

Dialog Print

= | e

Esito products are copyrighted and all rights are reserved by Esito. This document is also copyrighted and all rights are reserved. This document may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Esito. The information in this document is subject to change without notice, and Esito assumes no responsibility for any errors that may appear in this document. The references in this document to specific platforms supported are subject to change. Genova, Sysdul, Systemator are trademarks or service marks of Esito.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. IBM and Rational Rose are registered trademarks of IBM Corporation. GWT and GWT-based marks are trademarks or registered trademarks of Google Corporation in the U.S. and other countries. Windows, ASP and Visual Basic are registered trademarks of Microsoft Corporation. Enterprise Architect is a registered trademark of Sparx Systems. Any other trademarks or trade names contained herein are the property of their respective owners.

Copyright 2005-2009 © All rights reserved

Esito AS
Postboks 191
N-1325 Lysaker
Norway

DIALOG PRINT – TOC

1	Introduction	1
2	From dialog to printed report	2
2.1	How to.....	2
2.2	Quickstart: Printing a dialog in a java application.....	2
3	Creating report templates.....	5
3.1	Print Settings.....	5
3.2	Fonts.....	7
4	The JasperReports target	8
4.1	Requirements	8
4.1.1	Required libraries.....	8
4.1.2	Font files	9
4.2	How dialog components are mapped.....	10
4.3	Template parameters.....	10
4.4	Compiling report templates.....	11
4.5	Generated files	12
5	Data extraction	13
5.1	Structure of extracted data	13
5.2	Java2XML.....	14
5.2.1	Requirements	14
5.2.2	Generated files.....	14
5.2.3	Custom filters and value converters	15
5.3	Sysdul2XML.....	15
5.3.1	Generated files.....	15
5.3.2	Custom filters and value converters	16
5.3.3	Compiling the generated XML converter routines.....	17
5.4	Data extraction outside print context	17
6	Printing dialogs from a Java application	19
6.1	Configuring the application	19
6.2	The action print	19
6.3	Printing from non-generated code.....	19
7	Exporting reports	20
8	The Print Enterprise Java Bean.....	21
9	Printing in Sysdul context.....	22
9.1	Sysdul print walkthrough.....	22
9.2	Deploying Mule and the print service.....	23

1 Introduction

This manual describes the Print targets in Genova.

- Chapter 1 ["Introduction" on page 1](#) is this chapter.
- Chapter 2 ["From dialog to printed report" on page 2](#) offers a brief overview of the print target, as well as a recipe for turning a dialog into a printed report in a java context.
- Chapter 3 ["Creating report templates" on page 5](#) describes how dialogs are made printable.
- Chapter 4 ["The JasperReports target" on page 8](#) describes the JasperReports target; how to configure it, and how dialog components are mapped into report items.
- Chapter 5 ["Data extraction" on page 13](#) describes how an instance of an object selection is transformed into an XML document; how this process is tailored by adding custom made filters and converters, and how this fits into the print context.
- Chapter 6 ["Printing dialogs from a Java application" on page 19](#) describes the support provided by the genova java run-time to print dialogs.
- Chapter 7 ["Exporting reports" on page 20](#) describes how to export printable screen dialogs from within a java environment.
- Chapter 9 ["Printing in Sysdul context" on page 22](#) describes how to print dialogs from a Sysdul environment.

2 From dialog to printed report

Genova grants developers and designers the option to easily enable dialogs to be sent to a printer. Currently, Genova offers this functionality by utilizing the JasperReports library.

2.1 How to

To make a dialog printable, a report design needs to be generated for that dialog during design time. Genova includes a client target for this task, and the result of it is one or more generated report design templates.

To add functionality to a dialog to make it print itself (i.e. a print button), Genova offers a print action that can be called when GUI events are processed. The default implementation of this action will send its target to the printer.

During run-time, several steps are carried out to turn a populated screen dialog into a printed report. First, a generated service object will transform the object selection instance into an XML document. Then, the report design templates for the target dialog (which may or may not be the dialog shown on the screen) are filled with the data in the XML document. The result will by default be sent to a printer.

It is, of course, quite possible to transform an object selection instance into an XML document outside the print context supported by Genova.

2.2 Quickstart: Printing a dialog in a java application

Here are the steps that needs to be followed in order to make a dialog in a java application printable.

This walk-through requires a workspace, and that the following 3 targets have been successfully generated: Java domain classes, Java/JFC client and Java service target.

1) Make sure the following libraries are available:

- JasperReports 3.0 or later
- Apache Commons BeanUtils 1.7 or later
- Apache Commons Collections 3.2 or later
- Apache Commons Logging 1.1 or later
- Apache Commons Digester 1.8 or later
- Apache Ant 1.7 or later
- Apache Xerces library 2.9 or later
- Apache Xalan 2.7 or later

- 2) Check the default print setting (in the resource database) to see if it can be used. If not, either modify the incorrect values, or create a new print setting:
 - Right-click *Print settings* in the resource database, and choose *New print setting*.
 - Double click the newly created print setting, and select a style. This style will only be used for header and footer text.
 - Set correct page size in millimeters (i.e. for A4 portrait, width is 210, and height is 297).
 - Set correct values for margins in millimeters.
 - (optional) Set default values for the various header and footer fields. These values must be given as java Strings (plain texts must be between double quotes). Each dialog is free to override these header and footer settings.
 - Click *OK*.
- 3) Make a dialog subject for printing. This is done by selecting a print setting for it.
 - Open the dialog model by double-clicking it.
 - Open the properties window of the dialog by double-clicking the root of the dialog model.
 - Select the *Print* tab.
 - Select a print setting from the drop-down menu.
 - Click *OK*, and then save the dialog.
- 4) Add a Print action to a GUI event for the dialog (i.e. a "print" button). The target dialog for the print action will point out which report template to use. This may be the same dialog that contains the Print action. Or, in case the printed report should differ from the screen dialog layout-wise, the target of the print action may be any other dialog, as long as the other dialog uses the same object selection as the screen dialog. Remember to save the changes made.
- 5) Generate the client code for the dialog with the print action.
- 6) Generate JasperReports templates for the dialog. Open and select the dialog, and from the menu bar, select *Dialog model* › *Other targets* › *Jasper Reports template*.

- 7) Compile the generated .jrxml files to .jasper files. This step depends on Ant and the Eclipse JDT compiler 3.2 or later.
 - Modify the generated build.xml file, and specify the location of the required libraries.
 - Build the "package" target. The result of this should be a .jar file containing the required .jasper files.
 - Make sure the created .jar file is included in the classpath of the java application.
- 8) Generate the *Java2XML* target for the object selection associated with the dialog.
 - Set correct value for the *ServicePackage* template parameter for the Java2XML target (ie. *example.service.xmlconvert*). This is done in the setup database under Serve Designer, Java2XML Conversion.
 - Open and select the object selection, and from the menu bar, select *Object selection › Other targets › Java2XML conversion*.
- 9) Configure the application. Open *genova.config.properties* and set the *ServiceApplication.<ApplicationName>.DefaultXmlConvertPackage* property to the root package of the XML conversion (ie. *example.service.xmlconvert*)

The application now contains a printable dialog. Compile and run to test the added functionality.

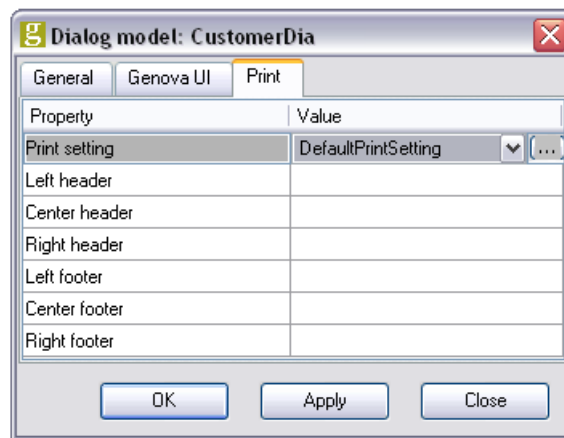
3 Creating report templates

The term *report template* in print context corresponds to the term *report design template* used by several reporting libraries; Files that define the layout of a particular report type.

Genova includes a client target that generates report templates usable by the JasperReports library. This target is described in [chapter 4 on page 8](#).

3.1 Print Settings

A PrintSetting needs to be assigned to each dialog that should be printable.



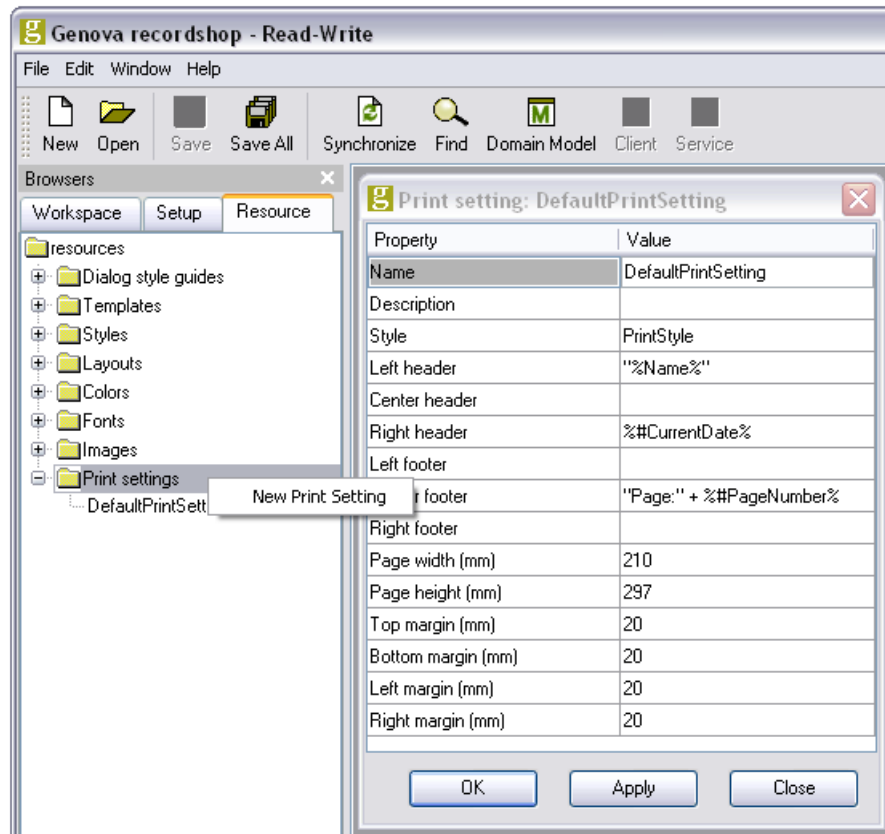
Each PrintSetting defines page size, margins, header and footer text and style. The header and footer texts can be overridden in the Print tab of the dialog model properties.

PrintSettings need to be defined, just like any other resource.

The figure on the next page shows the creation of a new print setting and the elements that can be set for each print setting.

The values for the various header and footer fields must be Java Strings. This means plain texts must be given in double quotes. This also gives access to Java expressions, as whatever can be used in the header and footer fields, as long as it evaluates to a Java String.

The substitution functionality presented by the generator can be used in header and footer fields. The mechanics behind this is explained in [chapter 8](#) of *Template Parser*. We will be on a DialogModel node in the iteration. The substitutions available at this node are listed in [section 4.4](#) of *Dialog Generator*. The report targets will always provide the following three variables for substitution; *#CurrentDate*, *#CurrentTime* and *#PageNumber*. These will be substituted to something that evaluates to a Java String. As always, substitution directives are given between %s, as *##PageNumber%*.



The following table lists some sample values for the header and footer fields.

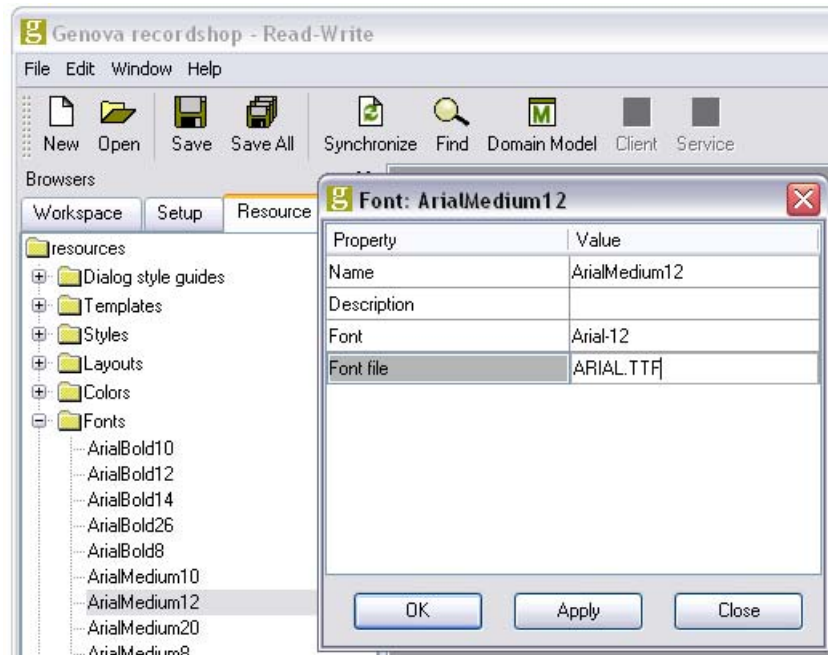
Sample value	Meaning
"© 2008 Esito AS"	Plain text
"Page:" + %#PageNumber%	String expression and substitution (page number)
%#CurrentDate%	Substitution (current date, as a String)
%#CurrentTime%	Substitution (current time, as a String)
System.getProperty("user.name")	String expression (user name)
"%Name%"	Substituted by generator (name of dialog)

The approach to get current date and time presented in the examples, is recommended over evaluating them through new Date(), since the latter may result in date and/or time changing across pages.

3.2 Fonts

A print template target may require additional information about fonts used in a dialog. The Font file property of the font resource should in those cases hold the required information. It may be required that this information is case sensitive (even in a Windows environment). It is therefore recommended that the value of this property always reflects the filename exactly as it is presented by the operating system.

Currently, this information is only needed if you wish to enable a dialog to be exported to a PDF file.



4 The JasperReports target

Genova supports generation of JasperReports report templates based on dialog designs.

JasperReports is an open-source Java library for creating reports from within a java environment. General documentation of the JasperReports project is beyond the scope of this document, and the readers should therefore consult the JasperReports project home, currently located at <http://jasperforge.org/projects/jasperreports>, to get a basic understanding of JasperReports.

The support for JasperReports is taken care of by the JasperReports client target, and by service code in the Genova runtime that invokes the JasperReports library to fill reports.

The JasperReports client target will create .jrxml files, which are report templates based on dialog models. These files must be validated and compiled into one or more JasperReports objects, and serialized as .jasper files. Genova will create a build.xml that calls upon an Ant task included in the JasperReports distribution to take care of this step. The result of all this is that there will be one or more .jasper files for each dialog that is printable.

The .jasper files are deployed by simply adding them to the classpath of the client application. They will then be accessible to the provided print action. The print action takes fully care of filling the report template with the object selection instance that populates a screen dialog, and sending the result to a printer.

4.1 Requirements

The following requirements need to be met in order to enable the JasperReports functionality in Genova context;

4.1.1 Required libraries

From the developer perspective, functionality offered by the JasperReports library is called upon in three different ways in Genova context;

First, an Ant task will verify and transform the generated .jrxml files into java classes, and compile and store the result as .jasper files. Second, during run-time, the genova runtime framework may fill a report template with data, and send the result to a printer. And last, during runtime, JasperReports may be called upon to fill a report template and export the result as a .pdf file.

The following libraries are required in some or all cases:

- **Genova runtime** (genova-common and genova-swing) 8.3 or later. Needed in all cases (including during report compilation time).
- **JasperReports** 3.0 or later. Needed in all cases. available from <http://sourceforge.net/projects/jasperreports/>
- **Apache Commons BeanUtils** 1.7 or later. Needed in all cases. Available from <http://commons.apache.org/beanutils/>
- **Apache Commons Logging** 1.1 or later. Needed in all cases. Available from <http://commons.apache.org/logging/>
- **Apache Xalan** 2.7 or later. Needed during run-time when reports are filled. Available from <http://xml.apache.org/xalan-j/>
- **iText** 2.0 or later. Used to create PDFs from filled reports. Available from <http://www.lowagie.com/iText/>
- **Apache Ant** 1.7 or later. Needed to compile .jrxml files into .jrxml. Available from <http://ant.apache.org/>
- **Apache Commons Collections** 3.2 or later. Used by JasperReports to create .jasper files from .jrxml. Available from <http://commons.apache.org/collections/>
- **Apache Commons Digester** 1.8 or later. Used by JasperReports to create .jasper files from .jrxml. Available from <http://commons.apache.org/digester/>
- **Eclipse JDT compiler** 3.2 or later. Used by JasperReports to create .jasper files from .jrxml. Reachable from <http://update.eclipse.org/downloads/>

4.1.2 Font files

As mentioned in [section 3.2](#), some print targets may require the filename of the used fonts to be specified in the Genova resource database. This holds true for the JasperReports target.

A warning will be issued during generation of report templates if a font that lacks this information is used in a printable dialog. This is because the iText library, used by JasperReports to create PDFs, needs access to the font files.

In other words, if it is desired to use JasperReports to create PDFs from screen dialogs, it is essential that all fonts used in those dialogs have correct filenames in the resource database. Furthermore, font files of all used fonts must be reachable through the classpath of the application.

4.2 How dialog components are mapped

The JasperReports target will attempt to make the report template look similar to the screen dialog. Here is a brief description of how dialog components are mapped into report elements.

Data items:

- **TextField, RadioGroup, ComboBox, ListBox, Scale and Stepper:** These will all appear as texts, like a TextField in a dialog.
- **CheckButton:** will be represented as a check box. The title of the check button will be placed to the right of the check box (just as they are for check buttons in screen dialogs).

Non-data items:

- **Button, MenuItem, ScrollBar and Spacer:** These will be silently ignored. However, their presence in the dialog may have an influence on the layout of the report template.
- **Label and Text:** These will appear as text, as they are in the dialog.
- **ImageBox, Separator, ViewPort and Component:** These are not supported, and a warning will be issued.

Containers:

- **WindowBlock and SimpleBlock:** The contents of these containers will be handled individually.
- **TableBlock and TreeView:** These containers are not supported, and a warning will be issued.
- **MenuBar, Menu, ToolBar and TreeNode:** These containers, and their contents, will be silently ignored.
- **Notebook:** All contained tab cards will be displayed subsequently below each other, and the titles of the tab cards will be used as labels above the content of the cards.
- **ListBlock:** Will result in a sub-report containing the elements of the list. The sub-report will be included in the main report.

4.3 Template parameters

It is possible to configure some aspects of how report elements are generated from dialog components.

The following optional parameters are recognized:

- **HeaderLine:** If *true*, a line will appear below the page header.
- **FooterLine:** If *true*, a line will appear above the page footer.

- **SpaceAboveHeader**: Space between page header and the top margin defined in a dialog's Print Setting. May be set to override the default 0 (mm).
- **SpaceBelowHeader**: Space between page header and body. May be set to override the default 8 (mm).
- **SpaceAboveFooter**: Space between page footer and body. May be set to override the default 8 (mm).
- **SpaceBelowFooter**: Space between page footer and body. May be set to override the default 8 (mm).
- **NotebookUnderlineLabel**: Set to *true* to underline notebook labels.
- **NotebookSpaceBeforeLabel**: Vertical space before a Notebook label. Default is 4 mm.
- **NotebookSpaceAfterLabel**: Vertical space after a Notebook label. Default is 1 mm.
- **CheckButtonTitleOffset**: Horizontal offset for check button title. Default is 10 mm.
- **ListRowShadingColor**: If set, rows of a ListBlock will be colored in groups (i.e. every other row is slightly darker). The value must be between #000000 and #FFFFFF.
- **ListRowShadingGroupSize**: How many subsequent rows to color and not color.
- **ListBlockNoDataText**: By default, empty tables are shown as a header only. If this parameter is set, the value of it will appear just below the header. Can be used to explicitly state that a table is empty.

4.4 Compiling report templates

All generated report templates need to be compiled from .jrxml to .jasper files. Genova will generate a build.xml that calls upon an Ant task shipped with JasperReports that takes care of this. Note that the build file will only be generated if it does not exist. The build process will compile all new and modified .jrxml files in the target directory.

Consult [section 4.1.1](#) (or the generated *build.xml*) to get a list of required libraries. The location of these libraries must be specified in the generated build file. The simplest way to achieve this, is to place all required jars in the same directory (ie. *c:\jasperbuildjars*), and change the *<path>* node of the *build.xml* to the following:

```
<path id="classpath">
  <fileset dir="c:/jasperbuildjars">
    <include name="**/*.jar"/>
  </fileset>
</path>
```

The *build.xml* file includes a "package" target for easy deployment. This target will first compile all new and modified .jrxml files, and then pack all .jasper files into a Java Archive (.jar) file. The resulting .jar file should be included in the classpath of the client application.

4.5 Generated files

The JasperReports target will generate one .jrxml file for each printable dialog, along with one .jrxml file for each ListBlock in that dialog. In addition, a build.xml file is created that calls for an Ant task shipped with JasperReports to compile all generated .jrxml files. The result of a successful build is one .jasper file for each generated .jrxml file.

By default the templates are directing the output into a subdirectory under the target directory: *print*. The *print* directory will contain all of the generated .jrxml files. If you want another or no substructures under the target directory this is achieved by assigning a value to the template variable *PrintDirectory*. Setting this variable to empty will make all output be directed into the target directory without using any substructure.

5 Data extraction

The term *data extraction* in Genova print context means transforming an instance of an object selection populating a screen dialog into XML.

Genova contains two service targets that generate code for this transformation: Java2XML and Sysdul2XML. The first one generates Java code that transforms an instance of an object selection into an `org.w3c.dom.Document` object. The latter converts an instance of an object selection into a Sysdul string containing the data as XML.

5.1 Structure of extracted data

Whether contained in a DOM object, or in a Sysdul string, the XML document will have the same structure.

The single *root element* of the document will have the name of the object selection.

This XML root element will contain one *collection element* for each object selection root node. Nothing else is contained in the XML root node. A similar collection node will also be generated for all other many-nodes, however they will be owned by the role that references them. The name of the collection element will in all cases be the role name, suffixed by "_collection".

Each collection element will contain one *domain object element* for each instance of the corresponding role. This XML element will have the same name as the object selection role.

Attributes of domain objects will appear in the corresponding XML element as leaf elements of their role, with the same name as the attribute.

Roles of the domain object will become children of the XML element corresponding to the referring domain object. If the role is one-related, the owned element will be a domain object element. If the role is many-related, the owned element will be a collection element containing one domain object element for each instance of the referenced object.

It is possible to add filters and converters to this process. A filter will remove a role or an attribute from the resulting XML, for instance, to prevent exposure of sensitive information. The role of a converter is to convert the data, for instance from some internal storage-format into presentable text. How converters and filters are added, depends on whether we are in a Java or Sysdul context.

5.2 Java2XML

The Java2XML target generates Java code that converts an object selection to an `org.w3c.dom.Document`.

5.2.1 Requirements

The generated classes will depend on Xerces-impl from the Apache Xerces library version 2.9 or later. This library is available from <http://xerces.apache.org/xerces2-j/>

The Java2XML target needs to be configured by setting the template parameter *ServicePackage* (ie, set it to *example.recordshop.service*). This setting dictates the super-package for all generated converters. A sub-package will be created for each object selection, with the same name as the object selection.

Furthermore, we must tell the application the name of the super-package of all generated XML converters. This is done in the *genova.config.properties* file. Simply add the following line:

```
ServiceApplication.<appName>.DefaultXmlConvertPackage=example.recordshop.service
```

The Java2XML target depends on all included attributes being accessible through bean-style accessors (*get* methods). If *is* is used instead of *get* as accessor prefix for boolean attributes, simply set the optional template parameter *BooleanAccessorPrefix* to *is* in the setup database.

5.2.2 Generated files

Two classes will be generated for each object selection. *<osName>DefaultXmlConverter* and *<osName>XmlConverter*.

The latter is a subclass of the first, and will not be generated if the source file already exist.

<osName>DefaultXmlConverter will, in turn, subclass *no.genova.support.xml.AbstractXmlConverter*.

By default the templates are directing the output into a subdirectory under the target directory: *java*. The *java* directory will contain any generated java source. By default this structure is used by all generators when generating code for a java oriented target. If you want another or no substructures under the target directory this is achieved by adding a template variable to the setup database for this target:

- `JavaDirectory`

Setting this variable to empty will make all output be directed into the target directory without using any substructure. See [section 8.2.5 on page 53](#) in the User Guide on how to create new template parameters.

5.2.3 Custom filters and value converters

The abstract base class of all XML converters provides functionality for adding value converters and filters.

Custom filters: Generated subclasses will call *isRoleIncluded* and *isAttributeIncluded* to check if a role or attribute should be included in the generated XML Document. This behavior can be used in two ways to add custom filters. The obvious way is to override these methods, and take care of all aspects of filtering in the subclass. However, there is support in the default implementation for adding filters. The documentation of *no.genova.support.xml.AbstractXmlConverter* (Javadoc) includes an example that shows how this support is utilized.

Custom value converters: The generated subclass will call *convertObjectToString* for all included attributes to convert them to strings. This method can be overridden to add custom value conversion. And as with filters, the abstract super class includes support for adding custom value converters for attributes. The documentation of *no.genova.support.xml.AbstractXmlConverter* (Javadoc) includes an example that shows how to take advantage of this support.

5.3 Sysdul2XML

The Sysdul2XML service target generates Sysdul source code which will convert object selection data into a string of type vartext. The XML string is unformatted, i.e. there are no extra line breaks or indentation.

5.3.1 Generated files

For each object selection, the following files will be generated:

<os>_driver.vsd: This file contains the main entry point for the XML conversion of the object selection, the *Q_XML_<os>* procedure. This procedure expects a vartext parameter which will contain the XML data on return. To obtain the object selection instance data, the procedure will call one procedure for each role in the object selection, *Q_OBT_<role>_<os>*, which will return an array with role instance data. The *Q_OBT* procedures will be generated by Grape for dialogs with Print actions. And they may also be generated if XML conversion is desired outside the print context.

<os>.vsd: This file contains a number of procedures responsible for writing object selection data in XML format. There is one set of procedures for each object selection role, which will write the start tag, the instance data,

and the end tag. There are hooks available which make the procedures customizable.

<os>.svp: This Svapp file is where the developer may override the default behavior of the XML conversion. Initially, the file only includes the **<os>_default.svp** file. This file will not be overwritten by subsequent Sysdul2XML target generation.

<os>_default.svp: This Svapp file has a set of macro definitions, providing the default Sysdul statements used by the conversion procedures to write XML. The Svapp macros may be overridden to customize the XML conversion.

<os>_formats.svp: The default XML conversion procedures uses the format macros from this file when writing XML data for role attributes. Initially, the file only includes the **<os>_default_formats.svp** file. This file will not be overwritten by subsequent Sysdul2XML target generation.

<os>_default_formats.svp: This file contains default format macros for role attributes. The format macros are for most data types based on the display rule given in the model. The format macros may be overridden to change the format of the role attributes used by the default XML conversion procedures.

In addition to the files generated for each object selection role, there is one application wide Sysdul file. This file, named **<app>.vsd**, contains two utility procedures. The first, **Q_ESCAPE_XML**, is called by the XML conversion procedures before writing to the XML output string. This is to ensure that the XML output string contains valid XML. The usage of the **Q_ESCAPE_XML** procedure may be overridden by customizing a Svapp macro. The other procedure, **Q_STORE_XML_FILE**, will store a XML string to file. This procedure is used by the code generated by Grape for Print actions.

5.3.2 Custom filters and value converters

By overriding the default Svapp macros generated by the Sysdul2XML templates, the developer may customize the behavior of the XML conversion. This includes filtering out unwanted role attributes or roles, possibly based on the instance data. The format of role attributes when written to the XML string may also be customized. Version 8.3 or later of Svapp supports overriding previously defined macros by defining a new macro with the same name.

The generated XML conversion routines does not support object selections with many-related roles nested more than one level. This limitation also applies to code generated by Grape. Object selections with more than one root are supported in the Sysdul2XML target, but not in Grape.

5.3.3 Compiling the generated XML converter routines

The generated Sysdul files (*.vsd*) includes the necessary Svapp files (*.svp*), and should be compiled as the other Sysdul files in the project.

5.4 Data extraction outside print context

There is a broad selection of software available in the market that transforms/processes XML documents. It is, of course, possible to convert an object selection instance to an XML document outside the print context. Here follows an example of how the data populating a screen dialog is transformed into XML.

Let's say we have a dialog *RecordDia*, and an object selection *RecordDia_os*. We wish to add a button to that dialog that, when clicked, stores the displayed data in a file. Here are the steps that needs to be followed to grant this wish.

- 1) Add a button to the dialog. Add an event to the button that activates a method called *xmlExport* when the button is clicked. Generate the Java target for that dialog to make the changes take effect.
- 2) Generate the Java2XML service target for the *RecordDia_os* object selection. This is done by opening and selecting the object selection, and then selecting *Object selection* › *Other targets* › *Java2XML conversion* from the menu.
- 3) Add the following method to *RecordDiaController.java*, and add appropriate Exception handling (omitted for simplicity).

```
public void xmlExport() {
    ObjectSelection os = obtainObjectSelection();
    ClientContext cContext = Application.getClientContext();
    Document doc = Application.getXmlConverter().convert(os, cContext);
    // Voila!! The data of the screen dialog is now an XML document.
    // Now store it as a file...
    OutputFormat format = new OutputFormat();
    format.setIndent(4);
    PrintStream ps = new PrintStream("someFile.xml");
    XMLSerializer ser = new XMLSerializer(ps, format);
    ser.serialize(doc);
    ps.close();
}
```

- 4) Add the following imports to *RecordDiaController.java*:

```
import java.io.PrintStream;
import no.genova.client.support.*;
import no.genova.support.*;
import org.w3c.dom.*;
import org.apache.xerces.dom.*;
import org.apache.xml.serialize.*;
```

5) Compile and run the application.

6 Printing dialogs from a Java application

The Genova runtime provides support for printing from Java applications. This support is present through the action print. It can be invoked manually as well as from generated code.

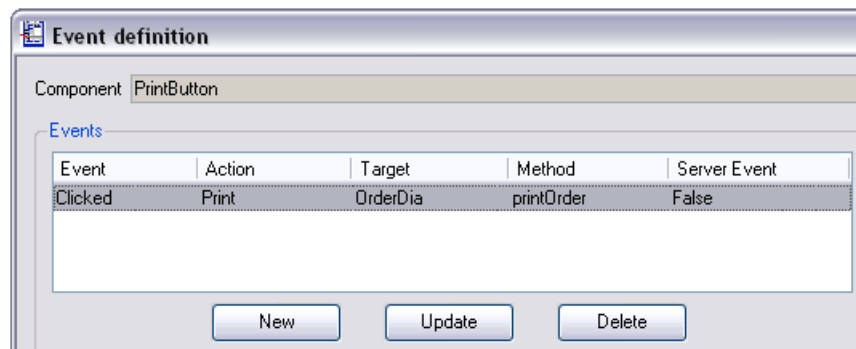
6.1 Configuring the application

The default configuration of applications generated with Genova will use Spring to locate the *PrintService* implementation. It is possible to override this behavior in the *genova.config.properties* file of the application. Simply add the following two lines to this file:

```
PrintService=no.genova.jgrape.print.PrintCentral
ExportService=no.genova.jgrape.print.PrintCentral
```

6.2 The action print

Genova offers the option to generate code that invokes the action print on GUI events.



Please consult [chapter 16](#) of the *Dialog Designer* document for detailed information about events and actions.

6.3 Printing from non-generated code

Printing can easily be done anywhere in an application. The following line of code will give access to the implementation of the *PrintService* the application is configured to use:

```
PrintService pService = Application.getPrintService();
```

Consult the javadoc of *no.genova.service.print.PrintService* and *no.genova.service.print.PrintContext* for a better understanding of how to print from manually written code.

7 Exporting reports

In this chapter, we will give an example of how to create a PDF file based on a screen dialog.

Let's say we have a dialog named *RecordDia*, with the corresponding object selection *RecordDia_os*. We wish to add a button to this dialog, that, when clicked, creates a PDF reflecting the contents of the screen dialog.

- 1) Generate and compile JasperReports design templates for the *RecordDia* Dialog by performing steps 1, 2, 3, 6 and 7 of [Quickstart: Printing a dialog in a java application](#). In addition to the libraries specified in step 1, make sure iText 2.0 or later is present for the application.
- 2) Add a button to the dialog, and add an event to the button that activates a method called *pdfExport* when the button is clicked.
- 3) For all fonts used in the dialog, specify (in the resource database) the filename of the font. This must be done case-sensitive, even in a Windows context. Furthermore, during run-time, the used fonts must be in the classpath.
- 4) Make sure the Java2XML service target for the *RecordDia_os* object selection is generated.
- 5) Add the following code to *RecordDiaController.java*, and add appropriate Exception handling (omitted for simplicity).

```
public void pdfExport() {
    ObjectSelection os = obtainObjectSelection();
    Document doc = Application.getXmlConverter().convert(os, null);
    Application.getExportService().exportToPDF(doc, "RecordDia", "some.pdf");
}
```

- 6) Add the following imports to *RecordDiaController.java*:

```
import no.genova.client.support.*;
import no.genova.support.*;
import org.w3c.dom.*;
```

- 7) Compile and run the application.

8 The Print Enterprise Java Bean

It is possible to deploy the Print services on an application server and a session EJB, *printEJB.jar*, is provided. See [Chapter 9 "Running the generated application with Enterprise Java Beans" on page 77](#) in the Java Target manual for information on how to use the EJB.

9 Printing in Sysdul context

To support printing of dialogs from a Sysdul client, the Sysdul run-time library depends on an external service to offer printing. The following steps are carried out in the background each time a Sysdul dialog is successfully printed:

- First, the print action procedure for the dialog will generate an XML file containing the dialog data by using the XML conversion routines from the *Sysdul2XML* target. The default file name for the XML file is *<host_ip>-<os>.xml*, where *<host_ip>* is the IP address of the host running the Squire process, and *<os>* is the name of the object selection.
- The print action procedure will then issue an HTTP GET call to the Squire host on a given port. The URL will contain the XML file name and the name of the printing dialog.
- A Mule instance running on the Squire host will listen for and accept the GET request. The request will be routed to a contained print service that will take care of filling the correct report template (pointed out in the URL) with the XML file (also pointed out in the URL).

All this will result in the Windows print dialog popping up, allowing the user to select printer and other print parameters before printing the document.

Note that the XML data file will not be deleted during or after a successful printing. This may be done routinely by a shell script or similar.

9.1 Sysdul print walkthrough

To be able to print data from Sysdul dialogs, the following steps need to be carried out:

- 1) Generate Jasper Report templates for the dialogs by performing steps 2, 3, 6 and 7 of [Quickstart: Printing a dialog in a java application](#).
- 2) Add a Print action to all printable dialogs. The print action is supported by the Sysdul Program Generator, see [Chapter 3.22 "Print action" on page 11](#) in the Sysdul Program Generator manual. The target dialog for the print action will point out which report template to use. This may be the same dialog that contains the Print action. Or, in case the printed report should differ from the screen dialog layout-wise, the target of the print action may be any other dialog, as long as the other dialog uses the same object selection as the screen dialog.
- 3) Generate the Sysdul XML conversion routines for the object selections of the print dialogs. For the application, right click and select *Generate services...* Select *Sysdul2XML Conversion* as the service target. Alternatively, for each printable dialog, open and select its object selection,

and from the menu bar, choose *Object Selection* › *Other targets* › *Sysdul2XML Conversion*.

- 4) Generate Sysdul code with Grape for the dialogs with print actions. If the print service of the Mule instance is listening to another HTTP port than the default (port 80), the port used by the Grape code must be overridden in a hook.
- 5) Build and deploy the Sysdul application.
- 6) Generate a new dialog description file (.ddf) for the application.
- 7) Start the Mule instance on the Squire host.
 - Make sure Mule is configured properly for the print service. An example Mule configuration file, named *jasperprint-config.xml* is distributed with Genova in the *%GENOVA_HOME%\misc* directory.
 - Make sure the files containing the XML data are available to the Mule instance. This is granted if Mule is started in the same directory where the XML files are written by the Sysdul application.

9.2 Deploying Mule and the print service

The following environment variables should be set before starting Mule:

- *MULE_HOME*: The value of this variable should be the directory where Mule is installed.
- *MULE_LIB*: The directory containing the Mule configuration file for the print service.

The print service depends on the following libraries:

- JasperReports 3.0 or later
- Apache Commons BeanUtils 1.7 or later
- Apache Commons Collections 3.2 or later
- Apache Commons Logging 1.1 or later
- Apache Commons Digester 1.8 or later
- Apache Xerces library 2.9 or later
- Apache Xalan 2.7 or later
- Genova runtime library 8.3 or later

The easiest way to make these libraries available, is to place a copy of them in the *%MULE_HOME%\lib\user* directory. Any jars containing report template files (.jasper) must be available for the Mule instance as well.

Mule is started from the command prompt with:

```
mule -config jasperprint-config.xml
```
