namics Developer Days – 21./22.6.2002

XSLT: Transformation von XML

Erik Wilde
ETH Zürich
http://dret.net/netdret/

Übersicht

- XML als Format für strukturierte Daten
  - Zugriff oftmals auf XML-spezifische Teile
- XSL als Stylesheet-Sprache für XML
  - Darstellung im Browser als Zielvorstellung
  - XSLT als ein Teil dieses Prozesses
- XSLT als Transformationssprache für XML
  - heute oftmals anders gebraucht als geplant
  - Kennenlernen und Berücksichtigen der Limitierungen

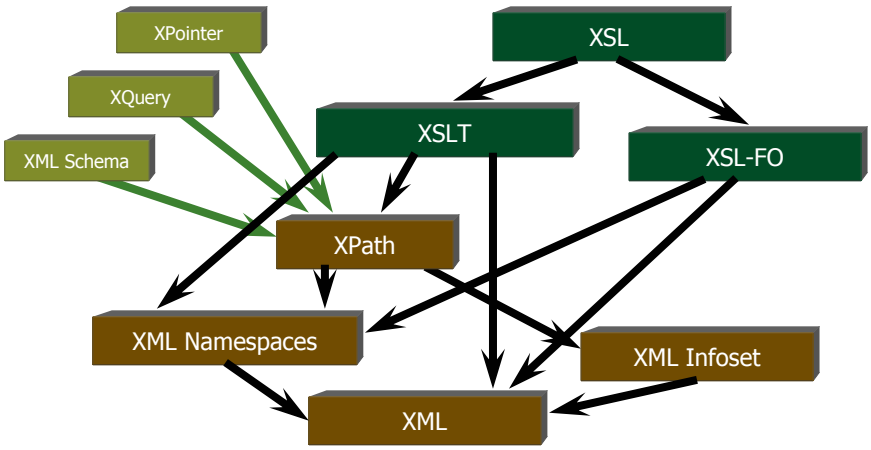
XML für strukturierte Daten

- XML definiert semistrukturierte Daten
  - grössere Variabilität als relationales Modell
  - mehr verschiedene Arten des Zugriffs
- verschiedene Anwendungsszenarien für XML
  - Zugriff auf bestimmte Teiles des XML
    - relatin einfach möglich z.B. über DOM XPath
  - Transformation orientiert an der Struktur des XML
    - komplexerer Zusammenhang zwischen XML und Code
    - populär für diese Anwendung ist das *Visitor* Design Pattern
      - ergibt sauber getrennten, aber unübersichtlichen Code
    - besser ist eine dedizierte Sprache für die Transformation

XSL Transformations (XSLT)

- ursprünglich Teil des XSL Standards
  - jetzt per Referenz in den Standard integriert
  - orientiert an Scheme, einem Lisp-Dialekt
- Potential wurde erkannt
  - Trennung von XSL in XSLT und XSL-FO
  - XSLT transformiert in *XSL Formatting Objects*
  - XSL-FO werden zur Darstellung benutzt
- XSLT wurde weiter geteilt
  - XPath für die Selektion von Teilen eines Dokuments
  - "der Rest", die Kontrollstrukturen
- besserer Name: *XML Transformation Language*

"Übersicht" XSL-Standards



XSLT Versionen

- aktueller Standard ist XSLT 1.0
  - stabile W3C Recommendation (11/1999)
- XSLT 1.1 wurde 2000 begonnen
  - inkrementelle Verbesserungen
    - mehrere Ausgabedokumente
    - Result Tree Fragments als Node Sets
    - Kompatibilitätsmechanismen
- XQuery braucht XPath
  - XPath 2.0 ist in Entwicklung
  - XSLT 2.0 benutzt XPath 2.0
  - weitere XSLT 1.1 Entwicklung wurde eingestellt

XSLT: My First Stylesheet!

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

</xsl:stylesheet>
```

- was an diesem Programm auffällt
  - es ist ein XML-Dokument
  - es benutzt XML Namespaces
  - es ist eine leere Hülle (nur ein Container-Element)
  - und es funktioniert trotzdem!

"First Stylesheet" unter der Lupe

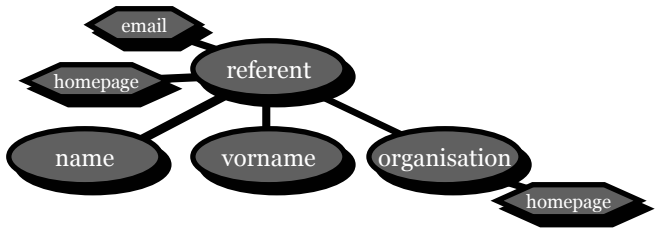
- Ergebnis der Anwendung auf ein Dokument
  - der "Text" des Dokuments erscheint
    - Text-Inhalt der Elemente, aber nicht der Attribute
  - funktioniert bei beliebigen Dokumenten
- Default-Verhalten
  - ungewöhnlich für eine Programmiersprache
  - praktisch für simple "Default-Formatierung"
- inkrementelle Entwicklung einfach möglich
  - beginnen mit einem leeren Stylesheet
  - stufenweise Verfeinerung in ausführbaren Schritten

XSLT: My Second Stylesheet!

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

<xsl:output method="text"/>

<xsl:template match="*">
  ( Element: <xsl:value-of select="local-name()"/>
  <xsl:apply-templates select="* | @*"/> )
</xsl:template>

<xsl:template match="@*">
  Attribute: <xsl:value-of select="local-name()"/>
</xsl:template>

</xsl:stylesheet>
```


Beispieldokument

```
<referent email="xml@dret.net" homepage="http://dret.net/">
   <vorname>Erik</vorname>
   <name>Wilde</name>
   <organisation homepage="http://www.tik.ee.ethz.ch/">ETH
   Zürich</organisation>
</referent>
```



Resultat des "Second Stylesheet"

```
( Element: referent
  Attribute: email
  Attribute: homepage
  ( Element: vorname )
  ( Element: name )
  ( Element: organisation
    Attribute: homepage ) )
```

- Whitespace nachträglich von Hand verändert!

"Second Stylesheet" unter der Lupe

- Steuerung der Ausgabe möglich
  - normalerweise XML als Ausgabe
  - Text-orientierte Ausgabe ebenfalls erlaubt
- Programmsteuerung durch *Template Rules*
  - Rekursion als Normalfall
  - Auswahl der Template Rules durch XSLT Prozessor
  - Ausführung durch das Dokument gesteuert
- inkrementelle Entwicklung
  - ohne die zweite Template Rule ebenfalls lauffähig
  - aber: Default-Verhalten in diesem Fall ungünstig

XSLT: Hello World!

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

<xsl:template match="/">
  <xsl:text>Hello World!</xsl:text>
</xsl:template>

</xsl:stylesheet>
```

- Abarbeitung beginnt beim Root Node
- Ausgabe eines Text Nodes
- Ende der Abarbeitung

XSLT als Programmiersprache

- Transformation von XML für die Darstellung
  - gedacht für den Einsatz im Browser
    - limitierte Ausführungsumgebung
    - keine Kontrolle der Ausführungsumgebung
  - spezialisiert auf ein Anwendungsgebiet
- XSLT wird vielfach falsch eingesetzt
  - missverstanden als general purpose Sprache
- XSLT und Software Engineering
  - viele implizite Typenkonvertierungen
  - Implementierungsfreiheiten für XSLT Prozessoren
  - erfordert viel Aufwand und Eigendisziplin

XSLT aus der Ferne betrachtet

- Eingabe ist ein XML-Dokument
  - etwas genauer betrachtet ein Node Tree
  - leichte Verallgemeinerungen gegenüber XPath
    - *External General Parsed Entities* sind auch erlaubt
  - *Whitespace Stripping* als erster Schritt
- Transformation als Ausführung des XSLT
  - beliebige Komplexität der Abarbeitung
- Ausgabe ist XML, HTML oder Text
  - XML ist der Normalfall (erlaubt Konkatenation)
  - HTML als populäres Präsentationsformat
  - Text ohne Markup-Struktur

Funktionales Programmieren

- XSLT ist eine funktionale Programmiersprache
  - für viele ein neues Konzept
  - bekannt aus Sprachen wie Lisp und Scheme
- alles ist eine Funktion
  - Funktionen sind *side effect free*
    - hängen einzig von den Eingabeparametern ab
    - können in beliebiger Reihenfolge aufgerufen werden
    - es gibt keine globalen Variablen und keinen Status
  - gute Unterstützung für inkrementelles Rendering
- keine Variablen in der Sprache
  - `<xsl:variable>` kann nicht geändert werden

Template Rules

- Templates werden auf zwei Arten aufgerufen
  - über das *Match Pattern* (`match` Attribut)
  - als *Named Template* (`name` Attribut)
- Templates können Parameter haben
  - werden beim Aufruf gesetzt
  - sind (neben dem Context Node) die Eingabe
  - werden als `<xsl:param>` deklariert (kein Typ!)
- enthalten einen *Template Rule Body*
  - ähnlich dem Block-Konzept in anderen Sprachen
  - wird an weiteren Stellen verwendet

Patterns

- sind eine Untermenge von XPath
  - müssen als Resultat ein Node Set liefern
  - werden von XSLT definiert (nicht von XPath)
- werden an vier Stellen verwendet
  - `<xsl:template match="...">`
  - `<xsl:key match="...">`
  - `<xsl:count from="..." to="...">`
- Menge von Selektionskriterien für Nodes
  - werden die Bedingungen erfüllt, matcht der Node
  - anderfalls nicht (das Pattern trifft nicht zu)

Template Rule Anwendung

- Templates werden auf zwei Arten aufgerufen
  - `<xsl:apply-templates>` selektiert Nodes
    - arbeitet die Nodes mit neuem Kontext ab
  - `<xsl:call-template>` für ein Named Template
    - keine Änderung des Kontext
- `<xsl:apply-templates>`
  - optional können die Nodes sortiert werden
  - Ausführung ist durch das Dokument bestimmt (push)
- `<xsl:call-template>`
  - ähnlich einem Prozeduraufruf
  - Ausführung durch das Stylesheet bestimmt (pull)

Built-in Template Rules

- fest eingebaut in jeden XSLT Prozessor
  - definiert im XSLT Standard
  - definieren das Default-Verhalten
    - deshalb erzeugt ein "leeres" XSLT eine Ausgabe
- streng genommen gibt es also immer Konflikte
  - ausser für Namespace Nodes
    - können nicht durch ein Pattern selektiert werden
  - Conflict Resolution ist essentiell in XSLT!
- Built-in Rules werden als importiert behandelt
  - vor allen anderen `<xsl:import>` Anweisungen
  - deshalb eine geringere *Import Precedence*

Eingaben in XSLT

- Eingabe ist immer ein XPath Node Tree
  - wird aus dem Infoset erzeugt
- XSLT Prozessor arbeitet auf dem Node Tree
  - XSLT sagt nichts über die Herkunft des Trees
  - oftmals ein vorgeschalteter XML Parser
  - kann auch ein synthetisiertes DOM sein
- u.U. kann der Parser wichtig sein
  - z.B. Validierung machen (`ID`/`IDREF`, Defaults)
  - z.B. Whitespace Handling (Stripping von Text Nodes)
  - Steuerung des Parsers manchmal notwendig

Ausgaben in XSLT

- Ziel abhängig von der Anwendung
  - oftmals ein XML-Dokument (Result Tree)
  - definiert durch `<xsl:output>`
- Varianten für die Ausgabe
  - Text-orientierte Ausgabe möglich
    - sinnvoll als Eingabe in Batch-Prozessen (z.B. LaTeX)
  - HTML-Ausgabe möglich
    - Ausgabe ist kein XML-Dokument
    - XHTML als Methode erst in XSLT 2.0 definiert
  - binäre Daten können nicht erzeugt werden
    - falls unbedingt nötig: Hex-Zeichen erzeugen und konvertieren
    - nicht das Anwendungsgebiet von XSLT

Iterationen

- <xsl:for-each> erlaubt Schleifen
  - Selektion eines Node Sets
    - absolut oder relativ zum aktuellen Kontext
  - Ausführung des Template Body für alle Nodes
  - die Nodes können sortiert werden
    - <xsl:sort> Element(e) innerhalb des <xsl:for-each>
- der Kontext wird für jeden Node neu gesetzt
  - falls alter Kontext gebraucht wird: Variable setzen
- ähnlicher Effekt zu <xsl:apply-templates>
  - Pull vs. Push Processing

Conditional Code

- <xsl:if> für einfache Bedingung (kein "else")

```
<xsl:if test="$somecondition">
  <xsl:text>$somecondition is true()</xsl:text>
</xsl:if>
```

- <xsl:choose> für Auswahl aus Alternativen
  - <xsl:when> darf wiederholt werden ("else if")
  - <xsl:otherwise> als letzter "else" Teil

```
<xsl:choose>
  <xsl:when test="$count > 2"><xsl:text>, and
    </xsl:text></xsl:when>
  <xsl:when test="$count > 1"><xsl:text> and
    </xsl:text></xsl:when>
  <xsl:otherwise><xsl:text> </xsl:text></xsl:otherwise>
</xsl:choose>
```

Variablen

- Zuweisung eines Wertes
  - mittels des select Attributs
    - `<xsl:variable name="name" select="'dret'"/>`
  - oder der Inhalt des `<xsl:variable>` Elements
    - komplette Template Bodys sind als Inhalt erlaubt
- globale und lokale Variablen
  - in beiden Fällen ein fester Wert pro Ausführung
  - global: pro Ausführung des Stylesheets
  - lokal: pro Ausführung des Templates
- Variablen können nicht verändert werden
  - aber sie variieren pro Ausführung

Verwendung von Variablen

1. Speicherung von häufig verwendeten Werten
   - machen ein Stylesheet übersichtlicher
   - machen die Ausführung u.U. effizienter
2. Speicherung von Kontext
   - Erhalten eines Kontexts in einem neuen Kontext
   - meistens bei Iterationen mit `<xsl:for-each>`
3. Speicherung eines *Temporary Tree*
   - erlaubt komplexe Zwischenresultate
   - Wiederverwendung eines Result Tree Teiles
   - Anwendung von XPaths (nur XSLT 1.1)

Node Sets und Variablen

- XSLT 1.0 definiert *Result Tree Fragments*
  - eine Variable kann einen Baum enthalten
  - nur einfache Funktionen sind erlaubt (keine Paths)
  - Grund waren Implementierungsüberlegungen
  - viele XSLT Prozessoren implementieren `node-set()`
    - nicht-standardisierte *Extension Function*
- XSLT 1.1 kennt keine Result Tree Fragments
  - eine Variable kann einen Baum enthalten
  - normale XPaths können darauf angewendet werden
    - z.B. `$variable//name[vorname='erik']`
  - sehr praktisch in vielen Situationen

Programmierphilosophie

- `<xsl:apply-templates>` für Match Patterns
- `<xsl:for-each>` für Iterationen
- Templates vs. explizite Schleifen
  - eine Frage der Methodik
  - eine Frage der Wartbarkeit
    - kann mit `<xsl:call-template>` verbessert werden
  - eine Frage der Schemas
  - eine Frage der Dokumente
  - eine Frage der Gewöhnung
- "relational" vs. semi-structured XML

<xsl:apply-templates>

- zweistufiger Prozess
  - Selektion von Nodes (`select` Attribut)
  - Suchen von passenden Templates (*Match Pattern*)
- vom Dokument gesteuert
  - Push-Processing
  - Ausführung u.U. abhängig von Imports
- besser geeignet für permissive Schemas
  - Inhalt eines Elementes stark variabel
  - Behandlung durch das Dokument gesteuert
    - dynamische Auswahl der passenden Templates

<xsl:for-each>

- einstufiger Prozess
  - Selektion von Nodes (`select` Attribut)
- vom Style-Sheet gesteuert
  - Pull-Processing
  - vorhersagbare Ausführung
- besser geeignet für restriktive Schemas
  - Inhalt eines Elementes recht klar eingegrenzt
  - Behandlung genau dieses Inhaltes
    - weniger flexibel hinsichtlich variabler Dokumente

Parameterübergabe

- Parameter übergeben Werte an Funktionen
  - Parameter des Stylesheets
    - `<xsl:param>` Element als Kind von `<xsl:stylesheet>`
  - Parameter von Templates
    - `<xsl:param>` Element als Kind von `<xsl:template>`
- Default-Wert kann angegeben werden
  - wird verwendet, falls kein Wert angegeben wird
    - falls kein Default und kein Wert: Empty String
- keine Typen (wie Variablen)
  - keine Typüberprüfung möglich
- Referenzierung wie eine Variable (`$name`)

XSLT Extensions

- XSLT ist eine einfache Sprache
  - Design für die Ausführung in einfacher Umgebung
  - Design für die Ausführung einfacher Programme
- XSLT wird für vieles andere eingesetzt
  - oft Server-seitig in B2B Anwendungen
  - wesentlich weitergehende Anforderungen
- viele XSLT Prozessoren bieten Erweiterungen
  - praktisch und viele Erleichterungen
  - Bindung an einen spezifischen XSLT Prozessor
  - Sammlung als EXSLT (`http://www.exslt.org`)

Extension Elemente und Attribute

- definieren zusätzliche Anweisungen
  - benutzen QNames aus einem Extension Namespace
  - sind spezifisch für einen XSLT Prozessor
- Saxon definiert
  - `<saxon:while>` für konditionale Iterationen
  - `<saxon:assign>` für Wertzuweisung zu Variablen
    - Variable muss auf `saxon:assignable="yes"` gesetzt sein
- Xalan definiert
  - `<xalan:write>` für mehrere Ausgabedokumente
  - `<xalan:for-each-token>` für Token-Schleifen
- weitere Prozessor-spezifische Erweiterungen

Extension Functions

- definieren zusätzliche Funktionen
  - benutzen QNames aus einem Extension Namespace
  - erweitern die XSLT/XPath Function Library
- Saxon definiert
  - `evaluate` für dynamische XPath Expressions
  - `path` generiert XPath für den Context Node
- Xalan definiert
  - `intersection` für Node Set Schnittmengen
  - `line-number` für Context Node Zeilennummer
- weitere Prozessor-spezifische Erweiterungen

Extensions selbstgemacht

- Integration eigener Funktionen in XSLT
  - in Form von eigenen Elementen
  - in Form von eigenen Attributen
- spezifisch für einen XSLT Prozessor
  - kein etablierter Standard für ein API
  - Implementierung ist nicht (einfach) portabel
- Implementierung eigener Elemente
  - kompliziertere Integration (mehrere Phasen)
- Implementierung eigener Funktionen
  - erhalten Context und Argumente übergeben

Zusammenfassung

- XSLT als einfache Programmiersprache
  - überschaubare Anzahl von Anweisungen
  - Komplexität in der Anwendung
- XSLT für einfache Anwendungen
  - nicht gut geeignet für grosse Projekte
  - Sprach-Design beruht auf Browser-Szenario
- komplexe Anwendungen u.U. strukturieren
  - Vorverarbeitungen mit anderen Sprachen
    - z.B. Textanalysen und Markup-Generierung mit Perl
  - XSLT nur für den Transformationsteil einsetzen