Die wundervolle Welt des XSLT: Refreshing

- Roman Bleier
- Torsten Roeder
- Patrick Sahle

Winter School: Digitale Editionen - Masterclass. Wuppertal, 23.02.2021

Willkürlich ausgewählte Themen

1. **Man liebt es oder man hasst es:** Namensräume (T)
2. **Alle Wege führen nach Rom:** Programmierstile (P)
3. **Was auch immer das Problem ist:** xsl:for-each-group (P)
4. **Nimm doch gleich alles:** copy() und copy-of() (T)
5. **Ich liebe Dich so, wie Du bist, aber:** Identity-Transform (T)
6. **X-Jonglage:** Template-Kaskaden (T)
7. **Mehr Eleganz bitte:** Named Templates und Funktionen (R)
8. **Das hebe ich mir auf:** Variablen (R)
9. **Das feine Skalpell:** String-Power mit regex (R)
10. **Die Weltherrschaft:** doc() und co. (P)

Man liebt es oder man hasst es

Namensräume

Torsten

Murphys Namespace-Law (frei erfunden):

»Wenn bei XSLT etwas nicht funktioniert und Du nicht verstehst warum, liegt es am Namensraum. Du suchst aber mindestens zwei Stunden woanders nach dem Fehler (oder so lange, wie Du benötigst, um das Skript komplett nochmal zu schreiben).«

- Jedes Dokument hat einen Namensraum.
  - Wenn keiner explizit definiert ist, heißt der Namensraum "".
- An jedem Elementknoten können beliebig viele weitere Namensräume definiert werden.
  - Das Schema könnte dabei ein Veto einlegen.
- Ab XSLT 2.0 kann für XPath ein Default-Namespace angegeben werden.
  - In XSLT 1.0 müssen hingegen explizite Namespaces verwendet werden.

Namensräume: Bugfixing

😩 Warum zum Kuckuck steht im Zieldokument überall `xmlns=""`?

- ist im Transformationsdokument der Default-Namensraum (für das Zieldokument) definiert?
- z.B. `xmlns="http://www.tei-c.org/ns/1.0"` unter `xsl:stylesheet` einfügen

😖 Warum steht im Zieldokument überall `xmlns="http://www.tei-c.org/ns/1.0"` ?

- ist im Quelldokument der Default-Namensraum definiert? (ggf. externe Quellen prüfen!)
- ist im Transformationsdokument der XPath-Default-Namespace definiert?

😭 Warum liefert mein XPath auf dem externen Dokument ein leeres Ergebnis?

- *Streberlösung:* alle Namespaces importieren (s.o.) und entsprechend im XPath angeben
- *quick and dirty:* jedem Element des XPath "`*:`" voranstellen

Namensräume: Checkliste

- Namespaces der Quelldokumente
  - ist der Default-Namespace gesetzt?
  - werden Namespaces aus anderen Dokumenten importiert?
- XPath-Default-Namespace des Transformationsdokuments
  - ggf. importierte Namespaces nochmal prüfen
  - Namespaces von importierten Attributen?
- Namespaces der Zieldokumente
  - meistens im Transformationsdokument unter `xsl:stylesheet`
  - ggf. auch unter `xsl:result-document` oder in `xsl:variable`
  - wird ggf. durch Import von Daten aus anderen Dokumenten definiert (s.o.)

Alle Wege führen nach Rom

Programmierstile

Patrick

Bei XSLT gibt es zwei grundlegend unterschiedlich ausgerichtete Strategien, Paradigmen, Stile …

1. Das Pull-Paradigma
   - Ein zentrales template *holt* gezielt die ~~Daten~~ Knoten zur weiteren Verarbeitung
   - Kennzeichen: ein oder nur wenige templates; viele for-each, if, choose
2. Das Push-Paradigma
   - Während der Prozessor durch den Datenbaum geht, *schiebt* er die Knoten durch die bereitstehenden templates
   - Kennzeichen: zahlreiche templates, die mit xsl:apply-templates verbunden sind; Vermeidung von for-each, if, choose

Programmierstile: Pull vs. Push

1. Das Pull-Paradigma
   - Ein zentrales template *holt* die Daten Knoten zur weiteren Verarbeitung
   - Kennzeichen: ein oder nur wenige templates
   - Vorteile:
     - Anfänglich intuitiver, scheinbar weniger komplex
     - Einfach inkrementell erweiterbar
     - Ähnlich zu anderen Programmiersprachen
   - Nachteile
     - Auf die Dauer komplexer und uneleganter
     - Einzelne Teile schlechter nachnutzbar / austauschbar
     - Entspricht eigentlich nicht der Logik von XML und XSLT

Programmierstile: Pull vs. Push

2. Das Push-Paradigma
   - Während der Prozessor durch den Datenbaum geht, *schiebt* er die Knoten durch die bereitstehenden templates
   - Kennzeichen: zahlreiche templates, die mit xsl:apply-templates verbunden sind
   - Nachteile:
     - Man sollte vorher einen Plan haben
     - Erfordert konsequentes Denken in Bäumen und processing flows
   - Vorteile:
     - “natural flow” - entspricht gut der Grundlogik von XML und XSLT
     - Letztlich modularer und dadurch übersichtlicher und besser nachnutzbar

Programmierstile: Pull vs. Push

In Wirklichkeit wird man am Ende fast immer mit einer Mischung beider Stile arbeiten …

- Vorab: natürlich kommt es auf den Einsatzzweck an! Detaillierte, spezielle Aufgaben: Pull - allgemeine, umfassende Aufgaben: Push
- Innerhalb von templates gibt es häufig weitere Pull-Aktionen
- Ein Mischansatz: xsl:call-template und xsl:with-param
- Wundersam: xsl:next-match (wie apply-templates, aber es lässt mehrere passende templates ausführen)

Programmierstile: Pull vs. Push

Nachlesen ...

- Push, Pull, Next!, Bob DuCharme, 2005
- Push and Pull Design, in: XSLT, part 2: Advanced features, David Birnbaum, 2018
- Push vs. Pull Stylesheets, in: XML - Hands-On XSLT & Co, Alex Düsel, 2019

Was auch immer dein Problem ist …

for-each-group

Patrick

1. Wir kennen alle xsl:for-each = tue etwas für jedes Element eines Knotensets
2. xsl:for-each-group macht im Prinzip das gleiche, nur mächtiger und eleganter
   - select ⇒ XPath-Ausdruck für das Knotenset
   - group-by ⇒ sortiert die Elemente nach einem Kriterium in Gruppen
   - man kann dann für jede Gruppe etwas tun; und dabei dann die Mitglieder der Gruppe verarbeiten …
   - current-grouping-key() ⇒ was war nochmal das Kennzeichen dieser Gruppe?
   - current-group() ⇒ die aktuelle Gruppe mit all ihren Elementen

Disclaimer: vermutlich lässt sich das Gleiche noch eleganter durch eine template-Kaskade lösen

for-each-group

Ein Beispiel ...

```
<xsl:for-each-group select="//person" group-by="substring(persName/surname,1,1)">
    <xsl:sort select="current-grouping-key()"/>
    <xsl:value-of select="current-grouping-key()"/>
    (<xsl:value-of select="count(current-group())"/> Personen)
    <xsl:for-each select="current-group()">
        <xsl:value-of select="surname"/> <!-- usw. -->
    </xsl:for-each>
</xsl:for-each-group>
```

Nimm doch gleich alles …

copy vs. copy-of

Torsten

- `<xsl:copy-of>` kopiert den aktuellen Kontext …
  - … und alles, was drin ist!
  - hilfreich, wenn man ein Element 1:1 übertragen will
  - erlaubt `@select` → kann also Teile aus anderen Dokumenten “importieren” (*pull*)
  - Anwendungsbeispiel: Daten aus TEI-Header oder mit `document()` “ranholen”

- `<xsl:copy>` kopiert den aktuellen Kontext …
  - … aber *ohne* den Inhalt!
  - hilfreich, wenn man ein Element modifizieren will → “Trojanisches Pferd”
  - erlaubt kein `@select` → nur kontextbezogen möglich (*push*)
  - Anwendungsbeispiel: “Identity Transform”

Ich liebe Dich, so wie Du bist, aber …

Identity Transform

Torsten

- interessant für die Verarbeitung eines XML-Dokuments …
  - wenn die Struktur intakt bleiben soll
  - aber einzelne Elemente erweitert oder modifiziert werden sollen
- Verarbeitungslogik:
  - grundsätzlich soll alles genauso erhalten bleiben, wie es ist
  - nur einzelne Knoten (identifiziert durch XPath) austauschen
  - anders gesagt: »kopiere alles außer …« (*push*)
- hilfreich dabei:
  - XPath-Ausdruck, der alles findet: Text, Attribute, Elemente, Kommentare, …
  - ein Befehl, der all dies genau so lässt, wie es ist

Identity Transform

kurz zur Auffrischung: XSL-Verarbeitung

- der XSL-Prozessor geht das Quelldokument durch und hat dabei die “Templates” im Hinterkopf
- `<xsl:template match=”...”>`
  - wird ausgeführt, wenn der XSL-Prozessor ein Element findet, das zu `@match` passt
- `<xsl:apply-templates select=”...”>`
  - wendet alle bekannten Templates rekursiv an der aktuellen Stelle des Quelldokuments an
  - lässt sich gezielt auf einzelne Knoten mit `@select` ansetzen (optional)
- Effekte:
  - Wiederverwendung desselben Templates an verschiedenen Stellen des Dokuments
  - ermöglicht template-orientiertes Programmieren (*push*)
  - Verarbeitung nach dem “Schachtelprinzip”

Identity Transform

```
01 <!-- identity transform -->
02 <xsl:template match="@* | node()">
03   <xsl:copy>
04     <xsl:apply-templates select="@* | node()"/>
05   </xsl:copy>
06 </xsl:template>
07
08 <!-- IDs einfügen -->
09 <xsl:template match="TEI/text//persName">
10   <persName ref="{generate-id()}">
11     <xsl:apply-templates select="@* | node()"/>
12   </persName>
13 </xsl:template>
14
15 <!-- Zeilenumbrüche entfernen -->
16 <xsl:template match="TEI/text//lb"/>
```

02 Attribute matchen: "`*`" findet nur Elemente, `node()` gilt nur für Elemente und Text.

03 `<copy>` kopiert den Kontext. (In Oxygen: “RELAX NG Standard-Attribute” abschalten!)

09 matcht jedes `<persName>` im TEI-Text.

10 automatisch generierte ID einfügen.

11 *identity transform* aufrufen, falls der Kontext weitere Attribute/Kindelemente hat.

16 `<lb>` wird bei Ausgabe übergangen.

Spoiler: mit XSLT 3.0 geht’s noch leichter …

X-Jonglage

Template-Kaskaden

- “Kaskade” in XSL: Verarbeitungsreihenfolge der Templates
- dabei gibt es unterschiedliche Prioritäten!
  - abhängig vom XPath
  - je spezifischer der XPath-Ausdruck ist, desto höher gewichtet
- Herausforderungen
  - konkurrierende Templates vermeiden (XSL-Prozessor bekommt sonst die Krise)
  - Templates wiederverwenden (Redundanz verringern)
- Möglichkeiten der Einflussnahme
  - XPath anpassen
  - Priorität manuell bestimmen
  - Modi verwenden

Template-Kaskaden

Welches XSL-Template gewinnt, wenn mehrere XPaths “matchen” würden?

- `match="test[@test='test']"`
- `match="test[@test]"`
- `match="test"`
- `match="*"`
- `match="node()"`

↑

spezifischer = höhere Priorität

allgemeiner = niedrigere Priorität

↓

ABER bei verzweigten Pfaden wie `match="test/test"` etc. …

- führen mehrdeutige Angaben zum Abbruch
- müssen XPaths komplementär formuliert werden
- kann die Priorität mittels `@priority` gesteuert werden
- zudem werden importierte Templates ebenfalls runtergestuft

Template-Kaskaden

- Szenario:
  - verschiedene Ausgabeformate für eine Edition herstellen
  - HTML diplomatisch mit Zeilenumbrüchen
  - HTML diplomatisch ohne Zeilenumbrüche
  - HTML normalisiert mit Zeilenumbrüchen
  - HTML normalisiert ohne Zeilenumbrüche
  - Plaintext diplomatisch mit Zeilenumbrüchen
  - etc. p.p.p.p.
- Horror: Für jedes Ausgabeformat ein einzelnes Skript (extrem redundant)
- Template-Kaskade so designen, dass jeder Aspekt getrennt gerendert wird
  - 1: HTML/Plaintext, 2: diplomatisch/normalisiert, 3: mit/ohne Zeilenumbrüche
  - Weichenstellung mithilfe von `@mode`

Mehr Eleganz bitte: Named Templates und Funktionen

Verwendung von Funktionen in der Programmierung:

- Um Programmcode zu strukturieren
- Um Programmcode (an mehreren Stellen im Programm) aufrufbar zu machen
- Vermeidung von Redundanz im Code
- Dadurch entsteht eine bessere Lesbarkeit und Wartbarkeit

Bestandteile von Funktionen:

- Funktionsname
- Übergabeparameter (optional)
- Ein Rückgabewert

Roman

Mehr Eleganz bitte: Named Templates und Funktionen

- Seit XSLT 1.0 haben Named Templates ähnliche Funktion wie Funktionen in anderen Programmiersprachen
- Verwendung: um XSLT-Code zu strukturieren und mehrfach aufrufbar zu machen

```
<xsl:call-template name="footer"/>

<xsl:template name="footer">
    <footer>...</footer>
</xsl:template>
```

Mehr Eleganz bitte: Named Templates und Funktionen

Named Templates können Parameter mitgegeben werden

```
<xsl:call-template name="footer">
        <xsl:with-param name="footerText"></xsl:with-param>
</xsl:call-template>

<xsl:template name="footer">
    <xsl:param name="footerText"></xsl:param>
    <footer> <xsl:value-of select="$footerText"/> </footer>
</xsl:template>
```

Mehr Eleganz bitte: Named Templates und Funktionen

- Seit XSLT 2.0 gibt aber noch eine anderen Möglichkeit Funktionen zu erzeugen
- `<xsl:function>` definiert eine Stylesheet-Funktion, die über einen XPath-Ausdruck aufgerufen wird
- `<xsl:function>` können (wie auch bei Named Templates) Parameter mitgegeben werden
- `<xsl:function>` hat einen Rückgabewert, der Typ des Rückgabewertes wird über das @as festgelegt.
- Hauptunterschied zum Named Template und `<xsl:function>` ist die Art des Aufrufs: `<xsl:call-template>` / im XPath-Ausdruck
- Unterschied in der Verwendung: Named Templates schaffen neue Knoten im Result Tree / während `<xsl:function>` auf der XPath-Ebene arbeitet => Adressierung von Knoten, Stringoperations und berechnen von Werten

Mehr Eleganz bitte: Named Templates und Funktionen

Beispiel einer einfachen Funktion:

```
<xsl:function name="myfunc:add" as="xs:integer">
        <xsl:param name="x" as="xs:integer"/>
        <xsl:param name="y" as="xs:integer"/>
        <xsl:sequence select="$x + $y"/>
</xsl:function>

<xsl:value-of select="6 + myfunc:add(1,3)"/>
```

Mehr Eleganz bitte: Named Templates und Funktionen

Roman

Bevor man sich seine eigenen Funktionen baut, sollte man in den existierenden Erweiterungs-bibliotheken nachschauen. Vielleicht gibt es die Funktion ja schon.

XSLT Erweiterungsbibliotheken (extension functions):

- EXSLT, von der EXSLT Initiative, Funktionen und Named Templates, Namespace: http://exslt.org/...
- Saxon Extension Functions, von Saxonica, Namespace: http://saxon.sf.net/
- EXPath Erweiterung, von der EXPath Iniziative, Namespace: http://expath.org/…

Mit den Named Templates und `<xsl:function>` hat aber jeder das Werkzeug sich eigene Bibliotheken zu bauen...

value-of vs. sequence

- zwei unterschiedliche Ausgabe-Methoden in XSL
- `<xsl:value-of>` operiert auf Textknoten
  - alle Textknoten des selektierten Baums aneinandergereiht
  - nervt möglicherweise wegen Whitespace
  - nervt möglicherweise wegen möglicher Siblings
- `<xsl:sequence>` operiert auf einem XML-Baum
  - nimmt alle Knoten des selektierten Baums mit
  - in Kombi mit `<xsl:variable>` lassen sich Bäume in Variablen bauen
  - plus in Kombi mit `document()` aus anderen Dokumenten direkt einlesen
- was wird gebraucht? Text oder Baum?

Das hebe ich mir auf: Variablen

- Variablen werden dazu verwendet, um Werte zu speichern und unter einem Namen abrufbar zu machen
- Ähnlich wie Variablen in anderen Programmiersprachen (Betonung auf ÄHNLICH). Gravierender (und verwirrender) Unterschied:
  - Man kann den Inhalt von Variablen in XSLT später nicht mehr ändern
- Aber: Man kann ganze XML-Bäume in Variable stecken!
- Gültigkeitsbereich einer Variable (variable scope):
  - Global: die Variable ist Kindelement von <xsl:stylesheet> definiert
  - Lokal: die Variable ist z.B. in einem Template

Roman

Das hebe ich mir auf: Variablen

`<xsl:variable>` vs `<xsl:param>`

- `<xsl:param>` wird zur Definition von Parametern, Parameter sind Werte die an Stylesheets, Templates und Funktionen übergeben werden
- Seit XSLT 2.0 können Datentypen über das @as für Variablen und Parameter definiert werden
- Entweder einfache Datentypen (xs:string, xs:int, xs:date, etc.) oder Knoten (element(), attribute(), text(), node())

```
<xsl:param name="backgroundColor" as="xs:string"/>
```

Das hebe ich mir auf: Variablen

Speichern von einfachen Datentypen:

```
<xsl:variable name="backgroundColor" as="xs:string" select=" '#F0F8FF' "/>
<xsl:variable name="monate" as="xs:integer" select="12"/>
```

Speichern von Elementen und XML-Bäumen:

```
<xsl:variable name="externesDokument" select="t:teiHeader"/>

<xsl:variable name="externesDokument" select="doc('uri')"/>
```

Das hebe ich mir auf: Variablen

Speichern von Sequenzen in Variablen:

```
<xsl:variable name="OneToTen" as="xs:integer*" select="1 to 10"/>

<xsl:variable name="months" as="xs:string*"
        select="('January', 'February', 'March', 'April',
        'May', 'June', 'July', 'August',
        'September', 'October', 'November',
        'December')"/>
```

`<xsl:value-of select="$months[3]"/>` würde das Ergebnis “March” produzieren

Das feine Skalpell: String-Power mit regex

Reguläre Ausdrücke werden dazu verwendet um Muster in Texten zu erkennen. z.B.:

- \d{2}-\d{2}-\d{4} um Zeichenketten wie 22-12-2018 oder 88-99-9999 zu finden
- [A-Z][a-z]+\s[A-Z][a-z]+ um einfache Namen nach dem Muster Xxxxx Xxxxxx zu finden

Symbole und Syntax für Regex:

\w	alphanumerisches Zeichen oder ein Unterstrich = [a-zA-Z0-9_]
\d	eine Ziffer = [0-9]
.	jedes Zeichen außer \n.
\n	neue Zeile
\s	Leerzeichen, Tabulatorzeichen, etc.
[abc]	Zeichenkette, eines der Zeichen in der Klammer kann vorkommen

Roman

Das feine Skalpell: String-Power mit regex

Häufigkeit des Vorkommens von Elementen

*	Null oder mehr Vorkommen des vorhergehenden Elements
+	Ein oder mehr Vorkommen des vorhergehenden Elements
?	Null oder ein Vorkommen des vorhergehenden Elements
{Wert}	Wie oft soll ein Element vorkommen z.B. zwei Ziffern \d{2}
{min,max}	Platzhalter für min bis max Vorkommen
^	Anker für Anfang der Zeichenkette
$	Anker für Ende der Zeichenkette
()	Bilden von Teilsuchmustern/Gruppen, z.B: jeweils eine Gruppe für Tag, Monat, Jahr: (\d{2})-(\d{2})-(\d{4})

Das feine Skalpell: String-Power mit regex

- XPath Funktionen:
  - `matches(subject, pattern, flags)` → Rückgabe? Boolean!
  - `replace(subject, pattern, replacement, flags)` → Rückgabe? String!
  - `tokenize(subject, pattern, flags)` → Rückgabe? Sequenz!
- XSLT Elemente
- `<xsl:analyze-string>` (XSLT 2.0)
- `<xsl:matching-substring>`, `<xsl:non-matching-substring>`
- XPath Funktion: `regex-group()`

Das feine Skalpell: String-Power mit regex

Einfaches Beispiel:

```
<xsl:analyze-string select="text" regex="\d+">
      <xsl:matching-substring>
         <zahl><xsl:value-of select="."/></zahl>
      </xsl:matching-substring>
      <xsl:non-matching-substring>
            <xsl:value-of select="."/>
      </xsl:non-matching-substring>
</xsl:analyze-string>
```

Das feine Skalpell: String-Power mit regex

Matchen von Gruppen:

```
<xsl:analyze-string select="beispiel" regex=" (\d\d)\.(\d\d)">
    <xsl:matching-substring>
        <zahl>Zahl 1: <xsl:value-of select=" regex-group(1)"/></zahl>
        <zahl>Zahl 2: <xsl:value-of select=" regex-group(2)"/></zahl>
    </xsl:matching-substring>
    <xsl:non-matching-substring>
         <rest><xsl:value-of select="."/></rest>
    </xsl:non-matching-substring>
</xsl:analyze-string>
```

Bsp. von https://www.data2type.de/xml-xslt-xslfo/xslt/xslt-referenz/matching-substring/

Die Weltherrschaft: doc(🌍)

Patrick

- Mit der XPath-Funktion **doc('ein URI')** kann ein externe Ressource (Datei) als Baum in die laufende Verarbeitung eingebunden werden
  - document() ist eine sehr ähnliche XSLT-Funktion
  - natürlich lassen wir doc() in Schleifen laufen; man kann auch 10.000 externe Dokumente verarbeiten; Parser sind gutmütige Wesen … (Achtung: manche Dienste sind es nicht)
  - **collection()** liefert ein ganzes set an Ressourcen (Dateien) als ein zusammengefasster Baum zurück. Damit kann man z.B. ganze Verzeichnisse verarbeiten. Man kann auch angeben, dass nur bestimmte Dateiarten verarbeitet werden sollen.
- Zwei Beispiele
  - Liebe Wikipedia, wieviele Einwohner hat Wuppertal?
    - doc('https://de.wikipedia.org/wiki/Wuppertal')//tr[contains(td[1],'Einwohner')]/td[2]
  - <xsl:copy-of select="collection('Verzeichnisname/?select=*.xml')"/>
    - → dies ist ein vollständiger File-Merger (alle XML-Dateien in einem Verzeichnis werden zusammengeführt)

Der Spiegel zu doc()

Mit doc() können beliebig viele externe Ressourcen in das eigene Stylesheet eingebunden werden (die Welt zu Gast bei Dir)

Mit **xsl:result-document** werden explizit Ergebnisdokumente geschrieben

- Die Datei (ihr gesamter Inhalt) ist innerhalb von xsl:result-document
- Wichtiges Attribut: href="Dateiname" (und Speicherort)
- Es gibt zahlreiche weitere Attribute, die die Eigenschaften der Ergebnisdatei beschreiben (encoding, method, type, indent und Attribute zur Namensraumbehandlung 😈)
- Wir schreiben result-doc in Schleifen oder in templates → beliebige Zahl an Ergebnisdokumenten

… alles gut, also ...

Die Welt ist schlecht ...

doc() funktioniert nur dann reibungslos, wenn die entfernte Ressource wohlgeformtes XML (also z.B. XHTML!) ist. Leider sind die allermeisten Dinge im Internet zwar HTML, aber nicht XHTML, was also sollen wir tun?

- doc-available() - prüft vorab, OB die Ressource erfolgreich zu einem Baum geparsed werden kann
- unparsed-text() - nimmt klaglos jeden HTML-/Text-Schrott an, man kann sich dann mit string-Funktionen die Teile rausschneiden, die man braucht (meistens scheitert HTML-Wohlgeformtheit an ein paar Kleinigkeiten), bis man einen wohlgeformten Baum hat
- Man könnte über die Benutzung von anderen Helferlein nachdenken, die Seiten vorab aufräumen (tidy)
- Twitter
- Manche Dienste liefern vorne HTML aus und haben hinten eine API, die XML ausliefert

Die Welt ist schlecht, aber …

- unparsed-text() - nimmt klaglos jeden HTML-/Text-Schrott an, man kann sich dann mit string-Funktionen die Teile rausschneiden, die man braucht (meistens scheitert HTML-Wohlgeformtheit an ein paar Kleinigkeiten), bis man einen wohlgeformten Baum hat
  → https://twitter.com/patrick_sahle/status/1352169831131451394
- Man könnte … über die Benutzung von anderen Persern oder Helferlein nachdenken, die Seiten
  - andere Techniken nutzen → https://twitter.com/szimr/status/1352190894028574722 & https://twitter.com/szimr/status/1352554095807258625
  - einen anderen Parser nehmen → https://twitter.com/bibliothekapp/status/1353077790761881603
  - eleganteres XSLT schreiben → https://twitter.com/chrlueck/status/1359621424915349508
- Manche Dienste liefern vorne HTML aus und haben hinten eine API, die XML ausliefert
  - https://www.wikidata.org/wiki/Q65145325 ist nicht wohlgeformt
  - https://www.wikidata.org/w/api.php?action=wbgetentities&ids=Q65145325&format=xml ist wohlgeformt

Willkürlich ausgewählte Themen

1. **Man liebt es oder man hasst es:** Namensräume (T)
2. **Alle Wege führen nach Rom:** Programmierstile (P)
3. **Was auch immer das Problem ist:** xsl:for-each-group (P)
4. **Nimm doch gleich alles:** copy() und copy-of() (T)
5. **Ich liebe Dich so, wie Du bist, aber:** Identity-Transform (T)
6. **X-Jonglage:** Template-Kaskaden (T)
7. **Mehr Eleganz bitte:** Named Templates und Funktionen (R)
8. **Das hebe ich mir auf:** Variablen (R)
9. **Das feine Skalpell:** String-Power mit regex (R)
10. **Die Weltherrschaft:** doc() und co. (P)