:

- It asks the adapter object for all properties. This means, it calls all get methods that are defined as public methods.
- If the get method returns a simple value, is marked to be transferred. (Whether it is really transferred, depends also on the delta management between the client and the server.)
- If the get method returns an object (e.g. an address object as used in the previous sections) or an array of objects, these objects are used for further drill down.

This mechanism is flexible on the one side, but dangerous on the other side: the Application Designer runtime will load *all objects* by following up the get methods.

Consequently, there is a certain access path restriction inside the Application Designer environment: if you generate a page (either by the Layout Painter or by the logical interfaces to the HTML generator) an access path restriction file is generated in addition. The HTML generator parses all tags

of a page; the controls themselves are bound to properties. This information is collected and written to a file.

This file is stored in the directory */accesspath* below the project directory. Please have a look at the files generated implicitly with your pages: the file contains a list of all access paths that are valid to be followed by runtime.

The name of the access path file is the same as the name of the page, but has the extension *.access*. Be aware of the fact that this access path file is inevitably important to avoid “mass loading” of data. Therefore, it must be a part of your software deployment.

6 Dynamic Access Property Binding

- Interface IDynamicAccess 26
- Example 26

Dynamic access binding is an additional binding technique that can be used together with **Java Bean binding** as described in the previous section. Dynamic access binding does not require an explicit definition of a set/get method for each property but is able to access the properties by generic data access functions.

Adapter objects, as well as embedded objects that are accessed inside the access path, may optionally support the interface `IDynamicAccess`. In this interface, you declare that there are additional properties to be accessed by generic access methods.

Interface IDynamicAccess

The interface definition looks as follows:

```
public interface IDynamicAccess
{
    public String[] findDynamicAccessProperties();
    public Class getClassForProperty(String property);
    public void setPropertyValue(String propertyName, Object value);
    public Object getPropertyValue(String propertyName);
    public void invokeMethod(String methodName);
}
```

It informs which dynamic properties are supported. In addition, you have to specify which class of a property it is. You can use any of the classes that are listed in ***Appendix C - Data Types to be Used by Adapter Properties*** for simple-value properties - and any class you desire for embedded object properties that you follow inside your access path. If you return a null value as a result of the `getClassForProperty()` method, the runtime returns the value as a String object.

Besides, you have to implement the generic set and get functions for property access.

There is a generic method invoked, e.g. when the user chooses a button in the page bound to a dynamically called method.

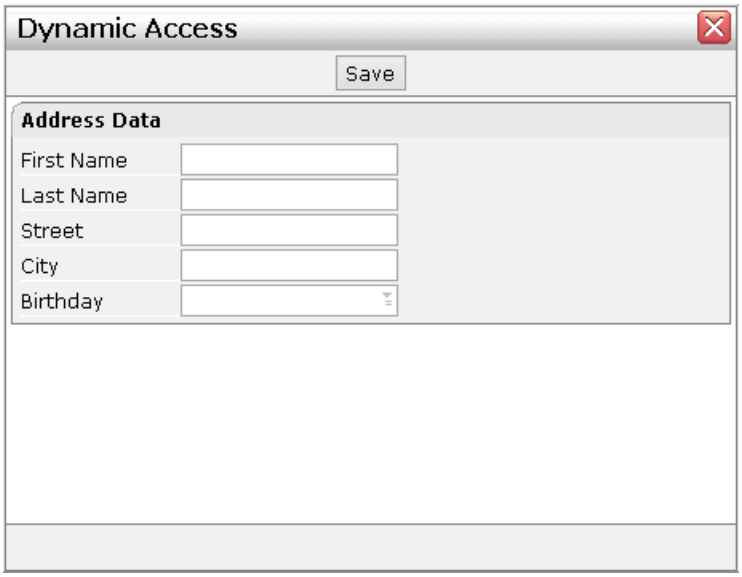
Example

The following example shows an adapter object that has two Bean properties (`firstName`, `lastName`) and three dynamic properties (`street`, `city`, `birthday`):

```
// This class is a generated one.

import com.softwareag.cis.server.Adapter;
import com.softwareag.cis.server.IDynamicAccess;
import com.softwareag.cis.util.CDate;

public class DynamicAccess_Adapter
    extends Adapter
    implements IDynamicAccess
{
    String m_firstName;
    String m_lastName;
    String m_street;
    String m_city;
    CDate m_birthday;

    public String[] findDynamicAccessProperties()
    {
        return new String[] {"street","city","birthday"};
    }

    public void setPropertyValue(String propertyName, Object value)
    {
        if (propertyName.equals("street")) m_street = (String)value;
        else if (propertyName.equals("city")) m_city = (String)value;
        else if (propertyName.equals("birthday")) m_birthday = (CDate)value;
        else throw new Error("No property " + propertyName + " available");
    }

    public Object getPropertyValue(String propertyName)
    {
        if (propertyName.equals("street")) return m_street;
        if (propertyName.equals("city")) return m_city;
        if (propertyName.equals("birthday")) return m_birthday;
        throw new Error("No property " + propertyName + " available");
    }

    public Class getClassForProperty(String propertyName)
    {
        if (propertyName.equals("birthday")) return CDate.class;
        //     default: null ==> String is assumed by runtime
        return null;
    }

    public void invokeMethod(String methodName) {}

    //  ------------------------------------------------------------------------
    //  bean properties
    //  ------------------------------------------------------------------------
    public void setFirstName(String value) { m_firstName = value; }
    public String getFirstName() { return m_firstName; }
    public void setLastName(String value) { m_lastName = value; }
```

```
    public String getLastName() { return m_lastName; }
}
```

Specifying a layout definition, there is no difference between the dynamic access properties and the bean properties.



The layout definition for the above page looks as follows:

```
<page model="DynamicAccess_Adapter">
    <titlebar name="Dynamic Access">
    </titlebar>
    <header withdistance="false">
        <button name="Save">
        </button>
    </header>
    <pagebody>
        <rowarea name="Address Data">
            <itr>
                <label name="First Name" width="100">
                </label>
                <field valueprop="firstName" length="20">
                </field>
            </itr>
            <itr>
                <label name="Last Name" width="100">
                </label>
                <field valueprop="lastName" length="20">
                </field>
            </itr>
            <itr>
                <label name="Street" width="100">
                </label>
```

```
                <field valueprop="street" length="20">
                </field>
            </itr>
            <itr>
                <label name="City" width="100">
                </label>
                <field valueprop="city" length="20">
                </field>
            </itr>
            <itr>
                <label name="Birthday" width="100">
                </label>
                <field valueprop="birthday" length="20" datatype="date">
                </field>
            </itr>
        </rowarea>
    </pagebody>
    <statusbar withdistance="false">
    </statusbar>
</page>
```

7 XML Property Binding

Use XML property binding with the following:

- ICONLIST control,
- MENU control,
- ROWTABSUBPAGES control, or
- for any simple property.

XML property binding uses XML files to access property values. Use the prefix "XML:" to indicate XML property binding.

```
<itr visibleprop="XML:isHomeAddressVisible">
   ...
</itr>
```

You see that the visibility of the row container is controlled by the XML property `isHomeAddressVisible`. An XML property is bound to a property tag (name-value pair).

```
<property name="isHomeAddressVisible" value="true">
</property>
```

The overall page layout is bound to an XML data file that contains all the property tags.

```
<xmlproperties>
   <property name="isHomeAddressVisible" value="true">
   </property>
   <property name="isBusinessAddressVisible" value="false">
   </property>
</xmlproperties>
```

The XML data file contains two property tags. With the first property, `isHomeAddressVisible` is set to "true"; with the second property, `isBusinessAddressVisible` is set to "false". At runtime,

you can switch between XML data files by changing the "XML data mode". Just use the following method in order to use the correct XML data file method:

```
Adapter.setXMLDataMode
```

The files are kept within directory <*webapp*>/<*project*>/*xmldata*. Each XML data mode is represented by a subdirectory. By default, the Application Designer server accesses the XML files within the directory *default*.

```
cis
    project
        xmldata
            default
                PersonInfoAdapter.xml
            fullinfo
                 PersonInfoAdapter.xml
```

8 Getting Information about Access Paths

Sometimes you need to get detailed information about a page accessing its adapter. Or, in other words: you want to get a detailed list of all the properties and objects which are referenced by your page definition to the corresponding adapter object.

For this reason, there is the class `CheckAccessPath` in the `com.softwareag.cis.server` package providing this information. The class has a `getInstance()` method to obtain an instance; the class has two additional methods:

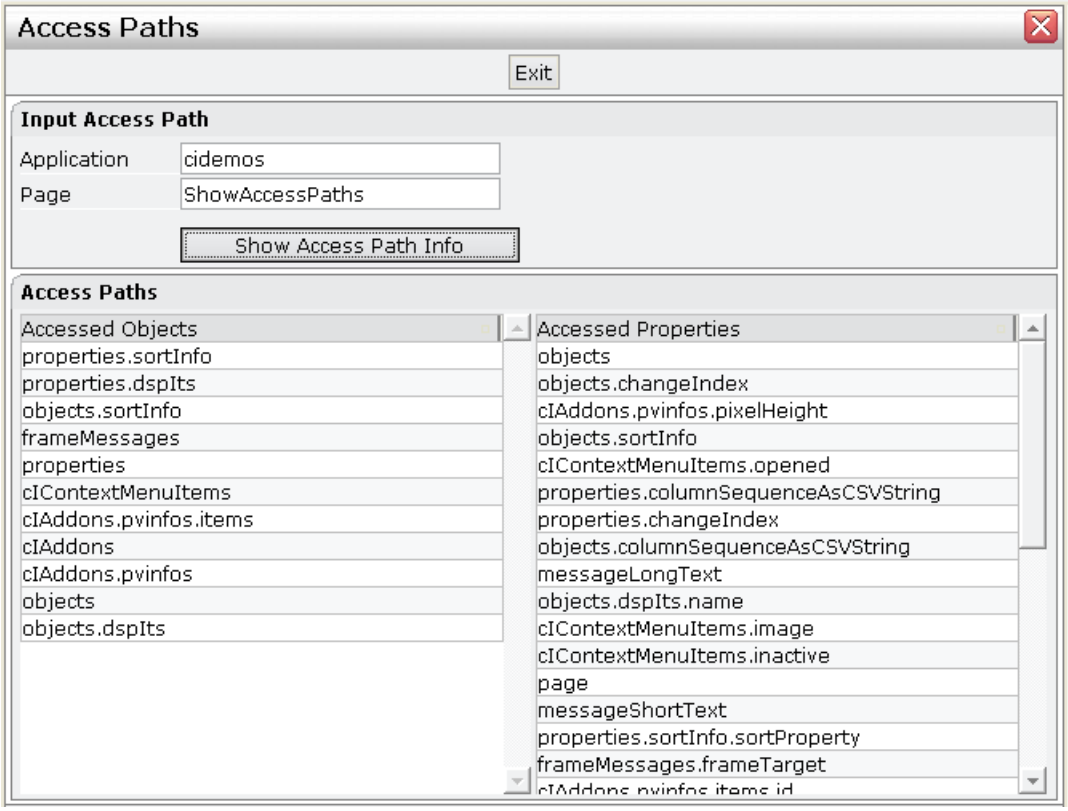
```
public String[] findAccessedObjects(String application,
                                    String reference);
public String[] findAccessedProperties(String application,
                                       String reference);
```

In both methods, the parameters are "application" and "reference". "application" is the application project in which a page is defined. "reference" is the name of the page itself, without ".xml".

The `findAccessedObjects` method returns a String array of all objects referenced in this page. An object is referenced if the properties are not directly plugged to the adapter itself but to subobjects. Example: if you bind a FIELD to the VALUEPROP "address.street" then "address" is the returned name.

The `findAccessedProperties` method returns a String array of all properties referenced in the page which are not complex properties, but simple value properties.

In the cisdemos project (which is part of the installation), there is a page "ShowAccessPaths" and the corresponding class `ShowAccessPathsAdapter` that shows an example of how to use the `CheckAccessPath` methods. The page allows you to enter the application name and the page name, and returns a list of referenced objects and properties:



In the class definition of the corresponding adapter, the code (finding the information about access paths) looks as follows:

```
// ---------------------------------------------------------------------------
// inner classes
// ---------------------------------------------------------------------------
 public class Info
{
    // property >objects[*].name<
    // property >properties[*].name<
    String m_name;
    public String getName() { return m_name; }
    public void setName(String value) { m_name = value; }
}

// ---------------------------------------------------------------------------
// property access
// ---------------------------------------------------------------------------

// property >app<
String m_app = "";
public String getApp() { return m_app; }
public void setApp(String value) { m_app = value; }

// property >page<
```

```
String m_page = "";
public String getPage() { return m_page; }
public void setPage(String value) { m_page = value; }

// array property >objects[*]<
TEXTGRIDCollection m_objects = new TEXTGRIDCollection();
public TEXTGRIDCollection getObjects() { return m_objects; }

// array property >properties[*]<
TEXTGRIDCollection m_properties = new TEXTGRIDCollection();
public TEXTGRIDCollection getProperties() { return m_properties; }

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

/** */
public void onShowAccessPath()
{
    // check
    if (m_app.trim().length() == 0)
    {
        outputMessage(MT_ERROR,"Please specify application project");
        return;
    }
    if (m_page.trim().length() == 0)
    {
        outputMessage(MT_ERROR,"Please specify page");
        return;
    }
    // fill data
    m_properties.clear();
    m_objects.clear();
    String[] objects = ↩
CheckAccessPath.getInstance().findAccessedObjects(m_app,m_page);
    String[] properties = ↩
CheckAccessPath.getInstance().findAccessedProperties(m_app,m_page);
    for (int i=0; i<objects.length; i++)
    {
        Info info = new Info();
        info.m_name = objects[i];
        m_objects.add(info);
    }
    for (int i=0; i<properties.length; i++)
    {
        Info info = new Info();
        info.m_name = properties[i];
        m_properties.add(info);
    }
  }
}
```

9 Exception Management Inside an Adapter Object

- Normal Exceptions are to be Handled by the Application 38
- Errors and Runtime Exceptions - The Default Behavior 39
- Interrupting the Application Designer Request Processing - AdapterNotAvailableError 40
- Errors and Runtime Exceptions - The Special Behavior 41

Application Designer binds its page processing to adapter objects providing properties and methods - as explained in the previous sections. What happens if an error happens at runtime, e.g. an error occurs in the method of an adapter object that is called after the user pressed a button?

Normal Exceptions are to be Handled by the Application

The first rule is: normal exceptions (i.e. no “Errors”, no “Runtime Exceptions”) are to be handled by the application itself.

This means: a property is provided by the corresponding set/get methods (or by an equivalent method when using dynamic binding). The methods must not throw any exception, i.e. in their declarations there is no "throws" element.

Example for a correct implementation:

```
public void setFirstName(String value)
{
    ...
    ...
}
public String getFirstName()
{
    ...
    ...
}
```

The following example is an *incorrect* implementation because application exceptions are thrown:

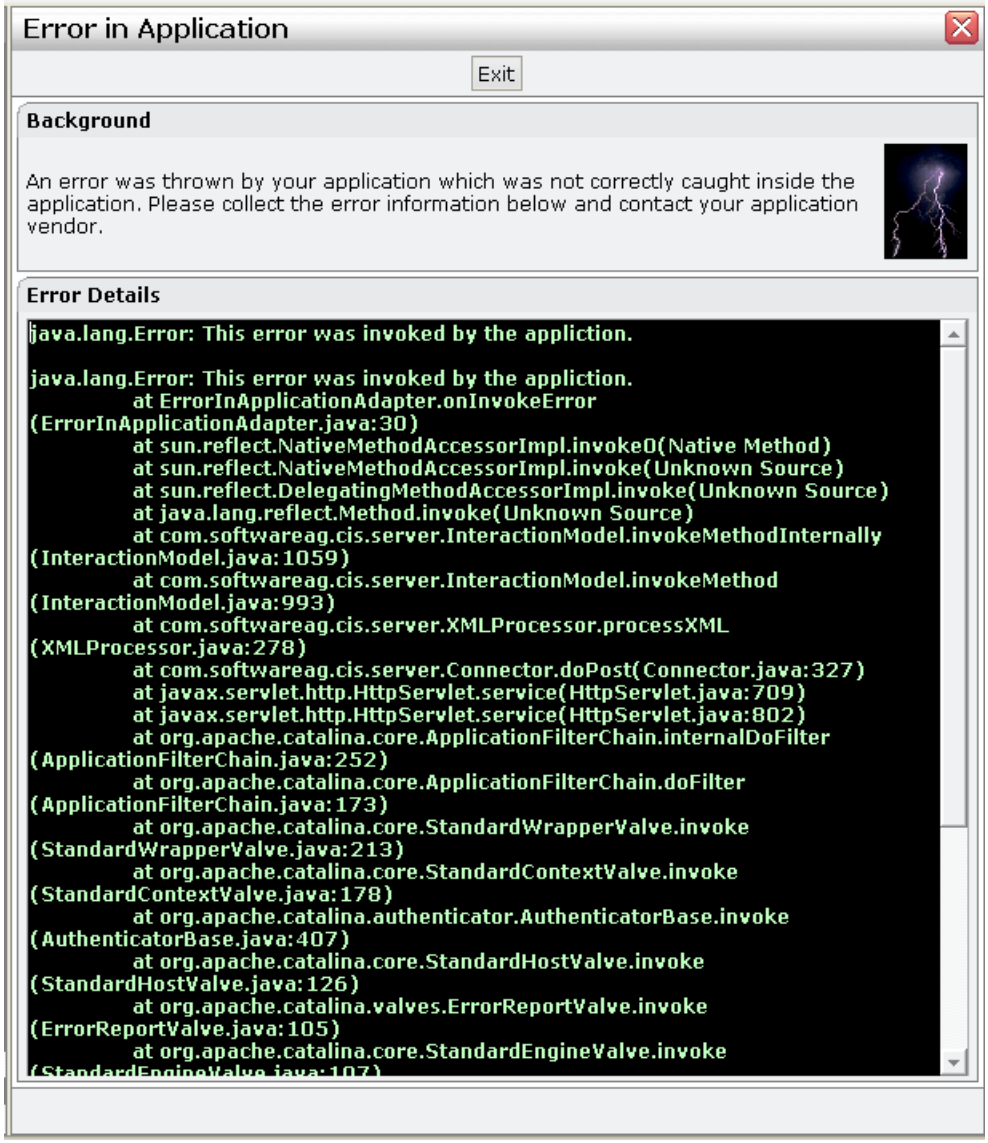
```
public void setFirstName(String value)
    throws ApplException
{
    ...
    ...
}
public String getFirstName()
    throws ApplException
{
    ...
    ...
}
```

Consequence: Application Designer passes values from the browser front end into the adapter object, invokes certain activities inside this object and collects data from the object to pass data changes back to the browser. Application exceptions are not relevant from Application Designer's point of view - they only affect the application internally.

Errors and Runtime Exceptions - The Default Behavior

Of course your application still can throw “Error” exceptions or “Runtime” exceptions. These are the exceptions that need not be declared inside a method's code - but that can be thrown any time at any place.

If Application Designer receives an error or runtime exception, Application Designer displays a page by default in which the error information is shown.



In addition, Application Designer writes a full stack dump into its runtime log.

Interrupting the Application Designer Request Processing - AdapterNotAvailableError

There may be some situations - within a special environment context - that do not allow to process the page at all. Maybe you have a page that requires a user to be logged on; if the Application Designer request processing starts now, you may decide with the method `reactOnDataCollectionStart` that you do not want to start the request since it does not make sense at all and just causes exceptions.

The only thing you want to do in such a scenario is to “escape” to a page which helps out of the situation. For example, if you miss logon information, you want to “escape” to the logon page and return to your original page afterwards.

The Java error class `AdapterNotAvailableError` is provided for this reason. The following adapter code shows an example on how to handle this error:

```
package com.softwareag.cis.demoapps;

...
...

public class Rescue1Adapter
    extends Adapter
{
    ...
    ...
    /** start of data transfer */
    public void reactOnDataTransferStart()
    {
        super.reactOnDataTransferStart();
        // fetch user and sytem info from session context
        m_user = (String)findSessionContext().lookup("rescueexample/user",false);
       m_system = (String)findSessionContext().lookup("rescueexample/system",false);
        // if not logged on ==> switch to rescue2 page
        if (m_user == null ||
            m_system == null)
        {
            // prepare Rescue2 page
           Rescue2Adapter r2a = (Rescue2Adapter)findAdapter(Rescue2Adapter.class);
            r2a.init("Rescue1.html");
            // throw error in order to interrupt normal processing and switch
            // to Rescue2 page
            throw new AdapterNotAvailableError("Rescue2.html");
        }
    }
    ...
```

```
    ...
}
```

Inside the `reactOnDataTransferStart` method, a user and a system variable are read from the session context. If one of them is null, the adapter decides to switch to page *Rescue2.html* and throws an `AdapterNotAvailableError` error. Before, it pre-fetches the page adapter of the page to escape to and initializes the page with certain information (in this example, it passes its own page name).

The error page is opened inside the same subsession as the one throwing the error.

Errors and Runtime Exceptions - The Special Behavior

There is a set of methods available in the adapter with which you can influence the standard error behavior:

- `handleErrorDuringInitPhase()`
- `handleErrorDuringSetPhase()`
- `handleErrorDuringInvokePhase()`
- `handleErrorDuringGetPhase()`

Depending on the method you have the following possibilities:

- `handleErrorDuringInitPhase()`
  This method is called when an error occurs in the `init` method of the adapter.
- `handleErrorDuringSetPhase()` **and**`handleErrorDuringGetPhase()`
  These methods are called when an error occurs during the set and the get phase of the adapter request processing.

  You may throw an `AdapterNotAvailableError` to navigate to a page of your own in order to present to the user detailed error information - and maybe also some way to solve the error.
- `handleErrorDuringInvokePhase()`
  This method is called when an error occurs during the invoke phase of the adapter request processing.

  You can decide whether normal Application Designer runtime processing continues, whether you want to navigate to an error handling page (via `PageNotAvailableError`), or whether the standard error processing of Application Designer is done.

See the API documentation (Java Doc) for further details.

10 Additional Interfaces

- Extending the Set of Simple Data Types 44
- Avoid the Getting of Certain Simple Data Type Properties 45
- Exchanging Objects by Converter Objects 46

Some additional interfaces are available which allow you to modify the binding behavior between a page and its adapter object. The interfaces are available via `com.softwareag.cis.server.IInteractionSessionMgr` which represents a general interface to the Application Designer runtime.

You receive an instance of `IInteractionSessionMgt` in the following way:

```
...
...
IInteractionSessionMgr iism = InteractionSessionMgrFactory.getInteractionSessionMgr();
...
... ↩
```

Extending the Set of Simple Data Types

As described previously in this part, Application Designer collects all the properties of an adapter object (and its contained objects) when collecting the data in order to respond to the browser client.

The way Application Designer collects the data for a certain object is:

- Application Designer collects all the properties that represent simple data types (int, float, String, BigDecimal, CDate, etc.; see *Appendix C - Data Types to be Used by Adapter Properties*).
- Application Designer investigates all properties that are non-simple datatypes and that are part of the access path of a certain page.

Sometimes you want to add a certain class to be managed as "Simple Datatype Class", i.e. Application Designer will not treat objects of this class as non-simple objects but will treat them as simple objects.

Simple objects have to provide a class implementation that

- provides a constructor in which the value is passed as a string object, and
- provides a `toString()` method to get the String representation of the contained value.

Example of a valid class:

```
public class ExtendedString
{
    String m_value;
    public ExtendedString(String value)
    {
        m_value = value;

    }
    public String toString()
    {
```

```
        return m_value;
    }
    ...
    ...
}
```

The class is registered by using the method `IInteractionSessionMgr.registerPropertyAccessSimpleDatatypeExtension()`:

```
IInteractionSessionMgr iism;
iism = InteractionSessionMgrFactory.getInteractionSessionMgr();
iism.registerPropertyAccessSimpleDatatypeExtension(ExtendedString.class);
```

Now having an adapter object (or follow-on object such as grid item) providing a property of type `ExtendedString`, Application Designer will not drill down the object but will use the object's `toString()` method to get its value and will use the object's constructor to pass new values to the application.

Avoid the Getting of Certain Simple Data Type Properties

In the previous sections, the general rule was explained: if Application Designer investigates an object during the get-data phase, then

- it takes all simple data type properties, and
- it takes those complex data type properties that are required by the corresponding page.

There is one possibility to fine-control the getting of simple data type properties: Every object that is investigated by Application Designer during the get phase (e.g. the adapter object) can implement the interface `com.softwareag.cis.server.IControlPropertyAccess`. The interface is defined as follows:

```
public interface IControlPropertyAccess
{
    public String[] findPropertiesNotToBeCollected();
}
```

When the interface is implemented, the get methods that are passed back by the `findPropertiesNotToBeCollected()` method are not processed.

Note that the method is called once per class - the first time Application Designer interacts with an object. You cannot tell Application Designer by this interface to sometimes use the property and sometimes not.

Exchanging Objects by Converter Objects

When Application Designer is accessing properties that are non-simple data type objects, there is the possibility to exchange the object and tell Application Designer to use a converter object instead.

The interfaces are:

- With `IInteractionSessionMgr.registerPropertyAccessConverter(Class forClass, IPropertyAccess Converter converter)` you can register a class (parameter `converter`) that is used as converter for another class (parameter `forClass`).
- The converter class itself must support the interface `IPropertyAccessConverter` that looks as follows:

```
public interface IPropertyAccessConverter
{
    public Object getConvertedObject(Object propertyValue);
}
```

For more details, see the JavaDoc API documentation.

II Details on Session Management

In *Working with Page Navigation*, there is a brief description on how Application Designer manages sessions. This part provides more details about session management.

In principle, the session management is hidden inside Application Designer. If you write normal applications running in the Application Designer workplace environment, you do not have to care about session management at all: you do not have to somehow collect data from a session object in order to work with it or do something similar.

However, reading this part is interesting for you if you want to know the following:

- What is the life cycle of an adapter?
- What amount of data is kept in an adapter?
- How does Application Designer internally arrange adapters?

This part is especially important for you if you:

- write a workplace-like application which serves as a frame for content applications;
- not only have Application Designer pages in your web application but also other servlets or JSP pages.

The information provided in this part is organized under the following headings:

HTTP Sessions - Application Designer Sessions

Application Designer Session - Application Designer Subsessions

Application Designer Subsession - Application Designer Adapter Objects

How Things Start

How Things End

Workplace Management

Saving Context Data

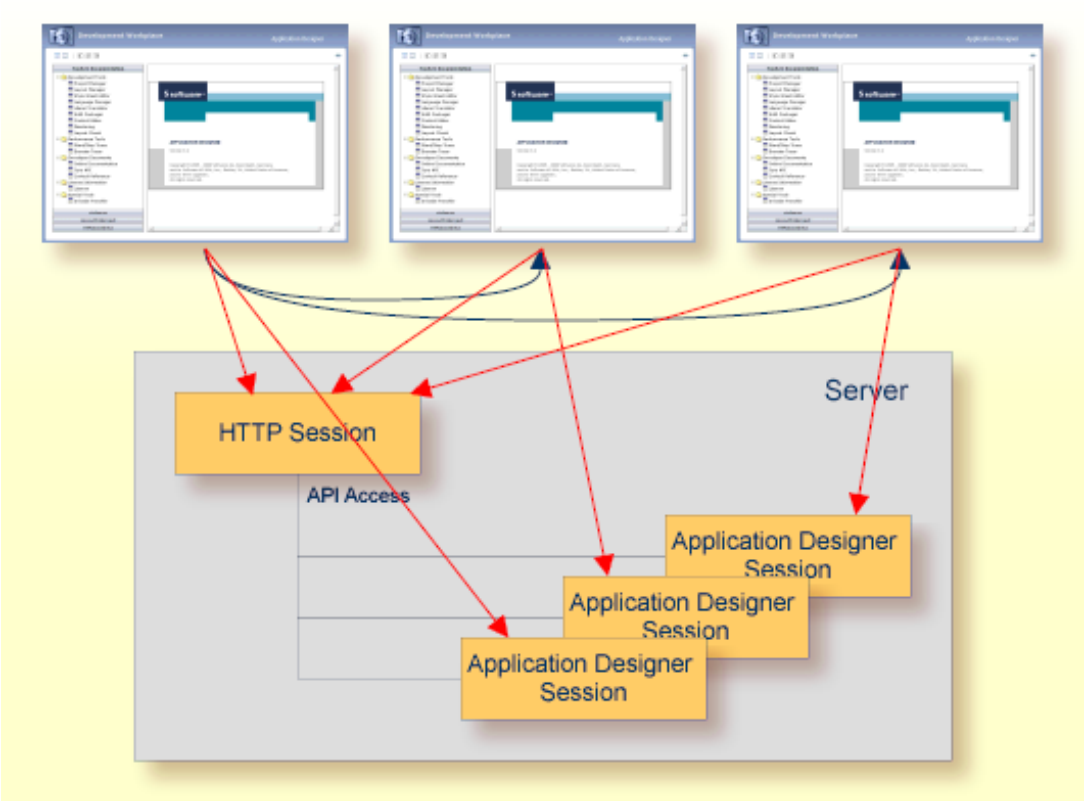
Session IDs

11 HTTP Sessions - Application Designer Sessions

If you have already developed servlets/JSPs, your first question will be: how do Application Designer sessions relate to HTTP sessions?

Application Designer adapters are living in sessions which are administered inside the Application Designer runtime environment. The sessions are kept in parallel to HTTP sessions, i.e. HTTP sessions may be used by other servlets/JSPs that may be part of your web application - but Application Designer itself does not require them. It is no problem to reach HTTP sessions from an adapter object via an API.

Why is Application Designer not using straight HTTP sessions? The problem is that HTTP sessions are sometimes the same for multiple browser instances. If you open a new browser instance from an existing browser instance (for example, with the Internet Explorer), then the corresponding session object on the server is shared between the browser instances. In the Application Designer session management, each instance of a browser (and if you want: each frame inside one browser) has its own clearly assigned session.



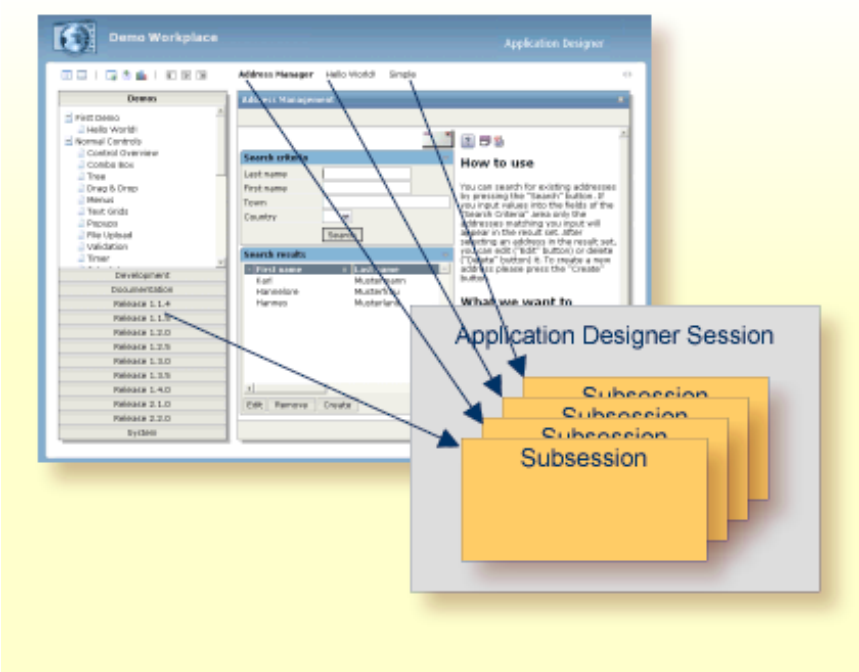
The above diagram shows the following:

- There are three browser instances sharing one HTTP session.
- Each browser instance has one related Application Designer session.
- There is an API from the Application Designer runtime to access the HTTP session.

12 Application Designer Session - Application Designer Subsessions

The Application Designer session concept knows one level below the Application Designer session: the Application Designer subsession. Adapter objects are living inside one subsession - and there can be multiple subsessions within one session.

Let us approach the subsessions by a practical example:

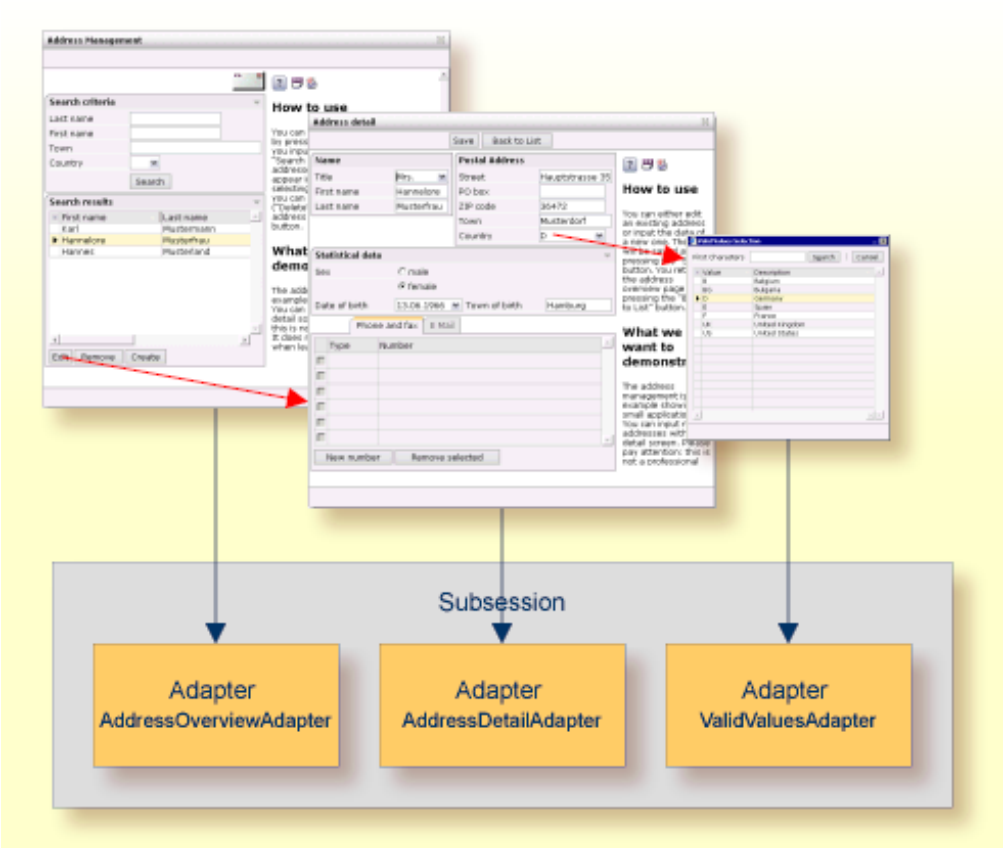


In the diagram, the Application Designer demo workplace is shown. Inside the workplace, three activities have been opened. The "Address Management" application is currently active. In addition, the workplace is also running.

See the next section ***Application Designer Subsession - Application Designer Adapter Objects*** for further information on this example.

13 Application Designer Subsession - Application Designer Adapter Objects

Each of the activities listed in the previous section *Application Designer Session - Application Designer Subsessions* is represented by a subsession on the server side. Each subsession itself is holding the adapters for the activity.



In the diagram above, the activity "Address Management" is shown in detail. It consists of several pages between which the user navigates. Each page belongs to one adapter of a certain class. The adapter instances are managed inside the subsession.

The general rules for administering adapter instances are:

- For each adapter class, there is one instance inside one subsession. This means: if you have several different pages between which you navigate inside one activity, and all pages are bound to the same Adapter class, then all pages are working with the same server side adapter instance.
- Adapter instances start to live when they are first accessed (e.g. by a page requesting them). They are kept as instances for the whole life cycle of a subsession - if not explicitly destroyed by the application via an API call.

Basically, there are two types of sessions:

- Each browser connected to Application Designer opens a new session inside the server. When closing the browser or navigating to another web page, this session is automatically destroyed at the server side.
- Within a session there are subsessions. Each subsession represents the state of one interaction process inside the browser. In the Application Designer workplace environment, you can open multiple parallel interaction processes, and you can switch from one to another. You may have other environments in which you do not want to offer the multi-interaction process management - and only have one subsession for the whole life cycle of a session.

Inside a subsession, the adapter instances are created. All navigation is done between pages that belong to the same subsession. See also *Working with Page Navigation*.

14 How Things Start

- Starting an Application Designer Session 56
- Starting Additional Application Designer Subsessions 56

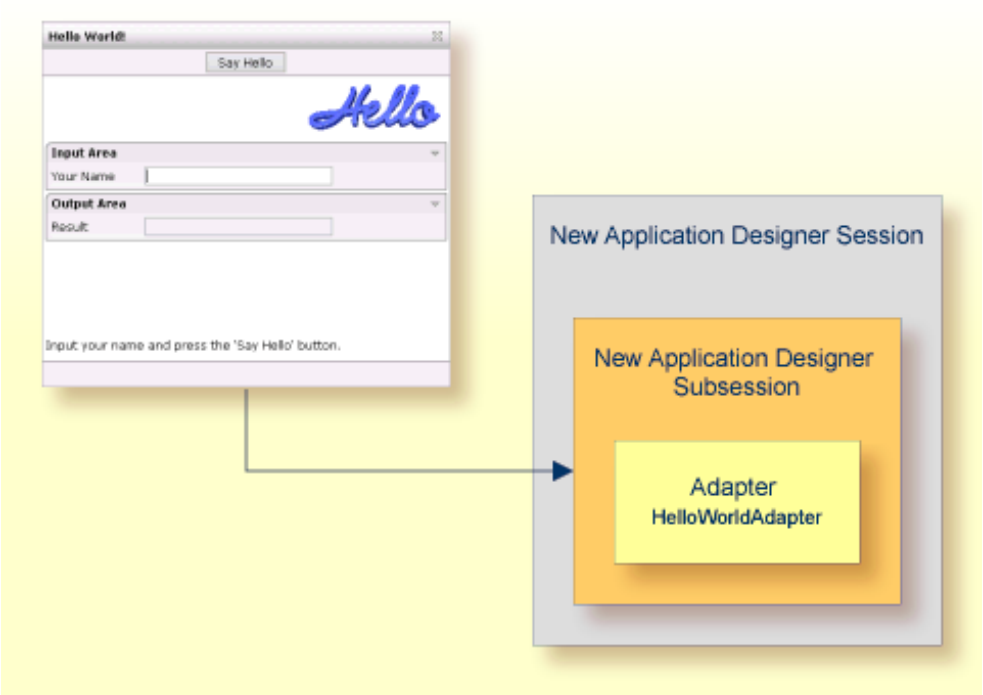
Starting an Application Designer Session

The proper start of a session is to open an Application Designer page via the StartCISPage servlet.

Example: If you start the "Hello World!" page with the following URL, a new Application Designer session object with a new session ID is automatically created on the server side:

```
http://localhost:51000/cis/servlet/StartCISPage?PAGEURL=/cisyourfirstproject/helloworld.html
```

The logical counter part of the page - the HelloWorldAdapter object - is opened inside a subsession that is automatically created inside the Application Designer session.



You see that inside one Application Designer session, there is always at least one subsession.

Starting Additional Application Designer Subsessions

You may use your pages in a mode in which you always work inside one Application Designer subsession - the one which was created during the `StartCISPage` procedure. But maybe you want to start additional subsessions.

There are two good reasons for starting additional subsessions:

- **Separate life cycles of activities**
  A subsession is keeping all adapter instances which play a role inside the processing of a certain activity. By closing the subsession, all adapter objects belonging to the subsession are released and can be caught by the garbage collector. In other words: a subsession is something like the life cycle manager for its contained adapters.

  Consequence: if you have multiple activities running in parallel, then each activity has its own life cycle, e.g. it can be closed individually without any consequence for the life cycle of the other activities.
- **Isolated activities**
  The adapter objects are created per subsession. This means: you can run one and the same activity in parallel - represented by two subsessions. In both subsessions, adapters are built up in parallel - completely isolated from one another.

  Programming applications inside “multi document interface”-like programs, (e.g. applications inside the Application Designer workplace) is therefore simple: each document (activity) is associated with its own subsession. The workplace just coordinates that the correct page is linked to the correct subsession at the appropriate point of time.

The starting of a new Application Designer subsession is done by opening a page inside a frame or inside an Application Designer subpage via Application Designer APIs.

Application Designer offers APIs (in class `com.softwareag.cis.server.Adapter`) to open Application Designer pages in a certain frame. These APIs always have one “simple” variant and one “complex” variant:

- **Simple Variant**

```
protected void openCISPageInTarget(String pageURL,
                                   String target)
```

  By calling this method, you open a certain page in a certain frame. The page is automatically linked to the subsession of the adapter calling this method.
- **Complex Variant:**

```
protected void openCISPageInTarget(String pageURL,
                                   String subsessionId,
                                   String target)
```

  By calling this method you open a certain page in a certain frame. But now you can explicitly pass a new `subsessionId` to be used for the page's adapter.

The proper call for a page which should belong to a new subsession is:

```
...
...

public void onOpenNewPage()
{
    // create new subsession id
    String newSSID = UniqueIdMgmt.createPseudoGUID();
    openCISPageInTarget("...URL...",newSSID,"...TARGET...");
}
...
...
```

15 How Things End

- End of an Application Designer Session 60
- End of an Application Designer Subsession 60
- End of an Application Designer Adapter 60

End of an Application Designer Session

A session normally ends if the page which was opened with the StartCISPage servlet is closed. This happens for example:

- if the user shuts down the browser,
- if the user loads a new page into the frame in which the StartCISPage servlet was called previously.

In other words: the session is normally kept alive as long as the user stays in the Application Designer environment.

Why "normally"? If a session is without user interaction for a long time, the session is timed out on the server side. When the user comes back to continue interaction, a corresponding message appears. The duration until a session is timed out is configurable; see the description of the *ciscon-fig.xml* file in the *Configuration* documentation for details.

If a session ends, all its subsessions and all adapters in the subsessions are automatically ended.

End of an Application Designer Subsession

A subsession is ended via an API. There are two APIs available:

- Via the interface `com.softwareag.cis.server.IInteractionManager`.
- Via the method `endProcess()` which your adapters inherit from the `Adapter` class.

For further information, see the JavaDoc documentation.

End of an Application Designer Adapter

Adapters typically stay alive until the subsession ends in which they are living. There is also an API available to directly end adapters:

- Method `Adapter.markThisAdapterForDestroy()`.
- Via the interface `IInteractionProcess` which you receive inside an adapter via `this.m_interactionProcess`.

16 Workplace Management

After reading the previous sections, you may now see in a better way what the task of a workplace management inside Application Designer is: a workplace is an application on its own having the task to administer content applications both from the graphical and the session management point of view.

The workplace management is responsible for the proper assignment of subsessions to activities. The life cycle of subsessions is typically controlled by the workplace.

17 Saving Context Data

- Different Levels of Context 64
- Accessing the Context 64
- Typical Usage Scenarios 65

Sometimes it is useful to save context data centrally inside a session context and to use these data like a session-global variable. You should be very restrictive with this option - otherwise you may end up in a scenario in which any kind of data exchange is done by the context.

Different Levels of Context

The session management allows you to hold context information at two levels:

- **Session Context**
  Within the session context, save data that you want to access from everywhere inside your adapters.
- **Subsession Context**
  Within the subsession context, save data which you want to access from everywhere inside a subsession.

  Two different subsessions have also two different subsession contexts, i.e. the saved data are kept independent per subsession.

Accessing the Context

You obtain the context(s) by calling methods which are inherited from the `Adapter` class:

- `findSessionContext()`
  Returns a context which is held for each session.
- `findSubSessionContext()`
  Returns a context which is held for each subsession.

Both methods return a `com.softwareag.cis.context.ILookupContext` interface. This interface offers the possibility to bind and look up any objects.

```
public interface ILookupContext
{
    public Object lookup(String s, boolean reactWithErrorIfNotExist);
    public void bind(String s, Object o);
    public void releaseAllReferences();
}
```

When binding objects to a context, use a naming convention that is similar to the naming of your Java classes to avoid naming conflicts. Example:

```
...
findSessionContext.bind("com/yourcompany/application/parameter");
...
```

The context can be cleaned up by the `releaseAllReferences()` method. It is integrated into Application Designer's session management.

Typical Usage Scenarios

Examples of typical data that you save at the session context level:

- Name of the user who is currently logged on.
- Name of the system to which the user is currently logged on.
- Language in which the user is logged on.

Examples of typical data that you save at the subsession context level:

- ID of the object you are processing.
- Temporary data you want to pass from one page to another.

18 Session IDs

Each session - session or subsession - holds an ID.

- **Session**
  The ID of the session is unique inside one instance of Application Designer. If you have two Application Designer installations running, the same ID may be used inside both servers.
- **Subsession**
  The ID of a subsession is unique inside one session. If you have multiple sessions running inside one Application Designer instance, the same subsession ID may be used in two sessions.

You can access the IDs from your adapter in the following way:

```
public class TestAdapter
    extends com.softwareag.cis.server.Adapter
{
    ...
    ...
    public void xxx()
    {
        ...
        String sessionId = this.m_interactionProcess.getSessionId();
        String subsessionId = this.m_interactionProcess.getProcessId();
        ...
    }
    ...
    ...
}
```

III Becoming a Member of the Startup Process

There may be the demand to become a member of the startup process of Application Designer: for example, in some cases you have an application which is accessed by Application Designer - by corresponding adapter classes. Typically, you have to initialise this application, for example, by setting up some database connection.

This initialisation takes time and should be done on startup of Application Designer - instead of the first time a user interacts with the application.

The information provided in this part is organized under the following headings:

Overview
Startup Class
Registration

19 Overview

It is quite easy to integrate your application inside Application Designer at startup time. You have to

- provide a startup class supporting the interface `com.softwareag.cis.server.IServletInitHandler`,
- register this class by editing a configuration file inside Application Designer.

20 Startup Class

The following code shows a simple Java class that can be registered inside the startup process of Application Designer:

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

import javax.servlet.*;
import com.softwareag.cis.server.*;

public class StartDemo
    implements IServletInitHandler
{

    public void init(ServletConfig conf)
    {
        System.out.println("StartDemo: started!");
        System.out.println("StartDemo: started!");
        System.out.println("StartDemo: started!");
        System.out.println("StartDemo: started!");
        System.out.println("StartDemo: started!");
    }

    public void destroy()
    {
        System.out.println("StartDemo: destroyed!");
        System.out.println("StartDemo: destroyed!");
        System.out.println("StartDemo: destroyed!");
        System.out.println("StartDemo: destroyed!");
        System.out.println("StartDemo: destroyed!");
    }

}
```

It supports the interface `com.softwareag.cis.server.IServletInitHandler` that requires the implementation of the methods `init` and `destroy`. The `init` method takes the servlet configuration

as parameter with which the Application Designer's servlet itself is initialised. See the documentation of the servlet functions (e.g. in the reference documentation for the servlet API) for more details.

21 Registration

This class must be registered in the */config/statapps.xml* configuration file to be integrated into the startup process of Application Designer. The file looks as follows:

```
<startapps>
  <start class="com.softwareag.cis.test.StartDemo"/>
</startapps>
```

Just add a new "start" line and specify the class name. The class must be accessible during runtime.

IV Adapting the Look & Feel

One of the guiding principles of Application Designer is to provide high-quality controls by simply specifying tags inside a layout definition. Each tag is rendered when generating the intelligent HTML page into various HTML and JavaScript statements. The HTML statements contain the specification of the display style of each control. For example, a label is rendered into a table cell having a defined background (typically a bottom line), a defined text size, etc.

This part describes how to modify the default rendering with the help of style sheets in order to adapt the look and feel to your needs.

The information provided in this part is organized under the following headings:

Introduction

Style Sheet File

Writing a New Style Sheet File

Selecting the Right Style Sheet

Dynamic Selection of the Style Sheet File

Static Selection of the Style Sheet File

22 Introduction

There are different possibilities for adapting the look and feel - depending on what you want to do:

1. Overwrite the style definition in individual controls by specifying the `style` property. Offered for all controls holding text information inside (label, button, field, etc.) and for all container controls (areas, tables, rows, etc.).
2. Exchange the central style sheet file containing all style information for controls. Furthermore, specify your own style sheet: define a style sheet file for a page statically or switch between style sheets dynamically (e.g. user-dependent).
3. Create new controls by yourself and place them into the Application Designer design and runtime environment.

Option 1 is typically used if you like the default style provided by Application Designer - but you want to change it for some pages. For example, you want the text of the button to appear in red - instead of black for some buttons.

Option 2 is typically used if you have to adapt the style of the controls to some customer-specific style. For example, if you want to change the font "Verdana" that is used inside the Application Designer style, or if you want to introduce a new color scheme. Option 2 does not require any changes inside the page layout definitions - the style is completely separated from the layout. You do not have to regenerate your XML definitions at all.

Option 3 is used if you need new controls. There is an open API that allows you to add your own controls in a simple way.

Option 1 is discussed in *Working with Controls* (in the *Java Page Layout* documentation). Option 3 is explained in the *Java Custom Controls* documentation. This part focuses on option 2 - exchanging the style sheet.

23 Style Sheet File

The style information of all controls is defined in the file *<your-webapplication>/cis/styles/CIS_DEFAULT.css*. The style information is sorted alphabetically. Omit the prefix "ROW" or "COL" for container controls - e.g. you find the style information of the "ROWAREA" in "AREA".

```
.AREATable
{
    font-size: 10pt;
    border-width: 0;
    background-color: #E0D8C8;
    border: 1 solid #808080
}
.AREATitleCell
{
    font-size: 8pt;
    color: #808080;
    background-color: #00006C
}
.AREALeftFromTitleCell
{
    font-size: 8pt;
    color: #808080;
    background-color: #00006C
}
.AREARightFromTitleCell
{
    font-size: 8pt;
    color: #808080;
    background-color: #00006C
}
.AREALinks
{
    color: #FFFFFF;
    text-decoration: none
}
```

Take further information out of the comments describing when which style class used.

24 Writing a New Style Sheet File

Style sheet files should be created and maintained with the Style Sheet Editor. This tool covers style sheet manipulation on a very low level. Maintaining style sheets with the Style Sheet Editor means that all information that you enter is kept separate from the style sheet itself.

From release to release, Application Designer adds new controls to its control library. As a consequence, the style sheet template is typically enhanced with every new control. When you work with the Style Sheet Editor, this is done automatically. You just have to regenerate your own style sheet file.

Otherwise (if you have manually created your own style sheet file), you always have to have to embed the enhancements into your style sheet file when Application Designer does style sheet changes: you have to copy the additional Application Designer style classes from the standard Application Designer style sheet file (*CIS_DEFAULT.css*) into your own style sheet file. Use a diff-viewer/diff-editor to do this.

25 Selecting the Right Style Sheet

An intelligent HTML page (generated inside Application Designer) links to a style sheet file. The selection of the style sheet file is done in the following way:

- **Dynamic selection (default):**
  The name of the style sheet file is determined by a property `style` of your adapter class. If this is not specified, the default Application Designer style sheet is chosen. The `style` property is provided automatically. See ***Dynamic Selection of the Style Sheet File*** for further information.
- **Static selection:**
  The name of the style sheet file is defined in the page by specifying the `stylesheetfile` property of the "page" tag. See ***Static Selection of the Style Sheet File***.

Static selection takes precedence over dynamic selection, i.e. if static selection is defined, dynamic selection is not taken into consideration anymore.

Typically, you define the style sheet file name statically only for certain pages: for those pages you want to be sure that they do not differ from the defined look and feel.

26 Dynamic Selection of the Style Sheet File

- What You Can Do 88
- Example 88

The style sheet file is determined by your adapter:

- There is a property style with its corresponding `getStyle()` method implemented in the inherited class `com.softwareag.cis.server.Adapter`. The `style` property returns the URL of the used style sheet file.
- The `Adapter` class derives the URL of the style sheet file from the Application Designer session context. Access the Application Designer session context by the protected property `m_sessionContext`. The `m_sessionContext` object provides a `setStyle()` and `getStyle()` method. To change the style sheet file inside the adapter, do the following:

```
public void ...()
{
    ...
    m_sessionContext.setStyle("...yourStyleURL... ");
    ...
}
```

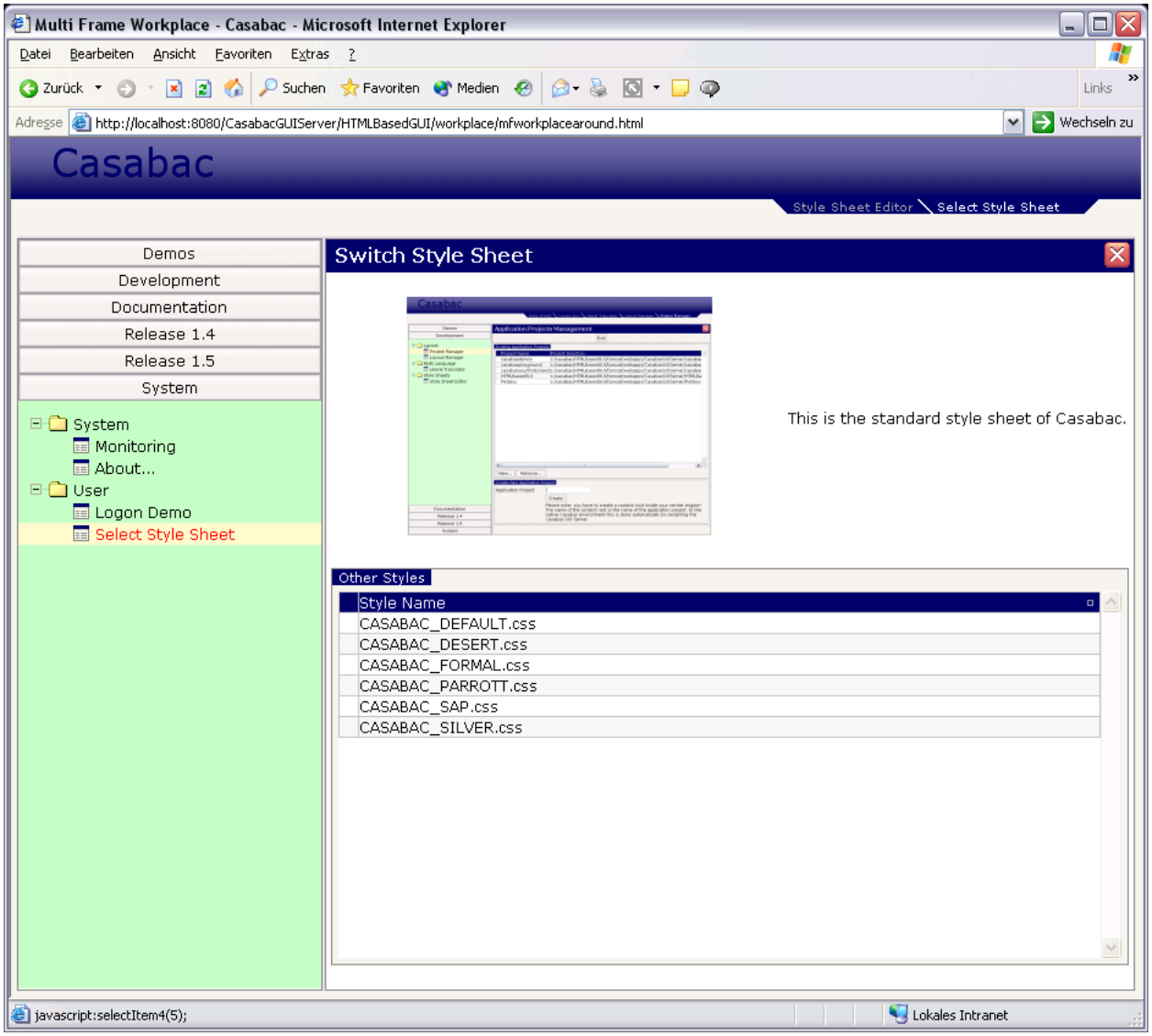
What You Can Do

There are two options that you can use in parallel:

- You can take over the `getStyle()` method in your adapter from the `Adapter` class. In this case, you can set the session's style sheet via `m_sessionContext.setStyle(...)`, as described.
- You can write your own `getStyle()` method and can apply any other rule you might think of on your own.

Example

Inside the Application Designer demo workplace, there is a function to select a style sheet for your current session:



The program lists all available style sheets in the directory *<webapp>/styles/*. If you select one style sheet file, then the selected style sheet is internally passed to the session context as described in the previous section.

Consequently, all pages in the content area of the workplace will be rendered with this style sheet.

The style of the workplace itself will not change: the workplace adapter overwrites the `getStyle()` method: with the workplace, you can pass its style sheet file when dynamically defining the workplace.

27 Static Selection of the Style Sheet File

It makes sense for some pages to define the style sheet file statically. In this case, it cannot be changed dynamically. This can be done inside the XML layout definition of the page with the "page" tag.

```
<page model="xyz"
      pagename="xyz.html"
      stylesheetfile="/HTMLBasedGUI/general/layout.css">
    ...
    ...
    ...
</page>
```

V Multi-Language Management in Java Applications

28 Multi-Language Management in Java Applications

- Defining the Language at Runtime 96
- Dealing with Literals inside Your Adapter 96

This chapter describes Java-specific customizations of multi-language management.

Detailed information on the multi-language management is provided in *Multi-Language Management*

Defining the Language at Runtime

With the protected member `m_sessionContext`, you find or set the currently active language used by the multi-language management:

```
...
m_sessionContext.setLanguage("de");
...
```

The value passed to the session context is only valid within the context of current session. Therefore, different users can be logged on to the system choosing different languages.

The string, which is passed to the `setLanguage()` method of `m_sessionContext`, represents the name of the directory in which the CSV files are stored. You are not bound to the "de" and "en" directories; you can add any other directories representing additional languages.

Dealing with Literals inside Your Adapter

Use method `replaceLiteral` of the inherited `Adapter` class to replace messages:

```
...
this.outputMessage("S",replaceLiteral("APP1","successFileSaved"));
...
```

The first parameter is an abbreviation - the file name of the multi-language file. The second parameter is the text ID that should be translated into text as described previously.

It is also possible to pass parameters of your application to the multi-language management. For example, if you want to show a success message which informs that file "xyz" was saved, proceed as follows:

```
...
String fileName = "xyz";
this.outputMessage("S",replaceLiteral("APP1","successFileSaved",fileName));
...
```

The corresponding line in the CSV file (*APP1.csv*) in the *\en* directory for English looks like:

```
...
successFileSave;File &1 was saved successfully
...
```

The "&1" is automatically replaced with the file name. There are other variants of the `replaceLiteral()` method available to pass 2 or 3 parameters. In this case, use &1, &2 and &3 in the text definition.

VI Online Help Management

The Application Designer environment provides for a simple but very useful online help management. You can easily plug help information behind controls or pages.

The information provided in this part is organized under the following headings:

Basics

Customizing the Online Help Pop-up

Other URL Rules?

Other Types of F1-Online Help?

29 Basics

- Supported Controls 102
- Way from Control to Online Help Page 102
- Content of HTML Page 104
- Where to Put the HTML Help Files 105
- HELPICON Properties 105

Supported Controls

Online help is accessible from the following controls:

- TITLEBAR (online help for a whole page)
- FIELD
- CHECKBOX
- RADIOBUTTON
- COMBOFIX
- COMBODYN2

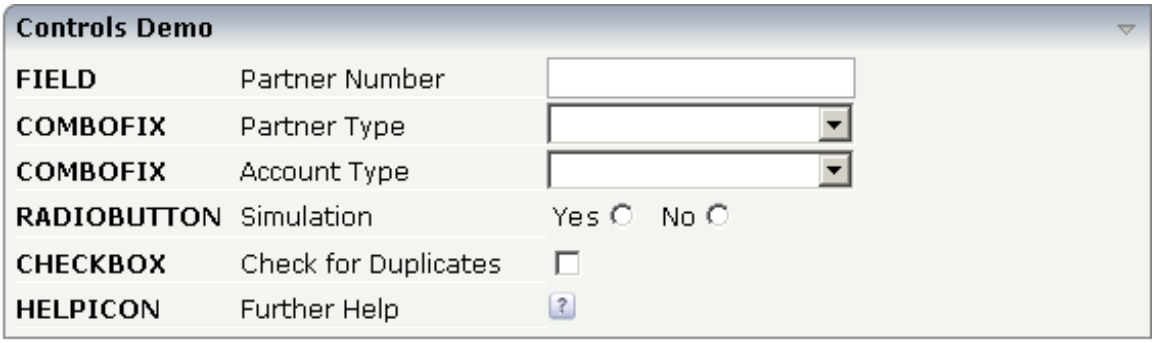
In addition, you can place special HELPICON controls at any point of your page.

Online help is either activated by pressing `F1` inside the controls or by clicking on the corresponding icon.

Way from Control to Online Help Page

Each control that supports online help offers a property `helpid`. In this property, you define an ID that is used for building a URL. This points to the page which appears when invoking the online help for the control.

Let us have a look at the following page:



Inside the XML layout definition, you see the `helpid` definitions:

```
<rowarea name="Controls Demo">
    <itr>
        <label name="FIELD" width="110" labelstyle="font-weight:bold">
        </label>
        <label name="Partner Number" width="150">
        </label>
        <field valueprop="businessPartnerNumber" width="150" ↩
helpid="BusinessPartnerInput">
        </field>
    </itr>
    <itr>
        <label name="COMBOFIX" width="110" labelstyle="font-weight:bold">
        </label>
        <label name="Partner Type" width="150">
        </label>
       <combofix valueprop="partnerType" width="150" size="1" helpid="PartnerType">
            <combooption name="Private" value="private">
            </combooption>
            <combooption name="Business" value="business">
            </combooption>
            <combooption name="Other" value="other">
            </combooption>
        </combofix>
    </itr>
    <itr>
        <label name="COMBOFIX" width="110" labelstyle="font-weight:bold">
        </label>
        <label name="Account Type" width="150">
        </label>
       <combodyn valueprop="accountType" optarrayprop="accountTypeOptions" size="1"
                  width="150" helpid="AccountType">
        </combodyn>
    </itr>
    <itr>
        <label name="RADIOBUTTON" width="110" labelstyle="font-weight:bold">
        </label>
        <label name="Simulation" width="150">
        </label>
        <hdist>
        </hdist>
        <label name="Yes" asplaintext="true">
        </label>
        <radiobutton valueprop="simulation" value="on" helpid="Simulation">
        </radiobutton>
        <hdist width="10">
        </hdist>
        <label name="No" asplaintext="true">
        </label>
        <radiobutton valueprop="simulation" value="off" helpid="Simulation">
        </radiobutton>
        <hdist>
        </hdist>
```

```
    </itr>
    <itr>
        <label name="CHECKBOX" width="110" labelstyle="font-weight:bold">
        </label>
        <label name="Check for Duplicates" width="150">
        </label>
        <checkbox valueprop="checkDuplicates" helpid="CheckForDuplicates">
        </checkbox>
    </itr>
    <itr>
        <label name="HELPICON" width="110" labelstyle="font-weight:bold">
        </label>
        <label name="Further Help" width="150">
        </label>
        <helpicon helpid="FurtherHelp">
        </helpicon>
        <hdist>
        </hdist>
    </itr>
</rowarea>
```

The default way how a URL is derived out of a help ID is:

```
URL = /<name of web application> +
      /<application project> +
      /help +
      /<language> +
      /<helpid>.html
```

Example: in the standard Application Designer installation, in the project "cisdemos", being logged on in English, the help ID "AccountType" is transferred to:

```
/cis/cisdemos/help/en/AccountType.html
```

Content of HTML Page

The content of the HTML page which is called is completely up to you.

Where to Put the HTML Help Files

Consequently, the files containing the online help are located inside the following directory:

```
<web application directory>/
  <application project>
    help/
      <language>/
        <helpid>.html
```

HELPICON Properties

Basic			
helpid	Help id that is passed to the online help management when the user clicks onto the icon.	Optional	
iconurl	URL of image that is displayed inside the control. Any image type (.gif, .jpg, ...) that your browser does understand is valid. Use the following options to specify the URL: (A) Define the URL relative to your page. Your page is generated directly into your project's folder. specifying "images/xyz.gif" will point into a directory parallel to your page. Specifying "../HTMLBasedGUI/images/new.gif" will point to an image of a neighbour project. (B) Define a complete URL, like "http://www.softwareag.com/images/logo.gif".	Optional	
title	Text that is shown as tooltip for the control. Either specify the text "hard" by using this TITLE property - or use the TITLETEXTID in order to define a language dependent literal.	Optional	
titletextid	Text ID that is passed to the multi language management - representing the tooltip text that is used for the control.	Optional	
visibleprop	Name of an adapter property that provides the information if this control is displayed or not. As consequence you can control the visibility of the control dynamically. The server side property needs to be of type "boolean".	Optional	
visibleprop	Name of the adapter parameter that provides the information if this control is displayed or not. As consequence you can control the visibility of the control dynamically.	Optional	

withdistance	If set to "true" then 2 pixels of distance are kept on the left and on the right of the icon. Reason behind: if arranging several icons inside one table row (ITR, TR) then a certain distance is kept between the icons when this property is set to "true".	Optional	true false
align	Horizontal alignment of control in its column. Each control is "packaged" into a column. The column itself is part of a row (e.g. ITR or TR). Sometimes the size of the column is bigger than the size of the control itself. In this case the "align" property specifies the position of the control inside the column. In most cases you do not require the align control to be explicitly defined because the size of the column around the controls exactly is sized in the same way as the contained control. If you want to directly control the alignment of text: in most text based controls there is an explicit property "textalign" in which you align the control's contained text.	Optional	left center right
valign	Vertical alignment of control in its column. Each control is "packaged" into a column. The column itself is part of a row (e.g. ITR or TR). Sometimes the size of the column is bigger than the size of the control. In this case the "align" property specify the position of the control inside the column.	Optional	top middle bottom
colspan	Column spanning of control. If you use TR table rows then you may sometimes want to control the number of columns your control occupies. By default it is "1" - but you may want to define the control to span over more than one columns. The property only makes sense in table rows that are synchronized within one container (i.e. TR, STR table rows). It does not make sense in ITR rows, because these rows are explicitly not synched.	Optional	1 2 3 4 5 50 int-value
rowspan	Row spanning of control. If you use TR table rows then you may sometimes want to control the number of rows your control occupies. By default it is "1" - but you may want to define the control to span over more than one columns. The property only makes sense in table rows that are synchronized within one container (i.e. TR, STR table rows). It does not make sense in ITR rows, because these rows are explicitly not synched.	Optional	1 2 3 4 5 50 int-value

comment	Comment without any effect on rendering and behaviour. The comment is shown in the layout editor's tree view.	Optional	

30 Customizing the Online Help Pop-up

- Creating a Project-Specific Pop-up 111
- Runtime Behavior 111

The HTML page that is shown when you press F1 or when you click on the HELPICON control is application-specific. Application Designer integrates this HTML page as a subpage into a corresponding pop-up. By default, a fixed pop-up is used for all Application Designer projects. For example:

The application-specific HTML page is used as the content of this pop-up. However, the pop-up itself has always a fixed look-and-feel and a fixed size.

A fixed pop-up size is not always the best match for all applications; and some applications might want to modify the appearance of the pop-up, for example, by adding an image. Therefore, it is also possible to use a project-specific pop-up instead of the default pop-up. This is explained in the topics below.

Creating a Project-Specific Pop-up

Each online help pop-up must have the name *popuponlinehelp.xml*. By default, the *popuponlinehelp.xml* file of the project **HTMLBasedGUI** is used.

To create a *popuponlinehelp.xml* file in your project

1. In the navigation frame of the development workplace, choose the button which represents your project.
2. Choose the **New Layout** command in the navigation frame.
3. In the **Name** text box of the resulting dialog, enter the name "popuponlinehelp.xml". A different name will not be accepted when you create a template for an online help pop-up.
4. Choose the layout template which is named "Online Help Pop-up".

   A new *popuponlinehelp.xml* file is created in the current project.

   The corresponding layout is shown in the Layout Painter. You can now customize your pop-up: you can modify all design-time properties. However, all bindings such as the `model` property of the page and the `valueprop` property of the subpage must not be modified.
5. To define a specific size, modify the pop-up properties `popupwidth` and `popupheight`.

   You can also add images (for example, a company logo) to the pop-up.

Runtime Behavior

Application Designer first checks whether a layout with the name "popuponlinehelp" exists in the application project which is currently executed. If such a layout is found, it will be used.

If a layout with the name "popuponlinehelp" is not found in the application project, the default "popuponlinehelp" layout of the project **HTMLBasedGUI** will be used.

31 Other URL Rules?

The rules defining how to build a URL based on a help ID are kept behind an interface:

```
public interface IOHManager
{
    /** returns the URL for the page to be opened. */
    public String getOnlineHelpURL(String project,
                                   String page,
                                   String helpId,
                                   String language);
}
```

You can write your own implementation of this interface in which you apply your rules. The result must be a valid URL which is opened inside a pop-up.

For more details, see the Java API documentation for the package `com.softwareag.cis.onlinehelp`.

32 Other Types of F1-Online Help?

Maybe the standard way of offering online help - displaying a modal pop-up containing corresponding text - is not the one that you want to use for your application. Maybe you want to open the help in a certain frame of a frameset definition or maybe you want to pass context information of your current adapter into the help system.

In this case, you can use the interface `com.softwareag.cis.onlinehelp.IExtendedOHManager`:

```
public interface IExtendedOHManager
{
    public void processOnlineHelpRequest(Adapter requestingAdapter,
                                         String project,
                                         String page,
                                         String helpId,
                                         String language);
}
```

See the Java API documentation for more information.

VII Appendices

The following appendices are available:

Appendix A - Call Sequence for Adapter	Describes how an incoming request by the browser client is processed inside an adapter.
Appendix B - Usage of Methods Inherited from the Adapter Class	Gives information about adapter classes and how to use them.
Appendix C - Data Types to be Used by Adapter Properties	Describes the various data types that can be used by adapter properties.
Appendix D - Class Loader Concepts	Gives information about class loader management.
Appendix E - StartCISPage Servlet	Describes the StartCISPage servlet that is used to open intelligent HTML pages.

33 Appendix A - Call Sequence for Adapter

- Normal Call Sequence 120
- Call Sequence when a Subsession is Destroyed 121
- Call Sequence when a Session is Destroyed 122
- Error/ Runtime Exceptions 122
- Pay Attention when Overwriting 122

This chapter describes how an incoming request by the browser client is processed inside an adapter. The request contains all the changes of properties that have been made at client side.

Normal Call Sequence

- `init()`
  This method is called only once - when creating the adapter inside a subsession. Before calling this method, Application Designer makes sure that the adapter instance is properly registered inside the Application Designer environment. Therefore - for example - you have access to the session management: use the `findSessionContext()` or `findSubSessionContext()` method in order to look for some values inside the `init()` method. It is not possible to use the `find...SessionContext()` methods inside the constructor of an adapter - since the session is not yet assigned to the adapter instance.

  When navigating between pages (using the `switchToPage()` or `openPopupPage()` method), the corresponding adapter objects are only created once. For example, if you navigate from page "A" to page "B" and back to page "A", the adapter of page "A" does not change. The `init()` method is only called once - at the time the adapter is instanciated.
- `activate(...)`
  This method is implemented by the `Adapter` class already. You only need to overwrite this method if you want to passivate the state between requests. In this case, you can activate this state inside your implemented method of your adapter class. If you use the adapter class to co-operate, for example, with components running in a container of an application server, you should synchronize the state passivation with the container's passivation.
- `reactOnDataTransferStart()`
  This method is called when the transfer of the changed properties starts. You can initialize some internal members at this time. If you overwrite this method, do not forget to include the method of the super-class (`Adapter.reactOnDataTransferStart()`) into your method implementation!
- `setXxx(), setYyy(),` **...**
  Now, the set methods of the changed properties of the browser client are transferred. It is very important that your implemented set methods never cause an exception or an error.
- `reactOnDataTransferEnd()`
  This method is called after setting the changed properties. Use this method to perform operations you always want to execute when processing a request.
- `invoke()`**-Method**
  If the request has a method call inside, the method is invoked now.
- `processAsDefault()`
  If the request has no method call, this standard method is called.

- `reactOnDataCollectionStart()`
  This method is called when the transfer of adapter properties starts. Use this method, for example, for performance improvements during the following get methods, for example, by building temporary objects.
- `getXxx(),getYyy()`
  All get methods of the adapter - including array elements which may be passed back by - are called.
- `reactOnDataColletionEnd()`
  This method is called when data collection is finished. Temporary objects - which you may have created for performance reasons - can be released for garbage collection now.
- `passivate(...)`
  This method is the counterpart of the activate method.

Call Sequence when a Subsession is Destroyed

`endProcess()`

This method is called inside the adapter if the user decides to terminate the subsession. For example, in the Application Designer workplace environment, this method is called whenever the user chooses the close button of a page.

You can deny closing a subsession in your implemented method:

```
public class ABCAdapter
    extends com.softwareag.cis.server.Adapter
{
    ...
    ...
    public void endProcess()
    {
        // veto the endProcess in case of unsaved data
        if (changedDataNotSaved == true)
        {
            this.outputMessage("E","Please save data first");
            return;
        }
        // close subsession
        super.endProcess();
    }
}
```

Call Sequence when a Session is Destroyed

If a session is removed from Application Designer - for example, if the user closes the browser or if a system administrator removes the session - the adapter instances are informed in the following way:

```
destroy()
```

In your implemented method, clean up all resources bound to your adapter instance. You cannot deny the destroying of the session - but you can react.

Error/ Runtime Exceptions

Error and runtime exceptions occurring during the adapter request processing may be handled centrally inside your adapter. For more details, see ***Binding between Page and Adapter***.

Pay Attention when Overwriting

The methods named above are already implemented with default behavior inside the class `com.softwareag.cis.server.Adapter`. Pay attention when overwriting these methods inside your adapter and always include the super-class's processing into your own implementation. The first statement inside your implementation should call the super-class method:

```
public class ABCAdapter
    extends com.softwareag.cis.server.Adapter
{
    ...
    ...
    public void reactOnDataTransferStart()
    {
        super.reactOnDataTransferStart();
        // now own implementation
        ...
        ...
    }
}
```

34 Appendix B - Usage of Methods Inherited from the Adapter Class

- Access to Lookup Session Context 124
- Access to Application Designer Session Context 125
- Access to other Adapters 125
- Error Output 125
- Page Navigation 126
- Opening of Pop-up Dialogs 126
- Frame Communication 126
- Closing of a Page 127
- Multi Language Management 127

Inside the Application Designer management, adapters have to provide a defined interface to be managed correctly by the system. This interface is declared by `com.softwareag.cis.Server.IAdapter`. In order to have a high level of comfort during developing adapters, you should derive your adapter classes from the super-class `com.softwareag.cis.Server.Adapter`. This class already provides some useful methods.

Access to Lookup Session Context

As you know, session management defines sessions (corresponding to one browser instance) and subsessions (corresponding to one process inside the Application Designer workplace). There is the possibility to bind and look for parameters on both levels:

- `Adapter.findSessionContext()` - returns the context which is on top of all subsessions. All adapters inside one session refer to the same session context.
- `Adapter.findSubSessionContext()` - returns the context which is held per subsession. Only adapters - belonging to the same subsession - share this context.

The result is a context supporting the interface `com.softwareag.cis.context.ILookupContext`. This interface provides two important methods:

```
public Object lookup(String s, boolean reactWithErrorIfNotExist);
public void bind(String s, Object o);
```

The session context is used, for example, to refer to the current user who is logged in, the chosen language, etc. The subsession context is used to share data inside a subsession.

Do not use the context as global variable buffers in a very intensive way. It will end up in programs relying on a lot of context information to be available - and sooner or later no one knows what has to be in the context when starting the program.

Via the methods

- `Adapter.findSessionId()`
- `Adapter.findSubsessionId()`

you can access the internally used representations of session ID and subsession ID.

Access to Application Designer Session Context

Application Designer uses its own lookup session management in order to store information of a session. You can access and manipulate this information by calling your adapter's method:

- `Adapter.findCISessionContext()` - returns a concrete session context object.

Inside the session context, the following parameters are kept:

- date format
- time format
- language
- style
- decimal separator
- and other information.

Have a look at the JavaDoc API documentation for more details.

Access to other Adapters

Access other adapters inside the same subsession by the methods:

- `Adapter.findAdapter(class)` - returns the adapter instance for a given class. Method `init()` is already called when passing back the instance - but only if the adapter was not used before.

Use this method before navigating between pages in order to prepare the adapter that will be used by the next page.

Error Output

You can display error messages inside the status bar (if it is defined in the page layout) by using the methods:

- `outputMessage(String, String (, String))`

First, pass a string for the type of message. This is needed to display a corresponding icon inside the status bar. There are constants defined inside the Adapter for specifying the type:

- `Adapter.MT_ERROR`

- `Adapter.MT_WARNING`
- `Adapter.MT_SUCCESS`

The second string is the message being shown.

The third string - which is optional - is the long text description of the message. It becomes visible by a dialog if the user clicks with the mouse on the message. If you do not specify a long description, the normal message is used.

Page Navigation

Navigate to a page by using the method:

- `switchToPage(String pageName)`

The "pageName" is the URL - either relative or absolute - of the next page.

Opening of Pop-up Dialogs

You can open a page inside a pop-up dialog by using the method:

- `openPopup(String pageName)`.

The "pageName" is the URL - either relative or absolute - of the page that is displayed inside the dialog.

You can specify pop-up parameters of the pop-up you open with `openPopup()` by using the methods:

- `setPopupTitle(String title)`
- `setPopupPageFeatures(String pageFeatures)`

Frame Communication

There are various methods to communicate to other frames:

- `openPageInTarget`
- `openCISPageInTarget`
- `invokeMethodInTarget`

- `refreshTarget`
- `sizeTarget`

Closing of a Page

The default method used for closing a page is `endProcess()`. It is provided by the `Adapter` class. The tasks performed by the `endProcess()` method are:

- The current subsession is closed and de-registered inside the session management.
- The current page is de-registered from the workplace management - if it was registered before.

Calling the `endProcess()` method ensures that all memory resources are released for the corresponding subsession.

The `endProcess()` method is called by clicking inside the page on the close icon at the top right corner of the page. You can also call it directly inside an adapter, e.g. if you want to close the subsession as reaction to the user's entered data.

Multi Language Management

You can access the multi language management using the methods:

- `replaceLiteral(String application, String textid)`
- `replaceLiteral(String application, String textid, String param1)`
- `replaceLiteral(String application, String textid, String param1, String param2)`
- `replaceLiteral(String application, String textid, String param1, String param2, String param3)`

The application is the name for the abbreviation of a defined application area for which literals are defined. In the file-based multi language management, it represents the name of a CSV file that holds the text identified by a text ID.

35 Appendix C - Data Types to be Used by Adapter Properties

- Supported Data Types 130
- Data Types for Managing Date and Time 130

The Application Designer management is very flexible by allowing various data types for properties of an adapter.

Supported Data Types

- String
- int, long, short, byte
- float, double
- BigDecimal
- boolean
- CDate
- CTime
- CTimeStamp

Data Types for Managing Date and Time

The `java.util.Time` class is very powerful, but also very complex to use for business applications. Therefore, three classes are introduced to deal with date and time:

- `com.softwareag.cis.util.CDate`
- `com.softwareag.cis.util.CTime`
- `com.softwareag.cis.util.CTimeStamp`

See the JavaDoc documentation for further details.

Dates and times are transferred as strings between Application Designer and the intelligent HTML page:

- YYYYMMDD format for dates.
- HHMMSS format for times.
- YYYYMMDDHHMMSSMMM format for timestamps.

The interpretation and formatting of these strings to valid formats is done automatically.

36 Appendix D - Class Loader Concepts

- Design Time - Runtime 132
- Class Loader Hierarchy 132
- Preparing for Runtime 135

An explicit class loader management was introduced to support the following scenarios:

- Classes are automatically found in the context of Application Designer without specifying a `CLASSPATH` variable.
- Classes can be stored inside an application project directory - separated from other application projects.
- During development time, easily run new pages together with the latest classes without restarting the server.

This chapter explains the class loader concepts used inside Application Designer.

Design Time - Runtime

The class loader concepts are designed to simplify the development of pages and their logical representations on the server side: adapters.

At runtime, they should only be used if you are not running in a cluster - i.e. if you do not distribute your application server on multiple nodes. When running in a cluster, classes should be located exactly there, where the application server specifications allow them to be located. Inside the Application Designer configuration, you can select which mode you are running in - for details, see *Design Time Mode and Runtime Mode* in the *Configuration and Administration* documentation.

After explaining the class loader concepts in this chapter, at the end we explain what to do in order to change a design time environment into a runtime environment.

Class Loader Hierarchy

Application Designer runs as a web application inside a servlet engine - by default, the Tomcat servlet engine is used. The class loader used by the servlet engine is called “web application loader” in the following text.

The Application Designer environment itself is running in the context of the web application loader. This class loader is looking for classes as specified by the servlet engine. Therefore the Application Designer runtime must be accessable by this class loader. For Tomcat, this is achieved by placing the *cis.jar* file inside the *<installdir>/tomcat/webapps/ROOT/WEB-INF/lib* directory.

The following topics are covered below:

- Application Class Loader
- Initialisation of Your Application
- Guidelines for Development
- Classpath Extensions in cisconfig.xml

- Loading Resource Files

Application Class Loader

The application classes (adapter classes) are loaded by the class loader management of Application Designer. This class loader looks for Java classes as follows:

- All *.class* files inside the directory:

  <webapp>/softwareag/appclasses/classes
- All *.jar* files inside the directory:

  <webapp>/softwareag/appclasses/lib
- All *.class* files inside any application project under the directory:

  <webapp>/<project>/appclasses/classes
- All *.jar* files inside any application project under the directory:

  /<webapp>/<project>/appclasses/lib
- All classes that are referenced in the classpath extension that can be defined in the Application Designer configuration (*cisconfig.xml*).

Unlike normal class loader hierarchies, the application class loader always tries to resolve a class inside its application directories first. Only if the class is not found, the parent class loader is called - the web application loader. The benefit is that application classes are totally separated from the servlet engine classes - e.g. by using XML parser libraries. You are not bound to the parser delivered with the servlet engine.

Inside the Application Designer session management, a session is bound to an application class loader instance. Therefore the application class loader - which was instanciated when the session was created - is kept in the session during its whole life cycle. All objects created inside this session use this instance of the class loader.

In case of changing classes inside the *softwareag/appclasses* or the corresponding application-project subdirectories, you can force to create a new class loader used in all sessions which are created afterwards. This means, that you can upgrade your system without disturbing running sessions. Old sessions are still using their old classes; new sessions are using new classes.

By choosing the button **Use latest Version of Applications for new Sessions**, a new class loader instance is generated.

A new class loader instance can also be created during development inside the Layout Painter. See also the "Hello World!" example in the *First Steps* and its section *If you Change the Adapter*.

Initialisation of Your Application

Every time a new instance of a class loader generated, the initialisation process of your application is also performed. This guarantees that, for example, all static variables you may use internally can be correctly initialised by your initialisation procedure.

The initialisation of applications is described in the ***Becoming a Member of the Startup Process*** section of the *Java Pages Development* documentation.

Guidelines for Development

The guidelines you have to follow during development are quite simple:

- Always put *all* your application/adapter classes inside the *softwareag/appclasses* directory or in the corresponding project directories. When using the project management (which is strongly recommended), store the classes in the project directories so that you can easily copy projects as self-containing units between different Application Designer installations.
- Do *not* put classes into the servlet engine's class loader's class path.
- Avoid class duplicates (a *.class* file in the */classes* subdirectory also contained in a *jar* file inside the */lib* subdirectory).
- Reload the classes by creating a new class loader instance. To see the effects re-logon. (The re-logon can be done by refreshing the browser.)

Classpath Extensions in cisconfig.xml

In the *cisconfig.xml* file, you can define the possibility to explicitly include defined directories or *jar*/*zip*/etc. files in the application class loader. The following example shows a *cisconfig.xml* file containing a class loader extension:

```
<cisconfig ...>
    <classpathextension path="c:/development/centralclasses/classes/"/>
    <classpathextension path="c:/development/centralclasses/libs/central.jar"/>
</cisconfig>
```

Consequence: you can also include classes that are located outside the web application's directory structure into the application class loader of Application Designer.

Pay attention: if defining directories that contain *.class* files, then the path definition inside the classpath extension must end with a slash (/).

Loading Resource Files

The Application Designer application class loader does only load classes to be loaded into the Java virtual machine. It is not able to load resource files that you might access from your code.

Place resource files into the web application class loader, below the directory *<webapps>/WEB-INF/classes/* so that they are loaded in a correct way.

Preparing for Runtime

The following topics are covered below:

- Basics
- Example

Basics

As explained in the previous section, the Application Designer class loader concepts are very useful for design time purposes. What is the price? The Application Designer class loader finds its classes by accessing the file system. It uses for this reason the `cis.home` parameter inside the *<webapp>/WEB-INF/web.xml* file in order to know the file root directory of the web application.

At runtime - especially if your application server distributes the load on several physical nodes - this is dangerous: each node may have its own directory structure and you cannot specify one root directory anymore in which the web application is located.

Consequence: for running in these scenarios, you have to prepare your application accordingly - i.e. you have to place your classes at the places where the application server definition defines them to be located.

The normal directories to put classes in are:

- *<webapp>/WEB-INF/lib* for libraries (*.jar* files).
- *<webapp>/WEB-INF/classes* for single class files (*.class* files).

In addition, you must switch off the flag "useownclassloader" inside the *cisconfig.xml*. Consequently, the Application Designer application class loader will not be used at all - all classes are loaded by the web application loader.

Example

Example: let us assume that you have set up the Application Designer application project "projectxyz". The classes for this project are located in

- *<webapp>/projectxyz/appclasses/classes/*.class* and
- *<webapp>/projectxyz/appclasses/lib/*.jar*

so that the Application Designer class loader can reach them.

For changing to the runtime scenario, just copy the **.class* and **.jar* files from your project directory into the corresponding standard directories.

37 Appendix E - StartCISPage Servlet

- Normal Calling of a Page 138
- Appending Application Parameters 138
- Controlling the Session Life Cycle 138
- Controlling the Session ID 139
- Setting Default Parameters 139
- Mixing Parameters 140
- Setting Parameters with the HTTP Method POST 140

The StartCISPage servlet is the central servlet that is used in order to open intelligent HTML pages. It was already mentioned several times in this documentation. This chapter describes certain attributes that you can pass inside the servlet call.

Normal Calling of a Page

A normal page is called in the following way:

```
http://<host>:<port>/<webapplication>/servlet/StartCISPage?PAGEURL=/<project>/<pagename>.html
```

The StartCISPage servlet creates a frameset page around the intelligent HTML page that provides for specific functions that are internally required.

Appending Application Parameters

Application parameters can be passed by just appending the name and the value of the parameters to the URL. Each parameter must be the name of a property that is provided for by the server side adapter.

Example: the adapter provides for a property `company`. When opening a page via

```
http://<host>:<port>/<webapplication>/servlet/StartCISPage?PAGEURL=/<project>/<pagename>.html&company=mycompany
```

then the `setCompany` method of the adapter is called and the value "mycompany" is passed.

This is a very simple and powerful way to pass parameters through the URL.

Controlling the Session Life Cycle

A page relates to adapters living inside a session on server side. A session is opened by default when referencing a page via StartCISPage. By default, it is closed when the initial StartCISPage page is removed - either by closing the browser or by loading a different URL into it.

You can explicitly control this automated removal of sessions with the parameter `ONUNLOADBEHAVIOR`. If you call a page in the following way, the session is not removed when the page is removed:

```
http://<host>:<port>/<webapplication>/servlet/StartCISPage?PAGEURL=/<project>/<pagename>.html&ONUNLOADBEHAVIOUR=NOTHING
```

Controlling the Session ID

By default, a new session ID is internally generated when opening a page by StartCISPage. But you can also pass the session ID and the subsession ID explicitly. This might be of interest if you require to control the Application Designer session management from outside.

Calling a page in the following way

```
http://<host>:<port>/<webapplication>/servlet/StartCISPage?PAGEURL=/<project>/<pagename>.html&SESSIONID=4711&SUBSESSIONID=5
```

will internally open the session with ID 4711 - or use 4711 if it already exists. The same applies on subsession level.

Pay attention: if you use this possibility, then you are responsible for managing session IDs in such a way that they are unique.

Setting Default Parameters

Language

As described in *Multi Language Management*, Application Designer internally holds a language per session. This language can be set from outside:

```
http://<host>:<port>/<webapplication>/servlet/StartCISPage?PAGEURL=/<project>/<pagename>.html&LANGUAGE=E
```

Default Style Sheet

By calling

```
http://<host>:<port>/<webapplication>/servlet/StartCISPage?PAGEURL=/<project>/<pagename>.html&SESSIONCSS=../softwareag/styles/CIS_PARROTT.css&DEFAULTCSS=../softwareag/styles/CIS_PARROTT.css
```

you define that the CIS_PARROTT style sheet is used instead of the default style sheet. Of course, you can reference any style sheet of your own.

Mixing Parameters

All parameters can be mixed without any restrictions.

Setting Parameters with the HTTP Method POST

Instead of adding the parameters to the URL, you can also use the HTTP method `POST` to set the parameters in an HTML form.

Example (similar to the example under ***Appending Application Parameters***, but with `POST`):

```
<html>
<head>
<title>Start Application Designer Demo Application</title>
<script type="text/javascript">
function submitStart() {
document.forms["myform"].submit();
}
</script>
</head>
<body>
 <form id="myform" name="myform" action="servlet/StartCISPage" method="post">
  <input type="hidden" name="PAGEURL" value="/<project>/<pagename>" />
   Company: <input type="input" name="company" value="mycompany" /><br/>
 </form>
 <a href="#" onclick="submitStart()">Start Demo</a>
 <div id="status">Click on Start Demo</div>
</body>
</html>
```