Chapter 6. June 2007

Welcome to the June 2007 edition of IBM Information Server Developer’s Notebook. This month we received one question in particular regarding XSLT (Extensible Style Language -Transforms), and we will use this deeper question to host a discussion of all things related to XML. This month we answer the following;

– What is XML, and what business value can I derive from it ?

– What are XSLT, XPath Expressions, XML Schema, and related ?

– How, Why, and When do I use DataStage/QualityStage (XML Input, XML Output, and XML Transform) ?

– How does Rational Data Architect figure into all of this ?

– And more-

In short, we are going to move data to and from relational and XML, model XML Schemas, and perform XML Style Sheet Transforms. These solutions involve many detailed steps and for that reason, we are going to divide the solutions offered into a number of distinct sub-tasks.

Software Versions, IBM Information Server

All of these solutions were *developed and tested* on IBM Information Server version 8.01, on the Microsoft Windows XP/SP2 platform. IBM Information Server allows for a single, consistent, and accurate view of data across the full width of the corporate enterprise, be it relational or non-relational, staged or live data. As a reminder, the IBM Information Server product contains the following major components;

> WebSphere Business Glossary AnyWhere™, WebSphere Information Analyzer™, WebSphere Information Services Director™, WebSphere DataStage™, WebSphere QualityStage™, WebSphere Metadata Server and Metabridges™, WebSphere Metadata Workbench™, WebSphere Federated Server™, Classic Federation™, Event Publisher™, and Replication Server™, and Rational Data Architect™.

Obviously, IBM Information Server is a large and capable product, addressing many strategic needs across the enterprise, and supporting different roles and responsibilities. This month we give focus to DataStage/QualityStage (Enterprise Edition), and Rational Data Architect components of IBM Information Server.

6.1 What is XML, and Why Do I Care ?

By means of analogy, consider the following;

Imagine you had to create a brand new on line transaction processing business application. This application is defined to contain 100 data entry screen forms, 100 printed reports, and a number of batch jobs. In every data entry screen form, you have to responsibility to deliver a *current list management* capability; that is, give the end user some means query, browse, create, update, and delete data. A given SQL table may contain millions of data records; which record or records does the end user currently want to update, delete, or output somewhere- What is your multi-user concurrency model, and how do you react to runtime errors, foreign characters sets, and also do all of this in a highly productive and maintainable manner-

Most business application programming environments will give you dynamic arrays, access to abstract and also distinct SQL data manipulation statements, error recovery, an ability to promote re-usability through library program code, and more. If you are really, really good, you create one master or template type data entry screen form, and that allows you to create the remaining 99 data entry screen forms more easily. That one master program is ten times harder to create than the stand alone programs, but it offers standardization (at least within your organization), higher productivity after its creation, and more.

Now imagine that your business application development tool gave you that current list management object, and that this smart object was an industry standard; open, extensible, and was already integrated into your entire Information Technology landscape. *XML is that smart object.* ASCII text files are the writing of 100 stand alone programs, or at best, writing the one master program, that is then proprietary to your organization.

XML Compared to other Technologies

XML (Extensible Markup Language) is based on the same parent technology as HTML (HyperText Markup Language). HTML tags are formatting (presentation) related, and do not give information about the *content* of a Web page. For this reason, it is hard to re-use the data contained in a given Web page; that data is largely only available for display.

XML looks a lot like HTML; each language having beginning and ending tags, modifying attributes, literal and derived values, and more. Where HTML is meant to describe Web pages, XML is meant to describe other languages, including the ability to describe data. XML tags identify data, and make that data available for integration and re-use by other business processes.

Still not sold on XML? Imagine if every file and spreadsheet you ever saw had a built in SQL database capability, could re-sort, output a report, or connect to other objects and work cooperatively with them- XML offers that.

Like every progressive area of Information Technology, XML has its own taxonomy of terms and objects, published APIs, standards bodies, and so on. Rather than review those items now, we will instead learn them *by example*, by their demonstrated use. In the next section of this document, we will use IBM Information Server DataStage/QualityStage to take a relational data source and output it in an XML format.

6.2 DataStage/QualityStage EE Job, XML Output

This section of this document was *developed and tested* using WebSphere DataStage/ QualityStage Enterprise Edition version 8.01 on the Microsoft Windows XP/SP2 platform. The steps outlined in this section *may work* on DataStage Server Edition 8.x or 7.x with only minor modification. Other conditions and assumptions are listed below:

– With this document being a periodic technical journal on IBM Information Server, it is assumed that the reader has at least basic proficiency with WebSphere DataStage/QualityStage.

– While we are discussing the example of taking a SQL-based relational data set and formatting it in XML, the example below will actually read from a (flat) sequential (ASCII text) file. Since most relational data sets are generally flat, we make this modification to simplify this example.

– The data set referenced in this section could be modeled as the relational table Customers, with three columns, CustomerNum, CompanyName and StateCode. For simplicity, all columns are of type String/Character.

  - Example data for CustomerNum includes; “101”, “102”, “103”, etcetera.
  - Example data for CompanyName includes; “Bay Sports”, “AA Athletics”, “Sports Spot”, etcetera.
  - Example data for StateCode includes; “CA”, “AZ”, “NJ”, etcetera.

– While the name Customers refers to the relational table Customers, or entire or subset of the Customers Table, the name Customer refers to one record within the Customers table, one instantiation of a record inside that table.

  The full hierarchy of objects in this stack is stated as: Customers => Customer => CustomerNum, CompanyName and/or StateCode.

– A resource dependency is used in this section that is not fully resolved until, Section 6.4, “Creating the XML Schema Definition used above” on page 36. This resource dependency is imported as a DataStage/ QualityStage Table Definition.

Figure-1, below, displays the DataStage/QualityStage Enterprise Edition Job that we will create to take a given data set and format it in XML.

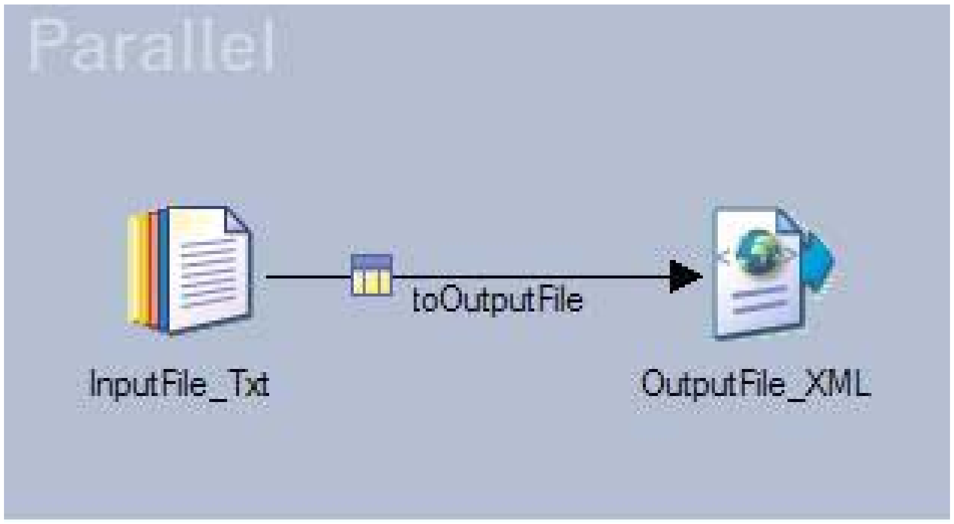


Figure-1, DataStage/QualityStage EE Job, Relational to XML.

Before we create the DataStage/QualityStage job above, we need to create two ASCII text files that we have dependencies with. One of the ASCII text files will serve as our input data source, (Customers.txt). The second ASCII text file is an XML Schema Definition (XSD) file, (Customers.xsd). In Section 6.4, “Creating the XML Schema Definition used above” on page 36, we will create the contents of (Customers.xsd) entirely by painting it with the XML Schema Editor, a graphical tool found to exist in the Rational Data Architect component of IBM Information Server. (Customers.xsd) is less than 27 lines long. For now we suggest you just create it by cutting and pasting from this document, or by using a text editor.

[Begin] Complete the following steps:

1. Create an ASCII Text input data file of a given format.

   Using a text editor or any other compatible method, create the (Customers.txt) ASCII text file. See Figure-2, below.

a. We used Notepad and a vertical bar delimiter (ASCII 127) as a field delimiter.

b. For best effect, the first column should contain unique values, a Customer.CustomerNum.

c. For best effect, the second column should contain unique values, a Customer.CompanyName.

d. For best effect, the third column should contain duplicate values, a Customer.StateCode.

e. For best effect, the rows should be in random sort order.

f. There should be no extra blank lines in this file.

2. Create an ASCII Text file in XML Schema Definition (XSD) file format.

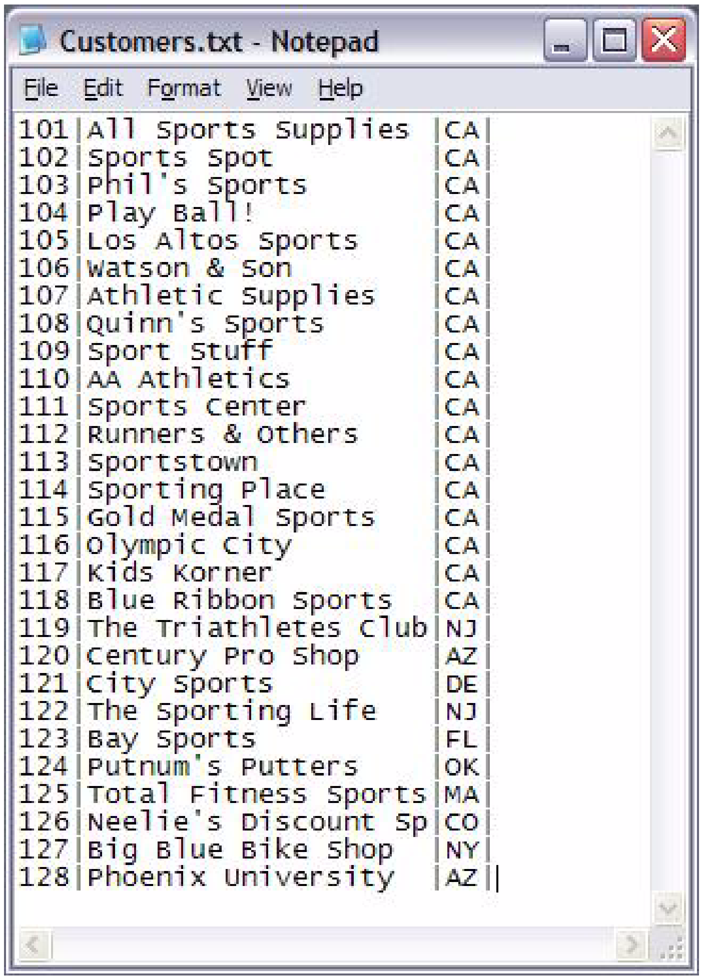
   Using a text editor or any compatible method, create the (Customers.xsd) ASCII text file. See Figure-3, below.

   a. For now we are creating this file manually. Later we will create this file via automatic means, via a graphical tool.

   b. Do not enter the line numbers displayed in this file; “001”, “002”, “003”, etcetera. These lines numbers are for our instructional purposes only.

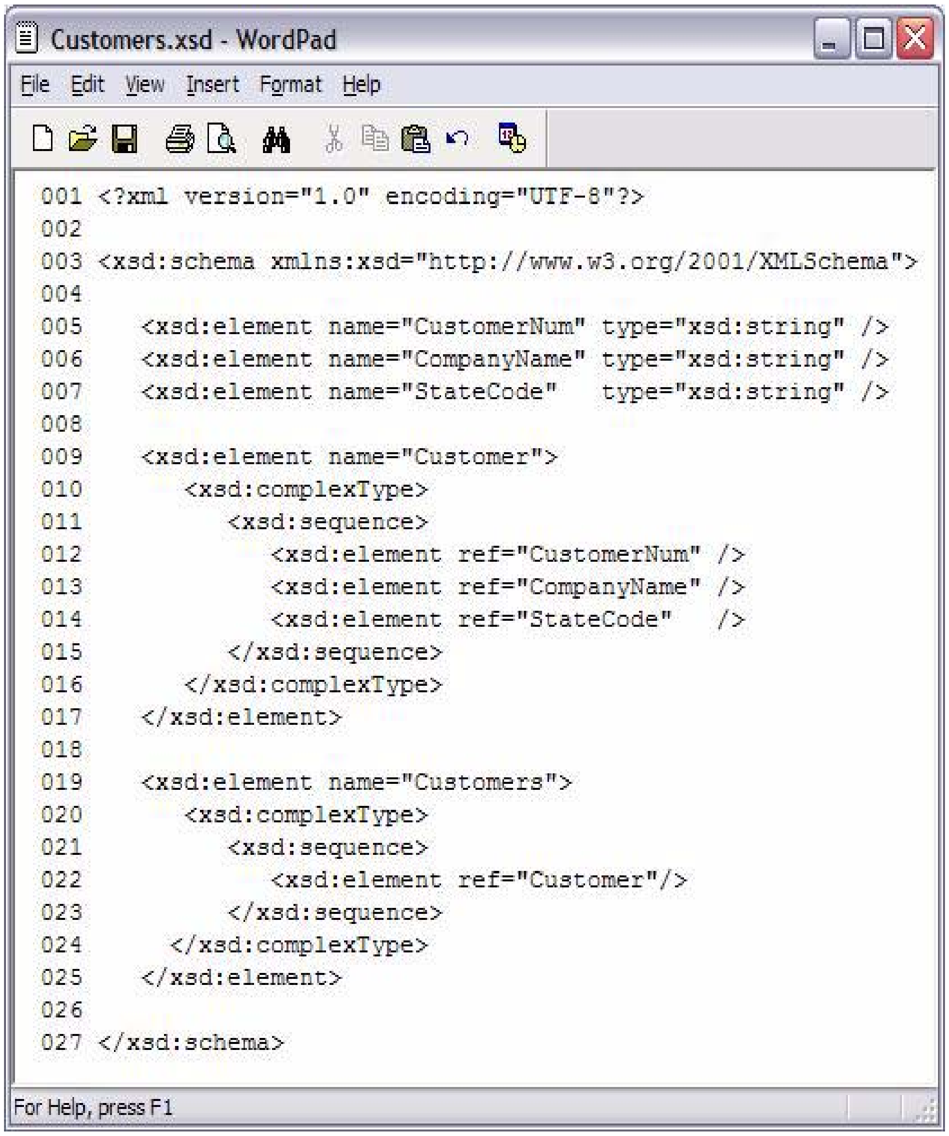
   c. A line by line code review explaining the contents of the (Customers.xsd) file appears later in this section.

[End] Complete the following steps:



```
101|All Sports Supplies |CA|
102|Sports Spot         |CA|
103|Phil's Sports       |CA|
104|Play Ball!          |CA|
105|Los Altos Sports    |CA|
106|Watson & Son        |CA|
107|Athletic Supplies   |CA|
108|Quinn's Sports      |CA|
109|Sport Stuff         |CA|
110|AA Athletics        |CA|
111|Sports Center       |CA|
112|Runners & Others    |CA|
113|Sportstown          |CA|
114|Sporting Place      |CA|
115|Gold Medal Sports   |CA|
116|Olympic City        |CA|
117|Kids Korner         |CA|
118|Blue Ribbon Sports  |CA|
119|The Triathletes Club|NJ|
120|Century Pro Shop    |AZ|
121|City Sports         |DE|
122|The Sporting Life   |NJ|
123|Bay Sports          |FL|
124|Putnum's Putters    |OK|
125|Total Fitness Sports|MA|
126|Neelie's Discount Sp|CO|
127|Big Blue Bike Shop  |NY|
128|Phoenix University  |AZ|
```

Figure-2, (Customers.txt) ASCII text file, to be used as an input data source.



```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <xsd:element name="CustomerNum" type="xsd:string" />
   <xsd:element name="CompanyName" type="xsd:string" />
   <xsd:element name="StateCode"   type="xsd:string" />

   <xsd:element name="Customer">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="CustomerNum" />
            <xsd:element ref="CompanyName" />
            <xsd:element ref="StateCode"   />
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="Customers">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="Customer"/>
         </xsd:sequence>
     </xsd:complexType>
   </xsd:element>

</xsd:schema>
```

Figure-3, (Customers.xsd) ASCII text, an XML Schema Definition file.

The following is a line by line code review of the (Customers.xsd) XML Schema Definition (XSD) file:

- **Again; Do not enter the line numbers displayed in this file; "001", "002", "003", etcetera. These lines numbers are for our instructional purposes only.**
- Line-001, is technically optional, but should be considered mandatory.
  - Much like the first line of an HTML file, this line declares the (XML) Version and (Language) Encoding.
  - XML lines that begin with "<?" and end with "?>" are called Processing Instructions. Aside from the example of declaring the Version and (Language) Encoding, Processing Instructions are rarely seen or used in XML.
- Line-003, this line is mandatory inside every XSD file. This line is paired with (terminated by) line-027, and is called the "(XML Schema) Namespace Declaration".
  - The URL entered on this line (this is a standard/reference value), and its standard associated Tag Prefix of "xsd" (also standard), refers to the "Schema of Schemas".

    The "Schema of Schemas" is to XML and XSD files, what the Object Class Repository is to Java; it defines the API and objects contained in the core XML/XSD language specification.
  - In effect, this line is an *include*; meaning, there are items we refer to inside this file that are defined elsewhere; items like "xsd:string", which is a base data type for the XML Schema Definition Language.
  - "xsd:string" is one of about eight to ten base data types; date, time, decimal, integer, etcetera. As a language, XML/XSD is also extensible, and you can further enhance (extend) any of the base data types. This topic is not expanded upon further here.
  - "Namespaces" are a core concept to XML and are worthy of further study. For example, lines-005 thru 007 define (columns) we could place into a central repository to be re-used by our business. We would associate these resources with an "xsd" type Tag Prefix, and import them (allow reference to) with a line similar to line-003.
- Lines-005 thru 007 define three "Elements", which are specific to our application.
  - There are many adjectives XML, and XSD in particular, use to refine the categorization of an Element. There are Simple Elements and at least four types of Complex Elements.

  - On lines-005 thru 007 we define three Elements which are similar in use to SQL database columns. More accurately, you could think of these Elements as user defined data types. While they start as an XML/XSD base data type of “xsd:string”, we could add data integrity and validation rules to these Elements, thus extending their (base) definition.
- Line-009, defines another Element, which is also specific to our application. This line is paired with (terminated by) line-017.
  - This element is similar in use to an SQL record or row.
  - Line-010, terminated by line-016, allows this Element to contain other Elements; namely the (column like) Elements we defined on lines-005 thru 007.
  - Line-011, terminated by line-015, states that the Elements contained in this Element will appear in order by name when present. “sequence” is one of several XML/XSD keywords that may be placed here, each with one specific operational effect. “sequence” is very common in its use.
  - Lines-012 thru 014 make reference to the Elements defined on lines-005 thru 007 by use of the “ref=[Element Name]” Attribute. In this manner, we inherit (import) the definitions on lines-005 thru 007.
- Line-019.
  - By this point in the file we have essentially defined a SQL style record (an Element of complexType) with three columns (three embedded Elements).
  - With the XML/XSD source code we have entered thus far, we could create an XML data file that contained one record, a single Customer. In order that we may create an XML file with numerous Customer records, we have to define (model) that. We need to define another Element which is the list of numerous Customer records, namely Customers.
- Lines-019 thru 025 model the Customers Element.
  - This section of the XML/XSD source file is very similar to lines-009 thru 017. The only difference is that this Element contains reference to only one (embedded) Element, not three. The (embedded) Element happens to be an Element of complexType, that itself contains three Simple Elements).
  - Again, this block allows our XML data file to contain numerous Customer records, in a structure called Customers.

Thus completes our creation of a source data file for input, and the creation and understanding of an XML Schema Definition Language (XSD) file. Again, we could have and will paint this XSD file later in this document. The point in manually creating this file now was to hopefully simplify this example by removing external dependencies.

At this point we are ready to create our DataStage/QualityStage Job that reads from an input data source, and outputs an XML formatted document.

[Begin] Complete the following steps:

1. Launch the DataStage/QualityStage component of IBM Information Server, and connect to an available Project.
2. Import the XML Schema Definition (XSD) file that we created above.

   The first thing we wish to do is import our XML Schema Definition (XSD) file, (Customers.xsd). The result of this import will be a DataStage/QualityStage Table Definition.

   a. From the DataStage/QualityStage Menu Bar, select; Import => Table-Definitions => XML Table Definitions. Example as shown in Figure-4. This action will produce the (DataStage/QualityStage) Meta Data Importer, displayed in Figure-5.

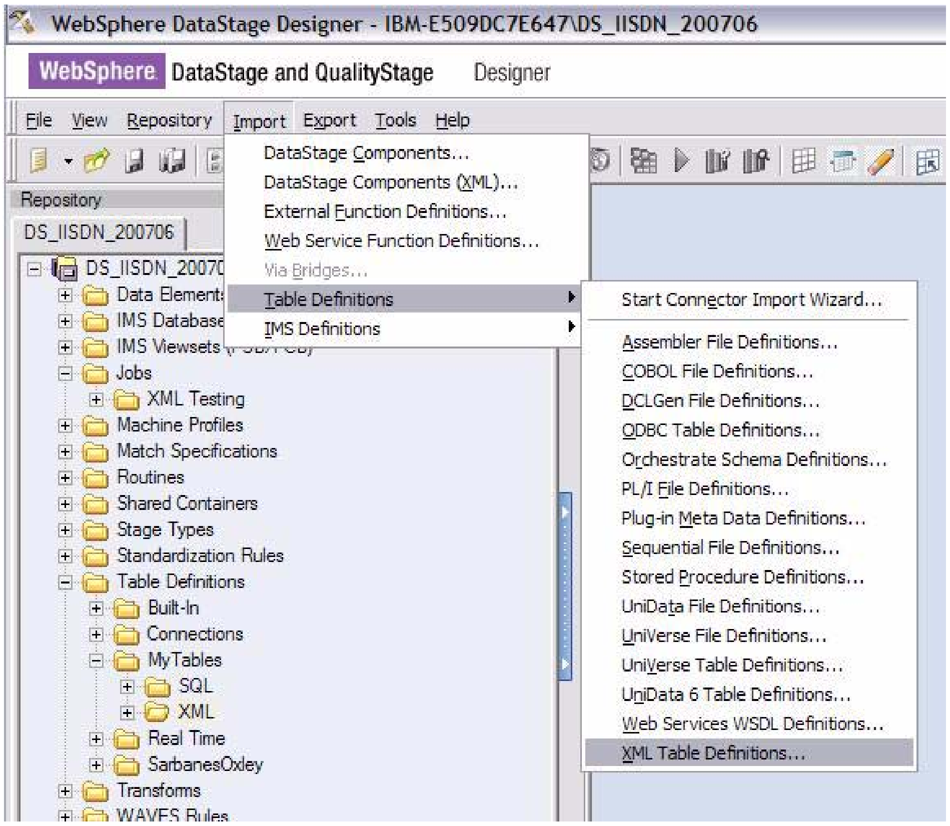
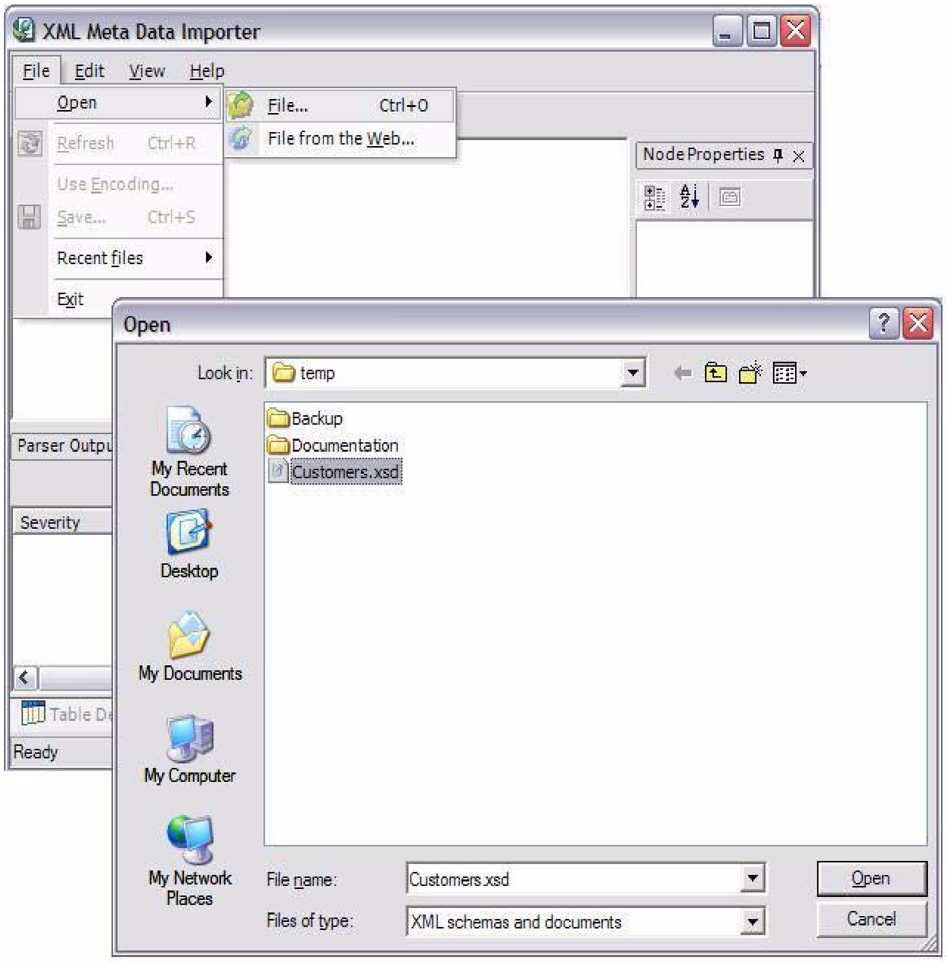

WebSphere DataStage Designer - IBM-E509DC7E647\DS_IISDN_200706
WebSphere DataStage and QualityStage Designer
File View Repository Import Export Tools Help
DataStage Components...
DataStage Components (XML)...
External Function Definitions...
Web Service Function Definitions...
Via Bridges...
Table Definitions
IMS Definitions
Start Connector Import Wizard...
Assembler File Definitions...
COBOL File Definitions...
DCLGen File Definitions...
ODBC Table Definitions...
Orchestrate Schema Definitions...
PL/I File Definitions...
Plug-in Meta Data Definitions...
Sequential File Definitions...
Stored Procedure Definitions...
UniData File Definitions...
UniVerse File Definitions...
UniVerse Table Definitions...
UniData 6 Table Definitions...
Web Services WSDL Definitions...
XML Table Definitions...
Repository
DS_IISDN_200706
Data Elements
IMS Databases
IMS Viewsets
Jobs
XML Testing
Machine Profiles
Match Specifications
Routines
Shared Containers
Stage Types
Standardization Rules
Table Definitions
Built-In
Connections
MyTables
SQL
XML
Real Time
SarbanesOxley
Transforms
WAVES Rules

Figure-4, Invoking DataStage/QualityStage XML Meta Data Importer.

b. From the DataStage/QualityStage XML Meta Data Importer Menu Bar, select; File => Open => File. Example as shown in Figure-5. This action will produce a (File) Open Dialog box.

c. Browse to the location where our previously created XML Schema Definition (XSD) file (Customers.xsd) resides. And click Open. This action produces the display in Figure-6.

   Notice that the Meta Data Importer can import XML Schema Definition (XSD) files *and* XML files. Using our XSD file now gives us an advantage over using any previously created XML (data file). We can import XML (data files) and reverse engineer them, however, using our XSD file saves us time.



Backup
Documentation
Customers.xsd

File name: Customers.xsd
Files of type: XML schemas and documents

Figure-5, Opening our previously created XML Schema Definition (XSD) file.

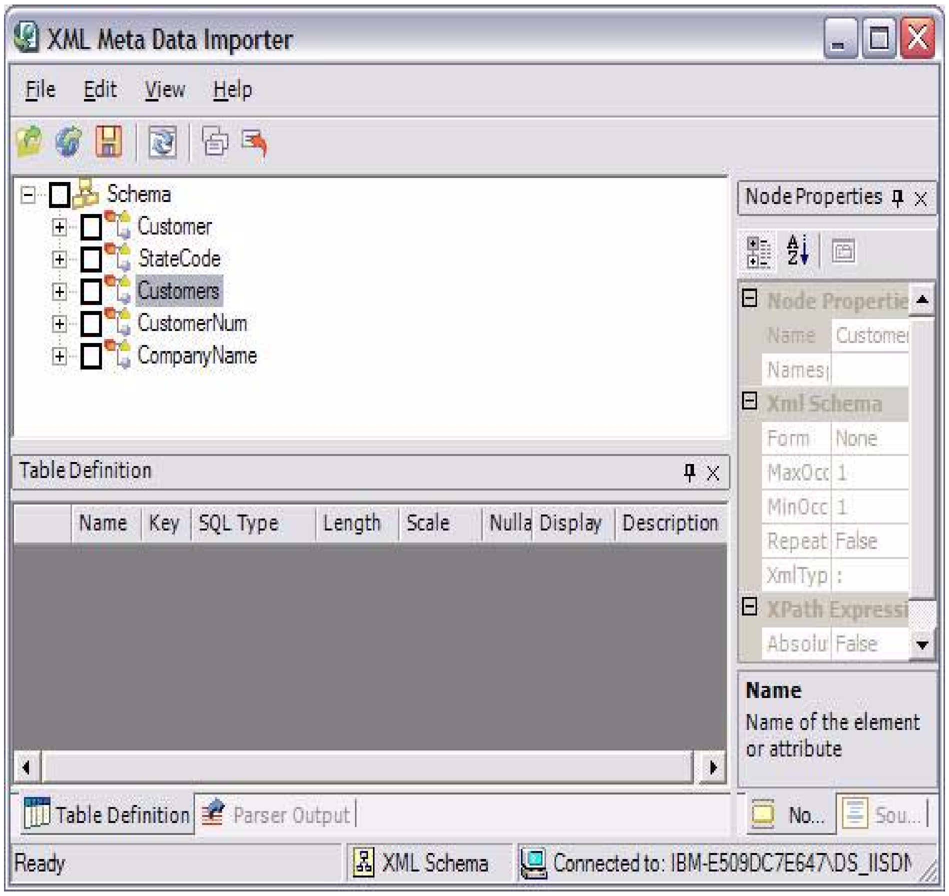


Figure-6, Basic presentation of DS/QS XML Meta Data Importer.

d. The upper left-most portion of the DS/QS XML Meta Data Importer display (Figure-6) is called the Tree View.

   Other views include; (XML/XSD) Source View, Node Properties View, (DS/QS) Table Definition View, Parser Output View.

   In our example, the Tree View is displaying the Elements we defined inside the (Customers.xsd) file.

e. Using the Tree view, manage the user interface so that you are able to Check the *three Data Type entries* for the CustomerNum, CompanyName and Statecode Elements.

For our example, we only want the lines with the “T:string” designator. This is the most common use. The other elements in this display serve to define the hierarchy of Customers => Customer => CustomerNum, CompanyName and StateCode. Example as shown in Figure-7.

Perhaps another way to reinforce why we are only selecting (checking) the Data Type entries for our desired Elements is to consider that we are moving from a hierarchical record model (XML) to one that is flat (relational data, a two dimensional presentation of data in Rows and Columns).

We will import the hierarchical knowledge as metadata, but the new model is flat.

Figure-8 displays changes that will automatically be made in the Table Definition View as we work in the Tree View.

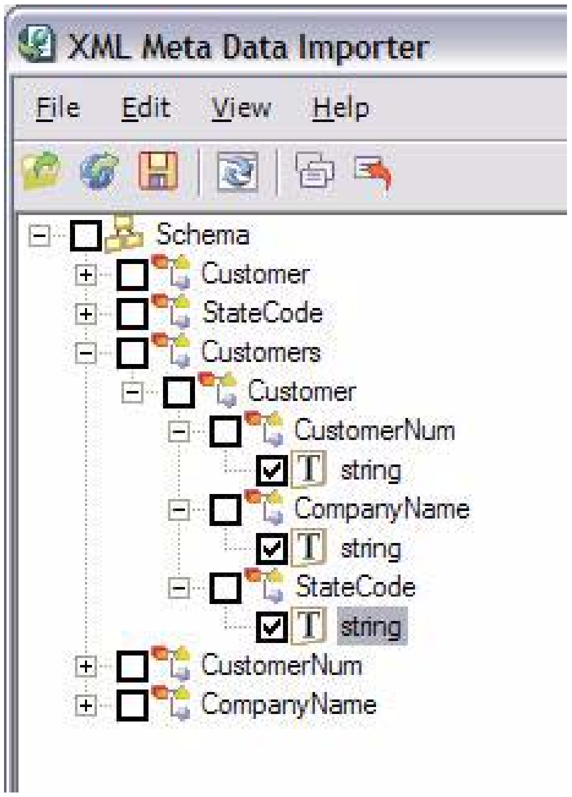
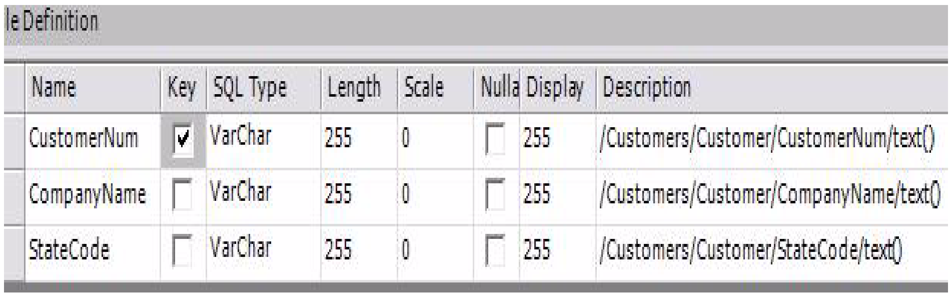


Figure-7, Selecting (Checking) the Data Type entries in the Tree View.


le Definition

	Name	Key	SQL Type	Length	Scale	Nulla	Display	Description
	CustomerNum	✓	VarChar	255	0	☐	255	/Customers/Customer/CustomerNum/text()
	CompanyName	☐	VarChar	255	0	☐	255	/Customers/Customer/CompanyName/text()
	StateCode	☐	VarChar	255	0	☐	255	/Customers/Customer/StateCode/text()

Figure-8, The Table Definition View, result of work in Tree View.

f. While it is stated that the Table Definition View will automatically reflect the changes made via our work in the Tree View, *there is one change we need to make within the Table Definition View.*

   One Element inside the Table Definition View must be selected as a Key Value.

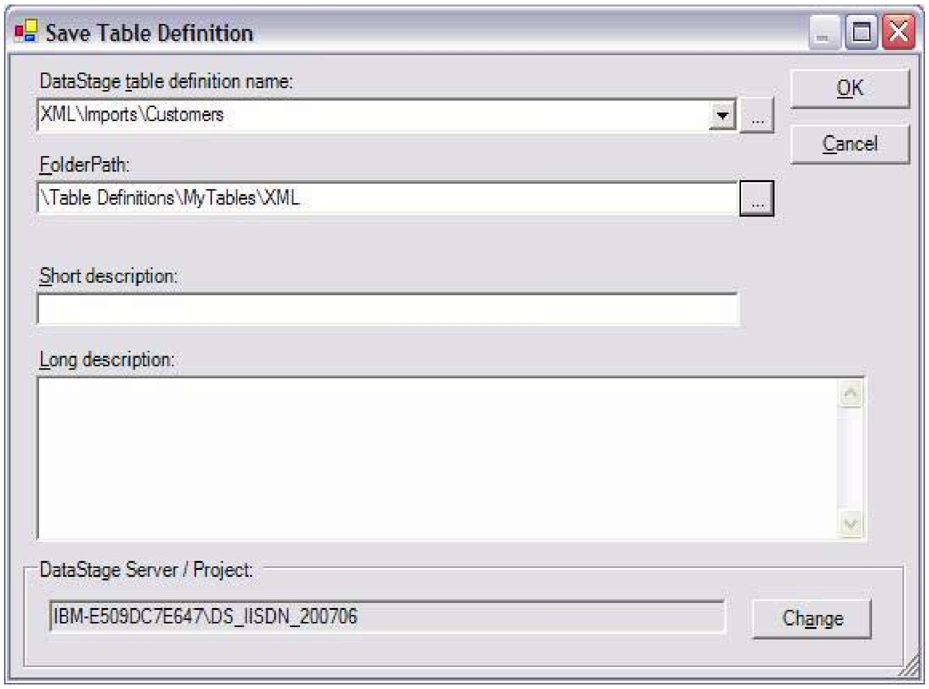
   In our example, we checked the CustomerNum field of our Table Definition. **Check that field now.**

g. We are ready to save our work and exit the DS/QS Meta Data Importer. From the Menu Bar, select; File => Save and complete the steps necessary to save our work. Example as shown in Figure-9.

   After the File => Save, you will also need to File => Exit.

   You may have noticed the automatically created entries in the Description Field inside the Table Definition View. The values in the Description Field are called (XML) XPath Expressions. (XML) XPath Expressions are pretty simple; in effect, XPath Expressions are used to designate Parents, Children, or then subsets of Children in an XML hierarchy. There is not much else you can do with XPath Expressions.

   While the values in the Description Field are required for our use (they are more than just comments, these values are functional), these values are automatically created and managed for us by the DS/QS XML Meta Data Importer.



DataStage table definition name: XML\Imports\Customers

FolderPath: \Table Definitions\MyTables\XML

Short description:

Long description:

DataStage Server / Project: IBM-E509DC7E647\DS_IISDN_200706

Figure-9, Saving the standard DataStage/QualityStage Table Definition.

3. Summary thus far-

   At this point you have exited the DataStage/QualityStage XML Meta Data Importer, and are in DataStage/QualityStage proper. As the result of the previous steps, you have defined a DataStage/QualityStage Table Definition.

   This Table Definition is similar to every previous Table Definition you may have created in DataStage/QualityStage, except for the fact that the Description Field for each column contains an (XML) XPath Expression.

   If you wanted to write these XPath Expressions manually, you did not need to complete Steps-2(a-g) above.

4. Create a DataStage/QualityStage Parallel Job.

   Now we are fully prepared to create our DataStage/QualityStage Job proper.

   a. From the Repository View, and then the Jobs (Folder), Right-Click and select; New => Parallel Job.

   b. From the Menu Bar, select; File => Save As, and give the Job a name. We called our Job "PJ01_TextToXML".

c. From the Palette View, File (Drawer), Drag and Drop a Sequential File object to the Parallel Canvas. We renamed this object to "InputFile_Txt".

d. From the Palette View, Real Time (Drawer), Drag and Drop an XML Output object to the Parallel Canvas. We renamed this object to "OutputFile_XML".

e. On the Parallel Canvas, Right-Click the Sequential File object (InputFile_Txt) and draw a Link to the XML Output object (OutputFile_XML). We renamed this Link to "toOutputFile".

   Figure-10 displays the proper result of these steps. (This is the same figure as Figure-1, displayed earlier.)

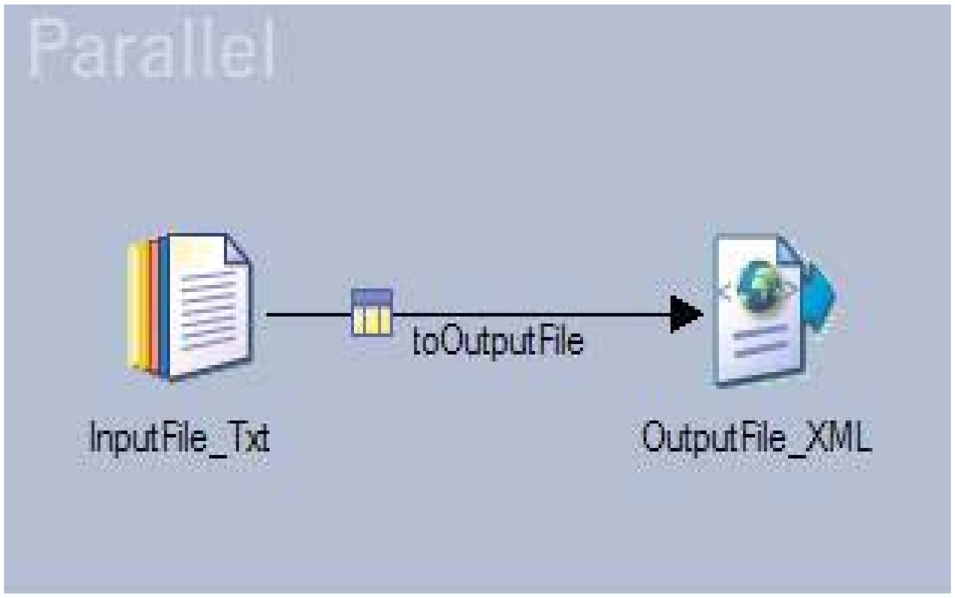


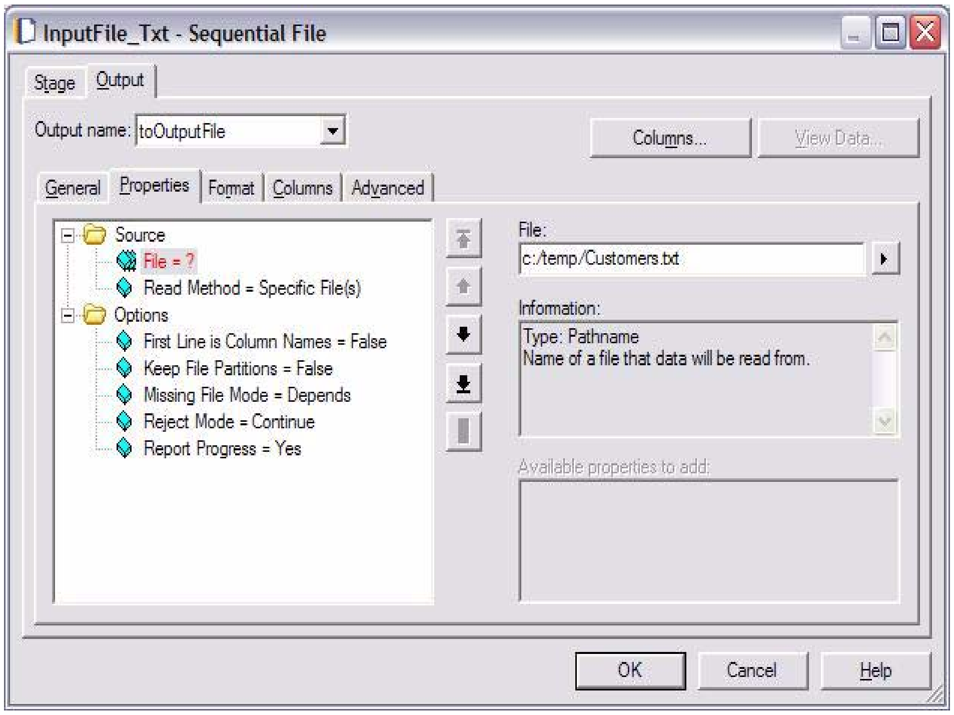
Figure-10, DataStage/QualityStage EE Job, Relational to XML.

5. Set Properties related to (input) Sequential File object.

   There are just a few Properties we need to set in each of the Sequential File object (InputFile_Txt) and the XML Output object (OutputFile_XML).

   a. Double-Click the Sequential File object (InputFile_Txt). This action produces the display in Figure-11. There are three distinct TABs we are going to access.

b. In the Output TAB and then Properties (sub) TAB, set the required File (name) Property. We set ours to "c:/temp/Customers.txt", the location of the ASCII Text input data file we created earlier.



Output name: toOutputFile

Source
File = ?
Read Method = Specific File(s)
Options
First Line is Column Names = False
Keep File Partitions = False
Missing File Mode = Depends
Reject Mode = Continue
Report Progress = Yes

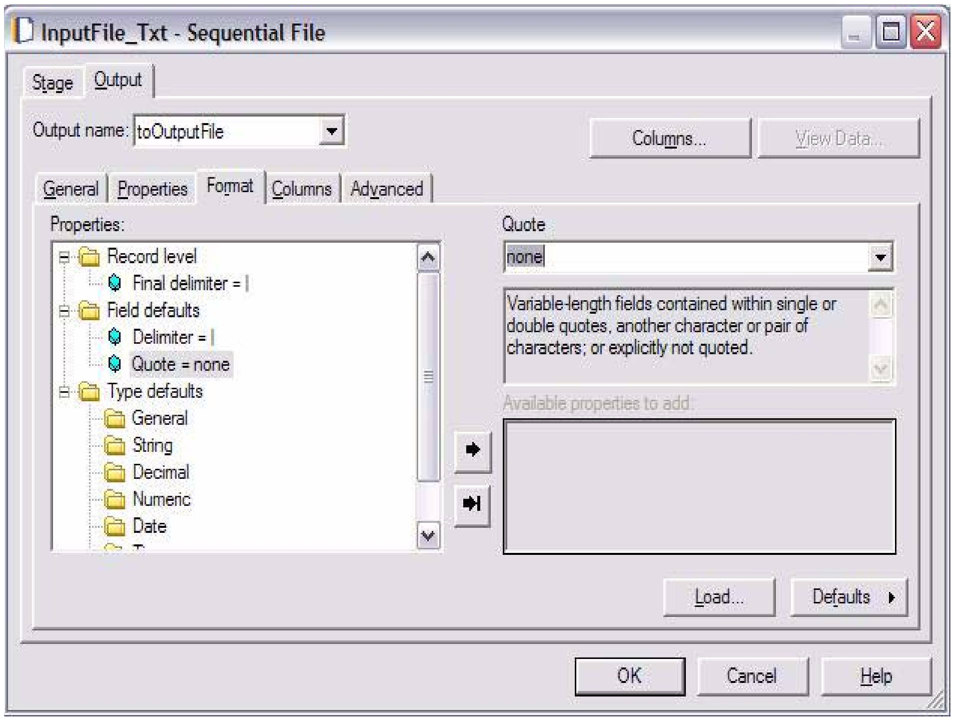
File: c:/temp/Customers.txt

Information:
Type: Pathname
Name of a file that data will be read from.

Figure-11, Sequential File object, Output TAB, Properties (sub) TAB.

c. In the Output TAB, and then Format (sub) TAB, we set three Properties, namely; Record level => Final delimiter, Field defaults => Delimiter, and Field defaults => Quote. Example as shown in Figure-12.

We set these properties to equal the conditions as they exist in our input data file (Customers.txt).



Output name: toOutputFile

Properties:
- Record level
  - Final delimiter = |
- Field defaults
  - Delimiter = |
  - Quote = none
- Type defaults
  - General
  - String
  - Decimal
  - Numeric
  - Date

Quote: none

Variable-length fields contained within single or double quotes, another character or pair of characters; or explicitly not quoted.

Available properties to add:

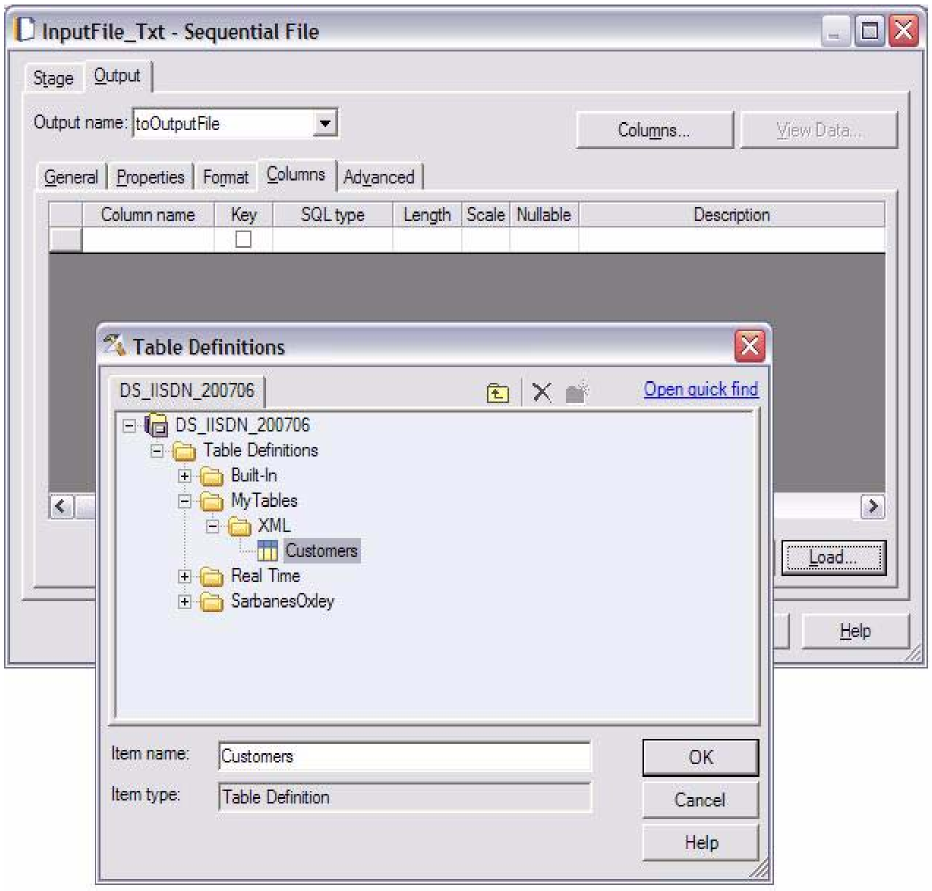
Figure-12, Sequential File object, Output TAB, Format (sub) TAB.

d. In the Output TAB, and then Columns (sub) TAB, we want to import the Table Definition we created earlier in Steps-2(a-g) above.

   In the Output TAB, Columns (sub) TAB, click the "Load ..." Button. This action produces the (Load) Table Definitions dialog box, as displayed in Figure-13.

   Navigate the (Load) Table Definitions dialog box and select the Customers Table Definition created earlier. Click OK.

   This action produces the Select Columns dialog box, displayed in Figure-14.

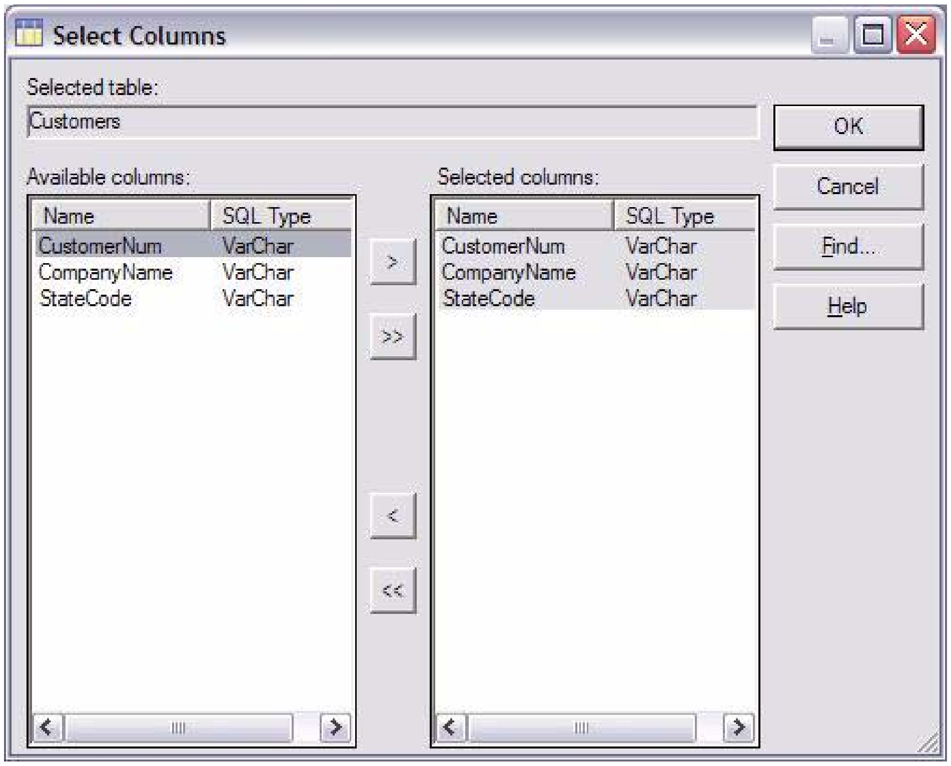


Item name: Customers
Item type: Table Definition

Figure-13, (Load) Table Definitions dialog box.

e. In Figure-14, we are fine selecting all of the columns from our Table Definition. Click OK.

   This action produces the display in Figure-15.



Name	SQL Type
CustomerNum	VarChar
CompanyName	VarChar
StateCode	VarChar

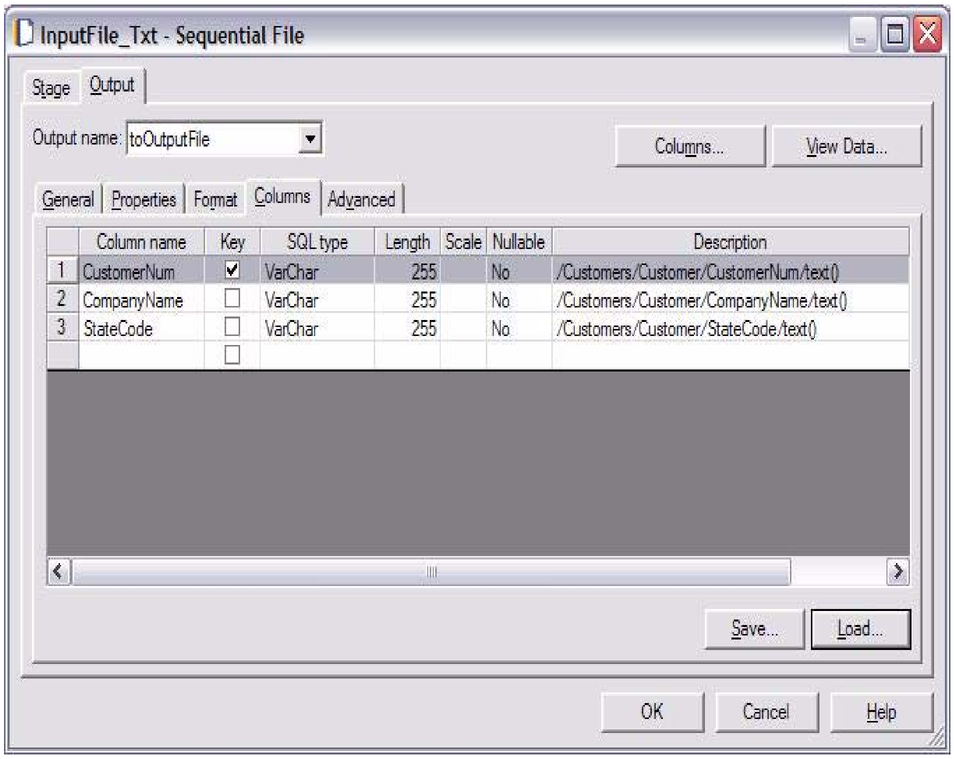
Figure-14, (Load Table Definitions) Select Columns dialog box.

f. In Figure-15, we are nearly done configuring the Properties on the Sequential File object (Customers.txt), our input data set.

   While the CustomerNum Column was defined as a Key in the Table Definition, there is no requirement we would use this column as a Key here.

   Check the Key attribute for the CustomerNum Column. Example as shown in Figure-15. (Or be certain that it was previously checked.)

g. Click OK to close the Sequential File Properties dialog box.



	Column name	Key	SQL type	Length	Scale	Nullable	Description
1	CustomerNum	☑	VarChar	255		No	/Customers/Customer/CustomerNum/text()
2	CompanyName	☐	VarChar	255		No	/Customers/Customer/CompanyName/text()
3	StateCode	☐	VarChar	255		No	/Customers/Customer/StateCode/text()
		☐					

Figure-15, Nearly final result, Sequential File object as configured.

6. Set Properties related to XML Output (in this case, file) object.

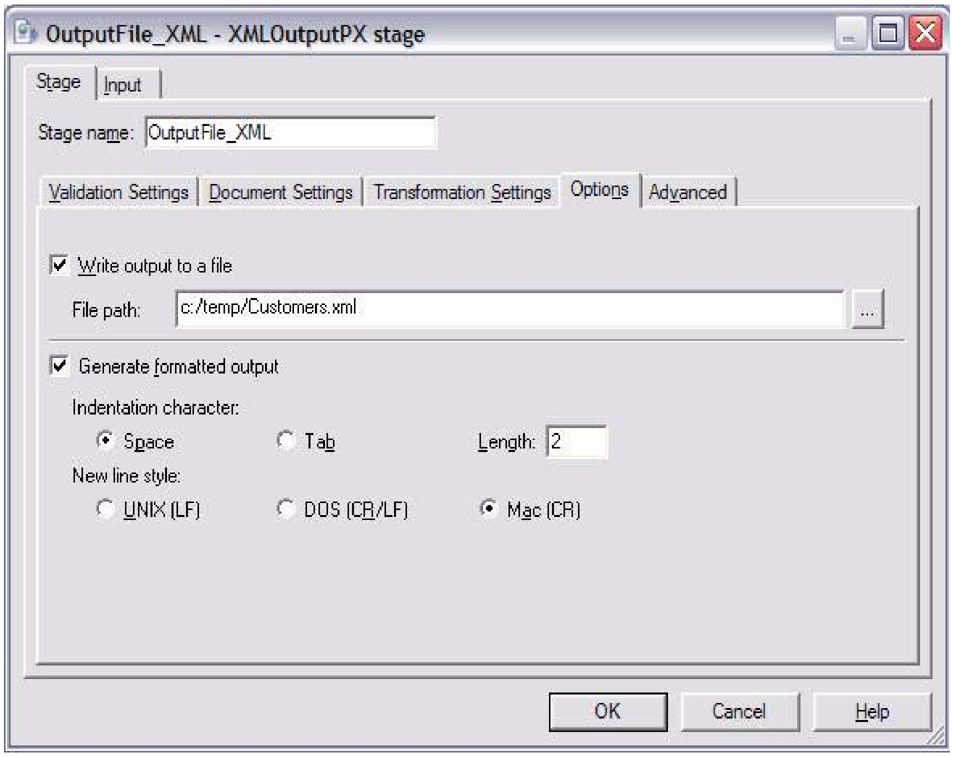
   Now we are ready to set Properties for the XML Output object (OutputFile_XML). There will be little to do here because most of what we need was set by the Table Definition import.

   a. Double-Click the XML Output object (OutputFile_XML). This action produces the display in Figure-16. There is one distinct TAB we are going to access, namely; Stage => Options.

   b. In the Stage => Options TAB, Check the "Write output to a file" visual control, and enter a valid value in the "File Path" text entry field. Example as shown in Figure-16.

      (Many times we would write to an output file here. The primary alternative would be to write to another DataStage/QualityStage operator; expectably the WISD Output object which allow us to be a real time Web Service Provider. But that is a topic for another time.)

c. Click OK.



Stage name: OutputFile_XML

[✓] Write output to a file

File path: c:/temp/Customers.xml

[✓] Generate formatted output

Indentation character: [✓ Space] Tab Length: 2

New line style: UNIX (LF) DOS (CR/LF) [✓ Mac (CR)]

Figure-16, XML Output object as configured.

7. Compile, Run, and Test our Job.

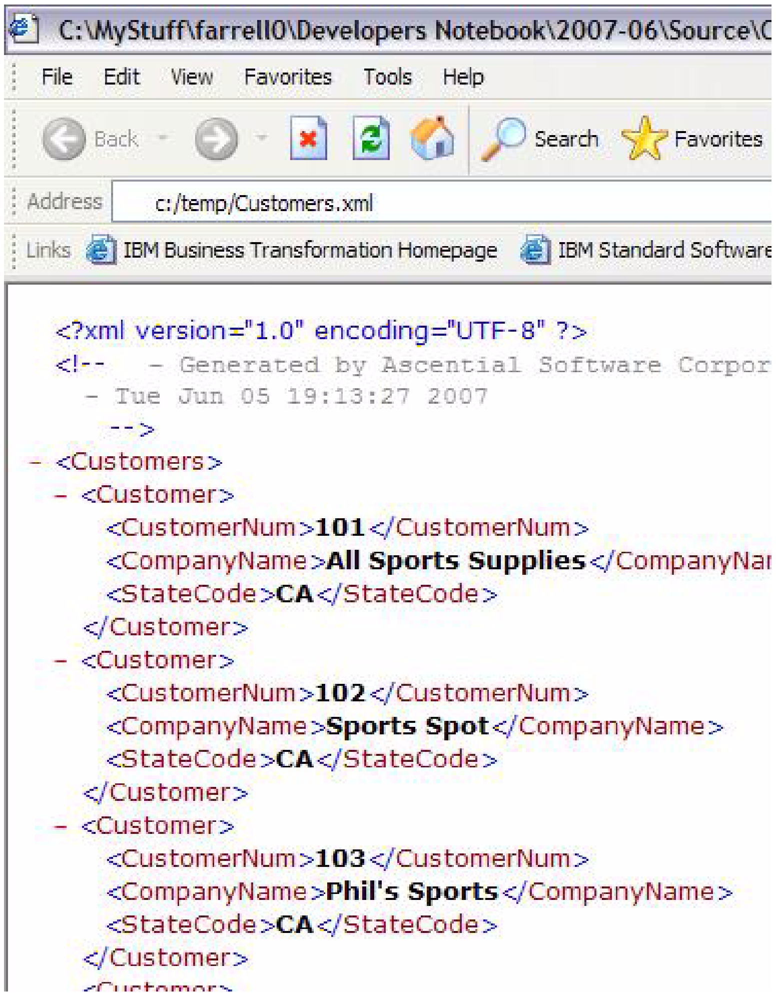
   Now we are ready to compile and run our Job, and view our results.

   a. From the Tool Bar, Compile and then Run this Job (PJ01_TextToXML). The output of this Job will be an XML formatted data file in the location you specified above. (We used "c:/temp/Customers.xml".)

   b. Open the XML Output file (Customers.xml) using a Web browser, Microsoft Internet Explorer or Mozilla Firefox. You should produce a display similar to Figure-17 below.

   c. We will use this XML formatted data file in the next section, where we create a DataStage/QualityStage Job that reads XML formatted data.

[End] Complete the following steps:


C:\MyStuff\farrell0\Developers Notebook\2007-06\Source\C
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
Links
IBM Business Transformation Homepage
IBM Standard Software

c:/temp/Customers.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--   - Generated by Ascential Software Corpor
  - Tue Jun 05 19:13:27 2007
    -->
- <Customers>
  - <Customer>
      <CustomerNum>101</CustomerNum>
      <CompanyName>All Sports Supplies</CompanyNar
      <StateCode>CA</StateCode>
    </Customer>
  - <Customer>
      <CustomerNum>102</CustomerNum>
      <CompanyName>Sports Spot</CompanyName>
      <StateCode>CA</StateCode>
    </Customer>
  - <Customer>
      <CustomerNum>103</CustomerNum>
      <CompanyName>Phil's Sports</CompanyName>
      <StateCode>CA</StateCode>
    </Customer>
```

Figure-17, Our outputted XML data file.

In this section of this document we completed the following:

- We created an XML Schema Definition (XSD) file by hand. (Later we will create a file of this type using graphical tools).
- Using the DataStage/QualityStage XML Meta Data Importer, we referenced the XML Schema Definition file above to create a DataStage/QualityStage Table Definition complete with XML XPath Expressions.

– We created and ran a DataStage/QualityStage XML Output object Job, and created an XML formatted data file.

The DataStage/QualityStage XML Output operator is one of three XML related operators within DataStage/QualityStage. The remaining two include; XML Input operator and XML Transform operator. In the next section, we create a DataStage/ QualityStage job that uses the XML Input operator.

6.3 DataStage/QualityStage EE Job, XML Input

This section of this document was *developed and tested* using WebSphere DataStage/ QualityStage Enterprise Edition version 8.01 on the Microsoft Windows XP/SP2 platform. The steps outlined in this section *may work* on DataStage Server Edition 8.x or 7.x with only minor modification. Other conditions and assumptions are listed below:

– It is assumed you have previously completed Section 6.2, “DataStage/QualityStage EE Job, XML Output” on page 4. We use the XML data file that was output in that Section to begin our work here.

– Also, we use the DataStage/QualityStage Table Definition that was created in that section.

[Begin] Complete the following steps:

1. Launch the DataStage/QualityStage component of the IBM Information Server, and connect to the Project where the tasks performed in Section 6.2, “DataStage/QualityStage EE Job, XML Output” on page 4 reside.
2. Create a DataStage/QualityStage Parallel Job.
   a. From the Repository View, and then Jobs (Folder), Right-Click and select; New => Parallel Job.
   b. From the Menu Bar, select; File => Save As, and give the Job a name. We called our Job “PJ02_XMLToText”.
   c. From the Palette View, File (Drawer), Drag and Drop a Sequential File object to the Parallel Canvas. We renamed this object to “InputFile_XML”.
   d. From the Palette View, Real Time (Drawer), Drag and Drop an XML Input object to the Parallel Canvas. We renamed this object to “XMLInput”.
   e. From the Palette View, File (Drawer), Drag and Drop a Sequential File object to the Parallel Canvas. We renamed this object to “OutputFile_Txt”.
   f. On the Parallel Canvas, Right-Click the first Sequential File object (InputFile_XML), and draw a Link to the XML Input object (XMLInput). We renamed this Link to “toXMLInput”.

g. On the Parallel Canvas, Right-Click the XML Input object (XMLInput), and draw a Link to the second Sequential File object (OutputFile_Txt). We renamed this Link to "toOutputFile".

h. Compare your work to that as displayed in Figure-18.

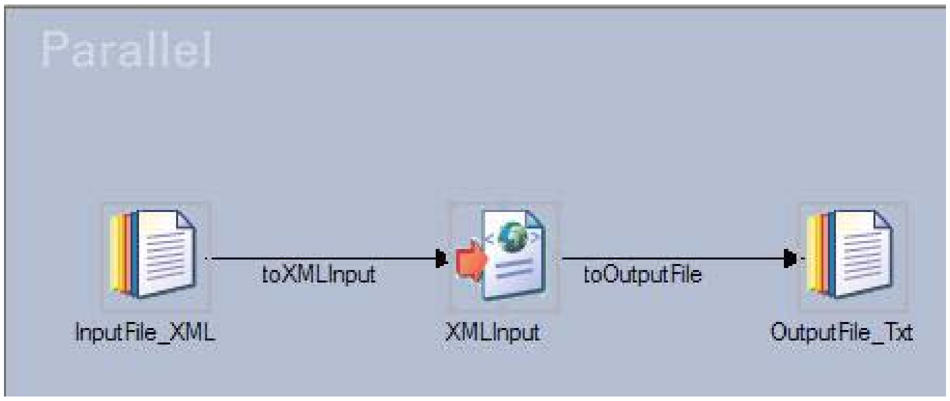


Figure-18, DataStage/QualityStage EE Job, XML to Relational.

3. Set Properties for the first (input) Sequential File object (InputFile_XML).

   a. On the Parallel Canvas, Double-Click the Sequential File object (InputFile_XML) and access the Output TAB, and then Properties (sub) TAB.

      Here set the Source => File (name). We set ours to "c:/temp/Customers.xml", the XML formatted data file we created previously in Section 6.2, "DataStage/QualityStage EE Job, XML Output" on page 4. See Figure-19 below.

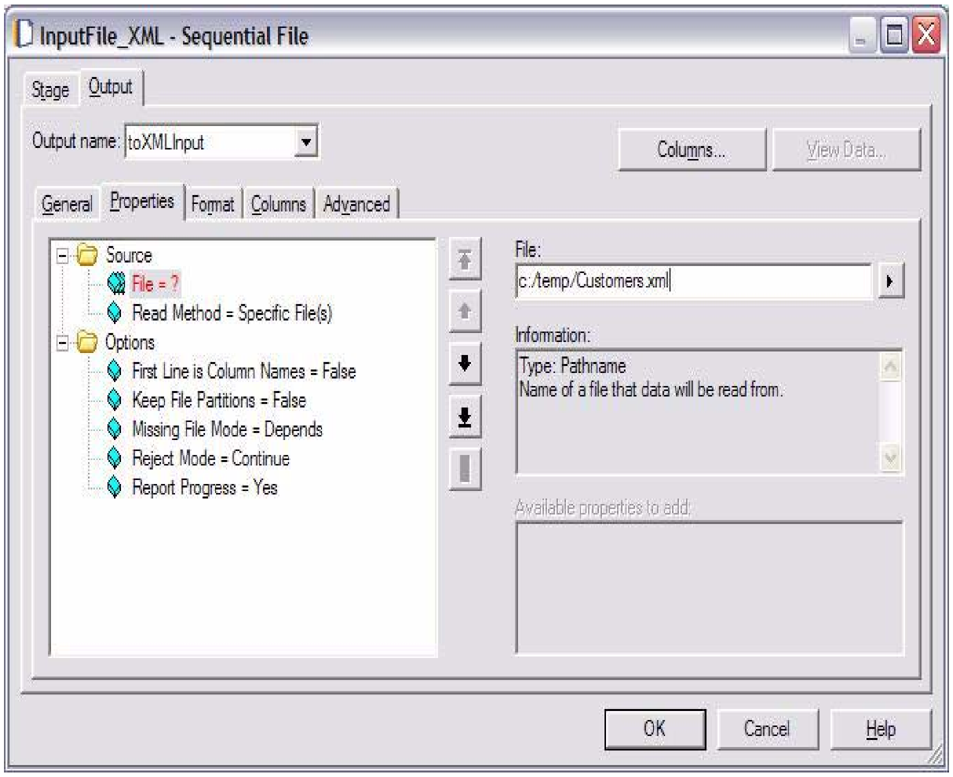
   b. Move to the Output TAB, and then Format (sub) TAB. Here we will set three Properties.

      Set Record level => Final delimiter to "end". In this context, "end" means "end of file".

      Set Field defaults => Delimiter to "none".

      Set Field defaults => Quote to "none".

      In effect, we are going to read this Sequential File object as one long uninterrupted string. See Figure-20 below.


InputFile_XML - Sequential File

Stage | Output

Output name: toXMLInput

Columns... | View Data...

General | Properties | Format | Columns | Advanced

- Source
  - File = ?
  - Read Method = Specific File(s)
- Options
  - First Line is Column Names = False
  - Keep File Partitions = False
  - Missing File Mode = Depends
  - Reject Mode = Continue
  - Report Progress = Yes

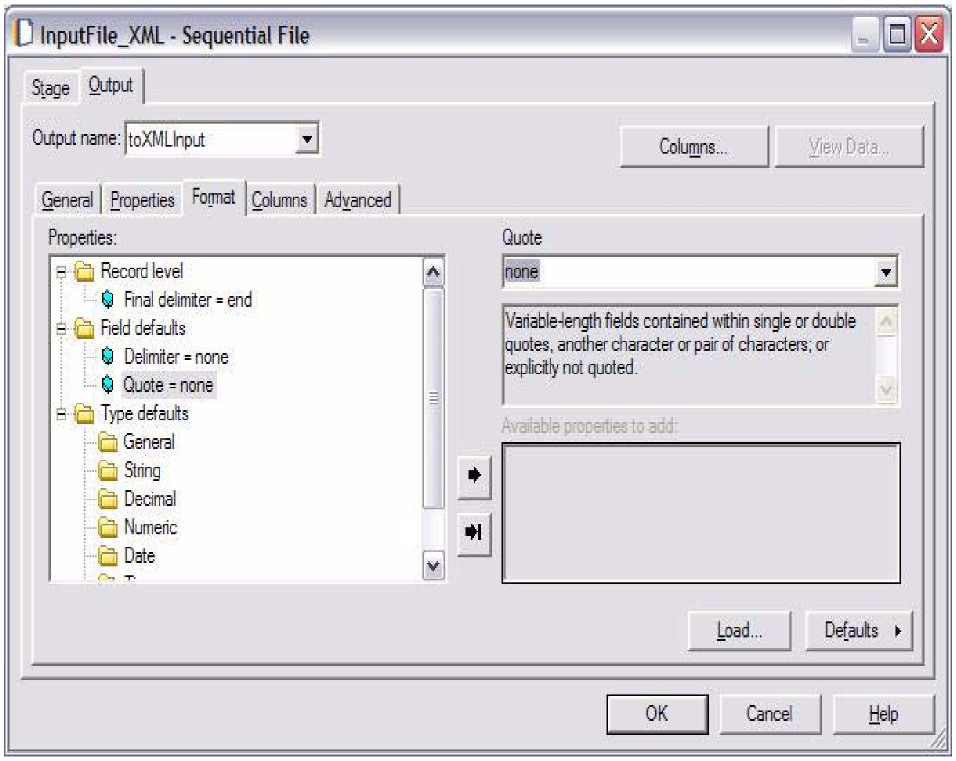
File:
c:/temp/Customers.xml

Information:
Type: Pathname
Name of a file that data will be read from.

Available properties to add:

OK | Cancel | Help

Figure-19, Sequential File object, Output TAB => Properties (sub) TAB.



Output name: toXMLInput

Properties:

- Record level
  - Final delimiter = end
- Field defaults
  - Delimiter = none
  - Quote = none
- Type defaults
  - General
  - String
  - Decimal
  - Numeric
  - Date

Quote

none

Variable-length fields contained within single or double quotes, another character or pair of characters; or explicitly not quoted.

Available properties to add:

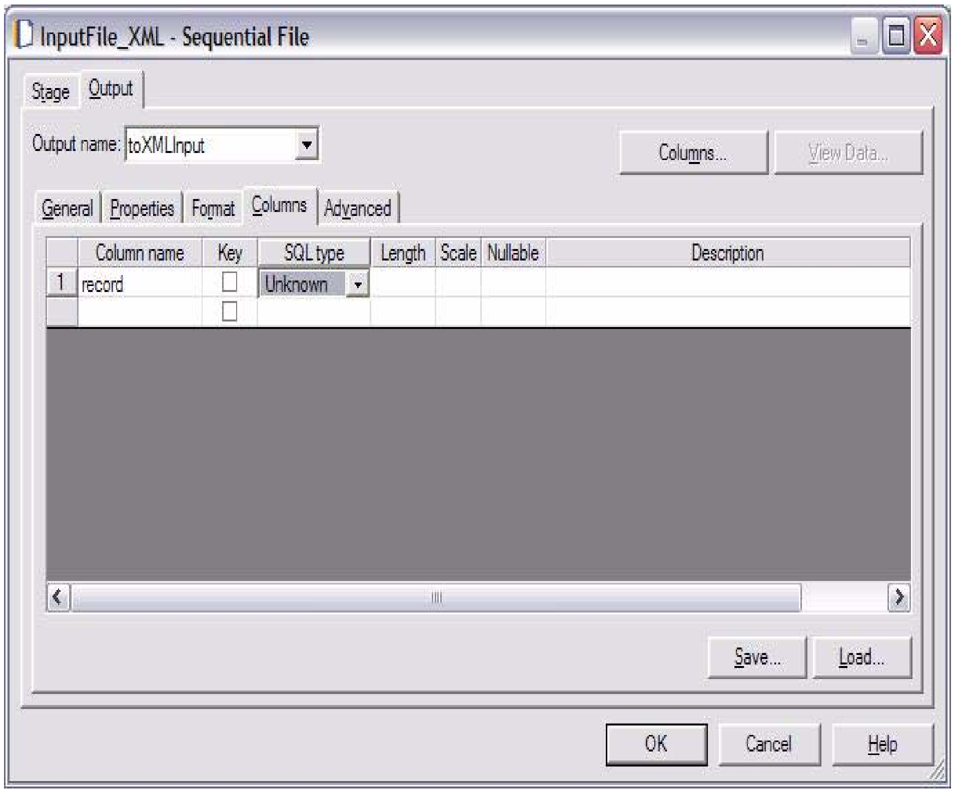
Figure-20, Sequential File object, Output TAB => Format (sub) TAB.

c. Move to the Output TAB, and then Columns (sub) TAB. Here we will create a new column of type "unknown". This is the source column from our Sequential File input; basically it is a big string. Other folks use VarChar(32K) or something to that effect. Unknown works also.

   We named our column "record". (The word "record" is not a keyword, we could have called this column anything.)

   Manage the visual controls in the Output TAB => Columns (sub) TAB to equal what is displayed in Figure-21.

   Press OK when you are done.



Output name: toXMLInput

	Column name	Key	SQL type	Length	Scale	Nullable	Description
1	record	☐	Unknown				
		☐					

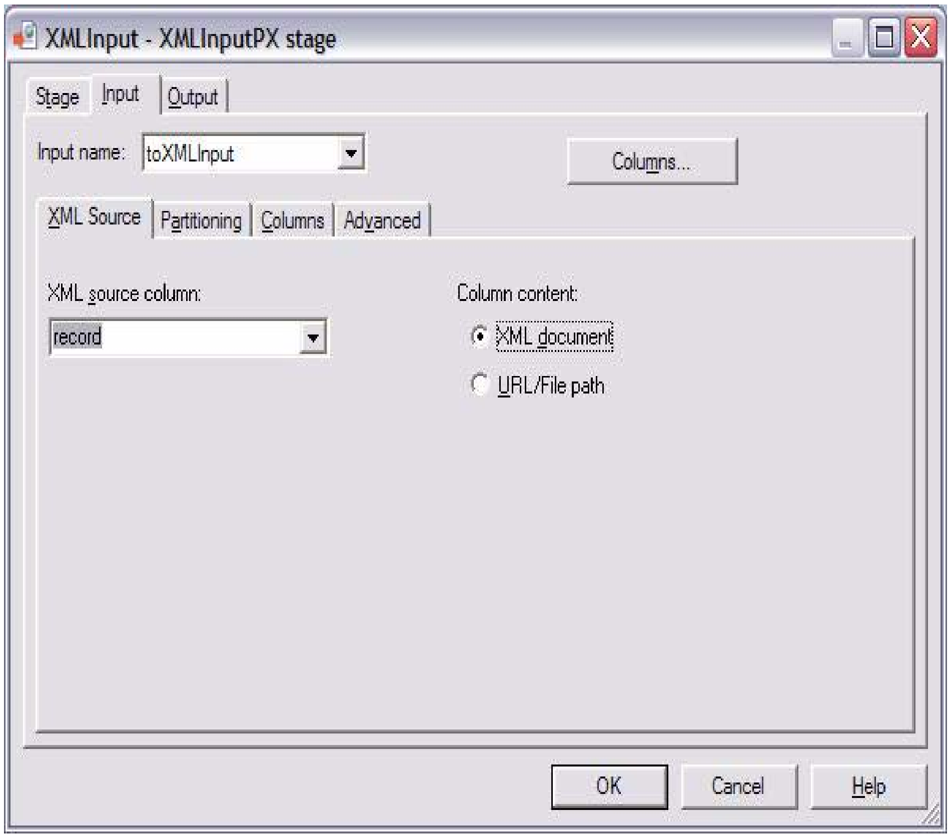
Figure-21, Sequential File object, Output TAB => Columns (sub) TAB.

4. Set Properties for the XML Input object (XMLInput).
   a. On the Parallel Canvas, Double-Click the XML Input object (XMLInput) and access the Input TAB, and then XML Source (sub) TAB.

      Here we need to specify an XML source input column. This is done by accessing the Drop Down List Box visual control named "XML source column". There should only be one choice here; the column name "record" comes from the previous step we had performed.

      In the Radio Button Group named "Column content", check the "XML document" button.

      Example as shown in Figure-22.



Input name: toXMLInput

XML source column: record

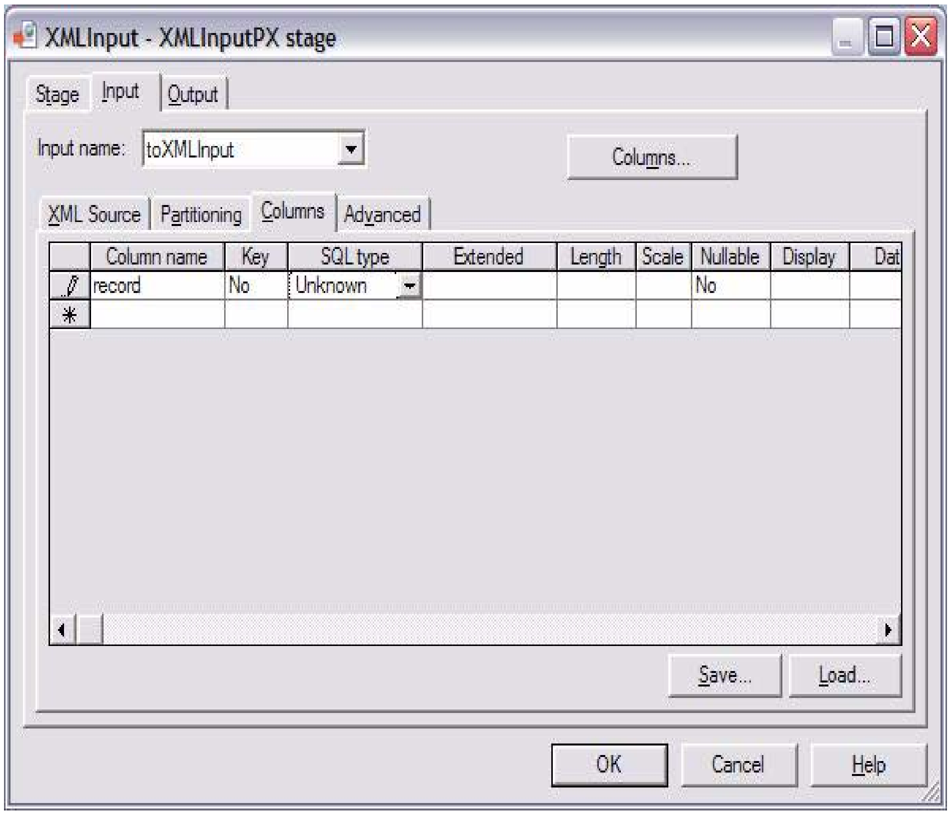
Column content: [✓ XML document] URL/File path

Figure-22, XML Input object, Input TAB => XML Source (sub) TAB.

b. Move to the Input TAB => Columns (sub) TAB. Example as shown in Figure-23.

   DataStage/QualityStage may have defaulted to an input column data type of Char. If so, change the input column data type to Unknown.

   Everything else on this TAB is fine.


XMLInput - XMLInputPX stage

Stage | Input | Output

Input name: toXMLInput

	Column name	Key	SQL type	Extended	Length	Scale	Nullable	Display	Dat
	record	No	Unknown				No		
*									

Figure-23, XML Input object, Input TAB => Columns (sub) TAB.

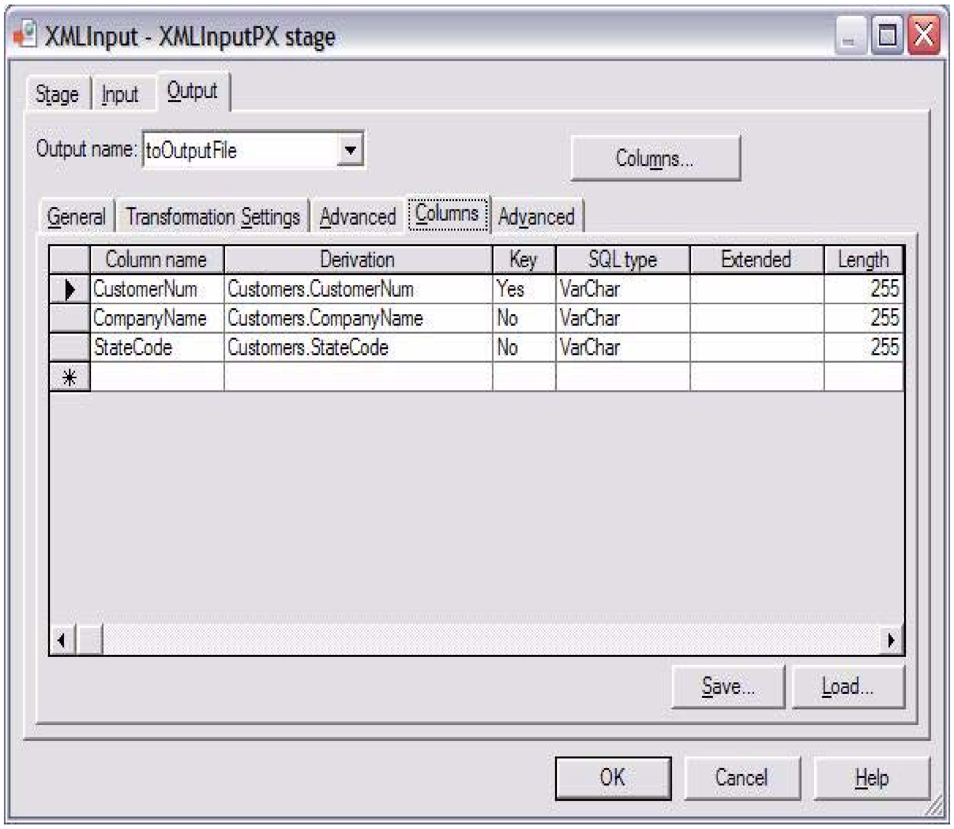
c. Move to the Output TAB => Columns (sub) TAB. Example as shown in Figure-23.

Initially this grid of columns are their associated attributes will be empty. Click the “Load...” Button and follow the prompts to load all columns from the DataStage/QualityStage Table Definition (Customers) that we had created in Section 6.2, “DataStage/QualityStage EE Job, XML Output” on page 4.

Use the Drop Down List Box visual control in the “Key” column, and for column “CustomerNum”. Change this value from No to Yes. See Figure-24 below.

In effect this setting instructs XML Input what the repeating (key) column is for XML Input. You could also view this as instructing XML Input what the most granular (lowest level) piece of data is in this list. It does not matter if the data in this column is unique, or is actually some sort of key value. Only one column should be selected here, never two.

Click OK when you are done.



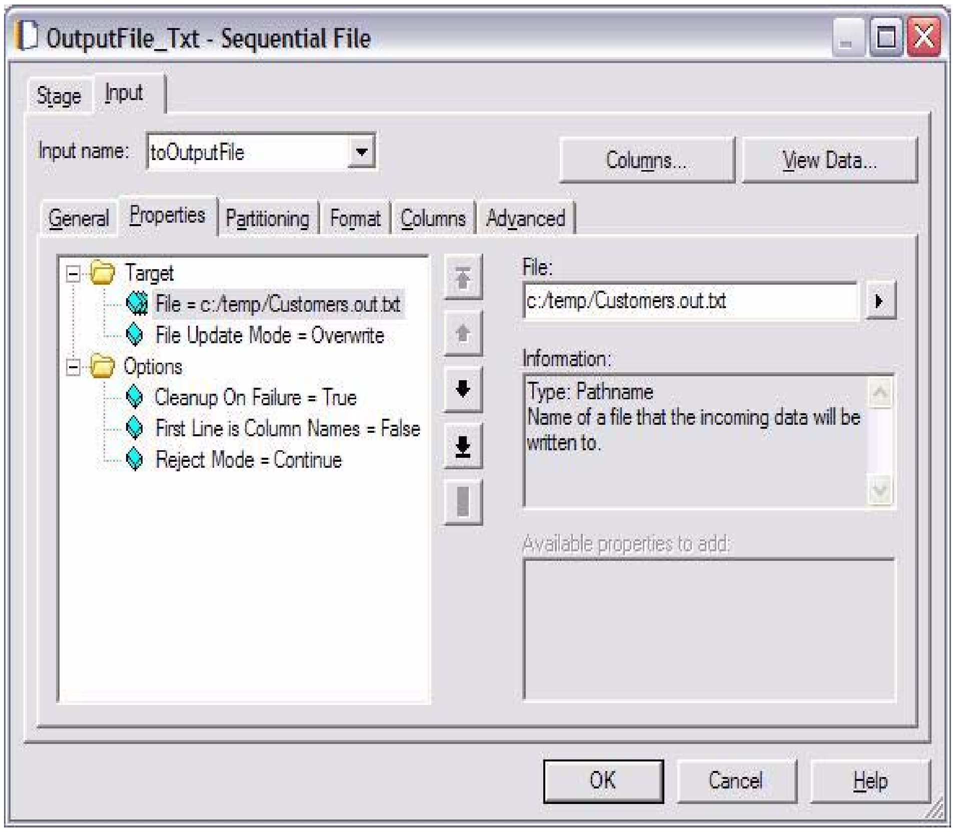
Column name	Derivation	Key	SQL type	Extended	Length
CustomerNum	Customers.CustomerNum	Yes	VarChar		255
CompanyName	Customers.CompanyName	No	VarChar		255
StateCode	Customers.StateCode	No	VarChar		255

Figure-24, XML Input object, Output TAB => Columns (sub) TAB.

5. Set Properties for the second (output) Sequential File object (OuputFile_Txt).
   a. On the Parallel Canvas, Double-Click the Sequential File object (OutputFile_Txt) and access the Input TAB, and then Properties (sub) TAB.

      The only Property we have to set in the Sequential File object (OutputFile_Txt) is the Input => Properties => Target => File (name). We set ours to "c:/temp/Customers.out.txt".
   b. You can set other Properties as you wish; field and record delimiters, etcetera.

c. Click OK to close the Sequential file Properties dialog box.


OutputFile_Txt - Sequential File

Input name: toOutputFile

- Target
  - File = c:/temp/Customers.out.txt
  - File Update Mode = Overwrite
- Options
  - Cleanup On Failure = True
  - First Line is Column Names = False
  - Reject Mode = Continue

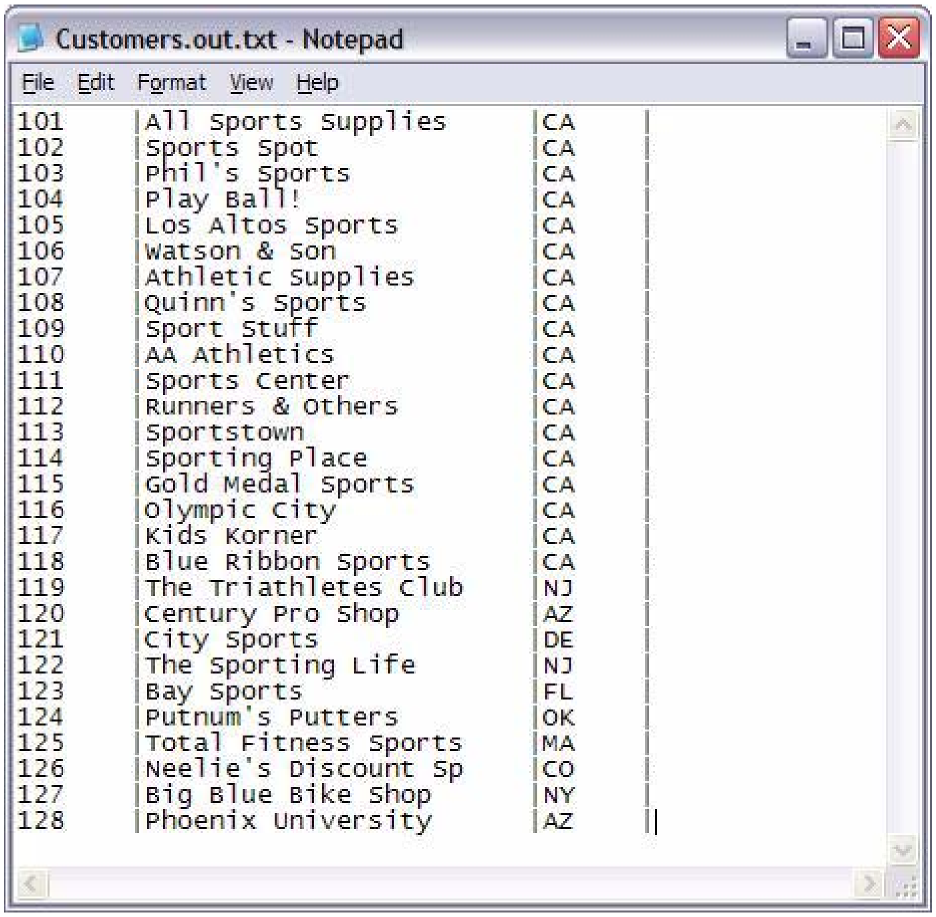
File: c:/temp/Customers.out.txt

Information:
Type: Pathname
Name of a file that the incoming data will be written to.

Figure-25, Sequential File, Input TAB => Properties (sub) TAB.

6. Save, Compile, and Run this Job (PJ02_XMLToTxt). A successful result is displayed in Figure-26.

[End] Complete the following steps:



```
101     |All Sports Supplies      |CA      |
102     |Sports Spot              |CA      |
103     |Phil's Sports            |CA      |
104     |Play Ball!               |CA      |
105     |Los Altos Sports         |CA      |
106     |Watson & Son             |CA      |
107     |Athletic Supplies        |CA      |
108     |Quinn's Sports           |CA      |
109     |Sport Stuff              |CA      |
110     |AA Athletics             |CA      |
111     |Sports Center            |CA      |
112     |Runners & Others         |CA      |
113     |Sportstown               |CA      |
114     |Sporting Place           |CA      |
115     |Gold Medal Sports        |CA      |
116     |Olympic City             |CA      |
117     |Kids Korner              |CA      |
118     |Blue Ribbon Sports       |CA      |
119     |The Triathletes Club     |NJ      |
120     |Century Pro Shop         |AZ      |
121     |City Sports              |DE      |
122     |The Sporting Life        |NJ      |
123     |Bay Sports               |FL      |
124     |Putnum's Putters         |OK      |
125     |Total Fitness Sports     |MA      |
126     |Neelie's Discount Sp     |CO      |
127     |Big Blue Bike Shop       |NY      |
128     |Phoenix University       |AZ      |
```

Figure-26, Output from (PJ02_XMLToTxt) Job.

In this section of this document we completed the following:

> We created and ran a DataStage/Quality Stage XML Input object Job, and read an XML formatted data file.

Thus far we have used the DataStage/QualityStage XML operators (XML Output and XML Input). We used XML Output first, because it reads data in the standard format with which we were familiar; flat, relational data in rows and columns. Since XML Output outputs XML formatted data, this output could then serve our work with XML Input, which expects to read XML.

If you are confused that XML Input and XML Output both have input and output capability, consider that the identifying keywords Input and Output refer to *formats*. XML Input accepts XML formatted data and outputs whatever. XML Output accepts whatever and outputs XML formatted data.

In our examples we were landing data to and from files. We used files to simplify our examples. Both XML Output and XML Input can read and write from Web Services; in fact, that is a standard use case for these DataStage/QualityStage operators.

6.4 Creating the XML Schema Definition used above

In both Sections 6.2 and 6.3, we had an external dependency which we will now resolve. Each of those Sections relied upon an XML Schema Definition (XSD) file. Because that file was small, 27 lines or less, we created this file by hand. We called this file (Customers.xsd). Creating XSD files by hand is similar to creating SQL Data Definition Language (SQL/DDL) command files by hand; it can be done, but at some point greater productivity can be gained by using a visual tool. Now, in this section, we will use a visual (graphical) tool to paint the contents of our XSD file.

In your shop, it might be normal for the DataStage/QualityStage person to be given any necessary XML Schema Definition (XSD) files, and that is perfectly fine. In this section, you can learn how to create or update these files yourself.

This section of this document was *developed and tested* using Rational Data Architect version 7.0.0.1 on the Microsoft Windows XP/SP2 platform. Rational Data Architect is a component of IBM Information Server version 8.01. Rational Data Architect is also released separately; hence, the reason why the version numbers currently differ.

Other conditions and assumptions are listed below:

- Rational Data Architect is an Eclipse based product.

  Eclipse is an open source developer’s workbench. If you have never used an Eclipse based tool before, there is a 30 minute to one hour learning curve. Eclipse has its own taxonomy of concepts and terms that must be learned; Views, Perspectives, Projects, Workspaces, Workbench, Roles, etcetera.

  If you are really going to learn Rational Data Architect, plan for an amount of time for general Eclipse and related skills development.

- Rational Data Architect does more than just create XML Schema Definition files; a lot more. Later in this document we will use Rational Data

Architect and Rational Application Developer to create an XML Style Language - Transform (XSLT) file.

However this is all somewhat misleading, Rational Data Architect is more famous for the relational and non-relational database modeling and support that it offers.

[Begin] Complete the following steps:

1. Launch the Rational Data Architect tool.
    a. If you are prompted for a Workspace, choose a new empty directory somewhere on your hard disk.

       If you are not prompted for a Workspace, go to the Menu Bar and select, File => Switch Workspace. Follow the dialog boxes to choose a new empty directory.

       In this context, a Workspace is a parent directory that will contain all of our generated source code and related artifacts. This (directory) will contain a hidden directory named (.metadata), which contains your preferences and other related settings. Our Workspace will contain one additional directory for every new Project we create.

       For our Workspace, we chose “c:/temp/Workspace”.
    b. Rational Data Architect will display a Welcome (View). Close that View by clicking on the red “X” in the title TAB Header. Example as shown in Figure-27.

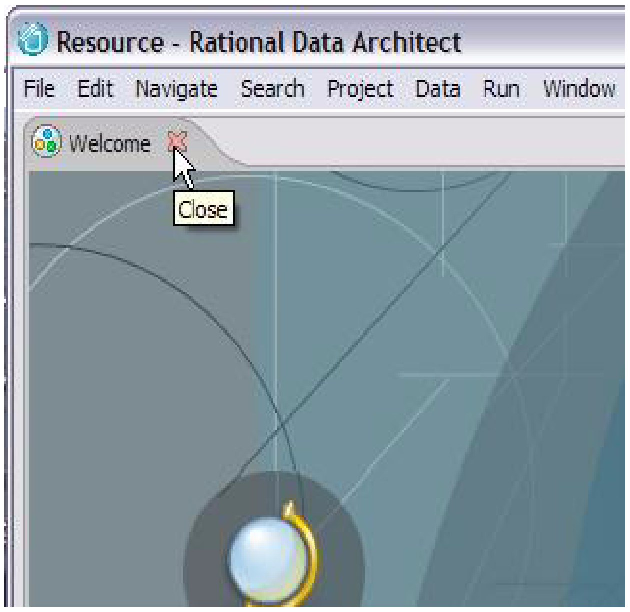

Resource - Rational Data Architect
File Edit Navigate Search Project Data Run Window
Welcome
Close

Figure-27, Rational Data Architect Welcome Screen.

2. Create a Project to contain what we are about to produce.

   All of the work we perform in this sub-section is done one time only; product configuration type activities.

   In Eclipse (which Rational Data Architect is based on), you can not do much of anything without having a Project. In this context, a Project is a collection of artifacts (source code) and metadata related to how the system can compile, deploy, etcetera, these artifacts.

   a. From the Menu Bar, select; File => New => Project. This action will produce the New Project dialog box.

   b. In the New Project dialog box, manage the display to select; Java => Java Project, and click Next. This action will produce the New Java Project dialog box.

      Don’t panic that our Project has Java in its identifier. XML Schema Definition (XSD) files are normally located inside Java Projects. XSD can be used to generate a good measure of Java program code; yet another benefit to XML, XSD, and related.

      There are numerous Project Types in Rational Data Architect; in part, the Project Type tells the Workbench what type of work you want to perform.

c. In the New Java Project dialog box, locate the text entry field named, "Project Name". Enter the value, "MyXMLProject". Click Finish.

d. You are likely to receive an Open Associated Perspective dialog box. This is normal. There are so many Views (specific regions of the display) inside Eclipse and Rational Data Architect, that these Views are grouped into logical collections called Perspectives. You can move Views around, define or save your own Perspectives, etcetera.

   For now, just Click Yes.

   This action returns you to the base display of Rational Application Developer.

e. From the Menu Bar, select; Window => Open View => Navigator. This action will open the Navigator View, most likely in the upper-left portion of the display.

   The Navigator View operates much like Microsoft Windows Explorer; a tree control displays a hierarchy of assorted folders and files. (Although initially our Project will be largely empty.)

f. In the Navigator View, Right-click our Project (MyXMLProject) to produce a context sensitive menu. From this menu, select; New => Package. Example as shown in Figure-28. This action will produce the New Java Package dialog box.

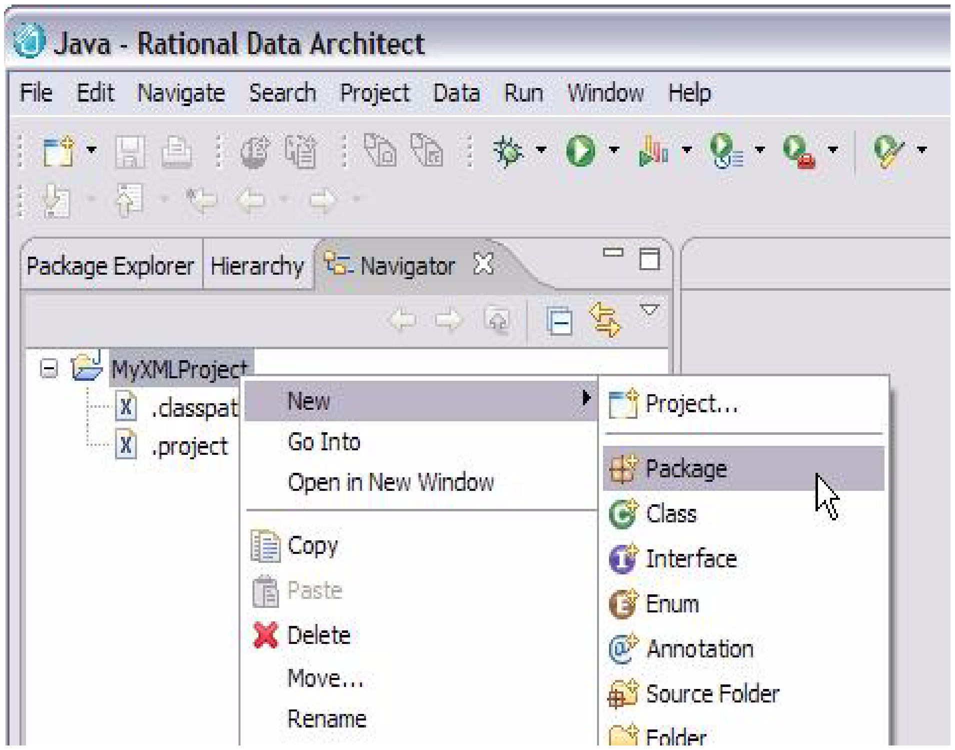


Figure-28, Creating a Package (sub-folder) within our Project.

g. In the New Java Package dialog box, enter the value "MyXSDFiles" in the text entry field named "name". Click Finish.

   We will use this Package to contain our XML Schema Definition (XSD) files. We could have named this Package anything. In this context, Packages are sub-folders under the parent folder which is our Project.

   The full hierarchy of folders is; Workspace (is a folder, rooted in this case from the "C:" drive) => Project Name (is a sub-folder with the same name as our Project, located under the Workspace) => Any named Packages (are sub-folders under the Project).

   We don't need this or any Package. These Packages were created for organization purposes alone.

h. Repeat the step above and create a second Package named "MyXSLTFiles". We will use this Package later to contain our XML Style Language - Transformation (XSLT) files.

i. Summary thus far-

We launched Rational Application Developer and specified a Workspace, (a parent directory to contain our produced artifacts). We created a Java Project. We opened the Navigator View, which functions like Microsoft Windows Explorer. And we created two Packages, (sub-folders under our Project).

At this point, your screen should look like Figure-29.

Don't worry about the entries entitled; ".classpath" and ".project". Those are Project Property files that we don't need to adjust.

[End] Complete the following steps:

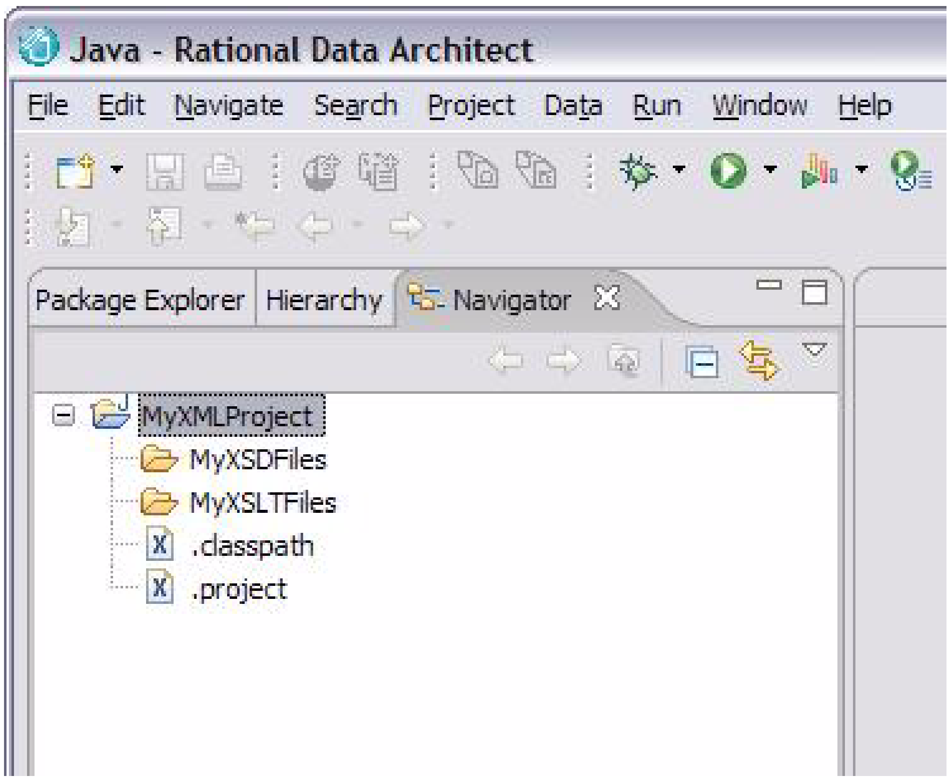


Figure-29, Our Project with two Packages (sub-folders).

All of the tasks we performed in Steps-2(a-i) we preparatory in nature; meaning, one time only type tasks. Now we are ready to begin the real work of creating an XML Schema Definition (XSD) file.

[Begin] Complete the following steps:

3. Create the XML Schema Definition (XSD) file using the XSD Wizard and Palette.
    a. In the Navigator View, Right-Click the "MyXSDFiles", under the Project named "MyXMLProject". This action produces a context sensitive menu.
    b. Select, New => Other. This action produces the New (Object Wizard).
    c. Navigate this visual control to expand the XML group, and then highlight XML Schema. Example as shown in Figure-30.
    d. Click Next.

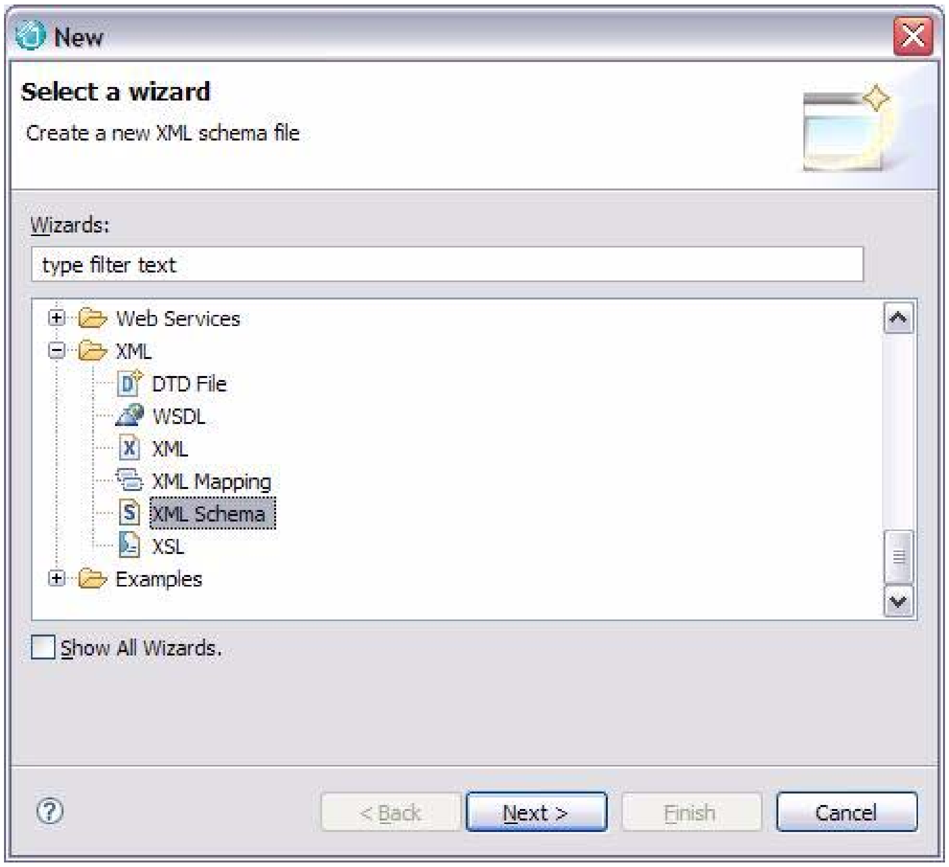


Figure-30, New (Object Wizard) dialog box.

e. In the Create XML Schema dialog box, enter the value "Customers.xsd" in the text entry field entitled "File name". Click Finish. This action produces the display as shown in Figure-31.

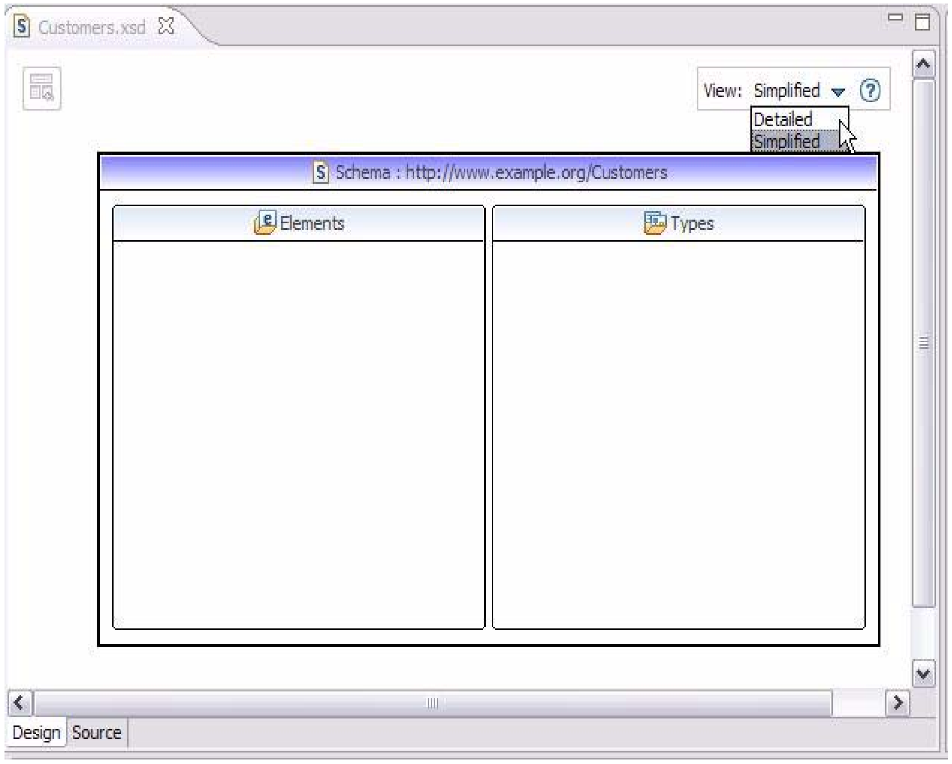


Figure-31, Subset of full screen. WSD Editor.

f. Next we wish to move the WSD Editor from Simplified (Mode) to Detailed (Mode). A Drop Down List Box style visual control entitled “View” allows us to do this. This control is displayed in the upper-right portion of Figure-31.

   Change the XSD Editor to Detailed Mode by using the “View” Drop Down List Box. You may receive an extra prompt when completing this task; Just Click OK.

g. We are creating this XSD file as a learning exercise, yet there are production ready system questions that will be asked of us. As a result, the XSD Editor sets some default values for us that we wish to delete.

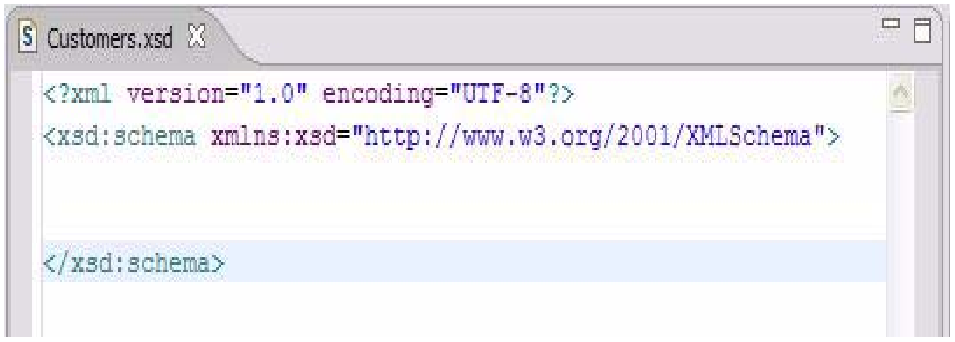
   In the bottom-left of Figure-31, you see two TABs; one labeled Design and one labeled Source. We are going to access the Source View as a means to complete this next step quickly.

   Click the Source TAB. This action will produce the display shown in Figure-32.

Edit the source code to equal what is displayed in Figure-32.

(In effect we are doing this to avoid having to set or give consideration to Namepsaces, and advanced topic that we wish to table and ignore for now.)

Return to the Design View by a Click on the Design TAB.



```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">


</xsd:schema>
```

Figure-32, Namespaces entries already corrected in WSD Editor.

h. If you wish, refresh yourself with the object we are about to create, our Customers.xsd file from Figure-3, far above.

i. First we are going to add our three Elements, CustomerNum, CompanyName and StateCode.

   Right-Click in the Elements Pane (area) of the XSD Editor, and select Add Element. Example as shown in Figure-33.

   A new Element will be created with the name "NewElement". We wish to rename this Element to "CustomerNum". This is accomplished by a Right-Click on NewElement, then select; Refactor => Rename. Example as shown in Figure-34.

   You may receive a prompt to save any changes thus far. Click OK.

   Eventually you will receive a Rename Wizard dialog box. Replace the original value NewElement with CustomerNum. Click OK.

   Repeat this step to Add and Rename CompanyName.

   Repeat this step to Add and Rename StateCode.

   Repeat this step to Add and Rename Customer.

   Report this step to Add and Rename Customers.

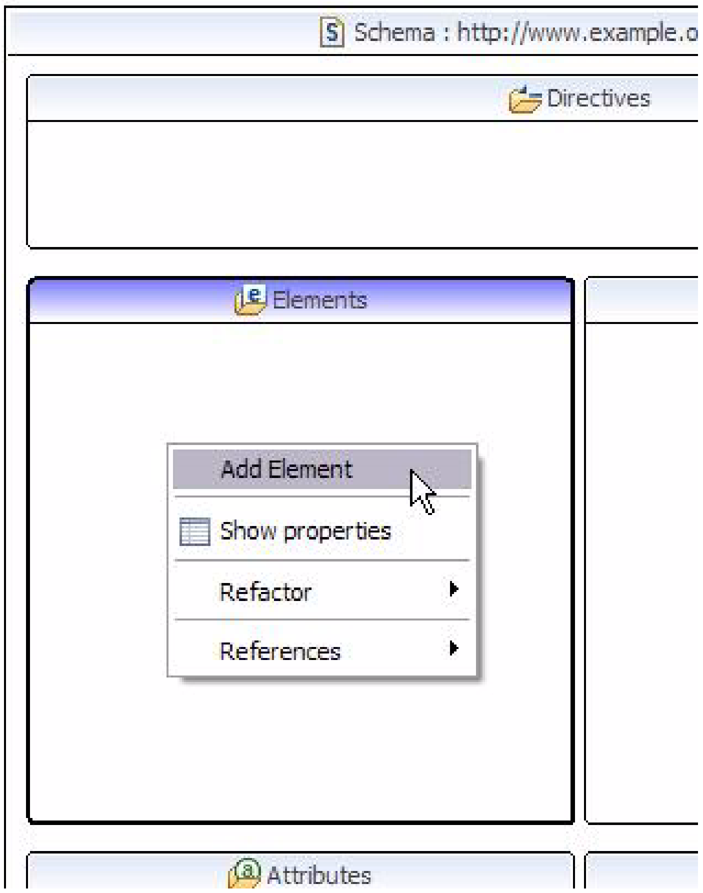


Figure-33, WSD Editor, Elements Pane, Adding an Element.

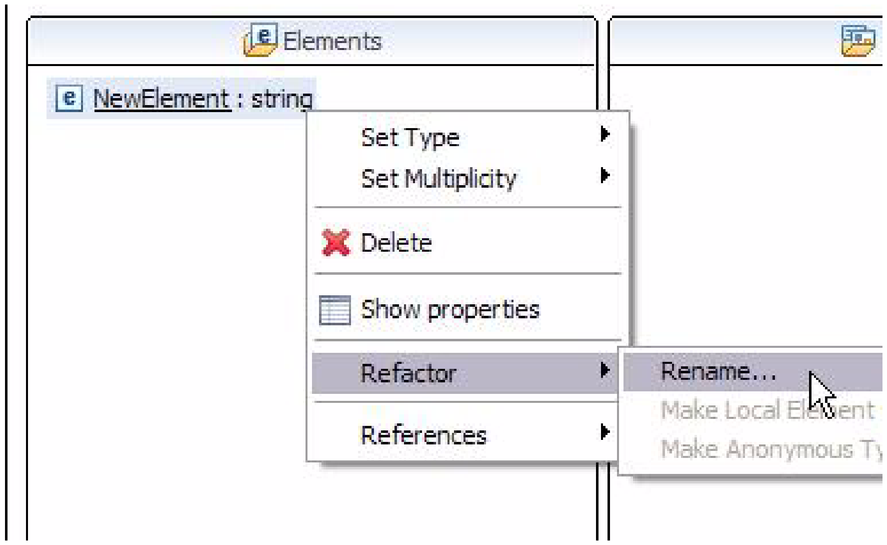
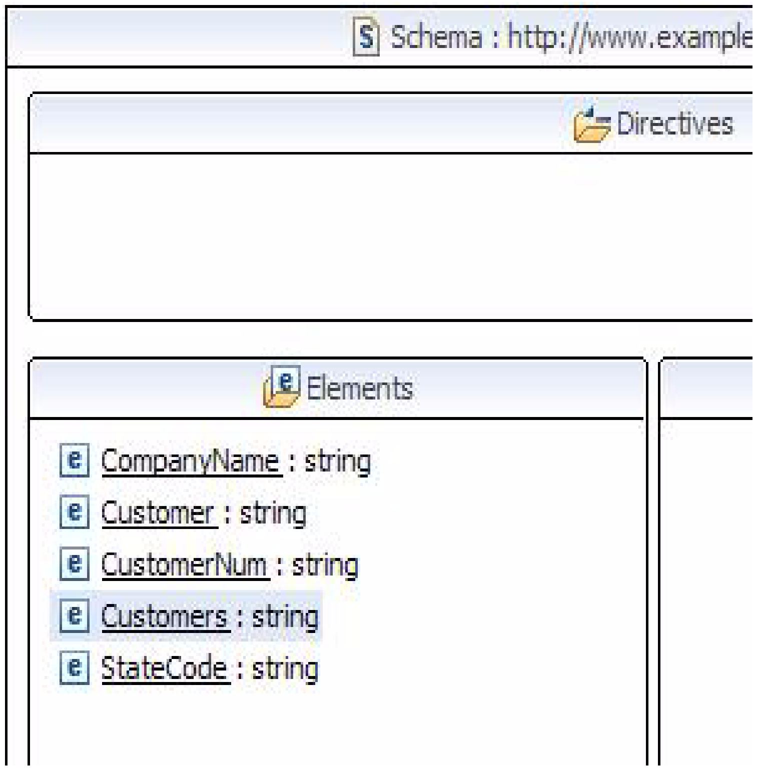


Figure-34, WSD Editor, Elements Pane, Renaming an Element.

j. Summary thus far-

   At this point, your screen should look like Figure-35.



CompanyName : string
Customer : string
CustomerNum : string
Customers : string
StateCode : string

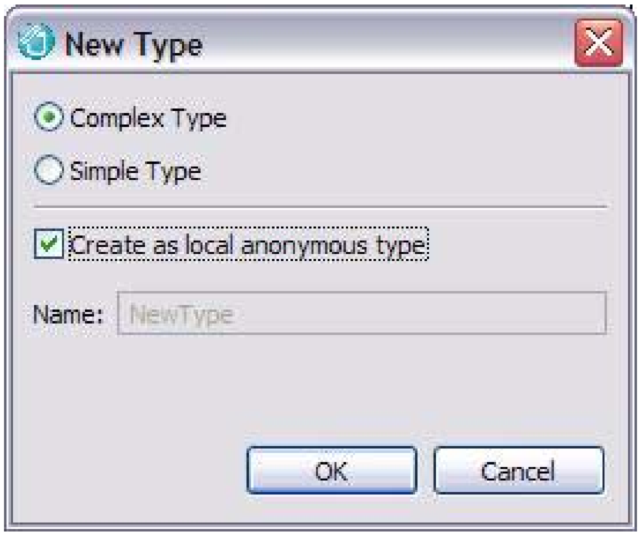
Figure-35, WSD Editor, Current status.

k. Now we are ready to change our Customer Element, so that it may contain the previously created CustomerNum, CompanyName and StateCode Elements.

   Right-Click Customer to produce a context sensitive menu, and select; Set Type => New.

   This action will produce the New Type dialog box as shown in Figure-36.

   Be certain the Complex Type Radio Button visual control is pressed. Check the Create Anonymous Type Check box, and Click OK.



(● Complex Type) ○ Simple Type

[✓] Create as local anonymous type

Name: NewType

Figure-36, WSD Editor, New Type dialog, modifying Customer.

l. As the result of our previous step, Customer is now an Element of Complex Type, and can have the CustomerNum, CompanyName and StateCode embedded within it. (By analogy; Company will act as a record and CustomerNum, CompanyName and StateCode will be its columns.)

   We need to manage the XSD Editor so that we may add (sub) Elements to Customer.

   Double-Click Customer so that we may move to work on Customer in greater detail.

   This action will produce the display as shown in Figure-37.

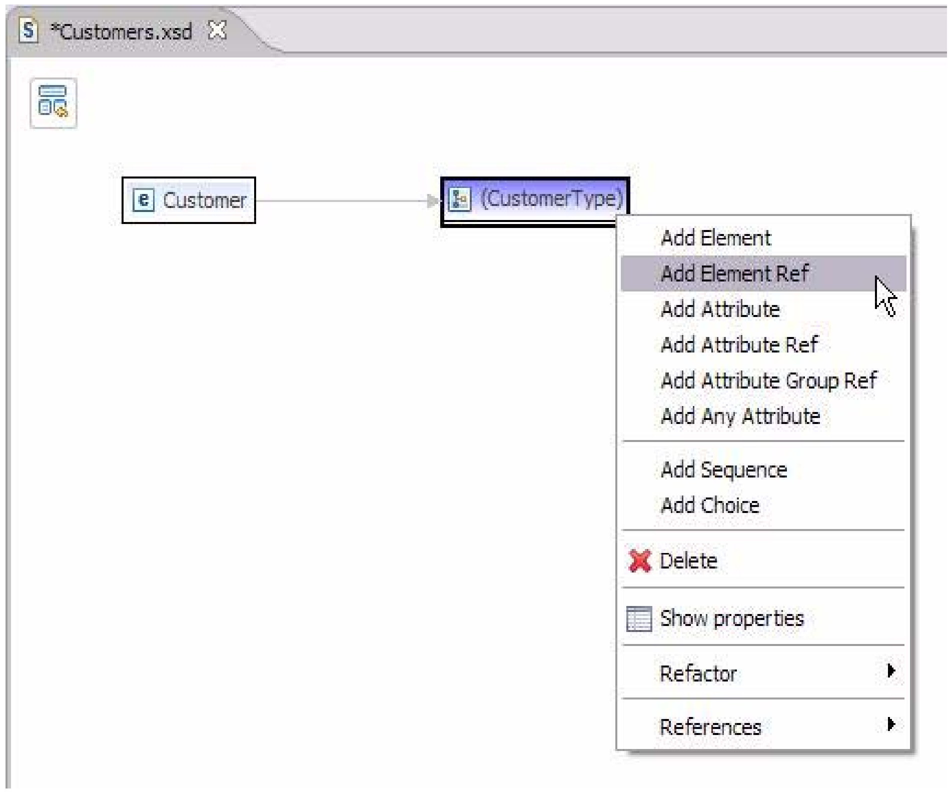


Figure-37, WSD Editor, adding Reference (sub) Elements to Customer.

m. From Figure-27, Right-Click the Customer-Type box to produce a context sensitive menu, and select; Add Element Ref (Reference).

Right-Click on the top row of the Customer-Type box, not inside the box.

This action will add the *first available* Element to Company, probably CompanyName. A Drop Down List Box is available by clicking on CompanyName. With this, our first add, we want CustomerNum. Later we will add CompanyName and StateCode, and will have to use this Drop Down List Box to make the second and third Elements we add to be CompanyName and StateCode, (not just the default first value of CompanyName).

Example as shown in Figure-38.

Use the Browse option to go off and browse for Elements.

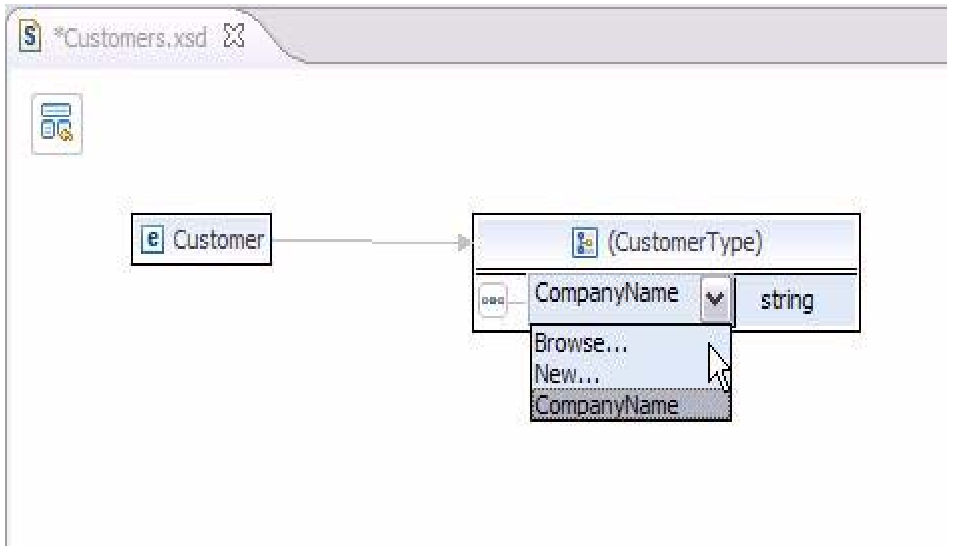


Figure-38, WSD Editor, adding Reference (sub) Elements to Customer.

n. Complete the steps to add CustomerNum, CompanyName and StateCode to Customer.

   When complete, your display should look like Figure-39.

   If you have difficulty getting a Drop Down List Box selection other than the default to remain selected (we did), worse case you can always edit the Source View directly. In effect you are then forced to edit one identifier. Example as shown in Figure-40.

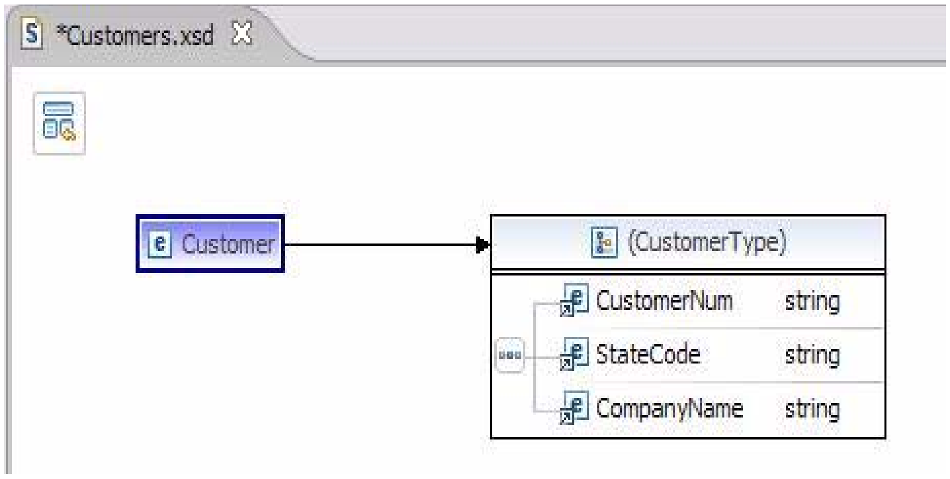
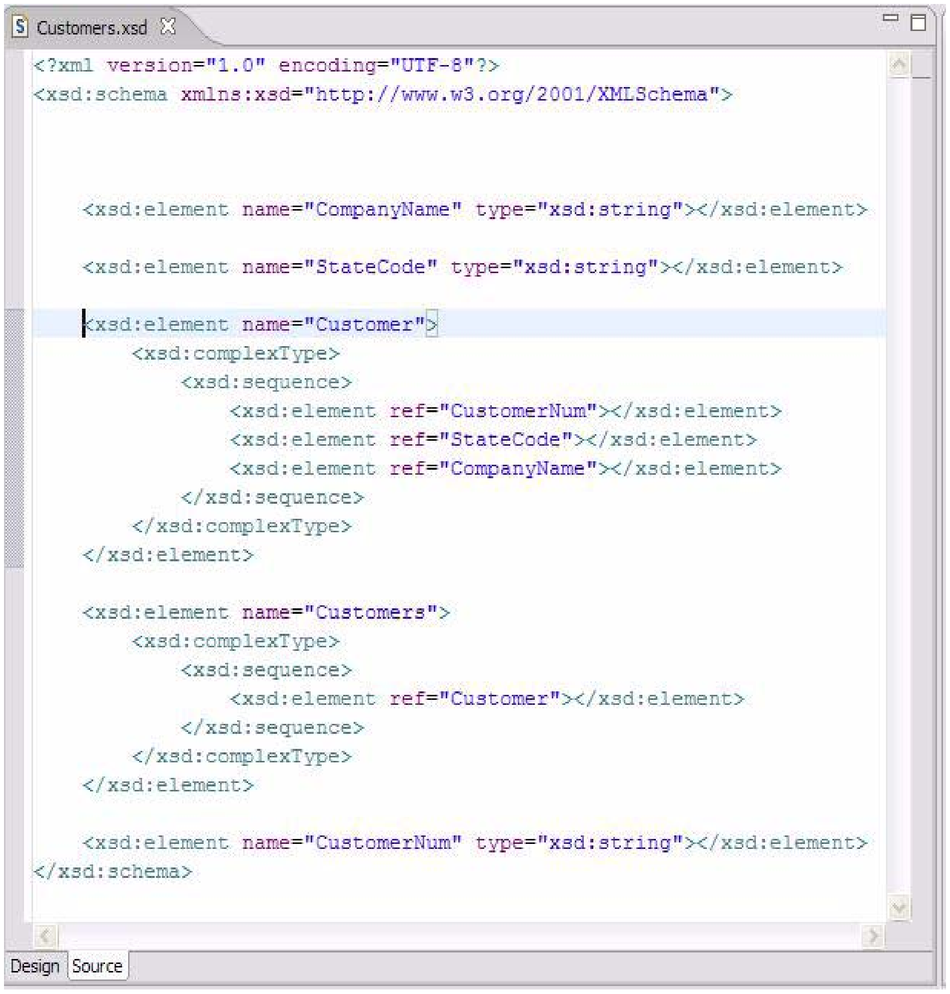


Figure-39, WSD Editor, Customer complete with adding (sub) Elements.



```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">



    <xsd:element name="CompanyName" type="xsd:string"></xsd:element>

    <xsd:element name="StateCode" type="xsd:string"></xsd:element>

    <xsd:element name="Customer">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="CustomerNum"></xsd:element>
                <xsd:element ref="StateCode"></xsd:element>
                <xsd:element ref="CompanyName"></xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="Customers">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="Customer"></xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="CustomerNum" type="xsd:string"></xsd:element>
</xsd:schema>
```

Figure-40, WSD Editor, Source View, Edit if necessary.

o. At this point, Customer is defined to contain the CustomerNum, CompanyName and StateCode Elements. And we are viewing a display where we can work on Customer in detail.

   We need to return to where we can work now on Customers.

   Figure-41 displays an Icon in the upper-left portion of the display that will allow us to return. Click that Icon.

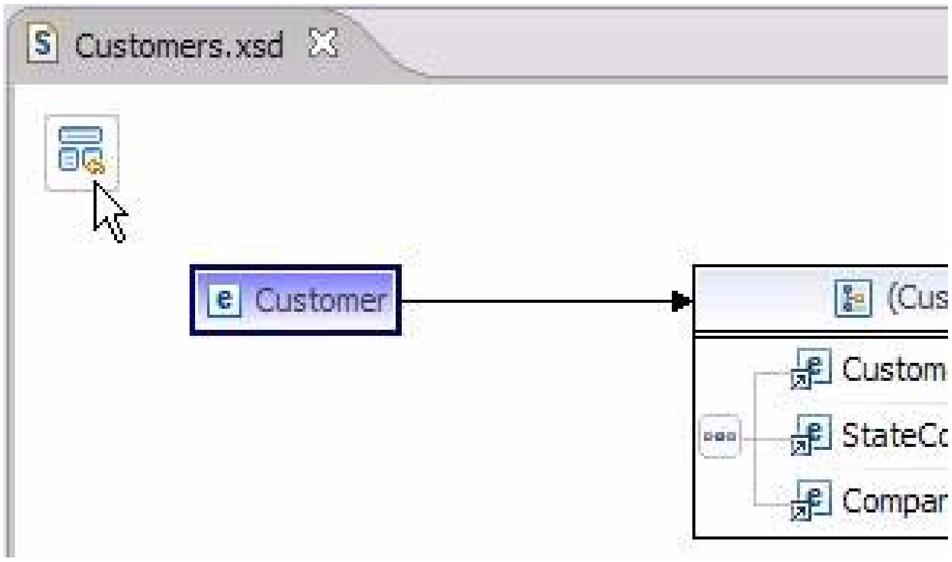


Figure-41, WSD Editor, Icon to return to remainder of model.

p. Now we are ready to do to Customers what we just did to Customer. Instead of adding the three Elements, CustomerNum, CompanyName and StateCode, we are going to add Customer to Customers.

   Right-Click Customers, select; Set Type => New.

   Check Create Anonymous, Click OK.

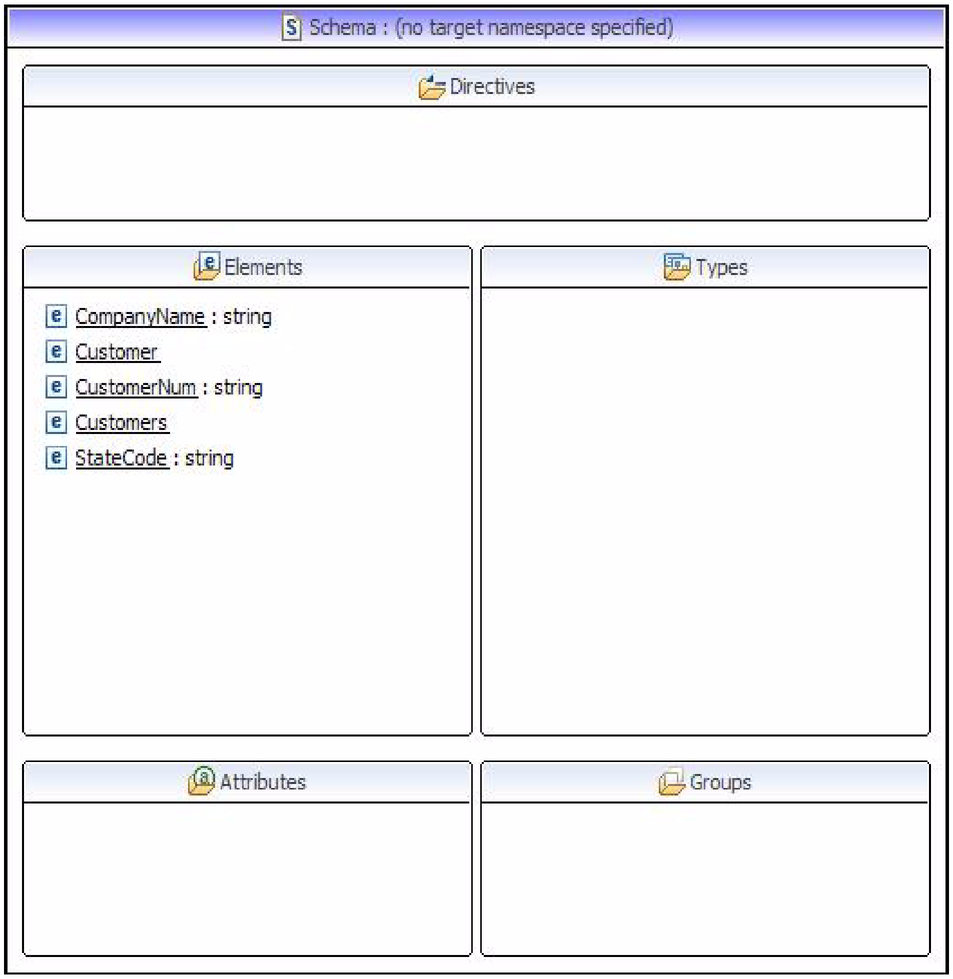
   Double-Click Customers.

   Right-click the top row of the CustomerType box, select; Add Element Ref (Reference).

   Use the Drop Down List Box, which probably contains the value CompanyName. Select Browse, and follow the user interface to select Customer.

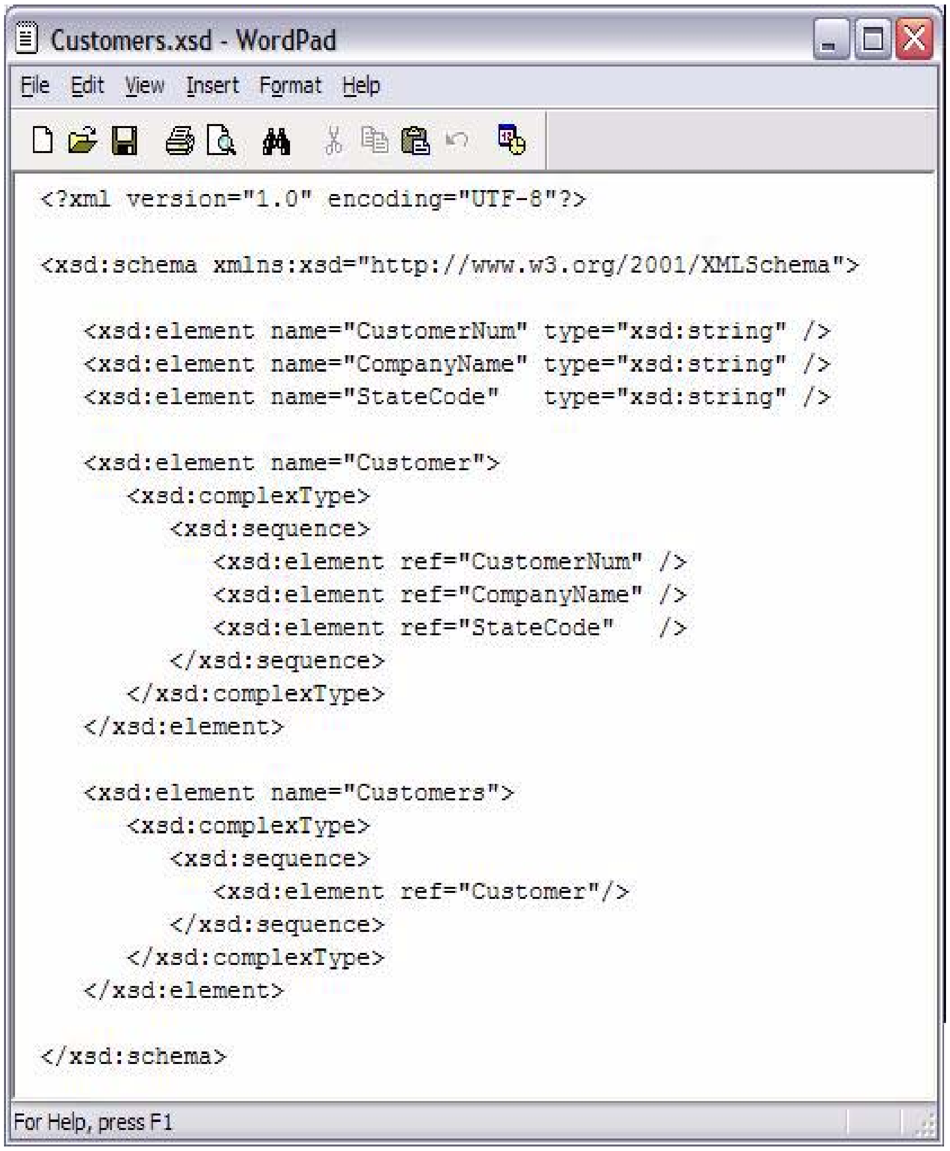
   Remember you can edit the Source View if you have difficulty with the Drop down List Box control.

   The finished product will look like Figure-42 and Figure-43.



CompanyName : string
Customer
CustomerNum : string
Customers
StateCode : string

Figure-42, WSD Editor, Design View, Finished XSD file.



```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <xsd:element name="CustomerNum" type="xsd:string" />
   <xsd:element name="CompanyName" type="xsd:string" />
   <xsd:element name="StateCode"   type="xsd:string" />

   <xsd:element name="Customer">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="CustomerNum" />
            <xsd:element ref="CompanyName" />
            <xsd:element ref="StateCode"   />
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="Customers">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="Customer"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>

</xsd:schema>
```

Figure-43, WSD Editor, Source View, Finished XSD file.

q. At this point we are nearly done. All that remains is to test our work.

   You can save the (Customers.xsd) file in a number of manners. From the Menu Bar select; File => Save All.

   Then from the Navigator View, Right-Click Customers.xsd. from the context sensitive menu that is produced, select; Validate.

   If Validate detects syntactical errors, these will be listed in the Problems View.

   Validate only checks the syntactical correctness of our XSD file. Validate can not tell us whether this file is accurate as a program deliverable; that requires testing.

   To truly test our newly created (Customers.xsd) file, re-run the steps in Sections 6.2 and 6.3 and see if DataStage/QualityStage Meta Data Importer likes our new (Customers.xsd) file and if it produces the correct results.

   The Navigator View works just like Microsoft Windows Explorer. just Right-Click the (Customers.xsd) file and select; Copy. Then paste this file on your Windows Desktop, or another location.

[End] Complete the following steps:

In this section of this document we completed the following:

– We used Rational Data Architect to paint an XML Schema Definition (XSD) file.

– In Rational Data Architect we; Created a Project, created Packages, used the XSD Wizard and XSD Editor, we learned to use an Eclipse based product, and more.

6.5 DataStage/QualityStage EE Job, XML Transformer

As a component to IBM Information Server, DataStage/QualityStage greatly excels at data transformation, doing to your data whatever you require; change its format, aggregate it, move or replicate it, repair it via some lookup or generation, and so on. It is exactly these types of activities that DataStage/ QualityStage is designed for.

As a data enhancing technology, XML can also do data transforms. You can do XML data transforms as a stand alone process outside of DataStage/ QualityStage, or you can do XML data transforms inside DataStage/ QualityStage.

In most cases when performing data transforms, DataStage/QualityStage with or without XML is going to have a huge performance advantage over XML executed outside of DataStage/QualityStage. If nothing else, DataStage/QualityStage is hugely parallel and parallel aware, where XML transforms outside of DataStage/ QualityStage are not. Since you are probably already familiar with the strategic advantages of DataStage/QualityStage, here now we will list random and closing thoughts on this topic:

– When you already have the XML/XSLT program code written and debugged to do the data transform, its makes sense to use that XML/XSLT program code. This code can be run inside of DataStage/QualityStage.

  If you don't already have the XML/XLST program code written, it makes more sense to create this Job on the DataStage/QualityStage Parallel Canvas.

  DataStage/ QualityStage allows you to *paint 96% or more* of these Jobs, making creation and maintenance of these Jobs much easier. XML/XSLT programs are *hand coded* in the XML/XSLT language, which increases your programming and maintenance burden.

– If you need to output HTML formatted data, some folks would argue that an XML/XSLT hand coded program would be better suited to this task than a DataStage/QualityStage painted Job. This is a judgement call; hard to prove or disprove either way.

– DataStage/QualityStage Jobs can run as part of a full Service Oriented Architecture (SOA), Web Services, and/or Java/J2EE application stack. So can XML/XSLT. XML/XSLT can more easily be embedded inside a Java/J2EE Servlet, which is more hand coding and programming. DataStage/ QualityStage offers its routines as Web Services. At that point, we are back to making a judgement call.

– And the list goes on.

DataStage/QualityStage XML Transformer

In previous sections of this document, we examined the DataStage/QualityStage operators named XML Input and XML Output. A third DataStage/QualityStage operator exists for XML related activity; namely, XML Transformer.

XML Input and XML Output can make use of an XML Schema Definition (XSD) file to provide XML formatted input and output. Rational Data Architect allows you to paint XSD files. (We did that activity in a prior section of this document as well.) XML Transformer uses another XML related artifact; namely, XML Style Language - Transforms (XSLT) files.

XSLT files are written in the XSLT programming language. XSLT is a standard's based and open technology, just like XML. In January 2007, release 2.0 of XSLT was finalized as an open specification. While Rational Data Architect supports modelling of XML, it does not support XML programming. In order for Rational Data Architect (RDA) to create an XSLT Program, you will also need the Rational Application Programmer (RAD) component. Installing RDA and RAD on the same workstation is common. The addtional Views and Wizards that come with RAD will simply compliment those that come with RDA. Rational Application Developer and also DataStage/QualityStage support some but not all of the new XSLT 2.0 language syntax, with product updates planned for both. As a language, XSLT somewhat resembles the Unix/Linux Awk(C) programming language. Both are pattern match languages, and declarative in design. (By means of comparison, SQL is a declarative programming language.)

In this section of this document, we will create a new DataStage/QualityStage Enterprise Edition Job that uses the XML Transformer operator. All XML Transformers have a dependency to make reference of an XSLT file. In this section of this document, we will hand code that XSLT file inside a standard editor (Notepad). In the next section of this document, we will use Rational Application Developer to create that XSLT file.

This section of this document was *developed and tested* using WebSphere DataStage/ QualityStage Enterprise Edition version 8.01 on the Microsoft Windows XP/SP2 platform. The steps outlined in this section *may work* on DataStage Server Edition 8.x or 7.x with only minor modification.

Figure-44 displays the DataStage/QualityStage EE Job that we are about to create.

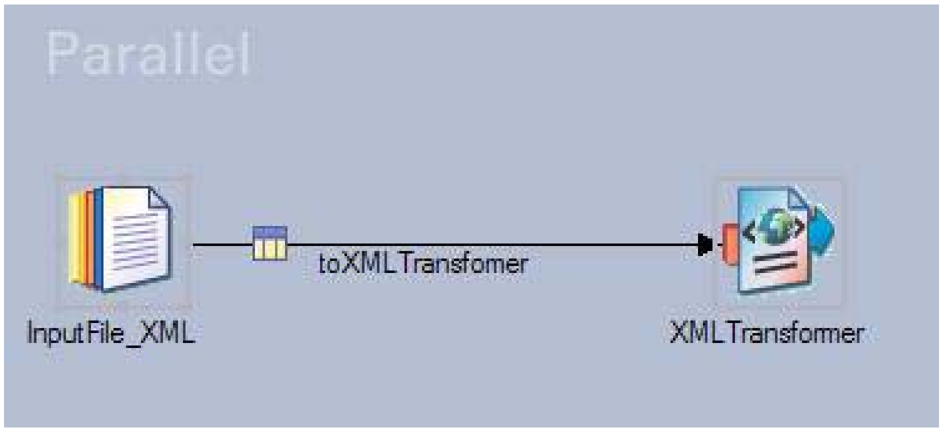
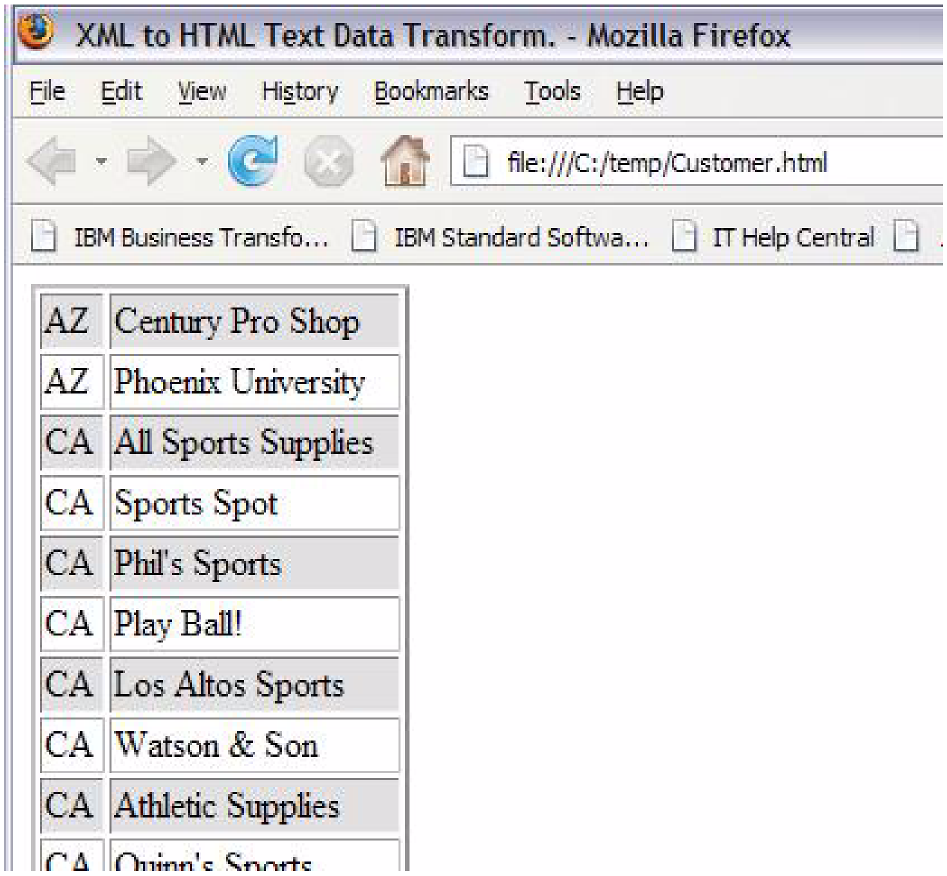


Figure-44, DataStage/QualityStage EE Job, XML Transformer.

Figure-45 displays what this DataStage/QualityStage Job will output; an HTML formatted file (Customers.html). The transform we will perform is from XML to HTML formatted data, including a re-sort of the data by a non-key value.



AZ	Century Pro Shop
AZ	Phoenix University
CA	All Sports Supplies
CA	Sports Spot
CA	Phil's Sports
CA	Play Ball!
CA	Los Altos Sports
CA	Watson & Son
CA	Athletic Supplies
CA	Quinn's Sports

Figure-45, Output from XML Transformer, (Customers.html).

[Begin] Complete the following steps:

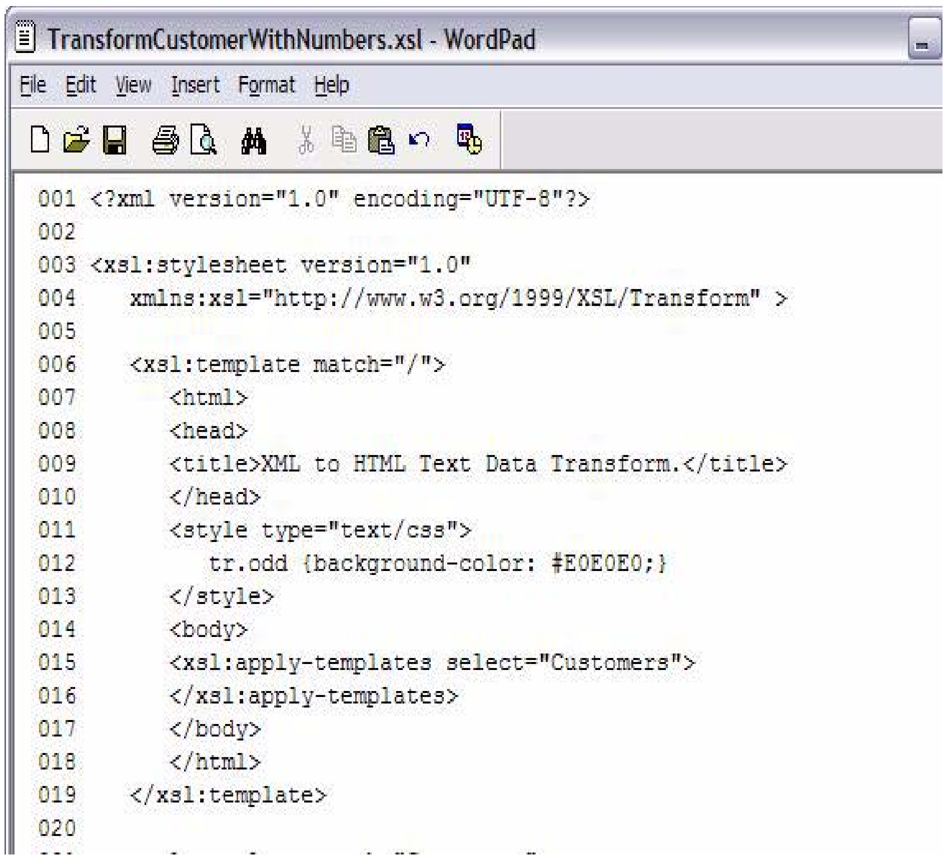
1. Create an XSLT file using a text editor.

   First, let's create the XML Style Language - Transform (XSLT) file we will need for this DataStage/QualityStage XML Transformer Job.

   In the next section, Section 6.6, "Create an XSLT file with Rational Application Developer" on page 72, we will create this XSLT file using Rational Application Developer.

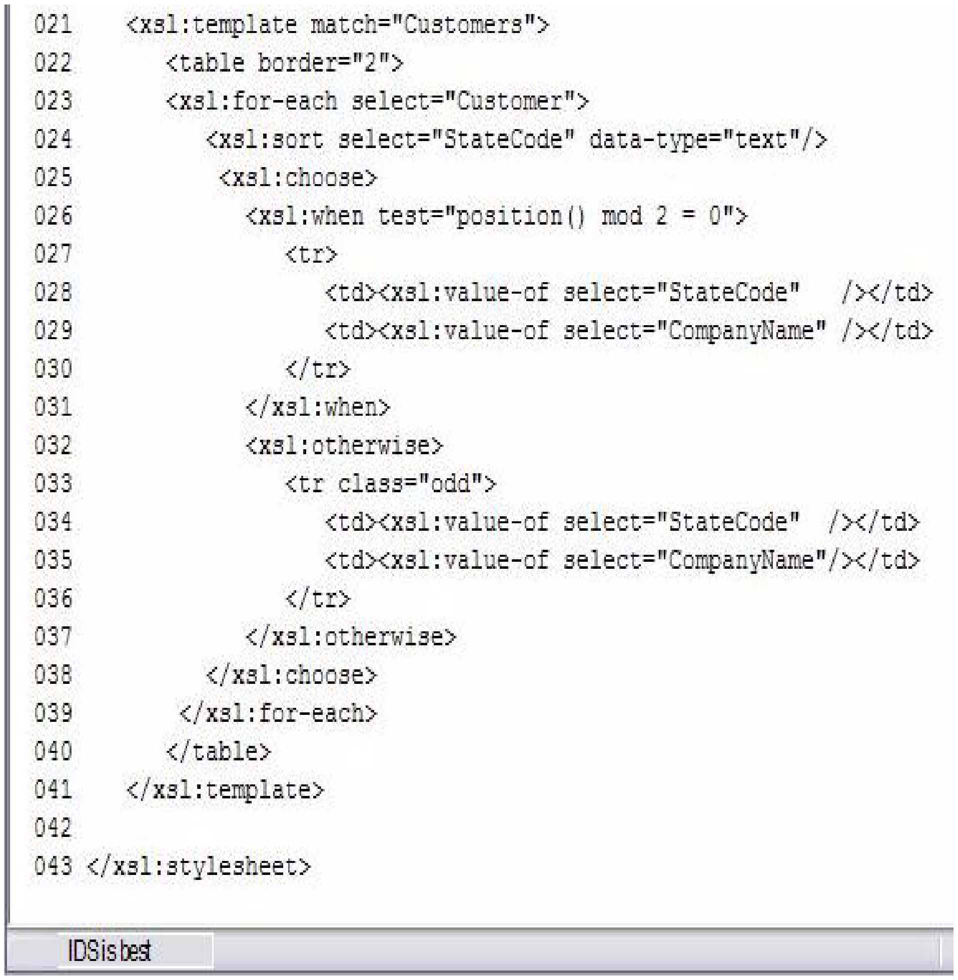
Using a text editor or any other compatible method, create the (TransformCustomers.xsl) ASCII Text File. See Figures-46/47 below.

The line numbers displayed in this file are for instructional purposes only. Do not enter the line numbers, "001", "002", "003", etcetera.


TransformCustomerWithNumbers.xsl - WordPad
File Edit View Insert Format Help

```
001 <?xml version="1.0" encoding="UTF-8"?>
002
003 <xsl:stylesheet version="1.0"
004    xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
005
006    <xsl:template match="/">
007       <html>
008       <head>
009       <title>XML to HTML Text Data Transform.</title>
010       </head>
011       <style type="text/css">
012          tr.odd {background-color: #E0E0E0;}
013       </style>
014       <body>
015       <xsl:apply-templates select="Customers">
016       </xsl:apply-templates>
017       </body>
018       </html>
019    </xsl:template>
020
```

Figure-46, (TransformCustomer.xsl) XSLT file, displaying page 1 of 2.



```
    <xsl:template match="Customers">
      <table border="2">
      <xsl:for-each select="Customer">
        <xsl:sort select="StateCode" data-type="text"/>
         <xsl:choose>
           <xsl:when test="position() mod 2 = 0">
             <tr>
               <td><xsl:value-of select="StateCode"   /></td>
               <td><xsl:value-of select="CompanyName" /></td>
             </tr>
           </xsl:when>
           <xsl:otherwise>
             <tr class="odd">
               <td><xsl:value-of select="StateCode"  /></td>
               <td><xsl:value-of select="CompanyName"/></td>
             </tr>
           </xsl:otherwise>
        </xsl:choose>
       </xsl:for-each>
      </table>
    </xsl:template>

</xsl:stylesheet>
```

Figure-47, (TransformCustomer.xsl) XSLT file, displaying page 2 of 2.

2. Launch the DataStage/QualityStage component of IBM Information Server, and connect to an available Project.
3. Create a DataStage/QualityStage Parallel Job.
   a. From the Repository View, and then the Jobs (Folder), Right-Click and select; New => Parallel Job.
   b. From the Menu Bar, select; File => Save As, and give the Job a name. We called our Job “PJ03_XMLTransformer”.

   c. From the Palette View, File (Drawer), Drag and Drop a Sequential File object to the Parallel Canvas. We renamed this object to "InputFile_XML".

   d. From the Palette View, Real Time (Drawer), Drag and Drop an XML Transformer object to the Parallel Canvas. We renamed this object to "XMLTransformer".

   e. On the Parallel Canvas, Right-Click the Sequential File object (InputFile_XML) and draw a Link to the XML Transformer object (XMLTransformer). We renamed this Link to "toXMLTransformer".

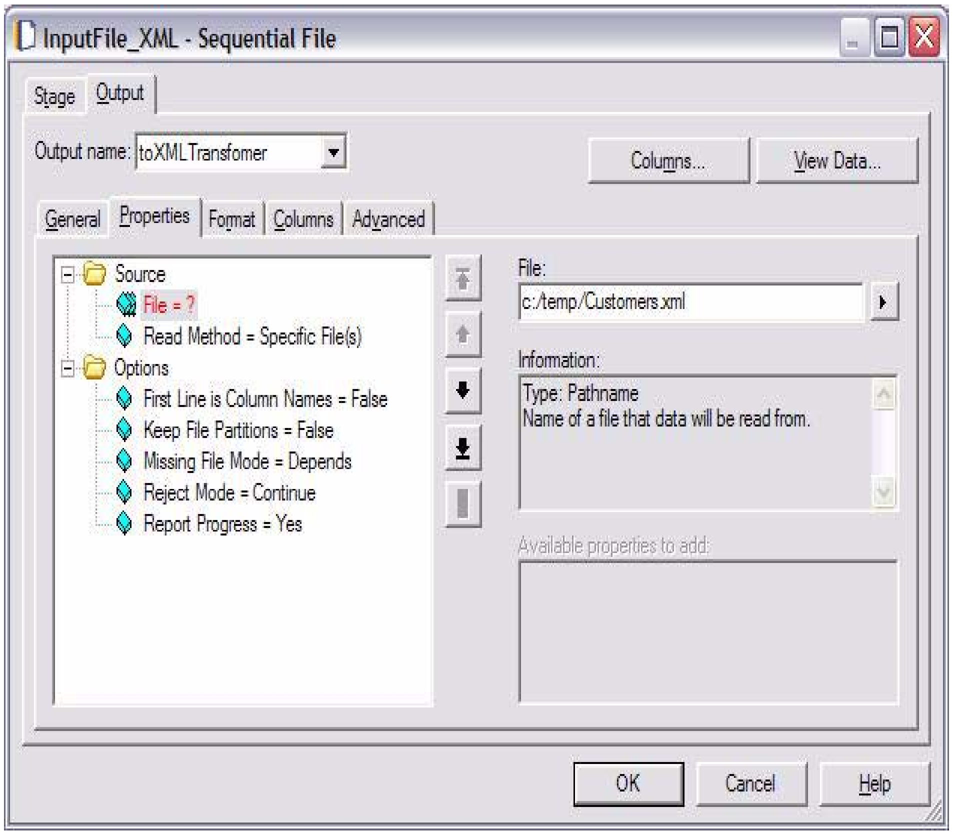
      Figure-44, above, displays the proper result of these steps.

4. Set Properties related to (input) Sequential File object.

   There are just a few Properties we need to set in each of the Sequential File object (InputFile_XML) and the XML Transformer object (XMLTransformer).

   a. Double-Click the Sequential File object (InputFile_XML). This action produces the display in Figure-48. There are three distinct TABs we are going to access.

      In Figure-48, change the Output TAB => Properties (sub) TAB => Source => File (name), to equal the path name to the (Customers.xml) XML formatted data file we create earlier.


InputFile_XML - Sequential File

Stage | Output

Output name: toXMLTransfomer

Columns... | View Data...

General | Properties | Format | Columns | Advanced

- Source
  - File = ?
  - Read Method = Specific File(s)
- Options
  - First Line is Column Names = False
  - Keep File Partitions = False
  - Missing File Mode = Depends
  - Reject Mode = Continue
  - Report Progress = Yes

File: c:/temp/Customers.xml

Information:
Type: Pathname
Name of a file that data will be read from.

Available properties to add:

OK | Cancel | Help

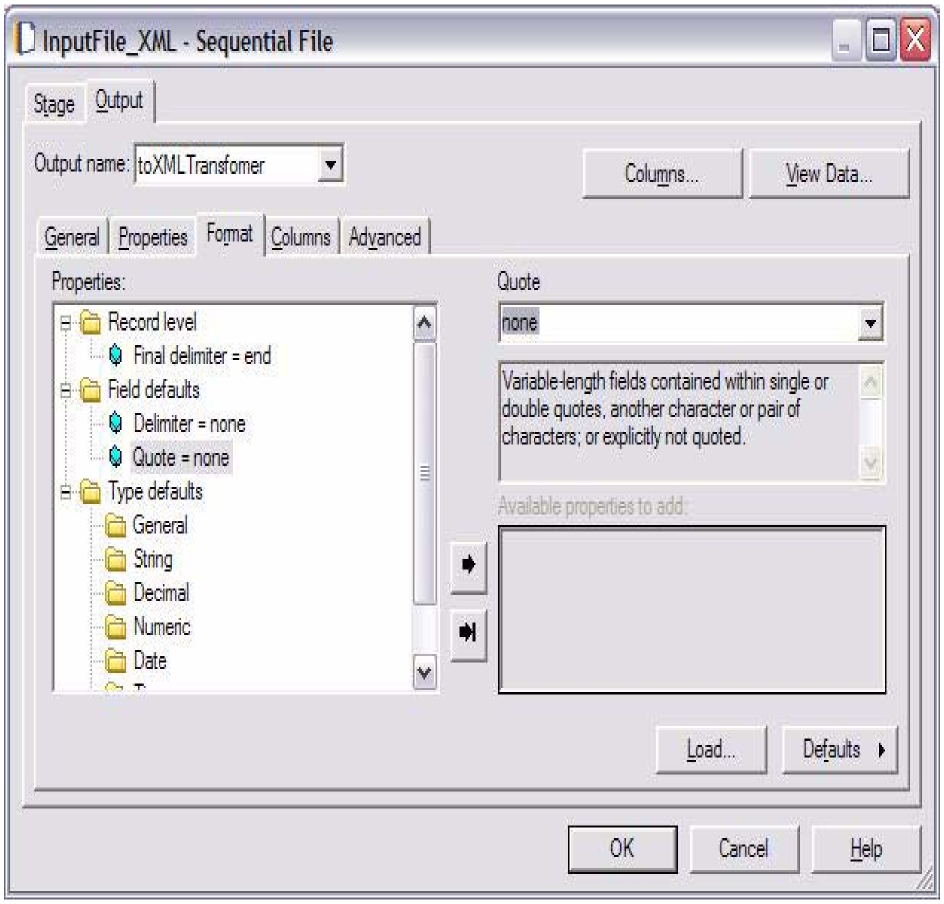
Figure-48, Sequential File object, Output TAB, Properties (sub) TAB.

b. If Figure-49, change the Output TAB, Format (sub) TAB, and then three properties here;

   Set the Record level => Final delimiter to "end". In this context, "end" means end of file.

   Set the Field defaults => Delimiter to "none". In the case of XML formatted input, which we have here, XML Input and XML Transformer use the XML beginning and ending Tags as field delimiters.

   Set the Field defaults => Quote to "none".


InputFile_XML - Sequential File

Output name: toXMLTransfomer

Properties:
- Record level
  - Final delimiter = end
- Field defaults
  - Delimiter = none
  - Quote = none
- Type defaults
  - General
  - String
  - Decimal
  - Numeric
  - Date

Quote: none

Variable-length fields contained within single or double quotes, another character or pair of characters; or explicitly not quoted.

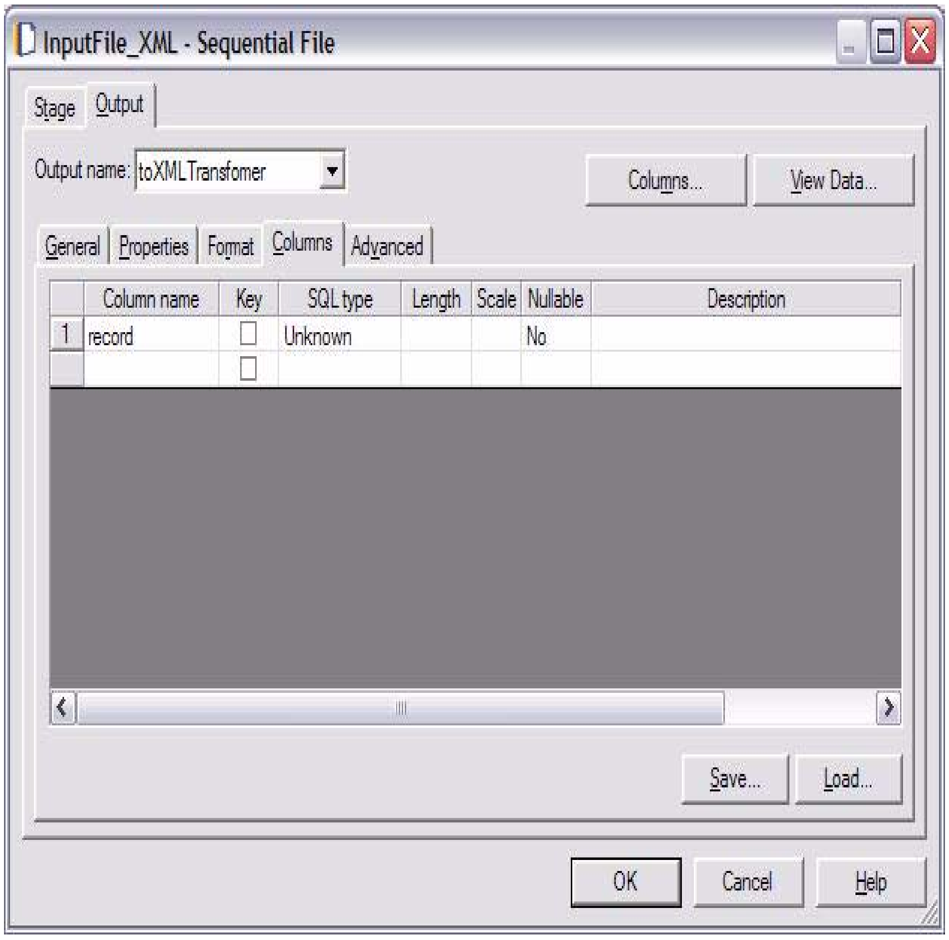
Figure-49, Sequential File object, Output TAB, Format (sub) TAB.

c. In Figure-50, change the Output TAB => Columns (sub) TAB as shown.

   Here we are following our standard practice of naming the input data column "record", and making it of type unknown. Again, "record" is not a keyword. We could use any value here.

   We are done with the Sequential file object (InputFile_XML).

   Click OK to close this dialog box.



Output name: toXMLTransfomer

	Column name	Key	SQL type	Length	Scale	Nullable	Description
1	record	☐	Unknown			No	
		☐					

Figure-50, Sequential File object, Output TAB, Columns (sub) TAB.

5. Set Properties related to XML Transformer object.

    Now we are ready to complete this Job. We only have a few properties to set within the XML Transformer object and we are ready for testing.

    In the case we are creating here, XML Transformer will modify our input stream per the instructions of an XSLT file. And the XML Transformer will also direct our output to a file, (Customers.html). In the real world, XML Transformer could output to another XML Transformer object (perform further data transformation), or XML Transformer could output to a Web Service (DataStage/Quality/Stage WISD Output object).

a. Double-Click the XML Transformer object (XMLTransformer). This action produces the display in Figure-51. There are three distinct TABs we are going to access.

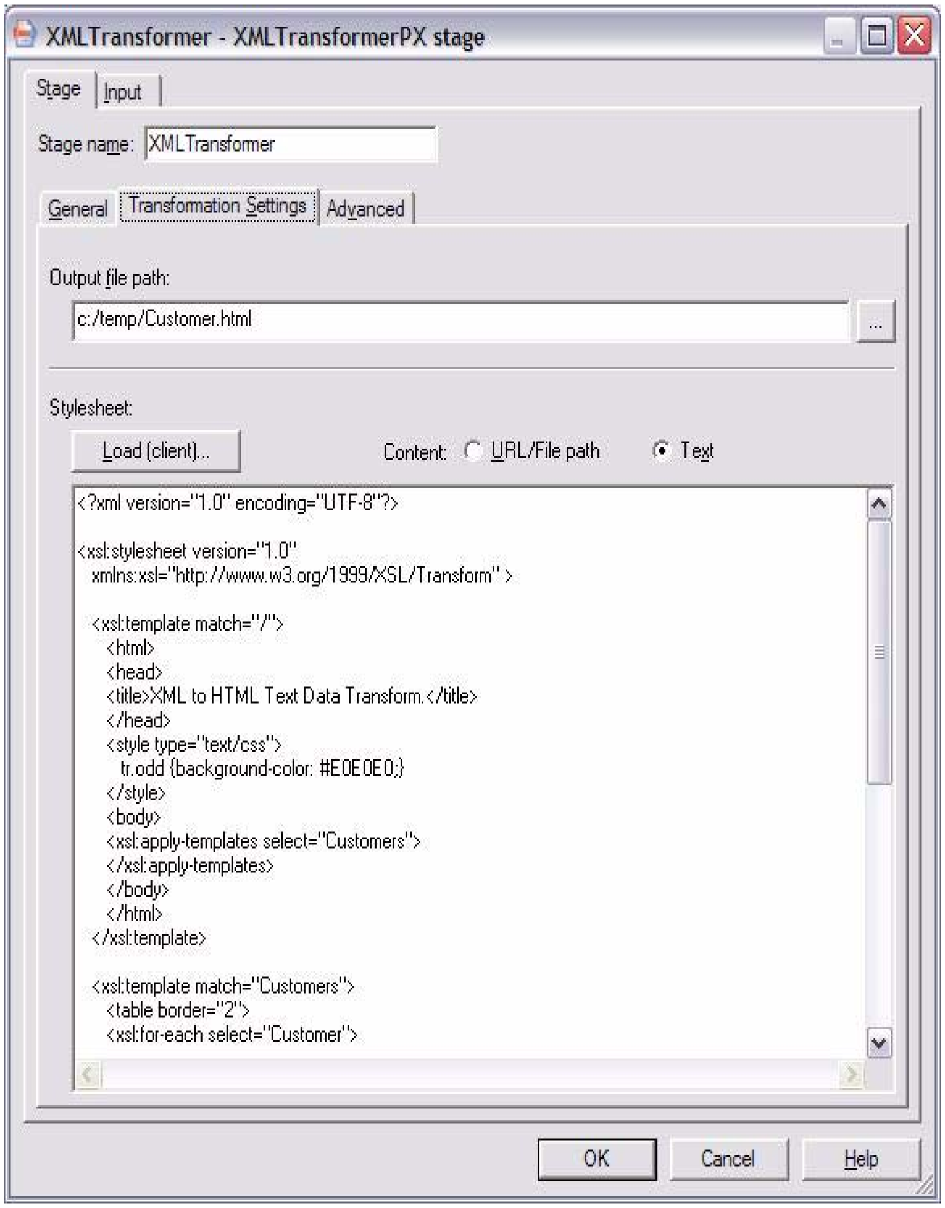
   In Figure-51, change the Stage TAB => Transformation Settings (sub) TAB => Output file path value to our desired output file name. We used (c:/temp/Customers.html).

   The XSLT code we are using outputs HTML; hence, our output file name suffix is HTML.

   Click the Stage TAB => Transformation Settings (sub) TAB => “Load Client..” Button, and Browse to the location of our previously created (TransformCustomer.xsl) file.

   All we are doing here is loading our previously created ASCII text file, which is in XSLT format.

   Make your display look like Figure-51.



Stage name: XMLTransformer

Output file path: c:/temp/Customer.html

Stylesheet:

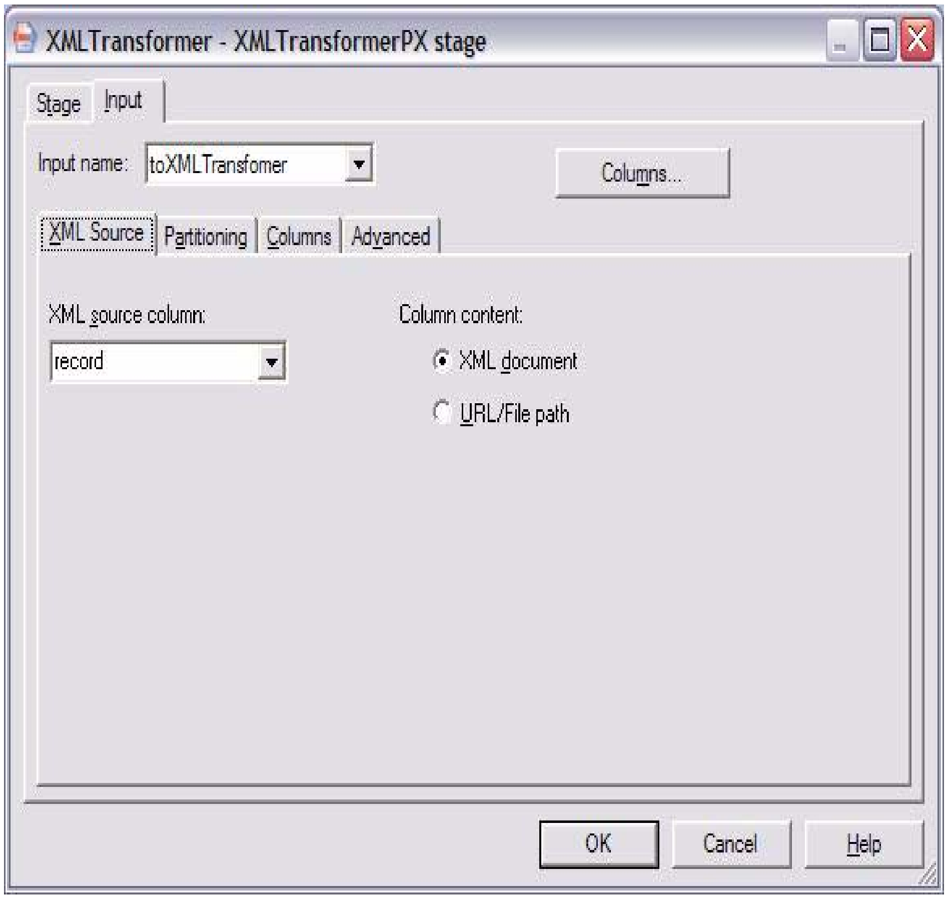
Content: ( ) URL/File path [✓ Text]

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >

  <xsl:template match="/">
    <html>
    <head>
    <title>XML to HTML Text Data Transform.</title>
    </head>
    <style type="text/css">
      tr.odd {background-color: #E0E0E0;}
    </style>
    <body>
    <xsl:apply-templates select="Customers">
    </xsl:apply-templates>
    </body>
    </html>
  </xsl:template>

  <xsl:template match="Customers">
    <table border="2">
    <xsl:for-each select="Customer">
```

Figure-51, XML Transformer object, Stage TAB, Transformation Settings (sub) TAB.

b. In Figure-52, change the XML Transformer => Input TAB => XML Source (sub) TAB => XML source column value to "record". This is the input column from our Sequential File object input data source.

   Click the XML Transformer => Input TAB => XML Source (sub) TAB => Column content Radio Button so that "XML document" is pressed.



Input name: toXMLTransfomer

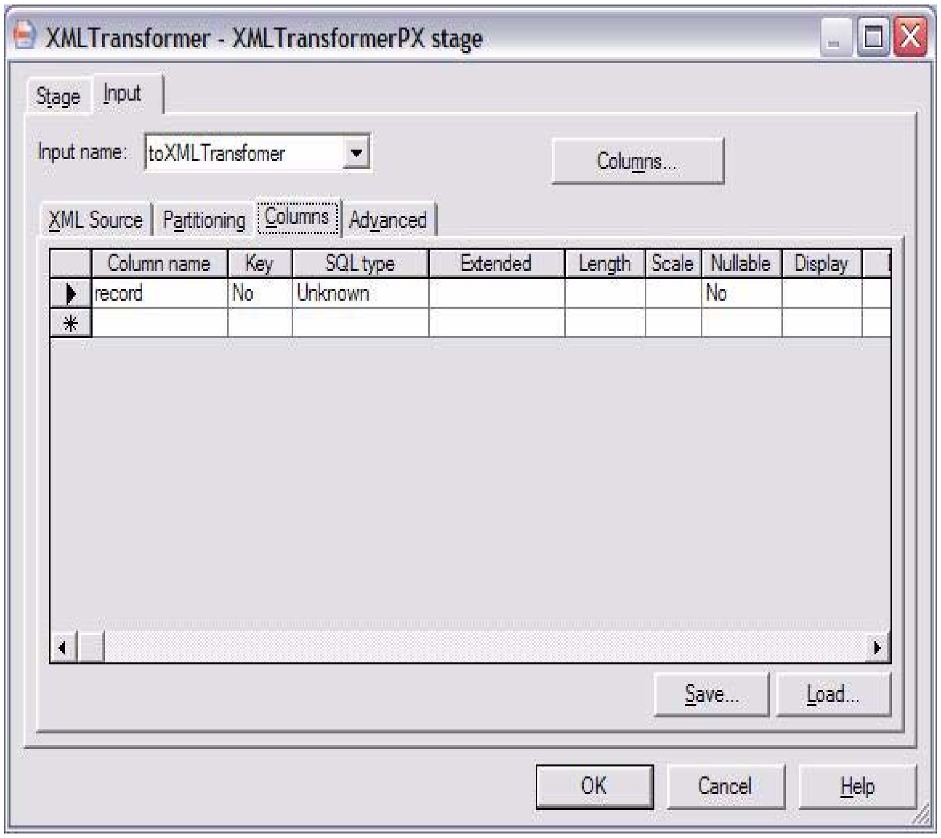
XML source column: record

Column content: [✓ XML document] URL/File path

Figure-52, XML Transformer object, Input TAB, XML Source (sub) TAB.

c. In Figure-53, ensure that the XML Transformer => Input TAB => Columns (sub) TAB appears as shown.

   When you are done, Click OK.



	Column name	Key	SQL type	Extended	Length	Scale	Nullable	Display
▶	record	No	Unknown				No	
*								

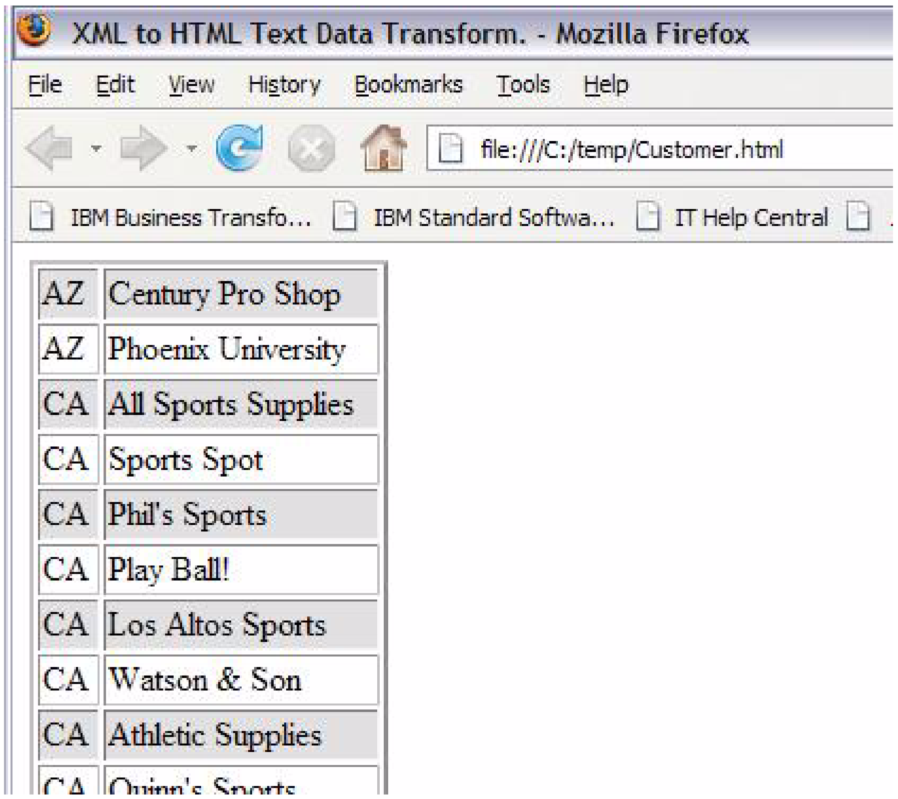
Figure-53, XML Transformer object, Input TAB, Columns (sub) TAB.

6. Compile, Run, and Test our Job.

   Now we are ready to compile and run our Job, and view our results.

   a. From the Tool Bar, Compile and then Run this Job (PJ03_XMLTransformer). The output of this Job will be an HTML formatted data file in the location you specified above. (We used "c:/temp/Customers.html".)

b. Open the HTML Output file (Customers.html) using a Web browser, Microsoft Internet Explorer or Mozilla Firefox. You should produce a display similar to Figure-54 below.


XML to HTML Text Data Transform. - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/temp/Customer.html
IBM Business Transfo...
IBM Standard Softwa...
IT Help Central

AZ	Century Pro Shop
AZ	Phoenix University
CA	All Sports Supplies
CA	Sports Spot
CA	Phil's Sports
CA	Play Ball!
CA	Los Altos Sports
CA	Watson & Son
CA	Athletic Supplies
CA	Quinn's Sports

Figure-54, (Customers.html), output from XML Transformer.

[End] Complete the following steps:

In this section of this document we completed the following:

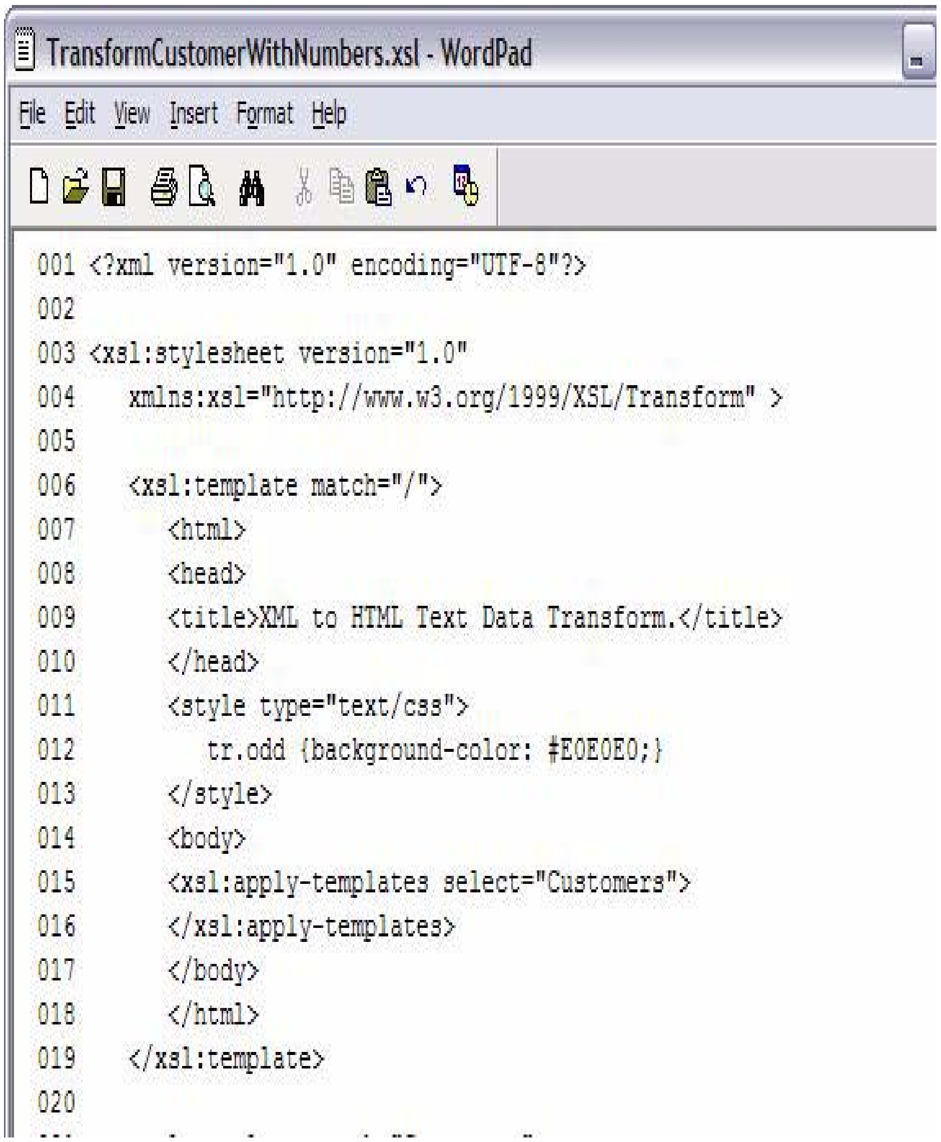
We created and ran a DataStage/Quality Stage XML Transformer object Job, read an XML formatted input data file, and output an HTML formatted data file.

We largely just threw that (Customers.xsl) file at you. In the next section, we will perform a line by line code review of that file (Customers.xsl), and create that file visually using Rational Data Architect and Rational Application Developer. Still, however, if you are going to a life of programming XML Style Language - Transformation (XSLT), you are going to have to learn XSLT. It may take you one hour to learn the basics of XSLT programming, and then maybe 30 minutes a week to keep those skills instantly recoverable.

6.6 Create an XSLT file with Rational Application Developer

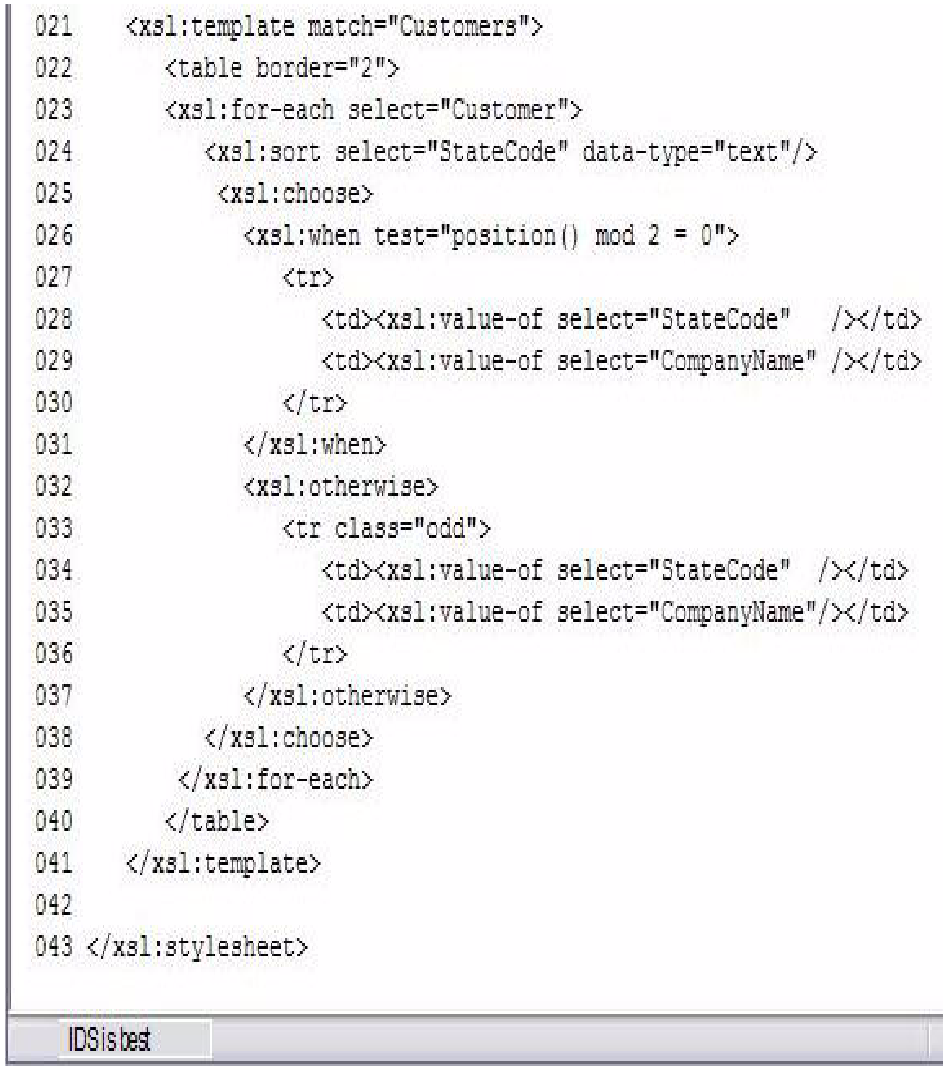
In this section we resolve a dependency we had above, from Section 6.5, "DataStage/QualityStage EE Job, XML Transformer" on page 57. In this section, we explain and the visually create our XML Style Language - Transformation (XSLT) file.

Figures-55/56 display the contents of our XML Style Language - Transformation (XSLT) file. We called this file (TransformCustomer.xsl). After display of this file, we will perform a line by line code review.


TransformCustomerWithNumbers.xsl - WordPad
File Edit View Insert Format Help

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >

   <xsl:template match="/">
      <html>
      <head>
      <title>XML to HTML Text Data Transform.</title>
      </head>
      <style type="text/css">
         tr.odd {background-color: #E0E0E0;}
      </style>
      <body>
      <xsl:apply-templates select="Customers">
      </xsl:apply-templates>
      </body>
      </html>
   </xsl:template>

```

Figure-55, (TransformCustomer.xsl) XSLT file, displaying page 1 of 2.



```
    <xsl:template match="Customers">
       <table border="2">
       <xsl:for-each select="Customer">
          <xsl:sort select="StateCode" data-type="text"/>
           <xsl:choose>
             <xsl:when test="position() mod 2 = 0">
                <tr>
                   <td><xsl:value-of select="StateCode"   /></td>
                   <td><xsl:value-of select="CompanyName" /></td>
                </tr>
             </xsl:when>
             <xsl:otherwise>
                <tr class="odd">
                   <td><xsl:value-of select="StateCode"  /></td>
                   <td><xsl:value-of select="CompanyName"/></td>
                </tr>
             </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each>
       </table>
    </xsl:template>

</xsl:stylesheet>
```

Figure-56, (TransformCustomer.xsl) XSLT file, displaying page 2 of 2.

The following is a line by line code review of the (TransformCustomer.xsl) XML Style Language - Transformation (XSLT) file:

- **Again: The line numbers displayed in this file are for instructional purposes only. Do not enter the line numbers, "001", "002", "003", etcetera.**
- Line-001, is technically optional, but should be considered mandatory.
  - Much like the first line of an HTML file, this line declares the (XML) Version and (Language) Encoding.
  - XML lines that begin with "<?" and end with "?>" are called Processing Instructions. Aside from the example of declaring the Version and (Language) Encoding, Processing Instructions are rarely seen or used in XML.
- Line-003, this line is mandatory inside every XSLT file. This line is paired with (terminated by) line-043, and is called the "(XML Stylesheet) Namespace Declaration".
  - The URL entered on this line (this is a standard/reference value), and its standard associated Tag Prefix of "xsl" (also standard), refers to the Base Schema and Language Specification for XML Stylesheets.

    The Base Schema and Language Specification for XML Stylesheets is to XML and XSLT files, what the Object Class Repository is to Java; it defines the API and objects contained in the core XML/XSD language specification.
  - In effect, this line is an *include*; meaning, there are items we can refer to inside this file that may be defined elsewhere.
  - Namespaces" are a core concept to XML and are worthy of further study. For example, we could place items (functions) into a central repository to be re-used by our business. We would associate these resources with an "xsl" type Tag Prefix, and import them (allow reference to) at a later time.
- Line-006, begins our first real area of XSLT programming in this file. This line is paired with (terminated by) line-019. If you can understand this block of XSLT code, then you have the means to understand all of XSLT.
  - This block begins with a "template match" phrase. With the match pattern of "/", we will match the Root Element of our XML document, (which happens to be Customers). All XML documents are organized in a hierarchy with a single Root Element, and then (zero or more sub or) nested Elements.

    Because there is only one Root Element to any well formed XML document, this block of code will only execute one time. This block of code will execute at the start of our file or input stream, which is where the Root Element is located.

- Remember that XSLT is a pattern match language. XSLT is organized around blocks of code (similar to function calls) that perform a given set of instructions. This block of code is executed when an (input) pattern is received. Since XML formatted data is organized in a hierarchy of matched Tags, XSLT will look for these beginning and ending (matched) Tags.
- Our input document, (our input data stream) contains XML formatted data in a hierarchical format. The whole list of data is called (tagged as) Customers. Within Customers we have zero or more Customer (records). A Customer (record) has three Elements embedded with it; CustomerNum, CompanyName, and StateCode. CustomerNum, CompanyName, and StateCode act as (columns) to Customer.

  We call Customer a (record) and CustomerNum a (column) to draw comparison to SQL, which we already know. Technically, Customers, and Customer are Elements of Complex Type, CustomerNum, CompanyName, and StateCode are Simple Elements.
- Above we match the pattern of "/", which means the root Element of our XML formatted document. In our case, the Root Element is Customers. Other patterns we could have matched on include: Customer, CustomerNum, CompanyName, and StateCode.

  As a Root Element, the Customers Tag will only be found one time. Any "template match" block for the Root element will only execute one time. Any "template match" block for other (sub) Elements, will execute once every time that that Tag is received in the XML file (input data stream).

– Lines-007 thru 014, are not XML. These lines are HTML.

  Lines-007 thru 014 are located in a "template match" block which is looking for the "/" pattern. This pattern only occurs one time, at the start of our file.

  As HTML source code, these lines form the top half of a standard HTML document. Lines-011 thru 013 are an HTML Inline Cascading Style Sheet definition which give us the alternating White and Grey lines in our HTML Table.

  XSLT will output these lines exactly as they appear, one time only.

– Lines-015 thru 016 are XSLT source code.

  Because lines-015 thru 016 are in this pattern block, they will execute only one time. These lines instruct XSLT to go execute the "Customers" pattern block, whatever that contains.

- Lines-017 thru 018 are also HTML source code, just like lines-007 thru 014 above. Lines-017 thru 018 form the bottom half of a standard HTML document.
- Line-019 terminates line-006 above.
- Line-021 begins our second "pattern match" block; essentially our second function call. Line-021 is paired with (terminated by) line-041.

  Any output from this block (function) will be placed in sequence with whatever is output after whatever line-014 outputs, and before whatever line-017 outputs.
- Line-022 is HTML source code.

  In HTML this line states to begin a presentation of (data) in a tabular form. This table will happen to have two columns, as determined by the count of "<td>" Tags (table data, or cell) between each pair of "<tr>" Tags (row tags).
- Line-023 begins a For-Loop. Line-023 is paired with (terminated by) line-039.

  Because this line is inside a block that is executed once, this loop will execute once. This loop is instructed to loop through Customer, which in our case, is our basic record of which we have many.
- Line-024 instructs XSLT to sort the input file (the input data stream) by the data contained in the XML formatted column "StateCode".

  This line may appear to be out of sequence. You may think it would be more natural to have line-024 appear above line-023. All we can say is that XSLT is not procedural programming language. XSLT is a declarative programming language. If it helps, try to think of this line as a modifier or attribute to the line above, line-023. Sort statements can not appear randomly in XSLT. Sort statements must directly follow lines like For-Loops, line-023, and related.

  Because this block executes one time at start up, this is a one time at start up For-Loop and sort. Because CustomerNum and CompanyName are bundled, in effect, with StateCode inside a Customer Element, those columns will follow the sort of StateCode.
- Line-025 begins what is basically a big Case statement, or If-Else block. In programming parlance, this is a Flow control statement (or block). Line-025 is paired with (terminated by) line-038.
  - Lines-026 and 032 are the test cases for this Case statement. If line-026 returns true upon test, then lines-027 thru 030 are output. If line-026 returns false, then lines-033 thru 036 are output.

  - “position()” is an XSLT language provided automatic counter of the number of Elements (records) counted by this “pattern block”. “position()” will count 1, 2, 3, and so on as each Customer Element is received.
  - By means of the “mod 2” evaluation, we have created an If statement that alternates every other row (every other Customer Element). This is how we get the alternating White and Gray bars in our HTML Table display.
- Line-028 is an example of how we retrieve (and effectively print) the value of any given Element. On line-034 we print StateCode. Other “value-of” lines are instructed to print (output) other element values.
- The remainder of the lines in this file are simply end Tags to Tags listed above.

Our example is a very basic XSLT (program). With the release of the XSLT 2.0 specification in January 2007, XSLT becomes very powerful. XSLT, and then XML in general, can input and output XML formatted data, Comma Separated Value (CSV) data, and much much more.

Prior to release 2.0 of the XSLT specification, certain things were very hard to do in XSLT; things like collapsing detail data and outputting aggregate summaries, handling disparate but concurrent multiple input and/or output data streams, and so on. (These are all things that DataStage/QualityStage has always done well.)

Now that we understand out XSLT program file, let’s create this file in Rational Data Architect and Rational Application Developer. Because the HTML we are using is very simple, we will just add this to our XSLT program code. If the HTML were more difficult, we could use Rational Application Developer, the sister product to Rational Data Architect, to first paint out HTML, and then add the XSLT language constructs.

[Begin] Complete the following steps:

1. Launch the Rational Data Architect tool with the Rational Applicaton Developer components also installed.

   Use the same Workspace and Project we used above in, Section 6.4, “Creating the XML Schema Definition used above” on page 36.
2. Create a new XML XSL(T) file, and open the XSLT Programming Editor.

a. In the Navigator View, under our Project (MyXMLProject), and then under our Package (sub-folder, named MyXSLTFiles), Right-Click to produce a context sensitive menu.

b. From the menu that is produced, select: New => Other. This action will produce the New (object) wizard.

c. In the New (object) wizard, navigate the visual control and select XML => XSL, and Click Next. This action will produce the New XSL File dialog box.

d. In the New XSL File dialog box, enter the value "TransformCustomer.xsl" in the "File name" text entry field. click Finish. This action will place you in the XSL Programming Editor.

3. Complete programming your XSLT file.

a. The Rational Data Architect and Rational Application Developer XSLT Program Wizard generated a beginning amount of XSLT program code for you. Similar to comments we made earlier in this document, we are still in development and testing of this XSLT Program, and do not need some of the Namespace related code (Tags) that the XSLT Program Wizard gave us by default.

   The net result is, edit line-002 of the default generated XSLT Program code to remove some of these Tags.

   Figure-57 displays our starting point for creating this new XSLT Program (TransformCustomer.xsl). To start off, make your XSLT Program look like Figure-57.

b. To perform iterative testing during the creation of our XSLT Program, use the Rational Data Architect / Rational Application Developer Navigator View.

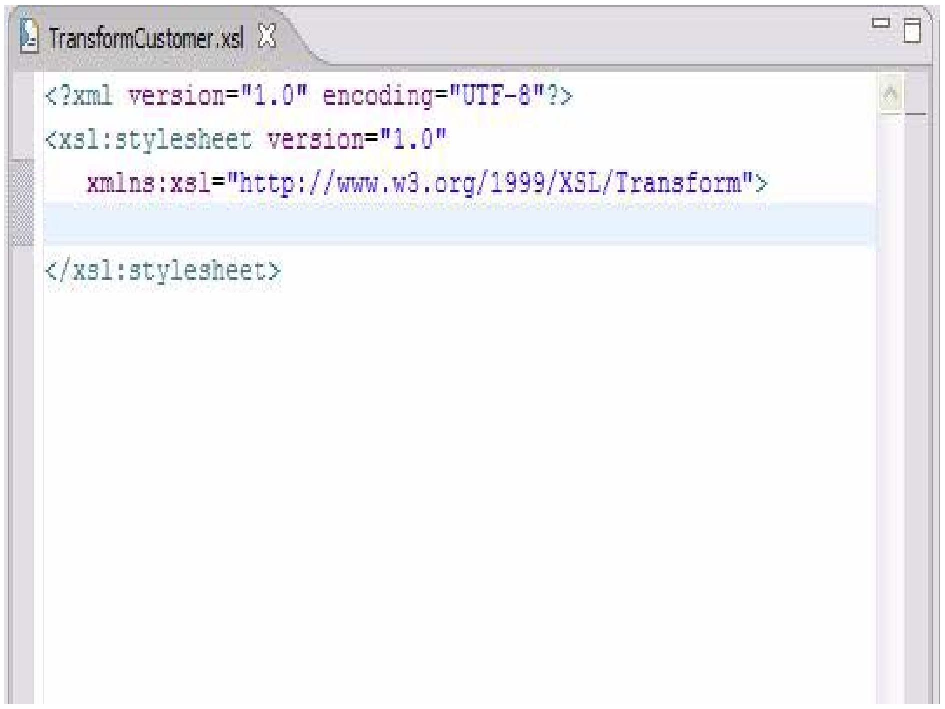
   Add the (Customers.xml) file to our Project. Using Windows Explorer, locate and then Copy this file. In Rational Data Architect / Rational Applicaiton Developer, add a new Package named (MyXMLFiles). Example as shown in Figure-58. After that Package is created, add (Customers.xml) via a Right-Click of (MyXMLFiles), and then Paste inside the Navigator View.

   As is standard in windowed programming environments, use a Control-Click to highlight multiple concurrent choices in the Navigator View. Control-Click (TransformCustomer.xsl) and then Control-Click (Customers.xml). This action will highlight both files concurrently.

   Right-Click (Customers.xml) to produce a context sensitive menu, and select; Run As => XSL Transformation. Example as shown in Figure-58.

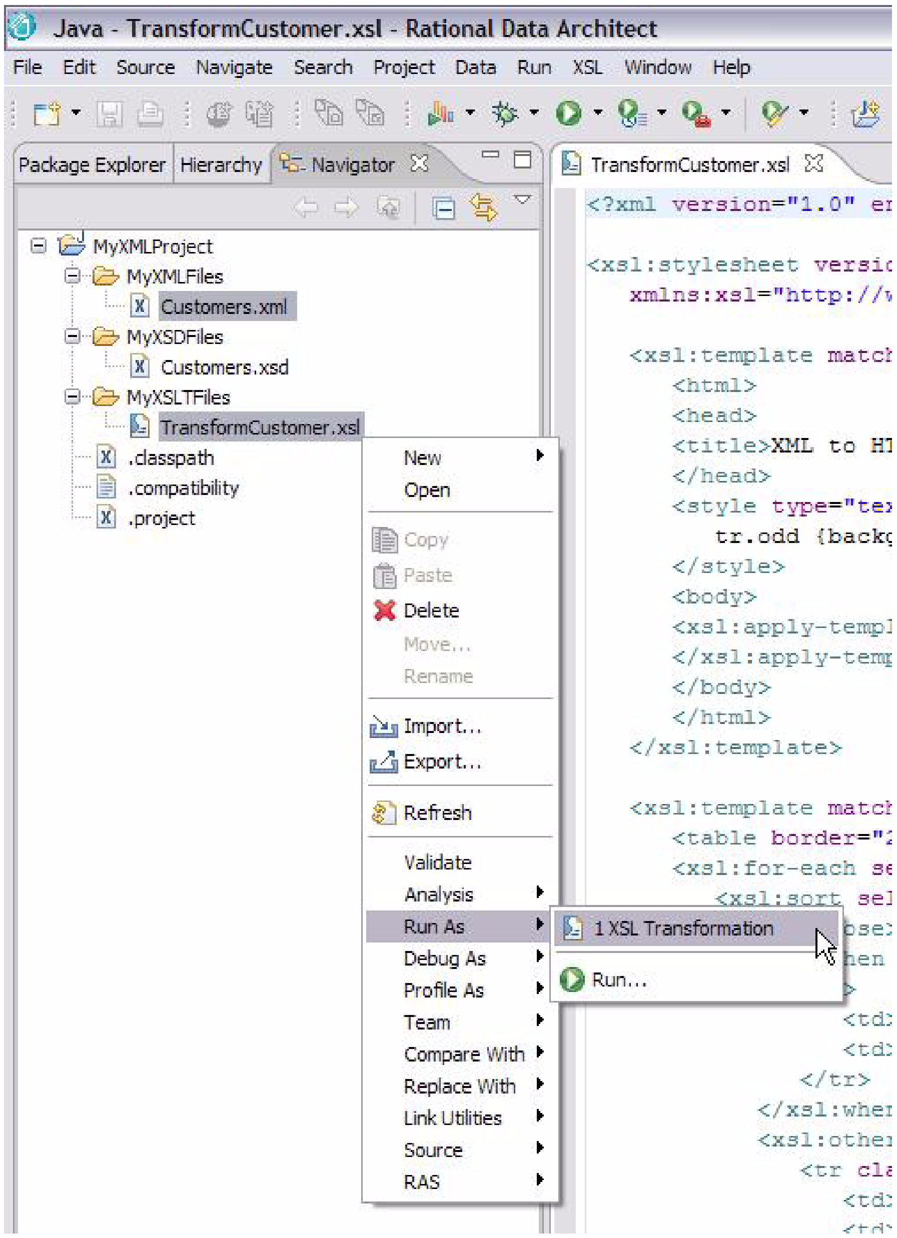
   The result of this operation will be a newly created file inside our Project with an XML file extension; XML Transformations create XML by default. The Console View will tell you the location and name of the output file. Our output file name was (_Customers_transform.xml) and was placed in the (MyXMLFiles) Package.

Because our formatted output is actually HTML, it might be best to then use a Web Browser to view our results file. Using the Navigator View, perform a Copy and then Rename of the outputted XML file and call this file (Something.html). A Double-Click on (something.html) will open this file in a Web browser, whichever one is currently configured for use by Rational Data Architect / Rational Application Developer.



```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

</xsl:stylesheet>
```

Figure-57, Starting Point to Create new XSLT Program.



```
<?xml version="1.0" en

<xsl:stylesheet versio
   xmlns:xsl="http://w

   <xsl:template match
      <html>
      <head>
      <title>XML to HT
      </head>
      <style type="tex
         tr.odd {backg
      </style>
      <body>
      <xsl:apply-templ
      </xsl:apply-temp
      </body>
      </html>
   </xsl:template>

   <xsl:template match
      <table border="1
      <xsl:for-each se
         <xsl:sort sel
                       ose
                       hen
                       >
                       <td
                       <td
                    </tr>
                 </xsl:when
                 <xsl:other
                    <tr cla
                       <td
                       <td
```

Figure-58, Iterative Testing of New XSLT Program.

c. The Rational Data Architect / Rational Application Developer XSLT Program Editor has many methods to assist you in creating an XSLT Program; additional Views (Outline View, Properties View, and more), and it has Code Snippets that you can Hot-Key and Import (those are programmable, you can add to and edit them), Content-Assist and more.

   At this point, we are only going to show you how to use Content-Assist. To learn the other Wizards, etcetera, inside Rational Data Architect / Rational Application Developer, we will list additional resources at the end of this document.

   All we have yet to accomplish is to create our (TransformCustomer.xsl) XSLT Program file. To do this, we are going to use our earlier copy of (TransformCustomer.xsl) as a guide, and we are going to use Content-Assist.

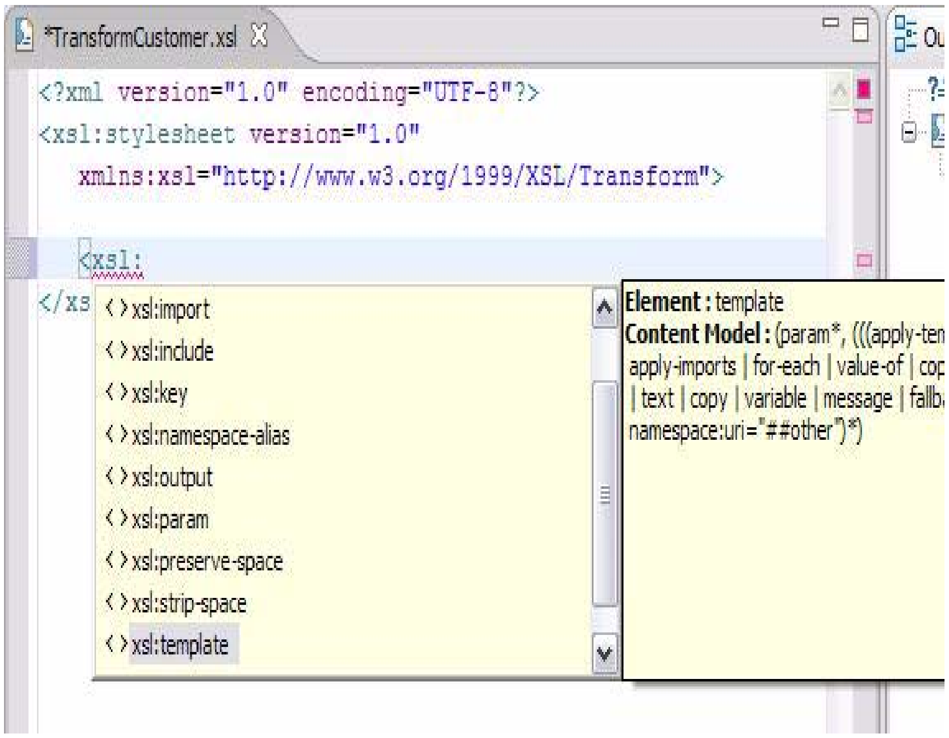
   See Figure-59. In Figure-59, we began typing line-003 from the (TransformCustomer.xsl), and then hit Control-SpaceBar.

   (Content-Assist can be configured to pop up automatically or only on command. You may experience automatic pop ups at first.)

   In Figure-59 we typed “<xsl:”, and then Content-Assist shows us the available and legal completions to this line. Content-Assist is content aware; in other words, it was illegal in the context of this XSLT file to put “<head” on line--03, and the XSLT Program Editor knew that.

   Practice using Content-Assist as you complete creation of the (TransformCustomer.xsl) file. Use our earlier copy of (TransformCustomer.xsl) as a guide.

   When you finish creating (TransformCustomer.xsl), test it by re-executing the steps we performed in Section 6.5, “DataStage/QualityStage EE Job, XML Transformer” on page 57.



```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:
</xs
```

```
<> xsl:import
<> xsl:include
<> xsl:key
<> xsl:namespace-alias
<> xsl:output
<> xsl:param
<> xsl:preserve-space
<> xsl:strip-space
<> xsl:template
```

Element : template
Content Model : (param*, (((apply-ten
apply-imports | for-each | value-of | cop
| text | copy | variable | message | fallb
namespace:uri="##other")*)

Figure-59, Using Content-Assist to Create new XSLT Program.

[End] Complete the following steps:

In this section of this document we performed the following:

We used Rational Data Architect / Rational Application Developer to create, mostly through Content-Assist, an XML Style Language - Transformation (XSLT) Program.

6.7 Summation

In this document, we reviewed or created:

– We created three DataStage/QualityStage XML related Parallel Jobs; one each for, XML Output, XML Input, and XML Transformer.

We these Jobs, we moved data to and from relational and XML, and used an XML Transform to output HTML code from an input data stream.

– We used Rational Data Architect / Rational Application Developer to create an XML Schema Definition (XSD) file, and an XML Style Language - Transformation (XSLT) file (Program).

Additional Resources:

– The IBM Information Server, DataStage/QualityStage component XML Pack product documentation is installed as part of your IBM Information Server product. The file name for the 8.01 XML Pack is, (i46dexml.pdf).

– There are several IBM Redbooks and Red Pieces on the XML related topics; visit http://www.Redbooks.IBM.com.

– Commercial resources and books-

- All of the Peachpit Press Visual QuickStart books are good; See ISBN-0-201-71098-6, “XML for the World Wide Web”. This book was published in 2001, but still offers a really good introductory overview to XML.
- Chapter-8 of ISBN-978-0-470-11487-2, “Beginning XML: 4th Edition”, was good.
- ISBN-1-931182-22-1, “An Introduction to Rational Application Developer”, is a very good introductory book to Rational Application Developer. (Rational Application Developer and Rational Data Architect share many components.)

Thank you to the persons who helped this month:

Ernie Ostic, Davor Gornic, Ray Daignault, and Allen Spayth.

Legal statements.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating trademarks that were owned by IBM at the time this information was published. A complete and current list of IBM trademarks is available on the Web at