XML Vorlesung FH Aargau, SS 2006

XSL Transformation (XSLT) Teil IV

Erik Wilde

3.7.2006

http://dret.net/lectures/xml-fhnw-ss06/

Übersicht

- XSLT und Schemas
  - Schemas für die Eingabe
  - Schemas für die Ausgabe
- Modularisierung
- Import Precedence
- XSLT Extensions

XSLT und Schemas für XML

- XSLT verarbeitet well-formed XML
  - Eingabe ist ein Infoset (oder DOM)
  - Verarbeitung gemäss der Regeln des Programms
- es gibt verschiedene Schemasprachen für XML
  - DTDs aus dem XML Standard selber
  - XML Schema als Weiterentwicklung von DTDs
    - separater Standard, zunehmende Akzeptanz
  - Schematron als regelbasierte Schemasprache
    - im Gegensatz zu den Grammatiken von DTD & XML Schema
  - in vielen Anwendungsfällen ist valides XML nötig
    - B2B Anwendungen erwarten korrekte Daten

Warum Schemas für die Eingabe?

- XSLT Code basiert auf Annahmen
  - die XPath Ausdrücke
    - `/personen/person/name/nachname` vs. `//nachname`
  - die Struktur der Templates
- invalides XML verletzt diese Annahmen
  - Fehler in der Verarbeitung
  - unvorhersehbare Resultate
- Annahmen sind wichtig beim Programmieren
  - ohne Schema programmiert es sich mühsam
  - Kenntnis des Schemas ermöglicht Effizienz

Schemas für die XSLT Eingabe

- Parser liefert dem XSLT Prozessor das DOM
  - manchmal austauschbar, manchmal eingebaut
- Parser kann parametrisiert werden
  - Validierung als Schritt vor dem XSLT Prozessor
  - DTD und/oder XML Schema Validierung
    - falls der Parser auch XML Schema unterstützt
    - kann per Parser Option eingestellt werden
- invalides XML bricht die Verarbeitung ab
  - Eingabebedingung nicht erfüllt
  - Korrektur oder Rückweisung (je nach Szenario)

Warum Schemas für die Ausgabe?

- oftmals wird das Resultat weiterverarbeitet
  - Eingabe in eine weitere Verarbeitungsstufe
    - die ihrerseits auf gewissen Annahmen basiert
  - Ausgabe sollte Regeln gehorchen
- Ausgabe wird dynamisch zusammengesetzt
  - abhängig vom Aufbau des Eingabedokuments
- Struktur der Ausgabe schwer vorhersehbar
  - einfacher bei *Pull-Processing* Style Sheets
  - schwieriger bei *Push-Processing* Style Sheets
  - immer schwierig bei nicht-trivialen Style Sheets

Schemas für die Ausgabe

- XSLT erzeugt einen Node Tree
  - dieser wird als well-formed XML serialisiert
  - u.U. wird er optimiert (z.B. Namespace Nodes)
  - u.U. wird er direkt als Baum (DOM) weitergegeben
- Validierung des Resultats ist einfach
  - Parser verwenden und validieren lassen
  - weiterer Schritt in der Verarbeitungspipeline
- Validierungen sind u.U. aufwendig
  - stark abhängig von Schema und Parser
  - nur an strategischen Stellen einsetzen

Modularisierung

- lange und unübersichtliche Style Sheets
  - sowieso problematisch wegen der XML-Syntax
  - Trennung in verschiedene Teile
- Verwendung mehrerer Teile
  - bessere Übersichtlichkeit
  - Wiederverwendung von Style Sheets
  - Ergänzung von Teilen in verschiedenen Kontexten
    - Wiederverwendung von Template Rules

Varianten der Modularisierung

- textuelles Einbinden
  - `<xsl:include>` als Top Level Element
- logisches Importieren
  - `<xsl:import>` für die logische Strukturierung
  - importierte Teile haben niedrigere Priorität
  - `<xsl:apply-imports>` für die Ausführung importierter Template Rules

Struktur eines XSLT Programms

- ein Style Sheet ist die Wurzel
  - als Style Sheet dem XSLT Prozessor übergeben
  - als Style Sheet im XML Dokument angegeben
- weitere Teile werden referenziert
  - `<xsl:include>` für textuelles Einbinden
  - `<xsl:import>` für logisches Importieren
  - beliebig weit rekursiv fortgesetzt
  - Resultat ist ein Baum von Style Sheets
- Zusammensetzung vor der Ausführung
  - alle referenzierten Teile müssen existieren

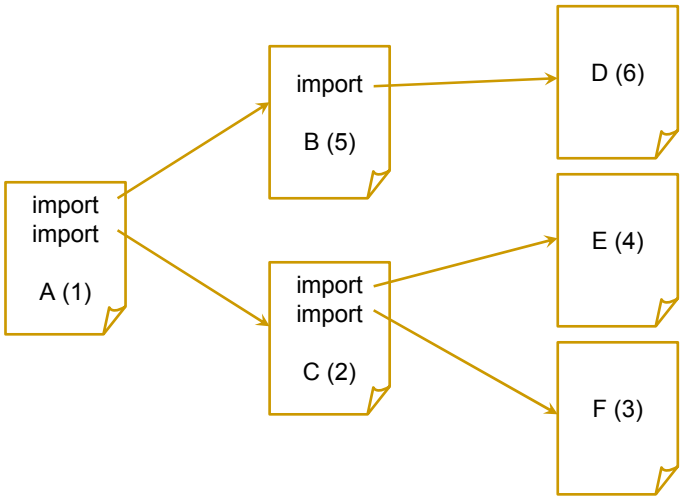
Textuelles Einbinden von Stylesheets

- `<xsl:include>` für textuelles Einbinden
  - muss auf dem Top Level erscheinen
  - entspricht dem Copy&Paste des Style Sheets
    - abgesehen von dessen Document Element
- nicht tauglich für lose gekoppelte Style Sheets
  - vergleichbar einem simplen Preprocessor
  - keinerlei Einfluss auf die Interpretation
- lose gekoppelte Style Sheets
  - logisches Einbinden mit niedrigerer Priorität
  - hat einen erwünschten Einfluss auf die Interpretation
    - definierte Dinge können überschrieben werden

Importieren von Stylesheets

- `<xsl:import>` für logisches Importieren
  - muss auf dem Top Level erscheinen
  - muss als erstes Element auf Top Level erscheinen
- Imports definieren die *Import Precedence*
  - Importiertes geringer gewichtet als das Importierende
  - zuerst importiert ist geringer gewichtet
- Includes ändern die Import Precedence nicht
  - gleich gewichtet wie das includende Style Sheet

Import Precedence am Beispiel



Ergänzen von Template Rules

- Redefinieren importierter Template Rules
  - von XSLT ausdrücklich erlaubt
  - wird durch die *Import Precedence* geregelt
- u.U. soll die Template Rule ergänzt werden
  - neuer Code im eigenen Style Sheet
  - bestehender Code im importierten Style Sheet
- `<xsl:apply-imports>` in Template Rule
  - suche nach einer passenden Template Rule
  - definiert anhand der *Current Template Rule*
    - gleicher Mode
    - in einem importierten Style Sheet

Zusammenfassung Modularisierung

- XSLT bietet Modularisierung
  - in zwei unterschiedlichen Varianten
- XSLT ist tendenziell unübersichtlich
  - Grund dafür ist die XML Syntax
  - eher kürzere Programme schreiben
- `include` ermöglicht textuelles Kombinieren
- `import` ermöglicht logisches Kombinieren
  - die *Import Precendence* definiert eine Hierarchie

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`)

Umgang mit Extensions

- Extensions werden mit Namespaces identifiziert
  - eindeutige Identifikation
  - Extensions werden mit QNames angesprochen
    - gilt für Elemente, Attribute und Funktionen
- Abfrage von vorhandenen Extensions
  - `element-available()` Funktion
  - `function-available()` Funktion
- definierter Fallback-Mechanismus
  - `<xsl:fallback>` innerhalb eines Elements
  - wird nur im Fehlerfall ausgeführt
    - kann Ersatzfunktionalität oder Terminierung veranlassen

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

- Übung macht den Meister
  - weniger Code, aber schwieriger zu schreiben
  - Umgewöhnung auf rekursive Algorithmen
- sehr nützliches Tool für den Umgang mit XML
- XSLT 2.0 wird 2005 aktuell
  - XSLT selber wird sich weniger stark ändern
  - XPath 2.0 wird XPath 1.0 stark erweitern