Workshop

XML-Workshop November 2002

XML-Workshop

Inhaltsverzeichnis

Einführungsbeispiel

Hello World

Workshop Vorbereitung zu
Einführungsbeispiel: Hello World

Werkzeuge:
MS Internet-Explorer 5.5 (SP2) oder 6
Einfacher Editor (Wordpad ...)
XMLSpy-Editor (www.xmlspy.com)

XML-Sprachfamilie:
XML-Elemente und Attribute
XML-Deklaration mit optionalem encoding-Attribut

```
<?xml version="1.0" encoding="iso-8859-1" ?>
```

Konzept der „Wellformedness" und „Validität"
Embedded Document Type Definition (DTD)
- Element-Deklaration und Inhaltsmodell
- Attribut-Deklaration und Typdefinition
- General Entities

XML-Processing Instruction (PI) für Stylesheet-Deklaration

```
<xml-stylesheet type="text/xsl" href="stylesheet.xsl"
```

XSLT- Stylesheet und XSLT-Vokabular

```
<xsl:stylesheet version="1"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

- Root-Template
- Templates allgemein

XSLT-Elements

```
xsl:template match="xpath-expression"
xsl:apply-template select="xpath-expression"
xsl:value-of select="xpath-expression"
```

XPath-Expressions

```
path[prediction]
```

- Punkt-Operator .
- Rekursiv-Operator //
- Ist-Operator =

XSLT-Functions

```
string(nodevalue)
substring(string, von, laenge)
contains(string, substring)
concatenate(string, string ...)
```

Zielsprache:
HTML und CSS

Einführungsbeispiel Hello-World

Aufgabenstellung:

Kennenlernen der XML-Sprachfamilie und XML-Werkzeuge sowie grundlegender Konzepte und Vorgehensweisen.

Zielsetzung:

Einfache „Document Type Definitions (DTD)“ modellieren, wellformed XML-Dokumente und Stylesheets schreiben können. Erkennen der Bedeutung einer DTD und der Vorteile des XML-Konzeptes: Strikte Trennung eines Dokumentes in Struktur, Inhalt und Präsentation.

Schritt 1: Erstellen eines XML-Dokumentes

Die Struktur und das Vokabular eines Gruß-Dokumentes (HelloWorld) wird frei erfunden. Für diese Arbeit wird ein einfacher Editor verwendet und das Ergebnis unter dem Namen `helloworld_1.xml` im entsprechenden Verzeichnis abgespeichert und anschließend mit einem Browser angezeigt.

```
<!–HelloWorld_1 Beispiel -->
<HelloWorld>
   <Greetings language="deutsch">Hallo Welt</Greetings>
   <Greetings language="deutsch">Hallo Welt</Greetings>
   <Greetings language="deutsch">Hallo Welt</Greetings>
</HelloWorld>
```

Schritt 2: Sinnvolles und Unsinniges

Wir ergänzen das Dokument mit Sinnvollem und Unsinnigem. Der Test im Browser zeigt ein unerwartetes Ergebnis (Zeichensatz-Problem). Wir erkennen die Bedeutung des XML-Prologs mit seinem optionalen encoding-Attribut.

```
<!–HelloWorld_2 Beispiel -->
<HelloWorld>
Freier Text kann hier sinnvoll sein...
   <Greetings language="deutsch">Hallo Welt</Greetings>
   <Greetings language="englisch">Hello World</Greetings>
   <Greetings language="persisch">Salam</Greetings>
   <Korb>Äpfel und Birnen haben hier nichts verloren!</Korb>
</HelloWorld>
```

Schritt 3: Erstellen einer Document Type Definition DTD

Die Struktur und das Vokabular eines Gruß-Dokumentes (HelloWorld) wird als Regelwerk definiert. Dieses Regelwerk wird in das XML-Dokument eingebettet und alles unter dem Namen `helloworld_3.xml` abgespeichert.

```
<?xml version="1.0" encoding="iso8859-1"?>
<!-- HelloWorld_3 Beispiel (Fehler!) -->
<!DOCTYPE HelloWorld [
   <!ELEMENT HelloWorld (Greetings)+>
   <!ELEMENT Greetings (#PCDATA)>
   <!ATTLIST Greetings language CDATA #REQUIRED>
]>
<HelloWorld>
Freier Text kann hier sinnvoll sein ...
   <Greetings language="deutsch">Hallo Welt</Greetings>
   <Greetings language="englisch">Hello World</Greetings>
   <Greetings language="persisch">Salam</Greetings>
   <Korb>Äpfel und Birnen haben hier nichts verloren</Korb>
</HelloWorld>
```

XML-Prolog mit encoding-Attribut

Embedded DTD

Schritt 4: Test mit einem validierenden Editor

Das XML-Dokument wird vom Browser korrekt angezeigt, obwohl es nicht unserem Regelwerk (DTD) entspricht. Mit einem validierenden Editor (zB. XML-Spy) lassen sich die Regelverstöße aufzeigen. Wir entfernen den „unsinnigen“ Korb aus unserem XML-Dokument und modellieren die DTD neu, um den „sinnvollen“ Freien Text zu erlauben. Für das language-Attribut wählen wir eine alternative Modellierung mit einer Wert-Enumeration.

```
<?xml version="1.0" encoding="iso8859-1"?>
<!-- HelloWorld_4 Beispiel -->
<!DOCTYPE HelloWorld [
   <!ELEMENT HelloWorld (#PCDATA | Greetings)*>
   <!ELEMENT Greetings (#PCDATA)>
   <!ATTLIST Greetings language (deutsch | englisch | persisch) "deutsch">
]>
<HelloWorld>
Freier Text kann hier sinnvoll sein ...
   <Greetings language="deutsch">Hallo Welt</Greetings>
   <Greetings language="englisch">Hello World</Greetings>
   <Greetings language="persisch">Salam</Greetings>
</HelloWorld>
```

Schritt 5: Externe DTD, Verwendung von Entities

Wir lagern das Regelwerk als externe DTD aus (`helloworld.dtd`). Das XML-Dokument enthält nun nur einen Verweis, wo die DTD zu finden ist: eine DTD-Deklaration mit dem Schlüsselwort SYSTEM.

Weiters weisen wir auf unser Copyright für das neu erfundene XML-Vokabular hin, dazu müssen wir aber erst das Copyright-Symbol neu erfinden, da es in XML nicht definiert ist. Wir erweitern dazu unsere DTD-Deklaration um sogenannte General-Entities und definieren damit das Copyright-Symbol und den Copyright-Inhaber. Das XML-Dokument speichern wir unter `helloworld_5.xml`.

```
<?xml version="1.0" encoding="iso8859-1"?>
<!-- HelloWorld DTD -->
<!ELEMENT HelloWorld (#PCDATA | Greetings)*>
<!ELEMENT Greetings (#PCDATA)>
<!ATTLIST Greetings
   language (deutsch | englisch | persisch) "deutsch" >
<!ENTITY copy "&#169;">
<!ENTITY dtd_autor "BRZ-Kurs">
```

Externe DTD
mit General Entities

```
<?xml version="1.0" encoding="iso8859-1"?>
<!-- HelloWorld_5 Beispiel -->
<!DOCTYPE HelloWorld SYSTEM "helloworld.dtd" [
   <!ENTITY dok_autor "Kursteilnehmer">
]>
<HelloWorld>
Freier Text kann hier sinnvoll sein ...
   <Greetings language="deutsch">Hallo Welt</Greetings>
   <Greetings language="englisch">Hello World</Greetings>
   <Greetings language="persisch">Salam</Greetings>
   &copy; Alle Rechte bei &dtd_autor; und &dok_autor;
</HelloWorld>
```

DTD-Deklaration
mit lokalem General Entity

Entity-Verwendung
(Instanzierung)

Die General-Entities `dtd_autor` und `dok_autor` zeigen, das oft verwendete, gleichlautende Textphrasen sinnvollerweise an zentraler Stelle definiert werden, um die Wartung der Dokumente zu vereinfachen. Entities werden, wie aus HTML bekannt, mit führendem Ampersand und abschließenden Strichpunkt im eigentlichen Text verwendet, das copy-Entity muß also zum Beispiel mit `&copy;` notiert werden.

Schritt 6: XSL-Stylesheet für die Anzeige des Dokuments

Für die Anzeige unseres XML-Dokumentes wählen wir HTML. Unser XSL-Stylesheet soll also das XML-Dokument in eine HTML-Seite transformieren. Dazu schreiben wir das Grundgerüst eines Stylesheets (`helloworld.xsl`) und eine Stylesheet-Deklaration in das XML-Dokument und testen das Ergebnis mit dem Browser.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!-- Root-Template -->
<xsl:template match="/">
   <html><body>
      <h1 align="center">Greetings</h1>
      <u>Push-Methode:</u><br />
      <xsl:apply-templates/>
      <br /><br />
   </body></html>
</xsl:template>

<!-- Weitere-Templates -->

</xsl:stylesheet>
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- HelloWorld_5 Beispiel -->
<!DOCTYPE HelloWorld SYSTEM "helloworld.dtd" [
   <!ENTITY dok_autor "Kursteilnehmer">
]>
<?xml-stylesheet type="text/xsl" href="helloworld.xsl"?>
<HelloWorld> . . . </HelloWorld>
```

Das Ergebnis ist eine unformatierte Ausgabe des Dokumentinhaltes. Wir ergänzen nun das Stylesheet um ein Greetings-Template, das unsere Grüße formatiert.

```
<!-- Weitere-Templates -->
<!-- Greetings-Template -->
<xsl:template match="Greetings">
   <div><xsl:value-of select="@language"/>:   <b><xsl:value-of select="."
/></b></div>
</xsl:template>
```

Wir erhalten nun die Grüße mit zugehöriger Sprachangabe (language-Attribut) angezeigt, und zwar in der Reihenfolge, wie sie im XML-Quelldokument gefunden werden. Ein Ergebnis der sogenannten Push-Methode, die wir im Root-Template angewendet haben.

Zum Vergleich verwenden wir nun die Pull-Methode, um gezielt auf den Dokumentinhalt zugreifen zu können. Dazu verwenden wir in der `apply-templates`-Anweisung einen XPath-Ausdruck mit einer Filter-Anweisung (Prediction) für das `language`-Attribut des `Greetings`-Element.

```
      <h1 align="center" style="color:#FF0000">Greetings</h1>
      <u>Push-Methode:</u><br />
      <xsl:apply-templates/>
      <br /><br />
      <u>Pull-Methode:</u><br />
      <xsl:apply-templates select="//Greetings[@language='persisch']"/>
      <xsl:apply-templates select="//Greetings[@language='deutsch']"/>
      <xsl:apply-templates select="//Greetings[@language='englisch']"/>
```

Eine interessante Spielart ist ein XPath-Ausdruck mit einem Suchmuster für das language-Attribut, das im gezeigten Beispiel englisch und persisch erkennt.

```
   <u>Pull-Methode mit Suchmuster "isch":</u><br />
   <xsl:apply-templates select="//Greetings[contains(@language,'isch')]"/>
```

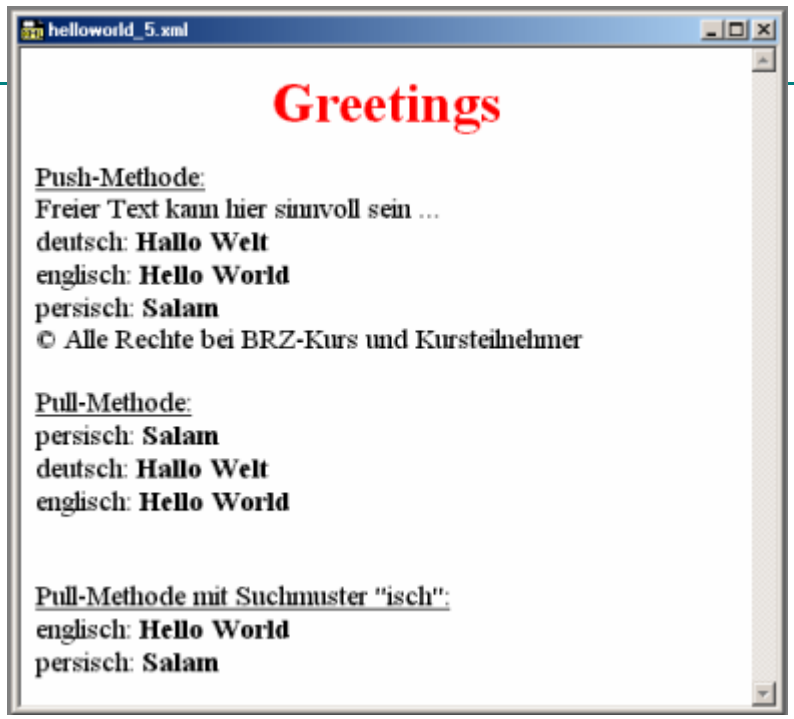
Das komplette Stylesheet und das Ergebnis der Browser-Anzeige sind nachfolgend dargestellt.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!-- Root-Template -->
<xsl:template match="/">
   <html>
   <body>
      <h1 align="center" style="color:#FF0000">Greetings</h1>
      <u>Push-Methode:</u><br />
      <xsl:apply-templates/>
      <br /><br />
      <u>Pull-Methode:</u><br />
      <xsl:apply-templates select="//Greetings[@language='persisch']"/>
      <xsl:apply-templates select="//Greetings[@language='deutsch']"/>
      <xsl:apply-templates select="//Greetings[@language='englisch']"/>
      <br /><br />
      <u>Pull-Methode mit Suchmuster "isch":</u><br />
      <xsl:apply-templates select="//Greetings[contains(@language, 'isch')]"/>
   </body>
   </html>
</xsl:template>

<xsl:template match="Greetings">
   <div><xsl:value-of    select="@language"/>:       <b><xsl:value-of    select="."
/></b></div>
</xsl:template>

</xsl:stylesheet>
```



Beispiel 1

Intelligentes Datendokument

Workshop Vorbereitung zu Beispiel 1: Intelligentes Datendokument

Werkzeuge:
MS Internet-Explorer 5.5 (SP2) oder 6
Einfacher Editor (Wordpad ...)
XMLSpy-Editor (www.xmlspy.com)

XML-Sprachfamilie:
CDATA-Sektionen für die Einbettung von Fremdkode

```
<![CDATA[ JavaScript in XML ...]]>
```

XSLT-Elements

```
xsl:sort select="xpath-expression"
xsl:if test="xpath-expression"
xsl:for-each select="xpath-expression"
xsl:param name="Name"
xsl:variable name="Name" select="xpath-expression"
```

XPath-Expressions

```
axis::path[prediction]
      parent-, ancestor-Achse
      self-Achse
      child-, descendand-Achse
      preceding-sibling, preceding-Achse
      following-sibling, following-Achse
substring(string, von, laenge)
path[prediction][index]
```

XSLT-Functions

```
substring(string, von, laenge)
not(xpath-expression)
count(xpath-expression)
sum(xpath-expression)
substring(string, von, laenge)
a mod b
a and b
```

Zielsprache:
HTML, JavaScript und CSS
DOM-Implementierung in JavaScript

XML-Tag in HTML (Microsoft)
load(XML-Dokument)
XML-Tag in HTML (Microsoft IE)
node = selectSingleNode(xpath-expression)
node.transformNode(xsl-stylesheet)
createNode(nodetype, nodename, namespace)
appendChild(node)

Beispiel 1: Intelligentes Datendokument

Aufgabenstellung:

Ein typisches Beispiel aus unserer alltäglichen Praxis, die Auswertung von Daten. Die übliche Lösung dafür, eine Datenbank und eine Anwendung. XML bietet dazu eine interessante Alternative: Das „intelligente Datendokument“, das sich selbst analysiert und auswertet. Nicht die Anwendung liest das Dokument, sondern das Dokument erzeugt sich selbst eine Anwendung.

Zielsetzung:

Entwurf einer „All-in-one Lösung“ für Dokumentanalyse, Benutzerdialog und Auswertung. Die erforderlichen Programme (JavaScript) werden durch Transformation erst erzeugt. Die technisch sehr anspruchsvolle Lösung zeigt unter anderem Strategien, wie Kommunikation und Parameterübergabe zwischen verschiedenen Sprachwelten (Javascript-XSLT) lösbar sind.

Schritt 1: Konsolidierung der XML-Rohdaten.

Die Rohdaten werden als Log-Dateien von einem Webservice produziert und in einem XML-Format täglich abgespeichert. Jedes Log-Ereignis (Entry) wird dabei mit „Append“ an das Ende der wachsenden XML-Datei gestellt.

```
<entry>
   <session>soap_60</session>
   <request>Abfrage-73a</request>
   <time>2002-09-04-20.41.34.545000</time>
   <provider>TMVIMD1</provider>
   <abfrager>Schneider</abfrager>
   <grund>Verlassenschaft</grund>
   <zeilen>15</zeilen>
   <zeichen>1481</zeichen>
</entry>
```

Das Ergebnis ist ein Kette von XML-Fragmenten, die erst zu einem wellformed XML-Dokument ergänzt werden müssen. Das geht recht einfach mit einem DOS-Skript, das alle Logdatein in einem Verzeichnis mit einem definierten XML-Dokument-Beginn und einem Dokument-Ende verbindet und unter dem Namen `ausw.xml` speichert:

```
rem DOS-Script mit Multiple COPY
COPY beginn.log+abf*.log ausw.xml
TYPE end.log >> ausw.xml
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- BEGINN.LOG -->
<?xml-stylesheet href="ausw.xsl" type="text/xsl" ?>
<auswertung>
```

Multiple COPY

```
<entry>
   <session>soap_60</session>
   <request>Abfrage-73a</request>
   <time>2002-09-04-20.41.34.545000</time>
   <provider>TMVIMD1</provider>
   <abfrager>Schneider</abfrager>
   <grund>Verlassenschaft</grund>
   <zeilen>15</zeilen>
   <zeichen>1481</zeichen>
</entry>
<entry> ... </entry>
```

```
<!-- END.LOG -->
</auswertung>
```

Schritt 2: Konzept der „All-in-one“-Lösung

Das konsolidierte XML-Datendokument ist nun für XML-Werkzeuge verwendbar und kann daher mit einem XSLT-Stylesheet bearbeitet werden. Die Stylesheet-Deklaration steht bereits vorsorglich im erzeugten XML-Dokument, das entsprechende Stylesheet `ausw.xsl` entwerfen wir nun. Das Konzept unserer „All-in-one“-Lösung soll folgende Funktionen abdecken:

- **Dokumentanalyse (1.Transformation)**
  Suche nach möglichen Auswertungskriterien im Quelldokument
  Erzeugung eines Benutzerdialogs mit wählbaren Auswertungsoptionen
- **Dokumentauswertung (2. Transformation)**
  Auswertung des Quelldokumentes nach gewählten Kriterien
  Anzeige des Ergebnisses

Wir erzeugen aus dem XML-Datendokument mittels XSLT-Transformation eine HTML-Seite mit zwei Tabellenzellen. Die linke Tabellenzelle ist für den Benutzerdialog vorgesehen (1. Transformation), die rechte Tabellenzelle, bzw. das eingebettete <p>-Element, für das spätere Auswerte-Ergebnis (2. Transformation).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet                                          version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!-- ========== Root template - start processing here ======== -->
   <xsl:template match="/">
   <HTML><BODY>
      <table width="640" border="0" cellpadding="10">
         <tr>
            <td id="menu" valign="top" width="160" bgcolor="#c0c0c0">
               <b>MENÜ</b><br/>
            </td>
            <td width="*" valign="top">
               <b>AUSWERTUNG</b><br/>
               <p id="content">Wählen Sie eine Auswertung</p>
            </td>
         </tr>
      </table>
   </BODY></HTML>
   </xsl:template>

<!-- ========== Auswertungs-Templates ======== -->

</xsl:stylesheet>
```

Schritt 3: Prinzipielle Dokument-Auswertung

Die Dokumentauswertung selbst ist kein besonderes Problem. Wir entwerfen ein exemplarisches Template für die Auswertung. Für einen ersten Test wird dieses Template mit fixen Parametern aufgerufen.

```
<b>AUSWERTUNG</b><br />
<p id="content">Wählen Sie eine Auswertung </p>
<xsl:apply-templates select="(//entry//provider[.='DATAKOMP'])[1]"/>
```

```
<!-- ========== Auswertungs-Templates ======== -->
<xsl:template match="provider">
<xsl:param name="arg" select="."/>
<b>Test Provider mit Fixparameter (<xsl:value-of select="$arg"/>)</b><br />
   Logs: <xsl:value-of select="count(//provider[.=$arg])"/>  
   Zeilen: <xsl:value-of select="sum(//zeilen[../provider=$arg])"/>  
   Zeichen: <xsl:value-of select="sum(//zeichen[../provider=$arg])"/>
</xsl:template>
```

Das Template erkennt durch Abfrage der gefundenen Dokumentstelle, für welches Kriterium es aufgerufen wurde, in unserem Beispiel „Provider-DATAKOMP“. Mit diesem Parameter wird nun eine Auswertung des gesamten Dokumentes durchgeführt, zB Anzahl der Logs und Summe der abgefragten Zeilen und Zeichen für diesen Provider.

Schritt 4: Suche nach Auswertungskriterien

Wir entwerfen nun den Analyse-Teil des Stylesheets und erwarten als Ergebnis eine Auswahl von Optionen, nach denen der Benutzer das Datendokument auswerten könnte. Im Wesentlichen sind das der Provider, der Grund der Abfrage, und der Abfrager-Name. Wir bereiten daher Eingabe- und Optionsfelder im Menü des Zieldokumentes vor und suchen nach Optionen zu diesen Kriterien im Quelldokument:

```
<b>MENÜ</b><br />
<form name="f1menu"  action="nop" method="get">
   Zeit von<br />
   <input name="von" value="2002-01-01"/><br />
   Zeit bis<br />
   <input name="bis" value="Alles"/><br /><br />
   Provider<br />
   <select name="provider">
      <option value="0" selected="y">Keine Auswertung</option>
      <xsl:for-each select="//entry/provider">
         <option value="{.}"><xsl:value-of select="."/></option>
      </xsl:for-each>
   </select>
    usw
```

Wir erhalten als Ergebnis eine unsortierte Optionsliste mit einer Vielzahl von gleichnamigen Optionen. Um dieses Problem zu lösen, benötigen wir ein tiefers Verständnis der XSLT-Mechanismen. Wir diskutieren es an nachfolgendem Beispiel:

```
Provider<br />
   <select name="get_provider">
   <option value="0" selected="y">Keine Auswertung</option>
   <xsl:for-each select="//entry/provider">
      <xsl:sort select="."/>
      <xsl:if test="node()[not(.=preceding::entry/provider)]">
         <option value="{.}"><xsl:value-of select="."/></option>
      </xsl:if>
   </xsl:for-each>
   </select>
```

Die `xsl:for-each`-Anweisung liefert als Ergebnis eine Liste, die automatisch abgearbeitet wird (Iterator-Funktion). Daher ist das Ergebnis mit der Anweisung `xsl:sort` sortierbar. Das Sortierkriterium ist in unserem Fall identisch mit dem `select`-Kriterium der `for-each`-Anweisung, was durch den Punkt-Operator im `select`-Attribute der `sort`-Anweisung festgelegt wird.

Schwieriger ist das Vermeiden von Mehrfach-Eintragungen in den Optionslisten. Dazu müssen wir jedes gefundene Kriterium mit allen vorangegangenen vergleichen, um gleichnamige auszusortieren. Dazu verwenden wir eine `xsl:if`-Abfrage, mit der wir den jeweils aktuellen Provider-Wert mit allen vorangegangenen Werten (`preceding`-Achse des Dokumentes) vergleichen. Dieser überaus elegante Ausdruck offenbart seine Mächtigkeit erst auf den zweiten Blick: Vom Ist-Operator der Vergleichsoperation wird nämlich ein Einzelwert erwartet und nicht eine Ergebnisliste, die Iterator-Funktion der preceding-Ergebnisliste sorgt aber automatisch für die Bereitstellung dieser Einzelwerte.

Diese Lösung wenden wir auch für die Kriterien „Abfrager“ und „Grund“ an. Damit erhalten wir ein vollständiges Auswahlmenü für unsere Auswertungs-Kriterien.

Schritt 5: Benutzerdialog

Dem Benutzer stehen nun auswählbare Kriterien in Menüform für eine Auswertung des Dokumentes zur Verfügung. Der Benutzerdialog wird in JavaScript abgewickelt, die eigentliche Auswertung des Dokumentes soll aber mit XSLT erfolgen. Dazu sind einige Vorbereitungen erforderlich.

- Das XML-Quelldokument und das XSL-Stylesheet muß für JavaScript ansprechbar sein, um eine neuerliche Transformation des Dokumentes zu ermöglichen.
- Von JavaScript müssen die ausgewählten Kriterien an XSLT übergeben werden. Nachdem diese Programme nicht direkt kommunizieren, ist dies nur auf Umwegen möglich.

Wir erweitern daher das Stylesheet um die erforderlichen Funktionen:

```
<!-- ========== Root template - start processing here ======== -->
<xsl:template match="/">
<HTML><HEAD>
   <SCRIPT LANGUAGE="JavaScript">
   <![CDATA[
      function auswertung() { alert(dok.xml); }
   ]]>
   </SCRIPT>
</HEAD>
<BODY>
   <xml id="dok" src="ausw.xml"/>
   <xml id="xsl" src="ausw.xsl"/>
   <table width="640" border="0" cellpadding="10">
      <tr>
         <td id="menu" valign="top" width="160" bgcolor="#c0c0c0">
            <b>MENÜ</b><br />
            <form name="menu_1"  action="nop" method="get">
               Auswahlmenü wie beschrieben ...
            </form>
            <input type="button" value="Auswertung"
                   onClick="javascript:auswertung()" />
         </td>
         <td width="*" valign="top">
            <b>AUSWERTUNG</b><br />
            <p id="content">Wählen Sie eine Auswertung</p>
         </td>
      </tr>
</table>
</BODY>
</HTML>
```

Wir erzeugen eine JavaScript-Funktion `auswertung()`, das ein alert-Fenster öffnet, im HEAD-Teil der HTML-Seite. Dazu verwenden wir die CDATA-Kapselung von XML, um den Stylesheet-Prozessor von diesem „fremden“ Scriptkode fernzuhalten.

Im BODY-Teil erzeugen wir mit den speziellen HTML-Tags <xml> Dokumentobjekte von unserem XML-Dokument und XSL-Stylesheet. Über die eindeutigen id-Attribute `dok` und `xsl` sind diese DOM-Objekte nun für JavaScript greifbar. Damit verlassen wir allerdings den Weg der puristischen XML-Programmierung, da diese Möglichkeit zur Zeit nur von Microsoft unterstützt wird. Um zu zeigen, dass dieser Mechanismus funktioniert, zeigen wir das XML-Dokument mit der DOM-Methode `dok.xml` im alert-Fenster an.

Außerhalb des Formulars sehen wir den bereits erwähnten Auswertung-Button vor, um die JavaScript-Funktion `auswertung()` mit dem `onClick`-Event aufrufen zu können. Diese Funktion soll eine Auswertungs-Transformation des XML-Dokumentes starten, dafür verwenden wir die `transformNode`-Methode des DOM-Objektes.

Schritt 6: transformNode-Methode des DOM-Objektes

Für den Neustart der XSLT-Transformation muß eine Dokument-Node mittels XPath-Expression gewählt werden, im einfachsten Fall der Dokumentbeginn (Root-Node). Der `transformNode`-Methode dieser DOM-Node kann nun ein beliebiges Stylesheet übergeben werden. In unserem Fall das gleiche Stylesheet (ausw.xsl), das wir bereits für die erste Transformation verwendet haben und das als DOM-Objekt mit der id „xsl“ vorbereitet wurde. Zu beachten ist, dass wir uns trotz der XML-ähnlichen Ausdrücke in der JavaScript-Welt befinden.

```
function auswertung() {
   if(dok.XMLDocument == null) {
      alert("Dokument noch nicht bereit!");
      return;
   }
   var dObj = dok.XMLDocument.selectSingleNode("XPath-Expression");
   document.all["content"].innerHTML = dObj.transformNode(xsl.XMLDocument);
}
```

Das Ergebnis der Transformation, ein HTML-Konstrukt, wird mit der JavaScript-Methode `innerHTML` dynamisch in die vorgesehene Tabellenzelle geschrieben. Für die XSLT-Transformation haben wir dazu kurz die JavaScript-Welt verlassen.

Das Ergebnis der Transformation wäre allerdings das gesamte XML-Datendokument, da wir die Root-Node gewählt haben. Wir wollen aber das Datendokument nach bestimmten Kriterien auswerten. Für diese Aufgabe müssen wir noch ein wichtiges Problem zu lösen, nämlich die Parameterübergabe an die XSLT-Transformation. Dazu gibt es prinzipiell zwei Möglichkeiten.

Experiment: Parameterübergabe mit XPath

Eine einfache Methode der Parameterübergabe ist direkt mittels XPath möglich. Dazu werden einfach die gewählten Kriterien als Filter (Prediction) für die entsprechende Node formuliert und dem Stylesheet-Prozessor beim Neustart übergeben. Um zum Beispiel das Datendokument gezielt nach einem Provider namens „DATAKOMP“ auszuwerten, müssen wir dieses Kriterium wie folgt als Übergabeparameter formulieren:

```
var dObj = dok.XMLDocument
              .selectSingleNode("//entry//provider[.='DATAKOMP'])[1]"/>
dObj.transformNode
```

Node | Filter | Index

Im Stylesheet würde damit das entsprechende Template (match=“provider“) aktiviert werden und auf eine zur Filteranweisung passende Dokumentstelle verweisen (Kontext). Diese Dokumentstelle kann nun gezielt nach dem Provider-Namen abgefragt werden.

Um zu verhindern, dass das Template mehrfach aufgerufen wird, enthält der Aufruf-Parameter zusätzlich eine Index-Angabe, damit nur eine Dokumentstelle gefunden wird.

```
<xsl:template match="provider">
<xsl:param name="arg" select="."/>
<b>Test Provider mit Fixparameter (<xsl:value-of select="$arg"/>)</b><br />
   Logs: <xsl:value-of select="count(//provider[.=$arg])"/>  
   Zeilen: <xsl:value-of select="sum(//zeilen[../provider=$arg])"/>  
   Zeichen: <xsl:value-of select="sum(//zeichen[../provider=$arg])"/>
</xsl:template>
```

Dieser Mechanismus versagt allerdings, wenn widersprüchliche Parameter als Filteranweisungen übergeben werden sollen, zB.eine Von-Zeit und eine Bis-Zeit.

Schritt 7: Parameterübergabe mittels DOM-Manipulation

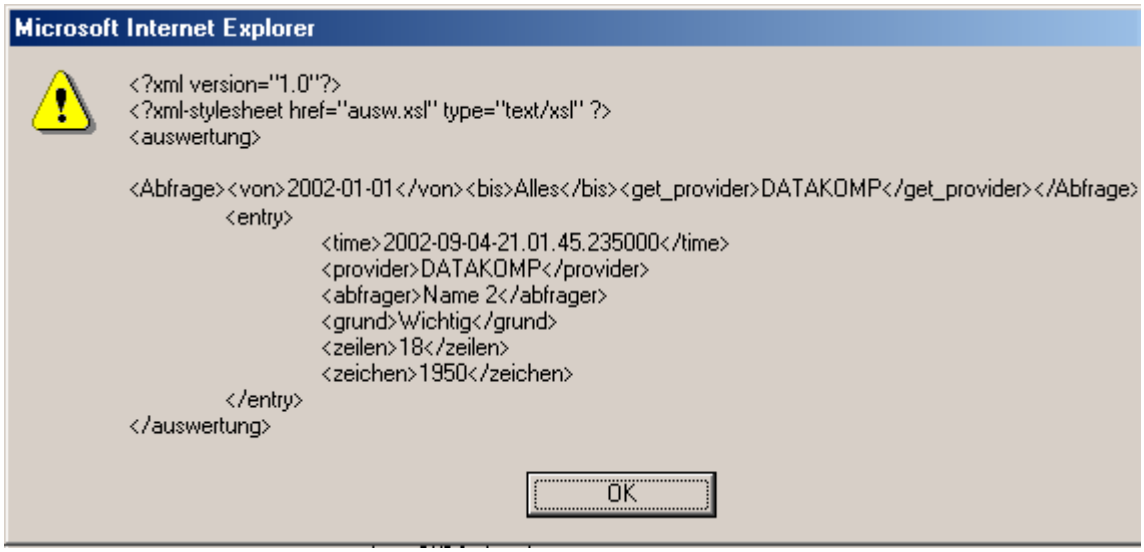
Eine universelle Lösung ist die Parameterübergabe mittels DOM-Manipulation. Die Abfragekriterien werden einfach als neue Elemente zum Datendokument hinzugefügt:

```
<SCRIPT LANGUAGE="JavaScript">
<![CDATA[
   function auswertung() {
      var root = dok.documentElement;
      var newAbfrage = root.insertBefore(dok.createNode(1, "Abfrage", ""),
                                                    root.childNodes.item(0));
      for(i=0; i<document.forms[0].length; ++i) {
         if(document.forms[0].elements[i].value != "0") {
            var wert = document.forms[0].elements[i].value;
            var was = document.forms[0].elements[i].name;
            var kriteria = newAbfrage.appendChild(dok.createNode(1, was, ""));
            kriteria.appendChild(dok.createTextNode(wert));
         }
      }
      alert(dok.xml); //Zeigt manipuliertes DOM an
      var dObj = dok.XMLDocument.selectSingleNode("//Abfrage[0]");
      document.all["content"].innerHTML = dObj.transformNode(xsl.XMLDocument);
   }
]]>
</SCRIPT>
```

Der Mechanismus im Detail:

- Am Beginn des XML-DOM wird ein mit `dok.createNode()` erzeugtes Element (Abfrage) mit der insert-Methode eingefügt.
- In einer for-Schleife werden alle Dialog-Elemente des Menüs (Provider, Abfrager, Grund) abgefragt, ob eine Auswahl getroffen wurde. Wenn ja, wird der Name des Dialog-Elementes (Variable was) verwendet, um ein Child-Element zu Abfrage zu erzeugen. Der Wert des Dialog-Elementes wird als Child-Inhalt verwendet (Variable wert).
- Abschließend wird die Transformation aufgerufen, diesmal mit der XPath-Expression "//Abfrage[0]", also dem Namen des neu eingefügten Dokument-Elementes (zu dem es allerdings noch kein passendes Template gibt).

Das Alert-Fenster im JavaScript zeigt das DOM-Ergebnis:



Bei aufeinanderfolgenden Aufrufen der Transformation wird jeweils ein neues Abfrage-Element mit neuen Parametern erzeugt und an den Beginn des Dokumentes gestellt. Daher die Index-Angabe bei der XPath-Expression "//Abfrage[0]", um das erste Abfrage-Element zu übergeben.

Zur Parameterübergabe fehlt jetzt noch ein entsprechendes Template, dass die Auswahlparameter übernimmt und die Auswertungen durchführt bzw. aufruft.

```
<xsl:template match="Abfrage">
   <xsl:apply-templates select="get_provider"/>
   <xsl:apply-templates select="get_abfrager"/>
   <xsl:apply-templates select="get_grund"/>
</xsl:template>

<xsl:template match="get_provider">
<xsl:param name="arg" select="."/>
<b>Summen nach Provider (<xsl:value-of select="$arg"/>)</b><br />
   Logs: <xsl:value-of select="count(//provider[.=$arg])"/>  
   Zeilensumme: <xsl:value-of select="sum(//zeilen[../provider=$arg])"/>  
   Zeichensumme: <xsl:value-of select="sum(//zeichen[../provider=$arg])"/>
</xsl:template>

<xsl:template match="get_abfrager">
<xsl:param name="arg" select="."/>
<b>Summen nach Abfrager (<xsl:value-of select="$arg"/>)</b><br />
...
</xsl:template>

<xsl:template match="get_grund">
<xsl:param name="arg" select="."/>
<b>Summen nach Grund (<xsl:value-of select="$arg"/>)</b><br />
...
</xsl:template>
```

In der Praxis scheitert die gezeigte Lösung allerdings bei großen Dokumenten an dem unglücklich gewählten Protokollformat der Log-Dateien, da die rekursive Suche nach Auswertekriterien quadratisch von der Dokumentgröße abhängt. Mit einer XML-XML Transformation ist dieses Problem aber im Normalfall behebbar (siehe Beispiel 4: XML-Transformationen).

Beispiel 2

Eingabeformular mit XML

Workshop Vorbereitung zu Beispiel 2: Eingabeformular mit XML

Werkzeuge:
MS Internet-Explorer 5.5 (SP2) oder 6
Einfacher Editor (Wordpad ...)
XMLSpy-Editor (www.xmlspy.com)

XML-Sprachfamilie:
CDATA-Sektionen für die Einbettung von Fremdkode

```
<![CDATA[ JavaScript in XML ...]]>
```

External Document Type Definition (DTD)
- Element-Deklaration und Inhaltsmodell
- Attribut-Deklaration und Typdefinition
- External Entities

XSLT-Elements

```
xsl:attribute
xsl:if test="xpath-expression"
xsl:for-each select="xpath-expression"
```

XPath-Expressions

```
attribute value template {}
```

XSLT-Functions

```
string(nodevalue)
substring(nodevalue, position, laenge)
```

Zielsprache:
HTML, JavaScript und CSS
DOM-Implementierung in JavaScript

```
getAttribute(name, is-case-sensitive)
```

Beispiel 2: Eingabeformular mit XML

Aufgabenstellung:

Dieses praxisnahe Beispiel eines Eingabeformulars für statistische Daten (Scheidungszählblatt) soll zeigen, daß XML auch für traditionelle Anwendungen wie HTML-Formulare sinnvoll einzusetzbar ist. Das XML-Konzept der Trennung von Struktur, Inhalt und Präsentation“ läßt sich zum Beispiel als Entwurfsmuster anwenden, das sich in der objektorientierten Welt unter dem Namen „Model-View-Controller“-Konzept bewährt hat.

Zielsetzung: Formulare nach dem Model-View-Controller Konzept

- Entwurf einer allgemeinen Formular-Beschreibungssprache und Realisierung des MVC-Konzeptes durch strikte Trennung in anwendungsspezifische Details (Model) und Standard-Formularfunktionen (View und Controller).
- Das Model wird eindeutig durch das XML-Dokument repräsentiert. Es enthält in einer für den Anwender verständlichen Markup-Sprache die vollständige formale Definition der Anwendung.
- Die View, also die Darstellung des Formulars, wird eindeutig durch das XSLT-Stylesheet realisiert. Im konkreten Fall die Transformation in ein anzeigbares HTML-Formular.
- Der Controller, also die DHTML-Funktionalität (JavaScript) des Formulars, ist im Sinne des MVC-Konzeptes zu organisieren:
  - Anwendungsspezifische Funktionen im Modell (XML-Dokument)
  - Allgemeine Formularfunktionen in der View (XSLT)

Schritt 1: Entwurf einer Formular-Beschreibungssprache

Die Struktur eines Formulars läßt sich durch eine relativ einfache DTD beschreiben. Im Wesentlichen sind für ein Formular Eingabefelder, Optionslisten und Checkboxes sinnvoll, und natürlich die Möglichkeit, diese sinnvoll zu gruppieren, mit Hilfetexten zu versehen und formale Datenprüfungen vorzusehen.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?xml-stylesheet href="scheidung.xsl" type="text/xsl"?>
<!DOCTYPE Formular [
   <!ELEMENT Formular (Beschreibung, Gruppe+, MetaData?)>
   <!ATTLIST Formular  titel CDATA #IMPLIED
>
   <!ELEMENT Beschreibung (#PCDATA)>
   <!ELEMENT Gruppe (Input)*>
   <!ELEMENT Input (Beschreibung)>
   <!ATTLIST Input
      id ID #REQUIRED
      size CDATA #REQUIRED
      typ (Text | Zahl) "Text"
      label CDATA #IMPLIED
      readonly (y | n | hidden) "n"
      required (y | n) "n"
      default CDATA #IMPLIED
>
   <!ELEMENT MetaData (Aktion?)>
   <!ELEMENT Aktion (#PCDATA)>
]>
```

Die DTD wird direkt in das XML-Dokument eingebettet, das Wurzelelement "Formular" bestimmt den Namen der neuen Dokumentklasse und besitzt folgendes Inhaltsmodell:

- Ein Beschreibung-Element zur Beschreibung des Formularzweckes
- Ein bis viele Gruppe-Element zur Gruppierung von Eingabe-Elementen
- Ein MetaData-Element zum Transport von JavaScript-Kode

Ein XML-Formular (Model) könnte demnach folgendermaßen aussehen:

```
<!-- ======= FORMULAR DOKUMENT =========  -->
<Formular titel="Scheidungszählblatt">
   <Beschreibung>Das Scheidungszählblatt dient zur ...</Beschreibung>
   <Gruppe>
      <Input id="BG" size="3" typ="Zahl" label="BG" required="y">
         <Beschreibung>Wählen Sie Ihr Bezirksgericht aus.</Beschreibung>
      </Input>
      <Input id="Aktenzahl" size="5" label="Aktenzahl" required="y">
         <Beschreibung>Aktenzahl (max. fünfstellig).</Beschreibung>
      </Input>
      <Input id="Jahr" size="4" typ="Zahl" label="Jahr (JJJJ)" required="y">
         <Beschreibung>Erfassen Sie das Anfallsjahr</Beschreibung>
      </Input>
      <Input id="Test" size="4" label="Test" default="Testfeld">
         <Beschreibung>Geben Sie freien Text ein</Beschreibung>
      </Input>
   </Gruppe>
   <Gruppe>
      <Input id="Datum" size="8" typ="Zahl" readonly="y">
         <Beschreibung>Datum wird automatisch gesetzt</Beschreibung>
      </Input>
   </Gruppe>
   <MetaData>
      <Aktion>
<![CDATA[
//Formular initialisieren
function initForm() {
   //Bewilligungsdatum setzen
   today = new Date();
   document.forms[0].Datum.value = today.getYear();
}
]]>
      </Aktion>
   </MetaData>
</Formular>
```

Aktuelles Datum wird automatisch gesetzt

Einfacherweise beschränken wir uns auf Eingabefelder. Andere Formularelemente wie Optionslisten und Checkboxes sind sinngemäß modelliert (siehe Anhang).

Das letzte Eingabefeld in unserem Beispiel hat eine besondere Funktion, es soll das aktuelle Tagesdatum enthalten. Damit lassen sich zwei Besonderheiten demonstrieren:

- Eingabefelder sollen auch passiv (readonly) oder überhaupt versteckt (hidden) im Formular vorsehbar sein.
- Anwendungsspezifische Aktionen sollem mit bestimmten Formularelementen durchführbar sein, zum Beispiel eben das Einfügen des aktuellen Datums in das fünfte Eingabefeld. Das bedeutet aber, dass „maßgeschneiderte“ JavaScript-Aktionen erforderlich sind, die nicht in einer Stylesheet-Bibliothek organisierbar sind. Solche Funktionen sollen mit dem Model transportiert werden (Model-Controller Beziehung).

Dafür haben wir mit dem MetaData-Element ein Vehikel vorgesehen, um solche Aktionen an die XSLT-Transformation übergeben zu können. Unser erster Entwurf eines XSLT-Stylesheets wird daher neben den grundlegenden HTML-Funktionen (statische Anzeige des Formulars) auch schon einige DHTML-Funktionen besitzen.

Für die Anzeige als HTML-Formular transformieren wir mittels XSLT-Stylesheet, für das wir zuerst ein Root-Template mit HTML-Skelett, CSS-Styles und JavaScript-Sektion entwerfen:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="html" indent="yes"/>

<!--                          ROOT TEMPLATE                          -->
<xsl:template match="/">
<html><head>
<style type="text/css">
   Style-Definitionen (siehe Quellkode)
</style>
<SCRIPT LANGUAGE="JavaScript">
<![CDATA[ //DHTML-Funktionen
   //Beschreibungstext anzeigen
   function showInfo(Text) { document.all["info"].innerText=Text; }
   //Formular prüfen
   function fertig() { alert("Formular prüfen"); }
]]>
</SCRIPT>
</head>

<body onLoad="javascript:initForm()">
<!-- EINGABE-FORMULAR -->
<form name="eingabe" action="nop" method="get">
<table border="0" cellspacing="0" cellpadding="2" width="100%">
   <xsl:apply-templates/>
</table>
</form>
<input type="button" value="Formular prüfen" onClick="javascript:fertig()" />
<br /><br />
<!-- INFO-BOX -->
<b style="color:#407090; ">Info</b>
<div id="info"> </div>
<xsl:apply-templates select="//Aktion"/>
</body>
</html>
</xsl:template>

<!--            TEMPLATES           -->

</xsl:stylesheet>
```

Das Formular wird durch die Transformation `xsl:apply-templates` innerhalb eines HTML `form`-Elementes erzeugt. Für die Initialisierung des Formulars verwenden wir die `onload`-Methode im `body`-Tag. Diese Methode ruft die anwendungsspezifische initForm-Funktion auf, die wir im Model eingebettet haben, und die eigentlich erst weiter unten im body-Teil „deklariert“ ist (Transformation mit `xsl:apply-templates select=“//Aktion“`). Das zugehörige Template für diese Transformation:

```
<!--            TEMPLATES           -->
...
<xsl:template match="//Aktion">
<SCRIPT LANGUAGE="JavaScript">
   showInfo('<xsl:value-of select="//Formular/Beschreibung"/>');
   <xsl:apply-templates/>
</SCRIPT>
</xsl:template>
```

Diese Reihenfolge „Aufruf vor Deklaration“ könnte in einem echten HTML-Formular zu Timing-Problemen führen, durch die XSLT-Transformation wird aber ein vollständiges DOM-Objekt an den Browser übergeben.

Für die einfache Anzeige des Formulars sind folgende Templates vorgesehen:

```
<!--            TEMPLATES           -->
<xsl:template match="Gruppe">
<tr>
   <xsl:apply-templates/>
</tr>
</xsl:template>

<xsl:template match="Input">
<td valign="top">
<div id="label"><xsl:value-of select="@label"/></div>
<input size="28" style="font-family:Arial; font-size:7pt"
   id="{@id}"
   value="{@default}"
/>
</td>
</xsl:template>

<!-- FILTER-TEMPLATES -->
<xsl:template match="Formular/Beschreibung"/>
<xsl:template match="MetaData"/>
```

Das Gruppe-Template erzeugt einfach jeweils eine Tabellenreihe, in der Tabellenzellen für jedes in der Gruppe befindliche Formularelement eingefügt wird (Transformation durch `xsl:apply-templates`). Das Input-Template wird dadurch aktiviert und führt die genannte Produktion durch, indem es ein HTML `input`-Element erzeugt und in eine Tabellenzelle einbettet. Die Bedeutung der Attribute wird im nächsten Schritt behandelt.

Zwei Filter-Templates verhindern, dass die Formularbeschreibung und der Inhalt von MetaData an unerwünschter Stelle in das HTML-Ergebnis transformiert werden. Damit haben wir die Grundlage für unsere eigentliche Aufgabenstellung, die Realisierung des MVC-Konzeptes, geschaffen.

Schritt 2: Model-View Beziehung am Beispiel eines Eingabefeldes

Als nächsten Schritt realisieren wir die vollständige View-Funktionalität des Formulars, indem wir alle dafür bestimmten Attribute des Models auswerten. Das ist eine HTML-Produktion, wir werden daher nur die für uns wesentlichen Details beachten:

```
<xsl:template match="Input">
<td valign="top">
<div id="label"><xsl:value-of select="@label"/></div>
<input  size="28"  style="font-family:'Verdana,  Arial';  font-size:7pt;
color:#604040;"
   id="{@id}"
   value="{@default}"
   dSize="{@size}"
   dTyp="{@typ}"
   onmouseover="javascript:showInfo(' {Beschreibung} ')"
   onmouseout="javascript:showInfo(' {//Formular/Beschreibung} ')"
>
<xsl:if test="@readonly[.='y']">
   <xsl:attribute name="readonly">true</xsl:attribute>
</xsl:if>
<xsl:if test="@readonly[.='hidden']">
   <xsl:attribute name="type">hidden</xsl:attribute>
</xsl:if>
<xsl:if test="@required[.='y']">
   <xsl:attribute name="pflicht">true</xsl:attribute>
</xsl:if>
</input>  
</td>
</xsl:template>
```

Pflichtattribute werden direkt zugeordnet

Hover-Help

Optionale Attribute werden durch if-Abfrage erkannt und zugeordnet

Die Attribute `id`, `value`, `dSize` und `dTyp` sind in unserem Modell als verpflichtende Attribute deklariert, daher können sie elegant mit einem sogenannten Attribute-Value Template in geschwungenen Klammern zugewiesen werden. Die verbleibenden optionalen Attribute bedingen eine if-Abfrage und müssen daher „umständlich“mit dem XSLT-Element `xsl:attribute` erzeugt werden.

Der Beschreibungstext wird mit dem onmouseover-Ereignis als Hover-Help erzeugt, dh in unserem Fall einfach in der Info-Box angezeigt. Wenn der Mauszeiger das Eingabefeld verlässt, wird wieder der Formulartext angezeigt (onmouseout- Ereignis).

Das eigentlich interessante Detail: Die Attribute `dSize`, `dTyp` und `pflicht` haben in HTML keinerlei Funktion. Wir erzeugen sie trotzdem, das Document Object Model DOM erlaubt es, und wir werden sie später für die Datenprüfung brauchen!

Schritt 3: View-Controller Beziehung (JavaScript im Stylesheet)

Die View-Controller Beziehung umfaßt alle allgemeinen Funktionen eines Formulars, wie zum Beispiel „Formulardaten senden“, „Formularfelder löschen“ oder formale Prüfungen (Pflichtfelder usw.). Diese Funktionen lassen sich direkt mit XSLT als „hardkodiertes“ HTML produzieren

```
//Formular prüfen
function fertig() {
   var s="";
   errorFlag = false;
   for(i=0; i<document.forms[0].length; ++i) {
      f = document.forms[0].elements[i];
      data = trimm(f.value);
      if((f.getAttribute("pflicht", "true")) == "true") {
         if(data.length == 0) {
            f.style.backgroundColor = "salmon";
         } else {
            f.style.backgroundColor = "white"; }
      }
      if(document.forms[0].elements[i].style.backgroundColor == "salmon")
         errorFlag = true;
      s = s + data;
   }
   if(errorFlag == false) { alert("Formulardaten :" + s); }
      else { alert("Markierte Felder sind fehlerhaft/nicht ausgefüllt!"); }
}
```

Als Ergebnis haben wir nun ein HTML-Formular mit DHTML-Funktionen. Das XSLT-Stylesheet erlaubt uns, Formulare künftig mit einem relativ einfachen Vokabular zu erzeugen. Dies ist im Sinne der Model-View Trennung - das Modell soll sich hauptsächlich mit fachlichen Aspekten beschäftigen, und nicht mit HTML- Kode oder JavaScript. Das Ergebnis ist allerdings bei Anwendung von XML selbstverständlich, da XML strikt zwischen Dokument und Präsentation (Model und View) trennt.

Mit den nächsten Schritten stellen wir allerdings diese puristische Auffassung in Frage. Um tatsächlich eine saubere Trennung zwischen Model und View zu erzielen, braucht der Formular-Autor ein Werkzeug, um anwendungsspezifische (logische) Funktionen im Model zu formulieren. Das Stylesheet selbst soll ja nur allgemeine (formale) Funktionen erfüllen.

Dieses Werkzeug kann nur JavaScript sein, da es ja zur Laufzeit des HTML-Formulars benötigt wird. Allerdings JavaScript in seiner einfachsten Form - einfache if-Abfragen und Aufrufe von Bibliotheksfunktionen – also zumutbare Programmierkenntnisse für einen Formular-Autor.

Schritt 4: Model-Controller Beziehung (JavaScript in XML)

Eine typische Model-Controller Beziehung ist die Datenprüfung, zum Beispiel korrekte Datums- und Aktenzahl-Formate, aber auch Abhängigkeiten wie zum Beispiel Heiratsdatum größer als Geburtsdatum. Dieses spezielle Wissen über die einzelnen Formularelemente ist nur im Model vorhanden, daher sollen die zugehörigen Prüfungen auch im Model als Funktionskode notiert werden.

Das folgende Beispiel zeigt die Model-Controller Beziehung für eine unmittelbare Datenprüfung bei Verlassen eines Eingabefeldes durch das JavaScript `onBlur`-Ereignis. Das XSLT-Stylesheet erzeugt für das HTML-Eingabefeld ein onBlur-Attribut mit dem Methodenaufruf `validate`. Die zugehörige Methode ist aber nicht in der JavaScript-Bibliothek des Stylesheets kodiert, sondern im XML-Quelldokument (Formular-Model).

XSLT-Stylesheet

```
<xsl:template match="Input">
<td valign="top">
<div id="label"><xsl:value-of select="@label"/></div>
<input  size="28"  style="font-family:'Verdana,  Arial';  font-size:7pt;
color:#604040;"
   id="{@id}"
   value="{@default}"
   dSize="{@size}"
   dTyp="{@typ}"
   onmouseover="javascript:showInfo('{Beschreibung} ')"
   onmouseout="javascript:showInfo('{//Formular/Beschreibung} ')"
   onblur="javascript:validate('{@id}')"
>
```

XML-Dokument

```
   <MetaData>
      <Aktion>
<![CDATA[
//Formular initialisieren
function initForm() {
   //Bewilligungsdatum setzen
   today = new Date();
   document.forms[0].Datum.value = today.getYear();
}

//Prüfen der Datenfelder nach Eingabe
function validate(field) {
   if(field == "BG" || field == "Aktenzahl") alert("Prüfe Feld " + field);
}
]]>
      </Aktion>
   </MetaData>
```

Bei der XSLT-Transformation wird der im Stylesheet notierte Javascript-Kode für das HTML-Eingabefeld direkt erzeugt, der JavaScript-Kode der `validate`-Funktion wird über ein einfaches Template aus dem XML-Dokument geholt und damit in das HTML-Formular exportiert.

```
<xsl:template match="//Aktion">
   <SCRIPT LANGUAGE="JavaScript">
      <xsl:apply-templates/>
   </SCRIPT>
</xsl:template>
```

Im Verzeichnis „schritt5“ ist das komplette Beispiel des Formulars zu finden, inklusive CGIs für den Mailversand.

Beispiel 3

Handbuch Konzept mit XML

Workshop Vorbereitung zu Beispiel 3: Handbuch-Konzept mit XML

Werkzeuge:
MS Internet-Explorer 5.5 (SP2) oder 6
Einfacher Editor (Wordpad ...)
XMLSpy-Editor (www.xmlspy.com)

XML-Sprachfamilie:
External Document Type Definition (DTD)
- Kapiteln als External Entities

XSLT-Elements
- `xsl:attribute`
- `xsl:if test="xpath-expression"`
- `xsl:for-each select="xpath-expression"`
- Konfliktlösung mit Template-Rules
- Konfliktlösung mit Priority-Attribut

XPath-Expressions
- `attribute value template {}`

XSLT-Functions
- `context()`
- `document(xlink-expression)`
- `translate(string, string)`

Zielsprache:
HTML, JavaScript und CSS

Beispiel 3: Handbuch-Konzept mit XML

Aufgabenstellung:

Benutzerhandbücher sind entweder didaktisch aufbereitete Tutorials oder systematische Referenzen. Diese unterschiedlichen Zielsetzungen lassen sich durch traditionelle Buchformen nur schwer vereinbaren. XML zeigt hier seine besonderen Stärken durch gezielten Strukturierung der Information und durch strikte Trennung zwischen Inhalt und Präsentation.

Zielsetzung:

- Entwurf einer DTD für Handbücher
- Erstellung unterschiedlicher Handbuch-Kapitel und eines Glossars
- Entwurf eines XSLT-Stylesheet für Tutorial/Referenzhandbücher
- Organisation der Kapitel zu einem Handbuch

Schritt 1: DTD, Kapitel- und Glossardokumente

Bei der Dokumenttyp-Definition DTD beschränken wir uns auf elementare Strukturen:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!ELEMENT Kapitel (Titel, (Kapitel | Inhalt)+)>
   <!ATTLIST Kapitel selectable (y | n) "n" >
<!ENTITY % txt "b | i | u | br | sp | Kode | Begriff | Link">
<!ELEMENT Titel (#PCDATA)>
<!ELEMENT Inhalt (#PCDATA | %txt; | Bild)*>
<!ELEMENT Bild EMPTY>
   <!ATTLIST Bild src CDATA #REQUIRED >
<!ELEMENT Link (#PCDATA)>
   <!ATTLIST Link href CDATA #REQUIRED >
<!ELEMENT Begriff (#PCDATA)>
   <!ATTLIST Begriff href CDATA #REQUIRED >
<!ELEMENT b (#PCDATA | %txt; )*>
<!ELEMENT i (#PCDATA | %txt; )*>
<!ELEMENT u (#PCDATA | %txt; )*>
<!ELEMENT br EMPTY>
<!ELEMENT sp EMPTY>
<!ELEMENT Kode (#PCDATA)>
```

- Das Dokumentmodell beschreibt den Aufbau eines Kapitels mit Titel, Inhalt und/oder beliebig tief verschachtelte Sub-Kapitel.
- Für Textformatierungen sind die bekannten HTML-Tags b (fett), i (kursiv), u (unterstrichen), br (Zeilenumbruch) und ein Nonbreaking-Space sp vorgesehen. Weiters sind für Text die Möglichkeiten vorgesehen, Begriffe (Glossar-Verweise), Links und einfaches Layout (Kode, entsprechend dem HTML-Tag pre) zu gestalten.
- Als Inhalt sind neben Text auch Bilder möglich
- Ein selectable-Attribut erlaubt es, Kapitel in einer besonders übersichtlichen und navigierbaren Form darzustellen (Details folgen)
- Es wäre sinnvoll, jedes Kapitel mit einem id-Attribut zu versehen, um die Inhalte eindeutig identifizierbar zu machen. Für unser einfaches Beispiel verzichten wir aber auf diese Möglichkeit.

Damit sind wir in der Lage, einfache Handbuch-Kapitel zu entwerfen, wie im nachfolgenden Beispiel dargestellt.

Ein einfaches Handbuch-Kapitel mit verschiedenen Gestaltungselementen sowie nachfolgend ein Glossar für Begriffserklärungen:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE Kapitel SYSTEM "kapitel.dtd">
<Kapitel>
   <Titel>Kapiteltitel 1</Titel>
   <Inhalt>
Freier Text gemischt mit <Begriff href="a1">Begriffen</Begriff>
und mit formatierten Texten wie zum Beispiel <b>Fettdruck</b>,
<i>Kursiven Texten</i> oder auch<sp/><sp/> Bildern:
      <br/><br/>
      <Bild src="image/bild1.gif"/>.
      <br/><br/>
      <Kode>
<![CDATA[
      Das ist ein
         Text eingebettet
      in eine CDATA-Sektion.
]]>
      </Kode>
   </Inhalt>
</Kapitel>
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<Glossar>
   <Begriff id="a1">Begriffserklärung 1</Begriff>
   <Begriff id="a2">Begriffserklärung 2</Begriff>
   <Begriff id="a3">Begriffserklärung 3</Begriff>
   <Begriff id="a4">Begriffserklärung 4</Begriff>
   <Begriff id="a5">Begriffserklärung 5</Begriff>
</Glossar>
```

Schritt 2: XSLT-Stylesheet

Das Root-Template des XSLT-Stylesheets besteht im wesentlichen aus einem HTML-Skelett und CSS-Styles für die verschiedenen Kapiteltitel-Hierarchien. Der HTML-body enthält den Aufruf zur eigentlichen Dokumenttransformation <xsl:apply-templates>

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

<xsl:template match="/">
<html>
<head>
   <STYLE TYPE="text/css">
      body { font-family:Arial; font-size:11pt; }
      td { font-family:Arial; font-size:11pt; }
      h1 { font-family:Arial; font-size:24pt; }
      h2 { font-family:Arial; font-size:20pt; color:#909090 }
   usw
   </STYLE>
</head>

<body id="DOKU">
   <xsl:apply-templates/>
</body>
</html>
</xsl:template>
```

Einige exemplarische Templates für die Textformatierungen bold, italic, underline und break (Zeilenumbruch):

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

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

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

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

Konfliktlösung mit Template-Rules

Für die Darstellung der verschiedenen Kapiteltitel-Hierarchien ist eine Konfliktlösung erforderlich, da das Pattern der rekursiven Kapiteltiteln (Kapitel/Titel) im Dokument nicht eindeutig ist. In unserem Fall erfolgt die Konfliktlösung durch die exakte Pfadangabe zu den einzelnen Kapitel-Hierarchien, wie im nachfolgenden XSLT-Kodeauschnitt gezeigt:

```
<xsl:template match="Buch/Titel">
   <h1><xsl:value-of select="."/></h1>
</xsl:template>

<xsl:template match="/Kapitel/Titel">
   <h2><xsl:value-of select="."/></h2>
</xsl:template>

<xsl:template match="/Kapitel/Kapitel/Titel">
   <h3><xsl:value-of select="."/></h3>
</xsl:template>

<xsl:template match="/Kapitel/Kapitel/Kapitel/Titel">
   <h4><xsl:value-of select="."/></h4>
</xsl:template>

<xsl:template match="Titel">
   <h5><xsl:value-of select="."/></h5>
</xsl:template>
```

Damit lassen sich die einzelnen Kapitel-Hierarchien unterscheiden, da immer das bestpassendste Pattern bevorzugt wird. Eine gezielte Priorität bei gleichwertigen Pattern läßt sich durch ein Priority-Attribut in der Template-Rule erzwingen.

Schritt 3: Tutorial-Aspekt des Handbuches

Der Tutorial-Aspekt des Handbuches läßt sich zb. durch ein Glossar mit zusätzlichen Informationen abdecken, das wie eingangs gezeigt, ganz einfach organisiert sein kann. Zur Anzeige dieser Begriffserklärungen bietet sich im elektronischen Handbuch ein Hover-Help an, daß beim Überfahren des Begriffs-Wortes automatisch aktiviert wird. Dazu erweitern wir die CSS-Styles um ein positionierbares Textelement (ToolTip) und „verstecken“ es mit dem `visibility`-Attribut. Die entsprechende HTML-Deklaration, ein -Element sehen wir unmittelbar nach dem body-Tag vor. Die erforderliche JavaScript-Funktion `zeigeToolTip` ist leider nicht trivial:.

```
<head>
   <STYLE TYPE="text/css">
   ...
      #ToolTip { position:absolute; visibility:hidden; z-index:8;
         background-color:#E8EBF4; padding:10; font-size:11pt; }
   </STYLE>
   <SCRIPT LANGUAGE="JavaScript">
   <![CDATA[
   function zeigeToolTip(text) {
      if(text == "NONE") {
         document.all["ToolTip"].style.visibility = "hidden";
      } else {
         document.all["ToolTip"].innerHTML=text;
         document.all["ToolTip"].style.top =  window.event.clientY + 20
                              + document.all.DOKU.scrollTop;
         document.all["ToolTip"].style.left = window.event.clientX -100;
         document.all["ToolTip"].style.visibility = "visible";
      }
   }
   ]]>
   </SCRIPT>
</head>
<body>
<div id =“ToolTip“> None </div>
```

Der eigentlich interessante Teil, das Begriff-Template mit direktem Zugriff auf das Glossar-Dokument über die XSLT-Funktion `document()`:

```
<xsl:template match="Begriff">
   <xsl:param name="tip0"
            select="document('glossar.xml')//Begriff[@id=current() /@href]"/>
   <b style="color:#303070; cursor:Help"
      onMouseout="javascript:zeigeToolTip( 'NONE' )"
      onMouseover="javascript:zeigeToolTip('{$tip}')">
      <xsl:apply-templates/>
   </b>
</xsl:template>
```

Diese dynamisch erzeugten Begriffserklärungen (sogenannte Tooltips) sind allerdings nur für einfache Hilfstexte geeignet. Probleme bereitet bereits ein Zeilenumbruch im Glossartext bei der Übergabe an JavaScript, daher müssen diese unzulässigen Whitespace-Zeichen aus dem Übergabestring ($tip) entfernt werden. Dazu ist die XSLT-Funktion `translate()` ideal:

```
<xsl:param name="tip0"
   select="document('glossar.xml')//Begriff[@id=current() /@href]"/>
<xsl:param name="tip" select="translate($tip0, '&#x0a;&#x0d;', ' ')"/>
```

Schritt 4: Referenz-Aspekte des Handbuches

Den Referenz-Charakter des elektronischen Handbuches wollen wir durch besondere Kapitelgestaltung erzielen, indem wir Unterkapitel mit systematischen Beschreibungen nicht direkt anzeigen, sondern nur eine Themenübersicht, die mit Mausklick die entsprechenden Kapitel einblendet. Dieses Modell ist besonders für Dialogbeschreibungen interessant, da der abgebildete Dialog immer sichtbar bleibt (kein Scrollen zum Beschreibungstext). Das erforderliche JavaScript ist in diesem Fall sehr einfach:

```
function zeige(box, name) {
   document.all[box].innerHTML=document.all[name].innerHTML;
}
```

Dafür ist das entsprechende Template für diese besonderen Kapitel etwas aufwendiger zu gestalten, die wichtigen Details sind im nachfolgenden Kode hervorgehoben:

```
<xsl:template match="Kapitel[@selectable='y']">
   <xsl:param name="kap" select="position()"/>
   <xsl:apply-templates select="Titel"/>
   <table width="600"><tr>

   <td width="120" valign="top">
   <xsl:for-each select="Kapitel">
      <small style="color:#000090; cursor:hand"
         onclick="javascript:zeige('box{$kap}', 'nr{$kap}{position()}')">
         <xsl:value-of select="Titel"/>
      </small>
      <br />
   </xsl:for-each>
   <br /><br /><br />

   </td>
   <td width="*" valign="top">
   <div style="position:absolute;" id="box{$kap}">
      <xsl:apply-templates select="Inhalt"/>
   </div>
   <xsl:for-each select="Kapitel">
   <div style="display:none" id="nr{$kap}{position()}">
      <xsl:apply-templates/>
   </div>
   </xsl:for-each>
   </td>
   </tr></table>
</xsl:template>
```

Tabellenzelle mit Themen-Navigation

Tabellenzelle mit dynamisch erzeugten Kapitelinhalten

Für den Test dieser Funktion verwenden wir ein komplexeres Kapitel (kapitel2.xml).

Schritt 5: Organisation des Handbuches

Die Organisation der Kapitel zu einem kompletten Handbuch erfolgt über ein kleines XML-Dokument, dass die Kapiteln als externe Entitäten importiert.

```
<!DOCTYPE  Buch [
   <!ELEMENT Buch (Titel, Kapitel*)>
   <!ENTITY  % kapitel SYSTEM "kapitel.dtd">
   %kapitel;
   <!ENTITY kap1 SYSTEM "kapitel1.xml">
   <!ENTITY kap2 SYSTEM "kapitel2.xml">
   <!ENTITY nop SYSTEM "nop.xml">
]>
<?xml-stylesheet type="text/xsl" href="kapitel.xsl"?>
<Buch>
   <Titel>Handbuch</Titel>
   &kap1;
   &kap2;
</Buch>
```

Die Kapitel-DTD wird als externes Vokabular eingebunden, die einzelnen Kapiteln als externe Entitäten

Workaround für Microsoft-Bug

Instanzierung der Kapiteln

An diesem kleinen Beispiel sieht man besonders eindrucksvoll die vielseitigen Aufgaben einer DTD: Das Minimal-Modell zur Beschreibung von Buch-Dokumenten wird durch die Kapitel-DTD ergänzt, die als externe Parameter-Entity (also externes Vokabular) eingebettet wird. Die einzelnen XML-Kapiteln werden ebenfalls in der DTD deklariert, aber erst im nachfolgenden XML-Buchdokument in der gewünschten Reihenfolge instanziert.

Beispiel 4

XSLT-Extensions verwenden

Workshop Vorbereitung zu Beispiel 4: XSLT-Extensions verwenden

Werkzeuge:
MS Internet-Explorer 5.5 (SP2) oder 6
Einfacher Editor (Wordpad ...)
XMLSpy-Editor (www.xmlspy.com)
XML-Parser Xerxes und XSL-Prozessor Xalan (www.apache.org)
Java JRE 1.3.x

XML-Sprachfamilie:
XSLT-Elements

XPath-Expressions

XSLT-Functions

```
xalan:write select="destination-file"
string(nodevalue)
concat(string, string ...)
```

Zielsprache:
HTML, JavaScript, Java

Beispiel 4: XSLT-Extensions verwenden

Redirection

XSLT-Redirection erlaubt die Ausgabe von Transformations-Ergebnissen in verschiedene Dateien. Dieser Mechanismus ist sinnvoll, wenn eine Transformation mehrfach genutzt werden soll, zum Beispiel zum Aufteilen großer Dokumente auf webgerechte kleine Teildokumente.

Wir werden Redirection im weiteren Verlauf des Workshops nutzen, daher interessiert uns im Moment nur die prinzipielle Anwendung. Für eine einfache Demonstration dieser Funktion nehmen wir daher das Hello World-Beispiel und erzeugen länderspezifische Ergebnisdokumente. Das erforderliche XSLT-Stylesheet sieht wie folgt aus:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xalan="org.apache.xalan.xslt.extensions.Redirect"
extension-element-prefixes="xalan"
>
<xsl:output method="html" indent="yes" encoding="iso-8859-1"/>

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

<xsl:template match="Greetings[@language='deutsch']">
   <html><body>
      <h1 align="center" style="color:#FF0000">Greetings</h1>
      <xsl:value-of select="@language"/>:
      <b><xsl:value-of select="." /></b>
   </body></html>
</xsl:template>

<xsl:template match="Greetings">
<xsl:param name="filename" select="@language"/>
<xalan:write select="concat($filename, string('.html'))">
   <html><body>
      <h1 align="center" style="color:#FF0000">Greetings</h1>
      <xsl:value-of select="@language"/>:
      <b><xsl:value-of select="." /></b>
   </body></html>
</xalan:write>
</xsl:template>

</xsl:stylesheet>
```

Diese Mehrfach-Transformation ist natürlich nicht mit dem Browser möglich, wir verwenden daher den Xalan-Prozessor,der über ein DOS-Kommando aufgerufen wird:

```
set savedClasspath=%CLASSPATH%

rem Workshop-Verzeichnis definieren
set xml=D:\brz\xml-workshop\xalan-j\
set CLASSPATH=.;%xml%xalan.jar;%xml%xerces.jar;%xml%bsf.jar;%CLASSPATH%
java org.apache.xalan.xslt.Process
       -xsl redirect.xsl
       -in helloworld_4.xml
       -out deutsch.html
set CLASSPATH=%savedClasspath%
```

Das „deutsche“ HTML-Dokument wird in diesem Fall als direktes Ergebnis der Xalan-Transformation erzeugt, die fremdsprachigen HTML-Dokumente durch Redirection.

Java-Binding

Java-Binding ist ebenfalls eine wichtige XSLT-Erweiterung. Damit steht der Sprachschatz einer mächtigen und objektorientierten Sprache samt ihrer Biblitheksfunktionen zur Verfügung. Wir wollen mit dieser Möglichkeit experimentieren.

Um die prinzipielle Funktion zu zeigen, erweitern wir unser HelloWorld-Beispiel mit einem Aufruf einer Java-Methode, dazu müssen wir nur das Stylesheet erweitern:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:rosen="HelloWorld">

<xsl:template match="/">
   <html>
   <body>
      <h1>Greetings</h1>
      <xsl:apply-templates select="//Greetings[contains(@language, 'isch')]"/>
      <xsl:apply-templates select="//Greetings[@language='deutsch']"/>
      <br /><br />
      Java-Greetings: 
      <xsl:value-of select="rosen:echo('Hallo')"/> 
      <xsl:value-of select="rosen:echo('Welt')"/>
   </body>
   </html>
</xsl:template>

<xsl:template match="Greetings">
   <p><xsl:value-of select="@language"/>:  <b><xsl:value-of select="." /></b></p>
</xsl:template>

</xsl:stylesheet>
```

Die Java-Klasse „HelloWorld“ wird mit einer Namespace-Anweisung unter einem beliebigen Namen deklariert (xmlns:rosen) und der Methodenaufruf erfolgt in einem select-Attribut einer xsl:value-of Anweisung. Das Java-Programm selbst ist einfach, es enthält nur eine einzige Methode „echo“, die einen übergebenen String retourniert:

```
public class HelloWorld {
      public String echo(String in)
            { return  in; }
}
```

Wir kompilieren das Java-Programm in einem DOS-Fenster mit `javac HelloWorld.java` und rufen anschließend den Xalan-Prozessor wie folgt auf:

```
set savedClasspath=%CLASSPATH%

rem Workshop-Verzeichnis definieren
set xml=D:\brz\xml-workshop\xalan-j\
set CLASSPATH=.;%xml%xalan.jar;%xml%xerces.jar;%xml%bsf.jar;%CLASSPATH%

java org.apache.xalan.xslt.Process
      -xsl helloworld_java.xsl
      -in helloworld_5.xml
      -out helloworld_java.html

set CLASSPATH=%savedClasspath%
```

Das Ergebnis ist ein HTML-Dokument „HelloWorld.html“ mit Java-Grüssen:

Java-Greetings: Hallo Welt

Verwendung der Java Encryption-API

Ein etwas anspruchsvolleres Beispiel einer Java-Anwendung befindet sich im Verzeichnis transformationen/crypto. Hier wird die Java Kryptographie-API verwendet, um ein XML-Dokument zu verschlüsseln. Die Java-Klasse ist in diesem Fall nicht mehr so trivial:

```
import java.io.*;
import java.security.*;
import sun.misc.*;
import javax.crypto.*;

public class Binding2 {

public static String encrypt(String in) throws Exception {
       ObjectInputStream savedKey = new ObjectInputStream(
                                    new FileInputStream("SecretKey.ser"));
       Key key = (Key)savedKey.readObject();
       Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
       cipher.init(Cipher.ENCRYPT_MODE, key);
       byte[] raw = cipher.doFinal(in.getBytes("iso-8859-1"));
       BASE64Encoder encoder = new BASE64Encoder();
       return encoder.encode(raw);
}

public static String decrypt(String in) throws Exception {
       ObjectInputStream savedKey = new ObjectInputStream(
                                    new FileInputStream("SecretKey.ser"));
       Key key = (Key)savedKey.readObject();
       savedKey.close();
       Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
       cipher.init(Cipher.DECRYPT_MODE, key);
       BASE64Decoder decoder = new BASE64Decoder();
       byte[] clearText = cipher.doFinal(decoder.decodeBuffer(in));
       return new String(clearText, "UTF8");
}
}// end class
```

Das zugehörige Stylesheet ist zwar relativ einfach, enthält aber ebenfalls ein etwas anspruchsvolleres Detail:

```
<!-- ========== Root template - start processing here ========== -->
<xsl:template match="/">
<HTML>
   <BODY>
      <xsl:variable name="text" select="string(/Dingsda/text)"/>
      <b>Originaltext</b>
      <p> <xsl:value-of select="$text"/> </p>
      <b>Verschluesselter Text</b>
      <xsl:variable name="cipher" select="rosen:encrypt($text)"/>
      <p> <xsl:value-of select="$cipher"/> </p>
      <b>Entschluesselter Text</b>
      <p> <xsl:value-of select="rosen:decrypt($cipher)"/> </p>
   </BODY>
</HTML>
</xsl:template>
```

Von der encrypt-Methode der Java-Klasse wird ein String als Parameter erwartet, das Ergebnis von `xsl:value-of` ist aber im Normalfall eine Document-Node, die XSLT-intern automatisch in einen String konvertiert wird, wenn es erforderlich ist. Um die Methoden-Signatur der Java-Klasse korrekt zu bedienen, muß daher die Document-Node gezielt in einen String überführt werden, was am einfachsten mit einer String-Operation geschieht.

Die Verwendung von Kryptographie-Funktionen ist im Zusammenhang mit digitalen Signaturen von Bedeutung.

Beispiel 5

XML-SQL Transformation

Workshop Vorbereitung zu Beispiel 5:
XML-SQL Transformation

Werkzeuge:
MS Internet-Explorer 5.5 (SP2) oder 6
Einfacher Editor (Wordpad ...)
XMLSpy-Editor (www.xmlspy.com)
XML-Parser Xerxes und XSL-Prozessor Xalan (www.apache.org)
Java JRE 1.3.x

XML-Sprachfamilie:
XSLT-Elements

```
xsl:output method="output-mode"
```

XPath-Expressions

XSLT-Functions

Zielsprache:
HTML, Text

Beispiel 5: XML-SQL Transformationen

Ein einfaches Beispiel für eine XML-Transformation ist die Erzeugung einer SQL-Datei, um Dokumentinhalte in eine Datenbank zu transferieren. Wir verwenden dafür ein beliebiges XML-Datendokument – im vorangegangenen Beispiel „Auswertung“ haben wir ja bereits ein interessantes Dokument behandelt, nämlich die protokollierten Zugriffe auf ein Webservice (ausw.xml).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?xml-stylesheet href="ausw.xsl" type="text/xsl" ?>
<auswertung>
<entry>
   <time>2002-09-04-20.41.34.545000</time>
   <provider>TMVIMD1</provider>
   <abfrager>Name 1</abfrager>
   <grund>Wichtig</grund>
   <zeilen>15</zeilen>
   <zeichen>1481</zeichen>
</entry>
<entry>
   ...
</entry>
</auswertung>
```

Das erforderliche XSLT-Stylesheet ist einfach zu formulieren:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="text" encoding="iso-8859-1"/>

<xsl:template match="/">
<xsl:for-each select="//entry">
insert into Tab
   Name (
   <xsl:for-each select=".//*[name()!='criteria']">
      <xsl:value-of select="name()"/>
      <xsl:if test="position()!=last()">, </xsl:if>
   </xsl:for-each>)
   value (
   <xsl:for-each select=".//*[name()!='criteria']">
      <xsl:value-of select="."/>
      <xsl:if test="position()!=last()">, </xsl:if></xsl:for-each>);
   </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Als Output-Methode wird mit der `<xsl:output>`-Anweisung der Textmodus gewählt. Wir werden später sehen, dass wir deswegen noch unser XSLT-Stylesheet „nachjustieren“ müssen, um unnötige Zeilenumbrüche in unserer Ausgabedatei zu verhindern.

Für die Produktion der SQL-Datei verwenden wir den Xalan-Prozessor, den wir uns beim Apache-Projekt besorgen (www.apache.org) und im Verzeichnis xml-workshop/xalan-j installieren. Xalan ist eine Java-API und setzt daher eine lauffähige Java-Installation voraus (JRE). Ob diese Voraussetzungen gegeben sind, sehen wir im DOS-Fenster mit dem Kommando `java -v`. Idealerweise erhalten wir die Bestätigung, dass Java 1.3.1 oder 1.4.1 vorhanden ist (für XML-Experimente ist leider die Release 1.4.0 unbrauchbar).

Xalan-Start über Kommandozeile (transform.bat)

```
set savedClasspath=%CLASSPATH%

rem Workshop-Verzeichnis definieren
set xml=D:\brz\xml-workshop\xalan-j\
set CLASSPATH=.;%xml%xalan.jar;%xml%xerces.jar;%xml%bsf.jar;%CLASSPATH%

java org.apache.xalan.xslt.Process -xsl sql.xsl -in ausw.xml -out ausw.sql

set CLASSPATH=%savedClasspath%
```

Diese kleine Kommandozeilen-Prozedur (BAT-Datei) erleichtert uns den Umgang mit Xalan, indem es die erforderlichen Pfadangaben zu den Xalan-Bibliotheken (jar-Dateien) herstellt und den Xalan-Prozessor mit den erforderlichen Argumenten aufruft:

-xsl sql.xsl	definiert das XSLT-Stylesheet
-in ausw.xml	definiert das XML-Datendokument
-out ausw.sql	definiert das Ergebnisdokument

Ergebnis ausw.sql

```
insert into
TabName (time, provider, abfrager, grund, zeilen, zeichen)
value (2002-09-04-20.41.34.545000, TMVIMD1, Name 1, Wichtig, 15, 1481);

insert into
TabName (time, provider, abfrager, grund, zeilen, zeichen)
value (2002-09-04-21.01.30.235000, DATAKOMP, Name 2, Unwichtig, 12, 1314);

insert into
TabName (time, provider, abfrager, grund, zeilen, zeichen)
value (2002-09-04-21.01.45.235000, DATAKOMP, Name 2, Wichtig, 18, 1950);
```

Das Ergebnis ist eine SQL-Datei, die in der entsprechenden Datenbank-Umgebung aufrufbar ist. Der „Umweg“ über so eine Zwischendatei ist normalerweise nicht erforderlich, da über JDBC (Java-ODBC) auch eine direkte Möglichkeit für Daten-Transaktionen geboten wird. Diese Möglichkeit werden wir in einem folgenden Beispiel nutzen (SAX-Parser Beispiel).

Beispiel 6

XML-PDF Transformation

Workshop Vorbereitung zu Beispiel 6:
XML-PDF Transformation

Werkzeuge:
MS Internet-Explorer 5.5 (SP2) oder 6
Einfacher Editor (Wordpad ...)
XMLSpy-Editor (www.xmlspy.com)
XML-Parser Xerxes und XSL-Prozessor Xalan (www.apache.org)
XSL-FO-Prozessor (www.apache.org)
Java JRE 1.3.x

XML-Sprachfamilie:
XSLT-Elements

XPath-Expressions

XSLT-Functions

XSL-FO
- fo:root
- fo:layout-master-set
- fo:simple-page-master
- fo:page-sequence-master
- fo:page-sequence
- fo:static-content
- fo:flow
- fo:block

Zielsprache:
HTML, XSL-FO (Formatting Objects), PDF

Beispiel 6: XML-PDF Transformation mit Apache FOP

Die mächtige Sprache XSL-FO (Formatting Objects) ist ideal für anspruchsvolle Dokumentproduktion – daher wollen wir uns mit den technischen Voraussetzungen für den Einsatz dieser Sprache bekannt machen.

Diese Aufgabenstellung unterscheidet sich grundlegend von den vorangegangenen HTML-Produktionen. Wir müssen dazu eine neue Sprache lernen, XSL-FO –eine sehr komplexe Seitenbeschreibungs-Sprache. Mangels Zeit und entsprechender Literatur (es ist tatsächlich eine neue Sprache, und daher noch wenig Literatur verfügbar), beschränken wir uns auf wesentliche Details.

XSL-FO-Stylesheet (xml2pdf1.xsl)

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
     xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:template match ="/">
   <fo:root>
   <!-- define page layout -->
   <fo:layout-master-set>
      <fo:simple-page-master master-name="odd">
      </fo:simple-page-master>
      <fo:simple-page-master master-name="even">
      </fo:simple-page-master>
      <!-- define page sequence -->
      <fo:page-sequence-master master-name="psm" >
      </fo:page-sequence-master>
   </fo:layout-master-set>
   <!-- execute page sequence -->
   <fo:page-sequence master-reference="psm" >
         ... <xsl:apply-templates/> ...
   </fo:page-sequence>
</fo:root>
</xsl:template>

<!-- Template Library -->
</xsl:stylesheet>
```

Ein XSL-FO-Stylesheet enthält zumindest folgende wichtige Elemente:

- Eine Namespace-Deklaration für das XSL-FO Vokabular
  xmlns:fo="http://www.w3.org/1999/XSL/Format"
- Ein FO-Root Element, eingebettet in das Root-Template des Stylesheets
- Ein Layout-Master-Set mit zumindest einem Simple-Page-Master. In unserem Beispiel wollen wir zwischen geraden und ungeraden Seiten unterscheiden, daher sind zwei Simple-Page-Master (odd, even) und ein zusätzlicher Page-Sequence-Master erforderlich
- Ein Page-Sequence Element mit einer Referenz auf einen Page-Master, in unserem Beispiel ein Page-Sequence-Master
- Eine Template-Bibliothek, wie wir sie schon von XSLT kennen

Diese grundlegenden Bestandteile bilden das Skelett eines XSL-FO-Dokumentes, vergleichbar einem HTML-Skelett (head, body). Wir werden allerdings kein FO-Dokument erzeugen, sondern über einen FO-Prozessor direkt ein PDF-Dokument. Doch vorher müssen wir diese Elemente im Detail formulieren.

Layout-Master-Set

Das Layout-Master-Set enthält einen oder mehrere Simple-Page-Master, die das Seitenlayout und die einzelnen Seiten-Regionen beschreiben, in unserem Fall für gerade und ungerade Seiten:

```
<!-- define page layout -->
<fo:layout-master-set>
   <fo:simple-page-master master-name="odd"
      page-height="29.7cm" page-width="21cm"
      margin-top="1.5cm" margin-bottom="1cm"
      margin-left="1.5cm" margin-right="1.5cm">
      <fo:region-before extent="1cm"/>
      <fo:region-after extent="1cm"/>
      <fo:region-body margin-top="1.5cm" margin-bottom="1.5cm" margin-left="3cm"/>
   </fo:simple-page-master>
   <fo:simple-page-master master-name="even"
      page-height="29.7cm" page-width="21cm"
      margin-top="1.5cm" margin-bottom="1cm"
      margin-left="1.5cm" margin-right="1.5cm">
      <fo:region-before extent="1cm"/>
      <fo:region-after extent="1cm"/>
      <fo:region-body margin-top="1.5cm" margin-bottom="1.5cm" margin-right="3cm"/>
   </fo:simple-page-master>
   <!-- define page sequence -->
</fo:layout-master-set>
```

Von den fünf möglichen Regionen (begin, end, before, after und body) brauchen wir nur drei: `xsl-region-before` für die Kopfzeile, `xsl-region-after` für die Seitennummern, und natürlich `xsl-region-body` für unseren Dokumentinhalt.

Page-Sequence-Master

Der Page-Sequence-Master ist im Layout-Master-Set eingebettet und dient in unserem Fall zur Unterscheidung von geraden und ungeraden Seiten-Layouts. Ein anderes Anwendungsbeispiel wäre die Unterscheidung zwischen erster Seite und Rest des Dokumentes mit `page-position` statt `odd-or-even`.

```
   <!-- define page sequence -->
   <fo:page-sequence-master master-name="psm" >
      <fo:repeatable-page-master-alternatives>
         <fo:conditional-page-master-reference master-reference="odd"
            odd-or-even="odd" />
         <fo:conditional-page-master-reference master-reference="even"
            odd-or-even="even" />
      </fo:repeatable-page-master-alternatives>
   </fo:page-sequence-master>
```

Damit ist das Layout-Master-Set definiert. Was jetzt noch fehlt, ist die eigentliche Dokumentbearbeitung. Diese erfolgt im Page-Sequence Element unter Verwendung einer Template-Bibliothek.

Page-Sequence Element

Das Page-Sequence Element produziert den Inhalt der einzelnen Seiten-Regionen. Dazu benötigt es die Information über das Seitenlayout, also eine Referenz auf einen Simple-Page-Master oder einen Page-Sequence-Master. Nachdem wir beide Möglichkeiten vorgesehen haben, können wir auch mit beiden Layout-Varianten experimentieren:

Simple-Page-Master für einheitliches Seitenlayout (odd oder even)

```
<fo:page-sequence master-reference="even" >
<fo:page-sequence master-reference="odd" >
```

oder Page-Sequence-Master für wechselweises Seitenlayout

```
<fo:page-sequence master-reference="psm" >
```

Der komplette Page-Sequence Block im Detail:

```
<!-- execute page sequence -->
<fo:page-sequence master-reference="odd" >
   <fo:static-content flow-name="xsl-region-before">
      <fo:block text-align="center" font-size="8pt">
         <xsl:value-of select="/Kapitel/Titel"/>
      </fo:block>
   </fo:static-content>
   <fo:static-content flow-name="xsl-region-after">
      <fo:block text-align="center" font-size="8pt">
         Seite <fo:page-number/>
      </fo:block>
   </fo:static-content>
   <fo:flow flow-name="xsl-region-body">
      <fo:block font-size="11pt" text-align="left">
         <xsl:apply-templates/>
      </fo:block>
   </fo:flow>
</fo:page-sequence>
```

Zwei `static-content` Elemente dienen zur Aufnahme des Kapiteltitels und der Seitennummer. Der Kapiteltitel wird mit `xsl:value-of` aus dem XML-Dokument geholt, die Seitennummer wird mit der Funktion `fo:page-number` erzeugt. Diese statischen Regionen werden auf allen Seiten wiederholt. Das flow Element ist der Behälter für den Dokumentinhalt, der mit `xsl:apply-templates` produziert wird. Es ist darüber hinaus ersichtlich, dass alle Produktionen in `fo:block` Elemente eingebettet sind. Dieses spezielle Element ist neben `fo:table` eines der wichtigsten FO-Elemente. Wir werden beide in unserer Template-Bibliothek verwenden.

Template-Bibliothek

```
<!-- Template Library -->
<xsl:template match ="Kapitel/Titel">
   <fo:block font-size="24pt" text-align="left"
         space-before.optimum="18pt" space-after.optimum="18pt">
      <xsl:apply-templates />
   </fo:block>
</xsl:template>

<xsl:template match ="p[@thema]">
<fo:table font-size="11pt" table-layout="fixed">
   <fo:table-column column-width="1cm" column-number="1"/>
   <fo:table-column column-width="14cm" column-number="2"/>
   <fo:table-body><fo:table-row>
      <fo:table-cell >
         <fo:block><xsl:value-of select="@thema"/></fo:block>
      </fo:table-cell>
      <fo:table-cell >
         <fo:block><xsl:apply-templates/></fo:block>
      </fo:table-cell>
   </fo:table-row></fo:table-body>
</fo:table>
</xsl:template>

<xsl:template match ="Bild">
   <fo:block font-size="10pt" text-align="left">
      <fo:external-graphic src="{@src}"/>
   </fo:block>
</xsl:template>

<xsl:template match ="b">
   <fo:inline font-weight="bold"><xsl:apply-templates /></fo:inline>
</xsl:template>
```

FOP-Start über Kommandozeile (fop1.bat)

Der Aufruf des Formatting-Object-Prozessors (FOP), den wir uns beim Apache-Projekt (www.apache.org) besorgt und im Verzeichnis xml-workshop/xml-fop installiert haben, ist nicht gerade trivial, daher verwenden wir wieder eine Startdatei:

```
rem FOP-Verzeichnis definieren
set fop= D:\brz\xml-workshop \xml-fop\

java -cp
   %fop%build\fop.jar;
   %fop%lib\batik.jar;
   %fop%lib\xalan-2.0.0.jar;
   %fop%lib\xerces-1.2.3.jar;
   %fop%lib\avalon-framework-4.0.jar;
   %fop%lib\logkit-1.0b4.jar;
   %fop%lib\jimi-1.0.jar
    org.apache.fop.apps.Fop -xml kapitel.xml -pdf kapitel.pdf -xsl xml2pdf1.xsl
```

Die erforderlichen Bibliotheken sind zur besseren Übersicht untereinander aufgelistet. Diese FOP-Release setzt auf eine ganz bestimmte Xalan-Implementierung auf, die im Packet mitgeliefert wird. Als Ergebnis erhalten wir ein PDF-Dokument (kapitel.pdf).

Einbettung von Unicode-Fonts

Neben den bekannten Vorteilen des plattform-unabhängigen PDF-Formats ist noch ein besonderer Aspekt interessant, nämlich die Möglichkeit, Unicode-Fonts einzubetten. Damit lassen sich „internationalisierte“ Dokument erstellen und verteilen. Diese wichtige Eigenschaft wollen wir uns genauer ansehen. Wir verwenden dafür ein fertiges XML-Dokument (kapitel.xml), das viele Sonderzeichen enthält, die nicht in einem normalen Font (zum Beispiel Arial) vorgesehen sind.

Um die Möglichkeit eingebetteter Unicode-Zeichen zu nutzen, muß der FO-Prozessor die entsprechenden Font-Metriken kennen. Diese Metrik-Daten werden über einen TrueType-Fontreader erzeugt, der im FOP-Packet mit enthalten ist. Für den Aufruf des Fontreaders verwenden wir die Startdatei metric.bat. Die erzeugte Fontmetrik, in unserem Beispiel lsansuni.xml für den Font „Lucida Sans Uni“, muß in die config.xml-Datei des FOP-Prozessors eingetragen werden:

metric.bat	Startdatei für Truetype-Reader
lsansuni.xml	Fontmetrik-Datei für Lucida-SansUnicode
xml-fop/conf/config.xml	Config-Datei des FO-Prozessors

Voraussetzung dafür ist natürlich auch die Installation des entsprechende Unicode-Fonts, allerdings nur am Produktionssystem des Autors. Damit sind wir nun in der Lage, Unicode-Zeichen aus diesem Font in unser Dokument einzubetten. Wir müssen nur mehr den Font in unserem Stylesheet xml2pdf1.xsl deklarieren und im flow Element verwenden:

```
<xsl:template match ="/">
<!-- xsl:param name="font" select="string('any')"/ -->
<xsl:param name="font" select="string('Lucida Sans Unicode')"/>
<fo:root>

   <fo:flow flow-name="xsl-region-body">
      <fo:block font-size="11pt" font-family="{$font}" text-align="left">
         <xsl:apply-templates/>
      </fo:block>
   </fo:flow>
```

Beispiel 7

XML-XML Transformation

Workshop Vorbereitung zu Beispiel 7: XML-XML Transformation

Werkzeuge:
MS Internet-Explorer 5.5 (SP2) oder 6
Einfacher Editor (Wordpad ...)
XMLSpy-Editor (www.xmlspy.com)
XML-Parser Xerxes und XSL-Prozessor Xalan (www.apache.org)
Java JRE 1.3.x

XML-Sprachfamilie:
XSLT-Elements

XPath-Expressions

XSLT-Functions

Zielsprache:
XML, HTML, JavaScript

Beispiel 7: XML-XML Transformation

Die Transformation eines XML-Dokumentes in ein anderes XML-Dokument ist aus unterschiedlichen Gründen sinnvoll. Zum Beispiel, um Dokumentinhalte in ein anderes XML-Vokabular zu verpacken (Identitätstransformation), oder um XML-Dokumente zu reorganisieren. Der zweite Fall ist für unseren Workshop interessant.

Wir haben am Beispiel des „Intelligenten Datendokumentes“ gesehen, dass wir mit einer zweiphasigen Transformation ein Dokument nach auswertbaren Kriterien analysieren und anschließend auswerten können. Das funktioniert allerdings nur bei kleinen Datenmengen (bis ca. 200 kByte). Bei größeren Dokumenten führt die rekursive Suche nach Auswerte-Kriterien zum Kollaps, da der Resourcenbedarf (Zeit und Speicher) quadratisch mit der Datenmenge ansteigt.

Für unser Experiment verwenden wir wieder den Xalan-Prozessor und als Datendokument die Log-Datei ausw.xml (Webservice-Zugriffsstatistik) in vier verschiedenen Größen:

- small.xml (82 kByte)
- medium1.xml (540 kByte)
- medium2.xml (972 kByte)
- big.xml (2018 kByte)

Versuch 1: Reorganisation der Log-Datei

Durch Entfernen unnötiger Einträge läßt sich das Quelldokument auf die Hälfte reduzieren (ausw1.xml). Das entsprechende Stylesheet ist praktisch nur eine einfache Liste mit allen zu kopierenden Elementen, die im Ergebnisdokument aufscheinen sollen:

```
<!-- Root Template -->
<xsl:template match="/">

   <!-- Make the Content-Document -->
<xsl:copy-of select="processing-instruction()"/>
<xsl:element name="auswertung">
<xsl:for-each select="//entry">
   <xsl:element name="entry">
      <!-- xsl:element name="session"> ignorieren </xsl:element -->
      <!-- xsl:element name="request"> ignorieren </xsl:element -->
      <xsl:element name="time">
         <xsl:value-of select="concat(substring(time,1,4), substring(time,6,2),
                                substring(time,9,2))"/>
      </xsl:element>
      <xsl:element name="provider"><xsl:value-of select="provider"/></xsl:element>
      <xsl:element name="abfrager"><xsl:value-of select="abfrager"/></xsl:element>
      <!-- xsl:element name="grund"> ignorieren </xsl:element -->
      <xsl:element name="zeilen"><xsl:value-of select="zeilen"/></xsl:element>
      <xsl:element name="zeichen"><xsl:value-of select="zeichen"/></xsl:element>
      <!-- xsl:copy-of select="criteria"/ -->
   </xsl:element>
</xsl:for-each>
</xsl:element>

</xsl:template>
```

Durch Verwendung von Attributen statt Elementen ist eine Reduktion auf ein Fünftel der Original-Dateigröße möglich (ausw2.xml), das entsprechende XSL-Kodeschnipsel:

```
   <xsl:element name="entry">
      ...
      <xsl:attribute name="p"><xsl:value-of select="provider"/></xsl:attribute>
      ...
   </xsl:element>
```

Versuch 2: Optimierung der Suchmethode

Trotz der erheblichen Textreduktion ist im Prinzip die Datenmenge erhalten geblieben, und somit die rekursive Suche nach wie vor ein Problemfall. Wir werden uns daher verschieden Methoden zur Optimierung des Suchalgorithmus ansehen.

Die erste Überlegung betrifft den Dokumentaufruf selbst, das XML-Dokument startet den Browser und wird anschließen als DOM-Objekt nochmals instanziert. Das DOM-Objekt benötigen wir für die Auswertung des Dokumentes, aber für den Start des Browsers könnten wir ein optimiertes „Menü-Dokument“ (StartMenu.xml) verwenden, daß nur die gefilterten Auswahloptionen enthält, wie nachfolgend gezeigt:

StartMenu.xml

```
<auswertung>
    <t>20020903</t>
    <t>20020904</t>
      ...
    <p>TMVIMD1</p>
    <p>DATAKOMT</p>
    <p>DATAKOMT1</p>
      ...
    <a>test</a>
    <a>99999</a>
    <a>R598011</a>
      ...
</auswertung>
```

Genau das ist aber unser Problem, wir können bei großen Datenmengen die vielen Einträge (Entries) nicht filtern, zumindest nicht in einer Transformation alleine. Wir brauchen irgendeine vorbereitende Transformation, zum Beispiel um die Menge der gleichartigen Kriterien zu sortieren oder zu reduzieren. Wir erweitern daher die Reorganisation des XML-Dokumentes (1.Transformation) um diese neue Funktion:

Reorganisation und vorbereitende Optimierung (transform.bat)

Die Methode der Dokument-Reorganisation aus dem ersten Versuch behalten wir bei und erzeugen ein wesentlich reduziertes Datendokument ausw.xml. Zusätzlich erzeugen wir in diesem Transformationsschritt eine sortierte Folge der Auswahlkriterien Zeit, Provider und Abfrager, die wir durch Redirection in einer zweiten Datei speichern (_ausw.xml).

```
   <!-- Make the sorted List -->
<xalan:write select="string('_ausw.xml')">
<xsl:copy-of select="processing-instruction()"/>
<xsl:element name="auswertung">

   <xsl:element name="t">
   <xsl:for-each select="($entry/time)">
      <xsl:sort select="."/>
      <xsl:element name="t"><xsl:value-of select="."/></xsl:element>
   </xsl:for-each>
   </xsl:element>

   <xsl:element name="p">
   <xsl:for-each select="($entry/provider)">
      <xsl:sort select="."/>
      <xsl:element name="p"><xsl:value-of select="."/></xsl:element>
   </xsl:for-each>
   </xsl:element>

   <xsl:element name="a">
   <xsl:for-each select="$entry/abfrager">
      <xsl:sort select="."/>
      <xsl:element name="a"><xsl:value-of select="."/></xsl:element>
   </xsl:for-each>
   </xsl:element>
</xsl:element>
</xalan:write>
```

Ein interessantes Detail: Obwohl es sich bei dieser Datei nur um ein Zwischenergebnis handelt – das gefilterte Menü-Dokument wollen wir ja in einer zweiten Transformation erzeugen, produzieren wir eine Processing-Instruction (PI) für das Anzeige-Stylesheet. Aus einem einfachen Grund, wir müssen uns die PI für die zweite Transformation merken, den im endgültigen Menü-Dokument soll sie ja vorhanden sein.

Und ein zweites Detail: Die Zeiteinträge sind auf Tausendstel Sekunden genau dargestellt, wir wollen aber Tage als Auswahlkriterium anbieten. Wir müssen daher das Zeitformat ändern. In diesem Zusammenhang führen wir überhaupt gleich einen radikalen Eingriff durch und verwerfen 90 % der Zeitstempel. Uns interessieren nur Tage mit mehr als 10 protokollierten Webservice-Zugriffen.

```
<xsl:element name="t">
<xsl:for-each select="($entry/time)[position() mod 10 = 0]">
   <xsl:sort select="."/>
   <xsl:element name="t">
      <xsl:value-of select="concat(substring(.,1,4), substring(.,6,2),
                                   substring(.,9,2))"/>
   </xsl:element>
</xsl:for-each>
</xsl:element>
```

Unsere Zwischendatei _ausw.xml enthält nun sortiert, aber ungefiltert alle gewünschten Abfrage-Kriterien. Der zusätzliche Zeitaufwand für das Sortieren beträgt einige Sekunden und ist damit praktisch vernachlässigbar. Nun beginnen wir mit unserem Optimierungs-Experiment. Unser Ziel ist, alle Mehrfacheinträge aus der Liste zu entfernen und nur jeweils einen Eintrag pro Tagesdatum, Providername und Abfragername zu haben.

1. Vergleich über preceding::entry/x (transform21.bat)
Diese ursprüngliche Methode basierte auf der Iteration des unsortierten preceding Node-Sets, ob bereits ein identer Eintrag in der abgearbeiteten Datenmenge vorliegt. Der nachfolgende Kodeausschnitt zeigt diese Methode am Beispiel des Zeiteintrages:

```
<xsl:for-each select="//entry[not(@t=preceding::entry/@t)]">
   <xsl:element name="t">
      <xsl:value-of select="@t"/>
   </xsl:element>
</xsl:for-each>
```

Zeit für Einträge filtern:

small.xml	3 850 ms
medium1.xml	358 170 ms !!!

2. Vergleich über preceding::x[1] (transform22.bat)
Diese Methode verwendet die sortierte Zwischendatei und vergleicht Einträge gezielt mit dem unmittelbar benachbarten Preceding-Element (in unserem sortierten Dokument verwenden wir jetzt wieder Elemente statt Attribute – siehe _ausw.xml):

```
<xsl:for-each select="$t/t[not(.=preceding::t[1])]">
   <xsl:element name="t">
      <xsl:value-of select="."/>
   </xsl:element>
</xsl:for-each>
```

Zeit für Einträge filtern:

small.xml	990 ms
medium1.xml	29 600 ms

Das Ergebnis ist um eine Größenordnung besser, statt 350 Sekunden für ein mittleres Dokument sind nur mehr 30 Sekunden erforderlich.

3. Vergleich über Index-Position (transform23.bat)
Diese Methode verwendet die sortierte Zwischendatei und vergleicht Einträge gezielt mit dem unmittelbar benachbarten Element (position() –1):

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

   <xsl:for-each select="$t/t">
      <xsl:variable name="x" select="position()"/>
      <xsl:if test="not(.=$t/t[$x - 1])">
         <xsl:element name="t">
            <xsl:value-of select="."/>
         </xsl:element>
      </xsl:if>
   </xsl:for-each>
```

Zeit für Einträge filtern:

small.xml	930 ms
medium1.xml	26 700 ms

Das Ergebnis ist um Nuancen besser als die preceding-Methode. Das liegt vermutlich an der Verwendung der lokalen Variablen $t auf beiden Seiten der Vergleichsoperation. Im vorangegangenen Beispiel werden für den Vergleich unterschiedliche Node-Sets verwendet, nämlich die Ergebnismenge der for-each Operation und die preceding-Menge.

4. Vergleich über Index-Hopping (transform24.bat)
Diese Methode verwendet die sortierte Zwischendatei und berechnet jeweils die Anzahl gleicher Einträge mit der count() Methode. Damit ist automatisch der Startindex des nächsten ungleichen Eintrags bestimmt. Die Realisierung dieser Methode in XSLT ist allerdings eine echte Herausforderung, da XSLT keine Variablen im herkömmlichen Sinne kennt. Das zuständige Template ruft sich rekursiv auf, solange es ein Folgeelement auf der berechneten neuen Indexposition gibt. Bei jedem Aufruf wird mit `with-param` der aktuelle Index für die Neuberechnung übergeben. Da das rekursiv aufgerufene Template einen neuen Scope besitzt, kann die lokale Parameter-Variable $z mehrfach genutzt werden, was einem Variablen-Update entspricht.

```
   <xsl:param name="z" select="'1'"/>
   <xsl:apply-templates select="(//t/t)[1]">
      <xsl:with-param name="z" select="$z"/>
   </xsl:apply-templates>
   ...
<xsl:template match="t">
   <xsl:param name="y">
      <xsl:value-of select="count(//t[.=current()])"/>
   </xsl:param>
   <xsl:element name="t"><xsl:value-of select="."/></xsl:element>
   <xsl:if test="(//t/t)[$y + $z]">
      <xsl:apply-templates select="(//t/t)[$y + $z]">
         <xsl:with-param name="z" select="$y + $z"/>
      </xsl:apply-templates>
   </xsl:if>
</xsl:template>
```

Zeit für Einträge filtern:

small.xml	3 410 ms
medium1.xml	131 550 ms

Auf den ersten Blick ein unattraktives Ergebnis. Diese Methode versagt beim Filtern vieler ungleicher Einträge, zum Beispiel den Abfrager-Namen. Wenn nur wenige verschiedene Einträge aus einer großen Datenmenge gefiltert werden müssen (zum Beispiel Provider-Name), ist diese Methode allerdings sehr effizient.

5. Rekursive Node-Set Reduktion (transform25.bat)

Diese Methode basiert auf der Überlegung, die Treffermenge eines rekursiv aufgerufenen Templates schrittweise zu reduzieren, indem das Match-Kriterium laufend eingeengt wird.

```
<xsl:call-template name="zeit">
   <xsl:with-param name="x" select="//t/t"/>
</xsl:call-template>

<xsl:template name="zeit">
   <xsl:if test="$x">
      <xsl:element name="t">
         <xsl:value-of select="$x[1]"/>
      </xsl:element>
      <xsl:call-template name="zeit">
         <xsl:with-param name="x" select="$x[not(.=$x[1])]"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>
```

Zeit für Einträge filtern:

small.xml	1 100 ms
medium1.xml	135 500 ms

Vom Ergebnis entspricht diese Methode dem vorangegangenen Beispiel (Index-Hopping).

6. Kombination aus Methode 3 und 4 (transform26.bat)

```
<!-- Root Template -->
<xsl:template match="/">
<xsl:variable name="t" select="/auswertung/t"/>
<xsl:variable name="p" select="/auswertung/p"/>
<xsl:variable name="a" select="/auswertung/a"/>
   <!-- Make the Menu-Document -->
<xsl:copy-of select="processing-instruction()"/>
<xsl:param name="z" select="'1'"/>
<xsl:element name="auswertung">
   <xsl:for-each select="$t/t">
      <xsl:variable name="x" select="position()"/>
      <xsl:if test="not(.=$t/t[$x - 1])">
         <xsl:element name="t"><xsl:value-of select="."/></xsl:element>
      </xsl:if>
   </xsl:for-each>
   <xsl:apply-templates select="(//p/p)[1]">
      <xsl:with-param name="z" select="$z"/>
   </xsl:apply-templates>
   <xsl:for-each select="$a/a">
      <xsl:variable name="x" select="position()"/>
      <xsl:if test="not(.=$a/a[$x - 1])">
         <xsl:element name="a"><xsl:value-of select="."/></xsl:element>
      </xsl:if>
   </xsl:for-each>
</xsl:element>
</xsl:template>

<xsl:template match="p">
   <xsl:param name="y">
      <xsl:value-of select="count(//p[.=current()])"/>
   </xsl:param>
   <xsl:element name="p"><xsl:value-of select="."/></xsl:element>
   <xsl:if test="(//p/p)[$y + $z]">
      <xsl:apply-templates select="(//p/p)[$y + $z]">
         <xsl:with-param name="z" select="$y + $z"/>
      </xsl:apply-templates>
   </xsl:if>
</xsl:template>
```

Methode 3

Methode 4

Methode 3

Zeit für Einträge filtern:

small.xml	1 210 ms
medium1.xml	18 180 ms
medium2.xml	62 780 ms

Durch die Verwendung eines sortierten Zwischendokumentes haben wir somit eine maximale Verbesserung um den Faktor 20 erzielt. Die Wartezeit von einer Minute bei einem ca 1 Mbyte großen Datendokument (medium2.xml) ist trotzdem unbefriedigend. Unser eigentlicher Gewinn ist eher das nun tiefergehende Verständnis für XSLT.

Versuch 3: Vorfiltern einer Teilmenge

Für unser eigentliches Problem hilft jetzt nur mehr ein pragmatischer Ansatz. Wir haben erkannt, das template-orientierte Programmiersprachen wie XSLT für solche Aufgaben schlecht geeignet sind. Das Fehlen echter Variablen in XSLT ist ein oft kritisierter Mangel und die angebotenen rekursive Möglichkeiten ein schlechter Ersatz. Wir müssen daher die Datenmenge irgendwie reduzieren, Sortieren alleine hilft nicht.

Hier kommt uns die Statistik zu Hilfe. Wir filtern ganz einfach eine kleine Teilmenge, zum Beispiel die ersten 200 Einträger der Log-Datei, und speichern diese gefilterte Teilmenge ebenfalls im Zwischendokument _ausw.xml ab. Mit etwas statistischem Glück haben wir aus der kleinen Teilmenge mit minimalem Zeitaufwand bereits einen Großteil der Kriterien gefiltert und können dieses Ergebnis direkt verwenden, und was besonders wichtig ist, mit einer not()-Klausel aus der Restmenge ausschließen:

```
<xsl:variable name="p0" select="/auswertung/p"/>
<xsl:variable name="p1" select="$p0/p1"/>
<xsl:variable name="p" select="$p0/p[not(.=$p1)]"/>

   <xsl:for-each select="//p1">
      <xsl:element name="p">
         <xsl:value-of select="."/>
   </xsl:element>
   </xsl:for-each>
   <xsl:for-each select="$p">
      <xsl:variable name="x" select="position()"/>
      <xsl:if test="not(.=$p[$x - 1])">
         <xsl:element name="p">
            <xsl:value-of select="."/>
         </xsl:element>
      </xsl:if>
   </xsl:for-each>
```

Vorgefilterte Menge p1

```
<p1>TMVIMD1</p1>
<p1>DATAKOMT1</p1>
<p1>DATAKOMP</p1>
<p1>DATAKOMT</p1>
<p1>IMDP</p1>
<p1>IMDT</p1>
```

Zeit für Einträge filtern:

small.xml	330 ms
medium1.xml	3 190 ms (statt 358 170 ms)
medium2.xml	8 900 ms
bigxml	43 610 ms

Die gefilterte Teilmenge p1 (für Providernamen) wird direkt in das Ergebnisdokument StartMenu.xml exportiert und anschließend der verbleibende Rest **`$p0/p[not(.=$p1)]`** gefiltert. Damit haben wir das ursprüngliche Ergebnis um mehr als 2 Größenordnungen verbessert, statt 358 170 ms nur mehr 3 190 ms für das Dokument medium2.xml.

Vermutlich haben wir damit die Grenze des Machbaren in XSLT erreicht. Wie einfach und performant unser Problem mit echten „lokalen Variablen“ lösbar wäre, zeigt der folgende Versuch 4. Aber lokale Variable sind in der puristischen XSLT-Welt ein vieldiskutiertes Tabu, sie stören nämlich das grundlegenden XSLT-Designprinzip: „No side-effects“!

XSLT-Designprinzip „No side-effects“

Originalzitat von Michael Kay, XSLT-Guru und Saxon-Entwickler:

The idea that XSLT should be a declarative language free of side-effects appears repeatedly in the early statements about the goals and design-principles of the language, but no-one ever seems to explain why: what should be the user benefits?

A function or procedure in a programming language is said to have side-effects if it makes changes to ist environment, for example if it can update a global variable that another function or procedure can read. If functions have side-effects, it becomes important to call them the right number of times and in the correct order. Functions that have no side-effects (pure functions) can be called any number of times and in any order.

It is possible to find hints at the reason why this was considered desirable in the statements that the language should be capable of **progressive rendering**. There is a concern that if you download a large XML-document, you wont be able to see anything on your screen until the last byte has been received. If a language has side-effects then the order of execution of the statements has to be defined,or the final result becomes unpredictable. Without side-effects, the statements can be executed in any order, which means it is possible to process parts of a stylesheet selectively and independently.

What being side-effect free mean in practice is that you cannot update a variable. This restriction is something you may find frustrating at first. But as you get the feel of the language and learn to think about using it the way it was designed to be used, rather then the way you are familiar with from other languages, you will find you stop thinking about this as a restriction.

XSLT is a rule-based language

The dominant feature of a XSLT stylesheet is that it consists of a sequence of template rules, each of which describes how a particular element type or other construct should be processed. The rules are not arranged in any particular order, the don’t have to match the order of the input or the output. It is this that makes XSLT a declarative language, because you specify what output should be produced when particular patterns occur in the input, as distinct from a procedural language, where you have to say what tasks to perform in what order.

The rule-based structure is very like CSS, but with the major difference that both the patterns (description of which node a rule applies) and the actions (description of what happens when the rule is matched) are mutch richer in functionality.

Also auf gut deutsch, es gibt einen nachvollziehbaren Aspekt im XSLT-Design, der die Verwendung von „echten“ Variablen unterbindet. Und es gibt selbstverständlich eine Hintertür dafür, um diese Einschränkung zu umgehen. Die sogenannten XSLT-Extensions bieten die Möglichkeit, ander Sprachen einzubinden, zum Beispiel Java (XSLT 1.0 konform, Saxon oder Xalan) oder JavaScript bzw VBScript (unterstützt von Microsofts MSXML).

Versuch 4: Verwendung einer JavaScript-Variablen in XSLT

Die Lösung unseres Problems ist die Verwendung einer echten Variablen, um ein gefundenes Kriterium aus einer sortierte Liste für den nachfolgenden Vergleich zwischenzuspeichern, also eine Funktion der folgenden Form:

```
var alt = "";
function keinVorgaenger(x) {
       if(x == alt)  return false;
       alt = x;
       return true;
}
```

Dafür verwenden wir die XSLT-Extension von Microsoft und binden eine JavaScript-Funktion in die Transformation ein, unser ursprüngliches Stylesheet ausw.xsl, das wir für die Dokumentauswertung erstellt haben, sieht demnach folgendermaßen aus:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:js="rosen:javascript">

<msxsl:script xmlns:msxsl="urn:schemas-microsoft-com:xslt"
   language="JavaScript" implements-prefix="js">
   var alt = "";
   function keinVorgaenger(x) {
      if(x == alt)  return false;
      alt = x;
      return true;
   }
</msxsl:script>

<!-- ========== Root template - start processing here ========== -->
<xsl:template match="/">
    ...
<!-- Make the Menu -->
 <xsl:variable name="ausw" select="//entry"/>
Zeit von<br />
   <select name="von" style="font-family:'Arial'; font-size:7pt">
   <option value="20010101" selected="y">Beginn</option>
   <xsl:for-each select="$ausw/@t">
      <xsl:sort select="."/>
      <xsl:if test="js:keinVorgaenger(string(.))">
         <option value="{.}"><xsl:value-of select="."/></option>
      </xsl:if>
   </xsl:for-each>
   </select>
<br />
```

Wir sortieren in einer for-each Schleife die zu prüfenden Datenelemente (gezeigt am Beispiel @t für den Zeiteintrag entry/time) und vergleichen mit der JavaScript-Funktion jeweils mit dem vorangegangenen Element dieser sortierten Menge. Der erste Vergleich findet mit einem in JavaScript initialisierten Leerstring statt.

Das Ergebnis ist überzeugend, in Sekundenschnelle wird das komplette Datendokument analysiert und das Auswahlmenü erzeugt. Wir ersparen uns den Umweg über eine sortierte Zwischendatei (_ausw.xml).

Beispiel 8

SAX-Parser und Document-Handler

Workshop Vorbereitung zu Beispiel 8:
SAX-Parser und Document-Handler

Werkzeuge:
MS Internet-Explorer 5.5 (SP2) oder 6
Einfacher Editor (Wordpad ...)
XMLSpy-Editor (www.xmlspy.com)
XML-Parser Xerxes und XSL-Prozessor Xalan (www.apache.org)
Java JRE 1.3.x
Java XML-Pack JAXP
Java ODBC (JDBC)

XML-Sprachfamilie:
XSLT-Elements

XPath-Expressions

XSLT-Functions

XSD-Schema
    Regular-Expressions (Patterns)

Zielsprache:
Java, SQL, HTML

Beispiel 8: SAX-Parser und Document-Handler

Aufgabenstellung:

Entwurf und Realisierung einer XML-Datenschnittstelle. Datendokumente unterscheiden sich grundlegend von Textdokumenten. Dieses Beispiel soll alle Aspekte einer Datenübermittlung behandeln, vom Entwurf eines Datenformates, Verwendung eines SAX-Parsers zur Dokumentbehandlung bis zur Datenbank-Anbindung über Java-ODBC (JDBC).

Zielsetzung:

- Entwurf eines für Datenübermittlung optimalen Dokumentmodells (DTD)
- Feinspezifikation des Modells mittels Schema (XSD)
- Verwendung eines SAX-Parsers und Entwurf einer Documenthandler-Klasse
- Objektorientierte Datenorganisation (Interface-Pattern und Java-Collections)
- Verwendung der Java-ODBC API (JDBC)

Im konkreten Beispiel bestimmen zwei markante Eigenschaften der zu übermittelnden Daten die Architektur der Schnittstelle:

- variable Datenmenge von unter hunder KByte bis zu einigen Mbyte
- Zwei verschiedenen Datenszenarien in einem Dokument vereint (Summenwerte und Einzelwerte)

Die erste Eigenschaft, also die möglicherweise sehr große Datenmenge, betrifft sowohl die Modellierung des XML-Dokumentes wie auch die Wahl des XML-Parsers. Anstelle eines komfortablen DOM-Parsers wird ein einfacher SAX-Parser verwendet, der wesentlich weniger Speicherresourcen beansprucht. Die zweite Eigenschaft, die unterschiedliche Datenstruktur, wird besonders die Funktionalität des Dokumenthandlers beeinflußen, da zwei verschiedene Datenobjekte aus dem Dokumentfluß befüllt werden müssen.

Dokumentmodell (DTD)

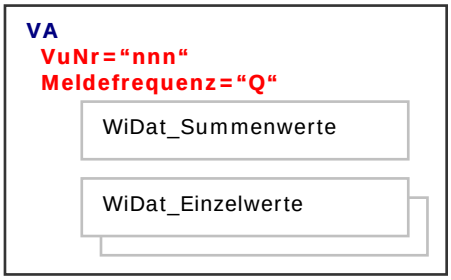
Für den prinzipiellen Entwurf des XML-Dokumentes verwenden wir eine DTD, also ein Werkzeug mit einfachem, aber logisch-prägnantem Beschreibungsvokabular:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!ELEMENT VA (Definitionen?, WiDat_Summenwerte?, WiDat_Einzelwerte*)>
<!ATTLIST VA
   VuNr CDATA #REQUIRED
   Meldefrequenz (J | M | Q | A) #REQUIRED
>
```

Das Wurzelelement VA enthält die für uns in erster Linie interessanten Datenstrukturen:

- WiDat_Summenwerte?
- WiDat_Einzelwerte*

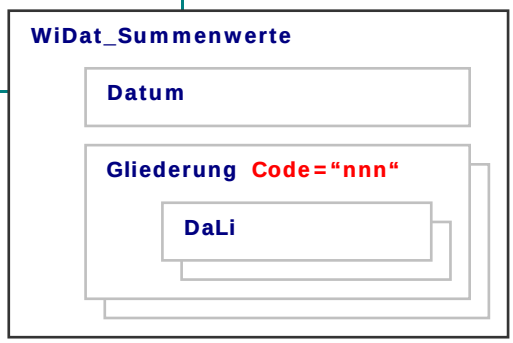
Beide Datenstrukturen sind optional, Summenwerte null bis einmal (?), Einzelwerte null bis viele mal (*).



Damit haben wir ein Vehikel, um kleine Datenmengen gemeinsam in einem Dokument zu transportieren, und um große Datenmengen auf mehrere Dokumente zu verteilen. Das zugrunde liegende Geschäftsmodel ist für uns nicht interessant, wir werden uns daher in weiterer Folge nur auf die wesentlichen Details beschränken.

Summenwerte-Modell

```
<!-- ++++ Wirtschaftsdaten ++++ -->
<!ELEMENT WiDat_Summenwerte (Datum, Gliederung+)>
<!ELEMENT Gliederung (DaLi+)>
<!ATTLIST Gliederung
   Code CDATA #REQUIRED
>
```



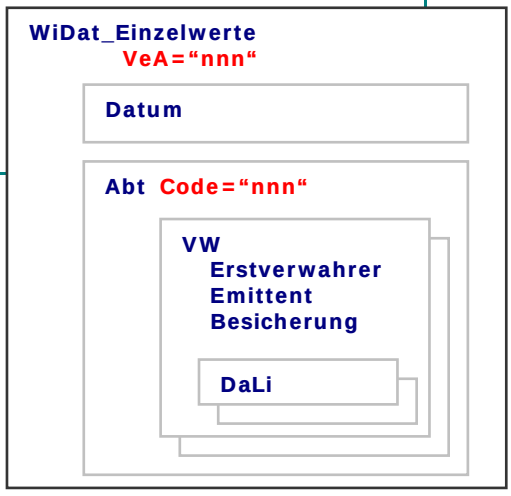
Aus technischer Sicht handelet es sich um eine doppelt verschachtelte Datenstruktur: ein bis viele Gliederung-Elemente enthalten jeweils ein bis viele Datenlisten (DaLi).

Jedes Gliederung-Element ist über ein Attribute (Code) eindeutig identifiziert.

Einzelwerte-Modell

Die zweite Datenstruktur WiDat_Einzelwerte ist sinngemäß in mehr Details aufgegliedert als Summenwerte, die für uns wichtigen Strukturdetails sind fett hervorgehoben:

```
<!-- ++++ Versicherungswerte ++++ -->
<!ELEMENT WiDat_Einzelwerte (Datum, Abt+)>
<!ATTLIST WiDat_Einzelwerte
   VeA (D|B|F|S) #REQUIRED
>
<!ELEMENT Abt (VW+)>
<!ATTLIST Abt
   Code CDATA #REQUIRED
>
<!ELEMENT VW (Erstverwahrer?, Emittent?, Besicherung*, DaLi+)>
<!ATTLIST VW
   Name CDATA #REQUIRED          Kennung CDATA #REQUIRED
   KennArt (W|I|S) #REQUIRED     AGR CDATA #IMPLIED
   Wg CDATA #REQUIRED            LaB CDATA #REQUIRED
   Komplex CDATA #IMPLIED        GZ CDATA #IMPLIED
>
<!ELEMENT Erstverwahrer EMPTY>
<!ATTLIST Erstverwahrer
   Name CDATA #REQUIRED          La CDATA #IMPLIED
>
<!ELEMENT Emittent EMPTY>
<!ATTLIST Emittent
   Name CDATA #REQUIRED          IdentNr CDATA #REQUIRED
   IdentNrArt (A|I) #REQUIRED    La CDATA #IMPLIED
>
<!ELEMENT Besicherung EMPTY>
<!ATTLIST Besicherung
   Name CDATA #REQUIRED
   Art CDATA #REQUIRED
   La CDATA #IMPLIED
>
```



Es ist ersichtlich, das die Daten praktisch nur über Attribute transportiert werden, entweder direkt im Element VW, oder mit leeren Sub-Elementen. Das ist eine Konzession an die möglicherweise große Datenmenge, wir versuchen Markup einzusparen, indem wir Attribute statt Elemente verwenden.

Datenliste Modell (DaLi)
In beiden Datenstrukturen (Summen- und Einzelwerte) kann als innerstes Element eine Datenliste (DaLi) beliebig oft vorkommen. Hier ist also eine besonders sorgfältige Modellierung erforderlich, um eine optimale Dokumentstruktur zu erzielen.

```
<!ELEMENT DaLi (UGL*)>
<!ATTLIST DaLi
   Code CDATA #REQUIRED
   Gliederung CDATA #IMPLIED
   W CDATA #IMPLIED>
```

Die Datenliste ist in zwei Strukturvarianten vorgesehen, entweder in Kurzform nur mit einem Code- und einem W-Attribut (für Wert), oder mit beliebig vielen Untergliedern (Subelementen UGL). Um den Markup-Anteil in diesem oft wiederholten Datenelement so minimal wie möglich zu halten, wird das deklarative Markup in die DTD ausgelagert, und im XML-Datendokument nur ein synonymes Kurz-Markup verwendet.

Deklaratives Markup mit FIXED-Attributen in der DTD

```
<!-- ++++ Modell ++++ -->

<!ELEMENT UGL EMPTY>
<!ATTLIST UGL
   W CDATA #IMPLIED
   GA (D | I | d | i) #IMPLIED
   La CDATA #IMPLIED
   LG CDATA #IMPLIED
   R CDATA #IMPLIED
   Wg CDATA #IMPLIED
   Z CDATA #IMPLIED
   ZG CDATA #IMPLIED
   P CDATA #IMPLIED
   VeA (D | B | F | S) #IMPLIED
   AGR CDATA #IMPLIED
   AA CDATA #IMPLIED>
```

```
<!-- ++++ Definitionen ++++ -->
<!ELEMENT Definitionen (dUGL)>
<!ELEMENT dUGL EMPTY>
<!ATTLIST dUGL
   W CDATA #FIXED "Wert"
   GA CDATA #FIXED "Geschäftsart"
   La CDATA #FIXED "Land"
   LG CDATA #FIXED "Ländergruppe"
   R CDATA #FIXED "Region"
   Wg CDATA #FIXED "Währung"
   Z CDATA #FIXED "Zweig"
   ZG CDATA #FIXED "Zweiggruppe"
   P CDATA #FIXED "Partner"
   VeA CDATA #FIXED "Vermögensart"
   AGR CDATA #FIXED "AGR"
   AA CDATA #FIXED "Auslandsaktivität">
```

Im XML-Dokument werden nur die Kurzsymbole transportiert, erst bei der Präsentation des Dokumentes werden vom XSLT-Stylesheet (vaviso1.xsl) die zugehörigen Langtexte verwendet. Damit dieser Mechanismus funktioniert, muß das XML-Dokument nur ein leeres Element (dUGL) enthalten, dessen Attribute in der DTD aufgelistet werden.

XML-Dokument Beispiel

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE VA SYSTEM "VAVISO1.DTD">
<?xml-stylesheet href="vaviso1.xsl" type="text/xsl"?>
<VA VuNr="151" Meldefrequenz="J">
   <Definitionen> <dUGL/> </Definitionen>
   <WiDat_Summenwerte>
      <Datum Art="Meldestichtag">
         <Jahr>2001</Jahr> <Monat>13</Monat>
      </Datum>
      <Gliederung Code="BA_1010">
         <DaLi Code="28">
            <UGL W="10000.00" GA="D" LG="EWR" ZG="S" VeA="B"/>
            <UGL W="20000.00" GA="I" LG="EWR" ZG="S" VeA="B"/>
         </DaLi>
         <DaLi Code="868" W="200000.00"/>
         <DaLi Code="1042" W="300000.00"/>
      </Gliederung>
   </WiDat_Summenwerte>
   <WiDat_Einzelwerte> ... </WiDat_Einzelwerte>
</VA>
```

Verweis auf deklaratives Markup

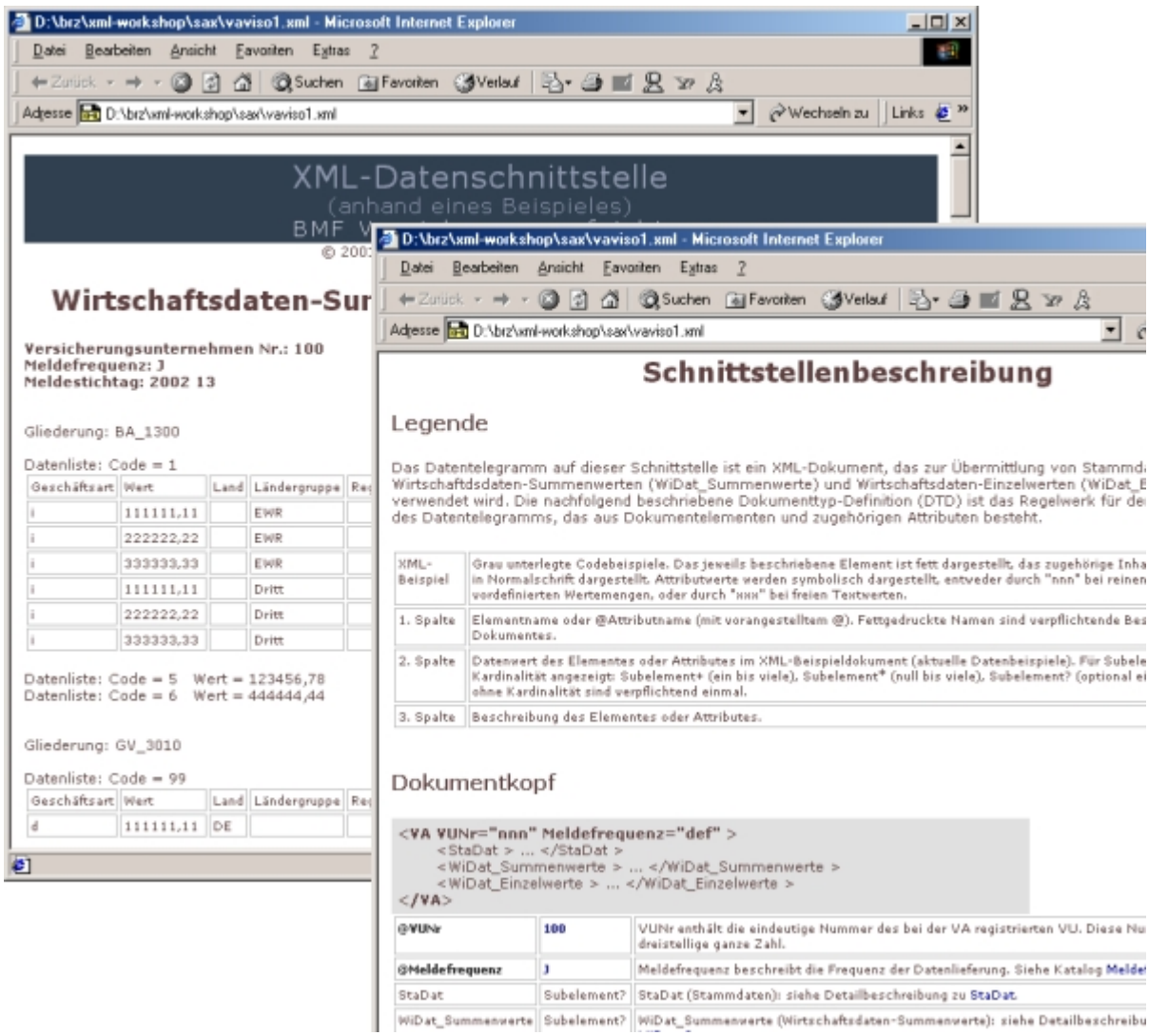
DaLi Strukturvarianten

Schnittstellendokumentation mit XSLT

Mit der Erstellung einer DTD ist die Datenschnittstelle bereits formal dokumentiert. Die DTD kann dem Anwender übergeben werden, und wenn er sich daran hält, ist alles in Ordnung. Leider umfaßt die DTD nur den technischen Aspekt der Anwendung, sie beschreibt nicht das Geschäftsmodell bzw. die Logik der einzelnen Dateninhalte.

Dieses Wissen wird zwar beim fachkundigen Anwender vorausgesetzt, er muß ja die Daten erbringen, aber wie vielfältig Datenformate interpretiert werden können, wissen wir alle aus der Praxis. Allein nur das Datumsformat besitzt viele Varianten.

Für XML sind alle Daten reine Texte, daher haben wir ein sehr tolerantes Datenvehikel. Wir könnten daher mit dem Anwender eine Datenformat-Konvention aushandeln, die wir ihm in schriftlicher Form zur Verfügung stellen. Damit diese Dokumentation immer auf dem letzten Stand ist, integrieren wir sie einfach in die Anwendung, in Form eines Stylesheets (vaviso1.xsl), das wir sowieso für die Anzeige der Datendokumente erstellen.



Eine andere Alternative wäre, das Datenformat bis ins Detail zu spezifizieren. Die XML-Familie bietet dafür ein passendes Werkzeug: XSD Schema.

Feinspezifikation des Modells mittels Schema (XSD)

Eine vorhandene DTD läßt sich jederzeit in ein Schema umwandeln (umgekehrt funktioniert es nicht). Wir verwenden dafür den XMLSpy-Editor. Unter dem Menüpunkt DTD/Schema finden wir die entsprechende Convert-Funktion. Das automatisch erzeugte Schema speichern wir unter vaviso1.xsd ab und können nun unserem XML-Dokument (vaviso1.xml) dieses Schema sofort zuweisen (Menü DTD/Schema - Assign Schema). Unser XML-Dokument besitzt nun folgenden Prolog:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- DOCTYPE VA SYSTEM "VAVISO1.DTD" -->
<?xml-stylesheet href="vaviso1.xsl" type="text/xsl"?>
<VA VuNr="998" Meldefrequenz="J"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="vaviso1.xsd">
```

Namensräume werden von Schemas unterstützt, das ist einer der Vorteile gegenüber DTDs. Für anspruchsvolle XML-Projekte ist der Schutz des Vokabulars durch Verwendung von Namensräumen sinnvoll, allerdings sind davon dann alle beteiligten Prozesse betroffen: XSLT-Transformation, Parser-Instanzierung und natürlich die Erstellung der XML-Dokumente (Authoring).

Die eingehende Behandlung von Schemas ist im Rahmen dieses Beispiels nicht möglich, wir beschränken uns auf das Detail der Element-Feinspezifikation. Dazu verwenden wir das Datum-Element, das in unserer DTD mit freiem Textinhalt (#PCDATA) definiert war:

```
<!ELEMENT Datum (Jahr, Monat, Tag?)>
<!ATTLIST Datum
   Art (Meldestichtag | andere | Ende_des_Meldezeitraumes) "Meldestichtag"
>
<!ELEMENT Jahr (#PCDATA)>
<!ELEMENT Monat (#PCDATA)>
<!ELEMENT Tag (#PCDATA)>
```

Das entsprechende Schema-Modell:

```
<xs:element name="Datum">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="Jahr"/>
         <xs:element ref="Monat"/>
         <xs:element ref="Tag" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="Art" default="Meldestichtag">
         <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
               <xs:enumeration value="Meldestichtag"/>
               <xs:enumeration value="andere"/>
               <xs:enumeration value="Ende_des_Meldezeitraumes"/>
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
   </xs:complexType>
</xs:element>
```

Für uns interessant sind Jahr, Monat und Tag. Für diese Elemente wollen wir den zulässigen Inhalt eindeutig formulieren:

Jahr:	2001 bis 2099
Monat:	01 bis 12
Tag:	01 bis 31

Datumsformat mit Patterns definieren

```
<xs:element name="Jahr">
   <xs:simpleType>
      <xs:restriction base="xs:gYear">
         <xs:minInclusive value="2001"/>
         <xs:maxInclusive value="2099"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>

<xs:element name="Monat">
   <xs:complexType>
      <xs:simpleContent>
         <xs:restriction base="xs:string">
            <xs:pattern value="0[1-9]|1[0-2]"/>
         </xs:restriction>
      </xs:simpleContent>
   </xs:complexType>
</xs:element>

<xs:element name="Tag">
   <xs:complexType>
      <xs:simpleContent>
         <xs:restriction base="xs:string">
            <xs:pattern value="0[1-9]|[1-2][0-9]|3[0-1]"/>
         </xs:restriction>
      </xs:simpleContent>
   </xs:complexType>
</xs:element>
```

Das Jahr läßt sich elegant durch den vorgesehenen Schema-Datentyp `xs:gYear` definieren, als zusätzliche Bedingung muß nur mehr der erlaubte Wertebereich durch `minInclusive` und `maxInclusive` eingeschränkt werden.

Monat und Tag betrachten wir wegen der führenden Null als String und entwerfen jeweils ein passendes Bedingungs-Pattern:

Monat: `0[1-9]|1[0-2]`
Wenn die erste Ziffer 0 ist, ist als zweite Ziffer 1-9 erlaubt, oder wenn die erste Ziffer 1 ist, ist 0-2 als zweite Ziffer erlaubt

Tag: `0[1-9]|[1-2][0-9]|3[0-1]`
Wenn die erste Ziffer 0, ist, ist als zweite Ziffer 1-9 erlaubt, oder wenn die erste Ziffer 1 oder 2 ist, ist 0-9 als zweite Ziffer erlaubt, oder wenn die erste Ziffer 3 ist, ist 0-1 als zweite Ziffer erlaubt

Mit diesen sogenannten Regular-Expressions lassen sich Dateninhalte genau spezifizieren, und validierende Parser können somit Datendokumente auch auf inhaltliche Kriterien prüfen.

SAX-Parser

Für unserer Java-Experiment brauchen wir die XML-Klassenbibliothek, die wir bei Sun (java.sun.com) unter dem Stichwort XML finden. Konkret brauchen wir das JAXP-Paket, das wir im ClassPath verfügbar machen müssen (siehe dazu die Datei start.bat).

Schritt1: SAX-Parser Instanzierung

Wir erstellen eine Java-Klasse mit dem Namen SaxTest1.java:

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class SaxTest1 {

  public static void main(String[] args) throws Exception {
    SAXParserFactory factory = SAXParserFactory.newInstance();
    //factory.setValidating(true);
    SAXParser parser = factory.newSAXParser();
    if (args[0] != "") {
      InputSource is = new InputSource(new FileReader(args[0]));
      //parser.parse(is, new DocumentHandler());
   }
    System.exit(0);
  }
}
```

Die Instanzierung eines SAX-Parsers erfolgt über eine Factory mit der Methode `factory.newSAXParser`. Die Factory selbst wird aus der SAXParser-Klasse mit `newInstance()` erzeugt. Weiters erzeugen wir uns eine InputSource für die spätere Übergabe des XML-Dokumentes an den SAX-Parser. Diese InputSource `is` wird über einen FileReader mit dem XML-Dokument befüllt. Der FileReader selbst erhält als Argument den Dateinamen des XML-Dokumentes, den wir als Kommandozeilen-Parameter (`args[0]`) übergeben haben.

Dieses kleine Programm compilieren wir mit der Kommandodatei start.bat, die den entsprechen Compiler-Aufruf und die ClassPath-Definition enthält. Damit stellen wir sicher, dass unsere Java-Umgebung für die weiteren Experimente funktioniert.

Schritt2: Einfacher Documenthandler

Der SAX-Parser erzeugt beim Bearbeiten des XML-Dokumentes jedesmal ein Ereignis (Event), wenn ein Dokumentteil aus dem Datenstrom verfügbar ist. Diese Events müssen von einem DocumentHandler verarbeitet werden. Wir erweitern daher unsere Java-Klasse zu einem DocumentHandler, indem wir von diesem Interface erben:

```
public class SaxTest2 extends DefaultHandler {

   public void startDocument() throws SAXException {  }
   public void endDocument() throws SAXException {  }
   public void setDocumentLocator(Locator locator) {  }
   public void characters(char[] ch, int start, int length)
      throws SAXException {  }
   public void startElement(String namespace, String localName,
                String qName, Attributes atts) throws SAXException {  }
   public void endElement(String namespace, String localName, String qName)
                throws SAXException {  }

   public static void main(String[] args) throws Exception {
     //wie SaxTest1
   }
}
```

Diese neue Klasse speichern wir unter dem Namen SaxTest2.java. Um damit überhaupt etwas Sinnvolles machen zu können, müssen wir die Methoden im DocumentHandler mit entsprechendem Funktionen befüllen:

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class SaxTest2 extends DefaultHandler {

  private String elementName = "";
  private Locator locator = null;

public void setDocumentLocator(Locator locator) {
    this.locator = locator;
  }
```

Wir deklarieren zwei Variablen elementName und locator und setzten den Document-Locator in der entsprechenden Methode. Wir selbst brauchen ihn nicht, aber das Interface zwingt uns zu diesem Formalismus.

```
public void startDocument() throws SAXException {
    System.out.println("XML-Parser Version 1.0 START");
  }

  public void endDocument() throws SAXException {
    System.out.println("END XML-Document");
  }
```

Die Methoden startDocument und endDocument benutzen wir vorerst nur, um diese Ereignisse mit System.out.println im Kommandofenster anzuzeigen. Wenn vom SAX-Parser Text erkannt wird, so ruft er die Methode characters() auf und übergibt ihr die Textzeichen in einem Character-Array. Dieses Array wollen wir ebenfalls im Kommandofenster anzeigen:

```
  public void characters(char[] ch, int start, int length)
    throws SAXException {
    String charString = new String(ch, start, length);
    System.out.println(charString);
  }
```

Jetzt fehlen nur mehr die Methoden für startElement und endElement, auch hier wollen wir die entsprechenden Ereignisse, nämlich die Element-Namen, im Kommandofenster anzeigen:

```
  public void startElement(String namespace, String localName,
    String qName, Attributes atts) throws SAXException {
    elementName = qName;
    System.out.print(elementName + ": ");
    for (int i=0; i < atts.getLength(); i++) {
       System.out.print(atts.getQName(i) + " " + atts.getValue(i) + "  ");
   }
    System.out.println();
  }

  public void endElement(String namespace, String localName, String qName)
    throws SAXException {
  }
```

Damit haben wir einen lauffähigen DocumentHandler, den wir compilieren und testen können.

Documenthandler

Jetzt beginnt der anspruchsvolle Teil unseres Beispiels. Wir müssen aus dem Datenstrom des SAX-Parsers ein Datenmodell aufbauen. Der wesentlich komfortablere DOM-Parser würde diese Aufgabe für uns erledigen, aber den können wir wegen der zu erwartenden großen Datenmenge nicht verwenden. Unser Problem läßt sich wie folgt skizzieren:

- Zwei grundverschiedene Datenstrukturen in einem XML-Dokument erfordern zwei verschiedene Datenmodelle, im konkreten Fall Summen- und Einzelwerte. Wir werden sie als Java-Klassen bereitstellen. Die bewährte Designregel „Low Coupling“ werden wir dabei strikt einhalten und für dies Klassen ein Interface vorsehen, um die Implementierung des Datenmodells zu verbergen.
- Die hierarchischen Datenstrukturen des XML-Dokumentes lassen sich nur mühsam aus dem serialisierten Datenstrom rekonstruieren. Hier hilft uns das „Expert“-Designpattern, wir werden es sinngemäß anwenden.

Designregel „Low-Coupling“ für die Datenmodelle

Die Designregel „Low-Coupling“ realisieren wir mit zwei Strategien: Durch Verwendung eines Interface wird nicht nur die Schnittstelle zu den Datenklassen festgelegt, sondern überhaupt die Existenz dieser Klassen verborgen. Der DocumentHandler sieht nur das Interface, mit dem er kommuniziert.

Interface VaWerte

```
public interface VaWerte {
  void initValues(String theName);
  void setValue(String theName, String value);
  StringBuffer getValues();
}
```

DocumentHandler

```
public class SaxTest2 extends DefaultHandler {
  ...
  private VaWerte vaWerte;
  private Summenwerte summenwerte = new Summenwerte();
  private Einzelwerte einzelwerte = new Einzelwerte();

public void startElement(String namespace, String localName,
  String qName, Attributes atts) throws SAXException {
  elementName = qName;
  if(qName.equals("WiDat_Summenwerte")) {
    vaWerte = summenwerte;
    vaWerte.initValues("");
  }
  else if(qName.equals("WiDat_Einzelwerte")) {
    vaWerte = einzelwerte;
    vaWerte.initValues("");
    for (int i=0; i < atts.getLength(); i++) {
      einzelwerte.setValue(elementName + atts.getQName(i), atts.getValue(i));
    }
  }
  else {
    for (int i=0; i < atts.getLength(); i++) {
      vaWerte.setValue(elementName + atts.getQName(i), atts.getValue(i));
    }
  }
}
```

Wenn im Datenstrom der Beginn der Summen- oder Einzelwerte erkannt wird, erhält die Variable vaWerte (vom Typ Interface) die Referenz auf das jeweilige Datenobjekt. Alle nachfolgenden Methodenaufrufe (initValues, setValues) werden somit gezielt auf das richtige Datenobjekt angewendet. Damit wäre das Prinzip „Low-Coupling“ bereits erfüllt, wir wollen (und müssen) aber noch eine weitere Strategie anwenden.

Unsere Datenobjekte sollen selbst entscheiden, welche Daten sie verwerten. Sie repräsentieren ja das Datenmodell und sind daher Experten für diese Aufgabe.

„Expert“-Designpattern für das Datenmodell

```
import java.util.*;
import java.text.*;

public class Summenwerte implements VaWerte {

  private String nul = "~";
  private HashMap colName = new HashMap(50);
  private ArrayList rowValues = new ArrayList();

  public Summenwerte() {  // Constructor
    colName.put(new String("timestamp"), new Integer(0));
    colName.put(new String("Status"), new Integer(1));
    colName.put(new String("VAVuNr"), new Integer(2));
    colName.put(new String("VAMeldefrequenz"), new Integer(3));
    colName.put(new String("Jahr"), new Integer(4));
    colName.put(new String("Monat"), new Integer(5));
    colName.put(new String("GliederungCode"), new Integer(6));
    colName.put(new String("DaLiCode"), new Integer(7));
    colName.put(new String("UGLGA"), new Integer(8));
    colName.put(new String("UGLLa"), new Integer(9));

    for(int i=0; i < colName.size(); i++) rowValues.add(nul);
}
```

Für die Lösung dieser Aufgabe verwenden wir zwei Collection-Typen aus der Java util-Bibliothek, eine ungeordnete HashMap und eine geordnete ArrayList. Die HashMap bildet das Eingangsfilter unseres Datenmodells, sie akzeptiert nur Dokument-Elemente, die als Hash-Keys eingetragen sind. Und diese Keys legen wir im Constructor der Klasse fest, damit definieren wir unser Datenmodell.

Als zugehöriges Objekt legen wir für jeden Hash-Key einen Indexverweis auf einen ArrayList-Eintrag an. Damit mappen wir die „freie“ Reihenfolge der Dokument-Elemente auf eine geordnete Reihenfolge von Insert-Values für die Datenbank. Damit ersparen wir uns später, in den SQL-Insert-Statements Namen angeben zu müssen. Der Document-Handler übergibt an die setValue-Methode der Summenwert-Klasse den Element- und Attributnamen als Keyword und den zugehörigen Datenwert. Paßt das Keyword zu den Hash-Keys, so wird aus der HashMap der Indexwert geholt, und die Daten in der ArrayList abgelegt.

```
public void setValue(String theName, String value) {
  int i;
  if(colName.containsKey(theName)) {
    i = ((Integer)colName.get(theName)).intValue();
    rowValues.set(i, (Object)value);
  }
}
```

Paßt das Keyword nicht, so werden die angebotenen Daten von der Summenwert-Klasse ignoriert. Eigentlich sollten ja keine falschen Daten angeboten werden, da über das Interface sowieso die richtige Klasse adressiert wird, aber wir haben in beiden Klassen gemeinsame Daten, zum Beispiel Datum und VuNr. Daher werden wir alle Daten jeweils beiden Klassen anbieten, sie sind ja jetzt Experten.

```
for (int i=0; i < atts.getLength(); i++) {
   //vaWerte.setValue(elementName + atts.getQName(i), atts.getValue(i));
   summenwerte.setValue(elementName + atts.getQName(i), atts.getValue(i));
   einzelwerte.setValue(elementName + atts.getQName(i), atts.getValue(i));
}
```

„Expert“-Designpattern für die Rekonstruktion der Datenstrukur
Der Experte für die Rekonstruktion der Datenstruktur ist der DocumentHandler selbst. Er wird daher die Aufgabe übernehmen, fertige Datensätze von den Datenklassen anzufordern, und die Datenklassen für neue Daten vorzubereiten (zu initialisieren).

```
public void endElement(String namespace, String localName, String qName)
    throws SAXException {
    if(vaWerte == summenwerte) endSummenwerte(qName);
    else if(vaWerte == einzelwerte) endEinzelwerte(qName);
  }

  public void endSummenwerte(String qName)
    throws SAXException {
    if((qName.equals("DaLi") && elementName.equals("DaLi")) ||
      (qName.equals("UGL") && elementName.equals("UGL"))) {
        StringBuffer sb = vaWerte.getValues();
        System.out.println(sb);
    }
    if(qName.equals("Gliederung") || qName.equals("DaLi") || qName.equals("UGL"))
      vaWerte.initValues(qName);
  }

  public void endEinzelwerte(String qName)
    throws SAXException {
    if(qName.equals("VW")) {
        StringBuffer sb = vaWerte.getValues();
        System.out.println(sb);
    }
    if(qName.equals("VW") || qName.equals("Abt"))
      vaWerte.initValues(qName);
  }
```

Die entsprechenden Methoden in der Datenklasse

```
public void initValues(String theName) {
    int begin = 6; //default startindex
    if(theName.equals("DaLi")) begin = 7;
    if(theName.equals("UGL")) begin = 8;
    for(int i=begin; i < colName.size(); i++)
      rowValues.set(i, nul);
    rowValues.set(0, timestamp);
  }

public StringBuffer getValues() {
    StringBuffer sb = new StringBuffer("insert into journalsummenwerte values (");
    for(int i=0; i < colName.size()-1; i++)
      sb.append("'" + rowValues.get(i) + "', ");
    sb.replace(sb.length()-2, sb.length()-1, ") ");
    return sb;
  }
```

Datenbank-Anbindung

Die Datenbankanbindung wird nur zur Komplettierung des Beispiels gezeigt. Da es eine reine Java-Anwendung ist, zählt sie nicht zum engeren Kursthema.

```
import java.io.*;
import java.sql.*;

public class JdbcHandler {

  private Connection con;
  private Statement statement;

  public JdbcHandler(String[] args)
  throws SQLException, IOException, ClassNotFoundException
  {
    super();
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); //Access und Oracle
    //Class.forName("oracle.jdbc.driver.OracleDriver"); //Oracle Server
    String dsURL = "jdbc:odbc:"; //Access arg[1]=vaviso1
    con = DriverManager.getConnection("jdbc:odbc:" + args[1]);
    con.setAutoCommit(false);
      statement = con.createStatement();
  }

  public void makeCommit() throws SQLException  {
    con.commit();
  }

  public void insertData(String s) throws SQLException  {
    statement.executeUpdate(s);
  }

  public void closeConnection() throws SQLException {
    statement.close();
    con.close();
  }

}
```

Der entsprechende Methodenaufruf im DocumentHandler

```
private JdbcHandler jh;
...
public void endSummenwerte(String qName)
   throws SAXException {
  if((qName.equals("DaLi") && elementName.equals("DaLi")) ||
    (qName.equals("UGL") && elementName.equals("UGL"))) {
      StringBuffer sb = vaWerte.getValues();
      //System.out.println(sb);
      try {
        if(sb != null) jh.insertData(sb.toString());
      }
      catch (SQLException sqle) {
            System.out.println("SQLException in insertData");}
  }
}
```