opentext™

OpenText™ Embedded Data Transformation Engine

User Guide

This document provides information about the features and functionality of Data Transformation Engine.

VDTOTS240200-UDT-EN-1

OpenText™ Embedded Data Transformation Engine
User Guide

VDTOTS240200-UDT-EN-1
Rev.: 2024-Apr-16

This documentation has been created for OpenText™ Embedded Data Transformation Engine CE 24.2.
It is also valid for subsequent software releases unless OpenText has made newer documentation available with the product, on an OpenText website, or by any other means.

Open Text Corporation

275 Frank Tompa Drive, Waterloo, Ontario, Canada, N2L 0A1

Tel: +1-519-888-7111
Toll Free Canada/USA: 1-800-499-6544 International: +800-4996-5440
Fax: +1-519-888-0677
Support: https://support.opentext.com
For more information, visit https://www.opentext.com

© 2024 Open Text

Patents may cover this product, see https://www.opentext.com/patents.

Disclaimer

No Warranties and Limitation of Liability

Every effort has been made to ensure the accuracy of the features and techniques presented in this publication. However, Open Text Corporation and its affiliates accept no responsibility and offer no warranty whether expressed or implied, for the accuracy of this publication.

Table of Contents

1 Introduction 11

1.1 Installing and Running Data Transformation Engine 11

1.2 Supported Databases 12

1.3 Data Transformation Engine tab 12

2 Data Transformation Engine window 13

2.1 Using the Data Transformation Engine window 13

3 Settings window 15

3.1 Managing Tab and Row commands 16

3.1.1 Tab Management commands 17

3.1.2 Row Management commands 18

3.2 Global Transform settings 19

3.2.1 Preferences tab 19

3.2.1.1 Template metadata 20

3.2.1.2 Error Handler 21

3.2.1.3 Package and file paths 21

3.2.1.4 Mapping lines 22

3.2.2 Variables tab 22

3.2.2.1 Variable nesting 22

3.2.2.2 Using a Variable in output 23

3.2.2.2.1 Large numbers of variables 24

3.2.2.3 Sorting the Variables dialog box 25

3.2.3 Custom Function tab 26

3.2.3.1 Custom Function Wizard 26

3.2.4 Find & Replace tab 26

3.2.4.1 Import a Look-up list 27

3.2.4.2 Load Look-Up table using a SQL statement 28

3.3 Global Data Format settings 28

3.3.1 JDBC Connection tab 28

3.3.1.1 JDBC Connection Validation and Cache settings 29

3.3.1.2 Setting up a Direct JDBC connection 30

3.3.1.3 Setting up a JDBC connection via Persistent Storage Pool 30

3.3.1.4 Setting up a JDBC connection via JNDI 31

3.3.2 JDBC Lookup Table tab 32

3.3.3 XML Formatter tab 33

3.3.4 XML PI tab 34

3.3.5 EDI Formatter tab 35

3.3.6 COBOL Formatter tab 36

3.3.7 COBOL IO Directive tab 37

3.3.8 Transaction tab 39

3.3.9 PDF Appended Page Formatter tab 40
3.3.10 XBRL tabs 40
3.3.10.1 XBRL Context tab 40
3.3.10.2 XBRL Entity tab 42
3.3.10.3 XBRL Period tab 43
3.3.10.4 XBRL Scenario tab 44
3.3.10.5 XBRL Unit tab 45
3.3.11 Transformations in multiple thread environments 46

4 Source pane 47
4.1 Source Structure 48
4.2 Source Instance 50
4.3 Source Configuration 51
4.4 Input Data Format 52
4.4.1 XDoc Input Data Format 53
4.4.2 XML Input Data Format 54
4.4.2.1 Recursive Schemas 55
4.4.3 CSV Input Data Format 55
4.4.4 PDF Input Data Format 55
4.4.5 EDIFACT Input Data Format 55
4.4.6 EDI X12 Input Data Format 56
4.4.7 EDI HIPAA Input Data Format 56
4.4.8 SWIFT Input Data Format 56
4.4.9 HL7 Input Data Format 57
4.4.10 IDOC Input Data Format 57
4.4.11 JDBC Input Data Format 57
4.4.11.1 JDBC Join Feature 58
4.4.12 COBOL Input Data Format 59
4.4.13 COBOL nodes 60
4.4.13.1 Level 01 (L01) nodes 60
4.4.13.2 L01 structure 61
4.4.13.3 Redefined nodes 61
4.4.13.4 User defined nodes 62
4.4.13.5 COBOL node properties 62
4.4.13.5.1 COBOL Enumerated Values 62
4.4.14 Flat Text Input Data Format 63
4.4.14.1 Flat Text Importer Wizard 64
4.4.15 JDBC Input SQL statement generation 64
4.4.16 SQL statements 64
4.4.16.1 SQL statement syntax 65
4.4.17 Multiple Inputs Data Format 66
4.4.17.1 Load button 67

4.4.18 Loading structures from other sources 67
4.5 Preparsers 67

5 Target pane 69
5.1 Target Structure 70
5.2 Target Result 72
5.3 Target Configuration 74
5.4 Target Data Format 75
5.4.1 Multiple Output Transformations 76
5.4.2 XDoc Output Data Format 76
5.4.3 XML Output Data Format 76
5.4.4 CSV Output Data Format 77
5.4.5 XBRL Output Data Format 78
5.4.6 HL7 Output Data Format 79
5.4.7 SWIFT Output Data Format 80
5.4.8 Flat Text Output Data Format 80
5.4.9 EDI X12 Output Data Format 81
5.4.10 EDI HIPAA Output Data Format 81
5.4.11 EDIFACT Output Data Format 82
5.4.12 JDBC Output Data Format 83
5.4.13 JDBC Output Exception Control 84
5.4.14 SQL Output Data Format 86
5.4.15 COBOL Output Data Format 87
5.4.16 PDF Output Data Format 87
5.4.16.1 PDF Form Field Names 88
5.4.16.2 Appending Pages to PDFs 88
5.4.17 Object Output Data Format 89
5.4.18 Customizing output formatting 90

6 Mappings 91
6.1 Drag-and-drop feature 91
6.2 Source Mapping 91
6.3 Target Mapping 92
6.3.1 Using variable values 93
6.4 Output structure and values 94
6.4.1 Output structure 94
6.4.1.1 Node icons 94
6.4.1.2 Required structure elements 95
6.4.1.3 Mapping report and node descriptions 96
6.4.1.4 Output item values 97
6.4.1.5 Mapping options 98
6.4.1.6 Parent items 108
6.4.1.7 Filter 114

6.4.1.8 Namespaces 115
6.4.1.9 Dynamic Load and Unload elements 116

7 Functions 119
7.1 Function Editor 119
7.1.1 Selecting your Function 120
7.1.2 Setting Function parameters 121
7.1.3 Add a parameter from the Function Editor 122
7.2 Pre-defined Functions 122
7.2.1 Date/Time Functions 122
7.2.1.1 ADD_DATE 122
7.2.1.2 ADD_TIME 123
7.2.1.3 DATE 124
7.2.1.4 DAY 125
7.2.1.5 DOW 125
7.2.1.6 DSTR 126
7.2.1.7 DVAL 126
7.2.1.8 EOM 127
7.2.1.9 EOY 127
7.2.1.10 HOUR 128
7.2.1.11 MINUTE 128
7.2.1.12 MONTH 129
7.2.1.13 SECOND 129
7.2.1.14 SOM 130
7.2.1.15 SOY 130
7.2.1.16 TIME 130
7.2.1.17 TSTR 131
7.2.1.18 YEAR 132
7.2.2 String/Character Functions 132
7.2.2.1 CONCAT 132
7.2.2.2 DECIMAL FORMAT 133
7.2.2.3 DELSTR 133
7.2.2.4 EQUALS 134
7.2.2.5 FILL 135
7.2.2.6 FLIP 135
7.2.2.7 Format string 136
7.2.2.8 HSTR 136
7.2.2.9 HVAL 137
7.2.2.10 INSERT 137
7.2.2.11 INSTR 138
7.2.2.12 LEFT 139
7.2.2.13 LEN 139

7.2.2.14 LOWER 140
7.2.2.15 LTRIM 140
7.2.2.16 MID 140
7.2.2.17 NOT_EQUALS 141
7.2.2.18 QUOTEGEN 142
7.2.2.19 REP 142
7.2.2.20 RIGHT 143
7.2.2.21 RTRIM 143
7.2.2.22 SINGLEQUOTE 144
7.2.2.23 STR 144
7.2.2.24 STRTOKEN 145
7.2.2.25 SUBSTR 145
7.2.2.26 TOSTR 145
7.2.2.27 TRIM 146
7.2.2.28 UPPER 146
7.2.3 Numeric Functions 147
7.2.3.1 ADD 147
7.2.3.2 DIVIDE 147
7.2.3.3 GREATER_THAN 148
7.2.3.4 GREATER_THAN_OR_EQUAL_TO 148
7.2.3.5 INTVAL 149
7.2.3.6 LESS_THAN 149
7.2.3.7 LESS_THAN_OR_EQUAL_TO 150
7.2.3.8 MOD 150
7.2.3.9 MULTIPLY 151
7.2.3.10 NUM 151
7.2.3.11 NUM_CHR 152
7.2.3.12 RANDOM 152
7.2.3.13 RANGE 153
7.2.3.14 ROUND 153
7.2.3.15 SUBTRACT 154
7.2.3.16 VAL 154
7.2.4 Processing Functions 155
7.2.4.1 CONTROL_CHARACTER 155
7.2.4.2 IF 156
7.2.4.3 ISNULL 157
7.2.4.4 NULL 157
7.2.4.5 REPLACE 157
7.2.4.6 SEQUENCE 158
7.2.4.7 SETVAR 159
7.2.4.8 SIMPLEREPLACE 159
7.2.4.9 THROW_EXCEPTION 160

7.2.5 Security Functions ... 161
7.2.5.1 CHKDGT ... 161
7.2.6 Group Functions ... 162
7.2.6.1 AVG ... 162
7.2.6.2 COUNT ... 163
7.2.6.3 COUNTING ... 164
7.2.6.4 MAX ... 165
7.2.6.5 MIN ... 166
7.2.6.6 SUM ... 167
7.2.7 EDI Functions ... 168
7.2.7.1 COUNT_GROUP ... 168
7.2.7.2 COUNT_SEGMENT ... 168
7.2.7.3 COUNT_TRANSACTION ... 169
7.2.8 DOM Functions ... 170
7.2.8.1 NODE ... 170
7.2.8.2 NODES ... 170
7.2.8.3 XPath ... 171
7.2.8.4 XPathEval ... 173
7.2.9 Database Functions ... 174
7.2.9.1 EXECUTE_QUERY ... 175
7.2.9.2 EXECUTE_STORED_FUNC ... 175
7.2.9.3 FIELD_EXISTS ... 176
7.2.9.4 ORASEQUENCE ... 177
7.2.9.5 ORASTOREDFUNC ... 178
7.2.9.6 RUN_FUNCTION ... 178
7.2.9.7 RUN_QUERY ... 179
7.2.9.8 STORED_FUNC ... 180
7.2.10 Picture masks ... 180
7.2.10.1 Numeric pictures ... 181
7.2.10.2 Date and Time pictures ... 182

8 Custom components ... 185
8.1 Custom preparsers ... 185
8.1.1 Writing custom preparsers ... 185
8.1.1.1 Import statement ... 186
8.1.2 Class declaration ... 186
8.1.3 Reading, manipulating, and sending data ... 187
8.1.3.1 Reading data ... 187
8.1.3.2 Manipulating data ... 187
8.1.3.3 Sending data ... 187
8.1.4 Customized parameters ... 188
8.2 Custom functions ... 189

8.2.1 Defining custom functions ... 189
8.2.2 Writing custom functions ... 190
8.2.2.1 Import statement ... 191
8.2.2.2 Class declaration ... 191
8.2.2.3 Constructor ... 191
8.2.2.4 execute() ... 191
8.2.2.5 Getting arguments ... 192
8.2.2.6 Controlling project variables ... 192
8.2.2.7 getReturnType() ... 193
8.2.3 Sample custom function ... 193
8.2.4 Custom Function Wizard ... 194
8.2.4.1 Custom Function directory ... 194
8.2.4.2 Custom Function tab ... 194
8.2.4.3 Custom Function Wizard setup ... 195
8.2.4.4 Custom Function Editor ... 196
8.2.5 Custom Function Component Wizard ... 198
8.2.6 Editing Custom Functions ... 200
8.2.6.1 Editing the Custom Function ... 200
8.2.6.2 Editing the Java settings ... 201
8.2.6.3 Compiling the Custom Function ... 201
8.3 Compiling your Java file ... 202
8.3.1 Compiling using the command prompt ... 202
8.3.2 Compiling in Output Transformation Designer ... 203

9 Dictionary files ... 205
9.1 Composing dictionary files ... 206
9.1.1 Composing EDI dictionary files ... 206
9.1.1.1 EDI dictionary XML declaration ... 207
9.1.1.2 EDI dictionary comments ... 207
9.1.1.3 EDI Node ... 207
9.1.1.4 EDI TransactionSet node ... 208
9.1.1.5 EDI Segment node ... 208
9.1.1.5.1 Segment Type attribute ... 209
9.1.1.6 EDI Loop node ... 210
9.1.1.7 EDI Element node ... 210
9.1.1.7.1 Req conditional requirement ... 212
9.1.1.8 Relational operators ... 212
9.1.1.9 Conditional operators ... 213
9.1.1.9.1 Conditional operators examples ... 213
9.1.1.10 EDI CompositeElement node ... 214
9.1.1.11 EDI ComponentElement node ... 214
9.1.2 Composing SWIFT dictionary files ... 216

9.1.2.1 Starting SWIFT messages ... 216
9.1.2.2 General structure ... 216
9.1.2.3 Basic header block ... 216
9.1.2.4 Application header block ... 218
9.1.2.5 User Header block ... 220
9.1.2.6 Text Block or Body ... 221
9.1.2.6.1 Usage rules for TransactionSet ... 222
9.1.2.6.2 Example ... 226
9.1.2.7 Trailer Block ... 227
9.1.2.7.1 SWIFT message example ... 228
9.1.2.7.2 Writing a Trailer Block ... 228
9.1.3 Composing HL7 dictionary files ... 229
9.1.3.1 HL7 dictionary XML declaration ... 230
9.1.3.2 HL7 dictionary comments ... 230
9.1.3.3 HL7 node ... 230
9.1.3.4 MessageStructure node ... 231
9.1.3.5 HL7 Segments node ... 231
9.1.3.6 HL7 Message node ... 231
9.1.3.7 HL7 Segment node ... 232
9.1.3.7.1 Segment nodes within MessageStructure portion ... 232
9.1.3.7.2 Segment nodes within Segments portion ... 233
9.1.3.8 HL7 Group node ... 233
9.1.3.9 HL7 Field node ... 234
9.1.3.10 HL7 FieldRepeat node ... 235
9.1.3.11 HL7 Component node ... 236
9.1.4 Composing Generic Flat Text dictionary files ... 237
9.1.5 Flat Text Importer Wizard ... 245
9.1.5.1 Flat Text Structure Editor ... 246
9.1.5.1.1 Tree Structure Definition pane ... 247
9.1.5.2 Flat Text Structure Record ... 248
9.1.5.2.1 Simple Instance ... 248
9.1.5.2.2 Complex Instance ... 250
9.1.5.2.3 Selecting a Record ... 252
9.1.5.2.4 Specifying Field Attributes ... 253
9.1.5.3 Nested fields ... 254
9.2 Validating dictionary files ... 256

Chapter 1

Introduction

OpenText Embedded Data Transformation Engine is a rules-based data transformation tool that can transform different structure data formats to one another. It consists of two primary components:

- Data Transformation Engine tab within Output Transformation Designer
- Data Transformation Engine run-time engine

The Data Transformation Engine tab's primary purpose is to create a rules template file that is used by the run-time engine to perform the data transformation. A transformation is defined as the conversion of one or more input files of one data format to one or more output files of one data format.

You can view the template file on the **Template** tab in the **Events** window. You can find more detailed information on the Events window in *OpenText Output Transformation Designer - User Guide (VDTOTS-H-UTD).*

The rules template file is an XML configuration file that consists of the following information:

- Input and output data formats
- Structure of the output data
- Operations required to build output data

The engine processes the input data according to the template file's rules and parameters, and produces the output. A template file is unique to one transformation. If a new transformation is desired, you need to reconfigure the engine by creating a new template file in Output Transformation Designer .

1.1 Installing and Running Data Transformation Engine

Data Transformation Engine can be installed only during the installation of OpenText Output Transformation Server. To ensure that Data Transformation Engine is included in the installation, you must select the Data Transformation Engine plugin on the **Choose Install Set** screen during the installation procedure. For detailed instructions, please refer to *OpenText Output Transformation Server Installation Guide*.

To run Data Transformation Engine, click the **Launch Transform** icon, , in Output Transformation Designer, or click on the **Tools** menu and select **New Data Transformation Project**.

Data Transformation Engine can also be called as part of a custom application by using its Java API. See OpenText Embedded Data Transformation Engine Developer's Guide for more information.

1.2 Supported Databases

MySQL databases are supported, but users who want to connect to a MySQL database must download and install **MySQL Connector/J**, the official JDBC driver for MySQL, before any connections can be made.

To download and install MySQL Connector/J:

1. In your preferred web browser, open https://dev.mysql.com/downloads/connector/j/5.1.html to access the **MySQL Download Connector/J** page.
2. From the **Generally Available (GA) Releases** list, locate the **Connector/J** driver containing your preferred archive file format and download it to your machine.
3. Extract the contents of the archive file to a suitable location.
4. From the extracted archive contents, copy the `mysql-connector-java-<version>.jar` file and paste it into the `<ots_home>\install\<version>\lib\common` directory.

Note: For a full listing of the latest supported databases and their versions, see the Release Notes.

1.3 Data Transformation Engine tab

The Data Transformation Engine tab in Output Transformation Designer is a user-friendly drag-and-drop based graphical application that helps build the transformation template. At any point during your template creation, all work can be saved in a transformation project for later retrieval. A key strength of Output Transformation Designer is that it can be used to run transformations which you can view instantly.

Chapter 2

Data Transformation Engine window

The Data Transformation Engine window in Output Transformation Designer is where you configure all of your input and output settings. It consists of two panes:

- **Source** (input). For more information, see “Source pane“ on page 47.
- **Target** (output). For more information, see “Target pane“ on page 69.

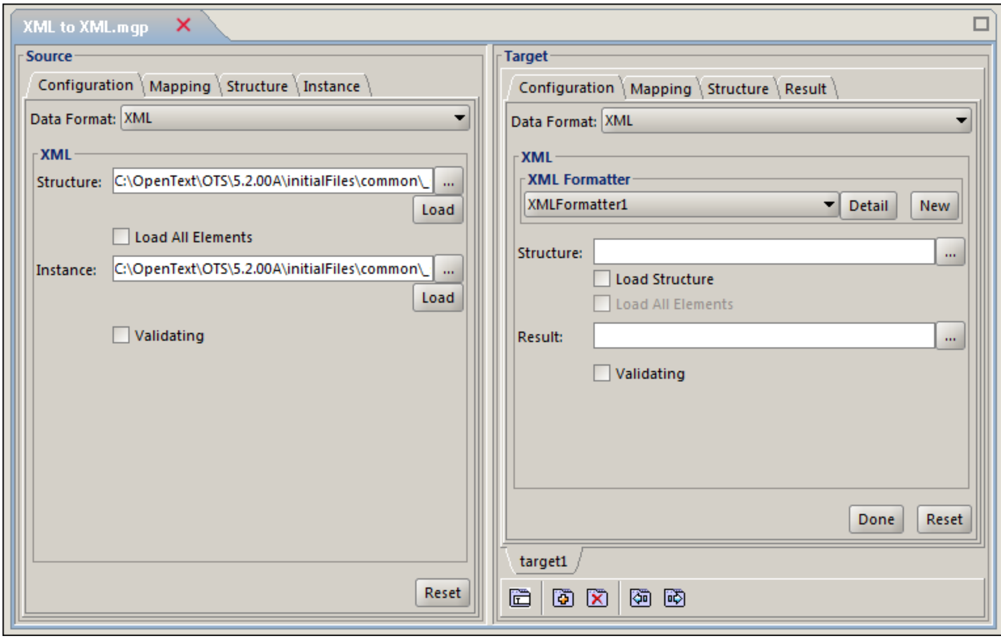


Figure 2-1: Data Transformation Engine overview

2.1 Using the Data Transformation Engine window

When creating your output structure in the **Target Mapping** tab of the **Transform Target** pane, the drag-and-drop feature allows you to move items instantly from the **Source** tab. For more information about the **Target Mapping** tab, see “Target Mapping” on page 92 and for more information about the **Source** tab, see “Source Mapping” on page 91.

You may perform multiple output transformations for any output data format with OpenText Output Transformation Server .

To add an additional output target to your transform project, click on the **Insert a tab** button, , at the bottom of the **Transform Target** pane.

Switch back and forth between targets by selecting the different tabs.

Note: You may use the value of a variable in any field that requires user input. The variable must first be defined in the **Variables** tab of the **Settings** window. For more information, see “Settings window“ on page 15

To insert the value of a variable in a field:

1. Right-click in that **field**.
2. Select **Variables** followed by the name of the variable you would like to use. The value of the variable will be determined and used when the transformation is run by the engine.

Chapter 3

Settings window

The **Settings** dialog enables you to set global properties for your project. To open the **Settings window**, you have several methods to choose from:

- Select the **MGP Settings and Details** button, , from the Output Transformation Designer toolbar.
- Click the **Detail** button on the "Target Configuration" on page 74 tab.

The **Settings** window is divided into two areas:

- "Global Transform settings" on page 19. This section has the following tabs available:
  - "Preferences tab" on page 19
  - "Variables tab" on page 22
  - "Custom Function tab" on page 26
  - "Find & Replace tab" on page 26
- "Global Data Format settings" on page 28. This section may have any of the following tabs available:
  - "JDBC Connection tab" on page 28
  - "JDBC Lookup Table tab" on page 32
  - "XML Formatter tab" on page 33
  - "XML PI tab" on page 34
  - "EDI Formatter tab" on page 35
  - "COBOL Formatter tab" on page 36
  - "COBOL IO Directive tab" on page 37
  - "Transaction tab" on page 39
  - "PDF Appended Page Formatter tab" on page 40
  - "XBRL tabs" on page 40
  - "XBRL Context tab" on page 40
  - "XBRL Entity tab" on page 42
  - "XBRL Period tab" on page 43
  - "XBRL Scenario tab" on page 44
  - "XBRL Unit tab" on page 45

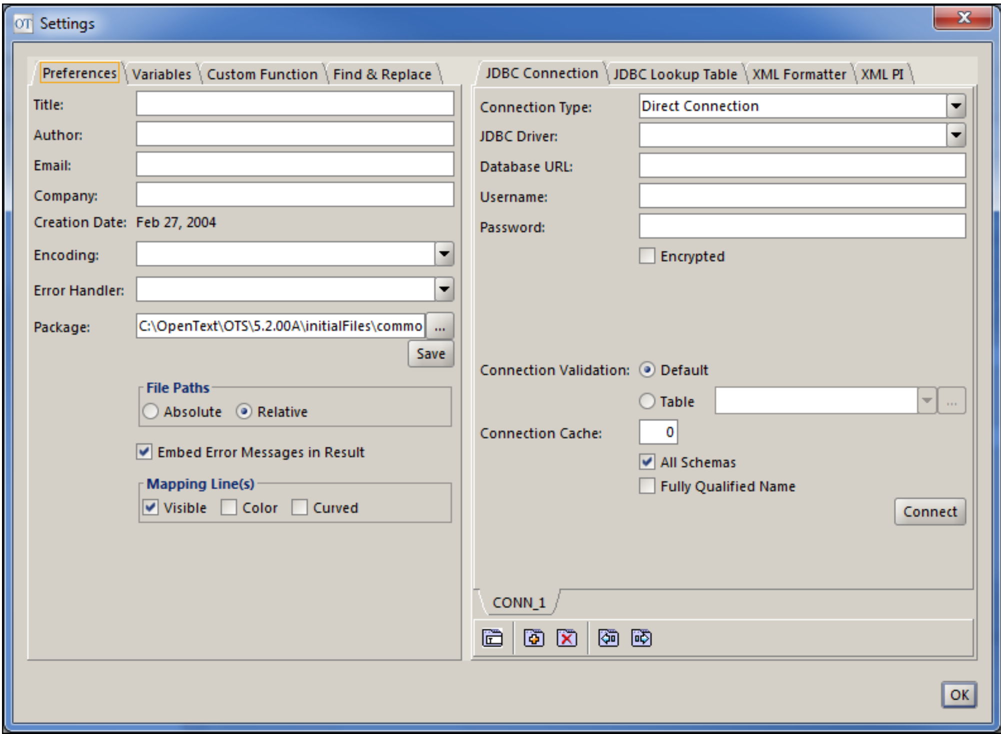


Figure 3-1: Settings window

3.1 Managing Tab and Row commands

You can define more than one transformation configuration for a single project run by managing tabs and rows within the “Settings window“ on page 15. The tab management functions appear at the bottom of the pane for many of the data format transformation configuration tabs, while the row management functions mainly exist within the configuration details tabs that require the setting of either variables or find and replace data. For more information, see “Tab Management commands” on page 17 and “Row Management commands” on page 18.

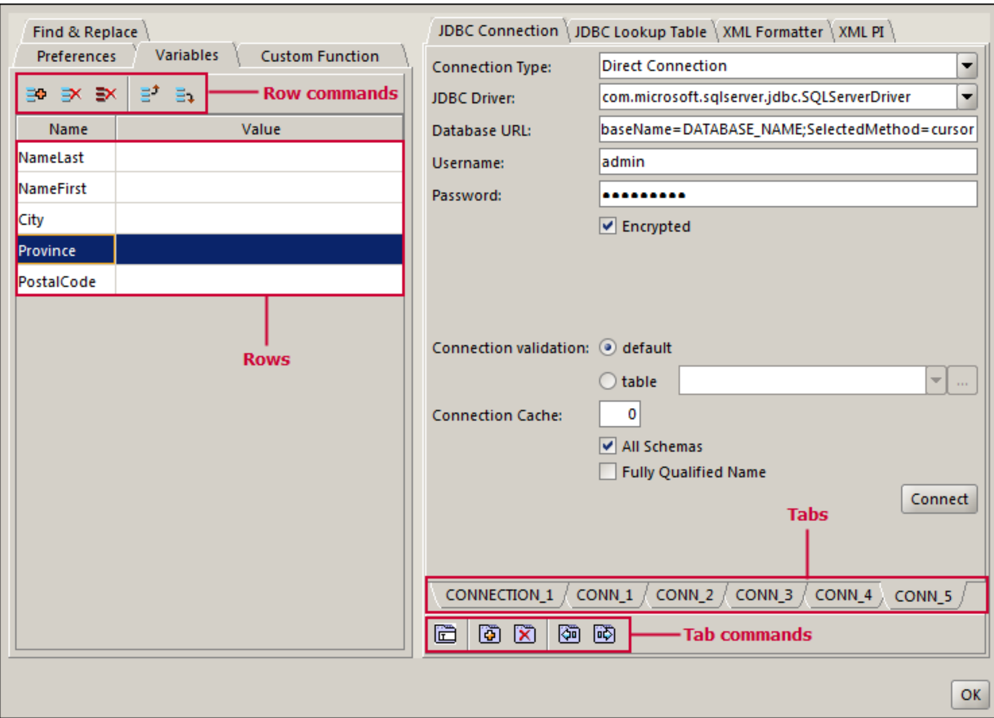
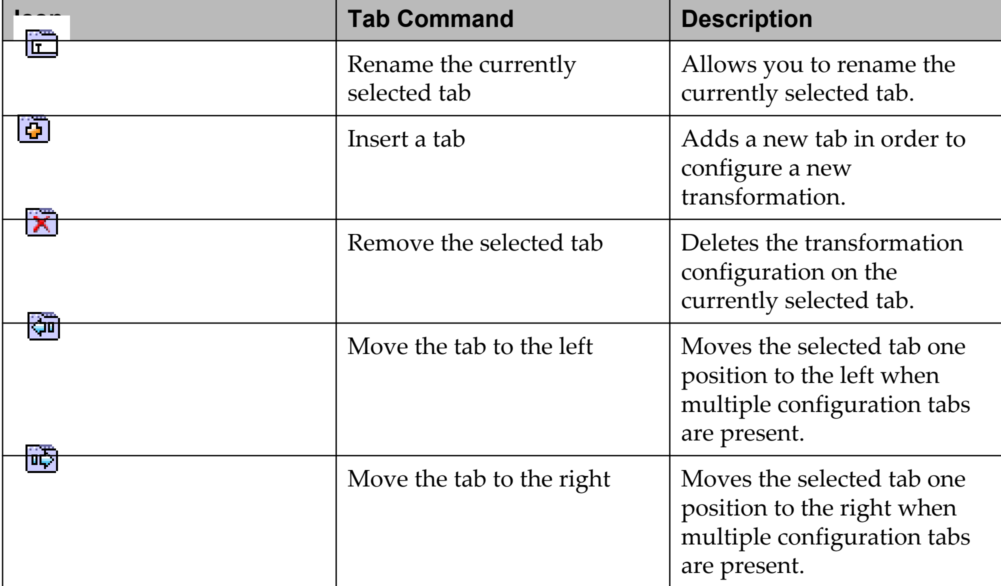


Figure 3-2: Sample Settings window

3.1.1 Tab Management commands



Icon	Tab Command	Description
	Rename the currently selected tab	Allows you to rename the currently selected tab.
	Insert a tab	Adds a new tab in order to configure a new transformation.
	Remove the selected tab	Deletes the transformation configuration on the currently selected tab.
	Move the tab to the left	Moves the selected tab one position to the left when multiple configuration tabs are present.
	Move the tab to the right	Moves the selected tab one position to the right when multiple configuration tabs are present.

Tip: The tab management commands can also be accessed by right-clicking the tab name and selecting an option from the context menu that appears.

3.1.2 Row Management commands

Icon	Tab Command	Description
	Insert a row	Adds a new data entry row on the transformation configuration tab.
	Remove the selected row	Deletes the currently selected data entry row.
	Remove all the rows	Deletes every data row on the selected transformation configuration tab.
	Move the selected row one row up	Moves the selected row one position up on the transformation configuration tab.
	Move the selected row one row down	Moves the selected row one position down on the transformation configuration tab.
	Refresh JDBC Lookup Table	Refreshes the data on the "JDBC Lookup Table tab" on page 32 with the latest match/replace entries. **Note:** This option is only available on the JDBC Lookup Table tab.
	Preview	Previews the XML processing instructions. **Note:** This option is only available on the "XML PI tab" on page 34.
	Load look-up table	Loads the JDBC Lookup Table.

Tip: Most of the row management commands can also be accessed by right-clicking the row entries and selecting an option from the context menu that appears.

3.2 Global Transform settings

The following tabs are available from the **Global Transform Settings** area of the **Settings window**.

- Preferences tab. Enables users to provide template metadata, optional data that does not affect the transformation, and file paths.
- Variables tab. Allows you to create and manage variables for use in most user-defined values within Data Transformation Engine .
- Custom Function tab. Allows you to specify your own pre-defined Java functions.
- Find & Replace tab. Matches and replaces input data values to be used in your output.

3.2.1 Preferences tab

The **Preferences tab** enables users to provide template metadata, an error handler, a file system package, and file path information.

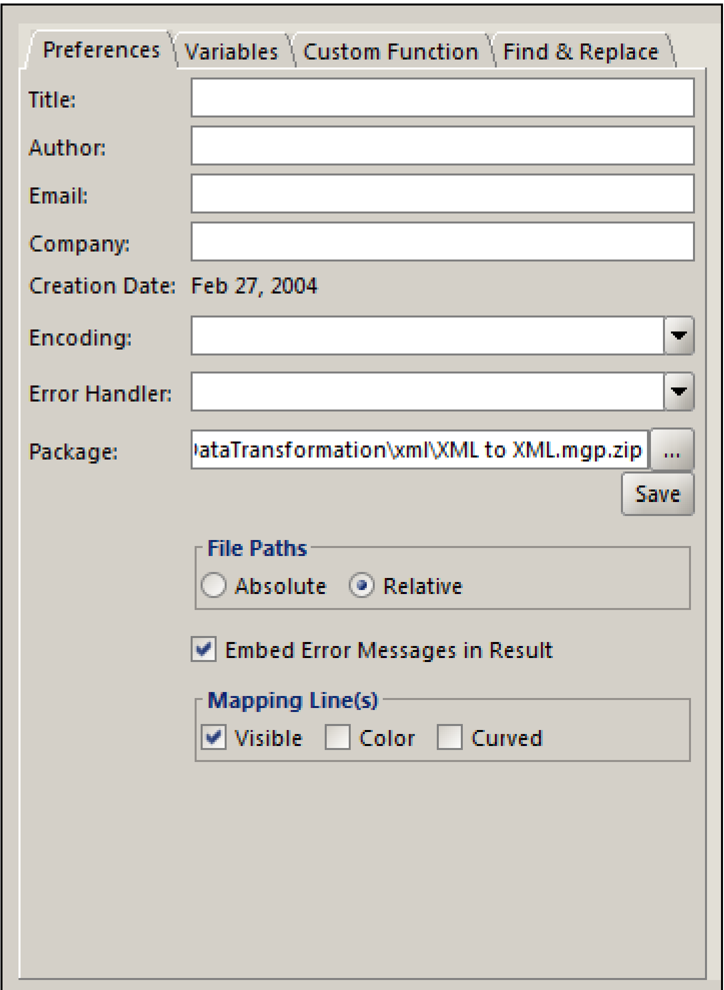


Figure 3-3: Preferences tab

3.2.1.1 Template metadata

The first five fields in this tab allow users to provide template metadata — optional data that does not affect the transformation. These include:

- **Title.** The project's title.
- **Author.** The project's author.
- **Email.** The project author's email address.
- **Company.** The company's name.
- **Encoding.** The coding that enables Data Transformation Engine to communicate with different languages and character sets. The default is UTF-8.

Note: Encoding affects the transformation, unlike the other four template metadata: Title, Author, Email, and Company.

The values you enter here are stored in your template file as template metadata, and are not added to your output.

3.2.1.2 Error Handler

Error Handler. Allows the user to specify their own error handler, which will be called when an error occurs within a transformation. This is done by entering a class name in the field. The class name implements `com.xenos.transform.ErrorHandler`. The **Stop Transform** option will stop the transformation once an error occurs.

3.2.1.3 Package and file paths

- **Package.** This configuration displays the file path where you locate your source information for the template. It is saved as a `.zip` file.
- **File Paths.** Configuration option enables you to choose between using **absolute** and **relative** file paths within your template.
- **Absolute.** Data Transformation Engine will only look at the specified file path, regardless of where the project is located. For example, by selecting Absolute file path, your project's input instance will be similar to:

  ```
  <install_home>\initialFiles\common\_sample\DataTransformation\xml\
  Input.xml
  ```

  For more information on the input instance, see “Source Configuration” on page 51.
- **Relative.** Data Transformation Engine will first look at your template's file path and then start looking for the file from there on. For example, if your template's file path is located at:

  ```
  <install_home>\initialFiles\common\_sample\DataTransformation\xml
  ```

  then by selecting **Relative file path**, your template's input instance will be similar to **Input.xml**. For more information on the input instance, see “Source Configuration” on page 51.

  The default file path is set to *Relative file path*. This will affect how filenames are written to the project, and therefore, how the run-time engine locates the files at run-time. If the template file is not found, the run-time engine will use the system's booting directory as the base path instead.
- **Embed Error Messages in Result.** This check box lets you decide whether error messages are to be written to the output or not.

3.2.1.4 Mapping lines

Using the checkboxes, you can set a few options for how mapping lines appear in your projects:

- **Visible.** Toggles the mapping lines on and off.
- **Color.** Displays mapping lines in uniform grey or color-coded to make it easier to follow the mappings.
- **Curved.** Displays curved mapping lines rather than lines with 90 degree corners.

3.2.2 Variables tab

You can create variables for use in most user-defined values within the Data Transformation Engine tab.

1. In the **Name** column, enter the name of the variable you wish to create.
2. In the **Value** column, enter the value for the variable.
3. Click **OK**.

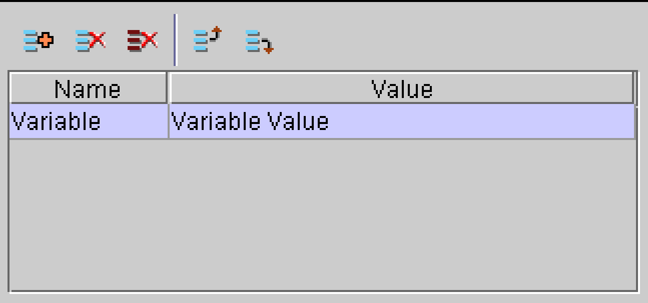


Figure 3-4: Variables tab

3.2.2.1 Variable nesting

Data Transformation Engine enables you to nest variables, that is, to use the value of a variable within the value of another variable. To insert the value of a variable you have already defined into the value of a variable you are defining, do the following:

1. Enter the $ into the **Value** field.
2. Right-click the **Value** field.
3. Select **Variables**.
4. Select the name of the pre-defined variable you wish to use.

Variable nesting is **only** supported by JDBC and SQL data formats. If a variable is nested and is used by other data formats, then the variable will be displayed as a string instead of its value.

For example, in the **Variables Configuration** pane we have:

Name	Value
condition	EMP01
table_name	Lookup
select_query	SELECT * FROM ${table_name} WHERE Value LIKE '${condition}'

If variable `select_query` is used in the XML, it will print the following:

```
<node>SELECT * FROM ${table_name} WHERE Value LIKE '${condition}'</node>
```

Whereas, if the variable `select_query` is used in the SQL or JDBC Configuration tab, then it will print the following:

```
SELECT * FROM Lookup WHERE Value LIKE 'EMP01'
```

3.2.2.2 Using a Variable in output

To use a variable in your output:

1. Right-click on the output item within the “Target Mapping” on page 92 tab.
2. Select **Map to Constant**.

   An empty field box appears.
3. Right-click within the empty field box to display the list of available variables and select the variable you wish to use.

   If there are more than 25 variables defined, see “Large numbers of variables” on page 24.

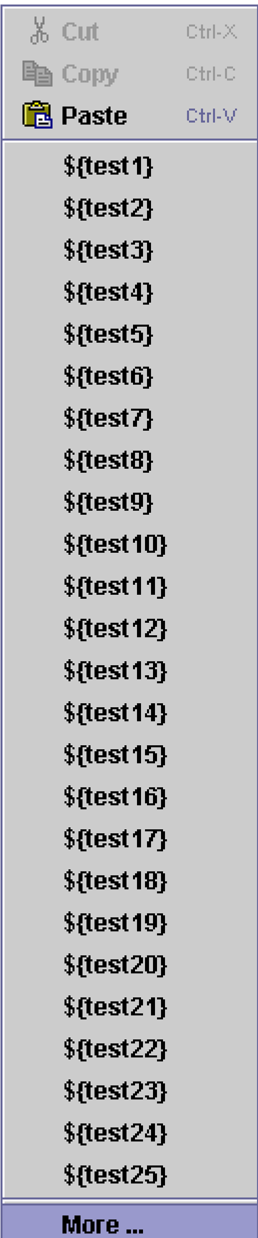
To use a variable in any other field within Data Transformation Engine :

1. Right-click in the **field**.
2. From the context menu that appears, choose the **Variables** option followed by the name of the variable you wish to use.

Note: The value of any variable may be changed when you run the actual transformation from the Data Transformation Engine run-time engine. See OpenText Embedded Data Transformation Engine Javadoc for more information, or see OpenText Embedded Data Transformation Engine Developer’s Guide .

3.2.2.2.1 Large numbers of variables

If your project has more than 25 variables defined, then when you select **Map to Constant** and right-click in the empty field box, the **Variables** context menu will include a **More...** option at the bottom.



To select a variable not included among the first 25 variables listed:

1. Click the **More...** option to open the **Variables** menu.

2. Select the variable you wish to use.
3. Click **Save**.

The variable returns the variable name, such as ${var1}.

3.2.2.3 Sorting the Variables dialog box

The **Variables** dialog box can be sorted by **Name** or **Value.** Click the table headers to sort by that criteria.

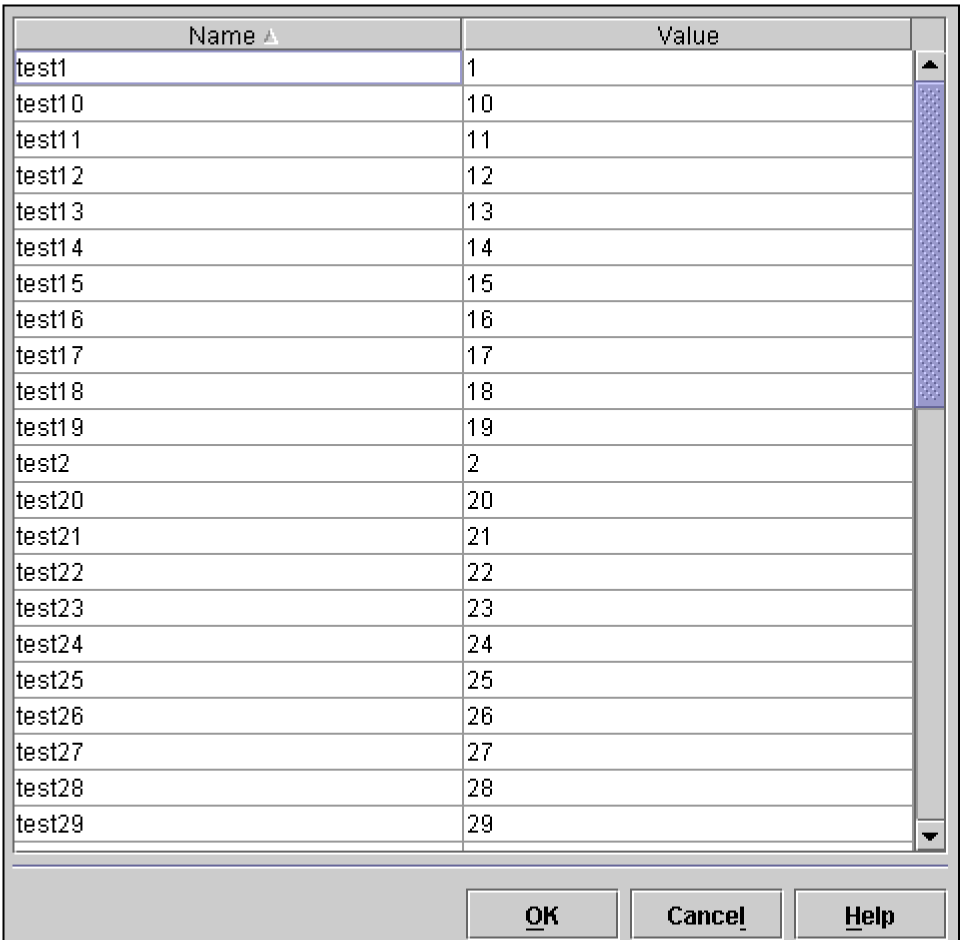


Figure 3-5: Variables dialog box

3.2.3 Custom Function tab

When a Data Transformation Engine pre-defined function does not exist to perform the task you need, you can write your own custom function.

The **Custom Function tab** lets you specify your own pre-defined Java functions by giving a function name, the number of arguments the function takes, and the location of the custom function within your system.

It is recommended that you store custom functions within the `\external_functions` directory, for example:

```
<install_home>\dev_studio\external_functions
```

If you do not store custom functions within the `\external_functions` directory, you must add the location of your custom function to the *classpath* used by Data Transformation Engine .

For more information about the Data Transformation Engine classpath, see *OpenText Embedded Data Transformation Engine - Developer's Guide (VDTOTS-H-PDT)* in OpenText Embedded Data Transformation Engine Developer's Guide.

3.2.3.1 Custom Function Wizard

At the bottom of the **Custom Function tab** is the **Custom Function Wizard** button, which opens the wizard's window. For more information on the wizard, see “Custom Function Component Wizard” on page 198.

For more information on custom functions, see “Defining custom functions” on page 189.

3.2.4 Find & Replace tab

The **Find & Replace tab** lets you match and replace input data values to be used in your output.

Note: Once you have defined your find/replace pair, you must apply it to the desired output item by means of the Replace function, as described in the list of pre-defined functions.

For example, suppose that your input data has a field called “r;car_age” whose value is either *new* or *used*. You have decided that you don't like the term *used* and want to use *pre-owned* instead. To perform this substitution:

1. Click the **Insert Row** button, , at the top of the pane.
2. In the row that appears, enter:
    - *used* under the **Find What** column.
    - *pre-owned* under the **Replace With** column.

3. If the output item you would like to apply this to is called *Age*, then right-click *Age* in the “Target Mapping” on page 92 tab.

4. Specify a **Replace** function whose parameters are the input item *car_age* and the name of the Find & Replace tab (located at the bottom of the pane).

3.2.4.1 Import a Look-up list

To import a look-up code list from SV file or from a JDBC Connection table, click the **Load Look-up Table...** button, , to bring up the **Load Look-up Table** dialog.

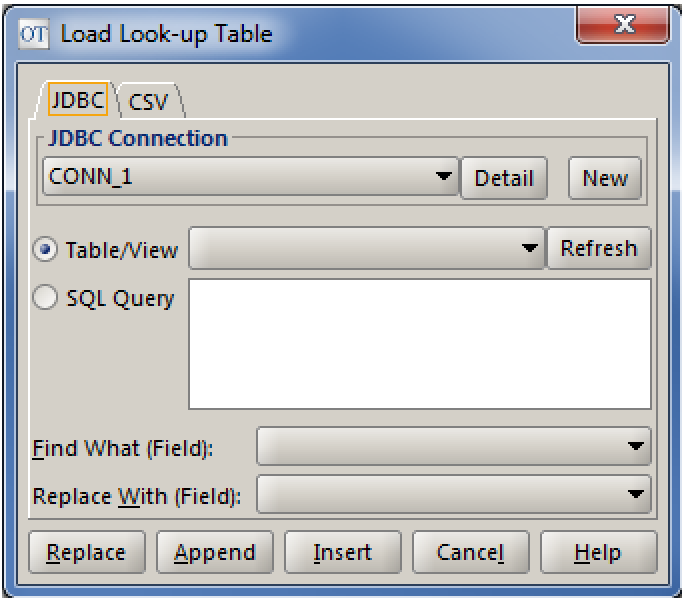


Figure 3-6: Load Look-up Table window

If the CSV file contains multiple values with the same key, the first key-value pair is imported and the successive ones are not imported.

There are five buttons:

- **Replace.** Replaces the values in the **Replace With** column with the new column specified, matching the values in the **Find What** column.
- **Append.** Adds the imported list to the end of the look-up table.
- **Insert.** Adds the imported list to the beginning of the selected row in the look-up table.
- **Cancel.** Stops the loading of the table and closes the **Load Look-up Table** window.
- **Help.** Launches the online help.

3.2.4.2 Load Look-Up table using a SQL statement

1. Select the **SQL Query** radio button.
2. Type an SQL Query statement without any semicolons in the end like:
    - SELECT **key value** FROM lookup, WHERE key='B'.
    - SELECT * **tablename**, WHERE tablename is the name of the database table.
3. Click the **Find What** and **Replace With** drop-down lists and select the appropriate columns.

3.3 Global Data Format settings

The tabs that appear in the Global Data Format Settings dialog box are dependent upon the data formats used. You can configure multiple format settings of the same type at one time. For more information on how to accomplish this, see “Managing Tab and Row commands” on page 16.

3.3.1 JDBC Connection tab

The **JDBC Connection** tab contains the configuration properties required for setting up transactions through the Java Database Connectivity (JDBC) method. If either your input or output data format is using JDBC, or you will be accessing databases to perform JDBC find and replace operations, you must provide the necessary information in order to access the database.

Data Transformation Engine also supports JDBC connections through the Output Transformation Server Persistent Storage Pool and JNDI. Additionally, when you connect to a database using the JNDI Connection option in Data Transformation Engine , you enable several applications within the same JEE server to access the same database connection pool. Refer to the topics in this section for further instructions on some supplementary configuration steps that need to be completed in order to facilitate the connections.

The supported JDBC connection formats share some common connection validation and cache configuration settings. For more information, see “JDBC Connection Validation and Cache settings” on page 29.

Once your connection settings are complete, remember to select which connection you would like to use for the transformation in the “Source Configuration” on page 51 and “Target Configuration” on page 74 tabs, or the “JDBC Lookup Table tab” on page 32 of the “Settings window“ on page 15.

3.3.1.1 JDBC Connection Validation and Cache settings

The connection validation and cache configuration settings are common to all supported JDBC connection methods. Their default settings are acceptable for most situations, but opting to establish these properties can help to maximize the performance of your transformations.

Connection Validation

Since Data Transformation Engine pools the database connections, these connections can occasionally time out. The Connection validation settings determine how to validate whether a connection is still active or whether it needs to be reconnected. This setting automatically verifies all connections before they are made, however, you can control which branch of the connection is tested.

When the **Default** option is selected, the application hits the Java method `connection.getAutoCommit()`, which works with most databases. When **Table** is selected, the query “SELECT 1 FROM table_name;” is run. If you are performing validation using the table method, you must enter all your database connection information in the appropriate fields because the table names must be retrieved from the database before the table selection dropdown menu can be populated.

Tip: It is recommended that you experiment with the Default scheme prior to trying the Table method as Default is more efficient.

Connection Cache

The **Connection cache** field specifies the number of JDBC connections to initialize in the global connection pool. The default value of **0** indicates that no connections are initialized in the pool and that any required JDBC connections should only be created during the time of the transformation. Subsequently, the size of the pool will expand to accommodate the maximum number of concurrent transformations that will occur. If you are running a few transformation threads concurrently, it is highly recommended that you initialize an adequate number of threads in order to increase performance.

Enabling the **All schemas** option loads all the tables for all the schemas found in the database. When this option is disabled, upon connecting to the database all available schemas from the database will appear in the **Database Schemas** window where you can select which schema tables to load. The default setting is to load all tables.

Using the **Fully Qualified Name** option presents both the schema and table name in order to help you distinguish between tables with the same name. Occasionally, you may encounter multiple tables with an identical name, all originating from different schemas, which is where displaying the fully qualified name for the JDBC tables proves indispensable. For example, if you are using an Oracle JDBC connection with the user name Smith and you create a new table called Employees, when the Fully Qualified Name is selected, the table name appears as `Smith.Employees` while it would only show as `Employees` when this option is not selected. By default, this setting is disabled.

Note: The **Fully Qualified Name** option can only be used for JDBC connections that support schemas. Microsoft Access, for example, does not support the use of schemas.

3.3.1.2 Setting up a Direct JDBC connection

To make a direct JDBC connection:

1. On the **JDBC** tab of the **Settings** screen, from the **Connection Type** dropdown menu, select **Direct Connection**.

   The options on the JDBC tab change to reflect the new connection type.

2. From the **JDBC Driver** dropdown menu, select the appropriate JDBC driver to use according to the type of database server you are connecting to.

3. In the **Database URL** field, a predefined string of text is present in the basic JDBC URL format, which changes depending on your selected JDBC driver. You must replace certain portions of the URL name with the pertinent information for your database. For example, if you are using the MySQL JDBC driver, `jdbc:mysql://HOST_NAME/DATABASE_NAME` appears in the Database URL field. You must supplant the HOST_NAME and DATABASE_NAME excerpts with the appropriate information for your database.

4. If applicable, enter your server's **User name** and **Password** information into their respective fields.

5. You must indicate whether the data should be encrypted during the transaction. If you require your data to be encoded, select the **Encrypted** checkbox. By default, this is disabled.

6. Configure the **Connection validation** and **Connection cache** settings for the JDBC connection, as required for your transaction. For more information on these settings, see “JDBC Connection Validation and Cache settings” on page 29.

7. When you are finished, click **Connect**.

   A dialog opens stating that the database connection was successfully made.

3.3.1.3 Setting up a JDBC connection via Persistent Storage Pool

The Persistent Storage Pool is the set of connections configured within the OpenText Output Transformation Server system configuration. If you are running Data Transformation Engine within Output Transformation Server , it is suggested that you set up the Persistent Storage Pool within Output Transformation Server instead of within Data Transformation Engine for better manageability of the JDBC connection.

To set up a JDBC connection via Persistent Storage Pool:

1. On the **JDBC** tab of the **Settings** screen, from the **Connection Type** dropdown menu, select **Persistent Storage Pool**.

The options on the JDBC tab change to reflect the new connection type.

2. If you already have some predefined Persistent Storage Pool configurations set up, you can select one from the **Persistent Storage** dropdown menu and skip to step 5.

   If you want to create a new set of Persistent Storage Pool properties, next to the **Persistent Storage** field, click the **Ellipsis** button.

   The **Modify Persistent Storage Pool Configuration** screen opens. Proceed to the next step.

3. On the **Modify Persistent Storage Pool Configuration** screen, right-click the **PersistentStorage** parameter name and from the context menu that appears, select **Add**.

   A new set of Persistent Storage Pool configuration parameters appears.

4. You must enter values for each of the parameters according to your server information. When you are finished, click **Save**.

   The **Settings** dialog displays.

5. Configure the **Connection validation** and **Connection cache** settings for the JDBC connection, as required for your transaction. For more information on these settings, see "JDBC Connection Validation and Cache settings" on page 29.

6. When you are finished, click **Connect**.

   A dialog opens stating that the database connection was successfully made.

3.3.1.4 Setting up a JDBC connection via JNDI

To make a JDBC connection via JNDI:

1. Ensure that you have the correct JDBC connector or API in the application server's classpath.

   Note: If you are using MySQL, you must be using `mysql-connector-java-5.0.8-bin.jar` or newer in order to properly retrieve the list of tables, otherwise an exception will be thrown.

2. On your application server, ensure that your **DataSource** is defined as a **JNDI connection**.

3. In your application server's connection pool, establish a minimum of 2 connections for Data Transformation Engine's use. If you have other applications that will also be using this JNDI connection, you must increase the number in your connection pool accordingly.

4. In Data Transformation Engine, access the **Settings** dialog for the JDBC connection you want to configure.

5. On the **JDBC Connection** tab, from the **Connection Type** dropdown menu, select **JNDI Connection**.

6. In the **JNDI name** field, you must supply your JNDI's name. Depending on your JNDI settings, it should appear in either java:/jdbc/test or jdbc/test format.

7. You must enter the proper context factory for your application server in the **Context factory** field.

Note: If you are using a secure JNDI connection or defining any roles/ permissions for your JNDI connection, you must keep in mind that your context factory may change. If either of these situations apply to you, you should provide the user name and password for accessing the JNDI connection so that links can be made without hindrance.

8. In the **URL** field, you must enter your URL connection information and the JNDI connection port number, based on your application server's settings. For instance, the entry should resemble the following example:

   WebSphere: t3:<localhost>:7001

9. If applicable, enter your server's **User name** and **Password** information into their respective fields.

10. You must indicate whether the data should be encrypted during the transaction. If you require your data to be encoded, select the **Encrypted** checkbox. By default, this is disabled.

11. Configure the **Connection validation** and **Connection cache** settings for the JDBC connection, as required for your transaction.

12. When you are finished, click **Connect**.

    A list of the database's tables is retrieved and a dialog appears stating that the database connection was successfully made.

3.3.2 JDBC Lookup Table tab

The **JDBC Lookup Table tab** enables you to set up match/replace pairs in a similar fashion to the “Find & Replace tab” on page 26. The difference is the values used here are found in a database. Once you have defined your find/replace pair, you must apply it to the desired output item by means of the “REPLACE” on page 157 function, as described in the list of pre-defined functions.

You may define more than one JDBC lookup configuration by adding new lookup tabs with the **Insert a tab** button, , at the bottom of the pane. For more information on configuring and working with multiple tabs and rows, see “Managing Tab and Row commands” on page 16.

From the drop-down box at the top of the tab, select which JDBC connection you will be using to connect to your database. You should have already defined the connection “JDBC Connection tab” on page 28 of the “Settings window“ on page 15.

The **Detail** button displays the settings of the currently selected JDBC connection, while the **New** button enables you to configure a new connection.

Fill in the SQL statement to access the table containing your match/replace pairs in the **SQL Statement** field. For each match/replace pair you enter in the **JDBC Find & Replace** section, you must enter a unique ID that you will use as a parameter in the Replace function later.

Note: You may not use the same ID in different lookup configurations. The values you enter in the **Find What** and **Replace With** columns are the names of the columns from which you will be retrieving your match and replace values, respectively.

If you decide to update your database lookup table while working on your project template, you will need to click the **Refresh JDBC Lookup Table** button, , to retrieve the updated lookup table.

3.3.3 XML Formatter tab

The **XML Formatter** tab enables you to add custom formatting to XML output.

You may define more than one XML output formatting configuration by adding new formatting tabs with the **Insert a tab** button, , at the bottom of the pane. For more information on configuring and working with multiple tabs and rows, see "Managing Tab and Row commands" on page 16. Once your settings are complete, **remember** to select which formatter you would like applied to your XML output in the "Target Configuration" on page 74 tab.

XML Formatter Public and System ID

Public and System ID are for XML Doctype Public ID and System ID, like:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/
strict.dtd">
```

PUBLIC ID: `-//W3C//DTD HTML 4.01//EN`

System ID: `http://www.w3.org/TR/html4/strict.dtd`

Stylesheet type

You can also specify which XML stylesheet (XSLT) file to use when viewing the XML output in an Internet browser. When generating the output XML file, Data Transformation Engine will add the XML stylesheet information to the XML output. You will see that the XML output is formatted when viewed in an Internet browser.

Currently, only text/xsl are supported. For example:

```
<?xml-stylesheet type="text/xsl" href="<installation_directory>\initialFiles\common
\DataTransformation\samples\xml\abc.xslt"?>
```

When the XML file is opened in a Web browser, the browser will refer to this stylesheet for rules on how to present the XML to the browser.

The designation of a stylesheet to use with a particular XML or HTML document is the most popular use of XML processing instructions. It is just one example of their

ability to pass along information that does not fit into a document's regular structure.

XML processing instruction

This is used mainly for including an XML stylesheet like the one in the “Stylesheet type” on page 33 example.

Example: `<?PI_1 acord="1.4"?>`

3.3.4 XML PI tab

To add the XML processing instruction, such as the following:

```
<?acord version="1.4.1"?>
```

In the Data Transformation Engine tab, follow this procedure:

1. Open the **Configuration** pane.
2. Select the **Details** button to open the “Settings window“ on page 15.

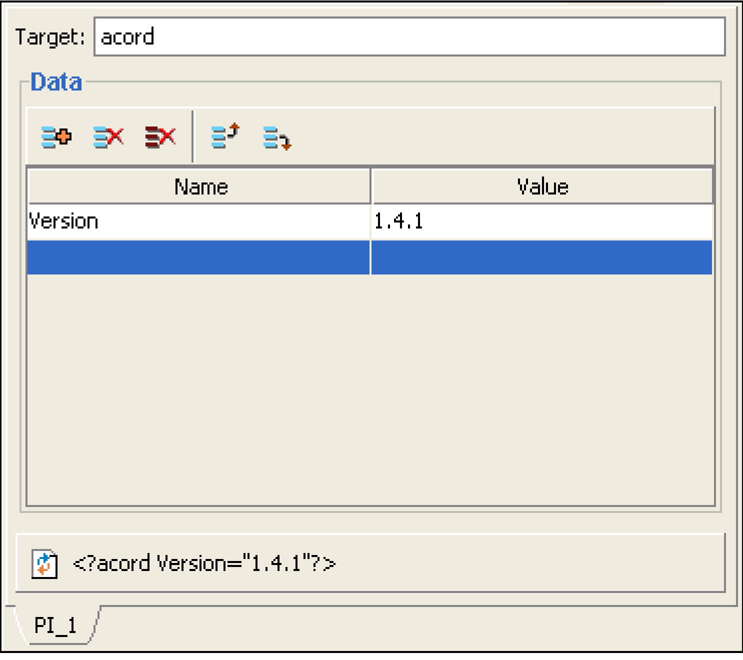


Figure 3-7: XML PI tab

3. Select the **XML PI** tab.

4. In the **Target** field, type `acord`.

5. Add a Row. Click **Insert a row**.

6. Enter the following in the table row:

   - Under **Name** enter `Version`.
   - Under **Value** enter `1.4.1`.

   To add more attributes, repeat Step 4 to Step 6.

7. Clicking the **Preview** button, , at the bottom of the screen will display a preview of the processing instructions that will be inserted in the top of the XML file.

8. Switch to the "XML Formatter tab" on page 33 of the "Settings window" on page 15.

9. Select the **PI_1** processing instruction.

10. Select the **XML_1 formatter** on the **Configuration** pane.

11. Run the transform. In the output XML file, you should now see the acord XML processing instruction:

```
<?acord version="1.4.1"?>
```

   Hold the **Ctrl** key and **left-click** on items in the list of processing instructions to select or deselect multiple items.

You may define more than one XML PI formatting configuration by adding new formatting tabs with the **Insert a tab** button, , at the bottom of the pane. For more information on configuring and working with multiple tabs and rows, see "Managing Tab and Row commands" on page 16.

3.3.5 EDI Formatter tab

When your output data format is EDI, use the **EDI Formatter** tab to tell Data Transformation Engine what symbols you would like to use as your delimiters.

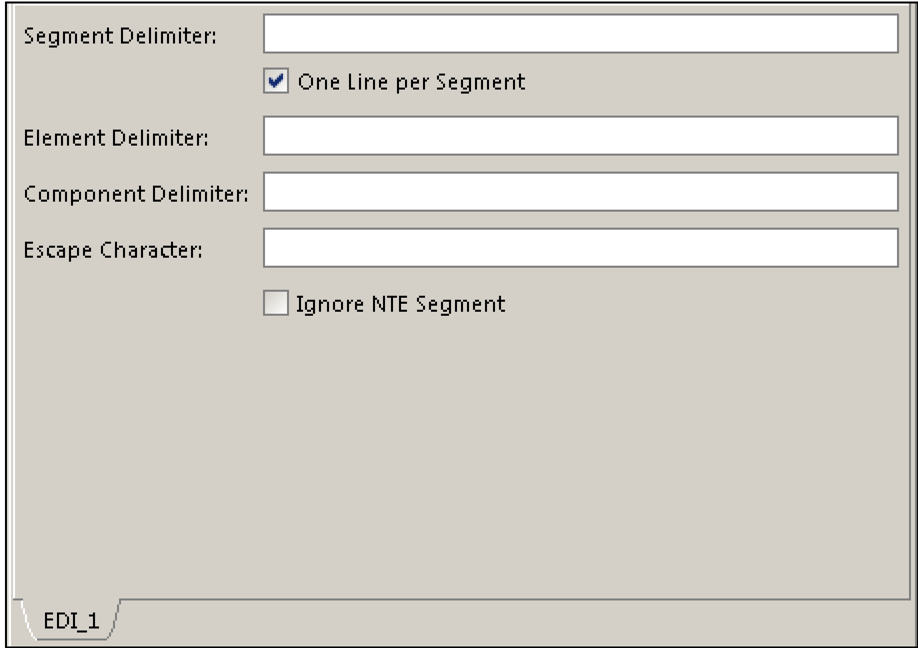


Figure 3-8: EDI Formatter tab

You may define more than one EDI output-formatting configuration by adding new formatting tabs with the **Insert a tab** button, , at the bottom of the pane. For more information on configuring and working with multiple tabs and rows, see "Managing Tab and Row commands" on page 16.

Once your settings are complete, remember to select which formatter you would like applied to your EDI output in the "Target Configuration" on page 74 tab.

3.3.6 COBOL Formatter tab

The **COBOL Formatter tab** enables you to add custom formatting to COBOL output.

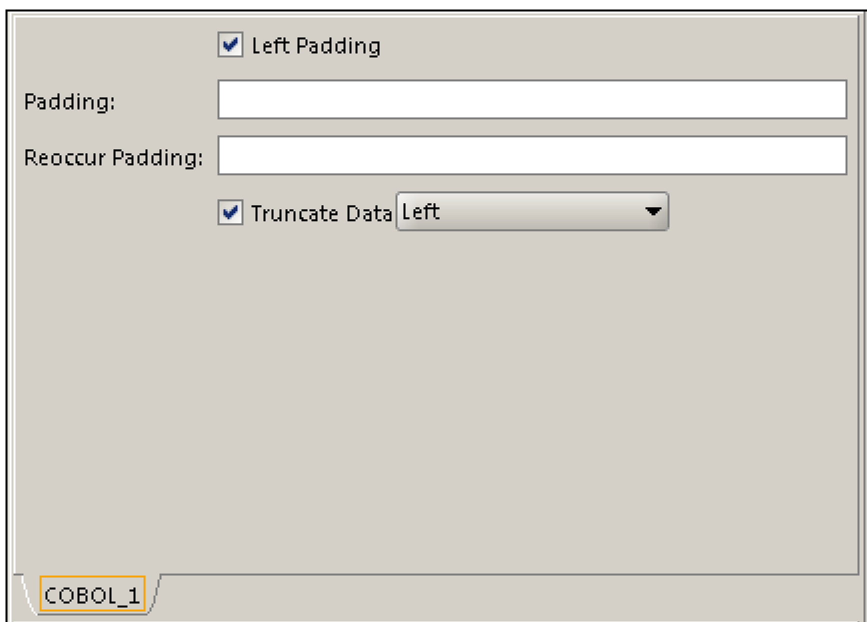


Figure 3-9: COBOL Formatter tab

You may define more than one COBOL output formatting configuration by adding new formatting tabs with the **Insert a tab** button, , at the bottom of the pane. For more information on configuring and working with multiple tabs and rows, see "Managing Tab and Row commands" on page 16.

If **Truncate Data** is checked, Data Transformation Engine will truncate strings that exceed the length as defined in the COBOL copybook structure.

- **Left**. Truncates data off the left end of the string.
- **Right**. Truncates data off the right end of the string.

Once your settings are complete, remember to select which formatter you would like applied to your COBOL output in the "Target Configuration" on page 74 tab.

3.3.7 COBOL IO Directive tab

When a COBOL IO directive is not chosen on the "Target Configuration" on page 74 tab and COBOL format is selected for the source, the default IO directive is a record based IO directive that does not contain record descriptor words.

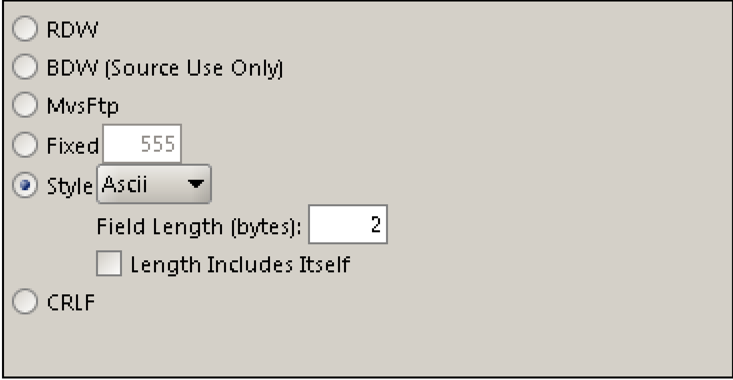


Figure 3-10: COBOL IO Directive tab

You may define more than one COBOL IO directive output formatting configuration by adding new formatting tabs with the **Insert a tab** button, , at the bottom of the pane. For more information on configuring and working with multiple tabs and rows, see "Managing Tab and Row commands" on page 16.

The **COBOL IO Directive tab** enables you to define the COBOL IO directives:

- **RDW.** Record descriptor words, common on MVS.
- **BDW.** Block descriptor words, common on MVS, input only.
- **MvsFtp.** A three byte length field used for MVS FTP file transfer.
- **Fixed.** All records are fixed length. Enter the record length, for example, 555.
- **Style.** Formats include:
  - **ASCII**.
  - **EBCDIC**.
  - **IntelInt (Intel integer, default).**
  - **MvsInt (MVS/Sun/Aix/HP integer).**
- **Field Length (bytes).** Indicates the number of bytes that take up the integer, typically 2 or 4.
- **Length Includes Itself.** Check this box when the field length is inclusive.
- **CRLF.** Line based, a DOS CRLF or Unix-style LF follows each record.

Note: It is recommended that you choose an appropriate COBOL IO directive instead of using the default IO directive.

3.3.8 Transaction tab

The **Transaction tab** enables you to customize how you commit your output to a database.

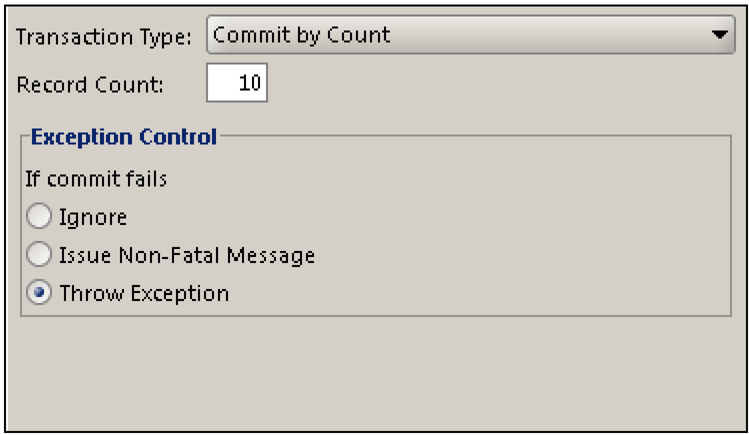


Figure 3-11: Transaction tab

There are three types of commit options to choose from:

- **Auto Commit.** This commits your output to the database after each record is processed.
- **Commit by Count.** With this option, your data will be committed to the database after a specified number of records have been processed. Define this number in the **Record Count** field.
- **Commit by Context.** This tells Data Transformation Engine to commit based on the successful processing of a complete level in your input hierarchy. If any single item at a particular level fails to be processed properly, none of the other items at that level is committed.

There is also the option of throwing an exception if a commit fails. By not selecting this box, Data Transformation Engine will continue with the transformation process when data fails to commit to the database. If selected, an exception will be thrown.

3.3.9 PDF Appended Page Formatter tab

The **PDF Appended Page Formatter** tab enables you to configure the layout of the appended PDF pages.

You may define more than one PDF appended page formatting configuration by adding new formatting tabs with the **Insert a tab** button, , at the bottom of the pane. For more information on configuring and working with multiple tabs and rows, see "Managing Tab and Row commands" on page 16.

Complete the following fields to configure the appended pages layout:

- **Column Count**
- **Row Count**
- **Margin Top**
- **Margin Left**
- **Font Size**
- **Font Color.** Click the **Ellipsis** button, , to launch the **Color Picker** window and select your desired color.
- **Line Height**
- **Word Wrap**
- **Tab Space**

3.3.10 XBRL tabs

XBRL tabs are used for configuring XBRL context and the sub-elements of the context, including Entity, Period, Scenario, and Unit (for numeric context only).

3.3.10.1 XBRL Context tab

The **Context** element contains information which is necessary for understanding a business fact captured as an XBRL item, such as information about the:

- **Entity being described**, see .
- **Reporting period**, see "XBRL Period tab" on page 43.
- **Reporting scenario**, see "XBRL Scenario tab" on page 44.

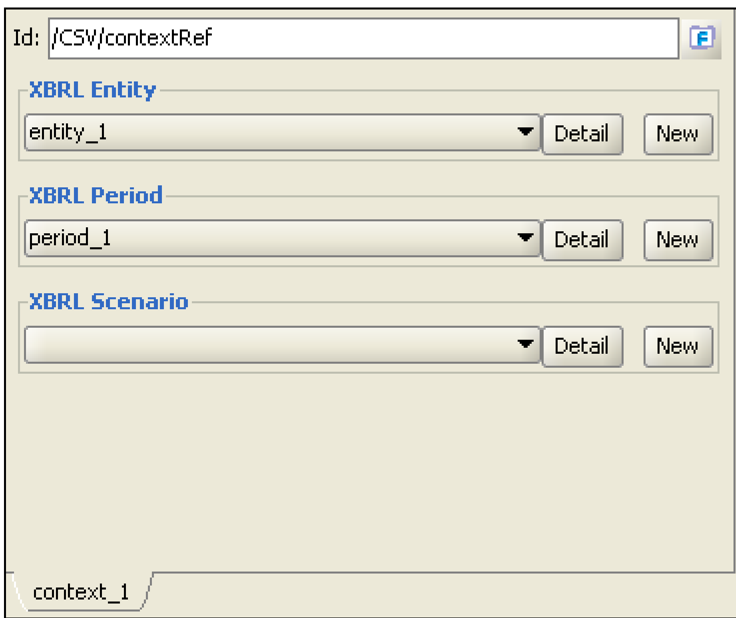


Figure 3-12: XBRL Context tab

You may define more than one XBRL context configuration by adding new formatting tabs with the **Insert a tab** button, , at the bottom of the pane. For more information on configuring and working with multiple tabs and rows, see "Managing Tab and Row commands" on page 16.

The following fields are to be completed:

- **Id**. The ID attribute identifies the context so that it may be referenced by item elements.

There are three methods for entering the context ID:

- Typing it manually in the **Id** field.
- Dragging a node from the **Mapping** pane onto the **Id** field.
- Using the "Function Editor" on page 119; click the **F** icon, , to the right of the **Id** field. For example, in the sample project `xml-xbrl.mgp`, you want the context ID to match the node `/node/xbrl@contextRef`. Instead of typing it in the **Id** field, you can drag the `contextRef` node from the "Source Mapping" on page 91 tab to the XBRL context Id field.

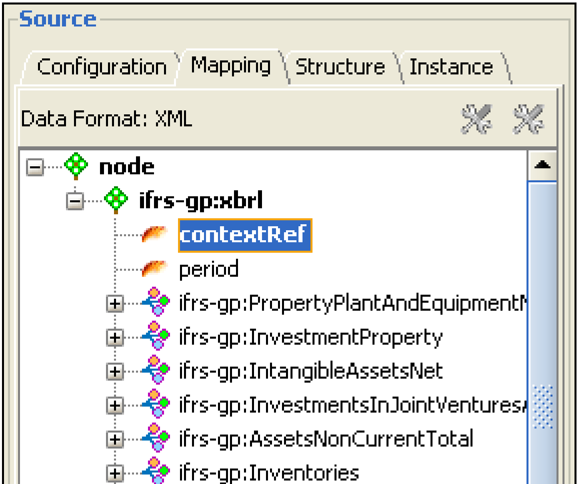


Figure 3-13: Context ID Node

Id can have the values:

- **@CONCAT(/node/xbrl/context@id)** for multiple contexts where `/node/xbrl/context@id="2003-12-31"` and `/node/xbrl/context@id="2002-12-31"`
- **"2002-12-31"** if you only have one context in the input.
- Select an **Entity**, **Period** and **Scenario** from the drop-down list.
- Click **Detail** to see the tab associated with the attribute.
- Click **New** to create a new entry.

3.3.10.2 XBRL Entity tab

The **Entity** element documents the entity (business, government department, individual, etc.) described by the business fact. The Entity element is required content of the context element.

Figure 3-14: XBRL Entity tab

You may define more than one XBRL entity configuration by adding new formatting tabs with the **Insert a tab** button, , at the bottom of the pane. For more information on configuring and working with multiple tabs and rows, see "Managing Tab and Row commands" on page 16.

3.3.10.3 XBRL Period tab

The **Period** element contains the instant or interval of time for reference by an item element. Period is mandatory for "XBRL Context tab" on page 40.

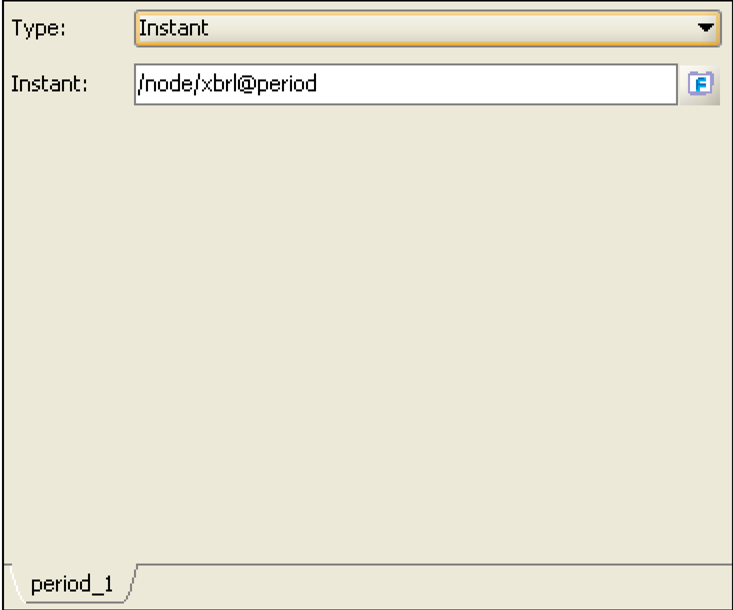


Figure 3-15: XBRL Period tab

You may define more than one XBRL period configuration by adding new formatting tabs with the **Insert a tab** button, , at the bottom of the pane. For more information on configuring and working with multiple tabs and rows, see “Managing Tab and Row commands” on page 16.

3.3.10.4 XBRL Scenario tab

The **Scenario** element is used to indicate the circumstances of the reported items. The Scenario element provides additional markup validation for a greater variety of additional metadata which preparsers can associate with items.

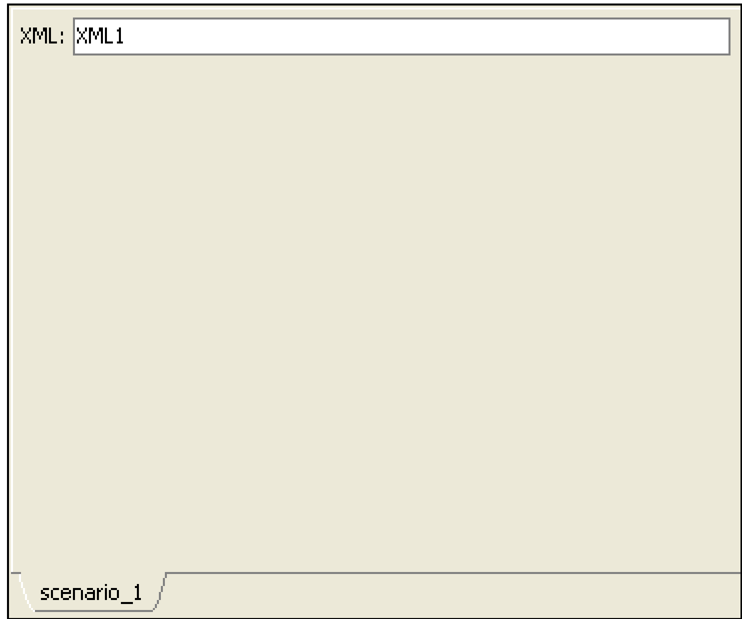


Figure 3-16: XBRL Scenario tab

You may define more than one XBRL scenario configuration by adding new formatting tabs with the **Insert a tab** button, , at the bottom of the pane. For more information on configuring and working with multiple tabs and rows, see "Managing Tab and Row commands" on page 16.

3.3.10.5 XBRL Unit tab

The **Unit** element specifies the units in which a numeric item has been measured.

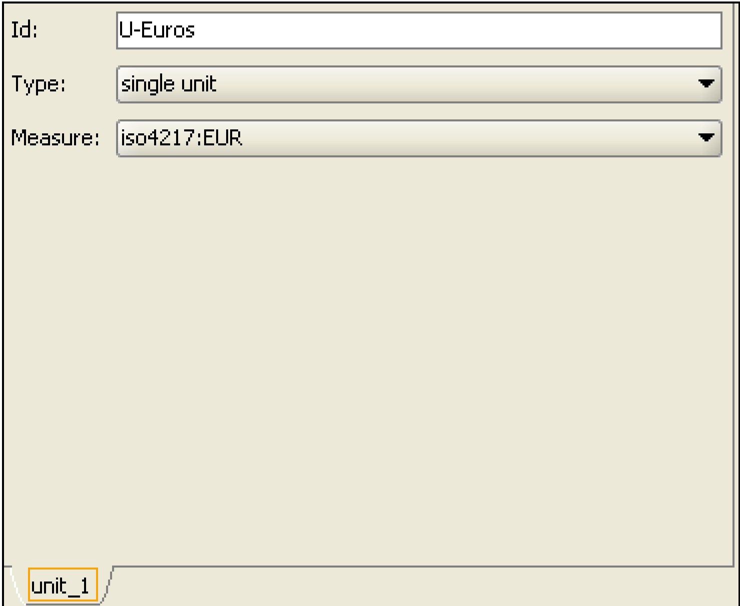


Figure 3-17: XBRL Unit tab

You may define more than one XBRL unit configuration by adding new formatting tabs with the **Insert a tab** button, , at the bottom of the pane. For more information on configuring and working with multiple tabs and rows, see “Managing Tab and Row commands” on page 16.

3.3.11 Transformations in multiple thread environments

You may find it useful to run several transformations in separate threads at the same time.

You can set up a transformation for XML to multiple databases, or from multiple databases to XML. For more information, see *OpenText Embedded Data Transformation Engine - Developer’s Guide (VDTOTS-H-PDT)*.

Chapter 4

Source pane

The Data Transformation Engine **Source** pane is divided into the following tabs:

- Source Configuration. The **Configuration** tab is where you define the data format and structure of your input, and optionally the input data source (instance) to be used for the transformation you can run within Output Transformation Designer .
- Source Mapping. The file you have chosen as your input structure is displayed graphically in the **Mapping** tab.
- Source Structure. The **Structure** tab displays the file you have loaded as your input structure file.
- Source Instance. The **Instance** tab displays the input data source you have chosen.

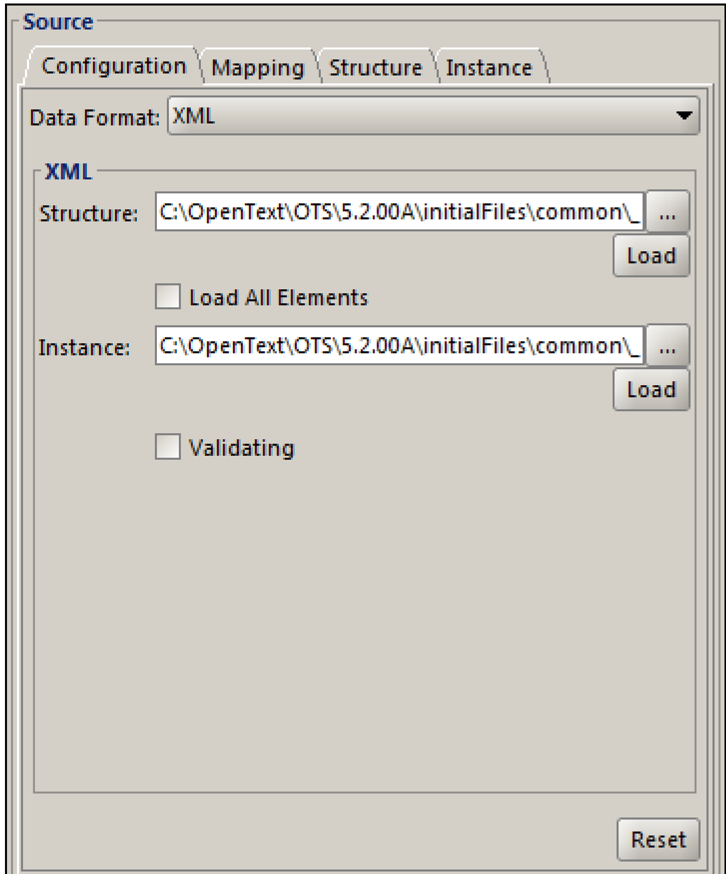


Figure 4-1: Source pane

4.1 Source Structure

The **Structure tab** displays the file you have loaded as your input structure file. The structure of this file is shown graphically in the “Source Mapping” on page 91 tab.

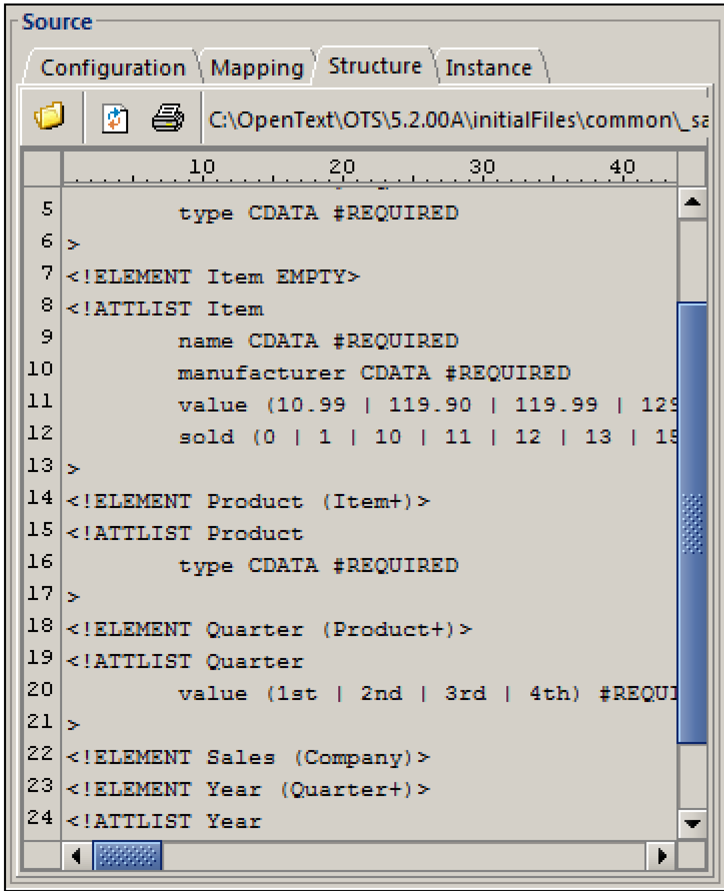


Figure 4-2: Source Structure tab

Underneath the tabs are the following buttons:

Icon	Menu Command	Description
	Open the file in a text editor	Opens up the source file in a separate editor within a new window for you to manually make changes to the code.
	Refresh the source structure	Refreshes the source code to show the most recent version.
	Print the source structure	Prints out a copy of the source structure.

4.2 Source Instance

The **Instance tab** displays the input data source you have chosen. This is the file that will be used in test transformations you run within the Data Transformation Engine window.

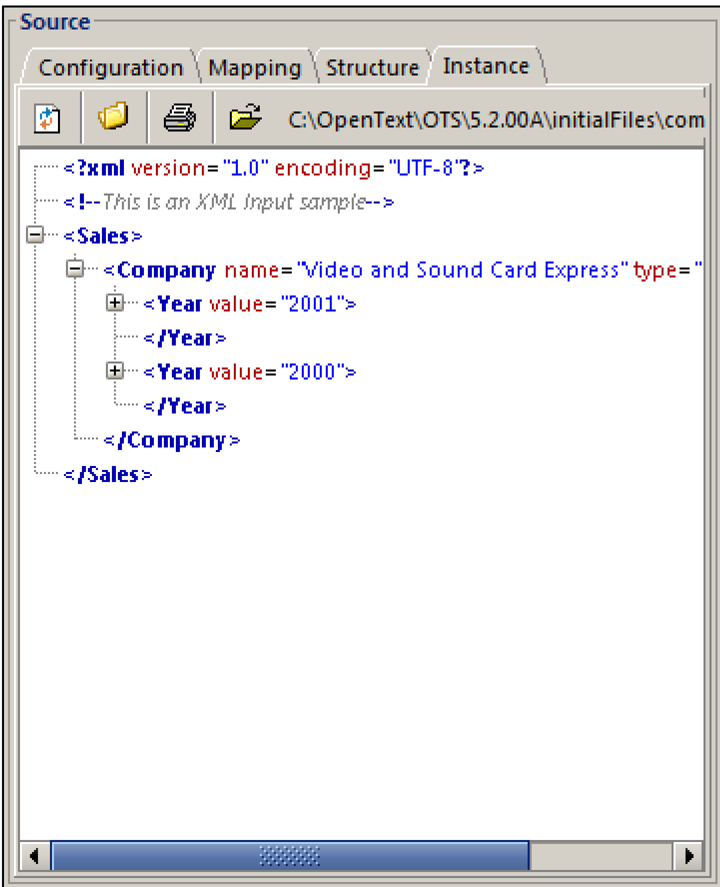


Figure 4-3: Source Instance tab

Below the row of tabs are buttons to enable you to do the following:

Icon	Menu Command	Description
	Refresh the source instance	Refreshes the source instance to show the most recent version.
	Open the file in a text editor	Opens up the source file in a separate editor within a new window for you to manually make changes to the code.

	Print the source instance	Prints out a copy of the source instance.
	Open instance...	Allows you to open another saved file.

4.3 Source Configuration

The **Source Configuration tab** is where you define the data format and structure of your input file and, optionally, the input data source (instance) to be used for the transformation you can run within Output Transformation Designer .

The **Source Configuration tab** has three important areas:

- "Input Data Format" on page 52. Choose your input type from the Data Format drop-down list at the top of the pane. Changing selections will change the fields within the **Source Configuration tab**.
- **Structure box**. This box displays the file you have loaded as your input structure file. The structure of this file is shown graphically in the "Source Mapping" on page 91 tab.
- **Instance box**. This box displays the path to the input data source you have chosen. This is the file that will be used in test transformations you run within the Data Transformation Engine window. Choose your desired location by clicking the **Ellipsis** button, , and browsing your file system.

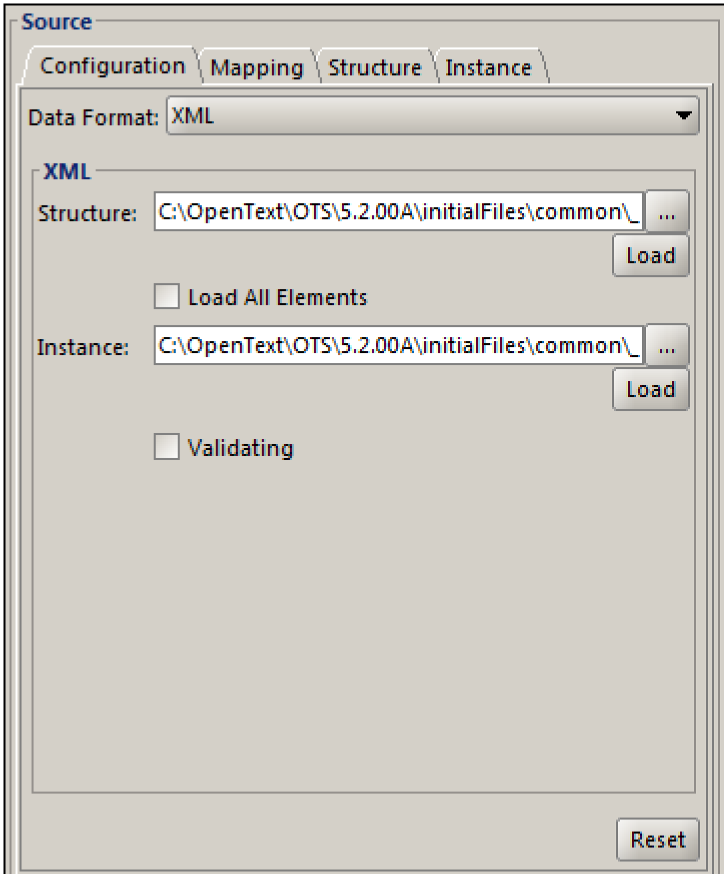


Figure 4-4: Source Configuration tab

4.4 Input Data Format

The **Data Format** box of the “Source Configuration” on page 51 tab is where you define the source data format. Choose your input type from the **Data Format** drop-down list at the top of the pane. Changing selections will change the fields available within “Source Configuration” on page 51.

The source structure is specified differently according to your input data format.

When you are finished setting the fields for Source Configuration, set the fields for “Target Configuration” on page 74 and click the **Done** button on it to save the changes. You could also click the **Reset** button of the Source Configuration tab to restore the previous entries.

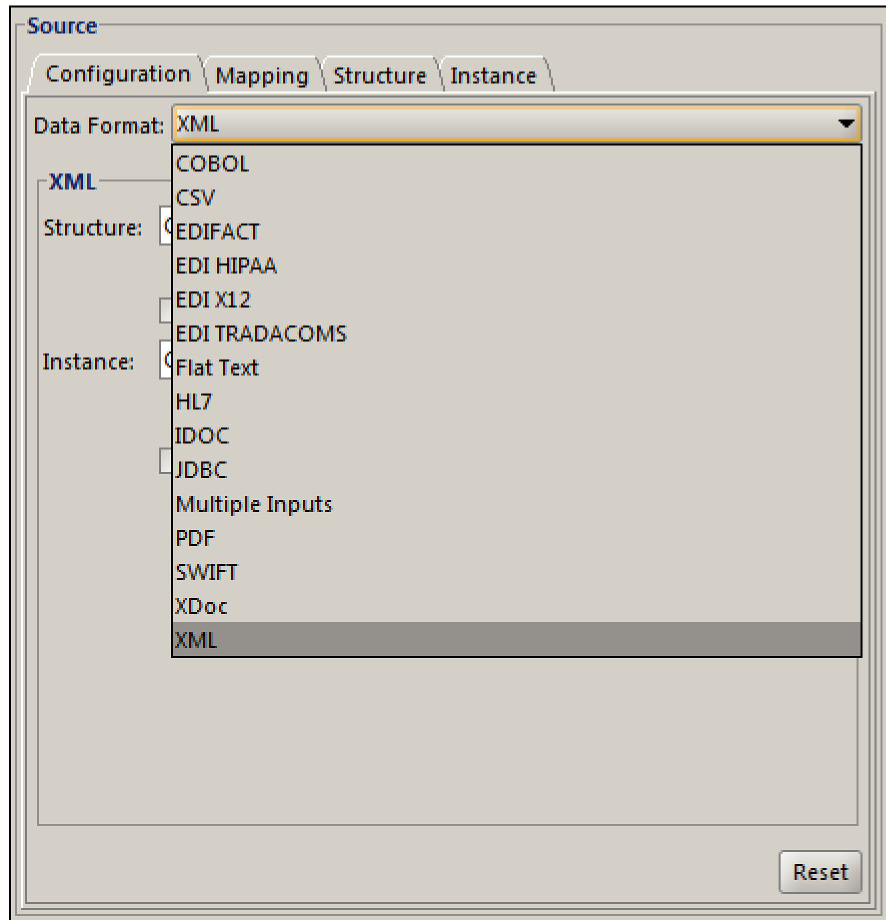


Figure 4-5: Source Configuration Data Formats

4.4.1 XDoc Input Data Format

An XDoc can be used as "unstructured" data. Users can stream in any file type, making it useful for embedding encoded or mixed data types into output data such as a database, SOAP message, or PDF image.

- **Instance.** The path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.
- **Encoding Mode.**
  - Choose **None** for text.
  - Choose **Encode Base64** for images and other encoded data.

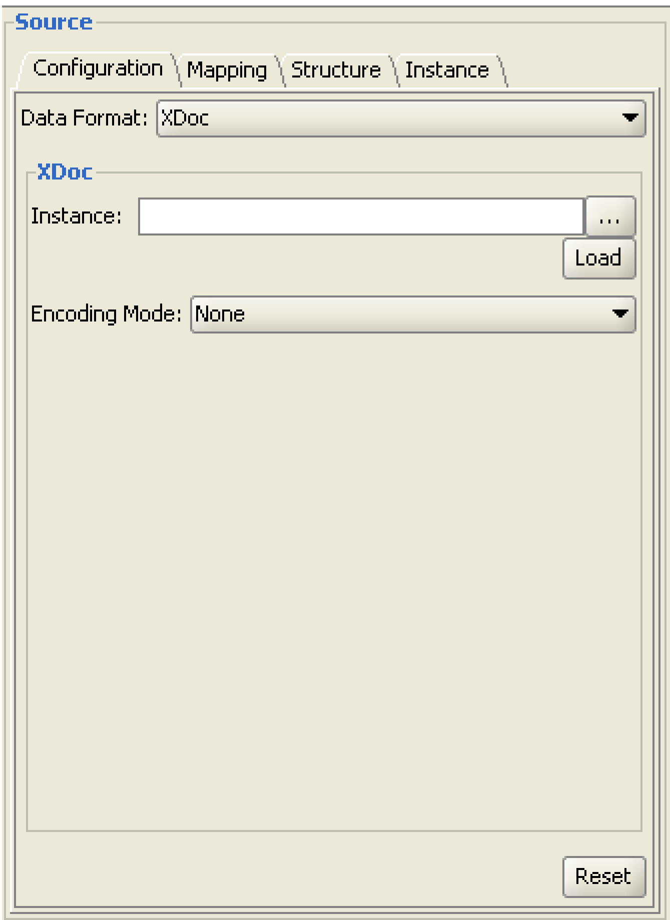


Figure 4-6: XDoc Input Data Format

4.4.2 XML Input Data Format

The **Structure** file is either the input data file, or a file which has an identical structure to your intended input data file(s). For XML this can also be a Document Type Definition (DTD) or schema file.

The **Instance** is the path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.

The **Validating** check box enables you to validate the XML source against the DTD or schema file. Select this check box to validate the XML source file.

Note: If an XML schema file (.xsd) is large, avoid selecting the **Load All Elements** check box, as doing so may consume all available memory.

4.4.2.1 Recursive Schemas

OpenText Output Transformation Server supports recursive schemas. You are prompted to enter the recursive level when you load the XML structure.

4.4.3 CSV Input Data Format

The input structure file is any existing CSV file that has the structure you want.

- For CSV, you must indicate the symbol acting as the **Delimiter** in your input file(s). By default this is a comma, ",".
- The **Structure** file is either the input data file itself, or a file which has an identical structure to your intended input data file(s).
- The **Instance** is the path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.
- The **Validating** check box enables you to validate the CSV structure based on the number of fields expected. Select this check box to validate the CSV source file.

4.4.4 PDF Input Data Format

OpenText Output Transformation Server will use the Adobe Portable Document Format (PDF) input structure to parse the data from the PDF input instance and send the data to any supported target data format. The PDF input **Structure** must have the same text field naming conventions as outlined in "PDF Output Data Format" on page 87 and "PDF Form Field Names" on page 88.

The **Instance** is the path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.

4.4.5 EDIFACT Input Data Format

EDI dictionary files are used to represent the **Structure** of the EDI transaction. They can be written in either our own XML format or in the industry standard, Standard Exchange Format (SEF). For more information on EDI dictionary files, see "Composing EDI dictionary files" on page 206. Please visit http://www.edidev.com/eval_SefFile.html for more information on the SEF format.

The **Header** is the path to the dictionary file that contains header information. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.

The **Instance** is the path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.

The **Validating** check box enables you to validate the EDIFACT structure against the dictionary files. Select this check box to validate the EDIFACT source file.

4.4.6 EDI X12 Input Data Format

EDI dictionary files are used to represent the **Structure** of the EDI transaction. They can be written in either our own XML format or in the industry standard, Standard Exchange Format (SEF). For more information on EDI dictionary files, see “Composing EDI dictionary files” on page 206. Please visit http://www.edidev.com/eval_SefFile.html for more information on the SEF format.

The **Header** is the path to the dictionary file that contains header information. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.

The **Instance** is the path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.

The **Validating** check box enables you to validate the EDI X12 source file against the dictionary files. Select this check box to validate the EDI X12 source file.

4.4.7 EDI HIPAA Input Data Format

EDI dictionary files are used to represent the **Structure** of the EDI transaction. They can be written in either our own XML format or in the industry standard, Standard Exchange Format (SEF). For more information on EDI dictionary files, see “Composing EDI dictionary files” on page 206. Please visit http://www.edidev.com/eval_SefFile.html for more information on the SEF format.

The **Header** is the path to the dictionary file that contains header information. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.

The **Instance** is the path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.

The **Validating** check box enables you to validate the EDI HIPAA source against the dictionary files. Select this check box to validate the EDI HIPAA source file.

4.4.8 SWIFT Input Data Format

SWIFT dictionary files, XML representations of each type of transaction, represent the **Structures** of these transactions. For more information on SWIFT dictionary files, see “Composing SWIFT dictionary files” on page 216

The **Instance** is the path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.

4.4.9 HL7 Input Data Format

HL7 dictionary files , XML representations of each type of transaction, represent the **Structures** of these transactions. For more information on HL7 dictionary files, see "Composing HL7 dictionary files" on page 229.

Select the **Use Batch** check box to enable the Batch setting. Enter the dictionary that contains header and trailer information on how messages in HL7 Structure are looped.

For **DataType,** enter the dictionary that contains the data type definitions used in HL7 Structure.

The **Instance** is the path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, [...], and browsing your file system.

4.4.10 IDOC Input Data Format

IDOC is a standard data **Structure** for EDI between application programs written for the popular SAP business system or between an SAP application and an external program. IDOCs serve as the vehicle for data transfer in SAPs ALE system.

The **Instance** is the path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, [...], and browsing your file system.

4.4.11 JDBC Input Data Format

The input structure for JDBC may be a database table or the result of a SQL statement run against the database. Select which JDBC connection you will be using to connect to your database in the JDBC Connection combo box. You should have already defined the connection in the JDBC Connection tab of the **Configuration** pane. The **Detail** button displays the settings of the currently selected JDBC connection, while the **New** button allows you to configure a new connection. For more information about the JDBC Connection tab, see "JDBC Connection tab" on page 28.

The **Instance** is the path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, [...], and browsing your file system.

Note: For JDBC, the database table or SQL result you specify for the input structure is assumed to be your input instance as well. All other formats allow you to specify an input data source different from the input structure. If you do not provide an input data source, the structure file will be used as the instance.

4.4.11.1 JDBC Join Feature

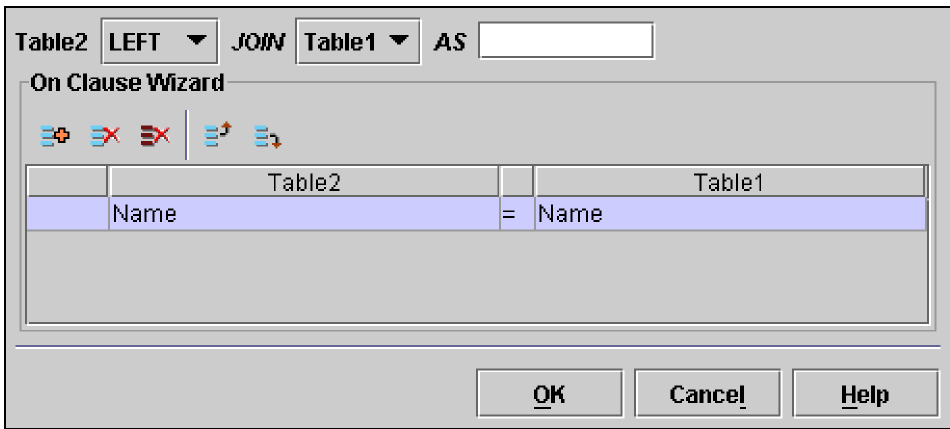
With the input **JDBC Join** feature, you can generate an XML-like structure by joining multiple tables.

To perform a JDBC input join:

1. Select the **JDBC data format** in the “Source Configuration” on page 51 tab.
2. Load the **root table**.
3. Select the “Source Mapping” on page 91 tab.
4. Right-click on the **root table name**, which is the first element.
5. Select **Properties** to enter an alias.

Or

1. Select **Join** to bring up the JDBC Join window.



2. Two types of joins are available: **LEFT** and **INNER**.

- **Join.** From the Join drop-down list, select the table to which you wish to join your JDBC data set.
- **AS.** This text field is where you specify an alias for the table. This is useful for doing SELF joins.
- **ON.** This text field is where you type the criteria for the JOIN. For example:

```
 SELECT Office.Name, Person.Name FROM (Person INNER JOIN Office ON
Person.PrimaryKey = Office.ForeignKey)
```

- **On Clause Wizard.** The wizard automatically selects the **JOIN** table to be the table that is referenced by the foreign key. It also selects the referenced column name automatically. If no reference is found, you can select the table and column name from the drop-down lists.

Note: For MS Access databases, the **On Clause Wizard** does **not** automatically select the referenced tables.

4.4.12 COBOL Input Data Format

The input structure for COBOL is determined by the copybook selected in the **Structure** box. The COBOL IO Directive determines how your data is interpreted by the COBOL parser.

The **Instance** is the path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, , and browsing your file system.

To use **COBOL85 Free Format**, select the check-box. When **not** using COBOL85 Free Format data, de-select the check-box and select **Fixed** or **Variable**.

Note: If a COBOL IO directive is not chosen, the default IO directive is a record-based IO directive that does not contain record descriptor words. For more on COBOL IO Directives, see “COBOL IO Directive tab” on page 37.

The **COBOL Copybook Parser** supports the following:

- COBOL85 format only.
- Multiple level 01 fields.
- Level 02-49 fields.
- Level 88 field: condition names,
- Usage clause: DISPLAY, COMP-3.
- All data types: 9 for numeric; X for alphanumeric; A for alphabetic.
- PIC clause.
- VALUE clause: for default value.
- VALUES.
- OCCURS clause.
- REDEFINES clause.
- FILLER field - full support (reference count). Option to show fillers through the Data Transformation Engine window.
- Variable occurs.

The COBOL Data Parser supports the following:

- Multiple level 01 field.
- Level 02-49 field.
- Level 88 field: condition names.
- Usage clause: DISPLAY, COMP-3.

- All data types: 9 for numeric; X for alphanumeric; A for alphabetic.
- PIC clause.
- VALUE clause: for default value.
- VALUES.
- OCCURS clause.
- REDEFINES clause.
- FILLER field - full support (reference count). Option to show fillers through the Data Transformation Engine window.
- Variable occurs.

4.4.13 COBOL nodes

The structure of COBOL files deserves special attention. This section discusses COBOL nodes and their structure.

4.4.13.1 Level 01 (L01) nodes

L01 nodes appear as blue colored nodes, . L01 nodes define the hierarchy of a record, indicating the byte offset of each COBOL field and substructure. For example:

```
01 HEADER.
05 TYPE PIC 99.
05 WHO.
10 FIRST_NAME PIC X(30).
10 LAST_NAME PIC X(30).
```

In the above example:

- **HEADER** is a COBOL L01 Structure.
- **WHO** is a substructure.
- **TYPE, FIRST_NAME** and **LAST_NAME** are fields.

4.4.13.2 L01 structure

A Loaded Level 1 node is a blue colored icon, , instead of green, . Properties are the same as user node defined properties with the exception that there are no Group Types, and there are **Condition** and **Occurs** fields.

You can access the COBOL Node Properties dialog by right-clicking on any L01 node and from the context menu that appears, choosing **Properties**.

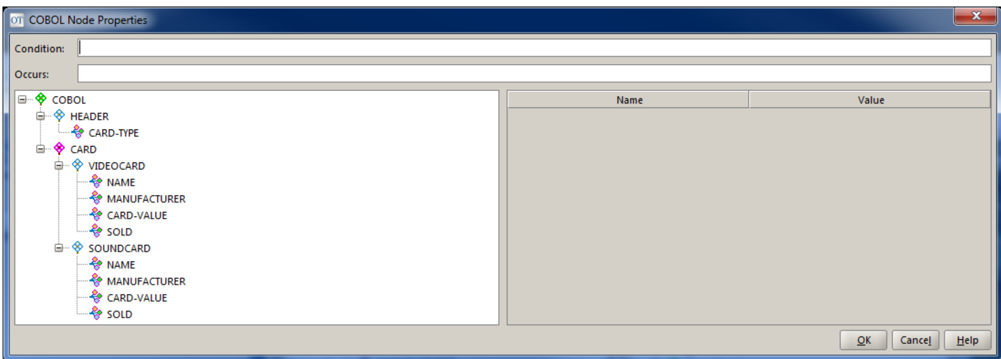


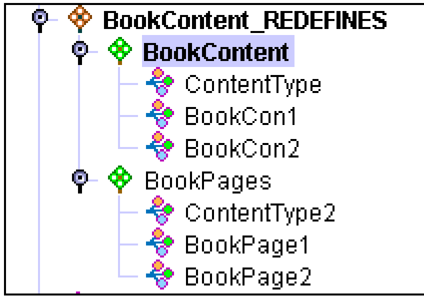
Figure 4-7: COBOL Node Properties window showing L01 node properties

Condition is a boolean expression. Drag a node into the **Condition** box and enter a Boolean expression operator and value after it. For example: `/COBOL/CARD/VIDEOCARD/CARD-VALUE=='1'`

The **Occurs** field is the node that contains the information on how many times it will loop, for example 01, 02, or 03.

4.4.13.3 Redefined nodes

Redefined nodes appear as brown colored nodes, . A redefined node uses the condition setting of the Node Properties window to redefine the structure of the data. For example, based on the Content Type element, a condition will determine if the data is BookContent or BookPages.



Condition for BookContent:

```
/COBOL/Book/BookContent_REDEFINES/BookContent/ContentType == '01'
```

Condition for BookPages:

```
/COBOL/Book/BookContent_REDEFINES/BookPages/ContentType2 == '02'
```

4.4.13.4 User defined nodes

User defined nodes appear as purple colored nodes, . These nodes are used to group the L01.

To add user defined nodes:

1. Right-click the **root node**.
2. Select **Add** child.

4.4.13.5 COBOL node properties

To view the properties of a user defined node, right-click the **User Defined Node** and select **Properties**. The **COBOL Node Properties** window opens.

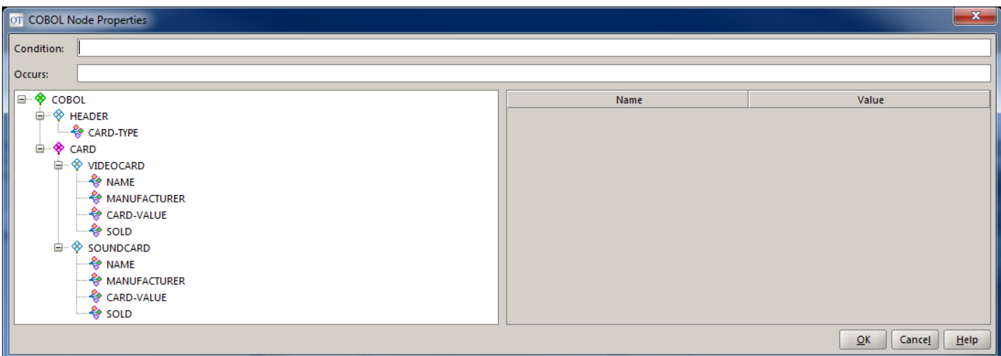


Figure 4-8: COBOL Node Properties Window Showing User Defined Node“r;Card”

4.4.13.5.1 COBOL Enumerated Values

Note: For more on **Enumerated Values**, see “Enumerated nodes” on page 105.

When COBOL contains enumerated values, the values are shown in the right-hand table of the **Node Properties window**.

The COBOL node Group Types are:

None	Default

Exclusive	If A and B are Level 01 exclusive, then A and B cannot be in the same record with the same header.
Sequence	Same as None.

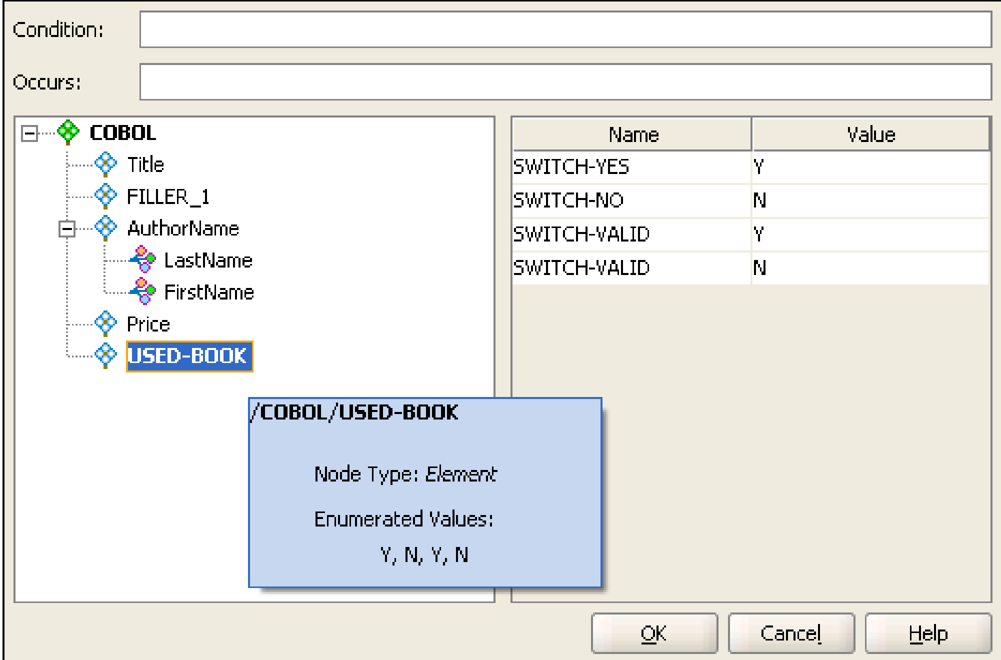


Figure 4-9: COBOL Node Properties window showing enumerated node

4.4.14 Flat Text Input Data Format

The input structure file is any existing Flat Text file that has the structure you want.

- The **Structure** file is either the input data file itself or a file that has an identical structure to your intended input data file(s).
- The **Instance** is the path to the input data source you have chosen. Choose your desired location by clicking the **Ellipsis** button, [...], and browsing your file system.
- The **Validating** check box enables you to validate the Flat Text structure against the XML dictionary file. Select this check box to validate the instance against the Flat Text dictionary.

4.4.14.1 Flat Text Importer Wizard

See "Flat Text Importer Wizard" on page 245.

4.4.15 JDBC Input SQL statement generation

The JDBC input format allows for better SQL statement generation. Variables can be used to determine the output.

Note: JDBC must be the source data format to select fields using SQL variables.

Connection parameters are defined in the "JDBC Connection tab" on page 28 of the "Settings window" on page 15.

To use the SQL Statement Generator:

1. Open the **Source window**.
2. Select the **Mapping tab**. For more information on the Source Mapping Pane, see "Source Mapping" on page 91.
3. Right-click on a value to query against and from the context menu that appears, select **Properties**.

   The **JDBC Source Node Properties** dialog appears.
4. In the **Condition** field, enter a SQL query (i.e. `"> 5"`, `"< 5"`, `"= someValue"`).
5. **Click OK** to save.

Note: For more information on SQL queries and statements, see "SQL statements" on page 64.

4.4.16 SQL statements

If your transformation is producing output to a database, your output structure may be a database table or the result of a SQL statement run against the database. Loading a database table as your output structure is described in "Loading structures from other sources" on page 67. OpenText Output Transformation Server accepts any proper SQL statement, but it is important to know the syntax used to refer to output item names, as described below.

Suppose you have a table called `people` in your database, into which you would like to insert some rows containing ID, name, and contact information for various people. A simple example of a SQL statement that would achieve this would be:

```
INSERT INTO names id,name,contactInfo VALUES (${id},${name},${contactInfo})
```

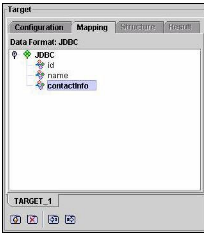


Figure 4-10: SQL statement sample

1. Enter the above **SQL statement** into the **SQL Statement** field of the "Target Configuration" on page 74 tab.
2. Ensure that the **Load Structure** check box is selected. The following output structure will be visible in the **Mapping** tab of the **Target** pane.
3. Set the values of your output items as usual. For more information, see "Output item values" on page 97.

4.4.16.1 SQL statement syntax

This statement takes the form:

```
<operation> <table name> (<column names>) VALUES (<output item names>)
```

The column names and output item names are comma-delimited.

Note: The column names and output item names do **not** have to be identical.

Suppose you enter the following SQL statement:

```
INSERT INTO names (id,name,contactInfo) VALUES (${myID},${myName},${myContactInfo})
```

This would still configure the transformation to enter values into the ID, name, and contactInfo columns. However, your output item values would be named myID, myName, and myContactInfo respectively.

You may write any proper SQL statement that uses this bracketing style and naming syntax.

4.4.17 Multiple Inputs Data Format

Choosing **Multiple Inputs** for your input type from the **Data Format** drop-down lists means you will have more than one input transformation. You can use more than one input source and/or type for your input.

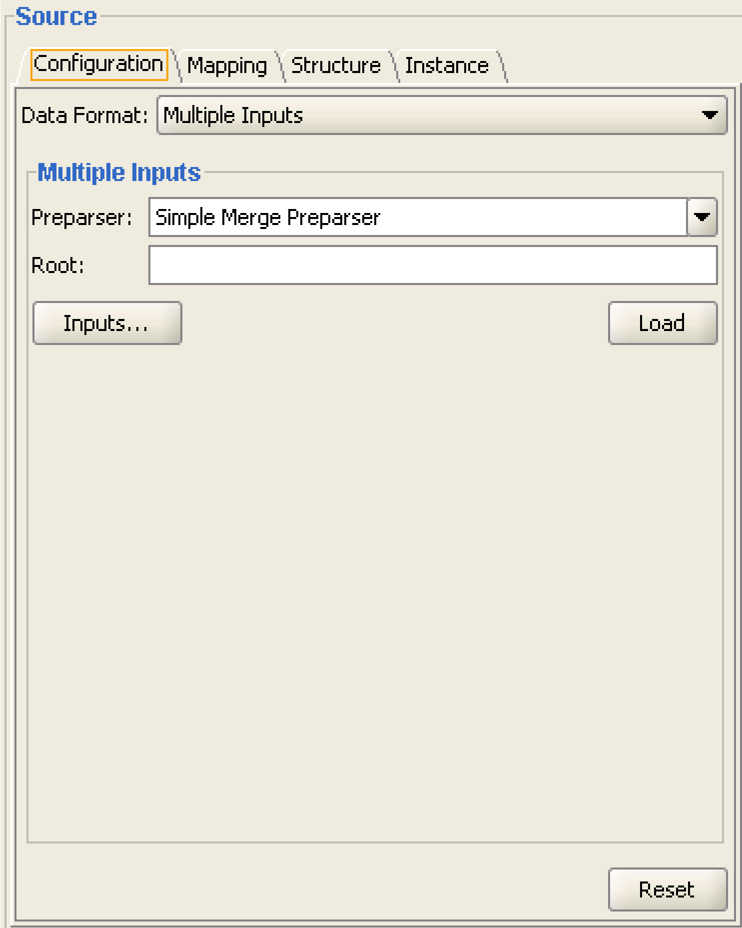


Figure 4-11: Multiple Inputs Data Format

To define multiple sources of input data:

1. Select **Multiple Inputs** from the **Data Format** drop-down list in the "Source Configuration" on page 51 tab.
2. Select the **Preparser** from the drop-down list. For more information see "Preparsers" on page 67.

   Note: If you have written a **custom preparser**, enter the name of the custom preparser.

3. Type a valid **XML name** for the name of the root node.
4. Click the **Inputs** button. The **Multiple Inputs Configuration** window appears.
5. Select the **Data Format** for each input as you would for a single input transformation.
6. Add additional inputs:
   a. Click the **Insert a tab** button, , to add an input tab.
   b. Select the **Data Format** for the input as you would for a single input transformation.

4.4.17.1 Load button

The Multiple Inputs **Load** button allows the structure of multiple inputs to be loaded without having to load each structure one by one.

This is useful when the structure is modified outside of Data Transformation Engine , in an application such as a text editor. Clicking **Load** will reload the structure into the project.

4.4.18 Loading structures from other sources

Structures can also be loaded from any of the following sources: URN, HTTP, `.jar` file, and FTP. To use these sources, enter the resource path, as in the examples below:

```
HTTP://domainname.com/resourcelocation/resource.jar
```

```
FTP://domainname.com/resourcelocation/resource.jar
```

4.5 Preparsers

There are two pre-defined preparsers: **Default** and **Simple Merge**.

Default preparser

The **Default** preparser produces output in sequence. In the output file, the data from each input source is concatenated without the use of logic, as is the case with the Simple Merge preparser.

Simple Merge preparser

The **Simple Merge preparser** will combine the input data based on a custom merge key. All data matching the custom key will be placed together as per the structure of the output file.

Note: Simple Merge multiple inputs **only** support JDBC, CSV, Flat Text, and COBOL. The `hasnext()` type methods are not available when writing custom preparsers for EDI/HL7 and SWIFT data formats under the Reading Data section. For more information, see “Writing custom preparsers” on page 185.

See the OpenText Output Transformation Server embed sample directory for an example of a preparser:

```
<install_home>\initialFiles\common\_sample\DataTransformation\embed_sample
```

Chapter 5

Target pane

The Data Transformation Engine Target pane is divided into the following tabs:

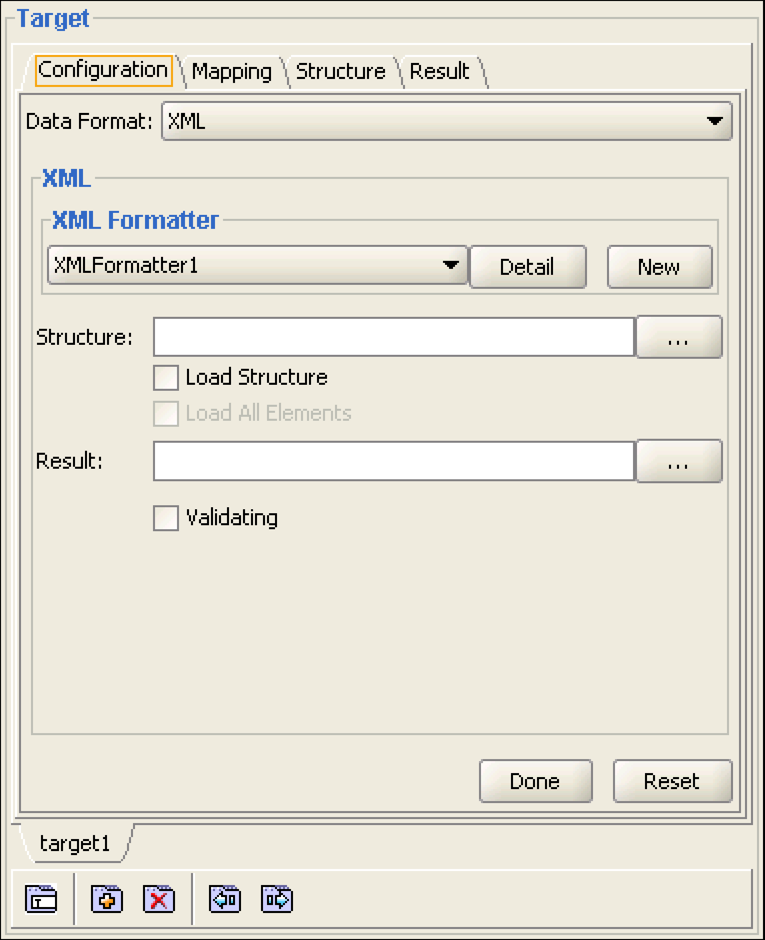


Figure 5-1: Target pane

- Target Configuration. The **Configuration** tab is where you define the data format and, optionally, the structure of your output.
- Target Mapping. The **Mapping** tab graphically displays your target structure.
- Target Structure. The **Structure** tab displays the file you have loaded as your target structure file.

- Target Result. The **Result** tab enables you, at any time, to see the output produced from your current configurations run on the input data source (instance) specified.

Note: The Structure and Result tabs are disabled with PDF output. Instead, a PDF reader, such as Acrobat Reader, is opened (if installed) to automatically view the transformed output.

5.1 Target Structure

The **Structure** tab is only used if you have loaded an output structure from file in the “Target Configuration” on page 74 tab. If so, the output structure file is displayed here.

Notes

- This tab is disabled when PDF output is selected. Instead, a PDF reader, such as Acrobat Reader, is opened (if installed) to automatically view the transformed output.
- This tab is disabled if you have JDBC as your output. When you select the **Load Structure** check box, the structure is loaded on the “Target Mapping” on page 92 tab instead.

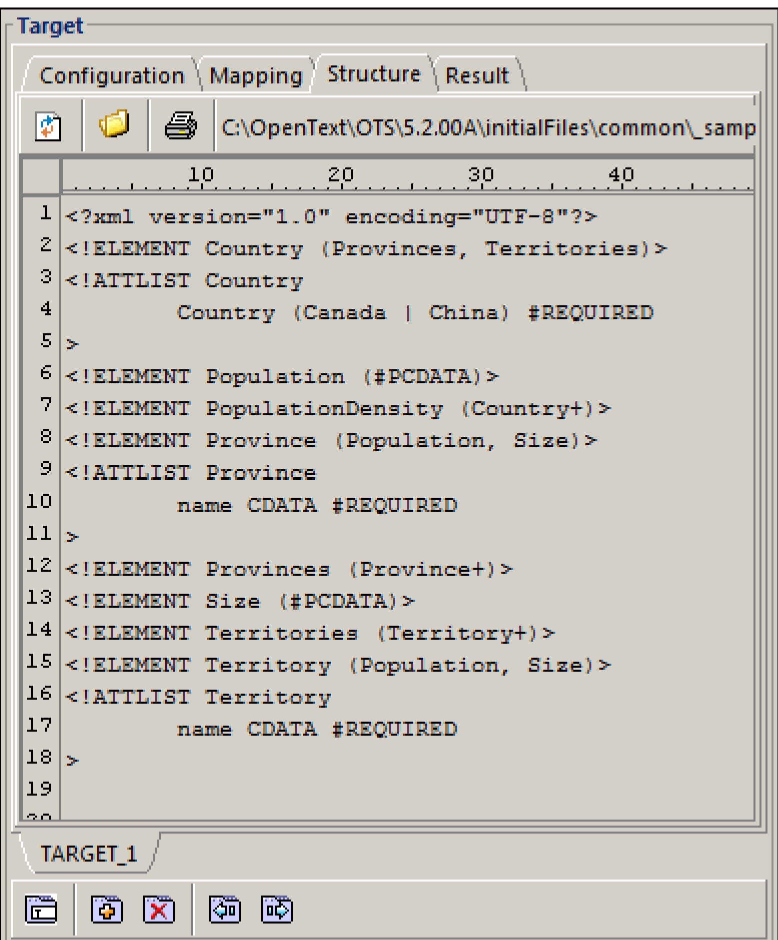


Figure 5-2: Target Structure tab

Near the top of the screen, you will find the following buttons:

Icon	Menu Command	Description
	Refresh the transform result	Refreshes the transform result to show the most recent version.
	Open the file in a text editor	Opens up the source file in a separate editor within a new window for you to manually make changes to the code.
	Print the transform result	Prints out a copy of the transform result.

At the bottom of the Target Structure pane are quick access buttons to:

Icon	Menu Command	Description
	Rename the currently selected tab	Renames the presently selected transform tab.
	Insert a tab	Inserts a new tab to allow for another transformation target.
	Remove the selected tab	Deletes the current transform tab.
	Move the tab to the left	Moves the presently selected tab one slot to the left.
	Move the tab to the right	Moves the presently selected tab one slot to the right.

5.2 Target Result

The **Result tab** allows you, at any time during your work, to see the output produced from the current configurations run on the input data source (instance) specified. If no input instance has been declared, the input data structure is used during transformation.

This tab will not refresh your transformation result every time you make a change. In order to see the output produced from your current settings, you may have to click the **Refresh** button, , in the upper left corner of this pane.

Note: This tab is disabled when PDF output is selected. Instead, a PDF reader, such as Acrobat Reader, is opened (if installed) to automatically view the transformed output.

This tab is also disabled if you have JDBC as your output. To do a transformation, click **Transform** on the toolbar.

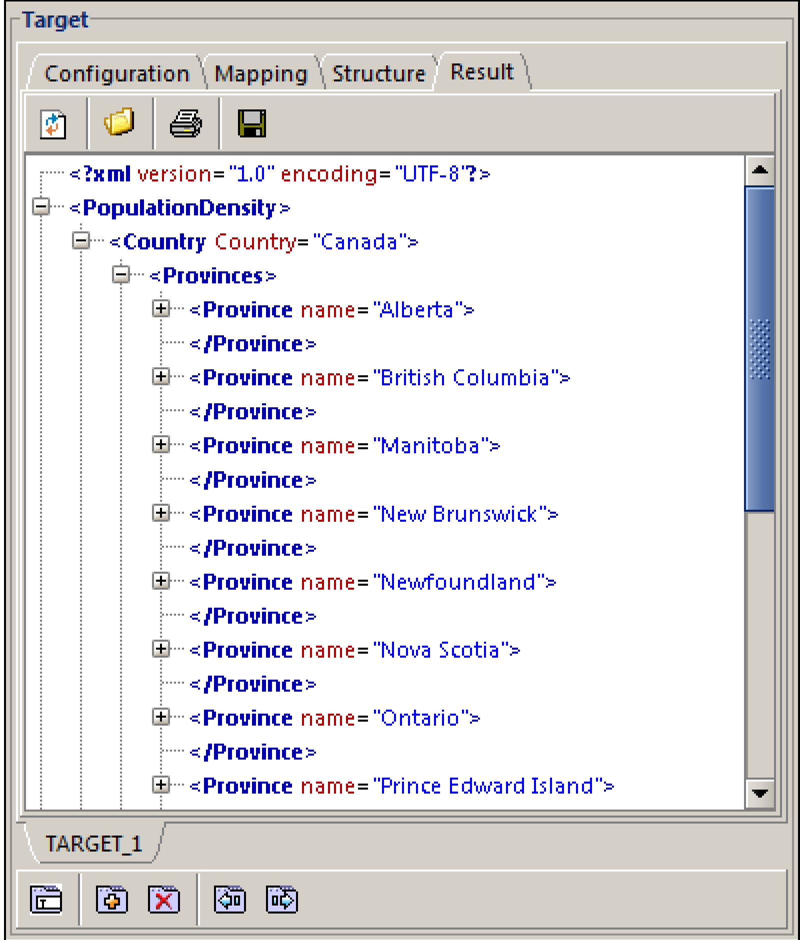


Figure 5-3: Target Results tab

Beneath the row of tabs at the top are buttons to perform the following functions:

Icon	Menu Command	Description
	Refresh the transform result	Refreshes the screen with the most recent code.
	Open the file in a text editor	Opens up the source file in a separate editor within a new window for you to manually make changes to the code.
	Print the transform result	Prints out a copy of the transformed output.
	Save target...	Saves the results of the transform.

At the bottom of the Target Structure pane are quick access buttons to:

Icon	Menu Command	Description
	Rename the currently selected tab	Renames the presently selected transform tab.
	Insert a tab	Inserts a new tab to allow for another transformation target.
	Remove the selected tab	Deletes the current transform tab.
	Move the tab to the left	Moves the presently selected tab one slot to the left.
	Move the tab to the right	Moves the presently selected tab one slot to the right.

5.3 Target Configuration

The **Target Configuration tab** is where you define the target data format and, optionally, the structure of your output.

Choose your output type from the **Data Format** drop-down list at the top of the tab. Changing the data format type will change the available fields within the **Target Configuration tab**.

The **Target Configuration tab** will change depending on which “Target Data Format” on page 75 you select. Some common elements to all data formats are as follows:

- **Data Format.** This drop-down list is where you choose the target data format.
- **Structure box.** This box displays the file you have loaded as your target structure file.

  Your target structure may be created in the “Target Mapping” on page 92 tab by manually adding items and using the drag-and-drop feature or the structure may be loaded from a file or database.
- **Result box**. This box displays the output file you have chosen.

  The Result field is provided for all data formats except JDBC. It allows you to specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.

5.4 Target Data Format

The **Data Format** box of the “Target Configuration” on page 74 tab is where you define the data format of your output. Choose your output type from the **Data Format** drop-down list at the top of the tab. Changing selections will change the fields available within the“Target Configuration” on page 74 tab.

When you are finished setting the fields of the “Target Configuration” on page 74 tab, click **Done** to save the changes or **Reset** to restore the previous entries.

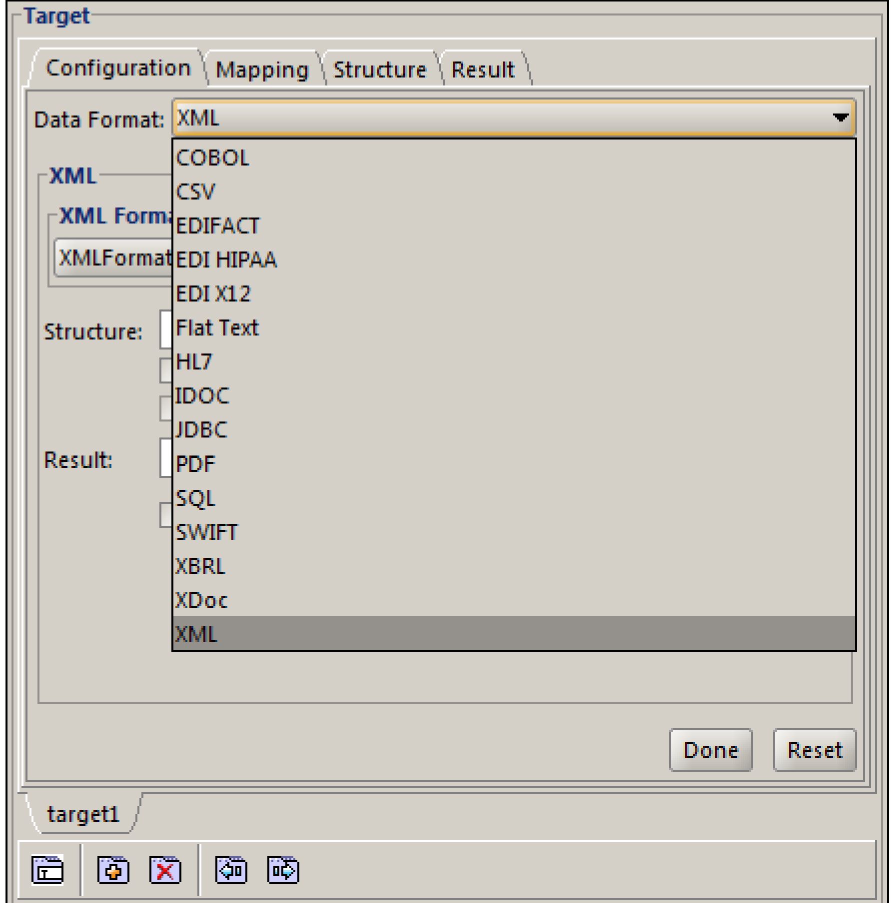


Figure 5-4: Target Configuration Data Format dropdown list

5.4.1 Multiple Output Transformations

You may perform multiple output transformations for any target data format with Data Transformation Engine . To add additional output targets to your project, click on the **Insert a tab** button,

, at the bottom of the pane. Then, you can switch between targets by selecting the different tabs.

5.4.2 XDoc Output Data Format

An XDoc can be used as “unstructured” data. Users can stream out any file type, making it useful for pulling messages from input data, such as from a database or a SOAP message.

- **Result**. Specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button, , and browsing your file system.
- **Encoding Mode.** Choose:
  - **None** for text.
  - **Decode Base64** for images and other encoded data.

5.4.3 XML Output Data Format

The output structure file is any existing XML file that has the structure you want. For XML this can also be a DTD or schema file.

Your target structure can be created in the **Mapping tab** by manually adding items and using the drag-and-drop feature, or it may be loaded from a file or database.

- If you choose to load your target structure, be sure that it complies with any rules and/or syntax of the data type chosen for your particular output data format.
- If you have specified a target structure, check the **Load Structure** box to load it; it will be displayed on the **Mapping tab**. You can load structure files from any URN, HTTP, `.jar` file, or FTP.
- The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button, , and browsing your file system.
- The **Validating** check box enables you to validate XML based on the results expected against the DTD or schema file. Select this check box to validate the XML source file.

Automatically mapping input PDF forms to XML output

When mapping PDF forms to XML output, users are required to load the PDF form before dragging the root node from the Source pane over to the Target pane. While

the method is functional for PDF forms with static mappings, this method is not practical when there are multiple types of forms since separate projects need to be created for each PDF form, which is a time-consuming task. If you have multiple types of PDF forms, you can automatically generate the required mappings thereby allowing one project to have the ability to process all the different form types. Automatic mapping of PDF forms can also be beneficial for users who do not have all the PDF forms ready or are working with forms that can be modified. When **Automatic Mapping** is enabled, the input PDF document is mapped to the XML structure and then the transformation is executed.

To create a PDF to XML project with automatic mapping:

1. In Output Transformation Designer, create a new Data Transformation Engine project.
2. On the Source pane's Configuration tab, set the **Data Format** as **PDF**.
3. **Load** the source PDF structure.
4. On the Target pane's Configuration tab, set the **Data Format** as **XML**.
5. Also in the Target pane, select the **Automatic Mapping** check box.
6. **Save** your project.
7. **Execute** the project and on the **Execution Options** screen, switch to the Sources tab and in the **Value** field, provide a different PDF file name.

   When the job has finished running, an XML file is created for each PDF form file.

5.4.4 CSV Output Data Format

The output structure file is any existing CSV file that has the structure you want.

- You must indicate the symbol you would like to be the **Delimiter** in your output file(s). By default this is a comma.

Your target structure can be created in the **Mapping** tab by manually adding items and using the drag-and-drop feature, or it may be loaded from a file or database.

- If you choose to load your target structure, be sure that it complies with any rules and/or syntax of the data type chosen for your particular output data format.
- If you have specified a target structure, check the **Load Structure** box to load it; it will be displayed on the **Mapping** tab. You can load structure files from any URN, HTTP, `.jar` file, or FTP.
- The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button,

  , and browsing your file system.

- The **Validating** check box enables you to validate the output CSV based on the results expected. Select this check box to validate the CSV file.

5.4.5 XBRL Output Data Format

XBRL is a powerful and flexible version of XML that has been defined specifically to meet the requirements of business and financial information.

For **Taxonomy**, enter the top-most level XSD, such as the XSD that imports XBRL label, presentation, calculation and reference schemas.

If you have specified a target structure, check the **Load Structure** box to load it; it will be displayed on the **Mapping** tab. You can load structure files from any URN, HTTP, `.jar` file, or FTP.

The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button,

, and browsing your file system.

XBRL properties

To configure XBRL properties:

1. Right-click a node in the “Target Mapping” on page 92 tab.
2. Select **XBRL Properties**.

   If you define the XBRL Properties on the target root node, the entire tree will inherit those properties.
3. Select the XBRL Context and XBRL Unit. Click:

   - **Detail** to see the Configuration tab associated with the attribute.
   - **New** to create a new attribute tab entry.

   For more information see “XBRL Context tab” on page 40 and “XBRL Unit tab” on page 45.
4. Enter an integer number or **INF** for **Decimal**.

   Note: The meaning of `decimals="INF"` is that the lexical representation of the number is the exact value of the fact being represented.
5. Enter a float number for **Scale**.

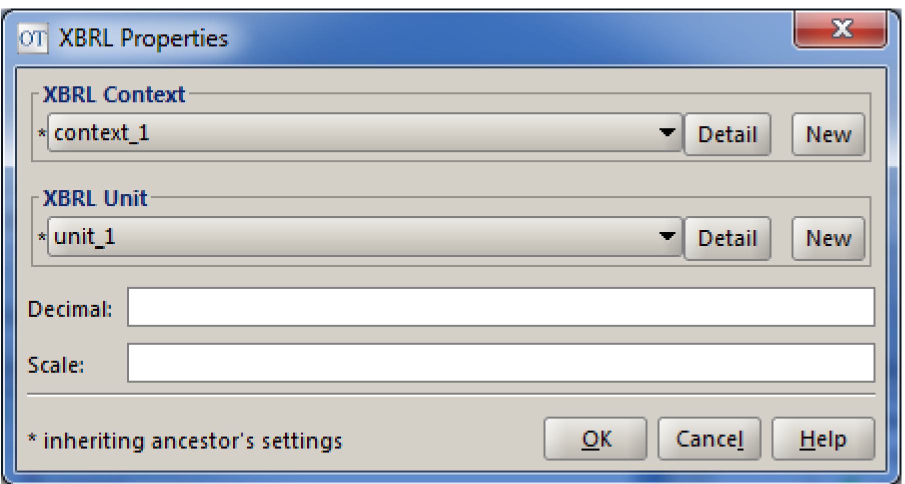


Figure 5-5: XBRLProperties dialog

5.4.6 HL7 Output Data Format

- **Batch.** Select the **Use Batch** check box to enable the Batch setting. Enter the dictionary that contains header/trailer information on how messages in HL7 Structure are looped.
- **Data Type.** Enter the dictionary that contains the data type definitions used in HL7 Structure.
- **Structure.** The output structure is a dictionary file, which is an XML representation of the type of transaction you are producing.

Your target structure can be created in the **Mapping tab** by manually adding items and using the drag-and-drop feature, or it may be loaded from a file or database. If you choose to load your target structure, be sure that it complies with any rules and/or syntax of the data type chosen for your particular output data format.

If you have specified a target structure, check the **Load Structure** box to load it; it will be displayed on the **Mapping tab**. You can load structure files from any URN, HTTP, `.jar` file, or FTP.

- The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button, ..., and browsing your file system.
- The **Validating** check box enables you to validate the output HL7 based on the results expected. Select this check box to validate the HL7 file.

5.4.7 SWIFT Output Data Format

The output structure is a dictionary file, which is an XML representation of the type of transaction you are producing.

Your target structure can be created in the **Mapping tab** by manually adding items and using the drag-and-drop feature, or it may be loaded from a file or database. If you choose to load your target structure, be sure that it complies with any rules and/or syntax of the data type chosen for your particular output data format.

If you have specified a target structure, check the **Load Structure** box to load it; it will be displayed on the **Mapping tab**. You can load structure files from any URN, HTTP, `.jar` file, or FTP.

The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button, [...]

, and browsing your file system.

5.4.8 Flat Text Output Data Format

The output structure is a dictionary file, which is an XML representation of the type of transaction you are producing.

Your target structure can be created in the **Mapping tab** by manually adding items and using the drag-and-drop feature, or it may be loaded from a file or database. If you choose to load your target structure, be sure that it complies with any rules and/or syntax of the data type chosen for your particular output data format.

If you have specified a target structure, check the **Load Structure** box to load it; it will be displayed on the **Mapping tab**. You can load structure files from any URN, HTTP, `.jar` file, or FTP.

The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button, [...]

, and browsing your file system.

The **Validating** check box enables you to validate the output Flat Text structure based on the results expected against the XML dictionary file. Select this check box to validate the result against the Flat Text dictionary file.

5.4.9 EDI X12 Output Data Format

The output structure is a dictionary file, which is an XML representation of the type of transaction you are producing.

Your output structure may be created in the **Mapping tab** by manually adding items and using the drag-and-drop feature as described in "Mappings" on page 91, or it may be loaded from a file or database. If you choose to load your output structure, be sure that it complies with the following for your particular output data format.

If you have specified an output structure, check the **Load Structure** box to load it into the **Mapping tab**. You can load structure files from any URN, HTTP, `.jar` file, and FTP.

The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button,

, and browsing your file system.

The **Validating** check box enables you to validate the output EDI X12 against the dictionary files. Select this check box to validate the EDI X12 file.

Note: IDOC (Intermediate Document) is a standard data structure for EDI (Electronic Data Interchange) between application programs written for the popular SAP business system or between an SAP application and an external program. IDOCs serve as the vehicle for data transfer in SAP's ALE (Application Link Enabling) system.

Loop Path. This text field is used when you want to pipeline (conserve memory by writing data immediately to disk) your EDI output. The **Loop Path** is the path in your target EDI structure that will be looping (pipelining).

To enable pipelining, you **must** also select **Match** on the **Target Node Properties** "Advanced tab" on page 111for the EDI root node. For more information, see Match on page 112.

Loop Path example: `/EDI/Group_Loop/APERAK`

5.4.10 EDI HIPAA Output Data Format

The output structure is a dictionary file, which is an XML representation of the type of transaction you are producing.

Your output structure may be created in the **Mapping tab** by manually adding items and using the drag-and-drop feature as described in "Mappings" on page 91, or it may be loaded from a file or database. If you choose to load your output structure, be sure that it complies with the following for your particular output data format.

If you have specified an output structure, check the **Load Structure** box to load it into the **Mapping tab**. You can load structure files from any URN, HTTP, `.jar` file, and FTP.

The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button,

, and browsing your file system.

The **Validating** check box enables you to validate the output EDI HIPAA result against the dictionary files. Select this check box to validate the resulting EDI HIPAA file.

Note: IDOC (Intermediate Document) is a standard data structure for EDI (Electronic Data Interchange) between application programs written for the popular SAP business system or between an SAP application and an external program. IDOCs serve as the vehicle for data transfer in SAP's ALE (Application Link Enabling) system.

Loop Path. This text field is used when you want to pipeline (conserve memory by writing data immediately to disk) your EDI output. The **Loop Path** is the path in your target EDI structure that will be looping (pipelining).

To enable pipelining, you **must** also select **Match** on the **Target Node Properties** “Advanced tab” on page 111 for the EDI root node. For more information, see Match on page 112..

Loop Path example: `/EDI/Group_Loop/APERAK`

5.4.11 EDIFACT Output Data Format

The output structure is a dictionary file, which is an XML representation of the type of transaction you are producing.

Your output structure may be created in the **Mapping tab** by manually adding items and using the drag-and-drop feature as described in “Mappings“ on page 91, or it may be loaded from a file or database. If you choose to load your output structure, be sure that it complies with the following for your particular output data format.

If you have specified an output structure, check the **Load Structure** box to load it into the **Mapping tab**. You can load structure files from any URN, HTTP, `.jar` file, and FTP.

The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button,

, and browsing your file system.

The **Validating** check box enables you to validate the output EDIFACT result against the dictionary files. Select this check box to validate the resulting EDIFACT file.

Note: IDOC (Intermediate Document) is a standard data structure for EDI (Electronic Data Interchange) between application programs written for the

popular SAP business system or between an SAP application and an external program. IDOCs serve as the vehicle for data transfer in SAP's ALE (Application Link Enabling) system.

Loop Path. This text field is used when you want to pipeline (conserve memory by writing data immediately to disk) your EDI output. The **Loop Path** is the path in your target EDI structure that will be looping (pipelining).

To enable pipelining, you **must** also select **Match** on the **Target Node Properties** "Advanced tab" on page 111 for the EDI root node. For more information, see Match on page 112.

Loop Path example: `/EDI/Group_Loop/APERAK`

5.4.12 JDBC Output Data Format

The output structure may be a database table or the result of a SQL statement run against the database.

To load your structure using a JDBC connection:

1. Choose the connection you will be using to connect to your database in the **JDBC Connection** drop-down list.

   You should have already defined the connection parameters in the "JDBC Connection tab" on page 28 of the "Settings window" on page 15. The **Detail** button displays the "Settings window" on page 15, while the **New** button creates a new connection tab on the "JDBC Connection tab" on page 28 and in the JDBC Connection Dropdown. The new connection will need to be configured.

2. Select **Table/View** as your output option.

3. Loading the output structure is essentially the same as writing an `Insert` SQL statement; however, Data Transformation Engine automatically generates the statement for you. For more information, see "SQL statements" on page 64.

   You may change this to be an `Update` statement. On the "Target Mapping" on page 92 tab, right-click the root of your loaded output structure and select **Properties** from the menu that appears. In this **Properties Window** you may also add **Where** keys.

   Data Transformation Engine allows for any proper SQL statement to be written in the **SQL Query** field, rather than just simple `Insert` and `Update` statements.

4. **Load Structure.** Select the **Load Structure** box to load it into the "Target Mapping" on page 92 tab. You can load structure files from any URN, HTTP, `.jar` file, and FTP.

   The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button,

   , and browsing your file system.

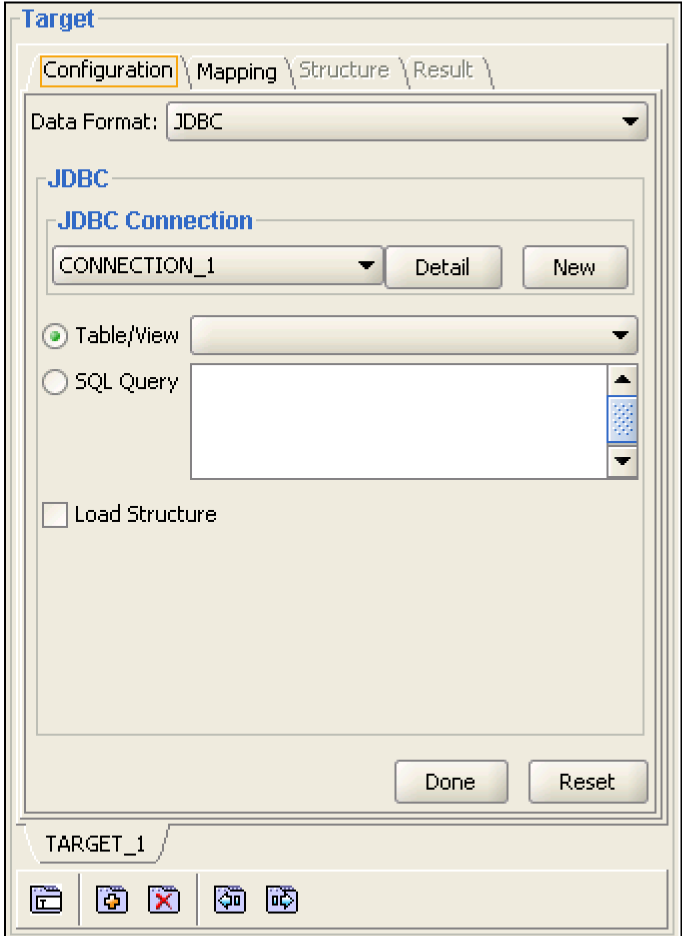


Figure 5-6: JDBC Output Data Format

5.4.13 JDBC Output Exception Control

The JDBC Output Data Format allows the user to create an exception control for JDBC transactions.

Note: JDBC must be the target data format to create an exception control.

To use the JDBC exception control:

1. Select the JDBC data format in the "Target Configuration" on page 74 tab.
2. Click the JDBC Connection **Detail** button.

   Connection parameters are defined in the "JDBC Connection tab" on page 28 of the "Settings window" on page 15. The **Detail** button displays the "Settings window" on page 15, while the **New** button creates a new connection tab on the "JDBC Connection tab" on page 28 and in the **JDBC Connection** drop-down

list. The new connection will need to be configured. For details, see “JDBC Connection tab” on page 28 and “JDBC Output Data Format” on page 83.

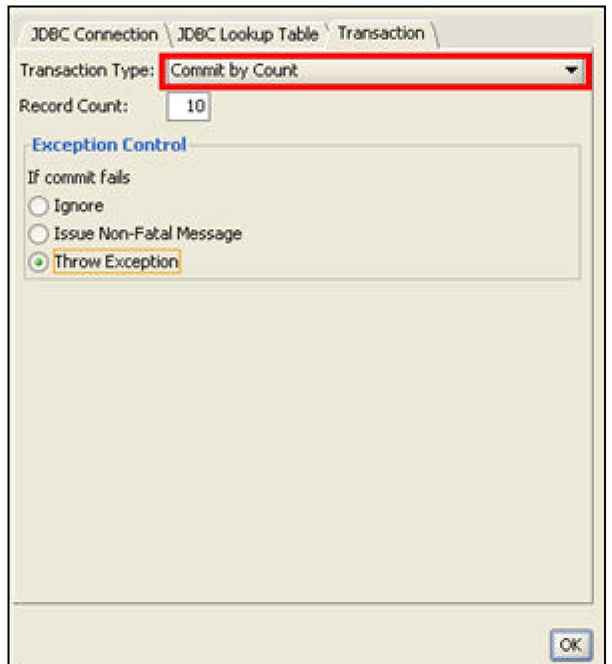


Figure 5-7: Transaction tab

3. From the “Settings window“ on page 15, open the “Transaction tab” on page 39.

4. Select a **Transaction Type**.

5. For exception control, If commit fails, click on one of three radio buttons:

   - **Ignore.** Exceptions will be ignored.
   - **Issue Non-Fatal Message.** Continue the transaction and send a message.
   - **Throw Exception.** This will stop the transaction at the point of the exception and roll back the database to the last commit based on “Record Count”.

6. Click **OK** to save.

5.4.14 SQL Output Data Format

The output structure can be loaded via a JDBC connection or from a SQL statement you provide in the **SQL Statement** field.

To load your structure using a JDBC connection:

1. Choose the connection you will be using to connect to your database in the JDBC Connection drop-down list.

   You should have already defined the connection parameters in the “JDBC Connection tab” on page 28 of the “Settings window“ on page 15. The **Detail** button displays the “Settings window“ on page 15, while the **New** button creates a new connection tab on the “JDBC Connection tab” on page 28 and in the **JDBC Connection** drop-down list. The new connection will need to be configured.

2. Select **Table/View** as your output option.

3. Loading the output structure is essentially the same as writing an `Insert` SQL statement; however, Data Transformation Engine automatically generates the statement for you. For more information, see “SQL statements” on page 64.

   You may change this to be an `Update` statement. On the “Target Mapping” on page 92 tab, right-click the root of your loaded output structure and select **Properties** from the menu that appears. In this **Properties Window** you may also add **Where** keys.

   Data Transformation Engine allows for any proper SQL statement to be written in the **SQL Query** field, rather than just simple `Insert` and `Update` statements.

4. **Load Structure.** Select the **Load Structure** box to load it into the “Target Mapping” on page 92. You can load structure files from any URN, HTTP, `.jar` file, and FTP.

5. The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button,

   , and browsing your file system.

5.4.15 COBOL Output Data Format

Select the COBOL Formatter and COBOL IO Directive from the drop-down lists.

You should have already configured these in the "COBOL Formatter tab" on page 36 and "COBOL IO Directive tab" on page 37 of the "Settings window" on page 15.

- The **Detail** button displays the "Settings window" on page 15.
- The **New** button creates either a new COBOL Formatter or IO Directive on the respective tab drop-down. New connections need to be configured from the "COBOL Formatter tab" on page 36 and "COBOL IO Directive tab" on page 37 of the "Settings window" on page 15.

Your target structure can be created in the **Mapping tab** by manually adding items and using the drag-and-drop feature, or it may be loaded from a file or database.

- If you choose to load your target structure, be sure that it complies with any rules and/or syntax of the data type chosen for your particular output data format.
- If you have specified a target structure, check the **Load Structure** box to load it; it will be displayed on the **Mapping tab**. You can load structure files from any URN, HTTP, `.jar` file, or FTP.
- The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button, , and browsing your file system.
- **Encoding** is either EBCDIC or ASCII.
- **COBOL85 Free Format** is either fixed or variable.

5.4.16 PDF Output Data Format

Your target structure can be created in the **Mapping tab** by manually adding items and using the drag-and-drop feature, or it may be loaded from a file or database. If you choose to load your target structure, be sure that it complies with any rules and/or syntax of the data type chosen for your particular output data format.

If you have specified a target structure, check the **Load Structure** box to load it; it will be displayed on the **Mapping tab**. You can load structure files from any URN, HTTP, `.jar` file, or FTP.

In the **Target** pane, along with setting the destination directory for your **Result**, you can also select from the following options:

- **Flattened** to produce flattened PDF files.
- **Repeat Fields in Overflow Pages** to print document information on each page of the PDF. If not selected, the information will print on the first page only.

- **Single output** to concatenate all produced PDF documents into a single file.

5.4.16.1 PDF Form Field Names

To create Portable Document Format (PDF) output, you must first create a PDF form with field names. To use the table overflow feature, name the fields with suffixes consisting of an underscore and incremental numbers, *fieldname_1*, *fieldname_2*, *fieldname_3*, etc. This will place overflowing data into duplicate forms.

Note: PDF files that contain no form fields have no structure. This applies to both PDF input and PDF output. To create PDF form fields, you need Adobe Acrobat Professional.

OUTPUT_ID

The OUTPUT_ID attribute is used when you wish to output to multiple PDF documents. The value of the OUTPUT_ID will be the PDF filename. Use the OUTPUT_ID and the “CONCAT” on page 132 function to concatenate the value OUTPUT_ID and another value.

Static Fields

Static fields will repeat on every page of the output. To make a node static, select the **Static Fields** check box on the **Target Node Properties** dialog for the individual node (for elemental nodes only).

Embedding Images

In order to insert an image into a PDF form field, you must change the properties of the form field to make it a button. To map an image to this field, the image must be in Base64 format and will usually be pulled from a database, a SOAP message or using the “XDoc Input Data Format” on page 53.

5.4.16.2 Appending Pages to PDFs

To append pages of data to a PDF file:

1. Click **Append Pages** on the “Target Configuration” on page 74 tab to open the **Append Pages** dialog.
2. From the **Data Format** drop-down menu, select the format. Only **Flat Text**, **PDF**, and **XML** are supported formats.

   Note: If you selected XML as the format, you must also choose a **XML Formatter** to use. For more information, see “XML Formatter tab” on page 33.
3. In the **Structure** field, you must indicate the Structure for your data type to adhere by. Click the button and from the **Open** dialog that appears, select a file with the appropriate

structure information. After you make a selection, click **Open to reopen t**he **Append Pages** dialog.

4. If you chose either **Flat Text** or **XML** as your Data Format, you must arrange the layout of appended PDF pages. Click the **Detail** button and the **PDF Appended Page Formatter** tab of the **Settings window** displays. For more information on its configuration, see "PDF Appended Page Formatter tab" on page 40.

   If you chose **PDF** as your Data Format, proceed to step 5.

5. Click **OK**.

   The **Target** pane reappears.

5.4.17 Object Output Data Format

One of the capabilities of Data Transformation Engine is allowing a `java.lang.Object` as either an input source or an output result.

If your output data format is **Object**, the transform method you call will take a `com.xenos.transform.ObjectResult` object as the output parameter.

The `ObjectTransform.java` sample, located in the `<install_home>\initialFiles\common_sample\DataTransform\embed_sample` directory, demonstrates a transformation that has Object output.

Your target structure can be created in the **Mapping tab** by manually adding items and using the drag-and-drop feature, or it may be loaded from a file or database. If you choose to load your target structure, be sure that it complies with any rules and/or syntax of the data type chosen for your particular output data format.

If you have specified a target structure, check the **Load Structure** box to load it; it will be displayed on the **Mapping tab**. You can load structure files from any URN, HTTP, `.jar` file, or FTP.

The **Result** box is where you specify a location to which you would like to write output you produce. Choose your desired location by clicking the **Ellipsis** button, [...], and browsing your file system.

Select the **Validating** check box to validate against the XML schema that was used to generate the object code.

XBean (short for XMLBean) is supported.

Castor, XBean, and JAXB are **XML-Java binding** frameworks. When selecting a **Binding** method:

- If the structure extension is .xml select from **Castor** or **XBean**. When **Castor** is selected, XBean is ignored.
- If the file extension is **not** .xml, then only **JAXB** are available.
- If the file structure is empty, then **JAXB** and **XBean** are available.

To generate the Java code, click **Generate Code**.

To compile the Java code, click **Compile Code**.

5.4.18 Customizing output formatting

You may customize the formatting of your output if your target data format is “XML Output Data Format” on page 76.

To customize output formatting:

1. Provide formatting details in the “XML Formatter tab” on page 33 of the “Settings window“ on page 15. Click the **Detail** button to open the “Settings window“ on page 15.

2. From the **XML Formatter** drop-down list, select the **XML Formatter** you defined on the **XML Formatter tab**.

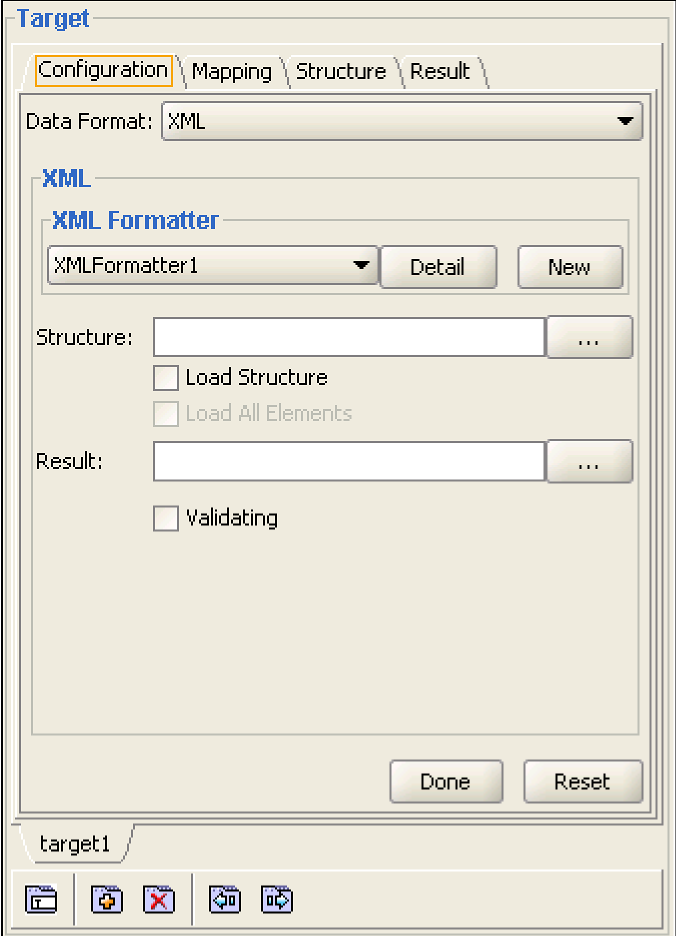


Figure 5-8: XML Output Data Format

Chapter 6

Mappings

The “Source Mapping” on page 91 and the “Target Mapping” on page 92 tabs graphically display the structure of your input and output files. In addition to a graphical display, the “Source Structure” on page 48 and “Target Structure” on page 70 tabs enable you to view the files you have loaded as structural models for your input and output.

The “Source Instance” on page 50 and “Target Result” on page 72 display the *source* and *target* data file contents respectively. Viewing the **Results** tab runs the transform to produce the output file.

6.1 Drag-and-drop feature

Any item on the “Source Mapping” on page 91 tab can be dropped onto the “Target Mapping” on page 92 tab to quickly build an output structure.

To drag-and-drop:

1. Select an item on the **Source Mapping** tab; continue to hold the mouse button down after selection.
2. While holding the mouse button down, drag the item to the **Target Mapping** tab and highlight the target node under which the item will be placed.
3. Release the mouse button. The item dragged will now appear under the selected target node.

6.2 Source Mapping

The **Source Mapping** tab graphically displays the file you have chosen as your source structure in the “Source Structure” on page 48 tab.

To learn more about the drag-and-drop functionality of the **Mapping** tab, see “Drag-and-drop feature” on page 91.

When creating your output structure in the “Target Mapping” on page 92 tab, source items can be moved instantly from the **Source Mapping** tab to the **Target Mapping** tab using the drag-and-drop feature.

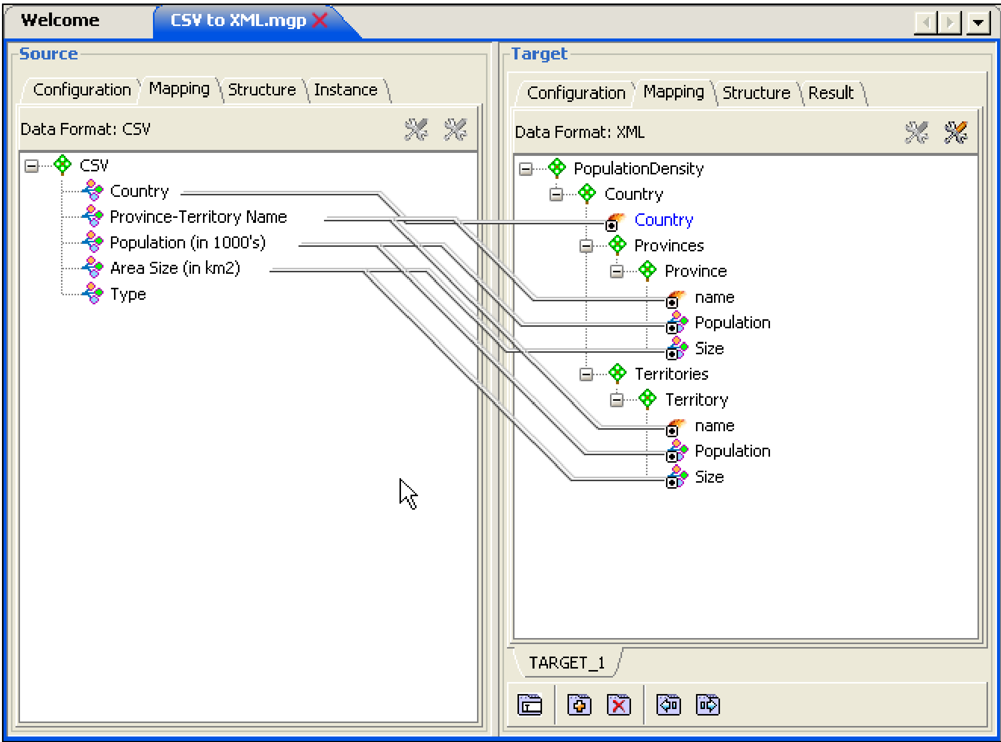


Figure 6-1: Source Mapping Tab

6.3 Target Mapping

The **Target Mapping tab** displays your output structure. If an output structure has been loaded from a file or database, it is shown automatically.

Note: : To learn more about the drag-and-drop functionality of the **Mapping tab**, see "Drag-and-drop feature" on page 91.

If you are creating your output structure, this is the place to do it. Items may be added here using the right-click functionality or by using the drag-and-drop feature.

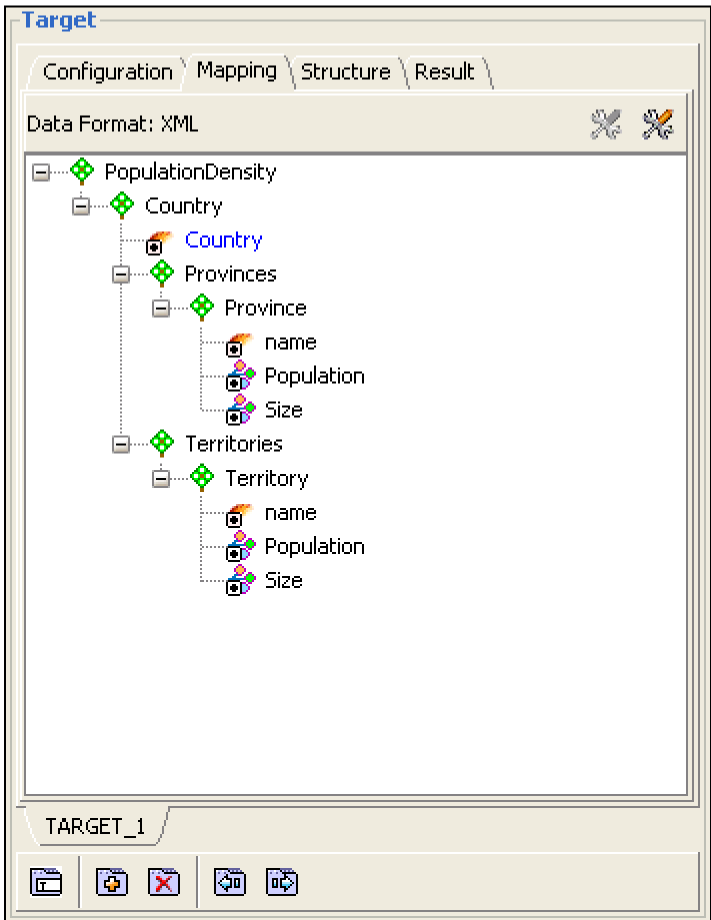


Figure 6-2: Target Mapping tab

6.3.1 Using variable values

You may use the value of a variable in any field that requires user input. The variable must first be defined in the “Variables tab” on page 22 of the “Settings window“ on page 15.

To insert the value of a variable in a field, see “Using a Variable in output” on page 23.

6.4 Output structure and values

This section discusses output structure and values.

Loading and configuring output items is performed in the “Target Configuration” on page 74 tab. It involves specifying an output data format and possibly loading an output structure.

6.4.1 Output structure

Output structure specifies exactly how the output data file is to be laid out. Output structure uses a parent/child system similar to XML to produce output blocks in the output file. An output block is one loop of output data in the output file for a parent and its children. Because parents can be nested within parents for some output data formats, output blocks can contain output blocks.

It is the output structure, not the actual output data, that you design in the “Target Mapping” on page 92 tab. Output structure can be designed from scratch or can be pre-loaded (from a file or database) when you specify the “Target Configuration” on page 74.

To design output structure from scratch, or to alter a pre-existing structure:

1. Select the “Target Mapping” on page 92 tab. Your current output structure will be displayed.
2. Right-click the **parent** of the item you will be inserting.
3. If no structure yet exists, right-click anywhere within the **Target Mapping** tab. A menu will appear.
4. From the menu that appears, select **Add Child**.
5. Select the type of output item you would like to create. See “Output item values” on page 97 for information on how to assign a value to your item.

To pre-load an output structure, see “Target Structure” on page 70.

6.4.1.1 Node icons

The following is a list of the icons you may see in the output structure:

	Parent node; bold font indicates all of a selected node's parent nodes.
	COBOL “Level 01 (L01) nodes” on page 60
	COBOL “User defined nodes” on page 62
	COBOL “Redefined nodes” on page 61
	Element

	Attribute
	Comment
	Content
	CDATA

The optimize feature is available for trimming all unmapped nodes. To optimize your mappings, click the **Optimize** button, , which is located in the top right corner of the output structure pane. The optimize feature is available for all tree-structure output formats except for XML output format.

6.4.1.2 Required structure elements

When mapping very large EDI and XML schemas, many elements are often left unmapped. There is an easy way to know which elements in the target structure are mandatory, so that you can ensure you have mapped an item to them.

When you load EDI dictionary files, the elements which are indicated as *required* are displayed in dark green font, as opposed to black. In addition, when you move the cursor over the nodes, the tool tip will display either:

- mandatory: **true**.
- mandatory: **false**.

The text tables of nodes are displayed in different colors, depending on the required attributes (Req= in the structure):

Attribute (Requirement)	**Node Color**
M (Mandatory)	Green
O (Optional)	Default color (black).
N (Not used)	Gray
Conditionally required	Purple

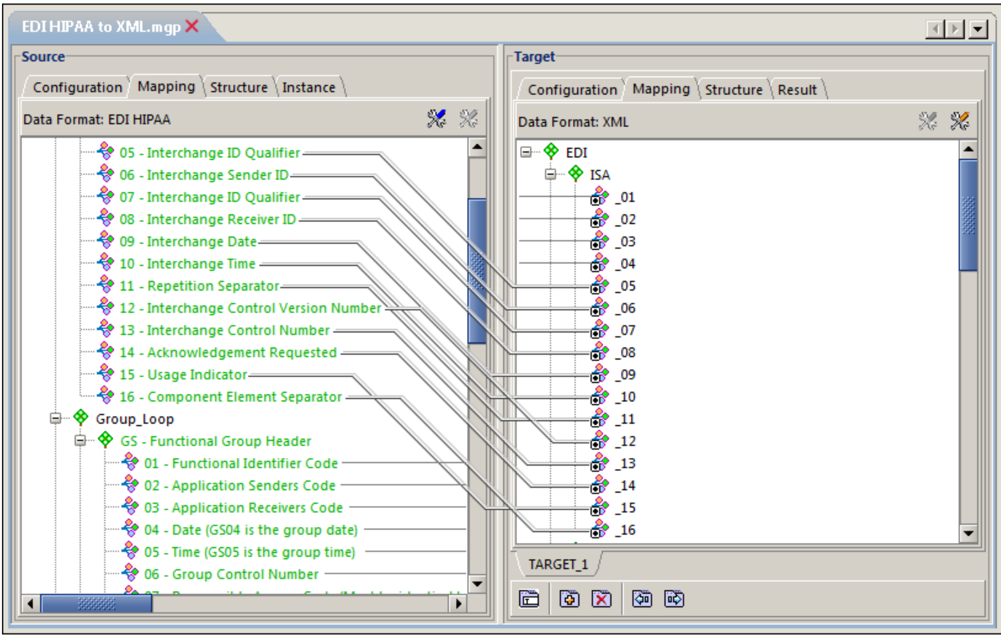


Figure 6-3: Colored nodes in Mapping tab

6.4.1.3 Mapping report and node descriptions

A description of a node can be entered which will display in the Mapping Report. The Mapping Report displays in an Internet browser the current template mappings in HTML format. It can be displayed by selecting the **Report** option of the **Run Menu**.

Node Description

To view, add, or change a description for a node:

1. Select a node displayed on the “Source Mapping” on page 91 or the “Target Mapping” on page 92 tab.
2. Right-click the selected node and select **Description**, or press **Ctrl** + **D**.

   The **Node Description** dialog window appears.
3. The description is entered and displayed in the text box of the **Node Description** dialog window. Either enter a new description or change the current one.

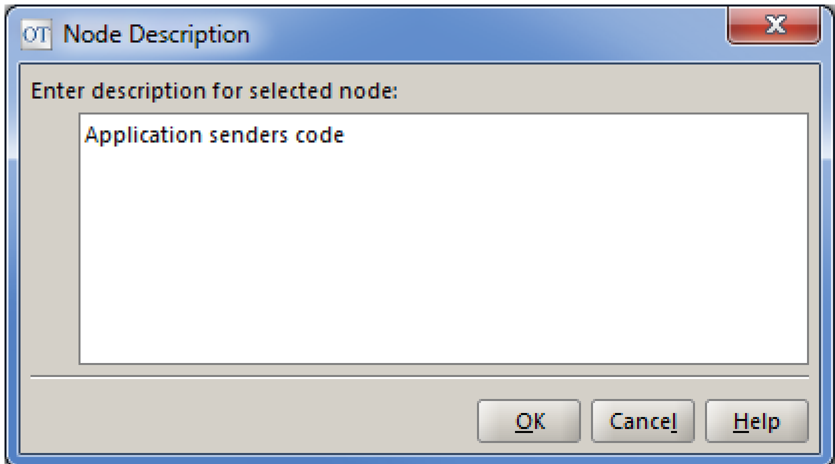


Figure 6-4: Sample Node Description dialog window

6.4.1.4 Output item values

Simply creating an output structure is not enough to perform a transformation. You must specify the value that each output item will take upon transformation.

Note: For information on building output item values from input items, see “Mapping options” on page 98.

An output item value can be built from the following:

- Constant Value
- Result of a Function
- Previous Mapping

Constant value

To set an output item value to a constant value:

1. Right-click on the **output item** in the “Target Mapping” on page 92 tab.
2. Choose **Map to Constant** from the menu that appears. You will be provided with an empty field in which to enter the constant value.

Result of a function

To set an output item value to the result of a function:

1. Right-click on the **output item** in the “Target Mapping” on page 92 tab.
2. Choose **Map to Function** from the menu that appears. The **Function Editor** will appear.

3. Configure the function as described in the “Function Editor” on page 119.

Previous mapping

To set an output item value to a previous mapping:

1. Right-click on the output item from which you want to copy the mapping in the “Target Mapping” on page 92 tab.
2. Select **Cut Mapping** or **Copy Mapping** to store the mapping into memory.
3. Right-click on the **output item** that will contain the copied mapping.
4. Select **Paste Mapping** to apply the new mapping.

6.4.1.5 Mapping options

A mapping specifies an output item value whereby the value of the output item takes (is mapped to) the value of a specified input item.

Your mapping options include:

- “Mapping a value to an existing output item” on page 99

  Allows you to set the value of an item already defined in your output structure to be that of a particular input item. This option is called **Set Mapping**.

- “Adding new output items with input item values” on page 100

  Allows you to drag an input item or input tree (a hierarchy of input items) to add to your current output structure. This option is called **Insert Value-Of Mapping**.

- “Setting structural mappings” on page 102

  If a segment of your output structure has the exact same form, but perhaps not the same names, as a segment of your input structure, you may create a value mapping from the input segment to the output segment. The output will now have the exact values of the input, but the output item names will remain different from those of the input. This option is called **Set Structural Mapping**.

- “Adding an exact copy of an input tree” on page 103

  If you would like an input tree (a hierarchy of input items) to be copied exactly to your output during transformation, you may use the Copy-Of mapping option. Using this tool, you do not have to know the precise structure of your input tree. You must only provide the name of the tree's root. This option is called **Insert "Copy-Of" Mapping**.

Toggle mapping lines

To view your mappings, be sure that the **Show Mappings** button, , on the toolbar is selected.

Select both the “Source Mapping” on page 91 and “Target Mapping” on page 92 tabs. The mappings are displayed by lines connecting source items to target items.

Mapping a value to an existing output item

Whether you have added items to your output structure manually or have loaded an output structure from file, this option is used when you have a target item already in your structure whose value is not defined.

To create a mapping to an existing output item, left-click on the desired input item and drag it onto the existing target item in the **Mapping tab** of the **Target** pane. When you create a mapping in this manner, the value of the target item is automatically set to take the value of the mapped source item.

For example, you can go from Figure 6-5 to Figure 6-6. In Example 2, a value mapping for the *name* output item has been created by dragging the *type* source item onto the *name* target item in the "Target Mapping" on page 92 tab.

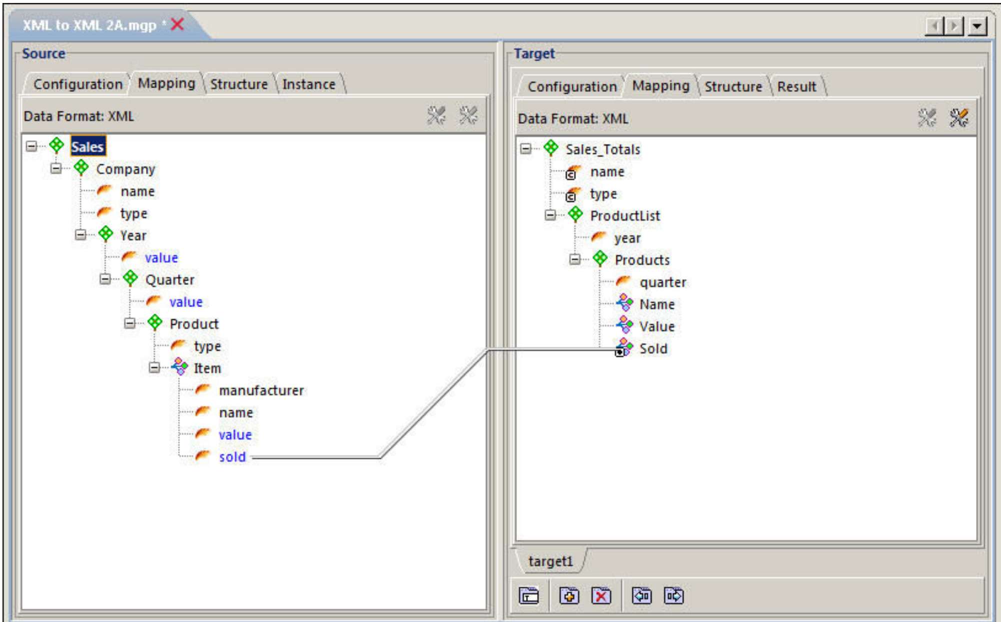


Figure 6-5: Example 1

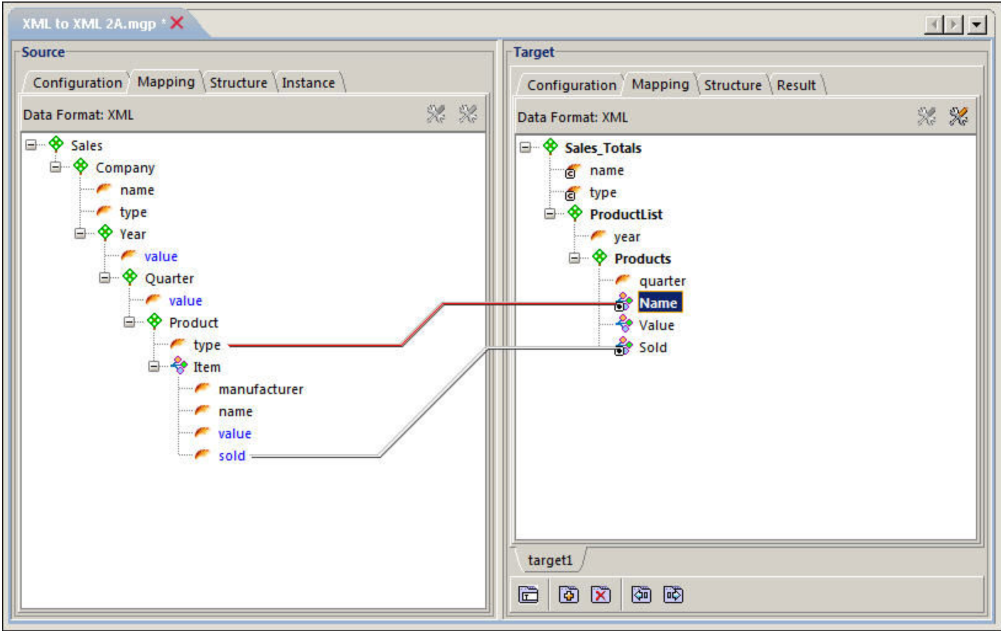


Figure 6-6: Example 2

Adding new output items with input item values

Do not add new items to your output structure if your output format is EDI (X12, HIPAA, and EDIFACT), SWIFT, HL7 or flat text. This alters the output structure loaded from the dictionary or structure file.

To create a new output item, drag the desired input item into the **Mapping** tab of the **Target** pane to the point within your output structure where you want the item to be inserted. When you create a mapping in this way, the value of the new output item(s) are automatically set to take the value of the mapped input item(s).

For example, you can go from Figure 6-7 to Figure 6-8. In Figure 6-8, the *Product* output parent item and its children have been created by left-clicking on the *Product* input item and dragging it onto the *Company* output parent item:

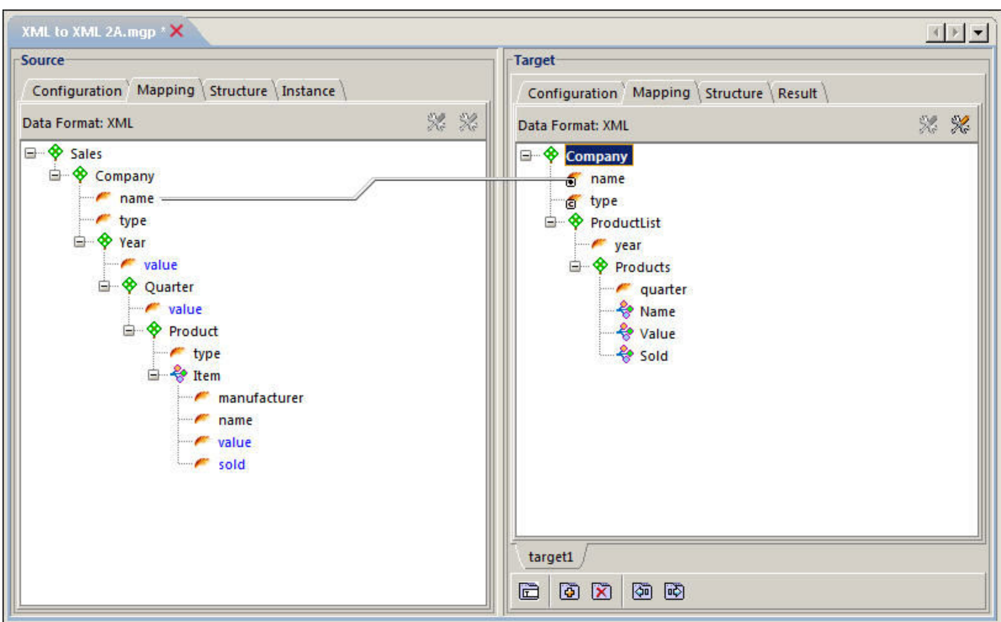


Figure 6-7: Example 1

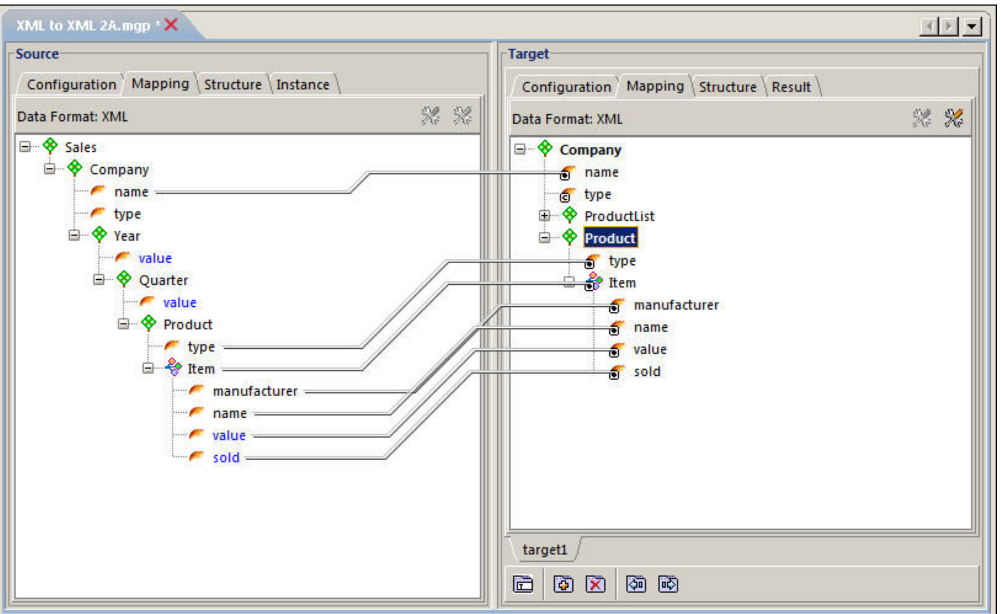


Figure 6-8: Example 2

When structure mapping, you can map an input element or parent item. If you structure map a parent, you include all children of that parent (as in the example above).

For XML output, you can right-drag a structure mapping to obtain finer control over the output items you are adding. If you right-drag an input item onto an output item element, you have the option of **Set Mapping** (set the output item's value) or **Insert Value-Of Mapping** (add the input item as a child of the selected output item).

Setting structural mappings

This option allows you to create a direct mapping from an input segment to an output segment, where the segments' structures are identical. The names of the items within each structure do not have to be the same.

If a segment of your output structure has the exact same form, but perhaps not the same names, as a segment of your input structure, you may create a value mapping from the input segment to the output segment. The output will now have the exact values of the input, but the output item names will remain different from those of the input. This option is called **Set Structural Mapping**.

For example, you can go from Figure 6-9 to Figure 6-10. In Figure 6-10, the *Product* output parent item and its children have had their values set by right-clicking on the *Item* input item, dragging it onto the *Product* output item, and selecting **Set Structural Mapping** from the list of options that appear:

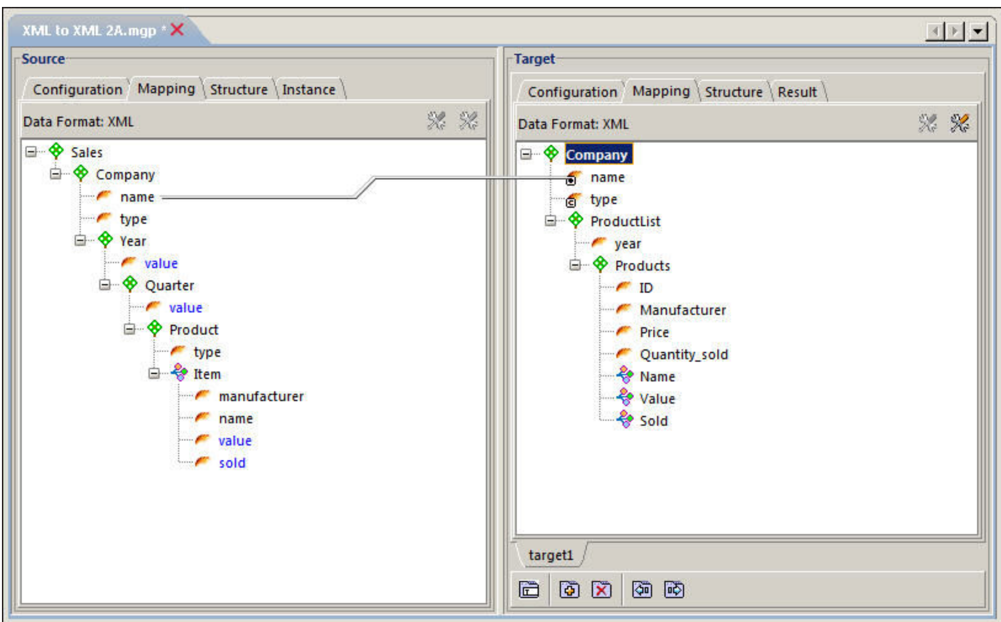


Figure 6-9: Example 1

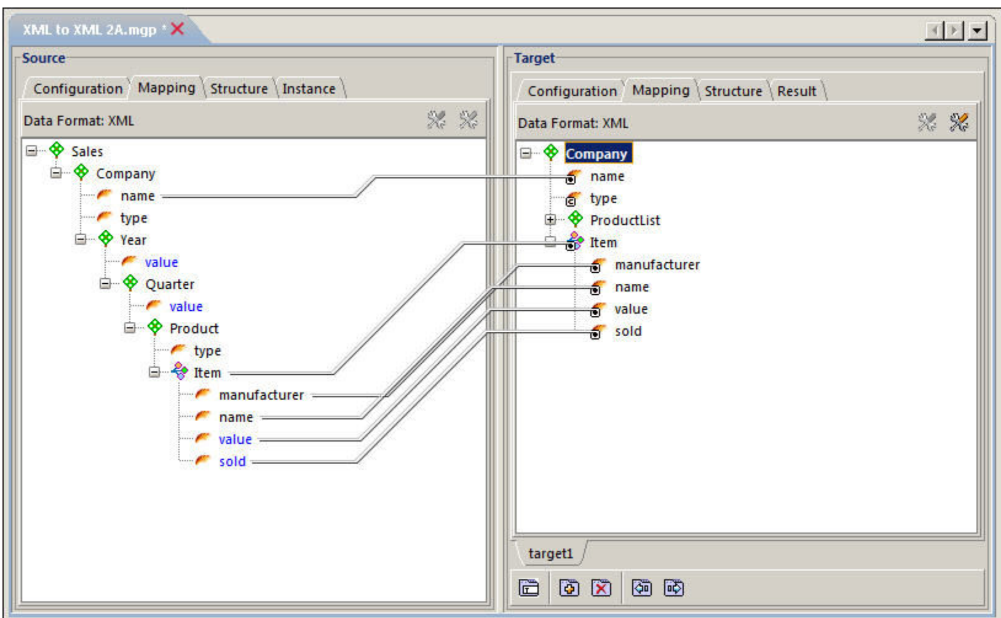


Figure 6-10: Example 2

Adding an exact copy of an input tree

This option allows you to insert an exact copy of a segment of your input, without having to know the precise structure of that portion of the input.

For example, you might have the input and output structures as illustrated in Figure 6-11 and you would like to add the *Item* input item as a child of your *Company* output parent item.

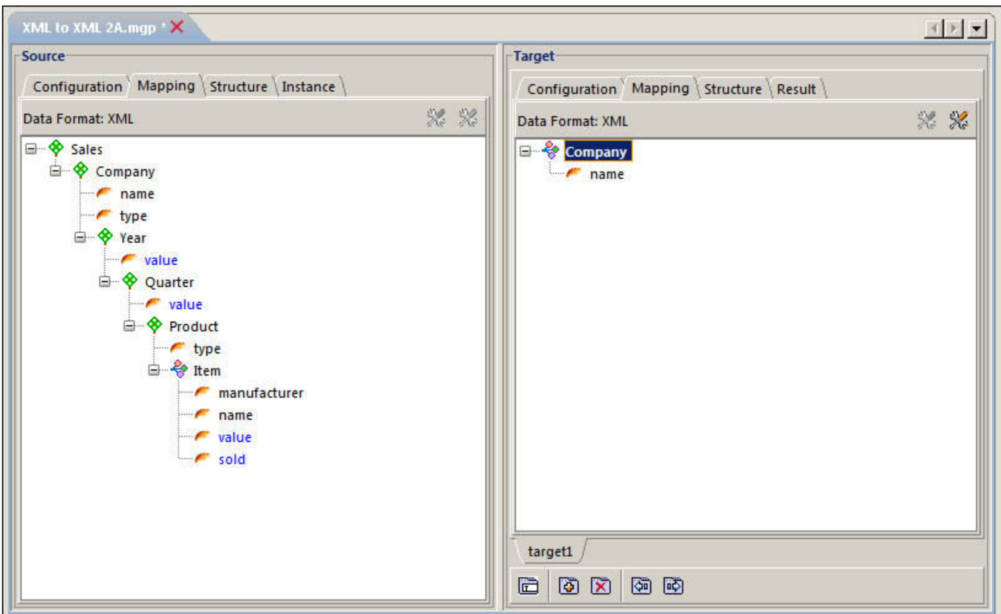


Figure 6-11: Example 1

Some of your input files might not agree exactly with the input structure you have loaded. They may have additional attributes under the *Item* item, or even additional children under *Item*.

In this case, you would right-click on Item in the input structure and drag it onto the *Company* parent item in your output. From the list of options that appears, you would select **Insert "Copy-Of" Mapping**.

Figure 6-12 would be the result. The actual output you produce would include all possible attributes and children that *Item* might contain in the input.

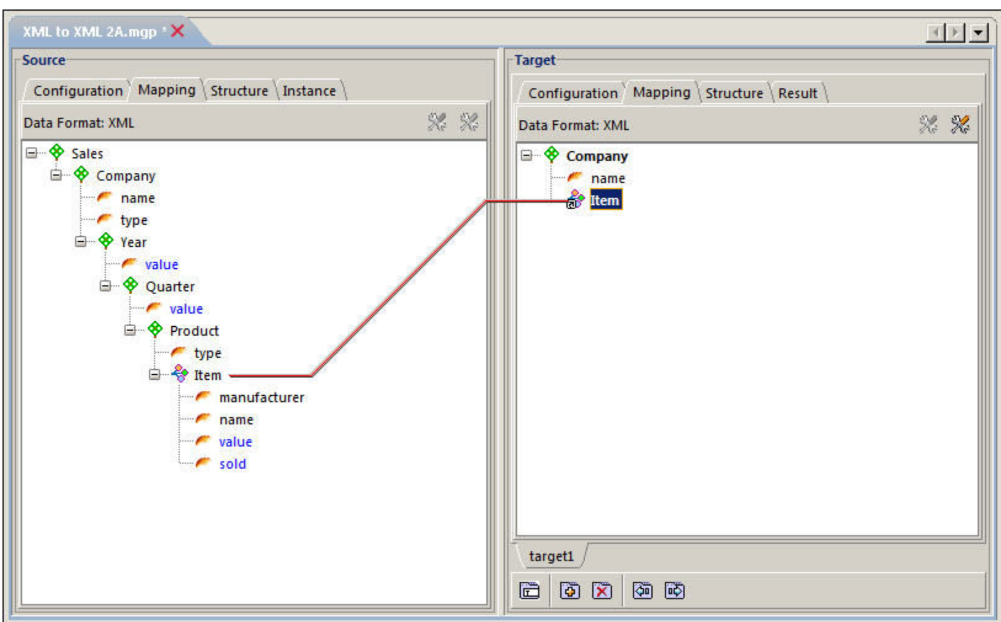


Figure 6-12: Example 2

Enumerated nodes

Enumerated nodes are shown in blue text in the following figure:

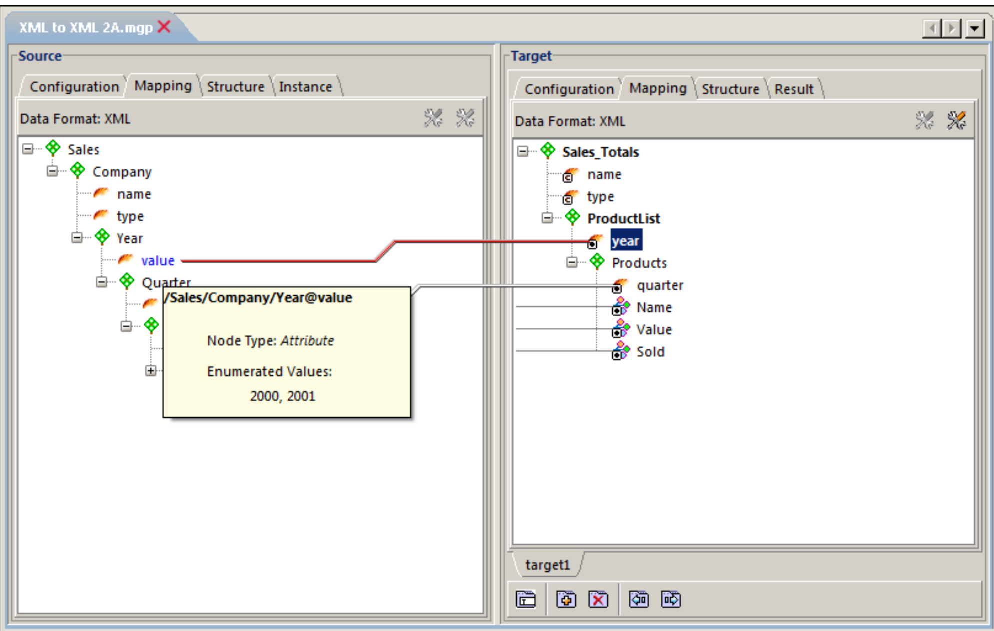


Figure 6-13: Enumerated nodes shown on Source and Target Mapping tabs

During a transformation an enumerated node is mapped to a value using the *_enum_#* (where, # is an incremental number or *_enum_#* can be renamed in Find & Replace) in the "Find & Replace tab" on page 26 of the **Configuration Pane.** Thus, using the enumerated nodes is a faster method of implementing find and replace.

Enumeration mapping

1. Right-click the **mapped enumeration node** in the **Target Mapping** tab, or press **F7** to open the **Enumeration Mapping** dialog.

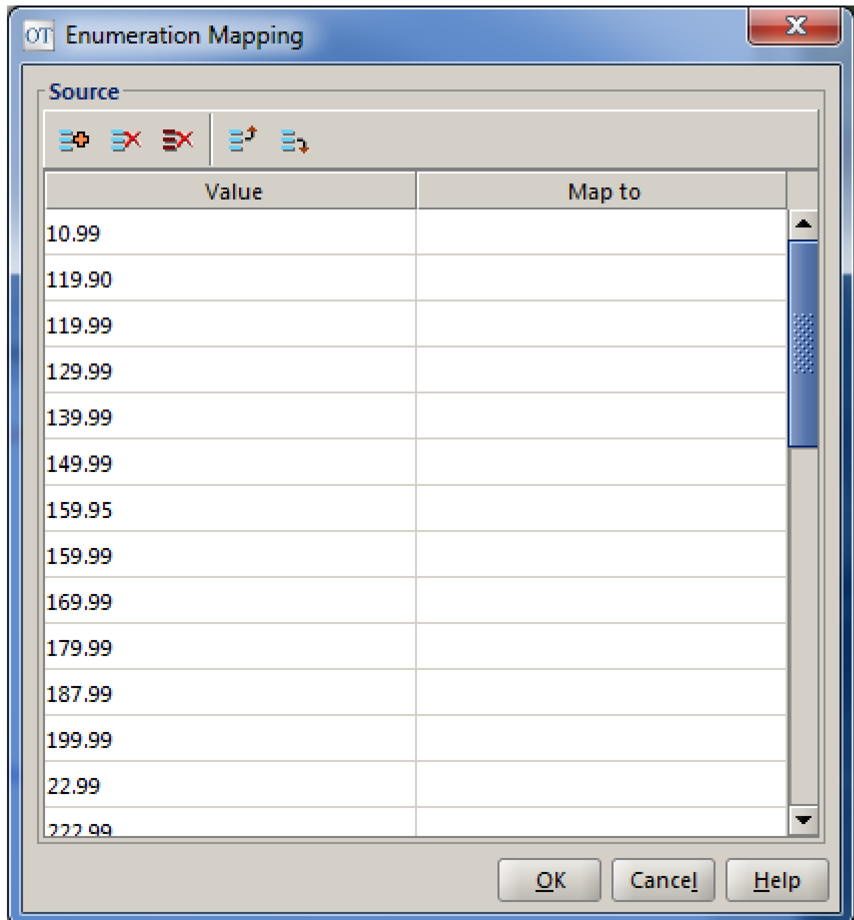


Figure 6-14: Sample Enumeration Mapping dialog

2. In the **Map to** column, enter the value you wish to replace the value listed in the **Value** column.

   By default, Data Transformation Engine populates the **Value** column with enumerated values and leaves the **Map to** column blank.

3. You must remove rows that have empty **Map to** columns, or click the **Cancel** button, to make the transformation more efficient.

Note: If this is not done, Data Transformation Engine will create a find and replace table, even if the **Map to** column is empty. For more information, see "Find & Replace tab" on page 26.

COBOL Enumerated Values

When COBOL contains enumerated values, the values are shown in the right-hand table of the **Node Properties** window.

Enumerated Output

When both the source data and the output data structure contains enumerated nodes and they are mapped to each other, the enumerated mapping window contains the source and target enumerated value tables.

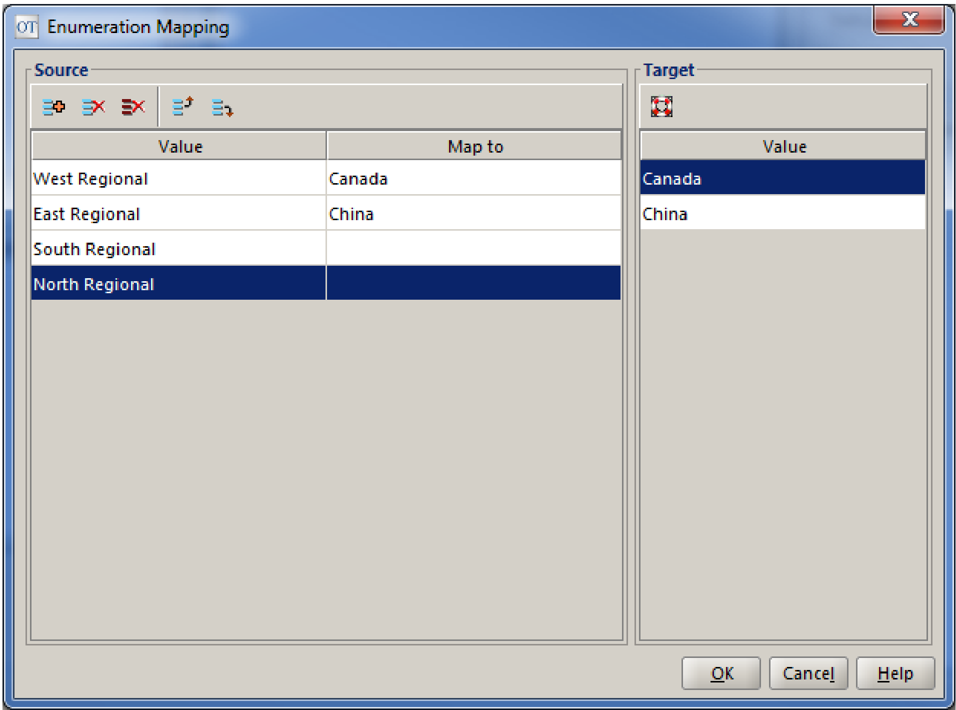


Figure 6-15: Enumeration Mapping dialog

To copy the values from the **Target** table to the **Map to** column of the **Source** table, click the **Enumerated Value Copy** button, .

The process during a transformation is the same as with non-enumerated output above.

6.4.1.6 Parent items

Every output structure, regardless of the output data format, has at least one output parent item, shown with the, ◆, icon. Bold font indicates all of a selected node's parents nodes.

A parent item marks the start of an output block by specifying the start of an output loop. The first item in your output structure is always a parent item. The first parent item tells Data Transformation Engine to produce output by looping through the entire output structure, while reading the input file and performing the transformation.

Extra Effects

In addition to signalling the start of an output loop, a parent item can have other effects depending on the output data format:

XML	The first parent item value specifies document root. Other parent item values specify an XML parent tag name. Parent items may be used to create complex output structures by means of nested output loops.
EDI	The parent item value specifies an XML parent tag name. Parent items may be used to create complex output structures by means of nested output loops.
Flat Text	Parent items may be added for looping and filtering purposes that otherwise could not be achieved. These items, however, will not be visible in the actual output data.

Note: The first parent item specified for JDBC and SQL is the output table name.

Parent properties

You can set **node properties** for a parent output item.

To view node properties:

1. Right-click on a **parent output item** in the “Target Mapping” on page 92 tab.
2. Select **Properties**.

   The **Target Node** window appears, which contains two tabs:

   - “General tab” on page 109
   - “Advanced tab” on page 111

Note: Properties differ depending on the output data format. Properties that do not pertain to your output data format are not shown in the properties window for an output parent item.

General tab

The General tab features several basic options.

Filter

The **Filter** parent property lets you set a condition on a parent, and therefore on each individual output block produced by that output loop. If the parent condition you enter evaluates to false for an output loop, then that particular block is not written to your output.

The expression you enter in the **Filter** field can be any complex logic statement consisting of a combination of values from your input, constants, operators, function calls and brackets. If you will be using the value of an input item, it should take the following form according to its type:

- Parent item:

```
/parent1/parent2/.../yourParentItem
```

- Element:

```
/parent1/parent2/.../yourElement
```

- Attribute:

```
/parent1/parent2/.../element@yourAttribute
```

The possible operators are:

Operator	Meaning
==	equal to
!=	not equal to
>=	greater than or equal to
<=	less than or equal to
>	greater than
<	less than
&&	and
||	or

Trim

The **Trim** option tells Data Transformation Engine how to deal with empty output tags (that is, output tags with no value and whose attributes have no values).

- **None.** All empty tags will be retained in output.
- **Child Blocks.** Tags whose children all have empty values, and whose own values are also empty, will be removed from output.
- **All Nodes.** All empty tags will be removed from output.

Unique Keys

You can control the uniqueness of a parent's output child items by defining items as unique. For example, assume that you will be generating the following non-unique XML output data:

```
<Product>
<ProductID>XR281</ProductID>
<Name>Green Rocket Vehicle</Name>
</Product>
<Product>
<ProductID>SR71</ProductID>
<Name>SR-71 Blackbird</Name>
</Product>
<Product>
<ProductID>XR281</ProductID>
<Name>Green Rocket Vehicle Again</Name>
</Product>
```

If you choose to make the output data unique by specifying the output item *ProductID* in the **Unique Keys** parent property, then your final output is the following:

```
<Product>
<ProductID>XR281</ProductID>
<Name>Green Rocket Vehicle</Name>
</Product>
<Product>
<ProductID>SR71</ProductID>
<Name>SR-71 Blackbird</Name>
</Product>
```

The second instance of the XR281 product has been thrown out since it was not unique to the output item *ProductID*.

Sorted By

The **Sorted By** section lets you choose a child element by which to sort the output. If you specify a child item in this section, the output is sorted in ascending or descending alphanumeric order by the element you choose.

To sort your output by a particular child item:

1. Open the Parent Properties window for that child's parent.
2. Click the **Insert Row** button, , in the Sorted By section. A row will be automatically added, with the parent's first child as the default sort value.

To change the child that will be used to sort:

1. Double-click on the **child's name**. A drop-down list of all of the children's names appears.
2. Select the **child** you wish to use.

Insert/Update

The **Insert/Update** radio button pair show up for output data formats JDBC and SQL. Your selection here determines the type of SQL statement written to your template file.

Where Keys

For output data formats JDBC and SQL, use the **Where Keys** option to add **Where** conditions to the SQL statement written in your template file. You may only add Where conditions when the SQL statement uses Update.

To add a Where condition, click the **Insert Row** button, , in the **Where Keys** section of the parent properties dialog. A row will be automatically added, with the parent's first child as the default Where value. To change the child that will be used:

1. Double-click on the **child's name**. A drop-down list of all of the children's names appears.
2. Select the **child** you wish to use.

Advanced tab

Loop

The **looping parent** property tells Data Transformation Engine how to deal with multiple instances of a parent item. All parent items default to **None**.

- **None.** Data Transformation Engine decides how your parent element will loop.
- **False.** Indicates that the element will not loop.
- **True.** Indicates that the element will loop as many times as is required by the input.

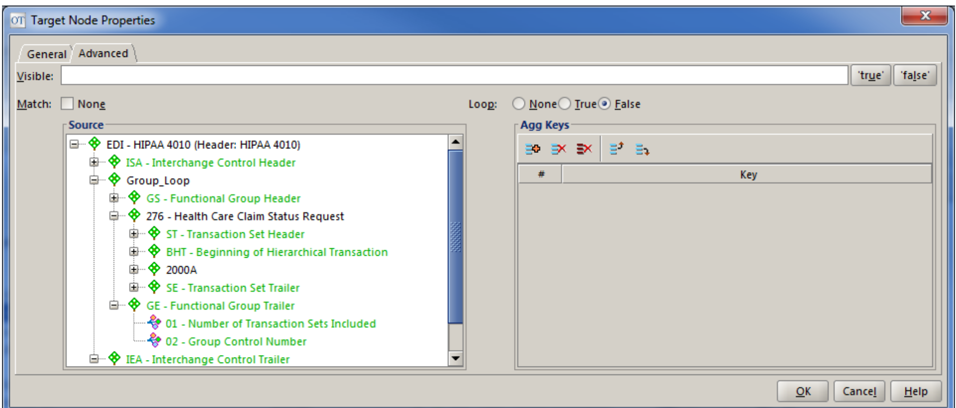


Figure 6-16: Target Node Properties - Advanced tab

Visible

One reason to have an invisible parent is if you have a set of sibling elements in your output, and would like to apply certain conditions to only some of these elements. You would then add a parent item whose **Visible** option you would configure, and make the relevant elements children of the specified parent. It would then be possible to apply the desired conditions to these elements while maintaining your previous output structure.

The **Visible** field may be set to **true** or **false**, or a condition. The condition takes the same form as the condition in the Filter parent property.

Match

Applying the **Match** parent property to a parent item in your output structure can solve looping complexity issues. It is especially useful in transformations in which elements from different levels in the input hierarchy are being mapped to the same level of output hierarchy, and one or more of the input elements loops. **Match** specifies to the Data Transformation Engine run-time engine where the block of input that is being used for looping begins. As a general rule, this should be set to the innermost parent block that encapsulates all of the data that you need to contain.

To set the level at which looping should begin during transformation, uncheck the **None** check box in the **Match** section of the Parent Properties dialog. The input structure will appear below. Select the input item from which looping shall begin. You may also use the **Find** tool within the **Match** window to locate items by name.

Note: To access the **Find** tool in the **Target Node Properties** dialog window, right-click the empty white space of the **Source** box.

Agg Keys

You can aggregate several attributes of the same element together by using the **Agg Keys** parent property. This is very useful if you have filtered out attributes and want to combine those attributes at a certain time. For example, you have filtered out the c2 and c3 attributes from the first Color element and filtered out the c1 attribute from the second element:

```
<Color c1="red">Rainbow</Color>
<Color c2="orange" c3="yellow">Rainbow</Color>
```

By adding the Color element as an **Agg Key**, you will get the output:

```
<Color c1="red" c2="orange" c3="yellow">Rainbow</Color>
```

There is only one Color element with three attributes.

Data typing

For output formats **JDBC** and **SQL**, Data Transformation Engine allows you to choose whether or not you want to apply data typing to your output elements. Data typing is a form of validation in which your actual output values are compared against the data types that you specify are allowed for those values.

Example 6-1: Data typing example

You have an element called Sold in your output that represents the number of items sold for a certain product. You may want to ensure that all output values attributed to this element are integers. After applying the integer data type to the Sold element, any values of Sold that are not integers (for example, floats or booleans), will result in transformation errors.

Data Types

Your choices when applying data types to output item values are the following:

• BIGINT	• LONGVARCHAR
• BIT	• NUMERIC
• CHAR	• REAL
• DATE	• SMALLINT
• DECIMAL	• TIME
• DOUBLE	• TIMESTAMP
• FLOAT	• TINYINT
• INTEGER	• VARCHAR

Applying data types to output item values

To apply a data type to a specific element or attribute in your JDBC or SQL output:

1. Right-click on the **item's name** in the **Mapping tab** of the **Target** pane.
2. From the context menu that appears, select **Properties**.

   The **Target Node Properties** window appears.
3. If your desired data type does not appear in the **Data Type** field, click the down arrow on the right. You will be presented with a list of system data types from which to choose.

4. After making your selection, click **OK**.

6.4.1.7 Filter

For all output data formats except **JDBC** and **SQL**, you may set a filter for any non-parent output item.

To set a condition for a non-parent output item:

1. Right-click the **output item** in the **Mapping** tab of the **Target** pane.
2. **Choose Properties** from the resulting pop-up menu. The **Properties** window for that output item is displayed.

The **Filter** field lets you set a condition on that output item. If the expression you enter is false as the engine runs through a particular loop of the input data, the element will not appear at all in your output for that iteration. Conversely, if the expression you enter is evaluated to be true, the element will appear in your output.

The expression you enter in the **Filter** field can be any complex logic statement consisting of a combination of values from your input, constants, operators, function calls and brackets. If you will be using the value of an input item, it should take the following form according to its type:

- Parent item:
```
/parent1/parent2:.../yourParentItem
```
- Element:
```
/parent1/parent2/.../yourElement
```
- Attribute:
```
/parent1/parent2/.../element@yourAttribute
```

The possible operators are:

Operator	Meaning
==	equal to
!=	not equal to
>=	greater than or equal to
<=	less than or equal to
>	greater than
<	less than
&&	and
||	or

Example

If you are mapping a Year element from your input to output, but only want those whose value attributes are '2001', you would enter

```
/Sales/Company/Year@value == '2001'
```

in the **Filter** field of your output Year item.

6.4.1.8 Namespaces

XML namespaces provide a way to distinguish between duplicate element type and attribute names. This duplication may occur, for example, in an XSLT stylesheet or in a document that contains element types and attributes from two different DTDs.

An XML namespace is a collection of element type and attribute names. In an XML namespace, an element type or attribute name is uniquely identified by a two-part name: the name of its XML namespace and its local name.

Using XML namespaces

If your output data format is XML, you may create and use your own namespaces within the **Mapping** tab of the **Target** pane.

There are two steps to using namespaces:

1. **Declare** (create) the namespaces you will be using.
2. **Apply** the namespaces to your output items.

Declare namespace

To declare a namespace:

1. Right-click on any **output item** in the **Mapping** tab.
2. Choose **Declare Namespaces** from the menu that appears.
3. In the dialog that appears, click on the **Insert Row** button, . A row will be added to your list of namespaces to which you may now enter the corresponding prefix and URI.
4. Click **OK** when you are done.

Apply namespace

To apply a namespace to particular output item:

1. Right-click an **item** in the **Mapping** tab.
2. **Choose Namespace** from the menu that appears. The list of namespaces you have declared will appear in a dialog.

3. Select a **namespace**.
4. Click **OK**.

XML pre-defined namespaces

With a pre-defined XML namespace prefix, you can create attributes with, for example, the name `xml:lang`.

To use the XML namespace prefix:

1. Right-click a **target node** on the“Target Mapping” on page 92 tab.
2. Select **Namespace.**
3. Select the predefined **XML** namespace from the list.
4. Click **OK**.

6.4.1.9 Dynamic Load and Unload elements

Nodes with their names displayed in purple font indicate that sub-elements are not loaded. Right-click the element and select **Load Elements** from the shortcut menu that appears.

Nodes with their names displayed in peach font indicate that sub-elements are loaded. Right-click the element and select **Unload Elements** from the pop-up.

Note: By default, the **Load All Elements** check box in the “Source Configuration” on page 51 and the “Target Configuration” on page 74 tabs is *unchecked*. Selecting this check box may cause Data Transformation Engine to issue an **out of memory** error.

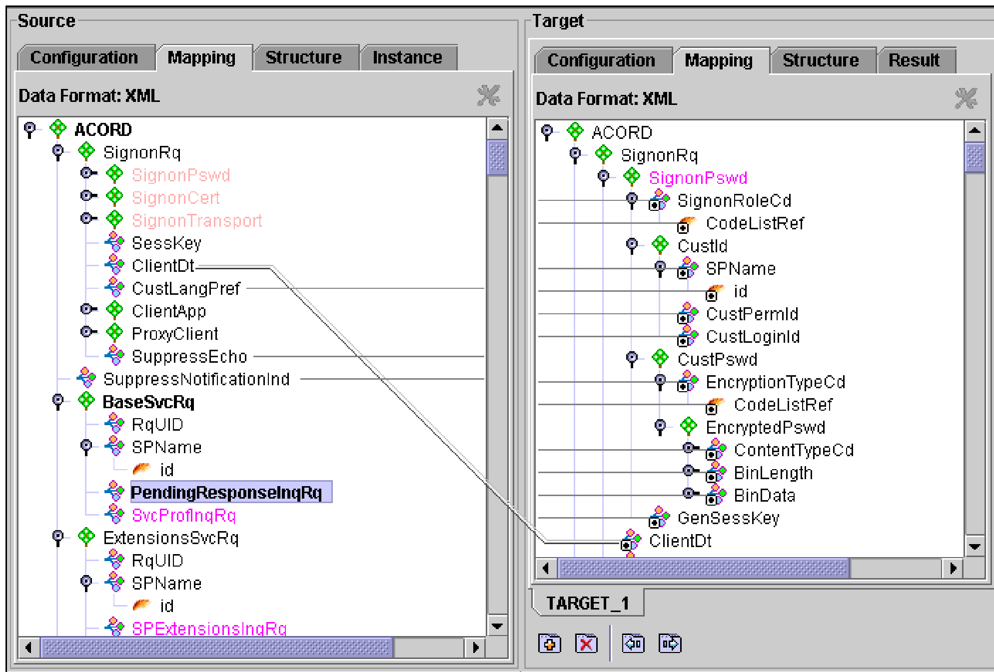


Figure 6-17: Dynamic XML Mapping

Chapter 7

Functions

A function can be used to produce the value of output by specifying that function within an output item value. A function is a piece of Java code written to perform a calculation upon, or manipulation of, input data. Data Transformation Engine provides many pre-defined functions. These are all described in the list of "Pre-defined Functions" on page 122.

You can also define your own custom functions. All custom functions must be written in Java. See "Defining custom functions" on page 189 for more information.

7.1 Function Editor

The Function Editor is a simple tool for building a function for an output item value. It is displayed when you select the Map to Function option on an output item in the "Target Mapping" on page 92 tab.

You can access "Pre-defined Functions" on page 122 and any "Custom functions" on page 189 that you have defined.

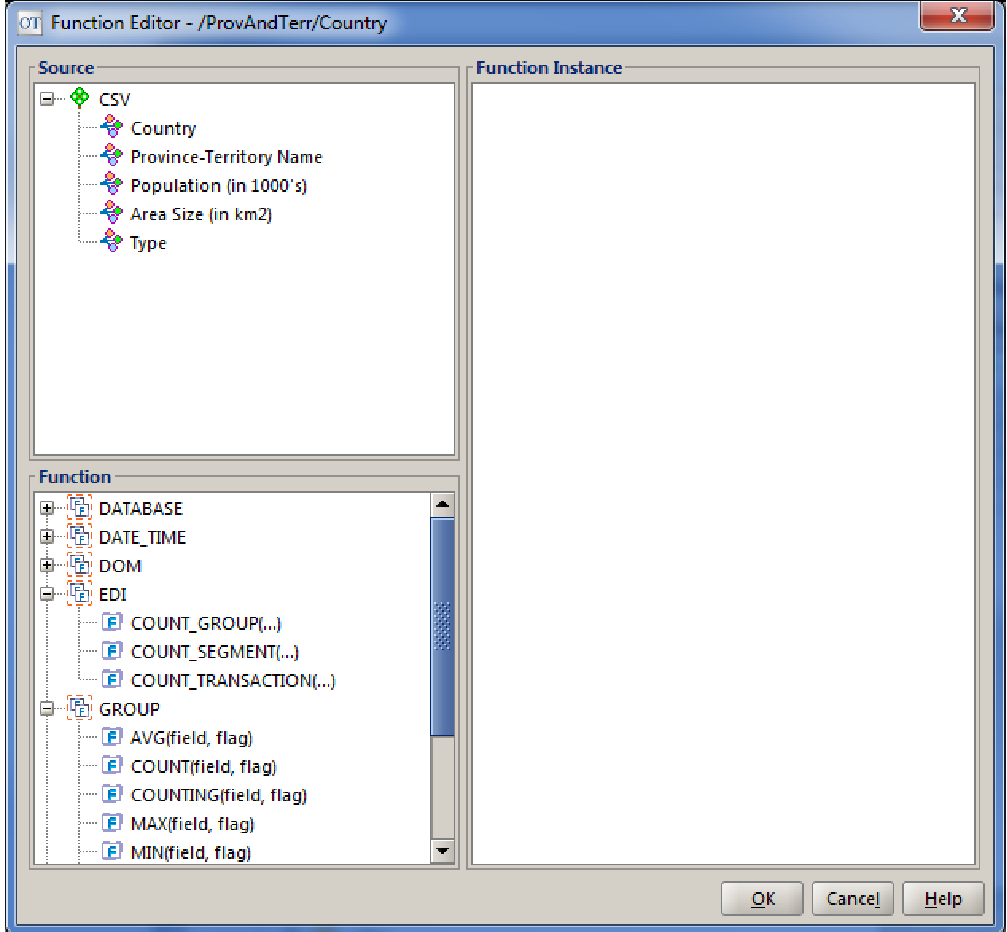


Figure 7-1: Function Editor window

7.1.1 Selecting your Function

Functions are divided into different categories depending on their functionality. To select a function to use, do the following:

1. Select the function you wish to use in the **Function** pane of the **Function Editor**.
2. Drag the **selected function** onto the **Function Instance** pane of the **Function Editor**.

To change the function you have selected, drag a **new function** onto the current function.

The next step is to set the function parameters.

7.1.2 Setting Function parameters

Most functions have associated parameters. If you do not know what parameters you must specify for your working function, you can learn about a function in the “Pre-defined Functions” on page 122 section.

Function parameters take three types of values:

- **An input item**

  To set a parameter to the value of an input item:

  1. Ensure you have selected your function. See “Selecting your Function” on page 120.
  2. Drag the input item of your choosing from the Source portion of the Function Editor onto the parameter you would like to set in the Instance section of the Editor. A line, indicating the mapping between input item and parameter, will appear.

- **A constant value**

  To set a parameter to a constant value:

  1. Ensure that you have selected your function. See “Selecting your Function” on page 120.
  2. Right-click on the parameter you would like to set, and choose **Map to Constant** from the menu that appears. A field will appear in which you may enter the constant value.

- **A nested function**

  To set a parameter to be a nested function:

  1. Ensure you have selected your function. See “Selecting your Function” on page 120.
  2. Drag the **function you wish to nest** from the Function portion onto the parameter in the Instance section of the Editor. A line, indicating the mapping between function and parameter, will appear. Ensure that you set the nested function's parameters.
  3. You can also nest a function that you have already created to another function. You can accomplish this by doing a right-click on the **function name** and selecting **Copy**.
  4. Now that you have copied the function, you can then create another function, right-click on the **function parameter's name** and select **Paste**. This will nest the previously copied function to the current function.

7.1.3 Add a parameter from the Function Editor

You can right-click on the function's name in the **Function Instance** pane and select **Add a Parameter** for the following functions:

- "CONCAT" on page 132
- "COUNT_TRANSACTION" on page 169
- "COUNT_SEGMENT" on page 168
- "COUNT_GROUP" on page 168
- "REPLACE" on page 157
- "XPath" on page 171
- "XPathEval" on page 173
- "ORASTOREDFUNC" on page 178

7.2 Pre-defined Functions

The "Function Editor" on page 119 allows you to use functions to set "Output item values" on page 97. You can use both "Custom functions" on page 189 and more than 90 pre-defined functions. When specifying a function through the **Function Wizard**, you must know what the specific function does and what parameters you must specify for it.

The pre-defined functions are grouped into separate categories as discussed in this section.

7.2.1 Date/Time Functions

Date/time functions are covered in this section.

7.2.1.1 ADD_DATE

The **@ADD_DATE** function performs calculation on a date variable, and constructs a date out of an input date and three values to be added to that date: years, months, and days. The result is always a valid date.

Return

The return type is java.lang.String.

Parameters

@ADD_DATE takes four parameters:

Date	Format: (MM/dd/yyyy).
Years	The number of years to add to parameter 1.

Months	The number of months to add to parameter 1.
Days	The number of days to add to parameter 1.

Example

`@ADD_DATE('06/01/2003', '1', '2', '3')` returns **08/04/2004 (Format: MM/dd/yyyy)**.

Note: Parameter values of zero (0) are ignored.

7.2.1.2 ADD_TIME

The **@ADD_TIME** function performs a calculation on a time variable. It constructs a time out of an input time and three values added to that time: hours, minutes, and seconds. The result is always a valid time.

Return

The return type is java.lang.String.

Parameters

@ADD_TIME takes four parameters:

Time	A Time Value.
Hours	The number of hours to add to parameter 1.
Minutes	The number of minutes to add to parameter 1.
Seconds	The number of seconds to add to parameter 1.

Example

`@ADD_TIME('12:00:00', '1', '2', '3')` returns **13:02:03**.

Note: Parameter values of zero (0) are ignored.

7.2.1.3 DATE

The **@DATE** function returns the system date.

Return

The return type is java.lang.String.

Parameters

@DATE takes one parameter:

Date	A date format. **Note:** @DATE does not allow alphabet characters except for: **G** (era designator) **y or Y** (year) **M or m** (month) **w** (week in year) **W** (week in month) **d or D** (day in month) **F** (day of week in month) **E** (day in week)

Example

If the system date is 01/28/1992, `@DATE('dd/MM/yyyy')` returns **28/01/1992**.

7.2.1.4 DAY

The **@DAY** function returns the day portion of a date (a number between 1 and 31).

Return

The return type is java.lang.String.

Parameters

@DAY takes one parameter:

- **Date**. A date or date expression.

Example

`@DAY('01/28/1992')` returns **28**.

The following expression displays the word “Overdue” if the system is later than the 15th of the month: `@IF(@DAY(@DATE('MM/dd/yyyy'))>'15','Overdue',@NULL())`

7.2.1.5 DOW

The **@DOW** function returns the number of the day of the week from an @DATE() function, where Sunday is 1, Monday is 2, and so on.

Return

The return type is java.lang.String.

Parameters

@DOW takes one parameter:

Date	A date (Format: MM/dd/yyyy).

Example

`@DOW('01/29/1992')` (representing a Wednesday) returns **4**.

The following expression displays a message if the system date is a Sunday:
`@IF(@DOW(@DATE('MM/dd/yyyy'))=='1','Sunday',@NULL())`

7.2.1.6 DSTR

The **@DSTR** function converts a date or date expression to a character string according to a picture mask.

Return

The return type is java.lang.String.

Parameters

@DSTR takes three parameters:

Date	A date.
Character String Picture Mask (parameter)	The character string picture mask (format) of parameter 1. See "Picture masks" on page 180 for more information.
Character String Picture Mask (date)	The character string picture mask (format) that the date will be converted to. See "Picture masks" on page 180 for more information.

Example

`@DSTR('01/28/2004','MM/dd/yyyy','EEEEEE, MMMMMM dd, yyyy')` returns '**Wednesday, January 28, 2004**'.

7.2.1.7 DVAL

The **@DVAL** function converts a date entered or stored as a character string to a numeric value of type string. The numeric value represents the number of days elapsed since the day before the first day of the 1st century (01/01/01) until the date that is being converted.

Return

The return type is java.lang.String.

Parameters

@DVAL takes two parameters:

Character String	A character string or an alpha expression that can be interpreted as a date (for example, '01/01/92', 'Jan 1, 1992').
Parameter Format	The format for parameter 1; this parameter is required for the system to read and interpret the character string or expression.

Example

`@DVAL('01/01/92', 'MM/dd/yy')` and `@DVAL('Jan 1, 1992', 'MMM dd, yyyy')` each return **727198**.

7.2.1.8 EOM

The **@EOM** function returns the date of the end of the month specified in the parameter.

Return

The return type is java.lang.String.

Parameters

@EOM takes one parameter:

Date	A date or date expression (MM/dd/yy).

Example

`@EOM ('05/05/93')` returns **05/31/93**.

7.2.1.9 EOY

The **@EOY** function returns the date of the end of the year specified in the parameter.

Return

The return type is java.lang.String.

Parameters

@EOY takes one parameter:

Date	A date or date expression (MM/dd/yy).

Example

`@EOY ('10/05/93')` returns '**12/31/93**'.

7.2.1.10 HOUR

The **@HOUR** function returns a number of type string that represents the hours part of a time value or a time expression.

Return

The return type is java.lang.String.

Parameters

@HOUR takes one parameter:

Time	A time value or a time expression (HH:mm:ss).

Example

`@HOUR('12:02:05')` returns **12**.

Where the variable A represents the time value 14:22:19, the expression `@HOUR(A)+2` returns **16**.

7.2.1.11 MINUTE

The **@MINUTE** function returns a number of type string that represents the minutes part of a time value or a time expression.

Return

The return type is java.lang.String.

Parameters

@MINUTE takes one parameter:

Time	A time value or a time expression (HH:mm:ss).

Example

`@MINUTE('12:02:05')` returns **2**.

If the variable A represents the time value 14:22:19, the expression `@MINUTE(A)+2` returns **24**.

7.2.1.12 MONTH

The **@MONTH** function returns the month portion of a date.

Return

The return type is java.lang.String.

Parameters

@MONTH takes one parameter:

Date	A date or a date expression (MM/dd/yy).

Example

`@MONTH('01/28/92')` returns **1**.

The following expression displays the word “Overdue” if the system month is later than February: `@IF(@MONTH(@DATE('MM/dd/yyyy'))>'2','Overdue',@NULL())`

7.2.1.13 SECOND

The **@SECOND** function returns a number of type string that represents the seconds part of a time value or a time expression.

Return

The return type is java.lang.String.

Parameters

@SECOND takes one parameter:

Time	A time value or a time expression (HH:mm:ss).

Example

`@SECOND('12:02:05')` returns **5**.

7.2.1.14 SOM

The **@SOM** function returns the date of the start of the month specified in the parameter.

Return

The return type is java.lang.String.

Parameters

@SOM takes one parameter:

Date	A date or date expression (MM/dd/yy).

Example

`@SOM ('05/18/93')` returns '**05/01/93**'.

7.2.1.15 SOY

The **@SOY** function returns the date of the start of the year specified in the parameter.

Return

The return type is java.lang.String.

Parameters

@SOY takes one parameter:

Date	A date or date expression (MM/dd/yy).

Example

`@SOY ('10/05/93')` returns '**01/01/93**'.

7.2.1.16 TIME

The **@TIME** function returns the system time.

Return

The return type is `java.lang.String`.

Parameters

@TIME takes one parameter:

Time	The format of the resulting character string. For more information, see "Picture masks" on page 180. @TIME does not allow alphabet characters except for: **a** (am/pm) **H or k** (hour in day 0-23) **h or K** (hour in am/pm 1-12) **m or M** (minute in hour) **s or S** (second in minute) **z** (general time zone) **F** (day of week in month) **Z** (day in week) **Note:** Only available in JDK 1.4.2 or higher.

Example

`@TIME ('HH:mm:ss')` returns **17:08:42**.

7.2.1.17 TSTR

The **@TSTR** function converts a time to an alpha string, according to a picture mask.

Return

The return type is `java.lang.String`.

Parameters

@TSTR takes three parameters:

Converted Time	A time to be converted.
Parameter Format	The format of parameter 1. For more information, see "Picture masks" on page 180.

Character String Format	The format of the resulting character string. For more information, see “Picture masks” on page 180.

Example

`@TSTR('3:51 PM','h:mm a','HH:mm')` returns '**15:51**'.

7.2.1.18 YEAR

The **@YEAR** function returns the year portion of a date.

Return

The return type is `java.lang.String`.

Parameters

@YEAR takes one parameter:

Date	A date or date expression (MM/dd/yyyy).

Example

`@YEAR('01/28/1992')` returns **1992**.

7.2.2 String/Character Functions

String/character functions are covered in this section.

7.2.2.1 CONCAT

The **@CONCAT** function allows you to join together (concatenate) two or more strings.

Return

The return type is `java.lang.String`.

Parameters

@CONCAT takes from two to four parameters:

String1	A string to be concatenated.
String2	The number of years to add to parameter 1.
String3	The number of months to add to parameter 1.
String4	The number of days to add to parameter 1.

Parameters can be added in the "Function Editor" on page 119.

Example

`@CONCAT('J. Alfred ', 'Prufrock')` returns '**J. Alfred Prufrock**'.

`@CONCAT('The cow ', 'jumped', ' over', ' the moon')` returns '**The cow jumped over the moon**'.

7.2.2.2 DECIMAL FORMAT

The **@DECIMAL_FORMAT** system function changes the mask of a string such as a phone number, birth date, SIN, telephone or salary.

Return

The results are string of type java.lang.String.

Parameters

@DECIMAL_FORMAT takes two parameters:

Parameter1	The input string.
Parameter2	The mask to use.

Parameters can be added in the Function Editor.

Note: Extra characters that fall outside the mask will either be applied to the end or the beginning of the returned string.

Example

`@DECIMAL_FORMAT('123.4', '#.00')` returns **123.40**.

`@DECIMAL_FORMAT('123.40', '#.#')` returns **123.4**.

7.2.2.3 DELSTR

The **@DELSTR** function deletes characters from an alpha string. If the sub-string that is to be deleted is not found in the input, the entire input string is returned.

Return

The return type is `java.lang.String`.

Parameters

@DELSTR takes three parameters:

Alpha String	An alpha string or an alpha string expression.
Position of First Deleted Character	The position of the first character to be deleted. **Note:** Position numbering starts at 1.
Number of Deleted Characters	The number of characters to be deleted, beginning with Param2 and proceeding rightward.

Example

`@DELSTR('ABCD', '2', '1')` deletes the second letter of the string, and returns '**ACD**'.

`@IF(Y<'0',@DELSTR(X,'1','1'),@DELSTR(X,'2','1'))` If X contains a character string of length greater than or equal to 2, the expression removes either the first or second character, or leaves the string intact, according to the value that appears in column Y (negative, positive, or zero).

7.2.2.4 EQUALS

The **@EQUALS** function compares two strings and returns 'true' if they are equal, or 'false' if they are not equal.

Return

The return type is `java.lang.String`.

Parameters

@EQUALS takes two parameters:

String1	String.
String2	String.

Example

`@EQUALS('BA', 'AB')` return '**r;false**'.

`@EQUALS('Ab', 'AB')` return '**r;false**'.

7.2.2.5 FILL

The **@FILL** function repeats an alpha string or expression 'n' times.

Return

The return type is `java.lang.String`.

Parameters

@FILL takes two parameters:

Alpha String	An alpha string or expression.
Strings Repeated	The number of times the string will be repeated.

Example

`@FILL('*',5)` creates a string of five asterisks *****.

Note: This function will accept a maximum of 32k.

7.2.2.6 FLIP

The **@FLIP** function reverses an alpha string, or the result of an alpha expression, to its mirror image.

Return

The return type is `java.lang.String`.

Parameters

@FLIP takes one parameter:

Alpha String	An alpha string or alpha string expression.

Example

`@FLIP('Good')` returns '**dooG**'.

7.2.2.7 Format string

The **@FORMAT_STRING** function allows the user to determine how an input string is modified.

Return

The results are string of type java.lang.String.

Parameters

@FORMAT_STRING takes three parameters:

Input String	The input string.
Mask	The mask it should be applied to.
Boolean	Boolean to determine whether we are working front to back or back to front.

Parameters can be added in the Function Editor.

Note: Extra characters that fall outside the mask will either be applied to the end or the beginning of the returned string.

Example

```
@FORMAT_STRING('1234567890', '###-###-###', 'true')
```

will return **123-456-7890**.

```
@FORMAT_STRING('1234567890', '###-###-###', 'false')
```

will return **1234-567-890**.

7.2.2.8 HSTR

The **@HSTR** function returns the hexadecimal (base 16) string value of a decimal (base 10) number.

Return

The return type is `java.lang.String`.

Parameters

@HSTR takes one parameter:

Decimal Number	A decimal number (a base 10 number), or a numeric expression that represents a decimal number.

Example

`@HSTR('15')` returns '**F**'.

`@HSTR('16')` returns '**10**'.

7.2.2.9 HVAL

The **@HVAL** function returns the decimal (base 10) value of a hexadecimal (base 16) number.

Return

The return type is `java.lang.String`.

Parameters

@HVAL takes one parameter:

Alpha String	An alpha string that represents a hexadecimal (base 16) number.

Example

`@HVAL('FF')` returns **255**.

`@HVAL('10')` returns **16**.

7.2.2.10 INSERT

The **@INSERT** function inserts one string into another. An argument value error will be returned as the result if:

- The target position is out of range.
- The number of characters to be inserted is invalid.

Return

The return type is `java.lang.String`.

Parameters

@INSERT takes four parameters:

Target String	An alpha string that represents the target string.
Source String	An alpha string that represents the source string.

Parameter 1 Number	A number that represents the position in parameter 1 at which the second string will be inserted. **Note:** Position numbering starts at 0.
Number of Parameter 2 Characters	A number that represents the number of characters from parameter 2 that will be inserted into parameter 1.

Example

`@INSERT('abcde', 'xxx', '3', '2')` returns '**abcxxde**'.

7.2.2.11 INSTR

The **@INSTR** function returns a number of type string that represents the first position (counting starts from 0) of a sub-string within an alpha string or alpha expression.

Return

The return type is `java.lang.String`.

Parameters

@INSTR takes two parameters:

Alpha String	An alpha string or alpha expression.
Alpha String	An alpha string which will be the search argument in parameter 1.

Example

`@INSTR('abcd', 'b')` returns 1. `@INSTR('ABCDEF', 'DE')` returns **3**.

Note: If the search argument is not found, the function will return **-1**.

7.2.2.12 LEFT

The **@LEFT** function returns a specified number of characters from an alpha string, starting from the leftmost character. If the number of characters to be returned is out of range, an argument value error will be returned as the result.

Return

The return type is `java.lang.String`.

Parameters

@LEFT takes two parameters:

Alpha String	An alpha string.
Number of Returned Characters	The number of characters to be returned, starting from the leftmost character.

Example

`@LEFT('abcdefg', '3')` returns '**abc**'.

7.2.2.13 LEN

The **@LEN** function returns the defined length of an alpha string.

Return

The return type is `java.lang.String`.

Parameters

@LEN takes one parameter:

Alpha String	Input alpha string.

Example

`@LEN ('abcdefg')` returns **7**.

Note: The function will remove both leading and trailing blanks before commencing.

7.2.2.14 LOWER

The **@LOWER** function converts a string to all lower case.

Return

The return type is `java.lang.String`.

Parameters

@LOWER takes one parameter:

Alpha String	An alpha string.

Example

`@LOWER('Who said THAT?')` returns '**who said that?**'.

7.2.2.15 LTRIM

The **@LTRIM** function removes leading white space (blanks, tabs, linefeeds) from an alpha string or an alpha expression.

Return

The return type is `java.lang.String`.

Parameters

@LTRIM takes one parameter:

Alpha String	Input alpha string.

Example

`@LTRIM(' John')` returns '**John**'.

7.2.2.16 MID

The **@MID** function extracts a specified number of characters (a sub-string) from an alpha string. If the sub-string is not found or the sub-string's length is greater than the input string, the entire input string is returned.

Note: The @SUBSTR function performs the same task.

Return

The return type is `java.lang.String`.

Parameters

@MID takes three parameters:

Alpha String	Input alpha string.
Number of Starting Position	Number representing the starting position of the sub-string within parameter 1. **Note:** Position numbering starts at 1.
Number of Characters	Number of characters to be extracted (length of sub-string).

Example

`@MID('John', '3', '2')` returns '**hn**'.

Note: If the sub-string to be extracted is longer than the input string, or if the input string is NULL, @MID will simply return the entire input string. For example:
`@MID('abcde','3','5')` returns the entire input string '**abcde**'.
while:
`@MID('abcde','3','2')` returns '**cd**'.

7.2.2.17 NOT_EQUALS

The **@NOT_EQUALS** function compares two strings and returns 'false' if they're equal or 'true' if they're not.

Return

The return type is `java.lang.String`.

Parameters

@NOT_EQUALS takes two parameters:

String	String.
String	String.

Example

`@NOT_EQUALS('BA', 'AB')` return '**r;true**'.

7.2.2.18 QUOTEGEN

The **@QUOTEGEN** function is designed for use in generating 'Where' SQL statements. The function will generate single quotes around a string if it is of type varchar, but not if it is of type integer or number.

Return

The return type is `java.lang.String`.

Parameters

@QUOTEGEN takes one parameter:

String	String; can be an input item, a constant, or the result of another function.

Example

`@QUOTEGEN('HELLO')` will return '**HELLO**'.

`@QUOTEGEN('1234')` will return **1234**.

`@CONCAT('SELECT column1 from table where column2 = ', @QUOTEGEN(parent/child/value))` will return:

- **SELECT column1 from table where column2 = 'value', IF parent/child/value type is varchar**
- **SELECT column1 from table where column2 = value, IF parent/child/value type is integer or number**

7.2.2.19 REP

The **@REP** function replaces an alpha sub-string within a string with another sub-string. If the target position or the number of characters specified is invalid, an argument value error will be returned as the result.

Return

The return type is `java.lang.String`.

Parameters

@REP takes four parameters:

Alpha String	The target alpha string or expression where the replacement will take place.
Alpha String	The alpha string or expression that provides the sub-string to be copied to parameter 1.

Position in Parameter 1	The first position in parameter 1 that will receive the sub-string from parameter 2. **Note:** Position numbering starts at 1.
Number of Characters	The number of characters that will be moved from parameter 2 to parameter 1, starting from the leftmost character of parameter 2.

Example

`@REP('12345', 'abcde', '3', '2')` returns '**12ab5**'.

7.2.2.20 RIGHT

The **@RIGHT** function returns a specified number of characters from an alpha string, starting with the right-most character. The function will NOT right-justify the string (remove trailing blanks) before commencing. If the number of characters to be returned is out of range, an argument value error will be returned as the result

Return

The return type is `java.lang.String`.

Parameters

@RIGHT takes two parameters:

Alpha String	An alpha string from which the characters will be taken.
Number of Retrieved Characters	The number of characters to be retrieved, starting from the rightmost character.

Example

`@RIGHT('abcdefg ', '3')` returns '**fg** '.

7.2.2.21 RTRIM

The **@RTRIM** function removes trailing white space (blanks, tabs, linefeeds) from an alpha string or an alpha expression.

Return

The return type is `java.lang.String`.

Parameters

@RTRIM takes one parameter:

Alpha String	Input alpha string.

Example

`@RTRIM('John ')` returns '**John**'.

7.2.2.22 SINGLEQUOTE

The **@SINGLEQUOTE** function generates a single quote (') character.

Return

The return type is `java.lang.String`.

Parameters

None.

Example

`@CONCAT('It',@SINGLEQUOTE(),'s here.')` returns '**It's here**'.

7.2.2.23 STR

The **@STR** function converts a number to an alpha string according to a picture mask. For more information on picture masks, see “Picture masks” on page 180.

Return

The return type is `java.lang.String`.

Parameters

@STR takes two parameters:

Number	The number to be converted into an alpha string.
Picture Mask	A picture mask format for the string.
	Note: If parameter 2 is not a recognized picture mask, the result will have the value of parameter 1 appended to parameter 2.

Example

`@STR(45.12, ##.#)` returns '**45.1**'.

`@STR('12.56','¤#,0000.0#')` returns '**$0012.56**'.

`@STR('1234567.89','#,0000.0# m/s')` returns '**123,4567.89 m/s**'.

7.2.2.24 STRTOKEN

The **@STRTOKEN** function returns a string token from a delimited string. Special cases are:

- If the token index is 0 or a negative number, an empty string will be returned.
- If the token index is greater than the number of tokens created, the last token will be returned.
- If the delimiter was not found in the input string, the entire input string will be returned as the result.

Return

The return type is `java.lang.String`.

Parameters

@STRTOKEN takes three parameters:

Alpha String	Delimited alpha string with tokens.
Index	Requested token index (numeric).
Alpha String	Alpha string containing the delimiter string.

Remarks

- Parameter 3 can have more than one character as the delimiter.
- Every delimiter will be counted for the index calculation (no repetition).

Example

If the variable `BA = abcd/cdef/ghik/lmnp`, then `@STRTOKEN(BA, 2, /)` returns **cdef**.

7.2.2.25 SUBSTR

@SUBSTR has the same functionality as the "MID" on page 140 function.

7.2.2.26 TOSTR

The **@TOSTR** function converts any data type to string type.

Return

The return type is `java.lang.String`.

Parameters

@TOSTR takes one parameter:

Input Data	Input data to be converted.

Example

`@CONCAT('5<10<11=',@TOSTR(@RANGE('10','5','11')))` returns '**5<10<11=true**'.

7.2.2.27 TRIM

The **@TRIM** function removes leading and trailing white space (blanks, tabs, linefeeds) from an alpha string or an alpha expression.

Return

The return type is `java.lang.String`.

Parameters

@TRIM takes one parameter:

Alpha String	An input alpha string.

Example

`@TRIM(' John ')` returns '**John**'.

7.2.2.28 UPPER

The **@UPPER** function converts a string to all upper case.

Return

The return type is `java.lang.String`.

Parameters

@UPPER takes one parameter:

Alpha String	An alpha string.

Example

`@UPPER('Pablo Picasso')` returns '**PABLO PICASSO**'.

7.2.3 Numeric Functions

Numeric functions are covered in this section.

7.2.3.1 ADD

The **@ADD** function adds two numbers to return a number as an alpha string.

Return

The return type is `java.lang.String`.

Parameters

@ADD takes two parameters:

Float	**Float number.**
Float	Float number.

Example

`@ADD(24.01, 12.02)` returns **36.03**.

7.2.3.2 DIVIDE

The **@DIVIDE** function divides two numbers to return a string.

Return

The return type is `java.lang.String`.

Parameters

@DIVIDE takes two parameters:

Float	The dividend (a float).
Float	The divisor (a float).

Example

`@DIVIDE('24.02', '12.01')` returns **2.0**.

7.2.3.3 GREATER_THAN

The **@GREATER_THAN** function determines whether a numeric value, *number1*, is greater than a second numeric value, *number2*.

Return

The return type is `java.lang.String`.

Parameters

@GREATER_THAN takes two parameters:

Number1	This is the numeric value you wish to determine whether it is greater than another numeric value.
Number2	This is the numeric value you are comparing the first numeric value to.

Example

```
GREATER_THAN(number1, number2)
```

`@GREATER_THAN(10, 2)` returns '**true**'.

`@GREATER_THAN(2, 10)` returns '**false**'.

7.2.3.4 GREATER_THAN_OR_EQUAL_TO

The **@GREATER_THAN_OR_EQUAL_TO** function determines whether a numeric value, *number1*, is greater than or equal to a second numeric value, *number2*.

Return

The return type is `java.lang.String`.

Parameters

@GREATER_THAN_OR_EQUAL_TO takes two parameters:

Number1	This is the numeric value you wish to determine whether it is greater than or equal to another numeric value.
Number2	This is the numeric value you are comparing the first numeric value to.

Example

```
GREATER_THAN_OR_EQUAL_TO(number1, number2)
```

`@GREATER_THAN_OR_EQUAL_TO(10, 2)` returns '**true**'.

`@GREATER_THAN_OR_EQUAL_TO(10, 10)` returns '**true**'.

`@GREATER_THAN_OR_EQUAL_TO(2, 10)` returns '**false**'.

7.2.3.5 INTVAL

The **@INTVAL** function converts a numeric string to an integer value of type string.

Return

The return type is `java.lang.String`.

Parameters

@INTVAL takes one parameter:

Numeric String	The numeric string to be converted to an integer value.

Example

`@INTVAL('45')` returns **45**.

7.2.3.6 LESS_THAN

The **@LESS_THAN** function determines if a numeric value, *number1*, is less than a second numeric value, *number2*.

Return

The return type is `java.lang.String`.

Parameters

@LESS_THAN takes two parameters:

Number1	This is the numeric value you wish to determine if it is less than another numeric value.
Number2	This is the numeric value you are comparing the first numeric value to.

Example

```
LESS_THAN(number1, number2)
```

`@LESS_THAN(10, 2)` returns '**false**'.

`@LESS_THAN(2, 10)` returns '**true**'.

7.2.3.7 LESS_THAN_OR_EQUAL_TO

The **@LESS_THAN_OR_EQUAL_TO** function determines if a numeric value, *number1*, is less than or equal to a second numeric value, *number2*.

Return

The return type is `java.lang.String`.

Parameters

@LESS_THAN_OR_EQUAL_TO takes two parameters:

Number1	This is the numeric value you wish to determine if it is less than or equal to another numeric value.
Number2	This is the numeric value you are comparing the first numeric value to.

Example

`LESS_THAN_OR_EQUAL_TO(number1, number2)`

`@LESS_THAN_OR_EQUAL_TO(10, 2)` returns '**false**'.

`@LESS_THAN_OR_EQUAL_TO(10, 10)` returns '**true**'.

`@LESS_THAN_OR_EQUAL_TO(2, 10)` returns '**true**'.

7.2.3.8 MOD

The **@MOD** function returns a string as a result of performing a modulus operation on parameter one by parameter two.

Return

The return type is `java.lang.String`.

Parameters

@MOD takes two parameters:

String1	A java.lang.String type. The value must be a float or an integer.
String2	A java.lang.String type. The value must be a float or an integer.

Note: If the string “Five” is passed to @MOD, an error will result because even though it is a string type, the value contains letters and not numbers.

Example

`@MOD('5','2')` = **1**.

7.2.3.9 MULTIPLY

The **@MULTIPLY** function multiplies two numbers to return a string.

Return

The return type is `java.lang.String`.

Parameters

@MULTIPLY takes two parameters:

Float1	A float number.
Float2	A float number.

Example

`@MULTIPLY ('12.01', '2.00')` returns '**24.02**'.

7.2.3.10 NUM

The **@NUM** function converts an integer string to a numeric value of type string.

Return

The return type is `java.lang.String`.

Parameters

@NUM takes one parameter:

Numeric String	The numeric string to be converted to an integer value.

Example

`@NUM('45')` returns **45**.

Note: @NUM is a specialized variant of @VAL, where the input is integer and picture mask defaults accordingly to '#n' where n is the number of digits in the integer string.

7.2.3.11 NUM_CHR

The **@NUM_CHR** function returns the ASCII character that corresponds to a number. If the integer is not between 31 and 127, the input parameter is returned.

Return

The return type is `java.lang.String`.

Parameters

@NUM_CHR takes one parameter:

Integer Number	An integer number between 31 and 127 to represent an ASCII number.

Example

`@NUM_CHR ('r;66')` returns '**r;B**'.

7.2.3.12 RANDOM

The **@RANDOM** function returns a pseudo random number of type string.

Return

The return type is `java.lang.String`.

Parameters

@RANDOM takes one parameter:

Number	A number used to seed the random number generator.
Number1	If the number is -1, the seed is initialized randomly.
Number2	If the number is 0, the next random number is generated based on the existing seed.
Number3	If the number is an integer not equal to 0 or -1, the seed is initialized to this number.

After initializing the seed, continue with `@RANDOM('0')` to generate the next random number.

7.2.3.13 RANGE

The **@RANGE** function checks if a number falls within a range, and returns the string true or false.

Return

The return type is `java.lang.String`.

Parameters

@RANGE takes three parameters:

Number	The value checked.
Number	A value that represents the lower limit of the range.
Number	A value that represents the upper limit of the range. **Note:** The range is given by parameter 2 to parameter 3, inclusive.

Example

`@RANGE('10', '5', '15')` evaluates to **true**.

7.2.3.14 ROUND

The **@ROUND** function extracts a specified part of a number and rounds the result to an integer and returns it as a string.

Return

The return type is `java.lang.String`.

Parameters

@ROUND takes three parameters:

Number	The number subjected to the operation.
Number	The number of digits to be extracted from the integer part of parameter 1. Digits are counted from the left of the decimal separator.
Number	The number of digits to be extracted from the decimal part of parameter 1. Digits are counted from the right of the decimal separator.

Example

`@ROUND('345.995', '2', '2')` returns **46.0**.

7.2.3.15 SUBTRACT

The **@SUBTRACT** function subtracts two numbers, returning the remainder as a string.

Return

The return type is `java.lang.String`.

Parameters

@SUBTRACT takes two parameters:

Float1	A float number.
Float2	A float number.

Example

`@SUBTRACT('24.02', '11.01')` returns **13.01**.

7.2.3.16 VAL

The **@VAL** function converts an alpha string to a numeric value of type string according to a picture mask. If the given picture mask does not match the input string, a `java.text.ParseException` will be returned as the result. For more information about picture masks, see “Picture masks” on page 180

Return

The return type is `java.lang.String`.

Parameters

@VAL takes two parameters:

Alpha String	An alpha string to be converted to a numeric value.
Number Format	The format in which the number is stored in the string. The format must be specified as a constant within the “Function Editor” on page 119.

Example

`@VAL(45.12, ##.#)` returns **45.12**.

`@VAL('$123.24','$###.##')` returns **123.24**.

`@VAL('12.3E-4','#')` returns **0.00123**.

7.2.4 Processing Functions

Processing functions are covered in this section.

7.2.4.1 CONTROL_CHARACTER

The **@CONTROL_CHARACTER** inserts a special character into a string. It takes Java control characters, unicode and hex.

Return

The return type is `java.lang.String`.

Parameters

@CONTROL_CHARACTER takes one parameter:

Character	The character to use.

This function returns the value of the character.

Parameters can be added in the “Function Editor” on page 119.

Example

- `n` - **new line**
- `\t` - **tab**
- `\r` - **carriage return**
- `\b` - **backspace**
- `\f` - **form feed**
- `\u00A9` - **©**
- `#0x1c;0x0d;` - **carriage return line feed**

7.2.4.2 IF

The **@IF** function allows for conditional selection. If the condition is true, it will return the True option, otherwise it will return the False option.

Return

The return type is `java.lang.String`.

Parameters

@IF takes five parameters:

Operand	Left operand.
Operator	Conditional operator.
Operand	Right operand.
String1	The string to be returned if the condition is true.
String2	The string to be returned if the condition is false.

The possible arguments for the parameter two condition are:

Argument	**Meaning**
==	equal to
!=	not equal to
>=	greater than or equal to
<=	less than or equal to
>	greater than
<	less than
&&	and
||	or

Example

The variable X has a value which fluctuates between 4 and 8. `@IF(X,<,'10','r;ABC','r;DEF')` will always return '**r;ABC**'.

7.2.4.3 ISNULL

The **@ISNULL** function checks if the value of an item is null or not.

Return

The return type is `java.lang.String`.

Parameters

@ISNULL takes one parameter:

Item	The item whose value will be checked.

Example

`@ISNULL('NotNull')` returns '**false**'.

7.2.4.4 NULL

Returns a null value.

Parameters

None.

Example

`@NULL()` returns a null value.

7.2.4.5 REPLACE

The **@REPLACE** function calls replace functions or JDBC replace functions you have defined in the Configuration pane. See “Find & Replace tab” on page 26 or “JDBC Lookup Table tab” on page 32 for more information on either of these. Once @REPLACE has matched a key (values under the **Find What** column) from the replace function, it will replace all matching words in the input item with the mapped value (values under the **Replace With** column) and will not process the remaining keys.

Return

The return type is `java.lang.String`.

Parameters

@REPLACE takes two parameters:

Input Item	Input item to make changes in.

Replace Function	If the replace function you are using has been defined in the Find & Replace tab (a normal replace function), this parameter is the name of the replace tab with your find-replace pairs in it. The name of the tab is at the bottom of the pane. If you are using a JDBC replace function, this parameter is the ID given to your find-replace pair.

Parameters can be added in the "Function Editor" on page 119.

Example

If you have the replace function called "replace1":

Find What	Replace With
apple	red
plum	black

`@REPLACE('apple,plum,apple','replace1')` returns '**red,plum,red**'.

Note: If you have multiple entries in your Find and Replace table, only the first instance in the Replace function will return a new entry; the second instance will not return the replace instance matching it.

7.2.4.6 SEQUENCE

The **@SEQUENCE** function increments the input variable by one every time the function gets called. You should define the variable in the template before using @SEQUENCE. The first result of @SEQUENCE will be one greater than the initial variable value.

Return

The return type is `java.lang.String`.

Parameters

@SEQUENCE takes one parameter:

Input Variable	The variable to be incremented.

Example using predefined variable

For this example, assume you have defined a variable *Count*, in the "Variables tab" on page 22 of the "Settings window" on page 15.

To use the variable *Count* in your target with the @SEQUENCE function, it cannot be entered in the usual manner. The variable will not be used correctly if it is entered using the method explained in "Using a Variable in output" on page 23.

The variable **must** be manually entered by typing the variable name once the @SEQUENCE function has been added to the **Function Instance** pane of the "Function Editor" on page 119.

```
<mg:column mg:name="TransmisionNumber">
<mg:value-of mg:select="@SEQUENCE('Count')" />
</mg:column>
```

Note: Because the @SEQUENCE function can update the system variable that you pass in, you have to call `setProperties(Map map)` after every transformation to retain the original value. More details about this function are specified in OpenText Embedded Data Transformation Engine Javadoc .

7.2.4.7 SETVAR

The **@SETVAR** will set a variable.

Return

The return type is `java.lang.String`.

Parameters

@SETVAR takes two parameters:

varName	The name of the variable to set.
value	The value to set the variable to.

This function returns the value that you are setting the variable to.

Parameters can be added in the "Function Editor" on page 119.

Example

If you have a variable (`count`) set at the beginning of the transformation and you call `@SETVAR('count', '0')`, the value will be set to **0**.

7.2.4.8 SIMPLEREPLACE

The **@SIMPLEREPLACE** function calls replace functions or JDBC replace functions you have defined in the Configuration pane. See "Find & Replace tab" on page 26 or "JDBC Lookup Table tab" on page 32 for more information on either of these. @SIMPLEREPLACE needs to match the entire input item with a key (values under the **Find What** column) for a replacement to occur.

Return

The return type is `java.lang.String`.

Parameters

@SIMPLEREPLACE takes two parameters:

Input Item	Input item to make changes in.
Replace Function	If the replace function you are using has been defined in the Find & Replace tab (a normal replace function), this parameter is the name of the replace tab with your find-replace pairs in it. The name of the tab is at the bottom of the pane. If you are using a JDBC replace function, this parameter is the ID given to your find-replace pair.

Example

If you have the replace function called "replace1":

Find What	Replace With
apple	red

`@SIMPLEREPLACE('apple,plum,apple','replace1')` returns '**apple,plum,apple**'.

`@SIMPLEREPLACE('apple','replace1')` returns '**red**'.

7.2.4.9 THROW_EXCEPTION

The **@THROW_EXCEPTION** function causes an exception to be thrown. It can be used to record an instance to the log, or it can completely halt the transformation if used in conjunction with the ErrorHandler. The ErrorHandler is located in the "Target Configuration" on page 74 pane of the Settings dialog. For details on the ErrorHandler, see "Preferences tab" on page 19.

Return

The return type is `java.lang.String`.

Parameters

@THROW_EXCEPTION takes one optional parameter:

errorMsg	Indicates the error message to throw. If this parameter is not used, a generic message will be issued (i.e. Error thrown by THROW_EXCEPTION).

This function returns nothing; it throws an exception.

Examples

If ErrorHandler is set to **Blank**, the error message generated is always displayed in the Events window while the creation of an output file depends on the source of the job's execution. If you are running your job from Output Transformation Designer , the output file can be viewed on the **Results** sub-tab of the **Output** tab by right-clicking the file name under the **Result Map** entry. If you are running it from Transform, an output file is shown on the Target Results tab for you to check your results, but to receive an actual output file you must specifically indicate the creation of an output file in the **Result** field on the **Target Configuration** tab.

If ErrorHandler is set to **Stop Transform**, the transformation terminates resulting in no output files being generated, and error messages are written to the application log.

7.2.5 Security Functions

Security functions are covered in this section.

7.2.5.1 CHKDGT

The **@CHKDGT** function will check if a specified alphanumeric character is a number or a letter, and return **True** if it is a number, or **False** if it is a letter.

Return

The return type is `java.lang.String`.

Parameters

@CHKDGT takes two parameters:

Alpha String	An alpha string that represents the number to be checked.
Character Position	The position of the character in parameter 1 to check; the position numbering starts at 0, from the left. Enter as a constant within the "Function Editor" on page 119.

Example

`@CHKDGT('4abc', '0')` returns **true**.

`@CHKDGT('r;6a89', '1')` returns **false**.

7.2.6 Group Functions

Group functions are covered in this section.

7.2.6.1 AVG

The **@AVG** function will return a string of type `java.lang.String`, which is the average of the numerical values of a specified input item.

Note: The input of this function is not `java.lang.String.class`, or any `java.lang` type. The input type is specific to Data Transformation Engine, so you cannot simply take the result of another function and pass it as input to @AVG. To specify the input, you must use the Function Editor and drag the elements across.

Return

The return type is `java.lang.String`.

Parameters

@AVG takes two parameters:

Number	An input item that has a numeric value.
Null Value	**true** if you would like null values to be counted as '0' (zero). **false** if you would like null values ignored.

Note: The looping settings of the parent and grandparent of the attribute or element that uses the @AVG function must be carefully set.

Example

If you have the following data:

```
<root>
<products>
<product>
<name>Product A</name>
<price>10.00</price>
</product>
<product>
<name>Product B</name>
<price>5.00</price>
</product>
</products>
</root>
```

then,

`@AVG(/root/products/price, 'true')` returns **7.50**.

7.2.6.2 COUNT

The **@COUNT** function returns a string of type `java.lang.String`, which is the number of instances of a particular input item.

Note: The input of this function is not `java.lang.String.class`, or any `java.lang` type. The input type is specific to Data Transformation Engine, so you cannot simply take the result of another function and pass it as input to @COUNT. To specify the input, you must use the Function Editor and drag the elements across.

Return

The return type is `java.lang.String`.

Parameters

@COUNT takes two parameters:

Number	The input item to be counted.
Null Value	**true** if you would like null values to be counted. **false** if you would like null values ignored.

Example

If you have the following data:

```
<root>
<products>
<product>
<name>Product A</name>
<price>10.00</price>
</product>
<product>
<name>Product B</name>
<price>5.00</price>
</product>
</products>
</root>
```

then,

`@COUNT(/root/products/price, 'true')` returns **2**.

7.2.6.3 COUNTING

The **@COUNTING** function returns a string of type `java.lang.String`, which is the number of input fields.

Note: The input of this function is not `java.lang.String.class`, or any `java.lang` type. The input type is specific to Data Transformation Engine, so you cannot simply take the result of another function and pass it as input to @COUNTING. To specify the input, you must use the “Function Editor” on page 119 and drag the elements across.

Return

The return type is `java.lang.String`.

Parameters

@COUNTING takes two parameters:

Number	An input item.
Null Value	**true** if you would like null values to be counted. **false** if you would like null values ignored.

Example

If you have the following data:

```
<root>
<products>
<product>
<name>Product A</name>
<price>10.00</price>
</product>
<product>
<name>Product B</name>
<price>5.00</price>
</product>
</products>
</root>
```

then,

`@COUNTING(/root/products/price, 'true')` returns **2**.

7.2.6.4 MAX

The **@MAX** function returns a string of type `java.lang.String`, which is the maximum value of a particular input item.

Note: The input of this function is not `java.lang.String.class`, or any `java.lang` type. The input type is specific to Data Transformation Engine, so you cannot simply take the result of another function and pass it as input to @MAX. To specify the input, you must use the Function Editor and drag the elements across.

Return

The return type is `java.lang.String`.

Parameters

@MAX takes two parameters:

Number	An input item.
Null Value	**true** if you would like null values to be counted as '0' (zero). **false** if you would like null values ignored.

Example

If you have the following data:

```
<root>
<products>
<product>
<name>Product A</name>
<price>10.00</price>
</product>
<product>
<name>Product B</name>
<price>5.00</price>
</product>
</products>
</root>
```

then,

`@MAX(/root/products/price, 'true')` returns **10.00**.

7.2.6.5 MIN

The **@MIN** function returns a string of type `java.lang.String`, which is the minimum value of a particular input item.

Note: The input of this function is not `java.lang.String.class`, or any `java.lang` type. The input type is specific to Data Transformation Engine, so you cannot simply take the result of another function and pass it as input to @MIN. To specify the input, you must use the Function Editor and drag the elements across.

Return

The return type is `java.lang.String`.

Parameters

@MIN takes two parameters:

Number	An input item.
Null Value	**true** if you would like null values to be counted as '0' (zero). **false** if you would like null values ignored.

Example

If you have the following data:

```
<root>
<products>
<product>
<name>Product A</name>
<price>10.00</price>
</product>
<product>
<name>Product B</name>
<price>5.00</price>
</product>
</products>
</root>
```

then,

`@MIN(/root/products/price, 'true')` returns **5.00**.

7.2.6.6 SUM

The **@SUM** function returns a string of type `java.lang.String`, which is the sum of all values of a particular input item.

Note: The input of this function is not `java.lang.String.class`, or any `java.lang` type. The input type is specific to Data Transformation Engine, so you cannot simply take the result of another function and pass it as input to @SUM. To specify the input, you must use the Function Editor and drag the elements across.

Return

The return type is `java.lang.String`.

Parameters

@SUM takes two parameters:

Number	An input item.
Null Value	**true** if you would like null values to be counted as '0' (zero). **false** if you would like null values ignored.

Example

If you have the following data:

```
 <root>
<products>
 <product>
 <name>Product A</name>
 <price>10.00</price>
 </product>
 <product>
 <name>Product B</name>
 <price>5.00</price>
 </product>
 </products>
 </root>
```

then,

`@SUM(/root/products/price, 'true')` returns **15.00**.

7.2.7 EDI Functions

EDI functions are covered in this section.

7.2.7.1 COUNT_GROUP

The **@COUNT_GROUP** function returns a string of type java.lang.String, which is the number of groups within your EDI output transaction.

Note: The input of this function is not java.lang.String.class, or any java.lang type. The input type is specific to the Data Transformation Engine , so you cannot simply take the result of another function and pass it as input to @COUNT_GROUP. To specify the input, you must use the “Function Editor” on page 119 and drag the elements across.

Return

The return type is `java.lang.String`.

Parameters

@COUNT_GROUP takes one parameter:

String	The minimum output length.

Example

If there are 25 groups, `@COUNT_GROUP('5')` returns **00025**.

Note: @COUNT_GROUP should only be used for EDI output transactions.

7.2.7.2 COUNT_SEGMENT

The **@COUNT_SEGMENT** function returns a string of type java.lang.String, which is the number of segments within your EDI output transaction.

Note: The input of this function is not java.lang.String.class, or any java.lang type. The input type is specific to Data Transformation Engine, so you cannot simply take the result of another function and pass it as input to @COUNT_SEGMENT. To specify the input, you must use the “Function Editor” on page 119 and drag the elements across.

Return

The return type is `java.lang.String`.

Parameters

@COUNT_SEGMENT takes one parameter:

String	The minimum output length.

Example

If there are 15 segments, `@COUNT_SEGMENT('3')` returns **015**.

Note: @COUNT_SEGMENT should only be used for EDI output transactions.

7.2.7.3 COUNT_TRANSACTION

The **@COUNT_TRANSACTION** function returns a string of type java.lang.String, which is the number of transactions within your EDI output transaction.

Note: The input of this function is not java.lang.String.class, or any java.lang type. The input type is specific to Data Transformation Engine, so you cannot simply take the result of another function and pass it as input to @COUNT_TRANSACTION. To specify the input, you must use the Function Editor and drag the elements across.

Return

The return type is `java.lang.String`.

Parameters

@COUNT_TRANSACTION takes one parameter:

String	The minimum output length.

Parameters can be added in the "Function Editor" on page 119.

Example

If there are three transactions, `@COUNT_TRANSACTION('2')` returns **03**.

Note: @COUNT_TRANSACTION should only be used for EDI output transactions.

7.2.8 DOM Functions

DOM functions are covered in this section.

7.2.8.1 NODE

The **@NODE** function returns a node from the specified XPath. Since the return value is a node and not a string, you will not be able to map @NODE directly to an output element. In order to use @NODE, you will need to write a custom function that will take in a node, do some processing with the node and return a string.

Return

@NODE's input and output are of type `org.w3c.dom`.

Parameters

@NODE takes one parameter:

Node	The XPath that points to a node.

Example

`@NODE(/A/B)` returns the node **/A/B**.

7.2.8.2 NODES

The **@NODES** function returns a list of nodes from the specified XPath. Since the return value is a list of nodes and not a string, you will not be able to map @NODES directly to an output element. In order to use @NODES, you will need to write a custom function that will take in a list of nodes, do some processing and return a string.

Return

@NODES' input and output are of type `java.util.List` containing Nodes of type `org.w3c.dom.Node`.

Parameters

@NODES takes one parameter:

Nodes	The XPath that points to a list of nodes.

Example

If parent element A contains the list of children B, `@NODES(/A/B)` returns the **node list of elements B**.

7.2.8.3 XPath

The **@XPath** function performs the XPath expression on the specified node and function returns a string of type `java.lang.String`. To return multiple results, use the Match on page 112 feature.

Note: The input of this function is not `java.lang.String.class`, or any java.lang type. The input type is specific to Data Transformation Engine, so you cannot simply take the result of another function and pass it as input to @XPath. To specify the input, you must use the Function Editor and drag the elements across.

Return

The results are string of type `java.lang.String`.

Parameters

@XPath takes three parameters:

Nodes	The XPath that contains the list of nodes.
Expression	The XPath expression.
Node	A node from which prefixes in the XPath will be resolved to namespaces or null if the XPath does not contain any prefixes.

Parameters can be added in the “Function Editor” on page 119.

Example 7-1:

If you have the structure:

```
<Root xmlns:c="http://c">
<A>
<c:B>1</c:B>
<c:B>5</c:B>
</A>
</Root>
```

then `@XPath(/Root/A,'c:B[text()>1]',@NSNode())` returns '**5**'.

@NSNode() is a custom function that you can create that will return a node with the namespace information. An example implementation of @NSNode() can be:

```
import org.w3c.dom.*;
import org.apache.xerces.dom.CoreDocumentImpl;
import com.xenos.transform.kernel.function.*;
```

```
public class NSNode extends AbstractFunction
{
private static final String DEFAULT_DESCRIPTION =
"get namespace node";
private static final String DESCRIPTION_KEY =
"function.NSNode.description";
Document doc;
public NSNode()
{
setName("NSNode");
setDescription(DESCRIPTION_KEY, DEFAULT_DESCRIPTION);
doc = new CoreDocumentImpl();
}
public Object execute() throws Exception
{
/* create a new element, nstag */
Element nstag = doc.createElement("nstag");
/* declare the namespace for the "c" prefix */
nstag.setAttribute("xmlns:c", "http://c");
/* return the element */
return nstag;
}
public Class getReturnType()
{
return org.w3c.dom.Element.class;
}
}
```

7.2.8.4 XPathEval

The **@XPathEval** function evaluates an XPath expression. @XPathEval lets you use functions in the XPath Function Library such as count(), concat(), and etc. in your transformations. The results are returned as a string of type `java.lang.String`.

Note: The input of this function is not `java.lang.String.class`, or any `java.lang` type. The input type is specific to Data Transformation Engine, so you cannot simply take the result of another function and pass it as input to @XPathEval. To specify the input, you must use the Function Editor and drag the elements across.

Return

The results are string of type `java.lang.String`.

Parameters

@XPathEval takes three parameters:

Nodes	The XPath that contains the list of nodes.
Expression	The XPath expression.
Node	A node from which prefixes in the XPath will be resolved to namespaces or null if the XPath does not contain any prefixes.

Parameters can be added in the “Function Editor” on page 119.

Example

If you have the structure:

```
<Root xmlns:c="http://c">
<A>
<c:B>1</c:B>
<c:B>5</c:B>
</A>
</Root>
```

then `@XPathEval(/Root/A,'count(c:B)',@NSNode())` returns '**2**'.

```
import org.w3c.dom.*;
import org.apache.xerces.dom.CoreDocumentImpl;
import com.xenos.transform.kernel.function.*;
public class NSNode extends AbstractFunction
{
```

```
private static final String DEFAULT_DESCRIPTION =
"get namespace node";
private static final String DESCRIPTION_KEY =
"function.NSNode.description";
Document doc;
public NSNode()
{
setName("NSNode");
setDescription(DESCRIPTION_KEY, DEFAULT_DESCRIPTION);
doc = new CoreDocumentImpl();
}
public Object execute() throws Exception
{
/* create a new element, nstag */
Element nstag = doc.createElement("nstag");
/* declare the namespace for the "c" prefix */
nstag.setAttribute("xmlns:c", "http://c");
/* return the element */
return nstag;
}
public Class getReturnType()
{
return org.w3c.dom.Element.class;
}
}
```

7.2.9 Database Functions

Database functions are covered in this section.

7.2.9.1 EXECUTE_QUERY

The **@EXECUTE_QUERY** function will run a query on a database.

Return

The return type is `java.lang.String`.

Parameters

@EXECUTE_QUERY takes three parameters:

db conn	The database connection ID.
query	The query to run.
reuse	Boolean for whether you want the query to be cached for quicker processing.

This function returns a string value.

Parameters can be added in the "Function Editor" on page 119.

Example

A table called fruit:

ID	Name
1	apple
2	orange
3	cherry

```
@EXECUTE_QUERY(CONN_1', 'SELECT name FROM fruit WHERE ID = 1')
```

will return '**apple**'.

7.2.9.2 EXECUTE_STORED_FUNC

The **@EXECUTE_STORED_FUNC** function will run a stored function from within a database.

Return

The results are string of type java.lang.String.

Parameters

@EXECUTE_STORED_FUNC takes two mandatory parameters:

db conn	The database connection ID.

FunctionName	The function name.

@EXECUTE_STORED_FUNC takes unlimited optional fields which act as parameters on the stored function within the database.

Parameters can be added in the “Function Editor” on page 119.

Example

A function in the database called concat that concatenates two varchars as parameters:

```
@STORED_FUNC('CONN_1', 'dbo.concat', '123', '456')
```

will return '**123456**'.

7.2.9.3 FIELD_EXISTS

The **@FIELD_EXISTS** function checks to see if a value exists in a database table.

Return

The return type is `java.lang.String`.

Parameters

@FIELD_EXISTS takes four parameters:

FieldName	The field in the database to lookup.
TableName	The table in the database to query.
Value	The value in the database to check for.
db conn	The database connection ID.

This function returns a boolean value.

Parameters can be added in the “Function Editor” on page 119.

Example

A table called fruit:

ID	Name
1	apple
2	orange
3	cherry

`@Field_Exists('name', 'fruit', 'orange', 'CONN_1')` will return **true**.

7.2.9.4 ORASEQUENCE

The **@ORASEQUENCE** function returns the string representation of Oracle's incremented sequence number.

```
@ORASEQUENCE(sequence_name, variable_name, JDBC_Connection_ID)
```

Note: Oracle's sequence does not guarantee a gap-free sequence number.

Return

The return type is java.lang.String.

Parameters

@ORASEQUENCE takes three parameters:

Sequence_name	The sequence name created in Oracle, such as `Individual_seq` and `Address_seq`. **Note:** If this is a number, then no error is thrown and the resulting value is the number itself.
Variable_name	The variable name created in the Data Transformation Engine template, usually used in children table as a foreign key value, such as `Individual_variable`. The Oracle sequence number will be stored in this variable.
JDBC_Connection_ID	The database connection defined in the Data Transformation Engine template, such as `CONN_1`.

Example

Suppose there's an Oracle sequence in JDBC connection CONN_1 called SEQ_ID that is declared as:

```
CREATE SEQUENCE SEQ_ID
MINVALUE 1
MAXVALUE 999999999999999999
START WITH 1
INCREMENT BY 1
NOCACHE
;
```

Assume there is no data for the sequence in the database, then:

`@ORASEQUENCE('SEQ_ID','PrimaryKey','CONN_1')` assigns the value 1 to the variable PrimaryKey and returns the number **1**.

7.2.9.5 ORASTOREDFUNC

The **@ORASTOREDFUNC** function will run a function in the database and return the returned value.

```
@ORASEQUENCE(JDBC_Connection_ID,Function_name, Parameter)
```

Return

The return type is java.lang.String.

Parameters

@ORASTOREDFUNC takes three parameters:

JDBC_Connection_ID	The database connection defined in the Data Transformation Engine template.
Function_name	The function name created in the database.
Parameter	The parameter to be used in the function. The number of parameters depends on the called function in the database.

Parameters can be added in the "Function Editor" on page 119.

Example

`ORASTOREDFUNC ('conn_1','msg_exists', '1', )` will return the value from the function **msg_exists** given the input param of 1.

The executed SQL query takes the form:

```
select functionName (parameter) from dual;
```

7.2.9.6 RUN_FUNCTION

The **@RUN_FUNCTION** provides the ability to run a function within a database.

Return

The results are string of type java.lang.String.

Parameters

@RUN_FUNCTION takes two mandatory parameters:

db conn	The database connection ID.
FunctionName	The function name.

@RUN_FUNCTION takes unlimited optional parameters where:

db conn	Parameter 1 is the connection ID.
FunctionName	Parameter 2 is the function name.
Parameter	Parameter 3 and greater are the required parameters for the function in the database.

The return value will be from the first result and first column of the query.

Parameters can be added in the "Function Editor" on page 119.

7.2.9.7 RUN_QUERY

The **@RUN_QUERY** function will run a query on a database.

Return

The return type is `java.lang.String`.

Parameters

@RUN_QUERY takes two parameters:

db conn	The database connection ID.
query	The query to run.

This function returns a string value.

Parameters can be added in the "Function Editor" on page 119.

Example

A table called fruit:

ID	Name
1	apple
2	orange
3	cherry

`@RUN_QUERY(CONN_1', 'SELECT name FROM fruit WHERE ID = 1')` will return '**apple**'.

7.2.9.8 STORED_FUNC

The **@STORED_FUNC** function will run a stored function from within a database.

Return

The return type is `java.lang.String`.

Parameters

@STORED_FUNC takes two mandatory parameters:

db conn	The database connection to use.
FunctionName	The name of the function to use in the database.

@STORED_FUNC takes unlimited optional parameters where parameter 1 is the connection ID, parameter 2 is the function name and parameters 3 and greater are the value represented by the parameters of the function.

This function returns a string value.

Parameters can be added in the "Function Editor" on page 119.

Example

A function in the database called concat that concatenates two varchars as parameters:

```
@STORED_FUNC('CONN_1', 'dbo.concat', '123', '456')
```

will return '**123456**'.

7.2.10 Picture masks

Picture masks are categorized by the basic data-typing element with which they can be used in combination. Each picture can be used vice-versa; for example, text->number or number->text.

Note: Special characters for picture masks are case sensitive.

7.2.10.1 Numeric pictures

The positional directives and mask characters for numeric pictures are as follows:

Positional Directives	Mask Characters
0	A digit (0-9 only), fills with leading zeroes to meet the minimum length.
#	A digit (0-9 only), trailing and leading zeros shown as absent, rounds the decimal part of the number to meet the maximum length.
.	Indicates the location of the decimal point. For example, '0000.000' defines a numeric variable of four whole digits and three decimal digits.
-	Minus sign.
,	Used as a grouping separator. If you have irregular groupings, the grouping that is closest to the decimal point is used. For example, if 100000000 is applied with #,##,### the result is 1,000,000 which has the grouping of 3.
E	Used to express scientific notation. If the integer part of the mask are all zeroes, the resulting mantissa will have the same number of digits as the integer part of the mask. For example, a numerical value, -123.45 with a mask of 00000.#E0 will give the result of -12345E-2. Else, the exponent will be a multiple of the length of the mask's integer part. For example, -12345.67 with a mask of ##0.#E0 will give the result of -12.35E3.
%	Multiplies the numeric value by 100 and adds the % sign in front.
¤	(Alt+0164) currency sign, replaced by currency symbol ($). If doubled, replaced by international currency symbol. If present in a pattern, the monetary decimal separator is used instead of the decimal separator.
'	Used as an escape character for prefixes and suffixes. Must be used in pairs. Double quotes ('') produces a single quote character in the output.

Examples of numeric pictures are shown in the following table:

Numeric Value	Picture	Resulting Numeric Value

-1234.56	######.##	-1234.56
-1234.56	000000.##	-001234.56
-1234.56	N######.##	-N1234.56
-1234.56	##,##.##	-12,34.56
-1234.56	#,#,##.#	-12,34.6
-1234567.89	###.###E0	-1.23457E6
0.12345	%00.00	%12.34
1234.56	¤	$1235
1234.56	¤¤0.00	USD1234.56
12345	'#¤'ABC#0	#¤ABC12345
12 o'clock	00o''clock	12o'clock

7.2.10.2 Date and Time pictures

The typical date formats are dd/MM/yyyy (European), MM/dd/yyyy (American), or yyyy/MM/dd (Scandinavian). The common time format is HH:mm:ss for an element of datatype Time. When you define the attribute Date/Time for a variable, you must also select the format for the date/time item (see below). You can change this default picture and place in it any positional directives and mask characters you need.

The positional directives and mask characters for numeric pictures are as follows:

Positional Directives	**Mask Characters**
G	Era designator, for example, AD
y	Year, for example, 1996; 96
M	Month in year, for example, July; Jul; 07
w	Week in year, for example, 27
W	Week in month, for example, 2
D	Day in year, for example, 189
d	Day in month, for example, 10
F	Day of week in month, for example, 2
E	Day in week, for example, Tuesday; Tue
a	a.m./p.m. marker, for example, PM
H or **k**	Hour in day (0-23), for example, 0
h or **K**	Hour in a.m./p.m. (1-12), for example, 12
m	Minute in hour, for example, 30
s or **S**	Second in minute, for example, 55

z	General time zone, for example, Pacific Standard Time; PST; GMT-08:00
Z	RFC 822 time zone, for example, -0800 **Note:** Only available in JDK 1.4.2 or higher.

Examples of date pictures are shown in the following table, using the date of 21 March 1992 and time 8:20:55 PM PDT:

The positional directives and mask characters for numeric pictures are as follows:

Picture	**Result and Notes**
MM/dd/yyyy	03/21/1992
EEEEE, MMMMM dd, ''yy G	Saturday, March 21, '92 AD
'WeekInMonth is ' W ', WeekInYear is ' w	Week In Month is 3, Week In Year is 12
EEE, F, DDD	Sat, 3, 081 where 3 means the 3rd week of March and 081 means the 81st day of the year
h:mm a	8:20 PM
HH:mm:ss Z	20:20:55 -0700
kk-mm-SS z	20-20-55 PDT

Chapter 8

Custom components

This section describes how you can create Data Transformation Engine custom components, such as custom preparsers and custom functions.

8.1 Custom preparsers

When a Data Transformation Engine pre-defined preparser does not exist to perform the task you need, you may want to write your own custom preparser. Custom preparsers must be written in Java and compiled to create a `.class` file. See "Compiling your Java file" on page 202 for more information. It is recommended that you store custom preparsers within the `//external_preparsers` directory because the locations are coded within Data Transformation Engine to look for them there or the classpath. For example:

```
<install_home>\dev-studio\external_preparsers
```

If you do not do so, you must add the location of your custom preparser to the classpath used by Data Transformation Engine . Custom preparsers should be tested by their creator and proven to work properly before attempting their use within Data Transformation Engine.

8.1.1 Writing custom preparsers

Note: The following instructions assume that you are familiar with the Java programming language and have a basic understanding of SAX events.

The custom preparser you design will be an implementation of the abstract class Preparser. You are required to implement the abstract method `parse()` that is declared in Preparser. You may also choose to add your own methods. The Java code for your custom preparser will have a similar format to the following:

```
import java.util.List;
import org.jdom.*;
import java.io.IOException;
import com.xenos.transform.Preparser;
import com.xenos.transform.CascadeException;
public class MY_PREPARSER_NAME extends Preparser {
public void parse() throws IOException, CascadeException {
// parsing goes here {
}
```

Compiled custom preparser Java classes **must** be placed under the `//external_preparsers` directory.

Note: See the following directory for an example of a preparser: `<install_home>\initialFiles\common_sample\DataTransformation\embed_sample`

8.1.1.1 Import statement

You must have the following statement as the top line of your custom preparser implementation:

```
import com.xenos.transform.Preparser;
```

This will allow you access to methods that you will need in your preparser's implementation.

8.1.2 Class declaration

As stated earlier, a custom preparser is an implementation of the abstract class `Preparser`. Therefore, your custom preparser must be declared as the following:

```
public class MY_PREPARSER_NAME extends Preparser {
...
}
parse()
```

The method `parse()` is the only abstract method declared in the `Preparser` class. Your custom preparser, an implementation of Preparser, must therefore implement `parse()` as follows:

```
public void parse() throws IOException, CascadeException {
//reading data
//data manipulation
//sending data
{
```

The body of this method is written as you would write any method in Java. Preparsing mainly involves three steps: reading data from the Data Transformation Engine run-time engine, manipulating data, and sending data back to the Data Transformation Engine run-time engine.

Compiled custom preparser java classes must be placed under the `\external_preparsers` directory.

8.1.3 Reading, manipulating, and sending data

8.1.3.1 Reading data

Reading input data can be accomplished with the help of the parsers that are already initialized by Data Transformation Engine . You can use a parser to iterate through the data like:

```
if (parsers[i].hasNext()) {
Element element = (Element)parsers[i].next();
…
}
```

Where:

- parsers[i] is the ith parser that you wish to use.
- parsers[i].next() returns an org.jdom.Element, which is defined through dictionaries such as XSD, DTD, and Flat Text.

Note: Simple Merge multiple inputs only support JDBC, CSV, Flat Text, and COBOL. The `hasNext()` type methods are not available when writing custom preparsers for EDI/HL7 and SWIFT data formats.

8.1.3.2 Manipulating data

Manipulating data will vary from user to user. The Preparser abstract class provides some helpful methods for easier data manipulation:

- `getFormats()` returns a list of preparser formats
- `getParseNum()` returns the number of source inputs with unique source IDs
- `setErrorHandler(ErrorHandler errorHandler)` sets up errorHandler to all the parsers.

Refer to OpenText Embedded Data Transformation Engine Javadoc for more details.

8.1.3.3 Sending data

After reconstructing the data, you can send the data to Data Transformation Engine by calling the `element2SAX(org.jdom.Element element)` or `elements2SAX(List elements)` methods, where element and elements contain the newly reconstructed data.

8.1.4 Customized parameters

You can create your customized parameters by calling the `getParameters(String format)` method. Data Transformation Engine will call this method when loading your custom preparser during runtime. You can add customized parameters like:

```
Map map = new java.util.HashMap();

map.put("PicPath", new Integer(FILE_PARAM));

map.put("PerfectPrefix", new Integer(STRING_PARAM));

map.put("NodeSelected", new Integer(NODE_PARAM));

map.put("HowManyNights", new Integer(INT_PARAM));

return map;
```

Where **map** contains the name of customized parameters as follows:

```
# "r;PicPath"
# "r;PerfectPrefix"
# "r;NodeSelected"
# "r;HowManyNights"
```

Note: The following case insensitive names are reserved for Data Transformation Engine use only and should not be used for custom preparsers: **fileName**, **LoadFiller**, **IsCOBOL85FreeFormat**, **structure**, **CopybookFormatType**, **id**, **format**, **delimiter**, **query**, **table**, **dbConnector**.

With their respected field types, as follows:

```
* "r;FILE_PARAM"
* "r;STRING_PARAM"
* "r;NODE_PARAM"
* "r;INT_PARAM"
```

To retrieve the parameter's value, override the method similar to the following example:

```
public Element getInputsStructure(Map[] sources)
throws CascadeException, Exception {
inputStructure = new Element(ROOT);
int size = sources.length;
for (int n = 0; n < size; n++) {
....
Object paramValue = (Object)sources[n].get(paramName);
.....
```

```
}
return inputStructure;
}
```

8.2 Custom functions

When a Data Transformation Engine pre-defined function does not exist to perform the task you need, you may want to write your own custom function. Custom functions must be written in Java and compiled to create a `.class` file. See "Compiling your Java file" on page 202 for more information. Output Transformation Designer includes two wizards to help you write your own custom function.

It is recommended that you store custom functions within the `//external_functions` directory because the locations are coded within Data Transformation Engine to look for them there or the classpath. For example:

`<installLocation>\dev-studio\external_functions`.

If you do not do so, you must add the location of your custom function to the classpath used by Data Transformation Engine . Custom functions should be tested by their creator and proven to work properly, before attempting their use within Data Transformation Engine .

For more information on the Data Transformation Engine classpath, see *OpenText Embedded Data Transformation Engine - Developer's Guide (VDTOTS-H-PDT)* in OpenText Embedded Data Transformation Engine Developer's Guide.

8.2.1 Defining custom functions

Defining a custom function in Data Transformation Engine is equivalent to creating a pointer to the actual custom function stored in the `//external_functions` directory. Custom functions are defined through the "Custom functions" on page 189 tab in the "Settings window" on page 15.

When defining your custom function on the **Custom Functions** tab, you must provide the following information:

- **Name.** The class name of the custom function. For example, if your custom function was the one used in "Sample custom function" on page 193, this value would be **ADD25**. This name is used in the "Function Editor" on page 119.
- **#Args.** The number of parameters that the function accepts.
- **Function Location.** The fully-qualified Java name of the custom function. For example, you might enter `com.myCompany.mySoftware.MyClass`.

Once you have completed the required information, your custom function may be used to assign an output item value.

Note: If you are running multiple instances of Data Transformation Engine within your JVM, you may have to watch for conflicting user-defined function names. See *OpenText Embedded Data Transformation Engine - Developer's Guide (VDTOTS-H-PDT)* for information.

8.2.2 Writing custom functions

Note: The following instructions assume that you are familiar with the Java programming language.

The custom function you design will be a subclass of the superclass `AbstractFunction`. You are required to implement a constructor for your class and the two abstract methods `execute()` and `getReturnType()` that are declared in `AbstractFunction`. You may also choose to add your own methods. The Java code for your custom function will have a similar format to the following:

```
import com.xenos.transform.kernel.function.*;
public class MY_FUNCTION_NAME extends AbstractFunction {
public MY_FUNCTION_NAME() {
setName("MY_FUNCTION_NAME");
setDescription("This is what my function does...");
}
protected Object execute() throws Exception {
// perform function's execution here
// and return desired output value
}
public Class getReturnType() {
try
{
return Class.forName("MY_RETURN_TYPE");
}
catch (ClassNotFoundException cnfe)
{
return null;
}
}
}
```

8.2.2.1 Import statement

You must have the following statement as the top line of your custom function implementation:

```
import com.xenos.transform.kernel.function.*;
```

This will allow you access to methods that you will need in your function's implementation, described below.

8.2.2.2 Class declaration

As stated earlier, a custom function is a subclass of the class `AbstractFunction`. Therefore, your custom function must be declared as the following:

```
public class MY_FUNCTION_NAME extends AbstractFunction {
...
}
```

8.2.2.3 Constructor

The only methods you need to call within your constructor are `setName(String name)`, and `setDescription(String description)` which have been pre-defined in the `AbstractFunction` class. The values you enter as the parameters in these methods are only used when you access your custom function through Data Transformation Engine . The function name used in Data Transformation Engine will be the parameter that you set here in the `setName` method, and the function description used by Data Transformation Engine will be the message you enter as the parameter in `setDescription`.

8.2.2.4 execute()

The method `execute()` is one of two abstract methods declared in the `AbstractFunction` class. Your custom function, a subclass of `AbstractFunction`, must therefore implement this method. `execute()` is declared as follows:

```
protected Object execute() throws Exception {
...
}
```

The body of this method is written as you would write any method in Java.

8.2.2.5 Getting arguments

Retrieving arguments passed to your custom function is done via the methods `getValueInstance()`, `getArgument(int index)`, and `getFunctionContext()` which are pre-defined in the functions package that your custom function imported. To store two arguments in String variables `argOne` and `argTwo`, you would write the following:

```
String argOne =
 (String)getFunctionContext().getArgument(0).getValueInstance();
```

```
String argTwo =
 (String)getFunctionContext().getArgument(1).getValueInstance();
```

Other methods that you may find useful in your implementation are `getNumberOfArguments()` and `getArguments()`. The first returns an integer indicating the number of arguments passed to your function by the user, and is used in conjunction with method `getFunctionContext()` as follows:

```
int num = getFunctionContext().getNumberOfArguments();
```

`getArguments()` returns an array containing all of the arguments passed to your function by the user. To store your arguments in an array of Objects called `myArray`, you would write the following:

```
Object[] myArray = getFunctionContext().getArguments();
```

8.2.2.6 Controlling project variables

Controlling project variables, variables defined in the Configuration pane, is made possible by the methods:

`getVariable(String variableName`

`setVariable(String variableName, String newValue)`

These methods are pre-defined in the functions package that your custom function imported.

If you wish to get the variable's value from the system variable table, you can use the method `getVariable(String variableName)` as follows:

```
String value = getVariable(variableName);
```

`setVariable(String variableName, String newValue)` allows you to update the variable in the system variable table by specifying its `variableName` together with the newValue as follows:

```
setVariable(variableName, "newValue");
```

8.2.2.7 getReturnType()

`getReturnType()` is the other abstract method declared in the `AbstractFunction` class. Your custom function, a subclass of `AbstractFunction`, must therefore implement this method. The value returned by `getReturnType()` indicates the type of the value you return in method `execute()`.

Choices for your return value are:

`java.lang.Boolean.class`

`java.lang.Character.class`

`java.lang.Double.class`

`java.lang.Float.class`

`java.lang.Integer.class`

`java.lang.Long.class`

`java.lang.String.class`

`java.lang.StringBuffer.class`

8.2.3 Sample custom function

Once you have read “Writing custom functions” on page 190, you are ready to write your own custom function. The following Java code is the implementation for a function that returns a string resulting from adding **25** to the function's input value.

Comments have been added for readability.

```
import com.xenos.transform.kernel.function.*;
public class ADD25 extends AbstractFunction
 {
 public ADD25() {
 /* specifies the function name to be used within Data Transformation */
 setName("ADD25");
 /* specifies the function description to be used within Data Transformation */
 setDescription("Return a string resulting from adding 25 to input value.");
 }
 public Object execute() throws Exception {
 /* retrieves the first argument and assigns it to String variable 'input' */
 String input = (String)getFunctionContext().getArgument(0).getValueInstance();
 /* convert the String to an int, add 25 and assign it to int variable 'answer' */
 int answer = Integer.parseInt(input) + 25;
 /* return the answer as a String */
 return answer + "";
 }
 public Class getReturnType() {
 /* specifies that method execute() returns a String */
 return java.lang.String.class;
 }
 }
```

8.2.4 Custom Function Wizard

Using this wizard, you can create a Output Transformation Designer custom function from within Data Transformation Engine that is used in Data Transformation Engine and OpenText Output Transformation Server .

The “Custom functions” on page 189 let you specify your own pre-defined Java functions by giving a function name, the number of arguments the function takes, and the location of the custom function within your system.

For more information, see “Defining custom functions” on page 189.

8.2.4.1 Custom Function directory

Accessing the Custom Function Wizard is a three-step process that involves changes to the following components:

- “Custom Function tab” on page 194
- “Custom Function Wizard setup” on page 195
- “Custom Function Editor” on page 196

It is recommended that you store custom functions within the `//external_functions` directory.

If you do not store custom functions within the `//external_functions` directory, you must add the location of your custom function to the classpath used by Data Transformation Engine . For example:

`<install_home>\initialFiles\common\com\xenos\add.Java`.

8.2.4.2 Custom Function tab

To add a custom function:

1. From inside Data Transformation Engine , open the MGP file for which you wish to use the Custom Function Wizard to create the custom function. This activates the MGP Settings and Details button, , on the Output Transformation Designer toolbar.
2. Click the **MGP Settings and Details** button.
3. Click the **Custom Function** tab.
4. Click the **Custom Function Wizard** button.

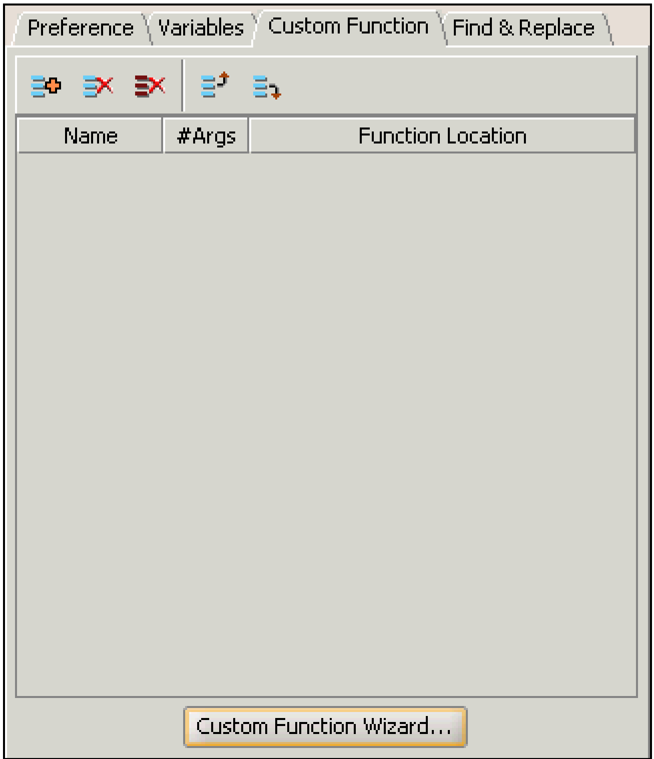


Figure 8-1: Empty instance of Custom Function tab

The **Custom Function Wizard** opens.

8.2.4.3 Custom Function Wizard setup

The Custom Function Wizard contains the following options:

- **Open an existing Function.** Looks for the file in the `//external_functions` directory that matches the function name and number of arguments.

  For example, if you have a custom function with package `com.xenos.functions` and name `ADD`, then the function name would be `com.xenos.functions.ADD`, and the file structure would be `com\xenos\functions\ADD.java`.

- **Create a new Function.** Complete the following when creating a new function:

  – **Package.** Custom function package name. For example, enter `com.xenos.functions`.

  – **Function.** Custom function name (mandatory). For example, enter `ADD`.

  – `#` **Arguments.** Number of function input arguments.

  – **Description.** Custom function description.

Click **OK** to continue.

The new function is saved to the `//external_functions` directory.

Figure 8-2: Custom Function Wizard

8.2.4.4 Custom Function Editor

Once a function has been added, the Custom Function Editor will appear with the default Java code for the function you chose to customize. To create the custom function:

1. Inside the file, modify the default Java code by typing new custom logic into the function.

   For example, under method `public Object execute(String parm1, String parm2...)` type `String a = "1"; return a;` to return “**1**”.

   You can use **Cut**, **Copy**, **Paste**, **Find**, **Replace**, **Undo**, and **Redo** commands in the text editor.

   See “Writing custom functions” on page 190 and “Sample custom function” on page 193 for tips on writing a custom function.

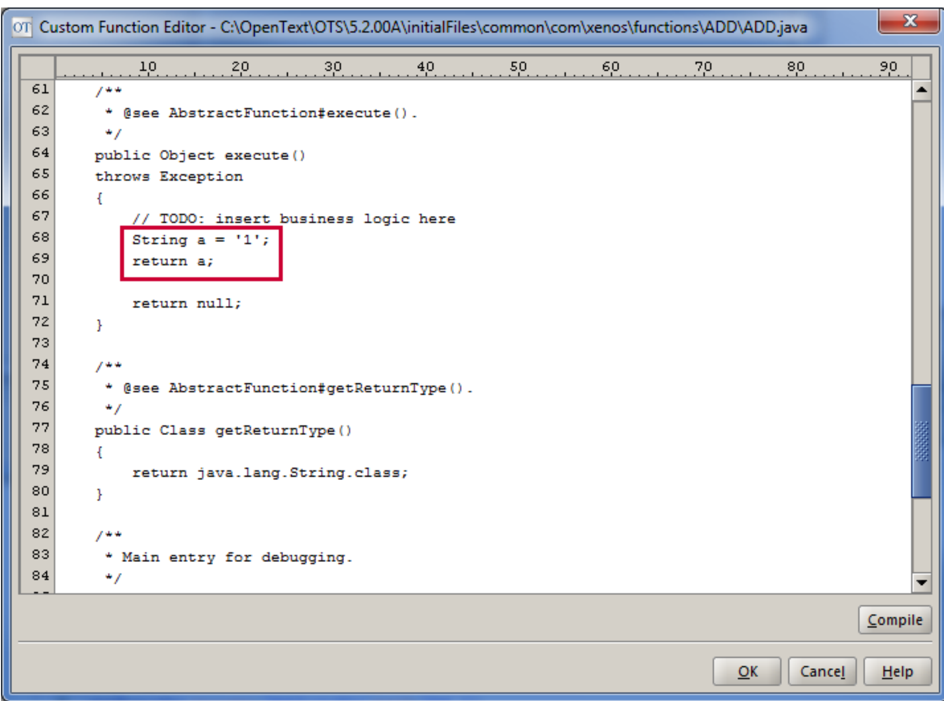


Figure 8-3: Custom Function Editor dialog

2. Click **Compile**. Save the source code to the `//external_functions` directory as `<function_name.java>`.

Note: The `JAVA_HOME` environment variable needs to be set before compiling.

3. Save the custom logic by clicking **OK**.

Note: If there is a Package, a new directory matching the Package will be created, for example, `<package/function_name.java>`.

Below the **Custom Function tab**, the new function is listed. Additional functions can be added as required.

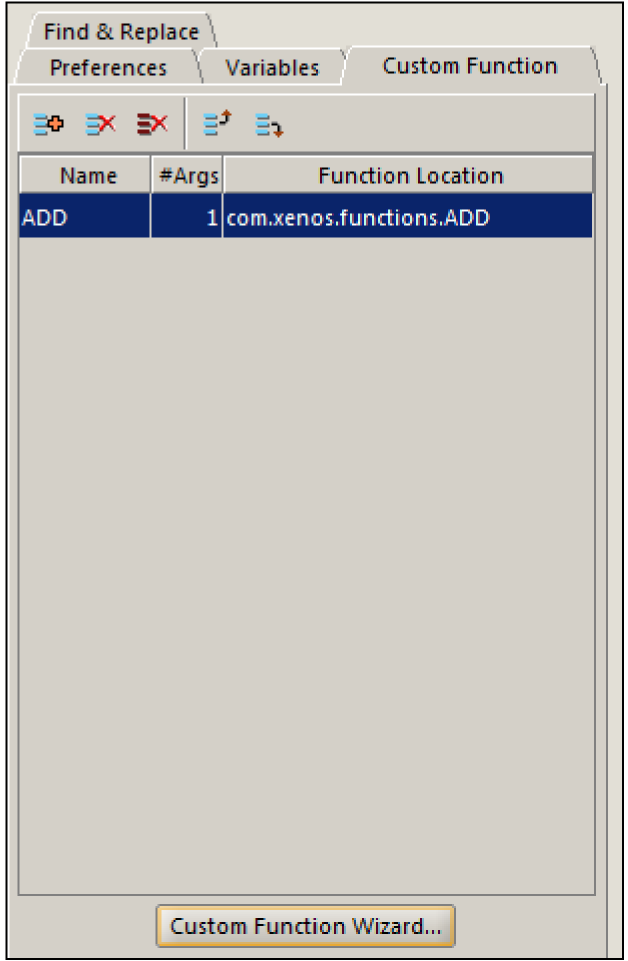


Figure 8-4: Custom Function tab with results

8.2.5 Custom Function Component Wizard

With this wizard, you can create a Output Transformation Designer custom function component that is used in Data Transformation Engine and OpenText Output Transformation Server .

To add a component:

1. Click the **New** button on the toolbar or select **New** (**Ctrl+N**) from the **File** menu. The **New Component Wizard** opens at the **Select a File Type** dialog window.

   Note: For detailed information on adding other components, see *OpenText Output Transformation Designer - User Guide (VDTOTS-H-UTD)*.

2. Open the **All Components** folder from the tree and then navigate to the **Processes** > **Transform** folder.

3. Select **Data Transformation Custom Function**.
4. Click **Next**.

   The **Data Transformation Custom Function Wizard** screen displays.
5. Select **Use Wizard** to configure the function with this wizard.
6. Click **Next** to open the **Custom Function dialog window**.
7. Complete the empty fields:
   - **File System.** Selects the file system that it will reside on, from the **File System** drop-down list.
   - **Package.** Indicates the package. If required, select an appropriate package that you need to call upon from the **Package** drop-down list.
   - **Function Name.** Specifies a function name.
   - **Description.** Indicates a description for your function.
   - **Argument Count.** Indicates the number parameters you want your function to use.

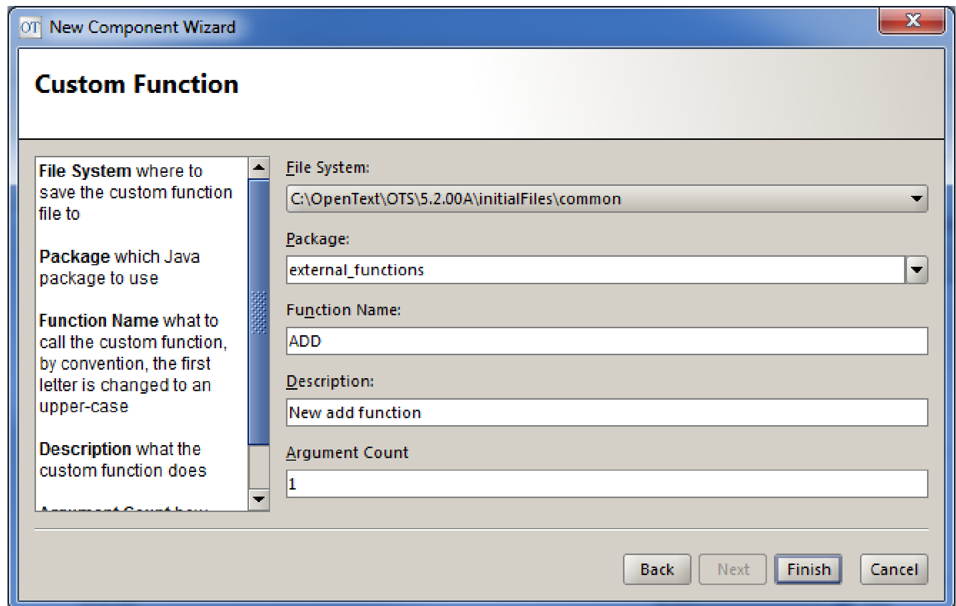


Figure 8-5: Custom Function Component Wizard: Custom Function screen

8. Click **Finish**.

   Your new custom function component appears in the **File Systems** window.

Once you have added a function, you can edit it using the Custom Function Editor, which is viewable in the **File Systems** tab inside the package that was selected. For more information, see “Editing Custom Functions” on page 200.

8.2.6 Editing Custom Functions

The Custom Function Editor enables you to modify the current Java code that defines your Output Transformation Designer custom function used in Data Transformation Engine and Output Transformation Server .

This page is a continuation of the **Custom Function Component Wizard**. In this section, you will be:

- Editing the Custom Function
- Editing the Java Settings
- Compiling the Custom Function

8.2.6.1 Editing the Custom Function

To edit and run the custom function:

1. Click the recently created file from the File Systems window to open it.
2. Inside the file, modify the default Java code by typing new custom logic into the function.

```
TEST_ADD.java
54      public ADD()
55      {
56          setName("ADD");
57          setDescription("");
58          argCount = 0;
59      }
60
61      /**
62       * @see AbstractFunction#execute().
63       */
64      public Object execute()
65      throws Exception
66      {
67          String a = "1";
68
69          return a;
70
71      }
72
73      /**
74       * @see AbstractFunction#getReturnType().
75       */
76      public Class getReturnType()
77      {
78          return java.lang.String.class;
1:1     Insert
```

Figure 8-6: Editing the Java code

For example, under method `public Object execute(String parm1, String parm2...)` type `String a = "1"; return a;` to return “**1**”to the calling method.

You can use **Cut**, **Copy**, **Paste**, **Find**, **Replace**, **Undo**, and **Redo** in the text editor.

For tips on writing a custom function, please see “Writing custom functions” on page 190 and “Sample custom function” on page 193.

Note: Remember to compile and save the custom logic. For more information, see “Compiling your Java file” on page 202.

8.2.6.2 Editing the Java settings

To change Java settings for a custom function:

1. From the **Tools** menu, select **Preferences**.
2. Inside the **Preferences window**, set the Java Preferences to where the directory is installed.
3. The default Java parameters are JavaHome and VmOptions. By clicking **JavaHome**, you may choose which Java Development Kit/Environment to run against.
4. Click **OK** to save your Java Settings, and then close your Java preferences.

8.2.6.3 Compiling the Custom Function

1. Verify that the File System is still open to the custom function.
2. Compile the custom function. The **Compiler tab** opens in the Events window displaying log or error messages, if needed. For more information, see “Compiling your Java file” on page 202.
3. After compilation, a message at the bottom of the Events window will be displayed notifying you that it has compiled successfully.

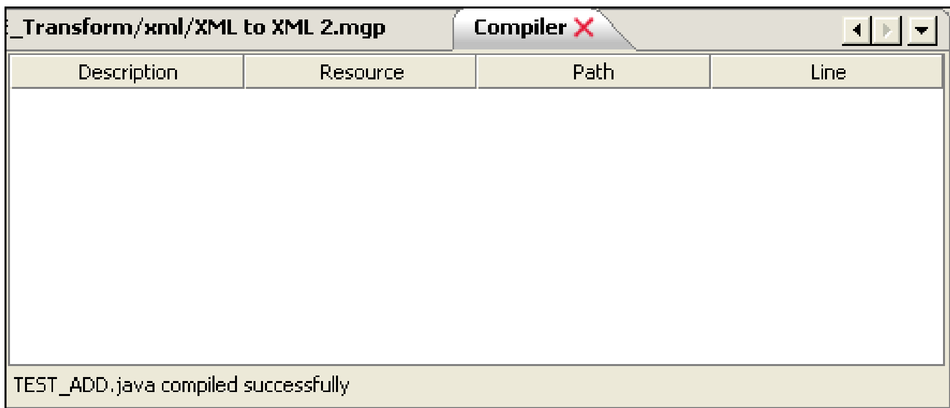


Figure 8-7: Events window - Compiler tab

For detailed information on the Events window, see *OpenText Output Transformation Designer - User Guide (VDTOTS-H-UTD)*.

8.3 Compiling your Java file

In order for your custom function to be available for use with Data Transformation Engine , you must compile your completed Java code to create a .class file. To do so, make sure your .java file is located in the `//external_functions` directory, or that you have added the location of your custom function to the classpath used by Data Transformation Engine . For more information on the OpenText Output Transformation Server classpath, refer to *OpenText Embedded Data Transformation Engine - Developer's Guide (VDTOTS-H-PDT)* in OpenText Output Transformation Server Developer's Guide.

You must add Data Transformation Engine 's `kernel.jar` to your classpath when compiling your source code. For example, you might enter the following at the command prompt:

```
%JAVA_HOME%\javac -classpath ..\lib\kernel.jar FUNCTION_NAME.java
```

where `%JAVA_HOME%` is the directory in which your `javac.exe` is located, and `FUNCTION_NAME` is the name of your custom function.

To make your custom component (custom function or custom preparser) available for use with Data Transformation Engine , you must compile your completed Java code to create a `.class` file. Your `.java` file must be located in either the `//external_functions` directory for custom functions or the `//external_preparsers` director for custom preparsers. You might also add either custom component to the classpath used by Data Transformation Engine because the locations are coded within Data Transformation Engine to look for them there or the classpath.

8.3.1 Compiling using the command prompt

When compiling your source code, add Data Transformation Engine 's `Xenos-t1transform.jar`, `Xenos-framework-util.jar`, and `Xenos-framework-engine.jar` to your classpath.

For example, for a custom component, you might enter the following at the command prompt:

```
%JAVA_HOME%\javac -classpath %install_path%\lib\common\Xenos-t1transform.jar;
%install_path%\lib\common\Xenos-framework-util.jar; %install_path%\lib\common\Xenos-
framework-engine.jar <FUNCTION_NAME or PREPARSER_NAME).java
```

where:

- `%JAVA_HOME%` is the directory in which your javac.exe is located.
- `%install_path%` is the install path.
- Select the custom component:
  - `FUNCTION_NAME` is the name of your custom function.
  - `PREPARSER_NAME` is the name of your custom preparser.

8.3.2 Compiling in Output Transformation Designer

To load the new custom component within Output Transformation Designer , you can compile the Java code from `.java` to `.class`. To compile your code do one of the following:

- If you are editing your custom component in Output Transformation Designer , select **Build** from the main menu bar and click **Compile** or right-click on the function within the file system and go compile

  Note: If you select **View** > **Toolbars** > **Build** from the main menu bar, **Compile**, , appears on the Output Transformation Designer toolbar, where you can also click it to compile the Java code.

- Right-click the newly created component in the directory and select **Compile**.
- If you are editing a custom function in the Custom Function Editor that you accessed through the Custom Function Wizard, click the **Compile** button at the bottom of the Custom Function Wizard.

Chapter 9

Dictionary files

Dictionary files are XML representations of EDI, SWIFT, HL7 and flat text transactions. A dictionary file is required for any Data Transformation Engine transformation in which EDI X12, EDI HIPAA, EDIFACT, SWIFT, HL7 or flat text is the input or output data format.

Data Transformation Engine makes use of dictionary files to allow for flexibility and ease of use in dealing with the differences between types of EDI, SWIFT, HL7 or flat text transactions. For example, although both are EDI HIPAA, the structures of EDI HIPAA 276 and EDI HIPAA 277 transactions are very different. The Data Transformation Engine run-time engine can handle these transactions when provided with a dictionary file for each one, since the Data Transformation Engine run-time engine is then essentially only dealing with XML.

How dictionary files work

If EDI, SWIFT, HL7 or flat text is your input format for a Data Transformation Engine transaction, the input is converted to XML using the structure defined in the appropriate dictionary file, and then another transformation into your desired output format is carried out.

If EDI, SWIFT, HL7 or flat text is your output type, your input file is transformed into XML, conforming exactly to the structure as defined in the specified dictionary file. Another conversion is made to the correct transaction for your output via the same dictionary file.

Getting a dictionary file for your transformation

Each EDI, SWIFT, HL7 and flat text transaction type must have a corresponding dictionary file in order to use that transaction with Data Transformation Engine.

Data Transformation Engine ships with sample EDI dictionary files. Find these in the `/_sample/DataTransformation/edi hipaa` and `/_sample/DataTransformation/edi x12 directories`.

9.1 Composing dictionary files

There are several types of dictionary files that you can compose.

9.1.1 Composing EDI dictionary files

The following instructions assume that you are familiar with EDI, XML, and the specific EDI transaction with which you are working.

Note: You may use EDI dictionaries shipped with Data Transformation Engine as reference for creating your own dictionary files.

A dictionary file is an XML file. Since it will be a representation of a specific EDI transaction, you must create the dictionary to match the structure of the transaction you are representing exactly. To get the structure you need, add a combination of these elements: **EDI**, **TransactionSet**, **Segment**, **Loop**, **Element**, **CompositeElement**, and **ComponentElement**.

EDI dictionary headers are written using the same guidelines as given below for dictionary files, without the TransactionSet node.

A dictionary file will have the following general node structure, which you will customize to conform exactly to the EDI structure you need:

```
<EDI>
<TransactionSet>
&lt;Segment&gt;
<Element/>
<Element/>
...
<CompositeElement>
<ComponentElement/>
...
</CompositeElement>
...
&lt;/Segment&gt;
<Loop>
&lt;Segment&gt;
<Element/>
...
<CompositeElement>
<ComponentElement/>
...
```

```
</CompositeElement>
...
...
<Loop>
...
</Loop>
</Loop>
...
</TransactionSet>
</EDI>
```

9.1.1.1 EDI dictionary XML declaration

As with all XML documents, the top line of your file must be the XML declaration. Simply insert the following as your first line of text:

```
<?xml version="1.0"?>
```

Everything below this line will be your structure of nodes.

9.1.1.2 EDI dictionary comments

You may find it useful to insert comments into your dictionary file as you write it for later reference. These take the form

```
<!-- My comment here... -->
```

and, when inserted, do not affect your overall EDI structure. As in any XML document, a comment must begin with <!-- and end with -->, and may contain any sequence of characters except the double hyphen (--) which is only permitted to close the comment.

9.1.1.3 EDI Node

This node will be at the top level of your structure's hierarchy. The `EDI` element must be given values for three attributes: **Type**, **Version**, and **Standard**.

Type	Defines the encoding that the text of your EDI transaction uses. For most purposes, this is ASCII.
Version	Identifies the version number of the standard that the transaction type follows.
Standard	Specifies the standard that the transaction type follows.

For example, to define a HIPAA 4010 transaction type that uses the ASCII character set, your EDI opening tag would be defined as follows:

```
<EDI Type="ASCII" Version="4010" Standard="HIPAA">
```

A closing tag, `</EDI>`, is required at the end of your file. The `EDI` element always contains exactly one `TransactionSet` child.

9.1.1.4 EDI TransactionSet node

The `TransactionSet` node is always a child element of the `EDI` node. It must be given values for attributes **ID** and **Name**. Optionally, you may add an attribute **Note**.

ID	Specifies a two- or three-character value that identifies the `transaction set`.
Name	Identifies the `transaction set`'s name.
Note	(optional) Is a value supplied that would contain information for your own reference.

For example, if you are building your dictionary file for the EDI HIPAA 276 transaction, Health Care Claim Status Request, your `TransactionSet` element would be defined as follows:

```
<TransactionSet ID="276" Name="Health Care Claim Status Request" Note="Optional
info...">
```

A closing tag, `</TransactionSet>`, is required at the end of your file just before the EDI closing tag. The `TransactionSet` element may contain as many `Segment` and `Loop` children as is required by your particular transaction.

9.1.1.5 EDI Segment node

A `segment` node is always a child element of either a `TransactionSet` or `Loop` node. It must be given values for four attributes: **ID**, **Name**, **Req** and **MaxUse**. Optionally, there are also two attributes: **Note** and **Type**.

ID	Specifies a two- or three-character value that identifies the `segment`.
XID	Is an alias for ID. This is useful if you wish to distinguish between multiple segments that have the same IDs but with different types.
Name	Identifies the `segment`'s name.
Req	Is the requirement, which must be either **M** (mandatory), or **O** (optional).
MaxUse	Sets the maximum allowed number of occurrences of the `segment`, which is set to null if the `segment` is allowed to exist an indefinite number of times.
Note	(optional) Is a value supplied that would contain information for your own reference.

Type	(optional) When multiple sibling loops exist in your transaction, it helps the Data Transformation Engine run-time engine differentiate between the loops. For detailed information see "Segment Type attribute" on page 209.

For example, if you wish to add an ST segment, the Transaction Set Header, which is mandatory and may only occur once, your `segment` element would be defined as follows:

```
&lt;Segment ID="ST" Name="Transaction Set Header" Req="M" MaxUse="1"&gt;
```

A closing tag, `</Segment>`, is required to close the segment. The `segment` element may contain as many `Element` and `CompositeElement` children as is required by the transaction.

9.1.1.5.1 Segment Type attribute

Type is only required when your dictionary file contains sibling loops that have the same first segment. For example, your dictionary file may have sibling loops `2000A` and `2000B` that both have a segment with `ID="HL"` as their first child segment. The Data Transformation Engine run-time engine requires that the `Type` attribute be added to these HL segments in order to determine to which loop the different parts of your input data belong.

The **Type** attribute specifies a condition that is always true for the segment to which it refers, and is unique to that loop. The condition could be that one of the segment's elements does or does not exist, or that one of its elements has a certain value. If the condition is based on the existence of a child element's value, it takes one of the forms:

- `Type="<ElementID>"` - child element with specified ID has a value.
- `Type="!<ElementID>" - child element with specified ID has null value.`

If the condition is based on the value of a child element, it takes one of the following forms:

- `Type="<ElementID>=='<value>'"` - child element with specified ID has given value.
- `Type="<ElementID>!='<value>'"` - child element with specified ID does not have given value.

For example, you may know that the third element of segment `HL` always has value `'20'` for loop `2000A` and always has value `'21'` for loop `2000B`. In this case, you would add `Type="03=='20'"` to segment `HL` in loop `2000A` and `Type="03=='21'"` to segment `HL` in loop `2000B`.

9.1.1.6 EDI Loop node

A `loop` node is always a child element of either a `TransactionSet` node, or another `loop` node. It must be given values for the following attributes:

ID	Specifies a two- or three-character value that identifies the `loop`.
MaxUse	Define its maximum number of occurrences. MaxUse is set to **null** if the loop is allowed to repeat an indefinite number of times.
XID	Is an alias for ID. This is useful if you wish to distinguish between multiple loops that have the same IDs.
Req	(Optional) Determines the requirement of the particular loop. Values for this may be either **M** (mandatory) or **O** (optional). If you do not include this attribute the loop is considered optional.
Random	Parses an EDI message that contains random order of segments. The Random attribute can have the values true or false. The default Random value is **false**.

For example, if you wish to create a loop `2000A` that is permitted to repeat as many times as is necessary, your `loop` element would be defined as follows:

```
<Loop ID="2000A" MaxUse="">
```

A closing tag, `</Loop>`, is required to close the loop. The `loop` element may contain as many `segment` and `loop` children as is required by the transaction.

9.1.1.7 EDI Element node

An `element` node is always a child element of a `Segment` node. It **must** be given values for six attributes: **ID**, **Name**, **Req**, **Type**, **MinLength** and **MaxLength**.

ID	Is a numerical value based on the element's position within its parent segment. The Element and CompositeElement items are given ID values sequentially within their parent segment. For example, in a segment containing two element nodes, one CompositeElement, and then another element, the respective ID values would be '01', '02', '03' and '04'.
Name	Specifies the element's name.

Req	Defines the requirement which must be either **M** (mandatory), **O** (optional), **N** (not used) or a conditional requirement. For detailed information, see “Req conditional requirement” on page 212.
Type	Specifies the type of the content that the element node will hold. Permitted values are:
AN - string	Element may contain a sequence of any characters from the basic or extended character sets.
DT - date	Element's value is either in format YYYYMMDD or YYMMDD, depending upon the MinLength and MaxLength.
ID - identifier	Element contains a value from a predefined list of codes that is maintained by the ASC X12 Committee.
Nn - numeric	(When implemented, **n** takes on an integer value, for example, N0 and N2 both work). Element contains a numeric value with an implied decimal point n positions from the right. For example, a transmitted value of 789 when specified as numeric type N2 represents a value of 7.89. A leading minus sign (-) is used to indicate a negative value.
R - decimal	Element may contain an explicit decimal point (unlike Type Nn) in its value if the value is not an integer value. For example, a transmitted value of 7.89 when specified as type R represents a value of 7.89. A leading minus sign (-) is used to indicate a negative value.
TM - time	Element's value is in general format HHMMSSd..d (where d are decimal seconds), and the precise format is determined by the MinLength and MaxLength attributes. For example, transmitted data of four characters denotes HHMM, and six characters denotes HHMMSS. Transmitted data of 9 characters denotes HHMMSSddd.
MinLength	Sets the minimum length permitted for the element's content.
MaxLength	Sets the maximum length permitted for the element's content. Optionally, you may add an attribute `Note` whose value would contain information for your own reference.

For example, if you wish to add a mandatory element that will have alpha-numeric content with minimum and maximum lengths of 3 and 3, and whose name is 'Transaction Set Identifier Code' as the first element in a segment, your `Element` element would be defined as follows:

```
<Element ID="01" Name="Transaction Set Identifier Code" Req="M" Type="AN" MinLength="3"
MaxLength="3" Note="'ST01' used to select appropriate transaction set definition"/>
```

No closing tag is required, but be sure to end all element tags with />. Element elements may not contain any children.

9.1.1.7.1 Req conditional requirement

If the requirement of an `Element` or `CompositeElement` depends upon the outcome of a condition, then its **Req** attribute should be a conditional requirement. This requirement will always evaluate to one of **M**, **O**, or **N**. A conditional requirement must take the form:

```
condition, trueReq, falseReq
```

A condition is comprised of a combination of terms, relational operators and conditional operators. Terms are either sibling element `ID`s (for example, `04` or `12`) or constant values (for example, `'3'` or `'foo'`).

9.1.1.8 Relational operators

Operator	Meaning	Example	Returns true if
==	is equal to	01=='A'	Sibling element 01 has value equal to 'A'.
!=	is not equal to	02!='9'	Sibling element 02 does not have value equal to '9'.
>	is greater than	03>'81'	Sibling element 03 has value greater than '81'.
<	is less than	04<'729'	Sibling element 04 has value less than '729'.

9.1.1.9 Conditional operators

Listed from highest to lowest precedence:

Operator	Meaning	Example	Returns true if
<elementID>	sibling element with given ID exists	01	Value exists for sibling element 01.
!	not	!02	Value does not exist for sibling element 02 (value is null).
&& *see note	and	03&&11	Value exists for both sibling elements 03 and 11.
||	or	08||26	A value exists for either sibling element 08 or 26, or both values exist.

- * **&&**; replaces the standard logic operator `&&` due to the special character status given to the `&` character in XML
- The **trueReq** is the requirement to be applied if the given condition evaluates to **true**. This may only take the value **M** (mandatory), **O** (optional), or **N** (not used), and may not be the same as the value given to `falseReq`.
- The **falseReq** is the requirement to be applied if the given condition evaluates to **false**. This may only take the value **M**, **O** or **N**, and may not be the same as the value given to `trueReq`.

9.1.1.9.1 Conditional operators examples

```
Req="04,M,O"
```

If a value exists for sibling element 04 then requirement is mandatory, otherwise requirement is optional.

```
Req="!05,M,O"
```

If the value of sibling element 05 is null, then requirement is mandatory, otherwise requirement is optional.

```
Req="02=='K'||03>'6'&amp;&amp;07||!09,M,O"
```

If sibling element 02 has value 'K', OR if sibling element 03 has value greater than '6' and a value for sibling element 07 exists, OR if the value of sibling element 09 is null, then requirement is mandatory, otherwise requirement is optional.

9.1.1.10 EDI CompositeElement node

A CompositeElement node is always a child element of a Segment node. It must be given values for three attributes: **ID**, **Name**, and **Req**.

ID	Is a numerical value based on the `CompositeElement`'s position within its parent segment. Elements and CompositeElements are given `CompositeElement` values sequentially within their parent segment. For example, in a segment containing two Elements, one CompositeElement, and then another Element, the respective ID values would be `'01'`, `'02'`, `'03'` and `'04'`.
Name	Specifies the CompositeElement's name.
Req	Defines the requirement which must be either **M** (mandatory), **O** (optional) or **N** (not used).

For example, if you wish to add a mandatory CompositeElement whose name is 'Composite Medical Procedure Identifier' as the first child in a segment, your CompositeElement element would be defined as follows:

```
<CompositeElement ID="01" Name="Composite Medical Procedure Identifier"
Req="M">
```

A closing tag, </CompositeElement>, is required to close this element. A CompositeElement may contain only ComponentElement children, but it may contain as many of these as is required by the transaction.

9.1.1.11 EDI ComponentElement node

A ComponentElement node is always a child element of a CompositeElement node. It must be given values for six attributes: **ID**, **Name**, **Req**, **Type**, **MinLength** and **MaxLength**.

ID	Is a numerical value based on the ComponentElement's position within its parent CompositeElement. For example, the first ComponentElement must have ID="01" and the fourth must have ID="04".
Name	Specifies the ComponentElement's name.
Req	Defines the requirement which must be either **M** (mandatory), **O** (optional), **N** (not used) or a conditional requirement. For detailed information, see "Req conditional requirement" on page 212 in *EDI Element Node*.

Type	Specifies the type of the content that the ComponentElement node will hold. Permitted values are:
AN - string	ComponentElement may contain a sequence of any characters from the basic or extended character sets.
DT - date	ComponentElement's value is either in format YYYYMMDD or YYMMDD, depending upon the MinLength and MaxLength.
ID - identifier	ComponentElement contains a value from a predefined list of codes that is maintained by the ASC X12 Committee.
Nn - numeric	(when implemented n takes on an integer value, for example, N0 and N2 both work). ComponentElement contains a numeric value with an implied decimal point n positions from the right. For example, a transmitted value of 789 when specified as numeric type N2 represents a value of 7.89. A leading minus sign (-) is used to indicate a negative value.
R - decimal	ComponentElement may contain an explicit decimal point (unlike Type Nn) in its value if the value is not an integer value. For example, a transmitted value of 7.89 when specified as type R represents a value of 7.89. A leading minus sign (-) is used to indicate a negative value.
TM - time	ComponentElement's value is in general format HHMMSSd..d (where d are decimal seconds), and the precise format is determined by the MinLength and MaxLength attributes. For example, transmitted data of four characters denotes HHMM, and six characters denotes HHMMSS. Transmitted data of 9 characters denotes HHMMSSddd.
MinLength	Sets the minimum length permitted for the ComponentElement's content.
MaxLength	Sets the maximum length permitted for the ComponentElement's content.

For example, if you wish to add an optional ComponentElement that will have alpha-numeric content with minimum and maximum lengths of 2 and 2, and whose name is 'Procedure Modifier' as the sixth element in a CompositeElement, your ComponentElement element would be defined as follows:

```
<ComponentElement ID="06" Name="Procedure Modifier" Req="O" Type="AN" MinLength="2"
MaxLength="2"/>
```

No closing tag is required, but be sure to end all ComponentElement tags with />. ComponentElement elements may not contain any children.

9.1.2 Composing SWIFT dictionary files

The following instructions assume that you are familiar with SWIFT syntax, XML, and the specific SWIFT transaction with which you are working.

The following sections describe how to create a dictionary for Data Transformation Engine to interpret SWIFT messages.

9.1.2.1 Starting SWIFT messages

All SWIFT messages are surrounded by the parent root tag called <SWIFT> with an ID parameter="SWIFT".

For example:

```
<SWIFT ID="SWIFT">
. . .
</SWIFT>
```

9.1.2.2 General structure

SWIFT tags contain the following general structure:

- "Basic header block" on page 216
- "Application header block" on page 218
- "User Header block" on page 220
- "Text Block or Body" on page 221
- "Trailer Block" on page 227

9.1.2.3 Basic header block

```
<BASIC ID="BASIC" Req="M">
...
</BASIC>
```

ID	Uses any word or character. It is recommended that you use **BASIC**.
Req	Contains either **M** (mandatory) or **O** (optional), which represents the requirement of that block.

SWIFT message example

```
{1: F 01 BANKBEBB 2222 123456}
(a) (b) (c) (d) (e)
```

(a) **Application ID**	F = FIN, A = GPA or L = GPA (logins, etc).
(b) **Service ID**	01 = FIN/GPA, 21 = ACK/NAK.
(c) **LT address**	12 Characters, must not have X in position 9.
(d) **Session number**	Added by the CBT, padded with zeroes.
(e) **Sequence number**	Added by the CBT, padded with zeroes.

Writing a Basic header (for a and b)

Note: (c), (d), and (e) are written in the same syntax.

A BASIC tag contains raw data. These data are represented in an Element tag which is written in this syntax:

```
<BASIC ID="BASIC" Req="O">
<Element ID="APPID" Name="Application ID" Req="M" Type="1!c"/>
<Element ID="SRVID" Name="Service ID" Req="M" Type="2!n"/>
. . .
</BASIC>
```

ID	Parameter should be a group of characters that references the Name parameter; this field is created by the author of the dictionary.
Name	Parameter specifies the raw data, for example, if raw data is 12345 and this is the account number of a client, then Name="Account Number".
Req	Parameter MUST be **M** for mandatory as specified by the user.
Type	Parameter tells the Data Transformation Engine run-time engine the types of characters and the amount of characters to expect. The final outcome for the two elements in the example above is that APPID will retrieve raw data "F", and SRVID will retrieve raw data "01".

9.1.2.4 Application header block

```
<APP ID="APP" Req="M">
...
</APP>
```

ID	Uses any word or character. It is recommended that you use **APP**.
Req	Contains either **M** (mandatory) or **O** (optional), which represents the requirement of that block.

Writing an Application header

An APP tag must have the following structure:

```
<APP ID="APP" Req="O">
&lt;Segment ID="I" Name="Input"&gt;
<Element ID="INOUT" Name="Input or Output flag" Req="M" Type="1!c"/>
<Element ID="MSGTP" Name="Message Type" Req="M" Type="3!n"/>
&lt;/Segment&gt;
&lt;Segment ID="I" Name="Input"&gt;
<Element ID="INOUT" Name="Input or Output flag" Req="M" Type="1!c"/>
<Element ID="MSGTP" Name="Message Type" Req="M" Type="3!n"/>
&lt;/Segment&gt;
</APP>
```

As many elements as are desired may be added within each segment, after the two required element tags shown above. If you include additional elements in your application header block, be sure to provide values for the following element attributes:

ID	Is a group of characters that references the Name parameter; this field is created by the author of the dictionary.
Name	Specifies the raw data, for example, if raw data is 12345 and this is the account number of a client, then Name="Account Number".
Req	Requires either **M** or **O**.
Type	Tells the Data Transformation Engine run-time engine the types and amount of characters to expect.

SWIFT message example

```
{2: I 100 BANKDEFFXXXX U 3 003}
(a) (b) (c) (d) (e) (f)
```

This corresponds with the following example APP block definition:

```
<APP ID="APP" Req="O">
&lt;Segment ID="I" Name="Input"&gt;
<Element ID="INOUT" Name="Input or Output flag" Req="M" Type="1!c"/>
<Element ID="MSGTP" Name="Message Type" Req="M" Type="3!n"/>
<Element ID="RCVAD" Name="Receive Address" Req="M" Type="12!c"/>
<Element ID="MSGPR" Name="Message Priority" Req="M" Type="1!c"/>
<Element ID="DLMON" Name="Delivery Monitoring" Req="M" Type="1!n"/>
<Element ID="OBSPD" Name="Obsolescence Period" Req="M" Type="3!n"/>
&lt;/Segment&gt;
&lt;Segment ID="O" Name="Output"&gt;
<Element ID="INOUT" Name="Input or Output flag" Req="M" Type="1!c"/>
<Element ID="MSGTP" Name="Message Type" Req="M" Type="3!n"/>
<Element ID="INTM" Name="Input Time" Req="M" Type="4!n"/>
<Element ID="MIRAD" Name="MIR and Senders Address" Req="M" Type="28!c"/>
<Element ID="OUTDT" Name="Output Date" Req="M" Type="6!n"/>
<Element ID="OUTTM" Name="Output Time" Req="M" Type="4!n"/>
<Element ID="MSGPR" Name="Message Priority" Req="M" Type="1!c"/>
&lt;/Segment&gt;
</APP>
```

(a) **Header**	Specifies whether the header is input or output: I = Input, O = Output
(b) **Message Type**	Message Type
(c) **Receiver's Address**	Receiver's address with X in position 9 and padded with X's if no branch
(d) **Message Priority**	S = System, N = Normal, U = Urgent
(e) **Delivery Monitoring**	Delivery Monitoring:
	1 Non Delivery Warning (MT010)
	2 Delivery Notification (MT011)
	3 Both Valid = U1 or U3, N2 or just N

(f) **Obsolescence Period**	When a non delivery notification is generated:
	Valid for U = 003 (15 minutes)
	Valid for N = 020 (100 minutes)

Note: If block 2 (APP) exists in the SWIFT message, the Data Transformation Engine run-time engine compares the value of APP/SEGMENT/MSGTP against the provided dictionary to ensure that the message type corresponds to the dictionary file.

9.1.2.5 User Header block

```
<USER ID="USER" Req="M">
...
</USER>
```

ID	Uses any word or character. It is recommended that you use **USER**.
Req	Contains either **M** (mandatory) or **O** (optional), which represents the requirement of that block.

SWIFT message example

```
{3: {113:xxxx} {108:abcdefgh12345678} }
(a) (b)
```

(a) **Code**	Is an optional banking priority code.
(b) **Data**	Specifies Message User Reference (**MUR**) used by applications for reconciliation with ACK.

Writing a User header

A User tag contains a LOOP tag that is written in this syntax:

```
<USER ID="USER" Req="O">
<Loop ID="Info">
. . .
</Loop>
</USER>
```

Note: The value of the ID parameter can be any word or characters specified by the author of the dictionary.

A LOOP tag contains Element tags that are written in this syntax:

```
<USER ID="USER" Req="O">
<Loop ID="INFO">
<Element ID="CODE" Name="Generic code" Req="M" Type=""/>
<Element ID="DATA" Name="Generic data" Req="M" Type=""/>
</Loop>
</USER>
```

Note: For a loop in a user header, you can specify only two elements.

ID	Parameter should be a group of characters that references the Name parameter; this field is made up by the author of the dictionary.
Name	Parameter specifies the raw data, for example, if raw data is 12345 and this is the account number of a client, then Name="Account Number".
Req	Parameter **must** be **M** as specified by the user.
Type	Parameter tells the Data Transformation Engine run-time engine the types and amount of characters to expect. In the example above, a type parameter is not specified because any value can be used.

The final outcome for (a) for the two elements in the example above is that CODE will retrieve raw data "113" and DATA will retrieve "xxxx". For (b), the final outcome of the two elements is that CODE will retrieve raw data "108" and DATA will retrieve abcdefgh12345678.

9.1.2.6 Text Block or Body

```
<TransactionSet ID="565" Name="Corporate Action Instructions">
...
</TransactionSet>
```

All text blocks are surrounded by a root tag called <TransactionSet>.

ID	Parameter should be a group of characters that references the SWIFT message type number. For example, if you are creating a SWIFT message type 565, then the ID="565". This ID number MUST be the message type number specified by SWIFT.

Name	Parameter specifies the name given to the SWIFT message type referenced by 565. For example, 565 is the Corporate Action Instruction, therefore Name="Corporate Action Instructions". For the name parameter, you can specify any name. However, it is recommended that you use the name specified by the SWIFT message type.

9.1.2.6.1 Usage rules for TransactionSet

The TransactionSet Tags Contain Loop Tags

Each sequence is a loop.

A loop is written in this syntax:

```
<Loop ID="A" Name="Sequence A" Req="M">
....
</Loop>
```

ID	Parameter should be a unique ID specified in the SWIFT Format Specification table (can be anything, but use the SWIFT Format Specification to be consistent).
Name	Parameter is just a name for the loop specified in the SWIFT Format Specification table (can be anything, but use the SWIFT Format Specification name to be consistent).
Req	Parameter MUST either be **M** (mandatory) or **O** (optional) specified in the SWIFT Format Specification table.

The Loop Tags Contain Segment and Element Tags

- Each SWIFT tag(s) surrounded by an arrow in the SWIFT Format Specification table is a loop.

A loop is written in the following syntax:

```
<Loop ID="L_99A" Name="Loop of 99A" Req="M">
....
</Loop>
</Loop>
```

ID	Parameter should be a unique ID created by the user.

Name	Parameter is just a name for the loop created by the user.
Req	Parameter MUST either be **M** or **O** specified in the SWIFT Format Specification table.

The Loop tags contain Segment and Element tags.

Each SWIFT tag that has a lowercase letter in the SWIFT Format Specification table has a loop surrounding it.

An example of a lowercase letter tag is 95a or 97a:

- 95a has 4 other options of 95P, 95Q, 95R, and 95S as shown in the Content column of the SWIFT Format Specification table.
- 97a has 2 other options of 97A and 97B as shown in the Content column of the SWIFT Format Specification table.

A loop is written in the following syntax:

```
<Loop ID="L_95a" Name="Loop of 95a" Req="M" Max="1">
....
</Loop>
```

ID	Parameter should be a unique ID created by the user.
Name	Parameter is just a name for the loop.
Req	Parameter MUST either be **M** or **O**, specified in the SWIFT Format Specification table for that specific tag.
Max	Parameter in this situation is always set to **1**. This means only looping once to validate whether the input SWIFT message is correct. This is optional, but is available just in case the SWIFT message is wrong (for example, if SWIFT has 95A and another 95A right after). This would be incorrect because there should only be one instance of 95A.

If this SWIFT tag is surrounded by an arrow in the SWIFT Format Specification table, then the Max parameter is not required.

- Each SWIFT tag is a segment tag.

A Segment tag is written in the following syntax:

```
&lt;Segment ID="20C" Name="Processing Reference" Req="M"&gt;
....
&lt;/Segment&gt;
```

ID	Parameter MUST be the SWIFT tag specified in the SWIFT Format Specification table.
Name	Parameter is the name given to this SWIFT tag specified in the SWIFT Format Specification table (can be anything, but use the SWIFT Format Specification name to be consistent).
Req	Parameter is either **M** or **O**, specified in the SWIFT Format Specification.

If a SWIFT tag is a Starting block then the Segment tag is written in the following syntax:

```
&lt;Segment ID="16R" Name="Starting Block" Req="M" Type="GENL"&gt;
....
&lt;/Segment&gt;
```

ID	Parameter MUST be the SWIFT tag specified in the SWIFT Format Specification table. It is safe to say that it is always “16R”.
Name	Parameter is the name given to this SWIFT tag specified in the SWIFT Format Specification table. It is safe to say that it is always “Starting Block” (can be anything, but use the SWIFT Format Specification name to be consistent).
Req	Parameter **must** either be **M** or **O**, specified in the SWIFT Format Specification table.
Type	Parameter **must** be the characters given in the CONTENT column of the SWIFT Format Specification table.

If a SWIFT tag is an Ending block, then the Segment tag is written in this syntax:

```
&lt;Segment ID="16S" Name="Ending Block" Req="M" Type="GENL"&gt;
....
&lt;/Segment&gt;
```

ID	Parameter **must** be the SWIFT tag specified in the SWIFT Format Specification table, it is safe to say that it is always “16S”.
Name	Parameter is the name given to this SWIFT tag specified in the SWIFT Format Specification table. It is safe to say that it is always “Ending Block” (can be anything, but use the SWIFT Format Specification name to be consistent).

Req	Parameter **must** either be **M** or **O**, specified in the SWIFT Format Specification table.
Type	Parameter **must** be the characters given in the CONTENT column of the SWIFT Format Specification table.
Type	Parameter is usually the same as the Starting Block Type parameter. It signifies to the Data Transformation Engine runtime engine that this block is done.

- If a SWIFT tag has a lowercase letter, there is a loop.

Now the Segment tag for this case is written in this syntax:

```
<Loop ID="L_97a" Name="Loop of 97a" Req="M" Max="1">
&lt;Segment ID="97A" Name="Account" Req="O"&gt;
....
&lt;/Segment&gt;
&lt;Segment ID="97B" Name="Account" Req="O"&gt;
....
&lt;/Segment&gt;
</Loop>
```

ID	Parameter MUST be the SWIFT tag specified in the SWIFT Format Specification table.
Name	Parameter is the name given to this SWIFT tag specified in the SWIFT Format Specification table (can be anything, but use the SWIFT Format Specification name to be consistent).
Req	Parameter is ALWAYS **O**, regardless of whether the lowercase letter tag (for example, 97a) is mandatory or optional. The **M** and the **O** should be specified in the Loop tag (for an example, see “EDI Loop node” on page 210, which has a SWIFT tag of lowercase letters).

- SWIFT tags contain SWIFT messages with data. Data can be characters that signify an action or just raw data. These data are represented in an Element tag.

An Element tag is written in this syntax:

```
&lt;Segment .....&gt;
<Element ID="QUALF" Name="Qualifier" Type=":4!c"/>
<Element ID="REF" Name="Reference" Type="//16x"/>
```

```
&lt;/Segment&gt;
```

ID	Parameter should be a group of characters that references the Name parameter; this field is made up by the author of the dictionary.
Name	Parameter is the name given to this SWIFT tag specified in the SWIFT Format Specification table (can be anything, but use the SWIFT Format Specification name to be consistent).
Type	Parameter tells the Data Transformation Engine run-time engine the types and amount of characters to expect.

Everything must be entered in the **Type** parameter, including /, //, :, and [].

For example, if the content is: 4!c/16x//12c/[4!a][/12c], then five Element tags are entered into the Type parameters accordingly:

```
(a) :4!c -> ... Type=":4!c"
(b) /16x -> ... Type="/16x"
(c) //12c -> ... Type="//12c"
(d) /[4!a] -> ... Type="/[4!a]"
(e) [/12c] -> ... Type="[/12c]"
```

Any time there is a new line between fields within a segment, you must place 'CRLF' as part of the field preceding the new line.

9.1.2.6.2 Example

The standard says

```
[/1!a][/34x]
```

```
4!a2!a2!c[3!c]
```

You should write your first field's Type attribute as

```
Type="[/1!a][/34x]CRLF"
```

and your second field's Type attribute as

```
Type="4!a2!a2!c[3!c]"
```

If neither of these first two values exists, the CRLF will be ignored. If either of the first two values exists, CRLF will be used.

Note: CRLF stands for carriage return + linefeed.

- We do NOT need the Req parameter, because the Type also states it, by the [] being optional otherwise mandatory. Thus (c) and (d) means optional, for example, they don't have to be provided in the SWIFT message.

- If the Content column in the SWIFT Format Specification table has two or more types on separate lines, then you need to specify this in the Type parameter, unless it is already specified in the SWIFT Format Specification table.

9.1.2.6.2.1 Example of One that is Not Specified

For the SWIFT tag 61, the content is:

```
6!n[4!n]2a[1!a]15d1!a3!c16x[//16x]
[34x]
```

The Type for the second line is:

```
Type="[1*34]"
```

The characters 1* lets the Data Transformation Engine runtime engine know that there is data on a new line of the SWIFT message that is part of the SWIFT tag 61.

9.1.2.6.2.2 Example of One that is Already Specified

For the SWIFT tag 35B, the content is:

```
[ISIN1!e12!c]
[4*35x]
```

The Type for the second line is:

```
Type="[4*35x]"
```

The characters 4*, already specified in the SWIFT Format Specification table, lets the Data Transformation Engine runtime engine know that on the next four new lines of the SWIFT message, the data is part of the SWIFT tag 35B.

9.1.2.7 Trailer Block

```
<TRAILER ID="TRAILER" Req="M">
...
</TRAILER>
```

ID	Uses any word or character. It is recommended that you use **TRAILER**.
Req	Contains either **M** (mandatory) or **O** (optional), which represents the requirement of that block.

9.1.2.7.1 SWIFT message example

```
{5: {MAC:12345678}{CHK:123456789ABC}
(a) (b)
```

9.1.2.7.2 Writing a Trailer Block

A LOOP tag is written in this syntax:

```
<Loop ID="INFO">
. . .
</Loop>
```

Note: The value of the ID parameter can be any word or characters specified by the author of the dictionary.

A LOOP tag contains Element tags that are written in this syntax:

```
<TRAILER ID="TRAILER" Req="O">
<Loop ID="INFO">
<Element ID="CODE" Name="Generic code" Req="M" Type=""/>
<Element ID="DATA" Name="Generic data" Req="M" Type=""/>
</Loop>
</TRAILER>
```

Note: For a loop in a trailer header, you can specify only two elements.

ID	Parameter should be a group of characters that references the Name parameter; this field is created by the author of the dictionary.
Name	Parameter specifies the raw data, for example, if raw data is 12345 and this is the account number of a client, then Name="Account Number".
Req	Parameter **must** be **M** as specified by the user.
Type	Parameter tells the Data Transformation Engine run-time engine the types amount of characters to expect. In the example above, a type parameter is not specified because any value can be used.

The final outcome for (a) of the two elements in the example above is that CODE will retrieve raw data "MAC" and DATA will retrieve "12345678". For (b), the final outcome of the two elements is that CODE will retrieve raw data "CHK" and DATA will retrieve "123456789ABC".

9.1.3 Composing HL7 dictionary files

The following instructions assume that you are familiar with HL7, XML, and the specific HL7 transaction with which you are working.

A dictionary file is an XML file. Since it will be a representation of a specific HL7 transaction, you must create the dictionary to match the structure of the transaction you are representing exactly. To get the structure you need, you can add a combination of these elements: **HL7**, **MessageStructure**, **Segments**, **Message**, **Segment**, **Group**, **Field**, **FieldRepeat** and **Component**.

A dictionary file will have the following general node structure, which you will customize to conform exactly to the HL7 structure you need:

```
<HL7>
<MessageStructure>
<Message>
&lt;Segment/&gt;
&lt;Segment/&gt;
...
<Group>
&lt;Segment/&gt;
...
</Group>
...
</Message>
...
</MessageStructure>
<Segments>
&lt;Segment&gt;
<Field/>
<Field/>
<Field>
<Component/>
...
</Field>
<FieldRepeat>
<Field/>
</FieldRepeat>
...
```

```
&lt;/Segment&gt;
&lt;Segment&gt;
...
&lt;/Segment&gt;
...
</Segments>
</HL7>
```

Note: Each HL7 segment is recorded in both the MessageStructure and Segments portions of the dictionary file. Segment order is declared in the MessageStructure section, and then the segments are defined with their respective fields in the Segments section.

9.1.3.1 HL7 dictionary XML declaration

As with all XML documents, the top line of your file must be the XML declaration. Simply insert the following as your first line of text:

```
<?xml version="1.0"?>
```

Everything below this line will be your structure of nodes.

9.1.3.2 HL7 dictionary comments

You may find it useful to insert comments into your dictionary file as you write it for later reference. These take the form

```
<!-- My comment here... -->
```

and, when inserted, do not affect your overall HL7 structure. As in any XML document, a comment must begin with <!-- and end with -->, and may contain any sequence of characters except the double hyphen (--) which is only permitted to close the comment.

9.1.3.3 HL7 node

Child of	None.
Parent of	Exactly one MessageStructure child, exactly one Segments child.

This node will be at the top level of your structure's hierarchy. The HL7 element must be given a value for the attribute version.

For example, to define an HL7 transaction based on the standard for HL7 version 2.3, your HL7 opening tag would be defined as follows:

```
<HL7 version="2.3">
```

A closing tag, </HL7>, is required at the end of your file. The HL7 element always contains exactly one MessageStructure and one Segments child.

9.1.3.4 MessageStructure node

Child of	HL7 element.
Parent of	Zero or more Message children.

Along with its subelements, the MessageStructure node declares the overall structure of the HL7 transaction you are representing. It does not take any attributes, but it may contain as many Message elements as is required by your particular transaction.

A closing tag, </MessageStructure>, is required before you begin the Segments node.

9.1.3.5 HL7 Segments node

Child of	HL7 element.
Parent of	Zero or more Segment children.

Each of the Segments node's Segment children must have been previously declared in the MessageStructure portion of the dictionary file.

9.1.3.6 HL7 Message node

Child of	MessageStructure element.
Parent of	Zero or more Segment children, zero or more Group children.

This node declares the structure of an HL7 message, and may contain as many Segment and Group elements as is required by your particular transaction. Each Segment declared in the Message must be completely defined in the Segments section of your dictionary file.

The Message node **must** be provided with values for two attributes: **Type** and **Event**. Optionally, you may add either of the two attributes: **Description** and **Note**.

Type	The abbreviated message name of the transaction.
Event	The identifier for the trigger event of the message.
Description	(optional) Details the purpose of the message.
Note	(optional) Provides any additional information you want for your own reference.

For example, if you are writing a dictionary for an ADT transaction which has been triggered by event A01, your Message element would resemble the following:

```
<Message Type="ADT" Event="A01">
```

A closing tag, </Message>, is required before any additional Message elements and before the MessageStructure closing tag.

9.1.3.7 HL7 Segment node

Child of	Message element or Group element or Segments element.
Parent of	None (if within MessageStructure portion); zero or more Field children (if within Segments portion).

Segment nodes exist both within the Message and Segments portions of HL7 dictionary files. Every Segment node you define must be declared in both sections. It must be declared in the Message section to show its location in the HL7 transaction hierarchy, and defined in the Segments section to identify its fields.

9.1.3.7.1 Segment nodes within MessageStructure portion

Within the MessageStructure portion of the dictionary, all Segment nodes will be children of either the Segments node or a Group node. The Segment element **must** be given values for two attributes: **ID** and **Req**.

Optionally, you may add the attribute **Note**.

ID	A three-character value that identifies the segment. This value is used to match the corresponding segment defined in the Segments section of the dictionary.
Req	Defines the requirement which must be either **R** (required), or **O** (optional).
Note	(optional) Provides any additional information you want for your own reference.

If a Segment, or a set of them, is permitted to repeat within the transaction, enclose the declaration within a Group node. For example, you might have two segments PR1 and ROL that are allowed to loop together indefinitely, and additionally the ROL segment can repeat indefinitely within that loop. Your declaration in the MessageStructure portion of the dictionary of these Segments would resemble the following:

```
<Group ID="PR1-ROL_Loop" Min="" Max="">
&lt;Segment ID="PR1" Req="O"/>
<Group ID="ROL_Loop" Min="" Max="">
```

```
&lt;Segment ID="ROL" Req="O"/&gt;
</Group>
</Group>
```

Segments within the MessageStructure portion of your dictionary file are empty elements (for example, they have no children), so no separate closing tag is required. However, be sure to end all Segment nodes with /> as shown above.

9.1.3.7.2 Segment nodes within Segments portion

Within the Segments portion of the dictionary, all Segment nodes will be immediate children of the Segments node. The Segment element **must** be given values for two attributes: **ID** and **Name**.

Optionally, you may add the attribute **Note**.

ID	A three-character value that identifies the segment. This value is used to match the corresponding segment defined in the MessageStructure section of the dictionary.
Name	Specifies the Segment's name.
Note	(optional) Provides any additional information you want for your own reference.

A closing tag, </Segment>, is required at the end of your segment definition.

9.1.3.8 HL7 Group node

Child of	Message element or Group element.
Parent of	Zero or more Segment children, zero or more Group children.

This node is used to allow its children elements to loop. It may contain as many Segment and Group elements as is required by your particular transaction.

The Group node **must** be provided with values for the following three attributes: **ID**, **Min** and **Max**.

Optionally, you may add an attribute **Note**.

ID	Identifies the particular group.
Min	Defines the minimum number of times the group must loop.

Max	Defines the maximum number of times the group may loop. If this limit is undefined, leave the value set to null (for example, Max="").
Note	(optional) Provides any additional information you want for your own reference.

For example, you might have two segments PR1 and ROL that are allowed to loop together indefinitely, and additionally the ROL segment can repeat indefinitely within that loop. Your declaration in the MessageStructure portion of the dictionary of these segments would resemble the following:

```
<Group ID="PR1-ROL_Loop" Min="" Max="">
&lt;Segment ID="PR1" Req="O"/&gt;
<Group ID="ROL_Loop" Min="" Max="">
&lt;Segment ID="ROL" Req="O"/&gt;
</Group>
</Group>
```

A closing tag, </Group>, is required before any additional Group elements and before the Message closing tag.

9.1.3.9 HL7 Field node

Child of	Segment element.
Parent of	Zero or more Component children.

This node is always a child of the Segment element and is used to define an HL7 field.

The Field node must be provided with values for the following attributes: **Name**, **Len**, **DT**, **Req**, **RP**, **TBL** and **Item**.

Optionally, you may add an attribute **Note**.

Name	The field's name.
Len	The length of the value given to the field. As per the HL7 standard, the given length is only a recommendation and is therefore not enforced.
DT	The datatype of the field's value. Examples of datatypes include ID, DT and PL.
Req	The requirement of the field. This may take the value **R** (required) or **O** (optional).

	Note: Field and component conditional requirements are currently not supported.
RP	The repetition of the field. This may take the value **Y** (infinite repetition), **N** (no repetition) or an integer representing the maximum number of times allowed for the field to repeat.
TBL	The HL7 table number. **Note:** Table lookup is currently not supported.
Item	The item number as per the HL7 standard.
Note	(optional) Provides any additional information you want for your own reference.

9.1.3.10 HL7 FieldRepeat node

Child of	Segments element.
Parent of	One Field child.

This node is always a child of the Segments element and is there to indicate when a field is permitted to repeat.

The FieldRepeat node must be provided with values for the attribute **Name**.

Optionally, you may add the attribute **Note**.

Name	The name of the FieldRepeat element. For example, if the field that was repeating was the ninth field of segment IN2 you may wish to make `Name="IN2_9_Repeat"` for your FieldRepeat element.
Note	(optional) Provides any additional information you want for your own reference.

A closing tag, `</FieldRepeat>`, is required after the FieldRepeat's Field child.

9.1.3.11 HL7 Component node

Child of	Field element.
Parent of	None.

This node is always a child of the Field element and is used to describe components of a field.

The Component node must be provided with values for the following attributes: **Seq**, **Name**, **DT**, **Req**, **TBL**, and **Format**.

Optionally, you may add an attribute **Note**.

Seq	An integer unique to that component in the field. The numbering starts at 1 and is incremented for each component that exists in the field.
Name	The component's name.
DT	The datatype of the component's value. Examples of datatypes include ID, DT and PL.
Req	The requirement of the field. This may take the value **R** (required) or **O** (optional).
	Note: Field and component conditional requirements are currently not supported.
TBL	The HL7 table number. **Note:** Table lookup is currently not supported.
Format	This is the format for the date or time value of the component (for example, MMMM-DD-YYYY). Note that this is component formatting is currently not validated by Data Transformation Engine .
Note	Provides any additional information you want for your own reference. (Optional)

9.1.4 Composing Generic Flat Text dictionary files

The following instructions assume that you are familiar with the flat text format, XML, and the exact transaction with which you are working.

Note: You may use any flat text dictionaries shipped with Data Transformation Engine as reference for creating your own dictionary files.

A dictionary file is an XML file. Since it will be a representation of a specific transaction, you must create the dictionary to match the structure of the transaction you are representing exactly. All flat text structures can be thought of in terms of fields, records and sets of records. To get the structure you need, you will add a combination of these elements: **FlatDataDef**, **RecordSet**, **RecordsDef**, **Record** and **Field**. Field is the only element to correspond to actual values within your flat text transactions. All others are used strictly to provide structure.

For example, suppose your flat text file contains the following information:

```
"Smith, Jane",dentist,1975

"Taylor, Guy",musician,1972
```

The information about each person would be one Record, and the Record elements are delimited with a new line character. Each Record has three Field children, delimited by commas. The set of Record elements is a RecordSet.

A dictionary file will have the following general node structure, which you will customize to conform exactly to the flat text structure you need:

```
<FlatDataDef>
<RecordSet>
<Record>
<Field/>
<Field/>
...
</Record>
...
<RecordSet>
<Record>
<Field/>
<Field/>
...
</Record>
...
</RecordSet>
```

```
...
</RecordSet>
...
<RecordsDef>
<Record>
<Field/>
...
</Record>
...
</RecordsDef>
</FlatDataDef>
```

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<FlatDataDef name="contacts_schema" recSep="\N" fieldSep="" escape=""
description="Schema for contacts flat text">
<RecordSet name="contacts" description="Company employee data transaction">
<Record ref="title_comment_rec" />
<RecordSet name="contact_transaction" description="Contact data transaction">
<Record name="contact_mark">
<Field name="mark" matchedValue="[contact]" length="9" />
</Record>
<RecordSet name="contact_data" sequence="false">
<Record name="name_rec">
<Field name="name_mark" matchedValue="name=" length="5" ignore="true" />
<Field name="name" length="25" />
</Record>
<Record name="comment_rec" index="false" ignore="false">
<Field name="comma" matchedValue=";" length="1" ignore="true" />
<Field name="comment" length="30" />
</Record>
<Record name="email_rec">
<Field name="email_mark" matchedValue="email=" length="6" ignore="true" />
<Field name="email" length="40" />
</Record>
<Record name="phone_rec">
```

```
<Field name="phone_mark" matchedValue="phone=" length="6" ignore="true" />
<Field name="phone" length="20" />
</Record>
</RecordSet>
</RecordSet>
</RecordSet>
<RecordsDef>
<Record name="title_comment_rec">
<Field name="comma" matchedValue=";" length="1" />
<Field name="comment" length="30" />
</Record>
</RecordsDef>
</FlatDataDef>
```

Flat Text dictionary XML declaration

As with all XML documents, the top line of your file must be the XML declaration. Simply insert the following as your first line of text:

```
<?xml version="1.0"?>
```

Everything below this line will be your structure of nodes.

Flat Text dictionary comments

You may find it useful to insert comments into your dictionary file as you write it for later reference. These take the form

```
<!-- My comment here... -->
```

When inserted, comments do not affect your overall flat text structure. As in any XML document, a comment must begin with <!-- and end with -->, and may contain any sequence of characters except the double hyphen (--) which is only permitted to close the comment.

FlatDataDef node

Child of	None.
Parent of	One or more RecordSet children, zero or more RecordsDef children.

This node will be at the top level of your structure's hierarchy.

The FlatDataDef node has one required attribute: **name**. The FlatDataDef has several optional attributes: **recSep**, **fieldSep**, **escape**, **description**, and **fieldAutoTrim**. If the flat text structure you are dealing with is determined by field lengths, you will only use the **name** attribute and possibly the **description** attribute.

name	The name for your flat text structure. This name will appear at the root of your structure tree.
recSep	(optional) The character(s) that act as the delimiter between records in your flat text files, often a new line character (\N).
fieldSep	(optional) The character(s) that act as the delimiter between fields in your flat text files.
escape	(optional) The character used in your flat text file to escape characters with special meaning. For example, suppose your field separator is a comma but the value of one of your fields is Smith, Jane. You do not want Data Transformation Engine to mistake this for two fields. Using quotation marks (“”) as your escape character and entering “Smith, Jane” as your field value would solve this problem.
description	(optional) Gives a description of the flat text transaction.
fieldAutoTrim	(optional) Trims the trailing whitespace off of the fields when reading.

Example

```
…
<FlatDataDef name="contacts_schema" recSep="\N" fieldSep="" escape=""
description="Schema for contacts flat text">
…
</FlatDataDef>
```

RecordSet node

Child of	FlatDataDef element or RecordSet element.
Parent of	Zero or more RecordSet children, zero or more Record children.

The RecordSet node gives you a means of grouping Record nodes together for looping purposes. A RecordSet may be nested within another RecordSet node.

RecordSet has one required attribute: **name**. Optionally, you may add four attributes: **sequence**, **description**, **required**, and **maximumUse**.

name	The name of this set of records. This name will appear in your structure tree in the Designer.

sequence	(optional) Whether or not the records belonging to the RecordSet are in the order as laid out in the dictionary file. Takes the values **true** and **false**.
description	(optional) Gives a description of the RecordSet.
required	(optional) The requirement of the field. This may take the value **M** (mandatory) or **O** (optional). Default value is **O**.
maximumUse	(optional) The upper limit to the number of RecordSets to be produced. Default value is **0**, which means no limit.

Example

```
…
<RecordSet name="contacts" description="Company employee data transaction">
…
<RecordSet name="contact_transaction" description="Contact data transaction">
…
<RecordSet name="contact_data" sequence="false">
…
</RecordSet>
</RecordSet>
</RecordSet>
…
```

RecordsDef node

Child of	FlatDataDef element.
Parent of	Zero or more Record children.

This node is used to define Record elements. You may define Record elements here and then reference them as many times as is required within any RecordSet nodes. Using the RecordsDef is never necessary when writing a dictionary file, but can be very useful if you have a Record node that appears several times at different points in your structure.

No attributes are required for the RecordsDef element. You may provide the optional attribute **description**.

Example

```
…
<RecordsDef>
```

```
<Record name="title_comment_rec">
<Field name="comma" matchedValue=";" length="1" />
<Field name="comment" length="30" />
</Record>
</RecordsDef>
…
```

Record node

Child of	RecordSet element or RecordsDef element.
Parent of	Zero or more Field children.

One Record is a collection of related or grouped Field elements. This element can be a child of either the RecordSet or RecordsDef element. You only put Record elements in the RecordsDef portion in order to pre-define them for use within the RecordSet section.

If you are going to pre-define a Record element in the RecordsDef portion, the Record definition under the RecordsDef parent has one required attribute: **name**. The corresponding Record element under a RecordSet parent has one required attribute, **ref**, whose value is the name of the Record defined in the RecordsDef section. A Record element that is simply a reference to another should have no other attributes except **ref**.

If you are simply inserting a Record under a RecordSet element without pre-defining it in the RecordsDef portion, the Record has one required attribute: **name**.

Optionally, you may add five attributes: **description**, **ignore**, **index**, **required**, **maximumUse**, **instance**, **start**, and **end**. When a default value is set, these attributes do not appear in the configuration file.

name	The name of the Record. This name will appear in your structure tree.
ref	The name of a Record defined in the RecordsDef portion of your dictionary file. This references the other Record and uses its definition.
description	(optional) Gives a description of the Record.
ignore	(optional) Specifies whether or not this element is to appear in the structure tree. Takes values of **true** and **false**. Default value is **false**.

index	(optional) Whether or not this Record element can loop. That is, whether or not it can appear multiple times within its parent RecordSet. A value of true means no looping, while false means it can loop. Default value is **false**.
required	(optional) The requirement of the field. This may take the value **M** (mandatory) or **O** (optional). Default value is **O**.
maximumUse	(optional) The upper limit to the number of Records to be produced. Default value is **0**, which means no limit.
instance	(optional) The file name attribute that is metadata for the flat text importer wizard.
start	(optional) The start character that is metadata for the flat text importer wizard.
end	(optional) The end character that is metadata for the flat text importer wizard.

Example

```
…
<Record ref="title_comment_rec" />
…
<Record name="contact_mark">
…
</Record>
…
<Record name="name_rec">
…
…
</Record>
…
```

Field node

Child of	Record element.
Parent of	None.

These elements are the part of your structure that define the actual values that will be seen in the flat text transactions. Field takes one required attribute: **name**. Optionally, there are five attributes: **matchedValue**, **ignore**, **length**, **fillerChar**, **align**, **dataType**, and **range**. The last three are only used if your flat text transaction is structured by field lengths.

name	The name of the Field. This name will appear in your structure tree.
matchedValue	(optional) The exact value of this Field, if you know it ahead of time. Supports regular expressions.
ignore	(optional) Specifies whether or not this element is to appear in the structure tree. Supports values of **true** and **false**.
length	(optional) The number of characters in this Field, if your flat text transaction is structured by length.
fillerChar	(optional) If your flat text transaction is structured by length, and there are certain fields in which the value of the Field does not completely fill the length given to the Field, this attribute specifies the character used to fill the extra space.
align	(optional) If your flat text transaction is structured by length, and there are certain fields in which the value of the Field does not completely fill the length given to the Field, this attribute specifies where the value should be positioned within the Field. This can take the values `left`, `right` and `middle`.
dataType	(optional) The attribute set to indicate validation. It is either set to **AN** (Alphanumeric) or **N** (Numeric).
range	(optional) The attribute set if the dataType attribute is **N** (Numeric). Range options are **<x**, **>x**, or **x-y**.

Example

```
…
<Field name="mark" matchedValue="[contact]" length="9" />
…
…
…
<Field name="name_mark" matchedValue="name=" length="5" ignore="true" />
<Field name="name" length="25" />
…
```

9.1.5 Flat Text Importer Wizard

Use the **Flat Text Importer Wizard** to create a simple **flat text dictionary**.

1. From the “Source Configuration” on page 51 tab or the “Target Configuration” on page 74 tab, select **Flat Text** as the data format. The **Importer Wizard** button appears at the bottom of the pane.
2. Select the **Validating** check box. Flat text will validate on whether the data type is alphanumeric (**AN**) or numeric (**N**). If numeric, it will validate via a number range, such as **<x**, **>x**, or **x** is within a numbered range **x-y**.
3. Click the **Importer Wizard** button.
4. Choose a **structure** option from the list below:
    - **Open existing structure file**.
    - **Create structure file by simple instance**. A “Simple Instance” on page 248 means creating a structure file for one record.
    - **Create structure file by complex instance**. A “Complex Instance” on page 250 means creating a structure file for multiple records.

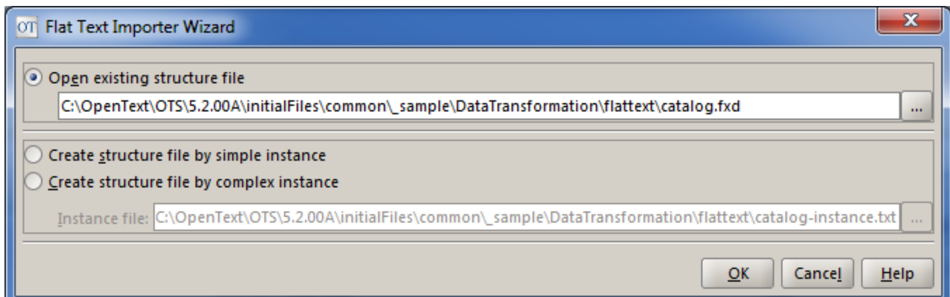


Figure 9-1: Flat Text Importer Wizard

5. Specify a **filename** under which to save the **Flat Text** dictionary.
6. Click **OK**.

   The “Flat Text Structure Editor” on page 246 opens.

9.1.5.1 Flat Text Structure Editor

The Flat Text Structure Editor is where you edit the tree structure for a flat text dictionary file.

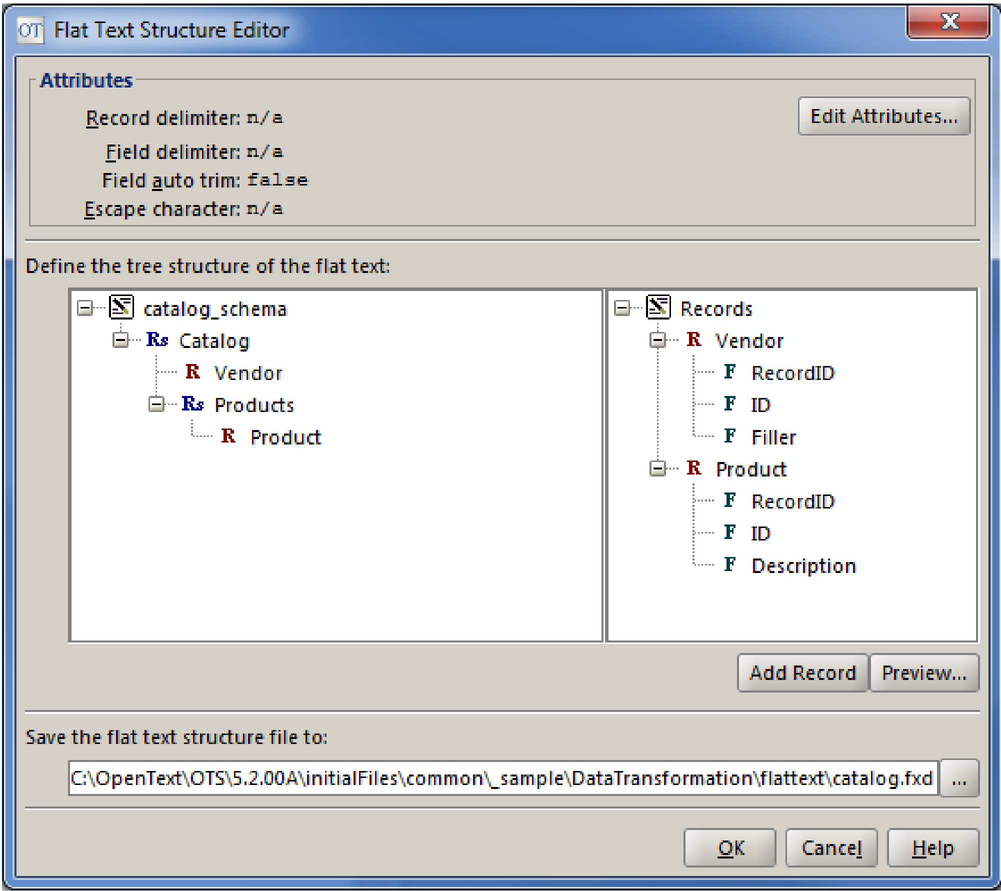


Figure 9-2: Flat Text Structure Editor dialog

Edit Attributes

Click the **Edit Attributes** button to open the **Edit Attributes** dialog.

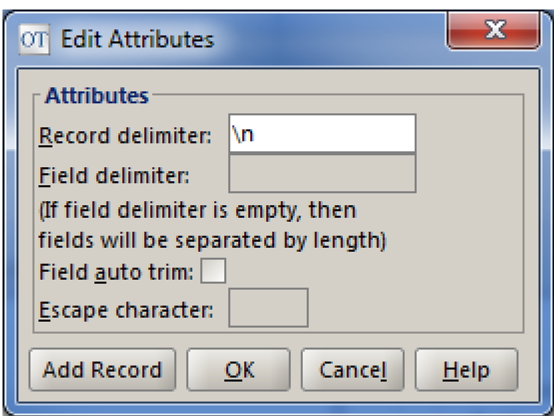


Figure 9-3: Edit Attributes dialog

When the **Edit Attributes** dialog opens, you can:

- change the record delimiter.
- change the field delimiter. For example, use a semicolon instead of a pair of single quotation marks. If there is no **Field Delimiter** set, Output Transformation Designer will use **Length** to delimit fields instead. Doing this also causes the Alignment, Filler, and Length columns to appear in the Flat Text Structure Record.
- set the field auto trim parameter. If selected, trailing whitespace will be removed from the value of the field.
- set the escape character. This character is used in your flat text file to escape characters with special meaning. For example, suppose your field separator is a comma, but the value of one of your fields is Smith, Jane. You do not want Data Transformation Engine to mistake this for two fields. Using quotation marks ("") as your escape character and entering "Smith, Jane" as your field value would solve this problem.

9.1.5.1.1 Tree Structure Definition pane

In the tree structure pane, the left side represents the **Flat Text Tree Structure** and the right side represents the **Record Field Definitions**.

Drag and drop a record from the right side to a record set on the left side; alternatively, you can copy a record from the right side and paste it to a record set on the left side.

Flat Text tree structure

The Flat Text Tree Structure is located on the left side of the Flat Text Structure Editor.

Right-click a **node**:

- Select **Add Child** to add a new record set.
- Select **Remove Child** to delete an existing record set.

Record Field definitions

The Record Field Definitions are located on the right side of the Flat Text Structure Editor.

Right-click **Records** on the right side of the Flat Text Editor:

- Select **More Records** to define more records.
- Right-click on any record and select **Properties** to display the **Flat Text Structure Record** window.

9.1.5.2 Flat Text Structure Record

The Flat Text Structure Record is where you edit a complex instance of a record in a flat text file. You can select a record and determine the length of its fields, specify common field attributes, set data type and range validation attribution, and create nested fields.

9.1.5.2.1 Simple Instance

A simple instance means creating a flat text structure file for one record. The **Flat Text Structure by Simple Instance** dialog displays the simple instance of the records by field.

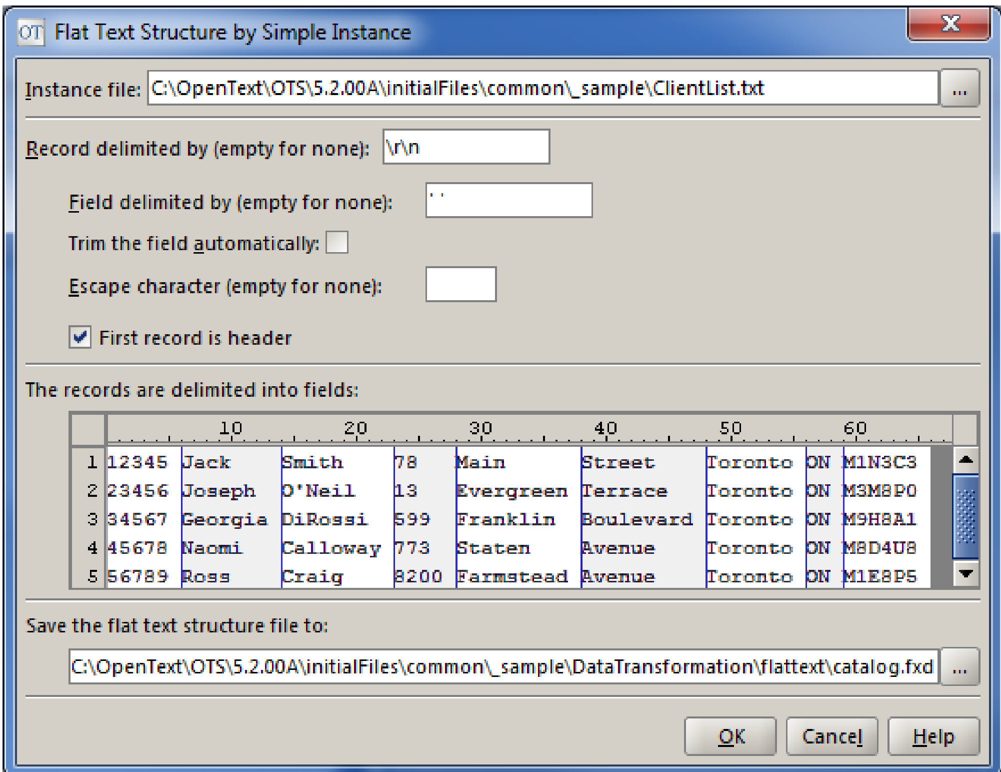


Figure 9-4: Flat Text Structure by Simple Instance with delimiter attribute set

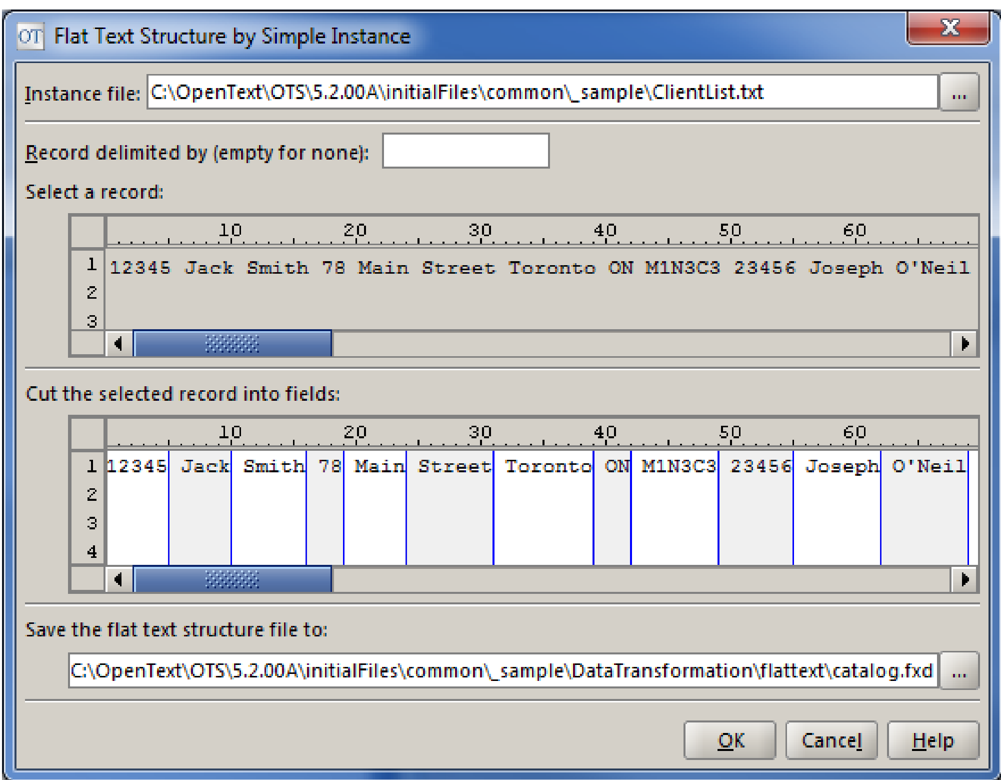


Figure 9-5: Flat Text Structure by Simple Instance without delimiter attribute set

9.1.5.2.2 Complex Instance

A complex instance means creating a flat text structure file for multiple records. The **Flat Text Structure Record** dialog displays the complex instance of multiple records in a flat text dictionary file.

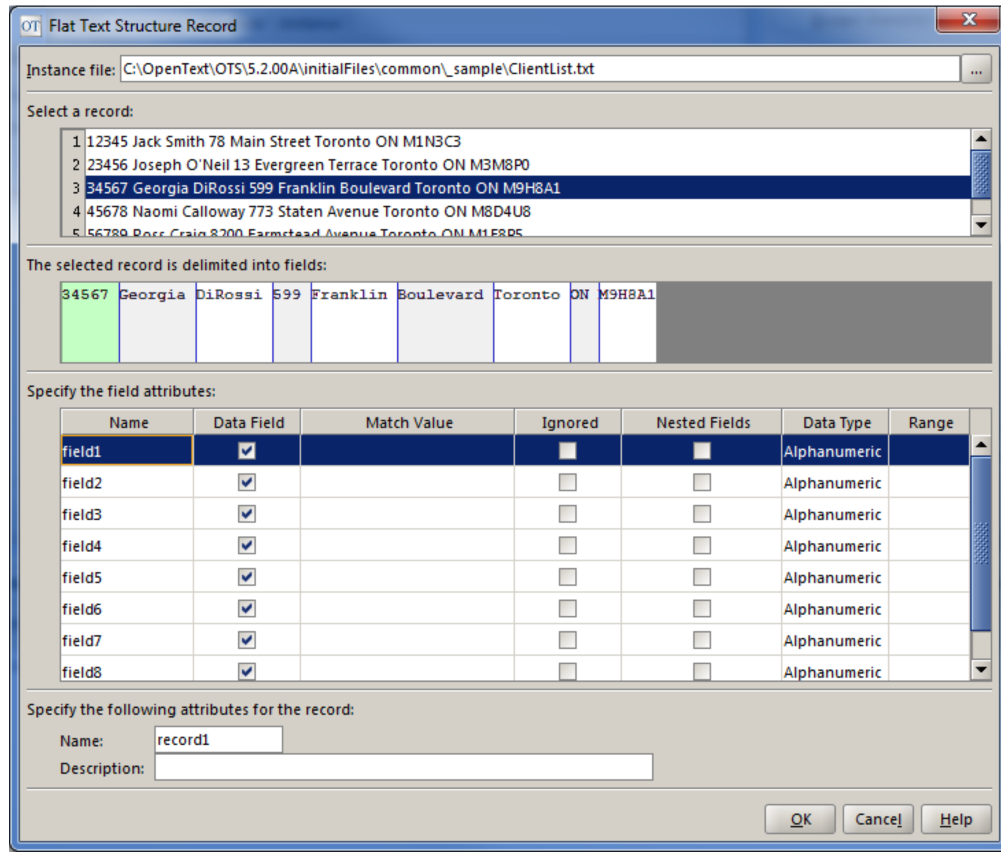


Figure 9-6: Flat Text Structure Record with field delimiter attribute set

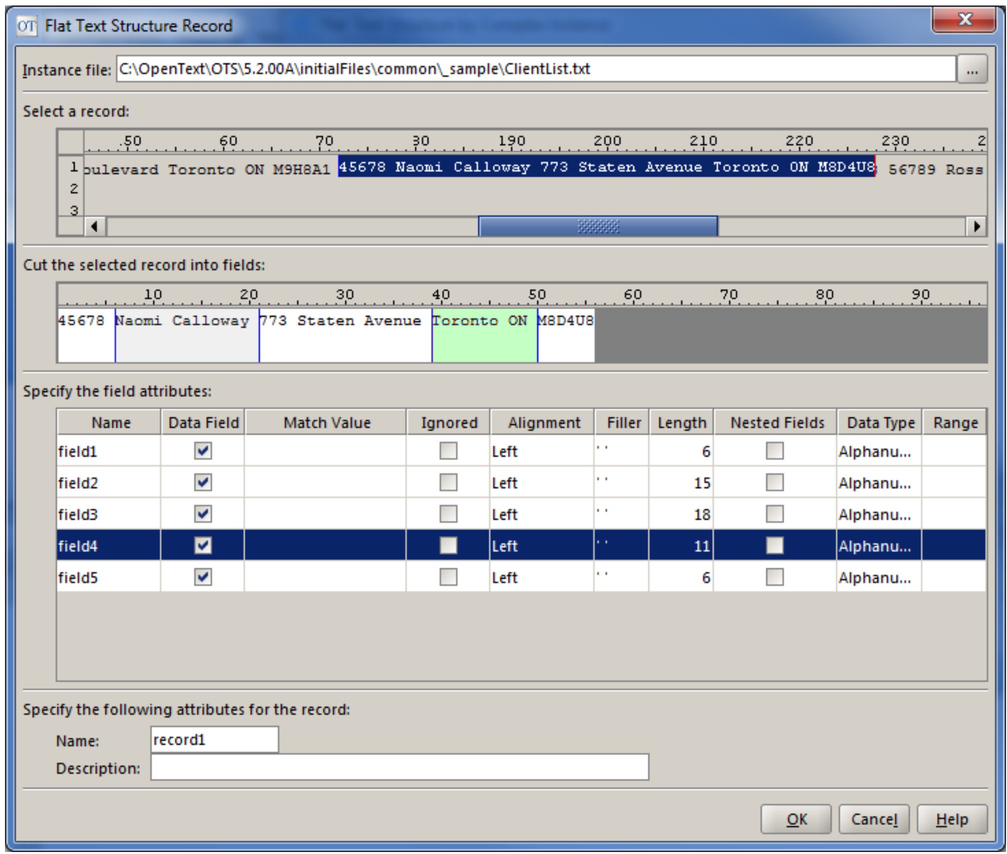


Figure 9-7: Flat Text Structure Record without the field delimiter attribute set

Note: You can only modify the field's length (see the vertical lines) if you have not entered anything in the Field Delimiter attribute.

9.1.5.2.3 Selecting a Record

Under the **Select a record** section, choosing a record displays its attributes in the **Specify the field attributes** section below. If you have a delimiter attribute set, you can simply click one of the records. If you do not have a delimiter attribute set, you must highlight a portion of the record from the input file to select the record. For more information on the **Specify the field attributes** section, see “Specifying Field Attributes” on page 253.

For a selected record, you can:

- break down the record into multiple fields by adjusting the vertical lines in between fields under **Cut the selected record into fields**.
- delete a vertical line by double-clicking on it.
- delete all lines by right-clicking and selecting **Clear All**.

- add a new vertical line by left-clicking on the desired position.
- reposition a vertical line by left-clicking and dragging it to the desired position.

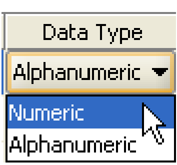
9.1.5.2.4 Specifying Field Attributes

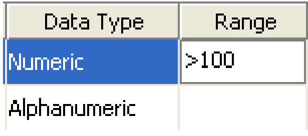
Each row in the table within the **Specify the field attributes** section corresponds with a field from your selected record. In this section, you can also:

- rename the field by double-clicking on the field attribute you want to change in the **Name** column.
- mark the field as a **Matched Value** or a **Data Field**.
- indicate whether or not a field will appear in the structure tree by clicking the check box in the **Ignored** column of the corresponding field.
- set the alignment by double-clicking the field in the **Alignment** column. A drop-down menu appears allowing you to select from **Left**, **Center** and **Right** justification options.
- input a character for the **Filler** attribute. If your flat text transaction is structured by length, and there are certain fields in which the value of the field does not completely fill the length given to the field (as defined by the **Length** attribute), the character inputted will fill in the extra spaces.
- indicate the number of characters in the field in the **Length** attribute.

Note: The **Alignment**, **Filler**, and **Length** columns appear only if you want to define your fields by length and if you leave the **Field Delimiter** attribute empty.

- validate your record by setting the **Data Type** attribute. If the data type is Alphanumeric (**AN**), then no **range** needs to be set for validation. If the data type is Numeric (**N**), then set **<x**, **>x**, or **x-y** as the range.





9.1.5.3 Nested fields

A nested field defines a field in more detail than the field's definition shows on its flat text structure record. For example, a field named Name may include a customer's first and last name. However, if defined, Name may also have two nested fields: First Name and Last Name. Nested fields have the same attributes as fields, but give a more precise field definition. For more information, see “Field node” on page 243.

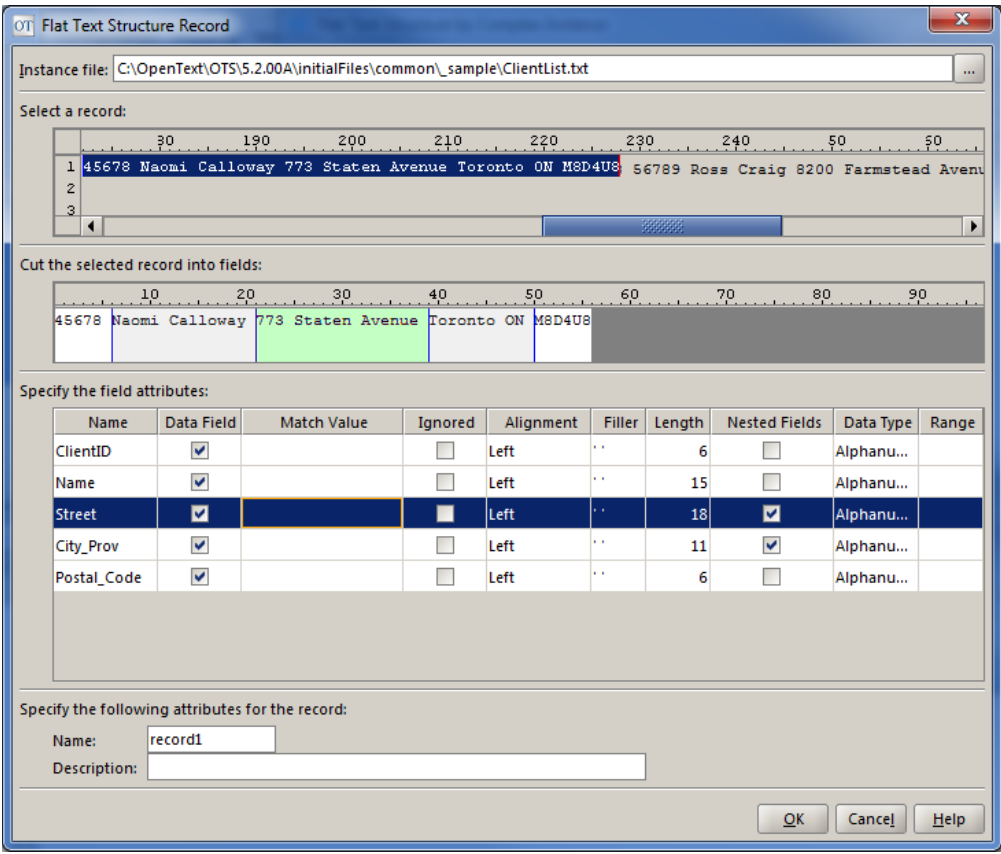


Figure 9-8: Nested field data on Flat Text Structure Record dialog

To add a nested field, double-click the unchecked cell in the **Nested Fields** column of the corresponding field and define its attributes in greater detail. The **Edit Nested Field** dialog opens.

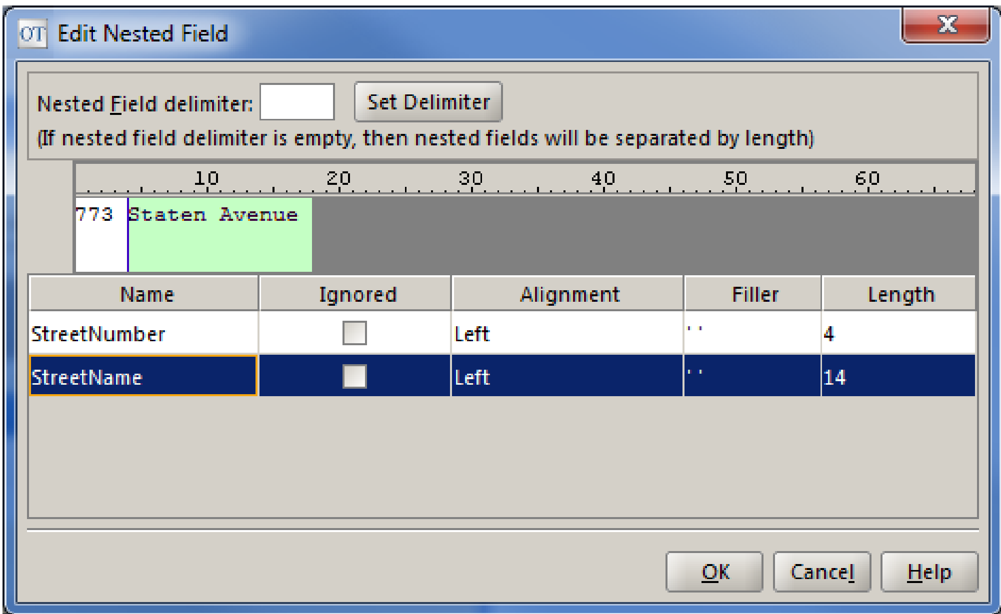


Figure 9-9: Edit Nested Field dialog

The **Nested Field delimiter** box sets the character(s) that act as the separator between nested fields. If a delimiter is not chosen, the nested fields will be separated by length. To set a delimiter:

1. Type the character(s) you want to use as the delimiter in the Nested Field delimiter box. The box sets the character(s) that act as the separator between nested fields. If a delimiter is not chosen, the nested fields will be separated by length.

2. Click **Set Delimiter**.

3. A **Designer Question** dialog box appears asking you to confirm your selection. Click **Yes** to proceed, or **No** to restore the original value.

   The Edit Nested Field dialog box works similar to the **Flat Text Structure Record** where you can:

   - Break down the record into multiple fields.
   - Edit the name of nested fields.
     - Ignore the field.
     - Align the field to the left, right or center.
     - Specify a filler character.
     - View a field's length.

4. Click **OK** to save your changes. The Flat Text Structure Record dialog returns with the Nested Fields check box selected.

To edit or remove a nested field, click the selected **Nested Field** you want to edit or remove and, when the **Designer Question** dialog box appears asking you to confirm your selection, click **Yes** to proceed or **No** to edit the value.

9.2 Validating dictionary files

Validating a dictionary file means that you are verifying that the structure of the input or output data transformation file matches the structure of the XML dictionary file it maps to. It validates based on the input or output data format against the dictionary.

The following input or output data formats need validation against a dictionary file:

- "CSV Input Data Format" on page 55 or "CSV Output Data Format" on page 77
- "EDI X12 Input Data Format" on page 56 or "EDI X12 Output Data Format" on page 81
- "EDI HIPAA Input Data Format" on page 56 or "EDI HIPAA Output Data Format" on page 81
- "EDIFACT Input Data Format" on page 55 or "EDIFACT Output Data Format" on page 82
- "HL7 Input Data Format" on page 57 or "HL7 Output Data Format" on page 79
- "Flat Text Input Data Format" on page 63 or "Flat Text Output Data Format" on page 80
- "XML Input Data Format" on page 54 or "XML Output Data Format" on page 76

To validate against a dictionary file, select the **Validating** check box that appears on the Source or Target Configuration tabs when you select one of the data formats above.