cheatsheet

Sarah Lang

3. Februar 2019

Inhaltsverzeichnis

1 Was ist Annotation?

Annotation und Erschließung

= Annotation mit dem Ziel der Erschließung

irgendwelche oder alle (?) (Meta-)Daten sammeln vs. gezielt spezifische Daten verfügbarmachen

Was ist Annotation?

Zusatzinfo ● Metadaten ● Bedeutung kodieren ● Strukturmerkmale, Wissen und Erkenntnisse (implizite Strukturen) für die Maschine explizit machen für Verarbeitung und Nachnutzung.
Im Druckwesen: inhaltliche Korrektur und typographische Anweisungen (Druckvorbereitung) ● **Computer:** Einbettung von Markup in elektronische Dokumente (log. Struktur beschreiben)

Visual vs. Generic Markup = Text mit Auszeichnung (Aussage über Bedeutung: Semantik, Struktur, Beziehungen – William W. Tunnicliffe (1967)

Visual = presentational & procedural

Generic deskriptiv

```
<name type="person">J. W. v. Goethe</name>
```

→ explizite Marker zur Unterscheidung für die Weiterverarbeitung

Bennenen, charakterisieren, annotieren (auf formalisierte Art und Weise)

Markup = Annotation = Kodierung

Stufen der Annotation

Basisannotation Formale Kriterien der Textgestalt, z.B. Überschriften, Paragraphen. Hinzufügen von Metadaten (bibliographisch, z.B.).

Tiefenerschließung Übergang formal-inhaltlich: Personen, Orte, *Named Entities*, GND – Zurückgreifen auf Standards und Normdateien zur Vernetzung vieler Projekte.

Übergeordnete Forschungsfrage Redeanteile der Monster? Wie wird charakterisiert? Kann man Spannung / Grusel 'messen'? Gibt es Signalwörter? Visualisierung der Ergebnisse

Markup / Annotation

Annotation = 'Auszeichnung' = Mark-up

> Eine Auszeichnungssprache (*markup language*, abgekürzt ML) ist eine maschinenlesbare Sprache für die Gliederung und Formatierung von Texten und anderen Daten. Der bekannteste Vertreter ist die *Hypertext Markup Language* (HTML), die Kernsprache des World Wide Webs.
>
> Mit Auszeichnungssprachen werden Eigenschaften, Zugehörigkeiten und Darstellungsformen von Abschnitten eines Textes (Zeichen, Wörtern, Absätzen usw. – "Elementen") oder einer Datenmenge beschrieben. Dies geschieht in der Regel, indem sie mit *Tags* markiert werden. […] mit der *Standard Generalized Markup Language* (SGML) empfohlene "Trennung von Struktur und Darstellung". (Wikipedia)

maschinenlesbare 'Auszeichnung'

SMGL **HTML** **XML** **…**

darstellend vs. beschreibend / semantisch:

- z.B. 'Schriftgröße 14' vs. 'Überschrift' (=Typus). (explizit vs. implizit)
- Textformatierung vs. Textbedeutung
- procedural, representative, descriptive / conceptual (sematisch)
- WYSIWYG-Textverarbeitung vs. WYSIWYM
- Vorteil, z.B. bei langen Dokumenten: Aussehen des Elements 1x umdefinieren
- verschiedene Repräsentationen eines Dokuments: beschreibend, dargestellt
- Webbrowser 'rendern' HTML: in 2 Sichten verfügbar (als Text und dargestellt)

Praktische Tipps

Tools zum 'Switchen' zwischen Markups/Markdown: z.B. Pandoc oder OxGarage (TEI-Fokus).
Download-Möglichkeiten überprüfen – evtl. gibt es das gewünschte Dokument bereits in Basis-Annotation? Hier allerdings potentiellen Korrektur-Aufwand berücksichtigen – falls Anpassungsbedarf: oft ist man mit einer .txt-Datei von Hand schneller. (Download oder per XSL 'herstellen')

Binäre Dokumentenformate wie *.doc*, *.pdf*, *.dvi* (TEX-Ausgabeformat) ≠ Auszeichnungssprachen.
Ziel: **Implizites explizieren**.

2 Annotation im Alltag

SGML

Trennung von Inhalt und Darstellung

Wie XML: Metasprache HTML und XML (Nachfolger von SGML genannt → SGML-basierend) ● Metasprache zur Definition von Auszeichnungssprachen für Dokumente ● *Genormte Verallgemeinerte Auszeichnungssprache* (en: **Standard Generalized Markup Language**

.SGML

RTF

Rich Text Format Microsoft 1987 ● Austauschformat zwischen Textverarbeitungsprogrammen (versch. Hersteller und Betriebssysteme). ● enthält im Gegensatz zu 'plain text' Markup zur Textformatierung

.RTF

```
{\rtf1
Guten Tag!
\line
{\i Dies} ist \b{\i ein
\i0 formatierter \b0Text}.
\par
\b Das \b0Ende.
}
```

JSON

JavaScript Object Notation

.JSON

ausgesprochen wie 'Jason' ● kompaktes Datenformat ● lesbar ● key-value-Paare ● Verschachtelung

Alternative zu XML? Unterschiede

XML = Struktur-beschreibend, JSON = Syntax-Konvention (nicht deklarativ) ● JSON: Definition von Instanzen strukturierter Daten ● JSON = sehr flexibel ● 'lightweight': geringer Overhead, für wenige Daten in strikter Key-Value-Struktur lesbarer ● valides JavaScript, per `eval()`-Funktion direkt als JS-Objekt umsetzbar

Fazit JSON im Vorteil, wo simple key-value-Paare (**'Einfachheit'**). XML hat und erlaubt mehr **Komplexität**.

XML = Auszeichnungssprache

JSON = Datenaustauschformat

```
{
  "Herausgeber": "Xema",
  "Nummer": "1234-5678-9012-3456",
  "Inhaber":
  {
    "Name": "Mustermann",
    "Vorname": "Max",
    "maennlich": true,
    "Hobbys": ["Reiten", "Golfen", "Lesen"],
    "Alter": 42,
    "Kinder": [],
    "Partner": null
  }
}
```

LaTeX

.tex Textsatz mit TEX durch Lamport-Makros (= 'Shortcuts') ● LATEX liest Makros (mit sprechenden Namen) ein – in der 'Produktion' entsteht die 'Illusion' rein deskriptiver Auszeichnung, im Hintergrund sind dies allerdings nur Stellvertreter für die komplexe prozedurale Sprache TEX. ● prozedural vs. präsentational: **WYIWYG** (*what you see is what you get*, z.B. Word) vs. **WYSIWYM** (*what you see is what you mean*) LATEX-Editoren)

Befehle unten ohne spaces vor der Klammer

`\textit {Italic}` ..... *Italic* (darst.)
`\emph {Italic}` ..... *Italic* (beschr. / semant.)
`\textbf {Bold}` ..... **Bold**
`\section {Titel}` ..... Überschrift 1
`\subsection {Titel}` ..... usw.
`\href {http://a.com}{Link}` ..... 'hidden' Link
`\includegraphics {bla.png}` ..... Bild
etc.

Markdown

Markdown in 60s ● simple text formatting

.md // darstellend

`*Italic*` ..... *Italic*
`**Bold**` ..... **Bold**
`## Heading 1` ..... Überschrift 1
`#### Heading 2` ..... usw.
`[Link]{http://a.com}` ..... 'versteckter' Link
`![Image]http://url/a.png` ..... Bild
`> Blockquote` ..... Zitat
`- List` ..... Liste (schachtelbar, 2 spaces)
`* List` ..... Alternative
`1. Aufzaehlung` ..... Aufzählung
`--` ..... Trennlinie
`‘Inline code‘` ..... Code ('backticks')
`‘‘‘code block‘‘‘` ..... Codeblock

PostScript

Seitenbeschreibungssprache ● 1980er (Adobe Systems) ● Vektorgraphikformat für Drucker ● aber auch: Turing-vollständige, stackorientierte Programmiersprache ● Standard der Druckindustrie ● heute von PDF (*Portable Dokument Format*) größtenteils abgelöst (ebenfalls Adobe, Weiterentwicklung von PS) ● kann per PostScript-Druckertreiber aus allen möglichen Dokumenten erstellt werden ● in Unix per 'Ghostscript' verarbeitet ● Beschreibung von Dokumenten als skalierbare Vektorgraphiken: verlustfreie Vergrößerung **.ps // präsentational**

Beispielprogramm schreibt 'Hallo Welt!' an Postition 50,50. Per Default Koordinatensystem beginnend links unten.

```
%!
/Courier findfont     % Schriftart
20 scalefont          % Schriftgröße 20
setfont               % festlegen
50 50 moveto          % (50, 50)=Schreibposition
(Hallo Welt!) show    % Text ausgeben
showpage              % Seite ausgeben
```

3 Annotation mit XML

XML: eXtensible Markup Language

W3Schools-Tutorial

Paradigma der Trennung von Form & Inhalt

XML: Metasprache

.XML

RSS **MathML** **GraphML** **XHTML**

XAML **SOAP** **RDF** **KML** **OSM**

Scalable Vector Graphics (SVG) (z.B.)

> Die **Extensible Markup Language** (dt. *Erweiterbare Auszeichnungssprache*), abgekürzt XML, ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten im Format einer Textdatei, die sowohl von Menschen als auch von Maschinen lesbar ist.
>
> XML wird auch für den plattform- und implementationsunabhängigen Austausch von Daten zwischen Computersystemen eingesetzt, insbesondere über das Internet, und wurde vom World Wide Web Consortium (W3C) am 10. Februar 1998 veröffentlicht. […] **XML ist eine Metasprache,** auf deren Basis durch strukturelle und inhaltliche Einschränkungen anwendungsspezifische Sprachen definiert werden. Diese Einschränkungen werden entweder durch eine **Document Type Description (DTD)** oder durch ein **XML Schema** ausgedrückt. (Wiki)

XML-Familie

XML strukturierte Datenbeschreibung

XPath Navigation in XMLs

XML Schema striktes Datenmodell

XSL eXtensible Style Language

XSLT Transformation von XML-Dokumenten

XSL-FO formatierte Ausgabe (z.B. für Druck)

XQuery XML-Datenbank-Abfragesprache

und mehr

XML-Vokabularien

(X)HTML Hypertext Markup Language ● **EAD** Encoded Archival Description ● **TEI** Text Encoding Initiative ● **CEI** Charters Encoding Initiative ● **MEI** Music Encoding Initiative ● **SVG** Scalable Vector Graphics ● **MathML** ● **CML** Chemical Markup Language, …

XML: eXtensible Markup Language

deskriptiv ≠ prozedural ● erweiterbar (nicht wie bei HTML): kein fixiertes Tag-Set ● UTF-8 kodiert

Bedeutung der Daten > Darstellung

expliziert Implizites oder eine Interpretation ● Hinzufügen von (Meta-)Information ● maschinelle Weiterverarbeitung ● Standard für Beschreibung und Austausch von Daten

universelle Metasprache

Nachteile 'geschwätzig', Overlapping-Problem, langsam verglichen mit JSON und relationalen Datenbanken – hat aber auch Features, die die nicht haben (& für GeWi sehr wichtig sind)

menschen- und maschinenlesbar

unabhängig von Ausgabeformat (Papier, Bildschirm)

XML-Regeln

Prolog

`<xml version="1.0" encoding="utf-8">` ..... XML-Deklaration

`<?xsl-stylesheet type="text/xsl" href="mein.xsl"?>` . Verarbeitungsanweisungen (optional)

evtl. Einbindung des Dokumentmodells (optional)
DTD, XML Schema, RelaxNG, Schematron

Entitäten 'geschützte Zeichen', da sie eine Bedeutung in der Metasprache und der Objektsprache haben; selbstdefiniert & vordefinierte, z.B.:

`<` ..... <
`>` ..... >
`&` ..... &

XML-Regeln

Prüfung auf **Wohlgeformtheit** (nach Regeln des XML-Standards) und **Validität / Gültigkeit** (wohlgeformt & Schema-Grammatik entsprechend): kann nur geparst werden, wenn wohlgeformt. Schema kann im Notfall ignoriert werden, auch wenn der Validator sich beschwert.
Es gibt Regeln für Elementnamen. Sollen hier nicht im Detail erklärt werden – Im Zweifelsfall wird man bei der Validierung erfahren, ob man sich einen Fehltritt geleistet hat.

Doppelkeks **russische Puppe**

`<key>value</key>` ..... XML als key-value-Notation

Schachtelung ● genau 1 Wurzelelement (eine äußerste Puppe)

Regeln Start- und End-Tag ● Tagnamen case-sensitive ● leere Elemente erlaubt (& abkürzbar) ● ordnungsgemäße hierarchische Schachtelung unter Wurzelknoten

Minimalbeispiel

```
<?xml version="1.0" ?>
<root>
<element attribute="value">
content
</element>
<!-- comment -->
</root>
```

Praktische Tipps

Von .docx zu TEI Worddokumente (.docx) sind XML-basiert und können, z.B. mithilfe von Pandoc oder OxGarage in einen gewünschten XML-Output umgewandelt werden. z.B. Pandoc oder OxGarage (TEI-Fokus) Mithilfe von XSLT-Transformationen kann man dieses "Roh-XML" dann noch anpassen.

Formatvorlagen Kursiv, Superskript, fett, etc. geht automatisch, der Rest muss extra angelegt werden. (Formatvorlage = .dotx-Datei aus einem Beispiel-Doc, wo alle Elemente vorkommen, erstellen. Um ein neues Dokument daraus zu erstellen: .dotx klicken).

`<p rend="lat. Zitat">dixit [...]</p>` Absatz-Formatvorlagen

`<hi rend="italic">kursiver Text</hi>` ..... Zeichen-Formatvorlagen

'Roh-XML' aus Word mit Formatvorlagen

```
<?xml version="1.0"?>
<xml>
    <p rend="la::Dichtung">
        <hi rend="la::Struktur-Heading">Schluss;;Epimythion::11</hi></p>
    <p rend="la::Dichtung">
        <hi rend="metrAnalyse">#senar::u-|u-|u//-|u/-|u-|ux</hi>
        sic totam praedam sola improbitas
        <note place="foot" xml:id="ftn15" n="15">
            <p rend="footnote text">
                Vokabel::improbitas,-atis f.: Schlechtigkeit (siehe auch.
                ↪ improbus)
            </p>
        </note>
        abstulit.
    </p>
    <p rend="de">
        <hi rend="de::Title">Die Kuh, die Ziege, das Schaf und der Löwe</hi>
    </p>
</xml>
```

4 Text Encoding Initiative

TEI Primer

Text Encoding Initiative

.XML

XML-Standard ● De-Facto-Standard in den *Digital Humanities* ● Kodierung von Druckwerken (Editionswissenschaft) ● bishin zu Linguistik

> Die Text Encoding Initiative (TEI) ist eine 1987 gegründete Organisation (seit 2000 als TEI-Konsortium organisiert) und ein gleichnamiges Dokumentenformat zur Kodierung und zum Austausch von Texten, das diese entwickelt hat und weiterentwickelt. (Wiki)

TEI-Grundstruktur

```
<TEI> <!-- Wurzelelement -->
    <teiHeader> ... </teiHeader> <!-- Autor, Titel, Datierung,
    ↪    Quelle(n), Editionsrichtlinien,
Versionierung, etc. -->
    <text> ... </text>
</TEI>
```

Ressourcen

Learn TEI ● Teach Yourself ● P5 = 5. Proposal ● MEI für Musik ● CEI für Charters (Urkunden) ● http://www.tei-c.org/ ● Standard zur Textkodierung ● Publikationswerkzeuge: Versioning Machine, TEI Boilerplate, TEICHI, TEI Stylesheets

TEI Header

fileDesc = Instrumentarium zur umfassenden bibliographischen Beschreibung der Inhalte eines TEI-Dokumentes

encodingDesc = beschreibt den Zusammenhang des elektronischen Textes mit dem Quelltext, z.B. diverse Regel bei der Transkription, sowie Erläuterungen zum Annotationsprozess

TEI-Header

```
<TEI> <!-- Wurzelelement -->
    <teiHeader>
        <fileDesc> ... </fileDesc> <!-- obligatorisch -->
        <encodingDesc> <!-- optional -->
        <profileDesc> <!-- optional -->
        <revisionDesc> <!-- optional -->
    </teiHeader>
    <text> ... </text>
</TEI>
```

profileDesc = Beschreibung aller nicht bibliographischen Aspekte des Textes, z.B. Informationen über die Entstehung des Textes, sowie über verwendete Sprachen u.Ä.

revisionDesc = Beschreibung aller Änderungen und Überarbeitungsschritte am Transskript des Quelltextes.

TEI Basics

TEI = **modular** → im Schema können Untermengen festgelegt werden ('Ich benutze Core und …') ● ROMA Schema ● sonst TEI P5 All

Gemeinsamkeiten von Textdokumenten:

TEI Core

- Identifikatoren (Seitenangaben, Signatur, Inventarnummer, Regalnummer, etc.)
- Abschnitte und Unterabschnitte (Gliederung)
- Abbildungen, Skizzen, grafische Elemente
- Schreibmodus (Prosa, Drama, Vers etc.)
- Strukturelle Einheiten (Absatz, Listen, Strophen, Zeilen, Reden, etc.)
- Textunterschiede oftmals durch unterschiedliche Formatierung gekennzeichnet (Titel, Überschriften, Zitate, Betonungen, etc.)
- Sachinformationen (Personen, Orte, etc.)
- Texteingriffe Korrekturen, Streichungen, Revisionen

Warum TEI verwenden oder nicht?

Nachteil: man muss Regeln befolgen – muss man aber sonst sinnvollerweise eh auch. Wollte man es wirklich gut machen, wäre es viel Arbeit! Andererseits: Flexibilität. TEI hat nicht immer etwas für Spezialzwecke.
Vorteil: die Regeln hat sich schon jemand überlegt, man muss sie nur nachschauen, verstehen und anwenden. Allgemeiner Standard in den Geisteswissenschaften, macht Daten unterschiedlicher Projekte interoperabel.

TEI verwenden

Gentle Intro to XML

TEI Core **div** (Abschnitt) ● **p** (Paragraph) ● **head** (Überschrift) ● **lb** (linebreak) ● **pb** (page break / beginning) ● **hi** (highlight) ● **l** (line) ● **lg** (line group) ● **list** ● **item** ● **listBibl** ● **bibl** (bibliograph. Angabe)

Attribute **@n** (label) ● **@type** (Typisierung) ● **xml:id** (eindeutige Kennung) ● **xml:lang** (Sprache) ● **@rend** (Darstellung) ● @ana (Interpretation)

Typische Anwendungen

```
<foreign xml:lang="en">word</foreign>
<term type="homonym"/>
<date when="2009-04-27"/>
<time when="12:00:00"/>
<name type="person"/>
<persName n="Caesar" xml:id="#44BC">Caesaris</persName> <!-- oder
↪    -->
<persName key="ID.01.208"/>
<person/>
<emph/> <hi rend="italic">kursiver Text</hi>
<seg/> <abbr type="acronym"/>
<placeName xml:id="#Whitby">Abbey</placeName>
```

Namensräume Identifikation über URI ● **<präfix:name>** ● z.B. <tei:p> ('Ich meine das <p> nach dem TEI-Standard').

Deklaration <element xmlns="URI"> …
<präfix:element xmlns:präfix="URI"> …
Z.B. <tei:p xmlns:tei="http://www.tei-c.org/ns/1.0">…

Spezialfälle: Rede & Briefe

Kodierung von Sprechakten (TEI-Referenz), falls Speaker davor steht, sonst TEIs 'said':

```
<sp who="#person">
    <speaker>1.</speaker> <p>Bla, bla, bla.</p>
</sp>

<said who="#Adolphe">- Alors, Albert, quoi de neuf?</said>
```

Briefe in TEI (Referenz)

```
<div type="letter" n="14">
    <head>Letter XIV: Miss Clarissa Harlowe to Miss Howe</head>
        <opener>
            <dateline>Thursday evening, March 2.</dateline>
            <salute>Hallo,</salute>
        </opener>
    <p>On Hannah's depositing my long letter ...</p>
    <closer>
        <salute>Yours more than my own,</salute>
        <signed>Clarissa Harlowe</signed>
    </closer>
</div>
```

Annotation fortgeschritten

Named Entities **& Indirekte Referenz**

TEI 13: Names, Dates, People, Places ● **persName** für Namensnennung, **<rs>** für *referring string* bei indirekter Nennung ('er', 'der Herr', etc.). Hier dann mit **@key** oder **@ref** spezifizieren, wer gemeint ist. (Referenz). **forename** ● **surname roleName** (z.B. 'König') ● **genName** ('der Ältere') ● **addName** ● **nameLink** ('von').

```
<name role="writer" type="person"
ref="http://d-nb.info/gnd/118540238">
Goethe</name>
<person>
  <addName type="Former">Murray</addName>
  <forename>Wilhelmina</forename>
  <addName type="nickname">Mina</addName>
</person
```

Eine **person** (selbst) ist nicht identisch mit dem **persName** (Name)! Dafür zur Verfügung stehende Elemente: **persName** zu **person**, **orgName** für **org** (Organisation), **placeName** für **place. geogName** (= geographical) = Lanschaftsmarker, Berge, etc. Problem: **Unterscheidliche Namensformen**, daher: **Normalisierung:** Zeigen, dass *dieselbe* Person gemeint ist. Wir können also eine sog. 'normalisierte' Form angeben, z.B. in Form einer Referenz auf die Personenliste im TEI-Header, auf deren **@xml:id** wir per **@ref** verweisen. Oder Normdaten der GND oder in Attribut (z.B. **@n** (*label*) oder **@ana** (Interpretation)).

Redundanz ist eine Fehlerquelle → Referenz auf eindeutigen Ort, wo die Info genau 1x vorhanden ist → Bei Fehlern nur 1x ausgebessern, nicht 200 Okkurenzen. Zur eindeutigen Referenzierung xml:ids, z.B. `<person xml:id="Mina">Mina</person>`, darauf referenzieren dann mithilfe des `@ref`-Attributs und einem Hashtag: `<persName ref="#Mina">Mina</persName>`.

5 Annotationspraxis

Hands-On gezielte Annotation

= Annotation mit dem Ziel der Erschließung

Vorgehensweise

1. Forschungsfrage formulieren
2. Überlegen, welche Daten zur Beantwortung benötigt werden
3. Begründen, warum und wie diese Daten zur Beantwortung beitragen
4. gezielt zu erschließende Quelle / Objekt (kann auch ein Gegenstand sein!) annotieren

Dazu ist (wahrscheinlich) **Vorbereitung** notwendig:

Beschaffung der Daten

1. Woher kann ich die Daten bekommen?
2. Online in welcher Form (xml, html, txt, etc.) verfügbar?
3. Falls ja: darf ich die Daten (für meine Zwecke) überhaupt verwenden? (Urheberrecht, Creative Commons, etc.)
4. selbst digitalisieren / transkribieren
5. gezielt zu erschließende Quelle / Objekt (kann auch ein Gegenstand sein!) annotieren

Sog. Pre-Processing

Kann je nach Daten und Bedürfnisse ganz unterschiedlich aussehen.

1. 'Junk' / 'Noise', der entfernt werden soll? (Zeichen aus dem HTML, kaputtes Encoding, irrelevante Daten, etc.)
2. Basis-Kodierung könnte man unter Pre-Processing fassen (Überschriften, Absätze)
3. Können Tätigkeiten automatisiert werden? Gibt es die Ressource schon online im richtigen Format?
4. Aber Achtung, übermotiviertes Automatisieren kann in Mehrarbeit enden!
5. Überprüfen, ob die 'Vereinfachung' tatsächlich das gewünschte Ergebnis erzielt!
6. Falls Online-Markup nicht passt: Evtl. ist es mit einem plain text Dokument im Endeffekt schneller.
7. Advanced: Automatisierung / Pre-Processing mit Python, XSLT, etc.

Annotation leichter machen

1. beim Annotieren nicht 'lesen', für das Grobe nur überfliegen
2. erst das Grobe (ohne zu lesen), dann eine 2. Runde für Details (*Named Entities*, etc.)
3. Keyboard-Shortcuts nutzen: per Tastatur navigieren lernen statt 'herumklicken'
4. Suchen & Ersetzen für häufige Orte / Namen
5. Patterns automatisch finden (RegEx, Suchen/Ersetzen)
6. vielleicht gibt es die Daten schon online??

Wir könnten alles in Word mit Formatvorlagen annotieren und per OxGarage umwandeln. Allerdings muss dort (bis auf Absätze) alles händisch markiert werden. Suchen & Ersetzen wäre da wohl effizienter.

1. **Basiskodierung I:** Daten aus Wikisource rauskopieren in Word. Alles markieren (Strg+A) und auf Formatvorlage 'Standard' klicken. Testen, ob es wirklich übernommen wurde. Im Word dann mit "Überschrift"-Formatvorlage die Kapitelüberschriften markieren. Würde auch mit Markdown schnell gehen.
2. **Transformation zu XML** / TEI mithilfe von OxGarage.
3. **Tipp:** Wenn ein Schema (z.B. TEI) mit dem Dokument verbunden ist, einfach öffnende Spitzklammer an eine Stelle schreiben und Oxygen bietet an, was dort stehen könnte (z.B. beim Header nützlich). Geht auch für Attribute, aber geht natürlich nur, wenn man mit dem Internet verbunden ist. Wenn man z.B. <p schreibt, dann zeigt es alle hier möglichen Elementnamen beginnend mit p, falls man schon ungefähr weiß, dass man z.B. eine Person sucht, die so benannt sein könnte.
4. **Suchen & Ersetzen nutzen,** um häufige Orte und Personennamen 'mit einem Schlag' zu annotieren. Sicherstellen, dass man nicht Dinge markiert, die keine eigentlichen Treffer waren (*false positives*): Zunächst ein paar Ergebnisse durchklicken, bevor man 'Alles ersetzen' macht. Vor Namen sicherheitshalber ein Leerzeichen setzen, damit nicht gleiche Wortteile mitgenommen werden (z.B. bei Mina durchaus möglich). Wenn Leute mit Vor- und Nachnamen vorkommen, immer zuerst die längstmögliche Version suchen, dann z.B. auch den Vornamen ohne Nachnamen. Allgemein, auch wenn man Suchen & Ersetzen zum Bereinigen unerwünschter Attribute verwendet, immer zuerst das längstmögliche, dann die kürzeren Varianten. Beim 'Bereinigen' schauen, ob man die unerwünschten Attribute nicht eh vielleicht brauchen kann (*<hi style="bold»* könnte z.B. *<hi rend="Name»* werden). (Für RegEx-Experten: XSLT / oXygen unterstützt nicht alle RegEx, die ihr evtl. aus anderen Programmiersprachen kennt – hier sorgfältig googeln: Falls es den Eindruck macht, dass der gewünschte Ausdruck nicht explizit irgendwo als XSLT-tauglich ausgewiesen wird, davon ausgehen, dass es ihn nicht gibt. Mangelnde Bereitschaft zur Akzeptanz dieses Umstands kann in stundenlanger erfolgloser Fehlersuche enden ☺)
5. Text durchgehen auf *Named Entities* (Namen, Orte), die nicht automatisch gefunden wurden. Per Hand nachziehen.

Vorgehensweise zur Basiskodierung

1. Annotation mit Word-Formatvorlagen und OxGarage-Konversion
2. XML selbst schreiben mit Tastatur-Shortcuts
3. im Internet vor-kodierten Text suchen / aus XHTML transformieren, etc. (funktionierte hier nicht so gut)

Dateien sinnvolle, aussagekräftige Dateinamen. Keine Leerzeichen in Dateinamen, keine Umlaute. Möglichst plattformunabhängige Dateiformate wählen.

Basis-Kodierung

1. **Überschriften** (*head*) und **Paragraphen** (*p*)
2. händisch erstellter, **minimaler TEI-Header**
3. **Personennamen, Orte** (mindestens direkte Verwendung, Verweise wie 'er' optional)
4. **direkte Reden mit eindeutigem Attribut** (*listPerson* im *teiHeader*), das angibt, wer hier spricht
5. **Briefe** (ganzer Brief, als Unter-Verschachtelung: Absender, Empfänger, Ort, Datum nach TEI)

TEI-Header

1. **fileDesc** ist verpflichtend, hier einfach die Autorschaft (ihr selbst und z.B. Bram Stoker) klarstellen und ggf. erklären
2. dann in der **profileDesc**: **listPerson** → **person** (mit Attribut xml:id="Minaöder so) → persName, evtl mit Untergliederung darunter nochmals in Vor-, Nach- und Spitzname. Für alle Hauptfiguren. TEI-Konventionen: addName type="nick"für Spitznamen, **roleName** für Graf, Doktor etc.
3. im Text Personen dann mit `<persName ref="Mina»` referenzieren (selbes entsprechend bei Orten)
4. Liste der wichtigsten Orte: **profileDesc** → **settingDesc** → **listPlace**: `<place xml:id="Londonäna="http://www.geonames.org/2643741/»` `<placeName> London` etc.
5. im Text: `<placeName ref="London»London</placeName>`

Tastatur-Shortcuts

`STRG+C` ..... copy
`STRG+Z` ..... undo
`STRG+Y` ..... redo
`STRG+A` ..... alles
`STRG+S` ..... save

`Alt+Tab` .. zw. Anwendungen springen
`Strg+Tab` .. zw. Tabs springen
`F5` ..... Browser-Refresh

Tastatur-Shortcuts

`STRG+Pfeiltasten` ..... in Sprüngen navigieren
`STRG+SHIFT+Pfeiltasten` .. in Sprüngen markieren)
`STRG+SHIFT+e` ..... Element (<oXygen/>)

RegEx

RegExr.com: vorher testen (potentielle Probleme z.B. = entweder alles oder nichts finden). Auch ein RegEx ist ein Modell → Abstraktion. z.B. **Jahreszahl finden** `\d{4}`

Ein Wort finden, das auch als Wortteil woanders vorkommen kann, z.B. 'bat': Suche von `bat(\W)`. D.h. die Zeichenkette beginnt mit 'bat' und es folgt ein Nicht-Wortzeiten (also Space, Punkt, Komma, Hyphen, etc.). `\w` gibt ein 'Wortzeichen', also Buchstaben, der Ausdruck in Groß jeweils das Gegenteil – also alles, was durch `\w` *nicht* erfasst wird. Das ganze kann man natürlich auch an den Wortanfang noch setzen, um auf Nummer sicher zu gehen: `(\W)bat(\W)`

Reguläre Ausdrücke (Regular Expressions, Regex) = Muster, die Zeichenketten beschreiben. → Finden und Bearbeiten von Zeichenfolgen, die auf sie zutreffen.

nachgestellt

`nichts` ..... genau 1x
`?` ..... 1x oder kein Mal
`*` ..... beliebig oft
`+` ..... mind. 1x
`{n}` ..... n-mal

Auswahl

`.` . beliebiges Zeichen

`[abc]` Zeichenauswahl
`[a-z]` `[0-9]` .. Auswahl
`\n` ... Zeilenumbruch

Gruppieren

`()` ... zur Gruppierung und Anwahl ($1)
`(.*?)` ..... beliebig (non-greedy)

`^` .. negiert Folgendes

`^` .. Start eines Strings

`\$` ..... String-Ende

`\\` .. Escape-Sequenz

`|` ..... Oder-Pipe

RegEx

`\s` ..... *space*: Leerzeichen, Tab, Zeilenumbruch
`\d` ..... Zahl (*digit*
`\w` ..... Wort (=Buchst.,Zahl, Unterstrich
`\D` `\W` `\S` jeweils das Gegenteil wie in klein

Reguläre Ausdrücke (RegEX)

Vorbereiten

Als erstes: Ein Leerzeichen markieren und durch ein Leerzeichen ersetzen. Dies verhindert Encoding-Probleme (z.B. Windows hat 'geschützte Leerzeichen', die im Hintergrund für den Computer anders aussehen. Diese können unerklärliche Fehler verursachen, daher lieber gleich am Anfang präventiv erledigen und die Fehlerquelle damit ausschließen.

Personen und Orte

Suchen und Ersetzen in Oxygen: Im OxygenXML-Suchfeld 'Reguläre Ausdrücke' und alles drunter anklicken.
Im oberen Suche-Feld:

```
( Mina|Mina Murray|Mina Harker)
```

Im Ersetzen-Feld greifen wir auf die Inhalte aus der Klammer folgendermaßen zu (mit $, das durchnummeriert wird, denn man kann mehrere Ausdrücke klammern und somit unterschiedlich weiterverarbeiten (z.B. Vor- und Nachname bereits richtig kodieren mit (Mina)(Murray|Harker), der Vorname ist dann in $1, der Nachname in $2 wiederzufinden):

```
<persName ref="#Mina">$1</persName>
```

Selbes für Ort:

```
<placeName ref="#London">$1</placeName>
```

Hier nicht vergessen, dass im Text immer vor der id der # sein muss, sonst kommt vom TEI-Schema ein Fehler gemeldet.

Kapitelstruktur

Im oberen Suche-Feld:

```
<p>(Letter|Chapter) (.*?)</p>
```

Im Ersetzen-Feld greifen wir auf die Inhalte aus der Klammer folgendermaßen zu (mit $):

```
<head>$1 $2</head>
```

Hinterher muss man um die heads (also immer beginnend vor head markieren bis kurz vor dem folgenden head, d.h. Ende Kapitel) markieren, Strg+E und ein div-Element erstellen für jedes Kapitel. Sonst nicht TEI-konform (gäbe auch komplexe Abfragen, mit denen dies automatisiert werden könnte).

Unerwünschte Zeilenumbrüche loswerden

Z.B. bei den Projekt-Gutenberg-txt-Dateien ist es uns passiert, dass unerwünschte Zeilenumbrüche im Text waren, die ohne RegEx nicht so einfach zu entfernen waren. Der RegEx, der das Problem löst, schaut folgendermaßen aus:

```
(\w+)(\n)(\w+)
```

D.h. er sucht ein oder mehrere Wörter, wo genau 1 Zeilenumbrucht (\n = newline) dazwischen steht. Die Logik dahinter: richtige Absätze werden durch mehrere Zeilenumbrüche abgebildet (bei einer Leerzeile zw. zwei Abschnitten finden sich mind. 2 newline-Zeichen, also 1x am Ende der letzten Zeile und 1x nach der Leerzeile, dort sind aber keine Wort-Zeichen dazwischen. Wir löschen also tatsächlich nur unerwünschte / überflüssige Umbrüche. Hier aber auch sichergehen, dass der Text tatsächlich so 'funktioniert' – wenn keine Leerzeilen zwischen Abschnitten sind, so funktioniert das nicht!

6 Webentwicklung

HTML

HTML = Hyper Text Markup Language

.HTML

Struktur von Webseiten. Ähnliche wie XML, aber fixiertes Set an Tags (viel weniger, die man sich merken muss und oft braucht).
Die wichtigsten HTML-Elemente: html, head, body. div, p, h1-6. span, ul, ol, li. table, tr, td.

HTML-Grundstruktur

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>

<style> /* inline CSS, lieber in extra file */
h1 {
  color: blue;
  font-family: verdana;
  font-size: 300%;
}
p  {
  color: red;
  font-family: courier;
  font-size: 160%;
}
p.important {
    color: green;
}
</style>
<script>
alert("Hello! I am an alert box!!");
</script>
</head>
<body>

<!
<h1>This is a Heading</h1>
<p>This is a paragraph. <br />
<a href="https://www.w3schools.com">This is a link</a>
</p>

<img src="img.jpg" width="500" height="600" />
<p style="color:red">I am a paragraph</p>

<p title="I'm a tooltip">
This is a paragraph.
</p>
<p>My mother has <span style="color:blue">blue</span> eyes.</p>
<p class="important">Note that this is an important paragraph.
 ↪     :)</p>


</body>
</html>
```

CSS

CSS = Cascading Style Sheets

.CSS

wird per Link in HTML eingebunden ● beschreibt die Darstellung von Webseiten ● Trennung von Inhalt und Darstellung: HTML hat den Inhalt in strukturierter Form, CSS macht das Layout.

Das **Bootstrap-Framework** bietet schon viele fertig anzuwendende Elemente. → Man muss nicht alles von Hand machen, daher sehr empfehlenswert. ● Box- und Grid-Modell
15min to Bootstrap
CSS 3 Cheat Sheet CSS3-Cheatsheet • W3 School W3schools CSS ● Tabelle der Eigenschaften
CSS ZenGarden

CSS Syntax

Selektor

```
h1 {
font-family: Arial
}
```

Deklarationsblock

```
p {
font-family: Arial ;
color: red
}
```

Schrift für `class`

```
.person {
    font-weight:bold;
    font-size:smaller;
    font-style:italic;
}
```

Bootstrap-Framework

Von diesem Link zu Bootstrap müssen ganz oben im **<head>** des HTML einerseits das Bootstrap-Stylesheet, andererseits JQuery-Plugins hineingeladen werden. (Achtung, **head** ist hier das Äquivalent zum **teiHeader**, Überschriften heißen in HTML **h1-h6**. Bei Examples gibt es Beispielseiten, die man als Basis verwenden und anpassen kann.

html head (verkürzt)

```
<html>
    <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="https://...bootstrap.min.css">
    <script
      ↪     src="https://code.jquery.com/jquery-3.2.1.slim.min.js"></script>
    </head>
    <body> [...] </body>
</html>
```

Source-Code des Starter Template (bei normalem Link: Rechtsklick 'Element untersuchen').

JavaScript

Dynamische Veränderung von Webseiten

.JS

JS kann Darstellung in Webseiten verändern, ohne dass man neu laden müsste (daher 'dynamisch').
Wenn wir z.B. Dinge per Checkbox wegklicken oder zuschalten wollen, verwenden wir JavaScript.

Hello World

```
document.getElementById("demo").innerHTML = "Hello JavaScript";
```

Web-Entwicklung-Intro

Ressourcen Lektionen 1 und Beginn von 2 auf Dash (Anmeldung nötig). ● Codecademy HTML (gut, aber eher ausführlich) Interneting is hard: Intro ● Kapitel 2: HTML Basics ● Mozilla Learn HTML & auf Deutsch ● W3Schools interaktives HTML oder wahlweise textlastiges HTML-Tutorial. ● Learn HTML.org ● Web Design in 4 minutes (CSS)

Responsive Web Design W3Schools-Tutorials

7 Datenmodellierung

Datenmodellierung

Model = Abbildung = Repräsenation ≠ Original, sondern nur ausgewählte Teile des Originals werden durch es wiedergegeben bzw. dargestellt. Die Wahl dieser abzubildenden Aspekte der Realität macht die Nützlichkeit oder Repräsentativität für einen bestimmten Anwendungsfall aus.

Subjektiv & abstrahiert → Muss auf eine spezielle Frage hin erstellt werden und nützt auch nur für diese. Wahrnehmung und Darstellung sind immer selektiv, so auch die Modelle. Ich kann nie alle Aspekte einer Vase gleichzeitig aufnehmen. Ein Foto zeigt sie nur von einer Seite.

Metadaten können zur Anreicherung von Datenmodellen gesammelt werden (Zusatz-Informationen über etwas, administrative & technische).

Disambiguierung notwendig, z.B. Normalisierung, Geonames, GND

Modell = Interpreatation Alles ab Basiskodierung, bereits teils bei *Named Enitites*. **Vorschlag zur Kenntlichmachung dieses Umstands:** `@ana` verwenden.
Wo endet die 'Abbilung von Realität', ab wo ist es nur noch eine Perspektive darauf?

Was ist für uns überhaupt das Original? Das Buch, dessen Materialität im Digitalen abstrahiert wird? Die Idee?

Modelle = vereinfachte Repräsentationen von Teilen der realen Welt.
Bsp. Photogrammetrie / Structure from Motion (= aus Fotos 3D-Modelle erstellen): Realität → mehr Info als in den Bildern. 3D-Modell: es braucht ca. 8 Bildpixel für 1 Punkt im Modell → weniger Information als die Summe der (mind. 50) Bilder.

Wechseln zw. Modellen

Nochmals zu Trennung von Inhalt und Darstellung Warum nicht gleich in HTML? → XML zu HTML ist Reduktion, da HTML ein ein viel beschränkteres Modell ist! Daher in den 'abstrakten' Rohdaten den Inhalt genau darstellen, dann Repärsentationen / Ausgabeformaten / Präsentationsformen in HTML, als PDF etc. automatisch generieren (sog. *single source* Prinzip).
Vom Buch zu unseren digitalen Daten, von den digitalen Daten wieder zum Buch. Bei der Konvertierung geht Information eventuell verloren. Z.B. in Anführungsstrichen", was bedeutet das jetzt? Der Werktitel? Was, wenn er dann kursiv genmacht werden soll? Wenn kein Markup mehr dabei ist, wird es schwerer, das automatisiert zu ändern. Daher auch möglichst keine Darstellunganweisenen ('in quotes') hardcoden (= direkt reinschreiben), sondern lieber in den Basisdaten, z.B. als `<q>` (*quote*).

Datenmodellierung

Modellbegriff „Alle Erkenntnis ist Erkenntnis in Modellen oder durch Modelle und jede menschliche Weltbegegnung überhaupt bedarf des Mediums Modell" (Herbert Stachowiak 1973)

Abbildungsmerkmal Ein Modell ist immer ein Abbild von etwas, eine Repräsentation natürlicher oder künstlicher Originale.

Verkürzungsmerkmal Ein Modell erfasst nicht alle Attribute des Originals, sondern nur diejenigen, die dem Modellschaffenden bzw. der Nutzerin/dem Nutzer relevant erscheinen. Ein Modell abstrahiert. Z.B. kann man eine Statue immer nur von einer Seite fotographieren (bzw. sogar sehen! D.h. ich sehe nie 'das Ding an sich', sondern einen Ausschnitt aus der Summe aller Eigenschaften, die es ausmachen).

Pragmatisches Merkmal Orientierung am Nützlichen. Ein Modell wird vom Modellschaffenden bzw. der Nutzerin/dem Nutzer innerhalb einer bestimmten Zeitspanne und zu einem bestimmten Verwendungszweck an Stelle eines Originals eingesetzt.

Datenmodellierung Modell = Ausschnitt aus der realen Welt, allerdings werden im Modell nur jene Attribute berücksichtigt, die für meine Fragestellung relevant sind. Modell und Realwelt weichen somit voneinander ab.

Modell = Abstraktion (=Klasse) Konkretisierungen (Kochrezepte, Briefe, Gedichte usw.) eines Modells werden als Instanzen bezeichnet.

Standardisierte Modelle ermöglichen die gemeinsame Auswertung, datenübergreifende Suche, Datenaustausch…

Modelle = vereinfachte Repräsentationen von Teilen der realen Welt.

McCarty

Willard McCarty, „Modeling: A Study in words and Meanings" (2004)

1. Ein Modell ist eine Repräsentation von etwas, z.B. für Studienzwecke.
2. Modellierung = heuristischer Prozess in dem Modelle kreiert und verwendet werden, um Probleme zu lösen.
3. *model of* (beschreibend) vs. *model for* (z.B. Plan zum Hausbau).
4. Modellierung als experiementeller Vorgang mit dem Ziel der Welterfassung.

Modellierung als iterativer Prozess

- Bestehendes Wissen einbringen
- Modell bilden
- Das Modell verwenden um das Thema zu untersuchen
- an den Diskrepanzen zw. Modell und Realität zeigt sich der Verfeinerungsbedarf, da das Modell ja immer nur eine Annährung ist. Diese zeigt aber wieder, wo noch nachgebessert werden muss.
- mit neuem Wissen aus Modell neu modellieren

8 Abfragen mit XPath

Intro

Mit XPath kann in XML-Dokumenten navigiert werden.
`//persName` = 'Gib mir alle Personennamen / alle Personen.'
→ alle Funde werden visuell markiert. Ich habe sie damit 'gefunden' und 'angesprochen'. Mit dieser Abfrage kann ich diese Elemente 'anwählen', z.B. zur Weiterverarbeitung in XSLT.

Um mit XPath etwas zu finden:

1. Auf Deutsch in einem vollständigen Satz formulieren, was man erreichen will. 'Ich möchte alle Vorkomnisse der Person Mina.'
2. Auf technische Struktur des Dokuments umlegen: Diese Vorkomnisse sind alle als `persName ref="Mina"` kodiert, womit ich das Mittel habe, sie per XPath zu addressieren.
3. Möchte ich Elemente ausgegeben haben, ihre Inhalte? Attribute, ihre Inhalte? Oder gar das Ergebnis einer Funktion ('Zähle alle Vorkomnisse und gib mir die Anzahl.')?
4. Gibt es Bedingungen, also will ich nur bestimmte Ergebnisse oder alles, was gefunden wird?

Jedes Vorkommen von Mina wäre 'übersetzt' also: Alle `persName`, die `@ref='Mina'` haben, *im ganzen Dokument*.

Pfade & Scope

Scope: relative & absolute Pfade
Ich kann, je nach Abfrage, **im ganzen Dokument** oder **in einem bestimmten Gültigkeitsbereich** suchen. Dazu unterscheide ich

relative Pfade nicht alles wird explizit aufgeschlüsselt, Lokalisierung abhängig davon, wo ich gerade im Dokument 'stehe'. → Das ist z.B. nützlich, weil ich nicht wissen muss, wo genau ich bin. Ist aber auch schlecht, weil ich evtl. nicht genau weiß, wo ich gerade bin und andere Ergebnisse bekomme, als ich erwarte. Falls man sich nicht sicher ist, lieber volle Pfade. In gewissen Funktionen von XSLT später unumgänglich und keine große Sache.

Exkurs: Scope Es kann auch sein, dass ich z.B. innerhalb einer `<xsl:for-each>`-Abfrage mit einer Sub-Menge des Dokuments weiterarbeite. Darin navigiere ich dann relativ, weil ich ja womöglich gar nicht wissen *kann*, wo die Elemente, auf die etwas zutrifft, eigentlich im Dokument stehen.

Ansonsten kann ein relativer Pfad aber auch dazu führen, dass man 'unerwünschte' Dinge findet: evtl. hat man sich verkalkuliert, man findet mehr oder weniger als gewollt. Vielleicht steht man an der falschen Stelle und die gewünschte Abfrage ergibt im aktuell zur Verfügung stehenden Umfang (*scope*) keine Ergebnisse.

absolute Pfade Jede Angabe ist explizit aufgeschlüsselt. Ich bekomme wirklich nur den erwünschten Pfad. Wird komplizierter, wenn die Ergebnismenge aus mehreren unterschiedlichen Bereich kommt (hier dann Oder-Verknüpfungen verwenden ||).

XPath Funktionsweise

Abfrageresultat = Knotenmenge Eine Pfad-Abfrage gibt eine Menge zurück. Mit einem Punkt kommt man an den Inhalt. Z.B. `matches(.,'([0-9]4)')`: Enthält 'alles' = aktueller Knoteninhalt eine vierstellige (Jahres-)Zahl? Antwort, ja/nein. Kein Zugriff auf die gefundenen Inhalte selbst, nur Ja/Nein-Antwort (!). `//*[matches(.,'([0-9]{4})')]` = 'beliebige Knoten im ganzen Dokument, wo eine vierstellige Zahl vorkommt' = gibt eine Knotenmenge zurück, der Elemente auf die die Antwort "ja"= "true"lautet.

Prädikate = Bedingungen Prädikate, also Bedingungen, sind in eckigen Klammern. Dort drin kann ich eine Funktion abfragen, die ein Ergebnis liefert (z.B.) ja/nein.

Funktionen Funktionen können auch um den ganzen Ausdruck herum, doch dann wird im Normalfall auch keine Knotenmenge ausgegeben, sondern eine Antwort (abhängig von der speziellen Funktion). Wie eine Funktion in Mathe hat sie ein Ergebnis. Funktion ungleich Ergebnismenge, sondern Lösung.

```
function(//lokalisierung, parameter)
```

Unterschied zwischen String und Knoten Wenn er mal im substring ist, hat er nur noch Zeichen, keine Elemente mehr. Auch bei Punkt(.).
Manche Funktionen geben eine Antwort, manche würden etwas verändern: die gehen nur in XSLT. Im XPath-Viewer geht nur das, was ich anzeigen kann. `XPath failed due to: A sequence of more than one item is not allowed as the first argument of`. Reinklicken kann man außerdem nur, wenn man eine Knotenmenge gewählt hat. Manche Funktionen nehmen nicht mehrere Knoten an, weil sie genau 1 Antwort geben sollen.

XPath

```
//person//surname
//persName[@ref]
//persName/@ref
//persName[@ref="bla"]
//persName[contains(.,'bla')]
contains(.,'bla') = ja/nein, keine Knotenausgabe
```

Vorteil gegenüber normalen Suchen und Ersetzen: man kann Bedingungen abfragen ("gibt mir alle Vorkommnisse von Mina, aber nur in Kapiteln, wo Dracula nicht vorkommt").
absoluter und relativer Pfad: in ein Div reinklicken und nur 'p' suchen.
Absoluter expliziter Pfad: z.B. title, wenn im Header auch bibliograph. Angaben sind: //title gibt alle Resultate, aber man will ja nur den Dokumententitel, daher
/TEI/teiHeader/ [hier entweder jeden Schritt eingeben oder z.B:]//titleStmt/title
Negation mit != oder not //persName[@ref] //persName[not(@ref)] – Personenname ohne ref-Attribut
//persname[@ref !='Mina'] – alle außer Mina

Einstieg & Tipps

XPath-Vorschau es ist möglich, dass nicht alle korrekten XPath-Abfragen auch in der Vorschau angezeigt werden können. Hier kommt dann keine Fehlermeldung und in XSLT sollte es gehen.

`substring()` Die String-Operationen (siehe 'Zeichenketten-Funktionen') gehen immer nur auf genau eine Zeichenkette und nicht etwa eine Menge.

Vorgehen Daher immer erst in einem normalen Satz formulieren, was mann will. Das dann zerlegen, was davon Pfad, was Bedingung und was Funktion sein könnte. Dann erst den Ausdruck schreiben.

Lokalen / globalen Scope testen Ausprobieren den Mauszeiger in ein `<div>` zu stellen und dann nur `p` in die XPath-Abfrage einzugeben: Man findet Knoten im lokalen Kontext. // gibt immer vom globalen Kontext aus an.

Antwort(menge) verstehen `not(//persname[@ref="Mina"])` gibt 'false' zurück, weil er fragt: Gibt es im ganzen Dokument einen `persname[@ref="Mina"]`, die Antwort ist ja. Daraufhin verneint `not()` alles, d.h. es stimmt nicht, dass es keinen `persname[@ref="Mina"]` im Dokument gibt.

Beispiel zum Einstieg

```
1. //div[@type="chapter"]   -- count(//head)
2. //head -- //div[@type='chapter']/head
3. substring-after(//div[type=....],'Chapter')
4. distinct-values(//placeName/@xml:id)
  ODER distinct-values(//placeName)
       --> was ist der Unterschied?
5. distinct-values(//persName[@ref="#Mina"]
7. //p[contains(.,'Christmas')]
  ODER //p[matches(.,[A-Z][a-z])]
8. //div[@type="chapter"][@n="4"]/head
  ODER  //div[@type="chapter"][4]/head
9. //persName[@ref != "#Mina"]
  ODER \\persname[not(@ref='#Mina')]
10. //div[@type='chapter'][1]/persName | //div[@type='chapter'][1]/placeName
ODER //person | //place
```

XPath

XML-Dokument = **Baum** → Zugriff / Navigation über Pfade ● Wurzelknoten (/) ≠ Wurzelelement, dieses ist Kind des Wurzelknotens. ● Weiters: Attributknoten, Textknoten, Elementknoten, (+ Namensraum + Kommentar + Verarbeitungsanweisung) ● Abfrage = Pfadstruktur, Ergebnis = Knotenmenge / Wert. ● Teil der XSL-Sprachfamilie (XPath, XSLT, XSL-FO, ...XQuery).
absolut (`/person/name`) oder **relativ** (`../../title`)

Syntax `Achsename::Knotentest[Prädikat]` ● Achse = Bewegungsrichtung ● Knotentest: Welcher Knoten? ● Prädikate: Bedingungen.
`/child::person[attribute::gender]/child::name` ... Langform
`/person[@gender]/name` ... Kurzform

Achsen
`::self` ... der aktuelle Knoten selbst (Kurzform: .)
`::ancestor` ... alle Vorfahren des aktuellen Knotens
`attribute::` ... die Attribute des aktuellen Knotens (**Kurzform: @**)
`child::` ... direkte Kindelemente des aktuellen Knotens
`descendant::` Nachkommen des aktuellen Knotens (**Kurzform: //**)
`parent::` ... Elternknoten des aktuellen Knotens (**Kurzform: ../**)
`preceding-sibling::` . vorhergehende Geschwister des aktuellen Knotens
`following-sibling::` .. nachfolgende Geschwister des aktuellen Knotens

Navigieren in XML

Zusätzliche Infos: IDE Kurzreferenzen

im Kontext ausgewertet ● absolute und relative Positionsangabe möglich (*dynamic & static context* ● Namespaces ● Funktionen, Variablen

XPATH

XPath-Basics
`/` ... Wurzelknoten / Schritt weiter im Baum (nächster Lokalisierungsschritt)
`::*` ... "*" = alle Elementtypen (kein bestimmter Elementname)
`@` ... Attribut
`[Zahl]` ... das wievielte Element (aus der Ergebnismenge)
`text()` ... Textinhalt eines Elementes
`::Elementtyp` ... Bedingung für den Elementtypen (Knotentest)
`//` ... beliebig tief im Baum
`[text() = 'Textinhalt']` ... Bedingung (Prädikat)
`.` ... self
`(/../` Eine Hierarchiestufe (mit beliebigem Namen) überspringen
`pfad//unterpfad` ... beliebige Tiefe dazwischen

XPath Prädikate

Bedingungen, die Submengen definieren = Prädikate

```
//person[@gender="female"]
//person[firstname = "Stefan"]
//person[firstname != "Tanja"]
//person[1]
//person[last()]/firstname
//person[position() = 2]
/participants/person[position()=5]/firstname
/participants/person[last()]/firstname
count(child::*)
```

XPath-Funktionen

Funktionen – Knotenmenge
`position()` ... Position des aktuellen Knotens, return: Zahl
`count()` ... Anzahl der Knoten der übergebenen Knotenmenge
`last()` ... letzter Knoten der gewählten Knotenmenge

Zeichenkettenfunktionen
`substring-before(value, substring)` .... Teilzeichenkette eines Textknotens (*value*), die vor dem angegebenen Zeichen (*substring*)
`substring-before(., 'mina')` ... Bsp.
`substring-after(value, substring)` ... wie before
`substring(value, start, length)` ... mit Start- und Länge
`string-length(.)` ... Länge des aktuellen Textknoten
`concat(value1, value2, ...)` ... zusammenfügen
`concat(nachname, ', ', vorname)` ... Bsp.
`translate(Ziel, string, Ersetzung)` ... Ersetzen
`noramlize-space(Ziel)` ... Entfernt trailing whitespace

XPath-Funktionen

Boolsche Funktionen
`starts-with(value, substring)` ... return: wahr/falsch
`starts-with(., 'D')` ... wahr/falsch
`contains(value, substring)` ... ist die Zeichenkette im aktuellen Textknoten enthalten, wird der Wert wahr zurückgeliefert
`not(Vergleich)` ... gibt ja, falls nicht XY

RegEx-fähige Funktionen
`matches(Ziel, 'RegEx')` wie `contains()`, aber nur exakte Treffer
`replace()` ... wie `translate()` nur RegEX-fähig
`tokenize()` ... Text in Einzelwörter auftrennen

Fortgeschrittene Anwendungen

Problembehebung

1. **[Nichts geht]** Im Zweifelsfall ist das Problem immer der **Namespace**. Oder man steht an einer anderen Stelle, als man dachte (bei der Verwendung relativer Pfade) → sicherheitshalber **absolute Pfade** wählen.
2. **[Nichts gefunden]** XML ist **hierarchisch**! Stehe ich an der **richtigen Stelle**? Erreiche ich den grad an **Tiefe**, den ich will (im Zweifelsfall `../` einbauen, um etwas unabhängiger von der Hierarchie zu sein.
3. **[Zu wenig gefunden?] Sprünge in der Pfad-Tiefe:** Ein Pfadausdruck (ohne zwischengeschaltetes `pfad//unterpfad`) überspringt keine Ebenen bzw. geht genau 1 Pfad weiter: Sind Zwischenebenen drin, die ich übersehen habe (z.B. eine weitere `div`-Verschachtelung)? Mit `//` dazwischen ist die Tiefe dann aber beliebig – falls eine spezielle gewünscht ist (z.B. 3 Stufen), dann lieber einen absoluten Pfad mit `../../../` bauen.
4. **[Ausdruck geht, passt das Ergebnis zur Anforderung?]** Möglichst spezifische Abfragen formulieren: Finde ich alles, was ich suche? Matcht die Abfrage womöglich auch noch andere, unerwünschte Elemente?
5. Es gibt vordefinierte Verarbeitungsregeln, d.h. alle Textinhalte werden ausgegeben, außer man unterdrückt es explizit.

Vorbereitung von Transformationen
Mithilfe von XPath zuerst alle im Dokument vorkommenden möglicherweise zu verarbeitenden Dinge ausgeben lassen (je 1x, (`distinct-values()`-Funktion) :
`distinct-values(//@*/name())` ... jede Art von Attribut
`distinct-values(//@*)` ... jeder Attributwert
`distinct-values(//@rend)` ... jeder `@rend`-Wert
`distinct-values(//name())` ... Elementtypen

9 Transformationen mit XSLT

Was ist XSLT?

XSL (*eXtensible Stylesheet Language*) ist eine Programmiersprache zur Transformation von XML-Daten. Diese erlaubt die Speicherung von abstrakten Basisdaten, aus denen dann unterschiedlichste Repräsentationen, z.B. die Strukturinformationen für HTML, automatisch generiert werden können (*single source*-Prinzip). **XSLT** (=XSL-Transformations) wird häufig synonym dazu verwendet, ist aber eigentlich nur ein Teil neben **XSL-FO** (*Formatting Objects*, zur Beschreibung von Druckdokumenten als PDF), das seit 2012 nicht mehr weiterentwickelt wird.

XSLT besteht aus Verarbeitungsmustern (sog. 'Schablonen' oder *templates*), wobei das XML-Dokument durchgegangen wird und Schablonen angewendet werden, sobald eine auf den aktuell angefundenen Inhalt passt. Die Verarbeitung durch den Parser beginnt beim Wurzelknoten, weswegen es immer eine erste Schablone gibt, die auf alles passt:

```
<xsl:template match="/">...</xsl:template>
```

Wird für angetroffene Inhalte kein Template gefunden, so werden die *Built-in*-Regeln angewendet, die die Inhalte der Elemente (inkl. Unterelemente!) ausgeben. Dies begegnet einem oft in dem Umstand, dass einfach der ganze Dokumenteninhalt drauflosgeprinted wird, was oft nicht eigentlich erwünscht ist. Z.B. möchte man oft ja nur den `body`-Inhalt eines Dokuments und nicht alle Metainformationen des `teiHeader`s unsortiert ausgedruckt haben. Dieses grundlegende Verhalten muss man daher bewusst unterbinden.

XSLT-Intro

XSL(T): eXtensible Stylesheet Language (Transformations)

.XSL

XSLT-Stylesheet (selbst XML-Dokument) beschreibt Regeln für den Transformationsprozess einer Eingabedatei (XML-Dokument) in eine oder mehrere Ausgabedateien (XML, XHTML, Text).

Ausgabeformate (x)HTML XML → z.B. Word, TEI und andere XML-Standards, RDF, SVG Text → z.B. LaTeX (→ PDF), RTF, MARC.

Warnung am Rande: LaTeX z.B. ist auch Markup, hat aber andere geschützte Entitäten, bzw. andere Konventionen, wie man Dinge, wie etwa den Ampersand escaped. `xsl:output` und nachkontrollieren!

XSLT Ressourcen

IDE Kurzreferenzen ● W3Schools-Tutorial

Einstieg

In Oxygen sowohl ein zu transformierendes XML als auch ein neues XSL-Stylesheet eröffnen.

Auto-generiertes Start-Dokument

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="2.0">

</xsl:stylesheet>
```

Dann in die XML-Datei reinklicken und ein Transformationsszenario konfigurieren (Schraubenzieher-Symbol):

1. 'Neu' auswählen (XSLT-Transformation).
2. Reiter XSLT: XML-Datei aussuchen, XSL-Datei aussuchen (kann man sonst auch auf massenweise Dokumente anwenden (Projekt-Optionen).
3. Transformator wählen: Saxon (9er Version, sonst egal welcher).
4. Reiter XSL-FO: Hier egal, weil das eine andere Transformation als XSLT ist (zum PDFs machen, was alternativ zu LaTeX dort gehen würde).
5. Reiter Ausgabedatei: 'Datei speichern unter' → Grünen Pfeil anklicken (`${cfn}` auswählen. Dann `-transform.xml` hinzufügen, damit die Orignaldatei nicht überschrieben wird. (Bei erneuter Transformation wird diese Datei immer wieder überschrieben, außer man benennt sie um).
6. Wahlweise 'Im Editor öffnen' und anzeigen als XML, falls XML oder XHTML, falls HTML.
7. Ok, dann anwenden.

Damit wird nun unser, bishe noch sehr leeres, Stylesheet auf die Datei angewendet. Was fällt auf? Das sich zeigende Verhalten ist das Standard-Verhalten von XSLT, wenn es keine zutreffenden Verarbeitungsregeln findet.

Standard-Transformationsablauf

Leeres Match-Root-Template

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="2.0">

    <xsl:template match="/">
        <!-- und jetzt? -->
    </xsl:template>

</xsl:stylesheet>
```

Mit dem folgenden äußern wir, dass wir möchten, dass die Standard-Verarbeitungsweise wieder verwendet wird. Der angefundene Text wird ohne Elemente hineingeschrieben. Achtung, das ist jetzt nicht eigentlich ein XML-Dokument, weil es keine Elemente oder hierarchische Struktur hat!

Standard-Regeln mit `apply-templates`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="2.0">

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

</xsl:stylesheet>
```

Struktur 'hardcoden'

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="2.0">

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

</xsl:stylesheet>
```

XSLT-Standardablauf II

Statt die Elementenhierarchie mühsam händisch wieder reinschreiben zu müssen, können wir auch Templates ('Schablonen') definieren, die immer aktiv werden, wenn ein bestimmtes Element gefunden wird. Lies: Wann immer Du `<p/>` findest, mach XY. Wir deklarieren vorher noch einen Namespace (`xmlns`), damit unsere TEI-Dateien erkannt werden:

```
xmlns:t="http://www.tei-c.org/ns/1.0"
```

Elemente müssen jetzt immer mit `t:` davor angesprochen. `t:p` lies: `<p>` aus dem Namespace, der mit `t:` abgekürzt ist (siehe Stylesheet-Deklaration oben).

`p`-Elemente matchen

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:t="http://www.tei-c.org/ns/1.0"
    exclude-result-prefixes="xs"
    version="2.0">

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

    <xsl:template match="t:p">
        <neu>
            <xsl:apply-templates/>
        </neu>
    </xsl:template>

</xsl:stylesheet>
```

XSLT-Basics III

Alle `p`s, inklusive ihrem Inhalt (dank `apply-templates`) werden erhalten, aber sie heißen nun statt `<p>` `<neu>` (sinnfrei & zu Demonstrationszwecken). Dieses Vorgehen jedenfalls nennt sich das sogenannte **Push-Paradigma**.

Das Gegenstück dazu, das **Pull-Paradigma** sehen wir jetzt beim hinzugefügten `<head>`. In dieses neu erzeugte Element holen wir uns bewusst den Inhalt vom `<title>`-Element rein (Achtung, hier XPath-Wissen anwenden: Wenn es mehrere `<title>`-Elemente gibt, nimmt er *alle*. Im Zweifelsfall explizit und mist absoluten Pfaden arbeiten. Außerdem ist `value-of select=` nicht dasselbe wie `apply-templates`. `apply-templates` schaut erst, ob es (weiter *unten* im Code!) noch Regeln gibt, die auf die Elemente zutreffen, die im gefundenen Ding eingeschachtelt sind und wendet ggf. diese an. `value-of select=` kopiert nur den aktuell angesprochenen Textinhalt; tiefere Verschachtelungen gehen verloren. `value-of select="."` gibt den aktuellen Knoteninhalt (Text). `attribut={@rend}` ist eine Kurzschreibweise dafür bei Attributen.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:t="http://www.tei-c.org/ns/1.0"
    exclude-result-prefixes="xs"
    version="2.0">

    <xsl:template match="/">
        <xml>
            <head><xsl:value-of select="//t:title"/></head>
            <xsl:apply-templates/>
        </xml>
    </xsl:template>

    <xsl:template match="t:p">
        <neu attribut={@rend}>
            <xsl:apply-templates/>
        </neu>
    </xsl:template>

</xsl:stylesheet>
```

Inhalte löschen

Per Automatismus werden in XSLT alle Inhalte (Werte) erhalten – aber wir haben schon gesehen, dass man mit leeren Regeln bewusst löschen kann. Wir löschen im Beispiel also probehalber alle `<hi>` (inklusive Inhalt!). Hier darauf achten, dass man nicht versehentlich auf die Quelldatei schreibt, sonst ist alles weg (siehe Transformationsszenario konfigurieren).

Bewusstes Löschen durch leere Regel

```xml
<xsl:template match="t:hi">
    <!-- loeschen -->
</xsl:template>
```

Das sind im Grunde schon alle Regeln, die man wissen muss. Daneben gibt es natürlich noch mehr Funktionen, falls man kompliziertere Sachen machen will. Diese finden sich im Folgenden erklärt, werden aber evtl. gar nicht gebraucht.

Gedicht → HTML-Webansicht

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

    <xsl:template match="/">
        <html>
            <head>
                <title><xsl:value-of select="div/head" /></title>
            </head>
            <body>
                <xsl:apply-templates select="div" />
            </body>
        </html>
    </xsl:template>

    <xsl:template match="div">
        <h1><xsl:value-of select="head" /></h1>
        <div>
            <xsl:apply-templates select="lg" />
        </div>
    </xsl:template>

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

    <xsl:template match="l">
        <xsl:value-of select="." />
        <br />
    </xsl:template>

</xsl:stylesheet>
```

Weitere Funktionen

Variablen

```xml
<xsl:variable name="platzhalter" select="inhalt">
```

XSLT führt bei Variablen einen sog. *pass by value*, nicht *by reference* aus, d.h. es wird nur der Wert übergeben, den das ausgewählte Element zu einem Zeitpunkt hat. Wenn ich die Variable zu früh setze, wähle ich dabei womöglich einen fixierten Wert. Also am besten immer so nah wie möglich am 'Ziel'.

Attribute `<xsl:attribute>` dient zum Setzen von Attributen im Resultat-Dokument. Dies ist nicht nötig, wenn man nur einen Inhalt will, dann kann auch die Kurzschreibweise verwendet werden.

plain text `<xsl:text>` setzt Text 'as is', d.h. fügt nicht unerwünschte Leerzeichen ein, was sonst passieren. Überhaupt geht XSLT mit Whitespace relativ frei um, außer man verbietet es explizt (`preserve-space`.

Sortieren Es gibt auch `<xsl:sort>` zum Sortieren.

Bedingungen `<xsl:choose>` zum Auswählen unterschiedlicher Verarbeitung je nachdem, was angefunden wird. `<xsl:if>` führt Anweisungen nur aus, wenn eine Bedingung erfüllt ist (z.B. falls es Element XY gibt, oder ein Element Inhalt XY hat).

Personenliste in HTML mit `for-each`

```xml
<ul> <!-- ungeordnete Liste in HTML -->
<xsl:for-each select="t:persName">
    <li> <!-- Listenelement -->
    <xsl:value-of select="." />
    </li>
</xsl:for-each>
</ul>
```

Attribute setzen

```xml
<xsl:for-each select="t:persName">
    <span class={@rend}><xsl:value-of select="."></span>
</xsl:for-each>
<!-- ODER -->

<span>
    <xsl:attribute name="id">
        <xsl:value-of select="@rend"><xsl:text>-person</xsl:text>
    <xsl:attribute>
    <xsl:attribute name="interpretation">
        <xsl:value-of select="concat(@rend,@ana)">
    <xsl:attribute>
    <xsl:apply-templates /> <!-- für den eig. Inhalt -->
<span>
<!-- Resultat z.B.
    <span id="monster-person" interpretation="monster-boese">
    Weird Sisters </span> -->
```

For-each-group

```xml
<!-- könnte man auch nach Alphabet sortieren oder so. Nur gezielt verwenden,
    ↪    macht in komplexen Dokumenten oft ominöse Fehler. -->

<xsl:for-each-group select"dings" group-starting-with="dings[@rend="Startdings]]"
    <xsl:apply-templates select="current-group()">
    <xsl:value-of select="current()/dingens">
</xsl:for-each-group>
```

Mehrere Quelldokumente zusammenführen (`document()`)

```xml
<xsl:apply-templates
    ↪    select="document('Letter1_TEI.xml')/tei:TEI/tei:text/tei:body/tei:div"/>
```

flow control / Bedingungen

Beispiel: Wähle jede Person (`//persName`) und generiere eine Aufzählung (`<ul>`) der Nachnamen (`lastname`):

Schleifen: for-each

```
<ul>
<xsl:for-each select="//persName">
    <xsl:sort select="lastname" order="ascending" />
    <li> <xsl:value-of select="lastname"/> </li>
</xsl:for-each>
</ul>
```

`xsl:if` führt Anweisungen nur aus, wenn die Bedingung in `@test` erfüllt ist.

Bedingungen I: Falls

```
<xsl:if test=" xpath-ausdruck "> ... </xsl:if>

<xsl:for-each select="//book">
    <xsl:if test=" author = 'Cicero' ">
        <li><xsl:value-of select="title"/></li>
    </xsl:if>
</xsl:for-each>
```

Auch die Unterscheidung mehrerer Fälle ist möglich. So kann ich z.B. für das HTML unterschiedlichen Personen unterschiedliche Farben zuweisen. Dazu frage ich, in `@test` welche ID ein angetroffener `persName` hat und verarbeite dann je nachdem unterschiedlich.

Bedingungen II: Auswählen

```
<xsl:choose>
    <xsl:when test=" xpath-ausdruck "> ... </xsl:when>
    <xsl:otherwise> ... </xsl:otherwise>
</xsl:choose>
```

Man kann zudem sortieren (`xsl:sort`), kopieren (`xsl:copy` - `xsl:copy-of`)und Variablen verwenden.

XSL-Paradigmen: Push

Push-Paradigma

```
<xsl:apply-templates select="xpath-ausdruck"/>
```

Geeignet vor allem: Für das Verarbeiten des Text-Bodys. Hier werden angetroffene Dinge einfach verarbeitet, wann immer sie auftreten. Man muss deren Reihenfolge nicht kennen. Die Dokumentenstruktur des 'Herkunftsdokuments' wird weitgehend erhalten (soweit es eben in den Template-Regeln definiert ist).

Push-Paradigma

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

Push Processing Jedes Element bekommt eine Regel oder *built-ins* werden verwendet. Nützlich falls Input- und Outputdokument dieselbe Struktur haben sollen.

Call Template / Push-Methode

```
<xsl:template match="/">
    <xsl:call-template name="irgendein-name ">
</xsl:template>
<xsl:template name="irgendein-name">
    ... macht etwas ...
</xsl:template>
```

XSL-Paradigmen: Pull

Pull-Paradigma

```
<xsl:call-templates name="name-eines-templates "/>
```

Geeignet vor allem: Herausziehen von Metadaten aus dem `teiHeader`, um sie irgendwo extra anzuzeigen. Nochmals 'über das Dokument drübergehen' und z.B. einen Personenindex machen, den man an einer bestimmten Stelle hingesetzt haben will (`<xsl:for-each>` ist z.B. ein typischer Befehl der Push-Methode.

Pull Processing Gewisse Knoten explizit auswählen. Kontrolle darüber, nur bestimmte Knoten im Output-Dokument weiterzuverwenden. Gut geeignet, falls das Output-Dokument eine völlig andere Struktur haben soll, als der Inhalt oder nur sehr selektiv Inhalte übernommen werden sollen (z.B. nur eine Liste aller enthaltenen Personennamen braucht nicht den ganzen restlichen Text dazu verarbeiten).

Pull-Paradigma

```
<xsl:value-of select="'pattern'"/>
<xsl:apply-templates select="'pattern'"/>
<xsl:for-each select="'pattern'"/>
```

Paradigmen in der Praxis

In der Praxis wird meist eine Kombination von beiden verwendet: Die Metadaten des `teiHeader` werden, z.B. in der Form eines Zitiervorschlags, mit dem Pull-Paradigma ausgelesen. Dann geht man an die Stelle des Output-Dokuments, wo der hauptsächliche Inhalt (*body*) hin soll und wählt aus, dass gar nicht mehr das ganze Dokument automatisch per Pull-Paradigma verarbeitet werden soll, sondern nur alles ab dem `body`.

apply-templates select

```
<xsl:apply templates select="//t:body" />
<xsl:apply-templates select="head" mode="toc"/>
```

Anderes Beispiel: Kapitelüberschriften innerhalb des Buches per Push-Paradigma, für das Inhaltsverzeichnis im Pull-Paradigma. Mithilfe von `modes` kann man zudem noch unterscheiden, dass in gewissen Situationen andere Templateregeln angewandt werden.
`<xsl:template match=>` verarbeitet ein Element, wo immer es angetroffen wird.
`<xsl:value-of select=>` gibt den Inhalt eines Elements als String aus – wenn noch Kindelemente sind, gehen die Knoten/Auszeichnungen verloren, weil nur mehr plain text kopiert wird.
`<xsl:apply-templates>` geht die Regeln durch, wo er dann alles findet, was unter `"match"` definiert ist – er sucht aber Unter-Verschachtelungen nur weiter, wenn man das explizit so angibt (mit weiterem `apply-templates`).

Facts

XSLT als Sprache ist deklarativ (wie CSS nur mächtiger) und dynamically-typed, d.h. es arbeitet eher mit Werten als mit Variablen (wie JavaScript).

Vorgehen bei Stylesheets Regelwerk nach Top-down-Prinzip, ausgehend vom Wurzelelement des Quelldokuments.

Wurzelelement `xsl:stylesheet` XSLT ist selbst XML, daher hat es auch ein alles umgebendes Wurzelelement. Hier werden einige Dinge deklariert (Namensräume, XSLT-Processor, etc.).

Exkurs: Markup & Programmierung

❝ Aber XSLT ist doch nicht Programmieren … ❞

Exkurs Turing-Vollständigkeit: Ist Markup Programmierung? Nein, aber …
Prozedurales PostScript und TEX= Turing-vollständig (d.h. man theoretisch kann alles damit programmieren).
Auch **XSLT & XQuery** = Turing-vollständige Sprache, 'Befehle' allerdings auf die Bearbeitung deskriptiven Markups in XML ausgelegt
● Sprache selbst in XML formuliert.