

Folien zum Lehrmodul

XSLT, Teil 2 (Stichworte)

Lernziele:

- Transformationsregeln im Detail kennen und benutzen können (Übungen!!)
- weitere zugehörige XSLT-Kommandos (include, import, apply-templates, copy) benutzen können

Inhaltsverzeichnis

1	Einleitung und Wiederholung	5
1.1	Bisheriger Lehrstoff	5
1.2	Neuer Lehrstoff - Übersicht	6
1.3	Transformationsregeln – Motivation	7
2	Definition und Verwaltung von Transformationsregeln	9
2.1	Konzeptueller Inhalt einer Transformationsregel	10
2.2	Spezifikationen im Parameter <code>match</code>	12
2.2.1	Grundidee	12
2.2.2	Beispiele	14
2.2.3	Spezifikation mehrerer Knotentypen mit <code>*</code>	15
2.2.4	Aufzählung mehrerer Knotentypen	18
2.2.5	Pfade in <code>LocationPathPatterns</code>	19
2.2.6	Prioritäten von Transformationsregeln	26
2.3	Verwaltung von Transformationsregeln	28
2.3.1	Kommando <code>xsl:include</code>	29

2.3.2	Kommando <code>xsl:import</code>	30
3	Expliziter Aufruf von Transformationsregeln	31
3.1	Kommando <code>xsl:apply-templates</code>	31
3.2	Beispiel	33
3.3	Vorgabewert für <code>select</code>	43
4	Vordefinierte Transformationsregeln	46
5	Die identische Transformation und Projektionen	52
5.1	Die identische Transformation	52
5.2	Kommando <code>xsl:copy</code>	54
5.3	Projektionen	55

1 Einleitung und Wiederholung

1.1 Bisheriger Lehrstoff

- nur 1 Transformationsregel für Dokumentwurzel, impliziter Aufruf dieser Transformationsregel bei Programmstart
- Kontext beim Aufruf einer Transformationsregel
- Ausführung einer Schablone (= Rumpf der Transformationsregel)
- Kommandos `value-of`, `text` und `for-each`
- `for-each`-Kommando hat innere Schablone; diese wird mehrfach in verschiedenen Kontexten ausgeführt

1.2 Neuer Lehrstoff - Übersicht

- Definition und Verwaltung vieler Transformationsregeln
- expliziter Aufruf von Transformationsregeln
- Priorität von Transformationsregeln
- vordefinierte Transformationsregeln
- identische Transformation und Projektionen

1.3 Transformationsregeln – Motivation

einfaches Beispiel einer “XML-Abfrage”:

- gegeben sei eine XML-Datei, die eine Adreßliste enthält
- gesucht: “Projektion” auf Nachname und Telefonnummer, genauer gesagt:
 - i.w. gleicher Syntaxbaum
 - viele Knoten müssen identisch kopiert werden
 - manche Knoten / Teilbäume werden weggelassen
 - manche Knoten werden umbenannt

resultierende **Anforderungen an eine XML-Abfragesprache:**

1. Wiederholung: der *komplette Ausgabebaum muß in 1 Verarbeitungsschritt aufgebaut werden*
2. es sollte nicht nötig sein, den Durchlauf durch den Eingabebaum manuell zu programmieren
(wäre immer wieder das gleiche Hauptprogramm)
→ möglichst nur *die Abweichungen* vom Standarddurchlauf angeben
3. das Kopieren von Teilbäumen / Baumfragmenten sollte einfach sein / wenig Schreibaufwand verursachen

2 Definition und Verwaltung von Transformationsregeln

Zur Erinnerung noch einmal die Syntax:

```
<xsl:template match=' Knotentyp ' >  
    .... Schablone ....  
</xsl:template>
```

2.1 Konzeptueller Inhalt einer Transformationsregel

1. Parameter match: *Spezifikation, auf welche Knotentypen die Regel anwendbar ist*
 - einfachster und häufigster Fall: genau ein Knotentyp
 - i.a.: Spezifikation einer *mehrerer* Knotentypen
d.h. 1 Regel kann auf viele Knotentypen anwendbar sein
 - ferner: Einschränkung auf Knoten, die in einem bestimmten *Kontext* stehen
sehr ähnlich zu XPath-Ausdrücken, aber einige Abweichungen

2. Inhalt des `template`-Elements: Schablone / “Textmuster”, das für einen Knoten dieses Typs auszugeben ist
- enthält Ausgabe- und Steuerkommandos,
 - darf *keine inneren Transformationsregeln* enthalten!
Es gibt keine “lokalen” Transformationsregeln

2.2 Spezifikationen im Parameter match

2.2.1 Grundidee

Zweck des `match`-Parameters: Spezifikation einer Teilmenge der Knoten des Eingabebaums anhand folgender Selektionskriterien:

- Knotentyp (meist das einzige Kriterium!)
- “Einbettung” des Knotens in Eltern und Vorfahren
- weitere inhaltliche Kriterien → Boolescher Ausdruck über Attributwerte u.a. Daten

Lösungskonzept: **LocationPathPatterns**

- eingeschränkte Pfadausdrücke: laufen nur in Richtung Blätter
- ein Template matcht auf Knoten X im Eingabebaum, wenn es einen Startknoten Y im Eingabebaum gibt, so daß das LocationPathPattern ausgehend von Y den Knoten X findet

2.2.2 Beispiele

Dokumentwurzel: `match=' / '`

Elemente: `match=' Elementtypname '`

Attribute: `match=' @Attributname '`

Textabschnitte: `match=' text() '`

Kommentare: `match=' comment() '`

Processing Instr.: `match=' processing-instruction() '`

2.2.3 Spezifikation mehrerer Knotentypen mit *

für Elementtypen und Attribute: “Namensmuster”

identisch mit Namensmustern in Knotentest in Navigationsschritten eines XPath-Pfads! s. XPath-Spezifikation:

```
[37] NameTest ::= '*' | NCName ':' '*' | QName
```

Beispiele:

```
match=' person '
```

```
match=' * '
```

```
match=' einNamensraum:lokalerName '
```

```
match=' einNamensraum:* '
```

Vorsicht: mit dem `*` kann man keine beliebigen regulären Ausdrücke über Typnamen bilden!! Beispiel: `abc*xyz` ist falsch!

letztlich ist nur folgendes erlaubt:

`*` selektiert alle Elemente

`NR:*` selektiert alle Elemente, deren Typname in dem Namensraum liegt, der durch den Namensraumbezeichner `NR` angegeben wird

`@*` selektiert alle Attribute

`@NR:*` selektiert alle Attribute, deren Name in dem Namensraum liegt, der durch den Namensraumbezeichner `NR` angegeben wird

Details s. XML-Namespace-Spezifikation

- [4] `NCName ::= NCNameStartChar NCNameChar*`
/* An XML Name, minus the ":"
(Non-Colon-Name) */
- [5] `NCNameChar ::= NameChar - ':'`
- [6] `NCNameStartChar ::= Letter | '_'`
- [7] `QName ::= PrefixedName | UnprefixedName`
- [8] `PrefixedName ::= Prefix ':' LocalPart`
- [9] `UnprefixedName ::= LocalPart`
- [10] `Prefix ::= NCName`
- [11] `LocalPart ::= NCName`

2.2.4 Aufzählung mehrerer Knotentypen

Angabe durch Aufzählung mit Trennzeichen `|`:

Beispiel: `match=' Name | Vorname | @* | meinNR:* '`

allgemeine Syntax von Patterns = Inhalt des Parameters `match`:

```
[1] Pattern ::=
      LocationPathPattern
      | Pattern '|' LocationPathPattern
```

Reihenfolge der `LocationPathPattern` in einem `Pattern` ist unerheblich

2.2.5 Pfade in `LocationPathPatterns`

zusätzliche Bedingung: Knoten müssen Kind / Nachfahre anderer Knoten sein¹; Beispiele:

`ol/li` Regel ist nur anwendbar auf `li`-Elemente, deren Elternknoten ein `ol`-Element ist

`DURCHFUEHRUNG/@dozentId` Regel ist nur anwendbar auf `dozentId`-Attribute, deren Elternknoten ein `DURCHFUEHRUNG`-Element ist

¹Man kann diese Einbettung auch als “Kontext” bezeichnen, allerdings kann die irritieren, denn der Begriff Kontext wird in XPath und XSLT noch mit anderen Bedeutungen benutzt.

allgemeine Form von *LocationPathPatterns*:

- sehen zwar aus wie XPath-Ausdrücke (und sind immer als syntaktisch korrekte XPath-Ausdrücke interpretierbar), *sind aber tatsächlich keine XPath-Ausdrücke!!*
- sind “*im Syntaxbaum nach unten gehende*” Navigationen

Syntax von LocationPathPatterns:

- ```
[2] LocationPathPattern ::=
 '/' RelativePathPattern?
 | '//'? RelativePathPattern |

[4] RelativePathPattern ::=
 StepPattern
 | RelativePathPattern '/' StepPattern
 | RelativePathPattern '//'? StepPattern

[5] StepPattern ::=
 ChildOrAttributeAxisSpecifier NodeTest Predicate*

[6] ChildOrAttributeAxisSpecifier ::=
 AbbreviatedAxisSpecifier
 | ('child' | 'attribute') '::'

[13] AbbreviatedAxisSpecifier ::= '@'?
```

## Unterschiede zu XPath bei den Navigationsrichtungen:

- explizit nur Navigationsrichtungen `child` und `attribute`, implizit (in abgekürzter Notation) ist `descendant` erlaubt
- bei `child` und `attribute` Vorgabe / Kurzform wie üblich
- `//` entspricht effektiv dem XPath-Navigationsschritt  
`/ descendant::node() /`  
(inkl. Begrenzer-Schrägstriche; also NICHT wie bei XPath  
`/ descendant-or-self::node() / !!`)
- `//` ist hier keine Abkürzung, sondern einzige Notation, die expandierte Notation `descendant::XYZ` ist nicht erlaubt

## Unterschiede zu XPath beim Knotentypstest `node()`:

- XSLT-Spezifikation: “*node() matches any node other than an attribute node and the root node.*”
- XPath, 2.3 Node Tests: *A node test node() is true for any node of any type whatsoever.*
- Unterschied irritierend, aber inhaltlich gegenstandslos, denn der Knotentypstest `node()` ist nur in den Navigationsrichtungen `child` und `descendant` erlaubt und dort können Attribut-Knoten und die Dokumentwurzel nicht vorkommen

**Bedeutung eines `LocationPathPattern`  $LPP$ :**

Eine Transformationsregel für gegebenes  $LPP$  ist anwendbar auf einen Knoten  $N$ , wenn:

- im Eingabebaum existiert ein Startknoten  $S$
- so daß  $N$  in der Treffermenge von  $LPP$  ausgehend von  $S$  liegt

Alle bisherigen Beispiele sind Sonderfälle dieser allgemeinen Regel

Weitere Beispiele für `match`-Angaben:

`li [ 1 ]` oder

`li [ position()=1 ]`

selektiert alle `li`-Elemente, die das erste Kindelement vom Typ `li` ihres Elternelements sind

`* [ position()=1 and self::li ]`

selektiert alle `li`-Elemente, die das erste Kind ihres Elternelements sind

`li [ last()=2 ]`

selektiert alle `li`-Elemente, deren Elternelement zwei Kindelemente vom Typ `li` hat

## 2.2.6 Prioritäten von Transformationsregeln

Problem: auf einen bestimmten Knoten können *mehrere Transformationsregeln anwendbar* sein, z.B.:

```
<xsl:template match=" * " >
 <!-- allgemeine Regel fuer alle Elementtypen -->

</xsl:template>
```

```
<xsl:template match=" TelNr " >
 <!-- spezielle Regel fuer einen Elementtyp -->

</xsl:template>
```

Entscheidung anhand von **Prioritäten** – kompliziert (diverse Möglichkeiten zur expliziten Steuerung von Prioritäten), hier nur rudimentär dargestellt

Grundregel: *speziellere Regeln haben höhere Priorität*

Regel R1 ist **spezieller als** Regel R2

$\iff$  R1 für kleinere Menge von Knoten anwendbar

Beispiele:

```
match=" node() "
```

```
match=" * "
```

```
match=" li "
```

```
match=" ol/li "
```

```
match=" ol/li [position()=2]"
```

## 2.3 Verwaltung von Transformationsregeln

- ein **Transformationsdokument** enthält i.a. eine *Menge* (keine Folge) von Transformationsregeln
- die *Reihenfolge* der Transformationsregeln im Transformationsdokument ist bedeutungslos, nur Prioritäten entscheiden  
→ dieser Teil von XSLT ist eine nichtprozedurale Sprache<sup>2</sup>
- ggf. viele Regeln auf mehrere Dateien aufteilen, zusammenführen mit `xsl:include` oder `xsl:import`

---

<sup>2</sup>man könnte auch von einer regelbasierten Sprache reden, aber das würde zu einer falschen Assoziation mit Prolog führen

### 2.3.1 Kommando `xsl:include`

Gesamtmenge der Transformationsregeln kann auf mehrere Dateien verteilt werden, die dann von der Hauptdatei inkludiert werden (ermöglicht Wiederverwendung von Regeln).

Syntax:

```
<xsl:include href='uri-reference' />
```

- darf nur als Top-Level-Element (direktes Kind des `transform`-Elements) verwendet werden
- `uri-reference` muß korrekt sein und zu einer Datei führen
- *kein Unterschied* zwischen inkludierten und lokal definierten Regeln

### 2.3.2 Kommando `xsl:import`

i.w. identisch zu `xsl:include`, aber die importierten Regeln haben *geringere Priorität* als die lokal vorhandenen (im Sinne von Voreinstellungen)

Syntax:

```
<xsl:import href='uri-reference' />
```

## 3 Expliziter Aufruf von Transformationsregeln

### 3.1 Kommando `xsl:apply-templates`

Kommando `xsl:apply-templates`:

- Syntax: `<xsl:apply-templates select='...' />`
- tritt innerhalb von Schablonen (z.B. von Transformationsregeln oder `for-each`-Kommandos) auf
- ist ähnlich wie `for-each` ein Ablaufsteuerungskommando
- Parameter `select`: enthält Pfad, der die Menge der Knoten angibt, die bearbeitet werden sollen

Implementierung in Form von Pseudocode:

1. bestimme die Menge der zu bearbeitenden Knoten gemäß dem XPath-Ausdruck im Parameter `select`;  
bearbeite die Knoten in der Reihenfolge gemäß Eingabe<sup>3</sup>
2. für jeden zu bearbeitenden Knoten  $N$ :
  1. bestimme den Typ von  $N$
  2. bestimme die Transformationsregeln, die für Knoten dieses Typs anwendbar sind;  
wähle darunter *die mit der höchsten Priorität für  $N$*
  3. rufe die Schablone aus dieser Transformationsregel mit passendem Kontext (insb.  $N$  als *Kontextknoten*) auf

---

<sup>3</sup>sofern nicht zusätzliche Sortierparameter angegeben sind

## 3.2 Beispiel

Eingabedaten:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
```

```
<Telefonliste>
```

```
 <Eintrag PNr="p1" >
```

```
 <Name>Meier</Name>
```

```
 <TelNr Vorwahl="0271" >891234</TelNr>
```

```
 </Eintrag>
```

```
 <Eintrag PNr="p2" >
```

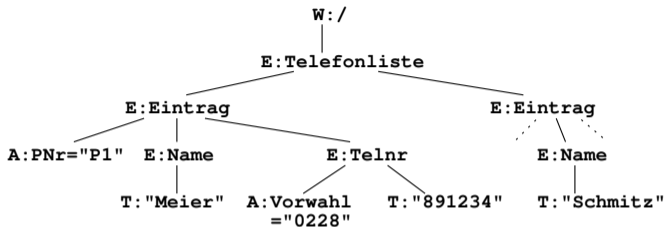
```
 <Name>Schmitz</Name>
```

```
 <TelNr Vorwahl="0228" >870887</TelNr>
```

```
 </Eintrag>
```

```
</Telefonliste>
```

zugehöriger Syntaxbaum (unvollständig):



Notation:

W: ... Knoten, der die Dokumentwurzel repräsentiert

E: ... Knoten, der ein Element repräsentiert

T: ... Knoten, der einen Text repräsentiert

A: ... Knoten, der eine Attributspezifikation repräsentiert

## Beispiel

Aufgabe: auch Vorwahl soll separates Unterelement werden, alles andere kopieren

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<xsl:transform version='1.0'
 xmlns:xsl='http://www.w3.org/1999/XSL/Transform' >

 <xsl:template match=' / ' >
 <xsl:apply-templates select='Telefonliste' />
 </xsl:template>
```

```
<xsl:template match=' Telefonliste ' >
 <Telefonliste>
 <xsl:apply-templates select='Eintrag' />
 </Telefonliste>
</xsl:template>
```

```
<xsl:template match=' Eintrag ' >
 <Eintrag>
 <xsl:apply-templates select='Name' />
 <xsl:apply-templates select='TelNr/@Vorwahl' />
 <xsl:apply-templates select='TelNr' />
 </Eintrag>
</xsl:template>
```

```
<xsl:template match=' Name ' >
 <Name>
 <xsl:value-of select='.' />
 </Name>
</xsl:template>
```

```
<xsl:template match=' TelNr ' >
 <TelNr>
 <xsl:apply-templates select='text()' />
 </TelNr>
</xsl:template>
```

```
<xsl:template match=' @Vorwahl ' >
 <Vorwahl>
 <xsl:value-of select='.' />
 </Vorwahl>
</xsl:template>
```

```
<xsl:template match=' text() ' >
 <xsl:value-of select='.' />
</xsl:template>
```

```
</xsl:transform>
```

## Erläuterungen / Beobachtungen:

1. die Dokumentwurzel des Ausgabebaums wird implizit erzeugt, sie kann nicht explizit erzeugt werden.

Die Transformationsregel mit `match=' / '` enthält daher *keine Anweisung, die den aktuell bearbeiten Knoten in der Ausgabe reproduziert* (im Gegensatz zu den Transformationsregeln für die Elemente)

2. die inneren Knoten des Eingabebaums müssen i.d.R. nur reproduziert (kopiert) werden;  
hierzu braucht man für *jeden Elementtyp* eine eigene Transformationsregel, die i.w. nach folgendem Schema aufgebaut ist:

```
<xsl:template match=' elementtyp ' >
 <elementtyp>
 <xsl:apply-templates select='Kinder' />
 </elementtyp>
</xsl:template>
```

- der Kontextknoten wird “manuell” in der Ausgabe rekonstruiert (durch öffnenden und schließenden Tag)
- die innerhalb dieser Tags liegenden Ausgabeanweisungen erzeugen Kinder dieses Knotens
- `<xsl:apply-templates select='Kinder' />` bearbeitet alle Kinder und ruft implizit die jeweils passende Transformationsregel auf

3. man kann mit `<xsl:apply-templates ... />` bis zu den Blättern des Eingabebaums herunterlaufen  
(Bsp.: Transformationsregeln für `TelNr` und `text()`)  
oder  
bei Elementen, die nur noch Kindknoten vom Typ Text haben, schon für diese Elemente eine Transformationsregel definieren, die `<xsl:value-of select='.' />` aufruft  
(Bsp.: Transformationsregel für `Name`)
4. viel monotone Schreiberei, um die Elementstrukturen zu rekonstruieren
5. Reihenfolge der Transformationsregeln hätte beliebig anders gewählt werden können

6. Attribut **PNr** ist verlorengegangen (Attribute mit Inhalt, der von der Eingabe abhängt, können wir mit dem bisher Erlernten nicht erzeugen)  
ebenso Kommentare

### 3.3 Vorgabewert für `select`

Der Parameter `select` im Kommando `xsl:apply-templates` kann weggelassen werden;

dann gültiger Vorgabewert: `child::node()`

m.a.W.:

```
<xsl:apply-templates />
```

ist äquivalent zu

```
<xsl:apply-templates select='child::node()' />
```

Beispiel: in der Transformationsregel für die Telefonliste hätte man `select` weglassen können

tückisch insofern, als hiermit *Attribute nicht selektiert* werden!!

**Testfrage [2 Minuten]:** erklären Sie den Unterschied zwischen

- `<xsl:template>` und
- `<xsl:apply-templates>`

Wo dürfen diese Elementtypen auftreten?

## Antwort:

- `<xsl:template>` definiert eine **Transformationsregel** für einen oder mehrere Knotentypen;  
darf nur als Kind eines `<xsl:transform>`-Knotens (also des Wurzelements eines Transformationsdokuments) auftreten
- `<xsl:apply-templates>` ist ein **Ablaufsteuerungskommando**, das angibt, daß auf die im Parameter `select` spezifizierten Knoten die Transformationsregel mit der jeweils höchsten Priorität angewandt werden soll  
darf nur in einer Schablone, also z.B. im “Rumpf” einer Transformationsregel, auftreten

## 4 Vordefinierte Transformationsregeln

Problem: Bei der Verarbeitung der Eingabe muß *für jeden Knotentyp wenigstens eine Transformationsregel definiert sein*, auch wenn das Transformationsdokument keine passende Regel definiert.

Beispiel: Text-Knoten

Lösung: Für alle Knotentypen (außer Namensraumknoten) sind Transformationsregeln vordefiniert; diese Regeln sind sehr allgemein und haben daher *geringste Priorität*;

übliche applikationsspezifische Regeln sind spezieller und haben höhere Priorität

Vordefinierte Transformationsregeln hängen vom Knotentyp ab:

1. für Elemente und die Dokumentwurzel:

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

bewirkt im Normalfall einen *rekursiven Abstieg durch den Baumstruktur* der Elemente (genauer: der Kindknoten; ohne Attribute!); Verarbeitung in der Eingabereihenfolge

## 2. für Textknoten und Attribute:

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

bewirkt Ausgabe des Inhalts als Text, *sofern* der Knoten überhaupt verarbeitet wird!

bei Attributen: nicht automatisch!

3. für Verarbeitungsanweisungen und Kommentare:

```
<xsl:template match=" processing-instruction() |
 comment() " />
```

werden *nicht ausgegeben*

**implizit vorhandenes voreingestelltes Hauptprogramm:**  
verarbeite Dokumentwurzel des Syntaxbaums gemäß zug. Transformationsregel:

```
<xsl:apply-templates select=' / ' />
```

**Testaufgabe:** Sei  $T$  ein Transformationsdokument, das keine einzige Transformationsregel enthält.

Beschreiben Sie die durch  $T$  definierte Transformation, also die Ausgabe, wenn Sie irgendeine XML-Datei mit  $T$  transformieren.

# 5 Die identische Transformation und Projektionen

## 5.1 Die identische Transformation

Ein Transformationsdokument, das nur die folgende Transformationsregel enthält, reproduziert den ganzen Eingabesyntaxbaum unverändert:

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

## Erläuterungen:

- `node()` | `@*` ist Kurzschreibweise von `attribute::*` | `child::node()`
- `node()` selektiert / ist anwendbar auf alle Knotentypen außer Attributen und der Dokumentwurzel
- `@*` selektiert / ist anwendbar auf alle Attribute
- Dokumentwurzel wird speziell behandelt - hier kein Thema

## 5.2 Kommando `xsl:copy`

Syntax:

```
<xsl:copy>
 Schablone
</xsl:copy>
```

- kopiert den aktuellen Kontextknoten in den Ausgabesyntaxbaum
- hat keinen `select`-Parameter
- der Inhalt von `xsl:copy` ist eine Schablone – diese wird ausgeführt und alle erzeugten Knoten werden an den kopierten Knoten als Kind angehängt

## 5.3 Projektionen

### “Entwurfsmuster” für Projektionen:

- identische Transformation als allgemeinste Regel definieren (lokal in der gleichen Datei oder in einem importierten oder inkludierten Transformationsdokument)
- für auszublendende Attribute oder Elemente (incl. darunterliegende Teilbäume!) Transformationsregeln mit leerer Schablone definieren

**Beispiel:** Lösche die TelNr-Elemente in der Telefonliste

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<xsl:transform version='1.0'
 xmlns:xsl='http://www.w3.org/1999/XSL/Transform' >

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

 <xsl:template match=" TelNr " />

</xsl:transform>
```