X P G

from XML2Any

Günther Brand

Edmund Haselwanter

Manfred Stocker

Stefan Thalauer

WS 2001/02

Abstract

Jeder, der schon einmal ein Dokument verfaßt hat, steht früher oder später vor der Aufgabe, dieses auch anderen Personen zugänglich zu machen. Meist ist allerdings der Inhalt wertvoller als das Aussehen des Dokumentes. Vor allem bei der Wiederverwendung des enthaltenen Wissens bzw. auch nur Teilen davon, steht man vor dem großen Problem der verschiedenen Dokumentformate, die von den unterschiedlichen Programmen, die zur Erstellung des Dokuments verwendet wurden, herrühren. Ein weiterer Problemkreis, der sich bald erschließt, ist eng mit dem Verteilungsmedium verbunden. Ein und der selbe Inhalt soll in gedruckter Form, in druckbarer Form aber elektronisch bzw. elektronisch aber nur Teile des Inhalts, zur Verfügung gestellt werden. In diesem Dokument wird ein in Java geschriebenes Framework beschrieben mit dessen Hilfe man Daten, die in einem XML-Format vorliegen, verarbeiten kann. Dabei ist es im Allgemeinen möglich beliebige Objekte aufzubauen, im Speziellen kann man die XML-Daten in jedes denkbare Ausgabeformat transformieren.

Keywords: Java, Dokumenterstellung, XML-Schema, XML, XSL, XSLT, Statemachine, endlicher Automat, Transformation, Dokumentverarbeitung

Inhaltsverzeichnis

Kapitel 1

Vorwort

Dieses Projekt wurde im Rahmen der XPG (eXtended Project Group) gestartet. Die eXtended Project Group ist eine Initiative des IICM (Institute for Information Processing and Computer supported new Media) an der TU-Graz, um Studierenden im Rahmen ihres Studiums das Arbeiten in großen Projekten und Teams zu ermöglichen. Eine genauere Beschreibung finden Sie unter [17].

1.1 Motivation

Die Motivation hinter XML2Any besteht darin, ein einheitliches Framework für den Umgang mit Dokumenten verschiedenster Art zu entwickeln. Die meisten der derzeitigen Produkte am Markt trennen die eigentliche Information, den *Inhalt* des Dokuments sehr unzureichend von der benötigten Information für deren Darstellung, dem *Layout* des Dokuments.

Dieser Problemkreis ist nicht neu. Vor allem in der Softwareentwicklung, die sich mit grafischen Benutzeroberflächen auseinandersetzt, wurde man schon recht bald vor die Aufgabe gestellt, Lokalisierungen, das heißt Anpassungen an die gewünschte Sprache vorzunehmen bzw. Eingabefelder, Tabellenfelder und andere Darstellungskomponenten mit Daten zu füllen. Aus diesen Erkenntnissen wurde das MVC (Model-View-Controller) Konzept abgeleitet [23]. Dieses Konzept hat auch bei der Verarbeitung bzw. Erstellung von Dokumenten seine Relevanz, da vermehrt die Daten entsprechend einem Modell (geeignet strukturierte Daten, zum Beispiel mit Hilfe von XML/XML-Schema, Datenbank, etc.) vorliegen, um mehrfach verwendet werden zu können.

Es ergibt sich ein natürliches Spannungsfeld zwischen dem Autor des Dokumentes und dem Betrachter, der vielleicht nur an einer bestimmten Sicht

auf den Inhalt des Dokumentes interessiert ist. Weiters kann der Autor bestimmte Anforderungen an das Layout (Aussehen) stellen, die in speziellen Fällen nicht eingehalten werden können (Schriftgröße vom Betrachter vorgegeben, Medium ohne Bilddarstellung, Übertragungsbandbreite, ...).

Dabei sind einerseits die Bedürfnisse des Autors in entsprechendem Ausmaß zu berücksichtigen, andererseits sollte dem Betrachter (Konsumenten) auch ein gewisser Einfluß auf die Präsentation des Inhaltes eingeräumt werden. Das schönste Layout nützt nichts, falls der Konsument nur Blindenschrift (siehe auch Braille [5]) *lesen* kann.

1.2 Kapitelübersicht

Kapitel 1 *Vorwort:* Dieses Kapitel beinhaltet das Zustandekommen dieses Projekts. Weiters wird die Motivation hinter XML2Any erklärt - aus welchen vorhandenen Problemstellungen heraus das Projekt abgeleitet wurde.

Kapitel 2 *Anforderungskatalog:* Zuerst muß die ganze Problematik genau erfasst werden - dazu wird ein umfangreicher Anforderungskatalog zusammengestellt. Weiters werden bereits vorhandene Lösungsansätze mit den enthaltenen Möglichkeiten betrachtet und die jeweiligen Probleme aufgezeigt.

Kapitel 3 *Betrachtete Technologien:* Die für dieses Anwendungsgebiet in Frage kommenden Technologien werden analysiert, um daraus dann jene auswählen zu können, mit deren Hilfe eine zielführende Lösung entwickelt werden kann.

Kapitel 4 *Das Framework:* Aus den gesammelten Anforderungen wurde ein Framework entwickelt, welches genau klassifizierte XML (eXtended Markup Language)-Dokumente in diverse andere Formate (z.B.: LaTeX , HTML (HyperText Markup Language), etc.) transformieren kann. An einem einfachen Beispiel wird die prinzipielle Funktionsweise erklärt. Danach wird das Design skizziert und die Implementierung der einzelnen Komponenten im Detail erläutert. Auch die im Framework verwendete Fehlerbehandlung wird eigens erklärt.

Kapitel 5 *Anwendungsbeispiele:* In diesem Kapitel werden einige Anwendungsbeispiele vorgestellt, die das entwickelte Framework verwenden. Unter anderem: Transformationen nach LaTeX und HTML, Entwurf eigener Dokumentklassen im Hinblick auf ein im Zielformat schon definiertes bzw .verwendbares Package (z.B. Dokumentstyle-Klasse bei LaTeX), Datenmanipulation bzw. Auswertung von Daten zum Erstellen und Einbinden von Graphiken. Dies deckt allerdings nur einen relativ kleinen Bereich der tatsächlichen Einsatzmöglichkeiten des entwickelten Frameworks ab (auf weitere wird

in Kapitel 6 hingewiesen).

Kapitel 6 *Zusammenfassung und Diskussion:* Es werden nochmals die mit diesem Framework erzielten Möglichkeiten und Lösungen zusammengefasst und Einsatzgebiete aufgezeigt. Weiters werden Ansätze gegeben, in welche Richtungen das Framework noch erweiterbar ist, und wo man ansetzten kann um es noch umfangreicher und universeller zu machen.

Kapitel 2

Anforderungskatalog

Einer der Hauptaspekte dieses Projektes ist es, eine Software zu entwickeln, die es ermöglicht - aus einer geeignet strukturierten Datenbasis - Dokumente in verschiedenen Zielformate zu transformieren. Dabei muß großer Wert auf ein modulares Konzept gelegt werden, um zu jedem beliebigen Zeitpunkt neue Zielformate hinzufügen zu können. Unter *Zielformate* sind unter anderem Dokumentformate wie HTML, WML (Wireless Markup Language) oder LaTeX zu verstehen.

2.1 User Requirements

Nachfolgend sind die wichtigsten Anforderungen, die sogenannten User Requirements, und teilweise die daraus resultierenden Designentscheidungen angeführt:

UR 1: Beschreibung der Daten in Mensch-lesbarer Form

Die Beschreibung der Daten soll in Mensch-lesbarer (human-readable) Form sein. Daraus folgt UR 2. Dadurch ist gewährleistet, dass das Quelldokument unabhängig vom System lesbar/bearbeitbar bleibt.

UR 2: Strukturierte Datenrepräsentation (XML)

Um eine Strukturierung der Daten zu erreichen, werden diese mit XML beschrieben. Aufgrund der Verwendung von XML ist die Plattformunabhängigkeit bezüglich der Strukturierung gegeben. (siehe auch UR 7)

UR 3: Validierbarkeit der Quelldaten

Die Definition des Quelldatenformats soll mittels XML-Schemata erfolgen. Das System unterstützt eine Validierung gegen diese Definition des Quelldatenformates.

UR 4: Trennung von Inhalt und Aussehen

Um verschiedene Ausgabeformate und Sichten auf die Daten aus den Quelldaten generieren zu können, ist es notwendig, den Inhalt und das Aussehen, respektive Layout strikt zu trennen.

UR 5: Transformation der Daten in wählbares Ausgabeformat

Aus UR 4 folgt, dass die die Quelldaten mit einer geeigneten Transformation in ein spezifisches - mit Layoutinformationen erweitertes - Ausgabeformat transformiert werden kann.

UR 5.1: Statische Transformation

Die Transformation der Quellen in die Ausgabeformate muß statisch (offline) erfolgen können.

UR 5.2: Dynamische Transformation

Die Architektur des Systems soll einfach auf eine dynamische Transformation erweiterbar sein.

UR 6: Unterschiedliche Ausgabeformate

Die Architektur des Systems soll verschiedene Ausgabeformate ermöglichen.

UR 7: Plattform Unabhängigkeit der Implementierung

Die Transformation soll plattformunabhängig erfolgen können. Um dies zu erreichen wird einerseits die Programmiersprache Java 1.3 und andererseits, wie in URD 2 beschrieben, XML verwendet.

UR 8: Mehrsprachigkeit

Es wird Mehrsprachigkeit des Kontents vom System unterstützt (z.B.: deutsch/englisch).

UR 9: Debug Möglichkeiten sollen vorhanden sein

Es gibt einen Debug Modus, der in geeigneter Weise Debug Informationen erzeugt. Dies dient unter anderem zur Analyse der Transformation.

UR 10: Transformation mitprotokolliert

Es werden die Transformationsschritte und -fehler protokolliert.

UR 11: Erweiterbarkeit

Es ist bei der Architektur des Systems auf folgende Erweiterungen Rücksicht zu nehmen:

UR 11.1: Modifikation der Definition des Quelldatenformates

Die Erweiterbarkeit im Bezug auf neue oder geänderte Elemente (Tags) und Transformationen muß gegeben sein.

UR 11.2: Neue Ausgabeformate

Die Erweiterbarkeit hinsichtlich neuer Ausgabeformate muß gegeben sein.

UR 12: Bandbreite der Deviceses berücksichtigen

Es muß dem Benutzer des Systems möglich sein, auf die Verschiedenen Gegebenheiten des Outputdevice Rücksicht zu nehmen. Am Beispiel eines Bildes: für HTML eine Version des Bildes mit niederer Auflösung/Dateigröße, für ein zum Ausdruck vorgesehenes Dokument eine hoch aufgelöste Version des Bildes.

Im Kapitel 5 werden dementsprechend implementierte Anwendungsbeispiele vorgestellt.

2.2 Ein Querschnitt über die Probleme und deren Lösungsansätze

Der Entwicklung des Frameworks ging eine umfangreiche Recherche voraus. Dabei wurde großes Augenmerk auf eine möglichst breite Palette von Anwendungsfällen gelegt. Diese floß auch in den Anforderungskatalog, als sogenannte URD (User Requirement Document) ein (siehe Abschnitt 2.1). Ziel dieser Vorarbeiten war einerseits die bereits vorhanden Lösungsansätze untereinander zu vergleichen und andererseits, diese Lösungsansätze in das URD einfließen zu lassen bzw. gegen das URD zu prüfen.

2.2.1 Das World Wide Web im Wandel

Vor allem das WorldWideWeb hat dazu beigetragen, dass sich im Bereich Dokumentmanagement einiges bewegt. Bei der Entwicklung immer umfangreicherer Webseiten entstand bald der Wunsch, die Funktionalität, die HTML bietet, zu erweitern. Vor allem stellte sich heraus, dass die Vermischung von Layout und Inhalt, wie es bei HTML passierte, viele Probleme aufwarf. Auch die Mehrsprachigkeit des Inhalts und die damit verbundene Vervielfachung der HTML-Dokumente erschwert die Wartung drastisch. Will oder muß man am Erscheinungsbild etwas ändern, ist meist jede einzelne HTML-Seite betroffen und somit zu bearbeiten.

Aus der in den frühen 80er Jahren entstandenen SGML (Standard Generalized Markup Language) [3] und unter Einbeziehung der Erfahrungen die mit HTML gemacht wurden, wurde XML entwickelt. XML stellt eine Möglichkeit dar, Daten strukturiert in einem mensch-lesbarem Textformat abzuspeichern. Damit ist ein Werkzeug für den Inhalt gegeben. Mit XSL, XSLT, XSL:FO (siehe Kapitel 3) werden Layoutinformationen kodiert. Dazu gibt es noch weitere Module wie zum Beispiel Xlink und XPointer die

gemeinsam eine Familie von Techniken im Umgang mit Dokumenten bieten. Mit Hilfe dieser Technologien ist möglich die Daten vom Layout zu trennen bzw. auch unterschiedliche Sichten auf die Daten zu erzeugen.

2.2.2 XT3D: a transformation system for XML

XT3D [26] stellt eine XML Sprache dar, mit der man Transformationen von einem Eingabe-XML-Dokument in ein Ausgabe-XML-Dokument spezifizieren kann. Dabei werden im Eingabe-XML-Beschreibungsdokument die Namen und Attribute als Muster für die Transformation markiert und im Ausgabe-XML-Beschreibungsdokument mit XT3D Elementen der Bildbereich der Transformation beschrieben. Für die eigentliche Transformation kommt dann ein in der funktionalen Sprache Scheme [25] implementierter Interpreter zum Einsatz.

Ein entscheidender Vorteil bei dieser Lösung ist, dass der Benutzer mit Hilfe von XML die Transformationen beschreibt, also nur eine "Auszeichnungssprache" erlernen muß. Ein gravierender Nachteil ist allerdings, dass mit dieser Transformation, die strenggenommen eine surjektive Abbildung darstellt, nur Veränderungen hinsichtlich der Struktur des Ausgangsdokumentes erzielt werden können. Es können also höchstens Teile des Ausgangsdokumentes weggelassen und/oder die Elementnamen geändert werden (vergl. XSLT, Abschnitt 3.5).

2.2.3 Displets: Managing Complex Documents over the World Wide Web

Bei manchen Dokumenten besteht eine große Schwierigkeit in der eigentlichen Darstellung (Rendering) der Information. Als triviale Lösung wird dann meist Text bzw. Text und Bild gemischt verwendet. In speziellen Fällen stößt man dabei an Grenzen, die auf den ersten Blick nicht offensichtlich sind. Vor allem bei wissenschaftlichen Dokumenten tritt der Fall auf, dass sie nicht oder nur sehr schwer textuell beschrieben werden können, damit eine elektronische Weiterverarbeitung möglich ist. Im Speziellen vertragen sich HTML und mathematische Formeln nicht, da es hierzu keine speziellen Tags in HTML gibt. Das W3C empfiehlt die Verwendung von MathML (Mathematical Markup Language) [6], einem XML-Dialekt, der aber noch nicht von allen Webbrowsern dargestellt werden kann. Für den Webbrowser Mozilla [9] wurde eine MathML Darstellungskomponente entwickelt.

Die Idee hinter Displets [14] ist es nun, dem WWW Browser eine zusätzliche Render Engine zur Verfügung zu stellen, die wie bei LaTeX aus der textuellen Beschreibung eine entsprechende Visualisierung generiert. Die Beschreibung

erfolgt dabei mit Hilfe eines entsprechendem XML Dialekts. Die Render Engine ist in Java implementiert und kann somit mit allen Java fähigen Browsern verwendet werden.

Bis auf die technische Machbarkeit bzw. den etwas anderen Lösungsweg stellt dieser Weg allerdings nicht gerade eine Vereinfachung in der Handhabung dar. Gegenüber einem LaTeX Dokument ist kein entscheidender Vorteil ersichtlich, da dieses auch einfach in ein PDF (Portable Document Format) [4] oder PostScript [1] Dokument übersetzt und weitergegeben werden kann.

2.2.4 FlexXML: Engineering a More Flexible and Adaptable Web

Oft besteht das Problem darin, dass dem Ausgabe/Anzeigegerät technische Grenzen gesetzt sind. So sind Übertragungsbandbreite, Bildschirmgröße und Bildschirmauflösung entscheidende Faktoren auf der Konsumentenseite. Abhängig vom Endgerät bzw. Bedürfnis des Konsumenten sollte Einflußnahme auf die Darstellung des Dokuments möglich sein. Im einfachsten Fall zum Beispiel durch die Auswahlmöglichkeit Text oder Text und Bild. Der damit verbundene Aufwand für den Autor sollte minimal sein. Wie eingangs erwähnt bietet es sich an, dem Konzept der Trennung von Layout und Inhalt hohe Aufmerksamkeit zu schenken.

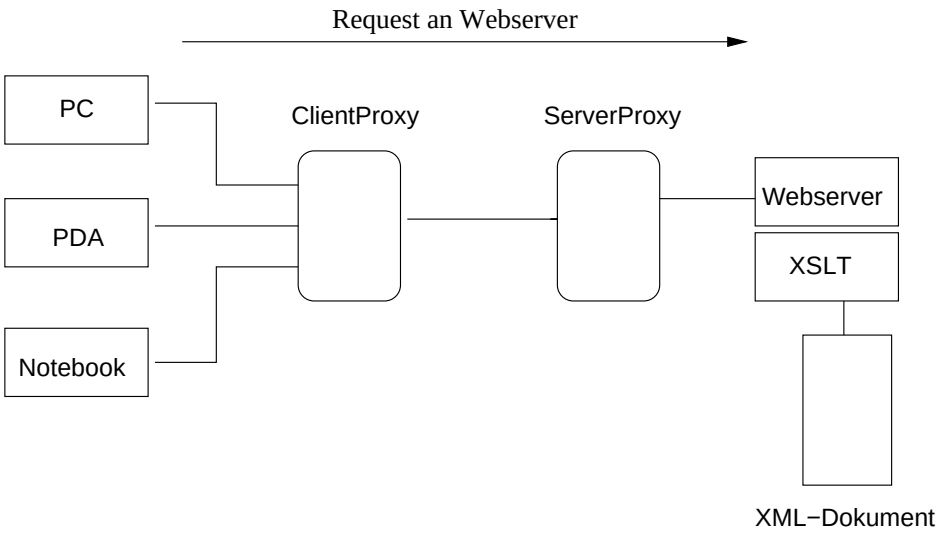

Abbildung 2.1: Das FlexXML Network

FlexXML [24] ist für das WWW entwickelt worden und verwendet für den Inhalt XML und für die Beschreibung des Layouts bzw. der Transformationsvorschrift XSL. Dabei werden für unterschiedliche Clients (Handy, PDA,

PC, Notebook) bzw. Netzanbindungen (drahtlos, Modem, LAN) entsprechende XSL Stylesheets verwendet. Damit erreicht man, dass abhängig vom Client das angeforderte Dokument im gewünschten Format ausgeliefert wird. Die Information, welches Stylesheet verwendet werden soll, wird bei FlexXML durch eine Protokollerweiterung des HyperTextTransferProtokolls (HTTP, RFC 1945 [11] bzw. RFC 2068 [22]) erreicht.

Der Client schickt hierzu seine Anfrage über einen Clientproxy, der den Request um die Usereinstellungen erweitert, an einen Server. Dieser Server muß nun mittels eines Serverproxys diese zusätzlichen Informationen aus der Anfrage herausholen und kann damit - und mit dem passendem XSL-Stylesheet - das Dokument entsprechend ausliefern.

Es stellt sich allerdings die Frage, ob die Protokollerweiterung den besten Weg darstellt, diese zusätzlich notwendigen Daten an den Server zu senden. Allerdings passiert der Zugriff auf die Daten dadurch für den Benutzer transparent. Mit ein und der selben Webadresse werden die angepaßten Dokumente an den Client ausgeliefert.

Kapitel 3

Betrachtete Technologien

In diesem Abschnitt werden Technologien vorgestellt, die im Hinblick auf Dokumentverarbeitung respektive Softwareentwicklung genauer untersucht wurden.

3.1 XML

Die *eXtensible Markup Language* (XML [13]) ist wie SGML eine Metasprache. Mit ihrer Hilfe lassen sich eigene, neue Sprachen zur Dokumentenbeschreibung erstellen. Beispielsweise lässt sich mit Hilfe von XML die Sprache HTML definieren. Eine Metasprache bietet Werkzeuge und eine normierte Syntax zur Beschreibung von Grammatiken.

Das Grundprinzip von Metasprachen basiert auf der Trennung von Inhalt, Struktur und Layout von Dokumenten, also der Trennung vom Informationsgehalt eines Dokumentes von seiner äußeren Form.

Dokumente bestehen aus dem Inhalt, also Text, Bilder usw., einer Formatierung, dem Layout, z.B. die gewählte Schriftart und anderen Informationen die die (visuelle) Darstellung bestimmen. Weiters einer strukturellen Unterteilung des Textes, also einer Aufteilung in Kapitel, Abschnitte usw. Sätze bestehen nicht einfach aus aneinandergereihten Wörtern, sondern auch Informationen über einzelne Wörter oder Satzteile. Handelt es sich um ein Zitat oder um eine wichtige Textstelle? Dies stellt die logische Information dar. Es muß also der Schwerpunkt nicht nur auf das Aussehen, sondern - für eine spätere Verarbeitung förderlich - auf die Struktur und die logischen Elemente gelegt werden.

Das *World Wide Web Konsortium* (W3C) hat im Februar 1998 die erste Empfehlung (recommendation) veröffentlicht.

In XML können eigene Markupbefehle und Attribute nach Bedarf definiert werden. Dokumentstrukturen können in ihrer Komplexität an die erforderlichen Informationen angepaßt werden. Jedes XML-Dokument kann weitere optionale Beschreibungen seiner Grammatik enthalten, mit deren Hilfe eine Applikation dann eine strukturelle Überprüfung durchführen kann.

Im Gegensatz zur sehr schwerfälligen und umfangreichen SGML-Sprache stellt XML eine Vereinfachung dar und ist so konstruiert, dass jedem Autor ermöglicht wird, eine auf die persönlichen Belange zugeschnittene Grammatik zu erstellen. Die SGML-Definition des W3C umfaßte 1986 über 500 Seiten. Die aktuelle XML-Definition kommt auf knappe 33 Seiten.

SGML wurde bereits 1986 als ISO-Norm 8879 ([3]) verabschiedet und bildet heute die Basis aller Auszeichnungssprachen. Mit Hilfe einer SGML-Deklaration gibt die SGML-Anwendung an, welche Zeichen als Daten und welche als Markup interpretiert werden sollen.

XML ist eine Teilmenge von SGML, also keine völlige Neuentwicklung oder gar Ablösung.

Voraussetzungen einer Sprachdefinition, die XML erfüllt:

- Flexible Datenstruktur: Die Sprache darf nicht auf einer festen Datenstruktur basieren, sondern muß ständig erweiterbar sein. Sie muß sich den wechselnden Gegebenheiten und Anforderungen, die der Benutzer insbesondere bei der Dateninterpretation an sie stellt, anpassen. Sie muß sich leicht in möglichst unterschiedliche Formate konvertieren lassen.
- Plattformunabhängigkeit: Die Daten, die mit dieser neuen Auszeichnungssprache definiert wurden, müssen flexibel zwischen den verschiedensten Plattformen austauschbar sein.
- Textorientiert: Die plattformübergreifende Struktur bedingt, dass man keine binäre Datenstruktur zur Speicherung verwendet. Ein Textformat ist hier deutlich im Vorteil. Die Kennzeichnung der einzelnen Daten muß dann natürlich auch durch Textmarken erzeugt werden. Meist beschränkt man sich auf die 128 Zeichen des US-ASCII-Zeichensatzes. Für landesspezifische Zeichen und Sonderzeichen muß man sich alternative Lösungen ausdenken, um dies mit Hilfe der vorhandenen Zeichen zu umschreiben.

Für spezielle Fachgebiete existieren bereits auf Basis von XML definierte Auszeichnungssprachen oder sind im Aufbau:

- SMIL[21] *Synchronized Multimedia Integration Language* Zur Integration und Synchronisation von Multimedia-Inhalten.

- CML *Chemical Markup Language* Zur Beschreibung komplexer chemischer Formeln.
- MathML [18] *Math Markup Language* Zur Beschreibung komplexer mathematischer Formeln.
- CDF [8] *Channel Definition Format* Von Microsoft eingeführtes Format das dazu dient, Anwender regelmäßig mit aktualisierten Inhalten von Websites zu versorgen
- WIDL [28] *Web Interface Definition Language* Neue objektorientierte Sprache zum Erstellen dynamischer Websites
- XQL *eXtensible Query Language* Zur Durchführung von Abfragen in Daten
- SVG [19] *Scalable Vector Graphics* Zur Darstellung von Vektorgrafiken
- X3D VRML-Standard für 3D

3.2 DTD

Eine so genannten *Document Type Definition* (DTD) beschreibt den strukturellen Aufbau von Dokumenten, die logischen Elemente einer Klasse von Dokumenten, genannt *Dokumenttyp* und es werden die Grundregeln für die Struktur eines Dokuments festgelegt. Die DTD definiert im Einzelnen einsetzbare Befehle und deren zusätzliche Optionen. Die Dokumenttyp-Definition kann innerhalb des Dokuments selbst oder in einer externen Datei untergebracht sein. Anhand der DTD ist eine spätere Überprüfung der Syntax durch den Parser möglich. Sofern ein validierender Parser eingesetzt wird, ist eine DTD unbedingt erforderlich.

Ein Dokument ist ein XML-Dokument, wenn es im Sinne dieser Spezifikation wohlgeformt ist, z.B. dürfen sich Tags nicht überlappen, jedes Tag muß beendet werden und die Groß- und Kleinschreibung ist zu berücksichtigen.

Ein wohlgeformtes XML-Dokument kann darüber hinaus gültig sein, sofern es bestimmten weiteren Einschränkungen genügt:

Jedes XML-Dokument hat sowohl eine logische als auch eine physikalische Struktur. Physikalisch besteht das Dokument aus einer Reihe von Einheiten, genannt Entities. Ein Entity kann auf andere Entities verweisen, um sie in das Dokument einzubinden. Jedes Dokument beginnt in einer Wurzel oder Dokument-Entity. Aus logischer Sicht besteht das Dokument aus Deklarationen, Elementen, Kommentaren, Zeichenreferenzen und Processing Instructions, die innerhalb des Dokuments durch explizites Markup

ausgezeichnet sind. Logische und physikalische Strukturen müssen korrekt verschachtelt sein.

Es gelten für Wohlgeformtheit eine Reihe von weitere Bedingungen. So schreibt die Recommendation auch vor, dass alle Elemente der XML-Instanz von einem klammernden sogenannten root-Element, umschlossen werden. Außerdem müssen beispielsweise alle Attributwerte in einfachen oder doppelten Anführungszeichen stehen.

Valide XML-Dokumente müssen wohlgeformt sein, und darüber hinaus von einer DTD (bzw. eines XML-Schemas) abgeleitet sein, und allen Deklarationen des Inhaltsmodells, die in der DTD definiert wurden, entsprechen.

Validierende Parser machen also einen Vergleich des XML-Dokumentes mit der dazugehörigen DTD. Alle weiteren Beschränkungen der Validität finden sich in der Recommendation der XML-Version 1.0.

Grenzen und Nachteile der DTDs:

- DTDs haben eine eigene (nicht XML konforme) Syntax. Das Erstellen von DTDs macht das Erlernen einer neuen Sprache notwendig, was kurzzeitig einen höheren Aufwand und damit höhere Kosten verursacht.

- Namensräume werden nicht unterstützt. Setzen sich die Beschreibungsregeln eines XML Dokuments aus mehreren DTDs aus verschiedenen Namensräumen zusammen, so lässt sich die Mehrfachverwendung von Bezeichnern nicht explizit ausschließen. Die so auftretenden Namenskollisionen verletzen die Bedingung der Wohlgeformtheit von XML Dokumenten.

- Stark eingegrenzter Typvorrat. Außer einiger Stringtypen und expliziter Aufzählung verfügen DTDs über keine Möglichkeit zur Beschreibung weiterer Datenformate (wie z.B. number, date, currency, etc.). Das Definieren eigener Datentypen ist damit ebenfalls nicht möglich. Dieses erweist sich vor allem bei der Kommunikation zweier Webapplikationen (e-commerce) als schwerer Nachteil, da es Aufgabe der Zielapplikation (z.B. einer Datenbank) bleibt, die Gültigkeit des empfangenen Dokumentes, bzw. der enthaltenen Daten zu prüfen. Damit geht ein großer Vorteil von XML, die Flexibilität, verloren.

- Rudimentäres Konzept zur Erweiterbarkeit. Die Erweiterungsmöglichkeiten von DTDs lassen sich im Wesentlichen reduzieren auf einfache String-Ersetzungen (über Parameter Entities). Eine systematische und von außen nachvollziehbare Modularisierung läßt sich damit nur schwer durchführen.

3.3 Schema

Betrachtet man im Verlauf der Entwicklung von Metasprachen die Inhaltsmodellierung in Form von DTDs, so wird im Hinblick auf die dynamische Entwicklung des Internets schnell klar, dass die Möglichkeiten von DTDs begrenzt sind.

Durch dynamische Seitengenerierung im Internet wird zugleich mehr Kontrolle aber auch mehr Flexibilität bezüglich der Inhaltsmodellierung notwendig. So möchte man beispielsweise ein Intervall von Werten genauer definieren können, als nur von Null bis unendlich. Gleichzeitig möchte man aber auch ein Element Postleitzahl eingrenzen auf den Datentyp Integer und den Bereich von 00000 bis 99999. Dies könnte man durch einen regulären Ausdruck überprüfen.

Es ist in Schema möglich, einem Element ein Intervall vorzugeben, welches von den eher vagen Häufigkeitsmodifikatoren in DTDs abweicht und wesentlich genauer ist. Schemas beschreiben eher Klassen von Objekten, als nur Dokumente. Denn es geht um die Modellierung von Daten und um weniger die semantische Struktur von Dokumenten.

Ein weiterer Vorteil von XML-Schema ist die Möglichkeit Vererbungen darzustellen und zu nutzen. Zwar gibt es in den DTDs bereits die Möglichkeit mit Parameter-Entities ein Inhaltsmodell an ein anderes Element zu übergeben, diese Beziehung ist aber keine verwandtschaftliche. Sie begründet kein Verhältnis von Klasse zu Subklasse.

Der Code eines Schemas ist - im Gegensatz zur DTD - ebenfalls in XML-Syntax gehalten. Dies hat auch den Vorteil, dass Parser, die XML-Dokumente gegen Schema statt gegen DTDs validieren, mit einem reduzierten Satz syntaktischer Regeln umgehen - mit den Regeln, denen XML-Dokumente ohnehin folgen müssen, um wohlgeformt zu sein. Das vereinfacht die Parser-Algorithmik und kommt tendentiell der Performanz der Verarbeitung von Dokumenten zugute. Der Gebrauch von Namensräumen, ermöglicht es, eventuelle Elemente, die in anderen Schemas definiert wurden, in das Inhaltsmodell einzubauen, ähnlich externen Entitäten in DTDs.

Somit bieten Schemas eine Menge Vorteile gegenüber DTDs, jedoch gerät durch verschiedene Implementierungen unter Berücksichtigung verschiedener Namensräume oder Elementdeklarationen dies möglicherweise auch bald zum Nachteil, weil Unübersichtlichkeit und die mangelnde Standardisierung problematisch werden könnten - welcher Parser unterstützt welche Schemas?

Die XML-Schema Recommendation 1.0 gliedert sich in zwei Teile:

- Teil 1 spezifiziert den strukturellen Teil der Schema-Sprache, d.h. die möglichen Objekttypen und ihre möglichen Beziehungen.

- Teil2 spezifiziert die möglichen Inhaltstypen von Objekttypen, d.h. Datentypen für Elemente und Attribute sowie einen Mechanismus zur Definition benutzerspezifischer Datentypen.

Wichtige Neuerungen von XML Schema:

- XML Schema ist XML. D.h. es existiert sowohl eine DTD zur Beschreibung von XML Schema als auch ein selbstbeschreibendes XML Schema. Ein gegen ein XML Schema geprüftes XML Dokument (document instance) kann wohlgeformt (well formed) sein oder wohlgeformt und gültig (valid). Der Test auf Gültigkeit beinhaltet neben der Überprüfung auf korrekte Reihenfolge und Struktur der Tags auch die Überprüfung auf Einhaltung der Wertebereiche und gültigen Datentypen.

- Umfangreicher Vorrat an vordefinierten, einfachen Datentypen. XML Schema bietet zahlreiche Datentypen an, zusammen mit der Möglichkeit, den Wertebereich explizit anzugeben. Mit diesen Basistypen können weitere, komplexe Elementtypen definiert werden. Die einmalige Definition von Daten- oder Elementtypen bei unter Umständen häufiger Verwendung fördert zum einen die Lesbarkeit des Schemas als auch die spätere Verarbeitung durch einen Parser/Validator, etc.

- XML Schema erlaubt explizites Gruppieren von Attributen. Attribute, die wiederkehrend in Gruppen verwendet werden (z.B. HTML: width und height) können nun als Attributgruppe einmalig definiert werden. Dieser Mechanismus, der bei DTDs mit Parameter Entities realisiert wurde, kann nun explizit angegeben und damit auch von einem Parser/Validator genutzt werden.

- XML Schema ermöglicht die Definition neuer Elementtypen auf Basis vorangegangener Definitionen (Vererbung). Durch die Erweiterung/Einschränkung bestehender komplexer Elementtypen um weitere Elemente oder Attribute lassen sich gleichartige Elementfamilien deklarieren.

- Unterstützung von Namensräumen (Namespaces). Durch die Berücksichtigung von Namensräumen lassen sich document instances erstellen, die auf Elemente in verschiedenen XML-Schema Beschreibungen zugreifen. Elemente mit gleichem Namen aber unterschiedlicher Struktur können, sofern durch Namensräume getrennt, in einer document instance erscheinen.

Beispiel einer Schemadefinition für eine E-Mail:

mail.xsd:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
  <xsd:complexType name="mailType">
    <xsd:sequence>
      <xsd:element name="from" type="xsd:string"/>
      <xsd:element name="to" type="xsd:string"/>
      <xsd:element name="cc" type="xsd:string" minOccurs="0"
maxOccurs="1"/>
      <xsd:element name="bcc" type="xsd:string" minOccurs="0"
maxOccurs="1"/>
      <xsd:element name="subject" type="xsd:string"/>
      <xsd:element name="body" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="mail" type="mailType"/>
</xsd:schema>
```

Das oben definierte Schema kann als Grundlage für eine E-Mail im XML Format dienen. Diese könnte folgendes Aussehen haben:

mail.xml:

```
<?xml version="1.0"?>
  <mail
    xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="mail.xsd">

  <from>bsp@www.beispiel.com</from>
  <to>recipient@www.beispiel.com</to>
  <subject>Schemabsp</subject>
  <body>
    Hi,
    Ein kleines Schemabeispiel!
  </body>
</mail>
```

Mit dem Attribut `xmlns:xsi` legt man innerhalb des XML Dokumentes fest, welche Art von Schemadefinitionssprache verwendet werden soll. Mit dem folgenden Attribut `xsi:noNamspaceLocation` verweist man dann auf das zu verwendende Schema. Ein validierender Parser ist nun in der Lage zu

prüfen, ob das Dokument den Angaben im Schema entspricht oder ob es Abweichungen gibt.

3.4 XSL

Ergänzend zu den *Cascading Stylesheets* (CSS) in HTML kümmert sich die Extensible Style Language (XSL) speziell um die Formatvorlagen zu XML. Sie bestimmt das Layout einer Seite und dient somit als Vorschrift für die Umwandlung der XML-Dokumente in andere Konstrukte.

Den Begriff Stylesheet kann man mit Formatvorlage oder Layoutvorlage übersetzten. Dabei handelt es sich um eine Vorlage zur Umwandlung der logischen Auszeichnungen in die physischen Auszeichnungen. Die Cascading Style Sheets, die für die Verwendung unter HTML vorgesehen sind, sind inzwischen in der Version 2.0 erschienen.

XSL ist abgeleitet von der *Document Style Semantics and Specification Language* (DSSSL), die ihren Ursprung in der SGML-Entwicklung hat. Die Sprache DSSSL wurde dazu entworfen, SGML-Dokumente zu verarbeiten (formatieren). Dabei wurde viel Wert auf weitgehende Flexibilität gelegt. So ist die Ausgabe von DSSSL grundsätzlich in beliebigen Formaten (RTF (Rich Text Format), PS (PostScript) usw.) möglich, sofern das Formatierungsprogramm dazu in der Lage ist. Wesentliche Grundzüge von CSS bzw. der neueren Version CSS 2.0 fanden in XSL Eingang.

Wichtige Aufgaben von XSL:

- Die Bildung einer Sprache mit der sich XML-Dokumente in andere Formate konvertieren lassen.
- Zur Verfügung stellen eines Vokabulars, das den semantischen Tags bestimmte Formatierungen zuweist.

Der Grundbaustein der XSL-Stylesheets ist das Template. Dieses beschreibt, wann ein XML-Elementknoten (dieses Element und alle darin enthaltenen Elemente) mit einem XSL-Elementknoten korrespondiert.

Eine solche Regel besteht aus:

- Einem Muster (pattern), das den XML-Knoten (Element) im XML-Dokument kennzeichnet. Durch das Muster wird festgelegt auf welchen (selbstdefinierten) XML-Befehl sich die dann folgende Stilanweisung (Action) bezieht. Das Muster ist das Auswahlkriterium, wann die definierte Ausgabeform auf den Inhalt eines Markups umzusetzen ist.

- Einer Aktion (Action). Wurde das angegeben Muster im XML-Dokument erkannt, dann folgt die Umwandlung des betreffenden Elements in die angegebene Ausgabeform. Eine Action-Anweisung kann neben passiven Elementen zur reinen Stilgestaltung auch dynamische Anweisungen, z.B. den Aufruf einer anderen Skriptsprache enthalten.

Der XML-Prozessor verarbeitet den XML-Dokumentbaum rekursiv, um einen Ausgabebaum anzulegen. Um zu verhindern, dass dieser rekursive Prozeß unterbrochen wird, wenn es keine geeignete Auswahlregel gibt, setzt der XSL-Prozessor eine Standardregel voraus.

3.5 XSLT

Die *eXtensible Style Language for Transformations* (XSLT, [20]) stellt eine Erweiterung der reinen Formatierungssprache XSL und den daran angeschlossenen Techniken dar. XSLT wurde vor allem dafür entwickelt, um dynamisch aus XML-Datenbeständen Teile zu extrahieren und sie anderen Anwendungen zur Verfügung zu stellen und sie dabei auch zugleich in neue Formate von Auszeichnungssprachen zu überführen.

Der XSLT Prozeß benötigt das zu transformierende Quelldokument, das ein XML Dokument sein muß und ein Stylesheet, das auch im XML Format vorliegt und in dem sich die Transformationsvorschriften befinden. Die Ausgabe beinhaltet das transformierte Dokument. Dieses kann, muß aber nicht ein XML Dokument sein.

Der Transformationsprozess arbeitet nicht direkt auf den Dokumenten, sondern auf einer DOM Repräsentation, also einem Baum, in dem die einzelnen Elemente des Quelldokumentes und auch des Stylesheets durch Knoten repräsentiert werden. Der Transformationsprozess beginnt an der Dokumentwurzel im Quelldokument. Jetzt wird im Stylesheet eine passende Regel gesucht, die beschreibt, was mit diesem Knoten geschehen soll. Diese Regel wird *Templaterule* genannt. Wird eine Templaterule gefunden, die für die Dokumentwurzel zuständig ist, wird sie instanziert, d.h. es wird der Teilbaum im Stylesheet durchlaufen, der den Inhalt der Templaterule bildet.

Jetzt wird für jedes Element in diesem Stylesheet-Fragment überprüft, ob es zur Anwendungsdomäne des XSLT-Prozessors gehört. Gehört das Element dazu, wird es ausgewertet. Gehört es zu einem Namensraum, der für Erweiterungen innerhalb des Stylesheets deklariert wurde, wird es an einen näher zu spezifizierenden Auswertungsprozess weitergegeben. Besitzt das Element keinen Namensraum oder ist dieser für den Prozessor nicht von Bedeutung, wird das Element nach erfolgter Attributexpansion in das Ergebnisdokument geschrieben.

Innerhalb der Templaterule wird jetzt die Auswahl von Elementen aus dem Quelldokument gesteuert. Hierzu kann man sich verschiedener XSLT-Konstrukte bedienen.

Bereits heute gibt es eine Fülle von Maschinen und Entwicklungsumgebungen, welche die Arbeit mit XSLT ermöglichen. Eine der ersten Maschinen die weltweit zum Einsatz kamen, war die von James Clark entwickelte XT [16].

Beim Apache XML Projekt wird seit einer Weile die Cocoon-Umgebung angeboten. Diese auf Servlets basierende Maschine ermöglicht XSLT-Transformationen in einer Serverumgebung. Benötigt wird dazu ein Webserver und eine Servletmaschine wie JRun, Apache JServ oder Jakarta Tomcat.

3.6 XLink

Wer Hypertext im Allgemeinen und HTML im besonderen kennt, will die wesentlichen Eigenschaften beider natürlich nicht missen: die Möglichkeit, Textstellen und Dateien miteinander über einen Link (Verbindung) zu verknüpfen. Auch in XML besteht diese Möglichkeit. Es existieren zwei Entwürfe, je einer zu *XLink* und *XPointer* , die zwischen einfachen und erweiterten Links, sowie solchen auf einzelne Bestandteile einer XML-Instanz unterscheiden.

Die Unterteilung in Links mit einfachem und erweitertem Funktionsumfang erleichtert die Handhabung und lässt zu, dass sich einfache Links ähnlich wie in HTML schnell und flexibel verwenden lassen.

Anders als die Syntax der Sprache ist die Spezifikation des Linking noch nicht abgeschlossen [31].

- Einfache XLinks entsprechen ungefähr den Hyperlinkfähigkeiten von HTML.
- Erweiterte XLinks. Diese erweiterten Links können nicht nur auf ein Dokument, sondern auf mehrere Quellen verweisen. Es lassen sich Links zu Daten einrichten, die selbst keine Linkfunktion unterstützen. Neu ist auch die Einrichtung von so genannten bidirektionalen Links, d.h. Links, die nicht nur vom Dokument zur Datenquelle verweisen, sondern auch in die andere Richtung von den Daten zum Dokument.

Beispiel:

```
<a xml:link="simple"
   href="http://www.bsp.com/index.xml">Linkbezeichnung</a>
```

Zu beachten ist, dass XML-Elementnamen unterschiedlich behandelt werden, wenn sie sich in Groß- und Kleinbuchstaben unterscheiden!

Im Gegensatz zum einfachen Link kann der erweiterte auf mehrere Ziele verweisen.

Sowohl das einfache als auch das erweiterte Linkelement können über die Adreßangabe (href) hinaus weitere Attribute enthalten, die die Bedeutung der Adreßangabe sowie der referenzierten Ressourcen und ihre Rolle betreffen. Es handelt sich dabei um:

- href
- inline
- (content-)title
- show
- actuate
- behavior

Bsp:

```
<einfach href="http://www.beispiel.com/index.xml"
     content-role="Anleitung"
     content-title="XML to any"
     show="new">Ein kleines Tutorial</einfach>
```

Ob und wie ein Browser oder eine andere Anwendung, die XML-Daten verarbeiten kann, mit Attributen wie content-role und content-title umgeht ist ihr überlassen. Zunächst liegt der Sinn von Attributen darin, das Markup zu strukturieren und prinzipiell die Möglichkeit zu schaffen, auf solche Informationen zugreifen zu können.

Besonderheiten der erweiterten Links:

- Es ist möglich, Verweise von Read-only-Medien oder Dateien aus zu anderen Stellen einzurichten (beispielsweise von einer auf CD-ROM gespeicherten Datei oder einer anderen Website aus ...).
- Man kann Links zu und von Daten erzeugen, die selbst kein Linking unterstützen.

Schließlich lassen sich Links zu Gruppen zusammenfassen: zu einer erweiterten Linkgruppe. Die Dokumente, die in solchen Gruppen referenziert werden, sind erweiterte Linkdokumente und müssen ebenfalls in einer DTD entsprechend deklariert werden. Linkgruppen haben als Wert für xml:link den Wert group, Linkdokumente document. Sie enthalten im href-Attribute die konkrete Adreßangaben.

3.7 XPointer

XLinks bietet bereits zahlreiche Möglichkeiten auf externe Dokumente zu verweisen und eine Webseite zu verknüpfen. *XPointer* können aber noch präziser auf die Struktur des Dokumentes reagieren und Verknüpfungen auf einzelne Elemente setzen.

XPointer [32] beschreiben einen Ort oder Bereich innerhalb einer XML-Instanz. Um dies zu tun, bedienen sie sich der Struktur der einzelnen Dokumente.

Es existieren verschiedene Möglichkeiten auf Elemente einer Seite zuzugreifen: Absolute Verweisausdrücke, relative Verweise und Verweise auf Attribute, Zeichenketten und Bereiche.

3.8 XPath

XPath ist aus der Bemühung entstanden, Funktionen, die sowohl in der XSL-Transformation, als auch in der XPointer Spezifikation [15] benötigt wurden, zu kapseln. XPath bietet Zugriff auf Teile des XML Dokumentbaumes. Die Syntax von XPath wurde kompakt gehalten, um sie in einer URL (Uniform Resource Locator) beziehungsweise als Attributwert zu speichern. Sie orientiert sich dabei an einer Pfadstruktur. Ausgehen von einem beliebigen Element lassen sich verschiedene Pfade anlegen.

3.9 Namensräume

Als Namensraum bezeichnet man den Raum, aus dem ein Dokument ein Detail - oder beliebig viele - seiner eigenen Dokumenttyp-Definition bezieht. Ein Problem bei der Erstellung von Dokumenten besteht immer dann, wenn Dokumente zu erstellen sind, die Teile von anderen XML-Dokumenten einsetzen. In diesem Fall kann es dazu kommen, dass Elementnamen doppelt vergeben sind, weil vielleicht mehrere Autoren an den Dokumenten gearbeitet haben. Dieses Problem wird mit Namensräumen (namespaces,[33])

gelöst. Bei Namensräumen geht es darum, für häufig vorkommende Elemente standardisierte Namen und Deklarationen zu vergeben und sie nicht immer neu zu definieren.

XML-Daten bestehen aus Elementen, auch Tags genannt, mit optionalen Attributen und deren Werten. Damit stellt sich aber ein Problem: wie soll ein Parser im Internet eine Unterscheidung treffen zwischen gleichen Namen, die zu verschiedenen logischen Zuordnungen gehören, und aus verschiedenen XML-Instanzen zusammengesetzt werden?

Angenommen, die beiden folgenden Elementgruppen kämen aus zwei verschiedenen DTDs, nämlich books und owners:

```
<book>
     <title>Buchtitel</title>
     <author>Name des Autors</author>
</book>

<owner>
     <name>Name einer Person</name>
     <title>Dr.</title>
</owner>
```

Hier wird das Problem deutlich, beide Elemente haben eine semantisch logische Bedeutung. Der erste Tag bezieht sich auf den Buchtitel, der zweite auf den akademischen Titel des Besitzers.

Genau hier setzt der Ansatz der XML-Namensräume an:

Es wird ein Präfix definiert, welches es ermöglicht die Namensräume der verschiedenen Elementgruppen zu unterscheiden.

```
<books:title xmlns:books "http://www.beispiel.com/books">
<owners:title xmlns:owners "http://www.beispiel.com/owners">

<books:book>
     <books:title>Buchtitel</books:title>
     <books:author>Name des Autors</books:author>
</books:book>
<owners:owner>
     <owners:name>Name einer Person</owners:name>
     <owners:title>Dr.</owners:title>
</owners:owner>
```

Hier ist also das Element title als Kindelement von book in die Namensraumvererbung mit einbezogen, das Element author aber nicht.

3.10 Formatobjekte

Die neueste XSL-Version stellt eigene Flußobjekte bereit, bezeichnet diese aber als Formatobjekte. Formatobjekte sind dem Namensraum *fo* zugeordnet. Das Formatobjekt wird auf den Ergebnisbaumknoten im Musterteil des Elements angewendet:

```
<xsl:template match="[muster]">
     <fo:[Formatobjekt] ([Stileigenschaft]="[Wert]")*>
     </fo:[Formatobjekt]>
</xsl:template>
```

Der aktuelle XSL-Entwurf beschreibt nur das page-sequence-Objekt und das simple-page-master-Objekt. Später werden weitere Formatobjekte eingeführt. Jedes Formatobjekt hat bestimmte Eigenschaften.

Zur Verarbeitung kann z.B. der *Formating Objects Processor* (FOP, [7]) verwendet werden. Dieser ist in Java geschrieben. Er interpretiert den *Formating Object Tree* und erzeugt daraus PDF Dateien oder andere Ausgabeformate. Dabei ist zu beachten, dass FOP nur einen Teil der *Formating Objects* ([10]) und deren Attribute implementiert, wie sie vom W3C als Empfehlung vom 15. Oktober 2001 für die *eXtensible Stylesheet Language* XSL vorgeschlagen wurden.

3.11 DOM

Die Grundidee von DOM ist es, die einzelnen Bestandteile eines XML-Dokuments über Objekte zu repräsentieren. Das *Document Object Model* ist eine Beschreibung der Baumstruktur eines XML-Datenmodells. Diese abstrakten Schnittstellen sind in der programmiersprachen-neutralen Sprache, der Interface Definition Language (IDL) der Object Management Group (OGM) abgefaßt. Ziel dieser Technologie ist es, den Baum analysieren zu können und auf einzelne Teile zugreifen und diese modifizieren zu können. Der Zugriff auf die Baumstruktur erfolgt über eine konkrete Implementierung dieses Document Object Models in einem *Application Programmers Interface* (API) für eine Programmiersprache.

Von einem Objektmodell ist im Zusammenhang von HTML und JavaScript oft die Rede. Der Zugriff auf im Dokument enthaltene Formulare und Grafiken erweitert die Möglichkeit von Web-Autoren. Innerhalb des W3C läuft die Arbeit an einem generellen Modell (*Document Object Model, DOM*), das sowohl für HTML, als auch für XML Grundfunktionen vorsieht.

Ein DOM Parser baut den XML Baum im Speicher auf wodurch ein sehr schneller Zugriff auf die Daten in XML ermöglicht wird. Für die Programmiersprache Java bietet sich vor allem das von IBM und Apache entworfene Xerces-API an. Diese beinhaltet sowohl Funktionalitäten aus dem DOM Level 1, aber inzwischen auch aus DOM Level 2.

Das DOM interpretiert eine XML-Instanz als Baum und, um es aus Sicht einer objektorientierten Programmiersprache wie Java verständlicher zu beschreiben, als Objekte. Eine DOM Implementierung überführt einen XML-Baum in ein Java Objekt, analysiert und modifiziert es, und transformiert das Objekt wiederum in einen XML-Baum. Mit DOM kann jedes beliebige XML-Dokument als Objekt- bzw. Datenstruktur aufbereitet werden, ohne dass dabei im Dokument enthaltene Informationen verloren gehen.

Dabei ist für das DOM der Typ eines Objekts entscheidend: Typen können beispielweise Elemente, Attribute, Kommentare oder CDATA Abschnitte sein. Diese Objekttypen sind im XML Baum wiederum - in der allgemeingültigsten Beschreibung - Knoten (nodes). Diese Knoten können, in Abhängigkeit vom spezifischen Typ, bestimmte Eigenschaften haben. So haben Kommentare keinen Namen und Processing Instructions haben keine Kindelemente. Daher erben alle abgeleiteten Klassen des abstrakten Objekttyps alle Eigenschaften, ohne sie konkret umsetzen zu müssen (d.h. sie liefern im Zweifelsfall null zurück).

Auch das gesamte XML Dokument ist ein Knoten und das umschließende Wurzelelement ist wiederum ein Kind des Dokumentknotens. Dieser Dokumentknoten kann noch weitere Kindelemente haben: Kommentare, Processsing Instructions und Dokument Typ Deklarationen. Dem Dokumentknoten sind zahlreiche Methoden zur Generierung von Dokumentinhalten in Form von Knoten und Elementen zugeordnet.

Als weitere Implementierung folgen dann die Elemente: Elemente können Elemente als Kindelemente haben oder auch Text, Kommentare oder ähnliches. Attribute sind als Knoten dem Dokumentbaum zugeordnet, sind aber keine möglichen Kinder von Knoten sondern vielmehr Eigenschaften von Elementen.

Um einen DOM-Baum als XML-Dokument zu speichern, muß der DOM-Objektbaum vollständig durchlaufen werden. Während der Travesierung wird die jeweilige XML-Repräsentation des gerade besuchten Baumknotens aufgebaut und ausgegeben. Der Baum wird - analog zur Schachtelung der in ihm enthaltenen XML-Strukturen - nach dem Depth-First-Prinzip durchlaufen. Knoten, die einen oder mehrere Unterknoten besitzen, werden dabei mehrfach besucht, und zwar vor und nach dem Durchlaufen jedes Unterknotens. Für ein Containerelement z.B. wird zunächst dessen öffnendes Tag erstellt: Nach der spitzen Klammer und dem Elementbezeichner werden die

Attribute des Elements ausgegeben; erst wenn alle Attributknoten durchlaufen wurden, wird die schließende spitze Klammer ausgegeben. Nun werden die im Containerelement enthaltenen Textblöcke und Elemente der Reihe nach ausgegeben, bevor das schließende Tag des Containerelements gesetzt wird. Dieses Vorgehen führt zur sequentiellen Ausgabe des gesamten Bauminhalts, d.h. der Objektbaum wird serialisiert.

3.12 SAX2

SAX steht für *Simple API for XML*. Dabei handelt es sich um eine Standardschnittstelle für ereignisbasiertes XML-Parsing. Das Dokument wird Element um Element abgearbeitet. Die Kommunikation mit anderen Anwendungen wird dabei über sogenannte Handler abgewickelt, weshalb man auch von einem ereignisgesteuerten Protokoll spricht. Es ist somit bei extrem großen XML-Dokumenten weniger speicherintensiv. Die Implementierung ist allerdings etwas komplexer.

Schnittstellen und Klassen in der SAX2-Distribution:

Teile dieser Schnittstellen sind insbesondere für die Parser-Entwickler vorgesehen. Andere dagegen für die Anwendungsentwickler gedacht:

- `DocumentHandler`: Dies ist der wichtigste Handler zur Verarbeitung überhaupt. In ihm werden die Methoden `startDocument` und `endDocument` implementiert, die jeweils zu Beginn und zum Ende eines Dokuments auftreten. Desweiteren gibt es die Methoden `startElement` und `endElement`, die aufgerufen werden, wenn der Parser ein einleitendes Tag bzw. ein schließendes Tag erkennt. Erkennt der Parser z.B. ein einleitendes Tag, wird die Methode `startElement` mit dem Namen des Tags und sämtlichen Attributwertpaaren als Parameter aufgerufen. In dieser Methode kann der Programmierer nun die Verarbeitung des Tags steuern, z.B. Name und Attribute des Tags ausgeben. Analog dazu wird die Methode `endElement` aufgerufen, wenn der Parser ein schließendes Tag erkennt. Erkennt der Parser hingegen freien Text, der zwischen Tags steht, wird die Methode `characters` aufgerufen, findet er eine Processing Instruction die Methode `processingInstruction`. Außerdem gibt es noch die Methode `setDocumentLocator`, mit der ein Locator-Objekt an den DocumentHandler übergeben wird. Dieser Locator gibt an, wo der Parser sich gerade im XML-Dokument befindet. Zur Behandlung von Whitespace gibt es die Methode `ignorableWhitespace`.

- `ErrorHandler`: Dieser Handler behandelt alle Fehler, die beim Parsen auftreten. Dafür gibt es die Methoden `warning`, `error` und `fatalEr-`

ror, die je nach Schwere des Fehlers beim Parsen aufgerufen werden. Als Parameter erhalten diese Methoden eine `SAXParseException`, in der nähere Angaben zur Art des Fehlers und der Position, an der er aufgetreten ist, gespeichert sind. Das SAX-API liefert zwar einen Default ErrorHandler in der HandlerBase Klasse, dieser wirft aber nur Exceptions bei fatalErrors, andere Fehler ignoriert er. Um Fehler in XML-Dokumenten zu erkennen, sollte man hier also auf jeden Fall seinen eigenen ErrorHandler mit speziellen Fehlerbehandlungen schreiben und beim Parser registrieren.

- `DTDHandler`: Dieser Handler kommt zur Anwendung, wenn der Parser in einer DTD auf ein `unparsed entity`, also Binärdaten mit einer zugehörigen Notation, trifft. Er ruft dann entweder die Methode `unparsedEntityDecl` oder `notationDecl` auf, in der der Programmierer seine eigene Behandlung der unparsed entities durchführen kann.

- `EntityResolver`: Der EntityResolver wird benötigt, wenn der Parser auf Referenzen zu externen Dateien, z.B. DTDs trifft, und diese lesen muß. Der Parser ruft dann die einzige Methode dieses Handlers, `resolveEntity` auf, die aus einer öffentlichen ID (URN=`Universal Resource Name`) eine System ID (URL=`Universal Resource Locator`) macht.

Einfache XML-Anwendungen können unter Verwendung von DocumentHandlern und gegebenenfalls ErrorHandlern entwickelt werden.

Nachfolgend sind die wichtigsten Methoden der DocumentHandler Schnittstelle aufgelistet:

- `startDocument()` - Zum Empfang von Benachrichtigungen über den Anfang eines Dokuments.

- `endDocument()` - Zum Empfang von Benachrichtigungen über das Ende eines Dokuments.

- `startElement(String name, AttributeList atts)` - Zum Empfang von Benachrichtigungen über den Anfang eines Elements.

- `endElement(String name)` - Zum Empfang von Benachrichtigungen über das Ende eines Elements.

- `characters(char ch[], int start, int length)` - Zum Empfang von Benachrichtigungen über Zeichendaten.

- `processingInstruction(String target, String data)` - Zum Empfang von Benachrichtigungen über eine Verarbeitungsanweisung.

Das SAX-Paket beinhaltet außerdem die Klasse DefaultHandler, die Standardimplementierungen der vier oben aufgeführten Schnittstellen bietet.

Die Anwendungsentwickler haben damit zwei Möglichkeiten:

- Sie können eine Klasse entwickeln, die die benötigten Schnittstellen implementiert.
- Sie können eine Unterklasse der Klasse DefaultHandler anlegen - die Standardimplementierungen der Schnittstelle enthält.

3.13 Servlets

Servlets sind in Java implementierte Anwendungskomponenten, die auf einem Server-System ausgeführt werden, um dort Anfragen von Clients zu empfangen und zu bearbeiten. Innerhalb von Web Anwendungen verarbeiten Servlets HTTP-Anfragen und liefern HTTP-Antworten zurück.

Ein Servlet realisiert üblicherweise einen einzelnen Dienst, der dann vom Server angeboten wird. Ein einzelnes Servlet ist daher in der Regel nicht als vollwertige Anwendung nutzbar, sondern benötigt für seine Arbeit zusätzlich Informationen, die vom Server bereitgestellt werden müssen. Außerdem sind Servlets normale Java-Objekte, d.h. sie können nicht direkt über das HTTP-Protokoll angesprochen werden: HTTP-Anfragen müssen in Methodenaufrufe umgesetzt werden, und die vom Servlet in Objektform gelieferten Ergebnisdaten müssen in eine HTTP-Antwort konvertiert werden. Für diese Aufgaben wird ein sogenannter *Servlet-Container* eingesetzt. Ein Servlet-Container arbeitet als Adapter zwischen Web-Server und Servlets und dient somit zur Integration von Web-Anwendungen in Web-Servern.

Empfängt der Web-Server eine HTTP-Anfrage, die an ein Servlet gerichtet ist, so leitet er die Anfrage an den Servlet-Container weiter, der seinerseits die HTTP-Anfrage analysiert. Die Analyse liefert Informationen darüber, welches Servlet die Anfrage verarbeiten soll, und welche Methode für die Datenübertragung gewählt wurde. Anhand dieser Information kann der Servlet-Container dann die gewünschte Verarbeitungsmethode des gesuchten Servlets aufrufen. Die HTTP-Anfrage wird dazu in ein *HttpServletRequest* Objekt umgesetzt und beim Methodenaufruf als Argument übergeben. Zusätzlich wird noch ein *HttpServletResponse* Objekt übergeben, in das das Servlet die Anfrageergebnisse einträgt. Nach Abschluß der Anfrageverarbeitung durch das Servlet erstellt der Servlet-Container aus den Daten in diesem Objekt die HTTP-Antwort.

Ein Servlet ist letzten Endes nichts anderes als eine Instanz einer Servlet-Klasse. Existiert noch kein geeignetes Servlet, um eine eingehende Anfrage

zu bearbeiten, ist es Aufgabe des Servlet-Containers, die richtige Klasse zu finden und zu instanzieren bzw. zu initialisieren. Der Servlet-Container stellt so automatisch sicher, dass alle korrekt formulierten Anfragen auch bearbeitet werden.

Kapitel 4

Das Framework

Für das Framework können drei unterschiedliche Gruppen von Benutzern der Software identifiziert werden: Der *Endanwender*, respektive der Autor von Dokumenten, der ausschließlich daran interessiert ist, seine Dokumente gemäß der entsprechenden XML-Struktur - d. h. unter Verwendung eines bereits genau definierten XML-Schemas - zu verfassen. Der *Dokumententwickler*, der verschiedene Dokumentarten klassifiziert, die jeweiligen Schematas dazu festlegt und die entsprechenden Transformationen entwickelt, mit denen er das Basislayout der Zielformate festlegt. Und der *Systementwickler*, der den Kern des Systems entwirft und diesem weitere Funktionalität hinzufügen kann. Um dem zu entsprechen, wurde ein Framework konzipiert, das es erlaubt, auf verschiedene Art und Weise erweitert und eingesetzt zu werden.

4.1 Ein einfaches Beispiel

An Hand eines einfaches Beispiels werden die Grundstrukturen und Vorgangsweisen des Frameworks skizziert. Es soll ein Eingabedokument (Input, Source), das im XML-Format vorliegt , in ein konkretes Zielformat - in diesem Fall HTML - übersetzt werden. Als erster Schritt muß eine Beschreibung des Sourcedokumentes - sozusagen einer Dokumentenklasse - modelliert werden. Der Einfachheit halber wird in diesem nur ein Element `<doc>` und ein Element `<title>` verarbeitet. Die Information, welche mit dem `<title>` Element umschlossen wird, soll in das HTML-Ausgabedokument übernommen werden.

In XML Syntax sieht dieses Beispiel dann folgendermaßen aus:

`simpledoc.xml`:

```
<doc
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="simpledoc.xsd">

  <title>Title Definition</title>
</doc>
```

Um garantieren zu können, dass die Verarbeitung dieses Dokumentes durch das Framework korrekt abläuft, muß zu dem Eingabedokument ein XML-Schema (siehe Abschnitt 3.3) entworfen werden. Dieses XML-Schema beschreibt, wie ein gültiges XML-Dokument aufgebaut sein muss.

`simpledoc.xsd`:

```
<?xml version="1.0" standalone="no"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="doc">
  <xsd:complexType>
      <xsd:all>
            <xsd:element name="title" type="xsd:string"/>
       </xsd:all>
    </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Im Folgenden wird die Abarbeitung dieses Dokumentes, im Hinblick auf das Ziel, daraus ein HTML-Dokument zu machen, betrachtet. Im Eingabedokument sind durch dessen XML-Struktur folgende Informationen implizit enthalten:

- `<doc>` - Drei Informationen: Erstens, dass das Dokument begonnen hat, zweitens das Auftreten eines `<doc>` Elementes und drittens, dass das File `simpledoc.xsd` das zugehörige XML-Schema ist.
- `<title>` Das Auftreten des `<title>`-Elementes
- "`Title Definition`" - Alle Zeichen zwischen dem `<title>` und dem `</title>` Element, in diesem Fall die Zeichenfolge "`Title Definition`"
- `</title>` - Das Auftreten des `</title>`-Elementes

- `</doc>` - Das Auftreten des `</doc>`-Elementes und das Ende des Dokumentes

Bei einer sequentiellen Abarbeitung, dem sogenannten *Parsen* des Eingabedokumentes, kann man die Folge der Informationen als Folge von Ereignissen (Events), die etwas bewirken können, auffassen (im Detail siehe Abschnitt 4.2.1).

Dieses XML-Dokument, das dem zugehörigem XML-Schema genügt, ist nur ein Dokument aus der Menge der Dokumente, die mit dem XML-Schema beschrieben werden. Wir müssen auf alle Kombinationen respektive Reihenfolgen dieser Ereignisse reagieren können, die das entworfene XML-Schema eindeutig und überprüfbar vorgibt. Somit ergibt sich für diese Dokumentenklasse das in Abbildung 4.1 dargestellte Ablaufdiagramm, das alle möglichen und gültigen Pfade enthält.

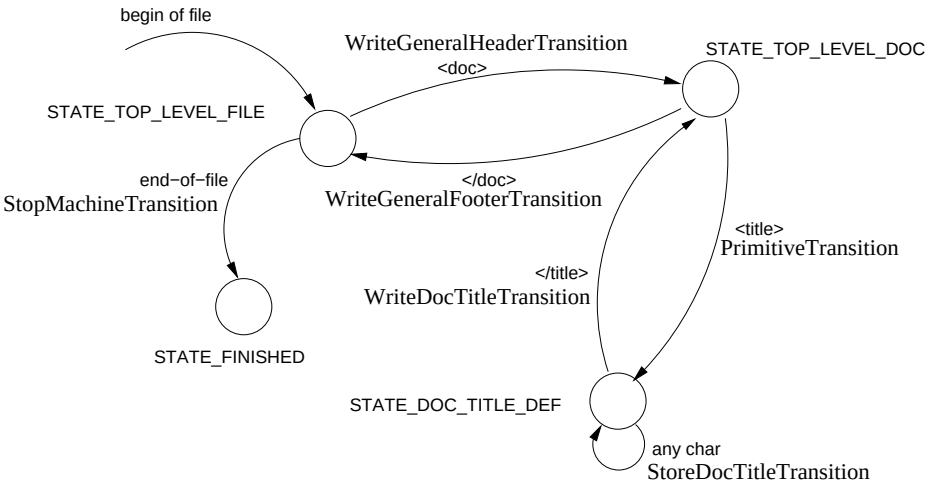

Abbildung 4.1: Das Ablaufdiagramm zum `<doc>` Beispiel

Das alleine stellt allerdings nur eine theoretische Betrachtung aller gewünschten Abläufe der Ereignisse dar. Wie man am Ablaufdiagramm leicht erkennt, kann dieses Verhalten mit einem dem Diagramm entsprechenden endlichen Automaten (Finite-State-Machine, in weiterer Folge *Statemachine* genannt) erreicht werden. Dabei bilden die Ereignisse die auslösenden Momente (Trigger, Transitionen) von einem gültigen Zwischenzustand (State) zum nächsten gültigen Zwischenzustand. Diese Zustände müssen nicht zwingend verschieden sein. Jedes Dokument wird zumindest zwei *ausgezeichnete* Zustände haben, den Dokument-Start-Zustand und den Dokument-Ende-Zustand.

Diese Statemachine ist das Kernstück des Frameworks (siehe Abschnitt 4.2). Für die Statemachine ist eine von obigem Schema abgeleitete Beschreibung

erforderlich, damit diese das Dokument (respektive die ganze mit obigem Schema beschriebene Dokumentenart) *versteht*. Diese Beschreibung erfolgt zweckmäßiger Weise auch mittels eines validierbaren XML-Dokumentes, da man verschiedene Dokumentklassen verarbeiten will und damit auch entsprechend unterschiedliche endliche Automaten aufbauen muss. Für die Validierung muß diese Beschreibung auch einem XML-Schema genügen, dem XML-Schema für die Statemachine (siehe Anhang A).

Bereits an diesem kleinem Beispiel kann man erkennen, dass die Anzahl der Zustände schnell sehr umfangreich wird. Allerdings muß dies nur einmal für eine Dokumentklasse gemacht werden. Nachfolgend sind die wesentlichen Abschnitte dieses XML-Dokumentes, dem sogenannten Konfigurationsfile der Statemaschine, für diese Dokumentklasse, dargestellt.

1. Angabe des Schemas

```
<?xml version="1.0" standalone="no"?>

<statemachine
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="StateMachineConfig.xsd">
```

2. Die Liste aller gültigen Zustände. Der Startzustand muß besonders ausgezeichnet sein, damit man einen gültigen Ausgangszustand der Statemachine hat.

```
  <states>
    <startstate>STATE_TOP_LEVEL_FILE</startstate>
    <state>STATE_TOP_LEVEL_DOC</state>
    <state>STATE_DOC_TITLE_DEF</state>
    <state>STATE_FINISHED</state>
  </states>
```

3. Die Liste aller zugelassenen Transitionen. Die Zustände des endlichen Automaten werden vor den Transitionen bekanntgegeben, damit schon beim Parsen festgestellt werden kann, ob die Transitionen eine gültige Zustandsänderung beschreiben.

```
  <transitions>
    <transition>
      <beginstate>STATE_TOP_LEVEL_FILE</beginstate>
      <nextstate>STATE_TOP_LEVEL_DOC</nextstate>
      <element type="start">doc</element>
      <classname>WriteGeneralHeaderTransition</classname>
```

```
    </transition>
    .
    .
    <transition>
      <beginstate>STATE_TOP_LEVEL_DOC</beginstate>
      <nextstate>STATE_TOP_LEVEL_FILE</nextstate>
      <element type="end">doc</element>
      <classname>WriteGeneralFooterTransition</classname>
    </transition>
    .
    .
    <transition>
      <beginstate>STATE_TOP_LEVEL_FILE</beginstate>
      <nextstate>STATE_FINISHED</nextstate>
      <element type="enddoc" />
      <classname>StopMachineTransition</classname>
    </transition>
  </transitions>
</statemachine>
```

Damit sind sämtliche für die Übersetzung des XML-Sourcedokuments notwendigen Dokumente (Schemata, Konfigurationsfile, ...) beschrieben.

Das Ziel ist, aus einem XML-Eingabedokument ein HTML-Ausgabedokument zu machen. Es fehlt noch der Programmcode bzw. die Anweisungen, welche die Generierung des Outputs in der Zielsprache - für dieses kleine Beispiel HTML - übernehmen. Das Hinzufügen dieser Anweisungen wird dadurch erreicht, dass die Transition als Interface `Transition` definiert ist, und der Implementierung dieses Interfaces der gewünschte Programmcode beigefügt wird (im Detail siehe Abschnitt 4.2.5). In Abbildung 4.1 sind die entsprechenden Klassennamen, bei den Übergängen von einem Zustand zum nächsten Zustand, eingetragen. Diese Klassennamen finden sich auch im XML-Konfigurationsdokument der Statemaschine wieder.

Die hier benötigten Anweisungen bzw. Klassen sind nun:

PrimitiveTransition Diese implementiert nur das Interface einer Transition und trägt nichts zum Output bei. Sie wird benötigt um von einem Zustand der Statemachine in einen anderen zu wechseln.

WriteGeneralHeaderTransition Diese schreibt den allgemeinen Kopf des Outputdokuments. Im diesem Beispiel mit HTML als Zielsprache ist dies das öffnende `<html>` Tag, das komplette `<head>` Element und das öffnende `<body>` Tag.

StoreDocTitleTransition Diese speichert den Inhalt des `<title>` Elements des XML-Eingabedokuments (im Detail siehe Abschnitt 4.2.2).

WriteDocTitleTransition Diese schreibt den Inhalt des `<title>` Elements des XML-Source in den Output, eingebettet in ein `<h1>` Element.

WriteGeneralFooterTransition Diese schreibt nun das generelle Dokumentende des Outputs; für das HTML Beispiel ist dies das schließende `</body>` und das schließende `</html>` Tag.

StopMachineTransition Diese wird aufgerufen, wenn der komplette Input abgearbeitet wurde (End-Of-File bei einem Eingabedokument). Sie stoppt die Statemachine und stellt somit sicher, dass das Framework korrekt terminieren kann.

Mit obigem XML-Konfigurationsdokument wird eine Statemachine aufgebaut, die Sourcedokumente, welche dem entsprechenden Schema genügen, verarbeiten kann.

Mit Hilfe der entsprechenden Transitionen generiert die Statemachine beim Abarbeiten des Eingabedokumentes das gewünschte Ausgabedokument. Für dieses einfache Beispiel wird der generierte Output dann folgendermaßen aussehen:

```
<html>
  <head>
    <title></title>
  </head>

  <body>
    <h1>Title Definition</h1>
  </body>
</html>
```

4.2 Das Design des Frameworks

Bereits für das vorhergehende einfache Beispiel ist ein gewisser Aufwand notwendig, um zu dem gewünschten Ergebnis zu gelangen. Zusätzlich zu dem eigentlich zu übersetzenden Eingabedokument mußten ein XML-Schema für dieses, ein XML-Schema für die Statemaschine, ein Konfigurationsfile für die Statemachine und eine Reihe von Klassen, für die Transitionen, erstellt

werden. Es wurde allerdings der gesamte Aufwand aller Benutzergruppen *Systementwickler*, *Dokumententwickler* und *Endanwender* dargestellt. Unterteilt man nun wieder in diese drei Gruppen von Benutzern der Software, relativiert sich dieser Aufwand (siehe Abbildung 4.2).

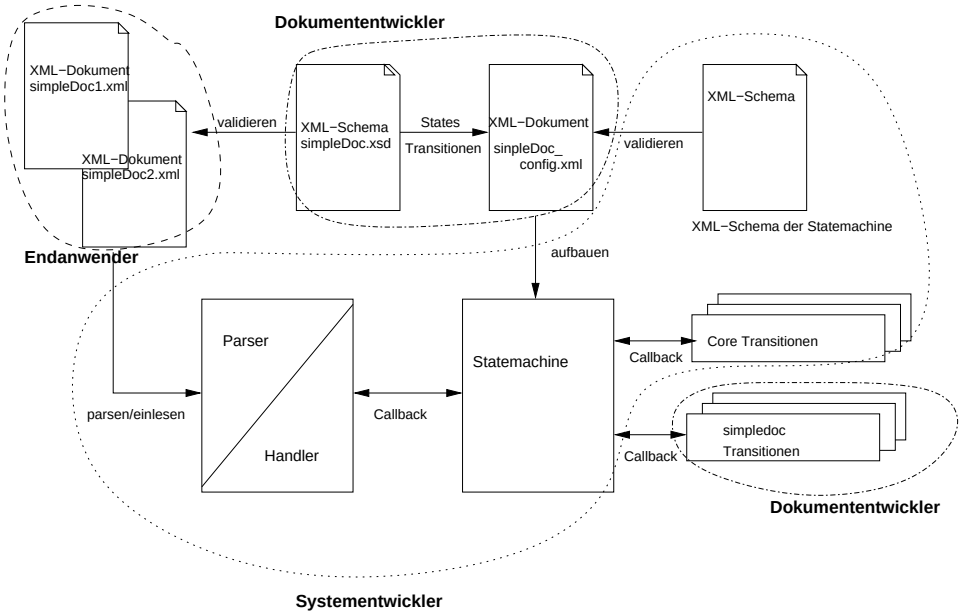

Abbildung 4.2: Framework: Die Benutzergruppen

Der *Systementwickler*, welcher den funktionalen Kern des Frameworks entwickelt, entwirft dazu auch das entsprechende XML-Schema für das XML--Konfigurationsfile der Statemachine. Dieses Statemachineschema wird zusammen mit dem Kern des System nur einmal geschrieben und ändert sich in Folge nur bei gravierenden Eingriffen in die Kernfunktionalität des Systems.

Der *Dokumententwickler* schreibt danach für eine ganze Klasse von Dokumenten ein dem Statemachineschema genügendes XML-Konfigurationsfile mit dem dann eine Statemachine aufgebaut wird, die die Dokumente dieser Klasse verarbeiten kann. Korrespondierend zu diesem XML-Konfigurationsfile entwirft der Dokumententwickler ein XML-Schema für die Dokumente. Auch implementiert er die notwendigen Transitionen (siehe auch Abschnitt 4.2.5), die den Output für diese Klasse von Dokumenten generieren sollen. Somit kreiert er für eine bestimmte Klasse von Sourcedokumenten ein komplettes wiederverwendbares Package (siehe Abschnitt 4.2.5) - bestehend aus dem XML-Schema der Dokumentklasse, dem XML-Konfigurationsfile der Statemaschine und Java-Klassen für die Transitionen - rund um den Systemkern.

Die Arbeit des Dokumententwicklers muss noch geeignet optimiert werden,

da zur Zeit alles noch *von Hand* gemacht werden muss.

Der *Endanwender* nimmt nun Systemkern und eines dieser Packages und muß nur das Sourcedokument erstellen - dieses muß allerdings dem im ausgewählten Package enthaltenen Schema entsprechen - und kann somit den Output in der gewünschten Zielsprache generieren.

Ähnlich der Unterteilung in verschiedene Benutzergruppen des Frameworks, kann dieses auch aus funktionaler Sicht in *Parser*, *Handler*, *Statemachine* und *Transitionen* bzw. *Packages von Transitionen* aufgeteilt werden (siehe Abbildung 4.3).

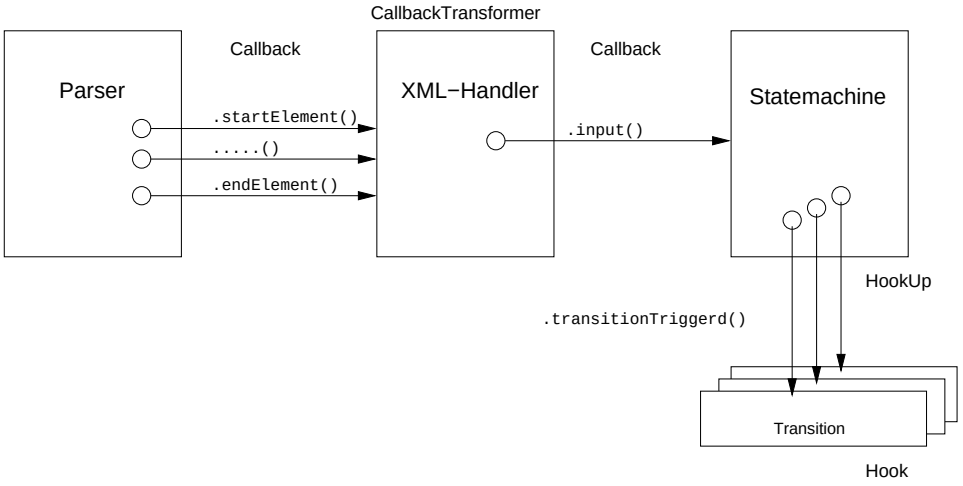

Abbildung 4.3: Framework: funktionale Sicht/Designpattern

Der *Parser* (siehe Abschnitt 4.2.1) liest das Sourcedokument ein. Er meldet das Auftreten der Ereignisse dem bei ihm registrierten *Handler* (Callback) und prüft gleichzeitig gegen das XML-Schema.

Der *XMLHandler* (siehe Abschnitt 4.2.3) speichert den XML-Input(siehe Abschnitt 4.2.2) und ruft seinerseits eine Callbackmethode der Statemachine auf (vgl. Callbacktransformer-Pattern [29]).

Die *Statemachine* (siehe Abschnitt 4.2.4 bzw. vgl. Hook-Pattern [29]) ruft die entsprechend als *Hook* registrierte Transition auf.

Die *Transitionen* (siehe Abschnitt 4.2.5) werden durch die *Statemachine* die deren *Hookup* darstellt, getriggert und erzeugen schlußendlich den gewünschten Output.

Bei den Transitionen kann man außerdem noch zwischen sogenannten *Core* -Transitionen - das sind gleichsam eine kleine Anzahl von Basistransitionen, die direkt zum Framework gehören - und den Transitionen in den einzelnen *Packages* (siehe Abschnitt 4.2.5) unterscheiden, die zur jeweils entsprechen-

den Klasse von Dokumenten, die damit verarbeitet werden soll, gehören.

In den folgenden Abschnitten werden die einzelnen Teile des Frameworks im Detail beschrieben.

4.2.1 Der Parser

Dieser Teil des Frameworks übernimmt das Einlesen, sprich Parsen, des XML-Sourcedokuments. Da beim Parsen mit DOM (Document Object Model) - Technologie die Gefahr besteht, dass es bei großen Dokumenten zu einem großen Speicherbedarf kommen kann, da der gesamte DOM-Baum des Dokumentes im Speicher gehalten werden muss, wird ein ereignisbasierter Parser verwendet.

Dazu wird SAX2 und Xerces, ein XML-Parser der das SAX (Simple API for XML) - Interface implementiert, verwendet (siehe auch Abschnitt 3.12)[2][27]. Zu diesem Zweck wird über die `XMLReaderFactory` der Xerces-SAXParser als `XMLReader` erzeugt. Der `XMLReader` gibt die auftretenden SAXParse-Events über eine Callback-Funktion an einen bei ihm zuvor registrierten Handler (siehe Abschnitt 4.2.3) weiter.

Für ein korrektes Funktionieren des Frameworks muß beim Parsen festgestellt werden, ob das Eingabedokument korrekt ist. Technisch kann das auf die zwei vorgestellten Arten, mit Hilfe einer DTD (siehe Abschnitt 3.2) oder eines XML-Schemas (siehe Abschnitt 3.3) geschehen. Aufgrund der Vorteile von XML-Schema wird dieses zur Spezifikation der Eingabedokumente verwendet.

Somit muß der `XMLReader` auf Validierung mit XML-Schema gestellt werden, damit auf fehlerhafte Eingabedokumente reagiert werden kann (siehe Abschnitt 4.3). Hierfür sieht SAX2 die Methode `setFeature` der Klasse `XMLReader` vor, wobei einer der Parameter ein String ist, der eine URI (Uniform Resource Identifier) repräsentiert. Die entsprechenden Parameter für Validierung und XML-Schema bei SAX2 bzw. Xerces sind nun:

```
http://xml.org/sax/features/validation
http://apache.org/xml/features/validation/schema
```

4.2.2 Verwaltung von Inputdaten

Da eventbasiert geparst wird, steht - im Gegensatz zum DOM-Parsen - die Information über aufgetretene Elemente, ihren Inhalt und eventuellen Attributen nur bei deren Auftreten zur Verfügung. Dies hat einerseits den Vorteil, dass sich nicht der ganze DOM-Baum bei der Verarbeitung im Arbeitspeicher befinden muss, jedoch andererseits den Nachteil, dass die Informationen

die noch nicht bei ihrem Auftreten während des Parsens verarbeitet werden können, oder mehrmals benötigt werden, in geeigneter Weise gespeichert werden müssen.

Dieses *Speichern* kann an zwei verschiedenen Punkten des Gesamtsystems passieren. Einerseits im Framework und andererseits in den Transitionen. Um beides zu ermöglichen, muss es ein Objekt geben, auf das sowohl vom Framework wie auch aus den Transitionen heraus zugegriffen werden kann, das `DataObject`.

Für einen naiven Designansatz würde also ein `DataObject`, in dem Daten gespeichert und wieder gelesen werden können, reichen. Da Inhalt, Name und zugehörige Attribute eines Elementes beim Parsen verschiedene Events sind, liegen diese Daten in einer Transition nie gleichzeitig vor. Werden diese Daten jedoch für die Verarbeitung zusammen benötigt, müssten sie in der Transition ihres Auftretens gespeichert werden. Bei dem Beispiel in Abschnitt 4.1 passiert das in der `StoreDocTitleTransition`. Das müsste bei allen auf diese Weise zu verarbeiteten Elementen in den zugehörigen Transitionen implementiert werden.

Es bietet sich für diese Problemstellung aber ein besserer Lösungsweg als der oben gezeigte an. Da eine der Forderungen in der XML-Recommendation des W3C (World Wide Web Consortium)[13] vorschreibt, dass Elemente in der umgekehrten Reihenfolge geschlossen werden müssen in der sie geöffnet wurden, bietet sich zur Zwischenspeicherung ein Stack an.

Als Datenstruktur zur Speicherung der relevanten Informationen eines Elementes dient das `XMLElement`. Es besteht aus dem Namen des Elementes, den Attributen und aus dem Wert, falls es sich hierbei um Zeichen (Charakters) und nicht um ein weiteres Element handelt.

Diese Speicherung übernimmt der Event-Handler des Parser (siehe Abschnitt 4.2.3). Der Stack ist Teil des oben erwähnten `DataObjects`, wobei es in den Transitionen nur lesenden Zugriff auf das oberste Element gibt. Es ist ersichtlich, dass sich der Transitionsprogrammierer um diese Zwischenspeicherung keine Gedanken mehr machen muss, da diese ja vom Handler des Parser übernommen wird, und in den Transitionen auf das oberste `XMLElement` des Stacks lesend zugegriffen werden kann. Die Einschränkung, dass Transitionen nicht auf den Stack schreiben dürfen, ergibt sich aus der Baumstruktur eines XML-Dokumentes. Dürften in den Transitionen Elemente auf dem Stack verändert werden, wäre die eindeutige Zuordnung zwischen dem Inhalt des Stacks und den auftretenden Events des Parsers nicht mehr gewährleistet.

Durch die Verwendung des Stacks stehen alle Informationen zu einem Element beim entsprechenden schließenden Tag zur Verfügung. Benötigt man diese Informationen jedoch während der Dokumentverarbeitung öfter oder

erst später, muss diese Information auf einem geeigneten Weg zusätzlich gespeichert werden. Diese Aufgabe übernimmt das *UserDefinedDataObject*, eine Hash-Map in welche Daten innerhalb von Transitionen abgelegt werden können, die bei ihrem Auftreten noch nicht verarbeitet werden können, oder öfters benötigt werden. Diese Daten können dann in späteren Transitionen wieder gelesen und verarbeitet werden. Es ist vom Transitionsprogrammierer nur darauf zu achten, dass er die Schlüsselwörter (Keys) zur Speicherung so vergibt, dass die zum Key passenden Daten wieder gefunden werden können.

Da das *UserDefinedDataObject* als Hash-Map implementiert ist, beschränken sich die speicherbaren Daten nicht nur auf Strings. Es wird hier zum Beispiel bei der Initialisierung die zu initialisierende Statemachine gespeichert bzw. man kann auch das Ausgabedokument hier aufbauen.

Das `DataObject` vereinigt den `XMLELement`-Stack und das *UserDefinedDataObject*. Dieses `DataObject` wird wie schon im naiven Designansatz jeder Transitionsklasse übergeben. Aus den Transitionen heraus hat man lesenden und schreibenden Zugriff auf das *UserDefinedDataObject* und wie aus den oben erwähnten Gründen nur lesenden Zugriff auf den Stack.

4.2.3 Der Event-Handler

Es wurde bis jetzt noch nicht erklärt wie die Ereignisse des Parsers im Framework verarbeitet werden.

Für die Verarbeitung der Events stellt SAX2 verschiedene Handler zur Verfügung (siehe Abschnitt 3.12). Es muss vor Beginn des Parsens dem `XMLReader` eine Klasse, die das `DocumentHandler`-Interface des SAX implementiert oder von der Referenzimplementierung des SAX, der Klasse `DefaultHandler` abgeleitet ist, bekanntgegeben werden. Für den Fall, dass die erste Möglichkeit gewählt wurde und eine Fehlerbehandlung durch den Parser möglich sein soll, muss die Klasse weiters das `ErrorHandler`-Interface der SAX implementieren. In unserem Fall haben wir uns für die zweite Möglichkeit entschieden und den `XMLHandler` von der Klasse `DefaultHandler`, welche unter anderem das `DefaultHandler`- und das `ErrorHandler`-Interface implementiert, abgeleitet.

Es muss im `XMLHandler` des Frameworks definiert werden, wie die auftretenden Events behandelt werden sollen. Im vorliegenden Design haben die Callback-Methoden des `XMLHandler` zwei Aufgaben. Einerseits die Verbindung mit der Statemachine und anderseits die Zwischenspeicherung der Elementinformationen am Stack des `DataObjects` (siehe Abschnitt 4.2.2). Um die Verbindung mit der Statemachine im Framework herzustellen, wird beim Auftreten eines Ereignisses in der entsprechenden Methode des Handlers der

Input, welcher diese Ereignisse erzeugt, der Statemachine gemeldet. Diese Weitermeldung entspricht eigentlich einer Callback-Methode der Statemachine.

Um die zweite geforderte Aufgabe zu erledigen, wird bei einem öffnenden Tag ein `XMLElement`, mit den zu diesem Zeitpunkt bekannten Informationen (Name und eventuell vorhandene Attribute) auf den Stack gelegt. Besteht das Element aus einzelnen Zeichen (Charakters), werden diese beim Auftreten als String im obersten `XMLElement` des Stacks gespeichert (vorausgesetzt der Stack wächst von unten nach oben, ansonsten bezieht sich die Aussage auf das unterste Element des Stacks). Wird das Element nun geschlossen, liegen alle relevanten Informationen nun im obersten `XMLElement` des Stacks vor und können jetzt weiterverarbeitet werden. Ein schließendes Tag bewirkt weiter, dass das oberste Element wieder vom Stack genommen wird. Daraus ist ersichtlich, dass sich im Speicher immer nur soviele Elemente befinden wie gerade geöffnet wurden, also nur ein Ast des Dokumentbaumes, und nicht der gesamte Dokumentbaum (vgl. Abschnitt 3.11). Es wird aber auch klar, dass bei einem schliessenden Tag zuerst die Callback-Methode der Statemachine aufgerufen werden muss, und erst dann das Element vom Stack genommen werden kann, da ansonsten die Elementinformationen in der von der Statemachine aufgerufenen Transition nicht mehr vorhanden sind.

4.2.4 Die Statemachine

Die Statemachine ist die Implementierung eines deterministischen endlichen Automaten. Für jede Dokumentklasse muss ein - dieser Dokumentklasse entsprechender - Automat aufgebaut werden. Es wird also eine Möglichkeit benötigt, unterschiedliche Automaten zu beschreiben und mit Hilfe dieser Beschreibung aufzubauen. Um möglichst flexibel zu bleiben, passiert dies mit einem XML-Konfigurationsdokument, welches dem Statemachine XML-Schema genügen muss.

Das Konfigurationsdokument enthält die States (`<state>`) für die Statemachine, inklusive einem ausgezeichneten Startstate (`<startstate>`).

Weiters enthält es die Transitionen (`<transition>`), wobei jede Transition aus einem Beginstate (`<beginstate>`), einem Nextstate (`<nextstate>`), einem Element (`<element>`), durch welches sie getriggert wird und dem Klassennamen (`<classname>`) der Transition, welche das *Transition Interfaces* implementiert.

Das `<element>` Element sieht ein Attribut vor, um eine Unterscheidung zwischen den verschiedenen Typen von Elementen bzw. Events zu definieren, da die SAX API hierfür unterschiedliche Callbackfunktionen vorsieht. Da-

mit muss bei Transitionen, die nicht durch ein öffnendes oder schliessendes `<element>` Element getriggert werden, wie zum Beispiel `startDocument`, `endDocument` und `characters`, nur der Typ mit dem Attribut angegeben werden, nicht aber der Name des Elements (siehe Beispiel in Abschnitt 4.1).

Weiters muss im Konfigurationsdokument noch der Suchpfad (`<path>`) für die Transitionen bzw. Transitionspackages angegeben werden, damit die Klassen, die sich ja in verschiedenen Packages befinden können, zur Laufzeit gefunden werden.

Das Konfigurationsdokument wird, wie schon bei der Erklärung der SAX API *eventbasiert* geparst. Die Statemachine stellt nach dem Hook-Pattern [29] einen Hookup dar. Die Transitionen werden demnach bei ihr als Hooks registriert. Zuvor werden der Statemachine alle möglichen Zustände (States) bekanntgegeben. Zum Zeitpunkt des Registrierens der Transitionen kann also überprüft werden, ob diese von einem registrierten Zustand zu einem registrierten Zustand führt.

Zum Registrieren der States und Transitionen in der Statemachine gibt es nun zwei Alternativen. Entweder wird dies direkt in die Callbacks des SAX Parsers hineinverpackt, oder man verwendet dazu wiederum eine Statemachine die das XML-Statemachinekonfigurationsfile verarbeiten kann. Diese Variante hat den Vorteil, dass man einheitlich im Bezug auf die Verarbeitung von Eingabedokumenten ist, jedoch den Nachteil, das diese *InitStateMachine* "hardcodiert" sein muss. Im vorliegenden Design des Frameworks wird die zweite Alternative mit der "hardcodierten" Statemachine verwendet.

Das XML-Schema für das Konfigurationsdokument ist unabhängig von den unterschiedlichen Verwendungszwecken immer das selbe. Da die Verarbeitung des Konfigurationsdokumentes hardcodiert ist, bedingt eine Änderung dieses XML-Schemas einen Eingriff in den Sourcecode!

Genau genommen werden allerdings nicht direkt die entsprechenden Aktionen/Klassen registriert sondern nur eine sogenannte *ClassLoaderTransition* welche den Namen der entsprechenden Klasse dann beinhaltet und bei Erhalt eines Triggers dann mittels des *Factory Patterns* [23] diese zur Laufzeit nachlädt und sogleich einen Trigger auf diese setzt (vgl. *Calltransformer* [29]), sie also auch gleich ausführt. Somit werden die Klassen erst beim Auftreten eines entsprechenden Triggers nachgeladen. Auch wenn eine Transition mehrmals benötigt wird wird sie nur einmalig nachgeladen, da diese Implementierung des Factory Patterns in Java intern die Klassen cacht betreibt.

4.2.5 Die Transitionen

Neben dem Parser, der den Input verarbeitet und die entsprechenden Events an die Statemachine weiterleitet, welche dann die Ablaufsteuerung darstellt, ist noch ein funktionaler Teil notwendig, welcher Aktionen setzt und somit den gewünschten Output generieren kann. Diesen Teil übernehmen die Transitionen.

Alle Transitionen müssen folgendes Interface implementieren:

```
public interface Transition
{
  public String transitionTriggered(State from_state,
                                    State to_state,
                                    DataObject data)
    throws TransitionException;
}
```

Darin ist die Methode `transitionTriggered` der Callback, welcher beim entsprechenden Event von der Statemachine aufgerufen wird und die gewünschten Aktionen dann durchführt. Der Parameter `from_state` beinhaltet den Zustand der Statemachine in jenem Zeitpunkt in dem sie den Event erhält und der Parameter `to_state` ist der Zustand in welchen sie nach abarbeiten der Transition übergehen wird. Das `DataObject data` beinhaltet das zum aktuellen Event gehörende `XMLElement` und stellt die Datenverwaltung zu Verfügung (siehe Abschnitt 4.2.2).

Aus funktionaler Sicht kann man die Transitionen im Groben in speichernde und schreibende unterteilen: Die speichernden legen den aktuellen Input (welcher im `XMLElement` enthalten ist) im `DataObject` ab - eventuell in bereits modifizierter Form. Im Gegensatz dazu sind die schreibenden Transitionen verantwortlich dafür, aus all den Informationen im `DataObject` sequentiell den gewünschten Output zu generieren. Dieser wird ebenfalls während seiner Generierung im `DataObject` zwischengespeichert.

Da sämtliche gewünschte Aktionen in den Transitionensklassen implementiert werden, können diese auch Funktionalität wie Filemanagement (das generierte Enddokument soll möglicherweise in ein File geschrieben werden), Datenmanipulation und Akkumulierung neuer Daten, Einbinden anderer Java-Packages mit beliebiger Funktionalität und der gleichen mehr enthalten.

Weiters kann man die Transitionen in sogenannte *Core*-Transitionen und Transitionen in den einzelnen Packages unterteilen:

Core-Transitionen

Diese haben entweder nur eine gewisse Basisfunktionalität oder werden in der Initialisierungsphase für die Statemachine benötigt. Sie können nichts zu einem etwaigen Output beitragen und gehören direkt zum Kern des Frameworks.

Core-Transitionen sind nun folgende:

PrimitiveTransition Sie implementiert nur das Interface einer Transition und beinhaltet sonst keine weitere Funktionalität. Mit dieser ist es möglich von einem Zustand der Statemachine in einen anderen zu wechseln.

RegisterStateTransition Wird in der Initialisierungsphase benötigt um Zustände in der Statemachine einzutragen.

RegisterStartStateTransition Registriert analog zur `RegisterStateTransition` einen Zustand in der Statemachine und markiert außerdem diesen als Startzustand.

ClassLoaderTransition Diese beinhaltet in Gegensatz zu den restlichen Transitionen zwei Membervariablen: eine `Factory` und einen Klassennamen. Diese beiden werden mit einem eigenen Konstruktor initialisiert. Ihre Funktionsweise ist dann, dass sie mittels `Factory` versucht eine Transition mit angegebenem Klassennamen zu laden und, wenn dies gelingt auf diese dann sogleich einen Trigger zu setzt - das heißt den Aufruf gleich weiterzuleiten und die implementierten Aktionen auszuführen.

SetSearchPathTransition Für das Laden von Transitionen mit der `Factory` kann in dieser ein Pfad im Dateiverzeichnis angegeben werden in welchen sie nach dieser suchen soll. Mit dieser Transition kann nun ein solcher Suchpfad in die `Factory` eingetragen werden.

StoreDataTransition Mit ihr werden die zum Registrieren einer Transition notwendigen Daten zwischengespeichert. Diese Daten sind die Zustände vor und nach einer Transition, der Event, auf welchen reagiert werden soll (inklusive der Art von diesem), und der Klassename der dazugehörenden Transition.

RegisterTransitionTransition Diese Transition trägt in der zu initialisierenden Statemachine entsprechend der dazugehörenden Daten eine `ClassLoaderTransition` ein, welche dann die eigentliche Transition enthält.

StopMachineTransition Diese Transition hält am Ende der ganzen Übersetzung die Statemachine an und stellt sicher, dass der ganze Prozess ordnungsgemäß terminiert.

Packages von Transitionen

Zum kompletten Framework mit seiner Basisfunktionalität können dann sogenannte Erweiterungs-Packages erstellt werden. Diese beschreiben entweder eine neue Klasse von Sourcedokumenten oder die Übersetzung in eine neue Zielsprache. Ein solches Package enthält nun ein XML-Schema für die damit verarbeitbare Dokumentenklasse, das dazugehörige Konfigurationsfile zum Aufsetzen der Statemachine und die gesamten zur Übersetzung notwendigen Transitionen.

4.3 Fehlerbehandlung

Bei der Verarbeitung von Dokumenten können verschiedene Arten von Fehlern auftreten.

Einerseits Fehler des Parsers und der Statemachine die aufgrund von "korrupten" Inputdateien auftreten können, andererseits Fehler die in Transitionen auftreten können.

4.3.1 Parser Fehler

Das W3C hat in der *XML Recommendation 1.0* [13] zwei Arten von Fehlern definiert.

error: *Nicht gültiges Dokument:* Es sind die Konsequenzen nicht definiert. Ein Beispiel hierfür ist, wenn das Dokument bei einem validierenden Parser nicht dem angegebenen Schema bzw. der DTD (Document Type Definition) entspricht.

fatal error: *Eine Verletzung der XML Spezifikation:* Es darf die normale Verarbeitung vom Parser nicht fortgesetzt werden. Diese Art von Fehler tritt zum Beispiel auf, wenn ein XML Dokument nicht wohlgeformt ist.

SAX stellt zur Verarbeitung dieser Fehler das Interface `ErrorHandler` zur Verfügung. Um vom Framework die Fehlerbehandlung vom Parse-Fehler zu gewährleisten, muss der `ErrorHandler` - im vorliegenden Fall implementiert der `XMLHandler` dieses Interface - beim `XMLReader` registriert werden. Es werden dann vom SAX Parser Fehler an diesen Handler gemeldet.

Es müssen laut diesem Interface die zwei Callback Funktionen `error` und `fatalError` implementiert werden. Die Informationen über den aufgetretenen Fehler werden den beiden Funktionen mittels `SAXParseException` als Parameter übergeben. Standardmäßig haben diese beiden Funktionen in der Referenzimplementierung des SAX, keine Funktion und müssen deshalb im `XMLHandler` überladen werden, da ein Ignorieren eventuell auftretender Fehler kein wünschenswertes Verhalten des Frameworks darstellt.
Dies wird dadurch ersichtlich, wenn man bedenkt, dass ein nicht wohlgeformtes bzw. nicht gültiges XML-Dokument falsche oder nicht vorhandene Transitionen auslösen könnte. Die Statemachine würde in einen undefinierten Zustand kommen und könnte die Verarbeitung nicht korrekt beenden. Nachdem diese Art von Fehler schon beim Parsen bemerken werden kann, wird er dort abgefangen. Es wird eine Fehlermeldung generiert und der Kern des Frameworks wird mit einer Exception benachrichtigt, um die Verarbeitung zu beenden.

4.3.2 Statemachine Fehler

Da die Statemachine dynamisch - in Abhängigkeit zu der verarbeitenden Dokumentklasse - aufgebaut wird, können verschiedene Fehler auftreten, die nicht vom Parser entdeckt und an das Framework gemeldet wurden. Hierbei handelt es sich einerseits um Fehler die bei der Initialisierung der Statemachine durch eine Konfigurationsdatei auftreten können, welche zwar wohlgeformt ist und dem XML-Schema der Konfigurationsdokumentklasse entspricht, jedoch trotzdem nicht korrekt verarbeitet werden kann. Solch ein Fehler könnte zum Beispiel der Versuch sein, mehrere States mit dem selben Namen zu registrieren, oder die Registrierung einer Transition, deren Anfangs- und/oder der Endstate nicht registriert wurde. Da durch ein XML Schema nur die Struktur eines Dokumentes vorgegeben, der Inhalt jedoch nur rudimentär eingeschränkt werden kann [12] können diese Fehler nicht vom Parser gefunden werden. Um die Transparenz, wie das Framework arbeitet, für den Dokumententwickler zu wahren, wurde die Designentscheidung getroffen, dass die Initialisierung der Statemachine bei einem solchen Fehler abgebrochen wird. Es könnte beim Versuch einen solchen Fehler bei der Initialisierung zu korrigieren nicht garantiert werden, dass die Statemachine das geforderte Verhalten zeigt.

Andererseits kann es bei der eigentlichen Verarbeitung jedoch auch zu Eingaben kommen die von der Statemachine nicht korrekt interpretiert werden können. Ein solcher Fehler tritt zum Beispiel auf, wenn es für einen Input im gerade aktuellen State keine gültige Transition gibt. Dies bewirkt, dass das Dokument nicht korrekt bis zum Ende verarbeitet werden kann und deshalb wird ein solcher Fehler wie ein Parsefehler behandelt. Warum kann dieses Verhalten nun auftreten wo doch der Parser validierend agiert? Einer der

Gründe könnte sein, dass die Statemachine für eine Dokumentklasse initialisiert wurde die nicht dem XML-Schema des zu verarbeitenden Dokuments entspricht. Für den Parser ist dieses Dokument valid, die Statemachine versteht es jedoch nicht und liefert dann einen Fehler.

Es kann jedoch auch passieren, dass eine Transitionsklasse nicht dynamisch mittels der Factory geladen werden kann. Dieser Fall kann auftreten, wenn im Konfigurationsdokument ein *Classname* einer Klasse angegeben wurde die es nicht gibt, oder wenn ein falscher Suchpfad für die Factory angegeben wurde. Diese Fehler führen auch zu einem Fehlerreport und zum geordneten Abbruch der Dokumentverarbeitung.

4.3.3 Transitions Fehler

Nachdem das Framework durch Transitionen erweitert werden kann, und die Möglichkeiten dieser Erweiterung fast unbegrenzt sind, sollte gewährleistet sein, dass Ausnahmesituationen in diesen Transition nicht den Absturz des Frameworks nach sich ziehen. Deshalb müssen Exceptions, die in neuen Transitionen nicht behandelt werden können in einer `TransitionException` verpackt werden. Da das Framework nicht weiss, warum diese `TransitionException` geworfen wurde, reagiert es auf eine solche Exception mit Beendigung der Verarbeitung und Fehlerausgabe, um wieder die Transparenz für den Transitionsprogrammierer zu wahren.

Kapitel 5

Anwendungsbeispiele

5.1 LaTeX: flexible Handhabung ganzer Dokumentblöcke

Während der Erstellung größerer Dokumente kann es immer wieder vorkommen, dass es notwendig ist, Teile eines Dokuments umzustrukturieren und dabei ganze Blöcke des Dokuments zu verschieben. Dabei kann es passieren, dass aus einem ursprünglich eigenen Kapitel nunmehr ein Unterkapitel wird, oder umgekehrt, aus einem anfangs nur als kleines Unterkapitel gedachten Block nun ein eigenes, großes Kapitel wird. Das bedeutet aber, dass Änderungen in der Dokumentenhierarchie vorgenommen werden. Nun ist es jedoch in Textbeschreibungssprachen wie LaTeX der Fall, dass diese Hierarchie eines Dokuments eine sehr genau vorgegebene Form haben muß - bei LaTeX sind dies der Reihe nach *part, chapter, section, subsection, subsubsection, paragraph, subparagraph.* Dies bedeutet nun aber, wenn bei solchen Umstrukturierungen aus Unterkapiteln eigene Kapitel werden (und analog umgekehrt), dass diese Modifikationen der Dokumenthierarchie händisch nachgebessert werden müssen: aus einem *section* wird ein *chapter*, aus dem darin enthaltenen *subsection* wird ein *section* und gleich weiter im ganzen verschobenen Block (respektive umgekehrt wenn aus einem Kapitel ein Unterkapitel wird).

Dieses händische Nachbessern wird obsolet, wenn man in der Dokumenthierarchie nur ein Element (z.B. *section*) kennt, unabhängig von der jeweiligen Hierarchieebene - man dieses also beliebig ineinander verschachteln kann. Aus der Tiefe dieser Verschachtelung erhält man dann die Ebene der Hierarchie für die Übersetzung in ein Zielformat (wie LaTeX) in dem man je Ebene einen unterschiedlichen Bezeichner hat.

Eine solche Dokumentenhierarchie kann nun zum Beispiel ein Aussehen wie

folgt haben:

```
<document>
  <section>
    <title>1. Kapitel</title>
    <para> ... </para>
  </section>
  <section>
    <title>2. Kapitel</title>
    <para> ... </para>
    <section>
      <title>1. Abschnitt</title>
      <para> ... </para>
    </section>
    <section>
      <title>2. Abschnitt</title>
      <para> ... </para>
    </section>
  </section>
</document>
```

Der eigentliche Inhalt der einzelnen Abschnitte befindet sich hier in den `<para>` Elementen, und wird entsprechend der weiters darin enthaltenen Elemente ins Zielformat übersetzt. Die `<title>` Elemente enthalten die Überschriften der einzelnen (Unter-)Kapitel. Für die `<section>` Elemente wird die jeweilige Verschachtelungstiefe akkumuliert (d.h. bei einen Starttag um eins erhöht und beim Endtag wieder um eins erniedrigt) und entsprechend dieser dann der entsprechende Bezeichner in der Zielsprache ausgewählt.

Zu obigem Beispiel kann nun die dazugehörige Dokumentenhierarchie in LaTeX folgendermaßen aussehen:

```
\begin{document}

\section{1. Kapitel}
  ...

\section{2. Kapitel}
  ...
\subsection{1. Abschnitt}
  ...
```

```
\subsection{2. Abschnitt}
  ...

\end{document}
```

5.2 Rechnen

Häufig kommt es vor, dass man eine ganze Menge von Daten als Input hat, sich jedoch nicht für diese gesamte Datenflut interessiert, sondern nur ein paar davon abgeleitete Daten benötigt, womit sich dann meist besser die gewünschten Aussagen belegen lassen. Das heißt, wenn als Input zum Beispiel eine ganze Tabelle mit Meßwerten zu Verfügung steht, ist es möglich daraus zusätzlich repräsentative Werte wie Maxima/Minima, Mittelwert, die Summe oder sonstige beliebige generierbare Daten zu ermitteln und in das Enddokument einfließen zu lassen. Das bedeutet aber gleichzeitig auch, dass bei einem etwaigen Update der Inputdaten mit einem neuerlichen Übersetzen des Dokuments diese generierten Daten im Output ebenfalls automatisch upgedatet werden.

Implementiert ist dies mit dem Framework in folgender Weise, dass aufgrund des Designs sämtliche Inputdaten der einzelnen XML-Elemente im zur Datenspeicherung vorgesehenen `DataObject` (siehe Abschnitt 4.2.2) verwaltet werden können. Es können somit also auch ganze Tabellen oder Listen von Inputdaten dort gesammelt abgelegt werden und dann daraus die benötigten Daten - wobei dies beginnend von Extremwerten, Mittelwerten oder Summen bis hin zu ganzen statistischen Berechnungen oder Verteilungen oder beliebige andere komplizierte Berechnungen sein können - akkumuliert und entsprechend in den Output eingefügt werden.

Als einfaches Beispiel für diese Funktionalität wurde eine Liste implementiert, aus deren Daten dann wahlweise der Mittelwert oder die Summe berechnet werden und abschließend jeweils der Liste angefügt werden. Das heißt, man hat wahlweise einen der folgenden Inputs:

Beispiel A:	Beispiel B:
<para> Liste mit Summe: </para> <list eval="sum"> <item>3.7</item> <item>2.8</item> <item>13.2</item> </list>	<para> Liste mit Durchschnitt: </para> <list eval="avg"> <item>3.7</item> <item>2.8</item> <item>13.2</item> </list>

Und daraus wird dann alternativ der folgende Output generiert:

Liste mit Summe:	Liste mit Durchschnitt
3.7 2.8 13.2	3.7 2.8 13.2
$\sum$ **19.7**	$\varnothing$ **6.57**

5.3 Lebenslauf - 2 verschiedene Zielformate

Ziel dieses auf den ersten Blick relativ einfachen Anwendungsbeispiels ist es, einen Lebenslauf zu erstellen und zu übersetzen. Dazu wurde auch ein eigenes Schema entworfen, dass das Eingabeformat desselben beschreibt. Es besteht aus einem ersten Teil, welcher die gesamten persönlichen Daten wie Name, Geburtsdaten, Adresse, Telefon und dergleichen enthält und anschließend einer Reihe von Listen zu den einzelnen Themen, wie zum Beispiel Schulbildung, beruflicher Werdegang, weitere Qualifikationen und ähnlichem mehr. Außerdem sind noch als Attribute eine Bezeichnung des Lebenslaufs und eine Sprachauswahl vorgesehen.

Dieser Aufbau für das Eingabeformat des Lebenslaufs wurde dadurch gewonnen, da als erstes Zielformat LaTeX in Zusammenspiel mit einem speziellen CV (Curriculum Vitae) Style gewählt wurde. Die Übersetzung des Lebenslaufes nach LaTeX wurde auch speziell auf die Verwendung dieses Styles abgestimmt.

Der Lebenslauf soll nun aber auch noch in andere Zielformate (z.B. HTML) übersetzt werden können. Dazu kann der Großteil der Transitionen für die Übersetzung nach LaTeX wiederverwendet werden und nur jene wenige, die

effektiv den Output formatieren, entsprechend adaptiert werden. Es können also, wenn einmal die Übersetzung in ein bestimmtes Format vorhanden ist, mit relativ geringem Aufwand Übersetzungen in andere Formate hinzugefügt werden.

5.4 Diagramme

In Geschäftsberichten, Jahresberichten, Marktanalysen, Erhebungen und noch vielen anderen Dokumenten ähnlicher Art ist es erforderlich, Diagramme verschiedenster Art (wie z.B. Kurvendiagramme, Tortendiagramme, Blockdiagramme, ...) ins gewünschte Enddokument einzufügen. Die Ausgangsbasis stellen hier eine bestimmte Menge an Daten dar, aus welchen dann ein Diagramm beliebiger Art erstellt werden soll, um dieses anschließend in das Dokument einzubinden.

Bei diesem Framework gibt es nun dahingehend eine wesentliche Erleichterung, dass es nur notwendig ist, in das Sourcedokument die gesammelten Daten für das Diagramm einzufügen, die gewünschte Art des Diagramms anzugeben und die dafür außer den Daten noch notwendigen Angaben zu machen. Beim Übesetzen in das gewünschte Zielformat wird dann automatisch das erforderliche Diagramm erstellt und in das Enddokument eingebunden. Diese komfortable Vorgehensweise erhielt man in diesem Beispiel durch das Einbinden des `Chart2D`-Packages, [30] mit welchem auf einfache Weise aus einer Reihe von Daten verschiedene Arten von Diagrammen erstellt werden können. Solche vom `Chart2D`-Package erstellten Graphiken werden dann im PNG (Portable Network Graphics)-Format in Files gespeichert und diese können dann recht komfortabel in den Output eingebunden werden.

Der Dateninput kann beispielsweise so aussehen:

```
<chart>
  <title>Quartalsumsätze 2001</title>

  <list chart="pie">
    <title>Quartalsumsätze</title>
    <item caption="1. Quartal">127</item>
    <item caption="2. Quartal">158</item>
    <item caption="3. Quartal">113</item>
    <item caption="4. Quartal">147</item>
  </list>
</chart>
```

Und dementsprechend wird dann eine Graphik wie in Abbildung 5.1 in das Enddokument eingebunden.

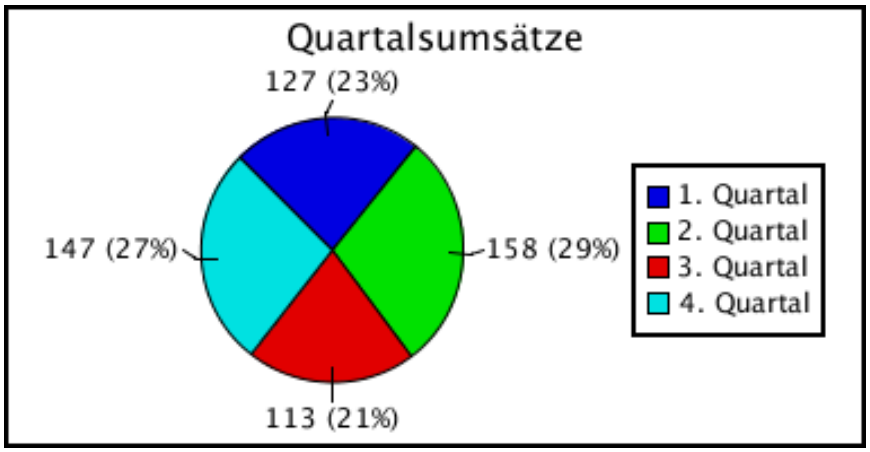

Abbildung 5.1: Beispiel: generiertes Tortendiagramm

Kapitel 6

Abschließende Betrachtungen

Es sollen nun noch einmal die Möglichkeiten und Vorteile des implementierten Frameworks zusammengefasst und aufgezeigt werden, welche Erweiterungsmöglichkeiten es bietet und an welchen Punkte dazu angesetzt werden kann.

6.1 Zusammenfassung und Diskussion

Die vorhandenen Technologien zur Verarbeitung von XML-Dokumenten, vorallem zur Erstellung neuer (druckbarer) Dokumente, bieten meist nicht den Funktionsumfang der in der heutigen Zeit zur professionellen Verarbeitung benötigt wird.

Es wurde versucht, ein Framework zu designen und zu implementieren, welches zur allgemeinen Verarbeitung von XML-Dokumenten verwendet werden kann und leicht erweiterbar ist.

Was sind nun im Einzelnen die Vorteile des implementierten Frameworks?

- *Die Verwendung von* XML *als Format für die Inputdaten* - Damit wird sichergestellt, dass in den Inputdaten ausschließlich Information und keine Formatierung oder dergleichen enthalten ist. Trotzdem ist dies eine für Menschen lesbare Repräsentation von Daten. Zudem wird mit der geforderten Verwendung von XML-Schematas auch die Validierung der Inputdaten sichergestellt.
- *Strikte Trennung von Inhalt, Layout und Logik* - Durch das Design des Framework werden diese drei Schichten strikt voneinander getrennt.

Dies ist einer der Hauptvorteile gegenüber Servlets, JSP (Java Server Pages) und ähnlichem.

- *Die Möglichkeit das selbe Dokument in verschiedene Formate zu transformieren* - Der Input ist vollkommen unabhängig davon, in welches Format man ihn übersetzen will. Ist jedoch einmal die Transformation in ein bestimmtes Zielformat vorhanden, so kann mit relativ geringem Aufwand auf weitere Formate erweitert werden. Dies funktioniert deshalb, weil die gesamte Ablaufsteuerung die selbe bleiben kann und nur jene Transitionen ausgetauscht werden müssen, welche effektiv Output in der Zielsprache erzeugen.

- *Beliebige Erweiterungsmöglichkeit mittels wiederverwendbarer Packages* - Das vorhandene Framework kann sehr einfach um eine weitere Dokumentklasse oder um ein weiteres Zielformat erweitert werden, indem man ein weiteres Package mit den dafür notwendigen Transitionen entwickelt, welches danach immer wieder verwendet werden kann.

- *Ermittlung neuer Daten aus den Inputdaten* - Es können nicht nur die Daten aus den Inputdateien einfach in den Output übernommen werden, sondern es können daraus auch neue Daten generiert werden, welche dann in den Output einfließen. Hierbei sind unter anderem mathematische Berechnungen, umfangreiche Statistiken und dergleichen mehr möglich.

- *Verwenden und Einbinden vorhandener Packages* - Es können sämtliche für beliebige Zwecke vorhandenen Java-Packages in das Framework (respektive in den Transitionen eines Erweiterungspackages zum Framework) eingebunden werden. Damit werden Funktionalitäten wie zum Beispiel das Erstellen von Diagrammen aus Rohdaten mit Hilfe des Chart2D-Packages und beliebig anderes mehr möglich.

- *Keine Beschränkung auf reine Dokumente als Output* - Das Framework kann nicht nur die Transformation von XML-Dokumenten in andere Dokumentenformate bewerkstelligen, sondern es ist noch weit mehr an Funktionalität damit erzielbar. Ein konkretes Beispiel hierfür ist bereits im Framework selbst inkludiert: das Aufsetzten einer neuen Statemachine.

6.2 Ausblick

Es gibt nun allerdings noch einige Punkte, an denen man ansetzen kann, um das vorhandene Framework zu erweitern und zu verbessern.

- *Entwicklung eines graphischen Interfaces* - Die derzeitige `main` Methode ersetzend, welche nur eine Eingabe auf der Kommandozeile zuläßt, kann ein GUI (Graphical User Interface) als Erweiterung zum bestehenden Framework implementiert werden, um das Arbeiten mit demselben komfortabler zu gestalten. Es ist zu diesem Zwecke allerdings nicht mehr erforderlich, den vorhandenen Systemkern zu verändern.

- *Automatisches Generieren der Konfigurationsdaten einer Statemachine aus dem Schema der zu verarbeitenden Dokumentklasse* - Derzeit müssen sowohl das Schema einer zu verarbeitenden Dokumentklasse als auch das dazugehörige Konfigurationsfile zum Aufbauen der dafür notwendigen Statemachine separat und von Hand geschrieben werden. Eine Technik wie aus dem Schema automatisch die Konfiguration der Statemachine ermittelt werden kann wurde noch nicht entwickelt.

- *Anbindung von Datenbanken an Dokumente* - Es sollte eine Möglichkeit geschaffen werden, dass beim Übersetzen spezieller Sourcedokumente auch die Verbindung zu Datenbanken geschaffen wird, aus welchen dann Daten für das gewünschte Enddokument extrahiert werden können.

- *Erstellen und Verwalten von Bibliographien und Literaturverzeichnissen* - Zu großen wissenschaftlichen Arbeiten ist es üblich, Bibliographien und Literaturverzeichnisse anzugeben. In LaTeX existiert hierzu ein recht komfortables Packages: *BibTeX*. Eine ähnliche Funktionalität könnte dem Framework hinzugefügt werden. Aus einer separaten Quelle sollen Daten für Literaturverzeichnis und Bibliographie zum Sourcedokument gewonnen werden und ins Enddokument eingebunden werden können. Beim Übersetzen nach LaTeX könnte man dann wieder auf das *BibTeX* -Package zurückgreifen.

- *Templates für Web-Formulare* - Ein gängiges Problem in der Webseitengestaltung bzw. Programmierung ist das Verarbeiten von Daten, die vom User eingegeben werden. Für dieses Problem sollte ein genereller Ansatz geschaffen werden, da die Abläufe immer dieselben sind und sich nur die Repräsentation und die verwendeten Daten ändern. Das Framework sollte nun dahingehend verwendbar sein, dass es für die jeweiligen Projekte bzw. Webseiten nur noch eine Datenbeschreibung in einer geeigneten Form benötigt. Der Grafiker kann dazu parallel Templates entwerfen, in die per Platzhalter die Datenfelder eingefügt werden können.
  Eine geeignete Lösung wäre dafür, die Datenfelder in einem geeigneten Format (XML bzw. XML Schema bieten sich an) zu beschreiben und die Erzeugung der eigentlichen Web Formulare passiert automatisiert.

Es kann dies eine Umwandlung von XML in PHP sein; Es wäre jedoch leicht möglich, auch z. B. JSP oder ASP damit zu erzeugen.

- *Erweiterung des Frameworks zu wiederverwendbarer Middleware* - Es kann das vorhandene Framework soweit erweitert werden, dass es im Zusammenhang mit verschiedenen Frontends und Backends Verwendung findet. Das bedeutet, dass man es zu Middleware erweitert.

- *Einsatz des Frameworks als Servlet* - Es wäre möglich, das vorhandene Framework dahingehend zu erweitern, dass es als Web-Applikation auf einem Servlet-Server (z.B. Apache) läuft und somit die XML Dokumente im Internet zu Verfügung stellt. Zusätzlich dazu, dass diese Dokumente immer aktuell sind, da sie ja erst beim entsprechenden Request generiert werden, wäre auch eine Auswahlmöglichkeit des gewünschten Dokumentformats (z.B. HTML oder PDF) möglich.

Es besitzt das vorhandene System nun bereits eine vielseitige Funktionalität, kann aber auch noch in viele Richtungen hin erweitert und verbessert werden. Das Design dieses Framework bietet also eine Unmenge an Möglichkeiten.

Literaturverzeichnis

[1] Postscript language reference, third edition. Addison-Wesley. available online: http://partners.adobe.com/asn/developer/pdfs/tn/PLRM.pdf.

[2] Xerces2 java parser 2.0.1. available online http://xml.apache.org/xerces2-j/.

[3] SGML international standard - iso8879, 1986.

[4] The PDF reference manual version 1.3, 3 1999. available online: http://www.pdfzone.com/pdfs/PDFSPEC13.PDF.

[5] Braille international standard - iso11548, 2001. Communication aids for blind persons – Identifiers, names and assignation to coded character sets for 8-dot Braille characters – Part 1: General guidelines for Braille identifiers and shift marks, Part 2: Latin alphabet based character sets.

[6] Mathematical markup language (mathml) version 2.0, 2 2001. available online: http://www.w3.org/TR/MathML2.

[7] Fop (formatting objects processor), 2002. available online http://xml.apache.org/fop.

[8] How to create channel definition format (cdf) files, 2002. available online http://msdn.microsoft.com/workshop/delivery/cdf/tutorials/generic.asp.

[9] Mozilla 1.0, June 2002. available online http://www.mozilla.org/.

[10] S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, and S. Zilles. Extensible stylesheet language (xsl), version 1.0, October 2001. available online http://www.w3.org/TR/2001/REC-xsl-20011015/slice6.html#fo-section.

[11] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext Transfer Protocol — HTTP/1.0, May 1996. Status: INFORMATIONAL.

[12] P. Biron and A. M. (Editors). Xml schema part 2: Datatypes. Technical report, World Wide Web Consortium (W3C), May 2001. available online `http://www.w3.org/TR/xmlschema-2/`.

[13] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. M. (Editors). Extensible markup language (xml) 1.0. Technical report, World Wide Web Consortium (W3C), October 2000. available online `http://www.w3.org/TR/REC-xml`.

[14] P. Ciancarini, F. Vitali, and C. Mascolo. Managing complex documents over the WWW: a case study for XML. *IEEE-Transactions-on-Knowledge-and-Data-Engineering*, 11(4):629–38, July-Aug 1999.

[15] J. Clark and S. D. (Editors). Xml path language (xpath), November 1999. available online `http://www.w3.org/TR/xpath`.

[16] J. Clark and B. Lindsey. Xt, a fast, free implementation of xslt in java, April 2002. available online `http://www.blnz.com/xt/index.html`.

[17] C. Dallermassl, K. Schmaranz, and H. Krottmaier. Xpg concept, September 2001. available online `http://courses.iicm.edu/xpg/docs/xpg_concept.pdf/`.

[18] R. M. David Carlisle, Patrick Ion and N. P. (Editors). Mathematical markup language (mathml) version 2.0, February 2001. available online `http://www.w3.org/TR/MathML2/`.

[19] J. F. (Editors). Scalable vector graphics (svg) 1.0 specification, September 2001. available online `http://www.w3.org/TR/SVG`.

[20] M. K. (Editors). Xsl transformations (xslt), version 2.0, April 2002. available online `http://www.w3.org/TR/xslt20`.

[21] P. H. (Editors). Synchronized multimedia integration language (smil) 1.0 specification, April 1998. available online `http://www.w3.org/TR/1998/PR-smil-19980409`.

[22] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. RFC 2068: Hypertext Transfer Protocol — HTTP/1.1, Jan. 1997. Status: PROPOSED STANDARD.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Longman, Inc., 1998. ISBN 0-201-63361-2.

[24] A. Kaplan and J. Lunn. FlexXML: engineering a more flexible and adaptable web. In C. U. Dept. of Comput. Sci., editor, *Proceedings International Conference on Information Technology: Coding and Computing.*, volume xiv+698, pages 405–10, SC, USA, 2001. IEEE Comput. Soc, Los Alamitos. Proceedings International Conference on Information Technology: Coding and Computing. 2-4 April 2001; Las Vegas, NV, USA. Sponsored by: IEEE Comput. Soc.

[25] R. Kelsey, W. Clinger, and J. R. (Editors). Revised(5) report on the algorithmic language scheme. available online: `http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html`.

[26] S. Krishnamurthi, K. E. Gray, and P. T.Graunke. Transformation-by-example for XML. In R. U. Dept. of Comput. Sci., editor, *Practical Aspects of Declarative Languages. Second International Workshop, PADL 2000. Proceedings (Lecture Notes in Computer Science Vol.1753)*, volume 1753, pages 249–62, Houston, TX, USA, 2000. Springer-Verlag, Berlin, Germany. Proceedings of 2nd Workshop on the Practical Aspects on Declarative Languages 2000. 17-18 Jan. 2000; Boston, MA, USA.

[27] D. Megginsion. Simple api for xml, May 2000. available online `http://www.saxproject.org/`.

[28] P. Merrick and C. Allen. Web interface definition language (widl), September 1997. available online `http://www.w3.org/TR/NOTE-widl`.

[29] K. Schmaranz. Ak-softwareentwicklung - programming for large libraries. online, 03 2001. available online `http://courses.iicm.edu/ak_swent/AKSWE_Skriptum.pdf`.

[30] J. Simas. Chart2d - java package, January 2002. available online `http://sourceforge.net/projects/chart2d`.

[31] E. M. Steve DeRose and D. O. (Editors). Xml linking language (xlink) version 1.0, June 2001. available online `http://www.w3.org/TR/xlink`.

[32] E. M. Steve DeRose and R. D. J. (Editors). Xml pointer language (xpointer) version 1.0, January 2001. available online `http://www.w3.org/TR/WD-xptr`.

[33] A. L. E. Tim Bray, Dave Hollander. Namespaces in xml, January 1999. available online `http://www.w3.org/TR/REC-xml-names`.

Anhang A

Schema der Statemachine

```
<?xml version="1.0" standalone="no"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!-- attention: all names must be adapted to the
                following nameing convention
     type names    : AllWordsCapitalizedWithoutUnderscores
     element names : alllowercasewithoutunderscores
-->

<xsd:element name="statemachine">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="path" type="xsd:string"
minOccurs="1" maxOccurs="unbounded"/>
      <xsd:element name="states" type="States"
minOccurs="1" maxOccurs="1"/>
      <xsd:element name="transitions" type="Transitions"
minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="States">
  <xsd:sequence>
    <xsd:element name="startstate" type="xsd:string"
minOccurs="1" maxOccurs="1"/>
    <xsd:element name="state" type="xsd:string"
minOccurs="1" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Transitions">
  <xsd:sequence>
    <xsd:element name="transition" type="Transition"
minOccurs="1" maxOccurs="unbounded"/>
```

```
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Transition">
  <xsd:sequence>
    <xsd:element name="beginstate" type="xsd:string"
minOccurs="1" maxOccurs="1"/>
    <xsd:element name="nextstate" type="xsd:string"
minOccurs="1" maxOccurs="1"/>
    <xsd:element name="element" minOccurs="0" maxOccurs="1">
       <xsd:complexType>
          <xsd:simpleContent>
            <xsd:extension base="xsd:string">
              <xsd:attribute name="type" type="ElementAttributes"
use="required"/>
            </xsd:extension>
          </xsd:simpleContent>
       </xsd:complexType>
    </xsd:element>
    <xsd:element name="classname" type="xsd:string"
minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="ElementAttributes">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="start"/>
    <xsd:enumeration value="end"/>
    <xsd:enumeration value="enddoc"/>
  </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```