A screenshot of a computer program

Description automatically generated

Create a new XML structure with all movies sorted by year. Actors shall be sorted alphabetically.

<movies>

<movie title=”” year=”” productioncompany=”” director=””>

<actor> actor’s name</actor>

A screenshot of a computer program

Description automatically generated

<xsl:attribute name=”year”>

<xsl:value-of select=”@year”>

A screenshot of a computer program

Description automatically generated

This XSLT 1.0 solution provides a way to restructure an XML database of movies. Here's a detailed breakdown of each part of the script:

1. **XML and XSLT Declaration**: The transformation starts with an XML declaration that specifies the version and encoding used. Then, it declares an XSLT transformation using XSLT version 1.0 and the XSL namespace.
2. **Output Method**: The **<xsl:output method="xml"/>** tag sets the output method to XML, meaning the transformation will produce XML content.
3. **Root Template**: The root template matches the root node **/** and creates a new XML structure starting with the **<movies>** element. It uses a variable **$doc** to store the document loaded from a path (assumed to be '../movies.xml') containing the original movies data.
4. **Applying Templates to Movies**:
   * **<xsl:apply-templates select="$doc//Movie">** applies templates to all **<Movie>** elements in the document stored in **$doc**.
   * **<xsl:sort select="@Year" order="ascending"/>** sorts these **<Movie>** elements by their **Year** attribute in ascending order, ensuring the movies are organized by year as requested.
5. **Movie Template**: This template matches each **<Movie>** element and constructs a new **<movie>** element for each:
   * Attributes such as **title**, **year**, **productioncompany**, and **director** are set using **<xsl:attribute>** tags, pulling values from the respective attributes and elements of the original **<Movie>** elements.
   * It also applies templates to all **<Actor>** elements associated with the current movie (matched by title), sorting them alphabetically by the actor's name.
6. **Actor Template**: This template matches each **<Actor>** element and creates a new **<actor>** element, inserting the actor's name as its content.

**Key Points:**

* The transformation is set to output XML.
* Movies are sorted by year, and actors within each movie are sorted by name.
* The script assumes a certain structure for the source XML and uses XPath extensively to navigate and extract data from it.

This XSLT script is designed to transform a list of movies (along with associated directors and actors) into a structured XML format that meets specific sorting criteria, organizing movies by year and listing actors alphabetically within each movie. This meets the requirements specified in the provided assignment image.

Top of Form

Understanding when to use **<xsl:template match="...">** and **<xsl:apply-templates>** is key to effectively leveraging XSLT for XML transformations. Here’s how they work and how you can use them in building solutions:

**1. <xsl:template match="...">**

The **<xsl:template match="...">** directive is used to define a template that XSLT applies to a specific type of node or a node that matches a specific pattern within the XML document. Each template is essentially a rule for how to handle and transform a particular piece of the XML.

* **When to Use**: Use **<xsl:template match="...">** when you want to define how different types of nodes (elements, attributes, text nodes, etc.) should be processed or transformed. It's ideal for setting specific transformation rules for different parts of your XML structure.
* **Example Use**: If you have an XML with different types of elements like **<Book>**, **<Author>**, and **<Price>**, and you want to transform each of these differently, you would create separate match templates for each.

**2. <xsl:apply-templates>**

The **<xsl:apply-templates>** directive is used within a template to process child nodes (or a specified subset of nodes) of the current node. It hands off the processing of these nodes to the appropriate template defined elsewhere in the XSLT, based on matching criteria.

* **When to Use**: Use **<xsl:apply-templates>** when you want to delegate the processing of nodes to the most appropriate **<xsl:template match="...">** defined in your stylesheet. It's particularly useful for recursive XML structures or when you want to maintain a separation of concerns within your transformation logic.
* **Example Use**: In a book catalog XML, within a template for **<Book>**, you might use **<xsl:apply-templates select="Author"/>** to process the **<Author>** element using a separate template.

**Constructing a Solution with XSLT**

When constructing an XSLT solution, the thought process generally involves:

1. **Analyzing the XML Structure**: Understand the structure of your input XML to determine what elements and attributes need to be transformed and how they are nested.
2. **Defining Templates**: For each type of node that requires specific handling, define a **<xsl:template match="...">**. Think about what needs to happen to each element type or attribute. Should it be copied, modified, removed, or reordered?
3. **Applying Templates**: Use **<xsl:apply-templates>** within your templates to process child elements. This allows you to modularize your transformations, making complex XML transformations easier to manage and understand.
4. **Sorting and Conditionals**: Integrate **<xsl:sort>** within your **<xsl:apply-templates>** to control the order of elements, and use conditionals like **<xsl:if>** or **<xsl:choose>** to handle variations in content or structure.
5. **Testing and Refinement**: As with any coding task, iterative testing and refinement based on the output are crucial. XSLT can produce unexpected results due to its recursive nature and the way it processes XML nodes, so iterative refinement is often necessary.

**Example Thought Process:**

If tasked with transforming a movie database XML:

* **Step 1**: Define a root template to wrap the entire output, say in a **<movies>** element.
* **Step 2**: Define a template for each **<Movie>** that structures each movie's details and uses **<xsl:apply-templates>** to delegate the processing of nested **<Actor>** elements.
* **Step 3**: Create a specific template for **<Actor>** to format actor details.
* **Step 4**: Use **<xsl:sort>** within **<xsl:apply-templates>** for **<Movie>** and **<Actor>** to ensure the output is ordered as required by the specifications.

This approach helps in building a modular, scalable, and manageable XSLT transformation stylesheet.

Yes, when you use **<xsl:apply-templates>** in an XSLT stylesheet, it can and often does match **<xsl:template>** definitions that appear later in the stylesheet. This is a fundamental aspect of how XSLT processors work. They are designed to handle the entire stylesheet as a cohesive unit, regardless of the order in which templates are defined. Here’s a more detailed explanation:

**How <xsl:apply-templates> Works**

* **Node Selection**: **<xsl:apply-templates>** can include a **select** attribute that specifies which nodes (children, specific nodes, etc.) to process. If the **select** attribute is omitted, it defaults to selecting all child nodes of the current node.
* **Template Matching**: The XSLT processor looks at all available **<xsl:template>** definitions to find the best match for each selected node. The matching is based on the XPath expression specified in the **match** attribute of each **<xsl:template>**

<xsl:variable name=”doc” select=”document(‘../movies.xml’)”>

The XSLT 2.0 solution provided for transforming an XML movie database introduces a few significant improvements and capabilities that are not available or are handled differently in XSLT 1.0. Here's a breakdown of the key differences and enhancements in this XSLT 2.0 solution compared to the XSLT 1.0 version:

**1. Version and Namespace**

* **XSLT 2.0** uses the same namespace as XSLT 1.0 but specifies **version="2.0"** in the **<xsl:transform>** tag, enabling newer syntax and features.

**2. Output Indentation**

* **XSLT 2.0**: The **indent="yes"** attribute in the **<xsl:output>** tag. This instructs the XSLT processor to output human-readable, indented XML, making the output easier to read and debug. This feature is available in XSLT 1.0 as well, but its implementation and effectiveness can vary between processors.

**3. Use of select Attribute in <xsl:attribute>**

* **XSLT 2.0**: Allows the use of the **select** attribute directly within **<xsl:attribute>** tags, streamlining the code and enhancing readability. For example:

xml

Copy code

<xsl:attribute name="title" select="@Title"/>

This is a more concise and clear way to set attribute values compared to XSLT 1.0, where you would typically use **<xsl:value-of>** inside an **<xsl:attribute>** block.

**4. Template Matching and Sorting**

* **XPath Context**: The XSLT 2.0 solution potentially uses a more robust context handling with XPath expressions, although this particular script appears to contain an inefficiency or error in its application of templates for actors:

xml

Copy code

<xsl:apply-templates select="//Movie[@Title=$movie\_name]/Actor">

This line intends to select actors from the current movie, but the use of **//Movie** searches the entire document again, which is not only inefficient but might also not work as intended if multiple movies have the same title. A better approach would be to use a relative path or a different method to ensure the context is limited to the current **Movie**.

**5. Error Handling and Robustness**

* **XSLT 2.0** offers better error handling and more built-in functions which can be used to manipulate strings, numbers, sequences, etc., more efficiently. Though not explicitly shown in this script, XSLT 2.0’s ability to directly manipulate and construct sequences can significantly simplify many common tasks.

**6. Performance and Capabilities**

* **XSLT 2.0** generally provides better performance and more capabilities, especially for complex documents and transformations, thanks to improved XPath and XSLT processing capabilities.

**Conclusion**

The XSLT 2.0 script you provided aims to be more concise, particularly in how attributes are set in the transformed XML. However, the attempt to use advanced features appears to be slightly misapplied in the context of selecting actors, which could affect both performance and accuracy. Generally, XSLT 2.0 allows for more powerful and efficient transformations thanks to its enhanced features and functions.

A screenshot of a computer program

Description automatically generated

<xsl:template match=”/”>

<movies>.

Show all the movies grouped by production company as HTML! One heading per company followed by a table with the corresponding movies. The table shall have one row with headings and one row per movie, with the following information included: Title, Year, Director's name, Actors (name and age at that time).

<ProductionCompany name as heading>

Table with movies

Title Year Director Actors

A screenshot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

< xsl: output method=”html”/>

All movies grouped by production company as HTML.

One heading per company followed by a table with movies.

A screenshot of a movie list

Description automatically generated

<xsl:with-param> is used within an <xsl:apply-templates> or <xsl:call-template> element to pass parameters to the template being applied or called. It allows you to send additional data into a template, which can then be accessed within that template using <xsl:param>. In this script, it's used to pass the year of the movie to the actor template, enabling the calculation of each actor's age at the time of the movie.

This mechanism for passing parameters is essential for creating reusable, modular templates in XSLT that can behave differently based on input parameters, enhancing the flexibility and power of XSLT transformations.

<xsl:template match=”/”>

<xsl:variable name=”doc” select=document(“../movies.xml”)>

<xsl:sort select=”@Year” order=”ascending”>

<xsl:sort select=”text()” order=”ascending” >

directly after <xsl:apply-templates select=”$doc//ProductionCompany[not(text()=preceding::text())]>

A red and green rectangle with black text

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

This XSLT (Extensible Stylesheet Language Transformations) code is designed to transform an XML document containing information about movies into an HTML page grouped by production companies. Here's a breakdown of the key parts:

1. **XSLT Declaration and Namespaces:**
   * The code declares an XSLT stylesheet using version 2.0. It specifies the namespace for XSLT as **http://www.w3.org/1999/XSL/Transform**.
2. **Variables and Output Method:**
   * The stylesheet sets the output method to "HTML".
   * A variable named "doc" is declared to hold the entire XML document (**document(../movies.xml)**), which allows the stylesheet to access data from an external XML document.
3. **HTML Structure Setup:**
   * The HTML structure begins with **<html>**, **<head>**, and **<title>** tags to set up the document head.
   * The title "Movies by production company" is set.
4. **Grouping and Sorting Movies:**
   * Movies are grouped by their **ProductionCompany** element using the **xsl:for-each-group** instruction, selecting all **Movie** elements from the XML.
   * The **current-grouping-key()** function is used to sort the groups in ascending order based on the production company's name.
5. **Table Creation for Grouped Movies:**
   * For each group, the code generates a header (**<h2>**) that displays the production company's name.
   * A table is then created with columns for Title, Year, Director, and Actors.
   * The **xsl:apply-templates** instruction is used to apply templates to the current group of movies.
6. **Templates for Movie and Actor Data:**
   * The template matching "Movie" creates a table row for each movie with cells for the title, year, director's name, and a list of actors.
   * It uses the **xsl:apply-templates** again for each **Actor** element, passing the movie's year as a parameter (**movie\_year**) to each actor's template.
7. **Template for Actors:**
   * The template for actors receives the **movie\_year** parameter.
   * It creates a list item (**<li>**) for each actor, displaying the actor's name and calculating their age in the movie's release year by subtracting the year of birth from the movie year.

This code efficiently organizes and presents data from an XML document containing movie details, allowing for easy viewing of movies grouped by the company with details about their actors formatted in a human-readable HTML format.

Top of Form

Show all the movies grouped by director as HTML! One big table with one heading row per director with name and country together in one cell and followed by one row per movie with four columns: Title, Year, Production company, Number of actors. Make sure the table is filled with cells correctly by using the html attribute colspan. The directors and the movies shall be sorted alphabetically.

<xsl:with-param name select> inside the <xsl:apply-templates>

With name and select

Then <xsl:param name=””> inside the <xsl:template match>

2nd question

A screenshot of a computer code

Description automatically generated

3rd question

A close-up of a message

Description automatically generatedall movies grouped by director as HTML.

One big table one heading row per director with name and country

One heading row per director with name and country together in one cell followed by one row per movie.

Movie – Title Year ProductionCompany NumberofActors

Movie

Movie

Movie

A screen shot of a movie

Description automatically generated

Directors and Movies sorted alphabetically

A screenshot of a computer code

Description automatically generated

<xsl:template match=”/”>

<xsl:variable name=”doc” select=”document(‘../movies.xml’)”>

<html><head> <title> </title></head>

<h2>Movies by directors</h2>

<table border>

<xsl:variable name=”unq\_directors” select=”$doc//Director[not @Name=preceding ::Director/@Name”>

<tr>

<th>

<th></tr>

<xsl:apply-templates select=”$unq\_directors”>

<xsl:sort select=”current()/@Name” order=”ascending”/>

</xsl:apply-templates>

</table>

</body>

</html>

</xsl:template>

<xsl:template match=”Director”>

A screenshot of a computer program

Description automatically generated

<xsl:template match=”Director”>

<xsl:variable name=”director\_name” select=”@Name”/>

<tr>

<td> <xsl:value-of select=”@Name”>

(<xsl:value-of select=”@Country”>)

</td> </tr>

<xsl:apply-templates select=”//Movie[Director/@Name=$director\_name]>

<xsl:sort select=”current()/@Title” order=”ascending”>

</xsl:apply-templates>

</xsl:template>

<xsl:template match=”Movie”>

<tr>

<td> <xsl:value-of select=”@Title”/></td>

<td> <xsl:value-of select=”@Year”/></td>

<td><xsl:value-of select=”ProductionCompany”/></td>

XSLT 2.0

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Table Setup**

* A table is initialized with headers for Title, Year, Production Company, and Number of Actors.

**Grouping and Sorting Movies by Director**

* **Grouping**: The **<xsl:for-each-group>** is used to group all **Movie** elements by the director's name (specified in an attribute).
* **Sorting**: Within the **<xsl:for-each-group>**, a sort order is specified to sort the directors alphabetically using **current-grouping-key()**, which in this context, refers to the director's name.