

Tobias Trapp

XML Data Exchange using ABAP

- ▶ Discover all aspects of XML data exchange: software development, specification and testing
- ▶ Maximize your use of all XML technologies in ABAP: XML Library, XSLT, and Simple Transformations (ST)
- ▶ Optimize the robustness of processes by validating XML documents using Java integration

XML Data Exchange using ABAP

Tobias Trapp

Contents

1 Introduction	5	3.4 Rendering XML Documents and Encodings	17
XML in E-Business	5	3.5 Validating Against DTDs	17
XML in Application Integration	5	3.6 Pitfalls Regarding Namespaces	20
Exchanging Data and Documents	5	Namespace Support of the XML Library	21
Structure of this Book	6	Identifying Elements with Namespaces	23
System Requirements	6	3.7 Event-Based Parsing	23
Acknowledgements	6	SAX Interface	24
2 XML Technologies and Data Exchange	7	Pitfalls	25
2.1 Important Standards of the XML Family	7	Using Rule Sets	25
XML 1.0	7	3.8 Useful Tips	26
XML Transformation Using XSLT	7	4 XSL Transformations	29
XML Schema for Data Modeling	8	4.1 Integrating XSLT into the ABAP Workbench	29
XML Framework in E-Business	8	4.2 Integrating XSLT in ABAP	31
2.2 Data Exchange in Business Processes ...	9	Calling XSLT from ABAP	32
XML in Integration Scenarios	9	Calling ABAP from XSLT	32
Data Exchange with External Partners	9	Exception Handling in ABAP Calls from XSLT	35
Conclusion	9	4.3 Transforming ABAP Data	36
2.3 Using XML Technologies in SAP Systems	10	Serializing ABAP Data	36
Application Integration	10	Sample Deserialization	37
3 XML Library	13	Realistic Scenarios	43
3.1 Important Interfaces	13	4.4 SAP-Specific Extensions	43
3.2 Data Sources	14	Numeric Functions	44
Encodings	14	String Functions	44
Unicode	15	XPath Operators and Path Expressions from XPath 2.0	44
3.3 DOM API	15	Commands for Nodesets	45
Response Scenario for a cXML Interface	16	Other Functions	45
		SAP-Specific Restrictions	45

4.5	XSLT 2.0 Support	46	5.7	Miscellaneous	77
	Grouping XML Elements	46		Literal Contents	77
	Definitions of XPath Functions and			Namespaces	77
	Conditional XPath Expressions	46	5.8	Useful Tips	79
	Several Input and Output				
	Documents	48	6	Java Integration	81
	Output Formatting	48	6.1	Validation Using JAXP	81
4.6	Generating Code	50	6.2	J2EE Infrastructure	84
	Text Templates	50		Session Beans	84
	Abstract Syntax Trees of XPath			Restrictions to the Use of Beans	84
	Expressions	51		ABAP-Java Integration	84
4.7	Useful Tips	52		Web Services	84
5	Simple Transformations	55	6.3	Creating a Web Service for	
5.1	Basic Structure	55		Validating XML Documents	85
	Subroutines and Parameters	56		Creating a Session Bean for the	
5.2	Accessing Data Objects	56		Validation	85
	Accessing Elementary Data Objects	56		Handling System Errors	86
	Data Roots and Data Nodes	57		Using JAXP 1.2	89
	Attributes	58		Deploying and Testing the	
	Structures	58		Web Service	90
	Internal Tables	59	6.4	Using Web Services through ABAP	93
	XML Representation of ABAP Data	61		Generating a Proxy Object	94
5.3	Variables and Parameters	61		Creating a Logical Port	95
	Assigning Variables and Parameters	62		ABAP Program for Calling the	
	Case Distinctions Using Variables	62		Web Service	95
5.4	Conditional Transformations	62	6.5	Discussing the Solution	96
	Optional Elements and Attributes	62			
	Preconditions, Conditions, and		7	Real-Life Scenarios	99
	Assertions	63	7.1	Designing Technical Processes	99
	tt:cond in Detail	65		Data Exchange in the Conflicting Areas	
	Case Distinction	66		of B2B and EAI	99
	Grouping	69		Message Services	99
5.5	Mappings	72		Using Established E-Business	
	Mapping Attribute	72		Standards	100
	Mapping with Case Distinctions	72		Proprietary Interface Formats	101
	Conditional Transformations with			Quality of the User Data	101
	Variables	73		Further Development and Versioning	
	Structural Mappings	74		of Interfaces	101
5.6	Modular Transformation Programs	76		Risk Management	102
	Subtemplates	76	7.2	DP Concept of the Process	102
	Including Transformations	77		Using the Strengths and Weaknesses	
	Calling External ST Programs	77		of XML Processes	102
				Choosing the Right Technology	102

Files as Data Sources	103	Note on the Design of	
Communication Protocols	103	the Sample File	108
Storing and Displaying XML		Part of the SPFLI Flight Data Model	112
Documents	104	A.2 Invoicing via cXML	113
Structure of the Data Transfer		Acknowledgement of an	
Interface	104	Invoice Receipt	113
7.3 Regression Tests	105	cXML-Based Sample Invoice	113
A Sample Scenarios	107	On the Probability of our Example	113
A.1 Sample Master Data Exchange		B Bibliography	117
Process	107	Index	119
Catalog of Offerings for Flight Meals ...	107		
Sample BMEcat Catalog	108		

4 XSL Transformations

Since Release 6.10 of the SAP Web Application Server (SAP Web AS), *XSL Transformations* (XSLT) have been integrated in ABAP via the `CALL TRANSFORMATION` command. When the XSLT processor was implemented, the current XSLT 2.0 specification was still under discussion. For this reason, a version was implemented that is based on the "W3C Working Draft," on April 30, 2002 (<http://www.w3.org/TR/2002/WD-xslt20-20020430>). Because we cannot assume that this working draft is generally well known, we'll compare it with the current version during the course of this chapter: the "W3C Candidate Recommendation" of November 3, 2005 (<http://www.w3.org/TR/2005/CR-xslt20-20051103>).

The importance of XSLT for data exchange stems from the fact that it is the most powerful and advanced technology available for the transformation of XML documents. XML data can be transformed into ABAP data structures and vice versa; however, XSLT is not limited to those types of output. You can also generate HTML documents or plain text files that are made available as loadable assets to other applications.

XSLT is widely used and well documented by a vast number of resources. Because you can easily integrate existing XML transformations in ABAP, you can also reuse existing XSLT-based data exchange solutions in an SAP system.

Elements and attributes in the tree structure of an XML document are addressed in a specific language: *XPath*

(<http://www.w3.org/TR/xpath>). In the following sections, we'll assume that you're familiar with *XPath 1.0* and we'll discuss only those extensions that SAP has implemented in the XSLT processor. Some of those extensions are only available as of SAP Kernel Release 6.20.

4.1 Integrating XSLT into the ABAP Workbench

You can create XSLT programs via Transaction SE80. To do that, you must go to the package view, then right-click on a package to open its context menu and select **Create • More... • Transformation**. An input dialog displays, as shown in Figure 4.1. Then you can edit the program in an editor.

Figure 4.2 shows a sample transformation that searches all elements of an XML document recursively and outputs a message for each A element. This example already shows some facts: The root element of the transformation is the `xsl:transform` element. This element has the same functionality as the `xsl:stylesheet` statement. Which command you want to use is purely subjective; however, it has become common practice to refer to transformations that format XML documents in a readable way as *stylesheets*, whereas in the data exchange context they are called *transformations*.

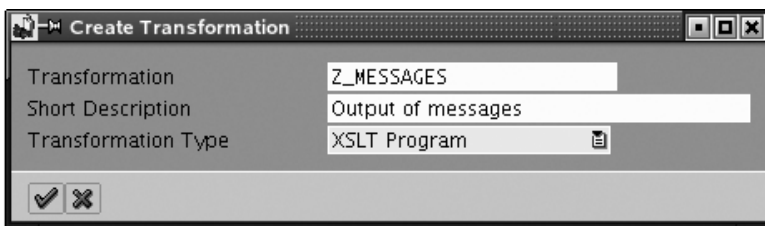


Figure 4.1 Creating XSLT Programs

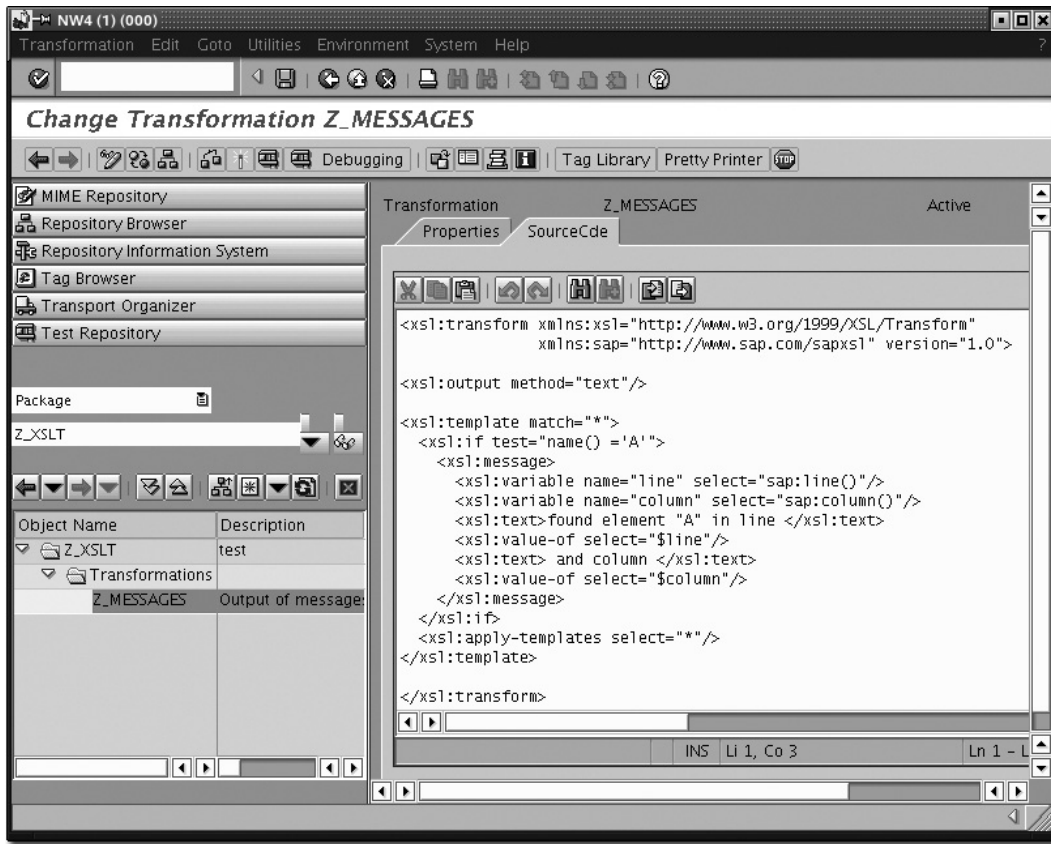


Figure 4.2 Sample Transformation

The `sap:line()` and `sap:column()` commands determine the line and column of the current context node in the original document. These commands are part of the `http://www.sap.com/sapxsl` namespace, which defines SAP-specific XSLT extensions. Section 4.4 discusses extensions in further detail.

We'll use the `sap:line()` and `sap:column()` commands in the sample transformation (see Figure 4.3). This transformation reads all elements of an XML document and outputs a message via the `xsl:message` command whenever an `A` element was found. The message specifies the position of the element in the input document.

You can test the transformation in the same way as you would an ABAP program. To do that, you must create a file with the following contents:

```
<?xml version="1.0"
  encoding="iso-8859-1"?>
<Root>
  <Level1>
```

```
    <A/>
  </Level1>
</A>
</Root>
```

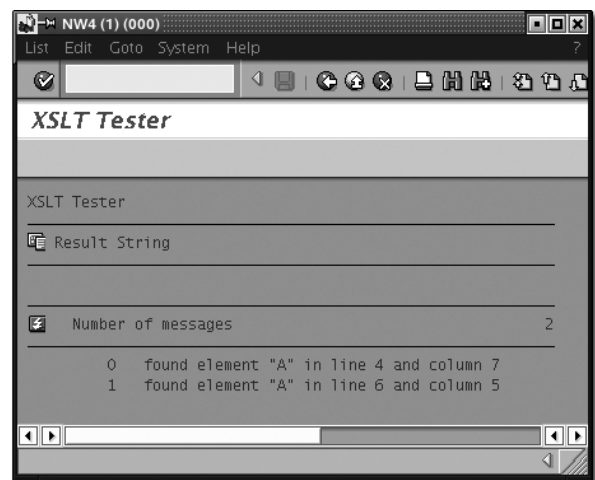


Figure 4.3 Output of the Sample Program

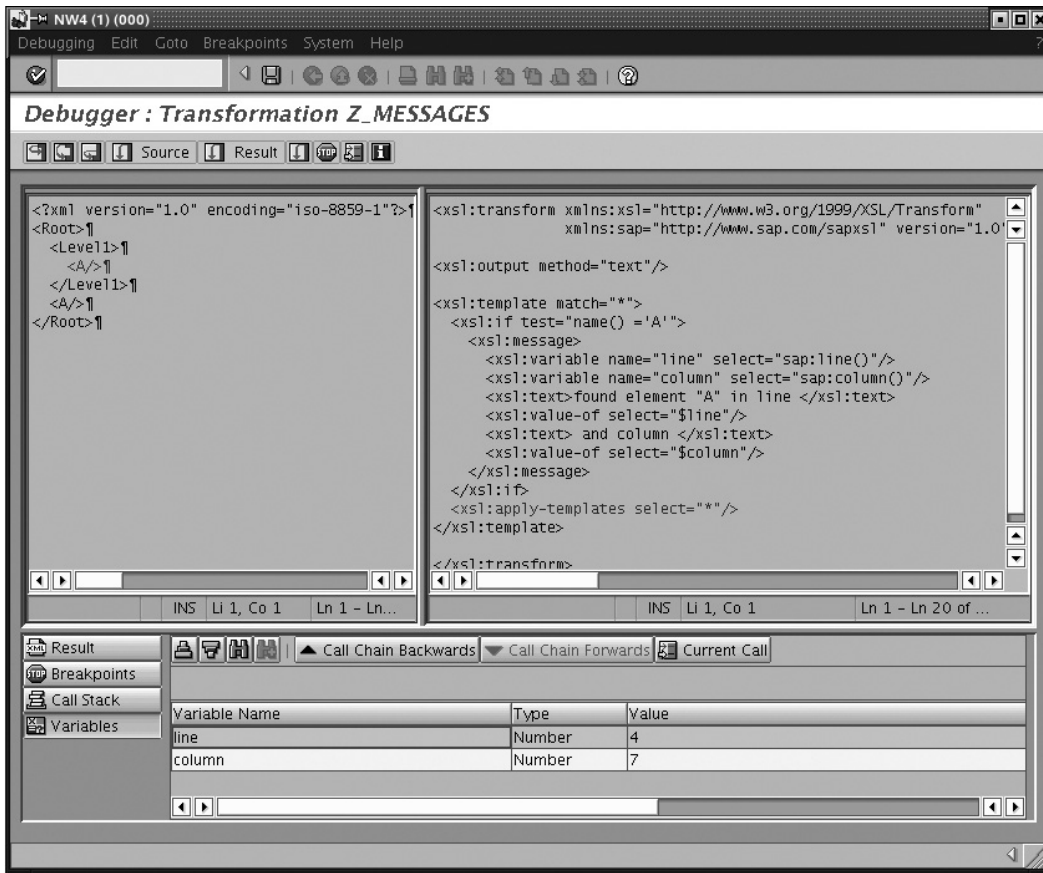


Figure 4.4 XSLT Debugger

When the *XSLT Tester* runs the transformation with the above input, it returns the output shown in Figure 4.3. You can call the XSLT Tester from Transaction SE80 by highlighting a transaction and clicking on the **Test XSLT Program** button.

You can also execute XSLT programs step by step. Unfortunately, you can only view the messages output via the `xsl:message` command when you use the XSLT Tester or the XSLT Debugger.¹

The debugger can also display the command line that is currently executed, the document position, and the values of variables. Moreover, you can set breakpoints.

¹ It is a general drawback of the XSLT specification that it doesn't describe where exactly the messages must be output. Consequently, each XSLT processor has its own output method. Strictly speaking, the SAP implementation also contains ABAP classes that can be used to evaluate the messages that have been output; however, we don't recommend that you use those classes.

Figure 4.4 shows an output of the debugger for the sample transformation.

The output of messages is necessary for testing the process and error behavior of a transformation. Because you can use messages for manual testing, Section 4.2 shows how you can do this via ABAP calls from XSLT.²

4.2 Integrating XSLT in ABAP

This section describes how to call XSLT programs from ABAP and alternatively, how to call ABAP from XSLT.

We'll use ABAP syntax as of Release 6.20 here. Language constructions that are considered "obsolete" are

² The `xsl:message` command is not very useful here. It often makes more sense to store logging information in separate elements in the XML output. You can find more useful information on test outputs and on debugging XSLT programs in the *XSLT Cookbook* (Sal Margano: O'Reilly 2005).

still supported for reasons of downward compatibility, however, they should no longer be used because there are better options available. Not only did the upgrade from Release 6.10 to 6.20 involve adding new concepts, but it also required the standardization of existing concepts. Developers should study these advanced developments so they can reduce the degree of complexity in the set of commands.

Calling XSLT from ABAP

You can call an XSLT program via the `CALL TRANSFORMATION` command. At this point, we only want to briefly describe the general syntax of the call. For further details such as exception handling, you should refer to *The Official ABAP Reference* (Horst Keller: SAP PRESS 2004).

```
CALL TRANSFORMATION transformation
    [PARAMETERS parameters]
    SOURCE source
    RESULT result.
```

The transformation source can be specified both statically and dynamically. You can transfer ABAP data objects to the transformation via `PARAMETERS`. Within the transformation, one of the following type specifications can be used for the parameter: `string`, `xstring`, `number`, `boolean`, `object`, or `nodeset`. The following section provides examples of how to transfer objects and use methods. To transfer nodesets, you must use an object of the `if_ixml_node` or `if_ixml_collection` type. This XML library interface was described in detail in Chapter 3.

Both the source and the result of a transformation consist of either ABAP data, which is available in the *asXML view* in the transformation, or XML documents.

The `CALL TRANSFORMATION` command enables you to transform ABAP data into ABAP data, XML documents into XML documents, and ABAP data to XML and vice versa. In the data source (`SOURCE` parameter) and target (`RESULT` parameter), XML documents are available as `string`, `xstrings`, or as reference variables of one of the following types: `if_ixml_istream`, `if_ixml_ostream`, `if_ixml_document`, `if_ixml_node`. You can also transfer ABAP data dynamically by using a variable of the type `ABAP_TRANS_RESBIND_TAB`.

Since Release 6.40 the `OPTIONS` parameter is available for the `CALL TRANSFORMATION` command. Among other

things, this parameter can be used for output formatting of XML documents.

Don't forget exception handling!

The `CALL TRANSFORMATION` command can trigger exceptions. Since Release 6.40, the common basic class `CX_TRANSFORMATION_ERROR`, is available for XSLT programs and simple transformations (see Chapter 5). Even though the listings shown here don't contain the handling of system errors due to space limitations, this is an indispensable component of a robust and defensive kind of programming. You will note that especially when transforming XML data into ABAP data, the assignment of an alphanumeric value to a numeric ABAP data type, for instance, triggers an exception that must be handled.

The following lines show the structure of the `CATCH` block that contains a transformation error:

```
DATA: l_rif_ex TYPE REF TO
        cx_xslt_runtime_error,
        l_var_text TYPE string,
        l_var_source_line TYPE i,
        l_var_program_name TYPE syrepid,
        l_var_include_name TYPE syrepid.

CATCH cx_xslt_runtime_error INTO l_rif_ex.
    l_var_text = l_rif_ex->get_text( ).
* Determine error position
CALL METHOD
    l_rif_ex->get_source_position
IMPORTING
    source_line = l_var_source_line
    program_name = l_var_program_name
    include_name = l_var_include_name.
* Log error
IF l_var_text IS NOT INITIAL.
    WRITE: / l_var_text.
    WRITE: / 'The error exists in line:',
            l_var_source_line.
ENDIF.
```

Calling ABAP from XSLT

To a certain extent, every XSLT processor supports extension mechanisms. For example, in our case, we can call

ABAP methods and function modules from a transformation. In one application of ABAP integration, we will demonstrate how you can store error messages in a logging object. Another typical application is the transformation in the data exchange process that discovers—during the transformation of an XML document—that a system error has occurred, or that the data does not meet the data quality requirements agreed upon. Once the transformation has been analyzed and additional error-handling mechanisms have been initiated, the logged error messages are collected in a main memory component.

For this purpose we'll define an object called `Z_CL_LOGGER` that contains a `save` method including the three parameters, `I_STRING`, `I_LINE`, and `I_COLUMN`. This object must possess another attribute that collects the data in an internal table in the main memory, as well as another method for saving the messages, for example, in the business application log. This object is transferred to the `PARAMETERS3` parameter of the `CALL TRANSFORMATION` statement.

The SAP-specific extensions, `sap:external-function` and `sap:call-external`, enable you to integrate ABAP in XSLT. At this point, you must understand that a combination of a procedural programming style in ABAP and a function-based style in XSLT often results in programs that can hardly be maintained. And yet the ABAP integration is generally unavoidable, particularly with regard to database access. Other areas of application include the implementation of counters and the access to number range objects.

Another useful way of applying the ABAP integration is the creation of error lists and process tracing. The transformation shown in Listing 4.1 processes an XML document recursively and logs the occurrence of `A` elements, including their position, in the source text.

The example in Listing 4.1 clearly illustrates the functionality of an XSLT program. An XSLT program consists of a set of templates that are defined via the `xsl:template` command. The `match="A"` attribute assigns a template rule to the above template. This rule states that the template can be used for processing the `A` elements.

At each point in time, the XSLT processor processes *nodes* of a context. At the beginning of the process, the context contains only the document root. For each node, the template rules are analyzed in order to find out which template is appropriate for the node. The identified template is then evaluated. If there is more than one template that meets the requirements, the template that meets the requirements most precisely is used. In our case, it means that the first `A` element is found, and the context of the process is newly set with regard to the element found. To continue the required recursive search for `A` elements, we use the `xsl:apply-templates` command. If no other element is found, the process terminates.

This type of processing is rather general, but not very efficient. Section 4.7 deals with this aspect of processing.

For the integration in ABAP log messages, we define an object using a `save` method. The instance is transferred to the transformation via `PARAMETERS object = l_rc1_log`. The `<xsl:param name="OBJECT" />` statement

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sap="http://www.sap.com/sapxsl" version="1.0">
  <xsl:param name="OBJECT" />
  <xsl:template match="A">
    <sap:call-external name="OBJECT" method="save">
      <sap:callvalue param="I_STRING" select="'A found'" />
      <sap:callvalue param="I_LINE" select="sap:line()" />
      <sap:callvalue param="I_COLUMN" select="sap:column()" />
    </sap:call-external>
    <xsl:apply-templates select="*" />
  </xsl:template>
</xsl:transform>
```

Listing 4.1 Transformation `z_messages_log`

³ The OBJECTS that were originally provided for this purpose have become obsolete since Release 6.20.

ensures that the parameter is published within the transformation:

```
DATA l_rc1_log
TYPE REF TO z_cl_logger.
DATA l_var_input TYPE string.
DATA l_var_output TYPE string.

CREATE OBJECT l_rc1_log.
l_var_input =
  `<R><Level1><A>a</A></Level1>
  <A>a</A></R>`.

CALL TRANSFORMATION z_messages_log
  PARAMETERS object = l_rc1_log
  SOURCE XML l_var_input
  RESULT XML l_var_output.
```

A common source of error is that the value of the transferred object is not capitalized in the `xsl:param` statement. The values that are transferred along with the transformation are translated into the corresponding ABAP values. In this context, a nodeset is automatically transformed into an object that is implemented by the `if_ixml_node` or `if_ixml_node_collection` interface respectively (see Section 3.1).

When transferring attributes using the `sap:callvalue` statement, you should determine whether you have to convert the attributes into a string first; otherwise, a nodeset may be transferred, which generally causes a type error. If a conversion is impossible, an exception—`CX_XSLT_ABAP_CALL_ERROR`—is triggered that must be absorbed by the surrounding ABAP program. The same holds true when an object call generates an exception.

If the method contains return parameters, you can bind those parameters to a variable by using the `sap:callvariable` command: The `<sap:callvariable name="variable" param="E_STRING"/>` command transfers the value of the `variable` variable into the exporting parameter, `E_STRING`. Here, too, an implicit conversion process is carried out. If this is not the outcome that we want, we can use the `type` attribute to specify a conversion that's described in the SAP Library. This conversion enables you to bind the return value to an XSLT variable. Please refer to the SAP Library for further details such as handling `CHANGING` parameters.

Similarly, the syntax required for calling class methods is also described in the SAP Library. In addition, you can call methods dynamically via the `method` attribute: `method="METHOD_{index}"`.⁴ This procedure is very general, but not robust if the index is encoded in the transformation instead of being determined at runtime by an ABAP object.

Compared to the `sap:call-external` command described earlier, the `sap:external-function` command has some advantages. This command can also be used in XPath expressions. Its syntax is more compact because it corresponds to a function definition at the *top level*⁵ of the XSLT program. However, the consequence of that is that a function value must be returned either as a `RETURNING` parameter or as an individual `EXPORTING` parameter. Other `EXPORTING` parameters are ignored. When you extend the `Z_CL_LOGGER` class by a `prot()` method that returns a `BOOLEAN` function value, you can define the call of this method as a top-level element by using the `sap:external-function` command, as shown in the XSLT program in Listing 4.2.

When the call is carried out via the `sap:external-function`, the attribute determines whether the method call is a constructor (`constructor`), a class method (`class`), or a method of the object (`instance`).

The call of the external function `prv:prot` corresponds to the convention that the first function argument is the name of the externally called object. This convention is commonly used in Java.

At this point, you could find it disturbing that the logging object is still contained in the interface of the `CALL TRANSFORMATION` call as well as in the `xsl:param` element of the transformation. We should therefore use the *factory pattern* now. We'll extend the `Z_CL_LOGGER` class by a static method, `create()`, that returns an instance of `Z_CL_LOGGER`. This method is called within a transformation, and the created object is stored in a variable. The logging method is then called via this variable. Listing 4.3 describes this process.

⁴ The contents of the `method` attribute are an *attribute value template* (AVT). The XSLT processor can use the AVT at runtime to determine a value that is contained in curly brackets (`{}`). With the `method` attribute, the value is stored in the `$index` variable.

⁵ Top-level elements are the child elements of `xsl:transform` and `xsl:stylesheet`.

```

<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sap="http://www.sap.com/sapxsl" xmlns:prv="urn:mime" version="1.0">
  <xsl:param name="OBJECT" />
  <sap:external-function name="prv:prot" method="prot" kind="instance" >
    <sap:argument param="i_string"/>
    <sap:argument param="i_line"/>
    <sap:argument param="i_column" />
    <sap:result param="e_return" type="boolean" />
  </sap:external-function>
  <xsl:template match="A">
    <xsl:value-of select="prv:prot($OBJECT, 'A found', sap:line(), sap:column())"/>
    <xsl:apply-templates select="A"/>
  </xsl:template>
</xsl:transform>

```

Listing 4.2 Calling a Method from XSLT

In the example in Listing 4.3, the solution may not appear to be adequate; however, typically, it can be considered good programming to create objects via factory methods, to store references to those objects, and lastly, to access the objects outside of the transformation.

The `sap:external-function` command does not support optional parameters. Consequently, you must program different `sap:external-function` commands for different calls.

Exception Handling in ABAP Calls from XSLT

There are ABAP programmers who complain that it is impossible to call function modules directly from XSLT.

Instead, they can only call objects that encapsulate those calls. This restriction, however, should not necessarily be regarded as a weakness as it exemplifies the efficiency of class-based exceptions in ABAP. If an error occurs in a deep software layer, it must be logged and then neutralized in the calling layer. If, for instance, an error occurs in an ABAP call from XSLT, because a data record is locked in the database or doesn't exist, often times the transformation is canceled, which, in turn, causes the calling ABAP program to react.

If there were only function modules available for that, we would have to forward the return codes and error logs in an additional interface. The concept of class-based

```

<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sap="http://www.sap.com/sapxsl" xmlns:prv="urn:mime" version="1.0">
  <sap:external-function name="prv:create" class="Z_CL_LOGGER" method="create" kind="class" >
    <sap:result param="e_object" type="external" />
  </sap:external-function>
  <sap:external-function name="prv:prot" method="prot" kind="instance" >
    <sap:argument param="i_string"/>
    <sap:argument param="i_line"/>
    <sap:argument param="i_column" />
    <sap:result param="e_return" type="boolean" />
  </sap:external-function>
  <xsl:variable name="instance" select="prv:create()"/>
  <xsl:template match="A">
    <xsl:value-of select="prv:prot($instance, 'A found', sap:line(), sap:column())"/>
    <xsl:apply-templates select="A"/>
  </xsl:template>
</xsl:transform>

```

Listing 4.3 Instantiating an Object from XSLT

exceptions, however, represents an elegant way of solving this kind of problem. When a serious error occurs in an ABAP function, we can report an exception by using the `RAISE EXCEPTION` command. The transformation cancels and we'll catch an exception of the type `CX_XSLT_ABAP_CALL_ERROR` in the `CATCH` block that follows `CALL TRANSFORMATION`. This exception is actually a chained exception in which you can recognize the original exception object by the `previous` attribute.

4.3 Transforming ABAP Data

To provide access to ABAP data transformations as sources and targets, SAP created a serialization format called *asXML*, which enables you to program the attributes of all basic and composite ABAP data types and even classes in XML format, and therefore as character strings. When transforming an XML document into ABAP data structures, you must implement an XSLT transformation that generates an asXML representation. The ABAP data structures are then filled in an implicit conversion step.

Serializing ABAP Data

If you want to perform a transformation from an XML document into an ABAP data structure, you must know the target data structure in the asXML representation. If you don't know that representation, you should simply generate it and analyze it. To do that, we recommend that you fill the ABAP target structure with values and then transfer it with the identical transformation to XML. The following call enables you to do just that:

```
CALL TRANSFORMATION id
  SOURCE p_1 = ... p_2 = ...
  RESULT XML l_var_xml_string.
```

This identical transformation is integrated in the SAP kernel and should not be mistaken for the transformation ID that's contained in the `SXSLT` package.

Now we'll analyze how we can convert basic ABAP data types to asXML. To do that, you must define a structure that contains the most common basic ABAP data types, and then apply the transformation `id` to them:

```
TYPES: BEGIN OF l_typ_test,
  chars  TYPE c LENGTH 10,
```

```
  string TYPE string,
  numc   TYPE n LENGTH 5,
  packed TYPE p LENGTH 4
           DECIMALS 2,
  float  TYPE f,
  date   TYPE d,
  time   TYPE t,
  xstring TYPE xstring,
END OF l_typ_test.
```

```
DATA: l_str_test TYPE l_test_typ,
      l_var_output TYPE string.
l_str_test-chars = ' ABC '.
l_str_test-string = ` ABC `.
l_str_test-numc = '01234'.
l_str_test-packed = 123.45.
l_str_test-float = 123.45.
l_str_test-date = '20060614'.
l_str_test-time = '1201'.
l_str_test-xstring = 'ABCDEFGF'.
CALL TRANSFORMATION id
  SOURCE root = l_str_test
  RESULT XML l_var_output.
```

The result of this transformation is the following XML document:

```
<?xml version="1.0"
  encoding="iso-8859-1"?>
<asx:abap
  xmlns:asx="http://www.sap.com/abapxml"
  version="1.0">
  <asx:values>
    <ROOT>
      <CHARS>ABC</CHARS>
      <STRING> ABC </STRING>
      <NUMC>01234</NUMC>
      <PACKED>123.45</PACKED>
      <FLOAT>1.2345E2</FLOAT>
      <DATE>2006-06-14</DATE>
      <TIME>12:01:00</TIME>
      <XSTRING>q83v</XSTRING>
    </ROOT>
  </asx:values>
</asx:abap>
```

The root element of an asXML document is `asx:abap`; the values for ABAP data types are `asx:values`. In our example, the only child element of `asx:values` is the `ROOT` element, which contains the values of the `SOURCE` parameter in the above `CALL TRANSFORMATION` call. This parameter is assigned a structure, `root`, whose components are the child elements of the XML element `ROOT`. The components correspond to the basic data types of the `l_typ_test` structure.

The basic data types are presented in compliance with integrated data types of the W3C XML Schema (<http://www.w3.org/TR/xmlschema-2/#built-in-datatypes>). You can see that the implementation of date and time types complies with ISO 8601,⁶ whereas `xstrings` are encoded according to Base64 (<http://www.ietf.org/rfc/rfc2045.txt>).

Sample Deserialization

As an example, we now want to describe a conversion of the fictitious master data scenario contained in Section A.1. Figure 4.5 shows a graphical display of the master data. This is a catalog of meals provided by a catering service for flight meals as a *BMEcat catalog*.⁷

Our goal is to assign each `ARTICLE` element an entry from the `SMEAL` table. The long and short texts from the associated `ARTICLE_DETAILS` element are transferred into this table and the associated `SMEALT` table.

The `CATALOG_STRUCTURE` element provides the structure of the contents of the `GROUP_ID` element. If the ID begins with the string "DT1010", then it refers to main courses, whereas "DT1020" refers to starters and "DT1030" to desserts. For each item, you can determine the assignment to one of the three product groups via the

type	GROUP_ID	GROUP_NAME	PARENT_ID	GROUP_ORDER
1 root	1	Catalog	0	1
2 node	DT10	Meals	1	2
3 node	DT1010	Main Courses	DT10	3
4 leaf	DT1010100	Asian	DT1010	4
5 leaf	DT1010200	European	DT1010	5
6 node	DT1020	Starters	DT10	6
7 leaf	DT1020100	Salads	DT1020	7
8 leaf	DT1020200	Soups	DT1020	8
9 leaf	DT1030	Desserts	DT10	9

mode	SUPPLIER_AID	ARTICLE_DETAILS	ARTICLE_ORDER...	ARTICLE_PRICE...
1 new	3179386	ARTICLE_DETAILS	ARTICLE_ORDER...	ARTICLE_PRICE...
2 new	1215483	ARTICLE_DETAILS	ARTICLE_ORDER...	ARTICLE_PRICE...
3 new	1206199	ARTICLE_DETAILS	ARTICLE_ORDER...	ARTICLE_PRICE...
4 new	1292092	ARTICLE_DETAILS	ARTICLE_ORDER...	ARTICLE_PRICE...
5 new	1270175	ARTICLE_DETAILS	ARTICLE_ORDER...	ARTICLE_PRICE...

ART_ID	CATALOG_GRO...
1 3179386	DT1010100
2 1215483	DT1010200
3 1206199	DT1020100
4 1292092	DT1020200
5 1270175	DT1030

Figure 4.5 Display of a BMEcat XML Document

⁶ Except that the year must range between 0 and 9999 in ABAP and the date of 00000000 is permitted.

⁷ BMEcat was developed by the Fraunhofer Institute IAO, the University of Essen, Germany, and the University of Linz, Austria, with the objective to create a uniform standard for catalog data in electronic procurement processes. Currently, there are over 150 different standards available for those processes.

child elements of the `ARTICLE_TO_CATALOGGROUP_MAP` element in order to find out if it is a starter, a main course, or a dessert. This system is used to create an entry for the corresponding transparent table.

The XSLT program in Listing 4.4 performs exactly this transformation.

At the beginning of the XSLT program in Listing 4.4, we define a local namespace using the prefix `BME` because a local namespace was defined in the original XML document for the `BME` root element and, recursively, for its children using the following URI: `http://www.bmecat.org/XMLSchema/1.2/bmecat_new_catalog`.

In the transformation, we analyze all catalog items (articles). Those items are programmed using the `BME:ARTICLE` elements. For each catalog item, we call the `CreateMeal` template to create an entry for the SAP tables, `SMEAL` and `SMEALT`. In our example, a catalog item can be a starter, a main course, and a dessert. Because each of those meals must be saved in different SAP tables (`SSTARTER`, `SMACOURSE`, or `SDESSERT`), we must first define the types. To do that, we use the item name to determine the associated ID catalog group via the `ARTICLE_TO_CATALOGGROUP_MAP` element, for which we have defined a `key_article` index to increase the system's performance.

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sap="http://www.sap.com/sapxsl" xmlns:asx="http://www.sap.com/abapxml"
  xmlns:BME="http://www.bmecat.org/XMLSchema/1.2/bmecat_new_catalog"
  exclude-result-prefixes="BME" version="1.0">
  <xsl:param name="CARRID"/>
  <xsl:strip-space elements="*" />
  <!-- Create key for fast access -->
  <xsl:key match="BME:CATALOG_GROUP_ID" name="key_article" use="string(..BME:ART_ID)"/>
  <xsl:template match="/">
    <asx:abap>
      <asx:values>
        <xsl:for-each select="/BME:BMECAT/BME:T_NEW_CATALOG/BME:ARTICLE">
          <xsl:variable name="article" select="."/>
          <!-- Determine item number -->
          <xsl:variable name="mealnumber" select="string(..BME:SUPPLIER_AID)"/>
          <!-- Determine catalog grouping -->
          <xsl:variable name="CatalogID"
            select="string(key('key_article', $mealnumber))"/>
          <!-- Create meal -->
          <SMEAL>
            <xsl:call-template name="CreateMeal">
              <xsl:with-param name="mealnumber" select="$mealnumber"/>
              <xsl:with-param name="article" select="$article"/>
            </xsl:call-template>
          </SMEAL>
          <!-- Create long text for meal -->
          <SMEALT>
            <xsl:call-template name="CreateMealText">
              <xsl:with-param name="mealnumber" select="$mealnumber"/>
              <xsl:with-param name="article" select="$article"/>
            </xsl:call-template>
          </SMEALT>
          <!-- Analyze meal type -->
          <xsl:choose>
            <xsl:when test="starts-with($CatalogID, 'DT1010')">
              <SMACOURSE>
```

Listing 4.4 Z_BME Transformation

Index

A

A2A integration 10
Abstract syntax trees 51
asXML 36, 61, 105
asXML representation 36
Attribute value template 34

B

Base64 37
bindingTemplate 85
BizTalk messages 5
BMEcat 5, 107, 108
BMEcat catalog 37
businessEntity 85, 93
businessService 85

C

CALL TRANSFORMATION 32, 37, 56
CDATA sections 25
Chained exceptions 36
Character sets 14, 52
CIF 5
CL_GUI_HTML_VIEWER 104
CL_IXML 14
Code generation 11, 50
Complex transformations 104
Content models
 Non-deterministic 61
current-group() 46
current-grouping-key() 46
cXML 5, 16, 59, 68, 69, 113

D

Data integration 9
Data quality 101
Datastore component 26
Default error handler 81

Default namespace 46
Deployment descriptors 84
Deserialization 57
Design pattern 26
Displaying XML documents 104
Diversifying development and testing 105
Document type definitions 8
DOM 15
DTD 17

E

E-business standards 100
ebXML 5, 99
eCl@ss 108
EDIFACT 7, 8
EJB 84
EJB specification
 restrictions 84
Encodings 14
Enterprise Application Integration 9, 99
Enterprise Application Project 90
Exceptions
 Chained 36

F

Factory pattern 34
function-available() 45

G

Generic object services 104
group-adjacent() 46
group-by() 46
group-ending-with() 46
group-starting-with() 46
gXML 5

H

Head-body pattern 27
HL7 messages 5
Home interface 84
HTTP requests 104

I

ICF 104
if_ixml 14
if_ixml_attribute 13
if_ixml_document 13, 15, 32
if_ixml_event 14, 24, 25
if_ixml_istream 14, 32
if_ixml_node 13, 27, 32, 34
if_ixml_node_collection 34
if_ixml_node_filter 27
if_ixml_ostream 14, 32
if_ixml_parser 14, 19
if_ixml_renderer 17
if_ixml_stream 14
if_ixml_stream_factory 14
if_serializable_object 10
Interface formats
 Proprietary 101
Internet Communication Framework (ICF)
 11
ISO-8859-1 14
ISO 4217 52
iXML library 13

J

J2EE 11, 84
J2EE connector architecture 84
Java 54, 81
JAXP 81
 JAXP 1.3 97
JCo 84

L

Literal result elements 46
 Literal text 62, 77
 Logical port 95
 LSMW 50

M

Markup 7
 Mass data 14, 27, 55, 101
 Message implementation guidelines 8
 Middleware 102
 Mime64 9
 Modular schemas 96

N

Namespace 20, 21, 23, 45, 96
 Namespace axis 45
 Non-deterministic content models 61
 Null namespace 46

O

Object identifiers 52
 Object services
 Generic 104
 Online text repository 45
 Optional elements 62
 Outside-in approach 94

P

PARAMETERS 33
 Pattern 65, 67, 72
 Payload 8
 Port
 logical 95
 Ports 11
 Process integration 9
 Proprietary interface formats 101
 Proxy class 95
 Proxy object 94

R

Regression tests 104
 RELAX NG 81
 Remote interface 84
 Request-response processes 8, 16, 99
 Resolving release dependencies 102

Result fragment tree 45
 Risk management 102
 RosettaNet 5

S

Safety facades 86
 sap:abs() 44
 sap:acos() 44
 sap:asin() 44
 sap:atan() 44
 sap:call-external 33, 34
 sap:callvalue 34
 sap:callvariable 34
 sap:column 30
 sap:concat 50
 sap:cos() 44
 sap:ends-with() 44
 sap:escape-uri() 44
 sap:exp() 44
 sap:external-function 33, 34, 35
 sap:find-first() 44
 sap:find-first-of() 44
 sap:find-last() 44
 sap:find-last-not-of() 44
 sap:find-last-of() 44
 sap:if() 48
 sap:intersection 45
 sap:let() 45
 sap:line 30
 sap:log() 44
 sap:log10() 44
 sap:lower-case() 44
 sap:max() 45
 sap:min() 45
 sap:node-set() 45
 sap:otr-string() 45
 sap:parse-xpath() 51
 sap:resolve-uri() 44
 sap:sin() 44
 sap:sqrt() 44
 sap:string-pad() 44
 sap:tan() 44
 sap:target 48
 sap:timestamp() 45, 52
 sap:upper-case() 44
 SAP NetWeaver Exchange Infrastructure
 10, 102
 SAP XML Toolkit for Java 23, 81, 89
 SAX 23
 SAXParserFactory 81

Schemas

 modular 96
 Serialization 57
 Session beans 84, 85
 Simple Transformations
 Sample applications 55
 Simple transformations 103
 ABAP structures 58
 Assertion 64
 Attributes 58
 Case distinctions 62, 66
 Case distinctions using variables 62
 Condition 63
 Conditional transformation 65
 Conditional transformations with
 variables 73
 Data nodes 57
 Data roots 57
 Expressiveness 74
 Grouping 69
 Internal tables 59
 Literal contents 77
 Mappings 72
 Modularization 76
 Namespace declarations 77
 Parameters 62
 Precondition 64, 66
 Symmetry 57, 79
 Variables 61
 SOAP 11, 99, 104
 Standard deserialization 72
 Standard serialization 67
 STX (Streaming Transformations for XML)
 103
 Symmetrical case distinctions 68

T

Tail recursion 52
 tModel 85, 93
 tt:apply 56, 77
 tt:assign 62, 74, 75
 tt:attribute 58
 tt:call 56, 74, 77
 tt:clear 62
 tt:cond 63, 64, 65, 67
 tt:cond-var 62, 74
 tt:context 61, 76
 tt:copy 61
 tt:d-cond 63, 66, 74
 tt:deserialize 73, 79

tt:group 69, 72
 tt:include 56, 77
 tt:lax 65
 tt:loop 59, 75
 tt:namespace 79
 tt:parameter 56
 tt:read 62
 tt:ref 57
 tt:root 56, 57
 tt:s-cond 63, 66
 tt:serialize 79
 tt:skip 63, 74, 77
 tt:switch 66, 67, 73
 tt:switch-var 73, 74
 tt:template 56, 76
 tt:transform 55, 61
 tt:value 56
 tt:variable 56, 61, 73
 tt:with-parameter 76
 tt:with-root 76
 tt:write 62

U

UDDI tool 93
 Unicode 14, 52
 User defined XPath functions 48
 UTF-16 14, 15
 UTF-8 14, 15

V

Validation 8, 18
 Versioning interfaces 101

W

W3C XML Schema 8, 100
 Expressiveness 8
 Web service interface 90
 Web services 11, 84, 93
 Calling from ABAP 95
 Webservices Navigator 94
 Wellformedness 7, 25, 102
 WSDL 11, 92, 94

X

xCBL 5
 XML 5
 XML 1.1 97
 XML declaration 50
 XML Encryption 81
 XML library 102
 XML signature 81
 XPath 29, 44
 XPath 2.0 43
 XPath 2.0 conformity 43
 XPath functions
 user defined 48

xs:include 96
 XSF data streams 10
 xsl:apply-imports 52, 53
 xsl:apply-templates 33
 xsl:for-each-group 42, 46, 53
 xsl:function 48
 xsl:import 52, 53
 xsl:include 52, 53
 xsl:message 30
 xsl:output 48
 xsl:result 48
 xsl:stylesheet 29
 xsl:transform 29
 XSLT 29, 103, 105
 ABAP extensions 33
 Calling from ABAP 32
 Exception handling 35
 Java extensions 54
 Limits of the SAP processor 45
 Modularization 52
 Output formatting 48, 50
 Platform-independent transformations
 53
 SAP-specific extensions 43
 XSLT 2.0 46, 48, 53
 XSLT 2.0 compliance 46, 48, 53
 XSLT 2.0 conformity 29
 XSLT Debugger 31