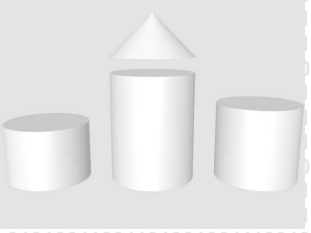

ODABA NG

Document Generation

run Software-Werkstatt GmbH
KÃ¶penicker Strasse 325
12555 Berlin

www.run-software.com

Tel: +49 (30) 609 853 44
e-mail: run@run-software.com

Berlin, September 2012

Table of Contents

This document has been generated from database topics using Open Office document generation.

1 Introduction

ODABA

ODABA is an terminology-oriented database system that allows storing objects and methods as well as causalities . As terminology-oriented database, ODABA supports complex object types (user-defined data types) defined in a terminology model, which reflect application relevant concepts.

ODABA applications are characterized by high flexibility. In addition to object type or context hierarchies, ODABA supports multifarious relations between object instances (master and detail relations, relations between independent object instances and others). This way, behavior of objects in the real world can be represented considerably better than in relational database systems.

ODABA supports event-driven applications concerning the graphical user interface as well as the database level. Thus, application design is tightly related to the experts or customers problem, since it refers to the same names and concepts as being defined by subject matter experts. This enables ODABA to solve highly complex jobs in administrative and knowledge areas.

Platforms

ODABA supports windows platforms (from Windows 95 up to Windows 7) as well as UNIX platforms (Linux, SUN Solaris). ODABA supports 64 and 32 bit technologies.

ODABA also runs well in heterogeneous client/server environments or with Internet servers.

Interfaces

ODABA supports several technical interfaces:

- C++, .Net as application program interface (this allows e.g. using ODABA in C# or VB scripts and applications)
- ODABA Script Interface (OSI) for accessing data via a script language, which is similar to C# or JAVA.
- Multiple storage support for using relational databases for storing ODABA data
- XML for supporting data exchange with complex data structures
- OIF (object interchange format), flat files and ESDF (extended self delimiter fields) for accessing data provided in external file formats
- Document exchange support for importing or exporting data from/to open office or Microsoft office documents.

Tools

ODABA provides a number of database maintenance tools, but also development tools in order to provide terminology model definitions, data model specifications, application design and others.

To support just-in-time documentation, all ODABA tools provide extended documentation facilities, which are the base for generating system and WEB documentation, but also online help systems.

2 Generate documents from database content

ODABA supports creating different kinds of documents:

- Creating LibreOffice documents (open document standard)
- Creating MS Office documents
- Creating HTML documentation

All kind of documents are based on document templates, which might be changed by the user. Document templates (LibreOffice, MS Office) are provided as external resources. HTML templates (OSI functions) are implemented in the **ode.dev** database and might be changed there or passed as external OSI files.

Document generation might be called from a command line, in which case the document template may define any kind of document for any type of object, but also from within an application program.

The behavior and environment for calling document generating actions is described in "Document actions" (see below). How to change document and HTML styles is described in "**Customizing document templates**".

2.1 Providing document templates

Document templates are supported as OSI templates for creating simple text or HTML files or as document template for generating Open Document documents.

The way of calling document generation from within an application program depends on the document type. Generating Open Document Standard compatible documents provides most flexible way of document generation. Calling HTML generation or any other type of document generation by means of OSI templates or functions simply requires an OSI function call. MS Office document may be created by calling MS Office Word and generating the document from within MS Office e.g. by means of VB Scripts.

In order to pass data from the application to the document template, appropriate option variables may be set or written to an ini-file which will be passed to the external program call.

2.1.1 OSI templates

OSI templates are a specific way of defining a function. Usually, OSI templates are used, when having a sort of text template, which is to be filled with data from the database.

Since one can use the **`SystemClass::`**`WriteResult()` or **`FileHandle::`**`Out()` functions in a OSI function, template functions are not really required, but are useful in many cases. In contrast to OSI functions, template functions are easier to read and easier to write.

OSI templates support conditional text generation as well as embedded OSI expressions. One may call templates from within a template etc. The template result will be generated as templates are called. Since one may access the template result at any time, one may also update the generated result during the generation process.

OSI templates can be considered as inverse functions, but there are some restrictions compared with functions. The example below shows a way of converting an OSI template into an OSI function. Since templates are just a simplified way of defining text functions, one may also mix templates and OSI functions or create a template result just by calling OSI functions.

Details for OSI template syntax are described in the Reference Documentation **OSI Template syntax**.

```
$template string Test()$
Dear $if (sex == "male")$Mr. $else$Mrs. $end$ $family_name$,
We just got your question concerning $product$, which you are using since
$buing_date$. We have forwarded your problem to
$responsible(0).first_name$\ $responsible(0).second_name$.
You will get an response during the next three days.
...
$return TemplateString$
$end$

// same template expressed as OSI function
function string Test () {
  WriteResult("Dear ",false);
  if (sex == "male") WriteResult("Mr. ",false);
  else WriteResult("Mrs. ",false);
  WriteResult(family_name,false);
  WriteResult(",",false);
  WriteResult("We just got your question concerning ",false);
  WriteResult(product,false);
  WriteResult("which you are using since ",false);
  WriteResult(buing_date,false);
  WriteResult(".",false);
  WriteResult("We have forwarded your problem to ",false);
  WriteResult(responsible(0).first_name,false);
  WriteResult(" ",false);
  WriteResult(responsible(0).second_name,false);
  WriteResult("You will get an response during the next three days.
\n...\n",false);

  return TemplateString;
}
```

2.1.1.1 ASCII Templates

OSI templates can be considered as inverse OSI functions, but there are some restrictions compared with OSI functions. In order to distinguish code from text, for inserting comments and for other reasons, some characters have been reserved and have to be used with care when appearing in the text.

One may create any sort of text output using template functions. The only reserved characters in a template are $ and \. When template text contains $ or \, one needs to add a \ before, i.e. \$ or \\ in the text will be converted to $ or \ respectively.

Fill characters

Blanks, new lines (0x0D0A or 0x0A) and tabs (0x09) are considered as fill characters. In some cases, fill characters are not displayed properly. Usually, all fill characters between text and embedded expressions are considered as part of the text constant.

This will also include a blank between first and family name (see example 1). Line breaks or tabs between embedded expressions or at the beginning of the text are also considered as fill characters to be displayed in the result (example 2).

For ignoring line breaks between embedded expressions one may use the line connector \. Adding a "\" at the end of a line causes the template to consider the next line as continuation of the current line. This also allows inserting line breaks in the template text, which may increase the readability of the template (example 3).

```
// template text 1
  ...
  $responsible(0).first_name$ $responsible(0).second_name$
  ...
// --> generated OSI code
  WriteResult(responsible(0).first_name,false);
  WriteResult(" ",false);
  WriteResult(responsible(0).second_name,false);

// template text 2
  ...
  $responsible(0).first_name$
  $responsible(0).second_name$
  ...
// --> generated OSI text
  WriteResult(responsible(0).first_name,false);
  WriteResult("\n",false);
  WriteResult(responsible(0).second_name,false);

// template text 3
  ...
  This is a longer text constant to be displayed in the result \
  in a single line. To make the template more readable, we can \
  add "\\" in the template text before line break.
  ...
```

Fill character sequences

All new lines found in the template before and after text constants are considered as text constants and included in the result. Thus, the template fragment

One may also add explicitly defined fill characters as \n, \t or \, which will have the same effect. In order to write characters as \n to the output, control characters have to be escaped by double backslash.

```
// Example 1: fixed text in template
  You will get an response during the next three days.
  ...
// will generate the following OSI function code
  WriteResult("You will get an response during the next three days.
\n...\n",false);

// example 2: explicit control characters
  You will get an response during the next three days.\n...\n
// will generate the same code as the example above
  WriteResult("You will get an response during the next three days.
\n...\n",false);

// example 3: write control sequences to output
  You will get an response during the next three days.\\n...\\n
// will generate the following OSI function code
  WriteResult("You will get an response during the next three days.
\\n...\\n",false);
```

Comments

Comments are line comments introduced by //. Comments can be placed at the end of a line, only, i.e. any text after the comment-begin is considered as part of the comment until the line end.

Comments within a text constant are not recognized as such but considered as part of the text constant. Adding comments in a template is possible at the beginning of the template or after embedded code or immediately after a fill character sequence (example 1)

In order to append a comment at the end of a text constant, you have to insert an explicit line break before the comment (example 2). For generating // sequences as template text, one may use \/, which will be converted to // (example 3)

```
Example 1
  // this is a valid line comment (at beginning of template)
  This is part of the template text // and this also
  $Data$      // display current data - valid comment

Example 2
  This is part of the template text \n// after line end this becomes a
comment
  $Data$    // display current data - valid comment
```

```
Example 3
  This is part of the template text \/ generated as //
  $Data$    // display current data - valid comment
```

2.1.1.2HTML templates

In case of HTML templates, however, text included in the function requires special treatment, because characters as < or > need to be converted. This is automatically done, when defining an HTML template as shown below:

When converting HTML templates to OSI functions, all text read from the database is converted to HTML by replacing reserved characters (see below). The difference is in calling the **`SystemClass::`**`WriteResult()` function, which passes **true** as second parameter to indicate HTML conversion.

```
  <template>
    <header> string Test() </header>
    <processing>
      <body>
Dear $if (sex == "male")$Mr. $else$Mrs. $end$ $family_name$,
We just got your question concerning $product$, which you are using
since $buying_date$. We have forwarded you problem to
$responsible(0).first_name$ $responsible(0).second_name$.
You will get an response during the next three days. <br/>
      </body>
      $return TemplateString$
    </processing>
  </template>

// will be converted to
function string Test () {
PROCESS
   WriteResult("    <body>\nDear ",false);
   if (sex == "male")
     WriteResult("Mr. "",false);
   else
     WriteResult("Mrs. ",false);
   WriteResult(family_name,true);
   WriteResult(",\n",false);
   WriteResult("We just got your question concerning ",false);
   WriteResult(product,true);
   WriteResult(", which you are using\n since ",false);
   WriteResult(buying_date,true);
   WriteResult(". We have forwarded you problem to \n",false);
   WriteResult(responsible(0).first_name,true);
   WriteResult(" ",false);
   WriteResult(responsible(0).second_name,true);
   WriteResult(".\n You will get an response during the next three days.
<br/>\n    </body>";
   return TemplateString;
}
```

Special characters

The rules for reserved template characters are the same as for ASCII templates, i.e. you must escape all special characters ($ or n), which are supposed to appear as such, in the fixed text.

Special HTML characters in the fixed text must be defined in an HTML conform way (e.g. &lt for <). Transformations are done only for the text read from the database.

In order to suppress HTML conversion, one may define an ASCII template instead. In order to suppress HTML conversion partially, one may modify the generated function code or define an explicit function.

New lines

New lines do not have any effect on the generated HTML page, but may make the generated code more readable. The template does not create line breaks
 or paragraphs <p> from new lines. Those must be defined explicit in the fixed text as all the other HTML tags.

2.1.1.3 Template specifications

In most cases, templates do have a `PROCESS` section, only. Templates may have, however, also a `VARIABLE` section. `ON_ERROR` and `FINAL` sections are not supported for templates.

General structure

ASCII and HTML templates can be defined as shown in the example below

Note, that an HTML template must not contain the `</process>` sequence as fixed text in the process section, since this will terminate the process section. This problem can easily be solved by calling a separate ASCII template, which just produces the `</process>` sequence or by using `>` instead of > and `<` instead of <.

In general, it is suggested to use ASCII templates rather than HTML templates.

```
// ASCII template
  $template string Test()$
  $VARIABLES$
    int       count = 0;
  $PROCESS$
    ... template text
  $END$

// HTML template
  HTML templates look a little bit different like:
  <template>
    <header> string Test() </header>
    <variables>
      int       count = 0;
    </variables>
    <process>
      ... template text
    </process>
  </template>
```

Template body

Template text (fixed text) can be entered in the template body (process section). Template text contains fixed text and embedded code.

Fixed text to be displayed in the template output is any sequence of characters (including fill characters as blanks or line breaks) except sequences enclosed in $... $, which are called embedded code. There are three types of embedded code expressions.

Output expression

Output expressions are operands (usually database property names), which are enclosed between $...$ (see example). Thus, one may enter template call for other templates or operations of any complexity in an output expression, but no statements terminated by semicolon.

The content of an output expression is directly written to the target string. In an HTML environment, it is converted to HTML before.

```
$first_name$
$responsible(0).first_name$
$responsible(0).first_name +     + responsible(0).second_name$
```

Embedded code

Embedded code does not directly create output, but is executed as expression code. Embedded code must be enclosed into ${...}$ (see example below).

Since code may contain `WriteResult` calls, code may also add data to the template result. This is one way to suppress HTML conversion for special texts, which have already HTML format.

Within embedded code one may refer to template variables defined in the variable section, to object variables, parameters and global variables like in an ordinary function.

```
${
  while ( messages.next )
    WriteResult(message.text,false); // no HTML conversion
}$
```

Control sequences

Control sequences are special expressions to control the text generation. Control sequences work similarly to the corresponding function constructs.

return

The return sequence is required, when the template is going to return a value as defined in the template header. The operand for the return value defines the value returned to the caller. No more text is generated after the return has been executed.

```
$return operand$
```

if else end

The if sequence defines a feature for conditional template generation: The else block is optional, but the end must be defined in any case. Note that line breaks after the condition become part of the fixed text and lead to line breaks in the template result. Thus, sometimes line breaks must not be inserted:

```
// each line create a line break
$if condition$
  Text generated when condition is true
$else$
  Text generated when condition is false
$end$

// conditional text without line break
$if (sex ==  male )$Mr. $else$Mrs. $end$
```

Switch case end

The switch block is an enhanced feature for conditional processing, since it allows defining any number of processing path. In contrast to function switch, template switch provides alternatives and does not support a default block. As well as for if-else, the end statement is mandatory.

In order to handle other (or default cases), the switch block may contain a `default` statement. The `default` statement is mandatory.

```
// switch by constants
$switch ( hair_color )$
$case  blue $
  Blue is not an accepted hair color. Use \ other\ , instead.
$case  yellow $
  Yellow is not an accepted hair color. Use \ other\ , instead.
$default$
  $hair_color§ is a valid value
$end$


// switch by operands
$switch condition$
$case operand$
  Text generated when the case operand matches the switch
$case operand$
  Text generated when the case operand matches the switch
$default$
  Text generated for other cases
$end$
```

while and for

While and for allow defining loops over arrays or collections. Similar to the conditional processing, an end statement is required in any case. One may also insert embedded code to change the condition: The same way one may define for-loops according to the for-syntax defined for OSI functions.

```
// simple while loop
$while messages.next $
    Message is: $messages.text$
$end$

// increase loop count within expression
${ rcount = 0; }$
$while rcount < 10 $
  Number is: $rcount$
  ${ ++rcount; }$
$end$

// for loop
$for rcount = 0; rcount < 10; ++rcount $
  Number is: $rcount$
$end$
```

Embedded code

In order to embed code within an OSI template one may use `${ statements }$` constructs. Each statement within the expression has to be terminated by semi-colon (also single statements). in contrast to variable or operation path references (`$operation$`), which add the (return) value to the template result, embedded code does not add anything to the template result string (except when calling `WriteResult()` within embedded code).

```
// embedded code
${
  ph.next;
  ++rcount;
}$
```

2.1.1.4Template result

The template result corresponds to the output created by `WriteResult`. The `WriteResult` function appends the text to the result string, i.e. it will collect the output from several templates.

Global template string

Template strings are thread variables, i.e. they are created separately for each thread. There is, however, only one template string for each thread, which can also be accessed as global variable `__template__result_`

It is, however, not suggested to refer to the template result via the global variable name, since the global variable name might be changed. A better way is referring to the template result via the functions describes below.

```
VARIABLES
  global    string   __template__result__;
```

SystemClass support

The **`SystemClass`** provides some functions in order to support handling the template result.

WriteResult

The `WriteResult()` function will append the data passed to the template string. Non-string values are converted to string according to the common conversion rules.

Write result supports converting data passed in the first parameter to HTML compatible data by converting HTML ^specific characters (e.g. < to < or & to & etc.)

During document generation data is passed to the document converter, which also converts Qt-HTML to Open Document standard. Qt-HTML is created when entering data in rich text edit control in the GUI frame work, i.e. most large text fields may contain Qt-HTML formatted text.

`WriteResult()` will be called automatically when running OSI templates. One may also call the function from within other OSI expressions.

```
// OSI
  WriteResult(data);      // appends content of data to template
  WriteResult(data,false);//  same as above
  WriteResult(data,true); // HTML convetion befor append data
```

Notes: When the function name is also a class member function it has to be prefixed with the **`SystemClass`** scope.

Reset template result

The `ResetResult()` function will clear the template string. The application is responsible to reset the template string at the beginning or when terminating the processing.

```
// OSI
  RestResult();
  MyTemplate/(; // run OSI template
  File.Out(TemplateString()); // write to file
```

Get template string

`TemplateString()` is a function that returns the template result as string value. Calling `TemplateString()`, one may display the template result or write it to file.

One may also call `TemplateString()` in order to add data directly to the template string or to reset the template string: The application is responsible to clear data in the template string when no longer being used, e.g. by calling `ResetResult()`.

```
// write template result to console
  Message(TemplateString);

// write template result to file
VARIABLES
  FileHandle    file;
PROCESS
  file.open(path,AccessModes::Write);
  file.append(TemplateString());
  file.close();
  ...

// append data to template string
VARIABLES
  string       &tsring &= TemplateString() // template string reference
```

```
PROCESS
  Message(tstring);
  tstring = '';
  Message(tstring);
  tstring += 'new value';
  Message(tstring);
```

2.1.1.5Debug templates

In order to debug templates, `OSI_DEBUG` option has to be set to **YES** (or **true**). Since templates are executed similar to OSI functions, one may also set break points in the template code. Since breakpoints are code, the most save way to define a break point is using embedded code. Since breakpoints are allowed in statements, only, one may add embedded code containing any stupid statement and setting a break point for this statement (see example below).

Debugging templates may become a little bit complicate, since errors are detected in the function generated from the template. Hence, the line numbers will not fit exactly to the template position. Even templates containing big amount of expression code may cause problems.

Those can be solved partially by viewing the system output, since when detecting an error in the generated code, OSI automatically writes the generated code to the system output.

```
$template void fragment$
here we will show the function result from ${# 1;}$$myFunction(ph)$.
```

2.1.1.6Creating documents by means of OSI templates

When running OSI templates, the result created by the template is written to a thread-global variable. This variable might be accessed from within the template or OSI function in order to write the result to a file (**`SystemClass::`**`TemplateResult()`) When not creating the output file by the OSI template or function, the template result might be retrieved by the application (see example below). In order to clear the template result, **`SystemClass::`**`ResetResult()` should be called before calling the OSI template.

The example below shows a simple fragment for calling an OSI template (e.g. for generating HTML documents).

```
bool ...fragment(Property &ph,odaba::String filename) {
odaba::String      html_template_name;
// set option variables requested by templates
  if ( ph.positioned() )
    Option("ItemLoid") = ph.instanceLoid();

  html_template_name = GetHTMLTemplateName(); // provide HTML template
name
  ph.executeExpression("SystenClass::ResetResult()");
  ph.executeExpression(html_template_name);
// write to file:
ph.executeExpression("SystenClass::TemplateResult()").toString();
}
```

2.1.2 Open Document templates

Document template are an enhance OSI template feature for using OSI within Open Document Standard documents. Several examples for document templates have been provided with the installation in the template folder (**..odaba/template** under MS Windows or **/usr/share/odaba/template** under Linux). When this directory is not available, templates might be downloaded from

www.odaba.com/content/downloads/demos/DocumentTemplates.zip

In order to create document templates, the document has to support specific styles, which are provided in **OSI-DocumentTemplate.ott**, which is also available in the template directory (or .zip file). The **SampleTemplate.odt** file contains a list of sample expressions for demonstrating template specifications for different template types with comments explaining the usage.

Documents generated from document templates are accessible by LibreOffice, but also by MS Office (with Open Document support).

2.1.2.1 Creating a document templates

Document templates are Open Document Standard documents (typically created with LibreOffice). Document templates require a number of specific paragraph styles, which must be available in the document template. Several styles are used for marking different template elements:

- **OSICode** - OSI code paragraphs
- **OSIComment** - template comments
- **OSIEnd** - end of an OSI template expression or block
- **OSIExpression** - OSI function header not creating document text
- **OSITable** - Table expression supporting row iteration
- **OSITableRow** - Row expression for defining single rows in mixed tables
- **OSITemplate** - Document template header
- **OSIVariables** - Template variable section

The style specification does not matter, but style names have to be defined as such. Moreover, list and graphic styles have to be provided in order to display embedded graphics and lists, which might be part of Qt rich text data stored in large text fields:

- **ListDefaultBullet** - display bullet lists
- **ListDefaultLowerCase** - display lists with lower case items
- **ListDefaultUpperCase** - display lists with upper case items
- **ListDefaultNumbering** - display lists with numbered items

- **GraphifDefaultCenter** - display centered graphics
- **GraphifDefaultLeft** - display left float graphics
- **GraphifDefaultRight** - display right float graphics

Template style specifications are provided in the **SampleTemplate.odt**, which might be copied, but also in the **OSI-DocumentTemplate.ott** file, which may be used directly for creating a new template. Style specifications might be changed as long as style names are not touched.

Notes: Document templates have been tested using LibreOffice, only. It is suggested to create document templates with LibreOffice.

2.1.2.1.1 Template expression

Template expressions within a document template start with a template header (similar to the function header for OSI functions). For the template header line (1) **OSITemplate** has to be assigned as paragraph style. The end of the template is indicated by an end statement. The paragraph style **OSIEnd** has to be assigned to the `end` (2) statement.

Any text outside template definitions will be ignored and does not become part of the generated document. Text after the template header line and before the `end` line becomes part of the generated document.

Document templates have to start with a main template or expression. The name of the main template has to be the same name as the template document name (without extension). Hence, template document name must not contain spaces or other special characters. One may also refer to a different entry point, which is defined as template in the template document. When referring to another entry point name, the entry point to the template must be declared when calling the template.

Template expressions may refer to option variables, which have been set in the configuration or ini-file or by the calling program. Option variables are referred to by name enclosed in %...%.

In order to support function variables, template variables (3) may be defined similar to OSI function variables in the `VARIABLES` section. In contrast to OSI functions, template variables do not require a special section, but the paragraph style **OSIVariables**. Variables have to be defined immediately after the template header. Variables must not be defined after first text or code line. As long as variables are not defined as global variables, those are defined in the scope of the template expression, i.e. between header and end, only.

After variable definitions template text, text references and template code may be defined. Defined fixed text (4) will be copied to the document with text formatting as being defined in the template (what you see is what you get). In order to invoke text from the database, operation paths (5) may be defined enclosed in $...$. For displaying the text provided via the operation path the current text formatting as being used for the operation path will be used. Any paragraph style except reserved **OSI...** styles may be assigned to fixed text and operation path references

Code lines (6) within the template function have to be defined with paragraph style **OSICode**. When defining code lined, no text will be generated from the code line. In order to generate text

Several specific types of template expressions are supported in addition:

- Template function - describes a function within document template
- Table template - describes a table with several rows
- Row template - describes a row in a composed table

```
(1) void SampleTemplate ()
(3) global int            count = 0;
(3) SET<HierarchyTopic>  &topics = HierarchyTopic(%ItemName%);
(4) This is my fixed text
(5) $topic(0).definition.definition.characteristic$
(6) topic.SubTopicTable();
(2) end
```

Notes: Changing text formatting within an operation path reference may cause errors in the generated document.

Template functions

Template expressions are used for executing a number of OSI statements. Often, template functions are used for calling subordinated template expressions or for calculating derived values.

Template functions start with a function header line, which requires the **OSIExpression** paragraph style. Template expressions are terminated with an `end` statement, which requires an **OSIEnd** paragraph style.

Any spaces or text within a template function will be ignored. Code lines within the template function have to be defined with paragraph style **OSICode**.

```
collection void DSC_Topic::SubTopicTable()
if ( tryGet(0) )
  if ( sub_topics.count > 0 )
    sub_topics.SubTopic1Table();
end
```

Table template

Table templates are used for displaying instances of a collection in table rows. Table templates (1) start with a table template header line, which requires the **OSITable** paragraph style. Table templates are terminated with an `end` statement (2), which requires an **OSIEnd** paragraph style.

Table templates are collection templates and will iterate for the last row found in the table definition. Table headlines (2) can be defined as first row(s) in the table. All table lines except the last (3) one are considered as static lines and will appear only once in the table. No text replacement will be done in static table lines, i.e. static table lines may refer to fixed text, only. The last table line (3) will be repeated for each instance of the collection referenced by the calling object.

A table template may contain an initial section before table begin, which will not iterate. Since no object instance is selected in the initial section (paragraph, headline), those may contain global, static or parameter data, only.

Code lines may be inserted before the first and after the last table row as well as in table cells. Code lines within the table template have to be defined with paragraph style **OSICode**. Code lines (4) immediately before and after the table will be executed for each table row generated for the last table row. Thus, this code can be used for calling other table row templates in order to create complex tables. Table

templates may contain the table definition and expression code, but should not contain text paragraphs or multiple tables.

In case of defining multiple row table templates, all rows are created for each instance in the collection. Conditional rows can be created by using expression (see example below)

```
(1) collection void DSC_Topic::SubTopic1Table()
(2) Title 1               Title 2
(3) $definition.name$     $definition.definition.characteristic$
(4) if ( definition.definition.example.count )
(4)   DSC_Topic::Example();
(4) sub_topics()->SubTopic2Table();
(2) end
```

Notes: Table templates should be marked as collection functions. Otherwise, template call may fail, when no instance is selected in the calling property handle.

Control table line count

Usually, all instances will be printed to the table. In order to restrict the number of lines to be printed, a filter might be set for the collection. One also may pass a count (`__count`) to the table template. The parameter has to be defined an integer (INT) in the table template headline and has to be set by the calling template.

When defining a `__count` parameter in the template head line, generating table lines starts with the instance currently selected in the collection. The `__count` parameter contains the number of instances to be processed.

When the table template returns, the next instance after the last printed to the table is positioned. When the end of the collection had been reached, no instance is positioned in the collection.

```
persons.first();      // position first instance in collection
while ( persons.positioned() )
  persons.Table2(10); // print next ten lines as long as table returns
true
...
collection void Person::Table2(int __count) // table template header
...
```

Row template

Row templates are usually called from within a table template in order to add additional rows for the selected object instance. This allows creating complex tables with different row formats. Row templates (1) start with a row template header line, which requires the **OSIRow** paragraph style. Row templates are terminated with an `end` statement (2), which requires an **OSIEnd** paragraph style.

The row (3) defined in the row template should fit into the table definition calling the row template. Row templates are used for creating tables with different generic

rows. A row template should contain only one row, which fits into the table calling the row template. Only the first row specification in the template will be considered as row template. Additional rows or paragraphs will be ignored.

In order to call further row templates, code lines may be inserted before and after the row as well as in row cells. Code lines (4) within the table template have to be defined with paragraph style **OSICode**. Code lines (4) may be used for calling subsequent row templates.

```
(1) void DSC_Topic::SubTopic2Table()
(3) $definition.name$         $definition.definition.characteristic$
(4) if ( definition.definition.example.count )
(4)   DSC_Topic::Example();
(2) end
```

2.1.2.1.2 Fixed text and text references

Templates typically refer to fixed text and text references. Moreover, one may use template conditions or template iteration in order to create conditional or iterative text output.

Fixed text

Fixed text will be copied to the target document using styles and local text formatting as being defined for the fixed text in the document. Any paragraph or text style defined for the document template may be used for fixed text, except OSI styles, which are reserved for marking OSI template elements. One may, however, refer also to list and image default styles provided in the template.

Text reference

In order to refer to text data provided by the database, one may define text references, which are operation paths enclosed in $...$. An operation path may refer to a single text field, but also to an operation. When referring to an operation, the operation result will be displayed. Operations referenced as text references should return elementary data (string, number date etc.).

Values which are not string types are converted to string values. All string values are converted to UTF8. Reserved XML characters as < and > are translated to corresponding XML values(> < etc).

Special support is provided for Qt rich text fields, which are usually stored in the database when editing text in rich text edit controls (GUI framework). Text formatting of rich text edit fields is converted to Open Document Standard text formatting, i.e. appropriate styles are selected or created when generating the document. .

When generating documents including Qt rich text edit fields, text formats and styles used in the formatted text may conflict with styles used in the document template. In general, paragraph styles are taken from the document template, while local formats (spans) are translated to document styles as long as possible.

Document templates support unordered and ordered lists defined within a rich text field as well as embedded graphics. Embedded graphics are copied to the document's image folder. Tables in rich text fields are not yet supported. Also special formatting options as line or block indent will be ignored. As long as rich text fields do not use very sophisticated text formatting, they are converted nearly 1:1 to the document.

When defining text references, only one style should be assigned to the text reference. Changing the style within the text reference may cause errors in the generated document.

```
$topics.topic(0).definition.name$
$topics.topic(0).GetExampleText()$
```

Conditional text output

In order to generate conditional text output, one may insert code lines defining the condition. This may cause problems, since text after a code line always starts a new paragraph. In order to create conditional text within a paragraph, template conditions may be defined. Template conditions must not be marked as **OSICode** and must not contain local formatting.

This is, however, rather sensitive, since any additional character as format information etc. will destroy the template expression. Hence, it is suggested to call a template expression, instead.

```
Dear $if sex == male$Mr.$else$Mrs.$end$ $last_name$,

Comment: better solution
My name is $GetTitle$ $last_name$.

Comment: template expression
STRING GetFirstName()
if ( sex == male )
  return ( "Mr." );
else
  return ( "Mrs. ");
end
```

Template iteration

In order to combine text elements within a text paragraph one may call template while construct. Since this is difficult to read, because all statements have to be written in one line, it is suggested to cal an OSI functions defined in the template, instead. The same solution is suggested for switch blocks an other more complex string computation.

```
My name is$while first_name.next$ $first_name$$end$ $last_name$.

Comment: better solution
My name is $GetFirstName$$last_name$.

Comment: template expression
STRING GetFirstName()
return ( first_name[0] + ' ' +
         first_name[1] + ' ' +
         first_name[2] );
end
```

2.1.2.1.3 Comments in document template

Any text outside template definitions is considered as comment. In order to add comments within template definitions, **OSIComment** style may be assigned to comment paragraphs. Comment are completely ignored and will not be written to the document, but also not to the template expression created from the document template.

2.1.2.2 Generating OO documents

In order to call LibreOffice document templates from a context action, a few option variables have to be set before calling the function. Generating LibreOffice documents requires the **OpenOffice** library, which is provided as dynamic link library with the ODABA installation.

The document generator is called via the **`SystemClass::`**`CreateDocument()` interface function by using an OSI expression. In order to be executed correctly, the function has to be executed with a property handle as calling object. In order to call the function properly, the complete document path and the complete template path have to be provided in internal option variables `__DocumentPath` and `__TemplatePath`. It is up to the application to provide proper location and file names.

Beside these mandatory option variables, usually a number of additional option variables has to be set, which are referenced in the document template. Since document templates are global functions, root object instance for the document are usually passed as instance identifier (loid or key value) or as access path.

The example below shows two fragments for an implementation of a context action for executing LibreOffice document generation.

```
// generate document in current process
bool ...fragment(Property &ph) {
// root object instance is selected in ph
// the following options are required by the document generatoin
interface
  Option("__DocumentPath") = GetDocumentPath(); // complete document path
  Option("__TemplatePath") = GetTemplatePath(); // complete template path

// set other option variables requested by the template
// ...
// running in current process
  ph.executeExpression("SystenClass::CreateDocument()");
}

// generate document in separate process
bool ...fragment(Property &ph) {
// root object instance is selected in ph

// create ini-file for CreateDocument
  fstream ini_file;
  ini_file.open ("test.ini", fstream::out | fstream::app);

  ini_file << "[SYSTEM]" << std::endl;
  ini_file << "DICTIONARY=" << Option("SYSDB").toString().data() << endl;

  ini_file << "[CreateDocument]" << std::endl;
  ini_file << "DICTIONARY=" << Option("SYSDB").toString().data() << endl;
  ini_file << "RESOURCES=" << Option("RESDB").toString().data() << endl;
  ini_file << "DATABASE=" << Option("DATDB").toString().data() << endl;
  ini_file << "ONLINE_VERSION=YES" << endl;
  ini_file << "ACCESS_MODE=Write" << endl;
  ini_file << "NET=YES" << endl;
  ini_file << "ODABA_ROOT=" << Option("ODABA_ROOT").toString().data() <<
endl;
  ini_file << "CTXI_DLL=" << Option("CTXI_DLL").toString().data()<< endl;
  ini_file << "TRACE=" << Option("TRACE").toString().data() << endl;
  ini_file << "DSC_Language=" << Option("DSC_Language").toString().data()
endl;
// create option variables for template options
  ini_file.close();

  odaba::String   path(Option("ODABA_ROOT"));
  path += "/CreateDocument.exe";
// depending on template reqirements additional optione might be set
  ph.instanceContext().executeProgram(path,"test.ini",GetTemplatePath(),G
etDocumentPath());

  return true;
}
```

2.1.2.3Debugging document templates

There are several reasons for errors when specifying document templates. This chapter explains typical error cases and different ways to solve the problem.

Since Document templates are converted first to OSI templates, which, again, are translated to OSI functions, error lines reported while running the template refer to OSI function lines rather than to document template lines. This may cause additional problems when trying to locate errors.

In order to avoid this problem, it is suggested to implement larger OSI functions in the resource database. The OSI functions may be checked before running, which reduces the risk of errors. This, however, is not possible for document templates of any kind. Hence one should try to reduce the code lines in document templates in order to keep templates transparent.

Names for document templates and expressions

When defining document templates as external resources, naming conflicts may easily happen, when calling the document template not via external program call but via internal function call. In order to optimize OSI function loading, OSI functions will be cached by the dictionary when being loaded. This will improve the performance, when calling document templates several times.

When the application generates different documents referring to global or class templates with the same name, loading the second document template will fail, since an OSI expression with the same name has already been loaded.

In order to reuse OSI functions, which are referenced in several templates, OSI functions might be implemented in the resource database. One cannot, however, store document templates in the resource database. Hence, document template names should be prefixed in order to obtain unique template names.

Syntax errors

Syntax errors are detected while converting the document template into an OSI template, while translating OSI templates into OSI functions and while loading OSI functions to the dictionary. In order to detect errors, it is suggested running `CreateDocument.exe`. When a syntax error has been detected, the console output will list the OSI function or template expression, which has been failed. Below the error message, the converted OSI template or function is listed, which usually allows to locate the problem (see example below).

```
Creating Document

2012-01-26 19:46:00 - Running L:\odet\CreateDocument.exe with:
  ini-file: l:\opa\tpl\ReferenceDocu.ini
  document:
  template:
Loading document template ... SOS Error :
Error at line 9, column 1
No match for 'basic_stmt' at: ... }$
   in: istatement
   in: statement
   in: block
   in: comp_expr
   in: imbedded_expr
   in: templ_string
   in: templ_text
   in: t ...

$template void ODC_Module::ReferenceDocuClasses()$
<text:h text:style-name="Heading_20_2" text:outline-level="2">
  Implementation classes
</text:h>

${
classes()->ReferenceDocu()
}$
$END$
```

Invalid operation

Referring to unresolved operation names is a typical error, which is rather easy to locate. Error template name, class and line causing the problem are listed in the console output (see below).

```
Creating Document

2012-01-26 20:30:29 - Running L:\odet\CreateDocument.exe with:
  ini-file: l:\opa\tpl\ReferenceDocu.ini
  document:
  template:
Loading document template ... Generate document ... DEBUG>PROCESS
DEBUG>run
-- Error in: collection void SDB_Member::ReferenceDocu(STRING title)
   at line 14, column 1: RDTopicTeyt(DataTypeCString);
   reason: operand or operation 'RDTopicTeyt' not defined in:
SDB_Attribute
```

Detecting malfunctions

In order to locate malfunctions one may run document generation in OSI debug mode. In order to activate the debug mode, `OSI_DEBUG=`**`YES`** has to be added to the `[CreateDocument]` section in the ini-file before calling `CreateDocument`.

In order to break at certain positions, one may insert break points in the document template in two different ways. In order to set a break point within a code line (with

paragraph style **OSICode**) one may simply insert an # character at first position of the code line. In order to set a break point within fixed text references, a statement has to be inserted, since break points can be set for statements, only. Usually, one simply inserts a dummy statement as ${# 1;}$, which contains a break point and debugging will stop at the marked text position.

```
Comment: break point in OSI code line
collection void SDB_Resource::RDTopicText(STRING types)
if ( positioned )
#  if ( resource_ref.tryGet(0) )
    resource_ref.description()->RDTopicText2(types);
end

Comment: break point in fixed text
2.3.1 $sys_ident$ - ${# 1;}$
$resource_ref(0).description(0).definition.name$
```

Document error

Somtimes, the document may contain invalid characters or unbalanced XML tags. When loading the document fails, LibreOffice write line number and position, where the error occurred. In order to locate the invalid text, one may change the extension for the document to .zip and browse the zip file content.

In the zip-file directory, the is the content.xml file, which contains the invalid line. Just open the content.xml file with a text editor supporting line numbers and locate the reported position. Usually, content errors are obvious when editing the content file.

2.1.3 MS Office document generation

A typical way for generating MS Office documents is using MS Word macro features. The **Terminus** application provides document generation actions, which may work, however, only when the required document templates have been installed. Document templates are not part of the ODABA installation and might be modified or rewritten according to specific requirements.

Typically, MS Office document templates refer to an initial document, which provides style definitions and title page for the document to be generated. This might easily be replaced by a more appropriate one. Moreover, styles might be changed, but not the style names, which are referred to from within the macros. Available macros and its resources are described below.

In order to support scripting languages as MS VB Script, ODABA .Net libraries have to be extended by a wrapper library. The .Net project and other resources required for MS document generation are available at following locations:

- MS Word helper functions

http://www.odaba.com/content/downloads/demos/odabaWordHelper.zip

- Document template

http://www.odaba.com/content/downlads/demos/DocumentTemplates.zip)

Those are just demos for schowing, how to get out some documents from an ODABA database. On the other hand, these demos are used by RUN for generating documentation from Terminus specifications. Usually, document templates are installed in the template directory (**/usr/share/odaba/template** under Linux and ...**odaba/template** under MS Windows).

Document templates are provided as **.dot** files for reference documentation (**ReferenceDocu.dot**), Terminology Model documentation (**TerminologyModel.dot**) and hierarchical topic documentation (**TopicsDocu.dot**). Document style definitions are provided for these templates in .doc files with appropriate names.

Notes: MS Office document generation by means of VB Script macros is one possible way, but it is rather slow and difficult to maintain. A better way is using Open Document templates, which generate documents that are accessible in MS Office as well as in LibreOffice.

MS Word helper functions

The MS Developer Studio 2010 solution provides an ODABA wrapper supporting ODABA database access and a few MS Word function for opening and closing word documents. Before compiling the solution, references for odaba-net.dll and dotnet-connector.dll have to be updated.

The wrapper library works with all MS Office versions from office 1997-2003 upwards. It has not been tested with older versions.

When opening a document (**`ODocument::`**`Open()`), an ini-file is required, that contains document and template name. The ini-file is, usually, generated when calling MS Word macros from within **Terminus**. Otherwise, an ini-file has to be provided, which contains a path the document to be created (option name passed in *`docname`*) and a path for a template for initializing the document (option name passed in *`templatename`*).

Two more functions (**`ODocument::`**`Find()` and **`ODocument::`**`ReplaceText()`) are available for convenience.

The odabaDBInterface file provides the ODABA database access function wrapper for accessing the database (**`ODatabase`**), for property handle support (**`OProperty`**) and for value access (**`OValue`**).

Terminology model template

The terminology model template provides a document template (TerminologyModel.dot) for generating an MS Word document for a terminology model defined in **Terminus**. (context menu for a terminology model in the Models tree **Generate Documents/Generate Word**). The action generates an ini-file and calls the word macro from the location as being defined in option `Options.Documentation.HTWordTemplate`.

One may also start the macro without running Terminus, but the ini-file has to be provided manually, in this case. An example for an ini-file is shown below (remove comments before running the ini-file).

Before running the template, one might update the initializing document TerminologyModel.doc in order to get a more appropriate document design. The document delivered is designed for generating ODABA documentation and includes specific title and RUN logo.

```
[SYSTEM]
DICTIONARY=odaba\ode.sys     - ODABA system dictionry

[DOCU]
DICTIONARY=odaba\ode.sys     - ODABA system dictionry
RESOURCES=odaba\ode.dev      - ODABA resource database
DATABASE=sample.dev          - my development database
ONLINE_VERSION=YES
ACCESS_MODE=Write
NET=YES
ODABA_ROOT=odaba
CTXI_DLL=AdkCtxi
TRACE=... logfile directory
DSC_Language=English
DOC_PATH=odaba\Projects\Sample\doc\TM.doc - final document location
DOC_TEMPLATE=odaba\template\TerminologyModel.doc - document
initialization
START_TOPIC=TM               - terminology model selected
```

Hierarchy topics template

The hierarchy topics template provides a document template (HierarchyTopicsDoc.dot) for generating an MS Word document for a topic hierarchy defined in **Terminus**. (context menu for a terminology model in the Themes tree **Generate Documents/Generate Word**). The action generates an ini-file and calls the word macro from the location as being defined in option `Options.Documentation.HTWordTemplate`.

One may also start the macro without running Terminus, but the ini-file has to be provided manually, in this case. An example for an ini-file is shown below (remove comments before running the ini-file).

Before running the template, one might update the initializing document HierarchyTopicsDoc.doc in order to get a more appropriate document design. The document delivered is designed for generating ODABA documentation and includes specific title and RUN logo.

```
[SYSTEM]
DICTIONARY=odaba\ode.sys     - ODABA system dictionry

[DOCU]
DICTIONARY=odaba\ode.sys     - ODABA system dictionry
RESOURCES=odaba\ode.dev      - ODABA resource database
DATABASE=sample.dev          - my development database
ONLINE_VERSION=YES
ACCESS_MODE=Write
NET=YES
ODABA_ROOT=odaba
CTXI_DLL=AdkCtxi
TRACE=... logfile directory
DSC_Language=English
DOC_PATH=odaba\Projects\Sample\doc\MainTopic.doc - final document
location
DOC_TEMPLATE=odaba\tpl\HierarchyTopicsDoc.doc - document initialization
START_TOPIC=MainTopic        - start topic in the topic tree
```

2.1.3.1 Call creating MS Office document

In order to call creating an MS Office document, MS Word has to be invoked. Since the technology for creating MS Office documents is based on MS Word macros (VB Script), the template document has to be called and executed. Since this is a different process, an ini-file has to be created and passed to the MS Word template (macro).

All information requested is passed via an ini-file, which is usually hard-coded in the document template macro. The document template examples provided in the **tpl** directory of the installation folder shows how to open a database by means of an ini-file.

When calling an MS Office template (.dot), the location for the template file has to be passed to the function call (`GetTemplatePath()` is just a symbolic function call in the example, which returns the complete path for the document template file). Other option file variables as location for output file or root object instance for document have to be set in additional option variables as requested by the document template.

```
// generate document in separate process
bool ...fragment(Property &ph) {// root object instance is selected in ph
// create ini-file for CreateDocument
  fstream ini_file;
  ini_file.open ("test.ini", fstream::out | fstream::app);

  ini_file << "[SYSTEM]" << std::endl;
  ini_file << "DICTIONARY=" << Option("SYSDB").toString().data() << endl;

  ini_file << "[DOCU]" << std::endl;
  ini_file << "DICTIONARY=" << Option("SYSDB").toString().data() << endl;
  ini_file << "RESOURCES=" << Option("RESDB").toString().data() << endl;
  ini_file << "DATABASE=" << Option("DATDB").toString().data() << endl;
  ini_file << "ONLINE_VERSION=YES" << endl;
  ini_file << "ACCESS_MODE=Write" << endl;
  ini_file << "NET=YES" << endl;
  ini_file << "ODABA_ROOT=" << Option("ODABA_ROOT").toString().data();
  ini_file << endl;
  ini_file << "CTXI_DLL=" << Option("CTXI_DLL").toString().data()<< endl;
  ini_file << "TRACE=" << Option("TRACE").toString().data() << endl;
  ini_file << "DSC_Language=" <<Option("DSC_Language").toString().data();
  ini_file << endl;
// create option variables for template options
  ini_file.close();

  odaba::String   path(Option("ODABA_ROOT"));
  path += "/CreateDocument.exe";
// depending on template reqirements additional optione might be set
  ph.instanceContext().executeShell("open",GetTemplatePath());
  return true;
}
```

Notes: In the example above, the the document template has to "know", where the ini-file has been stored.

2.2 Calling document generation from a command line

In order to start document generation from a command line, `CreateDocument` may be called:

...odaba/CreateDocument.exe ini_file [template] [document] [-q] [-h] [-p:type]

The configuration or ini-file contains the data source definition as well as specific options for running the document template. Especially options referred to by the template might be set in the ini-file (see example below). Data source definition options are described in Data Source Options. Propgramm parameters `DocumentPath` and `TemplatePath` might be set instead of passing *`document`* and *`template`* parameters Template options (e.g. `Collection` and `ItemName`) are option variables referred in the document template.

Command line document generation is rather helpful in order to detect document template options. In order to debug a document template, the `OSI_DEBUG` option has to be set in the `[CreateDocument]` section of the ini-file (`OSI_DEBUG=`**`YES`**).

Template files are internally converted into OSI template expressions. In order to check the generated OSI code, OSI templates may be stored to a file. The path (complete file path) for storing generated OSI templates has to be passed in `Options.Documentation.TemplateOutput`.

```
[SYSTEM]
DICTIONARY=L:\adk\ode.sys

[CreateDocument]
; data source specification
DICTIONARY=L:\adk\ode.sys
RESOURCES=L:\adk\ode.dev
DATABASE=L:\opa\opa.dev
ONLINE_VERSION=YES
ACCESS_MODE=Write
NET=YES
TRACE=e:\temp\reinhard
DSC_Language=English
; program environment
ODABA_ROOT=L:\odet\
CTXI_DLL=AdkCtxi
; debug options
OSI_DEBUG=YES
Options.Documentation.TemplateOutput=temp/ootemplate.osi
; program parameters
DocumentPath=L:\opa\doc\odabagui.odt
TemplatePath=l:\opa\tpl\ReferenceDocu.odt
; template options
Collection=ODC_Project
ItemName=odabagui
```