APPENDIX D
XSLT

XSLT is a programming language defined by the World Wide Web Consortium, W3C (http://www.w3.org/TR/xslt), that provides the mechanism to transform a given XML document to another XML document. In this appendix we will explain and provide examples of using XSLT to transform XML documents into HTML, since this is one of the most common uses of XSLT. The language contains many more functions, elements, and mechanisms that are used for transformation than will be covered here due to their complexity. The reader is encouraged to learn more about XSLT from one of many good XSLT references.

In addition to XSLT-specific elements that support the creation of the result document (transformation), the XSLT language implements XPath, which is another language defined by W3C (http://www.w3.org/TR/xpath). XPath defines navigation syntax over an XML document coupled with a very useful set of build-in functions. In this appendix we will not make a distinction between elements, expressions, and functions based on the language where they are defined. Instead, we will treat all syntax as XSLT syntax.

We assume that you gained some knowledge of XSLT while working through Chapter 9 and we will not go through the XSLT elements that were explained in that chapter.

Path and Patterns Expressions

At any giving time while a transformation is happening, the XSLT processor is pointing to a specific node in the source XML document. Retrieving the node value, or its sibling, child, parent or any other node in the document, requires a navigation expression. XSLT uses XPath and patterns to support navigation, selection, and matching nodes in the source XML document.

We will start with short example to illustrate how the processor reads an XPath expression and selects the applicable nodes from the XML document. For this example, and the following examples throughout this appendix, we will use an XML document that contains the inventory data show in Figure D.1. We will also assume that the current node is the "ProductGroup" element outlined in red.

Start Page | Form1.vb [Design] | bin\xslInventory.xsl | Form1.vb | bin\Inventory.xml*

```xml
<?xml version="1.0" ?>
<Root>
   <Warehouse City="Seattle">
      <ProductGroup name="Dairy Products">
         <Product name="Milk" unit="Quart" Quantity="25" Price="1.05" Amount="26.25" />
         <Product name="Milk" unit="Half Gallon" Quantity="18" Price="1.89" Amount="34.02" />
         <Product name="Butter" unit="LB" Quantity="42" Price="3.62" Amount="152.04" />
         <Product name="Cottage Cheese" unit="LB" Quantity="20" Price="2.00" Amount="40" />
      </ProductGroup>
      <ProductGroup name="Vegetables">
         <Product name="Potato" unit="LB" Quantity="220" Price="0.22" Amount="48.40" />
         <Product name="Parsley" unit="bunch" Quantity="34" Price="0.79" Amount="26.86" />
         <Product name="Onion" unit="LB" Quantity="78" Price="0.29" Amount="22.62" />
      </ProductGroup>
   </Warehouse>
   <Warehouse City="Portland">
      <ProductGroup name="Dairy Products">
         <Product name="Cream" unit="pint" Quantity="21" Price="2.29" Amount="48.09" />
         <Product name="Yogurt" unit="Quart" Quantity="30" Price="2.99" Amount="89.70" />
         <Product name="Cottage Cheese" unit="LB" Quantity="42" Price="1.69" Amount="70.98" />
      </ProductGroup>
      <ProductGroup name="Vegetables">
         <Product name="Potato" unit="LB" Quantity="180" Price="0.19" Amount="34.20" />
         <Product name="Onion" unit="LB" Quantity="105" Price="0.33" Amount="34.65" />
      </ProductGroup>
   </Warehouse>
</Root>
```

Figure D.1 XML Inventory document

Before we go any further, we need to recall that the document in Figure D.1 represents a “tree” structure. This tree is shown in Figure D.2. In addition, each node includes one or more attributes. Attributes allow information to be stored at a node based on an attribute name. For example, the Warehouse node has an attribute named “City” that stores the name of the city where the warehouse is located. Each Product node includes five attributes (name, unit, Quantity, Price, and Amount). These attributes store information about each product.

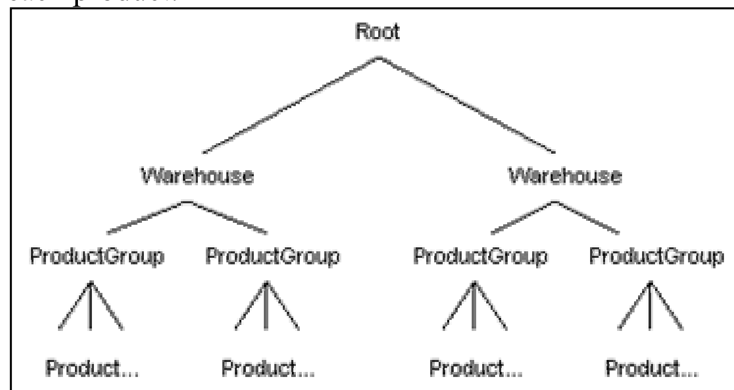


Figure D.2 Tree representing XML document from Figure D.1

The “current” node, as identified in red in Figure D.1, can be drawn as the tree segment in Figure D.3.

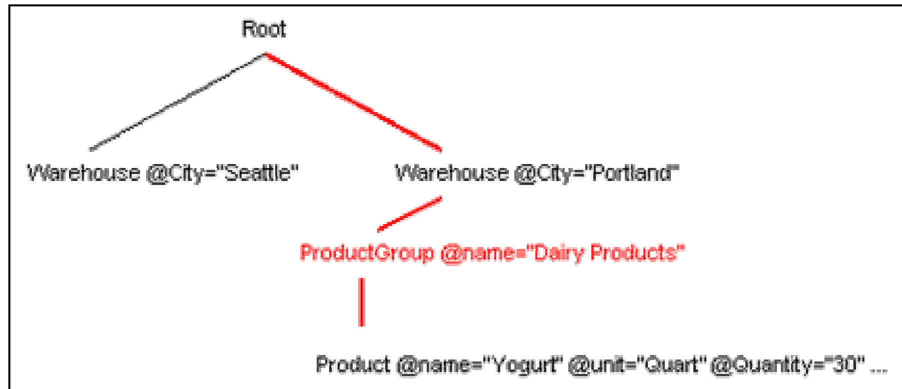
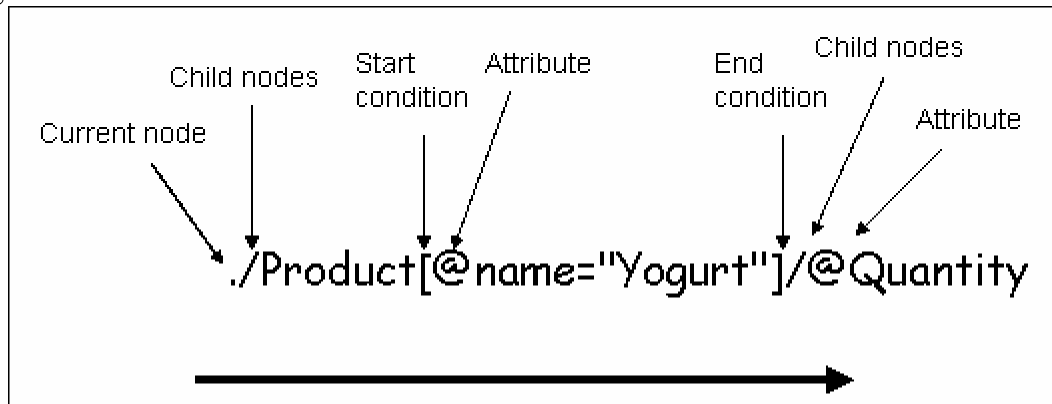


Figure D.3 Tree showing the current node (red)

The following XPath expression will select the "Quantity" attribute (whose value is 30) and is located in the "Product" node that has a name attribute with the value "Yogurt".



XPath expressions are read from left to right; the dot sign signifies the current node, single forward slash signifies the child nodes of the current node. Thus, we are currently pointing to the "Product" elements that are the children of the current node. Brackets signify a condition, the "@" sign signifies that we are looking for attribute named "name" with a value equal to "Yogurt". Since only the second "Product" element satisfies this condition, we select this element. Again the forward slash signifies child nodes (attributes are considered to be child nodes of an element). We thus select the "Quantity" attribute of the Product node whose name="Yogurt".

Table D.1 contains some of XPath expressions in their abbreviated form and patterns in XSLT. These expressions are either relative to the current node or select a node or a node set from the document regardless of the current node.

Table D.1 XSLT Path expressions

Path	**Semantics**	**Example**	**Result**
"@"	Signifies an attribute node	@name	Dairy Products
"."	Selects the current node	"."	<ProductGroup name="Dairy Products">
".."	Selects the parent of the current node	../@City	Portland
"/"	By itself selects the root of the document, also used to provide a step separator for relative path. Child is the default axis. Attributes considered	./Product/@Quantity	21

	as child nodes of an element.		
"//"	Selects all nodes that are descendant-or-self node of the current node. If not preceded with other expression will select all the matching nodes in the document.	//Product	<Product name="**Milk**" unit="Quart" Quantity="25" Price="1.05" Amount="26.25"></Product> <Product name="**Milk**" unit="Half Gallon" Quantity="18" Price="1.89" Amount="34.02"></Product> <Product name="**Butter**" unit="LB" Quantity="42" Price="3.62" Amount="152.04"></Product> <Product name="**Cottage Cheese**" unit="LB" Quantity="20" Price="2.00" Amount="40"></Product> <Product name="**Potato**" unit="LB" Quantity="220" Price="0.22" Amount="48.40"></Product> <Product name="**Parsley**" unit="bunch" Quantity="34" Price="0.79" Amount="26.86"></Product> <Product name="**Onion**" unit="LB" Quantity="78" Price="0.29" Amount="22.62"></Product> <Product name="**Cream**" unit="pint" Quantity="21" Price="2.29" Amount="48.09"></Product> <Product name="**Yogurt**" unit="Quart" Quantity="30" Price="2.99" Amount="89.70"></Product> <Product name="**Cottage Cheese**" unit="LB" Quantity="42" Price="1.69" Amount="70.98"></Product> <Product name="**Potato**" unit="LB" Quantity="180" Price="0.19" Amount="34.20"></Product> <Product name="**Onion**" unit="LB" Quantity="105" Price="0.33" Amount="34.65"></Product>
"[]"	Predicate expression, similar to a "where" clause in SQL. Filters the nodes where the condition expression is true. The following example should be read "Return all the Product nodes where the attribute 'name' is equal to 'Cottage Cheese'".	//Product[@name='Cottage Cheese']	<Product name="**Cottage Cheese**" unit="LB" Quantity="20" Price="2.00" Amount="40"></Product> <Product name="**Cottage Cheese**" unit="LB" Quantity="42" Price="1.69" Amount="70.98"></Product>

Functions

XSLT has a core library of functions that are divided into five groups: Nodeset, String, Boolean, Numeric and Other. These functions are similar to functions that are available with VB.NET. To call a function, we use the function name follow by a set of parenthesis that enclose the arguments (if there are any). Table D.2 presents some of the functions and provides an example based on the XML document in Figure D.1. Recall that the current node in Figure D.1 is marked in red.

Table D.2 XSLT Functions

Function	Semantics	Example	Result
Number **position**()	Returns the position of the current node in the node set. First node position ()= 1.	position()	3
Number **count**(node-set)	Returns the number of nodes in the argument node-set.	count(//Warehouse)	2
Number **last**()	Returns the number (position) of the last node in the current	Last()	4

	node's node-set. Since there are 4 <ProductGroup> nodes the result is 4.		
String **concat**(string, string, string*)	Returns the concatenation of its arguments. The third or more strings are optional	concat('The product group is ',./@name,' In ',../@City)	The product group is Dairy Products In Portland
String **name**()	Returns the name of the current node.	name()	ProductGroup
Boolean **contains**(string,string)	Returns true if the first argument string contains the second argument string, and otherwise returns false.	contains(./@name,'airy')	true
String **substring-before**(string, string)	Returns the substring of the first argument string that precedes the first occurrence of the second argument string in the first argument string.	substring-before(./@name,' Prod')	Dairy
String **substring-after**(string, string)	Returns the substring of the first argument string that follows the first occurrence of the second argument string in the first argument string.	substring-after(./@name,' Prod')	ucts
String **substring** (string, number, number?)	Returns the substring of the first argument starting at the position specified in the second argument with length specified in the third argument. If the third argument is missing it continues to the end of the string.	substring(./@name,7)	Products
Number **string-length**(string?)	Returns the number of characters in the string, if the argument is omitted, it defaults to the context node converted to a string.	string-length(./@name)	14
Boolean **not**(boolean)	Return true is the argument is evaluated to false and false otherwise. You can use '!=' to create the "not equal" expression.	not(last()=5) last() != 5	true
Number **round**(number)	Returns the number that is closest to the	round(./Product[position()=1]/@Price)	2

	argument and that is an integer.		
Number **sum**(node-set)	Return the result of summing each node in the argument node-set.	sum(./Product/@Quantity)	93
String **format-number**(number, string)	Converts its first argument to a string using the format pattern string specified by the second argument.	format-number(1625.726,'$#,###.00')	$1,625.73

Numeric Operators

XSLT expressions are evaluated with respect to their context to four basic data types: Node-sets, Boolean, Number and String. Like Visual Basic .NET, addition, subtraction, multiplication, division and mod operators are available to perform calculations during the transformation. If the expression is applied to node values that can not be converted to a valid number, the result will be NaN (not a number). For example, the expression "./@name * 2" will output NaN in the output document since "Dairy Products" (./@name) is not numeric. Since the forward slash "/" is heavily used for path expressions, the division operator is "div", for example, the result of "4 div 2" is 2.

Conditional Statements

XSLT has two elements that provide the ability to perform conditional processing based on the Boolean expressions. The first element is xsl:if that provides a simple if-then functionality. However, unlike Visual Basic .NETs "If…Then" statement, it does not support the "ElseIf" or "Else" clauses. The following example will output "This is the last element in the node-set" when current node is last in the node-set:

```
<xsl:if test="position()=last()">
    <b>This is the last element in the node-set</b>
</xsl:if>
```

The second element that performs conditional processing is xsl:choose. This supports the selection of one choice from several possibilities. It is similar to the Visual Basic .NET Select Case statement. It consists of a sequence of xsl:when elements, followed by an optional xsl:otherwise element. Each xsl:when element has a single attribute, which specifies an expression When an xsl:choose element is processed, each of the xsl:when elements is tested in turn, by evaluating the expression to Boolean. The content of the first, and only the first, xsl:when element whose test is true is executed. If none of the xsl:when elements are true, the content of the xsl:otherwise element is executed. If no xsl:when element is true, and no xsl:otherwise element is present, nothing happens. The following example will apply the color blue as the paragraph background color for each node whose position is even, and the color red as the background color for every other node.

```
<xsl:choose>
    <xsl:when test="(position() mod 2)=0">
        <P style="background-color:blue"></P>
    </xsl:when>
    <xsl:otherwise>
        <P style="background-color:red"></P>
    </xsl:otherwise>
</xsl:choose>
```

Variables and Params

Similar to Visual Basic .NET, you can create variables that are bound to an expression such as string, node or node-set. There are two elements that represent variables in XSLT**:** xsl:variable and xsl:param. A top-level variable-binding element declares a global variable that is visible everywhere within the XSLT document. In addition, template variables are allowed anywhere within a template (we will talk about templates soon). In this case, the variable is visible only within that template.

The xsl:variable is mainly used to compute values and to select and create nodes for use in other places in the transformation. Global variables are particularly useful when you would like to compute a value from the source XML document and use it more than once during a transformation. Global variables are also useful for declaring constants values. Template variables are useful when you need an intermediate calculation while processing a node-set. Unlike Visual Basic .NET, you cannot use a variable in its own expression. Thus, the Visual Basic .NET statement "x=x+1" is not valid in XSLT.

The xsl:param element is mainly used to pass values from another application to the XSLT stylesheet, and from template to template promoting reusability and abstraction. For example, passing user input to a stylesheet using a global param allows creating different reports from the same XML and XSLT documents.

Referring to a variable or param to retrieve its value is done by prefixing the variable or param name with a $. The following demonstrates declaring and retrieving values from variables.

Declare:

```
<xsl:variable name="x">2</xsl:variable>
<xsl:param name="y">3</xsl:param>
```

Retrieve:

```
<xsl:value-of select="$x * $y"/>
```

Result: 6

Templates (Procedural and Recursive)

XSLT provides two mechanisms for processing the source document to get the result tree. The first is procedural processing, which allows creating a transformation using control structures similar to Visual Basic .NET. To allow procedural-like processing, XSLT provides named templates, which are similar to calling a function or sub procedure in Visual Basic .NET. XSLT also has a for-each element that supports looping through a node-set. Named templates are invoked by the `<xsl:call-template>` element that can pass params to the called template. The `<xsl:for-each>` element supports the processing of a node-set directly within the template. The following example demonstrates processing of all the "Products" nodes that are child nodes of the

"ProductGroup" nodes. It uses a "for-each" element and a named template whose name is ProcessProducts (not shown) to control the processing.

```
<xsl:for-each select="ProductGroup">
     Do something here (optional)
     <xsl:call-template name="ProcessProducts">
          <xsl:with-param name="CurrentProduct" select="./Product"/>
     </xsl:call-template>
     Do something here (optional)
</xsl:for-each>
```

The second mechanism for processing the source document is recursive processing where a list of nodes is selected from the source tree by using an <xsl:apply-templates select=""> instruction. When the processor encounters an <xsl:apply-templates select="">, it searches for the best matching template to process the nodes. The template that best matches the node list may contain additional node list selections for processing and is instantiated with a current node and source node list. The process of matching the node list by template, applying the template rules and selecting new node list will continue recursively until no new source nodes are selected for processing. The "select" attribute in the apply-templates element is optional since the selecting of the child nodes is the default selection of nodes by the processor.

The following template will process all the "Products" nodes that are child nodes of the ProductGroup nodes by finding a template that matches "Product". It will produce the same result as the previous example. The select attribute, although redundant in this case, is added for clarity.

```
<xsl:template match="ProductGroup">
     Do something here (optional)
          <xsl:apply-templates select="Product"/>
     Do something here (optional)
</xsl:template>
```

Transforming an XML document recursively using xsl:apply-templates elements can be a complex and sometimes difficult task, however many XSLT programmers consider it as a more natural way to transform and process an XML document.

Finally, you can combine the procedural and recursive approaches to create the desired transformation. For example you can replace the call to a named template in the first example with xsl:apply-templates instruction in the second example to get the same results.

Creating HTML inventory report from XML document using XSLT

The following example will combine many elements and expressions of the XSLT language to create HTML report from the Inventory XML document in Figure D.1. We will show how to create the same report using two stylesheets. The first uses a named template and for-each elements to perform procedural processing similar to Visual Basic .NET programs. The other uses an apply-templates element to recursively process the XML tree to create the same report. Figure D.4 shows the results of the transformed XML as a rendered HTML document. As you can see, the report is fairly complex in terms of appearance and style. The XSLT document is responsible for adding the appropriate HTML tags to achieve this appearance.

XSLT Inventory Report Example

Inventory Reprot

Inventory of Seattle warehouse

Product Group	Product	Unit Size	Quantity	Price	Amount
Dairy Products	**Milk**	**Quart**	**25**	**$1.05**	**$26.25**
	Milk	Half Gallon	18	$1.89	$34.02
	Butter	**LB**	**42**	**$3.62**	**$152.04**
	Cottage Cheese	LB	20	$2.00	$40.00
Vegetables	**Potato**	**LB**	**220**	**$0.22**	**$48.40**
	Parsley	bunch	34	$0.79	$26.86
	Onion	**LB**	**78**	**$0.29**	**$22.62**

Total inventory of Seattle is $350 which is 56% of total report

Inventory of Portland warehouse

Product Group	Product	Unit Size	Quantity	Price	Amount
Dairy Products	**Cream**	**pint**	**21**	**$2.29**	**$48.09**
	Yogurt	Quart	30	$2.99	$89.70
	Cottage Cheese	**LB**	**42**	**$1.69**	**$70.98**
Vegetables	**Potato**	**LB**	**180**	**$0.19**	**$34.20**
	Onion	LB	105	$0.33	$34.65

Total inventory of Portland is $278 which is 44% of total report

Total inventory in this report: $628

Show Report Using Procedural StyleSheet   Show Report Using Recursive StyleSheet

Figure D.4 Inventory report as it appears in the windows application

XML declaration and global variables. We start by creating a new XML document and adding the XSLT namespace. Both XSLT stylesheets use the same set of global variables to define the report formatting. Defining the formatting definitions as global variables allows us to change the report appearance by changing a value in only one place. Figure D.5 shows the XML document declaration and the definition of the first of 7 global variables (`ColHeader`) included in the actual document (the other 6 global variables define additional styles and are not shown here). Later we use these variables to define the "style" attribute of the table cell "TD", using syntax like the following:

```
<TD style="{$Variable Name}">
```

The curly braces signify that during transformation, the value of the attribute needs to be evaluated to the result of the XSLT expression.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<!-- Declare global variables -->
<xsl:variable name="ColHeader">
{
   BORDER-BOTTOM: 0.08em solid #3f0600;
   BORDER-TOP: 0.08em solid #3f0600;
   PADDING-TOP: 0.15em;
   PADDING-BOTTOM: 0.15em;
   PADDING-LEFT: 0;
   FONT-SIZE: 0.8em;
   COLOR: #ffffff;
   BACKGROUND-COLOR: #00aadd;
   FONT-WEIGHT: bold;
}
</xsl:variable>
```

Figure D.5 XML document declaration and global variables

We then proceed and create the starting template that processes the root element of the document signified by the "/" symbol. Although it is not mandatory to have a template that process the root element, it is a highly recommended practice since it defines a clear starting point for processing the source document. It also allows us to easily define a root element in the result document (the "HTML" tag in our example). We will use an HTML table with its tags (Table, TR, TD) to display the report. Figure D.6 shows the start of the first template.

```
<xsl:template match="/">
<HTML>
  <Body>
    <Div style="{$ReportHeader}">Inventory Report</Div>
    <Div>
    <Table style="{$TablFormat}" cellspacing="0" cellpadding="0" width="100%">
```

Figure D.6 Start of the template

Approach #1: Using for-each and call-template to process the source document. In the first (procedural) stylesheet, we are using 3 nested for-each loops to iterate through each level of the tree to create the report table. The outer loop iterates through the "Warehouse" nodes and creates a table row for each warehouse. An inner loop iterates through each product group. Within this inner loop is a third loop that iterates through each product for the product group. If you look back at Figure D.4, you can see the impact of these three loops. After the inner loops finish, but before the outer loop repeats, a new row is created to show the total inventory for that warehouse and the percentage of total inventory represented by the warehouse. The "Sum" function and the “div" operator (for division) to are used to accomplish this task. Finally we format the result using number-format function:

```
<xsl:value-of select="concat('Total inventory of ', ./@City,' _is ',
format-number(sum(.//@Amount),'$#,###'),' which is ',
```

```
format-number(sum(.//@Amount) div sum(//@Amount),'#0%'), ' of total report')"/>
```

The inner loop that creates a row for each product group uses the rowSpan attribute of the TD element to merge the first cell in the ProductGroup row and the product rows that are created in the "product" inner loop. To calculate the number of rows to merge, we use the following statement:

```
rowspan="{count(./Product) + 1}"
```

This statement counts the child nodes of each ProductGroup and adds an additional row to account for the heading.

Within the product group loop, where the current node is a ProductGroup node, we call a named template and pass the current node using the Current() function as a param:

```
<xsl:call-template name="displayProductGroup">
    <xsl:with-param name="prodGroup" select="current()"/>
</xsl:call-template>
```

Figure D.7 shows the XSLT code that includes the outer "Warehouse" loop as well as the embedded "ProductGroup" loop.

```
    <xsl:for-each select="//Warehouse">
      <TR>
          <TD colspan="6" style="{$WHouse}">
            <xsl:value-of select="concat('Inventory of ', ./@City,' warehouse')"/>
          </TD>
      </TR>
      <TR>
          <TD style="{$ColHeader}">Product Group</TD><TD style="{$ColHeader}">Product</TD>
          <TD style="{$ColHeader}">Unit Size</TD><TD style="{$ColHeader}">Quantity</TD>
          <TD style="{$ColHeader}">Price</TD><TD style="{$ColHeader}">Amount</TD>
      </TR>
      <xsl:for-each select="./ProductGroup">
          <TR>
            <TD rowspan="{count(./Product) + 1}" style="{$ProdGroup}">
              <xsl:value-of select="./@name"/>
            </TD>
          </TR>
        <xsl:call-template name="displayProductGroup">
            <xsl:with-param name="prodGroup" select="current()"/>
        </xsl:call-template>
        </xsl:for-each>
        <TR>
          <TD colspan="6" style="{$CellValue2}">
            <xsl:value-of select="concat('Total inventory of ', ./@City,'  is ',
                          format-number(sum(.//@Amount),'$#,###'),' which is ',
                          format-number(sum(.//@Amount) div sum(//@Amount),'#0%')
                          , ' of total report')"/>
          </TD>
        </TR>
    </xsl:for-each>
    <TR>
        <TD colspan="6" style="{$WHouse}">
        <xsl:value-of select="concat('Total inventory in this report: '
                        , format-number(sum(//@Amount),'$#,###'))"/>
        </TD>
    </TR>
    </Table>
    </Div>
   </Body>
</HTML>
</xsl:template>
```

Figure D.7 Embedding for-each loops and creating the table

Within the inner “ProductGroup” loop, the “displayProductGroup” template is called. This template, shown in Figure D.8, includes the loop that iterates through each product.

```
<xsl:template name="displayProductGroup">
  <xsl:param name="prodGroup"/>
  <xsl:for-each select="$prodGroup/Product">
    <TR>
      <xsl:choose>
       <xsl:when test="(position() mod 2) = 0">
       <TD style="{$CellValue}"><xsl:value-of select="./@name"/></TD>
       <TD style="{$CellValue}"><xsl:value-of select="./@unit"/></TD>
       <TD style="{$CellValue}"><xsl:value-of select="./@Quantity"/></TD>
       <TD style="{$CellValue}"><xsl:value-of
                select="format-number(./@Price,'$#,##0.00')"/></TD>
       <TD style="{$CellValue}"><xsl:value-of
                select="format-number(./@Price * ./@Quantity,'$#,###.00')"/></TD>
      </xsl:when>
      <xsl:otherwise>
           <TD style="{$CellValue2}"><xsl:value-of select="./@name"/></TD>
       <TD style="{$CellValue2}"><xsl:value-of select="./@unit"/></TD>
       <TD style="{$CellValue2}"><xsl:value-of select="./@Quantity"/></TD>
       <TD style="{$CellValue2}"><xsl:value-of
              select="format-number(./@Price,'$#,##0.00')"/></TD>
       <TD style="{$CellValue2}"><xsl:value-of
              select="format-number(./@Price * ./@Quantity,'$#,###.00')"/></TD>
      </xsl:otherwise>
      </xsl:choose>
    </TR>
  </xsl:for-each>
</xsl:template>
```

Figure D.8 The “displayProductGroup” template

Approach #2: Using apply-templates to process the source document. In the second stylesheet, we use the `xsl:apply-templates` element, which allows us to recursively process the source document. We also replace the named template with three new templates that match the "Warehouse", "ProductGroup" and "Product" elements. The XSLT processor starts processing the source document from the root element and looks for a template that matches this element. If it doesn't find one, it will continue to look for templates that match the child nodes of this element. This process will continue until a matching template is found or until the source document is completely processed and no matching templates for its elements are found. Since our first template matches the root node (`<xsl:template match="/">`), the source document transformation starts with the template shown in Figure D.9. After inserting the "HTML", "Body" and “Table” tags into the result document, the processor encounters the `<xsl:apply-templates/>` instruction that starts the processing of the child nodes of the current node.

```
<xsl:template match="/">
<HTML>
  <Body>
    <Div style="{$ReportHeader}">Inventory Reprot</Div>
    <Div>
    <Table style="{$TablFormat}" cellspacing="0" cellpadding="0" width="100%">
    <xsl:apply-templates/>
    <TR>
        <TD colspan="6" style="{$WHouse}">
        <xsl:value-of select="concat('Total inventory in this report: '
                          , format-number(sum(//@Amount),'$#,###'))"/>
        </TD>
    </TR>
    </Table>
    </Div>
   </Body>
</HTML>
</xsl:template>
```

Figure D.9 Template that matches the root

The child nodes of the "Root" element are the "Warehouse" elements so the processor looks for a template that matches the "Warehouse" element. That template (see Figure D.10) outputs two rows to the result document before another `<xsl:apply-templates/>` instruction, which causes the processor to look for a matching template for the current element `<Warehouse City="Seattle">` child nodes ("ProductGroup" nodes).

```
<xsl:template match="Warehouse">
   <TR>
      <TD colspan="6" style="{$WHouse}">
         <xsl:value-of select="concat('Inventory of ', ./@City,' warehouse')"/>
      </TD>
  </TR>
  <TR>
      <TD style="{$ColHeader}">Product Group</TD><TD style="{$ColHeader}">Product</TD>
      <TD style="{$ColHeader}">Unit Size</TD><TD style="{$ColHeader}">Quantity</TD>
      <TD style="{$ColHeader}">Price</TD><TD style="{$ColHeader}">Amount</TD>
  </TR>
   <xsl:apply-templates/>
   <TR>
      <TD colspan="6" style="{$CellValue2}">
         <xsl:value-of select="concat('Total inventory of ', ./@City,'  is ',
                        format-number(sum(.//@Amount),'$#,###'),' which is ',
                        format-number(sum(.//@Amount) div sum(//@Amount),'#0%')
                        , ' of total report')"/>
      </TD>
   </TR>
</xsl:template>
```

Figure D.10 Template that processes the Warehouse nodes

The template that matches the ProductGroup elements is shown in Figure D.11.

```
<xsl:template match="ProductGroup">
  <TR>
    <TD rowspan="{count(./Product) + 1}" style="{$ProdGroup}">
      <xsl:value-of select="./@name"/>
    </TD>
  </TR>
  <xsl:apply-templates/>
</xsl:template>
```

Figure D.11 Template that processes the ProductGroup nodes

Processing of the child nodes of the "ProductGroup" elements i.e. the "Product" elements, will happen when the processor encounters `<xsl:apply-templates/>` in the "ProductGroup" template. Since the "Product" template, shown in Figure D.12, does not have any references to other templates, the processor will return to the "ProductGroup" template and from there to the "Warehouse" template. This process will continue recursively until there are no more nodes that match one of these templates in the source document.

```
<xsl:template match="Product">
  <TR>
    <xsl:choose>
    <xsl:when test="(position() mod 2) = 0">
     <TD style="{$CellValue}"><xsl:value-of select="./@name"/></TD>
     <TD style="{$CellValue}"><xsl:value-of select="./@unit"/></TD>
     <TD style="{$CellValue}"><xsl:value-of select="./@Quantity"/></TD>
     <TD style="{$CellValue}"><xsl:value-of
              select="format-number(./@Price,'$#,##0.00')"/></TD>
     <TD style="{$CellValue}"><xsl:value-of
              select="format-number(./@Price * ./@Quantity,'$#,###.00')"/></TD>
   </xsl:when>
   <xsl:otherwise>
       <TD style="{$CellValue2}"><xsl:value-of select="./@name"/></TD>
     <TD style="{$CellValue2}"><xsl:value-of select="./@unit"/></TD>
     <TD style="{$CellValue2}"><xsl:value-of select="./@Quantity"/></TD>
     <TD style="{$CellValue2}"><xsl:value-of
            select="format-number(./@Price,'$#,##0.00')"/></TD>
     <TD style="{$CellValue2}"><xsl:value-of
            select="format-number(./@Price * ./@Quantity,'$#,###.00')"/></TD>
   </xsl:otherwise>
   </xsl:choose>
  </TR>
</xsl:template>
```

Figure D.12 Template that processes the Product nodes

Summary

XSLT is extremely powerful language for transforming an XML document into other XML documents. Similar to Visual Basic .NET, it has a set of operators and functions as well as conditional and iteration statements. In addition, it supports recursion that provides a very natural way of processing XML tree. An XSLT developer can design stylesheets that are procedural or recursive or combine both methodologies to create the same transformation documents.

XSLT is a key technology in the XML world. While XML is a data format that is easily transferable between different computer systems and between organizations, XSLT provides the means to transform this data to useable format. In addition to transforming XML documents into HTML, it enables scenarios where two organizations can exchange XML data, and, using XSLT, transform the data into the appropriate format so that the data may be inserted into each organization's databases.