Semistrukturierte Daten

XSLT

Stefan Woltran
Emanuel Sallinger

Institut für Informationssysteme
Technische Universität Wien

Sommersemester 2014

Inhalt

XSLT

- Transformationssprache für XML Dokumente
  - XSLT steht für *Extensible Stylesheet Language Transformations*
- Hauptaufgaben
  - Informationsextraktion
  - Konvertieren von XML Dokumenten ...
    ... in andere XML Dokumente
    ... in HTML und andere Formate
    ... in Text
- Versionen
  - XSLT 1.0: W3C Recommendation seit 1999
  - XSLT 2.0: W3C Recommendation seit Jänner 2007 (basiert auf XPath 2.0)
  - XSLT 3.0: W3C Last Call Working Draft im Dezember 2013

Beispiel

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml">

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

  <xsl:template match="veranstaltung/titel">
    <i><xsl:value-of select="."/></i><br/>
  </xsl:template>

  <xsl:template match="text()"/>
</xsl:stylesheet>
```

Struktur von Stylesheets

- Stylesheets
- Deklarationen
- Instruktionen
- Vereinfachte Stylesheets

Stylesheets

- XSLT-Stylesheets sind selbst XML-Dokumente:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- Hier folgen die Deklarationen -->
</xsl:stylesheet>
```

- `<xsl:transform>` als gleichwertiger Alias zu `<xsl:stylesheet>` erlaubt.
- Verknüpfung zwischen Quelldokument und Stylesheet als Parameter beim Aufruf des XSLT-Prozessors oder durch PI im Quelldokument:

```
<?xml-stylesheet type='text/xsl' href='lva.xsl'?>
```

Deklarationen

`xsl:template`	definiert Regeln für die Transformation source tree → result tree
`xsl:output`	Ausgabemethode: XML, HTML, Text
`xsl:variable`	globale Variablen
`xsl:param`	globale Parameter

- Müssen top-level (direkt unter `xsl:stylesheet`) angegeben werden.
- Weitere Deklarationen: `xsl:import`, `xsl:include`, `xsl:attribute-set`, `xsl:key`, `xsl:decimal-format`, `xsl:strip-space`, ...

Instruktionen

`xsl:apply-templates`	definiert die Knotenmenge, bei der die Abarbeitung weitergeht.
`xsl:value-of`	definiert String-Wert mittels XPath-Ausdruck
`xsl:copy`	kopiert den momentanen Knoten
`xsl:copy-of`	kopiert den ganzen Sub–Baum
`xsl:for-each`	Schleife über eine Knotenmenge
`xsl:if`	bedingte Anweisung
`xsl:choose`	Alternativen (`xsl:when`, `xsl:otherwise`)

- Weitere Instruktionen: `xsl:sort`, `xsl:call-template`, `xsl:element`, `xsl:attribute`, `xsl:text`, `xsl:variable`, `xsl:param`, ...

Vereinfachte Stylesheets

- Falls nur ein Template verwendet werden soll, kann dieses auch weggelassen werden, und dessen Inhalt direkt angegeben werden.
- Dann können aber keine Deklarationen angegeben werden (z.B. `xsl:output` um die Ausgabe zu steuern)

```
<html xsl:version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml">
  <xsl:for-each select="//veranstaltung/titel">
    <i><xsl:value-of select="."/></i><br/>
  </xsl:for-each>
</html>
```

Templates

- Templates
- Built-in Templates
- Ablaufsteuerung
- Programmiersprache XSLT
- Select vs. Match
- Template-Auswahl

Templates

- Ein Stylesheet transformiert einen Source Tree in einen Result Tree (in XSLT 2.0 sind auch mehrere Source oder Result Trees möglich).
- Ein Stylesheet besteht aus Templates, die angeben, wie die Knoten des Source Trees verarbeitet werden sollen.

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

- Der Pattern ist ein eingeschränkter XPath-Ausdruck, der angibt auf welche Knoten das Template matcht, d.h. angewendet werden soll.
- Informell: Ein Pattern darf nur Kind-, Attributachsen und die Abkürzung // verwenden (innerhalb von Prädikaten allerdings beliebige Ausdrücke).

Templates

- Durch die `xsl:apply-templates` Instruktion werden die Knoten ausgewählt, auf die Templates angewendet werden sollen.

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

- Standardmäßig werden allen Kindelemente ausgewählt (das entspricht dem XPath Ausdruck "*").

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

- Mit dem `select` Attribut kann ein XPath Ausdruck angegeben werden, der die entsprechenden Knoten auswählt.

Built-In Templates

- In XSLT vordefinierte, sogenannte Built-In Templates geben das Standardverarbeitungsverhalten vor.
- Element- und Dokumentknoten erzeugen keine Ausgabe, und die Verarbeitung setzt bei den Kindelementen fort:

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

Built-In Templates

- Bei Attribut- und Textknoten wird der Stringwert in die Ausgabe geschrieben:

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

- Kommentare und Processing Instructions werden ignoriert:

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

Priorität

- Für die Verarbeitung eines Knotens wird immer genau ein Template ausgewählt.
- Falls mehrere Templates den Knoten matchen, dann bestimmt deren Priorität welches Template ausgewählt wird.
- Der Pattern im `match` Attribut bestimmt die Default-Priorität eines Templates.
- Allgemein gilt, je spezifischer dieser Ausdruck, desto höher die Default-Priorität.

Priorität

- Die exakten Regeln zur Berechnung sind komplex. Bei problematischen Fällen hilft ein Blick in die Spezifikation. Informell:
- Pattern testet ausschiesslich auf Knotenart: -0.5

```
*          @*          comment()
```

- Pattern testet auf qualifizierten Namen: 0

```
a          @b
```

- Pattern enthält Prädikate oder mehrere Steps: 0.5

```
a[1]          a/b
```

Priorität

- Die Priorität kann durch explizite Angabe einer Dezimalzahl im Attribut `priority` bestimmt werden.

```
<xsl:template match="..." priority="value">
```

- Das ist aber nur in seltenen Ausnahmefällen sinnvoll.

Ablaufsteuerung

- Es gibt immer ein context item, context position und context size (das ist auch der Auswertungscontext für XPath Ausdrücke).
- Zu Beginn ist das context item üblicherweise der Dokumentknoten des Source Trees.
- Es werden alle Templates gesucht, deren Patterns das aktuelle context item matchen, und abhängig von deren Priorität wird genau eines ausgewählt.

Ein Pattern matcht alle Knoten *a*, für die es irgendeinen Knoten *b* im Dokument gibt, für den der Pattern *a* selektieren würde.

- Diese informelle Definition gilt nur für Knoten mit Elternknoten, aber sinngemäß auch so, dass z.B. "/" den Dokumentknoten selektiert.

Ablaufsteuerung

- Ist ein Template ausgewählt, so werden die darin enthaltenen Instruktionen ausgeführt.
- Grundsätzlich ändern diese Instruktionen das context item nicht.
  Wichtigste Ausnahmen sind `xsl:apply-templates` und `xsl:for-each`.

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

- `xsl:apply-templates` führt dazu, dass für die selektierte Knotensequenz Templates angwendet werden.
- Das context item, context position und context size sowie die Reihenfolge der Templateaufrufe basieren auf dieser Knotensequenz.

Programmierparadigmen

- Imperative Programmiersprachen
  - Fallunterscheidungen mittels if/then/else
  - Wiederholung mittels Schleifen oder rekursiven Aufrufen
- XSLT als hauptsächlich deklarative Programmiersprache
  - Fallunterscheidung mit Pattern Matching (`<xsl:template match="...">`)
  - Wiederholung mit struktureller Rekursion (`<xsl:apply-templates/>`)
- Imperative Features in XSLT
  - `xsl:if`, `xs:for-each`, `xsl:choose` "vereinfachen" das Leben
  - XSLT ist auch ohne diese Kontrollstrukturen Turing-vollständig

Benannte Templates

- Falls ein imperativer Ansatz verfolgt werden soll, können benannte Templates definiert werden:

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

- Diese werden mit diesem Namen direkt aufgerufen:

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

- In der Praxis ist das aber nur in Ausnahmefällen sinnvoll.

select vs. match

- `select`-Attribut in `xsl:apply-templates`

Bestimme Menge aller Knoten Y, die man vom Knoten X aus mit dem `select`-Pfad selektieren kann. Für jedes Y wird dann ein passendes Template gesucht.

  - beliebige Achsen im Pfad erlaubt (macht z.B. Endlosschleifen möglich)
  - absolute Pfade möglich

- `match`-Attribut in `xsl:template`

Für einen bestimmten Knoten Y (der mittels `apply-templates` selektiert wurde) wird getestet, ob dieser von irgendeinem Knoten X aus mit dem `match`-Pfad selektiert würde.

  - nur child und attribute Achse sowie Abkürzung `//` erlaubt
  - absolute Pfade möglich (aber nicht immer sinnvoll)

Knoten-Erzeugung

- Literal Result Elements
- Attribute Value Templates
- `xsl:element`
- `xsl:attribute`
- Weitere Knoten-Typen
- `xsl:value-of`
- `xsl:copy`, `xsl:copy-of`
- Whitespace-Steuerung

Literal Result Elements

- Ein Literal Result Element (LRE) ist ein Element innerhalb eines Templates, dessen Namespace nicht der XSLT Namespace ist.
- Es wird inklusive Attributen und Namespaces in den Output durchgereicht.
- LRE darf selbst beliebigen Inhalt haben, insbesondere XSLT Instruktionen.

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

Attribute Value Templates

- Attribute innerhalb eines LRE sind möglich.
- Markup innerhalb eines Attributs ist nicht erlaubt:

```
<elem attr="<xsl:value-of select ='expr'/>">
```

- Ein Attribute Value Template (AVT) ist ein XPath Ausdruck in { }, der zur Erzeugung des Attributwerts ausgewertet wird.

```
<elem attr="{expr}"/>
```

xsl:element

- Erzeugt einen Element-Knoten.
- Das `name` Attribut legt den Elementnamen fest (kann ein AVT sein):

```
<xsl:element name="{@id}">
  ...
</xsl:element>
```

- Durch das optionale `namespace` Attribut wird der Namespace des erzeugten Elements festgelegt (kann ebenfalls ein AVT sein):

```
<xsl:element name="ex:autor" namespace="{@uri}">
  ...
</xsl:element>
```

- Das Element wird jedenfalls im angegebenen Namespace liegen, ob das Präfix erhalten bleibt ist implementierungsabhängig.

xsl:attribute

- Erzeugt einen Attribut-Knoten.
- Die Attribute `name` und `namespace` sind analog zu `xsl:element`.
- Kann im Unterschied zu AVTs den Namen bzw. Namespace dynamisch generieren und XSLT Instruktionen zur Erzeugung des Attributwerts enthalten.

```
<xsl:attribute name="{@id}" namespace="{@uri}">
  <xsl:apply-templates/>
</xsl:attribute>
```

- Ist sowohl bei LREs als auch bei `xsl:element` erlaubt.
- Muss vor Instruktionen oder LREs angegeben werden, die den Inhalt des Elements generieren.

Weitere Knoten-Typen

- `xsl:text` erzeugt Text, z.B. um den Whitespace besser zu steuern.

```
<xsl:text>, </xsl:text>
```

- `xsl:comment` erzeugt Kommentare, z.B. mittels Instruktionen:

```
<xsl:comment><xsl:value-of select="@id"/></xsl:comment>
```

- `xsl:processing-instruction` erzeugt PIs, z.B.:

```
<xsl:processing-instruction name="xml-stylesheet">
  "text/xsl" href="stylesheet.xsl"
</xsl:processing-instruction>
```

xsl:value-of

- `xsl:value-of` erzeugt einen Textknoten (der mit angrenzenden Text-Knoten verschmolzen wird).
- Der XPath Ausdruck des `select` Attributs wird ausgewertet und in einen String konvertiert (falls es nicht schon von diesem Typ ist).

```
<xsl:template match="person">
  <p>
    <xsl:value-of select="@vorname"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@nachname"/>
  </p>
</xsl:template>
```

xsl:copy

- `xsl:copy` kopiert das context item und nur dieses ("shallow copying")
- Bei Element-Knoten werden Namespace-Knoten ebenfalls kopiert, Attribut-Knoten und Kindknoten allerdings nicht!
- Das `xsl:copy`-Element selbst kann beliebigen Inhalt haben (ist nur relevant, wenn das context item ein Element-Knoten oder Dokument-Knoten ist).

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

xsl:copy-of

- `xsl:copy-of` kopiert alle durch das `select` Attribut angegebenen Knoten, inklusive deren etwaigen Attributen und Kindknoten ("deep copy").
- Bei String-Werten verhält sich `xsl:copy-of` gleich wie `xsl:value-of`.
- `xsl:copy-of` darf (im Gegensatz zu `xsl:copy-of`) selbst keine Kindelemente enthalten.

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

Whitespace-Steuerung

Whitespace-Verhalten ist steuerbar mit:

- `<xsl:text>`
- `<xsl:output indent='yes'/>`
- `normalize-space(expr)`
- `<xsl:strip-space elements='el1 el2'/>` definiert Elemente, bei denen Whitespaces getrimmt werden.
- Das Gegenteil ist `<xsl:preserve-space>` (Standardverhalten).
- Attribut `xml:space` in jedem Element erlaubt (mögliche Werte sind `default` bzw. `preserve`)

Kontrollstrukturen

- `xsl:for-each`
- `xsl:if`
- `xsl:choose`

xsl:for-each

- xsl:for-each ermöglicht Schleifen über einer Knotensequenz, die mittels XPath-Ausdruck des select-Attributs bestimmt wird.
- Die context size entspricht der Länge dieser Knotensequenz.
- Das context item und die context position werden für jedes item der Knotensequenz einzeln gebunden.

```
<xsl:template match='lehre'>
  <xsl:value-of select='veranstaltung[1]/titel'/>
  <xsl:for-each select='veranstaltung[position() &gt; 1]'>
    <xsl:text>, </xsl:text>
    <xsl:value-of select='titel' />
  </xsl:for-each>
</xsl:template>
```

for-each vs. apply-templates

- Üblicherweise austauschbar (voriges Beispiel mit `xsl:apply-templates`):

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

<xsl:template match='veranstaltung[1]'>
  <xsl:value-of select='titel' />
</xsl:template>

<xsl:template match='veranstaltung[position() &gt; 1]'>
  <xsl:text>, </xsl:text>
  <xsl:value-of select='titel' />
</xsl:template>
```

- Vergleich:
  - `for-each`: Einfache Formulierung von Joins (z.B.: mit Variablen)
  - `apply-templates`: Bessere Lesbarkeit durch geringere Verschachtelung

xsl:if

- xsl:if definitiert bedingte Anweisungen.
- test-Attribut enthält die Bedingung als XPath Ausdruck, der in einen boolschen Wert umgewandelt wird.
- Kein else-Zweig möglich.

```
<xsl:template match='veranstaltung'>
  <xsl:value-of select='titel'/>
  <xsl:if test='not(position()=last())'>, </xsl:if>
</xsl:template>
```

xsl:choose

- `xsl:choose` definiert Bedingte Anweisungen mit mehreren Alternativen.
- Enthält mindestens ein `xsl:when` Kindelement, das ausgeführt wird, wenn der XPath im `test`-Attribut erfüllt ist.
- Das optionale Kindelement `xsl:otherwise` gibt das Verhalten an, wenn kein `xsl:when` erfüllt wird.

```
<xsl:choose>
  <xsl:when test='expr1'>...das geschieht...</xsl:when>
  <xsl:when test='expr2'>...bzw. das...</xsl:when>
  <xsl:otherwise> und ansonsten das <xsl:otherwise>
</xsl:choose>
```

- Ausgeführt wird nur der erste Zweig, der erfüllt ist!

Variablen und Parameter

- Variablen
- `current()` Funktion
- Parameter

Variablen

- Das `xsl:variable` Element ist sowohl als (globale) Deklaration oder als Instruktion innerhalb eines Templates möglich.
- Variablen-Bindung mittels XPath Ausdruck im `select`-Attribut:

```
<xsl:variable name="var" select="expr"/>
```

- oder als Inhalt von `xsl:variable`:

```
<xsl:variable name="var"/>...</xsl:variable>
```

- Verwendung von Variablen mit $ vor dem Variablen-Namen:

```
<veranstaltung jahr="{$thisyear}">
```

Variablen

- Einmalige Zuweisung, d.h. Variablenwert wird nie mehr verändert (aber der Name kann in unterschiedlichen Kontexten unterschiedlich gebunden sein).
- Lokal definierte Variablen sind in aufgerufenen Templates nicht verfügbar (egal ob durch `xsl:apply-templates` oder `xsl:call-template`).
- Typische Anwendung bei Joins um unterschiedliche Kontexte zu verbinden:

```
<xsl:for-each select='/lehre/veranstaltung'>
  <xsl:variable name='x' select='.'/>
  <lva>
    <xsl:copy-of select='titel'/>
    <xsl:copy-of
    select='//mitarbeiter/veranstaltung[@nr=$x/@nr]/../name'/>
</lva>
</xsl:for-each>
```

current()

- Die `current()` Funktion gibt das context item zurück:

```
<xsl:value-of select ="current()"/>
```

- Auf äußerster Ebene äquivalent zu ".":

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

- Bleibt aber innerhalb der XPath Auswertung konstant, und erspart daher oft explizite Variablendefinitionen.
- Typische Anwendung sind daher Joins:

```
<xsl:value-of select="veranstaltung[@nr=current()/@nr]"/>
```

Parameter

- Das `xsl:param` Element entspricht bezüglich Definition und Verwendung genau dem `xsl:variable` Element.
- Der mittels `select` oder als Kindknoten definierte Inhalt ist aber nur für den Fall bestimmt, dass kein anderer Parameterwert übergeben wird.
- Die Bindung für globale Parameter wird durch den XSLT-Prozessor festgelegt (z.B. als Kommandozeilenargument).

Parameter

- Die Bindung lokaler Parameter erfolgt durch `xsl:with-param` als Kindelement von `xsl:apply-templates` oder `xsl:call-templates`:

```
<xsl:template match="teil[wichtig]">
  <xsl:call-template name='vielfaerbig'>
    <xsl:with-param name='farbe'>red</xsl:with-param>
  </xsl:call-template>
</xsl:template>

<xsl:template name='vielfaerbig'>
  <xsl:param name='farbe'>green</xsl:param>
  <teil farbe="{$farbe}" inhalt="{.}"/>
</xsl:template>
```

Weitere Features

- `xsl:sort`
- `document()` Funktion
- `generate-id()` Funktion
- XSLT 2.0
- `xsl:analyze-string`

xsl:sort

- xsl:sort ist als Kindelement von xsl:for-each oder xsl:apply-templates erlaubt.
- Bewirkt eine Veränderung der Ordnung der selektierten Knoten.
- Attribute von xsl:sort
  - select: string-expression als Sortier-Schlüssel
  - data-type: text, number
  - order: ascending, descending
  - case-order: lower-first, upper-first (d.h.: ob Groß- oder Kleinbuchstaben Vorrang haben)

xsl:sort

```
<xsl:for-each select='//mitarbeiter'>
  <xsl:sort select='name' order='ascending'/>
  <xsl:copy-of select='.'/>
</xsl:for-each>
```

- Mehrere `xsl:sort` Elemente können aufeinander folgen (definiert primäre, sekundäre, ... Sortierung):

```
<xsl:apply-templates select='employee'>
  <xsl:sort select='name/family'/>
  <xsl:sort select='name/given'/>
</xsl:apply-templates>
```

document()

- Die `document()` Funktion greift auf andere XML Dokumente zu.
- Mit `document("uri")` kann auf beliebige URIs zugegriffen werden.
- `document("")` erlaubt den Zugriff auf den Code des Stylesheets selbst.
- Man braucht aber für den Aufruf eines XSLT 1.0 Prozessors immer ein Quelldokument (kann auch ein "dummy" Dokument sein, das gar keinen Einfluss auf das Ergebnis hat).

```
<xsl:variable name='emps' select="document('merge2.xml')"/>
<xsl:template match='/'>
  <employees>
    <xsl:for-each select='$emps/employees/employee'>
      <xsl:copy-of select='.'/>
    </xsl:for-each>
  </employees>
</xsl:template>
```

generate-id()

- Die *string* generate-id(*node-set?*) Funktion erstellt einen eindeutigen Identifier-String für den ersten Knoten der Knotenmenge.
- Wenn das Argument fehlt, wird die id für das current item berechnet.
- Die id ist prozessorabhängig, muss aber für einen bestimmten Knoten bei wiederholten Aufrufen von generate-id immer gleich sein!
- Zum Beispiel für Links in einem HTML-Dokument:

```
<div id='{generate-id(expr)}'>...</div>
...
<a href='#{generate-id(expr)}'>...</a>
```

XSLT 2.0 und 3.0

Versionen von XSLT

- XSLT 1.0: Gute Unterstützung, z.B. in Browsern weit verbreitet
- XSLT 2.0: Neue Features, nicht in allen Browsern verfügbar
- XSLT 3.0: Noch nicht als W3C Recommendation verabschiedet

Neues in XSLT 2.0:

- Gravierende Änderungen des Datenmodells (wie XPath 2.0, XQuery 1.0)
- Verwendung von XPath 2.0
- `xsl:analyze-string`: Text mittels Regular Expressions behandeln
- `xsl:function`: Benutzerdefinierte Funktionen zur Verwendung in XPath
- Mehrere Ausgabedokumente möglich
- und vieles mehr (verbessertes Sortieren, Gruppieren, verarbeiten in Variablen gespeicherter XML-Fragmente, ...)

xsl:analyze-string (ab XSLT 2.0)

- Auf den String gegeben durch den Ausdruck im `select`-Attribut wird die Regular Expression im `regex`-Attribut angewendet.
- Der String wird in Substrings aufgeteilt, die jeweils der `regex` entsprechen bzw. nicht entsprechen.
- Für der `regex` entsprechende Teile wird das Ergebnis des Kindelements `xsl:matching-substring` ausgegeben, anderenfalls das von `xsl:non-matching-substring`.

```
<xsl:analyze-string select='abstract' regex='\n'>
  <xsl:matching-substring>
    <br/>
  </xsl:matching-substring>
  <xsl:non-matching-substring>
    <xsl:value-of select='.'/>
  </xsl:non-matching-substring>
</xsl:analyze-string>
```

XSLT und ...

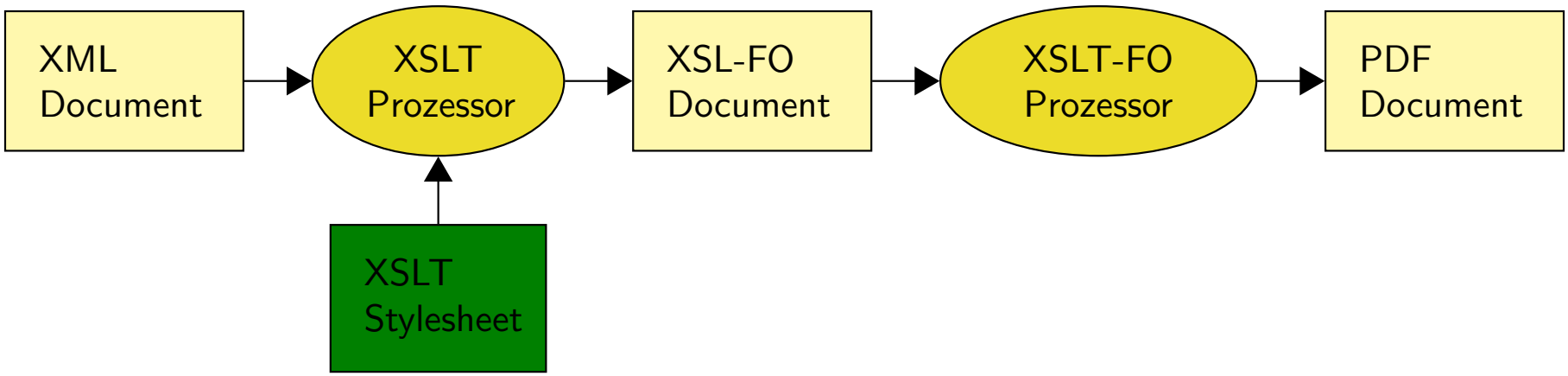
- XSLT und Stylesheetsprachen
- XSLT und XSL-FO
- XSLT Prozessoren
- XSLT und Java

XSLT und Stylesheetsprachen

- XML-Stylesheets:
  - CSS: Cascading Stylesheets
  - XSL: Extensible Stylesheet Language
- Bestandteile von XSL:
  - XSLT (XSL Transformations)
  - XSL-FO (XSL Formatting Objects)
  - baut sehr stark auf XPath auf
- XSL-FO:
  - W3C Recommendation seit Dezember 2006
  - Ursprünglich als Hauptteil von XSL gedacht
  - Hat wesentlich geringere Bedeutung als XSLT

XSLT und XSL-FO

- XSLT benötigt XSLT Prozessor
  - z.B. Xalan, Saxon, die meisten Browser
  - meist ohne XSL-FO: Erstellung von HTML, Text, SVG, etc.
- XSL-FO benötigt FO Prozessor
  - z.B.: Apache FOP für PDF Generierung
  - Formatierte Ausgabe (= Rendering)



XSLT Prozessoren

- Xalan (Teil des Apache Projekts; http://xml.apache.org/xalan-j/)

```
set CLASSPATH=xalan.jar;xercesImpl.jar
java org.apache.xalan.xslt.Process
    --IN test.xml -XSL test.xsl [-OUT out.xml]
```

- Saxon (von Michael Kay; http://saxon.sourceforge.net/)

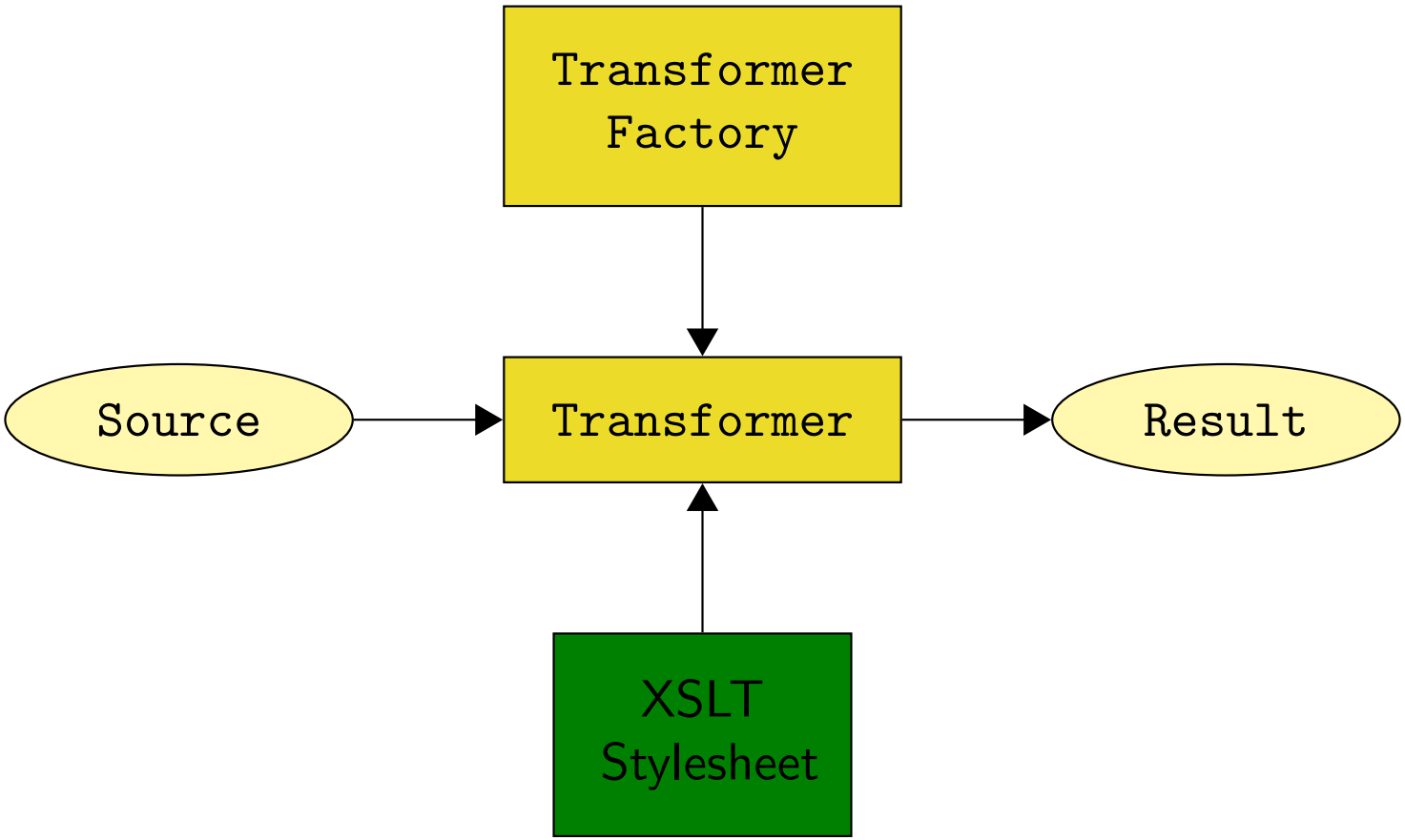
```
set CLASSPATH=saxon.jar;saxon-jdom.jar
java com.icl.saxon.StyleSheet
    test.xml test.xsl > out.xml
```

- XMLSpy (Altova; http://www.altova.com/xmlspy/)
  - enthält unter anderem: XSLT Prozessor, XSLT Debugger
  - Testversion für 30 Tage erhältlich

XSLT und Java

- Java API for XML Processing (JAXP)
- Die wichtigsten Klassen bzw. Interfaces:
  - `TransformerFactory`: wählt Implementierung aus und erzeugt neuen Transformer mittels XSLT Stylesheet
  - `Transformer`: die eigentliche Transformation
  - `Source`: Datei, SAX, DOM
  - `Result`: Datei, SAX, DOM

Überblick


Transformer Factory
Source
Transformer
Result
XSLT Stylesheet

Packages

- `javax.xml.transform`:
  Enthält die Klassen `TransformerFactory`, `Transformer`, `Source`, `Result`
- `javax.xml.transform.dom`:
  Enthält die Klassen `DOMSource` und `DOMResult` (= DOM- spezifische Implementierung der Interfaces `Source` und `Result`)
- `javax.xml.transform.sax`:
  Enthält die Klassen `SAXSource` und `SAXResult`
- `javax.xml.transform.stream`:
  Enthält die Klassen `StreamSource` und `StreamResult`

Mindest-Code

- Importe:

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
```

- Erzeugung der Sources und Results:

```
Source source = new StreamSource(in);
Result result = new StreamResult(out);
Source xslsource = new StreamSource(xsl);
```

- Factory/Transformer-Instanzierung, Transformation:

```
TransformerFactory tFactory =
  TransformerFactory.newInstance();
Transformer transformer =
  tFactory.newTransformer(xslsource);
transformer.transform(source, result);
```

DOM- und SAX-Input

- DOM:
  - `DOMSource` ist eine der drei möglichen Implementierungen des Source-Interface (package `javax.xml.transform.dom`)
  - Üblicherweise: `Source source = new DOMSource(doc);`
  - Ebenso möglich: `DOMSource` aus einem Sub-Tree des DOM-Baums erzeugen, d.h.: `Source source = new DOMSource(node);`
- SAX:
  - `SAXSource` ist eine der drei möglichen Implementierungen des Source-Interface (package `javax.xml.transform.sax`)
  - Erzeugung der `SAXSource`: benötigt `InputSource` und `Reader`:

```
inputSource isource = new InputSource(in);
SAXSource ssource = new SAXSource(reader, isource);
```

XML-Filter

- Erzeugung eines XML-Filters mittels XSLT-Stylesheet:
  - Dazu ist eine `SAXTransformerFactory` (Subklasse von `TransformerFactory`) erforderlich.
  - Mit `TransformerFactory.newInstance()` wird in den meisten Implementierungen eine `SAXTransformerFactory` erzeugt (d.h. Cast auf `SAXTransformerFactory` ist üblicherweise möglich).
  - Mit `newXMLFilter()` anstelle von `newTransformer()` wird der XML-Filter erzeugt.
- Beispiel:
  - SAX-Parser: Liest XML-Dokument
  - XML-Filter1: konsumiert Events des Parsers
  - XML-Filter2: konsumiert Events des XML-Filter1
  - Transformer: Wandelt den XML-Output des XML-Filter2 (als SAX-Events) in einen Stream um, der auf Datei geschrieben wird.