



# Entwickler Handbuch

XSLT Hands-On

XML - XSLT - XQuery



**(c) Alex Düsel 2019**

Creative Commons Namensnennung-Keine  
Bearbeitungen 4.0 International Public License  
[www.github.com/alexdd/XSLT-Buch](http://www.github.com/alexdd/XSLT-Buch)





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

Sonstige Features: Rechte- und Rollensystem, Versionskontrolle, konfigurierbarer Workflow mit Review & Approval Funktionen. Auf dem Entwicklerblog <sup>4)</sup> 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](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](https://en.wikipedia.org/wiki/Topic-based_authoring)

4) <http://www.tekturcms.de>



## Inhalt

<b>1 Theorie</b>	<b>7</b>
1.1 Push vs. Pull Stylesheets	7
1.2 Eindeutigkeit der Regelbasis	9
<b>2 Praxis</b>	<b>13</b>
2.1 Anwendungsbeispiele	13
2.2 Komplexe Anwendung	18
2.3 Vortransformationen	19
2.4 Validierung mit Schematron	22
2.5 Vererbung	26
2.6 Abfragen mit XQuery	28
2.6.1 Connector zu Marklogic in Oxygen	28
<b>3 Sonstiges</b>	<b>31</b>
3.1 GIT Einmaleins	31
3.2 XML Editoren	32
<b>4 Glossar</b>	<b>35</b>
<b>Index (Abb.)</b>	<b>37</b>
<b>Fussnoten</b>	<b>39</b>
<b>Index</b>	<b>41</b>



## 1 Theorie

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<sup>5)</sup> 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<sup>6) 7)</sup> 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.

### 1.1 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?

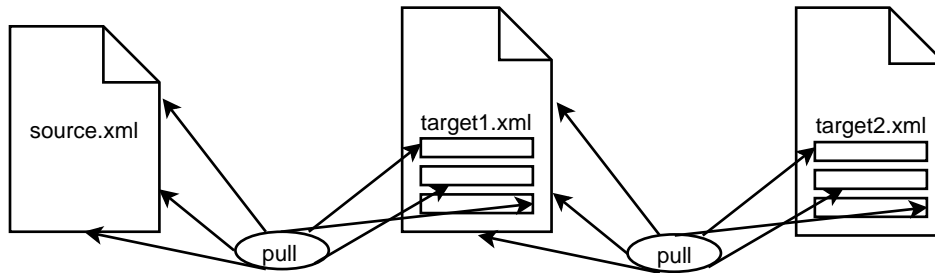
Um diesen Umstand aufzuklären ist ein bisschen Theorie notwendig:

5) <http://www.unidex.com/turing/utm.htm>

6) [https://de.wikipedia.org/wiki/XSL\\_Transformation](https://de.wikipedia.org/wiki/XSL_Transformation)

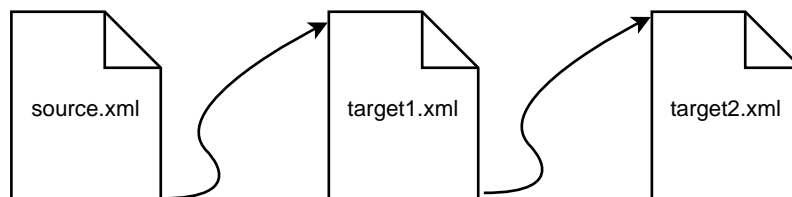
7) W3C Seiten zu The Extensible Stylesheet Language Family (XSL)

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.

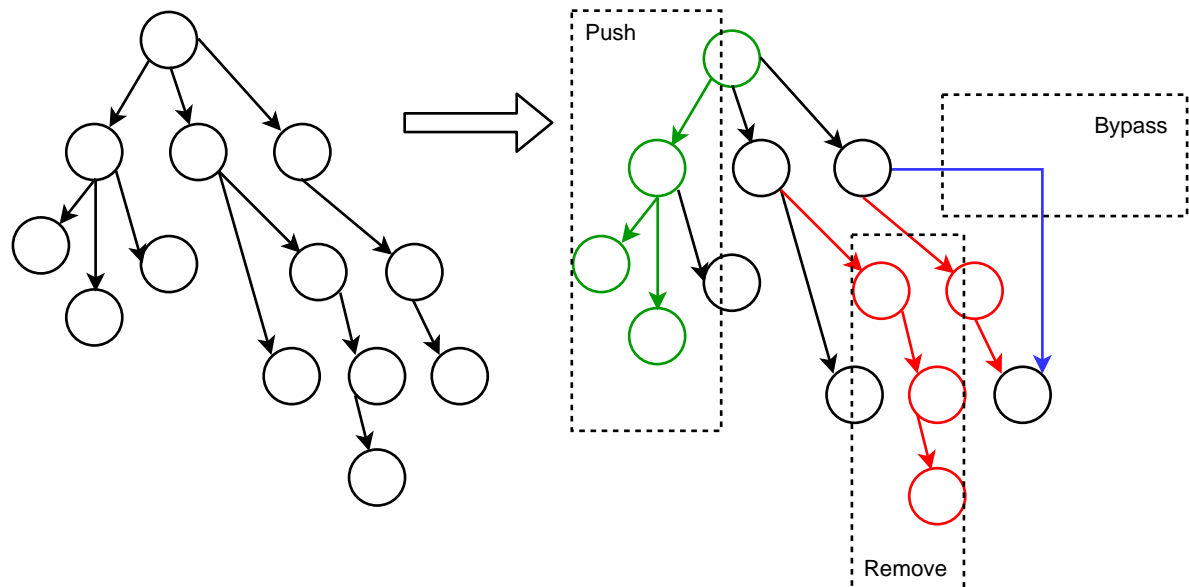


**Bild 2: Push Stylesheet**

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

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

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

Pull-Stylesheets werden gewöhnlich 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 Konvertierung relativ einfach sind, bspw. beim Auswerten / Konvertieren von Konfigurationsdateien. In allen anderen Fällen sind Push-Stylesheets vorzuziehen, d.h. möglichst wenige **for-each** loops und möglichst viele **Template-Match** Regeln.

## 1.2 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.

## 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 Präzedenz 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 wäre 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.
  - Am 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 7](#), 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.

## 2 Praxis

Hands-on Materiel mit Beispielquelltexten.

### 2.1 Anwendungsbeispiele

Einige Beispiele - aktuell und aus vergangenen Tagen.

#### 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 Auszeichnung 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 eingrücktes XML anzeigen - oder die Tags einfach ignorieren und den Textinhalt in einer Zeile darstellen. Fügt man eine Processing Instruction<sup>8)</sup> 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"?>
8) https://de.wikipedia.org/wiki/Verarbeitungsanweisung
```

```

<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>
  </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>

```

## Serverseitige Konvertierung

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

Auch eine serverseitige Konvertierung ist gebräuchlich. Ein Beispiel aus vergangenen Tagen - WAP-Seiten aus <sup>9)</sup> 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):

9) [https://de.wikipedia.org/wiki/Wireless\\_Application\\_Protocol](https://de.wikipedia.org/wiki/Wireless_Application_Protocol)

```
<?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>
    <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.<sup>10)</sup> 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<sup>11)</sup>, EclipseHelp<sup>12)</sup>, JavaHelp<sup>13)</sup>, ..., Print-Ausgabe Formate, wie PDF oder Adobe Framemaker<sup>14)</sup>, oder XML Standard Austauschformate, wie DITA, S1000D<sup>15)</sup>, PI-MOD<sup>16)</sup>, JATS<sup>17)</sup> oder TEI<sup>18)</sup>.

### Menschenlesbar machen

Kryptische XML Log-, Daten- oder Konfigurationsfiles können leicht mit XSLT menschenlesbar formatiert werden Ein 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 werden Ein 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
```

- 10) <https://de.wikipedia.org/wiki/EPUB>
- 11) [https://de.wikipedia.org/wiki/CHM\\_\(Dateiformat\)](https://de.wikipedia.org/wiki/CHM_(Dateiformat))
- 12) <https://www.ibm.com/developerworks/library/os-echelp/index.html>
- 13) <https://en.wikipedia.org/wiki/JavaHelp>
- 14) <https://de.wikipedia.org/wiki/FrameMaker>
- 15) <https://de.wikipedia.org/wiki/S1000D>
- 16) <https://www.i4icm.de/forschungstransfer/pi-mod/>
- 17) [https://de.wikipedia.org/wiki/Journal\\_Article\\_Tag\\_Suite](https://de.wikipedia.org/wiki/Journal_Article_Tag_Suite)
- 18) [https://de.wikipedia.org/wiki/Text\\_Encoding\\_Initiative](https://de.wikipedia.org/wiki/Text_Encoding_Initiative)



```
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:

```
<?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<sup>19)</sup>
- Knotentyp Visualisierung im Apache Jack Rabbit Projekt<sup>20)</sup>

## 2.2 Komplexe Anwendung

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 Wiederverwendung 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 Versio-

19) <http://surguy.net/articles/client-side-svg.xml>

20) <http://jackrabbit.apache.org/jcr/node-type-visualization.html>

nen Intelligente Querverweise: Ein Link zwischen einzelnen XML Topics bleibt versions-treu. Automatischer Satz, inkl. Zusammenhalte- 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.
- 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** <sup>21)</sup>.

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** <sup>22)</sup>

Es gibt aber auch einen schönen Artikel dazu im Java World Journal<sup>23)</sup>.

## 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<sup>24)</sup>.

## 2.3 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.

21) <http://argouml.tigris.org>

22) <http://butterflycode.sourceforge.net>

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

24) <http://www.saxonica.com/html/documentation/sourcedocs/streaming/>

- Die Zwischenergebnisse können für Debug-Zwecke ausgewertet werden 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 um 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>
    [...]
    <administrative-region id="31" name="Bavaria">
      <schools>
        <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>
        [...]
      </schools>
    </administrative-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>
  </teacher>
  [...]
</teachers>
```

```

    </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>

<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.

## 2.4 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.

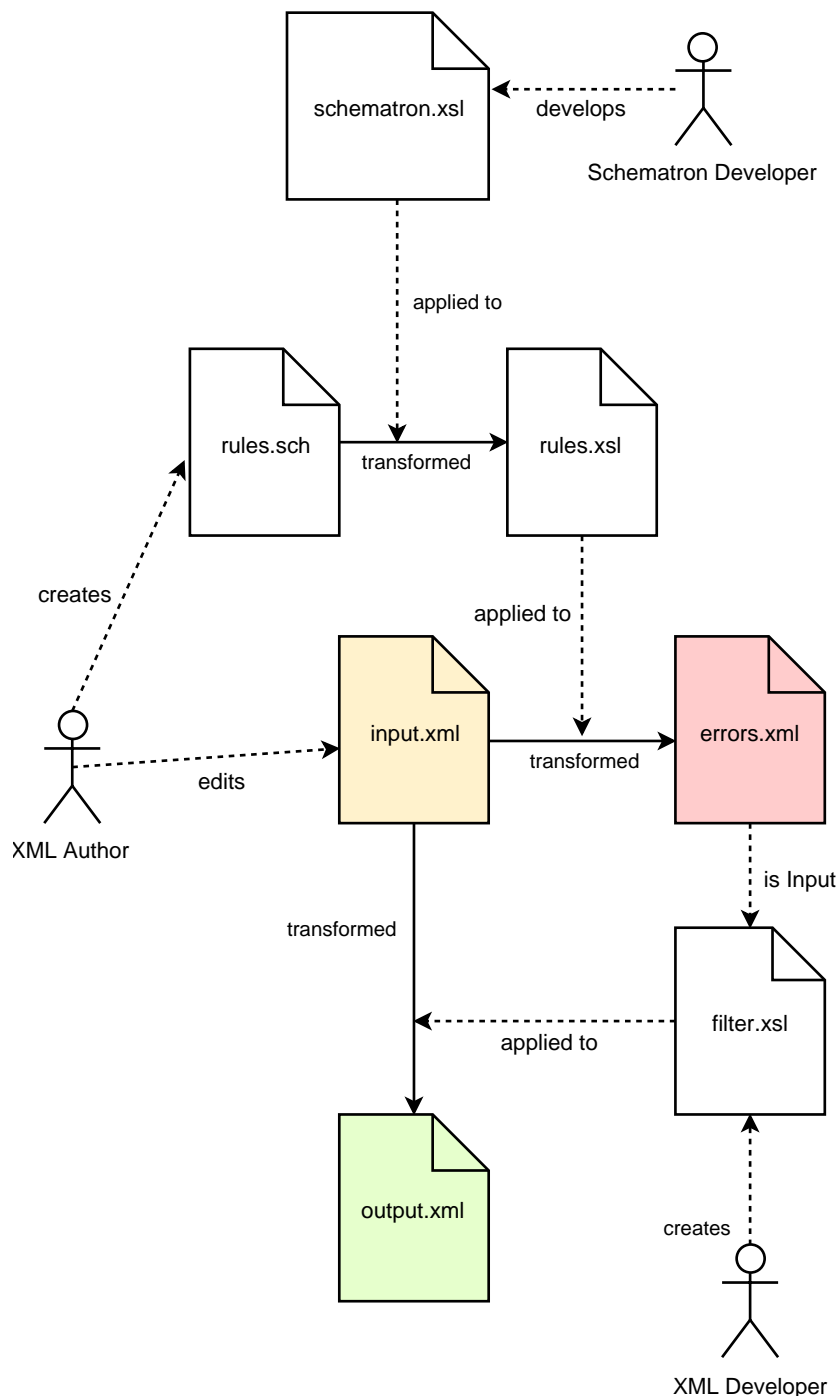
Das geht einerseits über die Angabe des Doctypes in der XML Instanz, andererseits bieten auch alle Editoren die Möglichkeit ein bestimmtes Schema explizit auszuwählen, um gegen dieses auf Anforderung zu validieren.

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**



## 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_dsd1_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 Validierungs-XSLT-Skripts ist dreistufig und wird über die folgenden XSLT Schritte abgearbeitet.

- iso\_dsd1\_include.xsl
- iso\_abstract\_expand.xsl
- iso\_svrl\_for\_xslt2.xsl

Herauszufinden, was in diesen Skripten passiert, sei dem geeigneten 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/dsd1/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]|1[2][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 abzufragen.

## 2.5 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 Stylesheet 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 Spagetti-Code ausartet.

Eine gute Option wäre, die XSLT Import Präzedenz auszunutzen, vgl. Kapitel [Eindeutigkeit der Regelbasis auf Seite 9](#).

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 einzellig 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.

## 2.6 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<sup>25)</sup> oder Marklogic<sup>26)</sup>, XQuery als Abfragesprache.

Aber auch einzelne XML Dokumente können z.B. in Oxygen XML Editor mit dem XQuery Builder Tool<sup>27)</sup> 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 angegeben 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<sup>28)</sup> können aber auch ganze Programme erstellt werden, die weit über die Funktionalität einer "Abfragesprache" hinausgehen.

### 2.6.1 Connector zu Marklogic in Oxygen

Marklogic bietet zwar auf Port 8000 per Default ein Query Console im Browser, mit der man bestimmte Sachen ausprobieren. Komfortabler arbeitet man aber mit einem Oxygen-Connector. Dazu öffnet man den *Data Source Explorer*:

25) <http://exist-db.org/exist/apps/homepage/index.html>

26) <https://de.marklogic.com/>

27) [https://www.oxygenxml.com/xml\\_editor/xquery\\_builder.html](https://www.oxygenxml.com/xml_editor/xquery_builder.html)

28) <http://cs.au.dk/~amoeller/XML/querying/flwrexp.html>

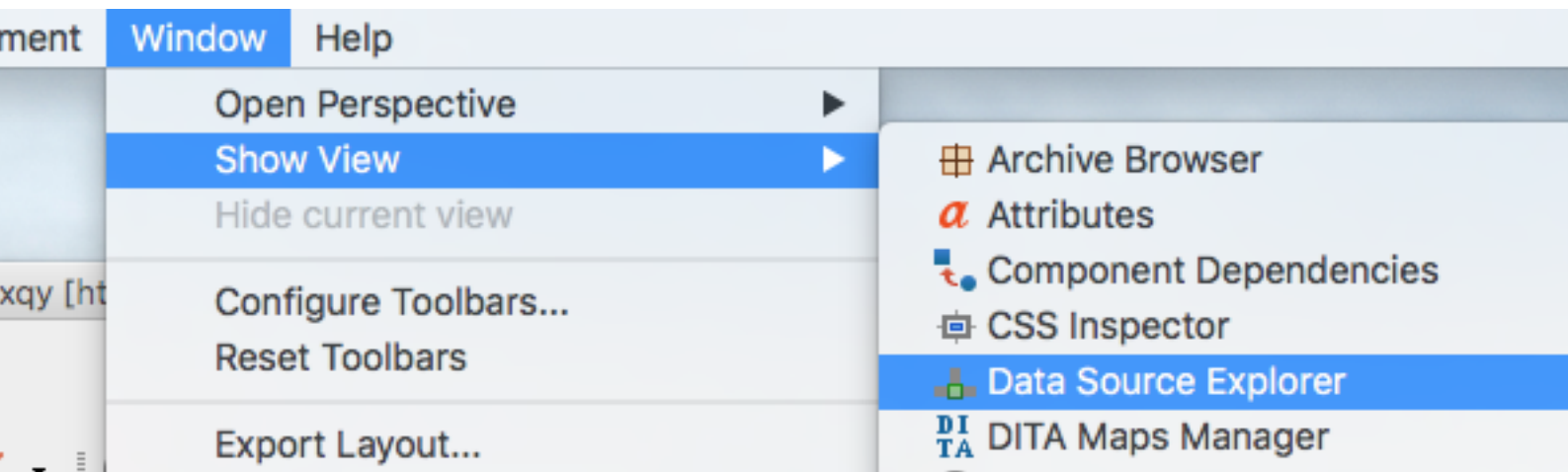


Bild 5: Data Source Explorer im Oxygen XML Editor öffnen

und konfiguriert eine neue Datenquelle:

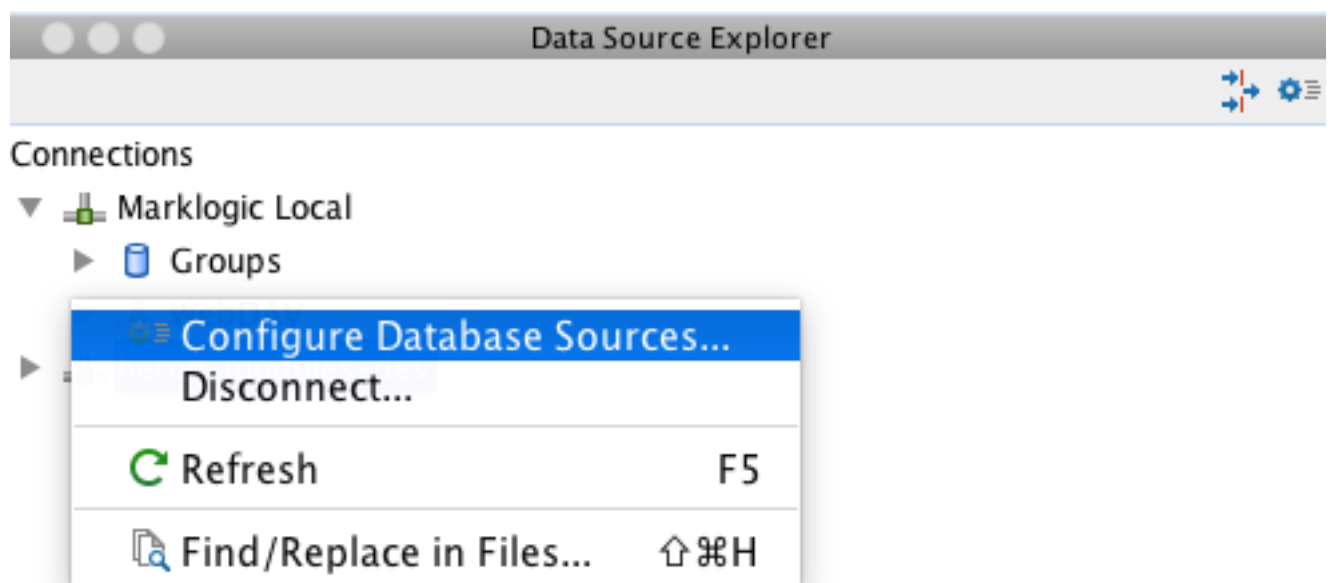


Bild 6: Neue Datenquelle im Oxygen XML Editor konfigurieren



## 3 Sonstiges

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

### 3.1 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
=> merge conflicts
```

#### TROUBLE SHOOTING

=====

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

## 3.2 XML Editoren

Der XSLT Stylesheet-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 <sup>[XM]</sup>	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 <sup>[EP]</sup>	Arbortext XML Editor, früher bekannt als EPIC ist sehr betagt. Ich hatte damit im Bereich Luftfahrt/Verteidigung zu tun. Bekannter-massen 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 <sup>[OX]</sup>	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 <sup>[FX]</sup>	FontoXML sieht schon fast aus wie Word. Neben der WYSIWYG/M Darstellung, kann auch die XML Struktur in einem Seitenpanel angezeigt werden.
XEditor <sup>[XE]</sup>	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 <sup>[XO]</sup>	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/>

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

## 4 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.)

Bild 1:	Pull Stylesheet
Bild 2:	Push Stylesheet
Bild 3:	Transformation des Quellbaums in den Zielbaum
Bild 4:	Schematron Validierung mit Filter
Bild 5:	Data Source Explorer im Oxygen XML Editor öffnen
Bild 6:	Neue Datenquelle im Oxygen XML Editor konfigurieren



## Fussnoten

- 1 [https://de.wikipedia.org/wiki/Darwin\\_Information\\_Typing\\_Architecture](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](https://en.wikipedia.org/wiki/Topic-based_authoring)
- 4 <http://www.tekturcms.de>
- 5 <http://www.unidex.com/turing/utm.htm>
- 6 [https://de.wikipedia.org/wiki/XSL\\_Transformation](https://de.wikipedia.org/wiki/XSL_Transformation)
- 7 W3C Seiten zu The Extensible Stylesheet Language Family (XSL)
- 8 <https://de.wikipedia.org/wiki/Verarbeitungsanweisung>
- 9 [https://de.wikipedia.org/wiki/Wireless\\_Application\\_Protocol](https://de.wikipedia.org/wiki/Wireless_Application_Protocol)
- 10 <https://de.wikipedia.org/wiki/EPUB>
- 11 [https://de.wikipedia.org/wiki/CHM\\_\(Dateiformat\)](https://de.wikipedia.org/wiki/CHM_(Dateiformat))
- 12 <https://www.ibm.com/developerworks/library/os-echelp/index.html>
- 13 <https://en.wikipedia.org/wiki/JavaHelp>
- 14 <https://de.wikipedia.org/wiki/FrameMaker>
- 15 <https://de.wikipedia.org/wiki/S1000D>
- 16 <https://www.i4icm.de/forschungstransfer/pi-mod/>
- 17 [https://de.wikipedia.org/wiki/Journal\\_Article\\_Tag\\_Suite](https://de.wikipedia.org/wiki/Journal_Article_Tag_Suite)
- 18 [https://de.wikipedia.org/wiki/Text\\_Encoding\\_Initiative](https://de.wikipedia.org/wiki/Text_Encoding_Initiative)
- 19 <http://surguy.net/articles/client-side-svg.xml>
- 20 <http://jackrabbit.apache.org/jcr/node-type-visualization.html>
- 21 <http://argouml.tigris.org>
- 22 <http://butterflycode.sourceforge.net>
- 23 <https://www.javaworld.com/article/2073998/java-web-development/generate-javabean-classes-dynamically-with-xsit.html>
- 24 <http://www.saxonica.com/html/documentation/sourcedocs/streaming/>
- 25 <http://exist-db.org/exist/apps/homepage/index.html>
- 26 <https://de.marklogic.com/>
- 27 [https://www.oxygenxml.com/xml\\_editor/xquery\\_builder.html](https://www.oxygenxml.com/xml_editor/xquery_builder.html)
- 28 <http://cs.au.dk/~amoeller/XML/querying/flwexp.html>
- [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



Tekur CCMS ist ein web-basiertes Component Content Management System und befindet sich noch in der Entwicklung. Blog: [www.tekturcms.de](http://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 Elementweise 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.