
# 181135 VU Semistrukturierte Daten 1

XSL - Extensible Stylesheet Language

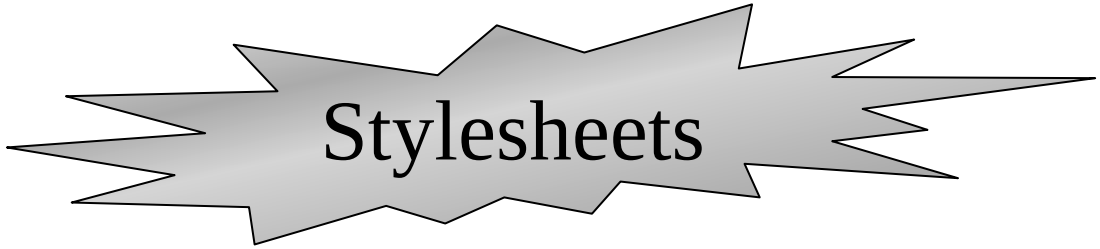
(Teil 1)

25.10.2005

*Reinhard Pichler*

# Inhalt

- **Stylesheets**
- **XSLT**
  - **Aufbau eines XSLT Stylesheets**
  - **Abarbeitung eines XSLT Stylesheets**
  - **Erzeugung des Result tree**
  - Kontrollstrukturen: for-each, if, choose
  - Sortieren, Nummerieren
  - Variablen, Parameter
  - zusätzliche Funktionen (gegenüber XPath)
- XSLFO




- Aufgaben von Stylesheets:
  - Transformation
  - Layout
  - Informationsextraktion/Abfragen
- Konvertieren von SGML Dokumenten
  - Dynamic Style Semantic and Specification Language (DSSSL)
- Konvertieren von XML Dokumenten
  - in andere XML Dokumente
  - in HTML und andere Formate
  - in Text

# Stylesheets

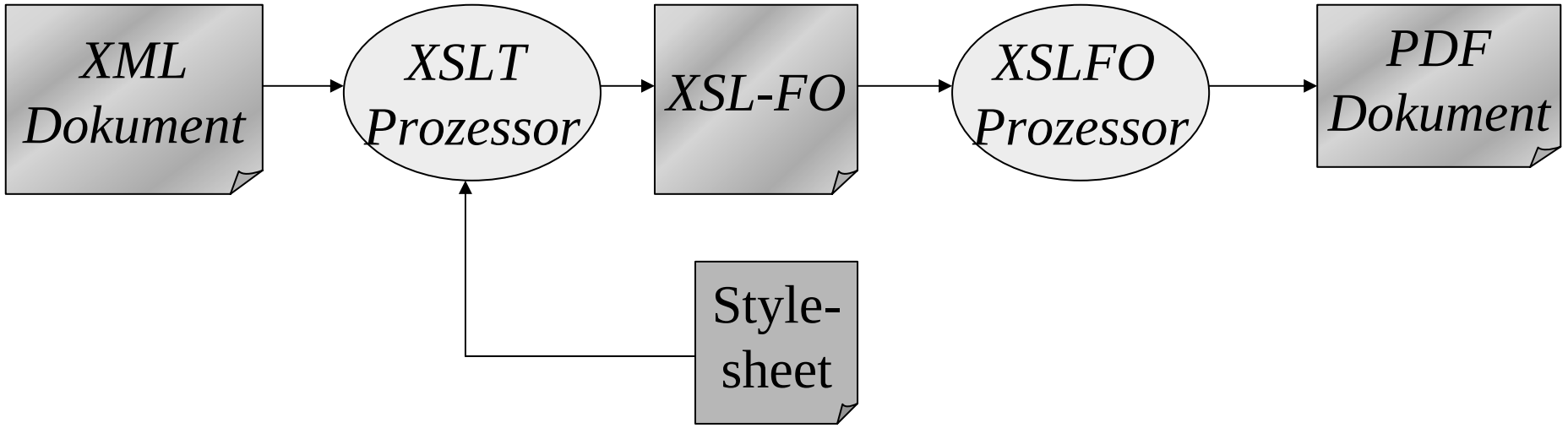
- Ideen
  - Trennung von Inhalt und Präsentation
  - deviceunabhängiges Layout
  - verschiedene Views
  - Wiederverwendbarkeit von Templates
- DSSSL
  - für SGML
  - ISO Standard 1996
  - umfassend, basierend auf LISP Dialekt
  - bestehend aus:
    - Transformationssprache
    - Formatierungssprache
    - Abfragesprache

# Stylesheets

- CSS
  - für HTML entwickelt
  - Regelmenge für Darstellung
  - p {font-size 10pt; font-style:italic}
  - weit verbreitet
  - einfaches Formatierungsmodell, nicht zur Restrukturierung
- XSL
  - verwendet XML Syntax
  - unterstützt Transformation zusätzlich zur Präsentation
  - Sprachteile von XSL: **XSLT:** Transformation
    **XSLFO:** Formatierungsobjekte
  - außerdem: **XPath:** Navigieren im Dokumentenbaum
- XSL mächtiger
  - CSS deckt nur den Teil der Formatierung ab

# XSL

- XSLT benötigt XSL Prozessor
  - z.B. IE6, Xalan, Saxon, XT (James Clark)
  - Transformation des Source Tree in einen Result Tree
  - In Praxis: meist ohne XSL:FO zur Erstellung von HTML, Text, SVG, etc.
- XSLFO benötigt FO Prozessor
  - Apache FOP für PDF Generierung
  - Formatieren des transformierten Dokumentes

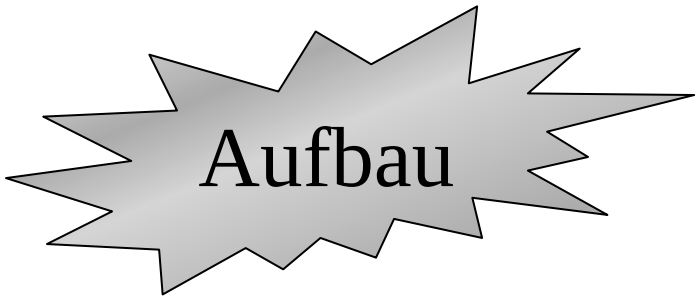



# XSLT and XPath – the keys to the XML kingdom

J.R.Gardner, Z.L.Rendon

# XSLT Prozessoren

- Xalan (Teil des Apache Projekts)
  - http://xml.apache.org/xalan-j/index.html (auch C++ vorhanden)
  - Stylesheet als Input oder in XML Dokument referenziert
  - **`set CLASSPATH=xalan.jar;xerces.jar`**
  - **`java org.apache.xalan.xslt.Process –IN test.xml -XSL test.xsl [-OUT out.xml]`**
- Saxon (von Michael Kay)
  - http://saxon.sourceforge.net/
  - **`set CLASSPATH=saxon.jar;saxon-jdom.jar`**
  - **`java com.icl.saxon.StyleSheet test.xml test.xsl > out.xml`**
- XML-Spy
  - enthält unter anderem: XSLT Prozessor, XSLT Debugger
  - Gratisversion für 30 Tage erhältlich
  - http://www.altova.com/products_ide.html




- XSLT-Stylesheets sind selbst XML-Dokumente

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0" >
   <!-- Hier kommen die top-level Elemente -->
</xsl:stylesheet>
```

- Aufbau:
  - Namespace: "http://www.w3.org/1999/XSL/Transform"
  - Präfix: üblicherweise xsl
  - Dokumentelement: <xsl:stylesheet> (oder <xsl:transform>)
- Verknüpfung XML-Dokument/Stylesheet:
  - mittels PI: `<?xml-stylesheet type="text/xsl" href="lva.xsl"?>`
  - oder als Parameter beim Aufruf des XSLT-Prozessors

# Top-level Elemente

| | |
|---|---|
| xsl:import | importiert ein anderes XSLT stylesheet |
| xsl:include | betrachtet die top-level Elemente eines anderen XSLT stylesheets als Teil des eigenen Stylesheets |
| xsl:strip-space | Lösche Textknoten mit reinem Whitespace |
| xsl:preserve-space | Erhalte Textknoten mit reinem Whitespace |
| xsl:output | Ausgabemethode: XML, HTML, Text |
| xsl:key | Definiere Schlüssel (Verbesserung gegenüber ID-Attributen) |
| xsl:decimal-format | Format von Dezimalzahlen |
| xsl:attribute-set | Definiere Attributmengen für das Output-Dokument |
| xsl:variable | globale Variablen |
| xsl:param | Parameter |
| xsl:template | wichtigste Elemente eines XSLT-Stylesheet: definieren Regeln für die Transformation source tree -> result tree |
| xsl:namespace-alias | Festlegen eines anderen NS-Präfix im Output-Dokument |
| literal result elements | Elemente außerhalb des xsl-Namespace: werden in den Output wörtlich durchgereicht |

Die Reihenfolge der Elemente ist irrelevant außer bei xsl:import-Elementen (wo die Reihenfolge über die import precedence entscheidet).

# Weitere XSL Elemente

| | |
|---|---|
| xsl:apply-templates | bewirkt rekursiven Abstieg im Source Tree |
| xsl:call-template | Aufruf eines Templates mit dessen Namen |
| xsl:element | erzeugt Element mit angegebenem Namen |
| xsl:attribute | erzeugt Attribut mit angegebenem Namen |
| xsl:attribute-set | für Attributmenge |
| xsl:text | erzeugt Textknoten |
| xsl:processing-instruction | erzeugt PI |
| xsl:comment | erzeugt Kommentar |
| | |
| xsl:value-of | Textknoten mit berechnetem String-Wert |
| xsl:copy | kopieren des momentanen Knoten |
| xsl:copy-of | kopieren eines ganzen Sub–Baums |
| xsl:for-each | Schleife über eine Knotenmenge |
| xsl:if | bedingte Anweisung |
| xsl:choose / xsl:when | |
| xsl:otherwise | Alternativen |
| xsl:sort | sortiert eine Knotenmenge |
| xsl:number | erstellt formatierte Zahl |

# Templates

- Attribute von xsl:template:
  - match="pattern": Aktion wird ausgeführt für alle Knoten, für die dieser Patternpfad erfüllt ist, d.h.: Alle Knoten, für die es einen context-node gibt, von dem aus sie mit diesem Patternpfad selektiert würden.
  - name: benanntes template

    (Aufruf **`<xsl:call-template name="name"/>`** )
  - mode: Modus, damit mehrere Templates für einen Knoten möglich sind
  - priority: Ändern der Default-Priorität
- Attribute von xsl:apply-templates:
  - select: wende Templates nur auf selektierte Knoten an
  - mode: wende nur Templates mit angegebenem Modus an

# Templates mit verschiedenen Modi

- Idee:
  - selber Knoten des Source Tree soll auf verschiedene Arten bearbeitet werden (z.B.: unterschiedliche Aufbereitung des Outputs)
  - benötigt Regeln mit verschiedenen „Modi“
- mode-Attribut:
  - in xsl:template
  - bei Aufruf mit xsl:apply-templates

```
<xsl:template match="/">....
  <xsl:apply-templates select="//titel" mode="modeA">
......
</xsl:template>

<xsl:template match="titel" mode="modeA">
....
</xsl:template>

<xsl:template match="titel" mode="modeB">
....
</xsl:template>
```

optional

# Beispiel

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns="http://www.w3.org/TR/REC-html40" version="1.0">

 <xsl:output method="html"/>

 <xsl:template match="/">
   <html><xsl:apply-templates/></html>
 </xsl:template>

 <xsl:template match="veranstaltung/titel">
        <i><xsl:value-of select="."/></i><br/>
 </xsl:template>

 <xsl:template match="text()">
 </xsl:template>

</xsl:stylesheet>
```

*Elementinhalt wird in den Output geschrieben*

*Zusätzliches Template benötigt, das den Text aller anderen Elemente nicht schreibt.*

Browser

*AK der IK 3*

*AK der AI 4*

*VU Datenbanken*

Xalan,
Saxon,
IE6

*Beispiel Xal01*

# Kurzschreibweise mittels LRE

- LRE (= literal result element):
  - d.h.: Element außerhalb des xsl-Namespace; LRE darf selbst beliebig komplexen Content (insbes. xsl-Elemente außer top-level Elementen) haben..
  - kann top-level element sein. In diesem Fall: erlaubte Kurzschreibweise für das Stylesheet, z.B.:

```
<?xml version = "1.0" encoding = "ISO-8859-1" ?>
<html xsl:version="1.0"
      xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
      xmlns = "http://www.w3.org/TR/xhtml1/strict">
  <head>
    <title>Lehrveranstaltungsübersicht</title>
  </head>
  <body>
    <p>Anzahl der LVAs:
       <xsl:value-of select="count(//veranstaltung)"/></p>
  </body>
</html>
```

- Ist äquivalent zu folgendem Stylesheet:

```
<?xml version = "1.0" encoding = "ISO-8859-1" ?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns="http://www.w3.org/TR/xhtml1/strict">
 <xsl:template match="/">
   <html>
     <head>
       <title>Lehrveranstaltungsübersicht</title>
     </head>
     <body>
       <p>Anzahl der LVAs:
          <xsl:value-of select="count(//veranstaltung)"/></p>
     </body>
   </html>
 </xsl:template>
</xsl:stylesheet>
```

# Abarbeitung

- Stylesheet ist deklarative Regelmenge (templates)

```
<xsl:template match="XPATH Ausdruck">
        Substitutionsausdruck
</xsl:template>
```


- XSLT Prozessor durchläuft Source Tree ausgehend von der Wurzel
  - Depth-First Traversal des Source Tree
  - es gibt immer einen current node und eine current node list
  - XSLT Prozessor sucht passendes Template für den current node
  - Wenn current node das XPath-Pattern matched, wird das Template ausgeführt
  - Immer nur 1 Template pro current node => conflict resolution (s.u.)
- apply-templates Element:
  - **Strukturelle Rekursion:** wende Templates rekursiv auf Kinder (bzw. Nachfahren) im Source Tree an.
  - Ohne „apply-templates“: Nach Abarbeitung eines Templates, würde die Bearbeitung des aktuellen Pfades enden und der Prozessor mit der current node list bzw. mit dem parent weiter machen.

# Select vs. Match

- **select-Attribut in xsl:apply-templates**
  - **Aufgabe des location path**: Bestimme die Menge aller Knoten *Y*, die man vom context node *X* aus mit dem select-Pfad selektieren kann. Für jeden dieser Knoten *Y* wird dann ein passendes Template gesucht wird.
  - auch absolute Pfade möglich -> nach Abarbeitung der dadurch aktivierten templates Rückkehr zum normalen Knotenabarbeitungsmechanismus
  - beliebige Achsen im Pfad erlaubt
  - Vorsicht: Endlosschleifen möglich
- **match-Attribut in xsl:template**
  - **Aufgabe des location path**: Für einen best. Knoten *Y* (der mittels apply-templates selektiert wurde) wird getestet, ob dieser von irgendeinem context-node *X* aus mit dem match-Pfad selektiert würde.
  - nur child und attribute Achse sowie Abkürzung // erlaubt!
  - absolute Pfade erlaubt.

# „Computational Power“

- Imperative Programmiersprachen:
  - Fallunterscheidungen mittels IF, IF ELSE, etc.
  - Wiederholung mittels Schleifen (oder Rekursion)
- XSLT (analog zu deklarativen Programmiersprachen wie PROLOG und LISP)
  - Pattern Matching
  - (structural) Recursion
- Kontrollstrukturen in XSLT:
  - if, for-each, choose „vereinfachen“ das Leben
  - XSLT ist Turing-vollständig (auch ohne Kontrollstrukturen)

# Built-In Templates

- Idee: Falls es für einen Knoten kein Template gibt, dann werden vom Prozessor built-in Templates verwendet.
- Folgende Templates sind (im XSLT Standard) vordefiniert:
  - **Elemente:** built-in template erstellt keine neuen Knoten, aber ruft rekursiv die Kinder des current node auf (damit strukturelle Rekursion weitergeht)

```
<xsl:template match="* | /">
  <xsl:apply-templates/>
</xsl:template>
```

  - **Attribute und Text-Knoten:** werden standardmäßig einfach in den Output geschrieben. Allerdings: Attribute werden beim Dokumentendurchlauf nicht durchlaufen, d.h.: „apply-templates“ damit built-in Template greift!

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```


- Modi:
  - Built-in Regeln verändern nicht den Modus (d.h.: als ob es für jeden Modus einen eigenen Satz von built-in Regeln geben würde)

# Conflict Resolution

- Prinzip:
  - Auf jeden Knoten wird immer nur exakt ein Template angewandt
  - Wenn mehrere Templates den current node matchen, muss ein Template ausgewählt werden.
- Auswahlkriterien:
  - import precedence: importierte Regeln kommen immer nach den Regeln des importierenden Stylesheets
  - priority: Regel mit der höchsten Priorität (entweder priority Attribut oder default priority) wird ausgeführt..
- Verhalten bei Clash
  - d.h.: wenn mit diesen Auswahlkriterien keine eindeutige Auswahl möglich ist
  - Verhalten prozessorabhängig
  - entweder Abbruch mit Error
  - oder Auswahl des letzten Templates im Stylesheet

# Default Priority

- Idee:
  - Je spezifischer eine Regel ist, umso höher ist die Priorität.
  - Der „Normalfall“ (d.h.: child/attribute-Achse + qualifizierter Name) bekommt Priorität 0.
  - Spezifischere Patterns haben positive Priorität.
  - Weniger spezifische Patterns haben negative Priorität.
- Berechnung der default priority:
  - Mit | getrennte Patterns werden wie 2 getrennte Templates behandelt
  - child/attribute-Achse + qualifizierter Name: Priority = 0
  - child/attribute-Achse + unqualifizierter Name: Priority = -0.25
  - child/attribute-Achse + NodeTest: Priority = -0.5
  - sonst (insbes.: > 1 location step) Priority = 0.5

# Umgehung der Precedence/Priority

- priority-Attribut bei xsl:template

```
<xsl:template match="expression" priority= "1.5">
```

- xsl:apply-imports
  - Wird ohne Attribute/Inhalt innerhalb eines Templates aufgerufen
  - Sucht importierte Regel, die den current node matcht

```
<xsl:template match="expression">
   ...
   <xsl:apply-imports/>
   ...
</xsl:template>
```

- xsl:call-template
  - Definiere template mit name-Attribut
  - Dieses Template kann dann mit diesem Namen direkt aufgerufen werden -> operiert auf dem current node.

```
<xsl:template match="expression" name="xxx"> .....

<xsl:call-template name="xxx"> .....
```

# Outputmethoden


- XML, HTML und Text unterstützt (XML ist der default)
  - Top level element, z.B.: **`<xsl:output method="HTML"/>`**
  - prozessorspezifisch (in Xalan):
    - XML – fügt XML Header ein
    - HTML – ohne Header
    - Text – ignoriert Tags
- weitere Attribute von xsl:output
  - omit-xml-declaration="yes|no"
  - standalone="yes|no"
  - version="..." (Default 1.0)
  - encoding="..."
  - indent="yes|no" (Wert der Einrückung nicht angebbar)

# Literal Result Elements

- LRE = Element, das nicht zum xsl-Namespace gehört
  - Typische Anwendung: Umwandlung xml -> html
- LREs werden (inclusive Attribute und Namespaces) direkt zum Output durchgereicht
- können xsl-Elemente enhalten (aber nicht top-level Elemente)
- Beispiel:

```
<xsl:template match="veranstaltung">
  <veranstaltung>
     <termin nr="1"> .... </termin>
     <termin nr="2"> .... </termin>
     <termin nr="3"> .... </termin>
  </veranstaltung>
</xsl:template>
```

# xsl:element

- Erzeugt ein Element im Result Tree
- Attribute von xsl:element:
  - name (mandatory): Name des Elements
  - namespace: URI-reference
    (Behandlung des Präfix ist prozessorabhängig)
  - use-attribute-sets: Liste von (global definierten) Attributmengen
- Vorteile gegenüber LRE:
  - Name kann mittels expression berechnet werden
- Beispiel:

```
<xsl:element name="lehrveranstaltungen"
  namespace="http://www.dbai.tuwien.ac.at/veranstaltungen">
  <xsl:apply-templates/>
</xsl:element>
```

# xsl:attribute


- Erzeugt ein Attribut im Result Tree
  - Als (erstes) Sub-Element des zugehörigen Elements
  - Sowohl bei LRE als auch bei xsl:element erlaubt
- Attribute von xsl:attribute:
  - name (mandatory): Name des Attributs
  - namespace: URI-reference
    (Behandlung des Präfix ist prozessorabhängig)
- Attributwert: als Content des xsl:attribute-Elements
- Vorteile gegenüber LRE + Attribut:
  - Name kann mittels expression berechnet werden
  - Beliebig komplexer Content (insbes. xsl-Elemente) des xsl:attribute-Elements zur Berechnung des Attributwerts (z.B. xsl:value-of, xsl:apply-templates)

# Beispiel

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

<xsl:template match="text()"/>

<xsl:template match="veranstaltung">
  <veranstaltung>
    <xsl:element name="termin">
      <xsl:attribute name="nr">1</xsl:attribute>
      .....
    </xsl:element>
    <xsl:element name="termin">
      <xsl:attribute name="nr">2</xsl:attribute>
      .....
    </xsl:element>
    <xsl:element name="termin">
      <xsl:attribute name="nr">3</xsl:attribute>
      .....
    </xsl:element>
  </veranstaltung>
</xsl:template>

</xsl:stylesheet>
```

# xsl:attribute-set

- xsl:attribute-set
  - Top-level Element im XSLT-Stylesheet
  - um Gruppe von Attributen zu definieren
  - Content: geschachtelte xsl:attribute-Elemente
- Attribute von xsl:attribute-set
  - name (mandatory): Name der Attribut-Menge
  - use-attribute-sets: um andere Attribut-Mengen zu referenzieren
- Verwendung (in Elementen oder anderen Attribut-Mengen)
  - Mittels Attribut use-attribute-sets
- Typische Anwendung:
  - verschiedende attribute-sets für unterschiedliche Format-Anweisungen im Output

# Beispiel

```
<xsl:attribute-set name="plain_table">
    <xsl:attribute name="border">0</xsl:attribute>
    <xsl:attribute name="cellpadding">3</xsl:attribute>
    <xsl:attribute name="cellspacing">4</xsl:attribute>
</xsl:attribute-set>
<xsl:attribute-set name="bordered_table">
    <xsl:attribute name="border">1</xsl:attribute>
    <xsl:attribute name="cellpadding">2</xsl:attribute>
    <xsl:attribute name="cellspacing">3</xsl:attribute>
</xsl:attribute-set>

<xsl:template match ="buecher">
  <xsl:element name="table" use-attribute-sets ="plain_table">
    <xsl:apply-templates/>
       <!-- z.B.: Tabelleneintrag fuer jedes buch-Element -->
  </xsl:element>
</xsl:template>
```

# Attributwerte berechnen

- nicht direkt möglich über

  ```
  <elem attr="<xsl:value-of select ='expr'.....>
  ```

  da kein Markup im Markup erlaubt ist

- Zwei Möglichkeiten:
  - AVT (=Attribute Value Template): XPath Ausdruck in { } angeben

    ```
    <elem attr="{expr}".....>
    ```

    Beispiel (wandelt Element-Inhalt in Attribut-Wert um):

    ```
    <xsl:template match="elem">
       <new-elem attr="{text()}"> ... </new-elem>
    </xsl:template>
    ```

  - xsl:attribute-Element: Attributwert = Content des xsl:attribute-Elements

# Andere Knoten-Typen

- xsl:text
  - Erzeugt einen Text-Knoten, z.B.:
    ```
    <xsl:text disable-output-escaping="yes"><&amp;</xsl:text>
    ```
  - Hauptanwendungen:
    - Whitespaces kontrollieren (z.B.: vermeide/erzwinge Zeilenumbrüche)
    - Escaping vermeiden (aber nur von den 5 predefined character entities in XML: & < > " ')
- xsl:comment
  - Erzeugt einen Kommentar, z.B.:
    ```
    <xsl:comment>Hier steht der Kommentar</xsl:comment>
    ```
- xsl:processing-instruction
  - Erzeugt eine PI, z.B.:
    ```
    <xsl:processing-instruction name="xml-stylesheet">"text/css"
    href="lehre.css"</xsl:processing-instruction>
    ```
- Content dieser Elemente:
  - xsl:text: PCDATA
  - xsl:comment und xsl:processing-intruction: bel. komplexer Content.

# xsl:namespace-alias


- xsl:namespace-alias
  - Top-level Element im XSLT-Stylesheet
  - definiert einen Namespace-URI als Alias für einen anderen
- Attribute von xsl:namespace-alias
  - stylesheet-prefix (mandatory): NS-URI oder `"#default"`
  - result-prefix (mandatory): NS-URI oder `"#default"`
- Typische Anwendung:
  - Wenn Result selbst ein XSLT-Stylesheet ist, z.B.:

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:myxsl="http://www.w3.org/1999/XSL/TransformAlias">
 <xsl:namespace-alias
    stylesheet-prefix="myxsl" result-prefix="xsl"/>
 <xsl:template match="/">
   <myxsl:stylesheet> ..... </myxsl:stylesheet>
 </xsl:template>
 .....
```

# xsl:value-of

- xsl:value-of
  - Erzeugt einen Textknoten im Result Tree (der mit angrenzenden Text-Knoten gemerged wird).
- Attribute von xsl:value-of
  - select (mandatory): expression mit Result Type String
    (andere Typen werden zu String konvertiert)
  - disable-output-escaping: wie bei xsl:text (d.h.: nur für & < > " ')
- Beispiel:

```
<xsl:template match="person">
   <p>
     <xsl:value-of select="@vorname"/>
     <xsl:text> </xsl:text>
     <xsl:value-of select="@nachname"/>
   </p>
</xsl:template>
```

# xsl:copy

- xsl:copy
  - Kopiert den current node in den Output („shallow copying“)
  - Bei Element-Knoten: NS-Knoten werden ebenfalls kopiert, aber weder Attribute noch Content
  - Das xsl:copy-Element selbst kann bel. Content haben (ist nur relevant, wenn der current node ein Element-Knoten oder der root node ist)
  - Optionales Attribut: use-attribute-sets
- Beispiel (identity transformation):

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

# xsl:copy-of

- Hat leeren Content
- Einziges Attribut: select
  - Wenn die expression des select-Attributs ein node-set ergibt:

    => Kopiert alle Bäume, deren Wurzel in der Knotenmengen liegen, in den Output (insbesondere auch Attribute und Content von Element-Knoten)
  - Bei Result Type string/boolean/number:
    => xsl:copy-of verhält sich gleich wie xsl:value-of
- Unterschiede zw. xsl:copy-of und xsl:copy:
  - xsl:copy kopiert nur den Markup, xsl:copy-of auch den Inhalt
  - xsl:copy kann selbst bel. komplexen Content enthalten, um den Element-Inhalt im Result Tree zu erzeugen. xsl:copy-of ist ein leeres Element.

# Whitespaces

Whitespace-Verhalten steuerbar mit:

- <xsl:text>
- <xsl:output indent="yes"/>
- normalize-space (expr)
- <xsl:strip-space elements="el1 el2"/> definiert Elemente, bei denen Whitespaces getrimmt werden
- Gegenteil: <xsl:preserve-space> (ist der Default)
- Attribut xml:space in jedem Element erlaubt: Werte preserve|strip

# Übungsbeispiel A

Betrachten Sie die HTML-Datei Teilnehmer.htm von Studenten, die eine Lehrveranstaltung besuchen.

1. Modifizieren Sie diese Datei so weit, dass eine wohlgeformte XML-Datei entsteht (z.B.: ersetze <br> durch <br/>, etc.) und testen Sie die Wohlgeformtheit mittels XML-Parser.

2. Erstellen Sie ein XSLT Stylesheet Htm2Xml.xsl, das die (modifizierte) Datei Teilnehmer.htm in ein XML-Dokument Teilnehmer.xml umformt.

   – Diese XML-Datei soll aus einem Dokumentenelement LVA mit einem Subelement TITEL und beliebig vielen Unterelementen TEILNEHMER bestehen.
   – Jedes TEILNEHMER-Element besteht aus Subelementen MatrNr, Nachname, Vorname, Kennzahl, Email.

Beachten Sie Folgendes beim XSLT-Stylesheet:

- Verzichten Sie auf xsl:for-each, xsl:if und xsl:choose Elemente.

# Übungsbeispiel B

Erstellen Sie ein XSLT-Stylesheet, das eine eins-zu-eins Kopie eines beliebigen XML-Dokuments erstellt.

- Verzichten Sie auf xsl:copy, xsl:copy-of und LREs
- Verwenden Sie statt dessen xsl:element, xsl:attribute, xsl:text, xsl:processing-instruction und xsl:comment.
- Testen Sie das XSLT-Stylesheet mit Hilfe der XML-Dateien lva.xml und lehre.xml