Freie Universität Berlin
Institut für Informatik

Diplomarbeit

Integration von XML in Content Management Systeme

Am Beispiel
des Network Productivity Systems (NPS)
der Infopark AG

Hinrich Boog

18. Oktober 2001

Sperrvermerk:

Die nachfolgende Diplomarbeit enthält vertrauliche Daten der Infopark AG. Veröffentlichungen oder Vervielfältigungen der Diplomarbeit - auch nur auszugsweise - sind ohne ausdrückliche Genehmigung der Infopark AG nicht gestattet. Die Diplomarbeit ist nur den Korrektoren sowie den Mitgliedern des Prüfungsausschusses zugänglich zu machen.

Inhaltsverzeichnis

Abkürzungsverzeichnis

HTML Hypertext Markup Language

XML Extensible Markup Language

SGML Standard Generalized Markup Language

CMS Content Management System

CM Content Manager (der Server)

DTD Document Type Definition

XSL Extensible Stylesheet Language

XSLT Extensible Stylesheet Language Transformations

DOM Document Object Modell

TCL Tool Command Language

FOP Formatting Objects Protocol

GUI Graphical User Interface, graphische Benutzerschnittstelle

HTML HyperText Markup Language

RDBMS Relational Database Managment System

SQL Structured Query Language

XQL XML Query Language

ICE Information and Content Exchange

1 Einleitung

In den letzten Jahren ist durch die zunehmende Vernetzung der IT-Welt ein Bedarf nach Standards und plattformübergreifenden Techniken entstanden. Einige Beispiele für Entwicklungen, die sich als weltweiter Standard auf verschiedenen Plattformen durchgesetzt haben, sind z.B. die Programmiersprache Java, das TCP/IP Protokoll oder die Auszeichnungssprache HTML. Mit Hilfe dieser Techniken lassen sich heute schon weitgehend Applikationen entwickeln, die auf fast jedem vernetzten Gerät laufen. Aus einer ähnlichen Motivation heraus wurde die eXtensible Markup Language (XML) entwickelt. XML ist zu einem Modewort in der Informationstechnologie geworden und fast jeder Hersteller von Software schreibt sich die XML-Unterstützung auf die Fahnen. In dieser Arbeit soll untersucht werden, inwiefern XML einen Fortschritt im speziellen Kontext von Content Management darstellen kann. Die Anforderungen an ein Content Management System, das die Verarbeitung von XML unterstützt, sollen herausgestellt werden und ein Teil des Systems soll als Prototyp implementiert werden.

1.1 Die eXtensible Markup Language (XML)

XML ist eine Auszeichnungssprache zur Strukturierung von Inhalten. Zwei der in der Spezifikation [Bra00b] genannten Ziele sind die einfache Benutzbarkeit über das Internet und die Unterstützung einer breiten Menge von Applikationen. Einige Eigenschaften der Sprache haben sich zur Unterstützung dieser Ziele herausgebildet, wie zum Beispiel die einfache Struktur und Kompatibilität zu SGML. Außerdem benutzt XML als Datenformat die Textdarstellung, was die menschliche Lesbarkeit und Plattformunabhängigkeit sicherstellt. Durch XML wird eine Klasse von Datenobjekten, genannt XML-Dokumente, definiert, auf die später noch näher eingegangen werden soll. Ein XML-Dokument besteht aus Zeichenfolgen und Markup (Tags). Diese bilden die XML-Elemente, die wiederum durch eine Schachtelung die zu behandelnden Daten strukturieren.

Der XML-Standard alleine nützt allerdings wenig, denn die Mächtigkeit von XML kommt erst im Zusammenwirken mit den flankierenden Standards zu tragen. Um

Applikationen mit XML zu entwickeln benötigt man bestimmte Mechanismen zur Bearbeitung, Speicherung und Umwandlung dieser Art von Dokumenten. Dafür existieren die schon in der XML-Spezifikation beschriebenen Parser, außerdem Prozessoren zur Umwandlung von XML, Datenbankimplementierungen und andere Tools, wobei die Parser und Prozessoren als Kernkomponenten angesehen werden können.

Unter den Parsern gibt es sogenannte "validierende" und "nicht-validierende" Parser. Ein nicht-validierender Parser muß ein Dokument lediglich auf seine Wohlgeformtheit prüfen, was bedeutet, daß jedes geöffnete Element auch wieder geschlossen werden muß und die Elemente eindeutig geschachtelt sind – hierdurch ergibt sich eine Baumdarstellung mit einem Wurzelelement für jedes Dokument. Validierende Parser hingegen müssen das Dokument nicht nur auf seine Wohlgeformtheit hin prüfen, sondern auch kontrollieren, ob die Struktur selber einer gewissen Vorgabe folgt. Um dies zu erreichen, können z.B. die aus SGML bekannten Document Type Definitions (DTDs) herangezogen werden.

Prozessoren dienen zur Transformation von einer XML-Struktur (also einer Baumstruktur) in eine andere mit Hilfe von Stylesheets – die zum Beispiel in der eXtensible Stylesheet Language (XSL) verfaßt sein können – und den sogenannten XSL-Transformations (XSLT). Hierauf soll später noch detaillierter eingegangen werden.

```
<document>          <document>
    <title>             <title>
        ....                ....
    </title>        </document>
</document>                 </title>
```

Abbildung 1.1: Wohlgeformtes (links) und nicht-wohlgeformtes XML

1.1.1 DTD vs. Schema

Durch eine DTD wird für eine Klasse von Dokumenten angegeben, wie die einzelnen Elemente geschachtelt sein dürfen und ob ein Subelement nicht, null, einmal oder mehrmals auftreten kann. Desweiteren kann angegeben werden, ob ein Element Zeichenketten enthalten darf oder leer sein muß und welche Attribute ein Element enthalten kann oder muß. Auch wenn DTDs schon die automatische Bearbeitung von XML-Dokumenten durch bestimmte Constraints ermöglichen oder vereinfachen, so haben sie doch einen entscheidenden Nachteil: Sie machen keine Aussagen über die Bedeutung der Elemente oder deren Inhalt. Auch die Notation von DTDs in einer EBNF-Form, die so eine andere Struktur haben als die XML-Dokumente selber, von

Applikationen getrennt verarbeitet werden müssen und auch nicht in denselben Datenbankschemata gehalten werden können, stellt einen Nachteil am DTD-Konzept dar.

Diese Nachteile versuchen die Entwicklungen von XML-Schemasprachen [Fal01] zu beseitigen. Ein XML-Schema beschreibt nämlich nicht nur die Struktur und Schachtelung der Elemente sondern kann auch Aussagen über die Semantik machen. Dazu gehört zum Beispiel die Definition von Datentypen analog zu herkömmlichen Programmiersprachen. Während ein validierender Parser anhand von einer DTD nicht feststellen kann, ob ein Element z.B. einen Wert eines ISO-Datums enthält, so ist dies mit einem XML-Schema problemlos möglich.

Durch Schemata werden bestimmte Grunddatentypen definiert, mit denen durch Kombination komplexere Datentypen gebildet werden können. Auch das aus der Objektorientierung bekannte Grundprinzip der Vererbung kann hier durch Erweiterung von vorhandenen Datentypen definiert werden. Im Gegensatz zu Programmiersprachen können durch XML- Schema zusätzlich noch so genannte “Facetten” von Datentypen angegeben werden, wie zum Beispiel der Wertebereich einer Integer-Zahl oder die Länge einer Zeichenkette. In Programmiersprachen sind diese Grenzen nur durch Eigenschaften des Betriebssystems eingeschränkt, selbstdefinierte Wertebereiche können nur durch Programmkonstrukte kontrolliert werden. Somit läßt sich mit Hilfe dieser Technik ein Teil der Logik eines Programms in eine Beschreibungsdatei mit Metainformationen verlagern. Außerdem ist XML-Schema selber auch ein XML-Dokument, was die Behandlung von Schemata vereinfacht.

Die in der Einleitung beschriebene XML-Unterstützung der Hersteller besteht bisher oft nur aus Import- und Exportfunktionen von XML-Dokumenten. Auch wenn das XML-Format sich gut verarbeiten läßt und sich dadurch als Austauschformat qualifiziert, so fehlt bei einem reinen Austausch von XML-Dokumenten dennoch ein entscheidender Teil, nämlich die Beschreibung. In dieser Form ist XML prinzipiell genauso nützlich wie jedes andere strukturierte Format. Wichtig ist die zugrundeliegende Beschreibung der Daten, also Metainformationen in Gestalt von XML-Schema, aufgrund deren man die weitere Behandlung der strukturierten Daten festlegen kann.

Wenn es Programmen möglich ist, beliebig strukturierte Daten mit Hilfe ihres Schemas zu bearbeiten und zu “verstehen”, so hätte man eine wirklich Plattform- und Applikationsunabhängigkeit der Daten erreicht.

2 Dokumentenverwaltung mit Content Management Systemen

Als Content Management bezeichnet man im allgemeinen die Kontrolle der Prozesskette, die bei der Erstellung, Bearbeitung, Veröffentlichung und Distribution von Inhalten durchlaufen wird und die dazugehörige Dokumentenverwaltung. Verschiedene Erkenntnisse der letzten Jahre haben die Entwicklung von Content Management Systemen (CMS) gefördert und begründen deren Notwendigkeit.

Zum einen hat sich durch die maschinelle Datenverarbeitung und das Internet das Datenaufkommen vervielfacht. Diese Mengen lassen sich nur schlecht mit herkömmlichen Methoden der Dokumentenverwaltung kontrollieren.

Weiterhin stehen Unternehmen unter einem großen Aktualitätsdruck bezüglich ihrer Produkte, sei es im traditionellen Publishing oder durch Web-Auftritte. Um bei dieser Aktualität und den schnell zu verarbeitenden Datenmengen Fehler zu vermeiden und Abläufe zu beschleunigen, werden Content Management Systeme eingesetzt. Die zunehmende Verbreitung von Online-Publishing fördert diese Entwicklung. Gerade bei der Online-Publikation ist eine ständige Kontrolle unerläßlich, da hier nicht einmalig veröffentlicht wird, sondern über einen bestimmten Zeitraum und dementsprechend die Inhalte nicht nur aufbereitet werden müssen, sondern auch wieder kontrolliert aus der Publikation entfernt werden müssen. Fehlendes Content Management macht sich auf vielen Webseiten durch veraltete oder inkonsistente Inhalte bemerkbar.

Ein weiteres Hauptargument für Content Management stellt die Linkverwaltung in Online-Publikationen dar: Gerade bei der multidimensionalen Verlinkung von Inhalten ist es ohne eine automatische und systematische Überprüfung dieser Links fast unmöglich, alle Fehler in der Navigation aufzudecken und eine komplett konsistente Publikation im Netz auf Dauer zu garantieren. Mit Content Management Systemen hingegen läßt sich zum Beispiel definieren, daß ein Objekt erst dann gelöscht werden kann, wenn es nicht mehr verlinkt ist oder dass die Hyperlinks auf dieses Objekt mit dem Objekt gelöscht werden.

Wie auch in [Man99] herausgestellt wird, leisten CMS gerade im Online-Bereich Aufbauhilfe für bestimmte Arten von Publishern. Dies wird deutlich, wenn man die Publikationen der verschiedenen Redaktionen in drei Kategorien aufteilt (siehe Abbildung 2.1):

- *Professional Publishers:* Firmen, deren Geschäft die Arbeit mit Inhalten ist und die Erfahrung im Verlagswesen haben
- *Accidental Publishers:* Unternehmen, die keine Erfahrung im Verlagswesen haben, im Gegensatz dazu aber relativ große Online-Publikationen halten
- *Publishing Playground:* Einzelpersonen, Arbeitsgruppen, einzelne Abteilungen mit kleinen Web-Auftritten

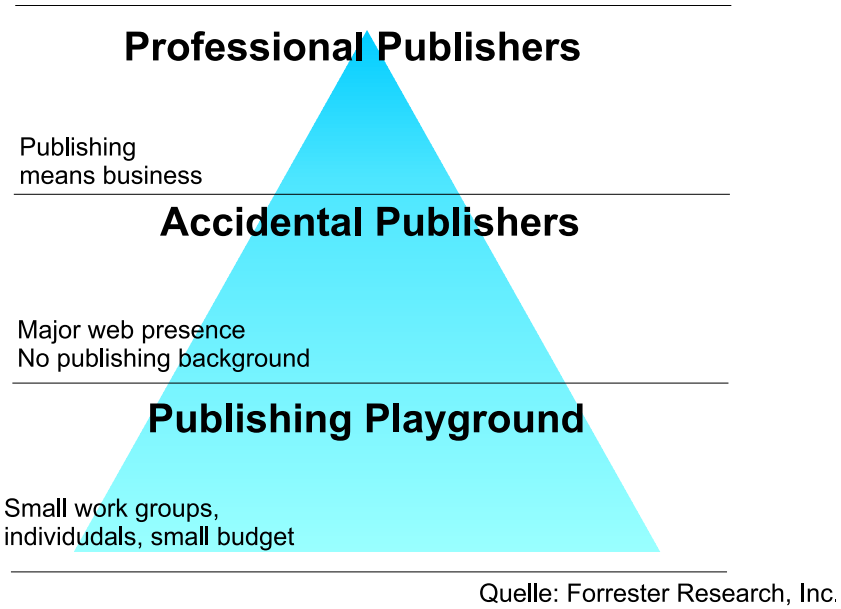

Abbildung 2.1: Die Publishing-Pyramide zur Veranschaulichung der verschiedenen Arten von Publishern

Für diese drei Gruppen leisten Content Management System unterschiedliche Aufgaben. Den professional Publishern helfen CMS einfach, ihr Datenaufkommen zu bewältigen, wo hingegen sie bei den anderen beiden Gruppen den professionellen Auftritt erst ermöglichen und auch eine technische Infrastruktur bereitstellen, die die benötigten Abläufe automatisiert.

Kennzeichnend für CMS ist der Begriff des Workflows: Durch den Workflow werden

bestimmte Zustände definiert, die ein Dokument vor der Veröffentlichung durchlaufen muss und Rollen, durch die bestimmte Nutzer spezifische Rollen in einem Workflow übernehmen. Dies kann beispielsweise bedeuten, daß eine bestimmte Gruppe von Redakteuren Dokumente erstellt und diese genau durch einen Vorgesetzen abgezeichnet werden müssen, danach noch vom Chefredakteur kontrolliert werden bevor das Layout und der Druck (die Online-Veröffentlichung) zum Tragen kommen. Diese möglichen Workflowzustände sind in Abbildung 2.2 zu sehen.

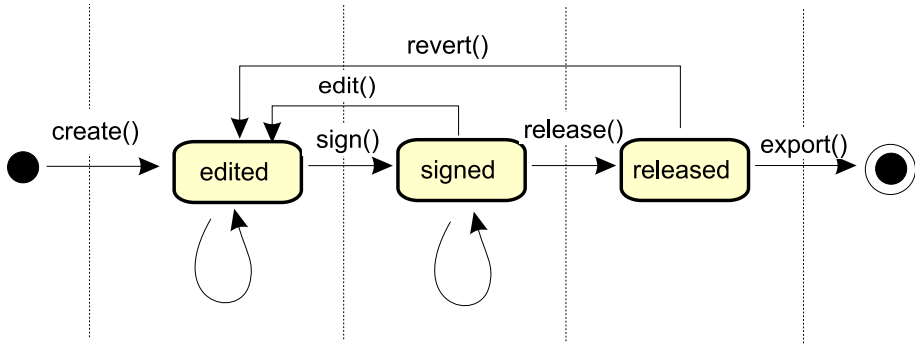

Abbildung 2.2: Mögliche Zustände eines Beispielworkflows

2.1 Das NPS der Infopark AG

Das NPS System der Infopark AG ist ein modulares Content Management System, welches verteilt betrieben werden kann. Die Hauptkomponente ist der NPS Content Management Server (im folgenden als CM bezeichnet). Dieser steuert die Kommunikation mit den Datenbanken und übernimmt die eigentlichen Aufgaben des Content Managements, also die Workflow-Kontrolle, Aufgabenverteilung, die Freigabe und das Zurückziehen von Inhalten, etc.. Der Content Management Server bietet als Außenanbindung eine Tcl-Schnittstelle und ein XML-API, über das die Kommunikation mit der grafischen Benutzerschnittstelle auf HTML-Basis (HTMLGUI) läuft. Durch das HTMLGUI ist die dezentrale Arbeit am System vom Webbrowser aus möglich. Weitere Komponenten sind zum Beispiel der Search Engine Server, der mit Hilfe der Suchmaschine Verity erweiterte Suchfunktionalitäten bietet, sowie die Template-Engine, die den Export von Inhalten ermöglicht und somit freigegebene Inhalte auf einen Webserver exportiert. Dieser Export lässt sich auch inkrementell durchführen, indem nur Inhalte exportiert werden, die sich seit einem definierten Zeitpunkt geändert haben.

2.1.1 NPS Objekte

Die Verwaltung von Objekten und Inhalten im NPS geschieht bisher durch die Definition von Objektklassen. Ähnlich zum Klassenkonzept in objektorientierten Programmiersprachen legt eine Objektklasse im NPS das Verhalten und die Ausprägung der einzelnen Objekte fest. Insbesondere werden über die Objektklassen die Attribute und Attributgruppen eines Objektes festgelegt.

Die Objektklassen im NPS werden jeweils von einem der fünf Basistypen *Publication, Document, Image, Generic* und *Template* abgeleitet. Diese Typen definieren weitestgehend den Verwendungszweck und die Behandlung eines Objekts. Objekte vom Typ *Publication* bilden dabei die Hierarchie des Dokumentenbaumes ab, d.h. eine Publikation kann beliebige Unterobjekte besitzen und selber Unterobjekt einer anderen Publikation sein. Wenn eine Publikation veröffentlicht wird, so wird dazu ihr Inhalt freigegeben und vom System z.B. auf einer Webseite zur Verfügung gestellt. Der Inhalt einer Publikation kann der Content selber oder auch die Contents der Unterobjekte darstellen. So kann die Online-Veröffentlichung einer Publikation zum Beispiel ein Inhaltsverzeichnis sein, in dem die Links auf seine Unterobjekte zu finden sind.

Objekte des Typs *Document* stellen einen Inhalt im klassischen Sinne dar, das heißt Dokumente beinhalten den eigentlichen Content. Der Content eines Dokumentes setzt sich zusammen aus einem Body, der den Text beinhaltet, dem HTML-Code zur Veröffentlichung, Attributen (gültig ab, gültig bis, Titel und Content Typ sowie frei definierte Attribute der Objektklasse), Links, die im Content auftreten können, und einem Status, der festlegt, ob der Content zur Zeit bearbeitet wird oder zur Veröffentlichung freigegeben ist. Somit beschreiben die Publikationen die Struktur einer Menge von Informationen und die Dokumente enthalten die Inhalte.

Die Objekttypen *Image* und *Generic* haben Eigenschaften ähnlich denen eines Dokumentes mit dem Hauptunterschied, dass hier der Content aus Binärdaten besteht. Der Typ Image wird für Grafiken verwendet, der Type Generic für andere Dokumente, die nicht direkt in der Publikation dargestellt werden, sondern typischerweise zum Download bereit gestellt werden, wie z.B. *.pdf oder *.zip Dateien. Der Objekttyp Template definiert das Layout der veröffentlichten Inhalte. Für Dokumente und Publikationen gibt es ein zentrales Mastertemplate, das bei der Veröffentlichung von Objekten dieser Typen abgearbeitet wird. Durch diese Abarbeitung können zum Beispiel bestimmte Attributwerte an bestimmten Stellen in Webseiten eingebaut werden oder andere Dokumente referenziert werden. Von diesem Mastertemplate aus können wiederum bestimmte Untertemplates aufgerufen werden, die z.B. eine Navigationsleiste o.ä. generieren. Außerdem können für Objektklassen sogenannte "Body-Templates" definiert werden, die das Aussehen des Objektes während seiner

gesamten Lebensdauer bestimmen. Bei Anwendung von Body-Templates kann der Content eines Objektes nicht mehr frei editiert werden, sondern lediglich die durch die Objektklasse vorgegebenen Attribute.

Alle Objekte im NPS verfügen über bestimmte Objektattribute (wie Name, Pfad, Objekttyp, Objektklasse, Version, Workflow, etc.), Objektrechte (die festlegen, welche administrativen Vorgänge am Objekt durch welche Personen oder Gruppen durchgeführt werden dürfen), eine Versionskontrolle, Links auf das Objekt und ein Protokoll. Dokumente und Publikationen verfügen zusätzlich noch über das Attribut "verfügbare Templates", über das die oben beschriebenen Templates zugeordnet werden.

Zusammenfassend kann man also alle Objekte bzw. deren Klassen in drei Kategorien einteilen:

- Publikationen zur Strukturbeschreibung
- Dokumente, Images und Generics mit den eigentlichen Inhalten
- Templates zur Formatierung

2.2 Grenzen der Darstellung in aktuellen CMS

Wie an den obigen Ausführungen deutlich wird, folgen die Dokumente größtenteils einer Dictionary-Struktur, bei der Attributwerte über Attributnamen referenziert werden. Die Attribute selber sind entweder primitive Datentypen oder Zeichenketten und lassen keine eigene Strukturierung zu. Für viele Anwendungsbereiche reicht dies aus, allerdings gibt es bestimmte Inhalte, die sich mit dieser Struktur nur sehr kompliziert oder gar nicht mehr abbilden lassen.

Ein gutes Beispiel hierfür ist die Online-Ausgabe eines Kataloges für Spezialgase. In diesem Katalog werden alle Arten von Industriegasen und Gasgemischen dargestellt. Während einerseits die Gase in verschiedene Produktgruppen aufgeteilt werden müssen und bestimmte Eigenschaften haben, können die Gase und Gasgemische wiederum Verunreinigungen durch andere Gase enthalten. Da das bisherige NPS für ein Dokument aber nur die oben beschriebene Dictionary-Struktur besitzt, lassen sich die in diesem Fall benötigten Strukturierungsinformationen nur durch Verwaltung von einzelnen Gasen als Publikation umsetzen.

Im Gegensatz z.B. zu einem Roman dient hier aber nicht die Publikationshierarchie zur Strukturierung der Inhalte, sondern die Inhalte selber sind derart komplex, daß

sie sich nur als Publikation abbilden lassen, obwohl man intuitiv wahrscheinlich sagen würde, daß ein Produkt einem Dokument entspricht. Diese unterschiedlichen Anforderungen an die Komplexität von Dokumenten lassen sich durch XML anpassen, d.h. man bekommt soviel Struktur wie benötigt.

2.3 Erweiterung des Dokumentenmodells durch den Einsatz von XML

Wenn wir die Objekte im NPS durch XML-Dokumente abbilden wollen, so bietet XML prinzipiell die Möglichkeit, auf die bisherige Aufteilung in Objekttypen weitestgehend zu verzichten. Durch geeignete Schemasprachen läßt sich die erlaubte Struktur von XML-Dokumenten beschreiben und einschränken. Binäre Inhalte können direkt in die Dokumente eingebunden werden oder durch Verweise referenziert werden. Dies kann zum Beispiel mit XLink (XML Linking Language [DMO01]) realisiert werden.

Somit lassen sich zunächst mal die Objekttypen Document, Image und Generic zusammenfassen, da sie sich vordergründig nur durch die Art ihres Inhalts unterscheiden, nicht jedoch durch die Anforderungen, die von Benutzer- oder Workflowseite an sie gestellt werden.

Die Struktur einer Menge von Objekten untereinander, die bisher durch den Objekttyp Publication dargestellt wird, kann ebenfalls in den gleichen XML-Dokumenten erfolgen. Um dies zu ermöglichen, müssen an mindestens einer Stelle des XML-Dokumentes Verweise auf andere Dokumente gleicher Struktur erlaubt sein. In einer Schemasprache bedeutet dies, rekursive Verweise auf die Klasse zuzulassen. Somit hätten wir auch den Objekttyp Publication erfasst.

Letztendlich bliebe der Objekttyp Template. Dieser erfüllt aus der XML-Perspektive zwei Funktionen: Zum einen schränkt er die Möglichkeiten zur Bearbeitung eines Objektes ein, wenn Body-Templates (also Schablonen für den Body einen Objektes) eingesetzt werden, zum anderen definieren Templates die Formatierung bei der Veröffentlichung und beim Export. Ersteres läßt sich in XML durch Einschränkungen im Schema bewerkstelligen: Anstelle der Definition eines Body-Templates wird ein bestimmter Bereich des Dokumentes zeitweilig oder endgültig durch restriktivere Regeln eingeschränkt. Die Erzeugung dieser Regeln ersetzt den Vorgang des Erstellens von Body-Templates. Die zweite Funktionalität von Templates im Kontext der NPS-eigenen Template-Logik dient zur Unterstützung des Exports, zum Beispiel zum Einfügen von dynamischen Inhalten. Gerade hier bieten sich durch den XML-Einsatz enorme Möglichkeiten die im weiteren Verlauf der Arbeit analysiert werden

sollen. Prinzipiell sollten vorhandene Techniken wie z.B. XSLT (eXtensible Stylesheet Language Transformations) den funktionalen Umfang der bisherigen Templates aber problemlos abdecken.

Somit können wir folgern dass es möglich ist, die vorhandenen Objekttypen auf einen Typ von XML-Dokument abzubilden, ohne einen Funktionalitäts- oder Informationsverlust in Kauf zu nehmen. Zusammen mit geeigneten Techniken zum Export und zur Weiterverarbeitung, zur Aufgabendefinition und Workflowkontrolle, hätten wir damit den vollen Funktionsumfang des bisherigen NPS abgedeckt.

3 Technische Aspekte eines Content Management Systems mit XML-Integration

Die Funktionen des NPS gliedern sich in die Aufgaben von zwei Hauptkomponenten, nämlich das Redaktions- und das Live-System. Das Redaktionssystem deckt die Arbeitsabläufe bis zur Veröffentlichung ab, d.h. hier werden Dokumente erstellt und bearbeitet. Das Live-System kontrolliert den Export und Re-Export von Dokumenten und Inhalten und übernimmt so die Administration von Webseiten. Welche Änderungen sind nun an den einzelnen Komponenten für eine erfolgreiche XML-Integration notwendig?

3.1 Aufbau des Systems

Der Aufbau des NPS Systems teilt sich in mehrere Komponenten, die in Abbildung 3.1 dargestellt sind. Die Bearbeitung von Dokumenten kann entweder über den Tcl-Client oder das HTMLGUI durchgeführt werden.

Bei der Bearbeitung via HTMLGUI wird eine Anfrage per HTTP vom Browser des Benutzers an das HTMLGUI geschickt, das als Teilkomponente einen Apache-Webserver mit Servlet-Engine besitzt[1]. Jetzt werden im HTMLGUI die benötigten Anfragen an die Datenbasis generiert und mit Hilfe des XML-basierten ICE[2]-Protokolls an den Content Management Server (CM-Master) geschickt. Dieser bearbeitet selber keine Anfragen, sondern gibt sie an einen seiner Slaves weiter. Der CM-Master dient hauptsächlich zu Steuerung und Lastverteilung zwischen den einzelnen Slaves. Die Slaves generieren dann entsprechende Anfragen an die Datenbanken und schicken die berechnete Antwort direkt an das anfragende HTMLGUI.

Neben den Anfrage an die Datenbanken können die Slaves auch Anfragen an den

[1] Es kann auch jeder andere CGI-fähige Webserver eingesetzt werden
[2] ICE: Information and Content Exchange (siehe auch [Dou98])

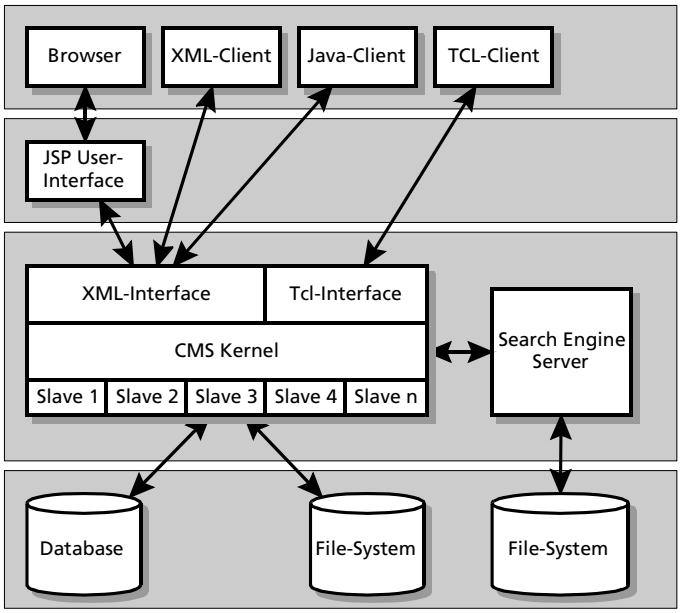

Abbildung 3.1: Aufbau des NPS Redaktionssystems

SES (Search-Engine Server) stellen, der mit Hilfe der Verity-Library schnellere Antworten bei bestimmten Anfragen (und einer tendenziell großen Datenbasis) liefern kann. Das HTMLGUI generiert aus den Antworten HTML-Seiten und schickt diese an den Browser zurück.

Der Tcl-Client steht im gleichen Verhältnis zum CM wie das HTMLGUI. Statt im Browser werden die Anfragen über eine Tcl-Shell gestellt und vom Tcl-Client zum CM durchgereicht.

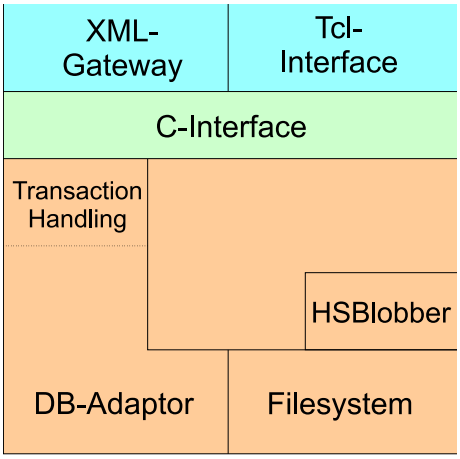

Abbildung 3.2: Der NPS-Kernel mit Schnittstellen für Datenbank, GUI und Tcl-Client

3.2 Kernel

Im CMS Kernel(Abbildung 3.2) sind folgende Komponenten vereinigt:

XML- und Tcl-Gateway: Hier laufen die Anfragen der Clients auf.

C-Interface (CIF): Der Zugriff auf den Server wird für obere Schichten über Standard C-Funktionen einheitlich zur Verfügung gestellt.

DB-Adaptor: Der Datenbankadapter nimmt SQL-Anfragen vom Kernel entgegen und reicht diese an die benutze Datenbank weiter. Dementsprechend verfügt der DB-Adaptor zur Zeit über Treiber für relationale Datenbanken von Oracle, Sybase, Informix und MS SQL. Der DB-Adaptor übernimmt außerdem das Transaction-Handling.

FS-Adaptor: Hier wird die Kommunikation zum Dateisystem sichergestellt. In der derzeitigen NPS-Version ist dem Administrator die Wahl überlassen, Blobs entweder im Dateisystem oder in der Datenbank zu speichern. Werden diese im Dateisystem gespeichert, so sichert der HSBlobber das Lesen und Schreiben von Blobs.

Für eine erfolgreiche XML-Integration sind im Kernel relativ gerinfügige Änderungen notwendig. Da die Kommunikation zwischen den Komponenten auf XML-Basis funktioniert, sind natürlich bereits einige Bibliotheken im System integriert, die für eine elementare XML-Funktionalität benötigt werden. So gibt es bereits Bibliotheken, die eine Behandlung von Datenstrukturen des Document Object Models (DOM)

```
1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.  <nps-document>
3.    <cm-header>
4.      <attributes>
5.        <id>2001</id>
6.        <validFrom>2001-08-20</validFrom>
7.        <validUntil />
8.        <workflow-id>4003</workflow-id>
9.        ...
10.     </attributes>
11.     <permissions>
12.       ....
13.     </permissions>
14.   </cm-header>
15.   <cm-content>
16.     <title>The Boston Tea-Party</title>
17.     <body>Once upon a time ...</body>
18.     <!-- Beliebige andere Daten gemaess Definition -->
19.       ...
20.   </cm-content>
21. </nps-document>
```

Abbildung 3.3: Ein Beispieldokument, das ein NPS-Objekt in XML abbildet

[Le 00] ermöglichen.

In die XML-Dokumente, die wie beschrieben die einzige Datenstruktur in einem XML System darstellen, müßten allerdings bestimmte Elemente oder Attribute eingebettet werden, die die Informationen zum eigentlichen Content Management enthalten. Dies könnte zum Beispiel wie in Abbildung 3.3 abgebildet werden.

Durch geeignete Mechanismen müssten diese Attribute dann aus den XML-Dokumenten statt aus der bisherigen Datenbank ausgelesen werden. Danach können die bisherigen Funktionen des Content Managements weiter benutzt werden.

Der CM-Server stellt außerdem eine Funktion zur Überprüfung der Dokumente vor dem Export zu Verfügung, den sogenannten Vollständigkeitscheck. Hiermit kann der Benutzer überprüfen, ob das Dokument den vorher festgelegten Vorgaben folgt. Um dieses Feature in XML abzubilden, wird ein Parser benötigt, der das Dokument gegen eine DTD oder ein Schema validieren kann.

In den Schnittstellen nach außen sind die Änderungen relativ gering, da die Kommunikation zwischen den einzelnen Komponenten des NPS bereits auf der Basis von

XML erfolgt. Dies erspart weitreichende Änderungen am Protokoll, denn prinzipiell lassen sich XML-Dokumente oder Teile davon in die bisherigen Anfragen einbetten. Änderungen müssen an den Teilen vorgenommen werden, die die Anfragen generieren. Hier müssen spezielle Accessor-Methoden für die Klasse XML-Document implementiert werden, sodaß zum Beispiel die oben beschriebenen Attribute direkt ausgelesen werden können.

Die weitreichensten Kernel-Änderungen müssen wahrscheinlich an den Datenbankadaptern vorgenommen werden, da man für XML-Dokumente andere Speicherstrukturen als bisher verwenden sollte.

3.3 Persistenz

Im bisherigen NPS werden über den DB-Adaptor Schnittstellen zu den relationalen Datenbanksystemen Oracle, Sybase, Informix und MS SQL zur Verfügung gestellt. Für eine XML-Integration stellen sich hier neue Anforderungen an die Datenbanken. Wie auch in [ST01] herausgestellt wird, enthalten XML-Dokumente Daten, die untypisch für relationale Datenbanken sind. Die typische Mischung aus Metadaten und Information in XML-Dokumenten, angereichert mit unterschiedlich großen Textabschnitten und Multimediadaten, läßt sich nicht ohne weiteres in relationalen Datenbanken speichern. Zusätzlich hat man das Problem, daß XML-Dokumente sehr unterschiedlich strukturiert sein können und damit der Platzbedarf eines Dokumentes schlecht abzuschätzen ist.

Um XML-Dokumente effizient speichern und lesen zu können, gibt es verschiedene Ansätze. Zum einen entwickeln sich native XML-Datenbanken , die XML-Daten verwalten und zusätzlich relationale Datenbanken einbinden können, zum anderen entwickeln sich Erweiterungen der relationalen Datenbanken, die diese XML-fähig machen bzw. die XML-Unterstützung von relationalen Datenbanken erhöhen.

3.3.1 XML-Datenbanken

Seit einiger Zeit gibt es XML Datenbanken verschiedener Hersteller. Diese bieten eine native XML-Schnittstelle und verringern so natürlich enorm die Komplexität eines Interfaces. Anfragen können direkt in XQL gestellt werden und desweiteren ist hier generell die komplette Funktionalität, die man benötigt um mit XML-Dokumenten zu arbeiten, vorhanden.

Vorteile:

- Natives XML Interface
- XML-Funktionalität auf DB-Seite
- Query und Update in z.B. XQL

Nachteile:

- Wenige Anbieter bisher
- XMLDB sind in der Regel nicht bei Kunden vorhanden und teuer.

- schlechtere Speicherung von relationalen Daten, d.h. Daten der ursprünglichen Applikationslogik müssen in XML eingebettet werden und können nicht mit der guten Performance von relationalen Datenbanksystemen behandelt werden.

- XMLDB stehen am Anfang ihrer Entwicklung (im Gegensatz zu relationalen Datenbanken, die seit vielen Jahren verbessert werden).

Als Beispiele können hier die kommerzielle Datenbank Tamino (Software AG) sowie die freie Datenbank dbXML (dbXML Group) genannt werden.

3.3.2 Mapping von XML Dokumenten auf relationale Datenbanken

Es besteht desweiteren die Möglichkeit, XML-Dokumente mittels eines Adapters auf mehrere Tabellen in relationalen Datenbanken zu mappen. Das eigentliche Mappen ist dabei nicht sehr kompliziert, allerdings ist die Performance stark vom Kontext abhängig. Hierbei stellt sich die Frage, wie komplex die zu behandelnden Dokumente sind und wie tief diese geschachtelt sind. Diese Komplexität schlägt sich sowohl bei Retrieval als auch beim Update nieder. Beim Retrieval müssen u.U. sehr viele Joins durchgeführt werden, um das ursprüngliche XML-Dokument wieder herzustellen, beim Update müssen u.U. sehr viele Schlüsselreferenzen verfolgt werden, um den eigentlichen Wert zu finden (siehe auch Abbilung 3.4).

Der große Vorteil bei dieser Art der Speicherung ist allerdings, dass man gängige relationale DBMS benutzen kann und durch ein SQL-Interface zur Datenbank auch herstellerunabhängig bleiben kann. Außerdem ist die Performance von heutigen RDBMS sehr gut, sodass eventuell trotz der eigentlich gegensätzlichen Strukturen eine gute Performance herausspringen kann. Statt die Adapter zu programmieren, kann man auch Middleware einsetzen, die das Mapping übernimmt.

Als Middleware bieten sich hier zum einen XML-Erweiterungen der Hersteller von RDBMS an(z.B. XML-Erweiterungen für Oracle 8i), zum anderen gibt es aber auch Drittprodukte, die als herstellerunabhängiges Interface das Mapping $XML \rightarrow RDBMS$ übernehmen, wie zum Beispiel *XDB* [ZVO00]. Beim Einsatz von Mapping-Tools ist allerdings Vorsicht geboten, da diese teilweise einen sehr naiven Mapping-Algorithmus benutzen, der zwar für jede Art von XML-Dokument funktioniert, aber oft viel Performance verschenkt, da der Anwendungskontext nicht berücksichtigt wird – aus diesem kann sich jedoch eine vollkommen andere Abbildung auf die relationale Datenbank als günstig erweisen[Bou01].

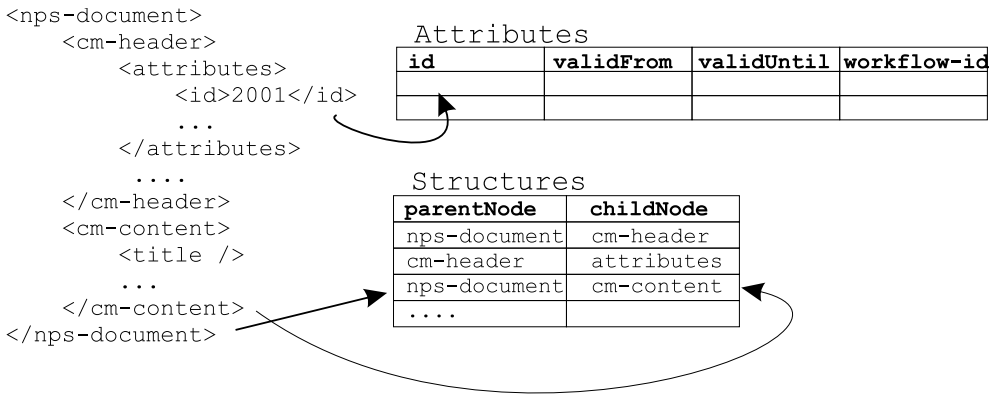

Abbildung 3.4: Beispiel für das Mapping eines XML-Dokumentes auf relationale Tabellen: Die Struktur kann in einer Tabelle mit Vater/Kind-Beziehungen abgebildet werden, die Attributwerte werden hier in einer separaten Tabelle gehalten.

Vorteile:

- RDBMS sind bei den meisten Kunden vorhanden
- Nicht-XML Daten lassen sich effizient speichern
- Ausgereifte Algorithmen, dadurch sehr performante RDBMS
- Durch SQL herstellerunabhängiges Interface

Nachteile:

- XML und relationale Daten haben prinzipiell widersprüchliche Strukturen
- Aufwendiges Speichern und Lesen durch viele Joins und viele Fremdschlüssel
- XML-Erweiterungen wie z.B. XQL sind nicht oder nur eingeschränkt nutzbar

Bei dieser Art der Speicherung ist es unbedingt notwendig, die Struktur der Dokumente vorher zu kennen, denn diese findet sich in einem optimalen relationalen Datenbankschema wieder. Jede Änderung an der Struktur der zu behandelnden Dokumente kann eine Änderung der Datenbanktabellen notwendig machen.

3.3.3 Speicherung der XML-Daten in Blobs und Anlegen von Sekundärindizes auf bestimmte Elemente

Diese Lösung stellt eine Mischung aus den ersten beiden Ansätzen dar: Man geht davon aus, dass sich XML-Daten nicht ohne weiteres auf relationale Datenbanken abbilden lassen, deshalb speichert man diese als Blobs in der Datenbank oder im Dateisystem. Zusätzlich legt man für bestimmte Elemente, die regelmäßig von den Applikationen benötigt werden, Sekundärindizes an, die in relationalen Tabellen abgelegt werden. Um ein bestimmtes XML-Dokument zu finden spart man sich so (bei geeigneten Indizes) in den meisten Fällen das Durchsuchen der Blobs und bekommt dadurch sofort das gewünschte Dokument.

Speziell für die Suche und Indexierung gibt es Standardlösungen, die man relativ einfach in ein System einbinden kann, wie zum Beispiel Oracle's *inter*Media oder das schon im NPS verwendete Verity-Cartridge.

Vorteile:

- Sehr schnelles Retrieval der Dokumente
- Einfaches und schnelles Speichern
- Herstellerunabhängigkeit durch SQL
- Unterschiedliches Speicherverhalten der einzelnen Produkte für Blobs kann in individuellen Datenbankadaptern realisiert werden

Nachteile:

- Sekundärindizes müssen angelegt und gepflegt werden
- Funktionalität des DBMS wird nicht genutzt, insbesondere bei Update von Teilstrukturen
- Alle Änderungen an den Dokumenten müssen im Hauptspeicher vom Adapter durchgeführt werden
- Schlechte Performance bei "Select" und gleichzeitigem "Update" von anderer Stelle, da immer der komplette Blob gelockt wird

Vom theoretischen Standpunkt aus lässt sich die Frage nach der Performance nicht ohne weiteres beantworten und hängt sehr stark mit der Art der zu speichernden Dokumente zusammen. Handelt es sich um "relativ flache" Dokumente, also XML-Dokumente, die eine geringe Schachtelungstiefe der Elemente besitzen, so bietet sich

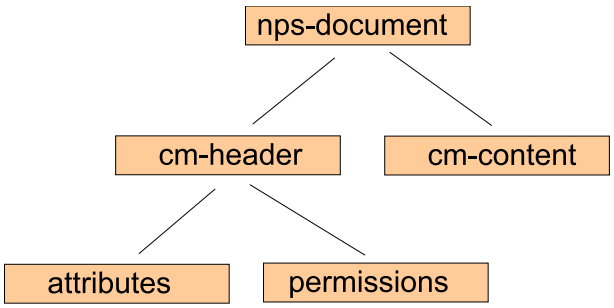

Abbildung 3.5: Abbildung des Beispieldokumentes auf eine hierarchische Datenbank

das Mapping auf relationale Datenbanken an. Wenn immer ganze Dokumente vom System benötigt werden, so ist die Speicherung in Blobs sicherlich die Wahlalternative. Selbst hier ließe sich eventuell noch eine Verbesserung durch "Teilblobs" erzielen, sodass nicht immer das gesamte Dokument im Speicher bearbeitet werden muss, sondern auch Teildokumente einzeln geladen werden können. Dies bringt sowohl Vorteile für das Speicher- als auch für das Locking-Verhalten.

Wenn über die zu speichernden Dokumente allerdings nichts bekannt ist und es sich um XML-Dokumente mit beliebiger Schachtelungstiefe handeln kann, so ist die Benutzung einer XML-Datenbank unverzichtbar. Zum einen sind hier die Update- und Retrieval Operationen zu nennen, zum anderen kann man ein reines Hauptspeicherproblem bekommen, wenn man aus beliebigen Dokumenten DOM-Bäume erzeugt, die innerhalb der Applikation bearbeitet werden.

3.3.4 Das hierarchische Datenmodell als Alternativansatz

Als alternativer Ansatz zur relationalen Speicherung bieten sich hierarchische Datenbanken an. Obwohl das hierarchische Datenmodell in der Literatur oft als historisch abgetan wird, sind die Parallelen zu XML-Strukturen unübersehbar und viele Funktionen, die hierarchische Datenbanken bieten, sind heute zur Speicherung von XML-Daten äußerst wünschenswert. So würde sich für unser Dokument "Rechnung" eine Hierarchie wie in Abbildung 3.5 ergeben.

Die in [Vos94] vorgestellten Funktionen zum Zugriff auf hierarchische Datenbanken beinhalten mit "getNext (GN)" und "getNextWithinParent (GNP)" genau die Art von Funktionen, die man sich für XML-Dokumente wünscht. Im Gegensatz zu den Daten, die ursprünglich in hierarchischen Datenbanksystemen gehalten wurden, bilden XML-Dokumente eine tatsächliche Baumstruktur. Virtuelle Einträge, die in hierarchischen Datenbanken noch benötigt wurden, um die wirklichen Zusammenhänge zu modellieren, könnten in XML-Dokumenten zur Realisierung von

Fremdschlüsselverweisen benutzt werden. Hierarchische Datenbanken sollen hier jedoch nur als theoretische Alternative genannt werden, denn sie sind in den meisten Umgebungen auch nicht vorhanden und stellen somit ein benötigtes Zusatzprodukt analog zu den (verwandten) XML-Datenbanken dar.

3.4 Benutzerschnittstelle

Wie schon beschrieben besitzt das NPS eine Benutzerschnittstelle auf HTML-Basis – die Tcl-Schnittstelle kann eher als "Administrationsinterface" gesehen werden. Gerade die grafische Darstellung im Webbrowser stellt sich als eine besondere Herausforderung dar, da zum einen durch XML die Flexibilität der Dokumente erhöht werden soll, zum anderen aber die Menüführung wie im bisherigen System möglichst intuitiv eindeutig und klar sein sollte.

Die grafische Darstellung verdeutlicht aber auch die eigentliche Universalität von XML. Während sich im Kernel relativ wenig ändert und die Speicherung von XML-Daten bei der derzeitigen Marktsituation zu Zeit eher noch Probleme bereitet als Verbesserungen bringt, so wird in der Benutzerschnittstelle die Mächtigkeit von XML schon direkt deutlich.

Dies läßt sich zum Beispiel durch ein XML-Schema bewerkstelligen, das dem Dokument oder der Klasse von Dokumenten zugrunde liegt:

Traditionell ist in der Benutzerschnittstelle ein hoher Programmieraufwand notwendig, um zum einen eine geeignete Menüführung zu schaffen und zum anderen die Eingaben des Benutzers auf einen bestimmten Datentyp oder Wertebereich hin zu überprüfen. Ein großer Teil dieser Logik kann durch XML-Schema und geeignete Tools in die Schemadateien, die die Metadaten des Dokumentes darstellen, verlagert werden.

Um die genauen Abläufe und deren XML-basiertes Verhalten zu beschreiben, muss man die einzelnen Hauptaufgaben getrennt betrachten.

3.4.1 Das Anlegen eines Dokumentes

Das Anlegen eines Dokumentes erfolgt entweder durch Import eines bestehenden Dokumentes oder durch manuelle Eingabe eines Textes.

Beim Import aus dem Dateisystem kommen schon im bisherigen System diverse Importfilter zum Einsatz, die den Dokumententyp erkennen und im wesentlichen den Textanteil eines Dokumentes extrahieren. Die jetzigen NPS-Objekte enthalten ein Attribut "Blob", in dem dieser Textanteil gespeichert wird. Dieses Attribut ist im wesentlichen der einzige Teil eines Objektes, der für den Benutzer frei editierbar ist, alle anderen Attribute haben gewisse Einschränkungen oder werden automatisch vom System vergeben und somit hat der User keinen Einfluß auf ihre Ausprägung.

Wenn wir das Beispieldokument aus der Abbildung 3.3 betrachten, so sind die

XML-Elemente des Headers Systemvorgaben, d.h. der Benutzer darf sie nicht frei editieren. Durch einen Validierer, der das Dokument nach dem Editieren durch den Benutzer gegen ein Schema prüft, kann sichergestellt werden, dass die Elemente auch nach der Eingabe noch ihren Vorgaben folgen. Somit ließ sich für unser Beispieldokument ein Schema wie in Abbildung 3.6 angeben.

In den Zeilen 29 und 30 wird hier festgelegt, dass unser Dokument genau einen Header hat, die Anzahl der verschiedenen Contents, die in Zukunft angelegt werden, ist hingegen nicht beschränkt. Das Element `<any />` in Zeile 22 definiert, dass hier beliebig strukturierte Contents eingesetzt werden können.

Die Benutzeroberfläche des NPS ist in Java und mit Java Server Pages implementiert. Für die HTML-Ausgabe werden zum einen natürlich direkt die Java Server Pages benutzt, in denen die wesentlichen Bestandteile statisch als HTML-Code realisiert sind, zum anderen werden die dynamischen Anteile durch Servlets in die Seiten eingefügt. Bei diesen dynamischen Anteilen handelt es sich nicht nur um Strings oder primitive Werte, sondern es gibt Bibliotheken, die für bestimmte Funktionalität die richtigen Bausteine zur Verfügung stellen. Dies kann zum Beispiel eine Gruppe von Checkboxen mit "OK" und "Cancel" Buttons sein, die an verschiedenen Stellen verwendet wird. Diese Bausteine werden als Views bezeichnet - ein View wird bei seiner Verwendung nur noch mit den benötigten Parametern gefüllt und dann individuell verwendet. Die in Java implementierten Views werden mit Hilfe der Apache ECS Libraries aufgebaut, die Bausteine für verschiedenen Auszeichnungssprachen wie zum Beispiel HTML als Java-Klassen definieren.

Der oben beschriebene Einsatz von XML-Schema ließe sich problemlos in dieses View-Konzept integrieren. In der jetzigen Situation entscheidet der Designer oder Programmierer, an welchen Stellen welche Bausteine eingesetzt werden – dies ist natürlich abhängig von den benötigten Eingaben und somit von den zurgundeliegenden Datenstrukturen und -typen. Diesen Vorgang könnte man automatisieren, indem man durch eine Analyse des Schemas den Datentyp oder die Datenstruktur bestimmt und dann den entsprechenden View anzeigt. Ein komplexer Datentyp der Art "Header" (Schema-Listing Zeilen 13 - 23) würde zunächst weiter aufgesplittet, bis man bei einem Datentyp ankommt, zu dem ein entsprechender View entweder in den NPS-Bibliotheken oder in den ECS-Libraries existiert. In diesem Fall würde sich folgende Behandlung der Header-Elemente ergeben:

id: Die Id wird vom System bei der Erzeugung vergeben, diese würde lediglich angezeigt.

workflow-id: Das Feld für die Workflow-Id kann editiert werden. Da hier in der

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
3.
4.  <complexType name="attributeType">
5. <sequence>
6.         <element name="id" type="integer"/>
7.         <element name="validFrom" type="date"/>
8.         <element name="validUntil" type="date"/>
9.         <element name="workflow-id" type="integer"/>
10. </sequence>
11. </complexType>
12.
13. <complexType name="headerType">
14. <sequence>
15. <element name="attributes" type="attributeType"/>
16. <element name="permissions" type="permissionType"/>
17. <xsd:element ref="email" minOccurs="0"/>
18. </sequence>
19. </complexType>
20.
21. <complexType name="contentType">
22.        <any />
23. </complexType>
24.
25.
26. <element name="nps-document">
27. <complexType>
28. <sequence>
29. <element name="cm-header" type="headerType"
30.                     minOccurs="1" maxOccurs="1"/>
31. <element name="cm-content" type="contentType"/>
32. </sequence>
33. </complexType>
34. </element>
35. </schema>
```

Abbildung 3.6: Ein mögliches XML-Schema zur Beschreibung des Beispieldokuments.

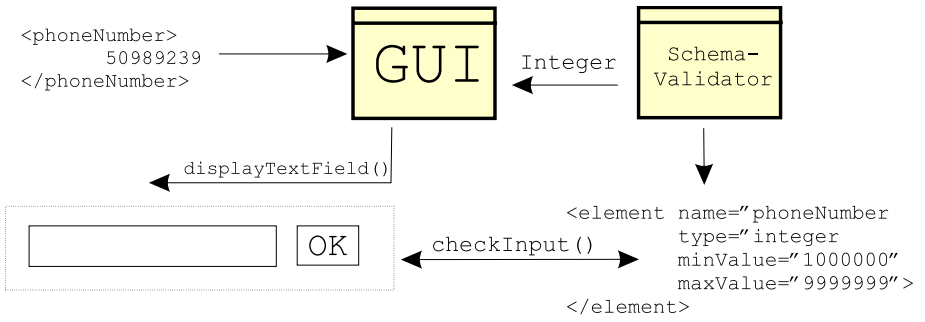

Abbildung 3.7: Dyamisches Erstellen der Eingabefelder in der Benutzerschnittstelle

Regel existierende Workflows referenziert werden, kann zum Beispiel eine Auswahlliste mit den bestehenden Workflows angezeigt werden.

validFrom, validUntil: Da es sich hier um Datumsangaben handelt, kann man wie in HTML-Formularen üblich, eine Kombination aus Auswahllisten zur Datumseingabe anzeigen.

Der entscheidende Punkt sind hier nicht die Anzeigefelder, sondern die Tatsache, dass diese dynamisch aus den Metadaten erzeugt werden können und somit nicht der Applikationsprogrammierer das endgültige Aussehen des GUIs bestimmt, sondern der Autor der Schema-Datei. Dies ist in Abbildung 3.7 beschrieben.

3.4.2 Das Bearbeiten eines Dokumentes

Im vorangegangenen Abschnitt wurde eine mögliche Grundstruktur zur Verwaltung eines Objektes auf XML-Basis beschrieben und das dazugehörige Schema bildet somit das Rahmengerüst, das eine einheitliche Behandlung überhaupt erst möglich macht. Wenn man ein geeignetes Schema zur Objektbehandlung gefunden hat, so wird man dies in der Regel nicht mehr ändern, da die Applikationslogik des Kernels darauf angewiesen ist. Die oben erwähnte Verlagerung der Applikationslogik beschränkt sich demnach auf die Typ- und Werteüberprüfung bei der Eingabe sowie die Anzeige von Elementen. Dies sollte aber nicht unterschätzt werden, da es einen entscheidenden Teil der Benutzeroberfläche darstellt.

Wenn man die Mechanismen zur Validierung und Schemaüberprüfung integriert hat, bieten sich auch bei der Bearbeitung von Contents weitreichende Möglichkeiten. Wie beschrieben, ist eine gewisse äußere Hülle notwendig, um die Abläufe des Content Managements überhaupt möglich zu machen. Danach kann man dem Benutzer aber ermöglichen, innerhalb des Contents selbstdefinierte Schemata zu verwenden, mit deren Hilfe das System dann die Eingaben beeinflußt. Dieses würde exakt die Aufgabe

erfüllen, die zur Zeit die in der Einleitung erwähnten Body-Templates innehaben.

Somit hätte man exzellente Vorraussetzungen geschaffen, um bestimmte Dokumentenprofile zu verwenden, die so zum Beispiel garantieren, dass bestimmte Rechte-, Design- oder Corporate Identity-Vorgaben immer erfüllt sind. Im Gegensatz zu dem derzeitigen Vollständigkeitscheck kann der User auf diese Weise durchgehend kontrolliert werden und falsche Eingaben können sehr früh abgefangen werden.

3.4.3 Der Import und Export von Dokumenten

Um Dokumente zu importieren kann ein großer Teil der bisherigen Programmlogik zunächst bestehen bleiben, denn das Importverhalten von proprietären Formaten wie z.B. Microsoft Word-Dokumenten wird sich nicht verändern - in diesen Fällen würden weiterhin die Blobs aus dem Dokument extrahiert und in die "Content"-Section des CMS-Objektes eingefügt. Da sich XML mittlerweile als Datenaustauschformat etabliert hat, wird aber der Import von XML-Dokumenten immer wichtiger. So bietet zum Beispiel das SAP R/3 System schon alle Schnittstellen und sogenannte *Remote Function Calls* auf XML-Basis an.

Wenn man ein Content Management System professionell einsetzen will, so sind die Schnittstellen zu anderen Systemen unerläßlich. Die Traditionelle Vorgehensweise ist hier, pro angebundenem System einen Adapter einzusetzen, der das interne Datenformat in das Datenformat des Fremdsystems umwandelt. Jedes System besitzt somit spezifische Schnittstellen für jedes andere System, mit dem es kommuniziert. Dies wird allgemein auch als EDI (**E**lectronic **D**ata **I**nterchange) bezeichnet, ein Konzept zum elektronischen Austausch strukturierter Daten. Wie in [RC01] beschrieben, steigt bei dieser Vorgehensweise die Anzahl der Schnittstellen oder Protokolle quadratisch mit $n \cdot (n - 1/2)$. Setzt man statt der individuellen Protokolle eine "Zwischensprache" ein, so steigt diese Zahl nur linear. Dies ist in Abbildung 3.8 verdeutlicht. Beim EDI gibt es einige standardisierte Zwischenformate, die jeweils auf bestimmte Anwendungsdomänen optimiert sind und somit die Anzahl der Schnittstellen bereits deutlich reduzieren.

Durch den Einsatz von XML als primäres Datenformat ergeben sich hier natürlich wesentliche Vereinfachungen. Ein zu importierendes Dokument kann mit Hilfe eines XSLT-Prozessors in das interne Datenformat umgewandelt werden und man benötigt nicht mehr jeweils ein Importmodul für jedes spezifische Datenformat, solange die Darstellung in XML gewählt ist. Analog dazu kann der Export durchgeführt werden. Somit benötigt man nur eine lineare Anzahl von Zwischensprachen und diese können zusätzlich per Stylesheet realisiert werden – das Programmieren einzelner Adapter entfällt somit.

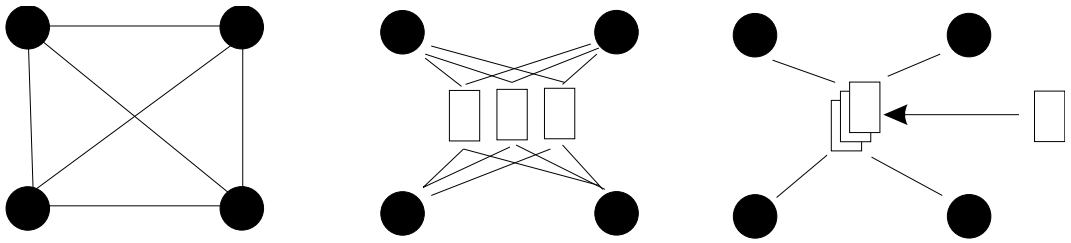

Abbildung 3.8: Die Anzahl der benötigten Interfaces steigt quadratisch ohne Zwischensprache (links), linear mit der Anzahl der verschiedenen Zwischensprachen (mitte). Rechts bleibt das Interface gleich, nur die Anzahl der Stylesheets verändert sich.

Der entscheidende Punkt ist hier genauso wie bei der Benutzerschnittstelle, dass man pro Dokumentenklasse, also einer Menge gleichartiger Dokumente, lediglich eine Schemadatei zur internen Verwaltung und ein Stylesheet pro Schnittstelle nach außen benötigt, ansonsten aber das System vollkommen unabhängig von dieser Dokumentenklasse ist, trotzdem aber individuell reagieren kann.

3.5 Einsatz von XML im Publishing Frontend

Die Möglichkeiten durch einen Einsatz von XML im Publishing-Frontend, also in auf der Export-Seite, sind ähnlich umfangreich wie die auf der Seite des GUIs. Durch die Möglichkeit, XML mit Stylesheets und Prozessoren zu verarbeiten, ergeben sich für Content Management Systeme eine direkte Erweiterung der Funktionalität. Dies äußert sich vor allem in zwei Punkten:

- Unterstützung diverser Ausgabeformate durch XSLT
- Stylesheets können unter die Dokumentenverwaltung des Content Management Systems gestellt werden.

Diese beiden Punkte stellen auch gleichzeitig die Schwächen derzeitiger CMS dar. Wenn man verschiedene Content Management Systeme vergleicht [BSW01], so wird man feststellen, dass das Zielformat mit einigen Ausnahmen primär HTML ist, also die Präsentation und Verwaltung von Webseiten. Das Konzept hinter Content Management Systemen schließt aber andere Verwendungen überhaupt nicht aus, in der Regel sind es technische Probleme, durch die CMS dem Einsatz beispielsweise in Print-Medien momentan noch nicht gerecht werden. Diese Beschränkungen lassen sich mit XML teilweise überwinden - wie weit dies möglich ist, soll in einer prototypischen Implementierung untersucht werden.

Eine eingebaute Stylesheetlogik kann enorme Vorteile für ein System bringen: In den heutigen Systemen gibt es in der Regel nur rudimentäre Template-Prozessoren, die es dem Benutzer ermöglichen, das HTML-Ausgabeformat für eine große Menge von Dokumenten festzulegen. Diese Template-Prozessoren sind in der Regel Eigenentwicklungen, die eine bestimmte Menge von Funktionen auf den Dokumenten zulassen. Da diese Funktionalität recht beschränkt ist, sind die Benutzer auf den Hersteller angewiesen, die benötigten Funktionen für die Anwendungsdomäne des Kunden zu implementieren. Dies verhält sich beim NPS ähnlich.

3.5.1 XML und XSL

Die eXtensible Stylesheet Language (XSL) wurde erstmalig 1997 als Antrag für eine Stylesheet-Sprache eingebracht, doch haben sich aus diesem ersten Vorschlage drei verschiedene Standards entwickelt.

Der erste dieser Standards, **XPath**, stellt einen Mechanismus dar, um Informationen innerhalb eines XML-Dokumentes zu finden. Viele der neueren XML-Datenbankanwendungen benutzen Anfragesprachen, die auf der Basis von XPath-Ausdrücken arbeiten. In Abbildung 3.9 sind einige Beispielanfragen im Bezug auf unser Beispieldokument

Finde alle <title> Elemente:
`XPathString: title`

Finde den Titel des Contents:
`XPathString: //nps-document/cm-content/title`

Finde den Content mit dem Titel "The Boston Tea-Party":
`XPathString: //content/title[. ='The Boston Tea-Party]`

Abbildung 3.9: Beispielanfragen in XQL unter Benutzung von XPath

gegeben.

Der zweite Standard, der sich aus der ursprünglichen XSL-Definition entwickelt hat, ist **XSLT**, die eXtensible Stylesheet Language Transformations. Dieser Standard definiert eine spezielle XML-Syntax über Templates, die zur Transformation von XML-Dokumenten verwendet werden können. Diese Stylesheets werden im Folgenden noch ausgiebig verwendet.

Der dritte Standard schließlich, die eXtensible Stylesheet Language **XSL**, macht Aussagen über die Formatierung eines XML-Dokumentes, indem bestimmte Definitionen für Blöcke des XML-Dokumentes angegeben werden. Diese Formatierung weist eine gewisse Ähnlichkeit zum HTML-Standard auf, da auch hier bestimmte Tags ein definiertes Layout hervorrufen - im Unterschied dazu gibt XSL zwar die Tags vor, das Layout, was diesem zugeordnet wird, kann im Gegensatz zu HTML aber frei definiert werden. Als Beispiel sei ein Block in XML und zwei mögliche Ausgabeformate, "print" und HTML, gegeben:

```
<block>Dies ist ein <block style=''bold''>Text Block</block'></block>
```

Dies ist ein **Text Block**

```
<P>Dies ist ein <B>Text Block</B></P>
```

Dieser Standard verletzt somit die eigentliche Intention, durch XML eine Trennung von Layout und Inhalten zu erzielen. Dies ist weniger gravierend, da diese Spezifikation eigentlich erst zur Layout-Zeit zum tragen kommt. Da die oben genannten Techniken alle aus der XSL-Spezifikation stammen, werden die Bezeichnungen oft inkonsistent benutzt. Die Grafik aus [Bra00a] in Abbildung 3.10 soll dies verdeutlichen.

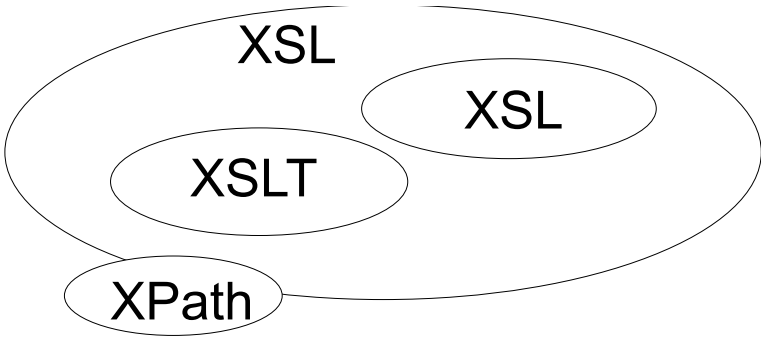

Abbildung 3.10: Die verschiedenen Verwendungen des XSL-Begriffs

3.5.2 XML und XSLT

Bei der Transformation mit Hilfe der eXtensible Stylesheet Language Transformations (XSLT) wird jeweils für eine Klasse von Dokumenten ein Stylesheet angewandt, dass die dokumentenspezifische Struktur des Ausgangsformates in bestimmte Zielformate umsetzen kann. Dazu durchläuft der eingesetzte XSLT-Prozessor den Dokumentenbaum eines XML-Dokuments und sucht für jedes auftretende Element eine passende Regel im Stylesheet. Diese Regel macht Aussagen darüber, unter welchen Bedingungen das Ausgangselement behandelt werden soll und welche Aktionen darauf ausgeführt werden sollen. Die Kriterien für die Behandlung eines Knoten werden in einem Pattern festgelegt, was der XPath-Notation [CD99] folgt. Die Transformationen mit Hilfe von XSLT zielen nicht notwendigerweise immer auf ein anderes XML-Format ab, sondern können auch beliebige andere Textformate sein. Um binäre Datenformate aus XML-Dokumenten zu erzeugen benötigt man hingegen die sogenannten Formatting Objects (FO). Bei XSL Transformationen handelt es sich immer um die in Abbildung 3.11 dargestellte Transformation von Bäumen von einer Darstellung in eine andere.

3.5.3 XML und Formatting Objects (FO)

XSL Formatting Objects definieren einen festen Namensraum (namespace) und sind Teil der XSL-Spezifikation. Bei der Transformation mit Hilfe von XSLT wird der XML-Baum in die Darstellung des gewünschten Formatting Objects umgewandelt. Dies kann zum Beispiel die Definition eines Absatzes, einer Seite oder einer Überschrift sein und enthält normalerweise das Layout betreffende Informationen, wie z.B. Einstellungen für Ränder oder Schriftgrößen und -typen. Auf dem Weg von der ursprünglichen XML-Darstellung bis zur formatierten Ausgabe sind in der Regel viele Formatting Objects beteiligt, von denen jedes einzelne für bestimmte Eigenschaften des Outputs verantwortlich ist. Diese Verantworlichkeiten können zum Beispiel

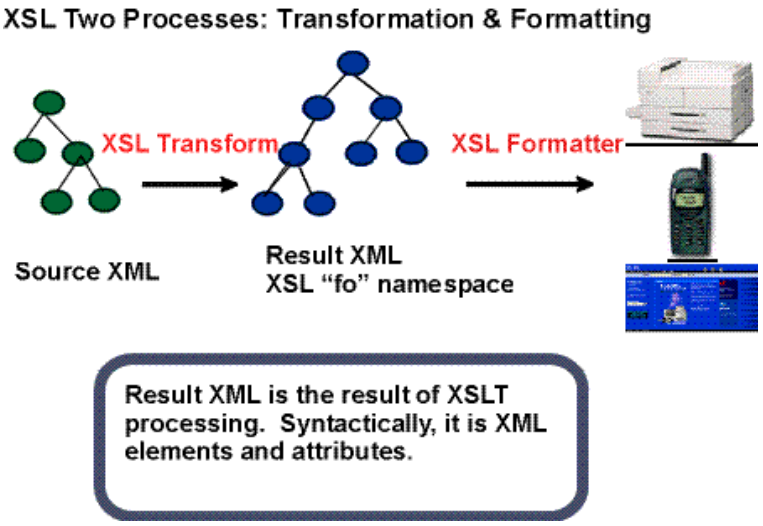

Abbildung 3.11: Transformation von XML-Baumstrukturen mit Hilfe von XSLT und XML-Formatting Objects (FO) [Quelle: W3C]

das Umbrechen der Zeilen oder die Aufteilung von Absätzen auf Seiten sein. Die Spezifikation des W3C für FOs abstrahiert von den konkreten Details und gibt lediglich vor, was die implementierenden FOs können müssen. Dementsprechend läßt die Spezifikation viel Platz für zukünftige FOs - derzeit existieren noch recht wenige davon. XML-Formatting Objects sind analog zu XSLT in einer XML-basierten Auszeichnungssprache formattiert, die durch den oben genannten Namensraum definiert wird und von einem XML-FO Prozessor interpretiert werden und dabei das gewünschte Datenformat erzeugen kann. Als Beispiel ist hier eine Definition eines Seitenrandes zu sehen (entlehnt aus [Eis01]).

```
<fo:layout-master-set>
    <fo:simple-page-master master-name="page"
         page-height="12cm"
         page-width="12cm"
         margin-top="0.5cm"
         margin-bottom="0.5cm"
         margin-left="1cm"
         margin-right="0.5cm">
     </fo:simple-page-master>
</fo:layout-master-set>
```

Mit Hilfe dieser Techniken sollte es möglich sein, den Funktionsumfang von Content Management Systemen bei der Ausgabe erheblich zu steigern. Jede Implementie-

rung eines Prozessors für Formatting Objects muss den XML-FO Namespace abdecken und somit das beschriebene XML-Format interpretieren können. Nach dem Interpretieren wird in dem XML-FO Prozessor ein Renderer aufgerufen, der das gewünschte Ausgabeformat generiert. Dies können sowohl Text- und XML-basierte Formate sein oder auch Binärformate wie Postscript oder das Portable Document Format (PDF) von Adobe.

3.5.4 Die NPSOBJ-Syntax und System-Execute Prozeduren

Im bisherigen NPS werden die Daten bei der Veröffentlichung durch die NPS-eigene Template-Logik verarbeitet. Diese Template-Logik ist auf die Erzeugung von HTML-Code spezialisiert und die zu veröffentlichenden Objekte haben in der Regel einen Content, der zusätzlich zum eigenlichen Text umschließende HTML-Elemente enthält. Außerdem können die Templates bei der Verarbeitung umhüllenden Code einfügen. Um dynamische Contents zu erzeugen, können in diese Templates, die die Grundlage für die fertigen HTML-Seiten darstellen, sogenannte NPSOBJ-Tags eingefügt werden, die bei der Vorschau sowie beim Export durch Daten aus dem Content Management Server ersetzt werden. Zur Benutzung der NPSOBJ-Syntax stehen verschiedene Befehle zur Verfügung, mit deren Hilfe zum Beispiel Inhaltverzeichnisse oder Navigationsleisten, die die Struktur der Dokumente im Content Management System abbilden, generiert werden können.

Einige Beispiele hierfür sind:

- parent: liefert die dem Objekt übergeordnete Publikation
- next: liefert das nächste Objekt in der aktuellen Publikation
- previous: analog zu next
- up: liefert das nächst höhere Objekt in der Publikationshierarchie

Ein sehr einfaches Template, indem nur ein <META>-Element mit dem aktuellen Titel des exportierten Objektes eingefügt wird, könnte zum Beispiel wie folgt aussehen:

```
<html>
   <head>
      <npsobj insertvalue="meta" name="keywords" content="title">
      </npsobj>
   </head>
   <body>
   </body>
</html>
```

Die NPSOBJ-Syntax ist somit im Wesentlichen eine künstliche Erweiterung von HTML: Die NPSOBJ-Elemente werden bei der Vorschau oder bei der Veröffentlichung vom System aus dem Code herausgeparst und durch die entsprechenden Werte ersetzt. Dieses beschränkt sich aber auf tatsächlich im Kernel implementierte Funktionen, d.h. wenn man die Template-Logik erweitern will, so muss man die neuen Funktionen im Kernel implementieren. Somit ist die Funktionalität schlecht erweiterbar und viele Konstrukte wie z.B. Schleifen lassen sich nur schwer umsetzen.

Um die NPSOBJ-Funktionalität bei Bedarf ohne Kerneländerungen zu erweitern, werden die sogenannten SystemExecute-Prozeduren benutzt. Diese bieten einen kontrollierten Ausstieg aus der Template-Logik und sorgen dafür, dass komplexere Aufgaben und Layouts durch Scripte abgearbeitet bzw. erzeugt werden können. Die SystemExecute-Prozeduren setzen auf der NPS-Tcl-Schnittstelle auf und sind somit genauso mächtig wie die Tcl-Umgebung.

Diese Prozeduren haben aber entscheidende Nachteile, da man durch die Benutzung Code ausführt, der außerhalb des Content Management Servers abläuft und deshalb nicht unter Kontrolle des NPS steht. Das bedeutet konkret, dass diese Prozeduren keinem Versionsmanagement unterliegen, nicht die Workflow-Mechanismen des Systems benutzen und auch beim Export auf das Live-System nicht automatisch mitgeliefert werden, sondern manuell kopiert werden müssen. Diese Probleme haben Templates nicht, da sie als normale NPS-Objekte mitverwaltet werden und somit auch bei Bedarf eine alte Version eines Templates problemlos wiederhergestellt werden kann. Durch die Benutzung der NSPOBJ-Syntax kann man somit erhebliche Konsistenzprobleme bekommen.

Im Gegensatz zur NPSOBJ-Syntax besitzt XSL eine deutlich umfangreichere Funktionalität, wie zum Beispiel nützliche Kontrollstrukturen, wie `<xsl:when />`, `<xsl:otherwise />` oder `<xsl:for-each />`.

3.5.5 Das resultierende Gesamtsystem

Als Gesamtsystem ergibt sich somit eine Architektur mit dem dynamischen View-Konzept auf der GUI-Seite, einer Speicherstruktur, die auf den jeweiligen Anwendungskontext hin optimiert ist und einer XSLT-Template Engine zusätzlich zur bestehenden Template-Engine für die bisherige NPSOBJ-Syntax. Der Template-Mechanismus auf der Publishing-Seite soll prototypisch implementiert werden.

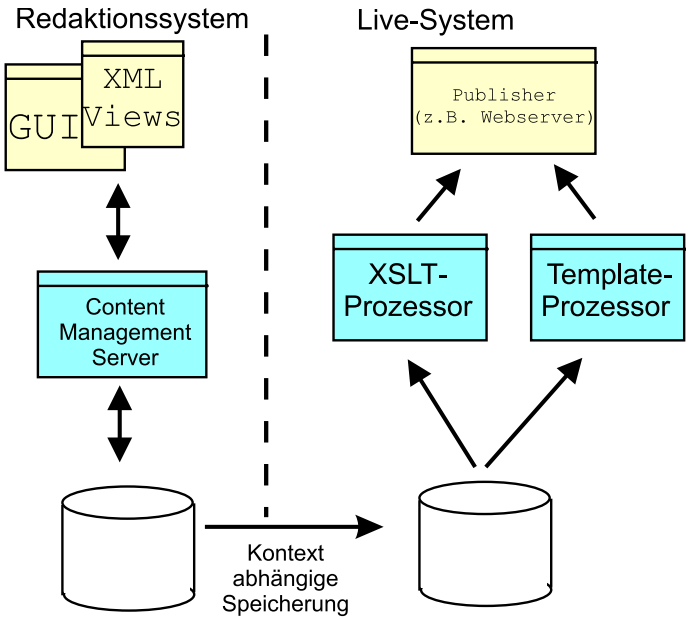

Abbildung 3.12: Aufbau des Gesamtsystems mit XML-Unterstützung

4 Auswahl eines XSLT-Prozessors zur Verwendung im NPS

Die Auswahl des Prozessors zur Verarbeitung von XML-Dokumenten mit XSL-Stylesheets im NPS ist ein entscheidendes Kriterium für den Erfolg dieses Features. Aufgrund des Aktualität der Techniken rund um die eXtensible Markup Language sind viele Implementierungen noch in der Entwicklung - als Folge davon sind oft nicht alle Anforderungen der XSL-Spezifikation [Cla99] umgesetzt, die Prozessoren haben noch Bugs oder die Performance lässt keinen praktischen Einsatz zu. Die Featureübersicht der einzelnen Implementierungen kann einfach verglichen werden, um die Geschwindigkeit zu beurteilen bedarf es jedoch ausgewogener Tests. Wie in anderen Bereichen der Software- und Rechnerentwicklung auch können auch für XSLT-Prozessoren Benchmark-Suiten zur Hilfe genommen werden. Im Rahmen dieser Arbeit soll die Benchmark-Anwendung der Firma Data Power Technology, Inc zur Bewertung herangezogen werden.

In den entsprechenden Quellen im WWW findet man oft Informationen über einen weiteren XSLT-Benchmark mit der Bezeichnung "XSLBench" [Jon00]. Die Tests von XSLBench sind jedoch in XSLTMark integriert und brauchen demnach nicht separat durchgeführt zu werden.

4.1 XSLTMark - Ein Benchmark für XSLT-Prozessoren

XSLTMark ist ein Benchmark um umfassend die Performance von XSLTProzessoren zu untersuchen. Dieser Benchmark besteht aus 40 Testszenarien, um die verschiedenen funktionalen Anforderungen an einen XSLT-Prozessor zu messen und vergleichbar zu machen.

4.1.1 Das Bewertungssystem von XSLT-Mark

Es gibt viele verschiedenen Ansätze zur Bewertung von Geschwindigkeit. Viele Benchmark-Tests benutzen abstrakte, einheitenlosen Ergebnisse um die Performace eines Systems zu bewerten. Hierbei werden oft die gewichteten Mittel einzelner Werte zu einem Gesamtwert kombiniert. Diese Bewertung eignet sich gut, um die relative Geschwindigkeit eines Systems zu beurteilen, allerdings lassen sich kaum Aussagen darüber treffen, wie schnell das System in der Praxis wirklich ist. Prominente Vertreter diese Art von Benchmark sind die SPEC-Benchmarks, bei denen ein System immer im Verhältnis zu einem Referenzsystem bewertet wird. Dieses Referenzsystem bekommt eine Performance von 1 zugeordnet [SPE01].

In anderen Benchmarks wird die mittlere Ausführungszeit als Einheit für eine kleine Nummer von Tests genommen - je geringer die Ausführungszeit, desto schneller das System. Die Autoren von [KD01] argumentieren dagegen, dass hierbei ebenso wie bei der oberen Methode keinerlei Aussagen über das tatsächliche Verhalten einer (in diesem Fall) Softwarekomponente in einem System getroffen werden. Da XML sich mittlerweile als Fomat zur Datenübertragung etabliert hat und somit ein Schwerpunkt auf dem Netzwerk liegt, wird bei XSLTMark die Performance in Kilobyte pro Sekunde gemessen. Diese Zahl ergibt sich aus dem Mittel von Input- und Output-Datenmenge. Weitere Einblicke in die tatsächliche Leistung kann man erhalten, wenn man das Verhältnis von Input, Output und Komplexität der durchgeführten Transformation näher betrachtet.

4.1.2 Das Auswertungsverfahren

Beim XSLTMark-Benchmark werden bei der Erstellung der Testergebnisse die Vollständigkeit der Prozessoren, also die Konformität zur XSLT-Spezifikation, berücksichtigt, da ansonsten Unregelmässigkeiten auftreten könnten. Ein Prozessor, der nur einige Features der Spezifikation implementiert, kann auf diese hin optimiert sein und demnach extrem gute Ergebnisse in wenigen Disziplinen erreichen – demnach geht eine vollständige Implementierung der W3C-Spezifikation in der Regel zuungunsten der Geschwindigkeit aus.

Diesem Verhalten wird beim XSLTMark-Benchmark durch den Wert “Conformance Score” Rechnung getragen. Dieser Wert berechnet sich aus der Anzahl der absolvierten Tests geteilt durch die Gesamtanzahl der Tests und gibt demnacht den Grad der Konformität zur XSLT-Spezifikation an. Zusätzlich zur Berücksichtigung der Features werden die erzeugten Dokumente noch mit Hilfe des Tools “dgnorm” normalisiert, was zum Beispiel das Entfernen von unnötigen Leerzeichen (Whitespace) beinhaltet, damit der gemessene Output der tatsächlichen Datenmenge entspricht.

4.2 Die Tests

Die Tests für XSLTMark sind auf das Anwendungsprofil typischer XML-Anwendungen zugeschnitten und reichen vom einfachen Ausfüllen eines nahezu statischen HTML-Formular bis hin zu komplexen Transformationsaufgaben. Die vier Hauptkomponenten sind:

- XSLT Template Pattern Matching und template Instanziierung
- XSLT Kontrollstrukturen und Parameterübergabe
- XPath Selektion von Knotenmengen und Attributen
- XPath Library Funktionen wie Operationen auf Strings und Knotenmengen

Einzelheiten zu den Tests sowie die Informationen über die verwendeten Stylesheets sind unter [KD01] zu finden, hier sollen jedoch nur Beispiele für das Spektrum des Benchmarks gegeben werden.

alphabetize: nimmt einen XML-Baum und sortiert die Elemente alphabetisch. Verwendung von xsl:select und xsl:sort.

breath: sucht ein einzelnes Element in einem grossen Dokumentenbaum.

dbonerow: wählt eine einzige Zeile aus einer Tabelle mit 10000 Einträgen aus.

metric: konvertiert metrische Maße in englische Maße.

oddtemplate: matcht verschiedene komplexe Pattern.

total: generiert Informationen über Sales-Daten.

tower: löst das Problem der “Türme von Hanoi”.

xpath: pattern matching auf der Basis von XPath.

Anhand dieser Auswahl kann man schon erkennen, dass die Zusammensetzung der Tests (wie für einen Benchmark notwendig) sehr vielfältig ist - das Spektrum reicht vom simplen Ersetzen bis zur Abarbeitung von komplexeren Algorithmen. Als Systemumgebung für die Tests steht ein Intel Celeron (Mendocino) mit 433 Mhz Taktfrequenz und 256 MB RAM zur Verfügung. Als Betriebssystem wird SuSe Linux 7.1 mit dem Kernel 2.2.14 verwendet.

Es soll ein möglichst breites Spektrum der zur Zeit verfügbaren XSLT-Prozessoren

getestet werden. Wie schon oben erwähnt kommen zur Integration im NPS tendenziell Prozessoren auf C/C++ Basis in Frage, allerdings sind noch zwei Java-Implementierungen (XalanJ und XT) unter den Testkandidaten, da man so den Vergleich zwischen den Sprachen ziehen kann. Für die Tests werden eingesetzt:

- Sablotron XSL Transformations Processor, Version 0.60, Ginger Alliance s.r.o.
- Xalan C++, Version 1.1, Apache Software Foundation
- Transformixx, Mozilla Projekt
- XalanJ, Version 2.0.0, Apache Software Foundation
- James Clark's XT

Der Java-Prozessor von Oracle kann nicht bewertet werden, da er beim Durchlauf der Tests regelmässig abstürzt, ebenso der C++ XSLT-Prozessor von Oracle, da hierfür kein Testtreiber vorhanden ist. Auch die Gnome-Bibliotheken lassen sich zur Zeit nicht mit den Benchmarks betreiben. Zusätzlich zu den bisher beschriebenen Prozessoren macht außerdem der Microsoft-Prozessor in der Literatur einen guten Eindruck. Da dieser aber auf die Windows-Plattform beschränkt ist, soll er hier nicht evaluiert werden.

Abbildung 4.1 zeigt das relative Verhältnis der einzelnen Prozessoren untereinander in KB/s an. Dieser Wert sagt allerdings nicht unbedingt viel aus, da einige Prozessoren , wie oben schon erwähnt, nur wenige Features unterstützen, diese dann aber sehr schnell bewältigen.

Um dieses Ergebnis zu bereinigen, haben die Autoren der Benchmark-Suite noch einen eigenen Wert, den des "Conformance-Score" hinzugefügt. Dies ist einfach nur die Anzahl der bestandenen Tests geteilt durch die Gesamtzahl der Tests, also eine prozentuale Angabe der bestandenen Tests. Dieser Conformance-Score ist in Abbildung 4.2 zu sehen.

Messergebnisse

Um einen brauchbaren Gesamtwert zur Beruteilung von Prozessoren zu bekommen, reichen die bisherigen Werte nicht aus. Wir haben zwar den Durchsatz und den 'Conformance-Score, diese stehen aber (noch) in keinem Verhältnis – das ist aber gerade der gewünschte Gesamtfaktor. Also wird hier ein weiterer Parameter eingeführt, der sogenannte "Usability-Faktor". Dieser errechnet sich in unserem Fall aus

Messungen nach Zeit

Driver:	Xalan Java Zeit[ms]	XT Zeit[ms]	Sablotron Zeit[ms]	Xalan C Zeit[ms]	Transformixx Zeit[ms]
alphabetize	8895	1526	5459	5275	8515
attsets	4901	1239	N/A	1445	5651
avts	12983	5017	12567	4505	18166
axis	3484	995	N/A	585	1425
backwards	6691	1811	4673	2819	7524
bottles	6263	1900	7510	5743	N/A
breadth	7391	1732	4781	2687	9787
brutal	5362	1223	4095	2113	4401
chart	5083	1365	4115	2071	4246
creation	11101	3931	15150	7583	37207
current	4335	770	1204	675	N/A
dbonerow	164917	35384	80025	52051	N/A
dbtail	10161	3040	14160	5984	10913
decoy	28830	7013	44862	27531	139370
depth	7014	1861	5412	3052	6948
encrypt	2561	560	3878	2703	17373
functions	39924	15306	33049	22835	15725
game	3134	1008	1820	771	1272
html	3059	658	1152	606	948
identity	13715	3963	14967	8624	344749
inventory	5009	1216	4203	2056	3139
metric	4157	N/A	2421	1359	2066
number	N/A	1195	N/A	1497	1518
oddtemplate	3044	603	1350	2658	1154
patterns	16553	5309	30781	20133	90147
prettyprint	14050	3342	23323	14249	126254
priority	3206	881	1774	790	1614
products	15070	1992	3751	1913	N/A
queens	3794	876	13288	5926	N/A
reverser	4026	3275	7676	4973	741
stringsort	14446	5521	16543	13846	217709
summarize	23887	1471	2097	1039	1362
total	3536	908	1251	605	764
tower	11717	3265	47163	25699	235
trend	45106	9072	23784	39603	39733
union	3224	597	1167	584	810
xpath	3239	607	1091	1197	1563
xslbench1	3634	1118	3288	1283	7205
xslbench2	13355	5366	13491	6450	157067
xslbench3	10328	4283	5963	3171	3641

Messungen nach Durchsatz (Kilobyte pro Sekunde)

Driver:	Xalan Java KB/s	XT KB/s	Sablotron KB/s	Xalan C KB/s	Transformixx KB/s
alphabetize	19,86	115,76	32,36	33,49	53,33
attsets	38,49	152,24	95,35	130,53	33,23
avts	120,27	311,24	124,25	346,61	85,91
axis	13,79	48,29	N/A	82,13	40,88
backwards	39,03	144,19	55,87	92,63	34,6
bottles	90,6	298,65	75,8	98,8	N/A
breadth	41,72	178,03	64,48	114,76	31,42
brutal	48,92	214,48	62,41	124,14	76,62
chart	47,29	176,1	56,56	116,07	87,89
creation	112,78	318,48	82,63	165,1	33,64
current	5,63	31,71	20,24	36,17	N/A
dbonerow	59,52	277,39	122,65	188,57	N/A
dbtail	121,54	406,24	87,21	206,38	113,09
decoy	68,34	280,95	43,92	71,57	13,85
depth	68,62	258,62	88,92	157,7	69,15
encrypt	76,8	351,21	50,71	72,76	11,32
functions	32,37	84,43	39,1	56,59	89,48
game	48,47	150,7	83,12	197,02	136,66
html	12,43	52,76	32,85	62,77	20,35
identity	143,1	495,22	131,13	227,57	5,69
inventory	38,62	159,09	46,02	94,09	36,13
metric	18,56	N/A	30,33	55,04	38,31
number	N/A	41,19	16,25	29,71	93,64
oddtemplate	5,31	26,8	12,44	6,08	15,4
patterns	119,03	371,13	64,02	97,86	21,41
prettyprint	57	239,63	27,72	56,2	19,4
priority	23,13	84,19	41,78	93,89	45,68
products	7,85	59,39	31,22	61,85	N/A
queens	2,31	10,01	0,66	1,48	N/A
reverser	32,2	39,58	16,88	26,07	94,43
stringsort	134,84	352,82	117,75	140,68	8,95
summarize	15,53	252,17	176,87	357,02	269,95
total	21,91	85,34	61,08	128,08	123,99
tower	58,99	211,71	14,66	26,9	10,43
trend	10,9	54,19	20,67	12,41	12,55
union	4,45	24,05	12,26	24,58	16,7
xpath	7,58	40,46	22,47	20,52	15,18
xslbench1	97,07	313,36	105,42	274,43	58,08

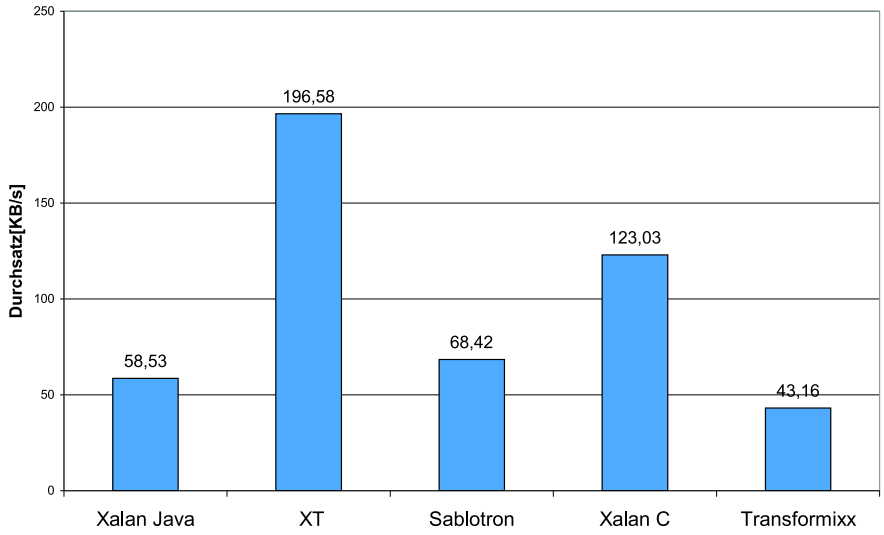

Abbildung 4.1: Durchsatz der einzelnen Prozessoren in KB/s

Messungen nach Durchsatz (Forts.)

Driver:	Xalan Java KB/s	XT KB/s	Sablotron KB/s	Xalan C KB/s	Transformixx KB/s
xslbench2	189,43	470,52	186,36	391,45	16,43
xslbench3	142,6	343,87	246,97	464,46	404,25
arithm.Mittel	58,53	196,58	68,42	123,03	43,16
geom. Mittel	36,36	135,16	44,95	75,93	28,78
harmon. Mittel	17,61	76,45	13,17	26,21	18,78
Conf. Score[%]	87,5	90	75	92,5	32,5

$$usability = conformance[\%] * speed[KB/s]$$

Dieser Faktor unterstützt somit zum einen die Geschwindigkeit, zum anderen aber auch die Vollständigkeit der Umsetzung der XSLT-Spezifikation. In wie weit die beiden Ausgangsgrößen hier noch mit unterschiedlichen Gewichtungen versehen werden müssen, hängt vom Anwendungskontext ab: Es mag Fälle geben, in denen nur simple Transformationen verwendet werden, dann sind die meisten Tests dieses Benchmarks für den Kontext nicht interessant und es würde sich hier eine ganz andere Gewichtung ergeben.

Dass dieser Faktor aber gerade in unserem Fall durchaus sinnvoll ist, lässt sich durch zwei Hauptargumente untermauern:

1. Geschwindikeit: Der Export eines sehr großen Webservers kann relativ lange ($>$ 1 Stunde) dauern und gerade bei einem Live-Betrieb ist dies ein kritischer Zeitraum. Wenn man also sämtliches Processing mit XSLT durchführen will, so ist ein schneller Prozessor unabdingbar.

2. Konformität: Ziel einer Verwendung von XSLT in Content Management Systemen kann nur sein, immer weniger proprietäre Template-Logik einzusetzen und stattdessen die volle Funktionalität der Templates in Stylesheets und Prozessoren zu verlagern. Deshalb sollte der Featureumfang so groß wie möglich sein und wird dementsprechend hier berücksichtigt.

Ein relativer Vergleich von Performance und Usability ist ebenfalls in Abbildung 4.2 dargestellt.

4.3 Ergebnisse

Eine Interpretation der Testergebnisse stellt gerade im XML-Bereich nur eine Momentaufnahme dar, da die Entwicklung permanent vorangetrieben wird. Auch viele der derzeitigen XSLT-Parser haben derzeit den Stand von Prototypen oder Entwicklungsversionen, was sich vor allem an dem Unterschied zwischen der eigentlichen "Geschwindigkeit" und der hier vorgestellten "usability" zeigt. Sowohl beim Datendurchsatz als auch bei der Kombination von Durchsatz und Konformität zur XSLT-Spezifikation stellen XT (für Java) und Xalan C (für C++) die Spitzengruppe dar.

Für die Ergebnisse der Oracle- und Gnome-Implementierungen muss auf die Ergebnisse von [KD01] zurückgegriffen werden. Diese zeigen, dass diese beiden Implementierungen nicht überragend schnell sind. Vor allem aber wird deutlich, dass

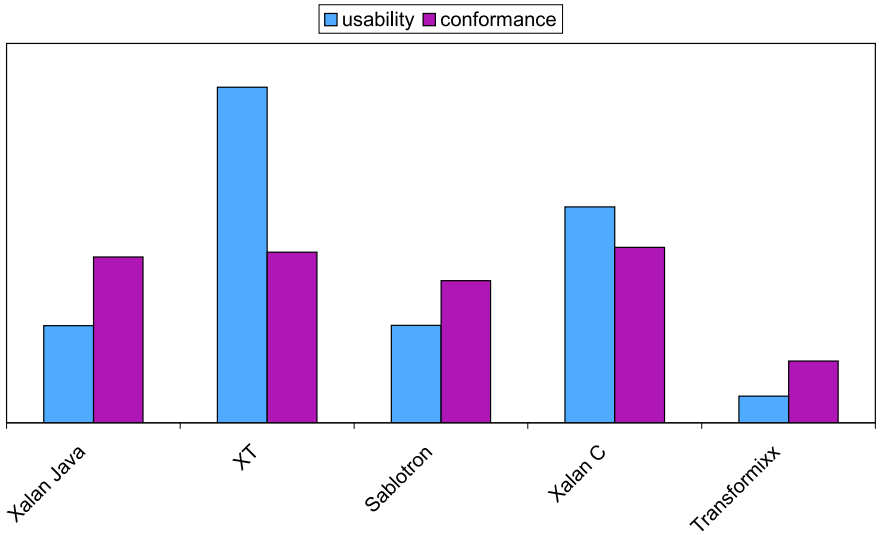

Abbildung 4.2: Verhältnis von “Usability” und “Conformance” der einzelnen Prozessoren untereinander

diese beiden einen schlechten Conformance-Score haben und damit die Spezifikation nur unzureichend erfüllen. Gerade dieser Faktor sollte in der derzeitigen Situation ein Entscheidungskriterium sein, da in Zukunft sicherlich noch einige neue Produkte und Versionen erscheinen werden, eine jetzige Verwendung aber nach möglichst konformen Prozessoren verlangt, um dauerhafte Planungssicherheit zu haben – der Prozessor kann später ausgewechselt werden, aber man sollte möglichst jetzt schon sein System mit dem vollen XSLT-Funktionsumfang ausstatten, um die Vorteile dieses Standards zu nutzen.

James Clark und die Apache Software Foundation werden ihrer Vorreiterrolle auf dem XML-Sektor gerecht und demnach wäre für eine Verwendung im derzeitigen NPS die C++ Implementierung von Xalan die Alternative der Wahl.

5 Implementierung eines Prototypen

5.1 Ziele einer prototypischen Implementierung

Die prototypische Implementierung einiger der oben vorgestellten Techniken soll eine bessere Bewertung der einzelnen Komponenten ermöglichen. Insbesondere sollen folgende Ziele erreicht werden:

- Integration einer Template-Engine, die die extern ablaufenden SystemExecute Prozeduren obsolet macht und somit durchgehendes Content Management für sämtliche Templates, inklusive Versionierung, ermöglicht.
- Öffnung des Systems für diverse XML-basierte Ausgabeformate durch Anwendung verschiedener Templates auf den gleichen Ausgangscontent
- Öffnung des Systems hin zu anderen Ausgabeformaten, z.B. zum Einsatz in Printmedien

Um dies zu erreichen, sollen zum einen XSL-Transformations eingesetzt werden, mit denen man jeden XML-Baum in einen anderen übersetzten kann. Außerdem sollen XSL-Formatting Objects integriert werden, die aus einem speziellen XML-Baum verschiedenen nicht XML-basierte Ausgabeformate erzeugen können.

Als Ergebnis der Tests aus dem vorangegangenen Kapitel wird für die XSLT - Transformationen die Xalan C++ Implementierung der Apache Software Foundation verwendet. Diese verfügt auch über ein C-API und kann demnach in der Objective C Umgebung des NPS-Kernels angesprochen werden. Für die Verarbeitung von XSL Formatting Objects steht zur Zeit nur eine freie Implementierung zur Verfügung, nämlich FOP der Apache Software Foundation in der Version 0.20.1, welches eine Java-Implementierung ist.

5.2 Eingliederung der XSLT-Komponenten im System

In Abbildung 3.12 ist der Aufbau des Gesamtsystems dargestellt. Dies ist der gewünschte Aufbau für eine unabhängige Verwendung entweder des herkömmlichen NPSOBJ-Prozessors oder des XSLT-Prozessors. Um dies zu erreichen, wird aber auch eine Infrastruktur vorausgesetzt, die beide Prozessoren gleichberechtigt behandelt.

5.2.1 Die Erzeugung von XML-Dokumenten im NPS

Um einen XSLT-Prozessor und XSL-Stylesheets zu benutzen, ist es zunächst notwendig, die relevanten Daten aus dem System als XML-Dokumente zur Verfügung zu stellen. Wie in Kapitel 3 beschrieben, stellt bei einer entsprechenden Infrastruktur die Generierung der XML-Daten kein großes Problem dar, da man zum Beispiel mit Datenbankanfragen in XQL die notwendigen Zusammenhänge direkt in einem XML Dokument darstellen kann. Da das NPS aber weiterhin mit relationalen Datenbanken arbeitet, benötigen wir in diesem Prototypen eine andere Methode, um diese Daten zu erhalten.

Diese Aufgabe kann der bisherige NPSOBJ-Prozessor übernehmen, da dieser über Funktionen verfügt, um auf den Content Management Server zuzugreifen. Das bedeutet für den Prototypen, dass man den NPSOBJ-Prozessor in der "Rendering-Pipeline" vor dem XSLT-Prozessor einhängt und damit dann die Beziehungen innerhalb des Content Management Systems verfolgen kann.

Die jetzige Template-Logik stellt verschiedene Befehle zur Verfügung, die bei genauerer Untersuchung in zwei Klassen unterteilt werden können.

Content Management Befehle: Diese bewirken den Zugriff auf die Objekte des Content Management Servers, wie zum Beispiel das Holen der Objekte selber, das Verfolgen von Links oder das Iterieren über Kinder eines Objekts.

Formatierungsbefehle: Diese kontrollieren die Ausgabe der Inhalte, also Formatierung und Sortierung von bestimmten Attributwerten, das Erstellen von HTML-Code, etc.

In diesem Prototypen sollen die Formatierungsmöglichkeiten untersucht werden, die mit Hilfe von XML möglich werden, die Änderungen im Bezug auf die Content-Management-Befehle können hier nur theoretisch behandelt werden, da sie weitreichende Modifizierungen am Gesamtsystem erfordern würden. Die Daten werden

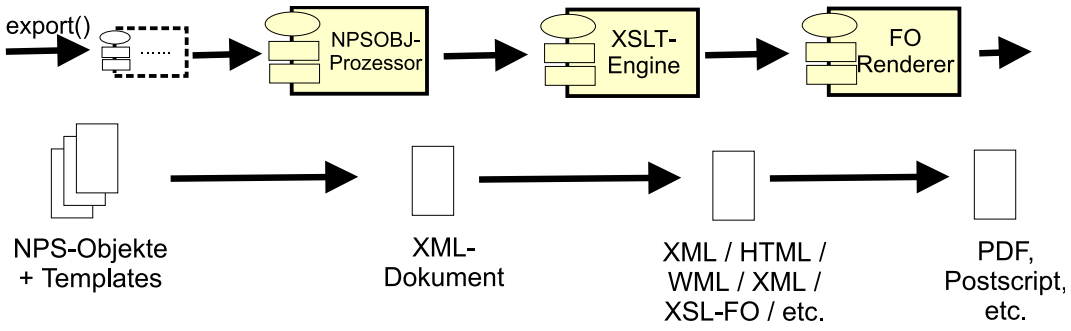

Abbildung 5.1: Die "Rendering-Pipeline" eines Prototyps mit XSLT-Prozessor

somit zunächst auf herkömmliche Weise verarbeitet und mit entsprechenden Templates werden hieraus XML-Dokumentenbäune generiert, die dann an den XSLT-Prozessor übergeben werden können. Dies ist in Abbildung 5.1 dargestellt. Somit hat der XSLT-Prozessor über den NPSOBJ-Prozessor uneingeschränkten Zugriff auf den Content Management Server.

5.2.2 Objective C Syntax

Da die Sprache Objective C nicht ähnlich verbreitet ist wie zum Beispiel Java oder C++, sollen hier einige Anmerkungen zum besseren Verständnis der Code-Beispiele beitragen.[1]

Objective C ist eine objektorientierte Sprache, die einige der objektorientierten Konzepte beinhaltet, wie Vererbung, Polymorphismus und dynamisches Binden. Die Kommunikation zwischen Objekten erfolgt über sogenannte Nachrichten:

`[`*Empfänger* `nachricht]`

Der hier dagestellte Empfänger ist das angesprochene Objekte und die Nachricht entspräche in Java der aufgrufenen Methode. Im Gegensatz zu Java erfolgt in Objective C auf die Typisierung dynamisch, d.h. es wird erst zur Laufzeit entschieden, ob das empfangende Objekt die Nachricht auch tatsächlich interpretieren kann. Die Argumente werden durch Doppelpunkte getrennt an die Nachricht angehängt. So könnte man zum Beispiel die Verschiebung eines Rechtecks zu bestimmten Koordinaten hin aufrufen mit

```
[myRectangle moveTo:30.0 :50.0]
```

[1] Siehe dazu auch [Ope95]

wobei entweder die Rechteckklasse oder eine ihrer Oberklassen die `moveTo` Methode implementieren muss.

```
- moveTo:(NXCoord)x :(NXCoord)y;
```

Der hier oft verwendete Objective-C-Datentyp **id** entspricht dem Java-Datentyp **Object**, also der allgemeinen Oberklasse.

5.2.3 Die Dokumententypen XMLPublication, XMLDocument und XMLTemplate

Im NPS legen die vordefinierten Objekttypen eine eventuelle Templateverarbeitung fest. So werden zum Beispiel Generics und Images gar nicht vom Template-Prozessor verarbeitet, Dokumente und Publikationen werden hingegen mit Hilfe des Dokumententyps *Template* verarbeitet. Ausserdem sind in den Dokumententypen, die als Klassen in Objective C definiert sind, voreingestellte Werte für die detaillierten Schritte während der Verarbeitung festgelegt, so zum Beispiel, ob interne Links über den sogenannten URI-Prozessor aufgelöst werden. Für alle bisherigen Typen wird der XSLT-Prozessor auch weiterhin nicht benötigt. Um die neue Funktionalität automatisch auf Dokumente anzuwenden, werden drei neue Dokumententypen eingeführt, nämlich *CMXMLDocument*, *CMXMLPublication* und *CMXMLTemplate*. Die Position in der Vererbungshierarchie der NPS-Objekte ist in Abbildung 5.2 zu sehen.

Die Klassen XMLPublication und XMLDocument verhalten sich im Wesentlichen wie Publikation und Document. Wie in Abbildung 5.3 zu sehen ist, wird die Methode `supportedContentTypes` umgeschrieben, sodass auch der MimeType PDF als gültiger Content-Typ akzenptiert wird. Dies ist notwendig für das spätere Rendering mit Formatting Objects – das Objekt wechselt während der Verarbeitung seinen ContentTyp von XML zu PDF. Wenn wir weitere Renderer zum Beispiel für Postscript einsetzen wollen, müssen hier die resultierenden ContentTypen behandelt werden. Die neuen XMLTemplates haben fast das gleiche Verhalten wie die bisherigen Templates, der Typ ist aber wichtig für die weitere Verarbeitung.

Analog dazu wird der ContentTyp in der Klasse CMXmlPublication.m geändert. Damit diese neuen Objekttypen verwendet werden können, müssen im gesamten System noch Änderungen durchgeführt werden. Diese werden nun kurz beschrieben:

Änderungen am Kernel in Objective C:

CMClasses.m: Hier werden die möglichen Objekttypen abgefragt und dementspre-

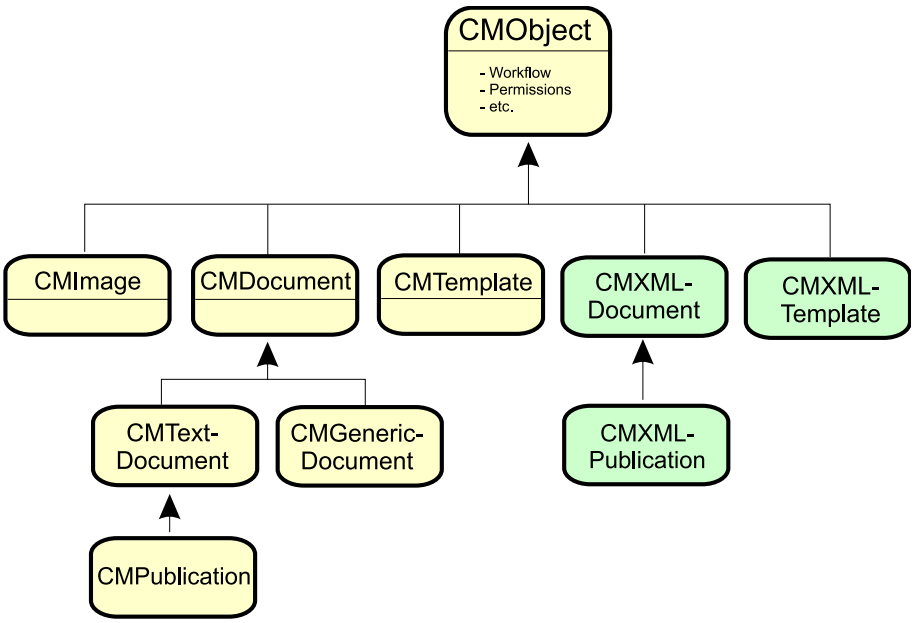

Abbildung 5.2: Die Vererbungshierarchie der einzelnen Objekttypen im NPS

chend müssen die neuen Typen hinzugefügt werden.

CMBase.h: Import-Statement für `CMXmlDocument und CMXmlPublication` hinzugefügt, damit dem CM diese Typen bekannt gemacht werden.

CMContent.m: Einige Änderungen in verschiedenen Methoden, die aber alle keine spezielle Behandlung der neuen Typen erfordern.

CMObject.m: Die Makros `DEFINE_CLASS_CODE` und `DEFINE_TCL_CODE` wurden für die neuen Typen hinzugefügt. Der `CLASS_CODE` setzt für jeden Objekttyp einen Character, der den Objekttyp in der Datenbank klassifiziert.

Änderungen am GUI in Java:

CMObjectAccessor.java: Die Accessoren stellen die Verbindung über das XML-Gateway. Dementsprechend müssen hier Keys für die neuen Typen angelegt werden.

CMObjClassMainEditPageServlet.java: Das Servlet zur Objektklasseneditierung. Da die Objektklassen von einem Objekttyp abgeleitet werden, müssen hier die neuen Typen angeboten werden.

```
1  #import "Base/CMBase.h"
2  #pragma .h #import "Base/CMDocument.h"
3
4  @implementation CMXmlDocument:CMObject
5  {
6  }
7  #pragma .h #include <CMCore/Base/CMCXmlDocument.h.include>
8  #include <CMCore/Base/CMCXmlDocument.m.include>
9
10 + (NSArray *)supportedContentTypes
11 {
12     id mimeTypeConfiguration = [[APP systemConfig]
                objectForKey: @"mimeTypes"];
13     id enumerator = [mimeTypeConfiguration keyEnumerator];
14     id contentTypes = [NSMutableArray array];
15     id contentType;
16
17     while (nil != (contentType = [enumerator nextObject])) {
18         id mimeType = [mimeTypeConfiguration
                objectForKey: contentType];
19
20         if ([mimeType hasSuffix: @"/html"]) {
21             [contentTypes addObject:contentType];
22         }
23         if ([mimeType hasSuffix: @"/pdf"]) {
24             [contentTypes addObject:contentType];
25         }
26     }
27     return contentTypes;
28 }
29 @end
```

Abbildung 5.3: Die Klasse CMXmlDocument.m

CMObjectViewPageServlet.java: "Ansicht" der Objekte. Damit der Content editiert werden kann, muss auch für die neuen Typen der Editor zu Verfügung stehen.

Sonstige Änderungen:

Icons: Um die neuen Typen in der Benutzeroberfläche behandeln zu könne, werden *.gif-Dateien für jeden Typ angelegt.

Startscript: Das Startscript zur ersten Initialisierung des Content Management Servers wurde um die neuen Typen erweitert, sodass von jedem Typ eine Standard-Objektklasse initial zur Verfügung steht.

Die zusätzliche Bool'sche Funktion `wantsNpsxmlProcessedInExportBlobFromContent`, die über die Datei `CMCXmlDocument.m.include` bzw. `CMXmlPublication.m.include` eingebunden wird, gibt an, dass für die neuen Dokumententypen die Verarbeitung nach Passieren des NPSOBJ-Prozessors noch nicht abgeschlossen ist, sondern danach noch der NPSXML-Prozessor durchlaufen wird.

Die Beziehung zwischen Objekten und Contents schlägt sich in den Klassen CMObject.m und CMContent.m nieder. Die Template-Verarbeitung für Preview und Export wird im Content selber angestoßen. Im CM wird viel mit Dictionary-Strukturen gearbeitet und so wird auch die Verarbeitung gestartet, wenn der Blob des gesuchten Objekts in der Methode `objectForKey: key` angefordert wird. Hier wird nun für herkömmliche Dokumente und Publikationen die Methode `exportBlobWithTemplate`, für die XML-Dokumententypen die Methode `exportXMLBlobWithTemplate` aufgerufen und so die oben beschriebene sequentielle Verarbeitung ausgeführt.

```
1  - objectForKey: key withViewDict: (NSDictionary*)viewDict
2  {
3      // Falls nach dem blob des Contents gesucht wird
4
5    if (keyCString && (strcmp(keyCString, "blob") == 0 &&
6            (!isBinary || isBinaryDataAllowed))) {
7        ....
8        ....
9        // Falls der Objekttyp "document" oder "publication" ist
10
11       } if (keyCString && (strcmp(keyCString, "exportBlob") == 0 &&
12               (!isBinary || isBinaryDataAllowed)) &&
13               (strcmp(objTypeCString, "publication") == 0 ||
14                strcmp(objTypeCString, "document") == 0)) {
15           return [self exportBlobWithTemplate: nil allowsEditedTemplate:
```

```
16                  [self isEditedContent] withViewDict: viewDict];
17
18      // Falls der Objekttyp "xmldocument" oder "xmlpublication" ist
19
20      } else if (keyCString && (strcmp(keyCString, "exportBlob") == 0 &&
21              (!isBinary || isBinaryDataAllowed)) &&
22              (strcmp(objTypeCString, "xmldocument") == 0  ||
23              strcmp(objTypeCString, "xmlpublication") == 0)) {
24          id result;
25          result = [self exportXMLBlobWithTemplate: nil allowsEditedTemplate:
26                  [self isEditedContent] withViewDict: viewDict];
27      ....
28      ....
29 }
```

Im folgenden ist die Methode `exportXMLBlobWithTemplate` abgebildet. Hier werden wie zuvor die einzelnen Verarbeitungseinheiten als Streams verbunden, dann wird die herkömmliche Verarbeitung angestoßen und das Ergebnis dem NPSXMLProzessor übergeben, der das Ergebnis an die aktuelle Instanz von CMContent.m zurückgibt.

```
1  - exportXMLBlobWithTemplate: (NSString *)templateName
2  {
3      id obj = [self object];
4      id result = nil;
5      id tempResult = nil;
6
7      if ([obj wantsUriProcessedInBlobFromContent]) {
8          id pool = [[NSAutoreleasePool alloc] init];
9          id uriExp = nil;
10
11         // Settings for processing the internal URIs
12         ...
13         ...
14         // Then the target is set to the resulting byte stream
15
16         uriExp = [CMViewHandler uriExporterForViewDict: viewDict
17                 andContent: self];
18         [uriExp setTarget: [HSByteStream stream]];
19
20         if ([obj wantsNpsxmlProcessedInExportBlobFromContent]) {
21
22          // Create a processor each for NPSOBJ and XSLT
23             id cmExp = [CMNpsobjProcessor stream];
24             id xmlExp = [CMNpsxmlProcessor stream];
```

```
25              id templ; // Template for NPSOBJ
26              id xmltempl; // Template for XSLT
27              id objProcessedData;
28
29              if (!IsNil(templateName)) {
30                 [cmExp setTemplateName: templateName];
31              }
32              [cmExp setExportContent: self];
33
34              templ = [obj parsedTemplateArrayWithName: [cmExp templateName]
35                      allowsEditedTemplate: allowsEditedTemplate];
36
37              // Settings for the NPSOBJ-Processor
38              ...
39              // Target on URI-Exporter
40              [cmExp setTarget: uriExp];
41              [cmExp writeObject: templ];
42              if ([ERRORHANDLER errorState] != APP_OK) {
43                  [ERRORHANDLER clearErrorStack];
44              }
45              // Ask the object for the needed XSL-Template
46              xmltempl = [obj xmltemplateWithName: [xmlExp templateName]
47                      allowsEditedTemplate: allowsEditedTemplate];
48
49              // Pass the template's contents to the XSLT-Processor
50              [xmlExp setTemplateContents: [[xmltempl objectForKey: @"blob"]
51                        stringValue]];
52
53              // Get the NPSOBJ-processed data
54              objProcessedData = [[[[uriExp target] target] stringValue] retain];
55             // Let the XSLT-Processor do the processing
56             tempResult = [xmlExp processXSLTemplate:objProcessedData];
57          } else {
58              [uriExp writeObject: [[self _blobInAttrDict] flatParsedHtmlArray]];
59              tempResult = [[[[uriExp target] target] stringValue] retain];
60          }
61          result = tempResult;
62          [pool release];
63          [result autorelease];
64      } else {
65          result = [self _blobInAttrDict];
66     }
67      return result;
68 }
```

Unter der Vorraussetzung, dass alle Bedingungen erfüllt sind, wird in Zeile 54 das NPSOBJ-Processing abgeschlossen. Nun haben wir ein durch das Template generiertes XML-Dokument vorliegen. Dieses kann beliebige Informationen aus dem Content Management Server enthalten und auch die genaue Form des XML-Dokuments ist unerheblich, jedoch sollte jede Informationseinheit, jede Entität des CM, in einzelnen XML-Tags eingeschlossen sein, damit kein Informationsverlust auftritt.

Bei der bisherigen Template-Verarbeitung wird bei einem ankommenden Dokument zunächst überprüft, ob dem Dokument ein spezifisches Template zugewiesen ist. Ist dies nicht der Fall, so wird der Dokumentenbaum nach oben hin durchlaufen und es wird das erste auftretende Template mit dem Namen "mastertemplate" genommen. Analog dazu wird nun mit XML-Dokumenten verfahren, bei denen das erste auftretende XMLTemplate mit dem Namen "xmlmastertemplate" genommen wird, was in Zeile 46 des oben abgebildeten Codes von `CMContent.m` passiert.

5.2.4 Der NPSXMLProzessor

Der XSLT-Prozessor für die Dokumente ist in CMNpsxmlProcessor.m definiert. Nach dem die oben erwähnte Xalan-Distribution in der Produktionsumgebung integriert worden ist, können wir die Komponenten jetzt benutzen. Wie bereits erwähnt, muss aus Objective C das C-API von Xalan angesprochen werden. Dieses bietet im Gegensatz zum C++ API eine reduzierte Schnittstelle, weshalb auch die Transformation mit Hilfe von temporären Dateien durchgeführt werden müssen. Die Erzeugung von temporären Dateien kann wiederum mit Methoden der NPS-Libraries durchgeführt werden.

Im folgenden ist die Methode `processXSLTemplate` des XMLProzessors abgebildet, in der die Templates verarbeitet werden.

```
#define XalanHandle xalan;

1  - (NSString *)processXSLTemplate:(NSString *)xmldata
2  {
3
4      BOOL isOk = YES;
5
6      char *xmlfilename ;
7      char *xslfilename;
8      char *outfilename;
9
10     NSString *xmlfilename1;
11     NSString *xslfilename1;
```

```
12     NSString *outfilename1;
13
14     NSData *xsldata;
15     NSData *outdata;
16
17     xmlfilename1 = [OSFilePath uniqueTmpFilename];
18
19    isOk = [xmldata writeToFile:xmlfilename1 atomically:YES];
20     xmlfilename = malloc([xmlfilename1 cStringLength]+1);
21    strcpy(xmlfilename, [xmlfilename1 cString]);
22
23    xsldata = [[self templateContents] asData];
24    xslfilename1 = [OSFilePath uniqueTmpFilename];
25
26     isOk = [xsldata writeToFile:xslfilename1 atomically:YES];
27
28     xslfilename = malloc([xslfilename1 cStringLength]+1);
29     strcpy(xslfilename, [xslfilename1 cString]);
30
31    XalanInitialize();
32    xalan = CreateXalanTransformer();
33
34    outfilename1 = [OSFilePath uniqueTmpFilename];
35    outfilename = malloc([outfilename1 cStringLength]+1);
36    strcpy(outfilename, [outfilename1 cString]);
37
38    theResult = XalanTransformToFile(xmlfilename, xslfilename,
39                                     outfilename, xalan);
40    XalanTerminate();
41    DeleteXalanTransformer(xalan);
42
43    outdata = [NSData dataWithContentsOfFile:outfilename1];
44
45    free (xmlfilename);
46    free (xslfilename);
47    free (outfilename);
48
49    return [outdata stringValue];
50 }
```

Nach dieser Abarbeitung können wir XML-basierte Formate in jeder Ausprägung erzeugen und den vollen Funktionsumfang der XSLT-Spezifikation nutzen. An dieser Stelle könnte statt des Xalan APIs jeder beliebige XSLT-Prozessor eingebunden werden. Wenn kein Binärformat erzeugt werden soll, wären wir an dieser Stelle

fertig - es kann jedoch jetzt noch eine weitere Bearbeitung mit XSL-Formatting Objects erfolgen und so ein großes Spektrum an zum Beispiel Printformaten abgedeckt werden.

5.2.5 Die Rendering Engine für das Formatting Objects Protocol

Das Formatting Objects Protokoll (FOP) spezifiert auf der Basis eines speziellen XML-Namensraumes bestimmte Blöcke, die jeweils einen Teil eines Dokumenten-Layouts definieren, wie zum Beispiel Ränder, Absätze, Überschriften, etc.. So ist hier ein Textblock mit Textgröße, Schriftart und Zeilenhöhe definiert.

```
<fo:block font-size="14pt" font-family="serif" line-height="16pt">
        This is a sample text
</fo:block>
```

Ein Dokument, das der FOP-Spezifikation folgt, muss somit nach der Umwandlung mit XSLT noch an eine Rendering-Engine weitergegeben werden. Diese Entscheidung erfolgt im Prototyp anhand des Content-Typs, der auch als MimeType an die externe Applikation weitergegeben wird, die das fertige Dokument erhält, wie zum Beispiel ein Web-Browser. Somit werden Dokumente, die als Content-Typ HTML oder XML haben, nicht an den Renderer geschickt, Dokumente, die jedoch den Content-Typ PDF haben, werden durch den Renderer in ein Binärformat umgewandelt.

Die FOP-Implementierung der Apache SoftwareFoundation unterstützt zur Zeit folgende Ausgabeformate:

- Standard-Textformat
- Portable Document Format (PDF) von Adobe
- Printer Control Language (PCL) für Laserdrucker von Hewlett-Packard und dazu kompatible
- MapInfo Data Interchange Format (MIF), ein Vektororientiertes Datenaustauschformat
- PostScript Language Format (PS)
- Klassen des Java Abstract Windowing Toolkits (AWT), komponenten zur Erzeugung von grafischen Benutzerschnittstellen.

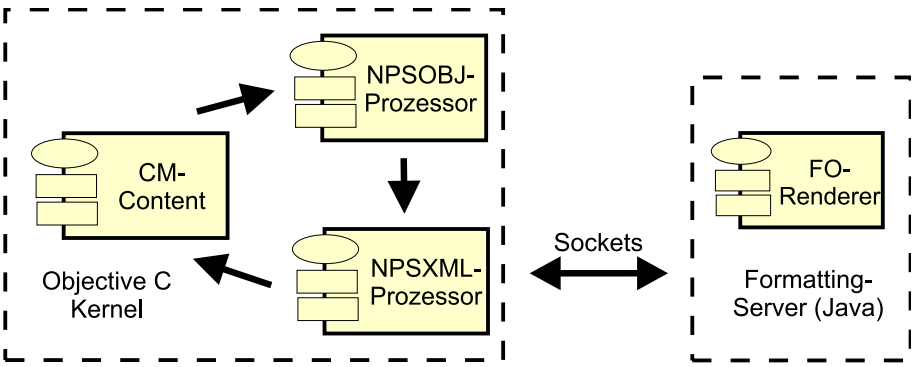

Abbildung 5.4: Der Kernel und der Java-Server zu Formatierung

- Direktausgabe auf einen Drucker
- XML-Format

Demnach können wir mit einem einzigen Stylesheet, das die Umwandlung nach XSL-FO vornimmt, alle oben genannten Formate erzeugen. Für die Formate Text und XML ist unser Renderer allerdings nicht notwendig, da wir diese Formate schon durch eine einfache XSLT-Transformation erzeugen können [2].

Wie bereits erwähnt gibt es für das Formatting Objects Protocol bislang nur eine freie Implementierung der Apache Software Foundation. Dies ist eine Java-Implementierung, die aus dem Objective-C Kernel angesprochen werden muss. Eine Möglichkeit wäre sicherlich, die Java-Klassen als eigenen Prozess direkt aus dem Kernel zu starten. Da aber das Starten der Virtuellen Maschine (JVM) bereits ca. 10 Sekunden dauert, ist dieser Overhead nicht hinnehmbar. Stattdessen wird ein kleiner Server aufgesetzt, der für die Formatierung zuständig ist. Der Kernel kann die Dokumente in xsl-fo Syntax über einen Socket an diesen weitergeben, anschließend werden die Dokumente formatiert und das erzeugte Binärformat wird zurückgeschickt. Der Aufbau ist in Abbildung 5.4 dargestellt.

Diesem Server übergeben wir nun das zu formatierende Dokument und zusätzlich einen String, der den gewünschten Renderer, also das gewünschte Ausgabeformat, angibt. Dieser String wird im Kernel anhand des eingestellten Content-Typen ermittelt. Das so durchgeführte Rendering benötigt somit für ein 7-seitiges PDF-Dokument Werte zwischen 150 und 300 Millisekunden, was für eine PDF-Erzeugung in Echtzeit durchaus akzeptabel erscheint.

[2] In diesem Kontext ist auch das Textformat ein XML-Format - es enthält lediglich einen Knoten

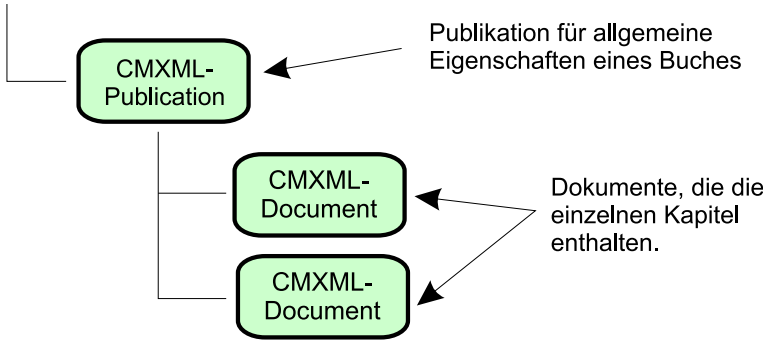

Abbildung 5.5: Eine mögliche Hierarchie zur Modellierung eines Buches

5.3 Anwendung des Prototypen auf ein Beispieldokument

Um die Abläufe im Prototypen zu testen, soll ein Buch aus mehreren NPS-Objekten generiert werden. Als Referenztext werden Ausschnitte aus dem Buch "The Lost World" von Arthur Conan Doyle genommen.

Es gibt verschiedenen Möglichkeiten, um die Anforderungen an ein Buch-Format abzudecken. In diesem Fall soll eine Publikation den Rahmen bilden und die generellen Eigenschaften eines Buches enthalten, die einzelnen Kapitel werden als Unterobjekte dieser Publikation modelliert. Für die Publikation definieren wir eine Dokumentenklasse, die vom Typ XMLPublikation ist und über Attribute verfügt, die in der Regel Eigenschaften eines Buches klassifizieren, wie zum Beispiel "Autor", "Verleger", "Titel" oder "Erscheinungsjahr". Die benötigten Attribute für ein Kapitel sind "Titel" und "Content", die bereits als Standardattribute in der Standardklasse "XMLDokument" enthalten sind und somit muss für die Kapitel keine spezielle Klasse definiert werden. Somit ergibt sich eine Hierarchie wie in Abbildung 5.5.

5.3.1 Schritt 1: Generierung des XML-Quelldokuments

Der erste Schritt in der Rendering-Pipeline aus Abbildung 5.1 ist die Abarbeitung des Mastertemplates durch den NPSOBJ-Prozessor. Hierbei handelt es sich um die klassischen Content Management Befehle, jegliche Formatierung wird später durch den XSLT-Prozessor durchgefügrt. Im folgenden ist der Inhalt des Mastertemplates dargestellt, dieses fügt die Attribute der Publikation direkt ein und schachtelt die Attribute der einzelnen Kapitel innerhalb von `<chapter>` Elementen.

```
<?xml version="1.0"?>
<book>
  <title>
    <npsobj name="title" insertvalue="var" />
  </title>
  <author>
    <npsobj name="author" insertvalue="var" />
  </author>
  <year>
    <npsobj name="year" insertvalue="var" />
  </year>

  <npsobj list="children">
    <npsobj name="objType" value="xmldocument" condition="isEqual">
       <chapter>
         <title>
           <npsobj name="title" insertvalue="var" />
         </title>
         <name>
           <npsobj name="name" insertvalue="var" />
         </name>
         <text>
           <npsobj name="body" insertvalue="var" />
         </text>
       </chapter>
    </npsobj>
  </npsobj>
</book>
```

Als Ausgabe des NPSOBJ-Prozessors erhalten wir die gewünschten Informationen im XML-Format.

```
<?xml version="1.0"?>
<book>
  <title>
       The Lost World
  </title>
  <author>
       Arthur Conan Doyle
  </author>
  <year>
       2001
  </year>
  <chapter>
```

```
     <title>
          There Are Heroisms All Round Us
      </title>
      <text>
         <p>Mr. Hungerton, her father, really was the most
              tactless person upon earth...</p>
         <!-- more paragraphs -->
      </text>
   </chapter>
   <!-- more chapters -->
</book>
```

Im bisherigen NPS-System hätten wir hier eine HTML-Ausgabe erzeugt und das Processing wäre abgeschlossen gewesen. In unserem neuen System kann das Dokument jetzt weitergereicht werden.

5.3.2 Schritt 2: Generierung des XML-Ausgabeformates

Auf das XML-Dokument können jetzt beliebige XSLT-Ausdrücke angewandt werden, die auch "Templates" genannt werden. Ein Stylesheet besteht aus vielen verschiedenen Templates, die jeweils auf ein bestimmtes Konstrukt des Quelldokuments passen. Somit steigen wir mit Hilfe des Templates den XML-Baum hinab und wenden auf jedes Element die gewünschte Transformation an. Da gewünschte Informationsmenge oft unabhängig vom Ausgabeformat ist (z.B. will man in einem PDF-Dokument die gleichen Inhalte haben wie in einem HTML-Dokument), sind die Regeln zum auffinden der Informationseinheiten dieselben, die Stylesheets unterscheiden sich im Wesentlichen durch die neuen Elemente, die um diese Informationen gruppiert werden. So könnte zum Beispiel für das `<H1>` Element einer HTML-Ausgabe ein `<block fontsize=18>` - Element für eine PDF-Ausgabe eingesetzt werden.

Im folgenden Ausschnitt eines Stylesheets ist zu erkennen, dass der Buchtitel als HTML-Titel gesetzt wird. Die einzelnen Kapitel werden hier in einer Tabelle platziert und die Titel der Kapitel werden ein der Größe `<H2>` dargestellt.

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

<xsl:template match="book">
  <html>
    <head>
       <title>
```

```
          <xsl:value-of select="title" />
        </title>
    </head>
    <body bgcolor="FFFFFF" text="#000000">
       <table width="70%" border="0">
        <!-- insertion of author, etc. -->
        <xsl:for-each select="chapter">
          <tr>
            <td>
              <xsl:apply-templates />
             </td>
          </tr>
        </xsl:for-each>
        </table>
      </body>
   </html>
</xsl:template>

<xsl:template match="book/chapter/text/title">
    <h2><xsl:value-of select="." /></h2>
</xsl:template>

</xsl:stylesheet>
```

Das Dokument kann jetzt entweder ausgegeben werden oder bei einer Formatierung mit Formatting Objects an den XSL-FO-Prozessor weitergereicht werden. Bei einer HTML-Ausgabe hätten wir jetzt ein sauber formatiertes Dokument gemäß der HTML 4.0-Spezifikation, im anderen Falle des Content-Typs “PDF” ein Dokument, in der initial die Formatierungen der einzelnen Arten von Textblöcken festgelegt werden und anschließend die Textblöcke selber auftreten.

```
<?xml version="1.0" encoding="UTF-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
     <fo:simple-page-master margin-right="2.5cm" margin-left="2.5cm"
        margin-bottom="2cm" margin-top="1cm" page-width="21cm"
        page-height="29.7cm" master-name="first">
       <fo:region-body margin-top="3cm"/>
       <fo:region-before extent="3cm"/>
       <fo:region-after extent="1.5cm"/>
      </fo:simple-page-master>

  <fo:page-sequence master-name="basicPSM">
```

```
     <fo:flow flow-name="xsl-region-body">
        <fo:block line-height="22pt" font-family="serif" font-size="20pt">
            The Lost World
           <fo:block line-height="12pt" font-family="serif"
                   font-size="12pt" id="sec1">
               by Arthur Conan Doyle
           </fo:block>
        </fo:block>
<fo:block line-height="20pt" font-family="serif" font-size="16pt">
   There Are Heroisms All Round Us
</fo:block>
    Chapter_1
<fo:block font-family="serif">
    Mr. Hungerton, her father ...
    <!-- the chapters follow here -->
        </fo:block>
      </fo:flow>
   </fo:page-sequence>
</fo:root>
```

5.3.3 Schritt 3: Generierung eines Binären Ausgabeformates

Das oben erzeugte Dokument wird nun an den Formatter weitergereicht, dieser verarbeitet die XSL-FO Syntax und gibt ein binäres Format zurück, in unserem Beispiel PDF. Dieser Vorgang wurde durch den Content-Typen des Ausgangsobjektes angestoßen, daher kann auch diese Content-Typ-Angabe an den Browser zurückgeschickt werden und dieser öffnet dann das Dokument mit Hilfe eines Plugins oder bietet einen Dialog zum Speichern an. Das oben beschriebene Beispieldokument ergibt so das in Abbildung 5.6 dargestellte PDF-Dokument.

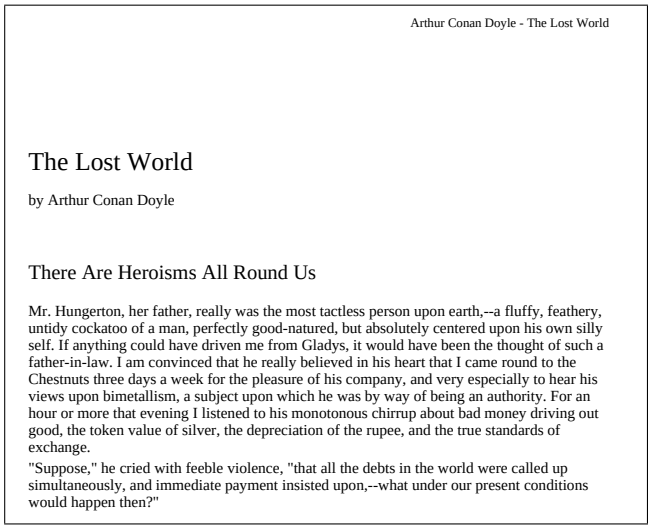

Abbildung 5.6: Ein PDF-Dokument, das mit XSL-Formatting Objects gerendert wurde

6 Anwendung der neuen Template-Logik auf Kundenszenarien

Wenn man sich noch einmal die in Abschnitt 5.1 formulierten Ziele betrachtet, so wurden die Punkte zwei und drei im letzten Kapitel erreicht: Wir können sowohl durch den Einsatz von XSLT als Ausgabeformat jede beliebige XML-Teilmenge produzieren als auch andere binäre Ausgabeformate durch XSL-FO erzeugen. Auch werden alle bisher in den Beispiel verwendeten Templates als NPS-Objekte verwaltet und stehen somit unter der Kontrolle des Content Management Systems. Dieses bringt einen entscheidenen Fortschritt in der kontrollierten Verwaltung der Daten.

Es muss allerdings noch überprüft werden, ob die beschriebenen SystemExecute-Prozeduren tatsächlich nicht mehr benötigt werden bzw. ob wir durch die neue Template-Engine die volle Funktionalität innerhalb des Systems nutzen können. Hierzu sollen Kundenszenarios eingesetzt werden, die bisher nicht alleine mit NPSOBJ-Logik zu bewältigen waren und mit externen Tcl-Prozeduren realisiert werden mussten.

6.1 Szenario 1: Das “Schachbrett”-Template

Ein relative einfaches Kundenszenario, in dem die NPSOBJ-Syntax nicht mehr ausreicht, war eine Anfrage bezüglich einer Webseite in einer “Schachbrett” -Darstellung, d.h. bei einer Tabelle sind die Felder alternierend entweder schwarz oder weiss und die Inhalte dementsprechend andersfarbig. Um dies zu erreichen, benötigt man ein Kriterium, um festzustellen, ob das aktuelle Feld zu den Feldern 1,3,5,... oder zu den Feldern 2,4,6,... gehört. Da die NPSOBJ-Logik nur einfache Ersetzungen zuläßt, ist die hier benötigte konditionale Verarbeitung nicht mehr ohne die externen SystemExecute-Prozeduren möglich.

Durch XSLT läßt sich dies aber auf verschiedenen Wegen innerhalb des Systems

realisieren, wie zum Beispiel durch die Benutzung von Variablen, die bei jeder Zelle inkrementiert werden und dann auf gerade oder ungerade Werte geprüft werden.

Für diese Beispiel könnten wir ein Ausgangsdokument nehmen, in dem die einzelnen Tabellenzeilen jeweils ein Produkt darstellen. Analog zu einer relationalen Datenbank haben die einzelnen Produktattribute ein spezifisches Tag, das den Spaltennamen angibt und die Produkte selber seien durch `<item>`-Tags abgetrennt.

```
<item>
   <itemname>
       Card Box Paper
   </itemname>
   <number>
       300231
   </number>
   <descr>
       Enables advanced wrapping and packaging
   </descr>
</item>
```

Um unser Kriterium festzulegen, muss die Position des aktuellen Elements modulo 2 festgestellt werden - anhängig davon kann dann die Tabellenzeile eingefärbt werden. In XSLT kann man durch `<xsl:variable>` Variablen definieren und diesen Werten zuweisen. Ausserdem kann mit `count=`*"Elementname"* festgestellt werden, zum wievielten Male das Element auftritt. Diese Zuweisungen nehmen wir in den Zeilen 3-5 des folgenden Listings vor. In den Zeilen 6-8 wird dann der Modulo-Wert einer Variablen zugewiesen. Dies ist ohne Probleme möglich, da in der XSLT-Spezifikation auch umfangreiche mathematische Funktionen spezifiziert sind [1]. Variablen in XSL sind innerhalb eines Templates lokal. Da wir den ermittelten Wert aber unmittelbar bei der Erzeugung unserer Tabellenzeile benötigen, müssen wir den Wert an das Template, das die Zelle erstellt, übergeben. Bei expliziten Template-Aufrufen mit `<xsl:call-template>` können diesen Parameter übergeben werden, was in Zeile 10-12 passiert.

```
1. <xsl:template match="item">
2.   <tr>
3.      <xsl:variable name="current">
4.         <xsl:number count="item"  />
5.     </xsl:variable>
6.     <xsl:variable name="color">
7.         <xsl:value-of select="$current mod 2" />
```

[1] Siehe dazu auch [DuC01]

```
8.      </xsl:variable>
9.      <xsl:call-template name="itemname">
10.         <xsl:with-param name="color">
11.             <xsl:value-of select="$color" />
12.         </xsl:with-param>
13.     </xsl:call-template>
14.     <!-- ... more template calls ... -->
15.   </tr>
16.</xsl:template>
```

Das so aufgerufenene Template für das Element `itemname` kann nun den Parameter auswerten und anschließend die Tabellenzelle und die Schriftart einfärben. Die im folgenden verwendeten Variablen `$schwarz` und `$weiss` sind globale Variablen, die im Stylesheet ausserhalb eines Templates definiert sind und auf die demnach ohne Variablenübergabe zugegriffen werden kann.

```
1. <xsl:template name="itemname">
2.      <xsl:param name="color"/>
3.      <td align="center">
4.          <xsl:if test="$color='1'">
5.              <xsl:attribute name="bgcolor">
6.              <xsl:value-of select="$schwarz" />
7.              </xsl:attribute>
8.          </xsl:if>
9.          <xsl:if test="$color='0'">
10.             <xsl:attribute name="bgcolor">
11.             <xsl:value-of select="$weiss" />
12.             </xsl:attribute>
13.         </xsl:if>
14. <font>
15.         <xsl:if test="$color='0'">
16.             <xsl:attribute name="color">
17.             <xsl:value-of select="$schwarz" />
18.             </xsl:attribute>
19.         </xsl:if>
20.         <xsl:if test="$color='1'">
21.             <xsl:attribute name="color">
22.             <xsl:value-of select="$weiss" />
23.             </xsl:attribute>
24.         </xsl:if>
25.     <h3><xsl:value-of select="itemname" /></h3></font>
26.     </td>
27.</xsl:template>
```

Name	Product Number	Description
Surf Wax	876532	Speed up grease
Underwater Helmet	897654	Protection device
Liquid Spirits	200542	Needed for cleaning and maintenance
Card Box Paper	300231	Enables advanced wrapping and packaging
Rubber Gloves	786122	To work with acids and other dangerous goods

Abbildung 6.1: Das generierte Schachbrettmuster als Beispiel für konditionale Verarbeitung

Nach Abarbeitung dieser Templates erhalten wir das oben beschriebene Schachbrettmuster, wie in Abbildung 6.1 zu sehen ist.

6.2 Szenario 2: Darstellung von Suchergebnissen

Als ein weiteres Beispiel für die eingeschränkten Möglichkeiten der NPSOBJ-Syntax kann die Darstellung von Suchergebnissen herangezogen werden. Ohne SystemExecute-Prozeduren ist es nicht möglich, Einfluss auf die Darstellung von Textinhalten zu nehmen, die vom Content Management Server geliefert werden – lediglich einzelne Attributwerte können unterschiedlich dargestellt werden, innerhalb der Attribute kann aber nicht navigiert werden. Dies ist mit Hilfe von XSL problemlos möglich. In vielen Umgebungen, in denen eine Volltextsuche stattfindet, ist es erwünscht, die Stellen, an denen der Suchausdruck auftaucht, gesondert hervorzuheben. Dies kann erreicht werden, indem man ein rekursives Template definiert, welches im folgenden abgebildet ist.

```
1. <xsl:template name="highlighter">
2.     <xsl:param name="content" />
3.     <xsl:choose>
4.         <xsl:when test="contains($content,$expr)">
5.             <xsl:value-of select="substring-before($content, $expr)" />
6.             <font color="#FF0000" size="+2"><B>
7.                 <xsl:value-of select="$expr" />
8.             </B></font>
9.             <xsl:call-template name="highlighter">
10.                <xsl:with-param name="content">
11.                   <xsl:value-of select="substring-after($content, $expr)" />
```

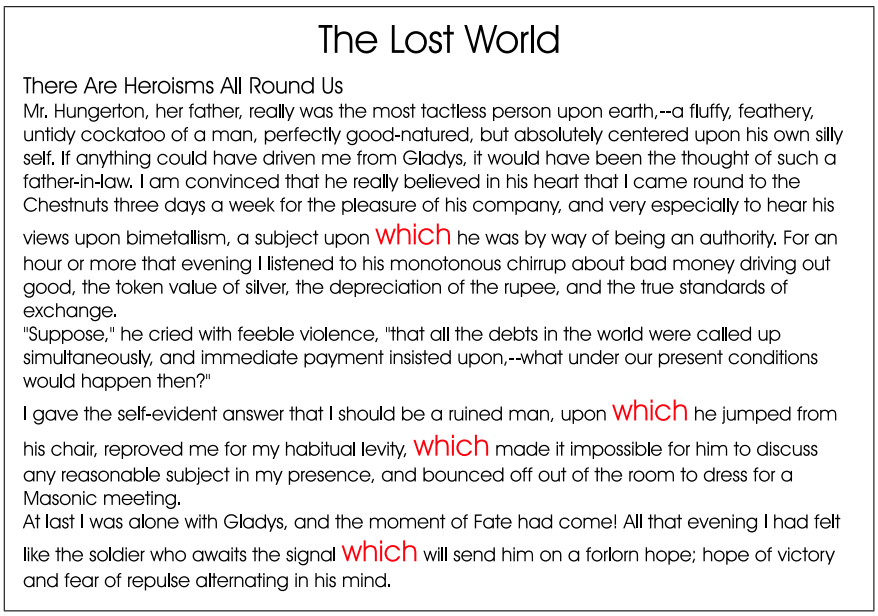

Abbildung 6.2: Hervorhebung von Ausdrücken durch rekursive Templateverarbeitung

```
12.                </xsl:with-param>
13.            </xsl:call-template>
14.        </xsl:when>
15.        <xsl:otherwise>
16.            <xsl:value-of select="$content" />
17.        </xsl:otherwise>
18.    </xsl:choose>
19.</xsl:template>
```

In diesem Template ist mit `$expr` der gewünschte Suchausdruck als globale Variable definiert. Wird nun im übergeordneten Template ein `<text>`-Element gefunden, so wird dieses Template aufgerufen. Durch `<xsl:choose>` wird eine Verarbeitung mit `when/otherwise` angestoßen. Wird in Zeile 5 nun der gewünschte Ausdruck gefunden, so wird zunächst der bisherige Text gedruckt, anschließend der neue Ausdruck mit gesonderter Formatierung und schließlich erfolgt ein erneuter rekrusiver Aufruf des Templates. In diesem Fall wird der Suchausdruck fett,in roter Schrift und größer dargestellt. Bei einem negativen Suchergebnis wird einfach der gesamte Text ausgegeben, was in Zeile 16 definiert ist.

6.3 Weitere XSLT-Funktionalität

Wie man schon an den obigen Beispielen sieht, stellt XSLT weit mehr als ein Werkzeug zur Formatierung dar, obwohl dies im Moment noch der Anwendungsschwerpunkt zu sein scheint. In der XSLT-Spezifikation, die ja auch schon von einigen Prozessoren nahezu vollständig implemenentiert wird, finden sich wesentlich mehr Sprachkonstrukte. Einige sollen hier genannt werden.

Umwandlung von XML-Bäumen: Jeder beliebige Baum kann in jeden anderen transformiert werden. Somit läßt sich jedes XML-basierte Ausgabeformat erzeigen, wie zum Beispiel HTML, XML, WML, etc.. Attribute können in Elemente umgewandelt werden und umgekehrt, jeder Baum kann mit Hilfe von XPath Ausdrücken traversiert werden, etc. und somit lassen sich Informationen in jeder gewünschten Form darstellen.

Variablen, Templates und Funktionen: Wie in gängigen Programmiersprachen können auch in XSL Variablen und Funktionen verwendet werden. Funktionen können frei definiert werden und Variablen zugewiesen werden. Funktionen und Templates können auch rekursiv aufgerufen werden.

Vererbung: Durch Inklusion von anderen Stylesheets kann Vererbung realisiert werden.

Konditionale Verarbeitung: Die oben gezeigte konditionale Verarbeitung ermöglicht beliebige Entscheidungen, die von Attribut-, Element- oder Textwerten abhängen können.

Auswertung von Ausdrücken: Verschiedenste Ausdrücke können durch XSLT berechnet werden. Hierzu gehören Vergleiche auf Zeichenketten, Elementen und Atributen, verschiedenste mathematische Ausdrücke, wie Addition, Subtraktion, Multiplikation oder Variablen.

Sortierung, Nummerierung und Aufzählungen: Durch XSL lassen sich beliebige Counter definieren und bei Bedarf abhängig von Ausdrücken anzeigen oder verändern. Elemente können anhängig von Kindern oder von Attributwerten sortiert werden, diese Sortierung kann zum Beispiel numerisch oder alphanumerisch erfolgen. Nummerierungen und Sortierungen können wie fast alle Konstrukte beliebig geschachtelt werden, wodurch sich zum Beispiel sehr komplexe Nummerierungen oder Sortierungen über mehrere Ebenen realisieren lassen.

Dies sind nur einige der Features von XSL – zusätzlich bieten sich sowohl innerhalb der XSLT-Spezifikation sowie durch Erweiterungen oder verwandte Standards nahezu unbegrenzte Möglickeiten im Bezug auf XML-basierte Daten.

6.4 Abschließende Bewertung

Nach der Arbeit mit XML und den dazugehörigen Standards dürften einige Vorteile von XML offensichtlich geworden sein.

Wie im Kapitel 3 beschrieben, kann XML einen entscheidenden Beitrag zur Trennung von Daten und Layout beitragen. Durch Implementierung und Einbindung von verschiedenen Tools zur Behandlung können Benutzerschnittstellen dynamisch in Abhängigkeit der zu behandelden Daten entworfen werden. In diesem Zusammenhang kann ein großer Teil der Programmlogik, die lediglich zur Überprüfung des Benutzerverhaltens dient, durch validieren von zugrundeliegenden Schemata ersetzt werden. Auf diese Weise lassen sich Benutzerschnittstellen sowohl generisch als auch stabil konstruieren.

Weiterhin bringt der Einsatz von XML und XSLT auf der Publishing-Seite einen enormen Funktionsumfang mit sich, der nicht nur durch die weitreichenden Möglichkeiten sondern auch wesentlich durch stabile Standards charakterisiert wird. Dies ist natürlich gerade im Umfeld von Content Management Systemen ein "Killer-Feature", was an den Beispielen im Zusammenhang mit dem Bau des Prototypen deutlich wird. Wenn man sich ein weiteres Mal die in Abschnitt 5.1 definierten Ziele ansieht, so sind diese voll erfüllt worden und es zeichnen sich weitergehende Vorteile ab:

- Durch den XSLT-Prozessor lassen sich die Inhalte unter völlig neuen Gesichtspunkten exportieren. Die fehleranfälligen SystemExcecute-Prozeduren lassen sich durch den enormen Funktionsumfang von XSLT komplett ersetzen, wodurch ein durchgehendes Content Management sowohl für die Inhalte als auch für die Templates erst möglich wird. Dies schliesst auch die durchgehende Versionierung und die Rechtevergabe ein, denn im Gegensatz zu SystemExecute-Prozeduren lagern die XSLTemplates nicht im Dateisystem, sondern in der Datenbank des CMS und somit kann individueller Zugriff für einzelne Benutzer und Gruppen auf die Ausgabeformatierung ermöglicht werden.

- Die Verarbeitung mit XSLT endet nicht bei HTML, sondern bietet Möglichkeiten zur Erzeugung jedes XML-basierten Ausgabeformates. Auch andere Textbasierte Formate können auf diese Weise erzeugt werden, da man bei einer Bearbeitung mit XSLT auch die Grundeigenschaft von XML, nämlich die Wohlgeformtheit, mit Hilfe von `disable-output-escaping` durchbrechen kann. Dies kann natürlich einen Informationsverlust zu Folge haben, aber in bestimmten Situationen notwendig sein.

- Durch die Einbindung von Formatting Objects ist die Funktionalität des Systems noch einmal entscheidend vergrößert worden. Auch wenn diese nur einen bestimmten Namespace definieren, so sind sie doch durch den Standard selbst

enorm mächtig, da von verschiedensten Seiten Renderer für Formatting Object entwickelt werden können. Bereits in dem hier vorgestellten Prototypen kann man mit der Implementierung von Apache bereits acht verschiedene Ausgabeformate, darunter auch eine Ausgabe als Java-Klassen des Java AWT, den Klassenbibliotheken für grafische Oberflächen, generieren – ein weiterer Schritt in Richtung generische Benutzeroberflächen. Gerade im Bereich Content Management können FOP entscheidende Verbesserungen im Bezug auf Features und Usability bringen.

- Der Einsatz von XML bringt wesentlich flexiblere Strukturen mit sich, wenn das System komplett auf dieser Basis realisiert wird, da man nicht mehr auf feste Tabellen und Strukturen angewiesen ist.

Auch im Bezug auf Schnittstellen zu anderen Systemen leistet XML bereits jetzt einen entscheidenden Beitrag, da allein schon durch die Wohlgeformtheit ein großes Fehlerpotenzial im Bezug auf unvollständige oder korrupte Datenübermittlung ausgeschlossen wird. Dies wird auch im NPS durch den XML-Gateway zwischen den Komponenten genutzt. Der Nutzen für die Kommunikation zwischen Prozessen kann aber noch erheblich gesteigert werden, wenn man XSLT nicht nur auf der Ausgabeseite einsetzt, sondern bidirektional in die Schnittstellen einbindet. So könnte ein Stylesheet zum Beispiel die Übersetzung vom NPS-XML zu einem SAP-XML durchführen, ein anderes Stylesheet übersetzt die Kommunikation zu Intershop Enfinity. Damit werden Schnittstellen wie in Abbildung 3.8 Realität.

Am 15.Oktober 2001 wurde schließlich auch noch die Version 1.0 der XSL-Spezifikation vom W3C als Empfehlung verabschiedet, was die Verbreitung der hier behandelten Techniken weiter stützen sollte, da dies ein Indiz für eine baldige Standardisierung ist. Mit dieser Arbeit sollte klar geworden sein, dass XML nicht nur ein Modewort ist, sondern durch gute Konzepte und Eigenschaften enorme Veränderungen im Bereich der Webanwendungen, insbesondere auch im Bereich Content Management, möglich macht.

Abbildungsverzeichnis

7 Literaturverzeichnis

[Adl00] Sharon Adler et al. Extensible stylesheet language (xsl). *World Wide Web Consortium*, 2000. URL:http://www.w3.org/TR/2000/CR-xsl-20001121.

[BHL99] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in xml. *World Wide Web Consortium*, 1999. URL:http://www.w3.org/TR/1999/REC-xml-names-19990114/.

[Bou00] Ronald Bourret. Namespace myths exploded. *Oreilly XML.com*, 2000. URL:http://www.xml.com/pub/a/2000/03/08/namespaces/index.html.

[Bou01] Ronald Bourret. Mapping dtds to databases. *Oreilly xml.com*, 2001. URL:http://www.xml.com/pub/a/2001/05/09/dtdtodbs.html.

[Bra00a] Neil Bradley. *The XSL Companion*. Addison-Wesley / Pearson Education, 2000.

[Bra00b] Tim Bray et al. The extensible markup language (xml) 1.0 (second edition). *World Wide Web Consortium*, 2000. URL:http://www.w3.org/TR/REC-xml.

[BSW01] Hans-Jörg Bullinger (Hrsg.), Erwin Schuster, and Stephan Wilhelm. Content management systeme - auswahlstrategien, architekturen und produkte. *Fraunhofer IAO, Verlagsgruppe Handelsblatt*, 2001.

[CD99] James Clark and Steve DeRose. Xml path language (xpath) version 1.0. *World Wide Web Consortium*, 1999. URL:http://www.w3.org/TR/xpath.

[Cla99] James Clark. Xsl transformations (xslt) version 1.0. *World Wide Web Consortium*, 1999. URL:http://www.w3.org/TR/xslt.html.

[Cla01] James Clark. Trex - tree regular expressions for xml. *Thai Open Source Software Center*, 2001. URL:http://thaiopensource.com/trex/spec.html.

[DMO01] Steve DeRose, Eve Maler, and David Orchard. Xml linking language (xlink) version 1.0. *World Wide Web Consortium*, 2001. URL:http://www.w3.org/TR/xlink/.

[Dou98] Dale Dougherty. Ice breaker - new xml-based protocol for content syndication. *Oreilly XML.com*, 1998. URL:http://www.xml.com/pub/a/98/10/ice.html.

[DuC01] Bob DuCharme. *Math and XSLT*. XML.com, 2001. URL:http://www.xml.com/pub/a/2001/05/07/xsltmath.html.

[Eis01] David J. Eisenberg. Using xsl formatting objects. *Oreilly xml.com*, 2001. URL:http://www.xml.com/pub/a/2001/01/17/xsl-fo/.

[Fal01] David C. Fallside (IBM). Xml-schema. *World Wide Web Consortium*, 2001. URL:http://www.w3.org/TR/schema-0/.

[FK99a] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing xml data in a relational database. *Institut National de Recherche en Informatique et en Automatique, France*, 1999.

[FK99b] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing xml data in a relational database. *Institut National de Recherche en Informatique et en Automatique, France*, 1999.

[Jon00] Kevin Jones. Xslbench - a xslt processor benchmark. 2000. URL:http://www.tfi-technology.com/xml/xslbench.html.

[KD01] Eugene Kuznetsov and Cyrus Dolph. Xslt processor benchmarks. *Oreilly XML.com*, 2001. URL:http://www.xml.com/pub/a/2001/03/28/xsltmark/index.html.

[Le 00] Arnaud Le Hors et al. Document object model (dom) level 2 core specification. *World Wide Web Consortium*, 2000. URL:http://www.w3.org/TR/DOM-Level-2-Core/.

[Mak00] Murata Makoto. Regular language description for xml (relax): Relax core. *JIS Technical Report*, 2000. URL:http://www.egroups.co.jp/files/reldeve/JISTRtranslation.pdf.

[Man99] Harley Manning et al. Managing web content. *Forrester Research, Inc.*, 1999.

[McL00] Brett McLaughlin. *Java and XML.* O'Reilly & Associates, Inc., 2000. URL:http://www.oreilly.com/catalog/javaxml/.

[Ope95] Openstep. *Object-Orientated Programming and the Objective-C Language.* NeXT Computer, Inc, 1995. URL:http://www.toodarkpark.org/computers/objc/objctoc.html.

[RC01] G. Rothfuss and C.Ried. *Content Management mit XML - Grundlagen und Anwendungen.* Springer-Verlag, 2001. URL:http://www.xml-content.de/.

[RSC99] Arnon Rosenthal, Len Seligman, and Roger Costello. Xml, databases and interoperability. *Federal Database Colloquium, AFCEA, San Diego*, 1999.

[SPE01] Standard Performance Evaluation Corporation SPEC. 2001. URL:http://www.spec.org.

[ST01] Airi Salminen and Frank Wm. Tompa. System desiderata for xml databases. *VLDB Submission Nr.284, Rome*, 2001.

[Vos94] Gottfried Vossen. *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme.* Addison-Wesley, 1994.

[ZVO00] ZVON.org. Xdb: Xml database. *ZVON.org*, 2000. URL:http://www.zvon.org/index.php?nav_id=61.

8 Danksagung