Entwickler Handbuch

XSLT Hands-On

XML - XSLT - XQuery

TEKTUR

(c) Alex Düsel 2019

Creative Commons Namensnennung-Keine Bearbeitungen 4.0 International Public License

www.github.com/alexdd/Buch

Version: 28

Thu Jan 17 2019 16:32:40 / de-DE

Dieses Buch wurde mit Tektur CCMS erstellt. Tektur ist ein einfach zu bedienender kollaborativer Editor um **DITA** [1] Inhalte erstellen, als PDF ausgeben und pflegen zu können. Die Eingabe erfolgt dabei per **WYSIWYG** [2] mit geführter Benutzerinteraktion. Die Inhalte werden als einzelne Topics verwaltet, die in verschiedenen Maps referenziert werden können; Stichwort: **Topic Based Authoring** [3].

Sonstige Features: Rechte- und Rollensystem, Versionskontrolle, konfigurierbarer Workflow mit Review & Approval Fuktionen. Auf dem Entwicklerblog[4] kann man sich über den Fortschritt informieren.

► *Dieses Buch ist **WORK IN PROGRESS** und dient in erster Linie als Test für Tektur CCMS. Der Feinschliff kommt noch! Momentan schreibe ich alles was mir irgendwie interessant erscheint - je nach zeitlicher Möglichkeit mehr oder weniger umfangreich - auf und binde die Topics in eine Map ein, um ein PDF erzeugen zu können.*

Wie schliesslich Struktur und Inhalt genau aussehen sollen, werde ich mir zu einem späteren Zeitpunkt noch genau überlegen :-]

1) https://de.wikipedia.org/wiki/Darwin_Information_Typing_Architecture
2) https://de.wikipedia.org/wiki/WYSIWYG
3) https://en.wikipedia.org/wiki/Topic-based_authoring
4) http://www.tekturcms.de

Inhalt

1 Anwendungsgebiete

XML, XSLT, **XPATH**, **XSL-FO** und **XQuery** sind Techniken um baumstrukturierte Daten - im Vergleich zu relationalen Daten - aus verschiedenen Quellen ineinander zu überführen, abzuspeichern, zu versenden, darzustellen und auszuwerten. Einfach gesprochen geht es um die Weiterverarbeitung solcher Daten - das XML kann dabei sowohl die Nutzdaten als auch die Steuerdaten tragen.

Vom Aussehen her sind XML Daten im Prinzip Textdaten. Sie können sehr einfach mit einem Texteditor erstellt werden. Im Gegensatz zu Multimedia-Daten sind keine komplexen Tools, wie z.B. ein Grafikeditor, erforderlich.

Auch relationale Daten können in Form von Tabellen, als Excel Tabelle oder bspw. als kommaseparierte Textdatei, aus einem System ausgespult und weiterverarbeitet werden. XML erlaubt es jedoch die Daten semantisch auszuzeichnen. Das geschieht durch das Klammern semantisch zusammengehöriger Elemente mittels Klammer-Tags und weiterer Kategorisierung dieser Informationseinheiten mittels weiterer Properties (Attribute) an diesen Tags. Durch das Verschachteln dieser geklammerten Komponenten entsteht ein Baum, der die Hierarchische Ordnung der Daten widerspiegeln sollte.

Diese Baumstrukturen sind maschinell lesbar und die Daten können, bevor sie von einem Versender zu einem Empfänger gehen, mittels eines automatischen Prozesses validiert werden. Dabei können sowohl der Inhalt als auch die Syntax anhand von definierten Regeln (Schemas) genau überprüft werden.

Der XML Standard ist mittlerweile 20 Jahre alt. Zuvor gab es SGML, das zum Beispiel auch nicht abgeschlossene Tags erlaubt.

Der Übergang von SGML zu XML hat die Sache ein bisschen vereinfacht, eine weitere Vereinfachung brachte JSON als Standard. JSON wird gerne im Webbereich eingesetzt um baumstrukturierte Daten zu verarbeiten. JSON ist jedoch nicht so gut maschinenlesbar und es gibt noch nicht so viele Werkzeuge wie z.B. Code Editoren dafür.

Folgend eine kurze Erläuterung zu den eingangs erwähnten Schlüsselwörtern, um die es sich im weiteren Text drehen wird:

- **XML** ist das Datenformat. Auf XML arbeiten die anderen Technologien. XML ist immer Input für diese Tools.
- **XSLT** transformiert eine XML Instanz in eine andere. Plain Text ist eine Aneinanderreihung von Text Knoten
- **XPATH** erlaubt es, bestimmte Knoten in einem XML Dokument über bedingte Pfadausdrücke zu selektieren.
- **XSL-FO** ist eine weitere XML basierte Auszeichnungssprache, die ein XSL-FO Prozessor einlesen kann, um daraus z.B. ein PDF zu generieren.
- **XQuery** ist eine Abfragesprache ähnlich zu SQL, jedoch werden damit nicht relationale Daten abgefragt sondern baumstrukturierte.

1.1 Einige aktuelle und vergangene Szenarien

Einige Beispiele - aktuell und aus vergangenen Tagen:

1 Anwendungsgebiete

XML Webseiten

Einen XSLT Prozessor hat jeder Browser eingebaut. Es gab mal eine Zeit, in der es sehr populär war, Webseiten vom Server als XML auszuliefern. XML erlaubt die semantische Auszeichung des Inhalts, und die strikte Trennung des Inhalts von Layout und Design. Wesentlich besser als dies mit HTML und CSS jemals möglich wäre.Ich denke, u.a. wegen des exzessiven Einsatzes von Javascript (auch inline), hat sich diese Idee nie vollständig durchgesetzt. Schliesslich wurde XHTML spezifiziert und jetzt gibt es HTML5.

Betrachten wir das folgende einfache XML Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
<title>Das ultimative Zwei-Kapitel Dokument</title>
  <chapter>
    <title>Kapitel 1</title>
    <intro>In Kapitel 1 wird kurz gesagt was Sache ist.</intro>
    <content>Um es kurz zu machen, wie der Hase läuft steht in Kapitel 2.</content>
  </chapter>
  <chapter>
    <title>Kapitel 2</title>
    <intro>Hier wird erklärt, wie der Hase läuft.</intro>
    <content>Im Prinzip ist es ganz einfach.</content>
  </chapter>
</document>
```

Ohne XSLT Stylesheet Zuweisung wird der Browser eine Datei mit diesem Inhalt als eingerücktes XML anzeigen - oder die Tags einfach ignorieren und den Textinhalt in einer Zeile darstellen.Fügt man eine Processing Instruction[5] am Anfang ein, wird ein XSLT Stylesheet vom Browser herangezogen und vor der Darstellung im Browser wird die so deklarierte XML Transformation ausgeführt:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="formatiermich.xsl" ?>
<document>
  <title>Das ultimative Zwei-Kapitel Dokument</title>
  <chapter>
[...]
```

Jetzt kann man das XML einfach im Browser öffnen und alles wird schön formatiert - je nachdem welche Regeln in formatiermich.xsl gesetzt sind - angezeigt:

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

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

  <xsl:template match="document">
  <body>
      <xsl:apply-templates/>
    </body>
```

5) https://de.wikipedia.org/wiki/Verarbeitungsanweisung

```
  </xsl:template>

  <xsl:template match="document/title">
    <h1>
      <xsl:apply-templates/>
    </h1>
  </xsl:template>

  <xsl:template match="chapter">
    <div class="chapter">
      <xsl:apply-templates/>
    </div>
  </xsl:template>

  <xsl:template match="chapter/title">
    <h2>
      <xsl:apply-templates/>
    </h2>
  </xsl:template>

  <xsl:template match="chapter/intro">
    <div class="intro">
      <i><xsl:apply-templates/></i>
    </div>
  </xsl:template>

  <xsl:template match="chapter/content">
    <p><xsl:apply-templates/></p>
  </xsl:template>
</xsl:stylesheet>
```

Die Processing Instruction hat keinen Einfluss auf den XML Inhalt und wird in einer anderen Eingabeverarbeitung nicht herangezogen.

Serverseitige Konvertierung

Auch eine serverseitige Konvertierung ist gebräuchlich. Ein Beispiel aus vergangenen Tagen - WAP-Seiten
aus [6] für unterschiedliche Handy-Modelle.

Damals hatten die Handys sehr unterschiedliche Displaygrößen. Handybrowser konnten noch nicht ausreichend Javascript und die Skalierung der WAP-Seite für das jeweilige Handy passierte nicht im Handy, sondern vor der Auslieferung auf der Serverseite. Dazu wurde eine XML Quelle mittels verschiedener XSLT Stylesheets in unterschiedliche WML WAP Repräsentationen transformiert.

So würde das Zwei-Kapitel Beispiel von oben im WML Format aussehen (recht einfach gehalten):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN" "http://www.wap.org/DTD/wml_1.1.xml">
<wml>
  <head>
    <meta name="title" content="Das ultimative Zwei-Kapitel Dokument"/>
  </head>
  <card id="chapter1" title="Kapitel 1">
    <p><i>In Kapitel 1 wird kurz gesagt was Sache ist.</i></p>
    <p>Um es kurz zu machen, wie der Hase läuft steht in Kapitel 2.</p>
  </card>
  <card id="chapter2" title="Kapitel 2">
    <p><i>Hier wird erklärt, wie der Hase läuft.</i></p>
```

6) https://de.wikipedia.org/wiki/Wireless_Application_Protocol

```
    <p>Im Prinzip ist es ganz einfach.</p>
  </card>
</wml>
```

Eine XSLT Transformation, die die XML Daten von oben in diese WML Darstellung überführt, könnte z.B. so implementiert werden:

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

  <xsl:output
    doctype-public="-//WAPFORUM//DTD WML 1.2//EN"
    doctype-system="http://www.wapforum.org/DTD/wml12.dtd"
    indent="yes"/>

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

  <xsl:template match="document/title">
    <head>
      <meta name="title">
        <xsl:attribute name="content">
          <xsl:value-of select="."/>
        </xsl:attribute>
      </meta>
    </head>
  </xsl:template>

  <xsl:template match="chapter">
    <card id="{concat('chapter',count(preceding-sibling::chapter)+1)}">
      <xsl:attribute name="title">
        <xsl:value-of select="title"/>
      </xsl:attribute>
      <xsl:apply-templates select="*[not(self::title)]"/>
    </card>
  </xsl:template>

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

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

  <xsl:template match="intro">
    <p><i><xsl:apply-templates/></i></p>
  </xsl:template>

  <xsl:template match="content">
    <p><xsl:apply-templates/></p>
  </xsl:template>
</xsl:stylesheet>
```

Ausgabeformate

Aus einer XML Quelle können auch leicht weitere Format erzeugt werden, bspw. ePub. EPUB
bspw.[7] ist das Standardformat für eBooks und neben Tags zur Formatierung für den Content, gibt es bspw. auch Anweisungen zum Erzeugen des Inhaltsverzeichnisses oder anderer Navigationsstrukturen.

Weitere gängige Formate sind neben dem oben gezeigten veralteten WML Format, elektronische Ausgabe-Formate wie: CHM[8], EclipseHelp[9], JavaHelp[10], ..., Print-Ausgabe Formate, wie PDF oder Adobe Framemaker[11], oder XML Standard Austauschformate, wie DITA, S1000D[12], PI-MOD[13], JATS[14] oder TEI[15].

Menschenlesbar machen

Kryptische XML Log-, Daten- oder Konfigurationsfiles können leicht mit XSLT menschenlesbar formatiert werdenEin Arbeitskollege im neuen Job kam kürzlich auf mich zu, ob ich um eine Möglichkeit wüsste, wie man sein kryptisches Datenfile für einen Übersetzungsdienst menschenlesbar formatieren könnte - XSLT to the Rescue:
mit XSLT menschenlesbar formatiert werdenEin Arbeitskollege im neuen Job kam kürzlich auf mich zu, ob ich um eine Möglichkeit wüsste, wie
man sein kryptisches Datenfile für einen Übersetzungsdienst

```
<?xml version="1.0" encoding="UTF-8"?>?<?xml-stylesheet type="text/xsl" href="de.xsl"?>
<jcr:root xmlns:sling="http://sling.apache.org/jcr/sling/1.0"
  xmlns:jcr="http://www.jcp.org/jcr/1.0"
  xmlns:mix="http://www.jcp.org/jcr/mix/1.0"
  xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
  jcr:language="de"
  jcr:mixinTypes="[mix:language]"
  jcr:primaryType="sling:Folder">
<b_manual
  jcr:primaryType="sling:MessageEntry"
  sling:message="Bedienungsanleitung"/>
  <b_warning
    jcr:primaryType="sling:MessageEntry"
    sling:message="Warnung"/>
  <b_danger
    jcr:primaryType="sling:MessageEntry"
    sling:message="Vorsicht"/>
  <b_note
    jcr:primaryType="sling:MessageEntry"
    sling:message="Notiz"/>
  <b_notice
    jcr:primaryType="sling:MessageEntry"
    sling:message="Hinweis"/>
  [...]
```

Mit einem eingehängten XSLT Stylesheet de.xsl wird so ein XML Datenfile schön formatiert als Tabelle angezeigt:

7) https://de.wikipedia.org/wiki/EPUB
8) https://de.wikipedia.org/wiki/CHM_(Dateiformat)
9) https://www.ibm.com/developerworks/library/os-echelp/index.html
10) https://en.wikipedia.org/wiki/JavaHelp
11) https://de.wikipedia.org/wiki/FrameMaker
12) https://de.wikipedia.org/wiki/S1000D
13) https://www.i4icm.de/forschungstransfer/pi-mod/
14) https://de.wikipedia.org/wiki/Journal_Article_Tag_Suite
15) https://de.wikipedia.org/wiki/Text_Encoding_Initiative

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:jcr="http://www.jcp.org/jcr/1.0"
  xmlns:sling="http://sling.apache.org/jcr/sling/1.0">

  <xsl:template match="jcr:root">
    <html>
      <table border="1" cellpadding="5" cellspacing="5">
        <xsl:apply-templates/>
      </table>
    </html>
  </xsl:template>

  <xsl:template match="*">
    <tr>
      <td>
        <xsl:value-of select="concat(count(preceding::*[@sling:message]) + 1,'.')"/>
      </td>
      <td>
        <xsl:value-of select="name()"/>
      </td>
      <td contenteditable="true">
        <xsl:value-of select="@sling:message"/>
      </td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

Hängt man an dieses Beispiel noch ein bisschen Javascript Logik und macht die Felder für die Übersetzungen mittels des HTML5 contenteditable Attributs editierbar, dann bräuchte man nur noch eine Rücktransformation HTML nach XML und hätte schon einen kleinen webbasierten XML Editor gebaut. Genau nach diesem Prinzip funktionieren einige aktuelle XML Editoren.

Diagramme

Nachdem eine SVG Grafik im XML Format vorliegt kann diese auch direkt aus XML Daten mittels XSLT erzeugt werden.Über das HTML5 <svg> Element kann so eine erzeugte Grafik inline in das ebenfalls durch das XSLT generierte HTML Dokument eingebunden werden. XSLT erzeugt werden.Über das HTML5 <svg> Element kann so eine erzeugte Grafik inline in das ebenfalls durch das XSLT generierte HTML

Betrachten wir unser Beispiel von oben, erweitert um drei neue <block> Elemente:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="chart.xsl" ?>
<document>
  <title>Das ultimative Zwei-Kapitel Dokument</title>
  <chapter>
    <title>Kapitel 1</title>
    <intro>In Kapitel 1 wird kurz gesagt was Sache ist.</intro>
    <content>Um es kurz zu machen, wie der Hase läuft steht in Kapitel 2.</content>
  </chapter>
  <chapter>
    <title>Kapitel 2</title>
    <intro>Hier wird erklärt, wie der Hase läuft.</intro>
    <content>Im Prinzip ist es ganz einfach. Betrachten wir doch drei gelbe Blöcke:
    </content>
    <block/>
```

```
    <block/>
    <block/>
  </chapter>
</document>
```

Wenn wir das XSLT Stylesheet von oben noch um eine Regel für das neue <block> Element ergänzen, so wie hier:

```
<xsl:template match="block">
   <svg style="background-color:yellow" width="30" height="30"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns="http://www.w3.org/2000/svg"/>
  <br/>
  <br/>
</xsl:template>
```

Dann erhalten wir drei schön formatierte gelbe SVG Blöcke.

Weiterführende Links:

- Client-side image generation with SVG and XSLT[16)]
- Knotentyp Visualisierung im Apache Jack Rabbit Projekt[17)]

1.2 Professionelle XML Verarbeitung

Vom Single-Source Publishing bis zur Generierung von Java Code aus Klassendiagrammen.

Single Source Publishing

Gängige Formate in der Technischen Dokumentation sind elektronische Ausgabe-Formate wie: CHM, EclipseHelp, JavaHelp, ePub, ..., Print-Ausgabe Formate, wie PDF oder Adobe Framemaker, oder XML Standard Austauschformate, wie DITA, S1000D, PI-MOD oder TEI. Vorteile:

- Bei einer Änderung in der XML Quelle werden auch automatisch alle anschließenden Formate aktualisiert.
- Strikte Trennung von Content (/ Semantik) und Layout/Design.
- Auf der XML Quelle sind XML Features möglich, wie: Modularisierung: Erlaubt die feingranulare Wiederverwendung von Content-Bausteinen, sowie das Verlinken, Filtern, Suchen und Exportieren derselben. Generalisierung ist ein DITA Konzept, welches die Wiederverendung von angepassten Topics in anderen DITA Systemen ermöglicht. Gültigkeiten erlauben die bedingte Anwendung von Content-Bestandteilen auf Satz und Wort-Ebene. Versionierung und Diffing - Vergleich von Änderungen zwischen Versionen Intelligente Querverweise: Ein Link zwischen einzelnen XML Topics bleibt versionstreu. Automatischer Satz, inkl. Zusamenhalte- und Trennregeln für Seiten, Absätze und Blöcke (Listen, Tabellen, etc).
- Veraltete Formate können ausgetauscht werden, ohne dass der Content geändert werden muss oder verlorengeht.
- Die XML Quelle kann ohne Aufbereitung in anderen Systemen wiederverwendet werden.

16) http://surguy.net/articles/client-side-svg.xml
17) http://jackrabbit.apache.org/jcr/node-type-visualization.html

- Es gibt weit verbreitete Standards zur Struktur der XML Quelle.
- Nur das XML muss in der Datenhaltung persistiert werden
- Es gibt spezialisierte XML Datenbanken, die besonders gut auf Baumstrukturen arbeiten. (Dokumente sind per se baum-strukturiert und sind eigentlich für eine relationale Datenbank ungeeignet)

Die Redaktionssysteme der Technischen Dokumentation der führenden Hersteller in Deutschland haben XML unter der Haube und setzen auf die Single-Source-Multi-Channel Strategie.

Code Generierung

Nachdem man bei XSLT im Format der Ausgabe frei ist, kann auch direkt Plain-Text mit XSLT Regeln generiert werden. Daher liegt es nahe sich jegliche Form von Quelltext aus einer XML Repräsentation erzeugen zu lassen.

Beispielsweise speichern gängige CASE Tools (Computer Aided Software Engineering) UML Diagramme im XML Format ab, so z.B. **ArgoUML** [18].

Diese Klassendiagramme lassen sich mittels XSLT direkt in Java-Code transformieren, wie z.B. in einem kleinen Open Source Projekt (aus vergangenen Tagen) : **Butterfly Code Generator** [19]

Es gibt aber auch einen schönen Artikel dazu im Java World Journal[20].

Migrationen und Konvertierungen

Für jede erdenkliche Art der Migration eines XML Datenbestands oder eines Datenbank-Dumps / -Exports im XML Format, zwischen Produktversionen oder zwischen Dienstleister- und Dienstnutzer-Systemen bietet sich XSLT als Mittel der Wahl zur schnellen und komplexen Transformation an.

Dabei ist zu beachten, dass XSLT besonders schnell und gut auf verschachtelten Strukturen arbeitet. Entartet ein Baum zur Liste und/oder sind nur geringe Strukturanpassungen notwendig, wird man sich mit einem schnellen SAX Parser leichter tun. Mittels der XSLT3.0 Streaming Option können auch sehr große XML Quellen (Big Data) verarbeitet werden. Saxon bietet bspw. diese Streaming Option[21].

18) http://argouml.tigris.org
19) http://butterflycode.sourceforge.net
20) https://www.javaworld.com/article/2073998/java-web-development/generate-javabean-classes-dynamically-with-xslt.html
21) http://www.saxonica.com/html/documentation/sourcedocs/streaming/

2 Wichtige Konzepte

XSL Stylesheets finden hauptsächlich Anwendung in der Technischen Dokumentation. Beispielsweise werden die Autohandbücher führender Hersteller mittels XSL gesetzt, deren Eingabedaten aufbereitet und zur Weiterverarbeitung transformiert.

- XSLT hat gerade noch den Status "Programmiersprache", weil man damit eine Turing Maschine[22)] programmieren kann.
- Mit HTML oder einer Templater Sprache (z.B. JSP) würde das nicht funktionieren.
- XSLT benötigt normalerweise immer eine XML Eingabe. Zumindest ein XML Knoten muss verarbeitet werden. Es gibt aber auch den Spezialfall, dass bspw. mit dem XSLT Prozessor Saxon keine Eingabe notwendig ist, bzw. diese vom XSLT Stylesheet selbst erzeugt wird.
- XSLT ist keine imperative Sprache, d.h es werden keine Anweisungen der Reihe nach abgearbeitet, sondern eine deklarative Sprache, d.h für jedes Ereignis (besser gesagt: für jeden durchlaufenen DOM Knoten) wird eine gefundene - und vom Programmierer deklarierte - Regel angewendet.
- Ausserdem gibt es funktionale Anteile, um bspw. die deklarierten Regeln rekursiv anwenden zu können.
- XSLT wird oft mit XSL gleichgesetzt. Aber XSL[23) 24)] ist mehr:
  - Zum einen kommt noch XPATH hinzu: XPATH erlaubt komplizierte Berechnungen und Selektionen auf den DOM Knoten eines XML Dokuments.
  - Zum anderen ist auch XSL-FO Bestandteil der XSL Spezifikation. XSL-FO Tags sind Anweisungen für einen XSL-FO Prozessor, der aus einem XSL-FO Dokument ein PDF Dokument generiert. Es sind auch andere Ausgabe-Formate, wie bspw. RTF möglich.

2.1 Push vs. Pull Stylesheets

XSLT ist eine ereignisgesteuerte, regelbasierte Umgebung zur Konvertierung von XML Daten. Gerade der Vorteil des regelbasierten Ansatzes ist vielen Entwicklern nicht bewusst, und es entsteht Quellcode der aussieht, wie mit XPath angereicherter PHP Code.

Ich frage mich an dieser Stelle immer, wieso nimmt man dann überhaupt XSLT, wenn man keine Template-Match Regeln verwendet, oder nur spärlich verwendet?

Um diesen Umstand aufzuklären ist ein bisschen Theorie notwendig:

22) http://www.unidex.com/turing/utm.htm
23) https://de.wikipedia.org/wiki/XSL_Transformation
24) W3C Seiten zu The Extensible Stylesheet Language Family (XSL)

2 Wichtige Konzepte

Beim "Pull" werden Elemente in der Quellinstanz selektiert und an einer passenden Stelle in der Zielinstanz eingefügt. Diese Vorgehensweise ist vergleichbar mit derer von Template-Engines, wie JSP oder ASP. Das kann in mehreren Stufen erfolgen, bis schrittweise die Quellinstanz in die finale Zielinstanz überführt wurde.

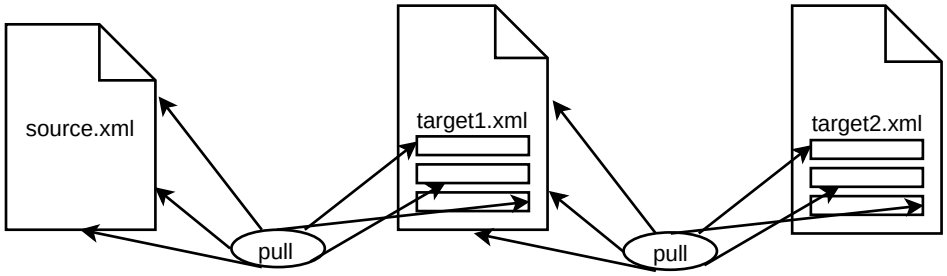

Bild 1: Pull Stylesheet

Beim "Push" werden die Quelldaten schrittweise in die Zieldaten konvertiert. Diese Vorgehensweise kann explorativ erfolgen und beim Transformieren in einen Zwischenschritt entstehen Erkenntnisse, die bei der Weiterverarbeitung nützlich sind. Merke: XSLT steht für eXtensible Stylesheet Transformation.

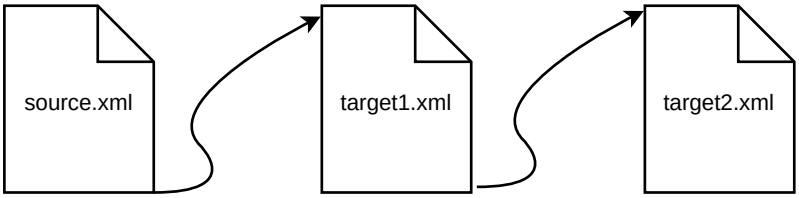

Bild 2: Push Stylesheet

Das bisher Gesagte verdeutlicht zwar den "Pull" Ansatz, was genau aber ge"pusht" wird, ist vermutlich noch unklar. Betrachten wir XML in der Baumdarstellung.

Der XSLT Prozessor unternimmt einen Tiefensuchlauf und überprüft bei jedem Knoten den er betritt, ob in seiner Regelbasis eine Regel existiert, die auf diesen Knoten "matched". Dabei gibt es drei grundsätzliche Möglichkeiten, wie die Knoten des Quellbaums in den Zielbaum kopiert - oder eben nicht kopiert - werden können.

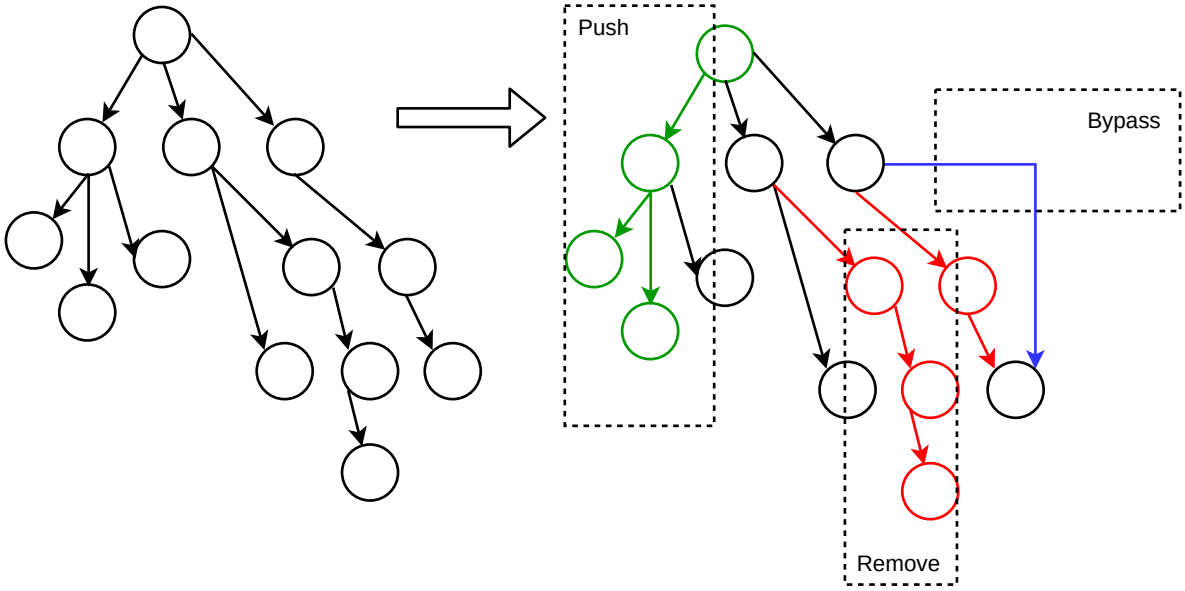

Bild 3: Transformation des Quellbaums in den Zielbaum

Remove ... Beim Betreten einer leeren Match-Regel wird ein Teilbaum nicht kopiert

Push ... Durch gezielte Auswahl einer Knotenmenge auf der Descendant-Achse wird der XSLT Prozessor in eine bestimmte Richtung ge"pusht".

Bypass ... Beim Betreten einer Match-Regel ohne Kopieranweisung wird der Knoten übersprungen

Pull-Stylesheets werden gewöhlich mit **for-each** Loops programmiert. Dieser Ansatz ist meiner Ansicht nach gebräuchlich, wenn keine großen DTD Änderungen zu erwarten sind, der XML Baum flach strukturiert ist und die Anforderungen an die Knvertierung relativ einfach sind, bspw. beim Auswerten / Konvertieren von Konfigurationsdateien. In alles anderen Fällen sind Push-Stylesheets vorzuziehen, d.h. möglichst wenige **for-each** loops und möglichst viele **Template-Match** Regeln.

2.2 Eindeutigkeit der Regelbasis

Die Regelbasis der XSLT ereignisgesteuerten Laufzeitumgebung (Wir erinnern uns: Jedes Mal wenn der XSLT Prozessor einen Knoten betritt, wird ein Event ausgelöst) kann unendlich viele Regeln aufnehmen. Für die Vollständigkeit, Eindeutigkeit und Konsistenz der Regelbasis ist der Programmierer selbst verantwortlich.

Um die Eindeutigkeit der Regeln zu gewährleisten, gibt es verschiedene Mechanismen.

2 Wichtige Konzepte

Reihenfolge der Match-Regeln

Im Normalfall sollte auf einen bestimmten Knoten in einem bestimmten Szenario genau eine Regel matchen. Falls es einen Konflikt gibt, wird zumindest bei Saxon diejenige Regel herangezogen, die im Stylesheet zuletzt deklariert wurde.

Diesen Umstand zu kennen, ist genau dann wichtig, wenn man einen bestehenden Stylesheet-Code übernehmen muss. Getreu dem Motto "Never change a running system" sollte man die Sache diesbzgl. sehr behutsam aufräumen.

Präzedenz der Auswertung

Match-Regeln werden gemäß ihrer Spezifität sortiert und diejenige, die auf einem Knoten in einem bestimmten Szenario am besten zutrifft wird zur Auswertung herangezogen. Grds. werden die Regeln anhand folgender Kriterien sortiert:

1. Importierte Template Regeln haben immer eine niedrigere Priorität als die Regeln des importierenden Stylesheets.
2. Templates mit einem höheren Priority Attribut haben Vorrang.
3. Templates ohne Priorität bekommen automatisch eine Default-Priorität. Die höchste Default-Priorität ist 0.5.
4. Diese Default Priorität errechnet sich anhand der Bedingungen oder Wildcards, die an einen Match-Regel geknüpft sind:
   - Wenn mehrere Templates matchen, dann wird das am meisten spezifische zur Auswertung herangezogen.
   - Das am meisten spezifische Template wird anhand der Prioritäten berechnet.
   - Einfache Elementnamen (z.B. "para") haben Prio 0.
   - Wildcards (z.B. *, @*) haben Priorität -0.25
   - Knoten-Tests für andere Knoten (e.g. comment(), node(), etc.) haben Priorität -0.5
   - In allen anderen Fällen ist die Prio 0.5.

   Beispiele:
   - para -> 0
   - h:* -> -0.25
   - * -> -0.25
   - node() -> -0.25
   - contents/para -> 0.5
   - contents/* -> 0.5
5. Mit einer Kommandozeilen-Option kann bei Saxon festgelegt werden, dass die Transformation abbricht, sobald es einen Konflikt bei der Regelauswertung gibt.

Import Präzendenz und Default-Regel

Wie in der obigen Sektion unter Punkt 1. angegeben, haben alle Regeln in einem importierten Stylesheet eine geringere Priorität als im importierenden Stylesheet. Diesen Umstand kann man sich zunutze machen, um eine Default-Regel einzubinden, bspw:

```
<xsl:template match="*" mode="#all"/>
```

Da sie sich in einem importierten Stylesheet befindet, hat sie geringere Priorität als alle anderen Regeln und greift nur dann, wenn für einen betretenen Knoten keine andere Match-Regel definiert ist.

Das ist z.B. praktisch, um nicht "gehandelte" Element zu identifizieren - dazu wäre die obige Regel nicht leer, sondern würde bspw. einen gelb markierten Warntext direkt in das Ausgabeformat schreiben.

Eine leere Default-Regel ist dagegen gut, wenn bspw. in einer XML-2-XML Migration automatisch Knoten im XML Baum abgetrennt werden sollen, für die keine Match-Regel existiert.

Prioritäten

Wie oben schon verdeutlicht werden alle Match-Regeln mit einer Priorität ausgestattet. Der Stylesheet-Entwickler hat die Möglichkeit diese Priorität zu überschreiben. Dazu wird das Attribut @priority an der Match-Regel verwendet. Ein Use-Case für die Prioritäten wÃ¤re bspw. folgendes Szenario:

- Die Eingabeinstanz soll in einer Vorprozessierung gefiltert werden.
- Dabei sollen Seminar-Elemente markiert werden, die nicht besonderen Bedingungen entsprechen:
  - Das Seminar-Element hat ein Feld "Ende-Datum" das abgelaufen ist.
  - Am Seminar-Element sind mehrere Dozenten angestellt, obwohl das Seminar-Element vom Type "Single" ist.
  - em Seminar-Element ist kein Dozent zugeordnet.
- Sicherlich kann es Seminar-Elemente geben, die alle drei Bedingungen erfüllen. Um das Error-Log aber nicht zu überfüllen, sollen die Filter nach ihren Prioritäten ausgeführt werden.

In Templates überführt, könnte diese Anforderung so umgesetzt werden:

```
<xsl:template match="Seminar[Ende-Datum/xs:date(.) le current-date()]"
              priority="30" mode="filter-network">
  <xsl:element name="Filtered-Seminar" namespace="{namespace-uri()}">
    <xsl:attribute name="reason">termed-seminar</xsl:attribute>
    <xsl:apply-templates select="node()|@*" mode="filter-network"/>
  </xsl:element>
</xsl:template>

<xsl:template match="Seminar[Type eq 'SINGLE' and count(dozenten/dozent) gt 1]"
              priority="20" mode="filter-network">
  <xsl:element name="filtered-Seminar" namespace="{namespace-uri()}">
    <xsl:attribute name="reason">dozenten-count</xsl:attribute>
    <xsl:apply-templates select="node()|@*" mode="filter-network"/>
  </xsl:element>
</xsl:template>

<xsl:template match="Seminar[not(dozenten/dozent)]" mode="filter-network">
  <xsl:element name="filtered-Seminar" namespace="{namespace-uri()}">
    <xsl:attribute name="reason">dozenten-missing</xsl:attribute>
    <xsl:apply-templates select="node()|@*" mode="filter-network"/>
  </xsl:element>
</xsl:template>
```

Modus Attribute

An allen Templates hat man die Möglichkeit einen selbst deklarierten Modus anzugeben. Wenn dann der XSLT Prozessor in eine bestimmte Richtung gepusht, vgl. *Push vs. Pull Stylesheets auf Seite 15*, wird, werden nur diejenigen Regeln zur Auswertung herangezogen, die im selben Modus sind, wie der apply-templates Call.

Beispielsweise möchte man die Titel im Kapitel anders behandeln als die Kapitel im Inhaltsverzeichnis, denn im TOC sollen z.B. keine Fussnoten-Marker angezeigt werden.

In Templates formuliert würde diese Anweisung folgendermassen aussehen:

```
<xsl:template match="title" mode="toc">
  <div class="toc-entry">
    <xsl:apply-templates select="*[not(self::footnote)]"/>
  </div>
</xsl:template>

<xsl:template match="title">
  <h1>
    <xsl:apply-templates/>
  </h1>
</xsl:template>
```

Die Generierung des TOC könnte dann folgendermassen ablaufen:

```
<xsl:for-each select="chapter">
  <xsl:apply-templates select="title" mode="toc">
</xsl:for-each>
```

Bzgl. der Eindeutigkeit der Regelbasis kann man also auch noch anhand des Mode-Attributes Ausführungs-Gruppen bilden. Wie auch bei Angabe der Priorities kann man auf diese Weise Regeln setzen, die nie ausgeführt wurden, weil sie vllt. im Zuge einer Refactoring-Massnahme abgeklemmt und dann vergessen wurden.

Auch das mode-Attribut ist also mit Vorsicht zu geniessen und sparsam einzusetzen.

3 Ausgewählte Themen

Hands-on Materiel mit Beispielquelltexten.

3.1 Vortransformationen

Bei einer komplexen Transformation ist es ratsam und sogar manchmal unabdingbar die Konvertierung in einzelne Stufen aufzuteilen. Das hat folgende Vorteile:

- Der Prozess ist transparenter, da die einzelnen Stufen leichter überschaubar sind.
- Die Zwischenergebnisse können für Debug-Zwecke ausgewertet werde oder dienen als Eingabe für andere Prozesse.
- Nicht-relevante oder invalide Teilbäume können aus der Eingabeinstanz gefiltert werden, um so die weitere Verarbeitung zu beschleunigen.
- Hilfskonstrukte können erzeugt werden. Diese erleichtern die weitere Verarbeitung.

Es gibt zwei Möglichkeiten, wie eine Vortransformation eingebunden werden kann:

- In einem separaten File bzw. einer XML Instanz, die vom XSLT Prozessor vor der eigentlichen Transformation aufgerufen wird uns einen Zwischenstand produziert, der als Eingabe für den Haupttransformationsschritt dient.
- Innerhalb des eigentlichen XSLT Stylesheets. Hier wird das Ergebnis der Vortransformation in einer Variablen erzeugt.

Den zweiten Punkt möchte ich anhand einer Beispiel XSLT Skripts vorführen. Betrachten wir folgende Input Daten:

```
<education-system>
  <administrative-regions>
    [...]
    <dministrative-region id="31" name="Bavaria">
      <shools>
        <school id="45">
          <teachers>
            <teacher id="576"/>
            <teacher id="345"/>
            <teacher id="12"/>
          </teachers>
        </school>
        <school id="36">
          <teachers>
            <teacher id="576"/>
            <teacher id="8"/>
          </teachers>
        </school>
        [...]
      </shools>
    </dministrative-region>
    [...]
  </administrative-regions>
</education-system>
```

Die erste Datei beinhaltet eine Zuordnung von Lehrern zu Schulen in verschiedenen Regierungsbezirken. Um die Daten zu den beiden referenzierten Objekten einzusehen müssen zwei weitere Dateien konsultiert werden. Die Datei, welche die Lehrer auflistet:

```
<teachers>
  [...]
  <teacher id="576">
    <first-name>Alfons</first-name>
    <last-name>Blimetsrieder</last-name>
    <subjects>
      <subject>Biology</subject>
      <subject>Math</subject>
      <subject>Sport</subject>
    </subjects>
    <suspended>2017-12-31</suspended>
    [...]
  </teacher>
  [...]
</teachers>
```

Und die Datei, welche die Schulen auflistet:

```
<schools>
  [...]
  <school id="45">
    <name>Gymnasium Bad Aibling</name>
    <type>Oberschule</type>
    [...]
  </school>
  [...]
</schools>
```

Um diese Daten verarbeiten zu können ist es sinnvoll, die drei Dateien in einem ersten "Resolver" Schritt zusammenzuführen und ggf. irrelevante Strukturen zu entfernen. Lehrer aus obigem Beispiel können beispielsweise suspendiert worden sein. Das folgende Skript erledigt dies mittels einer zusätzlichen Transformation in eine Variable:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  exclude-result-prefixes="#all">

  <xsl:output indent="yes" method="xml"/>

  <xsl:strip-space elements="*"/>

  <xsl:param name="file-1" required="yes"/>
  <xsl:param name="file-2" required="yes"/>
  <xsl:param name="file-3" required="yes"/>

  <xsl:variable name="files" select="(doc($file-1), doc($file-2), doc($file-3))"/>
  <xsl:variable name="bavaria-region-ids" select="(31, 58)"/>

  <xsl:key name="teachers" match="teacher" use="@id"/>
  <xsl:key name="schools" match="school" use="@id"/>

  <xsl:template name="main">
```

```
    <xsl:variable name="resolve-result">
      <xsl:apply-templates select="$files/administrative-regions" mode="resolve"/>
    </xsl:variable>
    <xsl:apply-templates select="$resolve-result/administrative-regions"/>
  </xsl:template>

  <xsl:template match="administrative-region[not(@id = $bavaria-region-ids)]"
                mode="resolve"/>

  <xsl:template match="school" mode="resolve">
    <xsl:copy>
      <xsl:copy-of select="key('schools',@id, $files/schools[1]/root())/node()"/>
      <xsl:apply-templates select="node()|@*" mode="resolve"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="teacher" mode="resolve">
    <xsl:copy-of select="key('teachers',@id, $files/teachers[1]/root())/node()"/>
  </xsl:template>

  <xsl:template match="teacher[suspended/xs:date(.) le current-date()]"/>

  <xsl:template match="node()|@*" mode="#all">
    <xsl:copy>
      <xsl:apply-templates mode="#current"/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Im ersten Resolve-Schritt werden die Referenzen zu den Lehrer- und Schul-Objekten aufgelöst, d.h. die Attribute des Schul-Objekts werden in die Struktur aus der ersten Datei kopiert. Die Liste der Lehrer an diesen Schul-Objekten bleibt erhalten und wird mit dem Inhalt aus der zweiten Datei bestückt.

Zusätzlich werden alle Regierungsbezirke entfernt, die nicht zu Bayern gehören, was die weitere Verarbeitung wesentlich beschleunigen wird. Lehrer die suspendiert worden sind fliegen ebenfalls raus.

3.2 Validierung mit Schematron

Um die Korrektheit einer XML Instanz zu prüfen, gib es verschiedene Schemata, wie XSD, RNG oder DTD, welche der Parser beim Aufbau des DOM Baums heranzieht. Eine Validierung mit Apache Xerces könnte beispielsweise als Java Code folgendermaßen angestossen werden:

```
URL schemaFile = new URL("http://host:port/filename.xsd");
Source xmlFile = new StreamSource(new File("web.xml"));
SchemaFactory schemaFactory = SchemaFactory
    .newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
try {
  Schema schema = schemaFactory.newSchema(schemaFile);
  Validator validator = schema.newValidator();
  validator.validate(xmlFile);
  System.out.println(xmlFile.getSystemId() + " is valid");
} catch (SAXException e) {
  System.out.println(xmlFile.getSystemId() + " is NOT valid reason:" + e);
```

```
} catch (IOException e) {
```

Schema Dateien können aber auch in XML Editoren eingebunden werden, um schon während der Eingabe der XML Instanz die Korrektheit zu überprüfen.

Das geht einerseits über die Angabe des Doctypes in der XML Instanz, anderseits bieten auch alle Editoren die Möglichkeit ein bestimmtes Schema explizit auszuwählen, um gegen dieses auf Anforderung zu validieren.

Gilt es komplexere Businessregeln zu überprüfen, die über Syntax-, Konsistenz- und einfache Korrektheitschecks hinausgehen, empfiehlt sich eine Validierung mit Schematron Regeln.

Schematron ist XSLT

Bei einer Schematron Validierung wird eine XML Instanz mit Hilfe eines automatisch generierten XSLT Stylesheets überprüft. Dieses kontextabhängige Stylesheet wird aus einer in der Schematron Syntax vom Autor verfassten Regelbasis, die wiederum in XML vorliegt, über ein zweites XSLT Stylesheet generiert - Dieses zweite XSLT Stylesheet ist sozusagen das eigentliche Schematron Programm.

Das folgende Diagramm veranschaulicht die Vorgehensweise anhand eines Filter-Szenarios, bei dem ein XML Dokument mit einigen ungültigen Passagen in eine gefilterte Darstellung überführt wird.

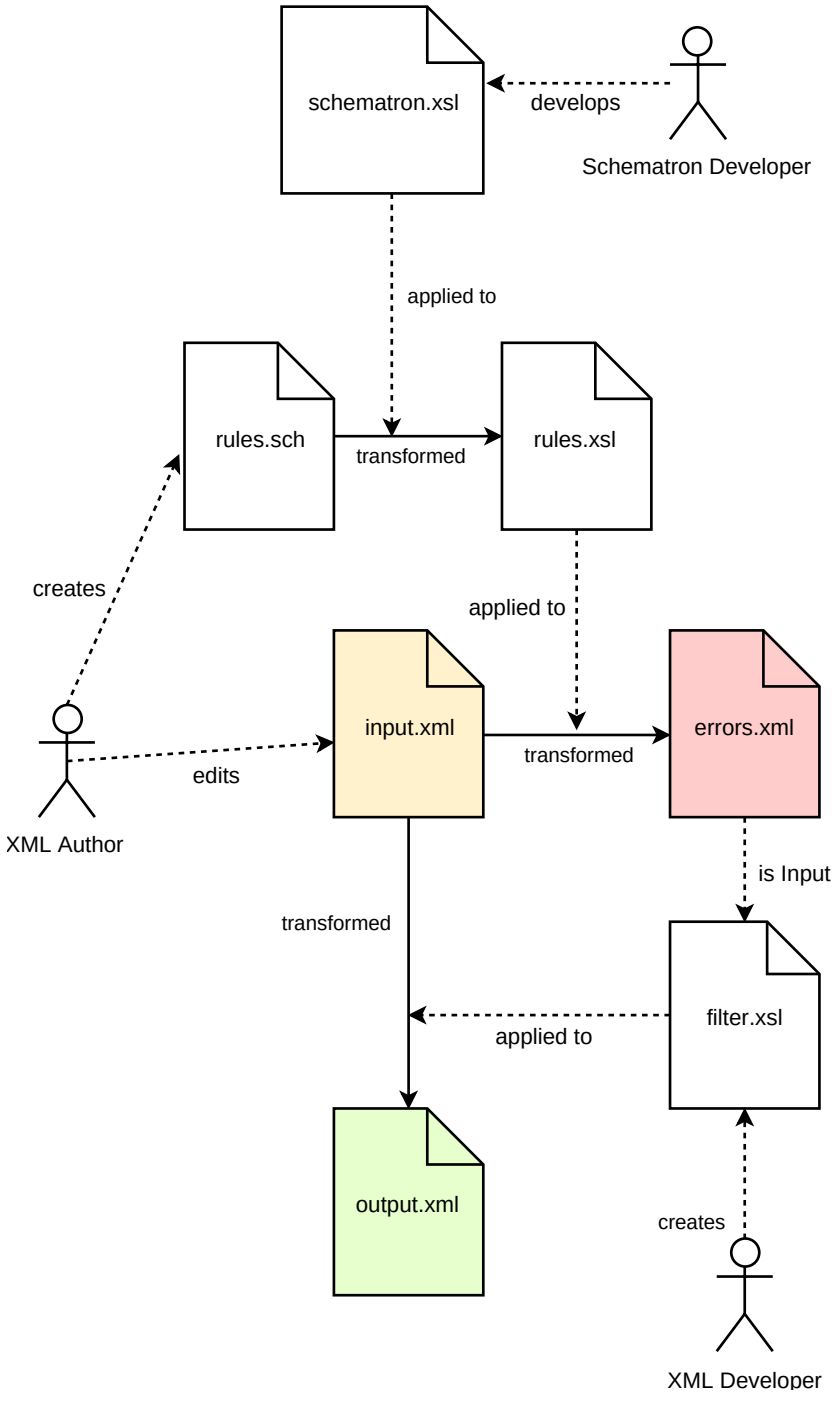

Bild 4: Schematron Validierung mit Filter

Zu finden ist das Schematron Programm auf Github: https://github.com/Schematron/schematron. Dieses Repo ist etwas unübersichtlich. Der relevante Teil des Sourcecodes befindet sich unter: **schematron/code**

3 Ausgewählte Themen

CLI Verwendung

Um die Schematron XSLT Skripte in eine eigene XSLT Anwendung per Kommandozeile einzubinden, könnte man folgendermassen vorgehen:

- Im eigenen GIT Projekt das Schematron Projekt als Submodule referenzieren.
- Eine Regelbasis anlegen, beispielsweise **$project_name.sch**.
- Zwei Batch-Skripte anlegen, beispielsweise **generate_schema.sh** und **validate.sh**.

Mittels des Skripts **generate_schema.sh** wird aus der Schematron Regelbasis das Schematron XSLT Stylesheet generiert. Der Inhalt dieser Batchdatei könnte zum Beispiel so aussehen:

```
saxon $script_dir/$project_name_validation.sch $script_dir/schematron/iso_dsdl_include.xsl
 | \ saxon -s:- $script_dir/schematron/iso_abstract_expand.xsl | \
saxon -s:- $script_dir/schematron/iso_svrl_for_xslt2.xsl \
  generate-fired-rule=false > $script_dir/$project_name_validation.xsl
```

Der Prozess zum Erzeugen des projektspezifischen Validerungs-XSLT-Skripts ist dreistufig und wird über die folgenden XSLT Schritte abgearbeitet.

- iso_dsdl_include.xsl
- iso_abstract_expand.xsl
- iso_svrl_for_xslt2.xsl

Herauszufinden, was in diesen Skripten passiert, sei dem geneigten Leser selbst überlassen. Uns interessiert an dieser Stelle nur das Resultat, nämlich das XSLT Stylesheet **$project_name_validation.xsl.**

Dieses Skript wird in der Batchdatei **validate.sh** aufgerufen:

```
saxon $xml_instance_to_check.xml $script_dir/$project_name_validation.xsl \
  > $validation-result.xml
```

Die Ausgabe dieses Prüfprozesses ist eine XML Datei mit den Fehlern in der Eingabe-XML-Instanz, die weiterverarbeitet werden kann, beispielsweise als Filterkriterium für einen nachfolgenden Prozessschritt. Ihr Inhalt dieser Datei sieht z.B. wie folgt aus:

```
<svrl:schematron-output xmlns:svrl="http://purl.oclc.org/dsdl/svrl" [...]
  <svrl:active-pattern document="file:/Users/alex/xml_instance_to_check.xml"
   id="default" name="default"/>
  <svrl:failed-assert test="count(key('unique-ids', current()))=1">
    <svrl:text>ID is not unique!</svrl:text>
    <svrl:diagnostic-reference diagnostic="default">
      <bk:id xmlns:bk="http://tekturcms/namespaces/book">1234-5678-9</my:id>
    </svrl:diagnostic-reference>
  </svrl:failed-assert>
  [...]
```

Neben den **svrl:failed-assert** Elementen, die angeben, was bei der überprüften XML-Instanz fehlgeschlagen ist, gibt es auch die Möglichkeit sich positive Ergebnisse anzeigen zu lassen - über das Element **svrl:successful-report**.

Konkret bedeutet das obige XML Schnipsel, dass unsere **id** mit dem Wert **1234-5688-9** im geprüften XML Dokument nicht eindeutig ist. Die Schematron Regelbasis, die wir zur Überprüfung angegebenen haben, sieht so aus:

```
<schema xmlns:sch="http://purl.oclc.org/dsdl/schematron" [...]
  <xsl:key name="unique-ids" match="bk:id" use="."/>
  <sch:let name="date-regex" value="'^((19|2[0-9])[0-9]{2})-(0[1-9]|1[012])
                                    -(0[1-9]|[12][0-9]|3[01])$'"/>
  <sch:pattern id="default">
    <sch:rule context="book">
      <sch:assert id="check-book-id" role="error" test="count(key('unique-ids', bk:id))=1"
                  diagnostics="default">ID is not unique!</sch:assert>
      <sch:assert id="check-book-published" role="error"
                  test="matches(bk:published,$date-regex)
    </sch:rule>
    [...]
  </sch:pattern>
  <sch:diagnostics>
    <sch:diagnostic id="default">
      <xsl:element name="bk:id">
        <xsl:value-of select="bk:id"/>
      </xsl:element>
    </sch:diagnostic>
  </sch:diagnostics>
```

Neben der "successful" und "failed" Regeln ist auch die Deklaration von Funktionen und Variablen im Body der Regelbasis erlaubt. Dies ermöglicht komplexe Bedingungen, bespw. durch das Nachschlagen in einer Lookup-Tabelle abzuprüfen.

3.3 Erste Schritte mit Xspec

XSpec ist ein **Test-Framework** [25)] für XSLT, XQuery und Schematron. Um beispielsweise komplexe Schematron Regeln zu testen, hinterlegt man in einem **Test-Szenario** Erwartungswerte für positive und negative Testfälle in Form von XML Schnippseln.

```
<test-szenario>
  <testfall>
    <personen>
      <person>
        <vorname>Horst</vorname>
        <nachname>Schlämmer</nachname>
        <gewicht>100</gewicht>
      </person>
      <person>
        <vorname>Gundula</vorname>
        <nachname></nachname>
        <gewicht>60</gewicht>
      </person>
    </personen>
  </testfall>
</test-szenario>
```

in einer XSpec Datei *. werden **Assert- und Not-Assert-Methoden** deklariert:

25) https://github.com/xspec

```
<x:description xslt-version="2.0" xmlns:x="http://www.jenitennison.com/xslt/xspec"
               schematron="test.sch">
  <x:scenario label="ALL">
    <x:context href="test.xml"/>
    <x:expect-not-assert id="person-nachname-rule" location="//person[1]/nachname"/>
    <x:expect-assert id="person-nachname-rule" location="//person[2]/nachname"/>
  </x:scenario>
</x:description>
```

Grds. bdeutet ein Assert, dass das Mapping zwischen tatsächlichem Wert und Erwartungswert des Testfalls positiv erfüllt ist. Beim Not-Assert ist das Gegenteil der Fall. Im obigen Beispiel reichen zwei Regeln, um den Testfall vollständig abzudecken.

Wenn man Schematron Regeln mit Hilfe von XSpec testen will, dann muss man ein bisschen um die Ecke denken. Denn auch diese Regeln werden mittels Assert und Not-Assert modelliert.

```
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform" queryBinding="xslt2">
  <sch:pattern id="main">
    <sch:rule context="nachname">
      <sch:assert id="person-nachname-rule" role="error" test="normalize-space(.)">
        Der Nachname der Person mit ID: <sch:value-of select="@id"/> fehlt!
      </sch:assert>
    </sch:rule>
  </sch:pattern>
</sch:schema>
```

In der Schematron-Regel wird zugesichert (Assert), dass jede Person einen Nachnamen hat.

Hat sie keinen Nachnamen so wird der Bericht zum Fehlerfall in die Schematron Ergebnisdatei geschrieben. Diese Datei wertet nun Xspec aus.

> ► *Erscheint ein Fehler (= das Feld **nachname** ist leer), so greift bei Xspec die Assert-Regel! Das ist die umgekehrte Logik zu den Schematron Regeln.*

Als Eselsbrücke kann man ein Assert in der Xspec Datei gleichsetzen mit **Appear** und ein Not-Assert mit **Not-Appear**.

Ein Assert sichert also zu, dass sich ein Fehlerbericht in der Schematron Ergebnisdatei zum Testfall befindet. Ein Not-Assert sichert zu, dass sich kein Fehlerbericht befindet.

Wie man sich leicht vorstellen kann, sind Assert-Regeln in diesem Fall leicht zu finden, dazu muss man nur die Schematron Testregeln ins Leere zeigen lassen. Alles ist grün und alles ist gut - dem Augenschein nach.

3.4 Vererbung

Mit XSLT kann man Konstrukte nachbilden, so wie sie in anderen Programmiersprachen auch vorhanden sind. Bspw. die Vererbung. Dabei wird in einer Spezialisierung eine schon bereits getätigte Implementierung übernommen und erweitert oder eingeschränkt.

Der Vorteil dabei ist, dass man nicht alles nochmal neu schreiben muss. Das verkleinert die Redundanz, führt zu einer besseren Wartbarkeit und einer geringeren Fehleranfälligkeit.

Beispiel: Parameterisierung

Gewöhnlich implementiert man ein Stylsheet für ein bestimmtes Ausgabeformat und eine Produktvariante. Schrittweise werden dann weitere Varianten und Formate hinzugefügt.

Am komfortabelsten hat man es natürlich, wenn zu Beginn der Implementierung eine vollständige Spezifikation vorliegt... Das ist aber natürlich eher selten der Fall.

Aus diesem Grund ist es wichtig, sich eine gute Strategie zu überlegen, damit die Architektur nicht in Spagetthi-Code auswartet.

Eine gute Option wäre, die XSLT Import Präzedenz auszunutzen, vgl. Kapitel *Eindeutigkeit der Regelbasis auf Seite 17* .

Angenommen es geht darum zu einem späteren Zeitpunkt weitere Parameter einzuführen. Ein Switch, wie der folgende, müsste dann an mehreren Stellen im Code aktualisiert werden.

```
<xsl:choose>
    <xsl:when test="$myParameter='this_option'">
        <!-- do this -->
    </xsl:when>
    <xsl:when test="$myParameter='that_option'">
        <!-- do that -->
    </xsl:when>
        [...]
</xsl:choose>
```

Besser ist es, wenn man ein Core-Stylesheet pflegt, das für ein Format und eine Produktvariante gut ausgetestet ist. Dieses Core-Stylesheet wird dann einfach für eine neue Variante importiert und relevante Teile werden für die neue "Spezialisierung" überschrieben. Beispielsweise könnte eine Regel zum Setzen des Headers auf jeder Seite so implementiert sein:

```
<xsl:template name="render-header">
    <!-- print logo on the left side spanning two rows-->
    <!-- print some metadata right side first row -->
    <!-- print a running header right side second row -->
</xsl:template>
```

Will man in einem neuen Format, bspw. A5, diese Logik austauschen und nur eine Zeile drucken, z.B. weil man nicht so viel Platz hat, so würde in einem "abgeleiteten" Stylesheet einfach die Regel noch einmal implementiert.

```
<xsl:choose>
<xsl:template name="render-header">
    <!-- print a running header on left side -->
    <!-- print logo on right side -->
</xsl:template>
```

Dieses Template hat nun Vorrang und wird zur Auswertung herangezogen, mit der Konsequenz, dass der Header nur einzeilig gedruckt wird. Das schöne an diesen "Named-Templates" ist auch, dass man sie innerhalb von Variablen verwenden kann:

```
<xsl:variable name="margin-width">
    <xsl:call-template name="get-margin-width"/>
</xsl:variable>
```

Das Template "get-margin-width" kann in einem "Sub"-Stylesheet überschrieben werden ohne dass die Variablen-Zugriffe im Core-Stylesheet angepasst werden müssten. Eine Zuweisung, wie:

```
width="{$margin-width}"
```

müsste nirgendwo im Code nochmal angefasst werden.

3.5 Abfragen mit XQuery

Xquery führt im Publishing-Bereich ein Schattendasein. In meiner Zeit als XSL Programmierer für zwei Publishing Firmen hatte ich damit nie zu tun. Erst als ich näher an den eigentlichen Daten war und mit XML Datenbanken zu tun hatte, kam ich mit XQuery in Berührung.

Während relationale Datenbanken mit SQL abgefragt werden, verwendet man bei XML Datenbanken, wie eXist[26] oder Marklogic[27], XQuery als Abfragesprache.

Aber auch einzelne XML Dokumente können z.B. in Oxygen XML Editor mit dem XQuery Builder Tool[28] oder auch per Saxon Kommandozeile abgefragt werden:

```
java -cp usr/lib/saxon/saxon.jar net.sf.saxon.Query
     -s:"schulen.xml"
     -qs:"/schulen/schule[id='6']"
     -o:"/Users/Alex/Desktop/schule_6.xml"
```

Mit der Option **-qs** kann hier der Querystring angebenen werden.

Wie man an dem einfachen Beispiel schon sieht, ist XQuery mit XPATH verwandt. XQuery umfasst den Sprachumfang von XPATH bietet aber zusätzlich die FLOWR Syntax um mächtigere Abfragen stellen zu können. Mittels weiterer Extensions[29] können aber auch ganze Programme erstellt werden, die weit über die Funktionalität einer "Abfragesprache" hinausgehen.

26) http://exist-db.org/exist/apps/homepage/index.html
27) https://de.marklogic.com/
28) https://www.oxygenxml.com/xml_editor/xquery_builder.html
29) http://cs.au.dk/~amoeller/XML/querying/flwrexp.html

XQuery Builder

Oxygen XML Editor bietet eine schöne Möglichkeit XQuery-Abfragen auf einem geladenen XML Dokument auszuführen. Dazu kann man seine Query in das betreffende Eingabefenster schreiben.

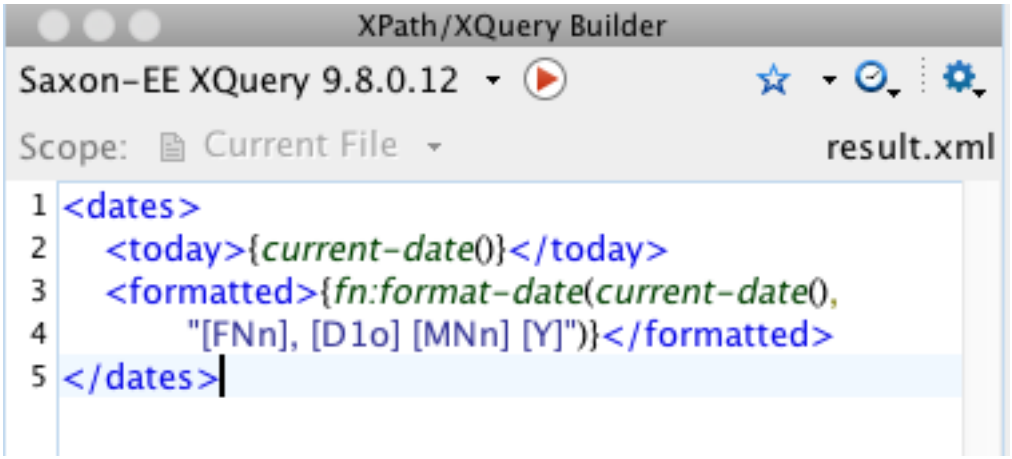

Bild 5: oXygen XQuery Builder

Mit folgendem Ergebnis:

```
<dates>
  <today>2019-01-16+01:00</today>
  <formatted>Wednesday, 16th January 2019</formatted>
</dates>
```

FLOWR Expression

FLOWR steht für for, let, where, order by, return. Das sind die Query-Anweisungen, die in dem Ausdruck erlaubt sind - in genau dieser Reihenfolge.

```
let $bibliothek := .
for $x in $bibliothek//buecher,
    $y in $bibliothek//autoren/autor
where starts-with($autor, 'Grass')
    and $x/@autorId = $y/@id
return $x/titel
```

In dieser Query werden die Titel aller Bücher von Grass zurückgeliefert. Bemerkenswert ist hier die Syntax.

> ▶ *Normalerweise würde man zwischen den einzelnen Anweisungen einen Blockabschluss, wie ein Semikolon erwarten. Da wir aber hier funtional prorgammieren, ist die Sache etwas anders...*

XML per XQuery

Es ist aber auch möglich XML zu erzeugen, wobei natürlich für eine Transformation XSLT vorzuziehen ist. Dazu werden Tags direkt in die Expression geschrieben, wie z.B. hier:

```
declare variable $nachname as xs:string external;
```

```
<buecher autor="{$nachname}">
{
  let $bibliothek := .
  for $x in $bibliothek/buecher//buch,
      $y in $bibliothek/autoren//autor
  where starts-with($y, $nachname)
      and $x/@autorId = $y/@id
  order by $x/ausgabe
  return
  <buch ausgabe="{$x/ausgabe}">
    {$x/titel}
  </buch>
}
</buecher>
```

Speichert man dieses Schnippsel in einer Datei buecher.xquery ab, so kann man mit der folgenden Kommandozeile auf einer buecher.xml Datei als Eingabe suchen:

```
java -cp usr/lib/saxon/saxon.jar net.sf.saxon.Query -t -s:buecher.xml
                                                       -q:buecher.xquery
                                                       -o:ergebnis.xml
                                                       nachname=grass
```

Document Projection

Document Projection[30] ist ein verstecktes Saxon XQuery Feature. Es funktioniert nur für eine einzige Abfrage. Das kann schon recht hilfreich sein, wenn man ein mehrere 100MB großes Dokument durchsuchen will.

Ohne Projection würde das Beispiel von oben so verarbeitet:

```
java -cp usr/lib/saxon/saxon.jar net.sf.saxon.Query -t
     -s:buecher.xml
     -q:buecher.xquery
     -o:ergebnis.xml
     -projection:off
     nachname=grass
Saxon-EE 9.7.0.20J from Saxonica
Java version 1.8.0_60
Using license serial number V005095
Analyzing query from Desktop/buecher.xquery
Generating byte code...
Analysis time: 201.10095 milliseconds
Processing file:/Users/Alex/buecher.xml
Using parser com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser
Building tree for file:/Users/Alex/buecher.xml
using class net.sf.saxon.tree.tiny.TinyBuilder
Tree built in 3.482278ms
Tree size: 46 nodes, 58 characters, 6 attributes
Execution time: 27.137589ms
Memory used: 67031664
```

Mit der Option -projection:on verändert sich die Ausführungszeit signifikant:

```
[...]

Document projection for file:/Users/Alex/buecher.xml
```

30) http://www.saxonica.com/documentation/#!sourcedocs/projection

```
-- Input nodes 50; output nodes 27; reduction = 46%
Tree built in 3.80615ms
Tree size: 26 nodes, 58 characters, 3 attributes
Execution time: 15.83463ms
Memory used: 64339064
```

3.6 XSLT Streaming

Bei grossen flach strukturierten Datenmengen gibt es zwei Möglichkeiten:

1. Für einfache Sammel- und Auswertungsaufgaben schreibt man sich am besten einen kleinen Parser, z.B. mit der Python sgmllib[31].
2. Für komplexere Aufgaben, in denen man nicht an jeder Stelle über den ganzen XML Baum navigiert und sich die Werte zusammensuchen suchen muss, kann man die Streaming Funktion des Saxon XSLT Prozessors verwenden.

XSLT Streaming ist in der XSLT Version 3.0 neu hinzugekommen[32] und in der kommerziellen Saxon-EE Löunsg implementiert[33]. Bei dieser Methode wird kein Eingabebaum im Speicher aufgebaut, was zu einer drastischen Performanzsteigerung führt.

Es gibt ein paar Regeln, die man bei der Verarbeitung großer Datenmengen über die Streaming Funktionen beachten sollte:

- Bei einer XPATH Auswertung sollte nur ein einfacher Ausdruck mit höchstens einer konsumierenden Selektion gegeben sein. Konsumieren heißt, dass vom Kontextknoten aus eine Knotenmenge abwärts selektiert wird. Dagegen bleibt die Information bzgl. der Ancestor-Achse erhalten.
- Bei einer Selektion sollte man aber darauf achten nur atomarische Werte auszuwählen.
- Knotenmengen, die über die Streaming Option eingelesen wurden, können nicht einer Funktion übergeben werden. Sie sind auch nicht einer Variablen zuweisbar.
- "Crawler”-Ausdrücke, wie //section sind nicht nicht zu verwenden, ebenso ein rekursiver Abstieg mit Selektion, wie bspw. mit einem Apply-Templates Call.

Zu Beginn der Streaming-Aktion kann man sich auf konventionelle Art und Weise Teilbäume, die nicht so performanzlastig aufgebaut werden, in einer Variablen abspeichern und im Verlauf der Streaming-Verarbeitung z.B. für einen Vergleich auswerten.

Ein einfaches Streaming Stylesheet könnte z.B. so aussehen:

```
<xsl:stylesheet version="3.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="#all">

  <xsl:output method="xml" indent="yes"/>

  <xsl:mode on-no-match="shallow-copy" use-accumulators="entry-count" streamable="true"/>

  <xsl:accumulator name="entry-count" as="xs:integer" initial-value="0"
                   streamable="yes">
    <xsl:accumulator-rule match="entry" select="$value + 1"/>
```

31) https://docs.python.org/2/library/sgmllib.html
32) https://www.saxonica.com/html/documentation/sourcedocs/streaming/xslt-streaming.html
33) https://www.saxonica.com/html/documentation/sourcedocs/streaming/

```
  </xsl:accumulator>

  <xsl:template match="/">
    <result>
      <xsl:apply-templates/>
      <count>
        <xsl:value-of select="accumulator-after('entry-count')"/>
      </count>
    </result>
  </xsl:template>

</xsl:stylesheet>
```

Diese Stylesheet hat einige Besonderheiten:

Zum einen wird darin ein Default-Modus deklariert, der jeden Knoten der Eingabeinstanz über eine implizite Identity-Transformation (shallow-copy)[34] in die Ausgabeinstanz kopiert.

Auf herkömmlichem Weg würde man dafür ein Templates wie dieses verwenden:

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

Zum anderen wird ein Akkumulator verwendet. Normalerweise gibt es in XSLT keine Variablen, sondern nur Konstanten, so wie das auch bei funktionalen Programmiersprachen der Fall ist.

Es gab zwar schon länger eine Saxon-Erweiterung, die die mehrmalige Zuweisung eines Wertes an eine Variable erlaubte, im Normallfall braucht man diese Eigenschaft aber nicht.

Ber der Verarbeitung sehr großer Datemengen, ist es aber unumgänglich, denn sonst würde der Laufzeitstapel schnell an seine Grenzen gelangen.

Ein Akkumulator akkumuliert Werte, wie der Name schon sagt. Das können atomare Typen sein, wie im obigen Beispiel, aber auch Datenstrukturen können aufgebaut werden, wie bspw. das Abspeichern des gerade prozessierten Teilbaums in einem Dictionary zur späteren Auswertung bzw. Gruppierung der Key-Elemente.

Auch im Akkumulator muss das streamable="yes" Property gesetzt sein, wenn er im Streaming-Modus arbeiten soll. In diesem Modus kann der Akkumulatorwert erst ausgelesen werden, wenn der untersuchte Baum vollständig durchlaufen wurde.

Um die Unterschiede zum "normalen" XSLT Betrieb festzustellen, können im obigen Beispiel einige offensichtlich korrekte Änderungen vorgenommen werden, die der Streaming Prozessor allerdings nicht akzeptiert.

```
Cannot call accumulator-after except during the post-descent
phase of a streaming template
```

Diese Fehlermeldung erscheint, wenn man den Apply-Templates Call entfernt. Der Akkumulator wird also nur befüllt, wenn der Baum auch explizit durchlaufen wurde. Dieser Durchlauf

34) https://www.saxonica.com/html/documentation/xsl-elements/mode.html

kann auch ein reines Kopieren sein, bspw. kann man den Apply-Templates Call auch durch ein

```
<xsl:copy-of select="."/>
```

ersetzen, was gleichbedeutend mit der Mode Einstellung

```
on-no-match="deep-copy"
```

wäre. Wie man sieht hat sich in XSLT 3.0 viel bzgl. der Handhabung verschiedener Verarbeitungsmodi getan. Anstatt Default-Match Regeln zu schreiben, kann man ganz oben am Stylesheet Modus Properties setzen, die den Baumdurchlauf auf verschiedene Arten realisieren.

Die Verarbeitung großer Datenmenden ist aber mit Streaming etwas tricky und es sollte geprüft werden, ob ggf. konventionelles Performanz-optimiertes XSLT für den Anwendungsfall ausreichen würde.

3.7 Namespaces

Wenn man XML Instanzen aus unterschiedlichen Quellen mit XSLT verarbeiten will, wird man sich wohl oder übel mit dem Thema Namespaces (NS) auseinander setzen müssen, um Konflikte in den Elementselektoren zu vermeiden.

Gerade bei hintereinandergeschalteten Transformationen kann es auch passieren, dass unerwartet ein Namespace in die Ausgabe generiert wird, den der folgende Prozesschritt nicht versteht, weil er dort nicht deklariert wurde.

Es gibt meherere Möglichkeiten einen Namespace im Stylesheet zu deklarieren. Gehen wir davon aus, dass in einem Transformationsschritt genau eine Quelle und max. eine Konfigurationsdatei verarbeitet wird, dann kann das Stylsheet-Element bspw. so aussehen:

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tektur="https://namespace-eigener-xslt-funktionen"
  xmlns="http://namespace-in-der-xml-eingabe.com/"
  xpath-default-namespace="https://namespace-der-konfigdatei.com/"
  exclude-result-prefixes="#all">
```

- Der **xsl** Namespace ist klar
- Der **xs** Namespace ist notwendig, wenn man typisiert arbeiten will. Er erlaubt das Einbinden von Datentypen nach der XML Schema Spezifikation[35)] und somit die bessere Validierung des Stylesheets zur Compile-Zeit.
- Die Deklaration eines eigenen geprefixten Namespaces erlaubt das Einbinden von eigenen XSLT Funktionen, wie z.B. auch das Einbinden der FunctX Bibliothek[36)]
- Der Nicht-geprefixte Namespace ist der Default-Namespace und kann einen NS aus der Eingabe handeln

35) https://de.wikipedia.org/wiki/XML_Schema
36) http://www.xsltfunctions.com/

– Das Attribut **xpath-default-namespace** gibt einen weiteren NS an, der in XPATH Funktionen verwendet werden kann. In diesem Feld würde ich den NS einer Konfigurations- oder separaten Datendatei angeben.

Mehr als einen NS in der Eingabe sollte man aus meiner Sicht bei der XML Verarbeitung nit XSLT vermeiden wenn es geht. GGf. empfiehlt es sich, die Eingabe vor der Verabeitung zu normalisieren und Elemente ggf. umzubennen.

Befinden sich in den Eingabedaten Namespaces, die man in den XSLT Stylesheets nicht handelt - der Namespace kann auch nur an einem ganz bestimmten Element hängen - so kannp bei der Transformation - ohne Fehlermeldung - zu unerwarteten Ergebnissen kommen.

► Deshalb sollte man die Daten im Vorfeld bzgl. Namespaces sehr genau analysieren.

Namespaces in der Eingabe werden über die Kopfdeklaration in der Stylesheetdatei gehandelt, welcher Namspace schliesslich in die Ausgabe geschrieben wird, hängt vom aktuell verarbeiteten Kontextknoten ab:

– Elemente, die man erzeugt, erhalten automatisch den Default-Namespace, wenn man nicht explizit einen NS angibt.
– Elemente, die man kopiert, transportieren den Namespace, den sie in der Eingabe hatten, wenn man dies nicht explizit verhindert.

Um diese beiden Default Einstellungen zu steuern (bzw. zu überschreiben) gibt es mehrere Möglichkeiten:

```
<xsl:element name="{local-name()}" namespace="{namespace-uri()}">
```

Hier wird ein Element mit dem un-geprefixten Namespace des Kontextknotens deklariert. Wenn der Kontextknoten keinen anderen Namespace hat, so wird hierdurch sichergestellt, dass der Default NAmespace auch tatsächlich in die Ausgabe kommt.

```
<xsl:element name="meinelement" namespace="">
```

Hier wird ein Element ohne Namespace Angabe in die Ausgabe geschrieben. Es würde dann den Default-Namespace der Ausgabe erben. Es gibt auch ein Attribut am xsl:copy Element, das den Vorgang des Namespace-Kopierens steuern kann:

```
<xsl:template match="p">
  <xsl:copy copy-namespaces="no">
     <xsl:apply-templates/>
   </xsl:copy>
</xsl:template>
```

Hier wird der Namespace am p Element nicht in die Ausgabe geschrieben. Ggf. funktiniert diese Funktion aber mit unerwarteten Ergebnissen, deshalb sollte man sich nicht darauf verlassen.

Ebenso kann eine Default-Kopierregel verwendet werden, die es verbietet einen Namespace weiterzuvererben:

```
<xsl:template match="@* | node()">
   <xsl:copy inherit-namespaces="no">
     <xsl:apply-templates select="@* | node()"/>
   </xsl:copy>
</xsl:template>
```

Namespaces in XQuery

Während XSLT dazu dienen sollte, XML Daten in andere (XML-) Formate zu transformieren, dient XQuery z.B. dazu auf einer NoSQL Datenbank Daten aus unterschiedlichen Quellen zu selektieren, zu harmonisieren und an verarbeitende Prozesse weiterzugeben.

Deshalb ist es für mich nicht so erstaunlich, dass das Namespace Konzept in XQuery irgendwie besser funktioniert.

Damit man überhaupt Daten auf einem mit Namespaces versehenen XML Dokument selektieren kann, müssen alle Namspaces am Anfang des XQuery Ausdrucks angegeben werden, das sieht so aus:

```
xquery version "1.0-ml";

import module namespace tektur = "http://www.teturcms.de/xquery/common"
                                 at "common.xqy";
import module namespace mem = "http://xqdev.com/in-mem-update"
                                 at '/MarkLogic/appservices/utils/in-mem-update.xqy';
declare namespace local = "https://lambdawerk.com/code/alex-sandbox/1.0";
declare namespace weiredns = "https://weired-ns-in-input-data.com/weired/ns";
declare namespace xs = "http://www.w3.org/2001/XMLSchema";
```

Hier werden zuerst Funktionen aus anderen Modulen eingebunden, nämlich die in einer Datei common.xqy im selben Verzeichnis aus der eigenen Entwicklung, sowie die Bibliothek mem aus der MArklogic Umgebung. Danach wird ein NS local deklariert, den man verwenden wird, wenn im weiteren Verlauf eigene XQuery Funktionen verwendet werden sollen, sowie der NS weiredns, der in den Eingabedaten vorhanden ist. Der NS xs ist analog zum XSLT Beispiel gesetzt.

4 Zusätzliches Know-How

Unsortierte Notizen, die für jeden XSLT-Programmierer interessant sein könnten.

4.1 GIT Einmaleins

Im Fachbereich Technische Dokumentation wird neue Technologie ein bisschen langsamer adaptiert, als in anderen Disziplinen der IT - schliesslich muss ja erst alles dokumentiert werden ...

Bei Versionsverwaltungssystemen ist das nicht anders. Doch inzwischen hat auch GIT in manchen Entwicklerbüros Einzug gehalten und verdrängt Subversion.

Ein Grund an dieser Stelle einmal zumindest die wichtigsten Befehle aufzulisten - Note to self:

```
GIT COMMANDS

CLONE
=====
git clone https://github.com/libgit2/libgit2
SIMPLE
======
git init
git add *
git commit -m "Form validation added”
[changes]
git add [changed files] or git commit -a
git diff —cached
git branch experimental
git checkout experimental
git commit -a -m "only in the experimental branch”
git checkout master
git merge experimental
git branch -d experimental

COLLABORATION
=============
push new branch git push -u origin newLocalBranch
bob$ git clone /home/alice/project myrepo
(edit files)
bob$ git commit -a
alice$ cd /home/alice/project
alice$ git pull /home/bob/myrepo master  (fetch & merge)
(fetch and not merge)
alice$ git fetch /home/bob/myrepo master
alice$ git log -p HEAD..FETCH_HEAD
(visualization only)
gitk HEAD..FETCH_HEAD
gitk HEAD…FETCH_HEAD
—
git log
git show ea14 (firs chars is enough)
git grep "headline_font_size” v2.5

PULL REQUESTS
=============
```

```
(checkout branch)
git pull origin simple-examples
(changes)
git push origin simple-examples
Create a pull request using the form on the Github page and assign it to a Reviewer

MERGE CONFLICTS
===============
git pull origin master
=> merga conflicts

TROUBLE SHOOTING
================
git reset --hard origin/master
git tree
git status
git checkout 118886ee3f06738b53f089433078d35f4d70a8f9 vendor/error-report.xsl
git log --diff-filter=D --summary
```

4.2 XML Editoren

Der XSLT Styesheet-Entwickler wird sich gewöhnlich mit Eingabedaten beschäftigen, die entweder automatisch mittels irgendeines Prozesses erzeugt wurden, oder die durch einen menschlichen Autor mit einem XML Editor eingegeben wurden.

Aus diesem Grund ist es ganz nützlich, die wichtigsten Editoren zu kennen. Wir unterscheiden zwischen Desktopapplikationen und Webanwendungen. Ausserdem unterscheiden wir noch ob der Editor WYSIWIG (**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et) oder WYSIWYM (**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **M**ean) unterstützt oder eine Mischung aus beidem darstellt.

WYSIWYM Desktop

Editor	Beschreibung
XMetal [XM]	XMetal ist wahrscheinlich der am weitesten verbreitete reine WYSIWYM Editor. Er hat Schnittstellen zu COM und Java und kann daher in eigene CMS integriert werden.
Arbortext XML Editor [EP]	Arbortext XML Editor, früher bekannt als EPIC ist sehr betagt. Ich hatte damit im Bereich Luftfahrt/Verteidigung zu tun. Bekanntermassen ist sein Tabelleneditor etwas buggy.

WYSIWYG Desktop

XMetal kann so konfiguriert werden, dass bei einer einfachen DTD der Content Bereich wie Word aussieht. Auch Code Editoren, wie OxygenXML bieten diese Möglichkeit. Das Key-Handling bei dieser Variante zeigt aber schnell, dass die UX noch weit von herkömmlichen Textverarbeitungssystem, wie Word oder OpenOffice entfernt ist.

WYSIWYM Online

Editor	Beschreibung
Oxygen XML WebAuthor [OX]	Dieser Online-Editor verwendet auf der Serverseite dieselbe Logik, wie das Desktop Programm des Herstellers. Das führt dazu, dass bei jedem Tastendruck eine Verbindung zum Server aufgebaut wird, und die Verarbeitung langsam werden kann. Zum Betrieb und bzgl. Customizing ist einschlägiges Java-Know-How erforderlich.
FontoXML [FX]	FontoXML sieht schon fast aus wie Word. Neben der WYSWYG/M Darstellung, kann auch die XML Struktur in einem Seitenpanel angezeigt werden.
XEditor [XE]	Xeditor benutzt XSLT Transformationen, um aus der Eingabe die Editoransicht zu generieren. Beim Abspeichern wird der umgekehrte Weg bestritten. Das mag zwar auf den ersten Blick etwas holprig erscheinen, wie aber auch Tektur beweist, funktioniert das ganze recht gut und schnell.
Xopus [XO]	Xopus ist wohl der älteste web-basierte XML Editor. Ich hatte damit schon 2008 zu tun, als er für ein Redaktionssystem evaluiert wurde. Wir haben uns dann für eine eigene nicht-generische Lösung basierend auf dem Webeditor CKEditor entschieden.

Das Customizing dieser Editoren erfordert einen sehr hohen Aufwand. Es müssen diverse Ressourcen angepasst werden, wie XSLT Skripte, XSD Schemas, CSS und Javascript. Das Schema wird meist über Kommandozeilentools in eine JS Repräsentation überführt.

Aus diesem Grund bieten einige Hersteller spezielle Schulungen an, wo man die Bedienung erlernen kann. Aus meiner Sicht ist das Problem "Webbasierter XML Editor" weltweit noch nicht ausreichend gelöst.

[XM] https://xmetal.com/
[EP] https://www.ptc.com/en/products/service-lifecycle-management/arbortext/editor
[OX] https://www.oxygenxml.com/oxygen-xml-web-author/app/oxygen.html
[FX] https://www.fontoxml.com/
[XE] http://www.xeditor.com/portal
[XO] http://xopusfiddle.net/VT7T/3/

4 Zusätzliches Know-How

Die Kosten für den Betrieb rangieren um die 1000 EUR monatl. für ein 20 Benutzer-Setup.

5 Glossar

Begriff	Beschreibung
Core-Stylesheet	In einem Stylesheet-Projekt bezeichnet das Core-Stylesheet eine bereits ausgiebig getestete Variante, die mittels Sub-Stylesheet unter Ausnutzung der XSLT Import Präzedenz überschrieben wird.
Sub-Stylesheet	Ein Sub-Stylesheet spezialisiert das Core-Stylesheet, damit Redundanz vermieden wird und somit die Wartbarkeit gewährleistet werden kann.
Parameterisierung	Bei der Parameterisierung wird ein bestehendes Stylesheet mit Parametern versehen, um für möglichst viele Produktvarianten und Ausgabeformate die gleiche Codebasis wiederverwenden zu können. Dadurch soll Redundanz eingespart werden und der Aufruf vereinfacht werden.

Index (Abb.)

Fussnoten

1 https://de.wikipedia.org/wiki/Darwin_Information_Typing_Architecture

2 https://de.wikipedia.org/wiki/WYSIWYG

3 https://en.wikipedia.org/wiki/Topic-based_authoring

4 http://www.tekturcms.de

5 https://de.wikipedia.org/wiki/Verarbeitungsanweisung

6 https://de.wikipedia.org/wiki/Wireless_Application_Protocol

7 https://de.wikipedia.org/wiki/EPUB

8 https://de.wikipedia.org/wiki/CHM_(Dateiformat)

9 https://www.ibm.com/developerworks/library/os-echelp/index.html

10 https://en.wikipedia.org/wiki/JavaHelp

11 https://de.wikipedia.org/wiki/FrameMaker

12 https://de.wikipedia.org/wiki/S1000D

13 https://www.i4icm.de/forschungstransfer/pi-mod/

14 https://de.wikipedia.org/wiki/Journal_Article_Tag_Suite

15 https://de.wikipedia.org/wiki/Text_Encoding_Initiative

16 http://surguy.net/articles/client-side-svg.xml

17 http://jackrabbit.apache.org/jcr/node-type-visualization.html

18 http://argouml.tigris.org

19 http://butterflycode.sourceforge.net

20 https://www.javaworld.com/article/2073998/java-web-development/generate-javabean-classes-dynamically-with-xslt.html

21 http://www.saxonica.com/html/documentation/sourcedocs/streaming/

22 http://www.unidex.com/turing/utm.htm

23 https://de.wikipedia.org/wiki/XSL_Transformation

24 W3C Seiten zu The Extensible Stylesheet Language Family (XSL)

25 https://github.com/xspec

26 http://exist-db.org/exist/apps/homepage/index.html

27 https://de.marklogic.com/

28 https://www.oxygenxml.com/xml_editor/xquery_builder.html

29 http://cs.au.dk/~amoeller/XML/querying/flwrexp.html

30 http://www.saxonica.com/documentation/#!sourcedocs/projection

31 https://docs.python.org/2/library/sgmllib.html

32 https://www.saxonica.com/html/documentation/sourcedocs/streaming/xslt-streaming.html

7 Fussnoten

33 https://www.saxonica.com/html/documentation/sourcedocs/streaming/

34 https://www.saxonica.com/html/documentation/xsl-elements/mode.html

35 https://de.wikipedia.org/wiki/XML_Schema

36 http://www.xsltfunctions.com/

[XM] https://xmetal.com/

[EP] https://www.ptc.com/en/products/service-lifecycle-management/arbortext/editor

[OX] https://www.oxygenxml.com/oxygen-xml-web-author/app/oxygen.html

[FX] https://www.fontoxml.com/

[XE] http://www.xeditor.com/portal

[XO] http://xopusfiddle.net/VT7T/3/

Index

(c) Alex Düsel 2019

Creative Commons Namensnennung-Keine Bearbeitungen 4.0 International Public License
www.github.com/alexdd/Buch

Tekur CCMS ist ein web-basiertes Component Content Management System und befindet sich noch in der Entwicklung. Blog: **www.tekturcms.de**

Hier sind einige Random Features:

- Die Inhalte werden nach dem DITA Content Model eingegeben. Die Ausgabe erfolgt über ein automatisches Satzsystem.
- Grafiken können für die PDF-Ausgabe seitenbreit, spaltenbreit und in der Marginalie gesetzt werden.
- Die Breite der Marginalie ist stufenlos einstellbar; die PDF-Ausgabe ist bzgl. der Formatierung weitestgehend konfigurierbar.
- Layoutoptionen bzgl. Papierformat, Bemassung und Schriftgrößen können über einen einfachen Dialog eingestellt werden.
- TOC und mehrstufige Register werden automatisch in der PDF-Ausgabe erzeugt.
- Die Zellenbreite von CALS Tabellen kann mit der Maus eingestellt werden; Funktionen auf Zellen sind weitestgehend implementiert.
- Copy 'n Paste funktioniert Element-weise und topic-übergreifend.
- Paras, Listitems und Sections können mit den Pfeilbuttons in der Toolbar nach oben und unten verschoben werden.
- Verlinkung auf andere Topics funktioniert über Referenzen und ein Linktext wird automatisch aktualisiert, wenn sich der Topic-Titel ändert.
- Die DITA-Map kann u.a. mittels Drag 'n Drop editiert werden; Im Topic Editor gibt es an jeder Stelle ein dynamisches Kontextmenü für weitere Optionen.
- Valide DITA Strukturen können exportiert und importiert werden.
- Topics, Tasks und Maps können vom Autor an Reviewer und Approver für einen Kommentar- und Freigabeprozess überwiesen werden.