XSLT

Patryk Czarnik

XML and Applications 2016/2017
Lecture 10 – 19.05.2017

XSLT – where does it come from?

- XSL – E**x**tensible **S**tylesheet **L**anguage
  - Presentation of XML documents by transformation
- XSLT – XSL **T**ransformations
  - Language (XML) to define transformations
    - Transformation of a source XML document tree to a result tree
  - Designed as one of XSL components
  - General enough to be used for other purposes

XSLT – status

- Version 1.0
  - October 1999
  - Makes use of XPath 1.0
  - Popular and widely supported by tools
- Version 2.0
  - January 2007
  - Makes use of XPath 2.0, related to XQuery 1.0
  - More general (and specified in more consistent way) data model
  - More features
  - Less popular, little (but existing) support
- Version 3.0
  - Motivated mainly by development of XQuery 3.0

XSLT – availability of tools

- XSLT 1.0 processors:
  - Internet browsers (IE, Mozilla/Firefox, Opera, Chrome)
  - Apache Xalan (for Java and C++)
  - xsltproc (Linux and related OSs)
  - XML extensions of database engines
  - DTP tools
- XSLT 2.0 / 3.0 processors:
  - Saxon (for Java and .NET)
    - basic version free (Open Source)
    - full (*schema aware*) version paid
  - commercial tools: XML Spy, oXygen
- Authoring tools:
  - Hundreds of plain text editors (with syntax highlighting etc.)
  - Advanced programmer environments (Eclipse, Idea, ...)
  - Commercial XML-specialised tools (XML Spy, oXygen, ...)

Structure of XSLT stylesheet

- **Stylesheet** (*arkusz*) consists of templates
- **Template** (*szablon*) – building block of XSLT
  - transformation of single source node to result tree fragment
  - may be called many times for different nodes
- Within template:
  - text and elements out of XSLT namespace → copied to result
  - XSLT instructions → control flow, copying content from source, inserting computed content, and more features (in 2.0 even more)
  - XPath expressions in some instructions → accessing source document, checking conditions, calculating numbers, etc.
- XSLT can be seen as a programming language
  - with a bit non-standard syntax
  - convenient for processing and creating XML content

Structure of stylesheet – example (1)

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

  <xsl:template match="/">
    <html>
      <head>
        <title>Kindergarten pupils</title>
      </head>
      <body>
        <h1>Kindergarten pupils</h1>
        <xsl:apply-templates />
      </body>
    </html>
  </xsl:template>
```

Structure of stylesheet – example (2)

```
  <xsl:template match="kids">
    <ul>
      <xsl:apply-templates />
    </ul>
  </xsl:template>

  <xsl:template match="girl">
    <li color="pink">
      <xsl:apply-templates />
    </li>
  </xsl:template>

  <xsl:template match="boy">
    <li color="blue">
      <xsl:apply-templates />
    </li>
  </xsl:template>
</xsl:stylesheet>
```

XSLT in run

- Processing based on tree nature of XML documents
- Start: running template matching source document root (`/`)
  - such template exists even if not declared
- Other templates run when applied with `apply-templates` instruction
  - and so on, recursively
  - usually following shape of source document
  - matching nodes to patterns and selecting appropriate templates
  - template result pasted into result tree
- Other ways of flow control
  - `call-template` – invoking templates without pattern matching
  - `for-each` – iteration over sequences of nodes
  - `if` and `choose` – conditional processing

Pattern matching and template selection

- `match` attribute of template
  - **pattern**, restricted form of XPath expression
  - “What are the nodes this template applies to?”
- `select` attribute of `apply-templates` instruction
  - XPath expression
  - “Which nodes to process now?”
  - optional, children of current node selected if not given
- Matching
  - for each node to be processed, **independently**:
  - from all templates which patterns match the node
    - at least one always exists
  - select one with the highest priority
    - usually – one with the strictest pattern
  - in case of many templates with the same priority – conflict
    - error or latter template chosen, depending on implementation

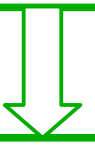
Pattern matching and template selection – example 1

```
<xsl:template match="kids">
  <ul>
    <xsl:apply-templates />
  </ul>
</xsl:template>

<xsl:template match="girl">
  <li color="pink">
    <xsl:apply-templates />
  </li>
</xsl:template>

<xsl:template match="boy">
  <li color="blue">
    <xsl:apply-templates />
  </li>
</xsl:template>
```

```
<kids>
  <girl>Alice</girl>
  <boy>Bob</boy>
  <girl>Cecil</girl>
  <girl>Dorothy</girl>
</kids>
```



```
<ul>
  <li color="pink">Alice</li>
  <li color="blue">Bob</li>
  <li color="pink">Cecil</li>
  <li color="pink">Dorothy</li>
</ul>
```

Pattern matching and template selection – example 2

```
<xsl:template match="cars">
  <ul> <xsl:apply-templates select="car"/> </ul>
</xsl:template>

<xsl:template match="car[@vmax > 200]">
  <li> <xsl:value-of select="name"/> (sports car) </li>
</xsl:template>

<xsl:template match="car">
  <li> <xsl:value-of select="name"/> </li>
</xsl:template>
```

```
<cars>
  <car vmax="290" name="Ferrari"/>
  <car vmax="180" name="Fiat"/>
  <car vmax="310" name="Jaguar"/>
</cars>
```

```
<ul>
  <li>Ferrari (sports car)</li>
  <li>Fiat</li>
  <li>Jaguar (sports car)</li>
</ul>
```

Built-in templates

- Templates applied if no user-provided template matches node
  - lowest priority
  - for each node there exists a template matching the node
- For document root and elements:
  - apply templates to children
    - preserve current mode
    - passing down all parameters passed to the current template
  - does not process attributes
- For text nodes and attributes:
  - copy text value to result
- For comments and processing instructions
  - do (and return) nothing

Processing modes

- `mode` attribute in `template` and `apply-temlates`
  - mode identified with an author-defined name
  - in XSLT 2.0 additional keywords: `#default`, `#any`, `#current`
- Alternative templates for the same nodes
- Possible applications:
  - processing the same nodes differently during the same transformation, e.g. details and summary
  - preparing library of different templates in advance and using selected subset in particular transformation

Processing modes – example

```
<xsl:template match="person">
  <li><xsl:apply-templates /></li>
</xsl:template>

<xsl:template match="person" mode="table">
  <tr><td><xsl:value-of select="fname" /> </td>
      <td><xsl:value-of select="surname"/></td></tr>
</xsl:template>

<xsl:template match="department">
  <table> ...
    <xsl:apply-templates select="person" mode="table"/>
  </table>
</xsl:template>
```

Combining stylesheets

Stylesheet files can be combined using top-level elements:

- `xsl:include`
  - simple inclusion of external stylesheet elements into current one
- `xsl:import`
  - importing templates from external stylesheet with lower precedence
  - acceptable only at the beginning of stylesheet
  - current and imported stylesheets form a tree – the further a template is defined, the lower precedence it has

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

  <xsl:import href="http://extraxsl.net/article.xsl"/>
  <xsl:include href="common.xsl"/>
  ...
</xsl:stylesheet>
```

Conditional processing – `if`

- Logical condition `test` checked (*Effective Boolean Value*)
- If true → body processed and result inserted into result tree
- No `else`

```
<xsl:template match="section">
  <xsl:if test="title">
    <h2>Title: <xsl:value-of select="title"/></h2>
  </xsl:if>
</xsl:template>
```

Conditional processing – choose

- Many branches
- Conditions (`test` in `when`) checked in order of occurrence
- (Only) first branch with satisfied condition processed
- Optional `otherwise` branch – used if no condition satisfied

```
<xsl:template match="account">
Account debit is
  <xsl:choose>
  <xsl:when test="debit &gt; 0"> positive </xsl:when>
  <xsl:when test="debit &lt; 0"> negative </xsl:when>
  <xsl:otherwise> zero </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Iteration over sequence – `for-each`

- For each node from `select`-ed set
  - node becomes the “current node”
  - body processed, result inserted into result tree
- Alternative way of walking through source document tree, but:
  - no pattern matching, each node processed in the same way
  - no separation of concerns, one blob of XSLT code in extreme case

```
<xsl:template match="department">
  <ul>
    <xsl:for-each select="person">
      <li><xsl:value-of select="fname"/>
          <xsl:value-of select="surname"/></li>
    </xsl:for-each>
  </ul>
</xsl:template>
```

Sorting

- **`sort`** instruction (“declaration”?) in `for-each`, `for-each-group` and `apply-templates`
- Sorting options in attributes:
  - `select` – what values are compared – processed nodes by default
  - `data-type`, `lang` – data type (`number` | `text`) and natural language
  - `order`, `case-order`, `stable`

```
<xsl:template match="osoby">
  <ul>
    <xsl:apply-templates select="osoba">
      <xsl:sort select="nazwisko"/>
      <xsl:sort select="imię"/>
    </xsl:apply-templates>
  </ul>
</xsl:template>
```

Grouping (XSLT 2.0)

- **for-each-group** instruction
- **current-group** and **current-grouping-key** functions

Example from Recommendation

```
<xsl:for-each-group select="cities/city" group-by="@country">
  <tr>
    <td><xsl:value-of select="position()"/></td>
    <td><xsl:value-of select="@country"/></td>
    <td>
      <xsl:value-of select="current-group()/@name"
                    separator=", "/>
    </td>
    <td><xsl:value-of
                select="sum(current-group()/@pop)"/></td>
  </tr>
</xsl:for-each-group>
```

Creating result tree

- Inserting result content into template body
  - convenient (hence “templates”)
- Creating nodes by constructor instructions
  - general
- Copying nodes from source document
  or computing values using XPath expressions
  - expressible

Inserting content into template body

- Content from template body copied to result:
  - text nodes, except whitespace-only text nodes (indentation etc.)
  - elements, except elements in XSLT namespace
    - together with their attributes
    - elements content processed recursively
- Applies to any “sequence constructor” in general
  - content of block instructions (`for-each`, `if`, etc.)
  - function body (XSLT 2.0), parameter and variable body

```
<xsl:template match="smthng">
  Some text
  <elem atrr="attribute value">
    Some other text
    <xsl:if test="1=1"> <other_element/> </xsl:if>
  </elem>
  <!-- Comments and PIs are not copied -->
</xsl:template>
```

Constructor instructions

- Instructions creating result nodes:

element **attribute** **text**

comment **processing-instruction**

```
<xsl:template match="person">
  <xsl:element name="{if parent::department/name='accountancy'
      then 'accountant' else 'specialist'}">

    <xsl:if test="@position = 'manager'">
      <xsl:attribute name="boss">yes</xsl:attribute>
    </xsl:if>
    <xsl:text>Text without leading/trailing spaces</xsl:text>
    <xsl:text> </xsl:text>
    <xsl:comment>Modification date:
        <xsl:value-of select="current-date()"/></xsl:comment>
</xsl:element>
<xsl:processing-instruction target="xml-stylesheet">
  type="text/css" href="styl.css"</xsl:processing-instruction>
```

Typical applications of constructor instructions

- Inserting processing instructions and comments
- Inserting sole whitespace characters
- Inserting text fragments without unwanted whitespace characters
- Element or attribute name established at runtime
- Inserting attribute conditionally

Shallow copying

- **copy** instruction
- Creating node “the same as” the current one
  - same kind
  - same name and namespace (if applicable)
- Content of current element not copied
- Content of `copy` instruction processed as usual
  - may be used to copy / process source element content, if needed

```
<xsl:copy>
    <xsl:if test="$process_contents">
      <xsl:apply-templates />
    </xsl:if>
</xsl:copy>
```

Inserting XPath expression result

- **`copy-of`** and **`value-of`** instructions
  - in XSLT 2.0 also **`sequence`**
- XPath expression given in `select` attribute
- Expression result inserted into transformation result, depending on instruction, as:
  - `copy-of` – deep copy of result
  - `value-of` – text node containing text representation of result
  - `sequence` – unprocessed sequence (matters only theoretically)

Examples

```
<xsl:copy-of  select="//person[@position='chief']"/>

<xsl:value-of select="//person[@position='chief']/surname"/>

<xsl:value-of select="salary + sales * commision div 100"/>
```

value-of in XSLT 1.0 – details

- If expression evaluates to simple value – that value printed
- If expression evaluates to node set – **only first node** from set cast to string and printed
- As slightly unintuitive – often leads to errors

```
<person><fname>Patryk</fname><surname>Czarnik</surname></person>
<person><fname>Radek</fname><surname>Bartosiak</surname></person>

<result><xsl:value-of select="//person"/></result>

<result>PatrykCzarnik</result>
```

`value-of` in XSLT 2.0 – details

- Expression evaluates to sequence (as always in XPath 2.0)
- Sequence flattened and atomized
- Result text:
  - string representations of all sequence items
  - separated with text provided in `separator` attribute
  - default separator – single space
- Incompatible with 1.0 behaviour

```
<person><fname>Patryk</fname><surname>Czarnik</surname></person>
<person><fname>Radek</fname><surname>Bartosiak</surname></person>

<result><xsl:value-of select="//person"/></result>

<result>PatrykCzarnik RadekBartosiak</result>
```

Inserting XPath result into attribute

- “Attribute value template”
- Given within braces
- Result of XPath expressions processed in the same way as with `value-of`

```
<img src="{$image_server}/{@id}.jpg"/>

<xsl:element name="h{count(ancestror-or-self::section)}">
 ...
</xsl:element>
```

`value-of` etc. – best practices

- Instructions with `select` – all processing XPath result somehow...
- Which one to use – depending on what is processed

value-of

- Value of simple type (number, string, date/time), in particular when computed by an expression or function
- Value of single text node / simple-type element / attribute
- Elements? – only when we want to flatten them to plain text
  - fully aware of unintuitive behaviour in XSLT 1.0
  - and incompatibilities between 1.0 and 2.0
  - ! Do not make your default manner of the last one, it is heavily overused in practice

`value-of` etc. – best practices

apply-templates

- Source document nodes
  - using `apply-templates` we enable them to be processed
  - by default text content printed
    - built-in templates used
    - similar to `value-of`, but more intuitive (all nodes printed)
  - templates – allow us to impact the treatment of nodes

copy-of

- Nodes copied directly from source
- Nodes created in advance, e.g. variable values
- Worth to use for variables/parameters even if they are text-only now (`value-of` would also work), preparing ourselves to enrich their structure in the future (then `value-of` would not work any more)

Local variables

- **`variable`** instruction
- “Declarative” approach – variables cannot change their values
  - They may assume different values in subsequent invocations
  - Analogous to `let` in functional programming or `final` variables in Java

```
<xsl:template match="account">
  <xsl:variable name="sign">
    <xsl:choose>
    <xsl:when test="debit > 0">positive</xsl:when>
    <xsl:when test="debit = 0">zero</xsl:when>
    <xsl:when test="debit &lt; 0">negative</xsl:when>
    </xsl:choose>
  </xsl:variable>

  Account debit is <xsl:value-of select="$sign"/>.
</xsl:template>
```

Consequences of declarative approach

```
<xsl:choose>
  <xsl:when test="debit > 0">
    <xsl:variable name="sign">positive</xsl:variable>
  </xsl:when>
  <xsl:otherwise>
    <xsl:variable name="sign">not positive</xsl:variable>
  </xsl:otherwise>
</xsl:choose>
Account debit is <xsl:value-of select="$sign"/>.
```

```
<xsl:variable name="sign">not positive</xsl:variable>
<xsl:if test="saldo > 0">
  <xsl:variable name="sign">positive</xsl:variable>
</xsl:if>
Account debit is <xsl:value-of select="$sign"/>.
```

Global variables and parameters

- Parameter value set from outside...
  - execution environment, command line, API call
- Variables computed once per transformation

```
<xsl:param name="name"/>
<xsl:variable name="elements-count"
  select="count(//element()[name() = $name)"/>
<xsl:variable name="message">
  <p>The document contains
    <xsl:value-of select="$elements-count"/> elements.</p>
</xsl:variable>
...
<xsl:template match="/">
  ... <xsl:copy-of select="$message"/> ...
</xsl:template>
```

Template parameters

- Declaration in template: **param**
- Value specification on invocation: **with-param**

```
<xsl:template match="department">
  <ul> <xsl:apply-templates select="person">
        <xsl:with-param name="prefix">Employee:</...>
       </xsl:apply-templates> </ul>
</xsl:template>

<xsl:template match="person">
  <xsl:param name="prefix"/>
  <li><xsl:copy-of select="$prefix"/>
      <xsl:apply-templates /></li>
</xsl:template>
```

Named templates

- `name` attribute of `template`
- **`call-template`** invokes
- Recursion permitted – high power of expression

```
<xsl:template name="describe-element">
  <p>Element with name <xsl:value-of select="name()"/>.</p>
</xsl:template>

<xsl:template match="/">
  <html><body>
    <xsl:for-each select="//*">
      <xsl:call-template name="describe-element"/>
    </xsl:for-each>
  </body></html>
</xsl:template>
```

Recursion and parameters

- Enable “programming” in XSLT (even 1.0)

Factorial

```
<xsl:template name="factorial">
  <xsl:param name="n"/>
  <xsl:param name="res" select="1"/>

  <xsl:choose>
  <xsl:when test="$n > 1">
    <xsl:call-template name="factorial">
      <xsl:with-param name="n" select="$n - 1"/>
      <xsl:with-param name="res" select="$n * $res"/>
    </xsl:call-template>
  </xsl:when>
  <xsl:otherwise>  <xsl:value-of select="$res"/>
  </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Custom function definitions (XSLT 2.0)

Factorial – naive recursive definition

```
<xsl:function name="loc:factorial">
    <xsl:param name="n"/>
    <xsl:sequence select="if($n &lt;= 1)
      then 1
      else $n * loc:factorial($n - 1)"/>
</xsl:function>
```

Place of use

```
<p>Factorial of 5 is equal to
  <xsl:value-of select="loc:factorial(5)"/>
</p>
```

Serialisation of transformation result

- Transformation takes place at level of document tree
- **Serialisation** – converting result tree into sequence of bytes
- Several formats (“methods”) supported:
  - `xml`
    - ordinary XML format, “general parsed entity”
    - proper XML document only if exactly one top-level element exists
  - `html`
    - uses only starting tags for empty elements (`<br>` rather than `<br/>`)
    - automatically adds `Content-Type` metadata to HTML `head` tag
  - `text`
    - omits all tags and print only content of result text nodes
  - `xhtml` (XSLT 2.0)
    - much like `xml`, but allows transformers to control result structure

Serialisation parameters

- `method` (`xml` | `html` | `text` | `xhtml`)
- `encoding` – character encoding
- `indent` (`yes` | `no`) – whether to automatically format output
- `version` – version of XML or HTML
- `doctype-public`, `doctype-system` – DOCTYPE declaration

Example: XHTML 1.0 declaration in XSLT 1.0

```
<xsl:output method="xml" version="1.0" encoding="utf-8"
  doctype-public="-//W3C//DTD XHTML 1.1//EN"
  doctype-system="http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"/>
```

Beginning of result document

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE PUBLIC "-//W3C//DTD XHTML 1.1//EN"
                 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"/>
```

Result of transformation – common applications

- XSL Formatting Objects:
  - according to original XSL approach
  - especially useful for printed publication
- HTML or XHTML
  - most popular
  - especially useful for Web publication
- Arbitrary XML, e.g.:
  - migration to new / other format
  - acquiring and processing data (alternative to XQuery)
  - generating XSLT as result of XSLT
- Plain text, e.g.:
  - CSV and other text data formats
  - scripts and configuration files
  - converting text documents to non-XML text formats

Temporary result tree fragments

- XSLT 1.0 – distinct types *node set* and *result tree fragment*
  - not allowed to mix
  - not allowed to process result tree fragment again
- XSLT 2.0 – no such restriction
  - In particular: result (e.g. of value-of or apply-templates) can be read and processed again
  - this is the real programming and full recursion!

XSLT 2.0 but not XSLT 1.0

```
<xsl:variable name="tmp">
  <xsl:apply-templates select="document"/>
</xsl:variable>

<xsl:apply-templates select="$tmp" mode="correct"/>
```

Some typical patterns

- Recommended structure of stylesheet depends on application
- Typical patterns and best practices for
  - text document presentation
  - data processing and presentation
  - copying and filtering

Text document presentation

(this subject is going to be extended in the next lecture)

- Processing driven by source document structure (*push*)
- Many simple templates
  - one template for one type of input elements
  - translating source element into target element and processing its content with `apply-templates`
- More elaborated templates for elements requiring special treatment...
- Shape of result tree based on shape of source tree
- Works well with mixed content typical to text documents

Text document presentation

Root template

```
<xsl:template match="/">
  <html>
    <head>...</head>
    <body>
      <xsl:apply-templates />
  </body></html>
</xsl:template>
```

Typical simple template

```
<xsl:template match="item">
  <li>
   <xsl:apply-templates />
  </li>
</xsl:template>
```

More elaborated template

```
<xsl:template match="person">
  <div class="person">
     <xsl:apply-templates select="fname"/>
   <xsl:text> </xsl:text>
     <xsl:apply-templates select="surname"/>
   <br />
     <xsl:apply-templates select="contact"/>
  </div>
</xsl:template>
```

Data processing and presentation

- Processing driven by intended result structure (*pull*)
- One (or a few) large template
  - fixes structure of result
  - inserts acquired / computed data into intended places using `value-of`
  - iterates over monomorphic sequences using `for-each`
- Complex XPath expressions
  - or even “XSLT programming” to compute more complex things

Data processing and presentation

One big template

```
<xsl:template match="/company">
  <div class="report">
    <p>Number of employees:
      <xsl:value-of select="count(.//person)" /></p>

    <p>Personal expenditures annually:
      <xsl:value-of select="12*sum(.//person/salary)" /></p>

    <p>Expenditures per department:</p>
    <dl>
      <xsl:for-each select="department">
        <xsl:sort select="name"/>
        <dt><xsl:value-of select="name" /></dt>
        <dd><xsl:value-of
                  select="12*sum(person/salary)" /></dd>
      </xsl:for-each>
    </dl>
  </div>
</xsl:template>
```

Filtering and copying

- Returning subset of source document

Two styles, depending on needs:

- Big step style (analogous to target-driven processing)
  - large fragments of input copied as whole
  - `copy-of` within big template
- Small step style (analogous to source-driven processing)
  - recursive processing with exclusion or specialised transformation of selected nodes
  - `copy` as default and specialized templates for elements to be processed

Copying and filtering – big step approach

Coarse-grained filtering template

```
<xsl:template match="/company">
  <company>
    <xsl:copy-of select="department[count(person) > 100]"/>
  </company>
</xsl:template>
```

Copying and filtering – small step approach

Standard copy template

```
<xsl:template match="@*|node()">
  <xsl:copy>
    <xsl:apply-templates select="@*"/>
    <xsl:apply-templates />
  </xsl:copy>
</xsl:template>
```

Example: template changing element name

```
<xsl:template match="collection/item">
  <collection-item>
    <xsl:apply-templates select="@*" />
    <xsl:apply-templates />
  </collection-item>
</xsl:template>
```

“Advanced” template

```
<xsl:template match="para">
  <p>Paragraph of size
        <xsl:value-of select="string-length(string(.))"/>
  </p>
</xsl:template>
```