INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Ludwig
Maximilians–
Universität
München

LMU

Präsentation von XML-Daten – ein generischer Ansatz zur Layout-Spezifikation

Ulf Müller-Heinemann

Diplomarbeit

Beginn der Arbeit: 04.02.2002
Abgabe der Arbeit: 31.03.2002
Betreuer: Prof. Dr. François Bry
Dr. Norbert Eisinger

Erklärung

Hiermit versichere ich, daß ich diese Diplomarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 31.3.2002 Ulf Müller-Heinemann

Zusammenfassung

In dieser Arbeit wurde ein Prozeß entwickelt, der XML-Daten – mit dem konkreten Anwendungsfall der Verwaltungsdaten einer Universitätseinrichtung – in Präsentationen unterschiedlicher Beschreibungssprachen, Layouts und Landessprachen transformiert, wobei großer Wert darauf gelegt wurde, daß der Prozeß leicht an sich verändernde Rahmenbedingungen und Aufgabenstellungen angepaßt werden kann.

Im Zuge dessen wurde eine neue Abgrenzung der Begriffe „Daten“, „Layout“ und „Style“ vorgenommen und ein Ansatz entwickelt, mit dem das Layout von Präsentationen generisch spezifiziert werden kann und die Semantik der zugrundeliegenden Daten nach der Erstellung des Layouts erhalten bleibt.

Um die Tragfähigkeit des entwickelten Prozesses zu prüfen, wurde anschließend eine prototypische Implementierung vorgenommen, die darüberhinaus untersuchen sollte, inwieweit bestehende Werkzeuge und Transformationssprachen vorhanden und für eine Implementierung geeignet sind.

Abstract

The aim of this thesis was to create a process that transforms XML data into presentations of different description languages, layouts and spoken languages – with focus on the data used in the administration of university facilities. As the requirements to process change frequently, the possibility to easily adapt the process was especially emphasized.

In the course of building this process, the meanings of the terms „data“, „layout“ and „style“ were newly revised. An approach was developed in a way that allows to give a generic specification of a presentation's layout and preserves the semantics of the data after building the layout.

In order to prove the usability of the process a prototype was implemented. In addition to that existing transformation tools and languages were investigated and compared.

Inhaltsverzeichnis

1. Einleitung **5**

1.1. Ausgangspunkt 5

1.2. Ziele 6

1.3. Abgrenzung zu bestehenden Ansätzen 6

1.3.1. XSL Formatting Objects 6

1.3.2. DocBook 8

1.3.3. Vergleich 8

1.4. Aufbau 8

2. Das Datenmodell der Lehrangebots-Verwaltungsdaten – Besonderheiten **11**

2.1. Modellierungskonzepte 11

2.1.1. Abstraktion 11

2.1.2. Rekursion 12

2.1.3. Vererbung/Verschattung 13

2.2. Weitere Konzepte 14

2.2.1. Mehrere Bezeichnungen, Vielsprachigkeit 14

2.2.2. Strukturbaum der möglichen Lehreinheiten 14

2.2.3. Metadaten für die Datenbearbeitung 15

2.3. Definition von Begriffen zur Zeit 15

3. Präsentations-Erzeugungsprozeß **17**

3.1. Überblick 17

3.1.1. Datenquelle 18

3.1.2. Präsentation 18

3.1.3. Prozessoren 18

3.1.4. Prozeßschritte 19

3.1.5. Prozeßabschnitte 19

3.1.6. Gesamter Prozeß 22

3.2. Daten . . . 23
3.2.1. Datenschnittstelle . . . 23
3.2.2. Aufbereitung der Daten . . . 24
3.2.3. Sortierung und Transformation der Daten nach Layout-Klassen . . . 25
3.3. Layout . . . 27
3.3.1. Motivation des Layoutbaumes . . . 27
3.3.2. Generierung des Layoutbaums . . . 32
3.3.3. Aufbereitung des Layoutbaums . . . 32
3.3.4. Generische Ausdrücke . . . 34
3.4. Style . . . 36
3.4.1. Templatemechanismus . . . 36
3.4.2. Aufbereitung der Präsentation . . . 38
3.4.3. Transformation der Präsentation in die Zielsprache . . . 39

4. Rahmenbedingungen einer Implementierung 41
4.1. Erforderliche Sprachmittel . . . 41
4.1.1. Baumbasierte Transformationssprache . . . 41
4.1.2. Sprache zur Definition von Layoutbeschreibungsbäumen . . . 41
4.1.3. Sprache zur Kombination von Prozessoren . . . 42
4.1.4. Generische Ausdrücke . . . 42
4.1.5. Sprache zum Zugriff auf Inhalte der Präsentation in einem Template . . . 42
4.2. Erforderliche Werkzeuge . . . 43
4.2.1. Kombination von Prozessoren . . . 43
4.2.2. Prozessoren für Transformationen . . . 43
4.3. Verfügbare Sprachmittel und Werkzeuge . . . 44
4.3.1. Baumbasierte Transformationssprache . . . 44
4.3.2. Sprache zur Definition von Layoutbeschreibungsbäumen . . . 51
4.3.3. Sprache zur Kombination von Prozessoren . . . 51
4.3.4. Generische Ausdrücke . . . 51
4.3.5. Sprache zum Zugriff auf Inhalte der Präsentation in einem Template . . . 52

5. Prototypische Implementierung PEP 53
5.1. Implementierung des Prozesses . . . 55
5.1.1. Realisierung der Prozessoren . . . 55
5.1.2. Datenschnittstelle der Prozessoren . . . 56

5.1.3. Registrierung der Prozessoren 56
5.1.4. Realisierung des Prozesses 57
5.1.5. Vorteile 58
5.2. Bibliothek von generischen Ausdrücken 58
5.2.1. Generische Layoutkonstrukte 59
5.2.2. Generische Styleanweisungen 60
5.2.3. Generische Sonderzeichen 60
5.3. Implementierung ausgesuchter Prozessoren 61
5.3.1. Klassifikation der verwendeten Regelsprachen 61
5.3.2. Prozessoren mit mehreren Ausprägungen 68
5.3.3. Die implementierten Prozessoren 69
5.4. Vergleich der Werkzeuge am Beispiel des Prozessors zur Erzeugung des Layoutbaums 70
5.4.1. XSLT 70
5.4.2. fxt 75
5.4.3. XQuery 79
5.4.4. xcerpt 81

6. Zusammenfassung und Ausblick **83**

Anhang **85**
Abbildungsverzeichnis 85
Literaturverzeichnis 87

1. Einleitung

Für die Administration der Verwaltungsdaten des Lehrangebots am Institut für Informatik der LMU wird zur Zeit eine Software eingesetzt, die vor einigen Jahren am selben Institut entstanden ist. Allerdings waren bei der Planung dieses Systems viele der – teils sehr speziellen – Anforderungen noch nicht bekannt. Sie wurden nach und nach in das System integriert; aufgrund von bereits getätigten, grundsätzlichen Designentscheidungen konnten die Änderungen zum Teil allerdings nur in Form von Workarounds oder Sonderfall-Behandlungen vorgenommen werden.

Diese Arbeit beschäftigt sich nun mit einer Neukonzeption dieses Systems, die natürlich die bisher hinzugekommenen Anforderungen beinhaltet, aber daneben auch so offen sein soll, daß zukünftige Änderungen leicht zu integrieren sind.

1.1. Ausgangspunkt

Die vorliegende Arbeit stützt sich auf die Ergebnisse einer vorangegangenen Projektarbeit [20], deren Ziel es war, die Prozesse und Daten zu formalisieren, die bei der Verwaltung des Lehrangebots anfallen. Ausgehend von den spezifischen Anforderungen einer zukünftigen Implementierung wurde als Formalismus für die Beschreibung der Daten XML gewählt[1].

Im Rahmen der Projektarbeit entstanden ein Datenmodell für die Verwaltungsdaten in der Beschreibungssprache XML-Schema sowie detaillierte textuelle Beschreibungen der Prozesse in der Verwaltung. Die Beschreibung der Prozesse beginnt mit der Beschaffung der Verwaltungsdaten, geht über die Bearbeitung der Daten, wie z.B. die Generierung des Zeitplans der Veranstaltungen und die Beantragung dafür nötiger Räume, und endet mit der Erstellung einer Vielzahl von Präsentationen wie z.B. Vorlesungsverzeichnissen.

Diese Arbeit nun beschäftigt sich mit dem Teilbereich der *Erstellung der Präsentationen*. Die Bereiche der Datenerhebung, -erfassung und -bearbeitung werden hier außen vor gelassen; die Erfassung erfolgt zur Zeit rein manuell. Eine programmunterstützte Eingabe von Daten, z.B. über Eingabemasken mit einem Vorschlag- und Auswahlsystem der Werte, könnte ein Thema für eine spätere Arbeit sein. Ebenso eine Neukonzeption des Systems zum Entwurf eines Zeitplans.

[1] Die Hauptgründe, die zur Verwendung von XML geführt haben, waren die Notwendigkeit, die Daten textuell bearbeiten zu können (unter anderem wegen einer rationelleren Erfassung, z.B. durch Verwendung von *Copy and Paste*) und die Möglichkeit, bei der Erfassung der Daten bewußte Abweichungen vom Datenmodell (wie z.B. das Weglassen von noch nicht bekannten Elementen) in Kauf nehmen zu können.

1.2. Ziele

Mit dieser Arbeit soll ein Prozeß entwickelt werden, der XML-Daten – hier mit dem konkreten Anwendungsfall der Verwaltungsdaten – in Präsentationen unterschiedlicher

- Beschreibungssprachen (LaTeX, HTML, Text etc.),
- Layouts (z.B. eine „Stundenplanansicht“ und eine „Listenansicht“) und
- Landessprachen (Deutsch, Englisch etc.)

transformiert. Von großer Bedeutung ist dabei, daß der Prozeß leicht an sich verändernde Rahmenbedingungen und Aufgabenstellungen angepaßt werden kann.

Ein wichtiger Augenmerk liegt auf einer strikten Trennung von Datenzugriff, Erzeugung des Layout und des Style.

Die nötige Trennung von Inhalt und Style ist eine Erkenntnis, die sich mittlerweile allgemein durchgesetzt hat, egal ob im DTP- oder im Web-Publishing-Bereich. Deutlich sichtbar ist sie z.B. in der Entwicklung von HTML, das in seinen frühen Versionen beides undifferenziert nebeneinander benutzte. In seiner gegenwärtigen Version HTML 4.01 nimmt es durchaus eine Trennung von Inhalt (das Markup) und Style (Stylesheets) vor [29, 30, 34, 36].

Auch hat sich die Erkenntnis etabliert, daß es von Vorteil ist, Inhalte semantisch zu beschreiben, so daß das Ausgabemedium oder überhaupt der Ausgabezweck nicht bekannt sein muß; was unter anderem zur Einführung von XML geführt hat. Über Transformationen, die wieder über sogenannte Stylesheets [42] spezifiziert werden, sollen die Daten in beliebige Zielformate transformiert werden können.

Allerdings wird dabei unter Style sowohl das Erscheinungsbild der Inhalte als auch deren räumliche Anordnung verstanden. Diese Arbeit verfeinert das etablierte Verständnis des Stylebegriffes in sofern, daß sie eine Trennung dieser beiden Gesichtspunkte vornimmt: in Layout, das die räumliche Anordnung, und Style, der das Erscheinungsbild beschreibt.

Ein weiterer Augenmerk liegt auf einer sauberen Definition des Layouts, die anstelle einer imperativen ad-hoc-Lösung, bei der das Layout z.B. durch Druckanweisungen entsteht, eine deskriptive Beschreibung verwendet, aus der das Layout mit Hilfe von möglichst deklarativ formulierten Regeln aufgebaut wird.

1.3. Abgrenzung zu bestehenden Ansätzen

Um die Inhalte von Präsentationen zu beschreiben gibt es mittlerweile eine Reihe von Ansätzen. Die wohl bekanntesten und verbreitetsten sind XSL Formatting Objects [41] und DocBook [21].

1.3.1. XSL Formatting Objects

XSL-FO beschreibt die „physischen“ Bestandteile einer Präsentation, also Seiten, Boxen, Tabellen usw., bis hinunter zu den einzelnen Zeichen.

Diese Bestandteile werden in XSL-FO als *formatting objects* bezeichnet (daher auch der Name). Der Formalismus für die Beschreibung ist XML. XSL-FO definiert in dem Namensraum `http://www.w3.org/1999/XSL/Format` eine Reihe von XML-Elementen für die *formatting objects*. Anhand von *formatting properties* können die *formatting objects* an die Bedürfnisse der Präsentation angepaßt werden (Farben, Abstände, Ränder etc.).

Die *formatting objects* werden in folgende Gruppen eingeteilt:

1. *Declaration and Pagination and Layout Formatting Objects:* definieren Seitenvorlagen und Vorlagen für Seitenfolgen, z.B. eine Vorlage für Standardseiten und zusätzliche Vorlagen für die erste und die letzte Seite.

2. *Block Formatting Objects:* formatieren Textblöcke, wie z.B. Absätze, Überschriften oder Bildunterschriften.

3. *Inline Formatting Objects:* formatieren oder generieren einzelne Textbestandteile, wie z.B. Formatierung der erste Zeile eines Absatzes, Generieren des Textes "Seite " vor jeder Seitenzahl usw.

4. *Table Formatting Objects:* beschreiben Tabellen und ihre Bestandteile (z.B. Tabellenköpfe, -zeilen oder -zellen).

5. *List Formatting Objects:* beschreiben Listen und ihre Bestandteile (Items).

6. *Link and Multi Formatting Objects:* ermöglichen eine dynamische Anpassung der Darstellung und des Verhaltens von Teilen eines Dokuments, z.B. Verweise auf andere Dokumente, Wechsel zwischen verschiedenen Teilbäumen der *formatting objects* (z.B. zum Wechsel zwischen Kurz- und Langansicht einer Liste), Umschalten von Einstellungen (z.B. Schriftgröße) usw.

7. *Out-of-line Formatting Objects:* Objekte, die nicht unbedingt an der Position erscheinen, an der sie definiert wurden, z.B. Fußnoten und schwebende Objekte, wie Bilder.

8. *Other Formatting Objects:* dienen unter anderem dazu, andere Objekte zusammenzufassen oder Markierungen einzufügen und abzufragen.

Die Arbeitsweise von XSL-FO, um aus einem XML-Dokument eine Präsentation zu erstellen, ist grob geschildert wie folgend:

1. Das Quell-XML-Dokument wird mittels Transformationen in ein Dokument aus dem XSL-FO-Namensraum überführt. Diese Transformation wird als *tree transformation* bezeichnet.

2. Ein Baum, der die geometrischen Teilbereiche einer Präsentation beschreibt, der *area tree*, wird erzeugt. Dies erfolgt durch den *formatter*.

3. Mittels eines *renderers* kann der *area tree* auf dem jeweiligen Medium ausgegeben bzw. eine Präsentation in einer Zielsprache erzeugt werden.

1.3.2. DocBook

DocBook wurde mit der Absicht entworfen, Bücher und deren Bestandteile beschreiben zu können, wobei das Hauptaugenmerk auf Soft- und Hardwaredokumentationen liegt.

DocBook ist ein SGML-Dokumenttyp, in dem als Elemente verschiedene Präsentationstypen (Buch, Artikel, Unix-Man-Page und eine Zusammenstellung von Präsentationen) und deren Bestandteile definiert sind.

Die zu beschreibenden Bestandteile können Eigenschaften der Präsentationen (Autor, Titel, ...), ganze Gliederungseinheiten (Kapitel, Abschnitte, Inhaltsverzeichnisse, ...) oder diverse Layout-Konstrukte (Listen, Abbildungen, Absätze, ...) bis hin zu Inline-Elementen (Fußnoten, Verweise, ...) sein.

Mittlerweile gibt es auch eine XML-Version von DocBook [27].

Um Präsentationen zu erstellen, werden die Daten in der DocBook-Syntax formuliert und dann mit einem der vorhandenen DSSSL- [15], bzw. XSLT-Stylesheets [42] und einem entsprechenden Prozessor (z.B. Jade [10] für DSSSL, oder Saxon [16] für XSLT) in das Zielformat (LaTeX, HTML, XSL-FO u.a.) transformiert.

1.3.3. Vergleich

Stellt man den Ansatz, den XSL-FO vertritt, also die Beschreibung von physischen Seitenbestandteilen, dem hier zu entwickelnden Ansatz gegenüber, so drängt sich der Vergleich von einer Assemblersprache und einer höheren Programmiersprache auf. D.h. letztendlich müssen die Präsentationen, die mit dem hier zu entwickelnden Ansatz erstellt werden sollen, zwar physisch dargestellt werden, jedoch hängen die layoutbildenden Transformationen, die hier durchgeführt werden sollen, alle von der *Semantik* der darzustellenden Daten ab. Die Semantik der Daten kennt XSL-FO jedoch nicht.

Daraus folgt, daß XSL-FO zwar durchaus als ein mögliches Zielformat (neben LaTeX, HTML u.a.) angesehen werden kann, sich jedoch auf keinen Fall dazu eignet, den Prozeß zu beschreiben, mit dem das Layout der Präsentationen erzeugt wird.

Der Ansatz von DocBook stellt sehr generische Konstrukte für die Beschreibung von Präsentationen zur Verfügung und übernimmt die Entscheidung über die Art und Weise, wie diese dargestellt werden sollen, selbst. Einstellungen zur physischen Darstellung sind zwar möglich, entsprechen allerdings nicht unbedingt der Philosophie von DocBook. Hinzu kommt, daß die von DocBook intendierten Präsentationen Bücher und ähnliche Dokumentationen sind, nicht aber Stundenpläne, Veranstaltungslisten usw. mit feinerer Strukturierung.

DocBook könnte im Prinzip eine Anwendung des hier vorgestellten Ansatzes sein. DocBook beschreibt XML-Dokumente (bzw. SGML-Dokumente), die Bücher darstellen und Transformationen, um Layout und Style der Bücher zu bilden. Der hier vorgestellte Ansatz beschreibt generisch, wie Layout und Style für beliebige XML-Dokumente gebildet werden können.

1.4. Aufbau

In Kapitel 2 werden zunächst einige Besonderheiten des zugrundeliegenden Datenmodells erläutert; dies sind zum einen grundlegende Konzepte, die nicht nur für den vorliegenden Anwendungsfall von

Interesse sind (Abstraktion, Rekursion und Vererbung/Verschattung), und zum anderen Lösungen für etwas speziellere Probleme bei der Präsentation von Lehrangebotsdaten (Mehrsprachigkeit, unterschiedliche Texte für unterschiedliche Präsentationen).

Der Hauptteil der Arbeit liegt auf der Entwicklung des Präsentations-Erzeugungsprozesses in Kapitel 3, der aus den Daten, die in einem beliebigen Datenformat vorliegen (XML, Tabellen einer relationalen Datenbank etc.), Präsentationen in unterschiedlichen Beschreibungssprachen erstellt. Der Prozeß besteht aus drei Teilen: Im ersten werden die Daten – wenn nötig – in das XML-Format übertragen und nach bestimmten Gesichtspunkten aufbereitet, im zweiten wird das Layout der Präsentation aufgebaut und im dritten wird das Layout mit Styleinformationen angereichert und in eine Beschreibungssprache übersetzt.

Daraufhin wird in Kapitel 4 untersucht, welche unterschiedlichen Sprachmittel für eine Implementierung der Komponenten des Präsentations-Erzeugungsprozesses nötig sind und welche Sprachen und Werkzeuge diese Anforderungen erfüllen.

In Kapitel 5 wird die prototypische Implementierung PEP vorgestellt, mit der eine Implementierung eines vereinfachten, durchgängigen Präsentations-Erzeugungspozesses vorgenommen wurde. Ergänzend werden Implementierungsmöglichkeiten in unterschiedlichen Anfragesprachen verglichen.

In Kapitel 6 werden mögliche Anküpfungspunkte an diese Arbeit und zukünftige technische Entwicklungen, die für die hier gezeigten Ansätze interessant sein könnten, angesprochen.

2. Das Datenmodell der Lehrangebots-Verwaltungsdaten – Besonderheiten

2.1. Modellierungskonzepte

Bei der Modellierung wurden einige Konzepte eingesetzt, die unabhängig von der hier vorgestellen, speziellen Anwendung sind und als Lösung unterschiedlicher Probleme dienen. Dies sind die Konzepte Abstraktion, Rekursion und Vererbung/Verschattung. Sie werden im Folgenden kurz dargestellt.

2.1.1. Abstraktion

Auf die Vielzahl der Änderungen, die an dem bisherigen System zur Verwaltung der Lehrangebotsdaten nötig wurden, wurde ja in der Einleitung bereits hingewiesen. Dies betraf allerdings nicht nur Arbeitsabläufe, Präsentationen und dergleichen, sondern leider auch die zugrundeliegenden Datenstrukturierungen.

Ein Beispiel: Die Veranstaltung „Informatik 1" besaß lange Zeit eine zweimal in der Woche stattfindende Vorlesung und mehrere Übungen, die die Veranstaltungsteilnehmer jeweils alternativ besuchen konnten. Die pragmatische Lösung bei der Modellierung des ursprünglichen Datenmodells war, einen Datentyp „Veranstaltung" zu modellieren, der jeweils mehrere Komponenten vom Datentyp „Sitzung" und „Übung" enthalten kann. Was passiert aber – wie es tatsächlich passiert ist –, wenn im Rahmen der Veranstaltung nun noch eine „Rechnerübung" abgehalten wird? Als „Übung" kann sie nicht erfaßt werden, erstens soll sie in den Veranstaltungslisten gesondert erscheinen und nicht als Übungsgruppe und zweitens ist sie keine Alternative zu den anderen Übungen, sondern findet zusätzlich statt. Eine weitere pragmatische Lösung wäre, daß neben „Sitzung" und „Übung" noch ein Datentyp „Rechnerübung" hinzukommt. Und wenn nun noch eine „Seniorenübung" eingeführt wird?

Das Beispiel hat in diesem Fall zwar nur mögliche unterschiedliche Übungsarten herausgestellt, aber genauso könnten sich natürlich Veranstaltungen grundlegend unterscheiden, indem sie keine Übungen, sondern „Exkursionen" oder ähnliches enthalten sollen. In den letzten Jahren ist etwa alle zwei bis drei Semester eine neue Veranstaltungsart im Lehrangebot des Instituts für Informatik aufgetaucht, die es vorher noch nie gab.

Was mit dem Beispiel klar geworden sein sollte, ist, daß eine Abbildung von sehr speziellen Anwendungsfällen im Datenmodell leicht zu einem wildwuchernden Datenmodell führen kann.

So war es bei der Erstellung des Datenmodells der Lehrangebotsdaten vorrangig, auf die Behandlung von Spezialfällen weitestgehend zu verzichten und stattdessen die Begriffe so weit wie möglich zu abstrahieren.

Das Datenmodell faßt alle Veranstaltungen und Gruppen und Teile von ihnen unter dem Begriff „*Lehreinheit*" zusammen. Dies macht es möglich, neue Veranstaltungstypen, die bei der Erstellung des Datenmodells noch nicht bekannt waren, ohne Änderung des Datenmodells einzuführen.

Die Zusammenfassung der Veranstaltungen unter einem Begriff ist allerdings nur eine Anwendung der Abstraktion. Im Datenmodell der Lehrangebotsdaten gibt es einen weiteren Anwendungsfall, und zwar die Gesamtheit der am Lehrbetrieb beteiligten Institutionen, also die Universität, Institute, Lehr- und Forschungseinheiten usw.. Diese wurden zu dem Begriff „Verwaltungseinheit" abstrahiert, so daß auch die gelegentlichen Erweiterungen und Umstrukturierungen der beteiligten Institutionen ohne Änderung des Datenmodells berücksichtigt werden können.

2.1.2. Rekursion

Mit der Abstraktion im letzten Abschnitt wurden zwar spezielle Begriffe wie „Übung", „Rechnerübung" usw. zu dem allgemeinen Begriff der „Lehreinheit" zusammengefaßt, allerdings hilft die Abstraktion alleine noch nicht weiter, wenn die Teilveranstaltungen einer Veranstaltung strukturiert werden sollen. Um bei dem Beispiel zu bleiben, das mit der Abstraktion eingeführt wurde: Beließe man es allein bei dem Konzept der Abstraktion, dann würde die Veranstaltung „Informatik 1" (die natürlich als Lehreinheit modelliert ist) zunächst als Übungen eine Gruppe von Komponenten mit dem Datentyp „Lehreinheit" enthalten; später würde dann das Datenmodell insofern geändert, daß sie eine zweite Gruppe von Komponenten mit den Datentypen „Lehreinheit", den Rechnerübungen, enthalten kann. Viel gewonnen wurde also durch die Abstraktion alleine noch nicht, außer daß es nur noch einen Datentyp gibt.

Stellt man eine Veranstaltung und ihre Unterveranstaltungen als Baum dar[1], so gibt es im Grunde zwei Dimensionen, in die die Veranstaltung wachsen (oder schrumpfen) kann: in die Breite, wenn weitere direkte Unterveranstaltungen hinzukommen, und in die Tiefe, wenn sich Unterveranstaltungen in mehrere Unter-Unterveranstaltungen aufteilen. Abbildung 2.1 zeigt zwei Beispiele dazu, die in jüngster Zeit tatsächlich vorgekommen sind.

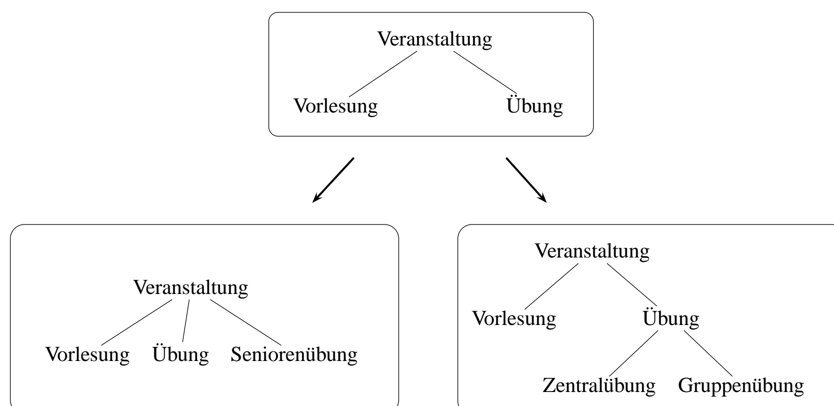


ABBILDUNG 2.1.: Beispiel zur Rekursion

[1] Die Darstellung als Baum ist natürlich nur vereinfachend, da sich z.B. Veranstaltungen Unterveranstaltungen teilen können, d.h. eine Darstellung als allgemeiner gerichteter Graph wäre zutreffend. Aus Gründen der besseren Darstellung wird hier allerdings darauf verzichtet.

Die Lösung, um die Hierarchie von Veranstaltungen offen zu halten, liegt auf der Hand: Veranstaltungen werden *rekursiv* definiert.

Das heißt, eine Veranstaltung (also eine „Lehreinheit") wird dahingehend modelliert, daß sie weitere Lehreinheiten enthalten kann, die wiederum weitere Lehreinheiten enthalten können und so fort. Damit wird im Datenmodell die Tiefe und Breite des Baumes offen gehalten. Nötige Strukturierungsknoten wie z.B. die rechte Übung im Beispiel, sind ganz einfach ebenfalls Lehreinheiten.

Das durch Rekursion gelöste Problem der zu strukturierenden Veranstaltungen ist kein Sonderfall. Alleine im hier vorgestellten Datenmodell kommt ein weiterer Anwendungsfall hinzu: die bereits kurz angesprochenen „Verwaltungseinheiten", die ebenfalls rekursiv definiert sind.

Und auch in der gegenwärtige Entwicklung der Verwaltungsdaten für den Lehr- und Forschungsbetrieb zeichnet sich ab, daß für die Darstellung von *Projekten* ebenfalls eine rekursive Darstellung am günstigsten ist.

2.1.3. Vererbung/Verschattung

Durch die Rekursion der Datenobjekte kann es zu ziemlich langen Ästen kommen.

Betrachtet man das Beispiel in Abbildung 2.2, so wird offensichtlich, daß es ziemlich fehlerträchtig wäre, in jeder der Lehreinheiten auf dem dargestellten Ast festzuhalten, daß es sich um eine Lehreinheit im „Sommersemester 2002" des „Instituts für Informatik" handelt.

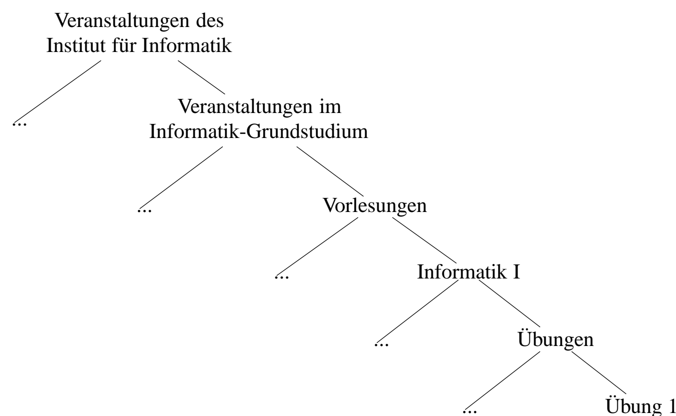


ABBILDUNG 2.2.: Beispiel zur Vererbung/Verschattung

Aus diesem Grund wurde eine *Vererbung auf Datenebene*, bzw. eine *Verschattung von Daten* eingeführt. Somit muß in dem Beispiel das Semester nur noch in der obersten Lehreinheit „Veranstaltungen des Instituts für Informatik" erfaßt werden und wird an alle darunterliegenden Lehreinheiten „vererbt".

Verschattung, also der Vorrang von Definitionen an tieferen Stellen in der Hierarchie vor Definitionen, die aus höheren Hierarchiestufen vererbt werden, tritt ebenfalls auf. Zum Beispiel ist eine Raumvergabestelle für das Gebäude „Oettingenstraße 67" spezifiziert, die an alle Räume des Gebäudes vererbt wird. Für den Besprechungsraum einer Lehr- und Forschungseinheit ist allerdings das Sekretariat dieser Lehr- und Forschungseinheit als Raumvergabestelle angegeben. Diese Angabe verschattet die Spezifikation, die vom Gebäude an den Raum vererbt würde.

2.2. Weitere Konzepte

Neben diesen allgemeinen Konzepten wurden noch einige speziellere Konzepte verwendet. Die Verwendung mehrerer Bezeichner bzw. Texte, abhängig vom Ausgabezweck und der Landessprache, ist eines davon, die Strukturierung der Lehreinheiten ein anderes.

2.2.1. Mehrere Bezeichnungen, Vielsprachigkeit

Die Präsentationen, die aus den Lehrangebotsdaten erstellt werden sollen, unterscheiden sich zum Teil sehr stark. Es gibt sehr ausführliche Veranstaltungslisten und es gibt Übersichtsstundenpläne, auf denen nur wenig Platz für jede Veranstaltung ist. In diesem zweiten Fall ist eine Kürzung der Bezeichnungen von Veranstaltungen, Personen oder Gebäuden unumgänglich.

Dafür gibt es prinzipiell zwei Vorgehensweisen: entweder wird die Kürzung bei der Erzeugung der Präsentation durch einen Algorithmus vorgenommen oder es wird bereits ein verkürzter Text hinterlegt und bei Bedarf darauf zurückgegriffen.

Da die Vorgehensweise mittels eines Algorithmus recht fehleranfällig ist und sowieso eine Möglichkeit zur Hinterlegung von Ausnahmen bedingen würde, wurde im Datenmodell gleich die Möglichkeit modelliert, mehrere Bezeichner, die anhand einer Markierung („kurz“, „lang“ etc.) unterschieden werden können, zu hinterlegen.

In die gleiche Kategorie fällt die Verwendung von unterschiedlichen Bezeichnern für unterschiedliche Landessprachen. Dies wurde mit dem gleichen Mechanismus gelöst: eine weitere Markierung bezeichnet die Landessprache des Bezeichners (“de“, „en“ etc.).

Über Selektoren, die entweder fest in den Transformationen für die Präsentationen vorgegeben sind (wie die Verwendung einer Kurz- oder Langform), oder über Parameter eingestellt werden können (wie die Landessprache der Präsentation), kann bei der Erstellung der Präsentationen auf die passenden Bezeichner zugegriffen werden.

2.2.2. Strukturbaum der möglichen Lehreinheiten

Die in den Abschnitten zur Abstraktion und Rekursion eingeführten Bäume aus „Lehreinheiten“ sind zwar ohne Frage syntaktisch stark strukturiert, was die Semantik der beschriebenen Lehreinheiten angeht, fehlt den Bäumen allerdings eine Strukturierung: Einer Lehreinheit ist nicht anzusehen, welcher Ebene von Lehreinheiten sie zuzuordnen ist, also ob sie z.B. in die Ebene der Veranstaltungen einzuordnen ist oder ob sie eine Übung einer Veranstaltung ist; an ihrer Bezeichnung kann man es auf jeden Fall nicht erkennen, da Veranstaltungen oft sehr unterschiedlich bezeichnet werden (“Informatik 1“, „Praktikum XML“, . . .) und Übungen oft gar keine Bezeichnung besitzen.

Was fehlt, ist eine Typisierung von Lehreinheiten, die allerdings nicht durch das Datenmodell vorgegeben ist, sondern in den Daten formuliert wird. Ansonsten gäbe es ja sofort wieder die Probleme, die mit den Konzepten der Abstraktion und Rekursion eigentlich beseitigt werden sollten.

Um nun in den Baum der Lehreinheiten eine vorgegebene Struktur zu bringen, wird analog zu der Lehreinheit ein *Lehreinheitstyp* definiert, dessen Definition ebenfalls rekursiv ist. Jeder Lehreinheit wird genau ein solcher Lehreinheitstyp zugewiesen, dadurch kann sie eindeutig einem Knoten in dem Baum der Lehreinheitstypen zugeordnet werden. Zur Illustration siehe Abbildung 2.3. Das Fettge-

druckte bezeichnet darin beispielhafte Lehreinheitstypen, darunter steht in Klammern eine mögliche entsprechende Lehreinheit.

Der Vorteil dieser Struktur zeigt sich, sobald man zum Beispiel Veranstaltungslisten ausgeben möchte: Anhand eines Präorderdurchlaufs durch einen Lehreinheitstypen-Baum können alle entsprechenden Veranstaltungen geordnet ausgegeben werden.

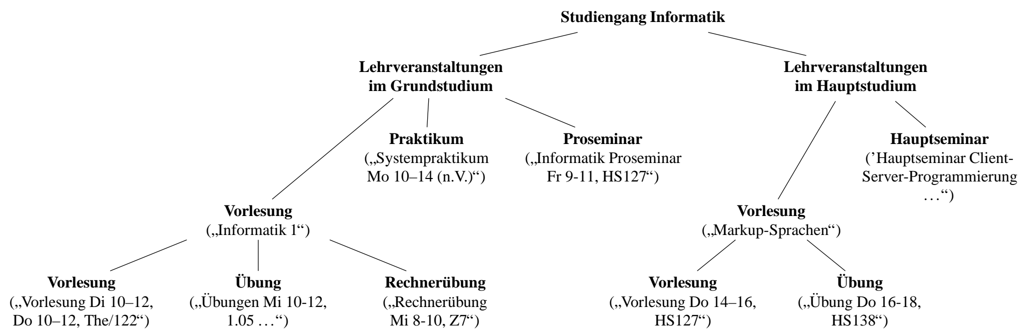


ABBILDUNG 2.3.: Beispiel zum Lehreinheitstyp

2.2.3. Metadaten für die Datenbearbeitung

Dieses Konzept hat zwar keine Bedeutung für die vorliegende Arbeit, soll aber trotzdem an dieser Stelle genannt werden: Für die Unterstützung des Workflows der Datenbearbeitung, mit der die Datenquelle erstellt wird, müssen zum Teil Informationen über den Zustand von Daten oder Bearbeitungsprozessen zusätzlich zu den Daten erfaßt werden können. Beispielsweise ist das die Information, daß ein spezielles Datum bereit telefonisch angefordert wurde und demnächst von einer bestimmten Person per E-Mail zugeschickt wird. Oder daß ein Datum noch geprüft werden muß usw..

Das Datenmodell sieht entsprechende Elemente für diese Metainformationen vor.

2.3. Definition von Begriffen zur Zeit

Gerade bei der Lehrangebotsverwaltung, in der sich sehr viel um Zeitenplanung dreht, spielen exakte Begrifflichkeiten zur Zeit eine große Rolle. Im Datenmodell wurden eine Reihe von Begriffen zur Zeit definiert.

Datum Ein Datum im ISO-Format `DD.MM.YYYY`.

Uhrzeit Eine Uhrzeit im ISO-Format `hh:mm`.

Wochentag Die textuelle Bezeichnung eines Tages, also `Montag`, `Dienstag`, usw.

Zeiteinheit Die Einheit, in der die Zeit gemessen wird: `sec`, `min`, `h`, `d`, `w`, `m`, `y`, `semester`.

Zeitzusatz Der im akademischen Bereich gebräuchliche Zusatz, um den Beginn einer Veranstaltung (nicht) zu verschieben, also `ct` um sie um „viertel nach“ stattfinden zu lassen, `st` um sie pünktlich zu der angegebenen Uhrzeit stattfinden zu lassen.

Zeitpunkt Eine Zeitangabe mit Datum und Uhrzeit.

Zeitintervall Zwei Zeitangaben für Start und Ende, jeweils mit Datum und Uhrzeit.

Dauer Numerische Dauer mit Angabe der Zeiteinheit.

Turnus Die regelmäßige Wiederholung eines Ereignisses in einem bestimmten Abstand von Zeiteinheiten

Vorkommen Die intensionale oder extensionale Beschreibung einer Menge von Zeitpunkten oder Zeitintervallen, die auch die Angabe von Ausschlüssen, wiederum in Form von Vorkommen, erlaubt.

Veranstaltungszeit Die im Universitätsumfeld gebräuchliche Notation, um zu beschreiben, wann eine Veranstaltung stattfindet. Sie besteht

- aus einem Wochentag, zu dem die Veranstaltung das Semester über stattfindet, einer Startuhrzeit, die noch mit einem Zeitzusatz versehen werden kann, einer Enduhrzeit und dem Vorkommen der einzelnen Sitzungen oder
- lediglich aus einer Dauer und dem Vorkommen der einzelnen Sitzungen

Zusätzlich können über Bemerkungen noch individuelle Vermerke (wie z.B. „nach Vereinbarung“) hinterlegt werden.

3. Präsentations-Erzeugungsprozeß

3.1. Überblick

Diese Arbeit beschäftigt sich mit der Erstellung von Präsentationen aus den Lehrangebots-Verwaltungsdaten eines Instituts, wobei sowohl die Daten als auch die Präsentationen häufigen und nicht vorhersehbaren Änderungen unterliegen.

Der Prozeß, der aus den Verwaltungsdaten Präsentationen erstellt, wird in dieser Arbeit mit *Präsentations-Erzeugungsprozeß* bezeichnet. Er wird im Folgenden vorgestellt.

Um den ganzen Präsentations-Erzeugungsprozeß zu strukturieren und um in Zukunft an dem System besser Anpassungen vornehmen zu können, wurde der Prozeß zunächst in drei Abschnitte unterteilt:

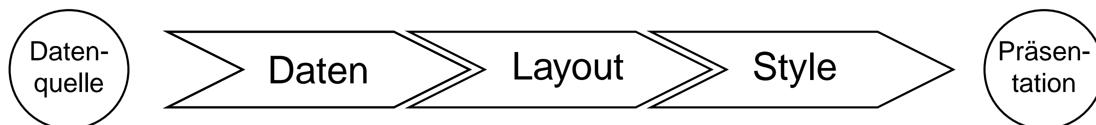


ABBILDUNG 3.1.: Unterteilung des Prozesses in Prozeßabschnitte

Die Bedeutung und Abgrenzung der drei Abschnitte wird weiter unten erkärt. An dieser Stelle geht es zunächst nur um die Strukturierungsprinzipien.

Die einzelnen Abschnitte wurden jeweils weiter in Prozeßschritte unterteilt, in denen beliebig viele Prozessoren die Daten entgegennehmen, bearbeiten und weitergeben.

ABBILDUNG 3.2.: Unterteilung eines Prozeßabschnittes in Prozeßschritte

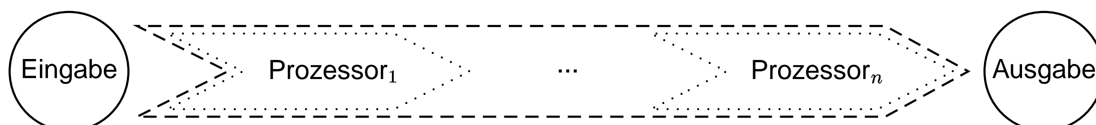


ABBILDUNG 3.3.: Unterteilung eines Prozeßschrittes in Prozessoren

Die Prozeßschritte dienen dabei – genau wie die Prozeßabschnitte – nur der logischen Gliederung, die Datenverarbeitung findet einzig und allein auf der Ebene der Prozessoren statt.

3.1.1. Datenquelle

Um den Präsentations-Erzeugungsprozeß nicht auf die Verarbeitung von Daten zu beschränken, die in Form von XML-Dokumenten vorliegen, wird der Begriff der *Datenquelle* eingeführt. Er abstrahiert die Form der Datenhaltung, die neben XML-Dokumenten nun auch eine relationale Datenbank, ein Dateisystem o.a. sein kann.

Da der Präsentations-Erzeugungsprozeß als internes Datenformat XML-Dokumente verwendet, müssen die Daten, die aus der Datenquelle kommen, zu Beginn des Prozesses über eine Schnittstelle in ein XML-Dokument transformiert werden.

Dies wird von dem allerersten Prozessor, im weiteren *Datenschnittstelle* genannt, erledigt. Ist die Form der Datenquelle bereits XML, so ist die Datenschnittstelle trivial.

3.1.2. Präsentation

Für die Ausgaben des Präsentations-Erzeugungsprozesses wird der Begriff der *Präsentation* eingeführt. Der Begriff beschränkt sich nicht auf die gedruckte Darstellung oder eine Anzeige am Bildschirm, sondern faßt alle möglichen Darstellungsformen (also auch gesprochene o.a.) und deren jeweiligen Beschreibungssprachen zusammen. Eine mögliche Präsentation wäre also z.B. ein tabellarischer Stundenplan, der in LaTeX formuliert ist.

Die Präsentation ist naheliegender Weise die Ausgabe des letzten Prozessors.

Logisch eigenständige Bestandteile einer Präsentation werden als *Inhalte einer Präsentation* bezeichnet. Damit sind Bestandteile gemeint, deren Anordnung innerhalb von Präsentationen in gewissem Maße voneinander unabhängig ist (wie z.B. eine Überschrift, eine Veranstaltungsliste und eine Bemerkung, die in verschiedensten Reihenfolgen in der Präsentation erscheinen können, also erst die Überschrift, dann die Bemerkung, dann die Veranstaltungsliste oder erst die Überschrift, dann die Veranstaltungsliste, dann die Bemerkung usw.).

Für den Ausdruck *Beschreibungssprache einer Präsentation* wird im weiteren Text das Synonym *Zielsprache* verwendet.

3.1.3. Prozessoren

Die Prozessoren sind Einheiten, die eine Eingabe in eine Ausgabe transformieren. Ihre Eingabe ist (mit Ausnahme des ersten Prozessors, dessen Eingabe die Datenquelle ist, die eine andere Form als XML haben kann) immer ein XML-Dokument, ihre Ausgabe ist (mit Ausnahme des letzten Prozessors, dessen Ausgabe in der jeweiligen Zielsprache formuliert ist) ebenfalls wieder ein XML-Dokument, das als Eingabe für den nächsten Prozessor dient.

In Kapitel 3.3 wird das Konzept des *Layoutbaums* eingeführt. Diese spezielle Klasse von XML-Dokumenten ist ab dem zweiten Prozeßabschnitt (dem Layout) durchgängig bis zum Schluß die Struktur, in der die Daten gehalten werden, wodurch eine einheitliche Schnittstelle zwischen den Prozessoren gegeben ist.

Durch die einheitliche Schnittstelle der Prozessoren können bei einer Änderung der Anforderungen an das System einfach die entsprechenden Prozessoren geändert bzw. neue Prozessoren hinzugefügt werden.

Ist für die XML-Dokumente, die als Ausgaben der Prozessoren entstehen, jeweils eine DTD vorhanden, so kann die korrekte Arbeitsweise der einzelnen Prozessoren validiert werden. Dies kann bei der Fehlersuche und -behebung sehr von Vorteil sein, da der Fehler schnell lokalisiert werden kann.

Die in diesem Kapitel vorgestellten Prozessoren sind keineswegs vollständig, sondern sollen nur Lösungen für typische Problemstellungen zeigen. Durch die offene Struktur des Präsentations-Erzeugungsprozesses ist jedoch ein problemloses Hinzufügen oder Ändern von Prozessoren möglich.

3.1.4. Prozeßschritte

Die Einteilung in Prozeßschritte hängt von der Struktur der internen XML-Dokumente (also den XML-Dokumenten, auf denen die Prozessoren ihre Transformationen durchführen) ab. Im Allgemeinen lassen sich die meisten Prozeßabschnitte in drei Prozeßschritte einteilen: Die Generierung der jeweiligen Struktur, deren Aufbereitung und die Generierung der Ausgabe des Prozeßabschnitts.

Jeder Prozeßschritt enthält dafür eine beliebige Anzahl von Prozessoren, die Transformationen auf der Struktur durchführen.

3.1.5. Prozeßabschnitte

3.1.5.1. Abgrenzung von Daten und Layout

Eine Abgrenzung der Datenquelle und der intern verwendeten Daten wurde in Abschnitt 3.1.1 vorgenommen. Wenn in dieser Arbeit von *Daten* gesprochen wird, sind immer die intern verwendeten Daten, also das von der Datenschnittstelle generierte XML-Dokument oder durch Transformationen daraus erzeugte XML-Dokumente gemeint.

In dieser Arbeit wird ein Ansatz eingeführt, bei dem die Daten nicht sofort in eine Präsentation transformiert werden, sondern zunächst ein Zwischenformat erzeugt wird, bei dem jedoch die Semantik der Daten nicht verloren geht: Dieses Zwischenformat ist der *Layoutbaum*. In ihm können sogenannte *Layout-Objekte* definiert werden.

Layout-Objekte beschreiben entweder ganze Präsentationen oder Inhalte von Präsentationen.

Anders ausgedrückt bedeutet dies, daß sich mit dem Wechsel vom Abschnitt Daten zum Abschnitt Layout der beschriebene Weltausschnitt ändert: im Abschnitt Daten werden die zugrundegelegten Anwendungsfälle beschrieben, also in der vorliegenden Arbeit die Verwaltung des Lehrangebots. Im Abschnitt Layout werden Bestandteile von Präsentationen dieser Anwendungsfälle beschrieben.

Die Gründe und die Vorteile dieser Vorgehensweise werden in Abschnitt 3.3 ausführlich dargelegt.

3.1.5.2. Abgrenzung von Layout und Style

Die Abgrenzung der Begriffe *Layout* und *Style*, die hier vorgenommen wird, basiert auf einem Untersuchungsgegenstand der Diplomarbeit von Michael Kraus [18], in der er schreibt:

> „The style model defines how to render data items, the layout model defines where to put them.“[1]

[1] Siehe [18], Seite 17

Dabei beschränken sich seine Aussagen über Layout und Style nicht nur auf die zweidimensionale gedruckte Präsentation, bzw. den zweidimensionalen Bildschirmaufbau, sondern sind viel allgemeiner gehalten:

> „Layout is not always a spatial problem: In the case of aural rendering, for example, the layout model has to decide when to render a certain element, that is the order of elements as well as pauses between them.
>
> Layout also has to deal with a varying number of dimensions: Monoaural rendering is one-dimensional, rendering on paper is two-dimensional, but has to deal with pagination issues, and rendering on a computer screen is three-dimensional, as a theoretically unlimited number of windows offers a third dimension.“[2]

Zur Veranschaulichung sollen die (fiktiven) Beispiele in Abbildung 3.4 betrachtet werden. Offensichtlich werden in den beiden Beispielen die selben Daten zugrundegelegt, wobei diese aber mit zwei unterschiedlichen Layouts ausgegeben werden:

- **Beispiel 1** stellt die Informationen als eine Auflistung von Veranstaltungen dar, zu denen jeweils alle zugehörigen Sitzungen (Vorlesungen, Übungen) angegeben sind.

  Die Layout-Objekte sind folgendermaßen angeordnet: Unter dem Titel der Präsentation steht die Veranstaltungsart, darunter die Veranstaltungsbezeichnung und die Dozenten, voneinander getrennt durch die Zeichenfolge ” – ” usw..

  Style-Informationen sind in Beispiel 1 unter anderem, daß die Überschriften größere Schriftarten und Fettdruck verwenden, die Sitzungen etwas eingerückt sind und die Zeilenabstände vor und nach den Überschriften und nach der URL etwas größer sind.

- **Beispiel 2** faßt die einzelnen Sitzungen nicht auf diese Weise zusammen, sondern stellt sie einzeln in einem Tabellenraster dar.

  Die Layout-Objekte sind folgendermaßen angeordnet: In einer Tabelle steht in der ersten Zeile der Titel der Präsentation, darunter steht eine Zeile mit statischem Text ”Mo”, ”Di”, . . . In der ersten Spalte stehen jeweils die statischen Texte der Anfangs- und Endestunden, getrennt durch die Zeichenfolge ” – ”. Die Veranstaltungen stehen jeweils in einer Zelle der Tabelle; unter der Veranstaltungsbezeichnung steht der erste Dozent und der Veranstaltungsort, getrennt durch die Zeichenfolge ”, ” usw..

  In Beispiel 2 sind Style-Informationen z.B., daß der Präsentationstitel fett gedruckt ist und die Zeiten rechtsbündig ausgegeben werden.

Was ebenfalls deutlich wird, ist, daß die beiden Beispiele unterschiedliche Teilmengen der Daten verwenden. Beispiel 1 zeigt z.B. eine lange Veranstaltungsbezeichnung, den vollen Namen des Dozenten und die URL. Beispiel 2 verwendet jeweils eine kurze Bezeichnung und zeigt keine URL.

Hier ist der Übergang von Layout zu Style fließend: Die Entscheidung, *ob* etwas angezeigt wird oder nicht, gehört in den Bereich Style, die Entscheidung *was* und *wo* es angezeigt wird, in den Bereich Layout.

In Abschnitt 3.2.3 wird der Begriff des Layouts noch einmal untersucht und ein weiterer Begriff eingeführt werden: die *Layoutklassen*, mit denen ähnliche Layouts zusammengefaßt werden können.

[2] Siehe [18], Seite 18

Lehrveranstaltungen des Informatik-Studiums

Vorlesungen

Informatik 1 – Prof. François Bry, Prof. Ohlbach
`http://www.pms.informatik.uni-muenchen.de/lehre/info1/00ss/`

Vorlesung

4-stündig
Dienstag, 10:00–12:00, Donnerstag, 10:00–12:00, The/122

Übung

2-stündig
Mittwoch, 10:00–12:00, 1.05

Praktika

Praktikum „XML und E-Commerce" – Prof. François Bry
`http://www.pms.informatik.uni-muenchen.de/lehre/xmlprakt/00ss/`

4-stündig
Montag, 9:00–13:00, Z7

(a) Beispiel 1

<table>
<tr><th colspan="6">Lehrveranstaltungen des Informatik-Studiums</th></tr>
<tr><td>Zeit</td><td>Mo</td><td>Di</td><td>Mi</td><td>Do</td><td>Fr</td></tr>
<tr><td>8 – 9</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>9 – 10</td><td rowspan="4">Prakt. XML
Bry, Z7</td><td></td><td></td><td></td><td></td></tr>
<tr><td>10 – 11</td><td rowspan="2">Info 1
Bry, The/122</td><td rowspan="2">Übung Info 1
Eisinger, 1.05</td><td rowspan="2">Info 1
Bry, The/122</td><td></td></tr>
<tr><td>11 – 12</td><td></td></tr>
<tr><td>12 – 13</td><td></td><td></td><td></td><td></td></tr>
<tr><td>13 – 14</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

(b) Beispiel 2

ABBILDUNG 3.4.: Beispiele zu Layout vs. Style

3.1.6. Gesamter Prozeß

Abbildung 3.5 gibt einen Überblick über den beschriebenen Prozeß:

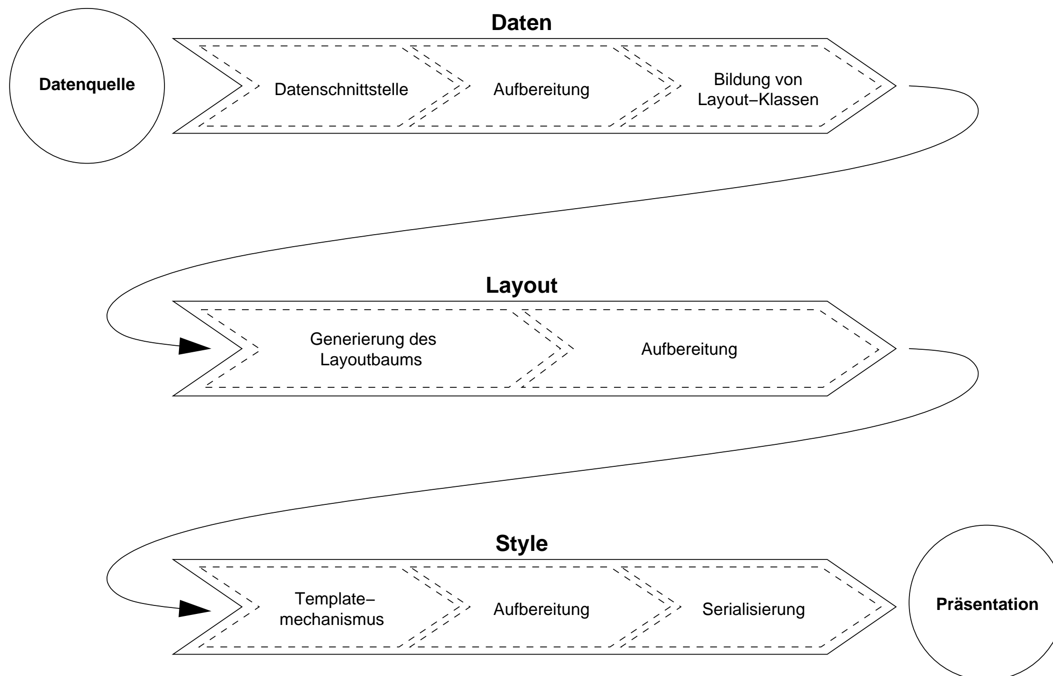


ABBILDUNG 3.5.: Übersicht über den gesamten Präsentations-Erzeugungsprozeß

Auf den folgenden Seiten wird auf die drei Abschnitte des Präsentations-Erzeugungsprozesses näher eingegangen.

3.2. Daten

Am Anfang des Prozesses findet der Zugriff auf die Datenquelle statt. Die Datenquelle besitzt ein beliebiges Format (z.B. bereits XML-Format, ist aber möglicherweise auch eine relationale Datenbank o.a.); eventuell muß zunächst eine Transformation in das XML-Format erfolgen.

Daraufhin werden die Daten aufbereitet. Z.B. können Werte materialisiert werden, die nur in Form von Ausdrücken, Verweisen o.a. vorhanden sind.

Zuletzt werden die Daten in unterschiedliche Layoutklassen transformiert und sortiert.

3.2.1. Datenschnittstelle

Die Datenschnittstelle nimmt eine sehr wichtige Rolle ein, da sie verantwortlich ist für die Flexibilität des Systems bezüglich der Form der Datenhaltung und somit auch für eine einfache Skalierung des gesamten Systems.

In der gegenwärtigen Situation ist für die Datenhaltung des in dieser Arbeit beschriebenen Systems eine einzelne XML-Datei völlig ausreichend. Doch sind schon Diskussionen im Gange, einen größeren Teil der Verwaltungsdaten als nur die des Lehrangebots, zu integrieren[3]. Aus Gründen der Effizienz und der Ermöglichung eines verteilten Zugriffes wäre dann möglicherweise die Speicherung der Daten in einem gemeinsamen (oder auch verteilten) Datenbanksystem nötig.

Durch den Einsatz eines Prozessors, der als Datenschnittstelle fungiert, beschränkt sich der Änderungsaufwand bei einem Wechsel der Art der Datenhaltung auf die Anpassung oder den Austausch dieses Prozessors. Alle weiter hinten liegenden Prozessoren bleiben unbeeinflußt.

3.2.1.1. Mögliche Datenbanksysteme

Prinzipiell sind für die Datenhaltung alle möglichen Datenbanksysteme und Dateisysteme geeignet. Allerdings kann es vorkommen, daß, wenn ein anderes Datenmodell als XML verwendet wird, manche Konstrukte, die in XML sehr natürlich erscheinen, nur umständlich zu formulieren sind, z.B. Rekursion und Alternativen. (Der umgekehrte Fall, also die schwierige Umsetzung von Konstrukten des Datenmodells in XML-Konstrukte, ist natürlich ebenfalls möglich.)

Grundsätzlich hängt die Einsetzbarkeit eines bestimmten Datenhaltungssystems nur von der Verfügbarkeit eines passenden Prozessors ab, der die Daten nach XML transformiert.

Sollte für den hier gegebenen Anwendungsfall ein Datenbanksystem eingesetzt werden, so sprächen für ein relationales System die ausgereiften und schnellen Datenbanksysteme, die auf dem Markt sind. Die Datenbanksysteme, die nativ XML verwenden (u.a. [2, 12, 24]), sind zur Zeit noch nicht ganz ausgereift und auch noch recht spärlich gesät. Erfahrungsberichte finden sich in [17, 19].

Für den Einsatz von XML-Dokumenten hingegen sprächen die gegenwärtigen Prämissen, die Daten textuell bearbeiten zu können, ein einheitliches Austauschformat zu haben und auch die Möglichkeit, gezielt Abweichungen vom vorgegebenen Datenmodell zulassen zu können.

Da jedoch die Datenmengen, die bei der Stundenplanerstellung momentan anfallen, nicht besonders groß sind und auch verteilter Zugriff und verteilte Datenmanipulation im Moment (noch) kein Thema

[3] PMSWeb-Taskforce, 2001/02; LFE PMS, LMU München

sind, spielen derzeit die Vorzüge eines Datenbanksystems wie Transaktionskonzepte, Synchronisation, Error-Recovery, Persistenz u.a. keine bedeutende Rolle. Somit gibt es zur Zeit keine Notwendigkeit eines Datenbanksystems.

Auf Grund dessen fiel für diese Arbeit die Wahl auf den Einsatz von XML-Dokumenten zur Datenhaltung.

3.2.1.2. Sonderzeichenproblematik

Neben der reinen Transformation von Datenfeldern in XML-Elemente und -Attribute gibt es noch ein sehr spezielles Problem und zwar der Einsatz von Sonderzeichen (z.B. Umlaute, aber auch mathematische Symbole). So wird z.B. das Zeichen „ç“ des Namens François in LaTeX mit dem Ausdruck „`\c{c}`“ dargestellt, in HTML mit „`ç`“ oder „`ç`“ usw.. Da ja später Präsentationen in unterschiedlichen Beschreibungssprachen erstellt werden sollen, ist es nicht vermeidbar, daß diese Zeichen in einer generischen Form vorliegen müssen, die am Schluß in die entsprechende Beschreibungssprache übersetzt werden kann. Diese generische Form kann sich an Standards orientieren, z.B. der UTF-8-Codierung für Unicode oder den Entities von HTML.

Natürlich sind Werkzeuge wünschenswert, die bei der Datenerfassung, also bei der Erstellung der Datenquelle, die Erzeugung dieser generischen Form unterstützen. Diese Arbeit geht aber von einer bereits erstellten Datenquelle aus und geht deshalb nicht weiter auf solche Werkzeuge ein.

Was in dieser Arbeit allerdings untersucht wird ist eine Biliothek von generischen Ausdrücken, die für die Sonderzeichen stehen. Mehr dazu in den Abschnitten 4.3.4 und 5.2.

3.2.2. Aufbereitung der Daten

Liegen die Daten nun in XML vor, so können allgemeine, noch nicht präsentationsspezifische, Transformationen durchgeführt werden. Hauptsächlich sind das Materialisierungen von Daten, die aus Effizienzgründen an dieser Stelle ein einziges Mal durchgeführt werden.

3.2.2.1. Materialisierung von Daten

Bei der Materialisierung werden Daten, die nur intensional in Form von bestimmten Regeln vorliegen, extensional in der Datenstruktur erzeugt. Danach erscheinen sie, wie alle anderen Daten, als Elemente, Attribute o.ä.. Oft ist die Frage, ob Daten materialisiert werden sollen (oder nicht), ein Abwägen zwischen Zeit- und Speicherplatzbedarf. Denn natürlich kann durch entsprechende Regeln ein sehr großer Datenumfang beschrieben werden (z.B. die Regel, daß ein Element namens „`<NatürlicheZahlen>`“ die Unterelemente „`<Zahl>`“ mit den aufsteigend geordneten Werten sämtlicher natürlicher Zahlen beinhalten soll).

In dieser Arbeit wird die Materialisierung für zwei Aufgaben eingesetzt: das Auflösen von Referenzen und die Anwendung von Vererbungsregeln.

3.2.2.1.1. Auflösen von Referenzen Um Redundanzen (und damit verbundene mögliche Inkonsistenzen) zu vermeiden, macht das Datenmodell dieser Arbeit regen Gebrauch von Referenzen mit `ID` und `IDREF`. Damit die späteren Transformationsprozesse einfacher gehalten werden können, werden diese Referenzen hier aufgelöst.

Das Auflösen der Referenzen geschieht folgendermaßen:

1. Suchen des über `IDREF` referenzierten Elements mit der entsprechenden `ID` und, falls es vorhanden ist,
2. Kopieren des referenzierten Elements an die Stelle der Referenz (das referenzierende Element selber verschwindet) und
3. Kopieren etwaiger Attribute des referenzierenden Elementes in das kopierte Element.

Zu beachten ist bei diesem Vorgehen, daß das Dokument nach dem Auflösen der Referenzen nicht mehr der ursprünglichen DTD entspricht. Allerdings erfordert beinahe jede der noch folgenden Transformationen eine eigene DTD, will man das entstandene Dokument validieren.

3.2.2.1.2. Vererbungsregeln anwenden Ebenfalls aus dem Grunde der Redundanzvermeidung wurde eine Vererbung auf Datenebene eingeführt, die durch Regeln beschrieben wird. Dabei sind Abhängigkeiten mit dem vorangegangenen Abschnitt, der Auflösung der Referenzen, zu beachten: So müssen in einigen Fällen zunächst

1. Vererbungsregeln für *referenzierte* Elemente angewandt, dann die
2. Dereferenzierung durchgeführt und daraufhin die
3. Vererbungsregeln für die *referenzierenden* Elemente

angewandt werden (und falls auf letztere wiederum referenziert wird, erneute Dereferenzierung etc.).

3.2.3. Sortierung und Transformation der Daten nach Layout-Klassen

Wie in den Beispielen in Abbildung 3.4 zu sehen ist, erfordern einige Präsentationen eine zum Teil grundlegend unterschiedliche Sortierung der Daten:
In dem ersten Beispiel sind die Daten wie folgend sortiert:

1. nach der Art der Veranstaltung (Vorlesungen, Praktika)
2. für die Veranstaltungen einer Art bleibt die in den Daten vorgegebene Sortierung erhalten (dies ist in dem Beispiel nicht ersichtlich, da nur jeweils eine Veranstaltung angegeben ist)
3. nach Art der Sitzung der jeweiligen Veranstaltung
4. nach der Veranstaltungszeit der Sitzungen

In dem zweiten Beispiel sind die Daten wie folgend sortiert:

1. nach der Uhrzeit der Sitzungen (zusammengefaßt anhand der Stunde)
2. nach dem Tag der einzelnen Sitzungen

3. nach dem zusammengesetzten Titel aus Art der Sitzung und Veranstaltungstitel (dies ist im Beispiel nicht ersichtlich, da es keine Überschneidungen gibt)

Weitere Sortierungen sind durchaus vorstellbar.

Neben der Sortierung unterscheiden sich die beiden Beispiele aber auch noch in der Granularität ihrer Daten: In Beispiel 1 sind es Veranstaltungen, in Beispiel 2 fehlt die Ebene der Veranstaltungen, hier sind es die Sitzungen, die sortiert werden.

Allerdings kann man beobachten, daß viele Präsentationen dieselbe Sortierung und Granularität erfordern. So konnten in dieser Arbeit zwei Klassen von Präsentationen gebildet werden:

- Die Präsentation der Veranstaltungen, sortiert nach Veranstaltungsarten, wie z.B. in Abbildung 3.4(a) dargestellt.

  Beispiele wären hierfür die Webseite der Veranstaltungen oder das kommentierte Vorlesungsverzeichnis.

- Die Präsentation der Sitzungen, sortiert nach Stunden und Tagen, wie z.B. in Abbildung 3.4(b) dargestellt.

  Beispiele wären hierfür der Aushangstundenplan und die Raumantragslisten.

Um zu vermeiden, daß die Sortierungen und Transformationen später unnötigerweise für jede Präsentation erneut durchgeführt werden müssen, werden die Präsentationen zu *Layout-Klassen* zusammengefaßt und die Sortierungen und Transformationen der Daten für jede Klasse an dieser Stelle einmal durchgeführt.

3.3. Layout

In diesem Abschnitt werden die Daten in Inhalte von Präsentationen transformiert (siehe auch Abschnitt 3.1.5.1). Das Mittel dazu ist der Layoutbaum. Er besteht aus Strukturierungsknoten und Textobjekt-Blättern und ist fortan die Datenstruktur, auf der alle Transformationen durchgeführt werden.

Die Textobjekte alleine reichen allerdings noch nicht aus, um die Inhalte genügend zu strukturieren: Die Textobjekte werden oft durch bestimmte Zeichenfolgen eingeleitet bzw. abgeschlossen. Und sie werden oftmals durch eine Zeichenfolge voneinander abgetrennt (z.B. durch ein einzelnes Leerzeichen oder durch ein Komma gefolgt von einem Leerzeichen etc.).

Aus diesem Grund werdem dem Layoutbaum in einer weiteren Transformation Trenntexte hinzugefügt, die durch Regeln vorgegeben wurden.

Gerade die Möglichkeit, diese Trenntexte angeben zu können, wird gegenüber dem imperativen Vorgehen (unter Verwendung von Druckanweisungen) als großer Vorzug der hier vorgestellten Vorgehensweise angesehen. Mehr dazu im folgenden Abschnitt.

3.3.1. Motivation des Layoutbaumes

Will man aus Daten Präsentationen erzeugen, so bieten sich im Grunde zwei Vorgehensweisen an: die imperative und die deklarative.

3.3.1.1. Imperative Vorgehensweise

Die imperative Vorgehensweise erzeugt die Präsentation durch eine Folge von Druckanweisungen. Sie ist für einfache, nicht tief strukturierte Präsentationen schnell und einfach zu implementieren.

Abbildung 3.6 zeigt zwei beispielhafte Prozeduren in einer Pseudosyntax, die eine Veranstaltung ausgeben, Abbildung 3.7 zeigt eine mögliche Präsentation, die durch wiederholte Aufrufe dieser Prozeduren erzeugt wird.

Die zugehörige Datenquelle ist nicht angegeben, aber aus der Präsentation sollte man erkennen, was in der Datenquelle repräsentiert ist. Eine XML-Darstellung der Datenquelle ist in Abbildung 5.1 zu finden.

Die Prozeduren gehen davon aus, daß bei der imperativen Vorgehensweise jede Veranstaltung intern in einem Verbund (bzw. record oder structure) `v` repräsentiert ist. Um mit der Implementierung unabhängig von der späteren Zielsprache der Präsentation zu bleiben, werden Formatierungsanweisungen (wie z.B. ein Zeilenwechsel) im Beispiel über Funktionsaufrufe eingefügt.

3.3.1.1.1. Problemstellung Allerdings hat bereits dieses sehr einfache Beispiel eine Reihe von Fehlern, die auf den ersten Blick nicht so einfach zu erkennen sind: Betrachtet man die letzte Zeile der Beispielpräsentation in Abbildung 3.7, so steht dort an erster Position ein Komma, das dort nicht erscheinen sollte. Daß dies trotzdem passieren konnte, liegt an einer getroffenen Annahme bei der Programmierung, nämlich, daß wenn eine Bemerkung ausgegeben wird, immer eine Dauer ausgegeben wird (das selbe Problem finden wir übrigens auch, wenn Sitzungen ausgegeben werden, ohne daß eine Dauer vorhanden ist).

```
procedure printVeranstaltung(v)
   print(v.Titel);
   print(" ");
   printBereiche(v.Bereiche);
   printZeilenwechsel();
   printDauer(v.Dauer);
   for s in v.Sitzungen do
       print(", ");
       printSitzung(s);
   end;
   if v.Bemerkung ≠ ∅ then
       print(", ");
       print(v.Bemerkung);
   end;
   printSpaltenwechsel();
   printDozenten(v.Dozenten);
end;

procedure printBereiche(bereiche)
   if bereiche ≠ ∅ then
       print("(");
       print(first(bereiche).Kurzform);
       for b in tail(bereiche) do
           print(", ");
           print(b.Kurzform);
       end;
       print(")");
   end;
end;
```

ABBILDUNG 3.6.: Programmbeispiel zur imperativen Vorgehensweise

Veranstaltungen	
Informatik I 4-stündig, Di 10–12, Do 10–12, The/122	Bry, Ohlbach
Praktikum „XML und E-Commerce“ (PG, A) 4-stündig, Mo 9–13, Z7	Bry
Fortgeschrittenenpraktikum 4-stündig, Beginn jederzeit möglich	Bry, Conrad, Hegering, Hofmann, Kriegel, Kröger, Linnhoff-Popien, Ohlbach, Wirsing, Zimmer
Praktikum Gruppendynamik , Aushang beachten	N.N.

ABBILDUNG 3.7.: Beispielausgabe zur imperativen Vorgehensweise

Nun könnte man argumentieren, daß diese Fälle durch ausreichend sorgfältiges Programmieren und Testen vermieden werden können. Leider hat sich das in der Praxis der bisherigen Verwaltung der Lehrangebotsdaten nicht bewahrheitet, zumal die tatsächlich verwendeten Daten weit komplexer und tiefer strukturiert sind.

3.3.1.1.2. Erkenntnis Untersucht man dieses Problem genauer, so kommt man auf eine interessante Erkenntnis: Faktum ist, daß die Inhalte von Präsentationen in einer bestimmten Anzahl von Dimensionen angeordnet werden können. Bei Präsentationen, die für das Papier bestimmt sind, sind es zwei, bei auditiven Präsentationen ist es eine, bei Webseiten können es zwei oder drei sein usw. (siehe Abschnitt 3.1.5.2).

Bedenkt man, daß viele Inhalte hierarchisch in Beziehung stehen (also z.B. ein Veranstaltungstitel, der sich aus Veranstaltungsbezeichnung und Dozentenangabe – die im Übrigen auch wieder aus Titel, Vorname und Nachname besteht – zusammensetzt und selber wieder in der Angabe der gesamten Veranstaltung vor der Auflistung der einzelnen Sitzungen steht), so könnte man noch eine weitere Dimension für diese Hierarchie hinzunehmen.

Die imperative Vorgehensweise besitzt jedoch nur eine einzige Dimension: *die Zeit, zu der eine Druckanweisung ausgeführt wird*. Das bedeutet, daß bei dieser Vorgehensweise mindestens eine Dimension fehlt.

Dies erklärt warum der Umgang mit den Trenntexten so komplex ist, da ja zwischen zwei tatsächlich erfolgten – anscheinend hintereinanderliegenden – Ausgaben beliebig viele nicht erfolgte Ausgaben liegen können und nicht mehr ohne weiteres festgestellt werden kann, auf welche Inhalte und welche Dimension sich ausgegebene oder auszugebende Trenntexte beziehen.

3.3.1.2. Deklarative Vorgehensweise

Möchte man die Komposition von Inhalten und die Erzeugung der Präsentation zeitlich entkoppeln, so bietet sich eine deklarative Lösung an. Und da die Daten als XML-Dokumente bereits in einer Baumstruktur vorliegen, liegt eine hierarchische Deklaration auf der Hand.

Das Mittel zur hierarchischen Beschreibung der Inhalte ist der Layoutbaum. Seine Blätter enthalten die einzelnen Daten, seine Knoten strukturieren die Daten als Inhalte der Präsentation.

Ein Beispiel zu einem Layoutbaum ist in Abbildung 3.8 zu finden.

Sowohl Blätter als auch Knoten besitzen eine Bezeichnung, so daß später gezielte Transformationen durchgeführt werden können. Blätter besitzen zudem noch eine Information über den Typ ihres Inhalts; in einem Layoutbaum kann dies entweder Text (TEXT) oder ein generischer Ausdruck (COMMAND) sein. Da die Blätter der Layoutbäume in den nachfolgenden Beispielen immer Text enthalten, wird dort zunächst auf die Typangabe verzichtet.

Das heißt, daß durch den Layoutbaum die Struktur einer Präsentation abstrakt beschrieben wird. Auch wird beschrieben, welche Reihenfolge die Inhalte in der Präsentation besitzen (die Reihenfolge der Blätter bei einem Pre-Order-Durchlauf durch den Baum).

Durch Transformationen auf dem Layoutbaum kann die Struktur der Präsentation geändert und angereichert werden.

Das Hinzufügen von räumlichen Beziehungen der Inhalte ist so eine Anreicherung (damit z.B. die Veranstaltungsbezeichnungen links und die Dozentennamen rechts stehen). Das Hilfsmittel dazu sind

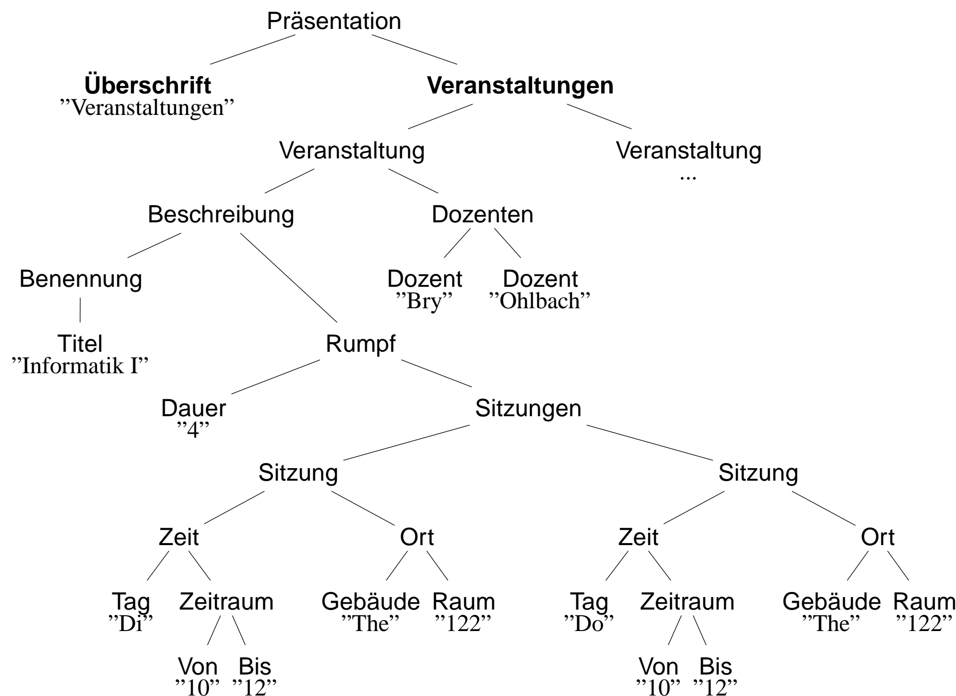


ABBILDUNG 3.8.: Exemplarischer Layoutbaum

die Trenntexte, die den einzelnen Knoten (und eventuell auch Blättern) hinzugefügt werden (siehe auch Abschnitt 3.3.3.3.

Wie entsteht nun ein Layoutbaum? Er wird durch eine Transformation der Daten erzeugt. Die Spezifikation dieser Transformation erfolgt durch den *Layoutbeschreibungsbaum*, der damit den Layoutbaum definiert. Dazu mehr im nächsten Abschnitt.

3.3.1.3. Definition des Layoutbaums

Der Layoutbaum wird durch den *Layoutbeschreibungsbaum* definiert. Dieser definiert im Einzelnen selber wieder Blätter und Strukturierungsknoten zur Strukturierung der Inhalte. Die Blätter können entweder statischen Text (TEXT), generische Ausdrücke (COMMAND) oder Selektoren für die Daten (SELECTOR) enthalten. Eine exemplarische Darstellung eines solchen Baumes (unter Verwendung der im vorangegangenen Abschnitt bereits dargestellten Beispieldaten) ist in Abbildung 3.9 zu sehen (man beachte die Selektoren für die Daten, die im Layoutbaum in Abbildung 3.8 nicht vorkommen).

Daneben können im Layoutbeschreibungsbaum ausgewählte Knoten oder Blätter als Layout-Objekte markiert werden. Die Markierung dient dazu, den Elementen eine eindeutige Identifikation hinzuzufügen, um sie später bei der Erzeugung der Präsentation gezielt ansprechen zu können (siehe Abbildung 3.13). Oft besitzen die Elemente bereits aus den Daten eine eindeutige Identifikation (in XML sind das die ID-Attribute, mit denen Verweise zwischen Elementen hergestellt werden), diese können später gleichermaßen zum Selektieren verwendet werden.

Trenntexte sind im Layoutbeschreibungsbaum noch nicht zu finden, da dieser möglichst kompakt gehalten werden soll. Sie werden dem Layoutbaum durch einen Prozessor in einem späteren Transformationsschritt, der in Abschnitt 3.3.3.3 erläutert wird, hinzugefügt. Im Gegensatz zu der imperativen

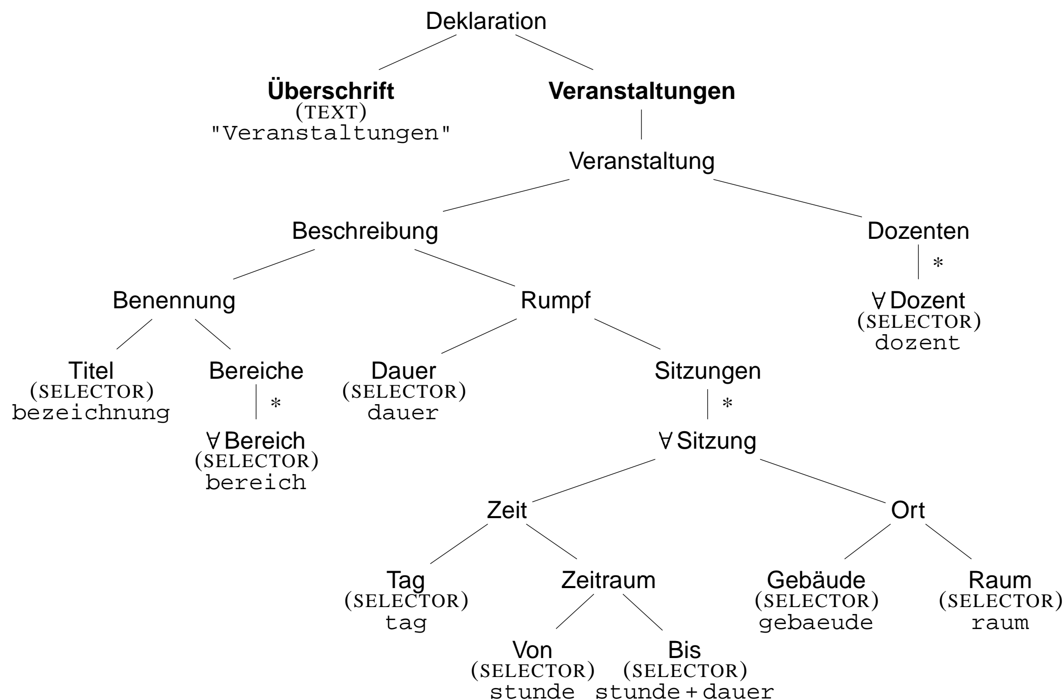


ABBILDUNG 3.9.: Exemplarischer Layoutbeschreibungsbaum

Vorgehensweise im vorigen Abschnitt, ist dies durch die verwendete strukturierte Baumdarstellung auch ohne weiteres möglich.

Der Layoutbeschreibungsbaum in Abbildung 3.9 besitzt ein künstliches Wurzelelement `Deklaration`, darunter befinden sich – hier durch Fettdruck gekennzeichnet – die beiden gesondert markierten Layout-Objekte `Überschrift` und `Veranstaltungen`. Die Positionen direkt unter dem Wurzelelement – bzw. die Markierungen überhaupt – sind nicht zwingend.

Die vorgestellte Darstellungsweise ist jedoch noch nicht ganz zufriedenstellend:

- Ein Formalismus zur Selektion der Daten (wie z.B. XPath [37]) wurde noch nicht nicht bestimmt
- Multiplizitäten können nur unzureichend dargestellt werden; sie werden hier mit „∀“ und „*“ dargestellt, was ausdrücken soll, daß zum einen mehrere Kindknoten dieses Typs vorkommen können („*“), und daß zum anderen der entsprechende Knoten sämtliche vorkommenden Ausprägungen seiner Blätter enthält („∀“).
- Alternative oder bedingte Teilbäume (zum Beispiel ein spezielles Text-Objekt `Bemerkung` im Teilbaum `Zeit` wenn der restliche Teilbaum leer ist) können gar nicht spezifiziert werden.

Die beste Lösung ist wohl der Einsatz einer entsprechend mächtigen Anfragesprache mit Konstruktoren, Selektoren und Filtern. Da in dieser Arbeit jedoch keine neue derartige Anfragesprache entwickelt werden soll, wird es hier bei dieser schematischen Darstellung des Layoutbeschreibungsbaumes belassen und die tatsächliche Implementierung durch bestehende oder bereits in der Entwicklung befindliche Anfragesprachen realisiert. Mehr dazu im Kapitel 5, in der die prototypische Implementierung von PEP erläutert wird.

3.3.2. Generierung des Layoutbaums

Der Layoutbaum entsteht anhand eines Prozessors, der auf den Daten die durch den Layoutbeschreibungsbaum beschriebenen Transformationen ausführt (siehe vorangegangener Abschnitt). Der Layoutbaum sieht im Prinzip aus wie der Layoutbeschreibungsbaum, nur daß die Selektoren an seinen Blättern durch die tatsächlichen Werte ersetzt wurden (der Typ des Inhalts der Blätter ändert sich in diesem Fall natürlich auch auf Text) und die Multiplizitäten aufgelöst wurden. Abbildung 3.8 zeigt ein Beispiel eines Layoutbaums, der aus dem Layoutbeschreibungsbaum in Abbildung 3.9 entsteht, wenn die gleichen Daten wie in Abbildung 3.7 vorausgesetzt werden.

3.3.3. Aufbereitung des Layoutbaums

Analog zu den Prozessoren für die Aufbereitung der Daten in Abschnitt 3.2.2 werden nun hier typische Prozessoren vorgestellt, die den Layoutbaum aufbereiten; dies sind Prozessoren zu Pruning und Anwendung des Distributivgesetzes.

3.3.3.1. Pruning

Bei der Erzeugung des Layoutbaumes können leere Teilbäume entstehen, also Teilbäume, die keine Daten enthalten. Sie werden mit diesem Prozessor entfernt.

Dabei ist zu unterscheiden in Blätter, die *keine* Daten enthalten, und Blätter, die *leere* Daten enthalten. Bei einem Blatt mit leeren Daten wurde bei der Datenerfassung bewußt ein leeres Datum eingegeben, z.B. eine Adresse, die keine Hausnummer besitzt. Bei einem Blatt ohne Daten fehlt diese Information noch.

In den Beispielen entstehen unter dem Knoten mit der Bezeichnung `Benennung` zunächst zwei Teilbäume mit den Wurzeln `Titel` und `Bereiche`. Letzterer ist aber im Fall der Veranstaltung mit dem Titel „Informatik 1“ leer, so daß dieser Teilbaum beim Pruning entfernt wird. Abbildung 3.8 zeigt also tatsächlich nicht den Layoutbaum, der direkt nach seiner Generierung entsteht, sondern die Ausgabe des Pruning-Prozessors, der diesen ersten Layoutbaum als Eingabe erhalten hat.

3.3.3.2. Umordnungen von Textobjekten

Durch die Baumstruktur des Layouts können einige interessante Umordnungen von Textobjekten durchgeführt werden, die bei einem imperativen Ansatz nur sehr umständlich durchzuführen wären.

3.3.3.2.1. Distributivgesetz auf Knoten Betrachtet man in Abbildung 3.4(a) oder 3.7 die Angabe der Dienstags- und Donnerstagssitzungen, so fällt auf, daß die Hörsaalangabe nur ein einziges Mal erfolgt, da beide Sitzungen im gleichen Hörsaal stattfinden. Diese Art von Verkürzung ist bei derartigen Veranstaltungslisten sehr gebräuchlich.

Sie läßt sich durch eine einfache Transformationsregel modellieren (siehe Abbildung 3.10).

Mathematisch gesehen ist dies einfach die Anwendung des Distributivgesetzes auf die Knoten `Zeit` und `Ort`.

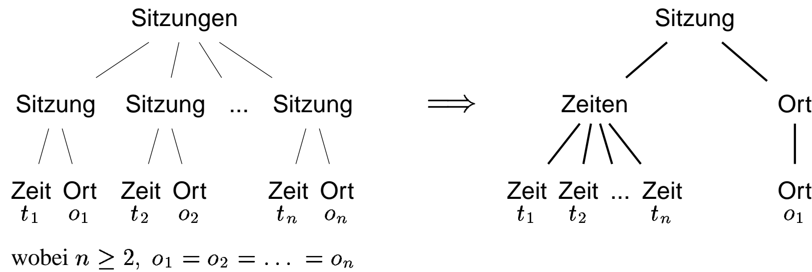


ABBILDUNG 3.10.: Distributivgesetz für Zeit und Ort

3.3.3.2.2. Weitere Umordnungen

Sehr ähnliche Probleme wären die Ausfaktorisierung von

- gemeinsamen Gebäudeangaben bei verschiedenen Hörsälen oder von
- Wochentags- und Ortsangaben bei Veranstaltungen, die zu mehreren Zeiten am gleichen Tag und im selben Hörsaal stattfinden usw.

Desweiteren könnten

- mehrere Dozentennamen in Präsentationen, bei denen wenig Platz dafür zur Verfügung steht (z.B. ein Stundenplan), nach bestimmten Kriterien zusammengefaßt werden, z.B. der Name des ersten Dozenten, gefolgt von dem Zeichen "+".

Von dieser Art gäbe es noch viele weitere Beispiele, auf deren Aufzählung hier aber verzichtet wird.

3.3.3.3. Trenntexte hinzufügen

Ein weiterer Prozessor fügt dem Layoutbaum Trenntexte, die mittels Regeln definiert wurden, hinzu. Diese sind im Abschnitt 3.3.1 motiviert worden, so daß die Notwendigkeit und Funktion solcher Trenntexte offensichtlich sein sollte. Für den vorgestellten exemplarischen Layoutbaum könnten die Regeln für Trenntexte schematisch wie in Abbildung 3.11 aussehen.

Element		Element	Prefix	Infix	Postfix
Überschrift	$\Longrightarrow$	Überschrift	""	""	"@[NEWLINE]"
Veranstaltungen	$\Longrightarrow$	Veranstaltungen	"@[BEGINTAB]"	""	"@[ENDTAB]"
Veranstaltung	$\Longrightarrow$	Veranstaltung	"@[NEWROW]"	"@[NEWCOL]"	"@[ENDROW]"
Beschreibung	$\Longrightarrow$	Beschreibung	""	"@[NEWLINE]"	""
Benennung	$\Longrightarrow$	Benennung	""	" "	""
Bereiche	$\Longrightarrow$	Bereiche	"("	", "	")"
Rumpf	$\Longrightarrow$	Rumpf	""	", "	""
Dauer	$\Longrightarrow$	Dauer	""	""	"-stündig"
Sitzungen	$\Longrightarrow$	Sitzungen	""	", "	""
Sitzung	$\Longrightarrow$	Sitzung	""	", "	""
Zeit	$\Longrightarrow$	Zeit	""	"@[CHAR(NBSP)]"	""
Zeitraum	$\Longrightarrow$	Zeitraum	""	"_"	""
Ort	$\Longrightarrow$	Ort	""	"/"	""
Dozenten	$\Longrightarrow$	Dozenten	""	", "	""

ABBILDUNG 3.11.: Regeln für Trenntexte

Auf die genaue Funktionsweise der verwendeten generischen Ausdrücke – hier in einer Form von `@[...]` dargestellt – wird im nächsten Abschnitt eingegangen. An dieser Stelle ist wichtig zu wissen, daß sie für eine entsprechende Anweisung in der Präsentations-Zielsprache stehen. Also z.B. `@[BEGINTAB]` für das Öffnen einer Tabelle.

Nach der Transformation, die die Trenntexte dem Layoutbaum hinzufügt, sähe der Layoutbaum aus Abbildung 3.8 nun aus wie in Abbildung 3.12 gezeigt. Die Reihenfolge, in der die Trenntexte darin aufgeführt sind (Prefix, Infix, Postfix), entspricht der aus Abbildung 3.11.

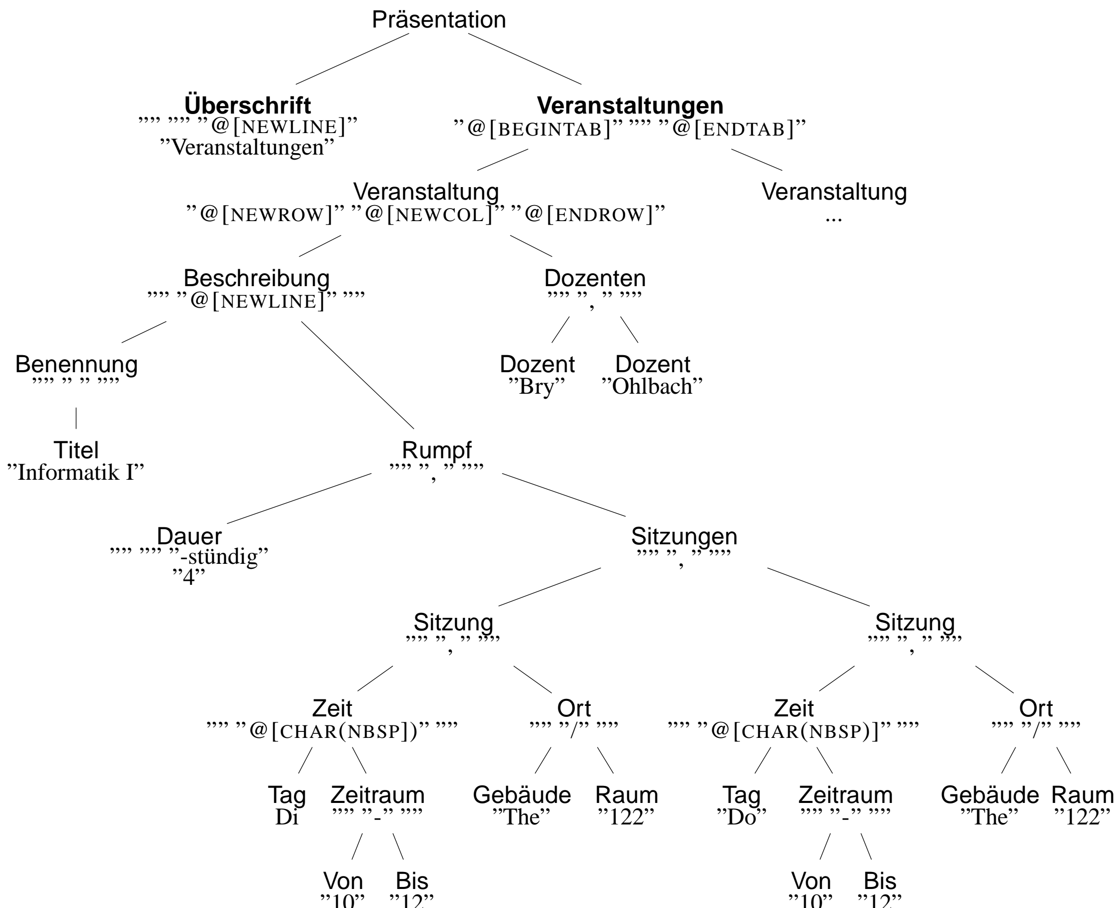


ABBILDUNG 3.12.: Layoutbaum mit Trenntexten

Dabei wird deutlich, daß nicht nur Knoten Trenntexte haben können, sondern auch Blätter. Zumindest einen einleitenden und abschließenden Text, wie im Beispiel mit dem Blatt `Dauer` gezeigt. Allerdings ist dies nicht zwingend, man könnte genausogut die Information in einem vorangehenden oder nachfolgenden Blatt als statischen Text ablegen.

3.3.4. Generische Ausdrücke

Durch die Unabhängigkeit der Layoutbeschreibung von bestimmten Zielsprachen ist es notwendig, Layoutkonstrukte generisch beschreiben und erst später in die entsprechende Zielsprache transformieren zu können. Unter *Layoutkonstrukten* werden im Weiteren alle Gliederungs-Hilfsmittel zur Darstellung von Daten verstanden, also Tabellen, Listen, Zeilenumbrüche usw..

Beschrieben werden die Layoutkonstrukte mittels generischer Ausdrücke. Sie könnten für eine Tabelle in etwa so aussehen:

`@[BEGINTAB]` ... Tabelleninhalt ... `@[ENDTAB]`

Im Abschnitt 3.2.1.2 zur Sonderzeichenproblematik wurde bereits auf eine weitere Anwendung der generischen Ausdrücke hingewiesen: Die Generalisierung von Sonderzeichen (Umlaute, mathematische Symbole etc.). Eine generische Entsprechung könnte für das Zeichen α (Alpha) etwa so aussehen: `@[CHAR(ALPHA)]`.

3.3.4.1. Unterscheidung von Text und generischen Ausdrücken

Ein großes Problem bei der Verwendung der generischen Ausdrücke ist es, sicherzustellen, daß die die Ausdrücke bezeichnenden Zeichenfolgen auch nur als solche im Text vorkommen und nicht an einer anderen Stelle wörtlich zu nehmen sind, also gewissermaßen „gequoted“ werden müßten.

Als Lösung könnte in einem eigenen Prozeßschritt der Datenschnittstelle geprüft werden, ob die Zeichenfolgen der verwendeten generischen Ausdrücke bereits im Dokument vorkommen und falls ja, sie umzubenennen.

Eine pragmatische Lösung besteht darin, sehr selten vorkommende Zeichenfolgen für die generischen Ausdrücke zu verwenden (wie in den gezeigten Beispielen, in denen die Ausdrücke von der in bisher sonst keiner Zielsprache vorkommenden Zeichenfolge `@[ ... ]` umfaßt sind), und mögliche Fehler zuzulassen.

In dieser Arbeit wird noch ein anderer Ansatz verwendet: Durch die Darstellung der Daten als Layoutbaum können generische Ausdrücke in eigenen Blättern abgelegt werden. Da in den Blättern der Typ des Inhalts ebenfalls gespeichert wird, können die generischen Ausdrücke ohne Probleme gesondert behandelt werden. Allerdings bezieht sich dies nur auf generische Ausdrücke, die während des Präsentations-Erzeugungsprozesses *hinzukommen*. Generische Ausdrücke, die bereits in der Datenquelle vorkommen, werden von der Datenschnittstelle auch genau als solche in die Datenstruktur eingefügt. Darum wurde dieser Ansatz mit der Lösung aus dem vorangeganenen Absatz kombiniert.

3.4. Style

Style ist der dritte Abschnitt des Präsentations-Erzeugungsprozesses. Die Daten wurden im vorangegangenen Abschnitt in einen Layoutbaum transformiert, dieser wurde weiter aufbereitet und mit Informationen über räumlichen Beziehungen seiner Elemente angereichert. Eventuell wurden spezielle Layout-Objekte markiert.

Über einen Templatemechanismus können nun einzelne Inhalte der Präsentation, die mit dem Layoutbaum beschrieben wurden, in Templates eingefügt werden. Die Templates sind bereits in der Zielsprache formuliert, mittels Platzhaltern werden die einzufügenden Inhalte selektiert.

In weiteren Transformationen können Style-Informationen hinzugefügt werden.

Zuletzt werden generische Ausdrücke aufgelöst und der Layoutbaum serialisiert. Woraus die endgültige Präsentation entsteht.

3.4.1. Templatemechanismus

Die Idee, die hinter dem Templatemechanismus steckt, ist folgende: Komplexe Dokumentdefinitionen sollen direkt in der jeweiligen Zielsprache implementiert, getestet und vor allem nachträglich leicht angepaßt werden können.

Templates sind Präsentationen in den jeweiligen Zielsprachen, die jedoch anstatt von Daten Platzhalter enthalten, die die Schnittstelle zu den Inhalten des Layoutbaums bilden.

Warum besteht überhaupt die Notwendigkeit, zur Erstellung von Präsentationen zusätzlich noch einen Templatemechanismus einzuführen? Zur Erzeugung von Präsentationen wurde ja bereits die Kombination Layout-Klassen, Layoutbeschreibungsbaum und Trenntexte vorgestellt, die prinzipiell schon ausreichen würde.

Zur Motivation sei folgendes Beispiel gegeben: Möchte man einen aufwendigen tabellarischen Stundenplan ausgeben, so reicht es nicht, die entsprechenden Werte aus den Daten einfach in ein entsprechendes Tabellenkonstrukt einzufügen und durch passende Trenntexte voneinander abzutrennen. Vielleicht sollen einzelne Spalten eine bestimmte Breite bekommen, die Tabelle gestaucht werden (damit der Stundenplan gerade noch auf eine Seite paßt) oder einmalig ein bestimmter Hinweistext eingeblendet werden. Desweiteren sind in einigen Zielsprachen die Einstellungen für manche Layoutkonstrukte recht komplex, wie z.B. Tabellen in LaTeX, so daß die Layoutobjekte nur umständlich als generische Ausdrücke formuliert werden können.

Kurz: Es besteht ein Bedarf, Präsentationen vorab – also ohne Daten – testen zu können und vor allem komplizierte Einstellungen (wie Datei-Header, Tabellendefinitionen usw.) an ihrem Ort der Verwendung definieren zu können und nicht in den viel allgemeiner gehaltenen Konstruktionsregeln für das Layout, wo sich eine kleine Änderung der Einstellungen auf sämtliche Präsentationen auswirkt und nicht nur auf die, in der z.B. eine Tabelle gerade mal etwas kleiner dargestellt werden soll.

Der Templatemechanismus ist eine pragmatische Lösung für diesen Bedarf.

3.4.1.1. Funktionsweise

Die Vorverarbeitung eines Templates läuft folgendermaßen ab: Das Template wird geparst und in einen *Template-Baum* transformiert. Der Template-Baum entspricht im Grunde einem Layoutbeschrei-

bungsbaum, nur daß er keine Strukturierungsknoten besitzt, also von der Tiefe 1 ist, und daß er anstatt von Selektoren für die Daten nun Seletoren für Inhalte besitzt.

In einem weiteren Schritt werden der Template-Baum und der bisherige Layoutbaum in einen weiteren Layoutbaum transformiert. Die Selektoren für die Inhalte werden dabei durch die entsprechenden Inhalte ersetzt. Das Beispiel in Abbildung 3.13 veranschaulicht diese Vorgehensweise.

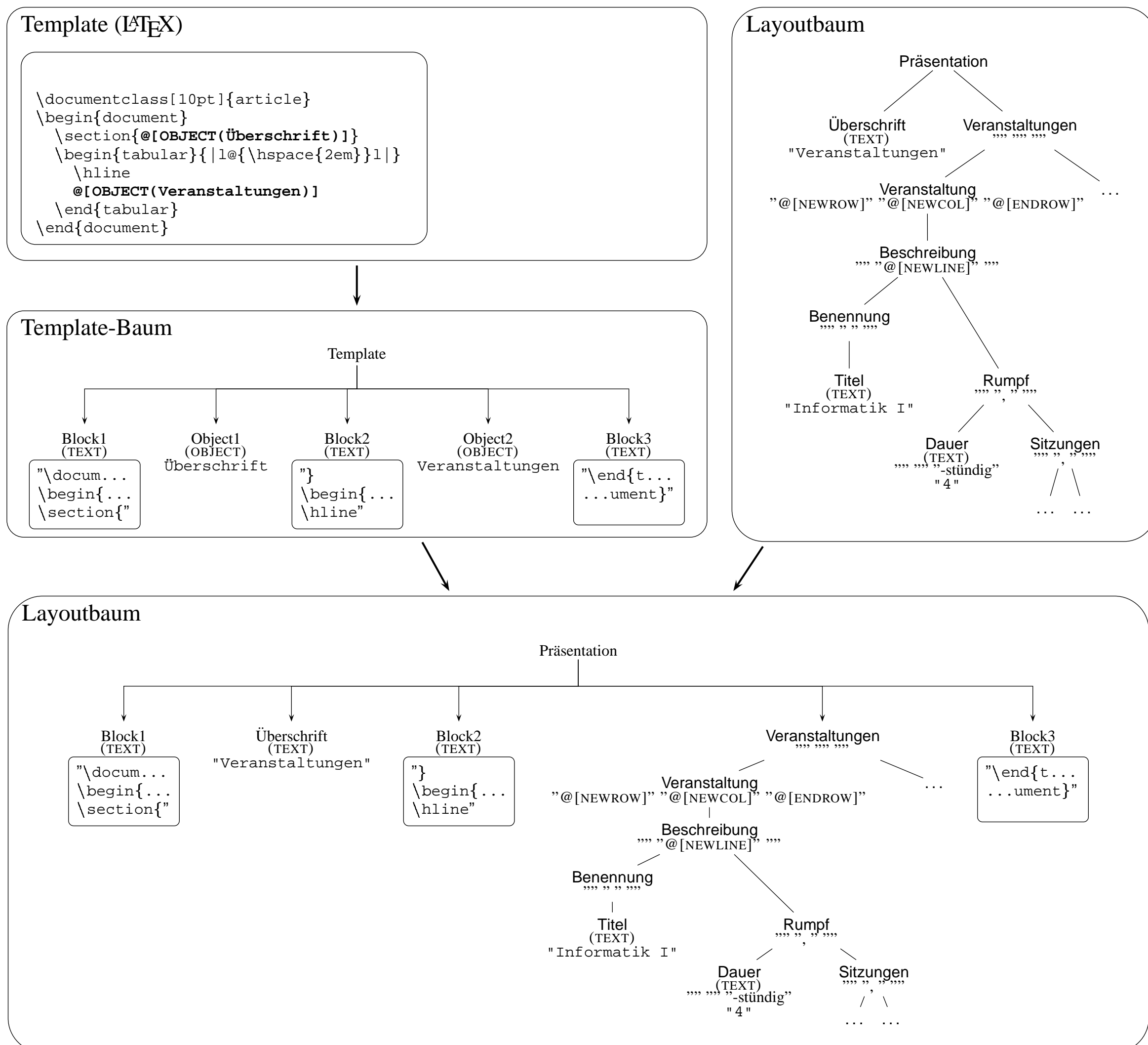


ABBILDUNG 3.13.: Templatemechanismus

3.4.1.2. Auflösung von Objekt-Referenzen

Im vorangegangenen Abschnitt wurde bereits angesprochen, daß die Templates Funktionen enthalten, die die Schnittstelle zu den Inhalten des Layoutbaums bilden. Wie werden diese Inhalte nun referenziert? Dies geschieht über eine eindeutige ID, die jeder Inhalt besitzen kann (diese ID entspricht in der vorliegenden Arbeit dem XML-Attribut ID, das bereits für die Verweise in den Daten verwendet wurde).

Diese ID wurde dem Inhalt entweder bereits bei der Datenerfassung hinzugefügt (z.B. eine bestimm-

te Veranstaltung mit der ID „id-veranstaltung-info1“) oder sie wurde im Layoutbeschreibungsbaum mittels eines Layout-Objektes definiert und dem Inhalt bei der Erzeugung des Layoutbaumes hinzugefügt.

Im Beispiel wird der Einfachheit halber angenommen, daß der Wert des ID-Attributes jeweils gleich dem Knotenbezeichner im Layoutbaum ist.

Das Auflösen der Referenzen bei der Transformation des Template-Baums und des alten Layoutbaumes in den neuen Layoutbaum ist simpel: Der Inhalt mit der entsprechenden ID wird im alten Layoutbaum gesucht und komplett – inklusive eventueller Teilbäume – in den neuen kopiert. Das referenzierende Element wird dabei überschrieben.

3.4.2. Aufbereitung der Präsentation

Nach dieser Transformation beschreibt der Layoutbaum im Prinzip bereits eine vollständige generische Präsentation. Allerdings fehlt der Präsentation noch der gesamte Teilbereich des Style: Bisher wurden die Daten in einem Layoutbaum zu Inhalten komponiert und die Inhalte wurden mittels Layout-Konstrukten strukturiert, d.h. es wurde bestimmt wo, bzw. in welcher Reihenfolge die Inhalte wiedergegeben werden sollen.

Was noch fehlt ist die Information *wie* die Inhalte wiedergegeben werden sollen (vergleiche Abschnitt 3.1.5.2). Dies geschieht nun, indem die Inhalte des Layoutbaums über weitere Prozessoren mit Styleinformationen angereichert werden.

Diese Prozessoren fügen dem Layoutbaum weitere stylebestimmende Strukturierungsknoten hinzu oder ändern, bzw. ergänzen bestehende Trenntexte mit Styleinformationen.

Eine Illustration zeigt Beispiel 3.14. Es zeigt drei Möglichkeiten, das Blatt `Überschrift` mit (generischen) Style-Informationen zu versehen, die es später in Fettruck erscheinen lassen.

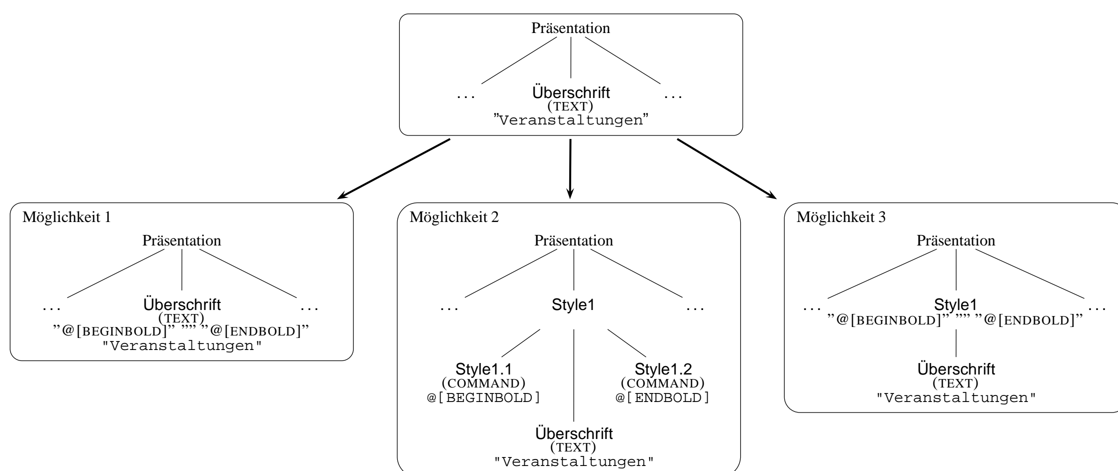


ABBILDUNG 3.14.: Hinzufügen von Style-Informationen

Möglichkeit 1 verändert die Trenntexte des Blattes, d.h. es können z.B. Trenntexte ergänzt oder bestehende durch speziellere ersetzt werden. Möglichkeit 2 fügt einen neuen Strukturierungsknoten mit dem bestehenden Blatt und je einem neuen Blatt davor und danach ein, die jeweils generische Ausdrücke beinhalten. Möglichkeit 3 fügt einen neuen Strukturierungsknoten mit neuen Trenntexten ein.

3.4.3. Transformation der Präsentation in die Zielsprache

Als abschließende Transformationen werden die generischen Ausdrücke in entsprechende Konstrukte der Zielsprache übersetzt und der Layoutbaum in eine Zeichenfolge serialisiert.

3.4.3.1. Auflösung generischer Ausdrücke

Zur Auflösung der generischen Ausdrücke müssen für jede gewünschte Zielsprache Transformationsregeln von den jeweiligen generischen Ausdrücken auf die entsprechenden Konstrukte in den Zielsprachen vorliegen.

Abbildung 3.15 gibt ein Beispiel für derartige Regeln. Sie zeigt eine beispielhafte generische Definition einer Tabelle unter Verwendung der in dieser Arbeit verwendeten Notation und daneben die Entsprechungen in LaTeX und HTML.

generisch		LaTeX	HTML
@[BEGINTAB(2)]		\begin{tabular}{*{2}{l}}	<table>
@[NEWROW]			<tr><td>
$Element_{11}$		$Element_{11}$	$Element_{11}$
@[NEWCOL]		&	</td><td>
$Element_{12}$		$Element_{12}$	$Element_{12}$
@[ENDROW]	$\Longrightarrow$	\\	</td></tr>
@[NEWROW]			<tr><td>
$Element_{21}$		$Element_{21}$	$Element_{21}$
@[NEWCOL]		&	</td><td>
$Element_{22}$		$Element_{22}$	$Element_{22}$
@[ENDROW]		\\	</td></tr>
@[ENDTAB]		\end{tabular}	</table>

ABBILDUNG 3.15.: Übersetzung generischer Ausdrücke

Die Transformation funktioniert folgendermaßen:

Die Transformationsregeln werden auf die COMMAND-Blätter des Layoutbaumes angewandt. Dabei werden die generischen Ausdrücke durch die Zielsprachen-Konstrukte ersetzt und der Typ des Blattinhalts auf TEXT geändert. Das bedeutet, daß nach Abschluß dieser Transformation nur noch Blätter mit dem Typ TEXT im Layoutbaum vorkommen.

Bei der Anwendung der Transformationsregeln kann auch auf bestimmte Blatt-Namen gefiltert werden, so daß Sonderbehandlungen möglich sind.

Abbildung 3.16 zeigt ein Beispiel für eine solche Transformation.

Generisch
Präsentation
...
Style1
...
Style1.1 (COMMAND) @[BEGINBOLD]
Überschrift (TEXT) "Veranstaltungen"
Style1.2 (COMMAND) @[ENDBOLD]

Zielsprache
Präsentation
...
Style1
...
Style1.1 (TEXT) "\textbf{"
Überschrift (TEXT) "Veranstaltungen"
Style1.2 (TEXT) "}"

ABBILDUNG 3.16.: Transformation in Zielsprache (LaTeX)

3.4.3.2. Serialisierung

Abschließend wird der Layoutbaum in eine Zeichenfolge serialisiert

Der Serialisierung geht eine Variante des Flatten-Algorithmus mit einem Pre-Order-Durchlauf durch den Layoutbaum unter Beachtung der Trenntexte voraus.

Dabei wird der Layoutbaum von der Wurzel bis zu den Blättern rekursiv durchlaufen und eine Liste von Blättern erzeugt, die aus den Pre-Trenntexten, den durch die Infix-Trenntexte separierten Werten der enthaltenen Knoten (Rekursion!) oder der Blätter und den Post-Trenntexten besteht.

Damit liegt also immer noch ein XML-Dokument vor, das eine Sequenz von Textknoten enthält. Der letzte Prozessor braucht nur noch die Konkatenation der Inhalte dieser Textknoten zu bilden, und damit ist die Präsentation in der Zielsprache erzeugt.

Das Beispiel in Abbildung 3.17 veranschaulicht die Arbeit der beiden letzten Prozessoren noch einmal: Die erste darin gezeigte Transformation zeigt die Serialisierung. Die zweite Transformation zeigt die Konkatenation.

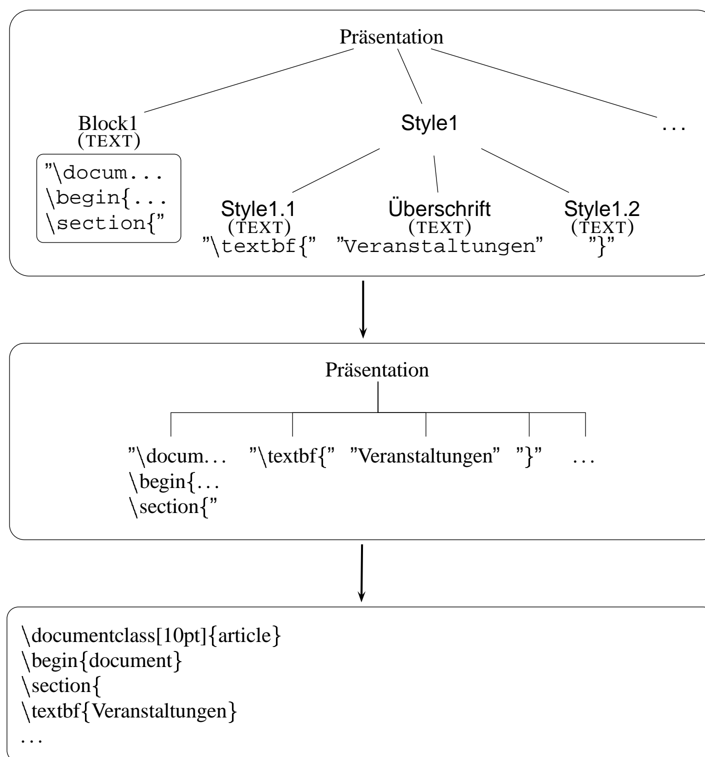


ABBILDUNG 3.17.: Serialisierung des Layoutbaums und Erzeugung der Präsentation in der Zielsprache

4. Rahmenbedingungen einer Implementierung

4.1. Erforderliche Sprachmittel

Dieser Abschnitt soll einen Überblick darüber geben, welche grundlegenden Sprachmittel benötigt werden, um den in Kapitel 3 beschriebenen Präsentations-Erzeugungsprozeß zu realisieren.

4.1.1. Baumbasierte Transformationssprache

Fast jeder Prozessor führt eine Transformation durch, die einen Eingabebaum in einen Ausgabebaum überführt. Dafür ist eine Sprache erforderlich, in der die Transformationsregeln in möglichst deklarativer Weise formuliert werden können.

Einige Transformationen (z.B. das Hinzufügen von Trenntexten) sind so trivial, daß sich daraus keine besonderen Anforderungen an die Transformationssprache ergeben. Andere (z.B. das Pruning) benötigen für eine natürliche Formulierung ein Rekursionsprinzip, mit dem die Ausgabe aus der Transformation von Teilbäumen als Eingabe weiterer Teiltransformationen dienen kann, die dann die Gesamtausgabe aus den Teilausgaben konstruieren.

Diese Art von Rekursion ist beispielsweise in XSLT nicht möglich.

Manche Transformationen (z.B. zum Ausfaktorisieren gemeinsamer Ortsangaben) lassen sich nur formulieren, wenn der Rumpf der Transformationsregel ein Muster nicht nur für einen Ast, sondern für einen komplexeren Teilbaum enthalten kann. Sprachen, die auf XPath basieren, sind dafür zu ausdrucksschwach. Gebraucht werden statt dessen Konstrukte wie das Pattern Matching in einigen funktionalen Programmiersprachen.

Es wäre interessant, wenn eine Menge von primitiven Operationen identifiziert werden könnte, aus denen sich die benötigten Transformationen mit höheren Sprachmitteln zusammensetzen lassen.

4.1.2. Sprache zur Definition von Layoutbeschreibungsbäumen

Im Gegensatz zu normalen Transformationsregeln, die nur einmal implementiert werden und dann – solange nicht grundlegende Änderungen am Prozeß vorgenommen wurden – unverändert bleiben können, unterliegt ein Layoutbeschreibungsbaum einer höheren Änderungshäufigkeit. Jedes Mal, wenn sich das Layout einer Präsentation ändern soll, muß er angepaßt werden. Noch dazu soll nicht nur eine einzige Präsentation erstellt werden. Für fast jede der Präsentationen ist ein eigener Layoutbeschreibungsbaum nötig.

Desweiteren ist die Struktur eines Layoutbeschreibungsbaumes sehr komplex.

So liegt die Forderung nahe, daß die Sprache zur Definition eines Layoutbeschreibungsbaums möglichst einfach, kompakt und übersichtlich sein soll.

4.1.3. Sprache zur Kombination von Prozessoren

Der beschriebene Präsentations-Erzeugungsprozeß geht davon aus, daß Prozessoren hintereinandergeschaltet werden können, so daß die Ausgabe des einen Prozessors als Eingabe des nachfolgenden dient.

Dies erfordert eine Sprache mit einem „Pipeline"-Operator. Im Moment sind keine Verzweigungen oder Iterationen von Prozessoren vorgesehen, aber sie könnten sich doch noch als notwendig erweisen, so daß es günstig wäre, wenn die Kombinationssprache auch dafür Unterstützung bieten würde.

4.1.4. Generische Ausdrücke

In den Beispielen wurden generische Ausdrücke mit einer Pseudosyntax wie @[NEWLINE] verwendet, um das Prinzip der formatunabhängigen Beschreibung zu illustrieren.

Für die betrachtete Anwendung ist eine pragmatisch gewählte Menge solcher generischen Ausdrücke erforderlich, die die auftretenden Fälle abdecken und sich einfach auf die vorkommenden Zielsprachen abbilden lassen. „Vollständigkeit" in dem Sinne, alle Möglichkeiten von XSL-FO, LaTeX und anderen Beschreibungssprachen abdecken zu wollen, wäre hingegen unrealistisch.

Sowohl die Syntax der generischen Ausdrücke als auch ihre Abbildungen auf die verschiedenen Zielsprachen sollten möglichst einfach spezifizierbar sein, um Erweiterungen zu erleichtern.

4.1.5. Sprache zum Zugriff auf Inhalte der Präsentation in einem Template

Die Templates enthalten den statischen Text einer Präsentation und Verweise auf einzufügende (dynamische) Inhalte aus dem Layoutbaum. Um zu spezifizieren, auf welche Inhalte verwiesen wird, ist eine Sprache zur Selektion nötig.

In dem vorangeganenen Kapitel wurde ein Ansatz beschrieben, mit dem Inhalte mittels einer eindeutigen Identifikation referenziert werden. Dies erfordert eine Sprache, die solche Referenzen auflösen kann.

Die Selektion von Inhalten über ihre strukturelle Anordnung im Baum (z.B. in Pfadnotation: `Präsentation/Überschrift`) ist problematisch, da sie nicht eindeutig ist, d.h. daß mehrere Inhalte unter dem gleichen Pfad erreichbar sein könnten. Eine pfadorientierte Sprache zur Selektion ist also nicht ausreichend.

Man könnte die Inhalte natürlich auch über andere (implizite) Merkmale selektieren. Also z.B. über einen enthaltenen Text. Allerdings ist der Aufwand für die Sicherstellung der Eindeutigkeit und der korrekten Selektion wesentlich höher als bei der Verwendung einer ID, der Nutzen hingegen fast der gleiche.

4.2. Erforderliche Werkzeuge

Dieser Abschnitt soll einen Überblick darüber geben, welche grundlegenden Werkzeuge benötigt werden, um den in Kapitel 3 beschriebenen Präsentations-Erzeugungsprozeß zu realisieren.

4.2.1. Kombination von Prozessoren

Auf diese Thematik wurde bereits im letzten Abschnitt eingegangen. Prinzipiell kann der Aufruf von Prozessoren auf zwei Weisen erfolgen:

1. Die Prozessoren werden durch Funktionsaufrufe realisiert. Die Steuerung übernimmt das Programm, in dem diese Funktionsaufrufe formuliert sind. Die Daten werden in einer gemeinsamen Datenstruktur (z.B. einem DOM-Baum) gehalten[1].

2. Die Prozessoren werden durch eigenständige Programme realisiert. Die Steuerung übernimmt ein Steuerungsprogramm, das die einzelnen Programme in einer definierten Abfolge aufruft. Die Daten werden in einem gemeinsamen Austauschformat (z.B. als textuelle XML-Dokumente) entweder über die Standard-Ein/Ausgabe oder über Dateien übergeben.

Vorteil von 1. ist eine bessere Laufzeiteffizienz, da nicht jeder Prozessor zunächst die erhaltenen Daten in eine interne Datenstruktur parsen muß und die Funktionen intern sehr schnell aufgerufen werden können. Der Nachteil ist jedoch, daß alle Prozessoren in einem einzigen monolithischen Programm formuliert werden, was die Austauschbarkeit erschwert. Ein Mittelweg wäre die Realisierung der Prozessoren in Form von extern implementierten Funktionen in APIs, wodurch allerdings die Wahl der Programmiersprache vorgegeben wäre.

Vorteil von 2. ist die Flexibilität des Prozesses, da die Prozessoren mit annähernd beliebigen Formalismen spezifiziert und beliebig ausgetauscht werden können. Nachteil ist das im vorangegangenen Absatz bereits angesprochene wiederholte Parsen der Dokumente.

Welcher Ansatz besser geeignet ist, hängt vom Anwendungszweck ab: Für ein Produktionssystem, bei dem der Prozeß relativ stabil ist, ist sicher der erste Ansatz besser geeignet, für ein Produktionssystem mit einem sich häufig ändernden Prozeß oder für einen Prototypen der zweite.

4.2.2. Prozessoren für Transformationen

Für viele deklarative Anfragesprachen gibt es Software, die die in dieser Sprache formulierten Anfragen auf Dokumenten durchführt. Diese Programme werden ebenfalls *Prozessoren* genannt. Daß der Begriff *Prozessor* in dieser Arbeit für die Einheiten des Präsentations-Erzeugungsprozesses verwendet wird, ist kein Zufall. Sie machen im Grunde das gleiche: Sie führen Transformationen auf dem Daten- bzw. Layoutbaum, hier also auf XML-Dokumenten, durch.

Existiert eine Transformation des Präsentations-Erzeugungsprozesses in einer Anfragesprache, die solche Prozessoren bietet, so liegt der Einsatz eines solchen Prozessors nahe.

[1] Natürlich kann es auch entfernte Funktionsaufrufe geben, wie z.B. mit CORBA. Allerdings erscheinen diese dem Programm selbst als lokale Aufrufe, da der Aufruf durch den *Client Stub* und dessen *data marshaling* transparent durchgeführt wird.

Manchmal ist es jedoch von Vorteil, eine Transformation in einer eigenen deklarativen Sprache zu formulieren, entweder weil es keinen Formalismus gibt, mit dem die Transformation klar ausgedrückt werden kann oder weil noch kein passender Prozessor verfügbar ist. (Ein solcher Fall ist z.B. der Layoutbeschreibungsbaum.)

Um diese eigene Sprache anzuwenden, muß sie entweder direkt interpretiert und ausgeführt werden, d.h. es muß also ein eigener Prozessor für diese Sprache entwickelt werden, oder sie muß in eine andere Anfragesprache übersetzt werden.

4.3. Verfügbare Sprachmittel und Werkzeuge

In den letzten beiden Abschnitten wurden die Anforderungen an die Sprachmittel und Werkzeuge für eine Implementierung untersucht. Nun soll eine Auswahl verfügbarer Sprachmittel und Werkzeuge vorgestellt und verglichen werden. Eine Entscheidung, welche Sprachmittel und Werkzeuge für die Implementierung des Präsentations-Erzeugungsprozesses verwendet werden, wird im nächsten Kapitel getroffen.

4.3.1. Baumbasierte Transformationssprache

Im Folgenden soll ein knapper Überblick über einige Ansätze zur Transformation von XML-Dokumenten gegeben werden. Die vorgestellten Ansätze sind weder vollständig noch ist ihre Darstellung erschöpfend behandelt, da dies über den Rahmen dieser Arbeit hinausgehen würde.

Die Ansätze werden nun zunächst erläutert und dann in einem Vergleich in Hinblick auf ihre Verfügbarkeit, der Natürlichkeit bei der Formulierung einer Transformation und ihrer Ausdrucksfähigkeit gegenübergestellt.

4.3.1.1. XSLT

XSLT ist ein regelbasierter Ansatz zur Transformation von XML-Dokumenten, der Ende 1999 von dem W3C als Recommendation[2], also gewissermaßen als Standard, eingestuft wurde [42].

XSLT transformiert ein XML-Dokument (*source tree*) in ein beliebig strukturiertes Zieldokument (*result tree*). Die Transformationsanweisungen dafür werden in einem sogenannten Stylesheet formuliert. Die Stylesheets sind ebenfalls XML-Dokumente, deren Elemente in dem Namensraum `http://www.w3.org/1999/XSL/Transform` definiert sind.

Sie bestehen aus einer Ansammlung von Regeln (*template rules*), die jeweils aus zwei Teilen bestehen: Einem *pattern* und einem *template*. Das *pattern* entspricht dem Selektor einer Anfragesprache, das *template* dem Konstruktor. Für die Spezifikation der *pattern* wird XPath [37] verwendet.

XSLT bietet Variablen an, allerdings entsprechen sie eher Konstanten, da sie fest an einen Ausdruck gebunden sind.

Abbildung 4.1 zeigt ein Beispiel für ein XSLT-Stylesheet. Darin wird eine HTML-Liste mit dem Inhalt aller `title`-Elemente, die sich in einem `section`-Element befinden, erzeugt.

[2] Der Weg zu einer fertigen Spezifikation (Recommendation) führt über die Stufen *Working Draft (WD)*, *Last Call Working Draft*, *Candidate Recommendation (CR)* und *Proposed Recommendation (PR)* hin zur *Recommendation (REC)*.

```
<?xml version="1.0"?>

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

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

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

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

</xsl:stylesheet>
```

ABBILDUNG 4.1.: Beispiel für Transformationen in XSLT

Bei der Transformation wird, ausgehend vom Wurzelknoten, diejenige Regel aktiviert, deren Selektor den momentanen Knoten matcht. Bei mehreren passenden Regeln wird diejenige mit dem spezifischeren Selektor ausgewählt.

Der Konstruktor der Regel spezifiziert die im Zieldokument einzufügenden Elemente. Er kann auch bestimmen, ob die Regelbearbeitung auf allen oder auf einigen der Nachfolger-Elemente oder auch in einem ganz anderen Teilbaum *rekursiv* fortgeführt wird oder daß sie für diesen Ast beendet wird. Allerdings bezieht sich dabei *Rekursion* nur auf den Aufruf der Regeln und nicht auf die Eingabedaten der Regeln: Die Eingabedaten sind immer Elemente des Quelldokuments.

4.3.1.2. fxt

fxt ist ein Ansatz, der an der Universität Trier entstanden ist [5]. Er ist (wie XSLT) ein regelbasierter Ansatz, dessen Regeln in XML formuliert werden. Anders als in XSLT dienen hier jedoch die formulierten Regeln nicht unmittelbar der Transformation des XML-Quelldokuments, sondern werden zunächst in einen sogenannten *transformer* übersetzt, mit dem dann das XML-Quelldokument transformiert werden kann.

fxt selbst ist in der funktionalen Programmiersprache SML implementiert und auch der *transformer* ist wieder ein SML-Programm. Dies ist einer der großen Vorteile dieses Ansatzes, denn in den Regeln können ebenfalls SML-Funktionen spezifiziert werden, so daß für komplexe Transformationen eine mächtige Programmiersprache im Hintergrund bereit steht.

Ein weiterer großer Vorteil dieses Ansatzes ist das Variablenkonzept: In fxt können globale Variablen definiert werden, deren Werte sich, im Gegensatz zu XSLT, während des Transformationsprozesses ändern können. Dazu sind die Variablen als Keller aufgebaut, in den die Werte mit einem `push`-Operator gelegt und aus dem sie mit einem `get`-Operator gelesen, bzw. mit einem `pop`-Operator entfernt werden können. Erlaubte Werte sind dabei sowohl atomare Werte als auch Bäume oder Wälder. Nachteil dieses Konzeptes mit globalen Variablen ist natürlich, daß die referentielle Transparenz der Regeln verloren geht. Vorteil des Kellerprinzips ist, daß echte Rekursion möglich wird.

Anstatt dem pfadorientierten XPath verwendet fxt einen ganz anderen Ansatz: *fxgrep* [3], das auf regulären Ausdrücken für Bäume basiert.

Abbildung 4.2 zeigt das XSLT-Beispiel aus Abbildung 4.1 als Transformations-Spezifikation für fxt.

```
<fxt:spec>

  <fxt:pat>/*</fxt:pat>
    <ul>
      <fxt:apply />
    </ul>

  <fxt:pat>//section/title/""</fxt:pat>
    <li>
      <fxt:current />
    </li>

 <fxt:pat>default</fxt:pat>
   <fxt:apply />

</fxt:spec>
```

ABBILDUNG 4.2.: Beispiel für Transformationen in fxt

4.3.1.3. HaXml

HaXml [25] implementiert zwei grundlegend unterschiedliche Ansätze, mit denen XML-Dokumente in der funktionalen Programmiersprache Haskell bearbeitet werden können. Der eine stellt eine Bibliothek von Funktionen zur Verfügung, mit der alle möglichen XML-Dokumente selektiert, generiert und transformiert werden können. Der andere implementiert eine Umgebung, in der aus einer vorhandenen DTD Definitionen für Haskell-Datentypen abgeleitet und Funktionen zur Verfügung gestellt werden, die die XML-Dokumente, die der DTD entsprechen, als typisierte Werte einlesen und schreiben.

Hier soll nur der erste Ansatz betrachtet werden. Der zweite ist zwar außer Frage sehr mächtig, allerdings fehlt ihm – naturgemäß – jede Möglichkeit, gesondert Transformationen deklarativ zu definieren. Wenn im folgenden von HaXml gesprochen wird, ist damit immer der erste Ansatz gemeint.

HaXml ist nicht wie XSLT oder fxt regelbasiert, sondern besitzt ein Konzept, das auf Filtern und Kombinatoren (*combinators*) basiert.

An Filtern gibt es sogenannte *predicate and selection filters*, das sind Filter, die Inhalte des XML-Quelldokuments annehmen und bestimmte Inhalte davon zurückliefern, und *construction filters*, das sind Filter, die Inhalte annehmen und daraus Teile des XML-Zieldokuments erzeugen (wie z.B. Elemente, Attribute oder Textinhalte).

Predicate filters sind z.B. `keep`, der jeden Inhalt einfach durchreicht, `tag <name>`, der nur das Element mit dem Namen `<name>` zurückliefert oder `txt`, der nur den Textinhalt eines Elements zurückgibt. *Construction filters* sind z.B. `mkElem`, der ein Element erzeugt oder `replaceAttrs`, der Attribute eines Elements ersetzt.

Mit den Kombinatoren werden mehrere Filter verknüpft. Der einfachste ist 'o', die sogenannte *irish composition*, bei dem der auf der linken Seite stehende Filter auf die Ergebnisse des auf der rechten Seite stehenden angewandt wird.

Die (pfadorientierte) Navigation im XML-Quelldokument wird ebenfalls über die Kombinatoren bewerkstelligt. Der Kombinator `/>` liefert die innere Struktur eines Unterelements, `</` liefert die äußere Struktur.

Auch Verzweigungen sind möglich, z.B. If-Then-Else-Verzweigungen, die ähnlich wie der conditional operator in C funktionieren: `if-Teil ?> then-Teil :> else-Teil`.

Um eine rekursive Bearbeitung von Bäumen zu ermöglichen, gibt es eine Reihe von recht mächtigen Kombinatoren: `deep`, um einen Filter rekursiv absteigend auf das jeweils erste passenden Element in einen Teilbaum anzuwenden, `deepest`, um einen Filter auf das jeweils tiefste passende Element im Teilbaum anzuwenden, `multi`, um einen Filter auf alle passenden Elemente eines Teilbaumes anzuwenden und `foldXml`, um einen Filter auf alle Ebenen eines Teilbaumes, bei den Blättern beginnend, anzuwenden.

Abbildung 4.3 zeigt das Beispiel aus Abbildung 4.1 als HaXml-Skript.

```
module Main where
import Xml
main = processXmlWith (
         mkElem "UL" [
           deep (
             tag "section" /> tag "title" ?>
               mkElem "LI" [ keep /> txt ] :>
               none
           )
         ]
       )
```

ABBILDUNG 4.3.: Beispiel für Transformationen in HaXml

4.3.1.4. XQuery

XQuery [40] ist eine Spezifikation einer funktionalen Anfragesprache für XML-Dokumente, die noch im Entwurfsstadium ist[3]. Aufgrund des frühen Stadiums der Spezifikation kann die nachfolgende Darstellung von XQuery nur einen Überblick über den *momentanen* Stand geben, da sich vieles noch ändern kann.

Der grundlegende Bestandteil einer XQuery-Anfrage ist der *Ausdruck*. Ein Ausdruck besteht aus einer Anzahl von Komponenten. Unter anderem sind dies:

- *Pfadausdrücke:* Sie sind durch XPath 2.0 spezifiziert. Dies umfaßt unter anderem die von XPath 1.0 bekannten absoluten und relativen Pfade und Achsen-Spezifizierer (*axis steps* und *general steps*) und zusätzlich noch ein Dereferenzierungs-Konstrukt (*dereference*) zum Auflösen von Referenzen mittels ID und IDREF.

- *Konstruktoren:* Diese können entweder Konstanten sein (formuliert als Elemente und Attribute in der bekannten XML-Notation) oder Ausdrücke, gekennzeichnet dadurch, daß sie von

[3]Die Spezifikation von XQuery hat momentan den Status „Working Draft“, was bedeutet, daß der Entwurf jederzeit ersetzt oder geändert werden kann.

geschweiften Klammern umgeben sind. Ausdrücke sind zum Beispiel Variablen, die mit Funktionen oder FLWR-Ausdrücken gebunden wurden.

Beispiel für einen Konstruktor:

```
  <book>
    <title>{ $b//title }</title>
    <note>Here is an example</note>
 </book>
```

- *FLWR-Ausdrücke:* Sie dienen der Deklaration und Bindung von Variablen. Sie bestehen aus For-, Let-, Where- und Return-Klauseln.
  - *For-Klauseln* binden Variablen der Reihe nach an alle Elemente, die durch einen Pfadausdruck bestimmt wurden (Iteration).
  - *Let-Klauseln* binden die Variablen an die ganze Sequenz der durch den Pfadausdruck bestimmten Elementen.
  - *Where-Klauseln* filtern die Variablenbindungen.
  - *Return-Klauseln* beinhalten einen Ausdruck, mit dem das Ergebnis eines FLWR-Ausdrucks gebildet wird; dies ist entweder ein Konstruktor oder auch ein primitiver Rückgabewert (Integer, Text, . . .).

  Beispiel für einen FLWR-Ausdruck:

```
for $b in document("bib.xml")//book
where $b/publisher = "Addison-Wesley"
      and $b/year = "2001"
return $b/title
```

  Das Beispiel liefert eine Sequenz aller Buchtitel von Addison-Wesley, die 2001 erschienen sind.

- *Quantifizierte Ausdrücke:* Sie beschreiben sowohl Allquantoren (`every`) als auch Existenzquantoren (`some`).

  Beispiele für quantifizierende Ausdrücke:

```
some $b in //book satisfies $b/year = "2001"
```

  Der Ausdruck ist wahr, wenn es mindestens ein Buch gibt, das 2001 erschienen ist.

```
every $b in //book satisfies $b/year = "2001"
```

  Der Ausdruck ist wahr, wenn alle vorkommenden Bücher 2001 erschienen sind.

XQuery läßt die Definition von benutzerspezifischen Funktionen zu. Die Funktionen können beliebige Werte über Parameter erhalten, also sowohl Werte mit primitivem Datentyp als z.B. auch Elemente, und ebensolche Werte zurückgeben. Der rekursive Aufruf von Funktionen ist ebenfalls erlaubt.

Beispiel für eine benutzerspezifische Funktion:

```
define function depth(element $e) returns xs:integer {
  if (empty($e/*)) then 1
  else max(for $c in $e/* return depth($c)) + 1
}
depth(document("partlist.xml"))
```

Der Funktionsaufruf im Beipiel liefert die maximale Tiefe des Dokuments „partlist.xml“.

4.3.1.5. xcerpt

xcerpt ist die Spezifikation einer Logikanfragesprache für XML-Dokumente, die, wie XQuery, noch im Entwurfsstadium ist. Sie entsteht zur Zeit an der LFE für Programmier- und Modellierungssprachen der LMU München [8, 7].

Im Gegensatz zu den funktionalen Anfragesprachen wie XSLT oder fxt, in denen Regeln rekursiv weitere Regeln aufrufen und aus deren Rückgabewerten, eigenen Konstruktorelementen und selektierten oder berechneten Elementen wieder einen Rückgabewert konstruieren und somit eine starke Vermischung von Konstruktoren und Selektoren aufweisen, trennt xcerpt mit seinem logischen Paradigma diese beiden Bereiche stark von einander ab:

xcerpt besitzt einen Selektorteil (`where`), in dem Variablen definiert werden und einen Konstruktorteil (`construct`), in den diese Variablen eingesetzt werden können. Gebunden werden die Variablen durch eine spezielle Form der Unifikation, genannt *simulation unification*. Diese spezielle Form der Unifikation ist für das pattern matching nötig, und um den Spezifika von Anfragen auf Bäumen gerecht zu werden, wie Beziehungen zu Nachfolgern beliebiger Tiefe (`desc`) oder Anfragemuster (`as`).

In xcerpt existieren ebenfalls quantifizierte Ausdrücke: `all` als Allquantor und `some` als Existenzquantor.

Ein Beispiel für eine Anfrage in xcerpt findet sich in Abbildung 4.4. Es zeigt den gleichen Sachverhalt wie die Beispiele für XSLT, fxt und HaXml.

```
construct
<ul>
  all <li>var T</li>
</ul>

where
desc <section>
  <title>var T</title>
</section>
```

ABBILDUNG 4.4.: Beispiel für Transformationen in xcerpt

Aufgrund des frühen Stadiums ist zu erwarten, daß unter anderem die Syntax der Sprache noch gewissen Änderungen unterliegt, so daß das Beispiel nur einen momentanen Stand wiederspiegelt.

4.3.1.6. Weitere Transformationssprachen

Natürlich sind noch eine Anzahl weiterer Anfragesprachen für XML verfügbar. Unter anderem sind dies Quilt [9], XML-QL [11], XQL [22], Lorel [1] und YATL [23]. Diese haben jedoch alle gemein, daß sie entweder in ihrem Verwendungszweck recht spezifisch ausgerichtet sind oder daß sie sich für die Verarbeitung von XML-Dokumenten nicht durchsetzen konnten. Ganz unbedeutend für die heutige Entwicklung sind sie natürlich auch wieder nicht: So basiert z.B. XQuery auf Quilt und Quilt selbst hat wiederum Anlehnungen an eine Reihe anderer der oben genannten Anfragesprachen.

Bewußt außen vor gelassen wurde die Einbettung von XML in Programiersprachen wie Java (mit einer baumorientierten Schnittstelle zu XML wie DOM oder einer eventbasierten wie SAX), Prolog (mit dem *SWI-Prolog SGML/XML parser* [28]), Haskell (mit der oben kurz genannten Übersetzung von DTDs in Haskell-Datentypen [25]) und anderen, da hier Ansätze zur deklarativen Spezifikation von Transformationen untersucht werden sollten.

4.3.1.7. Vergleich der Transformationssprachen

Abbildung 4.5 zeigt einen Vergleich der hier vorgestellen Transformationssprachen, in dem einige interessante Aspekte herausgegriffen wurden.

	XSLT	**fxt**	**HaXml**	**XQuery**	**Xcerpt**
Rekursive Bearbeitung des Quelldokuments	Ja. (Wiederholtes Anwenden von Regeln)	Ja. (Wiederholtes Anwenden von Regeln)	Ja. (Kombinatoren zur Unterstützung: `deep`, `deepest`, `multi`, `foldXML`)	Ja. (Iteration durch `for`)	Ja. (Wiederholtes Anwenden von Regeln)
Rekursives Bearbeiten der erzeugten Strukturen	Nein.	Ja. (Keller in globalen Variablen)	Ja. (Rekursiver Aufruf benannter Filter)	Ja. (Rekursiver Aufruf benutzerdefinierter Funktionen)	Ja. (Unifikation der Regeln)
Kontexte für Regeln mit gleichem Selektor	Ja.	Nein.	Ja.	Ja.	Ja.
Seiteneffekte	Nein.	Ja. (Verlust der referentiellen Transparenz durch globale Variablen)	Nein.	Möglich. (Je nach Definition der Sichtbarkeit von Variablen)	Nein.
Variablen	Ja. (An Ausdruck gebunden)	Ja. (Globale Variablen mit Kellerprinzip)	Nein.	Ja. (Werden durch `for`, `let` oder Funktionen gebunden)	Ja. (Werden durch Unifikation gebunden)
Verwendete Sprache für Selektion	XPath 1.0. (Pfadorientiert)	fxgrep. (Reguläre Ausdrücke auf Bäumen)	Filter und Kombinatoren. (Pfadorientiert)	XPath 2.0. (Pfadorientiert)	Auf allgemeinen Graphen basiert.
Eigene Funktionen definieren	Ja. (Benannte Templates)	Ja. (Definition von Funktionen inline und extern möglich. Sprache: SML)	Ja. (Definition von Filtern und Kombinatoren intern und extern möglich)	Ja. (Definition von Funktionen möglich. Sprache: XQuery)	Ja. (Benannte Regeln, externe Funktionen geplant)
Dereferenzierung	Ja. (`key`/ `key()`-Konzept)	Ja. (`key`/ `copyKey`-Konzept)	Nein.	Ja. (`dereference`-Operator: =>)	Geplant.
Verfügbar?	Ja.	Ja.	Ja.	Ja. (Technologiestudien)	Ja. (Prototyp)

ABBILDUNG 4.5.: Vergleich der Transformationssprachen

4.3.2. Sprache zur Definition von Layoutbeschreibungsbäumen

Die Generierung eines Layoutbaumes aus einem Layoutbeschreibungsbaum ist im Prinzip ein typisches Problem einer Anfragesprache: Anhand der Konstruktoren wird die Struktur des Layoutbaumes festgelegt, die Selektoren ergänzen die Daten aus dem Datenbaum und mit Filtern können die Daten eventuell noch eingeschränkt werden.

Allerdings ist eine einfache und übersichtliche Definition des Layoutbeschreibungsbaums sehr wichtig, wie bereits in Abschnitt 4.1.2 dargelegt wurde.

Aus diesem Grund kommen von den gezeigten Anfragesprachen nur XQuery und xcerpt in Frage, deren Konstruktoren sich sehr an der Baumstruktur des Zieldokuments orientieren. Eine weitere Möglichkeit ist der Einsatz einer eigenen Regelsprache, die in eine der gezeigten Anfragesprachen übersetzt wird. Da sowohl XQuery als auch xcerpt noch in der Entwicklung sind, ist dies wohl zur Zeit auch die einzige Möglichkeit.

4.3.3. Sprache zur Kombination von Prozessoren

In Abschnitt 4.2.1 wurden die unterschiedlichen Vorgehensweisen zur Kombination von Prozessen und ihre Vorzüge bereits ausführlich behandelt.

Für die prototypische Implementierung in dieser Arbeit ist sicher der Ansatz, in dem die Prozessoren durch eigenständige Programme realisiert werden, von Vorteil. Das Steuerprogramm für die Aufrufe kann prinzipiell in jeder Programmiersprache, die externe Programmaufrufe zuläßt, implementiert werden.

4.3.4. Generische Ausdrücke

Mit den generischen Ausdrücken werden in dieser Arbeit drei unterschiedliche Sachverhalte beschrieben:

1. Generische Layoutkonstrukte, wie z.B. Tabellen, Listen, Absätze usw.
2. Generische Styleanweisungen, wie z.B. Überschriften, Hervorhebung usw.
3. Generische Sonderzeichen, wie z.B. Umlaute, diverse Trennzeichen usw.

Für alle Sachverhalte müssen geeignete Bibliotheken geschaffen werden.

Zu 1.: Hier bietet sich eine Anlehnung an die in Abschnitt 1.3.1 beschriebenen XSL Formatting Objects an.

Zu 2.: Die Styleanweisungen beschreiben nicht nur "physische" Texteigenschaften (Schriftart, -größe, -farbe usw.), sondern auch semantische Texteigenschaften (Überschrift, Hervorhebung, Zitat usw.). XSL Formatting Objects allein ist somit für diesen Anwendungszweck nicht geeignet, da es nur auf die physischen Texteigenschaften eingeht.

Da leider noch keine Beschreibungssprache für Präsentationen existiert, die eine klare Trennung zwischen Layout- und Style-Auszeichnungen (im Sinne dieser Arbeit) macht, sondern immer eine starke Vermischung dieser beiden Sachverhalte besteht (u.a. in HTML und LaTeX), bietet sich hier an, aus

den Anweisungen der verschiedenen Präsentations-Zielsprachen (HTML, LaTeX, ...) eine pragmatische Teilmenge zu bilden, die aus denjenigen Anweisungen besteht, die solche semantischen Texteigenschaften beschreiben.

Zu 3.: Es bietet sich eine Anlehnung an bestehende Standards an. Insbesondere sind dies der ISO-Standard `8879:1986//ENTITIES Added Latin 1//EN`, in dem Entity Sets für Zeichen definiert werden und die HTML 4.01 Specification [34], in der Entity Sets definiert werden, die auf denen von ISO-8879 basieren und diese noch erweitern.

Allerdings wird nicht der volle Umfang der bestehenden Ansätze nötig sein, sondern lediglich eine Teilmenge.

4.3.5. Sprache zum Zugriff auf Inhalte der Präsentation in einem Template

Die Notwendigkeit, daß die Selektionssprache für die Inhalte mit Verweisen auf ID-Attribute zurechtkommen muß, wurde in Abschnitt 4.1.5 bereits dargelegt.

Von den vorgestellten Anfragesprachen unterstützen dies XSLT, fxt und XQuery und in eingeschränkter Form auch xcerpt.

5. Prototypische Implementierung PEP

Mit PEP[1] sollen Möglichkeiten der Implementierung des Präsentations-Erzeugungsprozesses untersucht werden.

Dafür wird zum einen eine Implementierung eines vereinfachten durchgängigen Prozesses vorgenommen, der, wie in Kapitel 3 beschrieben, eine Datenquelle als Eingabe besitzt und ein Dokument in einer beliebigen Zielsprache als Ausgabe generiert. Zum anderen wird die Implementierung eines einzelnen Prozessors in unterschiedlichen Transformationssprachen verglichen.

Für die Erläuterung des Prozesses und der Prozessoren wird in dieser Arbeit folgendes XML-Dokument als Datenquelle verwendet:

```
<?xml version="1.0"?>

<Lehrangebot>
  <Veranstaltungen>
    <Veranstaltung>
      <Titel>Informatik 1</Titel>
      <Dauer>4</Dauer>
      <Dozenten>
        <Dozent><Name>Bry</Name></Dozent>
        <Dozent><Name>Ohlbach</Name></Dozent>
      </Dozenten>
      <Sitzungen>
        <Sitzung>
          <Tag>Di</Tag>
          <Stunde>10</Stunde>
          <Dauer>2</Dauer>
          <Gebaeude>The</Gebaeude>
          <Raum>122</Raum>
        </Sitzung>
        <Sitzung>
          <Tag>Do</Tag>
          <Stunde>10</Stunde>
          <Dauer>2</Dauer>
          <Gebaeude>The</Gebaeude>
          <Raum>E51</Raum>
        </Sitzung>
      </Sitzungen>
    </Veranstaltung>

    <Veranstaltung>
      <Titel>Praktiukum "XML und E-Commerce"</Titel>
      <Bereiche>
        <Bereich>PG</Bereich>
```

[1] Der Name „PEP“ steht naheliegender Weise für „**P**räsentations-**E**rzeugungs**P**rozeß“.

```
        <Bereich>A</Bereich>
      </Bereiche>
      <Dauer>4</Dauer>
      <Dozenten>
        <Dozent><Name>Bry</Name></Dozent>
      </Dozenten>
      <Sitzungen>
        <Sitzung>
          <Tag>Mo</Tag>
          <Stunde>9</Stunde>
          <Dauer>4</Dauer>
          <Gebaeude>Oett</Gebaeude>
          <Raum>Z7</Raum>
        </Sitzung>
      </Sitzungen>
    </Veranstaltung>

    <Veranstaltung>
      <Titel>Fortgeschrittenenpraktikum</Titel>
      <Dauer>4</Dauer>
      <Dozenten>
        <Dozent><Name>Bry</Name></Dozent>
        <Dozent><Name>Conrad</Name></Dozent>
        <Dozent><Name>Hegering</Name></Dozent>
        <Dozent><Name>Hofmann</Name></Dozent>
        <Dozent><Name>Kriegel</Name></Dozent>
        <Dozent><Name>Kroeger</Name></Dozent>
        <Dozent><Name>Linnhoff-Popien</Name></Dozent>
        <Dozent><Name>Ohlbach</Name></Dozent>
        <Dozent><Name>Wirsing</Name></Dozent>
        <Dozent><Name>Zimmer</Name></Dozent>
      </Dozenten>
      <Bemerkung>Beginn jederzeit moeglich</Bemerkung>
    </Veranstaltung>

    <Veranstaltung>
      <Titel>Praktikum Gruppendynamik</Titel>
      <Dozenten>
        <Dozent><Name>N.N.</Name></Dozent>
      </Dozenten>
      <Bemerkung>Aushang beachten</Bemerkung>
    </Veranstaltung>
  </Veranstaltungen>
</Lehrangebot>
```

ABBILDUNG 5.1.: Beispielhafte Datenquelle für PEP

Das Dokument gibt einen stark vereinfachten Ausschnitt typischer Lehrangebotsdaten wieder. Da darauf verzichtet wurde, Prozessoren aus dem Prozeßabschnitt "Daten" (Abschnitt 3.2) zu implementieren, stellt das Dokument bereits eine durch den Abschnitt "Daten" aufbereitete Datenquelle dar.

Der Prozeß und die Prozessoren, die im Weiteren vorgestellt werden, sind großenteils natürlich nicht speziell auf dieses Dokument hin ausgerichtet.

Eine mögliche Ausgabe des Prozesses, die mit PEP aus der Datenquelle erzeugt wird – hier ein HTML-Dokument –, zeigt Abbildung 5.2.

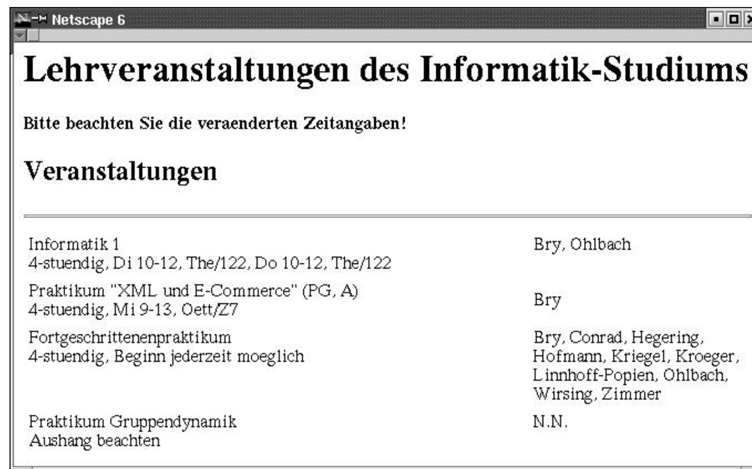


ABBILDUNG 5.2.: Beispielhafte Ausgabe von PEP in HTML, dargestellt mit einem HTML-Browser

5.1. Implementierung des Prozesses

Die Implementierung von PEP erfolgte für Linux. Für die Implementierung des Prozesses wurden nur Standardhilfsmittel von Linux eingesetzt, die in anderen Betriebssystemen ebenfalls existieren und sich leicht übertragen lassen. Auf lange Sicht ist es allerdings sinnvoll, die Definition des Prozesses in einer *betriebssystemunabhängigen* und *XML-basierten* Sprache zu formulieren.

Eine solche Spache ist z.B. mit XML Pipeline [26] gerade im Entstehen. Jedoch besitzt das momentan beim W3C veröffentlichte Dokument, das XML Pipeline beschreibt, gerade erst einmal den Status einer „Submission“, die zur Diskussion freigegeben ist. Deshalb wäre es verfrüht, XML Pipeline zu verwenden.

Die gegenwärtige Implementierung von PEP ist allerdings so offen angelegt, daß eine Ersetzung durch XML Pipeline jederzeit möglich wäre, ohne den Rest von PEP zu beeinflussen.

Für die Beschreibung der Prozessoren wurden unterschiedliche XML-Transformationssprachen (siehe Abschnitt 4.3) und entsprechende Werkzeuge, die die Transformationen ausführen, verwendet.

Wie die momentane Implementierung im Einzelnen aussieht, wird in den nächsten Abschnitten erläutert.

5.1.1. Realisierung der Prozessoren

Die Prozessoren wurden in Form von Shellskripten realisiert. Die Shellskripte für die einzelnen Prozessoren tragen einheitlich den Namen `prozessor.sh` und befindet sich jeweils in einem eigenen Verzeichnis, dessen Name die Aufgabe des Prozessors bezeichnet. Zusätzlich wurden diese Verzeichnisse noch durch übergeordnete Verzeichnisse zusammengefaßt, deren Namen die Bezeichnungen der eingesetzten Sprachen oder Werkzeuge sind.

Die entstandene Verzeichnisstruktur sieht ungefähr so aus wie in Abbildung 5.3 dargestellt.

```
src/
   fx/
      flatten/
      generischeausdruecke/
      layoutbaum/
      prettyprinting/
      pruning/
      serialisierung/
      template/
      trenntexteaufloesen/
      trenntextehinzufuegen/
   gema/
      template/
      zeichennachursprung/
      zeichennachxml/
   haxml/
      layoutbaum/
   xcerpt/
      layoutbaum/
   xquery/
      layoutbaum/
   xslt/
      layoutbaum/
```

ABBILDUNG 5.3.: Verzeichnisstruktur von PEP

5.1.2. Datenschnittstelle der Prozessoren

Die Prozessoren wurden so implementiert, daß sie Eingaben stets über die Standardeingabe entgegennehmen und Ausgaben auf die Standardausgabe schreiben. Eventuell können noch Parameter über die Kommandozeile übergeben werden.

Die Ein- und Ausgabe ist dabei immer ein XML-Dokument (mit der Ausnahme des ersten und letzten Prozessors, siehe Abschnitt 3.1.3).

Dies macht es möglich, mehrere Prozessoren über UNIX-Pipes als Kette hintereinanderzuhängen.

5.1.3. Registrierung der Prozessoren

Um den Speicherort der einzelnen Prozessoren transparent zu halten, werden sie als Shellvariablen registriert, die das Präfix PEPP_[2] und ihre Aufgabe als Namen tragen und deren Wert der komplette Aufruf des jeweiligen Shellskripts `prozessor.sh` ist.

Zentraler Ort für diese Registrierungsdaten ist eine Datei mit dem Namen `registrierung_prozessoren`, die im Wurzelverzeichnis von PEP liegt. Abbildung 5.4 zeigt einen Ausschnitt aus dieser Datei.

Die verwendetete Datei ist ein Shellskript für Shells der csh-Familie; darin werden zunächst einige Abkürzungen für Verzeichnisse gesetzt (um u.a. die Lage möglichst variabel zu halten) und dann die einzelnen Prozessoren definiert.

[2] PEPP steht für PEP Prozessor.

```
################################################################################
# Wurzelverzeichnis von PEP
setenv PEP_HOME                 $HOME"/work/da/anwendung/src"

################################################################################
# Pfade zu den Unterverzeichnissen
# fuer unterschiedliche Anfragesprachen
setenv PEP_HOME_fxt             $PEP_HOME"/fx"
setenv PEP_HOME_GEMA            $PEP_HOME"/gema"
setenv PEP_HOME_XSLT            $PEP_HOME"/xslt"

...

################################################################################
# Prozessoren
setenv PEPP_LAYOUTBAUM          $PEP_HOME_XSLT"/layoutbaum/prozessor.sh"
setenv PEPP_GENERISCHEAUSDRUECKE $PEP_HOME_fxt"/generischeausdruecke/prozessor.sh"
setenv PEPP_FLATTEN             $PEP_HOME_fxt"/flatten/prozessor.sh"
setenv PEPP_SERIALISIERUNG      $PEP_HOME_fxt"/serialisierung/prozessor.sh"
setenv PEPP_PRETTYPRINTING      $PEP_HOME_fxt"/prettyprinting/prozessor.sh"

...
```

ABBILDUNG 5.4.: Registrierung der Prozessoren

5.1.4. Realisierung des Prozesses

Der Prozeß wurde durch eine Hintereinanderschaltung von Prozessoren durch UNIX-Pipes realisiert. Die Hintereinanderschaltung kann entweder in einem gesonderten Shellskript definiert werden (z.B. für immer wiederkehrende Prozesse) oder aber auch direkt auf der Kommandozeile angegeben werden (z.B. für Testzwecke).

Ein solcher Aufruf sieht z.B. wie folgt aus:

```
$PEPP_LAYOUTBAUM < datenquelle.xml | $PEPP_PRUNING | $PEPP_PRETTYPRINTING
```

Mit dem Aufruf wird das XML-Dokument `datenquelle.xml` durch den Prozessor `layoutbaum` in einen Layoutbaum transformiert, aus dem anschließend durch den Prozessor `pruning` leere Äste entfernt werden und der zuletzt durch den Prozessor `prettyprinting` gut lesbar auf der Standardausgabe ausgegeben wird.

In der Entwicklungsphase, aber auch im Betrieb, kann es vorkommen, daß die Ausgabe nicht den Erwartungen entspricht. In einem solchen Fall kann man sich die Ausgabe eines bestimmten Prozessors ganz einfach anschauen, um z.B. zu überprüfen, ob das unerwartete Phänomen dort bereits auftritt oder durch einen weiter hinten in der Kette liegenden Prozessor verursacht wird.

Der Aufruf dafür sieht dann wie folgt aus:

```
$PEPP_LAYOUTBAUM < datenquelle.xml | $PEPP_PRETTYPRINTING
```

Bei dem Aufruf wurde auf ein Pruning des Layoutbaumes verzichtet und stattdessen der Layoutbaum gleich nach seiner Erzeugung ausgegeben.

5.1.5. Vorteile

Folgende Vorteile ergeben sich durch die hier vorgestellten Implementierungsentscheidungen:

5.1.5.1. Flexibilität

- Mit den Prozessoren wird gewissermaßen ein Baukasten von Grundtransformationen definiert. Durch die einheitliche Schnittstelle können alle Prozessoren gleich angesprochen und somit auch in einer nahezu beliebigen Ordnung aufgerufen werden. Das bedeutet, daß Prozesse sehr flexibel gebildet werden können.
- Durch die zentrale Registrierung der einzelnen Prozessoren können diese transparent über ihren Namen angesprochen werden. Wenn sich z.B. ihr Pfad ändert oder sie in einer anderen Anfragesprache formuliert werden hat dies keine Auswirkungen auf den Prozeß.

5.1.5.2. Erleichterte Entwicklung und Fehlersuche

- Dadurch, daß alle Prozessoren als eigenständige Einheiten implementiert werden, können sie isoliert entwickelt und getestet werden.
- Die textuelle Schnittstelle zwischen den Prozessoren erleichtert sowohl die Bereitstellung von Eingabedaten für das isolierte Testen von Prozessoren, als auch das Untersuchen der Ausgabedaten auf etwaige Fehler in der Funktionsweise des Prozessors.
- Der flexible Aufbau des Prozesses macht es möglich, den Prozeß an einer beliebigen Stelle zu unterbrechen, bzw. seine Ausgabe zur Erzeugung einer Datei mit Testausgaben aufzuspalten, um die bis dahin entstandenen Daten zu untersuchen.
- Im Gegensatz zu einem monolithischen System, das intern eine binäre Darstellung der gerade zu bearbeitenden Daten hält, ist es somit nicht nötig, eigene Programme für die Erzeugung von Testdaten implementieren und zur Überprüfung der internen Daten den Quellcode des Prozessors durch eingefügte Testausgabeanweisungen ändern zu müssen.

5.2. Bibliothek von generischen Ausdrücken

Für PEP wurde eine Bibliothek generischer Bezeichner für die folgenden drei Bereiche (siehe auch Abschnitt 4.3.4) geschaffen:

- Generische Layoutkonstrukte
- Generische Styleanweisungen
- Generische Sonderzeichen

Grundsätzlich wurde dabei die in Abschnitt 3.3.4 eingeführte Syntax für die generischen Ausdrücke beibehalten, also generische Ausdrücke in einer Form von `@[ ... ]`.

Im Folgenden werden die einzelnen Bereiche mit ihren jeweiligen Mengen von generischen Ausdrücken vorgestellt.

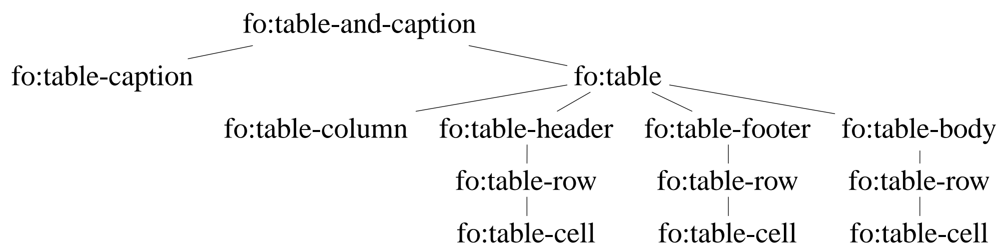
5.2.1. Generische Layoutkonstrukte

Für den Entwurf der generischen Layoutkonstrukte von PEP wurde eine Teilmenge der XSL Formatting Objects verwendet. In Abschnitt 1.3.1 wurden bereits die einzelnen Gruppen der Formatting Objects vorgestellt. Hier von Belang sind die *Block*, *Table* und *List Formatting Objects*.

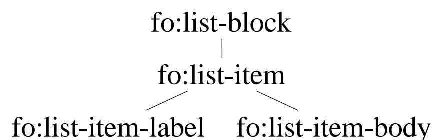
Bei den anderen ist entweder die Granularität zu grob (*Declaration*) oder zu fein (*Inline*) oder sie sind für PEP nicht von Bedeutung.

Im Folgenden sind die einzelnen relevanten Formatting Objects kurz aufgeführt:

- **Block Formatting Objects:**
  fo:block
- **Table Formatting Objects:**



- **List Formatting Objects:**



Die Bezeichnungen der Formatting Objects wurden für PEP weitgehend übernommen, allerdings wurde der Formalismus, wie Inhalte "geklammert" werden, an die Gegebenheiten der generischen Ausdrücke angepaßt.

Anstatt Inhalte also (wie in XML üblich) z.B. mit

```
<fo:table>
  ... Inhalt ...
</fo:table>
```

zu klammern, wird der generische Ausdruck jeweils mit dem Präfix `begin-`, bzw. dem Postfix `end-` versehen, und natürlich wird auf das Namespace-Präfix `fo` verzichtet.

Das Beispiel sieht mit generischen Ausdrücken dann folgendermaßen aus:

```
@[begin-table] ... Inhalt ... @[end-table]
```

5.2.2. Generische Styleanweisungen

Für den Entwurf der generischen Styleanweisungen von PEP wurde eine pragmatische Teilmenge der HTML 4-Auszeichnungen verwendet. Sie kommen aus den folgenden Bereichen:

- Headings,
- Phrase elements,
- Quotations,
- Subscripts and superscripts,
- Alignments und
- Font styles

Für PEP wurden die Bezeichnungen der jeweiligen HTML-Auszeichnungen weitgehend übernommen – mit den gleichen Anpassungen wie im letzten Abschnitt.

5.2.3. Generische Sonderzeichen

Für den Entwurf der generischen Sonderzeichen von PEP wurden die *character entity references* [31] von HTML 4 verwendet.

Die *character entity references* definieren Entities zu folgenden Bereichen:

- *ISO 8859-1 characters*: Sonderzeichen des ISO 8859-1 Zeichensatzes.
- *Symbols, mathematical symbols, and Greek letters*.
- *Markup-significant and internationalization characters:* Zeichen, die im Markup eine spezielle Bedeutung haben, wie z.B. die Zeichen "<" und ">" oder spezielle Zeichenabstände. Daneben werden noch Entities zur Internationalisierung definiert, wie z.B. eine unterschiedliche Schreibrichtung.

Für PEP wurde die Bezeichnungen der darin definierten Entities übernommen. Die generischen Ausdrücke zur Beschreibung der Sonderzeichen haben folgende Form:

```
@[char( ... )]
```

Anstelle der Punkte steht die Bezeichnung der jeweiligen Entity. Der generische Ausdruck für ein nicht-trennbares Leerzeichen ("*no*-*b*reak *sp*ace") sieht somit so aus:

```
@[char(nbsp)].
```

5.3. Implementierung ausgesuchter Prozessoren

Ziel bei der Auswahl der zu implementierenden Prozessoren war es einerseits, aus den Prozessoren einen durchgängigen Prozeß bilden zu können, und andererseits Prozessoren mit einem interessanten Verhalten bzw. mit einer interessanten Implementierung herauszugreifen.

5.3.1. Klassifikation der verwendeten Regelsprachen

Ein solcher interessanter Gesichtspunkt ist die vom Prozessor verwendete Regelsprache. Hier wurden zwei Klassen von Regelsprachen gebildet: bestehende Regelsprachen und eigene, anwendungsspezifische Regelsprachen.

5.3.1.1. Definition der Transformationen mit bestehenden Regelsprachen

Für feststehende Transformationen, bzw. für Transformationen, die nur sehr selten geändert werden, wurden die Transformationsanweisungen direkt in bestehenden Regelsprachen formuliert. Beispiele hierfür sind das Pruning und das Flatten, die einfach nach einem vorgegebenen Algorithmus arbeiten.

Transformationen, die sich häufig ändern, konnten direkt mit bestehenden Regelsprachen definiert werden, wenn diese Sprachen die Möglichkeit boten, Transformationen sehr einfach und übersichtlich zu definieren. Ein Beispiel hierfür ist der Layoutbeschreibungsbaum in XQuery.

5.3.1.2. Definition der Transformationen mit eigenen, anwendungsspezifischen Regelsprachen

Für eine Reihe von Aufgabestellungen, bei denen häufig Änderungen vorgenommen werden müssen, war jedoch eine direkte Defininition der Transformationen in den meisten der bestehenden Regelsprachen zu komplex und zu schlecht wartbar.

Vergleicht man in Abbildung 5.5 die beiden Ausschnitte aus den Regeln zum Hinzufügen von Trenntexten – einmal in fxt und einmal in einer eigenen (XML-basierten) Sprache – so wird der Vorteil einer eigenen, maßgeschneiderten Regelsprache recht deutlich.

Auf die Implementierung von Auswertern für die eigenen Regelsprachen wurde verzichtet. Stattdessen wurden Übersetzer in bestehende Regelsprachen implementiert.

Da die verwendeten bestehenden Regelsprachen alle auf XML aufbauen, wurden die eigenen Regelsprachen noch in Sprachen, die auf XML aufbauen und in andere unterschieden: Die anderen Sprachen müssen vor der Übersetzung zunächst in ein XML-Dokument transformiert werden, um von den auf XML basierten Werkzeugen übersetzt werden zu können.

5.3.1.2.1. Regelsprachen, die auf XML aufbauen Die Übersetzer für diese Regelsprachen sind in fxt oder XSLT formuliert, die Sprachen, in die übersetzt wird, sind jeweils dieselben.

Mit der Implementierung von PEP wurden folgende Regelsprachen eingeführt:

- PEP$_{layout}$ zur Definition von Layoutbeschreibungsbäumen
- PEP$_{separators}$ zur Definition der Regeln zum Hinzufügen der Trenntexte

Regeln in fxt:

```
<fxt:spec>
  <fxt:pat>/*</fxt:pat>
  <fxt:copyTagApply/>
  <fxt:pat>//Ueberschrift</fxt:pat>
  <fxt:copyTag>
    <fxt:addAttribute name="post" val="@[br]"/>
    <fxt:apply/>
  </fxt:copyTag>
  <fxt:pat>//Veranstaltungen</fxt:pat>
  <fxt:copyTag>
    <fxt:addAttribute name="pre" val="@[begin-table]"/>
    <fxt:addAttribute name="post" val="@[end-table]"/>
    <fxt:apply/>
  </fxt:copyTag>
  <fxt:pat>//Veranstaltung</fxt:pat>
  <fxt:copyTag>
    <fxt:addAttribute name="pre" val="@[begin-row]"/>
    <fxt:addAttribute name="post" val="@[end-row]"/>
    <fxt:apply/>
  </fxt:copyTag>
      ⋮
</fxt:spec>
```

Gleiche Regeln in eigener Sprache:

```
<transform>

  <separators selector = "//Ueberschrift"
              post = "@[br]"/>

  <separators selector = "//Veranstaltungen"
              pre  = "@[begin-table]"
              post = "@[end-table]"/>

  <separators selector = "//Veranstaltung"
              pre  = "@[begin-row]"
              post = "@[end-row]"/>
      ⋮
</transform>
```

ABBILDUNG 5.5.: Ausschnitte aus Regeln zum Hinzufügen von Trenntexten

- PEP$_{generic}$ zur Definition der Regeln zum Übersetzen der generischen Ausdrücke

Sie werden im Folgenden erläutert.

PEP$_{layout}$ Bei dem Entwurf von PEP$_{layout}$ stand die Prämisse im Vordergrund, die Layoutbeschreibungsbäume sehr einfach und übersichtlich definieren zu können.

Die Sprache besteht aus folgenden Komponenten:

- **Layoutbeschreibungsbaum** Enthält einen einfachen Strukturierungsknoten.
- **Strukturierungsknoten** Dient zur Strukturierung von Inhalten. Er ist entweder ein *einfacher Strukturierungsknoten* oder ein *iterierter Strukturierungsknoten*.
  - **Einfacher Strukturierungsknoten** Element aus einem beliebigen Namespace und mit einem beliebigen Namen. Es besitzt ein optionales Attribut `id`, dessen Wert zur späteren eindeutigen Identifizierung dient.

    Er kann entweder weitere *Strukturierungsknoten* oder einen *Wert* enthalten.
  - **Iterierter Strukturierungsknoten** Element aus einem beliebigen Namespace und mit einem beliebigen Namen. Es besitzt ein Attribut `pep:selector`, dessen Wert ein Ausdruck in der Selektionssprache des Übersetzers ist.

    Mit dem Attribut `pep:selector` wird auf eine Sequenz von Elementen im Eingabedokument verwiesen. Für jedes dieser Elemente wird ein *einfacher Strukturierungsnoten* mit dem Inhalt des iterierten Strukturierungsknotens eingefügt und der Kontext im Eingabedokument auf das selektierte Element gesetzt.

    Er kann entweder weitere *Strukturierungsknoten* oder einen *Wert* enthalten.
- **Wert** Beschreibt intensional oder extensional einzufügende Textinhalte. Er ist entweder ein *statischer Wert*, *berechneter Wert*, *selektierter Wert* oder ein *referenzierter Wert*.
  - **statischer Wert** Element `pep:value`. Es besitzt einen Textinhalt.
  - **berechneter Wert** Element `pep:value`. Es besitzt ein Attribut mit dem Namen `expression`, dessen Wert ein Ausdruck in der Selektionssprache der bestehenden Regelsprache ist, in die später übersetzt wird.
  - **selektierter Wert** Element `pep:value` Es besitzt ein Attribut `selector`, dessen Wert ein Ausdruck in der Selektionssprache des Übersetzers ist.

    Mit dem Attribut *selector* wird auf ein einzufügendes Element bzw. einen einzufügenden Elementinhalt im Eingabedokument verwiesen.
  - **referenzierter Wert** Element `pep:value`. Es besitzt ein Attribut mit dem Namen `reference`, dessen Wert, dem Wert eines `id`-Attributes im `Layoutbeschreibungsbaum` oder im Eingabedokument entspricht.

    Mit dem Attribut *reference* wird auf ein einzufügendes Element bzw. einen einzufügenden Elementinhalt im Eingabedokument verwiesen.

In Abbildung 5.6 wird ein Beispiel dieser Regelsprache gezeigt. Es implementiert genau den Layoutbeschreibungsbaum aus Abbildung 3.9.

```
<Praesentation>
  <Ueberschrift id="Ueberschrift">
    <pep:value>Veranstaltungen</pep:value>
  </Ueberschrift>
  <Veranstaltungen id="Veranstaltungen">
    <Veranstaltung pep:selector="/Lehrangebot/Veranstaltungen/Veranstaltung">
      <Beschreibung>
        <Benennung>
          <Titel>
            <pep:value selector=".//Titel/text()" />
          </Titel>
          <Bereiche>
            <Bereich pep:selector="Bereiche/Bereich">
              <pep:value selector="text()" />
            </Bereich>
          </Bereiche>
        </Benennung>
        <Rumpf>
          <Dauer>
            <pep:value selector="Dauer/text()" />
          </Dauer>
          <Sitzungen>
            <Sitzung pep:selector="Sitzungen/Sitzung">
              <Zeit>
                <Tag>
                  <pep:value selector=".//Tag/text()" />
                </Tag>
                <Zeitraum>
                  <Von>
                    <pep:value selector=".//Stunde/text()" />
                  </Von>
                  <Bis>
                    <pep:value
                      expression="sum((.//Stunde/text()|.//Dauer/text()))" />
                  </Bis>
                </Zeitraum>
              </Zeit>
              <Ort>
                <Gebaeude>
                  <pep:value selector=".//Gebaeude/text()" />
                </Gebaeude>
                <Raum>
                  <pep:value selector=".//Raum/text()" />
                </Raum>
              </Ort>
            </Sitzung>
          </Sitzungen>
        </Rumpf>
      </Beschreibung>
      <Dozenten>
        <Dozent pep:selector="Dozenten/Dozent">
          <pep:value selector=".//Name/text()" />
        </Dozent>
      </Dozenten>
    </Veranstaltung>
  </Veranstaltungen>
</Praesentation>
```

ABBILDUNG 5.6.: Beispiel für PEP$_{layout}$

PEP$_{separators}$ Ziel bei dem Entwurf der Regelsprache für die Trenntexte war eine kompakte Definition der Transformationen zum Hinzufügen der Trenntexte.

Die Sprache besteht aus folgenden Komponenten:

- **Transformationsanweisung** Element `transform`. Enthält einen oder mehrere Trenntexte.
- **Trenntext** Element `separators`. Es besitzt das Attribut `selector`, dessen Wert ein Ausdruck in der Selektionssprache des Übersetzers ist und die optionalen Attribute `pre`, `in` und `post`, die die Trenntexte enthalten.

  Mit dem Attribut `selector` wird auf das Element im Eingabedokument verwiesen, dem die Trenntexte hinzugefügt werden sollen.

In Abbildung 5.7 ist die DTD für diese Regelsprache dargestellt. Abbildung 5.8 zeigt ein Beispiel dieser Regelsprache.

PEP$_{generic}$ Ziel bei dem Entwurf der Regelsprache war eine kompakte Definition der Transformationen, die generischen Ausdrücke in ihre Zielsprachenäquivalente übersetzen.

Die Sprache besteht aus folgenden Komponenten:

- **Transformationsanweisung** Element `transform`. Enthält eine oder mehrere Sprachen.
- **Sprache** Element `language`. Es besitzt das Attribut `name`, dessen Wert die Bezeichnung der Zielsprachen ist, in denen Präsentationen erstellt werden. Also z.B. LaTeX, HTML usw..

  Die Sprache enthält Ausdrücke, die generische Ausdrücke und ihre Übersetzung in der Zielsprache enthalten.
- **Ausdruck** Element `expression`. Es besitzt das Attribut `generic`, dessen Wert der zu übersetzende generische Ausdruck ist, und das Attribut `destination`, dessen Wert der Text ist, in den übersetzt werden soll.

In Abbildung 5.9 ist die DTD für diese Regelsprache dargestellt. Abbildung 5.10 zeigt ein Beispiel dieser Regelsprache.

5.3.1.2.2. Regelsprachen, die nicht auf XML aufbauen In manchen Fällen wäre es für eine Regelsprache ungünstig, auf XML aufzubauen. Zum Beispiel dann, wenn dynamische Inhalte in umfangreiche statische Texte eingefügt werden sollen, kann überlegt werden, ob es nicht sinnvoll wäre, die Regeln in einem geeigneten Formalismus in den statischen Text einzubetten.

Eine Analogie gibt es z.B. in der Programmiersprache C mit der Ausgabefunktion `printf`: um die Zeichenfolge „Die nächste Veranstaltung findet am Freitag statt.“ auszugeben – wobei der jeweilige Tag (hier also der Freitag) durch eine Funktion `tag()` geliefert wird – könnte man folgende Formulierung verwenden:

```
printf("Die nächste Veranstaltung findet am ");
printf(tag());
printf("statt.");
```

```
<!ELEMENT transform (separators)+>
<!ELEMENT separators EMPTY>
<!ATTLIST separators selector CDATA #REQUIRED
                     pre CDATA #IMPLIED
                     in CDATA #IMPLIED
                     post CDATA #IMPLIED>
```

ABBILDUNG 5.7.: DTD von $\text{PEP}_{separators}$

```
<transform>
  <separators selector="//Ueberschrift"
              post="@[br]"/>
  <separators selector="//Veranstaltungen"
              pre="@[begin-table]"
              post="@[end-table]"/>
  <separators selector="//Veranstaltung"
              pre="@[begin-table-row]"
              post="@[end-table-row]"/>
  <separators selector="//Beschreibung"
              pre="@[begin-table-cell]"/>
              in="@[br]"/>
              post="@[end-table-cell]"/>
  <separators selector="//Benennung"
              in="@[char(nbsp)]"/>
  <separators selector="//Bereiche"
              pre="("
              in=", "
              post=")"/>
  <separators selector="//Rumpf"
              in=", "/>
  <separators selector="//Dauer"
              post="-stuendig"/>
  <separators selector="//Sitzungen"
              in=", "/>
  <separators selector="//Sitzung"
              in=", "/>
  <separators selector="//Zeit"
              in="@[char(nbsp)]"/>
  <separators selector="//Zeitraum"
              in="-"/>
  <separators selector="//Ort"
              in="/"/>
  <separators selector="//Dozenten"
              pre="@[begin-table-cell]"/>
              in=", "/>
              post="@[end-table-cell]"/>
</transform>
```

ABBILDUNG 5.8.: Beispiel für $\text{PEP}_{separators}$

```
<!ELEMENT transform (language)*>
<!ELEMENT language (expression)*>
<!ELEMENT expression EMPTY>
<!ATTLIST language name CDATA #REQUIRED>
<!ATTLIST expression generic CDATA #REQUIRED
                     destination CDATA #REQUIRED>
```

ABBILDUNG 5.9.: DTD von $\text{PEP}_{generic}$

```
<transform>
  <language name="html">
    <expression generic="@[br]"
                destination="<br />"/>
    <expression generic="@[char(nbsp)]"
                destination=" "/>
    <expression generic="@[begin-table]"
                destination="<table>"/>
    <expression generic="@[end-table]"
                destination="</table>"/>
    <expression generic="@[begin-table-row]"
                destination="<tr>"/>
    <expression generic="@[end-table-row]"
                destination="</tr>"/>
    <expression generic="@[begin-table-cell]"
                destination="<td>"/>
    <expression generic="@[end-table-cell]"
                destination="</td>"/>
  </language>

  <language name="latex">
    <expression generic="@[br]"
                destination="\newline"/>
    <expression generic="@[char(nbsp)]"
                destination="~"/>
    <expression generic="@[begin-table]"
                destination="\begin{tabular}"/>
    <expression generic="@[end-table]"
                destination="\end{tabular}"/>
    <expression generic="@[begin-table-row]"
                destination=""/>
    <expression generic="@[end-table-row]"
                destination="\\"/>
    <expression generic="@[begin-table-cell]"
                destination=""/>
    <expression generic="@[end-table-cell]"
                destination="&"/>
  </language>
</transform>
```

ABBILDUNG 5.10.: Beispiel für $\text{PEP}_{generic}$

Allerdings ist offensichtlich, daß diese Formulierung durch die mehrfachen Ausgabeanweisungen recht unübersichtlich und somit auch leicht fehleranfällig ist: Allein in dem kleinen Beispiel steckt bereits ein Fehler, der leicht zu übersehen ist.

Jedoch bietet `printf` auch die Möglichkeit, über Formatanweisungen und weitere Argumente eine gesamte Zeichenkette formatiert auszugeben. Das vorangegangenene Beispiel sähe dann so aus:

```
printf("Die nächste Veranstaltung findet am %s statt.", tag());
```

Die Zeichenfolge `%s` fungiert dabei als Platzhalter für die durch die Funktion `tag()` gelieferte Zeichenfolge. Die Ausgabe wird nicht zerrissen, was die Formulierung viel übersichtlicher erscheinen läßt.

Ganz ähnlich verhält es sich auch mit dem *Backquote* in LISP, nur daß sich das Backquote nicht auf Ausgabefunktionen, sondern auf Listen bezieht. Die Anweisung

```
(list (quote Die) (quote naechste) (quote Veranstaltung)
      (quote findet) (quote am) (tag) (quote statt))
```

kann mit Hilfe des Backquote folgendermaßen formuliert werden:

```
`(Die naechste Veranstaltung findet am ,(tag) statt)
```

Bei der Implementierung von PEP wurde diese Technik für die Templates verwendet.

PEP$_{template}$ Es wäre nicht sinnvoll, bei der Formulierung der Templates die jeweiligen Zielsprachen-Dokumente in einzelne Textblöcke zu zerteilen und in eine in XML formulierte Regelsprache einzufügen. Stattdessen werden in die Dokumente Platzhalter in Gestalt der bereits vorgestellten generischen Ausdrücke eingefügt, die auf einzufügende Inhalte verweisen.

Ein solches Template sieht z.B. so aus wie das in Abbildung 5.11

Die Sprache besteht aus folgenden Komponenten:

- **Text** Beliebiger Text, kann generische Ausdrücke enthalten.
- **Generischer Ausdruck** Entspricht den in dieser Arbeit eingeführten generischen Ausdrücken und hat die Form `@[OBJECT("`*<Referenz auf Inhalt>*`")]`.

  Die Referenz verweist auf den Wert eines ID-Attributes im Layoutbaum.

5.3.2. Prozessoren mit mehreren Ausprägungen

Für manche Aufgabenstellungen reicht es nicht aus, wenn es nur einen einzigen Prozessor dafür gibt. Betrachtet man zum Beispiel die Übersetzung der generischen Ausdrücke in Zielsprachen-Äquivalente, so fällt auf, daß es nicht nur eine einzige mögliche Übersetzung gibt, sondern für jede Zielsprache

```
<html>
  <head>
    <meta content="text/html; charset=ISO-8859-1"
          http-equiv="Content-Type" />
  </head>
  <body>
    <h1>Lehrveranstaltungen des Informatik-Studiums</h1>
     <b>Bitte beachten Sie die veranderten Zeitangaben!</b><br />
     <h2>@[OBJECT("Ueberschrift")]</h2>
     <hr />
     @[OBJECT("Veranstaltungen")]
   </body>
 </html>
```

ABBILDUNG 5.11.: Beispiel für PEP$_{template}$

eine. Oder bei der Verwendung von Templates: dort kann es nicht nur ein einziges Template geben, sondern beliebig viele.

In PEP wurde dies so implementiert, daß es einen gemeinsamen Prozessor gibt, der zunächst für diese Aufgabenstellung zuständig ist und anhand eines Kommandozeilenparameters entscheidet, welchem der alternativen Prozessoren er die Transformation übergibt.

Diese alternativen Prozessoren befinden sich in einem Unterverzeichnissen des gemeinsamen Prozessors, die nach der speziellen Aufgabenstellung benannt sind.

Bei dem Prozessor für die generischen Ausdrücke sieht die Verzeichnisstruktur so aus wie in Abbildung 5.12 gezeigt.

```
generischeausdruecke/
   html/
   latex/
   text/
```

ABBILDUNG 5.12.: Beispiel für Verzeichnisstruktur von Prozessoren mit mehreren Ausprägungen

5.3.3. Die implementierten Prozessoren

Im Folgenden wird kurz die Funktionsweise der einzelnen Prozessoren erläutert.

Layoutbaum Erzeugt aus einem Layoutbeschreibungsbaum, der entweder in PEP$_{layout}$ (für XSLT und fxt), in XQuery oder in xcerpt formuliert ist, und aus einem Eingabedokument einen Layoutbaum.

Pruning Entfernt leere Teiläste.

Trenntexte hinzufügen Fügt Elementen des Eingabebaums Attribute für Trenntexte hinzu. Die Regeln für die hinzuzufügenden Trenntexte sind in PEP$_{separators}$ formuliert.

Trenntexte auflösen Fügt gemäß den Attributen für Trenntexte vor, zwischen und nach den jeweiligen Elementen neue Elemente ein, die die Trenntexte als Inhalte enthalten.

Template Transformiert einen Layoutbaum gemäß einem in PEP$_{template}$ formulierten Template in einen neuen Layoutbaum.

Flatten Transformiert einen Eingabebaum in einen Baum der Tiefe 1, der nur noch das Wurzelelement und die Blätter des Layoutbaumes enthält.

Generische Ausdrücke Übersetzt generische Ausdrücke des Eingabebaums in unterschiedliche Zielsprachen. Die Zielsprache wird durch einen Kommandozeilenparameter (html, latex, . . .) ausgewählt. Die Regeln für die Übersetzung sind in PEP$_{generic}$ formuliert.

Serialisierung Transformiert einen Eingabebaum in eine Zeichenkette mit den konkatenierten Werten aller Blätter.

Pretty printing Transformiert einen Eingabebaum in einen gleichen, aber neu formatierten Baum.

5.4. Vergleich der Werkzeuge am Beispiel des Prozessors zur Erzeugung des Layoutbaums

Um die unterschiedlichen Werkzeuge vergleichen zu können, wurde der Prozessor zur Erzeugung des Layoutbaums jeweils für die einzelnen Werkzeuge implementiert. Daß genau dieser Prozessor für den Vergleich herangezogen wurde, lag daran, daß die Transformationen, die er durchführt, mit die komplexesten von PEP sind.

Dabei stellte sich allerdings bald heraus, daß eine direkte Formulierung in XSLT und fxt nicht den Anforderungen an einen Layoutbeschreibungsbaum entsprach (siehe 4.1.2), was zu dem Entwurf von PEP$_{layout}$ führte, das dann nach XSLT bzw. fxt übersetzt werden kann. XQuery und xcerpt hingegen entsprachen den Anforderungen.

So müßten eigentlich korrekterweise zum einen Vergleiche zwischen XSLT und fxt und zum anderen zwischen XQuery, xcerpt und PEP$_{layout}$ gemacht werden.

Um hier jedoch einen Vergleich zwischen allen Werkzeugen ziehen zu können, wurde die direkte Transformation von XQuery und xcerpt und die indirekte Transformation von XSLT und fxt als gleich betrachtet.

Für HaXml standen leider im Testzeitraum keine geeigneten Werkezuge zur Verfügung, so daß auf einen Vergleich verzichtet werden mußte.

5.4.1. XSLT

XSLT eignet sich nicht besonders gut zur direkten Formulierung eines Layoutbeschreibungsbaumes: Durch das Konzept, für jeden Selektor eine eigene Regel angeben zu müssen, würde der Layoutbeschreibungsbaum auf viele einzelne Regeln verteilt, was der Übersichtlichkeit sehr abträglich wäre.

Stattdessen wurden Regeln formuliert, die aus dem in PEP$_{layout}$ definierten Layoutbeschreibungsbaum wieder XSLT-Regeln erzeugen. Da diese Regeln nur ein einziges Mal formuliert werden müssen und dann auf alle Ausprägungen von PEP$_{layout}$ anwendbar sind, spielt dabei diesmal weniger die Einfachheit der Regeln eine Rolle, sondern vielmehr die Mächtigkeit von XSLT.

Die Transformation in erneute Regeln wird in XSLT recht leicht gemacht: XSLT sieht bereits die Möglichkeit vor, Namespace Aliase zu verwenden, die bei der Transformation in den entsprechenen Namespace übertragen werden. Somit schreibt sich eine zu erzeugende Regel z.B. so:

```
<axsl:template match="xyz">
  ...
</axsl:template>
```

Was beinahe der ursprünglichen Schreibweise entspricht.

Für das Wurzelelement und die iterierten Strukturierungsknoten wird jeweils eine eigene Regel eingefügt. Daneben müssen aber in die Regeln noch die einfachen Strukturierungsknoten und die Blätter eingefügt werden.

Dieses verschränkte Einfügen ist in XSLT mit einem einzigen Durchlauf nicht zu machen, sondern es müssen zuerst in jede Regel ihre enthaltenen Strukturierungsknoten eingefügt werden und dann in einem weiteren Durchlauf neue Regeln für möglicherweise vorgekommene iterierte Strukturierungsknoten erzeugt werden.

Dieser mehrmalige Aufruf wird in XSLT durch die *modes* unterstützt, mit denen Regeln mit gleichem Selektor unterschieden werden können.

Wenn im Layoutbeschreibungsbaum an unterschiedlichen Stellen iterierte Selektierungsknoten mit dem gleichen Selektor vorkommen, spielen die *modes* ebenfalls eine wichtige Rolle: Normalerweise würden zwei Regeln mit dem gleichen Selektor erzeugt werden, bei der zweiten Transformation würde aber nur eine davon verwendet werden bzw. ein Fehlerzustand auftreten. D.h. es würden, selbst wenn die Transformation durchgeführt würde, in den Layoutbaum falsche Strukturierungsknoten eingefügt. Die Lösung ist naheliegend: Wenn für jeden iterierten Strukturierungsknoten ein eigener *mode* verwendet wird, wird die richtige Regel verwendet.

In PEP$_{layout}$ wird auf drei unterschiedliche Arten auf einzufügende Daten verwiesen:

1. Durch ihre strukturelle Position in den Eingabedaten. Also z.B. ein Element `Dozent`, daß ein Vaterelement `Dozenten` hat. (In PEP$_{layout}$ sind das die `selector`-Attribute.)
2. Durch einen berechneten Ausdruck, der wiederum mehrere Verweise wie in 1. enthalten kann, z.B. die Summe der Inhalte der Elemente `Stunde` und `Dauer`. (In PEP$_{layout}$ sind das die `expression`-Attribute.)
3. Durch eine Referenz auf den Wert eines `id`-Attributes von ihnen, z.B. ein Element, das ein `id`-Attribut mit dem Wert “Info1” besitzt. (In PEP$_{layout}$ sind das die `reference`-Attribute.)

Die ersten beiden Arten werden in XSLT durch XPath 1.0 ermöglicht, die dritte durch einen *key*-Mechanismus.

Obwohl in XPath die Formulierung aller notwendigen Verweise möglich ist, enthält es doch eine Stolperfalle: Die XPath-Ausdrücke beziehen sich nicht zwingend auf den aktuellen Kontext des Eingabedokuments, sondern können auf beliebige Knoten im Eingabedokument verweisen – auch welche in einem anderen Teilbaum oder weiter oben im Baum. Dies erfordert besondere Sorgfalt bei der Formulierung der Selektoren. Andererseits ist man dadurch nicht darauf festgelegt, sich in einem Teilbaum nur abwärts zu bewegen, sondern kann in iterierten Strukturierungsknoten und Werten auf beliebige Elemente verweisen.

Der Zugriff auf Attributwerte fällt unter XSLT recht leicht: Das entsprechende Attribut wird über einen XPath-Ausdruck an eine Variable gebunden, die dann beliebig verwendet werden kann. Ist das Attribut nicht vorhanden, enthält die Variable eine leere Zeichenkette.

5.4.1.1. Transformationsanweisungen in XSLT zur Generierung von weiteren Transformationsanweisungen in XSLT, die den Layoutbaum erzeugen

Mit den folgenden, in XSLT formulierten, Transformationsanweisungen werden aus einem in PEP_{layout} definierten Layoutbeschreibungsbaum weitere XSLT-Transformationsanweisungen generiert. Mit diesen kann darauf ein entsprechendes XML-Dokument in einen Layoutbaum transformiert werden.

```
<xsl:stylesheet version="1.0"
                xmlns:axsl="http://www.w3.org/1999/XSL/TransformAlias"
                xmlns:pep="http://www.pms.informatik.uni-muenchen.de/xml/pep"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output encoding="ISO-8859-1" indent="yes" method="xml" />
  <xsl:namespace-alias result-prefix="xsl" stylesheet-prefix="axsl" />

  <!-- Template fuer das Wurzelelement: Rahmen fuer zu erstellendes
  Stylesheet und erstes Template darin erstellen -->
  <xsl:template match="/">
    <axsl:stylesheet version="1.0">
      <axsl:output encoding="ISO-8859-1" indent="yes" method="xml" />
      <axsl:key match="*[@id]" name="ID" use="@id" />
      <axsl:template match="*" />
      <axsl:template match="/">
        <xsl:apply-templates mode="fill-templates" />
      </axsl:template>
      <xsl:apply-templates mode="build-templates" select="//*[@pep:selector]" />
    </axsl:stylesheet>
  </xsl:template>

  <!-- Build-In-Templates ueberschreiben, da diese standardmaessig
  alle Knoten auswerten -->
  <xsl:template match="*">
  </xsl:template>

  <xsl:template match="*" mode="build-templates">
  </xsl:template>

  <xsl:template match="*" mode="fill-templates" priority="-1.0">
    <xsl:copy>
      <xsl:copy-of select="@id" />
      <xsl:apply-templates mode="fill-templates" />
    </xsl:copy>
  </xsl:template>

  <!-- Template um die Strukturierungsknoten zu erzeugen -->
  <xsl:template match="*[@pep:selector]" mode="fill-templates">
    <xsl:variable name="selector" select="@pep:selector" />
    <xsl:variable name="myname" select="name(.)" />
    <axsl:apply-templates mode="$myname" select="$selector" />
  </xsl:template>
```

```
  <!-- Templates fuer die Blaetter -->
  <xsl:template match="//pep:value[@selector]" mode="fill-templates">
    <xsl:variable name="selector" select="@selector|@pep:selector" />
    <axsl:copy-of select="$selector" />
  </xsl:template>

  <xsl:template match="//pep:value[text()]" mode="fill-templates">
    <xsl:copy-of select="text()" />
  </xsl:template>

  <xsl:template match="//pep:value[@expression|@pep:expression]"
                mode="fill-templates">
    <xsl:variable name="expression" select="@expression|@pep:expression" />
    <axsl:value-of select="$expression" />
  </xsl:template>

  <xsl:template match="//pep:value" mode="fill-templates" priority="-0.5">
    <axsl:copy-of select="text()" />
  </xsl:template>

  <xsl:template match="//pep:value[@reference|@pep:reference]"
                mode="fill-templates">
    <xsl:variable name="reference" select="@reference|@pep:reference" />
    <axsl:copy-of select="key('ID','$reference')" />
  </xsl:template>

  <!-- Template fuer die Erstellung der Templates -->
  <xsl:template match="*[@pep:selector]" mode="build-templates">
    <xsl:variable name="selector" select="@pep:selector" />
    <xsl:variable name="myname" select="name(.)" />
    <axsl:template match="$selector" mode="$myname">
      <xsl:copy>
        <xsl:apply-templates mode="fill-templates" />
      </xsl:copy>
    </axsl:template>
    <xsl:apply-templates mode="build-templates" />
  </xsl:template>

</xsl:stylesheet>
```

5.4.1.2. Generierte Transformationsanweisungen in XSLT, die den Layoutbaum erzeugen

```
<axsl:stylesheet version="1.0"
                 xmlns:axsl="http://www.w3.org/1999/XSL/Transform"
                 xmlns:pep="http://www.pms.informatik.uni-muenchen.de/xml/pep"
                 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <axsl:output encoding="ISO-8859-1" indent="yes" method="xml" />

  <axsl:key match="*[@id]" name="ID" use="@id" />

  <axsl:template match="*" />

  <axsl:template match="/">
    <Praesentation>
```

```
      <Ueberschrift id="Ueberschrift">Veranstaltungen</Ueberschrift>
      <Veranstaltungen id="Veranstaltungen">
        <axsl:apply-templates mode="Veranstaltung"
                              select="/Lehrangebot/Veranstaltungen/Veranstaltung" />
      </Veranstaltungen>
    </Praesentation>
  </axsl:template>

  <axsl:template match="/Lehrangebot/Veranstaltungen/Veranstaltung"
                 mode="Veranstaltung">
    <Veranstaltung>
      <Beschreibung>
        <Benennung>
          <Titel>
            <axsl:copy-of select=".//Titel/text()" />
          </Titel>
          <Bereiche>
            <axsl:apply-templates mode="Bereich" select="Bereiche/Bereich" />
          </Bereiche>
        </Benennung>
        <Rumpf>
          <Dauer>
            <axsl:copy-of select="Dauer/text()" />
          </Dauer>
          <Sitzungen>
            <axsl:apply-templates mode="Sitzung" select="Sitzungen/Sitzung" />
          </Sitzungen>
        </Rumpf>
      </Beschreibung>
      <Dozenten>
        <axsl:apply-templates mode="Dozent" select="Dozenten/Dozent" />
      </Dozenten>
    </Veranstaltung>
  </axsl:template>

  <axsl:template match="Bereiche/Bereich" mode="Bereich">
    <Bereich>
      <axsl:copy-of select="text()" />
    </Bereich>
  </axsl:template>

  <axsl:template match="Sitzungen/Sitzung" mode="Sitzung">
    <Sitzung>
      <Zeit>
        <Tag>
          <axsl:copy-of select=".//Tag/text()" />
        </Tag>
        <Zeitraum>
          <Von>
            <axsl:copy-of select=".//Stunde/text()" />
          </Von>
          <Bis>
            <axsl:value-of select="sum((.//Stunde/text()|.//Dauer/text()))" />
          </Bis>
        </Zeitraum>
      </Zeit>
      <Ort>
```

```
          <Gebaeude>
            <axsl:copy-of select=".//Gebaeude/text()" />
          </Gebaeude>
          <Raum>
            <axsl:copy-of select=".//Raum/text()" />
          </Raum>
        </Ort>
      </Sitzung>
    </axsl:template>

    <axsl:template match="Dozenten/Dozent" mode="Dozent">
      <Dozent>
        <axsl:copy-of select=".//Name/text()" />
      </Dozent>
    </axsl:template>
  </axsl:stylesheet>
```

5.4.2. fxt

Was im vorangeganenen Abschnitt zu XSLT über die direkte Formulierung eines Layoutbeschreibungsbaumes gesagt wurde, gilt ebenso für fxt, da fxt einen ähnlichen regelbasierten Ansatz besitzt.

Leider kennt fxt keine Namespace Aliase, so daß zu erzeugende Regeln umständlich mit Elementen definiert werden müssen, die berechnete Elemente oder Attribute erzeugen. (Ein ähnliches Konzept bietet XSLT übrigens ebenfalls an.) Das Beispiel aus dem XSLT-Abschnitt würde sich dann so schreiben:

```
<fxt:tag fxt:name='String2Vector "fxt:pat"'>
  xyz
</fxt:tag>
...
```

Es ist zwar in diesem Fall nicht viel länger, aber um einiges schwerer zu lesen. Zumal fxt noch eine Typkonvertierung in das interne Vector-Format erfordert.

Für das verschränkte Einfügen von Strukturierungsknoten und Regeln gibt es in fxt ein ganz anderes Konzept als in XSLT: Globale Variablen mit Kellerprinzip (push und pop von Werten), deren Werte auch Wälder sein können und die (anders als in XSLT) geändert werden können. Damit reduzieren sich die mehreren Durchläufe von XSLT auf einen einzigen, der folgendermaßen abläuft:

1. Die Strukturierungsknoten des Layoutbeschreibungsbaumes werden rekursiv durchlaufen,
2. für das Wurzelelement und jeden iterierten Strukturierungsknoten wird
   - eine neue Regel erzeugt, deren Inhalte durch den fortgesetzten Durchlauf gebildet werden,
   - diese Regel in eine globale Variable eingefügt und
   - ein Element erzeugt, das diese Regel aufruft,
3. für jeden einfachen Strukturierungsknoten und jedes Blatt wird einfach ein entsprechendes Element erzeugt und, wenn Kindknoten vorhanden sind, mit dem Durchlauf fortgefahren.
4. Zuletzt werden die erzeugten Regeln in das Zieldokument eingefügt.

Modes werden von fxt nicht unterstützt, so daß mehrfach vorkommende Selektoren im Layoutbeschreibungsbaum nur mit einigem Aufwand realisierbar wären (iterierte Strukturierungsknoten würden keine ganzen Regeln mehr erzeugen dürfen, sondern nur bedingte Blöcke in einer gemeinsamen Regel für jeden Selektor).

Der Verweis auf die einzufügenden Daten ist in fxt recht ähnlich wie in XSLT, nur, daß anstatt von XPath hier fxgrep als Selektionssprache und SML als Sprache zur Berechnung von Ausdrücken verwendet wird. Auch ein ähnlicher Key-Mechanismus ist vorhanden.

Allerdings ist das Selektionsprinzip von fxgrep durch seine regulären Ausdrücke auf Bäume um einiges mächtiger und trotzdem besser verständlich als das von XPath.

Auch bezieht sich fxgrep immer auf den aktuellen Kontext im Baum und läßt keine Aufwärtsbewegung zu Vorgängern des aktuellen Knotens zu. Dadurch wird die Formulierung von Selektoren einfacher, allerdings schließt sie eventuell manche Aufgabenstellungen aus, bei denen Knoten aus einem anderen Teilbaum nötig wären. Normalerweise dient hier das Variablenkonzept als Lösung. Bei der Erstellung von Regeln über Regeln sind solche Sonderfälle jedoch nur schwer zu handhaben.

Der Zugriff auf Attributwerte erfolgt unter fxt über Ausdrücke, die in SML formuliert werden. Dieses Konzept ist an sich recht mächtig, allerdings tritt bei dem Zugriff auf nicht vorhandene Variablen sofort ein Fehlerzustand auf, sodaß immer erst eine Überprüfung stattfinden sollte, ob die Variable vorhanden ist, was die Regeln unnötigerweise aufbläht. Auch ist, die oftmals nötige Typkonvertierung in SML-Typen recht lästig, hier wäre eine automatische Konvertierung sinnvoll gewesen.

5.4.2.1. Transformationsanweisungen in fxt zur Generierung von weiteren Transformationsanweisungen in fxt, die den Layoutbaum erzeugen

Mit den folgenden, in fxt formulierten, Transformationsanweisungen werden aus einem in PEP$_{layout}$ definierten Layoutbeschreibungsbaum weitere fxt-Transformationsanweisungen generiert. Mit diesen kann darauf ein entsprechendes XML-Dokument in einen Layoutbaum transformiert werden.

```
<fxt:spec>
  <fxt:global name="actions" type="Forest" />
  <fxt:push name="actions" val="emptyForest" />
  <fxt:global name="root" type="Forest" />
  <fxt:push name="root" val="emptyForest" />
  <fxt:global name="keys" type="Forest" />
  <fxt:push name="keys" val="emptyForest" />

  <!-- Startregel: Rahmen und Regel fuer Startpattern erzeugen und
  erstellte Regeln einfuegen -->
  <fxt:pat>/*</fxt:pat>
  <fxt:tag fxt:name="String2Vector "fxt:spec"">
    <fxt:setForest name="root">
      <fxt:tag fxt:name="String2Vector "fxt:pat"">/*</fxt:tag>
      <fxt:copyTag>
        <fxt:apply />
      </fxt:copyTag>
      <fxt:tag fxt:name="String2Vector "fxt:pat"">default</fxt:tag>
    </fxt:setForest>
    <fxt:get name="keys" />
    <fxt:get name="root" />
    <fxt:get name="actions" />
```

```
</fxt:tag>

<!-- Regel zum Einfuegen von Selektoren auf Werte -->
<fxt:pat>//pep:value[@selector]</fxt:pat>
<fxt:tag fxt:name="String2Vector "fxt:copyContent"">
  <fxt:addAttribute name="select"
                    valExp="getAttribute (String2Vector "selector") current" />
</fxt:tag>

<!-- Regel zum Einfuegen von Selektoren auf Elemente -->
<fxt:pat>//pep:value[@reference]</fxt:pat>
<fxt:setForest name="keys">
  <fxt:tag fxt:name="String2Vector "fxt:key"">
    <fxt:addAttribute name="name"
                   valExp="getAttribute (String2Vector "reference") current" />
    <fxt:addAttribute name="select" val="//*[@id]" />
    <fxt:addAttribute name="key" val="id" />
  </fxt:tag>
  <fxt:get name="keys" />
</fxt:setForest>
<fxt:tag fxt:name="String2Vector "fxt:copyKey"">
  <fxt:addAttribute name="name"
                 valExp="getAttribute (String2Vector "reference") current" />
  <fxt:addAttribute name="keyExp"
                    valExp="concatVectors((String2Vector "String2Vector "),
                      (getAttribute (String2Vector "reference") current))" />
</fxt:tag>

<!-- Regel zum Einfuegen von Ausdruecken -->
<fxt:pat>//pep:value[@expression]</fxt:pat>
<fxt:tag fxt:name="String2Vector "fxt:text"">
  <fxt:addAttribute name="code"
                 valExp="getAttribute (String2Vector "expression") current"/>
</fxt:tag>

<!-- Regel zum Einfuegen von Textinhalt -->
<fxt:pat>//pep:value[""]</fxt:pat>
<fxt:copyContent />

<!-- Regel zum Einfuegen des Inhalts des aktuellen Kontextes -->
<fxt:pat>//pep:value</fxt:pat>
<fxt:tag fxt:name="String2Vector "fxt:copyContent"" />

<!-- Regel zum Einfuegen der Strukturierungsknoten mit Iteration -->
<fxt:pat>//*[@pep:selector]</fxt:pat>
<!-- Regel erzeugen -->
<fxt:setForest name="actions">
  <fxt:tag fxt:name="String2Vector "fxt:pat"">
    <fxt:text code="getAttribute (String2Vector "pep:selector") current" />
  </fxt:tag>
  <fxt:copyTag>
    <fxt:deleteAttribute name="pep:selector" />
    <fxt:apply />
  </fxt:copyTag>
  <fxt:get name="actions" />
</fxt:setForest>
<!-- apply in umgebende Struktur einfuegen -->
```

```
  <fxt:tag fxt:name="String2Vector "fxt:apply"">
    <fxt:addAttribute name="select"
                  valExp="getAttribute (String2Vector "pep:selector") current" />
  </fxt:tag>

  <!-- Regel zum Einfuegen der Strukturierungsknoten ohne Iteration -->
  <fxt:pat>//*</fxt:pat>
  <fxt:copyTag>
    <fxt:apply />
  </fxt:copyTag>

  <fxt:pat>default</fxt:pat>
</fxt:spec>
```

5.4.2.2. Generierte Transformationsanweisungen in fxt, die den Layoutbaum erzeugen

```
<fxt:spec>
  <fxt:pat>/*</fxt:pat>
  <Praesentation>
    <Ueberschrift id="Ueberschrift">Veranstaltungen</Ueberschrift>
    <Veranstaltungen id="Veranstaltungen">
      <fxt:apply select="//Veranstaltung" />
    </Veranstaltungen>
  </Praesentation>

  <fxt:pat>default</fxt:pat>

  <fxt:pat>//Veranstaltung</fxt:pat>
  <Veranstaltung>
    <Beschreibung>
      <Benennung>
        <Titel>
          <fxt:copyContent select="//Titel/""" />
        </Titel>
        <Bereiche>
          <fxt:apply select="//Bereich">
          </fxt:apply>
        </Bereiche>
      </Benennung>
      <Rumpf>
        <Dauer>
          <fxt:copyContent select="Dauer/""" />
        </Dauer>
        <Sitzungen>
          <fxt:apply select="//Sitzung" />
        </Sitzungen>
      </Rumpf>
    </Beschreibung>
    <Dozenten>
      <fxt:apply select="//Dozent" />
    </Dozenten>
  </Veranstaltung>

  <fxt:pat>//Dozent</fxt:pat>
  <Dozent>
```

```
    <fxt:copyContent select="//Name/""" />
  </Dozent>

  <fxt:pat>//Sitzung</fxt:pat>
  <Sitzung>
    <Zeit>
      <Tag>
        <fxt:copyContent select="//Tag/""" />
      </Tag>
      <Zeitraum>
        <Von>
          <fxt:copyContent select="//Stunde/""" />
        </Von>
        <Bis>
          <fxt:text code="String2Vector "Bis"" />
        </Bis>
      </Zeitraum>
    </Zeit>
    <Ort>
      <Gebaeude>
        <fxt:copyContent select="//Gebaeude/""" />
      </Gebaeude>
      <Raum>
        <fxt:copyContent select="//Raum/""" />
      </Raum>
    </Ort>
  </Sitzung>

  <fxt:pat>//Bereich</fxt:pat>
  <Bereich>
    <fxt:copyContent />
  </Bereich>
</fxt:spec>
```

5.4.3. XQuery

Dadurch, daß Transformationen in XQuery in sehr deklarativer Weise angegeben werden können, konnte der Layoutbeschreibungsbaum direkt in XQuery definiert werden, und es konnte darauf verzichtet werden, ihn zunächst in PEP_{layout} zu definieren, um ihn in einem anschließendem Übersetzungschritt in die jeweilige Transformationssprache übersetzen zu müssen. Der Vorteil davon ist, daß eine wesentlich mächtigere Transformationssprache als die sehr spezialisierte Sprache von PEP_{layout} zur Verfügung steht, und daß ein Vorverarbeitungsschritt wegfällt.

Nachdem XQuery zur Selektion XPath 2.0 verwendet (eine Weiterentwicklung von dem in XSLT verwendeten XPath 1.0), gilt im Prinzip das Gleiche, was im Abschnitt zu XSLT zur Selektion von einzufügenden Daten gesagt wurde. Allerdings können die Variablen, die mit `for` oder `let` gebunden wurden, ebenfalls in den Selektoren verwendet werden, so daß die Selektoren wesentlich knapper und intuitiver formuliert werden können.

Was negativ auffällt ist die Selektion von Textinhalten: Selektiert man ein Element z.B. über den Selektor `$sitzung/tag`, so erhält man das Element `tag` zurück. Den Textinhalt des Elements `tag` erhält man über den Funktionsaufruf `text($sitzung/tag)`. Hier ist die Lösung von fxgrep `sitzung/tag/""` wesentlich intuitiver.

5.4.3.1. Transformationsanweisungen in XQuery, die den Layoutbaum erzeugen

```
<Praesentation>
  <Ueberschrift id="Ueberschrift">Veranstaltungen</Ueberschrift>
  <Veranstaltungen id="Veranstaltungen">
    {for $veranstaltung in
      document("/dev/stdin")/Lehrangebot/Veranstaltungen/Veranstaltung
    let $dauer := $veranstaltung/Dauer
    return
    <Veranstaltung>
      <Beschreibung>
        <Benennung>
          {for $titel in $veranstaltung/Titel
          return
          <Titel>{string($titel)}</Titel>
          }
          <Bereiche>
            {for $bereich in $veranstaltung/Bereiche/Bereich
            return
            <Bereich>{string($bereich)}</Bereich>
            }
          </Bereiche>
        </Benennung>
        <Rumpf>
          {$dauer}
          <Sitzungen>
            {for $sitzung in $veranstaltung/Sitzungen/Sitzung
            let $tag := $sitzung/Tag
            let $von := $sitzung/Stunde
            let $dauer := $sitzung/Dauer
            let $gebaeude := $sitzung/Gebaeude
            let $raum := $sitzung/Raum
            return
            <Sitzung>
              <Zeit>
                {$tag}
                <Zeitraum>
                  <Von>{string($von)}</Von>
                  <Bis>{number($von) + number($dauer)}</Bis>
                </Zeitraum>
              </Zeit>
              <Ort>
                {$gebaeude}
                {$raum}
              </Ort>
            </Sitzung>
            }
          </Sitzungen>
        </Rumpf>
      </Beschreibung>
      <Dozenten>
        {for $dozent in $veranstaltung/Dozenten/Dozent
        let $name := $dozent/Name
        return
        <Dozent>{string($name)}</Dozent>
        }
      </Dozenten>
```

```
    </Veranstaltung>
    }
  </Veranstaltungen>
</Praesentation>
```

5.4.4. xcerpt

xcerpt ist ebenfalls ein Ansatz, bei dem Transformationen auf eine sehr deklarative Weise angegeben werden können. Was zusätzlich zur Klarheit der angegebenen Regeln beiträgt ist der Umstand, daß in xcerpt die Selektion der Daten und die Konstruktion in zwei getrennten Teilen der Regel formuliert wird.

Auch verwendet xcerpt einen ganz anderen Mechanismus zur Selektion als die bisherigen Ansätze: Anstatt zu versuchen, komplexe Anfragen auf Inhalte eines XML-Eingabe-Dokuments in vielen linearen Anfrageausdrücken zu formulieren, besteht der Selektionsteil einer xcerpt-Anfrage aus einer Baumstruktur, die der Struktur des Eingabe-Dokuments, bzw. Teilstrukturen davon, entspricht; Inhalte werden selektiert, indem an der entsprechenden Stelle in der Baumstruktur eine Variable definiert wird.

Die Struktur des Konstruktionsteils entspricht der Struktur des XML-Ausgabe-Dokuments (oder Teilen davon bei mehreren Regeln). Die selektierten Daten werden eingefügt, indem die jeweilige Variable an der Stelle steht, an der eingefügt werden soll.

Um Iterationen über mehrere mögliche Ausprägungen von Variablen durchzuführen, können an beliebigen Stellen in der Struktur des Konstruktionsteiles `all`-Quantifizierer stehen. Für jede Ausprägung wird der Teilbaum unter dem Quantifizierer in das Ausgabedokument eingefügt.

Aufgrund des frühen Stadiums von xcerpt gibt es allerdings noch einige Einschränkungen:

So sind nur Anfragen erfolgreich, die dem Selektionsteil *vollständig* entsprechen. Fehlt nur ein einziges Unterelement, so liefert die Selektion nichts zurück. Allerdings werden unterschiedliche Lösungsmöglichkeiten wie Defaultwerte und Alternativen bereits untersucht.

Quantifizierer benötigen immer ein eindeutiges Element, auf das sie sich beziehen. Die Möglichkeit, daß sich Quantifizierer auf Attribute (z.B. das eindeutige Attribut `id`) beziehen, ist noch nicht vorhanden.

Das Auflösen von Keys ist zwar über das Gleichstellen zweier Variablen möglich, allerdings fehlt ein Mechanismus, der das Auflösen erledigt.

5.4.4.1. Transformationsanweisungen in xcerpt, die den Layoutbaum erzeugen

```
construct
<Praesentation>
  <Ueberschrift id="Ueberschrift">Veranstaltungen</Ueberschrift>
    <Veranstaltungen id="Veranstaltungen">
      all <Veranstaltung>
        <Beschreibung>
          <Benennung>
            <Titel>var Titel</Titel>
          <Bereiche>
            all <Bereich>var Bereich</Bereich>
          </Bereiche>
```

```
        </Benennung>
        <Rumpf>
          <Dauer>var Vdauer</Dauer>
          <Sitzungen>
            all <Sitzung>
              <Zeit>
                <Tag>var Tag</Tag>
                <Zeitraum>
                  <Von>var Stunde</Von>
                  <Bis>var Sdauer</Bis>
                </Zeitraum>
              </Zeit>
              <Ort>
                <Gebaeude>var Gebaeude</Gebaeude>
                <Raum>var Raum</Raum>
              </Ort>
            </Sitzung>
          </Sitzungen>
        </Rumpf>
      </Beschreibung>
      <Dozenten>
        all <Dozent>
          <Name>var Dozentname</Name>
        </Dozent>
      </Dozenten>
    </Veranstaltung>
  </Veranstaltungen>
</Praesentation>

where
<Lehrangebot>
  <Veranstaltungen>
    <Veranstaltung>
      <Titel>var Titel</Titel>
      <Bereiche>
        <Bereich>var Bereich</Bereich>
      </Bereiche>
      <Dauer>var Vdauer</Dauer>
      <Dozenten>
        <Dozent>
          <Name>var Dozentname</Name>
        </Dozent>
      </Dozenten>
      <Sitzungen>
        <Sitzung>
          <Tag>var Tag</Tag>
          <Stunde>var Stunde</Stunde>
          <Dauer>var Sdauer</Dauer>
          <Gebaeude>var Gebaeude</Gebaeude>
          <Raum>var Raum</Raum>
        </Sitzung>
      </Sitzungen>
    </Veranstaltung>
  </Veranstaltungen>
</Lehrangebot>
```

6. Zusammenfassung und Ausblick

Mit dieser Arbeit wurde ein Prozeß entwickelt (der *Präsentations-Erzeugungsprozeß*), der (semi) strukturierte Daten in Präsentationen unterschiedlicher Beschreibungssprachen, Layouts und Landessprachen transformiert – mit dem Fokus auf den Verwaltungsdaten des Lehrangebots einer Universitätseinrichtung.

Dabei wurde großer Wert darauf gelegt, daß der Prozeß leicht an sich verändernde Rahmenbedingungen und Aufgabenstellungen angepaßt werden kann.

Deshalb wurde der Prozeß in kleinere Einheiten untergliedert: zunächst in die drei Prozeßabschnitte „Daten", „Layout" und „Style", diese dann weiter in Prozeßschritte und diese wiederum in einzelne Prozessoren.

Der Grund für die Unterteilung in die drei Prozeßabschnitte war, daß der Prozeß zunächst von den zugrundeliegenden Daten und den zu erstellenden Präsentationen weitgehend unabhängig gemacht werden sollte.

Der erste Prozeßabschnitt „Daten" dient dazu, die Daten aus einer beliebigen Datenquelle in das XML-Format zu übertragen und nach unterschiedlichen Gesichtspunkten aufzubereiten.

Der zweite Prozeßabschnitt „Layout" dient dazu, die derart aufbereiteten Daten in eine semantisch reiches Zwischenformat, den Layoutbaum, zu übertragen, mit dem die Inhalte einer Präsentation beschrieben werden können. Diese Inhalte können weiter aufbereitet werden: so können ihnen z.B. über das Konzept der Trenntexte Informationen über ihre räumliche Anordnung hinzugefügt werden oder sie können umgruppiert werden.

Der dritte Prozeßabschnitt „Style" dient dazu, den Inhalten Informationen hinzuzufügen *wie* sie dargestellt werden sollen. Über das Konzept des Templatemechanismus können die Inhalte darauf in vorgefertigte Präsentationen eingefügt werden und über eine einfache Serialisierung des Layoutbaums – der zu diesem Zeitpunkt immer noch die verwendete Datenstruktur ist – kann dann zuletzt die Präsentation erstellt werden.

Um die Aufgaben, die innerhalb eines Prozeßabschnittes vorkommen, logisch nach ihrem Verwendungszweck zu gliedern, wurden die einzelnen Prozeßabschnitte in mehrere Prozeßschritte unterteit.

Und um nun den gesamten Prozeß durch Änderungen der Rahmenbedingungen und Aufgabestellungen letztendlich weitgehend unbeeinflußt zu lassen und Anpassungen lokal begrenzen zu können, wurde die weitere Unterteilung in Prozessoren vorgenommen.

Diese Prozessoren sind diejenigen Einheiten des Prozesses, die Transformationen auf den Datenstrukturen ausführen. Dafür nehmen sie Eingaben über eine einheitliche Schnittstelle entgegen, transformieren diese nach einer klar abgegrenzten Aufgabenstellung und geben das Ergebnis über eine einheitliche Schnittstelle wieder aus. Durch diese strenge Modularisierung können die Prozessoren problemlos ausgetauscht, weggelassen oder um weitere Prozessoren ergänzt werden.

Um die Tragfähigkeit dieses Ansatzes zu prüfen, wurde eine prototypische Implementierung des Prozesses vorgenommen:

Dazu wurde zunächst untersucht, welche Sprachen und Werkzeuge grundsätzlich für eine Implementierung nötig sind. Darauf aufbauend wurden mögliche Kandidaten der notwendigen Sprachen und Werkzeuge untersucht und miteinander verglichen.

Für die Implementierung selbst wurden einige interessante Prozessoren herausgegriffen und damit ein vereinfachter durchgängiger Prozeß realisiert. Daneben wurde ein ausgewählter Prozessor in unterschiedlichen Sprachen inplementiert und damit die Eignung der einzelnen Sprachen für die Implementierung des Prozesses verglichen.

Dabei hat sich gezeigt, daß eine Reihe von Ansätzen gerade im Entstehen ist, die für eine Implementierung des Prozesses sehr interessant sein könnten. Dies sind insbesondere xcerpt und XQuery für die Spezifikation der Transformationen der Prozessoren und XML Pipeline für die Spezifikation des Prozesses.

Mit der prototypischen Implementierung wurde deutlich, daß der Ansatz, der in dieser Arbeit entworfen wurde, leicht zu implementieren ist und die Anforderungen an den Prozeß durchaus erfüllt werden.

Was die in dieser Arbeit vorgestellten bzw. implementierten Prozessoren angeht, so muß gesagt sein, daß sie nur eine Teilmenge derjenigen Prozessoren bilden, die in einem tatsächlich einsetzbaren System notwendig wären. Es wäre interessant, eine Art Baukasten aus Prozessoren zu bilden, der einen Großteil der für die Erstellung von Präsentationen notwendigen Transformationen abdeckt.

Abbildungsverzeichnis

2.1. Beispiel zur Rekursion 12
2.2. Beispiel zur Vererbung/Verschattung 13
2.3. Beispiel zum Lehreinheitstyp 15

3.1. Unterteilung des Prozesses in Prozeßabschnitte 17
3.2. Unterteilung eines Prozeßabschnittes in Prozeßschritte 17
3.3. Unterteilung eines Prozeßschrittes in Prozessoren 17
3.4. Beispiele zu Layout vs. Style 21
3.5. Übersicht über den gesamten Präsentations-Erzeugungsprozeß 22
3.6. Programmbeispiel zur imperativen Vorgehensweise 28
3.7. Beispielausgabe zur imperativen Vorgehensweise 28
3.8. Exemplarischer Layoutbaum 30
3.9. Exemplarischer Layoutbeschreibungsbaum 31
3.10. Distributivgesetz für Zeit und Ort 33
3.11. Regeln für Trenntexte 33
3.12. Layoutbaum mit Trenntexten 34
3.13. Templatemechanismus 37
3.14. Hinzufügen von Style-Informationen 38
3.15. Übersetzung generischer Ausdrücke 39
3.16. Transformation in Zielsprache (LaTeX) 39
3.17. Serialisierung des Layoutbaums und Erzeugung der Präsentation in der Zielsprache . 40

4.1. Beispiel für Transformationen in XSLT 45
4.2. Beispiel für Transformationen in fxt 46
4.3. Beispiel für Transformationen in HaXml 47
4.4. Beispiel für Transformationen in xcerpt 49
4.5. Vergleich der Transformationssprachen 50

5.1. Beispielhafte Datenquelle für PEP 54

5.2. Beispielhafte Ausgabe von PEP in HTML, dargestellt mit einem HTML-Browser . . 55
5.3. Verzeichnisstruktur von PEP 56
5.4. Registrierung der Prozessoren 57
5.5. Ausschnitte aus Regeln zum Hinzufügen von Trenntexten 62
5.6. Beispiel für PEP$_{layout}$ 64
5.7. DTD von PEP$_{separators}$ 66
5.8. Beispiel für PEP$_{separators}$ 66
5.9. DTD von PEP$_{generic}$ 67
5.10. Beispiel für PEP$_{generic}$ 67
5.11. Beispiel für PEP$_{template}$ 69
5.12. Beispiel für Verzeichnisstruktur von Prozessoren mit mehreren Ausprägungen 69

Literaturverzeichnis

[1] ABITEBOUL, SERGE, DALLAN QUASS, JASON MCHUGH, JENNIFER WIDOM und L. WIENER: *The Lorel Query Language for Semistructured Data*. International Journal on Digital Libraries, 1(1):68–88, April 1997. Siehe http://www-db.stanfort.edu/~widom/pubs.html.

[2] APACHE SOFTWARE FOUNDATION: *Xindice 1.0*. Siehe http://xml.apache.org/xindice/.

[3] BERLEA, ALEXANDRU: *fxgrep - The Functional XML Querying Tool*. Department of Computer Science, University of Trier. Siehe http://www.informatik.uni-trier.de/~aberlea/Fxgrep/.

[4] BERLEA, ALEXANDRU: *fxp - The Program fxp*. Department of Computer Science, University of Trier. Siehe http://www.informatik.uni-trier.de/~aberlea/Fxp/.

[5] BERLEA, ALEXANDRU und HELMUT SEIDL: *fxt - The Functional XML Transformation Tool*. Department of Computer Science, University of Trier. Siehe http://www.informatik.uni-trier.de/~aberlea/Fxt/.

[6] BRY, FRANÇOIS: *XQuery: Eine Einführung*. Technical Report, Ludwig-Maximilians-Universität München, LFE für Programmier- und Modellierungssprachen, Juni 2001.

[7] BRY, FRANÇOIS und SEBASTIAN SCHAFFERT: *Pattern Queries for XML and Semistructured Data*. Technical Report, Ludwig-Maximilians-Universität München, LFE für Programmier- und Modellierungssprachen, März 2002. Siehe http://www.pms.informatik.uni-muenchen.de/publikationen.

[8] BRY, FRANÇOIS und SEBASTIAN SCHAFFERT: *Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification*. Technical Report, Ludwig-Maximilians-Universität München, LFE für Programmier- und Modellierungssprachen, Februar 2002. Siehe http://www.pms.informatik.uni-muenchen.de/publikationen.

[9] CHAMBERLIN, DON, JONATHAN ROBIE und DANIELA FLORESCU: *Quilt: an XML Query Language for Heterogenous Data Sources*. In *Lecture Notes in Computer Science*. Springer-Verlag, Dezember 2000. Siehe http://www.almaden.ibm.com/cs/people/chamberlin/quilt.html.

[10] CLARKE, J.: *Jade*. Siehe http://www.jclark.com/jade/.

[11] DEUTSCH, ALIN, MARY FERNANDEZ, DANIELA FLORESCU, ALON LEVY und DAN SUCIU: *A Query Language for XML*. Siehe http://www.research.att.com/~mff/files/final.html.

[12] EXCELON: *XIS - eXcelon's eXtensible Information Server (XIS)*. Siehe http://www.exceloncorp.com/.

[13] FERNANDEZ, MARY, JÉRÔME SIMÉON und PHILIP WADLER: *XML Query Languages: Experiences and Exemplars*. Siehe http://www-db.research.bell-labs.com/user/simeon/xquery.html.

[14] HUHMANN, JOCHEM: *Fast wie von selbst – XML: mit DocBook und Emacs arbeiten*. iX, 9/2001:134 ff., September 2001.

[15] ISO - INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO-10179:1996 - Document Style Semantics and Specification Language (DSSSL)*. Siehe http://www.iso.org.

[16] KAY, MICHAEL H.: *Saxon, Version 6.4.4*. Siehe http://saxon.sourceforge.net/.

[17] KIESLING, TOBIAS: *Building an XML Information System, Seite 36 f.* Projektarbeit, Ludwig-Maximilians-Universität München, LFE für Programmier- und Modellierungssprachen, 2001. Siehe http://www.pms.informatik.uni-muenchen.de/publikationen.

[18] KRAUS, MICHAEL: *A Toolkit for Advanced XML Browsing Functionalities*. Diplomarbeit, Ludwig-Maximilians-Universität München, LFE für Programmier- und Modellierungssprachen, November 2000. Siehe http://www.pms.informatik.uni-muenchen.de/publikationen.

[19] KRAUS, SEBASTIAN: *Tamino experience report, advantages and pitfalls of an XML DBMS*. Projektarbeit, Ludwig-Maximilians-Universität München, LFE für Programmier- und Modellierungssprachen, 2001. Siehe http://www.pms.informatik.uni-muenchen.de/publikationen. In Vorbereitung.

[20] MÜLLER-HEINEMANN, ULF: *XML-Modellierung für Online-Dienste – Lehrangebot*. Projektarbeit, Ludwig-Maximilians-Universität München, LFE für Programmier- und Modellierungssprachen, September 2001. Siehe http://www.pms.informatik.uni-muenchen.de/publikationen.

[21] OASIS: *DocBook*. Siehe http://www.oasis-open.org/docbook/.

[22] ROBIE, J., J. LAPP und D. SCHACH: *XML Query Language (XQL)*. Siehe http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[23] ROBIE, J., J. LAPP und D. SCHACH: *The New YATL: Design and Spezifications*. Technical Report, INRIA, 1999.

[24] SOFTWARE-AG: *Tamino*. Siehe http://www.softwareag.com/tamino/.

[25] WALLACE, MALCOLM und COLIN RUNCIMAN: *Haskel and XML: Generic Combinators or Type-Based Translation?*

[26] WALSH, NORMAN und EVE MALER: *XML Pipeline Definition Language Version 1.0*. World Wide Web Consortium. W3C Note. 28.2.2002. Siehe `http://www.w3.org/TR/2002/NOTE-xml-pipeline-20020228/`.

[27] WALSH, NORMAN und LEONARD MUELLNER: *DocBook – The Definitive Guide, Version 1.0.2*. Sebastopol, CA (O'Reilly and Associates), 1999.

[28] WIELEMAKER, JAN: *SWI-Prolog SGML/XML parser, Version 1.0.14*. SWI, University of Amsterdam.

[29] WORLD WIDE WEB CONSORTIUM: *Cascading Style Sheets, level 1*. W3C Recommendation. Dezember 1996, überarbeitet Januar 1999. Siehe `http://www.w3.org/TR/1999/REC-CSS1-19990111`.

[30] WORLD WIDE WEB CONSORTIUM: *Cascading Style Sheets, level 2*. W3C Recommendation. Mai 1998. Siehe `http://www.w3.org/TR/1998/REC-CSS2-19980512`.

[31] WORLD WIDE WEB CONSORTIUM: *Character entity references in HTML 4*. W3C Recommendation. Dezember 1999 Siehe `http://www.w3.org/TR/1999/REC-html401-19991224/sgml/entities.html`.

[32] WORLD WIDE WEB CONSORTIUM: *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation. Oktober 2000. Siehe `http://www.w3.org/TR/2000/REC-xml-20001006`.

[33] WORLD WIDE WEB CONSORTIUM: *Extensible Stylesheet Language (XSL) Version 1.0*. W3C Recommendation. Oktober 2001. Siehe `http://www.w3.org/TR/2001/REC-xsl-20011015`.

[34] WORLD WIDE WEB CONSORTIUM: *HTML 4.01 Specification*. W3C Recommendation. Dezember 1999 Siehe `http://www.w3.org/TR/1999/REC-html401-19991224`.

[35] WORLD WIDE WEB CONSORTIUM: *Namespaces in XML*. W3C Recommendation. Januar 1999. Siehe `http://www.w3.org/TR/1999/REC-xml-names-19990114`.

[36] WORLD WIDE WEB CONSORTIUM: *XHTML 1.0: The Extensible HyperText Markup Language*. W3C Recommendation. Januar 2000. Siehe `http://www.w3.org/TR/2000/REC-xhtml1-20000126`.

[37] WORLD WIDE WEB CONSORTIUM: *XML Path Language (XPath) Version 1.0*. W3C Recommendation. November 1999. Siehe `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[38] WORLD WIDE WEB CONSORTIUM: *XML Path Language (XPath) Version 2.0*. W3C Working Draft. Dezember 2001. Siehe `http://www.w3.org/TR/2001/WD-xpath20-20011220`.

[39] WORLD WIDE WEB CONSORTIUM: *XML Schema Part 0, 1 and 2*. W3C Recommendation. Mai 2001. Siehe `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`, `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/` und `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/`.

[40] WORLD WIDE WEB CONSORTIUM: *XQuery 1.0: An XML Query Language*. W3C Working Draft. Dezember 2001. Siehe `http://www.w3.org/TR/2001/WD-xquery-20011220`.

[41] WORLD WIDE WEB CONSORTIUM: *XSL 1.0 - Formatting Objects*. W3C Recommendation. Oktober 2001. Siehe `http://www.w3.org/TR/2001/REC-xsl-20011015/slice6.html`.

[42] WORLD WIDE WEB CONSORTIUM: *XSL Transformations (XSLT) 1.0*. W3C Recommendation. November 1999. Siehe `http://www.w3.org/TR/1999/REC-xslt-19991116`.

[43] WORLD WIDE WEB CONSORTIUM: *XSL Transformations (XSLT) 2.0*. W3C Working Draft. Dezember 2001. Siehe `http://www.w3.org/TR/2001/WD-xslt20-20011220/`.