Erstellung eines Programms zur Verwaltung und Generierung von Dokumentvorlagen für XSL- und LaTeX-Stylesheets

Bachelor-Thesis im Studiengang Technische Redaktion
Fakultät für Wirtschaftswissenschaften
Hochschule Karlsruhe - Technik und Wirtschaft (HS)

Ersteller	Alexander Fischer Matrikelnummer 20652 Grenzstr. 10 76227 Karlsruhe alexanderfischer@o2online.de (+49) 0176 / 235 570 31
Betreuer	Dipl.-Phys. Heinz Pommer bitplant.de GmbH & Co. IT-Services KG Fabrikstr. 15 89520 Heidenheim info@bitplant.de (+49) 0732 / 660 345
Erstprüfer der Hochschule	Prof.-Dr. Wolfgang Ziegler
Zweitprüfer der Hochschule	Prof. Sissi Closs
Tag der Anmeldung	25. August 2008
Tag der Abgabe	25. November 2008

Hiermit erkläre ich Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und bisher nicht veröffentlicht.

Karlsruhe, 25. November 2008

Inhaltsverzeichnis

1 Einführung **5**

2 Ausgangssituation **7**
2.1 Einführung 7
2.2 Einordnung verschiedener ausgewählter Publikationsprozesse 7
2.3 Ursprünge von TeXML 11
2.4 TEX und LATEX 11
2.5 TeXML-Publikationsprozesses 12
2.6 LATEX-Stile 14
2.7 Memory Stylesheets 16
2.8 Schlussfolgerung 18

3 Zielsetzung **21**
3.1 Einführung 21
3.2 Lasten und Vorüberlegungen 21
3.3 Integration in den Publikationsprozess 23

4 Dokumentvorlagen – TemplateXML **27**
4.1 Einführung 27
4.2 Namensraum 29
4.3 Elemente und Attribute von TemplateXML 30
4.3.1 Das Wurzelelement 30
4.3.2 Element designer 31
4.3.3 Element template 31
4.3.4 Element page 32
4.3.5 Element frame 33
4.3.6 Element content 33
4.3.7 Element parameter 36
4.3.8 Element description 36
4.3.9 Element dimension 36
4.3.10 Element position 37
4.3.11 Vorkommen und Verwendung der Elemente dimension und paper 38
4.3.12 Element paper 38
4.4 Schlussfolgerung 40

5 Template-Designer **43**
5.1 Einführung . . . 43
5.2 Umsetzung . . . 43
5.2.1 Begrüßungsdialog . . . 43
5.2.2 Hauptfenster . . . 44
5.2.3 Baumstruktur für Navigation . . . 45
5.2.4 Einstellungen zum jeweiligen Element . . . 47
5.2.5 Einstellungen für Dokumentvorlagen . . . 49
5.2.6 Einstellungen für Dokumentabschnitte . . . 50
5.2.7 Einstellungen für Elemente zur Seitenausgestaltung . . . 50
5.2.8 Einstellungsdialog . . . 54
5.2.9 Menüs und Werkzeugleisten . . . 54
5.3 Lizenz und Gedanken zu OpenSource . . . 55
5.4 Genutzte Drittanbieter-Software . . . 59
5.4.1 Python . . . 59
5.4.2 WxPython . . . 60
5.4.3 Ghostscript . . . 61
5.4.4 Python Image Library . . . 61
5.5 Integrierte Drittanbieter-Software . . . 62
5.5.1 Oxygen . . . 62
5.5.2 Keychain.py . . . 62
5.6 Verwendete Entwicklungswerkzeuge . . . 63
5.6.1 Eclipse . . . 63
5.6.2 Pydev . . . 63
5.6.3 Subclipse . . . 64
5.6.4 Web Tools Platform . . . 65
5.7 Schlussfolgerung . . . 65

6 Besondere Aspekte bei der Umsetzung des Template-Designers **67**
6.1 Einführung . . . 67
6.2 Bitplant-TemplateXML validieren . . . 67
6.2.1 Einführung . . . 67
6.2.2 Document Type Definition . . . 68
6.2.3 XML Schema 1.0 . . . 68
6.2.4 XSLT . . . 70
6.2.5 Schematron . . . 72
6.2.6 XML Schema 1.1 . . . 74
6.2.7 Python mit ElementTree . . . 75
6.2.8 Schlussfolgerung . . . 77
6.3 Dokumentation der Quelltexte . . . 78
6.3.1 Einführung . . . 78
6.3.2 System zur Verwaltung von Qelltext-Dokumentation . . . 79
6.3.3 Verknüpfung von Quelltext und Dokumentation . . . 80

6.3.4 Docstrings 82
6.3.4.1 Innere Struktur von Docstrings 83
6.3.4.2 PlainText 84
6.3.4.3 reStructuredText 86
6.3.4.4 Abschnitte 86
6.3.4.5 Absätze 87
6.3.4.6 Listen 88
6.3.4.7 Querverweise 90
6.3.5 Schlussfolgerung 90

7 Anhang **91**
7.1 Quelltexte 91
Literatur 99

1 Einführung

Im Bereich der Technischen Dokumentation begann sich in den letzten Jahren zunehmend und in Kombination mit Content-Management-Systemen (CMS) die Metasprache XML (eXtensible Markup Language) durchzusetzen. XML wurde vom World Wide Web Consortium (W3C [95]) im Jahr 1998 als Nachfolger von SGML entwickelt. Es sollte ursprünglich dazu dienen, Web-Inhalte auszutauschen und als eine standardisierte Schnittstelle zwischen verschiedensten weiteren Dokumentformaten fungieren.

Die besonderen Vorteile von XML sind der verhältnismäßig einfache Aufbau, seine enorme Flexibilität bei gleichzeitig unbedingter Treue zu einer vorgegebenen Struktur, und seine Layoutunabhängigkeit. XML gibt als Metasprache Regeln vor, um auf diesen Regeln aufbauend eigene XML-Dialekte zu entwickeln, die hochgradig strukturiert und dennoch flexibel an die eigenen Bedürfnisse angepasst sein können. Bei der Definierung eines XML-Dialektes werden Inhalten keine Layoutinformationen zugeordnet. Statt dessen werden Inhalte nach semantischen Gesichtspunkten strukturiert. Eine Überschrift wird beispielsweise als Überschrift gekennzeichnet, ein Autor als Autor.

Die Strukturierung kann genau so vorgenommen, wie es die Umstände erfordern. Dieser Umstand und die Möglichkeit Strukturen festzuschreiben und damit zu standardisieren haben besonders in der Technischen Dokumentaion enorme Vorteile. Jeder Redaktion ist es mit XML möglich, die eigenen Richtlinien zur Erzeugung der Technischen Dokumentation zu standardisieren. Mit XML kann automatisiert überprüft werden, ob diese Richtlinien auch eingehalten werden. Wegen der Layoutunabhängigkeit ist es zudem möglich Inhalte so aufzubereiten, dass diese wiederholt und in verschiedensten Ausgabeformaten verwendet werden können (Single-Source-Publishing [16]).

Dass XML Informationen layoutunabhängigkeit speichert, hat aber auch einen weiteren Schritt bei der Publikation von Technischer Dokumentation notwendig gemacht: den mit XML strukturierten Inhalte müssen vor der Publikation Layoutinformationen hinzugefügt werden. Für diesen Vorgang haben sich in den letzten Jahren verschiedenste Lösungen entwickelt, die sich entweder damit beschäftigen den XML-Strukturen Layoutinformationen beizuordnen (Adobe FrameMaker) oder die gegeben XML-Strukturen in ein Format zu konvertieren, dass die Möglichkeit bietet, die Inhalte anhand ihrer ursprünglichen Strukturierung mit Layoutinformationen zu versorgen (XSLT, XSL-FO, TeXML). Beide Lösungswege bedingen tiefgehende Kenntnisse bezüglich der Steuerung des Publikationsprozesses.

Diese Situation stellt besonders für kleinere Technische Redaktionen und Redaktionen, die nur geringe oder keinerlei Erfahrungen im Umgang mit XML haben, eine hohe Einstiegshürde dar.

Diese Thesis beschäftigt sich mit dem TeXML-Publikationsprozess als einer Möglichkeit zur Erzeugung von druckfertigen Dokumenten aus XML-Daten. Im Rahmen der Thesis wurde für den Publikationsprozess mit TeXML bzw. LaTeX und XSL-FO ein Programm zur Fertigung von Dokumentvorlagen erstellt. Diese Dokumentvorlagen sollen es ermöglichen, ohne tiefere Kenntnisse des Publikationsprozesses einfache Änderungen am Layout eines Dokumentes durchzuführen, um so die Hürde im Umgang mit den genannten Publikationslösungen zu senken. Die Thesis beschreibt ferner das erstellte Programm und befasst sich dabei mit Details und Überlegungen, die bei der Erstellung von Relevanz waren und für die Technische Dokumentation nach Meinung des Autors insgesamt von Interesse sind.

Besonderer Dank gilt der Firma bitplant.de GmbH & Co. IT-Services KG, die diese Thesis durch die Bereitstellung von Wissen, Zeit und Ressourcen ermöglichte sowie allen Entwicklern von OpenSource-Software. Ohne diese wäre die Thesis niemals möglich gewesen.

2 Ausgangssituation

2.1 Einführung

Die Firma bitplant.de GmbH & Co. IT-Services KG ist ein Dienstleister für Technische Dokumentation. Im Auftrag seiner Kunden wird deren Technische Dokumentation erstellt und gepflegt, wobei insbesondere XML-basierte Lösungen zum Einsatz kommen. Aufgrund des in der Technischen Dokumentation steten Drucks zu Innovation und Prozessoptimierung wurde deutlich, dass die bisher zur Verfügung stehenden Publikationsprozesse optimierbar waren und die Notwendigkeit bestand, eine Lösung zu entwickeln, die besser auf die Bedürfnisse der Technischen Dokumentation zugeschnitten ist. Der vorläufige Höhepunkt dieser Entwicklungsanstrengungen ist der TeXML-Publishing-Server. Nachfolgend wird diese Entwicklung technisch eingeordnet, genauer beschrieben und bei Bedarf mit anderen Lösungen, insbesondere XSL-FO verglichen.

2.2 Einordnung verschiedener ausgewählter Publikationsprozesse

Allen Methoden zur Publikation von XML-Daten ist, wie bereits erwähnt, die Problematik gemein, dass diese ein hohes Maß an Vorarbeit benötigen, um den gegebenen XML-Strukturen ein ansprechendes Aussehen zu geben. Vorarbeiten bestehen entweder darin, XML-Strukturen Layoutinformationen beizufügen oder so genannte Stylesheets zu programmieren. Stylesheets wandeln XML-Daten in Formate um, die mit Layoutinformationen versehen sind. Jede Publikationsmethode bietet individuelle Vor- und Nachteile, die im Folgenden beschrieben werden, um einen Überblick über die gegenwärtige Situation zu gewinnen.

Als erste und sehr populäre Publikationsmethode sei die Verwendung des Werkzeugs Adobe FrameMaker [51] genannt. Bei der Verwendung von FrameMaker zur Publikation von XML-Daten wird eine so genannte *Element Document Definition* (EDD) genutzt. Eine EDD ist ein von FrameMaker interpretierbarer Regelsatz, der aus einer *Document Type Definition* (DTD) eines XML-Dialektes heraus generiert wird und um von FrameMaker interpretierbare Layoutinformationen angereichert wird. Durch eine EDD ist es FrameMaker möglich beim Verfassen von Inhalten XML-ähnliche Strukturen zu

erzwingen und diese beim Speichern in den ursprünglichen XML-Dialekt zu übertragen. Gleichzeitg sieht der Technische Redakteur beim Verfassen der Inhalte deren Formatierung (*What You See Is What You Get*-Prinzip) und kann aus den Inhalten direkt und ohne zusätzliche Schritte hochwertige Drucksachen publizieren. Ist FrameMaker der Dialekt eines XML-Dokumentes durch das Vorhandensein einer passenden EDD bekannt, kann FrameMaker die XML-Daten direkt mit den zugeordneten Layoutinformationen darstellen. Adobe FrameMaker ermöglicht es mittels der EDD auf intuitive Weise, XML-Daten mit einem hochwertigen Layout zu produzieren.

An seine derzeitigen Grenzen stößt FrameMaker wenn es um die Publikation von Inhalten in Schriftsprachen geht, die von Rechts nach Links gelesen werden. Dies ist besonders für Technische Dokumentation relevant, die im asiatisch- und arabischsprachigen Raum Verbreitung finden soll.

Ein weiteres Problem stellt die derzeit übliche Zusammenstellung von XML-Daten aus Content-Management-Systemen heraus und ihre anschließende Publikation mit FrameMaker dar. Da XML-Daten selbst keine Layoutinformationen tragen, müssen Korrekturen am Layout, z.B. ungünstige Zeilen- oder Seitenumbrüche, bei erneuter Publikation aus dem genutzten Content-Management-System über FrameMaker immer wieder neu korregiert werden. Dies ist bei sehr umfangreicher Technischer Dokumentation mit einem hohen Zeit- und Kostenaufwand verbunden. Es besteht zwar prinzipiell die Möglichkeit, erforderliche Layoutkorrekturen bereits in den XML-Ausgangsdaten zu vermerken, so dies denn der verwendete XML-Dialekt gestattet. Jedoch gibt man damit die Layoutunabhängigkeit von XML, einer der zentralen Vorteile, auf. Außerdem läuft man Gefahr bei einer Änderung des gewünchten Ziellayouts alle im ursprünglichen Layout getätigten Änderungen korrigieren zu müssen.

Der Publikationsprozess von XML-Daten über eine EDD zum druckfertigen Dokument mittels Adobe FrameMaker stellt eine praktikable Lösung dar. Schwächen zeigt Adobe FrameMaker jedoch dann, wenn die Publikationsprozesse in einem hohen Maß automatisiert werden sollen und es um die Publikation in exotische Zielsprachen geht.

Eine weitere Publikationsmethode ist die Verwendung der vom World Wide Web Consortium (W3C) entworfenen eXtensible Stylesheet Language – Formatting Objects (XSL-FO). Diese ist verwandt mit der eXtensible Stylesheet Language – Transformation (XSLT) und selbst ein XML-Dialekt [89].

Bei dieser Publikationsmethode werden XML-Daten unter Zuhilfenahme von XSLT nach XSL-FO transformiert. Anschließend wird das XSL-FO-Dokument wiederum in ein druckfertiges Dokumentenformat konvertiert [73, Chapter 2 - A First Look at XSL-FO]. Die gewünschten Layouteigenschaften sind bei diesem Vorgang als Attribute im XSL-FO-Dokument hinterlegt, müssen also bereits dem XSLT-Stylesheet bekannt sein. XSL-FO eignet sich sehr gut für automatisierte Publikationsprozesse, weil sich die Transformation der XML-Daten skriptgesteuert, und von einem eventuell genutzten Content-Management-System angestoßen, abwickeln lässt. Wie bei FrameMaker ist eine nach-

trägliche Layoutkorrektur verhältnismäßig aufwendig, weil Änderungen des Layouts im XSL-FO Dokument durchgeführt werden müssen. Inhaltliche Korrekturen und Fortentwicklungen an den ursprünglichen XML-Daten bewirken, dass der Transformationsprozess über XSLT nach XSL-FO erneut notwendig wird, was auch erneute Korrekturen des Layouts bedingt.

Weil XSL-FO lediglich eine Spezifikation und keine Implementierung darstellt, wie es beispielsweise bei Adobe FrameMaker und LaTeX der Fall ist, existieren verschiedene Implementierungen verschiedener Anbieter [73, Appendix C, Today's Tools]. Dieser Zustand ist mit jenem zu vergleichen, der bei Web-Browsern vorzufinden ist. Weil es von Seiten des World Wide Web Consortium (W3C) keine Referenzimplementierung der Spezifikation gibt, muss man sich auf eine Implementierung beschränken, wenn man stets die selbe drucksetzerische Qualität erreichen will [50]. Dadurch ist man in der Regel auf die durch den gewählten Anbieter unterstützte Plattform beschränkt, was dem ursprünglichen Gedanken, XML als plattformunabhängiges Format zu nutzen, zuwiderläuft. Ob die derzeitige drucksetzerische Qualität der verfügbaren XSL-FO-Prozessoren insgesamt mit jener von Adobe FrameMaker oder LaTeX mithalten kann, ist fraglich. Dies ist vor allem deshalb gegeben, weil sowohl Adobe FrameMaker als auch und insbesondere LaTeX einen wesentlich längeren Entwicklungsprozess durchlaufen haben und gezielt darauf hin entwickelt wurden *gut auszusehen*.

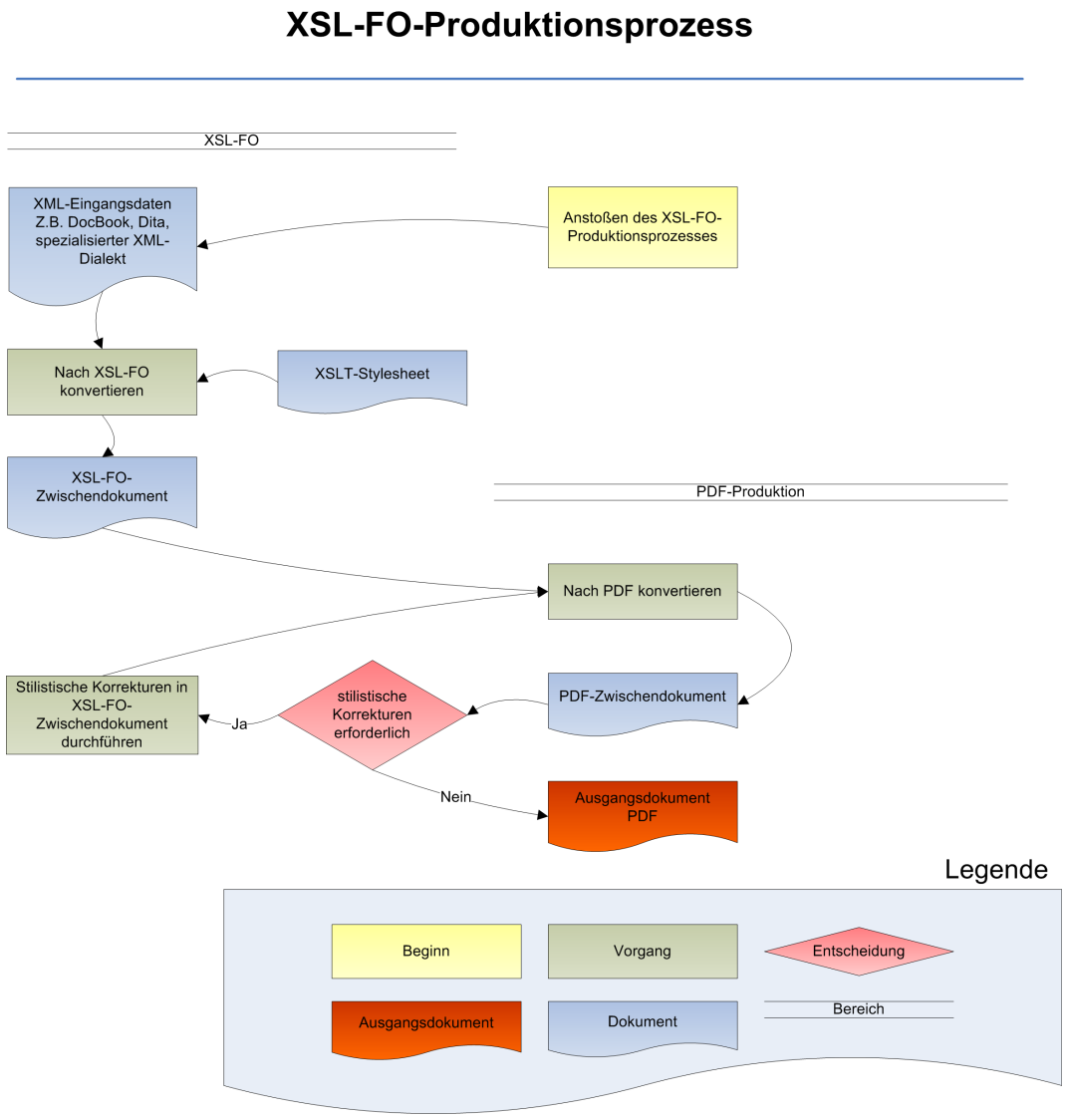

Abbildung 2.1: Überblick zum XSL-FO-Publikationsprozess

2.3 Ursprünge von TeXML

Aufgrund der vorig beschriebenen Defizite der Publikationsprozesse mit FrameMaker und XSL-FO griff die Firma Bitplant auf eine ursprünglich von IBM stammende Idee [59] zurück, die Ihre erste Implementierung im mittlerweile nicht mehr weiterentwickelten XML-Prozessor TeXMLatte [78] fand. Bei diesem Publikationsprozess geht es darum, XML-Daten mittels des LaTeX-Textsatzsystems [82] in ein druckfertiges Dokumentenformat zu konvertieren. Dabei werden die Ausgangsdaten mittels XSLT in einem ersten Schritt in den XML-Dialekt TeXML zu transfomiert, welcher anschließend in einem zweiten Schritt so transformiert wird, dass das Ergebnis vom LaTeX-Textsatzsystem in ein druckfertiges Dokumentenformat transformiert werden kann. In diesem zweistufigen Vorgang unterscheidet sich der Publikationsprozess mit TeXML somit nicht von dem mit XSL-FO.

2.4 TeX und LaTeX

TeX ist ein ursprünglich von Donald Ervin Knuth [1] entwickeltes Textsatzsystem, dessen Entwicklung bereits im Mai 1977 begann [83, 1.1 Ein kurzer Blick in die Vergangenheit]. Ziel der Entwicklung war es, ein Textsatzsystem zu entwickeln, dass alle bis dato verfügbaren Textprozessoren hinsichtlich der drucksetzerischen Qualität übertreffen sollte [57, With TeX the goal is to produce the *finest* quality]. Dabei setzt TeX auf das *What you see is what you mean*-Prinzip. Das bedeutet, dass der Anwender – im Gegensatz zu neueren Textprozessoren wie beispielsweise Adobe FrameMaker oder Microsoft Word – keine formatierte Ausgabe der zu setzenden Texte vor sich hat, sondern statt dessen mit einfachen Textdateien arbeitet. In diesen Dateien sind Befehle notiert, die dem Textsatzsystem vorgeben, wie dieser Inhalte nach der Umwandlung in ein Dokumentenformat mit Layouteigenschaften darstellen soll. Mit diesem Prinzip nahm TeX zu einem gewissen Grad die Entwicklung von HTML, CSS und XML vorweg. Schon bei der Entwicklung von TeX gab es das ausdrückliche Ziel, dass sich der Anwender mit den Inhalten und nicht mit deren Formatierung auseinandersetzen sollte [57, Yet you won't need to do much more work than would be involved if you were simply typing the manuscript on a ordinary typewriter]. Wenn auch dieser Schritt nicht so konsequent umgesetzt wurde, wie etwa bei XML. TeX fehlen insbesondere die Fähigkeiten von XML, eine bestimmte Struktur zu erzwingen.

TeX besteht heute aus einem stabilen Kern, der circa 300 Grundbefehle bereit hält und seit 1990 nicht mehr fortentwickelt, sondern nur noch gepflegt wird. Grund dessen war, dass zusätzliche Erweiterungen von TeX nach Ansicht der Entwickler nur dessen

[1] Emeritierter Professor für Informatik an der Stanford University, `http://www-cs-faculty.stanford.edu/~knuth/`

Stabilität gefährdet hätten und eine Optimierung der drucksetzerischen Fähigkeiten von TEX nicht mehr notwendig sei [2].

Da TEX selbst nicht mehr um neue Funktionen erweitert wird und dessen Bedienung nicht trivial ist, kam es ab dem Jahr 1985 zur Entwicklung von LATEX. LATEX stellt Befehle und Erweiterungen bereit, die auf den TEX-Kern zugreifen und die Bedienung des TEX-Textsatzsystems wesentlich vereinfachen sollten. Ab 1994 bestand zudem die Möglichkeit LATEX um eigene Entwicklungen in Form von sogenannten Paketen zu erweitern. Seitdem wird ständig von einer besonders aus dem universitären Umfeld stammenden Entwicklergemeinde an deren Pflege und Erweiterung gearbeitet. Heute sind als Vorteile des LATEX-Textsatzsystems besonders dessen Robustheit und Geschwindigkeit zu nennen, sowie der qualitätiv sehr hochwertige Schriftsatz mit der Möglichkeit Schriftglyphen mit einer Genauigkeit von weniger als 0,01 Mikrometern zu positionieren. Hinzukommt die Möglichkeit des sehr präzisen mathematischen Formelsatzes, der LATEX besonders im mathemathisch-naturwissenschaftlichen Bereich zum deFacto-Standard für das Setzen von Formeln macht. Das Textsatzsystem selbst ist plattformunabhängig, im Quelltext ohne Lizenzkosten frei verfügbar und auf maximale Interoperabilität ausgerichtet. Ziel der Entwicklung war ein auf jeder unterstützten Plattform identischer Textsatz. Weitere Informationen liefern [57, 83].

2.5 TeXML-Publikationsprozesses

Der TeXML-Publikationsprozess beginnt damit, dass XML-Daten mittels eines XSLT-Stylesheets nach TeXML transformiert werden. TeXML ist ein simpler XML-Dialekt, dessen Aufgabe es ist, die syntaktischen Merkmale von TEX abzubilden. TEX bezeichnet hier die Befehle zum Steuern des LATEX-Textsatzsystems und dessen Erweiterungen. Dennoch würde eine vollständige Beschreibung von TeXML den Rahmen der Thesis sprengen. Deshalb sei für eine vollständige Beschreibung von TeXML auf [72] verwiesen. Im Gegensatz zum Publikationsprozess mit XSL-FO ist TeXML nicht dazu gedacht, Layoutinformationen direkt zu tragen. Dies ist zwar möglich, jedoch ist die wesentlich bessere Lösung – änhlich dem Ansatz der Cascading Stylesheets (CSS [92]) aus der (X)HTML-Welt – einen eigenen Befehl zu definieren, dessen Interpretation anschließend vom LATEX-Textsatzsystem vorgenommen wird. An diesem Punkt ist LATEX deshalb interessanter Weise dem Grundgedanken von XML, nämlich Struktur bzw. Semantik und Layout zu trennen, näher als der XML-Dialekt XSL-FO, bei dem Angaben zur Layout und Formatierung und die eigentlichen Inhalte stets zusammen vorzufinden sind.

Ein praktisches Beispiel soll diese Ausführungen verdeutlichen. Listing 2.1 zeigt eine einfache XML-Datei, die mittels des in Listing 7.3 gezeigten XSLT-Stylesheets die in Listing 2.2 gezeigte TeXML-Datei erzeugt.

[2] http://www.tug.org/TUGboat/Articles/tb11-4/tb30knut.pdf

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE task SYSTEM "dtd/im/anhang/input.dtd">
<task language="deutsch" author="fial0015">

<goal>Kuehlschrank mit Lebensmitteln befuellen</goal>

<condition>Sie verfuegen ueber einen Kuehlschrank.</condition>
<condition>Sie verfuegen ueber Lebensmittel ausserhalb des Kuehlschranks
    .</condition>

<action>Oeffnen Sie die Kuehlschranktuer.</action>
<sidegoal>Die Kuehlschranktuer ist geoeffnet.</sidegoal>
<action>Legen Sie ihre Lebensmittel in den Kuehlschrank.</action>
<action>Schliessen Sie die Kuehlschranktuer.</action>
<sidegoal>Die Kuehlschranktuer ist geschlossen.</sidegoal>

<result>Sie haben den Kuehlschrank mit Lebensmitteln befuellt.</result>

</task>
```

Listing 2.1: Ein einfaches Beispiel-XML

```
<?xml version="1.0" encoding="utf-8"?>
<TeXML xmlns="http://getfo.sourceforge.net/texml/ns1">
  <cmd name="documentclass" nl2="1"><parm>dokumentklasse</parm></cmd>
  <cmd name="usepackage" nl2="1"><parm>dokumentstil</parm></cmd>
  <cmd xmlns="" name="title" nl2="1"><parm>Kuehlschrank mit
      Lebensmitteln befuellen</parm></cmd>
  <cmd xmlns="" name="author" nl2="1"><parm>fial0015</parm></cmd>

  <env name="document">
    <cmd xmlns="" name="section">
      <parm>Vorhaben: Kuehlschrank mit Lebensmitteln befuellen</parm>
    </cmd>

    <cmd name="condition">
      <parm>Sie verfuegen ueber einen Kuehlschrank.</parm>
    </cmd>
    <cmd name="condition">
      <parm>Sie verfuegen ueber Lebensmittel
          ausserhalb des Kuehlschranks.</parm>
    </cmd>

    <cmd name="action">
      <parm>Oeffnen Sie die Kuehlschranktuer.</parm>
    </cmd>
    <cmd xmlns="" name="sidegoal">
      <parm>Die Kuehlschranktuer ist geoeffnet.</parm>
    </cmd>
    <cmd name="action">
      <parm>Legen Sie ihre Lebensmittel in den Kuehlschrank.</parm>
    </cmd>
    <cmd name="action">
```

```
32        <parm>Schliessen Sie die Kuehlschranktuer.</parm>
33      </cmd>
34      <cmd name="sidegoal">
35        <parm>Die Kuehlschranktuer ist geschlossen.</parm>
36      </cmd>
37
38      <cmd name="sidegoal">
39        <parm>Sie haben den Kuehlschrank mit Lebensmitteln befuellt.</parm>
40      </cmd>
41
42    </env>
43  </TeXML>
```

Listing 2.2: TeXML-Dokument aus XSL-Transformation

TeXML setzt sich aus Elementen und Attributen zusammen, die grundsätzlich für drei verschiedene Aufgaben verantwortlich zeichnen. Zum Einen dienen manche Elemente (z.B. *cmd, env, opt, parm*) und Attribute (z.B. *name*) dazu, TeX-Strukturen in XML abzubilden. Weiter existieren Attribute (z.B. *nl1, gr*), die dazu dienen, das Aussehen des finalen LaTeX-Dokumentes zu steuern. Dies ist vor allem für eine eventuelle Nachbearbeitung des LaTeX-Dokumentes von hohem praktischen Wert. Und letztlich existieren Elemente (z.B. *pdf*), über die sich die Behandlung von Sonderzeichen und deren Umwandlung in für LaTeX interpretierbare Kodierungen steuern lässt. Damit unterscheidet sich TeXML deutlich von XSL-FO. XSL-FO wurde gezielt dazu entworfen, Layoutinformationen zu tragen und ist entsprechend umfangreich. TeXML dagegen verweist lediglich auf Layoutinformationen, die in LaTeX-Syntax notiert sind. Durch diese Trennung sind XSLT-Stylesheets für die Konvertierung nach TeXML wesentlich besser lesbar und auch kompakter, was deren Wartbarkeit vereinfacht. Grund dessen ist, dass sich der Stylesheet lediglich um die Anordnung der Inhalte und deren Auszeichnung kümmert und nicht direkt um das Layout.

2.6 LaTeX-Stile

Die mit TeXML erzeugten LaTeX-Befehle (in Listing 2.4 z.B. *action, sidegoal, result*) benötigen Stil-Definitionen, damit der LaTeX-Prozessor die Abarbeitung der Befehle erledigen kann. Stil-Definitionen werden in einer Textdatei mit der Dateiendung *sty* festgehalten und geben an, wie Inhalte darzustellen sind. Damit fungieren die mittels TeXML erzeugten LaTeX-Befehle lediglich als Platzhalter und Referenzen für die eigentlichen Stilangaben. Dennoch ist es möglich, auch LaTeX-Befehle zur direkten Formatierung von Inhalten in den für die TeXML-Transformation notwendigen Stylesheet aufzunehmen. Dieses Vorgehen ist jedoch nicht zu empfehlen, weil es den Stylesheet unnötiger Weise verkompliziert. Die eigentlich sinnvolle Trennung zwischen Auszeichnung und Aussehen würde aufgehoben, obwohl dies nicht notwendig ist. Außerdem ist es durch die Notation von Stilen in einer Stildatei möglich, dass Aussehen von zu publizierenden

Dokumenten zu ändern, ohne den XSLT-Stylesheet anfassen zu müssen. Es genügt dann, die entsprechenden LaTeX-Stildefinitionen zu verändern. Würde man das Aussehen von Inhalten direkt steuernde LaTeX-Befehle in den Stylesheet aufnehmen, ginge dieser Vorteil verloren.

Auf besagte Stildateien wird im Kopfbereich eines LaTeX-Dokumentes verwiesen. In Listing 2.4 nennt sich besagte Textdatei *dokumentstil.sty*. Eine ausführliche Syntaxbeschreibung aller möglichen LaTeX-Befehle zu liefern, würde den Rahmen dieser Thesis deutlich sprengen. Dennoch sei eine beispielhafte Erläuterung erlaubt.

```
1 \newcommand{\action}[1]{{\it #1}\parskip2pt\par}}
```

Listing 2.3: Bekanntmachen des Befehls *action* und Zuweisen von Aktionen zur Layoutsteuerung

Listing 2.3 erzeugt einen neuen LaTeX-Befehl mit dem Namen *action*. Die in eckigen Klammern notierte Ziffer *1* dient als Platzhalterzeichen für die eigentlichen Inhalte im LaTeX-Dokument. Diese werden wiederum an jener Stelle eingesetzt, an welcher *#1* notiert ist. Die Befehle *it*, *parskip2pt* und *par* bewirken, dass die eigentlichen Inhalte kursiv dargestellt werden, und zwischen aktuellem Absatz und nachfolgendem Absatz ein Abstand von zwei Punkt einzuhalten ist. Dieses Beispiel deutet die enorme Mächtigkeit von LaTeX an, weil es durch solche Definitionen möglich ist, schnell auf bereits bestehende Lösungen aufzusetzen und diese nach Belieben zu erweitern. Dagegen existiert für XSL-FO nur ein begrenzter Umfang spezifizierter Befehle, der nicht erweiterbar ist.

```
1  \documentclass{dokumentklasse}
2  \usepackage{dokumentstil}
3  \title{K\"{u}hlschrank mit Lebensmitteln bef\"{u}llen}
4  \author{fial0015}
5  \begin{document}
6  \section{VorhabenK\"{u}hlschrank mit Lebensmitteln bef\"{u}llen}
7  \condition{Sie verf\"{u}gen \"{u}ber einen K\"{u}hlschrank.}
8  \condition{Sie verf\"{u}gen \"{u}ber Lebensmittel au\ss{}erhalb des K\"{u
       }hlschranks.}
9  \action{\"{O}ffnen Sie die K\"{u}hlschrankt\"{u}r.}
10 \sidegoal{Die K\"{u}hlschrankt\"{u}r ist ge\"{o}ffnet.}
11 \action{Legen Sie ihre Lebensmittel in den K\"{u}hlschrank.}
12 \action{Schlie\ss{}en Sie die K\"{u}hlschrankt\"{u}r.}
13 \sidegoal{Die K\"{u}hlschrankt\"{u}r ist geschlossen.}
14 \result{Sie haben den K\"{u}hlschrank mit Lebensmitteln bef\"{u}llt.}
15 \end{document}
```

Listing 2.4: Fertiges TeX-Dokument aus TeXML-Transformation

```
\ProvidesFile{dokumentstil.sty}[2008/10/29 Beispielstil]

\newenvironment{para}[1]{\ignorespaces}{\par}

\setlength{\parindent}{0pt}
\setlength{\parskip}{10pt}

\RequirePackage[a4paper, twoside]{geometry}

\newcommand{\goal}[1]{{\sc #1}\parskip2pt\par}
\newcommand{\condition}[1]{{\tt #1}\parskip2pt\par}
\newcommand{\sidegoal}[1]{{\bf #1}\parskip6pt\par}
\newcommand{\action}[1]{{\it #1}\parskip2pt\par}
\newcommand{\result}[1]{{\rm #1}\parskip2pt\par}

\renewcommand{\familydefault}{\sfdefault}
```

Listing 2.5: Stildatei für das als Beispiel genutzte fertige TEX-Dokument

2.7 Memory Stylesheets

So sehr LATEX versucht, z.B. ungünstige Zeilen- oder Seitenumbrüche zu vermeiden, kann es wie bei jeder automatisierten Lösung dennoch passieren, dass Inhalte ein ungewolltes Aussehen annehmen. Hier besteht somit das selbe Problem wie bei den Publikationsprozessen mit Adobe FrameMaker oder XSL-FO. Jedoch kennt der TeXML-Publikationsprozess eine Hilfestellung gebende Lösung des Problems. Memory Stylesheets können helfen, die Bedeutung des Problems stark zu verringern und somit Zeit- und Kosten zu sparen. Das Prinzip hinter Memory Stylesheets ist, dass sich die Publikationslösung vom Benutzer zur Anpassung und Layoutkorrektur vorgenommene Änderungen merken kann und diese einmalig vorgenommenen Änderungen automatisch bei jeder weiteren Publikation anwendet.

Für diesen Vorgang kommen die Programme *diff* und *patch* zum Einsatz [40]. *Diff* dient dazu, Dateien zu vergleichen und Unterschiede zwischen den Dateien anzuzeigen bzw. zu speichern. *Patch* ist das Gegenstück zu *diff* und wird genutzt, die mit dem Programm *diff* erzeugten und jeweilige Unterschiede enthaltende Dateien auf andere Dateien anzuwenden und so die jeweiligen Veränderungen nachzuvollziehen. Veränderungen werden in den Memory-Stylesheets auf Zeilenebene gespeichert. Wenn der Benutzer eine Änderung im LATEX-Dokument vornimmt, wird die gesamte Zeile, in der die Änderung geschah, sowie die vorhergehende und nachfolgende Zeile in den Memory-Stylesheet geschrieben. Dadurch ist es möglich modifizierte Zeilen wieder zufinden, auch wenn sich die Zeilennummer, etwa durch das Hinzufügen von Inhalten, verändert hat.

Werden XML-Daten publiziert, überprüft die Publikationslösung nach der Umwandlung der XML-Daten in TEX, ob in einem bestehenden Memory-Stylesheet Zeilen vorhanden sind, die bereits vom Benutzer während einer manuellen Korrektur verändert wurden. Trifft dies zu, werden die Veränderungen automatisch für das neu erstellte LATEX-Dokument übernommen. Anschließend kann der Benutzer nach einer Umwandlung in das druckfertige Format manuell überprüfen, ob das Dokument seinen Vorstellungen entspricht. Sollte dies nicht der Fall sein, muss er an entsprechender Stelle im LATEX-Dokument Änderungen vornehmen und das LATEX-Dokument erneut in das druckfertige Format umwandeln. Vor dieser zweiten Umwandlung wird das LATEX-Dokument jedoch mit einer angelegten Sicherungskopie seiner selbst verglichen. Sollten dabei Änderungen zu Tage treten, werden diese automatisch in den Memory-Stylesheet übernommen. Damit sind die Änderungen bei einer folgenden Publikation der XML-Daten automatisch bereits vorhanden und werden entsprechend angewendet.

Memory-Stylesheets ermöglichen eine vollständige, hochwertige und automatische Publikation von XML-Daten, sofern die XML-Daten einmalig nach ihrer Umwandlung in TEX vom Benutzer korrigiert wurden. Diese Option steht sowohl bei der Publikation mit Adobe FrameMaker als auch XSL-FO nicht zur Verfügung. Damit vereinfachen und beschleunigen Memory-Stylesheets den Publikationsprozess von XML-Daten erheblich.

An seine Grenzen stößt das Verfahren, wenn Änderungen an der gewünschten Formatierung der XML-Daten vorgenommen werden. Wenn neue Schriften beispielsweise eine andere Lauflänge aufweisen oder sich die Länge von Zeilen auf Grund geänderter Abstände verändert, müssen bestehende Memory-Stylesheets eventuell verworfen werden, weil die manuell vorgenommenen Korrekturen keine Gültigkeit mehr besitzen. Weil dies jedoch ein eher seltener Vorgang ist, sind Memory-Stylesheets als wirkliche Erleichterung bei der Publikation von XML-Daten zu betrachten.

```
1  --- output.tex.backup  2008-11-19 05:30:07.092929739 +0100
2  +++ output.tex         2008-11-19 03:39:32.335984914 +0100
3  @@ -7,6 +7,10 @@
4   \condition{Sie verf\"{u}gen \"{u}ber einen K\"{u}hlschrank.}
5   \condition{Sie verf\"{u}gen \"{u}ber Lebensmittel au\ss{}erhalb des K\"{
        u}hlschranks.}
6   \action{\"{O}ffnen Sie die K\"{u}hlschrankt\"{u}r.}
7  +
8  +Das ist ein sinnfreier Absatz, um die Funktionsweise von Memory
9  +Stylesheets zu beschreiben.
10 +
11  \sidegoal{Die K\"{u}hlschrankt\"{u}r ist ge\"{o}ffnet.}
12  \action{Legen Sie ihre Lebensmittel in den K\"{u}hlschrank.}
13  \action{Schlie\ss{}en Sie die K\"{u}hlschrankt\"{u}r.}
```

Listing 2.6: Beispiel für einen Memory-Stylesheet. Der Anwender hat die Zeilen *Das ist ein sinnfreier Absatz, um die Funktionsweise von Memory Stylesheets zu beschreiben.* ergänzt

2.8 Schlussfolgerung

Der TeXML-Publikationsprozess stellt eine sinnvolle Bereicherung der Welt der bisherigen XML-Publikationsprozesse dar. Der Publikationsprozess kann mehrere Vorteile auf sich vereinigen. Durch die freie und damit langfristige Verfügbarkeit der eingesetzten Komponenten, ist sicher gestellt, dass Investitionen in diesen Publikationsprozess langfristig gesichert sind. Die Verwendung von LaTeX als Mittel zur Erzeugung des druckfertigen Formats sichert eine sehr hohe drucksetzerische Qualität und garantiert eine robuste und schnelle Publikation. Die Einführung der Memory-Stylesheets bewirkt eine echte Beschleunigung des Publikationsprozesses und ermöglicht eine vollständig automatisierte Publikation von XML-Daten in hoher drucksetzerischer Qualität.

Vorteilhaft für Adobe FrameMaker ist dessen Möglichkeit, die zu publzierenden Daten direkt nach dem WYSIWYG-Prinzip zu bearbeiten. Nachteilig ist jedoch gegenüber den Publikationsprozesse mit XSL-FO und TeXML die mangelnden Fähigkeiten zur vollständig automatisierten Publikation von XML-Daten.

XSL-FO kann für sich als Vorteil verbuchen, auch für die Publikation der XML-Daten die XML nicht zu verlassen. Dies ist jedoch bei genauerer Betrachtung nur dann von Vorteil, wenn man bereits über fundierte Kenntnisse um Umgang mit XML und XSL-FO bzw. XSL-T verfügt. Ist dies nicht der Fall, stellt die Verwendung von TeXML und LaTeX jedoch keine Einschränkung dar. Durch die mögliche strikte Trennung von Layout und Inhalt auch in LaTeX entspricht diese Lösung sogar eher den Grundgedanken von XML.

Dennoch machen die Ausführungen deutlich, dass keiner der genannten Publikationsprozesse wirklich trivial ist. Um diese Situation zu verbessern entstand die Idee des Template-Designers.

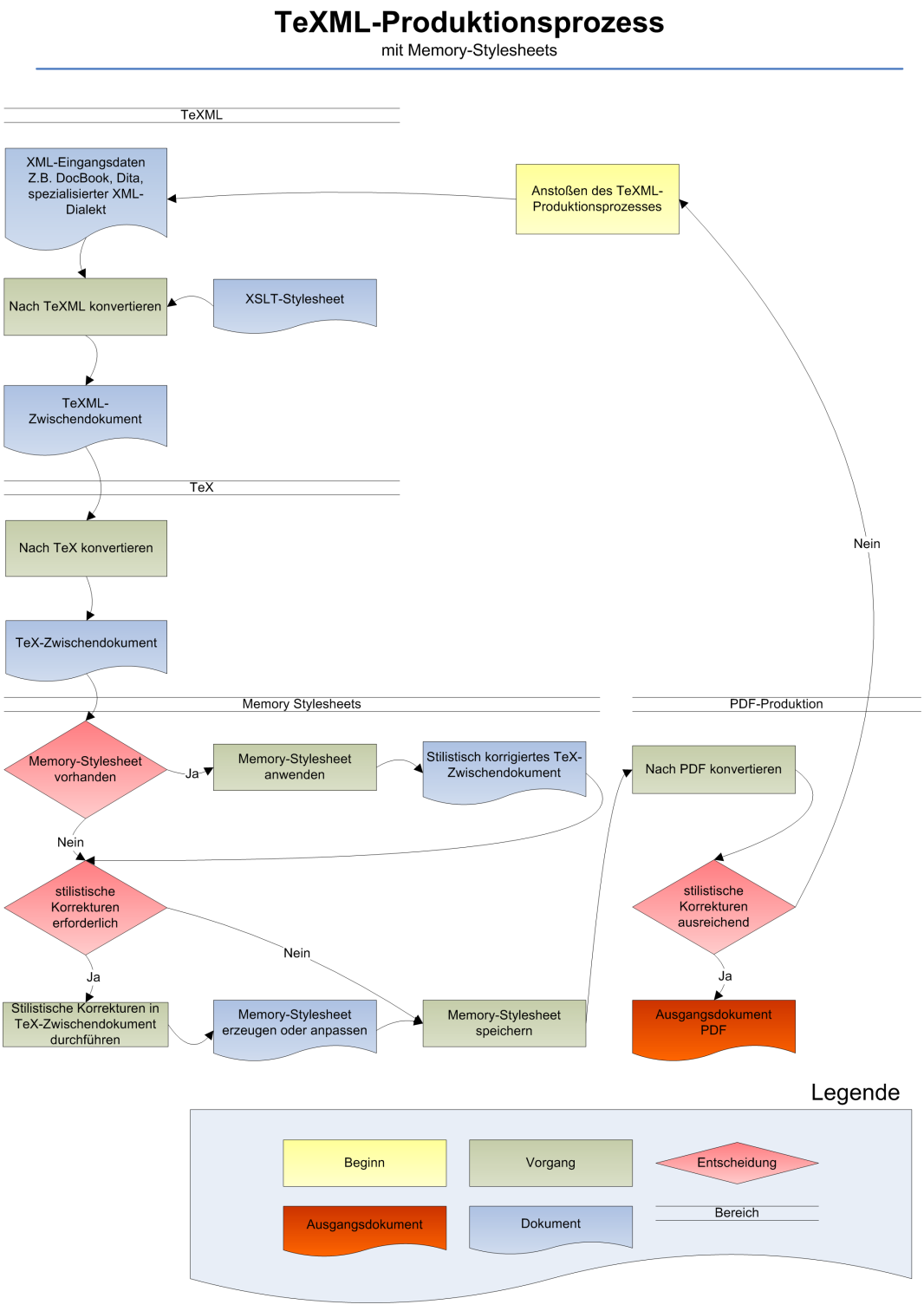

Abbildung 2.2: Überblick zum TeXML-Publikationsprozess

3 Zielsetzung

3.1 Einführung

Schwerpunkt der Bachelor-Thesis war die Erstellung eines Programms zur Erstellung von Dokumentvorlagen, die sich innerhalb eines Publikationsprozesses mit XSL-FO und LaTeX nutzen lassen. Ziel der Arbeit ist, einen Publikationsprozess in bestimmten Grenzen steuern und beeinflussen zu können, ohne über konkretes technisches Wissen zum jeweiligen Publikationsprozess zu verfügen. Die Erstellung von Dokumentvorlagen zur Gestaltung des allgemeinen Layouts der zu publizierenden Dokumente stellt einen ersten und wichtigen Schritt in diese Richtung dar.

3.2 Lasten und Vorüberlegungen

Für die Umsetzung des Programms, das den Arbeitstitel *Template-Designer* trägt, waren diverse Vorarbeiten und Überlegungen notwendig.

Bestehende Dokumente wurden analysiert und daraus abgeleitet welche Funktionalitäten des Template-Designers im gegebenen Zeitraum umsetzbar erschienen.

Der Template-Designer sollte prinzipiell plattformunabhängig sein. Jedoch sollte schwerpunktmäßig die gute Untersützung des Programms von Apples Mac OS X gegeben sein, weil zum gegenwärtigen Zeitpunkt der TeXML-Publishing-Server auf dieser Plattform arbeitet.

Der Template-Designer sollte Funktionalitäten bieten, um Position, Größe und Inhalt von Elementen zur Seitenausgestaltung festlegen zu können, die sich auf einer Seite befinden können. Dabei sollte nicht auf ein starres Konzept mit vorgegeben Bereichen gesetzt werden. Damit ist beispielsweise eine feste Definition von Kopf- und Fußzeilen mit verschiedenen Bereichen (links, mitte, rechts) gemeint. Dies hätte nicht gereicht, alle für die Analyse herangezogenen Dokumente umzusetzen. Beispielsweise wäre die Umsetzung einer Kustode bei diesem Vorgehen nicht möglich gewesen. Auch komplexere Dokumente – wie beispielsweise ein Brief oder Lieferschein mit fest positionierten und eventuell gar genormten Bereichen – wären auf diese einfache Art nicht umzuset-

zen gewesen. Daraus entstand ein Konzept zur Erzeugung von Rahmen, die jeweils mit individuellen Inhalten bestückt sind.

Position, Größe und Inhalt der Elemente zur Seitenausgestaltung sollten so präzise wie möglich festlegbar sein. Grund dessen ist, dass in der Regel für die in einem Publikationsprozesses mit XSL-FO und TeXML erzeugten Dokumente feste Regeln in einem Styleguide oder Redaktionsleitfaden definiert sind. Es sollte möglich sein, diesen so exakt wie möglich umzusetzen.

Der Template-Designer musste in der Lage sein, Vorlagen für Dokumente zu erzeugen, bei denen sich die Elemente zur Seitenausgestaltung in Abhängigkeit der eigentlichen Inhalte ändern können. Das bedeutet, die Dokumentvorlagen mussten eine Möglichkeit bieten, Dokumentabschnitte aufzunehmen und der Template-Designer musste in der Lage sein, diese zu verwalten.

Der Benutzer sollte sich um die Dokumentvorlagen selber kümmern, nicht darum, diese zusätzlich verwalten zu müssen. Deshalb sollte der Template-Designer in der Lage sein, mehrere Dokumentvorlagen gleichzeitig anzuzeigen bzw. für eine Bearbeitung bereitzuhalten. Diese Vorgabe führte dazu, dass ein zentraler Ort geschaffen werden musste, in dem die Dokumentvorlagen verfügbar sein sollten und den der Template-Designer verwalten konnte.

Um den Template-Designer vollständig in den Publikationsprozess mit XSL-FO bzw. LaTeX einbinden zu können, verlangte es eine konkrete Möglichkeit, auf die mit dem Template-Designer erzeugten Dokumentvorlagen zugreifen zu können. Dieser Umstand führte wie bereits der letztgenannte Punkt dazu, auf eine individuelle Verwaltung der Dokumentvorlagen durch den Benutzer zu verzichten.

Die Schaffung einer zentralen Stelle zur Ablage aller Dokumentvorlagen hatte den Nachteil, dass deren Verfügbarkeit nur auf der jeweils nutzbaren Plattform sichergestellt war. Die Verwendung mehrerer Plattformen zur Publikation hätte somit dazu geführt, dass Dokumentvorlagen mehrmals gewartet und gepflegt hätten werden müssen. Um dies zu vermeiden, ist der Template-Designer in der Lage, auf zwei Orte für die Verwaltung der bestehenden Dokumentvorlagen zugreifen zu können. Dadurch kann beispielsweise ein zentraler Ort geschaffen werden, der jene Dokumentvorlagen enthält, die überall verfügbar sein sollen, und es besteht die Möglichkeit an lokaler Stelle Dokumentvorlagen zu verwalten, um beispielsweise Tests durchzuführen oder sicher zu stellen, dass bestimmte Dokumente nur auf einem bestimmten System publiziert werden dürfen.

Durch den Umstand, dass der Benutzer selbst nicht mit der Verwaltung der Dokumentvorlagen betraut ist, musste der Template-Designer in der Lage sein, Dokumentvorlagen sowohl zu erzeugen, als auch zu löschen.

Um sicher zu stellen, dass bei der Arbeit mit den Dokumentvorlagen keine unbeabsichtigten Fehler passieren, sollte eine einfache Möglichkeit geschaffen werden, nur mit den in der jeweiligen Situation benötigten Rechten an den Dokumentvorlagen zu arbeiten. Dies musste der Template-Designer sicherstellen.

Die Dokumentvorlagen mussten der Art umgesetzt sein, dass sich dies mit möglichst geringen Aufwänden in die bestehenden Publikationsprozesse einbinden ließen.

3.3 Integration in den Publikationsprozess

Das im Rahmen der Thesis ausgearbeitete Programm zur Erstellung von Dokumentvorlagen genügt alleine nicht, um die Dokumentvorlagen auch nutzen zu können. Die erstellten Dokumentvorlagen müssen in die bestehenden Publikationsprozesse integriert werden. Für den TeXML-Publikationsprozess erfolgt eine Integration konkret derart, dass alle vorhandenen TemplateXML-Elemente in LaTeX-Stildefinitionen konvertiert werden und anschließend vom LaTeX-Prozessor interpretiert werden. Wie die Schritte zur Umwandlung der Dokumentvorlagen nach XSL-FO konkret aussehen werden, ist derzeit noch nicht klar, weil noch keine praktische Umsetzung hierfür existiert. Höchstwahrscheinlich werden die Informationen der Dokumentvorlagen über eine XSLT-Transformation einem bereits bestehenden XSL-FO-Dokument beigefügt.

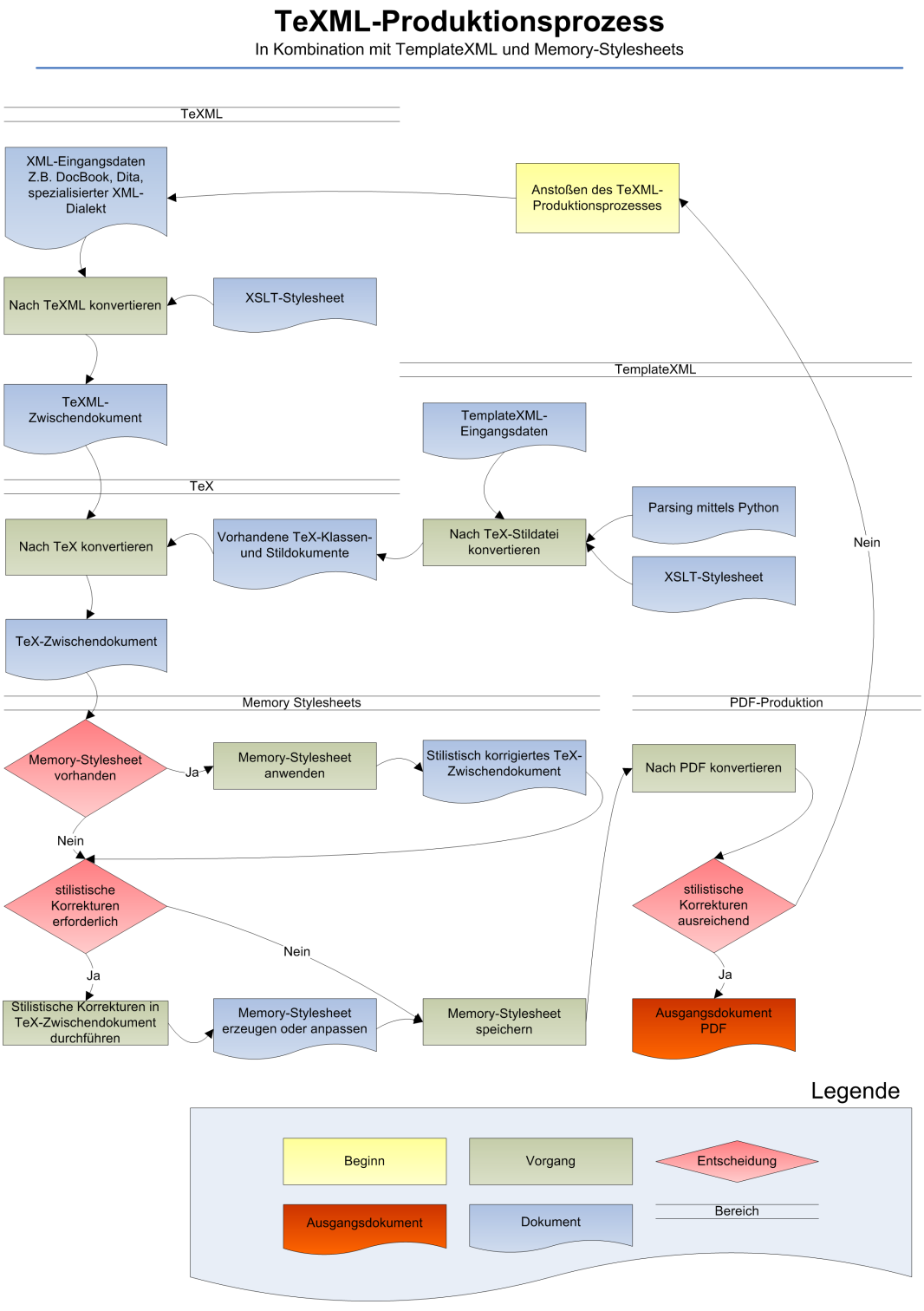

Abbildung 3.1: Überblick zum TeXML-Publikationsprozess mit TemplateXML-Integration

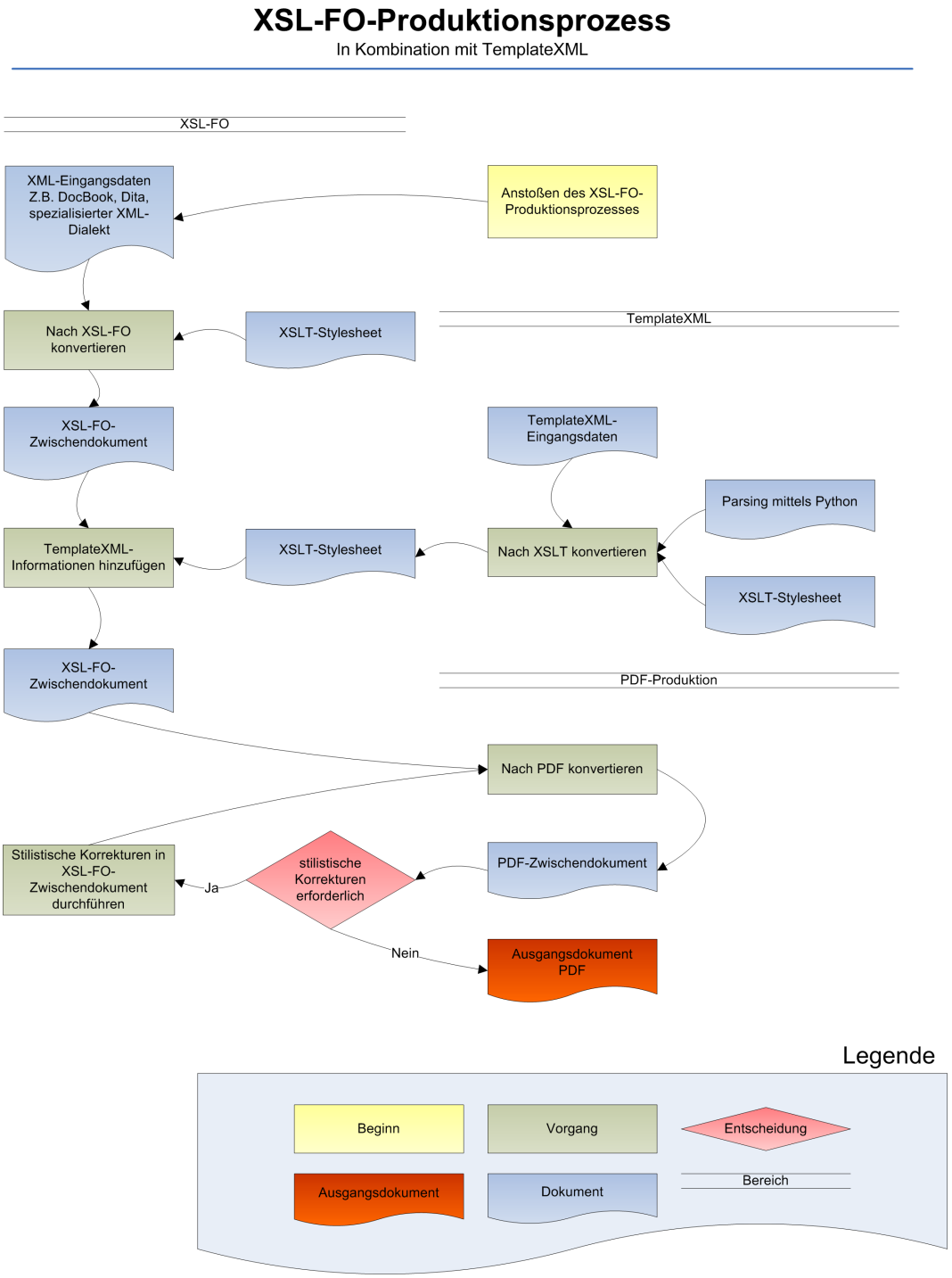

Abbildung 3.2: Überblick zum XSL-FO-Publikationsprozess mit TemplateXML-Integration

4 Dokumentvorlagen – TemplateXML

4.1 Einführung

Um in den Publikationsprozess mit TeXML oder XSL-FO einzugreifen und die gewünschten Vorlagen einzubinden, braucht es ein Speicherformat, dass die jeweils notwendigen Informationen zu den einzelnen Dokumentvorlagen zugänglich macht. Andernfalls müssten diese Informationen dem Publikationsprozess bei jeder einzelnen Publikation in irgendeiner Form manuell zugeführt werden. Das Speicherformat sorgt somit dafür, dass Informationen zu Dokumentvorlagen nur einmal erstellt werden müssen und anschließend immer wieder verwendet werden können.

Das Speicherformat muss dabei bestimmten Ansprüchen genügen. Es muss so einfach sein, dass das Programm zur Erstellung der Vorlageninformationen mit dem Speicherformat umgehen, es also lesen, bearbeiten und schreiben kann. Weiter muss das Speicherformat hinreichend komplex sein, um alle gewünschten Funktionen anbieten zu können. Und schließlich muss das Speicherformat so konzipiert sein, dass der Prozessor zur Umwandlung des Speicherformats in LaTeX oder XSL-FO Instruktionen auch fähig ist, dieses zu interpretieren und die Vorgaben umzusetzen. Um diesen Zielen möglichst gerecht zu werden, wurde als Speicherformat ein XML-Dialekt und eine einfache Verzeichnisstruktur um diesen XML-Dialekt herum entwickelt. Dieser XML-Dialekt erhiehlt den hier folgend verwendeten Namen TemplateXML.

Es erschien logisch auf einen XML-Dialekt zu setzen, weil XML aufgrund seiner hierarischen und verschachtelbaren Struktur sehr gut in der Lage ist, Dokumente abzubilden und somit auch die Vorlagen für diese Dokumente. Außerdem erschien es vernünftig innerhalb eines Publikationsprozesses von XML-Daten auch in diesem Bereich auf XML zu setzen. Hinzukommt die gute Unterstützung von XML durch die verwendeten Entwicklungswerkzeuge.

Alternativ wäre eine Abbildung der Strukturen in einer relationalen Datenbank, in von der verwendeten Programmiersprache interpretierbaren hierarischen Strukturen (z.B. multidimensionalen Arrays) oder die Verwendung von CSV-Dateien möglich gewesen. In Anbetracht der Alternativen hat XML hier weitere Vorzüge. Es ist in sich geschlossen. Eine vollständige XML Struktur kann genau eine vollständige Dokumentvorlage repräsentieren. Im Gegensatz dazu sind Datensätze in relationalen Datenbanken nur dann sinnvoll anwendbar sind, wenn diese in Beziehung zueinander gesetzt werden. Man

benötigt dazu also Einblick in die Datenbankstruktur. Bei XML ist die Struktur dagegen offensichtlich, weil XML durch seine XML-Tags die Struktur selbst vorgibt. Programmiersprachenеigene, hierarischen Strukturen haben den Vorzug, dass die Implementierung verhältnismäßig einfach erfolgen könnte. Jedoch hätten, um eine Interoperabilität zu anderen Anwendungen und Programmiersprachen sicher zu stellen, Schnittstellen definiert werden müssen, was einen ähnlichen Aufwand bedeutet hätte und komplexer gewesen wäre, als die Definition von XML. XML ist durch die sehr gute Unterstützung in nahezu jeder Programmiersprache und seine festgelegte innere Struktur faktisch unabhängig von spezifischen Schnittstellen. Es wurde als Austauschformat für Daten entwickelt und ist dahingehend vollständig akzeptiert. Die XML-Struktur definiert die Schnittstelle zum Datenaustausch und auf XML lässt sich mit allgemein verfügbaren Mitteln zugreifen. Das CSV-Format ließe sich ebenfalls sehr gut verarbeiten, lässt es jedoch nicht zu, geeignet komplexe Strukturen zu definieren, die eine Dokumentvorlage hinreichend abbilden könnten.

Ein weiterer Punkt war die Speicherung der Vorlageninformationen. Nachdem fest stand, dass XML für die Dokumentvorlagen verwendet werden sollte, musste noch entschieden werden, wie dieses XML als Ganzes zu verwalten sei. In Frage kamen das Dateisystem und eine relationale Datenbank. Entschieden wurde letztlich für das Dateisystem. Grund dessen war, dass dies keine weitere Infrastruktur – sprich Datenbankserver und zusätzliche Software zur Interaktion – notwendig macht, welche derzeit von keiner anderen Komponente im Publikationsprozess verwendet wird. Der Datenbankserver hätte jedoch für die Verwaltung Vorteile dahingehend gebracht, dass es für die auf die Daten zugreifenden Anwendungen keine Rolle gespielt hätte, wo die Vorlageninformationen liegen, solange ein Zugriff auf den Datenbankserver möglich gewesen wäre. Ein weiterer Vorteil der Lösung mit einem Datenbankserver wäre die einfachere Administration von Benutzern gewesen, weil alle eventuell in Frage kommenden Datenbankserver hier sehr feingliedrig justiert werden können. Alternativ wäre die Speicherung mittels SQLite [80] eine Option gewesen. Dies hätte aber keinerlei Vorteile gebracht, weil in diesem Fall trotzdem hätte bekannt sein müssen, wo eine entsprechende SQLite-Datenbankdatei liegt und ferner hätte es trotzdem die Einbindung externer Werkzeuge nötig gemacht. Außerdem verfügt SQLite nicht über die Möglichkeit mehrere Benutzer mit verschiedenen Zugriffsrechten zu verwalten.

Letztlich wurde als Speicherformat also ein eigenständig definiertes XML-Format gewählt, dass in einem Dateisystem als Datei abgelegt wird. Für den Zugriff auf die entsprechenden Dateien existiert eine Konfigurationsdatei, die die entsprechenden Verzeichnisangaben enthält.

TemplateXML wurde dazu entworfen, als Ausgangspunkt für die Konvertierung in LaTeX und XSL-FO Instruktionen zu dienen und gleichzeitig leicht verständlich zu sein. Dazu wurde versucht, sich – trotz gegenwärtiger Mängel bei den Möglichkeiten zur Validierung – auf möglichst wenige Elemente und Attribute zu beschränken und die Struktur möglichst einfach zu halten. TemplateXML versucht dabei die Struktur des Ausgangs-

dokumentes abzubilden: ohne dessen Inhalte, aber mit zusätzlichen Informationen zur Ausgestaltung eines Dokumentes. Im Folgenden wird die Struktur von TemplateXML diskutiert. Dabei wird, wo dies notwendig erscheint, auch auf Fragen der Implementierung eingegangen.

4.2 Namensraum

In XML dienen Namensräume dazu, XML-Dialekte eindeutig zu kennzeichnen, es können Elemente oder Attribute eindeutig anhand ihrer Funktion und inneren Struktur identifiziert werden. Die Konsequenz daraus ist, dass verschiedene XML-Dialekte in einem Dokument vereinigt werden können und trotzdem voneinander abgeschottet sind. Dies ermöglicht es, Attribute und Elemente aus verschiedenen XML-Dialekten aber mit identischen Namen im selben Dokument zu verwenden und trotzdem Ihre jeweils eindeutige Bestimmung aufrecht zu erhalten. Dabei beeinflussen sich die Elemente bzw. Attribute nicht gegenseitig.

Ein Namensraum wird über die Zuweisung eines sogenannten Internationalized Resource Identifiers (IRI) an ein Element oder ein Attribut gebildet. Weil IRIs aber sehr lang sein oder Zeichen enthalten können, die gemäß der XML-Recommendation nicht für Element- oder Attributnamen zulässig sind, wurden sogenannte Qualified Names eingeführt. Diese setzen sich aus einer kurzen Zeichenfolge, die dem IRI zugeordnet ist und dem Element- oder Attributnamen zusammen. Diese kurze Zeichenfolge wird Präfix genannt. Die Kombination aus Präfix und Element- oder Attributnamen kennzeichnet das Element oder Attribut in seiner Funktion. Ein Präfix wird einem Namensraum über Attribute zugeordnet die per Definition immer in XML verfügbar sind und mit den Zeichenfolgen *xmlns* oder *xmlns:* beginnen. Mittels Namensräumen ist es beispielsweise möglich, eine SVG-Grafik in das Bitplant-TemplateXML einzubinden, wenn das parsende Programm, den SVG-Namensraum kennt und die Bedeutung seiner Inhalte interpretieren kann.

TemplateXML benutzt den Namensraum *http://www.bitplant.de/template*. Wenn ein Präfix für den Namensraum verwendet wird, soll dieser *bit* lauten, um etwaige Verwirrungen zu verhindern. Der Template-Designer nutzt diesen Präfix.

Der Namenraum setzt sich in diesem Fall aus der Internetadresse der Firma Bitplant.de und dem Verzeichnis *template* zusammen, dessen Name als Abkürzung für TemplateXML verstanden werden kann. Wird die IRI des Namensraums als URL (Uniform Resource Locator) aufgefasst, findet sich darunter die TemplateXML-Dokumentation. Dass IRIs als URLs interpretiert werden können, ist der Tatsache geschuldet, dass URLs per Definition eine Teilmenge der IRIs bilden. [81, 1.1.3. URI, URL, and URN] [66, The term "Uniform Resource Locator" (URL) refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by descri-

bing its primary access mechanism (e.g., its network "location")]. Auf diese Weise geht ebenfalls das World Wide Web Consortium (W3C) bei der Definierung seiner Empfehlungen vor und es hat sich als allgemeine Praxis durchgesetzt.

4.3 Elemente und Attribute von TemplateXML

4.3.1 Das Wurzelelement

Für TemplateXML sind prinzipiell zwei verschiedene Wurzelelemente vorgesehen. Eines trägt den Namen *designer*, das andere Element trägt den Namen *template*. Der Unterschied besteht darin, dass das Element *designer* mehrere Elemente *template* aufnehmen kann und dafür Sorge trägt, dass verschiedene Vorlagen innerhalb einer XML-Datei definiert werden können. Diese Option soll den Datenaustausch vereinfachen, für eine gewisse Strukturierung sorgen und die Administrierung des Vorlagenbestandes variabler gestalten können.

Es ist durch die Sammlung von *template*-Elementen in einem *designer*-Element möglich, diese innerhalb des gesamten Datenbestandes nach verschiedenen Eigenschaften zu sortieren. Diese Eigenschaften können sich beispielsweise aus Inhalten der Dokumentvorlagen ableiten lassen (z.B. identische Logos). Weiterhin können die Eigenschaften von außen vorgegeben sein. So können zusammengefasste Vorlagen allesamt nur für einen Typ von Dokument vorgegeben sein. Auch administrativ kann durch diese Option variabler agiert werden. Es können dadurch Dokumentvorlagen zusammengefasst werden, die nur von einem bestimmten Nutzer bearbeitet oder genutzt werden dürfen. Durch den Wegfall mehrere Vorlagendateien lässt sich dieses Ziel schneller erreichen und auch der Sonderfall der Erzeugung neuer Vorlagen ist damit abgedeckt. Wenn Vorlagendateien individuell administriert werden müssten, wäre dies schwieriger zu bewerkstelligen, weil das Anlegen einer neuen Vorlage auch immer einen neuen Eingriff auf administrativer Ebene bedeuten würde. Durch die Nutzung einer Sammlung von Dokumentvorlagen genügt es, die Sammlung als solche zu administrieren, deren Inhalte jedoch dem Nutzer zu überlassen. Sicher gestellt wird der Zugriff bei diesem Vorgehen über die Zugriffsrechte des Dateisystems. Das Element *designer* soll somit auch zumindest in gewissem Maß, die Nichtverwendung eines Datenbankservers als Speicherort kompensieren. Die Einführung des *designer*-Elementes vergrößert zwar die Komplexität auf XML-Seite, sorgt aber andererseits für größere Flexibilität im Umgang mit den XML-Daten.

4.3.2 Element designer

Das Element *designer* ist ein strukturiertes Element, dass einzig entweder gar kein Element *template* beinhalten kann, oder beliebig viele Elemente *template*. Damit dient es, wie bereits in 4.3.1 beschrieben als Container für verschiedene Dokumentvorlagen. Dass auch keinerlei Kindelemente *template* gestattet sind, hat zum Ziel eine größere Flexibilität im Umgang mit den Vorlagen zu erreichen. So kann das XML-Dokument mit dem Wurzelelement zwar auf administrativer Ebene angelegt werden, dessen Inhalte, können aber alleinig vom Benutzer festgelegt werden. Erlaubte Attribute sind *name*, *id* und *lang*. Dahingehend entspricht das Element *designer* den Elementen *template*, *page* und *frame*. Das Attribut *id* ist in Anbetracht der Funktion des Elementes *designer* als Wurzelelement insofern zu erklären, dass es zwar keinen direkten Nutzen bringt, schließlich ist das Wurzelelement immer eindeutig, weil nur eines in einer XML-Struktur existieren darf, aber dessen Verfügbarkeit schadet auch nicht. Es ergibt vielmehr Sinn, das Attribut *id* auch hier einzuführen, weil damit alle strukturstiftenden Elemente (*designer*, *template*, *page*, *frame*) über dieses Element verfügen. Dadurch gestaltet sich die gesamte Struktur von TemplateXML insgesamt weniger komplex und eine Implementierung für den Zugriff auf TemplateXML muss bei iterativen oder rekursiven Prozessen nicht auf diese eventuelle Besonderheit Rücksicht nehmen.

4.3.3 Element template

Das Element *template* kann sowohl Wurzelelement eines TemplateXML-Dokumentes sein, als auch innerhalb des Elementes *designer* vorkommen. Die originäre Aufgabe des Elementes ist es, alle Informationen aufzunehmen, die für die Erzeugung einer Dokumentvorlage relevant sind. Als Attribute sind, wie beim Element *designer*, die Attribute *name*, *id* und *lang* gestattet. Dass das Attribut *id* für dieses Element gestattet ist, hat die schon beim Element *designer* genannten Gründe. Jedoch kommt hinzu, dass das Element *template* nicht nur als Wurzelelement, sondern auch als Kindelement des Elementes *designer* vorkommen kann. In diesem Fall lohnt sich die Einführung des Attributes, um das jeweilige Element *template* nötigenfalls eindeutig zu identifizieren.

Das Element *template* darf genau ein Kindelement *parameter* beinhalten, dem keines oder beliebig viele Elemente *page* folgen können. Das Element *parameter* beinhaltet Informationen zur Ausgestaltung der gesamten Dokumentvorlage. Damit liefert es quasi Grundeinstellungen für die Vorlage. Dass mehrere Elemente *page* gestattet sind, ist in Ihrer Funktion begründet, einen Abschnitt innerhalb des fertigen Dokumentes zu repräsentieren. Wenn kein Element *page* verhanden ist, soll eine Implementierung die Grundeinstellungen der Dokumentvorlage für das gesamte publizierende Dokument annehmen.

4.3.4 Element page

Ein Element *page* ist immer Kindelement des Elementes *template*. Die Aufgabe des Elementes *page* ist es, alle Vorlageninformationen aufzunehmen, die für einen Dokumentabschnitt gültig sind. Damit ermöglicht das Element *page* die Strukturierung eines Dokumentes. Ein Element *page* kann so einem Deckblatt entsprechen, einem Bereich mit Inhaltsverzeichnis, dem Bereich in dem die eigentlichen Inhalte zu finden sind oder einem Anhang. All diese Bereiche stellen in der Regel unterschiedliche Anforderungen an die Seitenausgestaltung. Das meint, dass jeder dieser Abschnitte unterschiedlich formatierte Kopf- und Fußzeilen aufweisen kann oder andere Abstände zum Seitenrand. Um diese Funktionen zu erfüllen sind als Kindelemente des Elementes *page* genau ein Element *parameter* und darauf folgend genau kein oder unbeschränkt viele Elemente *frame* erlaubt.

Das Element *parameter* übernimmt hier die selben Aufgaben, die es schon für das Element *template* ausübt. Hinzu kommt hier allerdings die Besonderheit, dass die Implementierung auf den Wert des Attributes *inherit* des Elternelementes *page* Rücksicht nehmen soll. Wenn das Attribut *inherit* den Wert *enable* trägt, soll die Implementierung keine Rücksicht auf die Inhalte des Kindelementes *parameter* nehmen. Dies ist die implizite Voreinstellung. Wenn der Wert allerdings *disable* lautet, soll eine Implementierung die Inhalte des Elementes *parameter* für den Abschnitt, den das Element *page* repräsentiert, anwenden. Dadurch sind dann die globalen Einstellungen des Elementes *template* quasi verschattet.

Es gab grundsätzlich mehrere Möglichkeiten hier vorzugehen. Zum Einen wäre es möglich gewesen, keinerlei Definitionen zur Seitenausgestaltung innerhalb des Elementes *page* zu zu lassen. Damit wären die Angaben des Elementes *template* für das gesamte Dokument gültig gewesen. Dies wäre aber sehr unflexibel, weil es verhindern würde, für einen Abschnitt eine divergierende Seitenausgestaltung anzuwenden. Das ist zwar bei einer Vielzahl von Dokumenten kein Problem, jedoch gibt es auch Dokumente, deren Anhang zum Beispiel einen anderen Seitenrand aufweist, als jener des Dokumenthauptteils. Die zweite Möglichkeit wäre es gewesen, auf die Definition globaler Einstellungen für die Seitenausgestaltung zu verzichten. Das hätte bedeutet, dass für jedes Element *page* Angaben bezüglich der Seitenausgestaltung notwendig geworden wären. Damit würde jedes Element *page* für sich selbst stehen und auch das Attribut *inherit* wäre überflüssig gewesen. Das wiederum hätte einen erhöhten Aufwand bei der Erstellung einer Dokumentvorlage nach sich gezogen, der in der Praxis nur schwer zu rechtfertigen wäre, weil eben doch bei vielen Dokumenten die meisten Abschnitte die selbe Seitenausgestaltung aufweisen. Globale Änderungen müssten bei dieser Umsetzung auf alle Abschnitte angewandt werden, was unnötig aufwändig wäre.

Als Lösung dieses Dilemmas wurde das Attribut *inherit* eingeführt. Dadurch ist es wegen des obig beschriebenen gewünschten Verhaltens der Implementierung möglich,

globale Vorgaben für die Seitenausgestaltung eines Dokumentes zu tätigen und dennoch bei Bedarf auf individuelle Einstellungen für den jeweiligen Abschnitt zu wechseln.

Ferner verfügt das Element *page* über die Attribute *name*, *lang* und *id*. Dabei ist das Attribut *name* obligatorisch. Dies ist besonders wichtig, weil im Laufe des Transformationsprozesses von XML in ein druckfertiges Dokumentformat anhand dieses Attributwertes entschieden wird, wann und wo die Inhalte des Elementes *page* erscheinen sollen.

Das Attribut *id* soll das jeweilige Element *page* eindeutig kennzeichnen. Das Element *lang* steht zur Verfügung um eventuell anderssprachige Beschreibungen oder Inhalte unterhalb des jeweiligen Elementes *page* zu kennzeichnen.

Der Name des Elementes *page* ist abgeleitet vom XSL-FO Element *simple-master-page*. Alternativ kamen die an LaTeX angelehnten Benennungen *section*, *chapter* oder *part* in Frage, haben sich jedoch nicht durchsetzen können. Dem eventuellen Problem, dass das Element *page* mit dem noch folgend erläuterten Element *paper* verwechselt werden könnte, sollte die vollständig andere innere Struktur im Wege stehen.

4.3.5 Element frame

Das Element *frame* beinhaltet alle Angaben zur Beschreibung genau eines Elementes zur Seitenausgestaltung. Mit Element sind beispielsweise Inhalte der Kopf- und Fußzeilen, Seitenzahlen, eine Grafik oder eine farbige Fläche gemeint. Das Element *frame* beinhaltet für die Beschreibung dessen genau ein Element *parameter*, gefolgt von genau einem Element *content*. Dabei dient das Element *parameter* dazu, Angaben zur Position und Dimension des Inhalts zu kapseln und und eine Beschreibung für den ganzen Frame aufzunehmen. Das Element *frame* verfügt wie die Elemente *template* und *page* und aus den selben Gründen über die Attribute *name*, *id* und *lang*. Das Attribut *name* ist auch hier obligatorisch, obwohl dies nicht unbedingt nötig erscheint. Auch hier war ausschlaggebend, dass eine Konsistenz zu den Elementen *template* und *page* bezüglich des Verhaltens gegeben sein sollte. Außerdem kann das Attribut dem Benutzer die Navigation erleichtern, weil er anhand des Attributwertes auf einen Blick erkennen kann, welche Inhalte das Element *frame* aufnimmt, wenn der Attributwert geschickt gewählt wurde.

4.3.6 Element content

Das Element *content* ist so definiert, dass es einzig unstrukturierte Inhalte und die Attribute *type* und *angle* aufnehmen kann. Das fakultative Attribut *angle* dient dazu, Inhalte als zu drehend zu markieren, wenn dessen Wert ungleich *0* ist. Dadurch soll es zum

Beispiel möglich sein, eine Kustode einzuführen. Die aktuellen Inhalte sollen sich damit in 90 Grad Schritten rotieren lassen. Ob mit diesem Attribut langfristig und an dieser Stelle zu rechnen ist, ist zu diskutieren. Im Laufe der Umsetzung begann sich die Idee durchzusetzen, es sei sinnvoller ein Attribut *style* einzuführen, dass auf eine irgendwie geartete Layoutinformation verweist. Diese würde dann wiederum Informationen zur Steuerung einer Rotation beinhalten. Gegenwärtig ist das Attribut dennoch vorhanden, um auf die Möglichkeiten einer feineren Layout-Steuerung der einzelnen Inhalte hinzuweisen. Das Attribut *type* soll kennzeichnen, wie die unstrukturierten Inhalte des Elementes *content* interpretiert werden sollen. Dabei sind prinzipiell vier verschiedene Inhaltstypen vorgesehen.

Wenn das Attribute *type* den Wert *text* trägt, sollen die unstrukturierten Inhalte so ausgegeben werden, wie diese im TemplateXML vermerkt sind. Es handelt sich dabei also um rein statische Inhalte, die nicht in irgendeiner Form interpretiert werden sollen. Diese statischen Inhalte können sinnvoll sein, um z.B. Kontaktdaten, also Informationen die sich verhältnismäßig selten ändern, aufzunehmen.

Anders soll sich die Implementierung verhalten, wenn der Wert des Attributes *type vartext* lautet. In diesem Fall sollen die Inhalte mittels Platzhalterzeichen und als LaTeX-Definitionen dynamisch interpretiert werden. Der Inhalt
hrule würde demnach eine horizontale Linie zeichnen, weil der entsprechende LaTeX-Befehl zum Zeichnen eben jener horizontalen Linie so lautet. Dieser Inhalt als vom Typ *text* gekennzeichnet würde hingegen auch die Ausgabe
hrule erzeugen.

Eine weitere und im Template-Designer angelegte Form der variablen Inhalte stellen Platzhalterzeichen dar. Diese werden mit einem Prozentzeichen % eingeleitet. Es besteht derzeit im Template-Designer die Möglichkeit sowohl variable Datums-, als auch Dokumenteigenschaften einzusetzen. Bei den Zeit- und Datumswerten wurden jene Möglichkeiten umgesetzt, welche die Norm Iso8601 [30] vorsieht. Als Dokumenteigenschaften wurden einige Gestaltungsmöglichkeiten für Seiten- und Kapitelzahlen vorgesehen. Beide Platzhalterkategorien lassen sich insofern als redundant beschreiben, weil sich diese Inhalte auch mit LaTeX darstellen lassen. Jedoch sind die Platzhalter kürzer notiert und genau deswegen auch eindeutiger und einfacher zu verstehen. Die Verwendung von direkten LaTeX-Befehlen bringt ferner die Einschränkung mit sich, dass dadurch die Option, XSL-FO zu nutzen weg fiele, würde es keinen LaTeX zu XSL-FO Konverter geben. Ob es etwas derartiges gibt, ist mir jedoch trotz Recherche unbekannt. Jedoch würde dieses Manko auch nur dann an Bedeutung gewinnen, wenn LaTeX bzw. TeXML und XSL-FO nebeneinander genützt würden, was auf Grund der in beiden Fällen verhältnismäßig komplexen Konfiguration des Gesamtsystems unwahrscheinlich und auch unnötig erscheint. Bei einem theoretischen Austausch von TemplateXML-Dokumenten zwischen verschiedenen Systemen, trifft die Einschränkung jedoch voll zu.

Eine dritte Form der Interpretation von unstrukturierten Inhalten soll in Kraft treten, wenn das Attribut *type* den Wert *color* trägt. In diesem Fall soll von der Implementierung angenommen werden, dass die unstrukturierten Inhalte in irgendeiner Form Farbinformationen tragen. Für diesen Zweck wurde ein Python Paket mit dem Namen *color* geschrieben, dass so tolerant wie möglich die eventuell vorhandenen Farbangaben interpretiert. Der Template-Designer nutzt dieses Paket. Wenn eine Erkennung der Farbwerte fehlschlägt, wird der Farbwert schwarz im Cmyk-Farbraum angenommen. Die Implementierung kann zwischen Farbwerten im Cmyk und RGB-Farbraum unterscheiden, wobei im RGB-Farbraum die Notation als RGB-Tripel, Hex-Wert und X11-Farbwert erkannt werden kann. Dies erschien notwendig, weil Redaktionsleitfäden oder Styleguides entweder nur jeweils Angaben zu einem Farbraum machen, oder zwischen den Publikationsmedien Bildschirm und Papier unterscheiden. Für den Bildschirm sind RGB-Farbwerte dabei das Mittel der Wahl, für den professionellen Druck Cmyk-Farbwerte.

Die Erkennung des Farbraumes erfolgt über reguläre Ausdrücke. Dabei wird der RGB-Farbraum über die Zeichenkette einleitende Zeichenkette *rgb* gekennzeichnet, dem wiederum ein Dreisatz von mittels Leerzeichen und oder Kommata getrennten Ziffern folgen muss. Beispiele hierfür sind die Notationen *rgb(12, 34, 45)* und *rgb 56 67 78*. Bei Hex-Werten verhält es sich ähnlich, wobei als Schlüssel zur Erkennung der Angabe nicht die Zeichenfolge *rgb* sondern das Zeichen *#* dient. Ferner sind hier hexadezimale Farbwerte, also die Zeichen 0-9 und A-F, statthaft. Für den Cmyk-Farbraum dient der Schlüssel *cmyk* als Erkennungsmerkmal. Als letzte mögliche Form einen Farbwert anzugeben, kann das Paket *color* X11-Farbwerte interpretieren. Dies sind eindeutige Bezeichnungen bzw. Festlegungen für Farbwerte, die auf eine Festlegung für das X11 Window System zurückgehen und mittlerweile auch im Working Draft des W3C für das CSS Color Module Level 3 zu finden sind [93]. Diese Variabilität wurde als notwendig erachtet, um ohne große Aufwände interoperabel mit eventuell zukünftig genutzten Komponenten von Drittanbietern zu sein, die andere Konventionen für die Benennung von Farbwerten und -räumen nutzen.

Der letzte verfügbare Inhaltstyp wird angenommen, wenn das Attribut *type* den Wert *image* trägt. In diesem Fall stehen die unstrukturierten Inhalte des Elementes *content* für den Namen einer einzubindenden Graphikdatei. Dabei geht die Implementierung des Template-Designers den Weg, keine absolute Pfadangabe für die Graphikdatei anzunehmen, sondern den unstrukturierten Inhalt durch zwei Doppelpunkte in drei Teile zu gliedern. Dabei dient der erste Teil dazu, einen Platzhaltertext *serverImage* oder *clientImage* aufzunehmen. Mit *serverImage* ist der Dateisystempfad gemeint, in dem die vom Template-Designer verwalteten Bilddateien für entfernte Vorlagen liegen. Dieser Pfad lässt sich von Drittanwendungen über den Zugriff auf die API des Template-Designers oder seine Konfigurationsdatei bestimmen. Der Platzhaltertext *clientImage* verfolgt das selbe Ziel für Graphikdateien, die lokal vorliegen. Der zweite Teil beinhaltet den eigentlichen Dateinamen. Dieser wird vom Template-Designer beim Import einer Graphikdatei variabel bestimmt, um eventuelle Kreuzungen mit anderen Dokumentvor-

lagen bzw. Dateinamen zu vermeiden, deren Graphikdateien im selben Verzeichnis liegen. Der dritte Teil beinhaltet den ursprünglichen Dateinamen. Dieser dritte Teil ist eigentlich nicht nötig für einen korrekten Funktionsablauf. Der Wert wird dennoch vom Template-Designer gespeichert, um vom Template-Designer angezeigt werden zu können. Dies soll dem Benutzer die Erkennung der verwendeten Graphikdatei neben einer Votschaufunktion weiter erleichtern. Die Verwendung dieser für XML atypischen Struktur wird in 6.2 näher beleuchtet.

4.3.7 Element parameter

Das Element *parameter* ist ein strukturiertes Element und muss Kindelement der Elemente *template*, *page* oder *frame* sein. Die Aufgabe des Elementes *parameter* ist es, alle Informationen zu kapseln, die das jeweilige Elternelement näher beschreiben. Deshalb sind innerhalb des Elementes die Kindelemente *description*, *dimension*, *position* und *paper* statthaft. Das Kindelement *paper* ist jedoch nicht gestattet, wenn *parameter* das Kindelement eines Elementes *frame* ist. Dies rührt daher, dass es für einen *frame* nicht notwendig ist und sinnfrei wäre, diesbezügliche Angaben zu tätigen.

4.3.8 Element description

Das Element *description* darf genau kein- oder genau einmal innerhalb eines Elementes *parameter* auftauchen. *Description* darf unstrukturierte Inhalte aufnehmen und dient dem Ziel, Platz für einen Text zu schaffen, der das jeweilige Elternelement *template*, *page* oder *frame* näher beschreibt und einordnet. Dies soll die Wiedererkennung erhöhen und die Handhabung der Dokumentvorlagen vereinfachen. Das Element *description* verfügt über keinerlei Attribute. Auch nicht über das Attribut *lang*. Es wurde hier bewusst darauf verzichtet, weil das Attribut dem übergeordneten Element *template*, *page* oder *frame* zugeordnet werden kann.

4.3.9 Element dimension

Das Element *dimension* ist ein leeres Element und verfügt über die obligatorischen Attribute *type* und *value* sowie über das fakultative Attribut *unit*. Unterhalb der übergeordneten Elemente *template* und *page* bestimmen die mit dem Element *dimension* angegebenen Werte die Größe des Textrahmens, in dem sich die Inhalte aus der zu publizierenden XML-Quelle befinden sollen bzw. werden.

Das Attribut *type* kann hier die Werte *width* oder *height* annehmen. *Width* definiert die Breite des Textrahmens und *height* die Höhe des Textrahmens. Beide Werte wurden ge-

wählt, weil diese sowohl in CSS als auch in XSL-FO genutzt werden. Das Attribut *value* beinhaltet einen Dezimalwert, welcher die eigentliche Breite bzw. Höhe festlegt. Dieser Dezimalwert benutzt den Punkt als Dezimaltrennzeichen. Das fakultative Element *unit* legt fest, in welcher Einheit der Wert des Attributes *value* gemessen werden soll. Das Attribut *value* darf die Werte *mm*, *cm*, *pt* und *inch* tragen, wobei *mm* als Vorgabewert angenommen wird.

Wert	**Bedeutung**	**Bemerkung**
mm	Millimeter	Voreinstellung
cm	Zentimeter	Entspricht 10 Millimetern
pt	Punkt	Entspricht 127/360 bzw. 0.353 Millimetern.
		Damit ist somit nicht der Didot-Punkt (0.375 Millimetern) gemeint
Inch	Inch	Entspricht 25,4 Millimetern

Unterhalb eines Elementes *frame* beziehen sich die innerhalb des Elementes *dimension* eingetragenen Werte nicht mehr auf die Größe des Textrahmens. Statt dessen legt *dimension* hier die Größe des Frames als solchen fest. Diese Unterscheidung ist zwingend, weil sich das Element *dimension* eines Frames gar nicht auf den Textrahmens beziehen kann. Dies erledigt bereits das Element *dimension* des übergeordneten Elementes *page*. *Frame* und Textrahmen stehen also in einer gedachten Hierarchie gleichwertig unterhalb des Elementes *page*. Es war beabsichtigt, das Element *dimension* trotz der erweiterten Bedeutung unterhalb des Elementes *frame* ebenso zu benennen, wie unterhalb der Elemente *template* und *page*. Die Begrifflichkeit *dimension* passt eindeutig auf beide Funktionen und auch die Validierung und das Parsen des Elementes ist identisch. Die lediglich bei der Interpretation der gegebenen Werte notwendige Fallunterscheidung hat hier nicht ausgereicht, um die Einführung eines neuen Elementnamens zu rechtfertigen.

4.3.10 Element position

Das Element *position* ist ein leeres Element und verfügt über die obligatorischen Attribute *type* und *value* sowie über das fakultative Attribut *unit*. Unterhalb der übergeordneten Elemente *template* und *page* bestimmen die mit dem Element *position* angegebenen Werte die Position des Textrahmens, in dem sich die Inhalte aus der zu publizierenden XML-Quelle befinden werden. Die Positionierung soll dabei in Abhängigkeit zu den Rändern des Papiers erfolgen.

Das Attribut *type* kann hier die Werte *left*, *top*, *right* oder *bottom* annehmen. *Left* bezieht sich auf den Abstand zwischen linkem bzw. innerem Seitenrand des Blattes und dem Textrahmen bzw. den Inhalten eines Frames. *Right* bezieht sich auf den Abstand zwischen rechtem bzw. äußerem Seitenrand des Blattes und dem Textrahmen bzw. den Inhalten eines Frames. In welchem Fall Links oder Innen bzw. Rechts oder Außen angenommen werden soll, entscheidet ein Attributwert des Elementes *paper*. *Top* und

bottom beziehen sich jeweils auf den oberen und unteren Abstand von Seitrand und Textkorpus. Beide Bezeichnungen wurden gewählt, weil diese sowohl in CSS als auch in XSL-FO genutzt werden und dies aufgrund ihrer Bedeutung in der englischen Sprache auch logisch erscheint. Das Attribut *value* beinhaltet einen Dezimalwert, welcher den eigentlichen Abstand festlegt. Dieser Dezimalwert benutzt den Punkt als Dezimaltrennzeichen. Das Attribut *unit* hat die selbe Bedeutung und Funktion wie unterhalb des Elementes *dimension*.

4.3.11 Vorkommen und Verwendung der Elemente dimension und paper

Es ist definiert, dass das Element *dimension* mindestens kein- und höchstens zweimal vorkommen darf. Das Element *position* darf mindestens zwei- und höchstens viermal vorkommen. Dieser Umstand ist deren Funktion geschuldet. Wenn man den Textkorpus oder *frame* auf einem Blatt unbestimmter Größe exakt ausrichten will, braucht man sowohl für die horizontale- wie für die vertikale Achse genau jeweils zwei Werte, um die Ausrichtung und Größe vollständig zu beschreiben.

Für die horizontale Achse ist dies die Breite des Textkorpus bzw. Frames (<dimension type=”width” ... />) und der Abstand zum linken (<position type=”left” ... />) bzw. rechten (<position type=”right” ... />) Seitenrand des Blattes. Für die vertikale Achse gilt das selbe in Bezug auf die Höhe des Textkorpus bzw. Frames und den Abstand zum oberen und unteren Seitenrand des Blattes. Das Vorkommen der Elemente *dimension* richtet sich demnach nach ihren Inhalten. Die Ausrichtung eines Textkorpus bzw. Frames lässt sich vollständig lediglich mit Inhalten des Elementes *position* beschreiben. Deshalb darf das Element *dimension* auch nicht vorkommen. Wenn man Höhe und Breite definiert, sind jedoch nur noch zwei Angaben für die Positionierung notwendig: eine für die horizontale- und eine für die vertikale Achse. Deshalb muss das Element *position* mindestens zweimal vorkommen. Damit es, zumindest nicht unabsichtlich, möglich ist, Werte doppelt zu vergeben, darf das Element *dimension* höchstens zweimal und das Element *position* höchstens viermal vorkommen. Damit können alle möglichen und sinnvollen Angaben getätigt werden. Dennoch ist es möglich, hier widersprüchliche Werte zu definieren. Dies ist dem Umstand geschuldet, dass sich mit XML Schema 1.0 keine sogenannten Co-Constraints bestimmen lassen. Also Strukturen, die nur in Abhängigkeit von gesetzten Werten anderer Strukturen Gültigkeit besitzen. Näheres beschreibt 6.2.

4.3.12 Element paper

Das Element *paper* ist ein leeres Element dessen Aufgabe es ist, als Kindelement des Elementes *parameter* Angaben zum zu verwendenden Seitenlayout zu machen. Dabei darf es aber nur in solchen Elementen *parameter* auftauchen, die selbst Kindelement

der Elemente *template* und *page* sind. Grund für diese Einschränkung ist, dass es für das Element *frame* keinen Sinn ergibt, Angaben zum Seitenlayout zu tragen. Der Name des Elementes wurde gewählt, weil es Angaben dazu macht, wie ein Dokument bzw. Abschnitt auf Papier zu drucken ist. Alternativ wurde noch der Name *print* diskutiert. Weil *print*, im Gegensatz zu allen anderen Elementen, jedoch ein Verb ist, wurde dies verworfen. Der Name *printer* kam nicht in Frage, weil er nicht hinreichend die Fuktion des Elementes wiederspiegelt. Somit wurde ganz bewusst in Kauf genommen, dass vom Namen her, nicht aber von der inneren Struktur her, eine gewisse Verwechselungsgefahr zum Element *page* besteht.

emphPaper verfügt, wie die Elemente *dimension* und *positio*n, über die obligatorischen Attribute *type* und *value*, jedoch nicht über das Attribut *unit*. Das Attribut *unit* ist an dieser Stelle nicht notwendig, weil die erlaubten Werte von *type* und *value* auch ohne Attribut *unit* für das Element *paper* eindeutig sind. Für das Attribut *type* sind die Werte *layout*, *orientation* und *format* zulässig. Für das Attribut *value* sind die Werte *oneside*, *twoside*, *portrait* und *landscape* sowie diverse Benennungen für gestattete Papierformate, z.B. *a5*, *a4*, *letter*, zulässig. Die Papierformate werden hier nicht im einzelnen beschrieben, weil dies den Rahmen sprengen würde. Sie lassen sich dem Schema im Anhang und dem beigefügten Datenträger entnehmen.

Der Wert *oneside* legt fest, dass der jeweilige Abschnitt, bzw. das gesamte Dokument einseitig erzeugt und gedruckt werden soll. Der Wert *twoside*, im Gegensatz dazu, legt einen zweiseitigen Druck des jeweiligen Abschnittes oder Dokumentes fest. Dies soll sich auch darauf auswirken, wie die jeweilige Implementierung die Angaben des Elementes *position* verarbeitet. Ein Attribute *type* mit dem Wert *left* soll sich in diesem Fall nicht auf den Abstand zum linken Seitenrand beziehen, sondern auf den Abstand zum inneren Seitenrand, an welchem sich die Bindung des Dokumentes befindet. Hat ein Attribut *type* den Wert *right*, soll der äußere Seitenrand als Bezug zur Positionierung des Texrahmens oder Frames herangezogen werden. Dieses Verhalten ist analog zum Verhalten von LaTeX bei Verwendung diesbezüglicher Angaben.

Das Element *paper* darf mindestens kein- und maximal dreimal unterhalb des Elementes *parameter* erscheinen. Dies rührt daher, dass für die Seitenausgestaltung jeweils nur maximal drei Angaben zweckmäßig sind. Dies trifft für den Fall zu, dass das Attribute emphtype für jedes vorkommende Element *paper* einen anderen Wert trägt. Bei dieser Definition besteht mit XML Schema 1.0 das selbe bereits für die Elemente *position* und *dimension* angedeutete Problem, der fehlenden Co-Constraints.

In der Implementierung des Template-Designers ist daher festgelegt, dass für das Attribut *value* die Werte *oneside* und *twoside* zulässig sind, wenn das Attribut *type* des selben Elementes *paper* den Wert *layout* trägt. Wenn das Attribut *type* den Wert *orientation* trägt, darf das zugehörige Attribut *value* entweder den Wert *portrait* oder *landscape* annehmen. Für den Wert *format* des Attributes *type* sind hingegen nur die angedeuteten Papierformate zulässig. Die Papierformate sind Benennungen für die wichtigsten

weltweit genormten Papierformate gemäß DIN476, ISO216, JIS P 0138-61 und ASME Y14.M-1995 sowie einiger traditioneller amerikanischer Papierformate. Nach Festlegung der gegenwärtigen Spezifikation des TemplateXML kam der Wunsch auf, auch direkte und von jeglicher Norm abweichende Größenangaben vorzunehmen. Diesem Wunsch wird jedoch erst in kommenden Versionen entsprochen.

4.4 Schlussfolgerung

Mit TemplateXML ist ein für seinen Verwendungszweck hin optimierter XML-Dialekt entstanden, der den gegenwärtigen Bedürfnissen voll entspricht. Jedoch ist abzusehen, dass sich hinsichtlich einiger Details wie dem Attribut *angle* des Elementes *content* oder der Definition der erlaubten Papierformate noch Änderungen ergeben werden. Weil der Template-Designer und die Komponenten zum Umwandeln des TemplateXML in LaTeX und zukünftig XSL-FO jedoch die einzigen Implementierung zur Nutzung darstellen, sind bei diesen Änderungen kaum Kompatibilitätsprobleme zu erwarten, weil deren Entwicklung mit der von TemplateXML Hand in Hand geht.

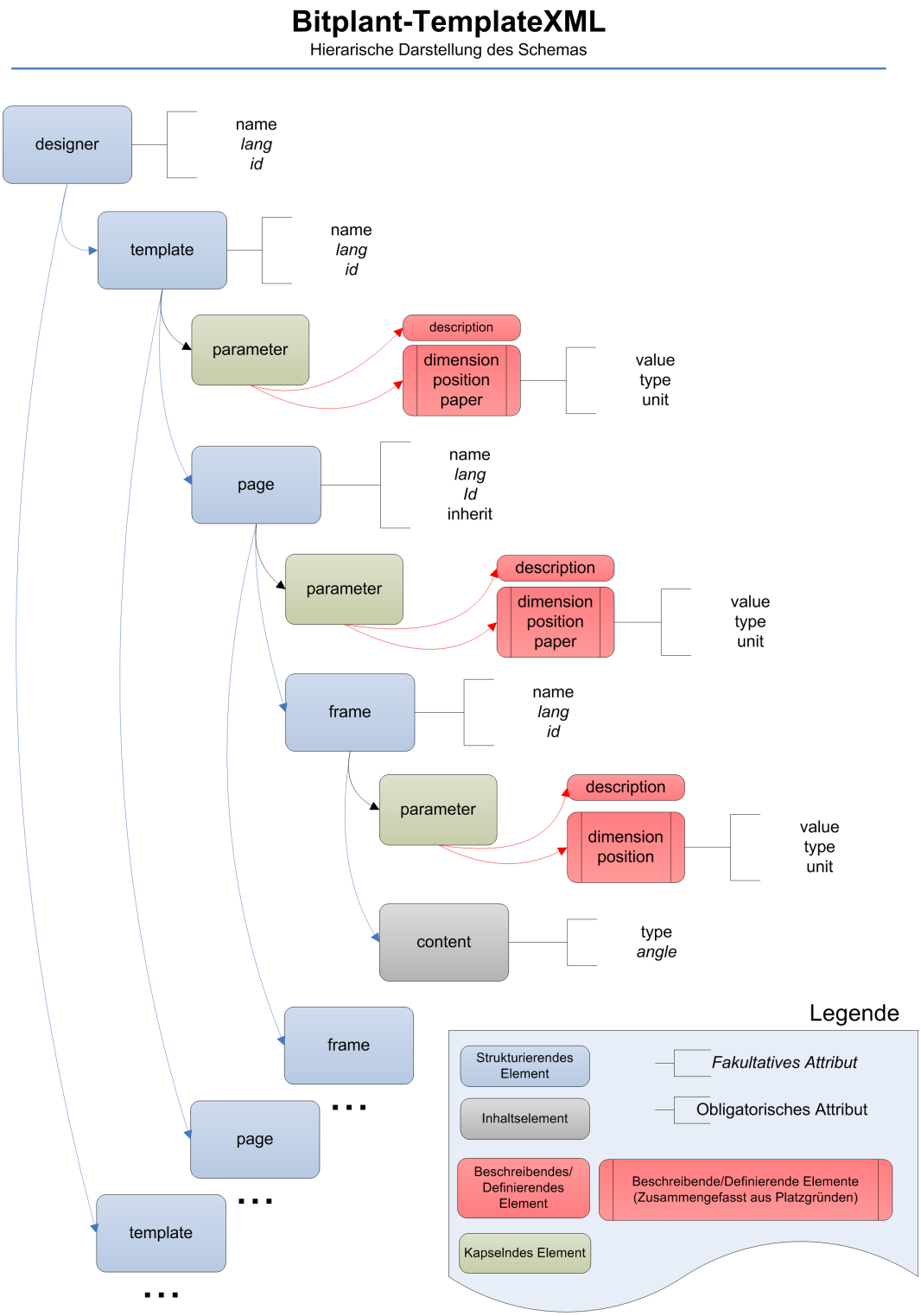

Abbildung 4.1: TemplateXML – Hierarchische Darstellung des Schemas

5 Template-Designer

5.1 Einführung

Im Folgenden wird auf relevante Punkte bei der Umsetzung des Template-Designer genannten Programms zur Erzeugung von Dokumentvorlagen zur Integration in Publikationsprozesse mit LaTeX und XSL-FO eingegangen. Dabei geht es im Besonderen um eine Beschreibung der umgesetzten Funktionalitäten und der verwendeten Werkzeuge.

5.2 Umsetzung

5.2.1 Begrüßungsdialog

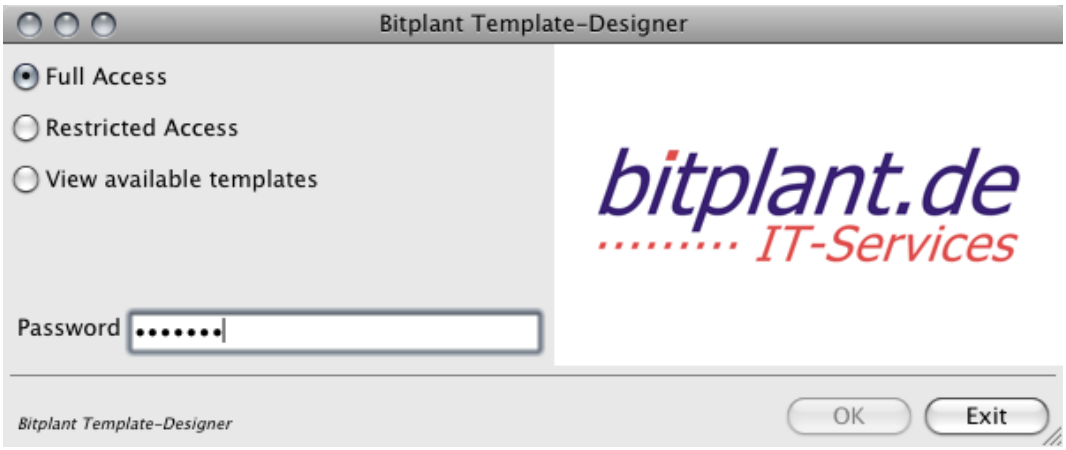

Abbildung 5.1: Begrüßungsdialog des Template-Designers

Nach dem Start des Template-Designers bekommt der Benutzer einen Begrüßungsdialog zu sehen, mit dessen Hilfe der Benutzer festlegen kann, in welchem Modus er den Template-Designer benutzen möchte. Dabei stehen drei Varianten zur Verfügung.

Der *Full Access* Modus erlaubt einem Benutzer das Anlegen, Verschieben, Bearbeiten und Löschen von Dokumentvorlagen. Dieser Modus hat eine primär administrative

Funktion, kann jedoch auch zur Bearbeitung von Dokumentvorlagen genutzt werden. Nur der *Full Access* Modus gestattet es, Einstellungen des Template-Designers zu ändern.

Der *Restriced Access* Modus dient dazu, vorhandene Dokumentvorlagen zu ändern. Dabei ist der Benutzer darauf beschränkt, die Inhalte der Elemente zur Seitenausgestaltung zu ändern. Es ist dem Benutzer in diesem Modus jedoch nicht gestattet, Elemente zur Seitenausgestaltung oder ganze Dokumentvorlagen anzulegen bzw. zu entfernen.

Der Modus *View available templates* gestattet dem Benutzer letztlich die Betrachtung der vorhandenen Dokumentvorlagen. Dieser Modus existiert, um die Einstellungen der einzelnen Dokumentvorlagen ohne Gefahr einer versehentlicher Manipulationen einzusehen.

Sollte für einen der drei Zugriffsmodi ein Passwort hinterlegt sein, aktiviert sich bei Auswahl des entsprechenden Modus das Feld zur Eingabe des benötigten Passworts. Erst wenn das benötigte Passwort korrekt eingegeben wurde, aktiviert sich die OK-Schaltfläche um den Template-Designer letztlich zu starten.

Die beschrieben Zugriffsmodi wurden umgesetzt, um die Möglichkeiten einer in das Programm integrierten Architektur zur rechteabhängigen Nutzung der einzelnen Funktionen des Template-Designers zu demonstrieren. Diese Architektur soll eine Anbindung an eventuell verfügbare Authentifizierungssysteme erleichtern, die häufig in größeren und stark arbeitsteiligen Technischen Redaktionen anzutreffen sind. Außerdem dient die aktuelle Umsetzung natürlich auch schon dem konkreten Ziel, den Zugriff auf die mit dem Template-Designer verwalteten Dokumentvorlagen nach Bedarf zu beschränken.

Die Graphik im rechten Bereich des Begrüßungsdialogs kann auf einfache Weise durch eine individuelle Graphik ersetzt werden, wenn dies durch den Benutzer erwünscht wird. So kann beispielsweise den Vorgaben einer firmeninternen Coporate Identity entsprochen werden.

5.2.2 Hauptfenster

Nachdem ein Benutzer den Begrüßungsdialog erfolgreich hinter sich gelassen hat, startet das Hauptfenster des Template-Designers. In diesem Fenster tätigt der Benutzer alle Aktionen, die relevant für die Verwaltung der Dokumentvorlagen sind. Das Hauptfenster ist in zwei Bereiche untergliedert.

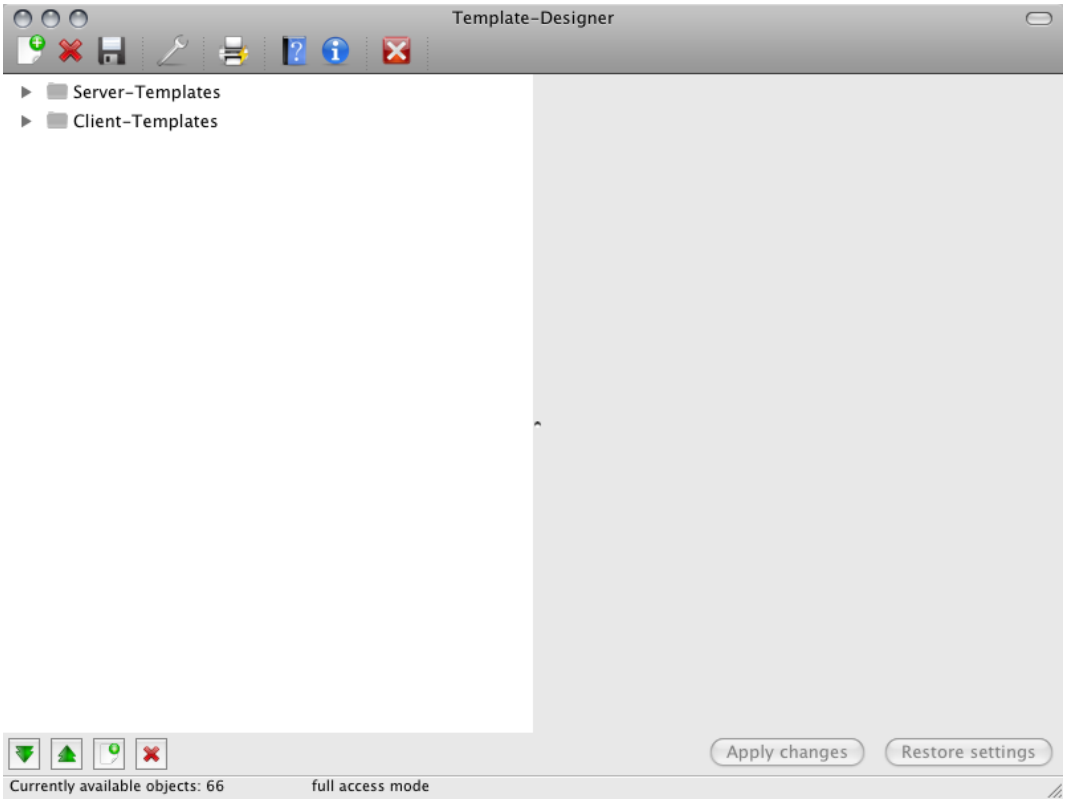

Abbildung 5.2: Template-Designer nach dem Start des Hauptfensters mit vollen Zugriffsrechten

5.2.3 Baumstruktur für Navigation

Im linken Bereich des Hauptfensters befindet sich ein sogenanntes TreeControl [26]. Dieses dient dazu, alle vorhandenen Dokumentvorlagen anzuzeigen und anzusteuern. Das TreeControl besteht aus ausklappbaren Listeneinträgen. Dieses Verhalten ist analog der Verästelung eines Baumes, die ebenfalls von einem ursprünglichen Element, dem Stamm, ausgehend immer feingliedriger wird. Auf der obersten Ebene des TreeControls befinden sich zwei Listeneinträge, die die Ablageorte der verwalteten Dokumentvorlagen repräsentieren. Die Namen der Listeneinträge, *Server-Templates* und *Client-Templates*, verraten dabei, für welchen Ablageort der Dokumentvorlagen die Einträge jeweils stehen.

Direkt unterhalb der Ablageorte befinden sich die verschiedenen Dokumentvorlagen und Zusammenstellungen der Dokumentvorlagen. An dieser Stelle wird auch ersichtlich, wie das TemplateXML-Element *designer* innerhalb des Template-Designers umgesetzt wird. Alle Dokumentvorlagen unterhalb einer Zusammenstellung befinden sich

auf XML-Ebene unterhalb eines Elementes *designer* und dementsprechend auch im TreeControl unterhalb einer eingezogenen Ebene, welche dieser Zusammenstellung entspricht. Bei TemplateXML-Dateien mit dem Wurzelelement *template* wird die Dokumentvorlage ohne die zusätzlich Ebene einer Zusammenstellung dargestellt.

Unterhalb der Dokumentvorlagen sind die jeweils vorhandenen Dokumentabschnitte zu finden. Diese entsprechen auf TemplateXML-Ebene jeweils einem Element *page*. Und letztlich befinden sich unterhalb eines Dokumentabschnittes alle für den jeweiligen Abschnitt gültigen Elemente zur Seitenausgestaltung, die dem XML-Element *frame* entsprechen. Weil die durch diese Anordnung insgesamt bis zu fünf anzutreffenden Verschachtelungsebenen drohten, auf Grund der schieren Menge an Information zu komplex zu werden, steht jedem Listeneintrag ein Symbol zur Seite, das dessen Typ identifiziert. Dies soll helfen, die Übersicht zu bewahren.

Um die Navigation innerhalb der Baumstruktur weiter zu vereinfachen, befinden sich unterhalb des TreeControls zusätzliche Elemente zur Steuerung. Diese dienen dazu, die gesamte Baumstruktur aus- bzw. einzuklappen und neue Elemente anzulegen bzw. bestehende Elemente zu entfernen, sofern dies durch den im Begrüßungsdialog gewählten Zugriffsmodus gestattet ist. Die Funktionen zum Aus- und Einklappen der Baumstruktur sollen helfen, dem Benutzer bei Bedarf einen schnelleren Zugriff auf die gewünschten Elemente zu gewähren. Entweder durch einen schnellen Blick auf alle verfügbaren Elemente oder um eine größere Übersicht zu gewinnen, indem man nach dem Zuklappen aller aufgeklappten Elemente nur jene Elemente öffnet, die gerade benötigt werden.

Neue Elemente innerhalb der Baumstruktur werden an jener Stelle erzeugt, die selektiert ist, wenn die Schaltfläche zur Erzeugung eines neuen Elementes gewählt wurde. Das bedeutet konkret, dass eine neue Zusammenstellung erzeugt wird, wenn vorher ein einen Ablageort repräsentierendes Baumelement selektiert war. Ein neuer Dokumentabschnitt wird unterhalb der jeweils ausgewählten Dokumentvorlage erzeugt. Sollte ein Element zur Seitenausgestaltung ausgewählt worden sein, wird beim Wählen der Schaltfläche zur Erzeugung eines neuen Elementes ein benachbartes Element zur Seitenausgestaltung erzeugt. Dieses Verhalten ist deshalb anderes, als das der übergeordneten Elemente, weil es unterhalb der Elemente zur Seitenausgestaltung (TemplateXML-Element *frame*) keine weiteren Strukturen gibt. Würden sich die übergeordneten Elemente ähnlich verhalten, wie die Elemente zur Seitenausgestaltung, wäre es nicht möglich neue Kindelemente anzulegen ohne eine weitere Schaltfläche für diese Funktion einführen zu müssen. Deshalb ist diese Lösung die praktikabelste.

Soll ein Element entfernt werden, wird ein Dialog zur Bestätigung des Vorgangs eingeblendet, um einem versehentlichen Entfernen vorzubeugen. Verfügt z.B. eine Dokumentvorlage über weitere Dokumentabschnitte und Elemente zur Seitenausgestaltung, so werden diese ebenfalls entfernt.

Mit Ausnahme der Ablageorte ist jedes Element der Baumstruktur per Drag&Drop verschiebbar. Dabei dürfen Elemente nur dorthin verschoben werden, wo diese gemäß Ihrem Typ auch erlaubt sind. Das meint, dass ein Dokumentabschnitt nur an eine Position unterhalb einer Dokumentvorlage gezogen werden darf usw. Dies soll den Umgang mit den Dokumentvorlagen und der Baumstruktur weiter vereinfachen.

Jedes Element der Baumstruktur verfügt über ein Kontextmenü, dass die selben Aktionen zulässt, wie die Schaltflächen unterhalb der Baumstruktur. Diese Option ermöglicht es dem Benutzer, so mit der Baumstruktur zu arbeiten, wie er dies gewohnt ist, oder erwartet. Würde dem Benutzer nur eine einzige Option zur Manipulation der Baumstruktur zur Verfügung stehen, stünde zu befürchten, dass der Aufwand zur Einarbeitung in das Programm größer ausfiele. Das sollte vermieden werden.

5.2.4 Einstellungen zum jeweiligen Element

Im rechten Bereich des Hauptfensters werden all jene Einstellungen angezeigt, die für das jeweils selektierte Element gültig sind. Diese Einstellungen entsprechen, wie nicht anders zu erwarten, jenen Kindelementen und Attributen, die für das jeweilige TemplateXML-Element verfügbar sind.

Für die Elemente zur Darstellung der Ablageorte existieren im rechten Bereich des Hauptfensters keinerlei Elemente zur Justage irgendwelcher Einstellungen. In Frage kämen hier lediglich Einstellungen der als Ablageort für die Dokumentvorlagen genutzten Vezeichnisse. Für diese Einstellungen wurde jedoch der ebenfalls verfügbare Einstellungsdialog gewählt. Dies geschah, weil eine Veränderung des Ablageortes ein wohl eher seltener Vorgang ist, der es nicht verdient, an solch prominenter Stelle innerhalb des Hauptfensters zugänglich zu sein.

Wählt der Benutzer eine Zusammenstellung aus, kann er den Namen der Zusammenstellung ändern, sofern er sich im *Full Access*-Modus befindet. Andernfalls ist diese Einstellung deaktiviert. Weitere Möglichkeiten zur Manipulation einer Zusammenstellung stehen nicht zur Verfügung. Dies hat als Ursache, dass TemplateXML selbst keine weiteren diesbezüglichen Einstellungen zulässt.

Für Dokumentvorlagen, Dokumentabschnitte sowie Elemente zur Seitenausgestaltung lassen sich deren Name und eine Beschreibung des jeweiligen Elementes verändern. Der Name entspricht dabei dem TemplateXML-Element-Attribut *name*. Die Beschreibung dem TemplateXML-Element *description*. Befindet sich der Template-Designer im Modus *Restriced Access*, so lässt sich lediglich die Beschreibung manipulieren. Dies rührt daher, dass der Name des jeweiligen Elementes für die Integration der Dokumentvorlage in den Publikationsprozess von Bedeutung ist. Anhand des Namens wird dabei entschieden, wann der jeweilige Abschnitt bzw. das jeweilige Element verwendet werden soll.

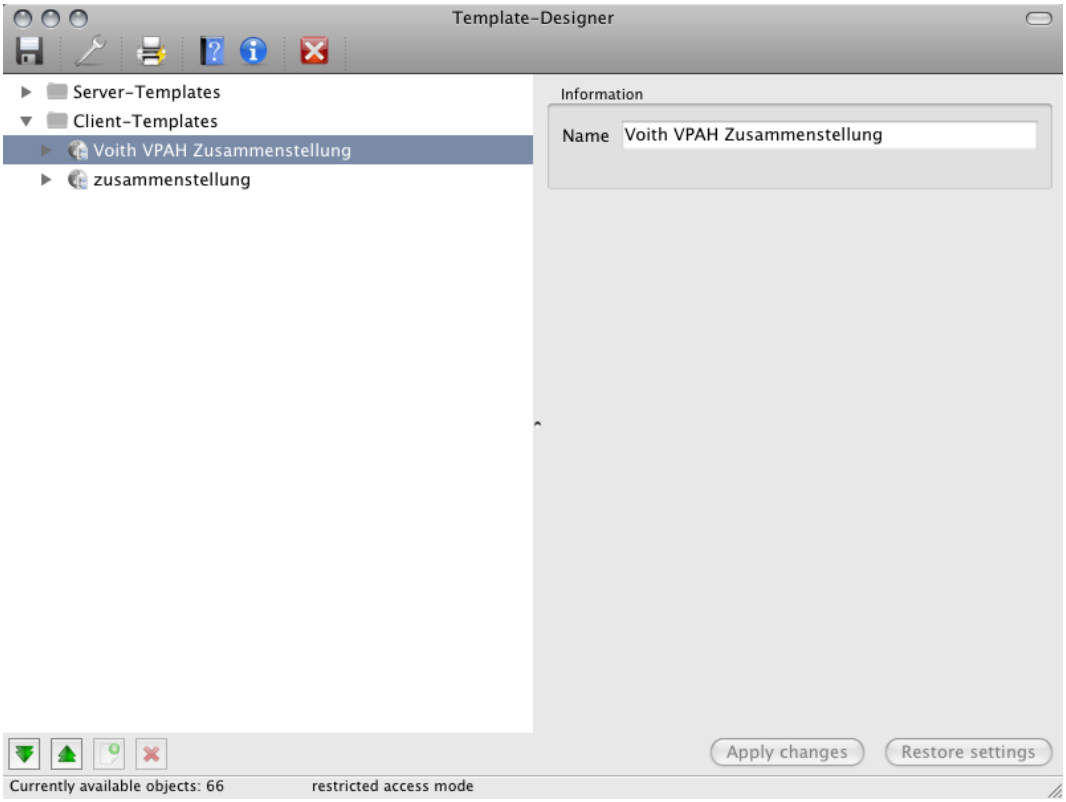

Abbildung 5.3: Template-Designer mit beschränkten Zugriffsrechten und sichtbaren Einstellungen für eine Zusammenstellung

Das Element *description* darf zwar gemäß TemplateXML-Definition nicht vorhanden sein, jedoch zeigt der Template-Designer einen Hinweis in warnender roter Schrift an, wenn der Benutzer keinen beschreibenden Text zu einem Element angegeben hat. Der Hinweis fordert den Benutzer auf, bitte eine Beschreibung abzugegeben. Der Template-Designer verhält sich so, weil interne Tests zeigten, dass Benutzer dazu tendieren, die Beschreibung als wenig relevant zu betrachten und dennoch Namen für benötigte Elemente wählten, die in manchen Fällen nicht vollständig eindeutig waren. Dieses Verhalten kann spätestens dann problematisch werden, wenn ein neuer Benutzer eine bestehende Dokumentvorlage ändern muss und dabei durch eine fehlende Beschreibung zumindest länger suchen muss, um das zu verändernde Element zu finden. Das Verhalten des Template-Designers in dieser Situation ist ein Kompromiss aus notwendiger Vollständigkeit der Daten und möglichst geringer Gängelei des Benutzers.

5.2.5 Einstellungen für Dokumentvorlagen

Für Dokumentvorlagen lassen sich entsprechend den Möglichkeiten von TemplateXML für die Dokumentvorlage global gültige Einstellungen bezüglich des zu nutzenden Papierformats sowie der Größe und Position des Textrahmens mit den eigentlichen Inhalten vornehmen. Dazu befinden sich unterhalb des Textfeldes zur Beschreibung der Dokumentvorlage weitere Kontrollstrukturen mit denen sich die entsprechenden Einstellungen vornehmen lassen. Die Kontrollstrukturen zur Positionierung und Größe beziehen sich im Falle der Dokumentvorlagen analog zur Vorgabe von TemplateXML (4.3.9, 4.3.10) auf den Textrahmen mit den eigentlichen Inhalten und nicht auf die Größe des Papiers.

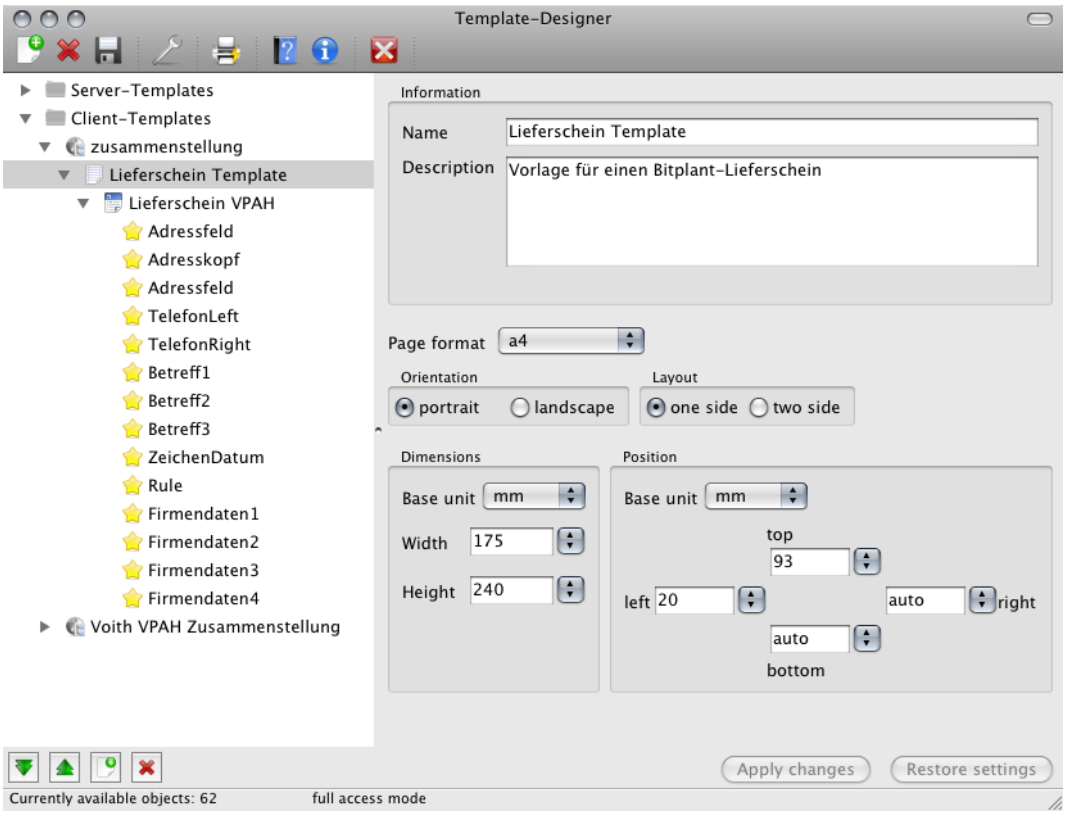

Abbildung 5.4: Hauptfenster des Template-Designers mit sichtbaren Einstellungen für eine Dokumentvorlage

Wegen des in 4.3.11 beschriebenen Umstandes, dass die vollständige Definition der Größe und Positionierung eines Rahmens weniger Angaben erfordert als zur Verfügung stehen, verhalten sich die Kontrollstrukturen zur Bestimmung jener Angaben ähnlich dem Textfeld zur Eingabe einer Beschreibung. Sollte die horizontale- bzw. vertikale Achse eines Rahmens über- bzw. unterdefiniert sein, erscheint ein Hinweis in roter

Schrift, um den Benutzer über diesen Umstand zu informieren. Alternativ wurde überlegt, einen warnenden Dialog an statt des Hinweises zu nutzen. Dies wurde jedoch verworfen, weil – wie beim Textfeld für eine Beschreibung – davon ausgegangen wird, dass der Benutzer weiß, was er tut. Es bestand ehedem die Gefahr, dass der Dialog weg geklickt wird, ohne ihn zu lesen.

5.2.6 Einstellungen für Dokumentabschnitte

Die Kontrollstrukturen zur Justage eines Dokumentabschnittes entsprechen exakt jenen einer Dokumentvorlage. Jedoch mit der Einschränkung, dass ein zusätzliche Checkbox existiert, über die sich die Vererbung der Einstellungen der übergeordneten Dokumentvorlage steuern lässt. Diese Checkbox regelt den Wert des Attributes *inherit* des zugehörigen Elementes *page*. Sollte die Vererbung der Einstellungen der Dokumentvorlagen aktiviert sein, so sind die Kontrollstrukturen zur Einstellung von Papierformat sowie Größe und Positionierung des Textrahmens mit den eigentlich zu publizierenden Inhalten deaktiviert, also ausgegraut. Schaltet der Benutzer mittels der Checkbox die Vererbung der Einstellungen der Dokumentvorlage aus, so werden die Kontrollstrukturen aktiviert.

5.2.7 Einstellungen für Elemente zur Seitenausgestaltung

Weil gemäß der Definition des TemplateXML vier verschiedene Inhaltstypen für das Element *frame* vorgesehen sind (über das Attribut *type*), existieren auch vier verschiedene Möglichkeiten ein Element zur Seitenausgestaltung einzurichten. Dazu existiert unterhalb des Textfeldes zur Beschreibung des Elementes eine DropDown-Liste, die den verschiedenen Inhaltstypen entspricht. Abhängig vom gewählten Inhaltstyp dieser DropDown-Liste zeigt der Template-Designer unterschiedliche Kontrollstrukturen an, um möglichst komfortabel die entsprechenden Einstellungen vorzunehmen. Für die Elemente zur Seitenausgestaltung stehen ebenfalls Kontrollstrukturen zur Manipulation von Größe und Positionierung bereit. Diese beziehen sich jedoch in diesem Fall auf Größe und Positionierung das Elementes selbst.

Für statische Inhalte, variable Inhalte und Bildinhalte existiert eine weitere Kontrollstruktur zum Drehen der jeweiligen Inhalte. Diese entspricht dem Attribut *angle* des TemplateXML-Elementes *content*. Elemente zur Seitenausgestaltung, die lediglich einen farbigen Inhalt bereitstellen existiert diese Kontrollstruktur nicht, weil die Rotation einer einfachen farbigen Fläche keinen Sinn ergeben würde. Der Template-Designer interpretiert in diesem Fall die fehlende Angabe als Vorgabewert, der gemäß der TemplateXML Definition *0* lautet und somit keine Rotation vorsieht.

Statische Inhalte können in ein Textfeld, ähnlich dem zur Eingabe der Beschreibung des Elementes, eingetragen werden. Variable Inhalte werden ebenfalls in ein solches Text-

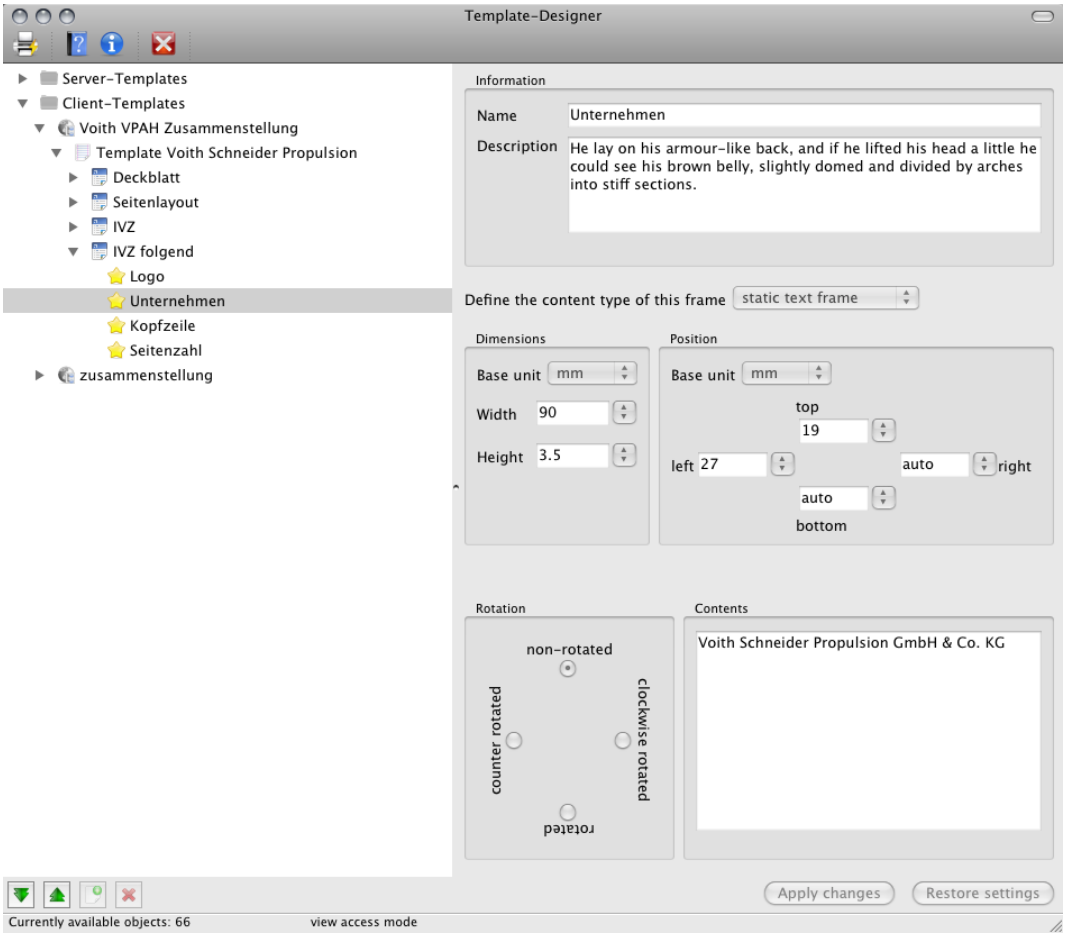

Abbildung 5.5: Template-Designer ohne Zugriffsrechte mit Einstellungen eines Elementes zur Seitenausgestaltung mit statischen Inhalten

feld eingetragen, jedoch existieren für diesen Fall zwei zusätzliche Funktionen in Form von Schaltflächen und zusätzlichen Dialogen. Der Template-Designer bietet für variable Inhalte zwei Dialoge, mit denen Platzhalter für variable Inhalte auf möglichst einfache Art und Weise ausgewählt werden können. Ein Dialog verfügt über Platzhalterzeichen zur Definition von Datums- und Zeitwerten. Für diesen Dialog wurden jene Datums- und Zeitwerte in den Template-Designer integriert, die gemäß *Iso 8601:2004(E), Third edition* [30] definiert sind. Dies wurde umgesetzt, um eine möglichst große Kompatibilität mit anderen Anwendungen sicher zu stellen, die zukünftig eventuell auf die mit dem Template-Designer erzeugten XML-Daten zugreifen. Die jeweils gewünschten Platzhalter sind in einer Baumstruktur ähnlich jener zur Navigation innerhalb der verschiedenen Dokumentvorlagen organisiert, um die Vielzahl an möglichen Platzhaltern besser überblicken zu können. Jedoch unterscheidet sich die Darstellung dahingehend, dass für die Platzhalter jeweils ein zusätzliches Beispiel notiert ist, um das Ergebnis der

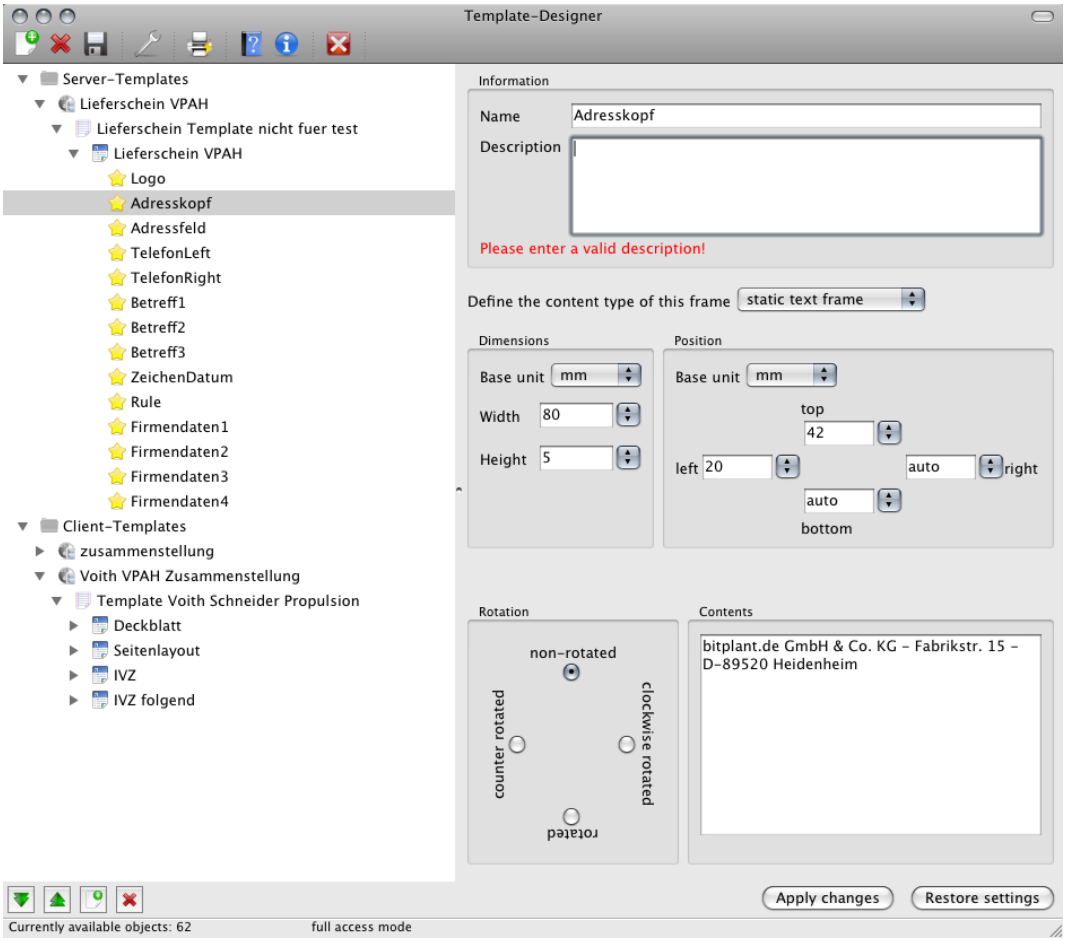

Abbildung 5.6: Bestimmen von statischen Inhalten eines Elementes zur Seitenausgestaltung

Ausgabe besser abschätzen zu können. Die Platzhalter werden durch einen einfachen Doppelklick auf den entsprechenden Platzhalter in das Textfeld für die variablen Inhalte übernommen. Ferner existiert eine zweite Gruppe von Platzhaltern, in der Platzhalter zu finden sind, die sich mit der jeweiligen Dokumentstruktur beschäftigen. Dazu gehören besonders diverse Notationen zu aktuellen Seiten- und Kapitelzahlen. Diese Platzhalter folgen keiner Norm oder Vorgabe, weil im gegebenen Zeitraum nichts entsprechendes gefunden werden konnte.

Für Bildinhalte existiert eine eigene Kontrollstruktur, die lediglich aus einer Schaltfläche zur Auswahl einer Bilddatei besteht. Hat der Benutzer eine Bilddatei ausgewählt, wird versucht, diese Bilddatei darzustellen. An dieser Stelle kommen die in 5.4.3 und 5.4.4 genannten externen Komponenten Ghostscript sowie die Python Image Library zum Einsatz.

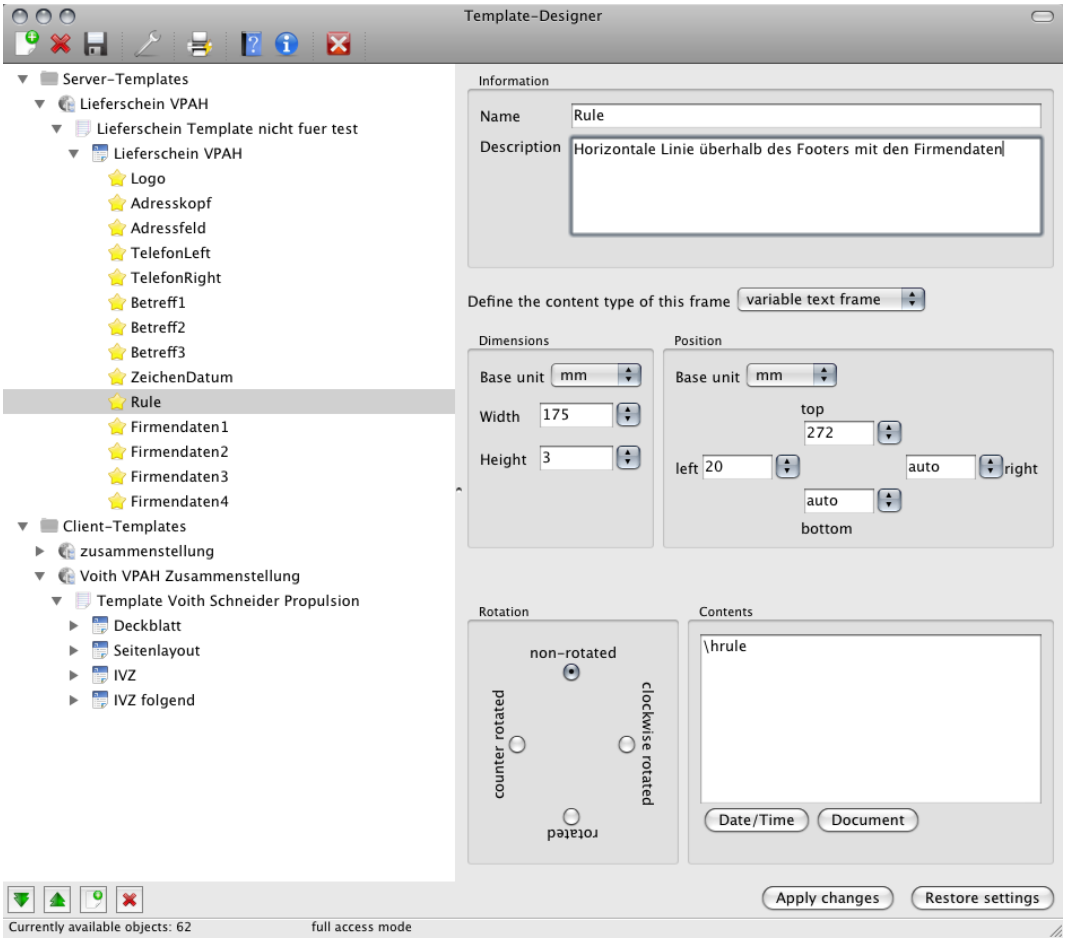

Abbildung 5.7: Bestimmen von variablen Inhalte eines Elementes zur Seitenausgestaltung

Für Farbinhalte existieren die komplexesten Kontrollstrukturen. Diese ermöglichen es, je nach Bedarf, Farbwerte für den Cmyk und Rgb-Farbraum einzutragen. Der Template-Designer versucht während der Eingabe automatisch, die angegeben Werte in den jeweils anderen Farbraum zu übertragen und anzuzeigen. Zusätzlich existiert eine Schaltfläche, um den plattformeigenen Dialog zur Auswahl eines Farbwertes aufzurufen. Dies ist vor allem dann interessant, wenn diesem Dialog bereits gespeicherte Farbwerte aus anderen Anwendungen vorhanden sind, die sich somit einfach übernehmen lassen.

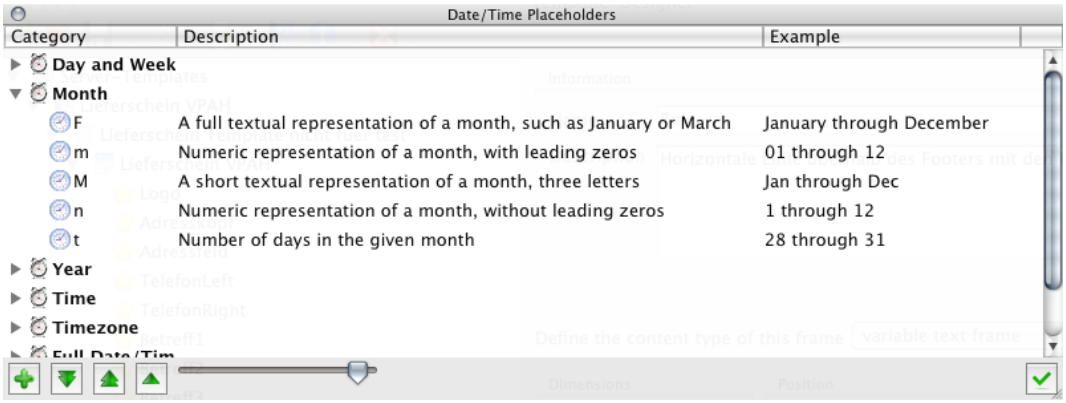

Abbildung 5.8: Dialog zum Einstetzen von Platzhaltern für Datums- und Zeitangaben gemäß Iso8601

5.2.8 Einstellungsdialog

Der Einstellungsdialog des Template-Designers beinhaltet Möglichkeiten zum Setzen von Passwörtern für die verschiedenen Zugriffsmodi sowie zur Justage der Ablageorte für die Dokumentvorlagen. Das Einstellen der Ablageorte ist zum Beispiel sinnvoll, wenn man die Dokumentvorlagen in eine bereits bestehende Strategie zur Datensicherung integrieren möchte. Einstellbar sind dabei lokale wie entfernte Verzeichnisse für die Dokumentvorlagen selber und die Verzeichnisse für Graphikdateien, die von den Dokumentvorlagen genutzt werden. Die Voreinstellung der Ablageorte sieht vor, dass die Ablageorte im jeweiligen Benutzerverzeichnis zu finden sind. Wenn mehrere Benutzer auf einer Plattform arbeiten oder eine zentrale Netzwerkfreigabe für die Verwaltung der Dokumentvorlagen genutzt werden soll, können diese Vorgaben entsprechend umgesetzt werden.

Passwörter werden gesetzt, indem diese zweimalig wiederholt werden müssen. Dabei müssen diese Passwörter sowohl Buchstaben als auch Ziffern enthalten und mindestens fünf Zeichen lang sein. Der Dialog zum Setzen von Passwörtern ist derzeit nur für die Plattform Mac OS X verfügbar, weil diese Priorität bei der Umsetzung des Template-Designers genoss und der gesteckte Zeitrahmen die Implementierung einer voll funkionsfähigen Passwortverwaltung für andere Plattformen verhinderte.

5.2.9 Menüs und Werkzeugleisten

In den Menüs und Werkzeugleisten des Template-Designers finden sich die selben Optionen zur Verwaltung der Dokumentvorlagen, die bereits mit dem TreeControl und dem Einstellungsdialog besprochen wurden. Hinzu kommt jedoch die Fähigkeit diese

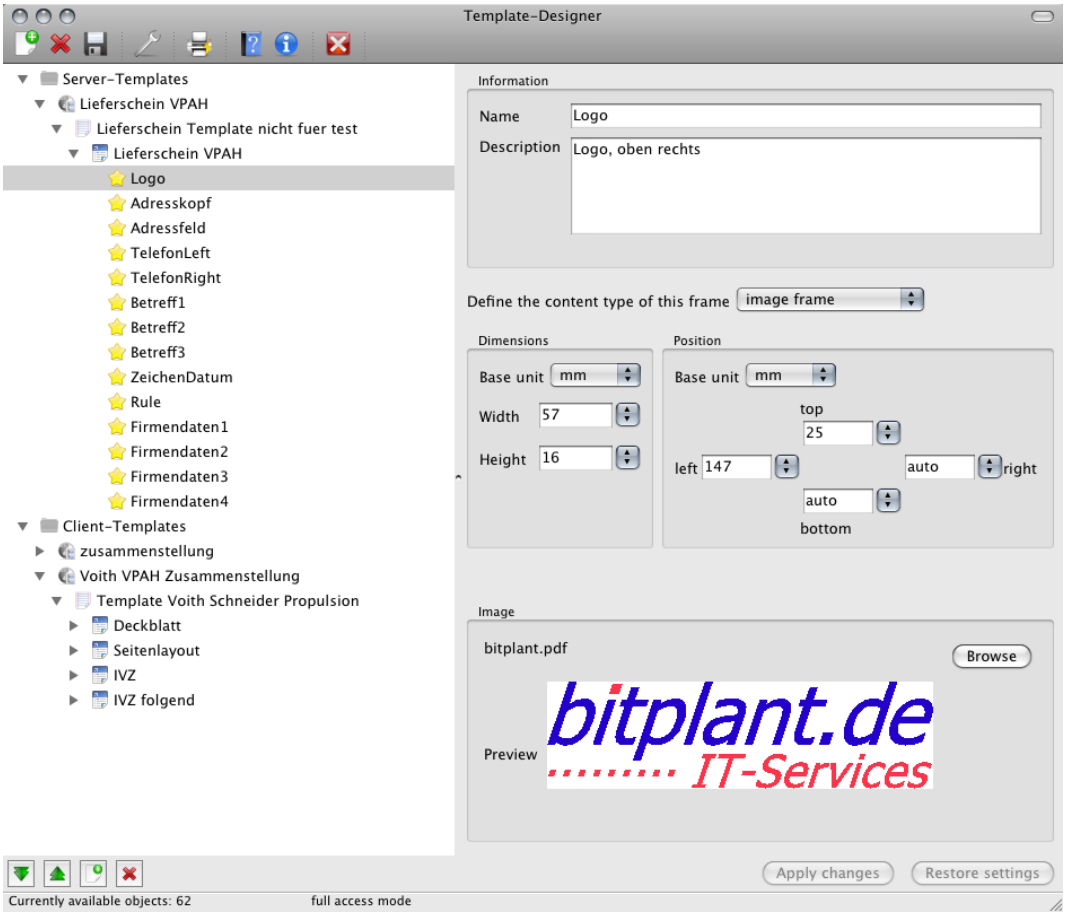

Abbildung 5.9: Bestimmen einer Bilddatei für einen Frame

Funktionen nicht nur mit der Maus, sondern auch mittels Tastenkombinationen anzusprechen. Dabei wurde versucht die in [54] formulierten Richtlinien umzusetzen. Es gab Fälle jedoch Fälle, zu denen [54] keinerlei Angaben machte. In diesen Fällen wurden Tastenkombinationen und Positionen für die jeweiligen Menüeinträge gewählt, die auch von einer überwiegenden Zahl anderer Programme verwendet werden.

5.3 Lizenz und Gedanken zu OpenSource

Um den Template-Designer im gesteckten Zeitrahmen umsetzen zu können, bedurfte es des Rückgriffs auf bereits existierende Software-Komponenten. Dabei wurde besonders auf deren freie Verfügbarkeit im Sinne von OpenSource und deren Plattformunabhängigkeit geachtet, weil der Template-Designer selbst diesen Ansprüchen unterworfen ist.

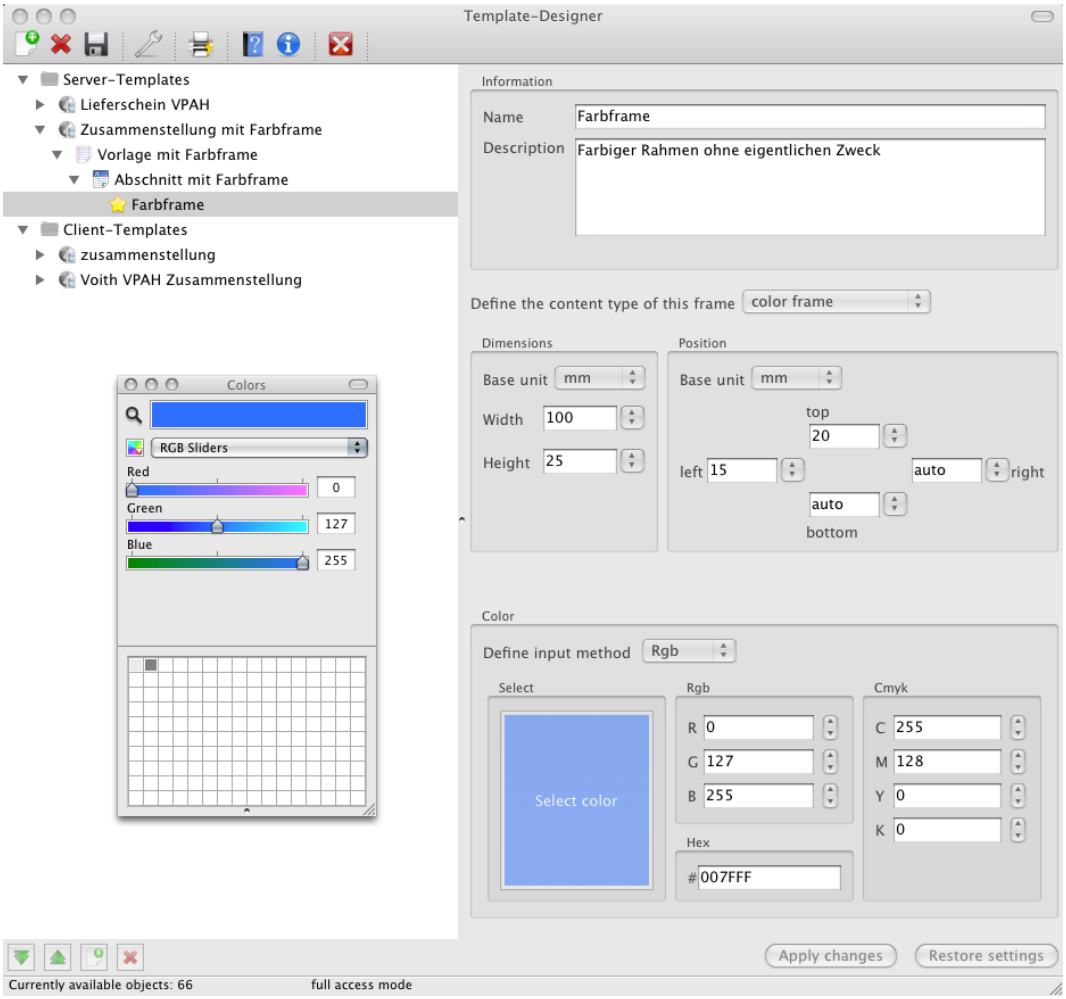

Abbildung 5.10: Bestimmen der Farbe eines Frames

Mit OpenSource ist gemäß [70] gemeint, dass die Komponenten im menschenlesbaren Quelltext vorliegen, die Komponenten frei und ohne zusätzliche Kosten weiterverteilt werden dürfen, die Komponenten an die eigenen Bedürfnisse angepasst werden dürfen und diese Regeln für jedermann gelten. Sinn und Zweck dieser Ansprüche ist es, dass die Entwicklung des Template-Designers und des gesamten TeXML-Publishing-Servers unabhängig vom Wohl und Wehe eines einzelnen Softwareherstellers verlaufen kann und die Verteilung des Template-Designers unabhängig von den Ansprüchen eines dritten Softwareherstellers erfolgt. Ferner erlaubt die Verwendung von derart lizenzierten Komponenten die Anpassung selbiger an die individuellen Bedürfnisse des Template-Designers. Zuletzt hat das Vorgehen den Vorteil, dass es die zukünftige Fortentwicklung des Template-Designers durch Dritte beschleunigen kann, weil die Teilnahme an

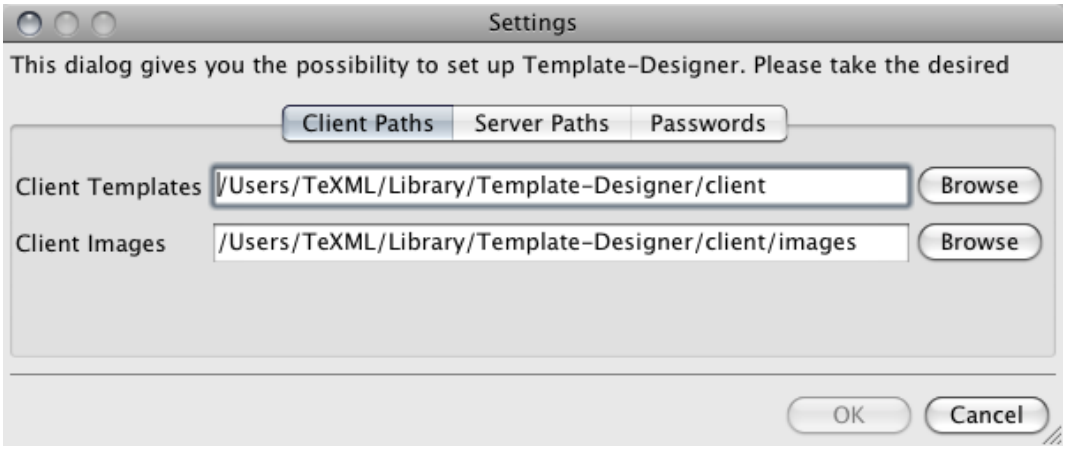

Abbildung 5.11: Festlegen des Ortes zur Speicherung lokaler Dokumentvorlagen

der Entwicklung nicht an die Bedingungen eines weiteren Softwareherstellers gebunden oder mit zusätzlichen Kosten verbunden ist.

Die Plattformunabhängigkeit soll gewährleistet sein, damit der Template-Designer zukünfitg und bei Bedarf an andere Plattformen, wie beispielsweise Linux oder Microsoft Windows, angepasst werden kann. Auch dies stärkt das Fortbestehen der Anwendung, weil zukünftige Entwickler nicht auf eine bestimmte Plattform gezwungen werden.

Der Template-Designer selbst steht unter der sogenannten MIT-Lizenz. Diese Lizenz erlegt dem Lizenznehmer keinerlei Pflichten oder Einschränkungen im Umgang mit der Software auf. Gleichzeitig werden die Autoren der Software von jeglicher Verantwortung in Bezug auf mögliche Probleme mit der Software freigesprochen. Bemerkenswert ist zusätzlich, dass diese Lizenz sozusagen auf Dateiebene wirkt. Das bedeutet, dass die Lizenz für jede einzelne Datei gilt, somit jede einzelne Datei als eigenständiges Werk gilt, auch wenn sie ihre volle Funktionsfähigkeit erst in Kombination mit anderen Dateien ausspielen kann. Das ist insoweit bemerkenswert, als das die Lizenz dadurch auch in jeder einzelnen Datei erscheinen muss. Dies unterscheidet die Lizenz von anderen OpenSource-Lizenzen, bei denen die Lizenz per Festschreibung einmalig für das jeweilige gesamte Projekt gilt. Die Festlegung der Lizenz auf Dateiebene ermöglicht es zusätzlich, dass bestimmte Quelltexte bei Bedarf unter einer anderen Lizenz erscheinen könnten. Dies trifft beispielsweise auf die verwendete Drittanbieter-Software keychain.py zu.

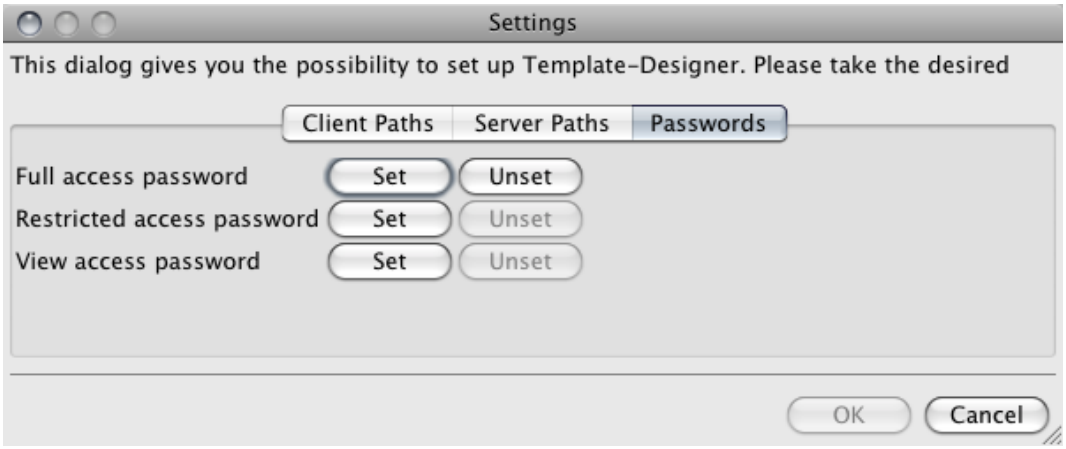

Abbildung 5.12: Festlegen von Passworten für einen beschränkten Zugriff auf die Dokumentvorlagen

```
1  Copyright (c) 2008 Bitplant.de
2
3  Permission is hereby granted, free of charge, to any person
4  obtaining a copy of this software and associated documentation
5  files (the "Software"), to deal in the Software without restriction,
6  including without limitation the rights to use, copy, modify, merge,
7  publish, distribute, sublicense, and/or sell copies of the Software,
8  and to permit persons to whom the Software is furnished to do so,
9  subject to the following conditions:
10
11 The above copyright notice and this permission notice shall be
12 included in all copies or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
15 KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
16 WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
17 PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
18 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
19 DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
20 CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
21 CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
22 DEALINGS IN THE SOFTWARE.
```

Listing 5.1: MIT-Lizenz des Template-Designers

5.4 Genutzte Drittanbieter-Software

5.4.1 Python

Der Template-Designer wurde in der Programmiersprache Python geschrieben. Bezüglich der Projektdefinition gab es keine Einschränkungen bzw. Festlegungen zur zu verwendeten Programmiersprache. Da die prinzipielle Plattformunabhängigkeit des Publishing-Servers erhalten bleiben sollte, standen die Programmiersprachen Java, Objective-C, Ruby und Python zur Wahl. Diese Auswahl ermöglicht die Erstellung von Programmen, die prinzipiell auf den Plattformen Mac OS X, Linux, BSD-Derivaten sowie Windows kompilier- bzw. interpretierbar sind.

Die Wahl viel auf Python, weil dieses in dem Ruf steht, leicht erlernbar zu sein. Python wird von der Python Software Foundation [38] entwickelt. Auf Grund der geringen Zahl von Schlüsselwörtern, seiner Übersichtlichkeit und einfachen Syntax, ist in Python geschriebener Quelltext schnell verstehbar [67, A Python Q&A Session - Why Do People Use Python][74, 2.3 Einsatzmöglichkeiten und Stärken]. Dies war deshalb Vorraussetzung, da das Projekt andernfalls nicht im gesetzten Zeitrahmen bewältigbar erschien. Ferner ist Python modularisierbar, was die Möglichkeit bietet, auf einfache Art und Weise Funktionen aus externen Quellen nutzen zu können. Ein Blick in die von den Python-Programmierern betriebenen Mailing-Listen machte deutlich, dass Python über eine große Entwicklergemeinde verfügt, was bei Problemen schnelle Hilfe versprach [67, A Python Q&A Session - Why Uses Python Today?][37]. Zuletzt wurde Python jedoch gewählt, weil bereits andere Teile des TeXML-Publishing-Servers in dieser Programmiersprache verfasst wurden. Dies gewährleistet zukünftig eine einfacherer Wartung des Template-Designer, weil das Verständnis der vorhandenen Quelltexte durch die Entwickler bereits gegeben ist. Hinzu kommt der Vorteil, dass Python unter einer OpenSource-Lizenz steht, die dessen freie Nutzung und Verteilung sowohl in privaten wie kommerziellen Projekten ohne Einschränkungen erlaubt [36].

Objective-C [55] schied als Programmiersprache wegen dessen Verwandtschaft zu C++ aus. Dieses hatte sich in kleineren früheren Projekten bei mir den Ruf erworben hat, komplex und schwer verstehbar zu sein. Jedoch hätte Objective-C den Vorteil gehabt, dass sich ein damit entwickeltes Projekt hervorragend in Apples Mac OS X integriert hätte, weil das von Apple verwendete Framework zur Erstellung graphischer Programme, Cocoa, selbst in Objective-C verfasst ist.

Java [56] hätte den Vorteil gehabt, dass das damit erstellte Programm auf Grund der Architektur von Java ohne erneute Übersetzung der Quelltexte plattformunabhängig ausführbar gewesen wäre. Ferner wäre von Vorteil gewesen, dass keine weitere externe Komponente zur Darstellung der graphischen Oberfläche des Template-Designers notwendig gewesen wäre, weil diese Funktionen bereits Teil des Java-Interpreters sind. Jedoch schied Java aus, weil sich die eben genannte graphische Oberfläche nur schlecht

an das jeweils genutzte Betriebssystem anpasst und die graphischen Komponenten von Java zudem bei vielen Anwendern einen unvorteilhaften Ruf bezüglich ihrer Geschwindigkeit inne haben [99].

Ruby [98] hat ähnliche Vorteile wie Python, wurde jedoch nicht gewählt, weil es im Gegensatz zur Verwendung von Python den Ruby-Interpreter als zusätzliche Komponente nötig gemacht hätte. Dies war mit Python nicht der Fall, weil dessen Interpreter bereits wegen seiner Verwendung durch andere Komponenten des TeXML-Publishings-Servers notwendig war. Sonstige Vorteile, die die Verwendung von Ruby nahe gelegt hätten, waren nicht erkennbar.

5.4.2 WxPython

Die Wahl der Programmiersprache Python bedingte die Wahl eines Frameworks zur Erstellung der benötigten graphischen Oberfläche des Template-Designers. Dieses Framework stellt Funktionen bereit, die es erlauben eine graphische Oberfläche mitsamt der benötigten Komponenten, auch Widgets genannt, für ein Programm zu erstellen. Python, genauer die Python-Standardbibliothek, stellt selbst bereits Funktionen zur Erzeugung einer graphischer Oberflächen bereit. Jedoch haben diese den Nachteil, nicht alle benötigten Widgets bereit zu stellen und sich zudem schlecht an das Aussehen der jeweils verwendeten Plattform anzupassen, besonders unter Linux und BSD-Derivaten. Deshalb viel die Wahl auf WxPython [68][28]. Dieses stellt eine Schnittstelle zum Framework WxWidgets [96] bereit, womit bereit ein erster Vorzug von Python, nämlich die Modularisier- und Erweiterbarkeit von Python, zum Tragen kam.

WxWidgets ist selbst in C++ geschrieben und abstrahiert wiederum die auf der jeweiligen Plattform verfügbaren Komponenten zur Darstellung einer graphischen Oberfläche. Dadurch passen sich die in WxPython verfügbaren Komponenten hervorragend an das jeweils genutzte Betriebssystem an.

Ebenfalls in die Überlegungen zum zu verwendenden Gui-Framework einbezogen waren die Frameworks QT4 und GTK. QT4 ist ein Framework, dass unter einer dualen Lizenz steht. Von Vorteil wäre bei QT4 ebenfalls die Plattformunabhängigkeit sowie die gute Integration in die jeweilige Plattform gewesen. Außerdem bestanden aus früheren Projekten bereits Erfahrungen mit diesem Framework. Letztlich wurde QT4 deshalb nicht berücksichtigt, weil die kostenfreie Benutzung von QT4 die Lizenzierung des Template-Designers unter der Lizenz Gnu GPL erforderlich gemacht hätte. Weil der Template-Designer, wie bereits der TeXML-Publishing-Server, jedoch unter der liberaleren MIT-Lizenz stehen sollte, schied dies Option aus. GTK wurd als Option bald verworfen, als klar wurde, dass die Untersützung von Mac OS X hinsichtlich Stabilität nicht den benötigten Ansprüchen gerecht wurde.

5.4.3 Ghostscript

Ghostscript ist ein weit verbreiteter Interpreter für PostScript (PS) und das Portable Document Format (PDF). Für die Darstellung dieser Seitenbeschreibungssprachen existieren mittels WxPython keine Funktionen. Dies macht den Einsatz von Ghostscript notwendig. Ghostscript wird deshalb für die Konvertierung von PostScript, Encapsulated PostScript und PDF-Dateien in ein Format genutzt, dass mittels WxPython dargestellt werden kann. Ghostscript selbst wird unter einer dualen Lizenz vertrieben. Das bedeutet im konkreten Fall, dass GhostScript sowohl unter den Bedingungen der Gnu General Public License (GPL)[39] als auch unter der kommerziellen Artifex Commercial License [3] vertrieben. Der Unterschied besteht darin, dass Ghostscript mit dem Erwerb der Artifex Commercial License auch in Software mit geschlossenen Quelltexten vertrieben werden darf, ohne diese offenlegen zu müssen. Unter den Bedingungen der Gnu GPL ist dies nicht möglich. Dies stellt eigentlich für den Gebrauch durch den Template-Designer kein Problem dar, weil dieser selbst unter einer OpenSource-Lizenz steht. Jedoch ist die für den Template-Designer gewählte MIT Lizenz liberaler als die Gnu GPL, weil die MIT-Lizenz es erlaubt, Veränderungen am Quelltext nicht offenlegen zu müssen. Die Gnu GPL erlaubt dies nicht. Dadurch ist es lediglich möglich unter MIT-Lizenz stehende Quelltexte innerhalb von GPL-Projekten zu nutzen, nicht jedoch umgekehrt.

Ghostscript wird vom Template-Designer lediglich über dessen Benutzerschnittstelle angesprochen. Es werden keine internen Funktionen von Ghostscript genutzt und die Quellen von Ghostscript wurden nicht verändert, um den Template-Designer nicht selbst unter die Lizenz Gnu GPL stellen zu müssen. Dies wäre andernfalls wegen des sogenannten CopyLeft-Prinzips der Fall gewesen, wonach ein von einem Gnu GPL Programm, also Ghostscript, abgeleitetes Programm, also der Template-Designer, nur dann verbreitet werden dürfte, wenn dieses ebenfalls unter der Gnu GPL stünde. Hinzukommt, dass die duale Lizenzierung vorsieht, dass auch eine Verteilung von Ghostscript nur unter den Bedingungen der Gnu GPL gestattet ist. Deshalb wird Ghostscript nicht zusammen mit dem Template-Designer ausgeliefert. Dieses muss im Einzelfall nachinstalliert werden.

Ghostscript wurde trotz dieser Bedingungen gewählt, weil nach derzeitigem Kenntnisstand keine derart unkompliziert zu beziehende, plattformunabhängige und gleichzeitig leistungsfähige Alternative verfügbar ist.

5.4.4 Python Image Library

Für die Vorschau-Funktionen des Template-Designers bezüglich einiger Pixel-Graphikformate greift dieser auch auf die Funktionen der Python Image Library [62] zurück. Die Python

Image Library erweitert Python direkt um Funktionen zum Öffnen und Speichern verschiedener Graphikformate.

Alternativ bestand die Option, auf Funktionen des ImageMagick-Projektes und dessen Anbindung an Python (PythonMagick) zurück zu greifen. Dies hätte den Vorteil gehabt auf eine ungleich höhere Zahl von Graphikformaten Zugriff zu haben. Jedoch steht ImageMagick, wie Ghostscript, unter der Lizenz Gnu GPL. Eine weitere, das CopyLeft nutzende Komponente sollte jedoch vermieden werden, um die rechtliche Situation bezüglich der Nutzung von Funktionen nicht weiter zu verkomplizieren. Die Python Image Library steht dagegen unter einer modifizierten X11-Lizenz, die mit der vom Template-Designer genutzten MIT-Lizenz sehr starke Ähnlichkeiten aufweist.

5.5 Integrierte Drittanbieter-Software

5.5.1 Oxygen

Der Template-Designer nutzt für die bessere Veranschaulichung von Funktionen der graphischen Oberfläche diverse Symbole. Für diese Symbole wurde das Oxygen Symbolpaket benutzt [76]. Dieses wurde gewählt, weil es mit den Zielen entwickelt wurde, modern, konsistent, klar und gut aussehend zu sein. Hinzu kam eine gewisse Ähnlichkeit mancher Symbole zu in Apples Mac OS X bereits verwendeten Symbolen, was deren Erkennungsrate steigern dürfte.

Oxygen wird ebenfalls von der Desktop-Umgebung KDE4 verwendet und hat somit bereits einen gewissen Verbreitungs- und Wiedererkennungsgrad erreicht. Zudem ermöglicht die Lizenzierung des Oxygen Symbolpaketes dessen freie Verwendung im Rahmen des Template-Designers. Es gab keine Unternehmungen, ein anderes Symbolpaket zu suchen oder zu nutzen, weil Oxygen den Ansprüchen des Template-Designers in nahezu idealer Weise entsprach. Die Benutzung der von Mac OS X genutzten Symbole schied wegen deren strenger Lizenzierung aus. Diese hätte den Gebrauch auf anderen Plattformen verhindert.

5.5.2 Keychain.py

Für den einfachen Zugriffsschutz des Template-Designers wurde auf das Python-Modul keychain.py zurück gegriffen [19]. Dieses in Python geschriebene Modul ermöglicht den Zugriff auf die Schlüsselbund-Funktionen der Plattform Mac OS X. Damit ist es möglich mehrere Passwörter in einem sogenannten Schlüsselbund abzulegen, der wiederum mit einem zentralen Passwort geschützt ist. Dadurch ist es auf einfache Art möglich Passwörter an zentraler Stelle zu verwalten. Insgesamt nachteilig an dieser Lösung ist,

dass diese Schlüsselbund-Funktionen nur unter Mac OS X zur Verfügung stehen. Jedoch wurde das Modul der Art in den Template-Designer integriert, dass auf dessen Funktionen abstrahiert zugegriffen wird. Dadurch ist es möglich für die spezifisch genutzte Plattform eine eigene Lösung zur Verwaltung von Passwörtern zu finden, ohne Kernfunktionen des Template-Designers verändern zu müssen.

5.6 Verwendete Entwicklungswerkzeuge

5.6.1 Eclipse

Für die Entwicklung des Template-Designers mussten geeignete Werkzeuge gefunden werden, die halfen dessen Entwicklung zur erleichtern und zu beschleunigen. Weil der Template-Designer primär auf der Plattform Mac OS X zur Verfügung stehen sollte, wurde diese Plattform auch als Plattform für die Entwicklung des Template-Designers gewählt. Grund war, dass dies das Testen der Funktionen des Template-Designers beschleunigte. Die Entscheidung für Mac OS X als Entwicklungsplattform beschränkte jedoch die Auswahl der geeigneten Entwicklungswerkzeuge. Diese sollten, wie der Template-Designer selbst, unter einer OpenSource-Lizenz stehen und zumindest Quelltexteditor und Funktionen zur Quelltextformatierung innerhalb einer integrierten Oberfläche bieten. Letztlich kam das sogenannte Eclipse-Framework [34] zum Einsatz, weil dies in vetretbarem Zeitraum die einzige Lösung für Mac OS X war, die den geforderten Ansprüchen gerecht wurde. Eclipse selbst stellt dabei Grundfunktionalitäten bereit, die durch Erweiterungen (Plugins) ergänzt werden. Für die effiziente Erstellung des Template-Designers waren mehrere Erweiterungen erforderlich.

5.6.2 Pydev

Pydev [77] ist eine Eclipse-Erweiterung, die Funktionen für einen höheren Komfort und bessere Effizienz beim Erstellen von Python-Quelltexten bietet. Zu diesen Funktionen gehören:

1. Tabs: Das meint ganz einfach, dass mehrere Dateien gleichzeitig innerhalb eines Fensters des genutzten Entwicklungswerkzeugs geöffnet werden. Dieses Verhalten ist analog zu dem von modernen Internet-Browsern. Die Tab-Funktionalität ist in Anwendungen für Mac OS X scheinbar nicht weit verbreitet, bietet jedoch den Vorteil, sehr schnell zwischen verschiedenen offenen Dateien hin und her zu springen.
2. Anzeige von Zeilennummern: Diese eigentlich triviale Funktionalität ist bei Fehlern in Quelltexten enorm hilfreich, weil sie eine schnelle Orientierung innerhalb der

Quelltexte ermöglicht und dabei hilft, die Fehlermeldungen des jeweils verwendeten Compilers bzw. Interpreters der verwendeten Programmiersprache auszuwerten. Dieser gibt im Regelfall neben einer Fehlermeldung auch eine Zeilennummer aus, in der der Fehler auftrat.

3. Syntax-Hervorhebungen: Dabei werden Schlüsselwörter und Strukturen in jeweils eigenen Farben dargestellt. Dadurch erhöht sich die Lesbarkeit der Quelltexte deutlich. Fehler, wie fehlende öffnende oder schließende Klammern, sind dadurch schneller erkennbar.

4. Code-Folding: Code-Folding ermöglicht es, Strukturen quasi zusammenzuklappen. Dadurch können Strukturen die gegenwärtig dem Verständis des Quelltextes im Weg stehen ausgeblendet werden, wodurch mehr Platz für die aktuell relevanten Quelltexte zur Verfügung steht.

5. Auto-Vervollständigung: Dies ermöglicht eine dramatische Beschleunigung bei der Erstellung von Quelltexten. Sobald die ersten Zeichen des Namens einer gewünschten Struktur geschrieben wurden, bietet die Auto-Vervollständigung alle sich aus den ersten Zeichen ergebenden Möglichkeiten an. Die Auswahl einer dieser Möglichkeiten fügt den vollständigen Namen der Struktur an der aktuellen Position ein. Um diese Funktionalität bereitstellen zu können, muss sowohl die Standardbibliothek der verwendeten Programmiersprache bekannt sein, als auch die aktuell geöffneten Quelltexte analysiert werden.

6. Funktionssprünge: Wird eine bestimmte selbst definierte Struktur in anderen Teilen einer Anwendung genutzt, ermöglichen es Funktionssprünge, durch einen Klick auf den Namen der jeweils verwendeten Struktur, zu deren Deklaration im Quelltext zu springen. Dies beschleunigt die Navigation innerhalb der Quelltexte deutlich, besonders bei der Fehleranalyse.

5.6.3 Subclipse

Subclipse [17] ist eine Eclipse-Erweiterung, die die Anbindung an ein Versionsverwaltungssystem gewährleistet. Versionsverwaltungssysteme ermöglichen es an zentraler Stelle verschiedene Versionsstände beliebiger Dateien, also auch die Quelltexte des Template-Designers, zu verwalten. Dies geschieht dadurch, dass Änderungen an Quelltexten zusammen mit dem Zeitpunkt der Änderung und einem Benutzernamen erkannt und gespeichert werden. Durch Subclipse war somit sichergestellt, auch auf alte und eventuell fehlerfreie Versionsstände einer Datei zurückzuspringen, wenn dies notwendig war. Ferner kann über Versionsverwaltungssysteme ein zentraler Ort für eine Datensicherung geschaffen werden. Innerhalb der Bitplant-Systemarchitektur bestand bereits die Möglichkeit auf das Versionsverwaltungssystem Subversion (SVN [18]) zuzugreifen. Subclipse ermöglichte den Zugriff darauf.

5.6.4 Web Tools Platform

Die Web Tools Platform [35] bietet eine Sammlung von Eclipse-Erweiterungen die Rahmen der Entwicklung des Template-Designers für die Erstellung und Bearbeitung von XML-Daten genutzt wurden. Hier ging es vor allem um die Erstellung der DTD und XML Schema Definitionen für TemplateXML sowie die Analyse von durch den Template-Designer erstellten XML-Daten. Die Web Tools Platform bieten in diesem Zusammenhang ähnliche Komfortfunktionen wie die Pydev Erweiterung sowie Möglichkeiten zur Visualisierung der jeweiligen XML-Daten.

5.7 Schlussfolgerung

Die Wahl der richtigen Entwicklungswerkzeuge war von großer Bedeutung bei der Erstellung des Template-Designers. Ohne die untersützenden Funktionen wäre es nicht möglich gewesen, den Template-Designer im geforderten Zeitrahmen umzusetzen. Gleichzeitig war es erstaunlich wie gering die Auswahl geeigneter Entwicklungswerkzeuge für die Plattform Mac OS X ist. Positiv ist besonders das Versionsverwaltungssystem aufgefallen. Ähnlich umfangreiche Funktionalitäten auch bei der Erstellung von Technischer Dokumentation und nicht nur für Software-Quelltexte erscheinen mir sehr sinnvoll.

Der Template-Designer deckt momentan nahezu alle von TemplateXML angebotenen Funktionen ab. Kürzere Praxistests haben die Nutzbarkeit des Template-Designers unter Beweis gestellt. Das Programm erfüllt die Aufgaben, die an es gestellt wurden. Zur Fortentwicklung des Programms dürften auf mittlere Sicht deshalb primär Fehlerkorrekturen gehören. Zukünfige Funktionen könnten sich mit einer leistungsfähigen Vorschaufunktion, ausgefeilten Import- und Exportfunktionen sowie der Unterstützung von Elementen aus anderen XML-Namensräumen, beispielsweise für das TemplateXML-Element *description* oder *content*, beschäftigen.

6 Besondere Aspekte bei der Umsetzung des Template-Designers

6.1 Einführung

Im Folgenden werden einige Aspekte bei der Umsetzung des Template-Designers behandelt, die im Rahmen der Technischen Dokumentation und in meinen Augen von besonderer Relevanz zu sein scheinen und deshalb im Detail ausgeführt werden. Dabei wird im Einzelnen auf die Probleme bei der Validierung des mit dem Template-Designer erzeugten TemplateXML eingegangen. Anschließend wird der Aspekt der Quelltext-Dokumentation des Template-Designers besprochen.

6.2 Bitplant-TemplateXML validieren

6.2.1 Einführung

Schon während der Planung von TemplateXML stellte sich die Frage, in welcher Form eine Validierung des erarbeiteten XML-Dialektes stattfinden sollte. Diese Validierung ist deshalb notwendig, weil nur danach sicher gestellt ist, dass alle mit den XML-Daten arbeitenden Komponenten (Template-Designer und Konverter nach LaTeX sowie XSL-FO) auch gewiss sein können, auch solche Daten zu erhalten, mit denen diese umgehen können. Damit soll eventueller Fehlgebrauch genauso vermieden werden, wie falsche oder nicht nachvollziehbare Interpretationen der zugeführten XML-Daten oder gar Programmabstürze auf Grund von unvorhergesehenen oder fehlenden Daten.

Die Validierung scheint zum aktuellen Zeitpunkt sekundär, weil der Template-Designer momentan die einzige Anwendung ist, die TemplateXML manipulieren kann. Doch schon sobald ein Benutzer eigenständig Hand an die XML-Daten legt oder eine weitere Anwendung hinzugezogen wird, die ebenfalls TemplateXML manipulieren kann, wird eine Validierung unumgänglich. Deshalb stellte sich das Problem der Validierung bereits beim Entwurf von TemplateXML und der Implementierung des Template-Designers.

6.2.2 Document Type Definition

Zuerst kam eine klassische Document Type Definition (DTD) [87] in Betracht. Nach deren rascher Umsetzung stellte sich jedoch heraus, dass diese nur unzureichend auf die Erfordernisse von TemplateXML einzugehen vermochte. Das erste Hauptproblem bei der Validierung des TemplateXML stellte die fehlende Unterstützung von Namensräumen bei DTDs dar. Die Fehlende Unterstützung von Namensräumen ist dem Umstand geschuldet, das die Spezifikation von DTDs älter ist, als jene von XML 1.0. Jedoch wurde die Syntax von DTDs der XML 1.0 Spezifikation hinzugefügt, ohne diese an die Erfordernisse von XML anzupassen. Die Unterstützung von Namensräumen ist zum gegenwärtigen Zeitpunkt zwar sekundärer Natur, weil diese derzeit nicht gebraucht werden. Jedoch ist es besonders in Kombination mit dem Elementen *description* und *content* zukünftig interessant auf diese zu zugreifen. Für das Element *description* könnten auf diesem Weg z.B. Auszeichnungselemente aus dem XHTML-Sprachraum eingebunden werden. Für das Element *content* ist besonders die Einbindung von Daten aus dem SVG- oder AI-Sprachraum interessant, wenn das zugehörige Attribut *type* den Wert *image* trägt. Dadurch könnten zukünftig Bilddokumente direkt in das TemplateXML eingebunden werden, ohne auf externe Dateien verweisen zu müssen. Auch ist eine Validierung eher umständlich, wenn bereits für den standardmäßigen Namensraum ein Präfix vergeben wird. An diesem Punkt lässt sich das zu einer DTD gehörige XML nur noch validieren, wenn dies in der Processing-Instruktion zur zugehörigen DTD vermerkt ist.

Als größerer und ausschlaggebender Nachteil der DTD stellte sich jedoch der Umstand heraus, dass DTDs nicht zwischen verschiedenen Datentypen unterscheiden können. So lässt sich nicht prüfen, ob z.B. der Wert eines Attributes ein Zahlenwert oder eine Zeichenkette ist. Somit ist für eine auswertende Anwendung nicht sichergestellt, ob diese einen Zahlenwert bekommt, wenn sie diesen erwartet. Dieser Wert lässt sich zwar anwendungsseitig untersuchen, bevor er verwendet wird, dies ist jedoch wenig elegant. Wo doch ohnehin vor der Benutzung des TemplateXML untersucht werden muss, ob dieses zumindest wohlgeformt ist, ist der Zeitpunkt des Einlesens der XML-Daten auch die bessere Wahl zur Untersuchung auf bestimmte Werte und Datentypen.

6.2.3 XML Schema 1.0

Nachdem sich die Nutzung einer DTD als ungünstig herausgestellt hatte, wurde begonnen, TemplateXML in Form eines XML Schemas zu gießen. XML Schema unterstützt sowohl Namensräume als auch verschiedenste benötigte Datentypen. [45, WG Description][46, 4. Design Principles][47, Scope]

Für die Validierung konnte, wie bei DTDs, auch das frei verfügbare Open-Source Programm *xmllint* zugegriffen werden. Jedoch ergab sich auch bei der Verwendung von

XML Schema ein Problem. Dieses bestand darin, dass es XML Schema genauso wie DTDs nicht gestattet, sowohl sogenannte Co-Constraints als auch Conditional Types zu nutzen. Co-Constraints sind Wertebereiche, die nur dann gültig sind, wenn ein Element oder typischer Weise Attribut einen bestimmten Wert besitzt. Conditional Types sind Strukturen, die nur dann gültig sind, wenn ein Element oder Attribut einen bestimmten Wert besitzt. Dies ist jedoch hinsichtlich einer vollständigen Validierung und optimalen Anpassung des TemplateXML an die gegebenen Erfordernisse sehr wünschenswert. Dabei geht es konkret um die Nutzung der Attribute *type* und *value* der Elemente *dimension*, *position* und *paper* und um das Element *content*, wenn dessen Attribut *type* mit dem Wert *image* belegt ist. Es ist mit XML Schema derzeit nicht möglich, zu daraufhin zu prüfen, ob ein Attributwert nur dann gestattet ist, wenn ein anderes Attribut einen bestimmten Wert annimmt und es ist nicht möglich Datentypen für ein Element in Abhängigkeit eines bestehenden Attributwertes festzulegen.

```
1 <paper type="orientation" value="a4"/>
2 <paper type="layout"      value="a4"/>
3 <paper type="format"      value="a4"/>
```

Listing 6.1: Veranschaulichung des Problems der Co-Constraints

Das Beispiel veranschaulicht das Problem der Co-Contraints. Gemäß XML Schema ist die obige Auszeichnung korrekt, aber nutzlos. Das Element *paper* kommt dreimal vor, es sind nur die erlaubten Attribute *type* und *value* gesetzt und deren Werte sind ebenfalls gültig. Jedoch ergibt die Kombination der Werte von *type* und *value* keinen Sinn. Für die Orientierung (type="orientation") sollten nur die Werte *portrait* und *landscape* für hoch- bzw. querformatiges Papier zulässig sein. für das Layout (type="layout") sollten hingegen nur die Werte *oneside* und *twoside* für einseitigen bzw. doppelseitigen Druck Gültigkeit besitzen. Nur für den letzten hier genannten Fall, das Format, ist der Wert *a4* sinnvoll.

Alternativ wäre die Einführung zusätzlicher Elemente ein Ausweg gewesen, jedoch war es erklärtes Ziel beim Entwurf von TemplateXML sich auf nur wenige Elemente und Attribute zu begrenzen, den XML-Dialekt nicht unnötig aufzublähen.

```
1 <paper-orientation value="portrait"/>
2 <paper-layout      value="oneside"/>
3 <paper-format      value="a4"/>
```

Listing 6.2: Alternative zum Problem der Co-Constraints

Das zweite Problem der Conditional Types lässt sich an der Definition des Elementes *content* beschreiben. Listing 6.3 zeigt die gegenwärtige Situation. *Content* ist als unstrukturiertes Element definiert. Das hat jedoch den Nachteil, dass die Anforderungen an den Inhalt nicht zufriedenstellend erfüllt werden können, sollte der Inhalt vom Typ Bild sein (type="image"), denn dann wären strukturierte Inhalte wünschenswert. So begnügt sich die Definition gegenwärtig mit unstrukturierten Inhalten und führt statt

dessen eine interne von XML unabhängige Struktur ein. Dies ließe sich gegenwärtig nur umgehen, in dem *content* so definiert würde, dass es Mixed Content aufnehmen könnte. Jedoch wäre es dann auch möglich, XML Strukturen zum Beispiel für statische Inhalte zu benutzen. Dies ist genauso wenig wünschenswert.

```
1 <content type="text">Ich bin statischer Inhalt</content>
2
3 ...
4
5 <content type="image">serverImage:dateiname.png:originalername.png</
    content>
```

Listing 6.3: Veranschaulichung des Problems der Conditional Types

```
1 <content type="text">Ich bin statischer Inhalt</content>
2
3 ...
4
5 <content type="image">
6   <path>serverImage</path>
7   <name>dateiname.png</name>
8   <oriname>originalerdateiname.png</oriname>
9 </content>
```

Listing 6.4: Ideale Lösung des Problems der Conditional Types

6.2.4 XSLT

Zur Lösung der Probleme gab es verschiedene Ansätze. Die erste Möglichkeit bestand darin, eine Prüfung der sinnvollen und somit erlaubten Werte mittels XSLT durchzuführen. Einige Beispiele demonstrieren das Vorgehen für einen kleinen Teil der bei diesem Vorgehen notwendigen Tests. In Listing 6.5 wird überprüft, ob innerhalb des Elementes *paper* das Attribut *value* mit den sinnvollen Werten *portrait* und *landscape* belegt ist, wenn das zugehörige Attribute *type* den Wert *orientation* trägt. In Listing 6.6 wird überprüft ob für jedes unterhalb eines *parameter* vorkommende Element *paper* der Wert des jeweils zugehörigen Attributes *type* auch einmalig vergeben ist. Beide Beispiele geben eine Fehlermeldung aus, sollte der Fehlerfall eintreffen. Andernfalls erfolgt keinerlei Ausgabe. Dieses Vorgehen erscheint elegant, findet die Validierung doch nur anhand von Notationen statt, die selbst in XML geschrieben. Jedoch ergeben sich bei diesem Vorgehen praktische Nachteile. Zum einen ergaben im konkreten Fall die möglichen zu untersuchenden Fälle eine hohe Zahl von benötigten XSLT-Stylesheets um diese ab zu decken. Dies Fälle abzudecken und zu verwalten wäre eine im Verhältnis zu aufwendige Angelegenheit geworden. Weiter war dieses Vorgehen bezüglich der Ausführungsgeschwindigkeit im Vergleich zu anderen Methoden verhältnismäßig langsam.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:bit="http://www.bitplant.de/template"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  extension-element-prefixes="bit">
  <xsl:output method="text"    version="1.0"
              encoding="UTF-8" indent="no"/>

  <xsl:template match="/">
    <xsl:call-template name="testpaper"/>
  </xsl:template>

  <xsl:template name="testpaper">
    <xsl:for-each select="//bit:paper[@type='orientation']">
      <xsl:if test="@value!='portrait' and @value!='landscape'">
        <xsl:text>Warning: Unrecommended value &amp;quot;</xsl:text>
        <xsl:value-of select="@value"/>
        <xsl:text>&amp;quot; of attribute value.</xsl:text>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

Listing 6.5: Prüfung auf Erfüllung eines Co-Constraints mit XSLT, Beispiel 1

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:bit="http://www.bitplant.de/template"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  extension-element-prefixes="bit">
  <xsl:output method="text"    version="1.0"
              encoding="UTF-8" indent="no"/>

  <xsl:template match="/">
    <xsl:call-template name="testparameter"/>
  </xsl:template>

  <xsl:template name="testparameter">
    <xsl:for-each select="//bit:parameter">
      <xsl:call-template name="testpaper">
        <xsl:with-param name="typevalue">orientation</xsl:with-param>
      </xsl:call-template>
      <xsl:call-template name="testpaper">
        <xsl:with-param name="typevalue">layout</xsl:with-param>
      </xsl:call-template>
      <xsl:call-template name="testpaper">
        <xsl:with-param name="typevalue">format</xsl:with-param>
      </xsl:call-template>
    </xsl:for-each>
  </xsl:template>

  <xsl:template name="testpaper">
```

```
28      <xsl:param name="typevalue">orientation</xsl:param>
29      <xsl:variable name="count" select="count(bit:paper[@type=$typevalue])
            "/>
30      <xsl:if test="$count&gt;1">
31          <xsl:text>Warning: value &quot;</xsl:text>
32          <xsl:value-of select="bit:paper/@type"/>
33          <xsl:text>&quot; of attribute type appears </xsl:text>
34          <xsl:value-of select="$count"/>
35          <xsl:text> times.</xsl:text>
36        <xsl:value-of select="$count"/>
37      </xsl:if>
38    </xsl:template>
39
40  </xsl:stylesheet>
```

Listing 6.6: Prüfung auf Erfüllung eines Co-Constraints mit XSLT, Beispiel 2

6.2.5 Schematron

Eine weitere Möglichkeit besteht darin ein XML Schema mit der Schema-Sprache Schematron zu erweitern. ISO Schematron ist ein XML-Dialekt, der besonders dazu geeignet ist, XML-Dokumente zu validieren. Dies geschieht primär über XPath-Ausdrücke. ISO Schematron unterscheidet sich von Document Type Definitions und XML Schema vor allem dahingehend, dass es nicht dazu entworfen wurde, eine XML-Struktur zu definieren, sondern bestimmte XML-Strukturen zu validieren. Mit ISO Schematron lassen sich Strukturen überprüfen und gegebenenfalls danach handeln. ISO Schematron ist spezifiziert als offizieller ISO/IEC-Standard 19757-3:2006. Durch die Beschränkung von Schematron auf die Validierung eignet es sich besonders gut, um in definierende Schema-Sprachen wie DTD, XML Schema oder RelaxNG eingebunden zu werden, so dies notwendig ist und vom Validator unterstützt wird.

Bei diesem Vorgehen wird Schematron direkt einem bestehenden XML Schema hinzugefügt, indem die Schematron-Regeln innerhalb des XML Schema Elementes *annotation* eingefügt und mit einem eigenen Namensraum versehen werden. Ein Validator, der die Schematron-Regeln interpretieren kann, gibt im Listing 6.7 die innerhalb des *assert* Elementes notierte Fehlermeldung aus, sollte die als XPath notierte Bedingung des zugehörigen Attributes *test* erfüllt werden. Im Listing ist das wiederum ein Test darauf, ob das Attribut *value* mit einem der geforderten Werte *portrait* oder *landscape* bestückt ist, wenn das Attribut *type* den Wert *orientation* trägt. Diese Lösung kam nicht zum Einsatz, weil im selbst gesteckten Zeitrahmen keine OpenSource-Implementierung eines Validators gefunden werden konnte, die Schematron in Kombination mit XML Schema beherrschte.

```
<xs:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.bitplant.de/template"
            xmlns="http://www.bitplant.de/template"
            elementFormDefault="unqualified"
            attributeFormDefault="unqualified">

  <xs:annotation>
    <xs:appinfo>
     <sch:title>TemplateXML Validation</sch:title>
     <sch:ns prefix="bit" uri="http://www.bitplant.de/template"/>
    </xs:appinfo>
   </xs:annotation>

  ...

  <xs:element name="parameter">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="description" type="descriptionType"
                    minOccurs="0" maxOccurs="1"/>
        <xs:element name="dimension" type="dimensionType"
                    minOccurs="0" maxOccurs="2"/>
        <xs:element name="position" type="positionType"
                    minOccurs="0" maxOccurs="4"/>
        <xs:element name="paper" type="paperType"
                    minOccurs="0" maxOccurs="3">
        <xs:annotation>
          <xs:appinfo>
            <sch:pattern name="typeTests">
              <sch:rule context="//bit:paper">
                <sch:assert test="@type='orientation' and
                        (@value!='portrait' and @value!='landscape')">
                Value of attribute value must be &quot;portrait&quot;
                or &quot;landscape&quot; if value of attribute
                &quot;type&quot; is &quot;orientation&quot;.
                </sch:assert>
              </sch:rule>
            </sch:pattern>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  ...

</xs:schema>
```

Listing 6.7: Prüfung auf Erfüllung eines Co-Constraints mit Schematron

6.2.6 XML Schema 1.1

Als dritte und momentan theoretische Überlegung kam XML Schema 1.1 in Betracht. Da die momentan aktuelle Recommendation(Empfehlung) XML Schema 1.0 [85] keine Unterstützung für Co-Constraints bietet, ist diese Eigenschaft im aktuell verfügbaren Working-Draft [94] vom 20. Juni 2008 des World Wide Web Consortium (W3C) vorhanden. Gemäß dem aktuellen Working-Draft sähe ein Auszug des XML Schemas nach Version 1.1 aus, wie in Listing 6.8 demonstriert. Zuerst war die direkte Definierung des TemplateXML in XML Schema 1.1 eine ernsthaft Alternative, weil nach allem Dafürhalten die XML Schema Version 1.1 als Recommendation Ende des Jahres 2008 bzw. Anfang des Jahres 2009 erscheinen sollte. Jedoch musste auch bedacht werden, dass möglichst zeitnah eine nutzbare Implementierung eines Prozessors für XML Schema 1.1 verfügbar sein sollte. Weil dies nicht abzusehen war, wurde auch diese Variante letztlich nicht berücksichtigt. Jedoch ist diese die mittefristig zukunftsträchstigste Lösung für das bestehende Problem und wird wohl zukünftig auch Verwendung finden.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3      targetNamespace="http://www.bitplant.de/template"
4      xmlns="http://www.bitplant.de/template"
5      elementFormDefault="unqualified"
6      attributeFormDefault="unqualified">
7
8  ...
9
10   <xs:element name="parameter" minOccurs="1" maxOccurs="1">
11     <xs:complexType>
12       <xs:sequence>
13         <xs:element name="description" type="descriptionType"
14                     minOccurs="0" maxOccurs="1"/>
15         <xs:element name="dimension" type="dimensionType"
16                     minOccurs="0" maxOccurs="2"/>
17         <xs:element name="position" type="positionType"
18                     minOccurs="0" maxOccurs="4"/>
19         <xs:element name="paper" minOccurs="0" maxOccurs="3">
20           <xs:complexType>
21             <xs:attribute name="type" type="paperAttrType" use="required
                   "/>
22             <xs:attribute name="value" type="valuePaperType" use="
                   required"/>
23           </xs:complexType>
24           <xs:assert test="@type='orientation' and
25                     (@value!='portrait' and @value!='landscape')"/>
26         </xs:element>
27       </xs:sequence>
28     </xs:complexType>
29   </xs:element>
30
31 ...
32
```

```
33   <xs:element name="content" minOccurs="1" maxOccurs="1">
34     <xs:complexType>
35       <xs:simpleContent>
36         <xs:extension base="xs:string">
37           <xs:attribute name="type" type="contentAttrType" use="required
                 "/>
38           <xs:attribute name="angle" type="angleAttrType" default="0"/>
39         </xs:extension>
40       </xs:simpleContent>
41     </xs:complexType>
42     <xs:alternative test="@type='image'">
43       <xs:complexType>
44         <xs:sequence>
45           <xs:element name="path" type="pathType"
46                       minOccurs="1" maxOccurs="1"/>
47           <xs:element name="name" type="nameType"
48                       minOccurs="1" maxOccurs="1"/>
49           <xs:element name="oriname" type="orinameType"
50                       minOccurs="1" maxOccurs="1"/>
51         </xs:sequence>
52       </xs:complexType>
53     </xs:alternative>
54   </xs:element>
55
56 ...
57
58 </xs:schema>
```

Listing 6.8: Prüfung auf Erfüllung von Co-Constraints und Conditional Types mit XML Schema 1.1

Die erste Element-Definition *parameter* zeigt die Berücksichtigung eines Co-Constraints unter Zuhilfenahme des Elementes *assert*. Der Name scheint dabei direkt aus der Schema-Sprache Schematron entliehen zu sein. Hier wie dort wird über einen XPath-Ausdrück eine Bedingung angegeben. Deren Ergebnis bestimmt, ob die aktuell validierte XML-Struktur valide ist. Das zweite gezeigte Element-Definition zeigt die gegenwärtige Lösung für das Problem der Conditional Types. Hier wird über das Element *alternative* eine Struktur vorgegeben, die gültig ist, sobald der XPath-Ausdruck des Attributes *test* wahr ist. In diesem Fall werden strukturierte Inhalte für das Element *content* angefordert, sobald dessen Attribut *type* mit dem Wert *image* belegt ist.

6.2.7 Python mit ElementTree

Die sich letztlich durchsetzende Lösung zur Validierung des TemplateXML wurde mittels Python und der XML-Erweiterung ElementTree [60] umgesetzt. Dies ist die für die gegebene Aufgabenstellung bis zur Nutzbarmachung von XML Schema 1.1 praktikabelste Möglichkeit, die gesteckten Ziele zu erreichen. Bei diesem Vorgang werden die

XML-Daten zuerst auf Wohlgeformtheit hin überprüft. Wenn diese Prüfung bestanden wurde, werden die XML-Daten validiert. Dazu wird beim XML-Wurzelelement beginnend jegliche sich darunter befindliche Struktur (Elemente, Attribute, Werte) gesucht, identifiziert und die notwendigen Tests durchgeführt. Dieser Vorgang wiederholt sich für jede Struktur, die weitere Strukturen beinhaltet.

Die beispielhaft gezeigte Methode in Listing 6.9 demonstriert die Überprüfung zweier Elemente *paper*. Der erste Vergleich mit der Instanz *paperFalsch* schlägt fehl, weil der Wert *orientation* des Attributes *type* für das Attribut *value* die Werte *portrait* und *landscape* erfordert. Stattdessen wird jedoch der Wert *oneside* übergeben. Dieser Umstand löst in der Realität keinen Fehler aus, sondern lediglich eine Warnung, weil diese Zuweisung gemäß den Festlegungen in der XML Schema Definition nicht falsch ist. Trotzdem wird der Wert des Attributes *value* in einem späteren Schritt ignoriert und der Vorgabewert *portrait* angenommen.

```
""" Die Zuweisung sieht in der Realitaet anders aus.
    Sie dient hier lediglich der Veranschaulichung. """
paperFalsch  = '<paper type="orientation" value="oneside"/>'
paperRichtig = '<paper type="layout"      value="oneside"/>'

def checkDependendEnum(self, elem, attrib1_name, attrib1_value, attrib2_
    name, attrib2_enum):
  """ Handle two attribute siblings.

  This function shall check for something similar to a co-constraint.
  It get's an attribute with a value and a second attributewith an
      enumeration.
  Then it checks, if value of attribute one is x.
  After that, if value of attribute y is in the given enumeration.

  -. elem          is a ElementTree element instance
  -. attr1_name    is a string defining the name of the first attribute
  -. attrib1_value is a string defining the value of the first attribute
  -. attr2_name    is a string defining the name of the second attribute
  -. attr2_value   is a list or tuple defining all possible values
                   of attribute two, depend on attribute value one

  """
  if elem.get(attrib1_name) and elem.get(attrib2_name):
    if elem.get(attrib1_name) == attrib1_value \
    and not elem.get(attrib2_name) in attrib2_enum:
      self.mb.setMessage("warning", _(u"Originally long but stripped
          warning text.")
      return False
    else:
      return True
  else:
      return True

checkDependendEnum(paperFalsch, "type", "orientation", "value", ["
    portrait", "landscape"])
```

```
33 # return False
34
35 checkDependendEnum(paperRichtig, "type", "orientation", "value", ["
       portrait", "landscape"])
36 # return True
```

Listing 6.9: Prüfung auf Erfüllung von Co-Constraints mit Python

6.2.8 Schlussfolgerung

Eine geeignete Möglichkeit zu finden, TemplateXML so vorzubereiten, dass es ohne größere Anstrengungen validiert und genutzt werden kann, war verhältnismäßig aufwendig. Ursache dessen war einerseits der Wunsch, TemplateXML möglichst kompakt zu halten. Das führte dazu, dass einige Situationen bei der Validierung schwieriger zu bewältigen sind, als das eigentlich erforderlich ist. Andererseits sind die Schwächen der Document Type Definition und von XML Schema 1.0 sehr deutlich geworden. Die Definierung von XML, welches primär Einstellungen und weniger semantisch strukturierte Inhalte enthält – wie das bei TemplateXML der Fall ist – ist mit der Document Type Definition sehr schwierig bis unmöglich im Detail festzulegen. Besonders die fehlende Unterstützung von Datentypen fällt hier unangenehm auf. XML Schema hat dahingehend die Situation wesentlich verbessert, doch hat die fehlende Unterstützung von Co-Constraints und Conditional Types einen sehr hohen Mehraufwand erfordert. Hier bleibt zu hoffen, dass XML Schema 1.1 sehr schnell veröffentlicht wird, damit diese Defizte beseitigt werden können.

6.3 Dokumentation der Quelltexte

6.3.1 Einführung

Die Dokumentation der Quelltexte des Template-Designers ist ein wichtiger Punkt zur Sicherung der Qualität und der zukünftigen Nutz- und Erweiterbarkeit des Programms. Quelltextdokumentation existiert aus dem Grund heraus, dass nur Sie es ermöglicht, Quelltexte und Schnittstellen eines Programms in kurzer Zeit zu verstehen, zu nutzen und gegebenenfalls zu verändern. Derjenige, der eine bestimmte und bereits bestehende Funktion/Struktur verwenden möchte, soll mittels Quelltextdokumentation in die Lage versetzt, diese Funktion/Struktur zu nutzen, ohne die exakte Art und Weise des Funktionierens, bzw. den korrespondieren Quelltext, zu kennen und zu verstehen.

Dies ermöglicht es, Programme schneller zu erweitern und Fehler zu korrigieren. Die schnelle Erweiterung wird dadurch ermöglicht, dass der Entwickler nicht mehr alle Quelltexte kennen muss, die er benutzt. Es ist nur noch wichtig, wie er diese anspricht. Fehler können deshalb schneller korrigiert werden, weil sich anhand des Vergleiches zwischen der in der Dokumentation definierten Aufgabe einer Funktion/Struktur und dem, was die Funktion/Struktur wirklich tut, erkennen lässt, ob die genutzte Funktion/Struktur korrekt funktioniert.

Das Finden und die Korrektur des Fehlers beschränkt sich dann ausschließlich auf den jeweils zur Dokumentation gehörenden Bereich des Quelltextes. Ohne Quelltextdokumentation müsste der auf einer bestehenden Funktion aufbauende Entwickler alle involvierten Quelltexte untersuchen bzw. anhand des Namens der Funktion raten, was diese tut. Dies wäre ein sehr zeitaufwendiges Unterfangen.

Aus diesen Gründen der praktischen Umsetzung dieses Themas mit dem Template-Designer und der allgemeinen Relevanz für die Technische Dokumentation werden im Folgenden Details zur Erstellung von Quelltextdokumentation behandelt. Diese sind bei der Erstellung des Template-Designers berücksichtigt worden, um das Projekt Template-Designer für andere Entwickler wegen des dadurch erleichterten Einstiegs attraktiver zu machen.

Für den Zweck der Dokumentation existieren für die Programmiersprache Python mehrere verschiedene Ansätze [43]. Im Folgenden wird eine kurze Einführung in den verwendeten Ansatz geliefert. Weiter werden einige der verwendeten Methoden exemplarisch und hinsichtlich ihrer Bedeutung für die Technische Dokumentation diskutiert. Ziel ist, die Systematik der Quelltextdokumentation zu verständlichen. Auch können die folgenden Inhalte dabei helfen, die Qualität einer Anwendung zu beurteilen. Gut dokumentierte Quelltexte sind zumindest ein starkes Indiz für gewissenhaft formulierte Quelltexte und damit gute Programme. Auch und gerade in der Technischen Dokumentation ist dies interessant, weil eine gute Quelltextdokumentation vom Wissen und

Verständnis der Entwickler um die Probleme im Umgang mit Technischer Dokumentation zeugt. Bei der Bewertung von Software für die Technische Dokumentation kann dies daher durchaus aufschlussreich sein.

Grundsätzlich dient das Dokumentationssystem mit Python dem Ziel, Benutzer mit ihren verschiedenen Ansprüchen zufrieden zu stellen. Das sind:

- die jeweiligen Entwickler
- das System zur Verwaltung der Dokumentation
- die Benutzer der Dokumentation, die Endanwender

Die Ansprüche der Benutzer ergänzen sich in Teilen, können sich aber auch widersprechen. Der Entwickler des jeweiligen Quelltextes fordert hinsichtlich Syntax und Integration in die bestehenden Quelltexte ein möglichst einfaches und schlankes System, dass ihn nicht in irgendeiner Form bevormundet und flexibel genug ist, auf die jeweils gültigen Ansprüche einzugehen. Das Verwaltungssystem der Dokumentation fordert einen gewissen Grad an Strukturierung, um die Dokumentation schnell und ohne größere Aufwände in eine Form zu bringen, die es verarbeiten und nutzen kann. Aufwendige Strukturierung und große Flexibilität stellen jedoch häufig bei der Umsetzung von Lösungen Zielkonflikte dar, die gelöst werden müssen. Ein weiteres Beispiel für solche Zielkonflikte sind XML-Editoren, die die Nutzer dahingehend unterstützen, XML-Tags nicht selbst schreiben zu müssen, sondern den Anwendern per Mausklick ermöglichen, Inhalte zu strukturieren, wenn Sie dies wünschen.

Der Anwender, was in den meisten Fällen andere Entwickler sein dürften, aber auch Technische Redakteure sein können, fordert eine möglichst ausführliche und dennoch schnell zugängliche und zu überblickende Dokumentation. Dies verstärkt die obig genannte Problematik weiter, denn Ausführlichkeit und schnelle Informationsfindung sind schwierig zu vereinen.

6.3.2 System zur Verwaltung von Qelltext-Dokumentation

Ein ideales System zur Verwaltung von Quelltextdokumentation besteht nach [43] und [41] aus mehreren Komponenten, die jeweils unterschiedliche Aufgaben übernehmen. Das System lässt sich in insgesamt sechs Komponenten unterteilen und funktioniert prinzipiell wie folgend beschrieben.

Am Anfang steht eine *Reader*-Komponente, die eine weitere Komponente *Input* beauftragt, die Dokumentation eines Programms einzulesen und anschließend an den Reader weiterzuleiten. Die Input Komponente ist somit auch dafür verantwortlich die

verschiedenen Speicherformen der Dokumentation transparent zugänglich zu machen. Formen der Speicherung können einfache Dateien, mehrere Dateien oder die Abholung der Dateien von einem Webserver oder Versionskontrollsystem sein.

Der Reader gibt die Dokumentation nach deren Erhalt an eine weitere Komponente, den *Parser*, weiter. Der Parser extrahiert aus den an ihn gelieferten Inhalten die relevanten Teile und bringt die Dokumentation in eine Struktur, die von allen weiteren Komponenten des Systems gleichsam verstanden werden kann. Diese Struktur wird *Document Tree* genannt. Anschließend liefert der Parser die in den Document Tree übertragene Dokumentation an den Reader zurück.

Der Reader übergibt die Dokumentation nun an die vierte Komponente des Systems, den *Transformer*. Der Transformer ist dafür zuständig, die Dokumentation so zu verändern, dass diese den aktuellen Notwendigkeiten entspricht. Was die Notwendigkeiten sind, wird dem Transformer von außen vorgegeben. Das kann die Form von Verweisen, Fußnoten oder dynamisch generiertem Text sein, aber auch einfach die Auflistung bzw. Existenz von Informationen insgesamt. Wenn bestimmte Informationen im Ergebnis nicht gewünscht sind, werden diese hier entfernt. Wenn Informationen in einer anderen Reihenfolge erscheinen sollen, wird die Reihenfolge an dieser Stelle geändert. Auch das Hinzufügen von Informationen aus anderen Quellen geschieht hier. Nachdem der Transformer alle an Ihn übertragenen Aufgaben durchgeführt hat, wird die Dokumentation an die Komponente *Writer* übergeben.

Diese Komponente wandelt den Document Tree der Dokumentation in das gewünschte Ausgabeformat um. Welches das Ausgabeformat ist, bestimmt der Writer. Aufgrund der modularen Struktur des gesamten Systems, können verschiedene Writer zur Verfügung stehen, die unterschiedliche Ausgabeformate realisieren. Es existieren Writer-Komponenten für PlainText, XML, LaTeX und weitere Formate.

Nach der Beendigung der Umwandlung in das Ausgabeformat wird die Dokumentation schließlich an die letzte Komponente *Output* des Systems übergeben. Diese ist dafür zuständig, die Daten auszugeben. Mögliche Wege der Ausgabe sind die Darstellung der Informationen als reinen Bildschirmtext, die Speicherung in einfachen Dateien oder mehreren Dateien, der Versand per E-Mail oder die Publikation in einer Datenbank oder auf einem Web-Server.

Das DocUtils-Paket setzt das soeben beschriebene System um, weshalb es für das Projekt Template-Designer zum Einsatz kam.

6.3.3 Verknüpfung von Quelltext und Dokumentation

Der erste und fundamentalste Schritt ist es, die Dokumentation in die Quelltexte einer Anwendung zu integrieren. Diese interne Dokumentation erfolgt so, dass die Dokumen-

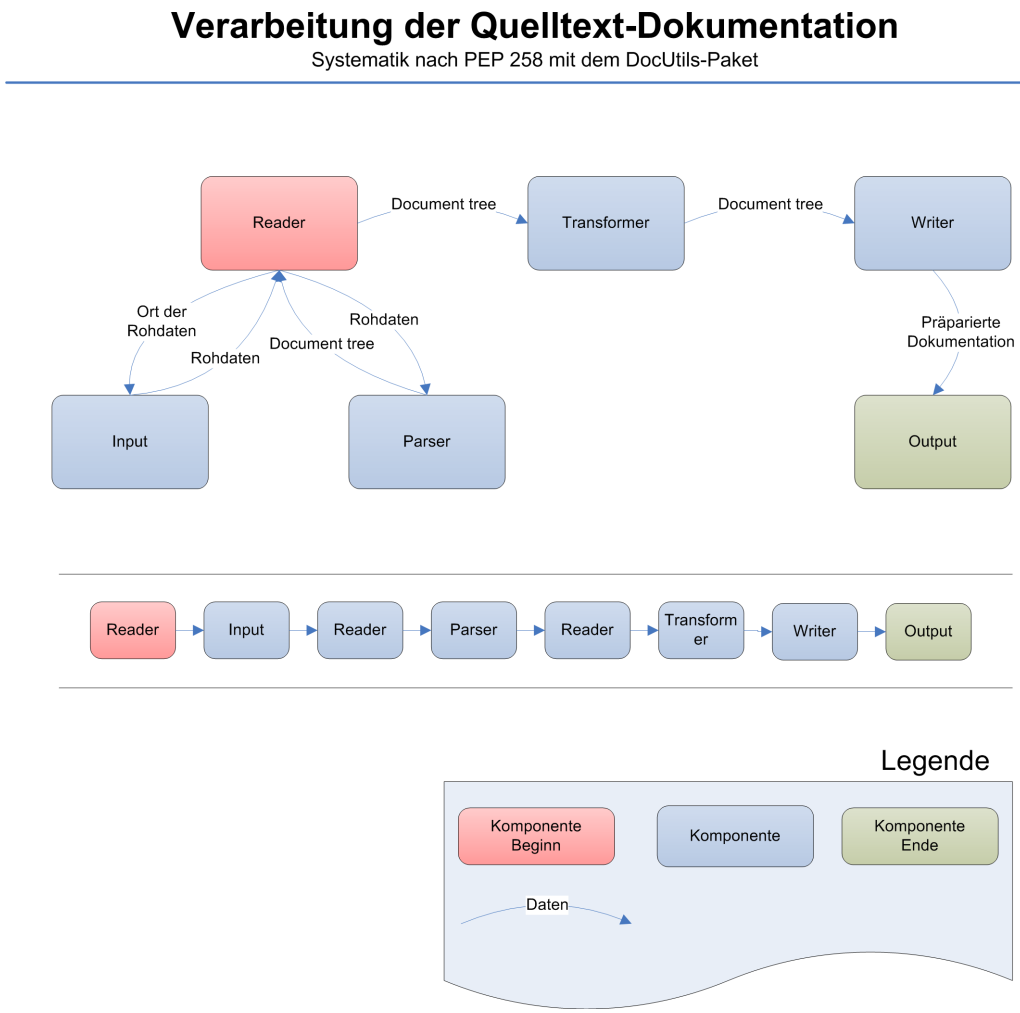

Abbildung 6.1: Verarbeitung der Quelltextdokumentation

tation genau an jener Stelle im Dokument verfasst wird, wo die Dokumentation auch aktuell ist. Das hat mehrere Vorteile. Zum einen muss der Entwickler nicht mit mehreren Werkzeugen arbeiten, Quelltext und Dokumentation befinden sich ja in der gleichen Datei, zum anderen ist dafür gesorgt, dass die Dokumentation stets auf dem aktuellsten Stand ist. Änderungen am Quelltext wirken sich direkt auf die Dokumentation aus. Die Entfernung einer überflüssig gewordenen Funktion beispielsweise entfernt die Dokumentation gleich mit. Würde die Dokumentation extern gepflegt, wäre die umständliche Pflege aller Quelltextänderungen die Folge, was extrem ineffizient wäre. Würde die Dokumentation zu einer Funktion zwar in der selben Datei wie der Quelltext erfasst werden, aber nicht dort wo die Funktion niedergeschrieben ist, wäre das ebenfalls nicht zweckdienlich, weil auch in diesem Fall zwei Strukturen, die des Quelltextes und der Dokumentation, gepflegt werden müssten.

Nachteilig an dieser Art Dokumentation ist jedoch, dass jegliche Strukturierung der Dokumentation nur auf Zeichenebene möglich ist. Deshalb wäre das Vorgehen, die Dokumentation zentral im Quelltext zu hinterlegen, doppelt nachteilig. Der Vorteil einer werkzeuggestützten Struktuierung, der ja bei externer Pflege von Dokumentation, z.B. Endanwenderdokumention mit Text- oder XML-Proezessoren, gegeben ist, würde aufgegeben, um anschließend durch die Mühen der Pflege von zwei Strukturen keinen weiteren Vorteil aus dem Vorgehen zu ziehen zu können.

Weil die Strukturierung von interner Dokumentation nur auf Zeichenebene möglich ist, müssen Regeln definiert werden, die bei der Erstellung der Quelltextdokumentation zu beachten sind. Der Entwickler muss die Dokumentation der von ihm definierten Strukturen in einer festgelegten Form in seine Quelltexte einbinden und die Dokumentation auch in ihrer Inneren Form einer Strukturierung unterziehen. Dies geschieht mit dem Ziel, den Nutzwert für den Anwender möglichst groß zu gestalten und die Auswertung durch den oben beschriebenen Parser zu erleichtern, indem man dem Parser das Finden der Dokumentation erleichtert.

6.3.4 Docstrings

Jegliche Dokumentation mit dem DocUtils-Paket geschieht über Stringliterale, auch Zeichenfolgen/-ketten genannt, und zwar direkt im Quelltext. In Python werden Stringliterale, die sich nur über eine Zeile erstrecken, mit dem Raute-Zeichen (#) eingeleitet. Alle Zeichen, die bis zum Zeilenende auf die Raute folgen, werden nicht als Quelltext interpretiert. Stattdessen wird dieser durch die Raute gewährte Platz für die Dokumentation genutzt. Das für den Template-Designer angewendete und auch in [25] empfohlene Vorgehen ist es jedoch, mehrzeilige Kommentare zu nutzen. Diese werden mit einem dreifachen doppelten Hochkomma umschlossen und dürfen sich über mehrere Zeilen erstrecken. Auch für eigentlich einzeilige Dokumentation wird das dreifache doppelte Hochkomma empfohlen, weil sich auf diese Art eine eventuelle Erweiterung der Dokumentation viel schneller realisieren lässt. Würde man die Raute verwenden, müsste man diese bei einer Erweiterung der Dokumentation erst entfernen und durch die drei doppelten Hochkommata ersetzen: Ein ungleich höherer Aufwand also.

```
1 # Einzeiliger Kommentar
2
3 """ Mehrzeiliger Kommentar
4
5 Umschlossen von dreifachen
6 Hochkommata
7 """
```

Listing 6.10: Beispiele für einzeilige und mehrzeilige Docstrings

Texte, die in Python dokumentierenden Charakter haben, also mit einer Raute eingeleitet oder von drei doppelten Hochkommata umschlossen sind, werden auch *Docstring* genannt. Damit der DocUtils-Parser die Docstrings ausliest, müssen diese direkt auf die Deklaration eines Moduls, einer Klasse, Methode oder Funktion folgen. Diese Festlegung erscheint logisch, weil sie den Entwickler bei der Erstellung der Dokumentation zwingt, einen Docstring zu erzeugen. Würde der Docstring vor der Deklaration einer der eben genannten Strukturen erfolgen, könnte die Dokumentation beim Verschieben von Quelltexten verloren gehen. Ein Docstring am Ende dieser Strukturen hätte den Nachteil, dass er die schnelle Erweiterung der Struktur behindern könnte und auch hier besteht die Gefahr bei Umstrukturierungen der Quelltexte verloren zu gehen. Die Behinderung bestünde darin, dass erst durch das Einfügen neuer Zeilen Platz für neuen Quelltext geschaffen werden müsste. Ferner birgt diese konvention für den Parser Vorteile bei der Suchgeschwindigkeit, weil auf diese Art und Weise nur der erste Docstring nach einer Deklaration ausgewertet werden muss. Alle Weiteren DocStrings können einfach ignoriert werden und sind somit nur für die Entwickler interessant, die sich nicht mit den Schnittstellen, sondern mit den Quelltexten selbst auseinandersetzen.

Eine weitere Form der Docstrings, stellen sogenannte *Attribute Docstrings* dar. Diese werden genauso gebildet wie Docstrings, sind jedoch immer nach einer Zuweisung zufinden, die den Wert einer im aktuellen Kontext global gültigen Variable oder Konstante festlegt. Dadurch können diese wichtigen Strukturen auch erfasst und ihre jeweilige Bedeutung dokumentiert werden. Dies ist sehr wichtig, weil der Wert dieser Variablen/-Konstanten oft entscheidend das Verhalten einer Struktur beeinflusst.

Eine letzte Form der Docstrings sind die *Additional Docstrings*. Diese sind direkt nach der Notation eines regulären Docstrings zu finden und werden ebenso gebildet, beinhalten jedoch zusätzliche Informationen, die zwar interessant sein können, jedoch keinen direkten Einfluss auf das Verhalten der mit dem Docstring beschriebenen Struktur haben. Über diesen Mechanismus hat der Entwickler somit bereits die Möglichkeit, eine Gewichtung der Informationen nach deren Bedeutung vorzunehmen.

6.3.4.1 Innere Struktur von Docstrings

Die innere Struktur von Docstrings kann als eigenes Thema angesehen. Hier reichen die Möglichkeiten über der Eingabe von PlainText ohne Berücksichtigung irgendeiner inneren Strukur bis zur direkten Eingabe von XML- oder TeX-Strukturen bzw. -Daten. Die direkte Eingabe von z.B. XML-Daten würde zwar für eine sehr strenge Strukturierung sorgen und auch die Abarbeitung der Strukturen in der Parser und Writer Komponente wäre weniger anspruchsvoll, jedoch wäre die Dokumentation um einiges unübersichtlicher und auch schwerer interpretierbar als PlainText, wenn an den Quelltexten gearbeitet wird. In diesem Fall wären die XML-Tags hinderlich beim Lesen und auch die Erstellung würde ohne spezielle Werkzeuge einen höheren Zeitaufwand bedeuten.

Es besteht ebenfalls die Möglichkeit dem Parser mitzuteilen, welche innere Struktur in einer Datei verwendet wird. Dies ermöglicht es, das Quelltext unterschiedlichen Ursprungs, der jeweils eine andere innere Struktur aufweist, auf eine einheitliche Art präsentiert werden kann. Dazu muss am Anfang jeder Datei die Variable __docformat__ mit einem Wert belegt werden, den der Parser verstehen kann und daran sein Verhalten ausrichtet.

```
1 __docformat__ = "restructuredtext"
```

Listing 6.11: Bekanntmachung der inneren Struktur eines Docstrings, hier RST

Außerdem kann dem Parser über diese Variable auch die in den Docstrings verwendete Sprache mitgeteilt werden. Dazu muss nach dem Schlüsselwort für die innere Struktur der Docstrings ein Leerzeichen und anschließend der Sprachcode der jeweils verwendeten Sprache gemäß [2] und [33] angefügt werden.

```
1 __docformat__ = "plaintext de"
```

Listing 6.12: Bekanntmachung der inneren Struktur eines Docstrings, hier Plaintext in deutscher Sprache

Für den Template-Designer wurde sowohl PlainText, als auch der Ansatz reStructuredText (RST) gewählt.

6.3.4.2 PlainText

PlainText haften, gemäß seinem Namen, keine weiteren Bedeutungsstrukturen an. Dennoch empfiehlt [25] auch für PlainText einige grundlegende Regeln, die die Nutzbarkeit der Docstrings erhöhen sollen und diese zu einem gewissen Grad vereinheitlichen. Bereits genannt wurde die Empfehlung, nur dreifache Hochkommata zu verwenden und auf einzeilige Docstrings zu verzichten.

Eine weitere Empfehlung ist es, in einzeiligen Docstrings auf überflüssige Zeilenumbrüche zu verzichten. Das bedeutet, der dokumentierende Texte soll in der Zeile beginnen, in der der Docstring beginnt, und der Docstring soll enden, wo der dokumentierende Text endet. Dies ergibt durchaus Sinn, verknappt es doch den benötigten Platz auf ein Minimum und erhöht somit die Lesbarkeit deutlich, weil das Auge des Entwicklers geringere Wege zurücklegen muss. Weiter brächten solche Zeilenumbrüche auch dem Anwender keinen Vorteil, weil diese durch das Verwaltungssystem sowieso entfernt würden bzw. beim Einsatz von XML als Ausgabeformat irrelevant wären.

DocStrings sollten in kurzer prägnanter Sprache formuliert sein und mit einem Punkt abgeschlossen werden. Betrachtet der Entwickler den reinen Quelltext, erscheint der Punkt im ersten Augenblick überflüssig, weil die dreifachen Hochkommata den DocString

sowieso begrenzen und moderne Editoren eine Syntaxhervorhebung besitzen, die den DocString klar vom übrigen Quelltext abhebt. Jedoch besitzt der Anwender diese Hilfen nicht. Hier ist der Punkt eine Stütze im Verständnis der Dokumentation und außerdem gehört korrekte Interpunktion zum guten Ton. Interessant ist dieser Punkt deshalb, weil der Vorschlag, den Punkt zu setzen und somit die inhaltliche Qualität zu steigern, in einem Dokument ausgeführt wird, dass sich ansonsten eher den technischen Aspekten der Quelltextdokumentation widmet.

Eine weitere inhaltliche Anmerkung betrifft die Formulierung des DocStrings. Texte sollten in kurzer prägnanter Sprache formuliert sein. Sie sollen nicht beschreiben, sondern so formuliert sein, wie die aktuelle Struktur auch innerhalb der Anwendung Verwendung findet. Ein Beispiel erläutert dies. Eine unerwünschte Formulierung wäre demnach zum Beispiel:

```
1 """ This method returns returns an integer
2 representing the current status. """
```

Listing 6.13: Fehlerhaft formulierte Funktionsbeschreibung

An dieser Formulierung ist prinzipiell nichts auszusetzen, jedoch erscheint die Aussage ziemlich lang, wenn man die vorgeschlagene Schreibweise betrachet.

```
1 """ Return the current status as integer value. """
```

Listing 6.14: Korrekt formulierte Funktionsbeschreibung

Diese Aufforderdung spricht interessanter Weise nicht den Anwender, sprich Leser, an, sondern die Struktur als solche. Ob dass im ersten Augenblick dem Verständnis des unbedarften Lesers dient, müsste untersucht werden, jedoch erspart es dem versierten Leser aufgrund der Kürze und Prägnanz wertvolle Zeit und beinhaltet dennoch den selben Informationsgehalt, wie erstere Formulierung.

Für mehrzeilige Kommentare wird weiterhin eine Regelung vorgeschlagen, um Inhalte zu gewichten. Demnach enthält die erste Zeile dann eine zusammenfassende Beschreibung des Tuns der Funktion, während alle weiteren nach einer folgenden Leerzeile folgenden Zeilen, detailiertere Beschreibungen enthalten, die z.B. auf die Funktionsweise und die Schnittstellen einer Funktion eingehen. Das ist ein interessanter Ansatz, ermöglicht er doch auf intuitive Art eine klare Strukturierung. Die erste Zeile wirkt dadurch wie eine Überschrift. Eine andere Möglichkeit wäre es gewesen, auf die Zusammenfassung in der ersten Zeile zu verzichten, schließlich beinhalten die detailierteren Beschreibungen prinzipiell die selben Informationen und die Erstellung des Docstrings würde ebenfalls beschleunigt. Jedoch kommt man hier ganz klar den Interessen des Anwenders entgegen, der so schon auf den ersten Blick beurteilen kann, ob die jeweilige Struktur für ihn von Belang ist. Im nächsten Schritt kann er dann überprüfen, ob ihm die Funktion wirklich weiterhilft.

6.3.4.3 reStructuredText

Die zweite verwendete Form, *reStructuredText* (RST), verfolgt den sogenannten WYSIWYM (*What You See Is What You Mean*)-Ansatz. Dabei wird versucht auf komplexe Verwaltungsstrukturen, wie sie bei XML mit seinen Tags zu finden sind, zu verzichten und den Inhalten stattdessen durch ihre Anordnung eine formative Bedeutung zu geben. RST funktioniert damit ähnlichen Prinzipien wie wie LaTeX.

ReStructuredText wird auch in [42] explizit genannt und stellt somit eine gute Empfehlung dar, um Docstrings über das Niveau von PlainText zu heben. Im Folgenden sind einige Beispiele zu Nutzung RST beschrieben, die die Vorzüge und Nachteile von RST demonstrieren sollen.

6.3.4.4 Abschnitte

ReStructuredText bietet bei ausführlicheren Dokumentationsbestrebungen die Möglichkeit Abschnitte in Texte einzufügen. Dazu werden Überschriften notiert, denen eine neue Zeile mit einer bestimmten Zahl eines der folgenden Zeichen folg.

```
= - ‘ : ’ " ~ ^ _ * + # < >
```

Listing 6.15: Erlaubte Zeichen zur Auszeichnung von Überschriften

Die Anzahl muss mindestens so groß sein, wie die Anzahl der Buchstaben und Ziffern des Überschriftentextes. Auf welcher Gliederungsebene sich die jeweilige Überschrift befindet, hängt einzig von der Reihenfolge der eben genannten Zeichen ab. Bekommt eine angenommene erste Überschrift die Unterstreichung mit *:*-Zeichen und eine weitere Überschrift eine Unterstreichung mit *+*-Zeichen, so sind fortan alle Überschriften der ersten Gliederungsebene mit Doppelpunkten zu unterstreichen und alle Überschriften der zweiten Gliederungsebene mit Plus-Zeichen.

```
""" ...
Contents
::::::::

Das sind Inhalte zur Ueberschrift erster Ordnung

License
+++++++

Das sind Inhalte zur Ueberschrift zweiter Ordnung

... """
```

Listing 6.16: Beispiel für die Auszeichnung von Überschriften

Insgesamt ist zu begrüßen, dass die Möglichkeit dieser Strukturierung besteht, jedoch muss man hier in Ansätzen auf das später genutzte Ausgabemedium der Dokumentation achten. RST nimmt dem Entwickler nicht ab, die jeweils eingeführten Abschnitte zu nummerieren. Wird die spätere Ausgabe eine primär XML-basierte Lösung sein, so wäre es kritisch, die Nummerierung der Abschnitte in den Überschriften zu nennen, weil die Gliederung schon durch die Unterstreichungen vorgegeben ist. Dies könnte auch bei Übersetzungen unter zu Hilfenahme eines TMS problematisch sein. Verzichtet man jedoch aus den eben genannten Gründen auf eine explizite Nummerierung, so haben Anwender, die die Dokumentation als PlainText lesen, Probleme sich innerhalb der Abschnitte zurecht zufinden. Hier muss im Idealfall eine projektinterne Lösung, sprich einmalige Festlegung, hinsichtlich der Ziele der Dokumentation getroffen werden. Dabei ist auch zu berücksichtigen, wie eventuell von Dritten erstellte und eingebundene Quelltexte dokumentiert sind. Außerdem zeigt sich an diesem Beispiel die Komplexität des Parsings, denn die Unterstreichung darf in keinem Fall kürzer sein, als der Name des Abschnitts. Im schlimmsten Fall kann eine Dokumentation nicht zusammengestellt werden, weil es Fehler in der Struktur eines Docstrings gibt.

6.3.4.5 Absätze

Ein Absatz ist ein zusammenhängender Text, der nicht durch eine Leerzeile getrennt ist und bei dem jede neue Zeile auf der selben Einrückungsebene beginnt. Die Einrückungsebene bezeichnet die Zahl der Leerzeichen am Beginn einer Zeile, die vor dem eigentlichen Text stehen.

[caption=Korrekt formulierte Absatz, label=fig:qdrightp]

```
""" Ich bin ein Absatz.

Ich bin ein weiterer Absatz. """
```

[caption=Fehlerhaft formulierte Absatz, label=fig:qdwrongp]

```
""" Ich bin ein fehlerhafter Absatz, weil meine Einrueckungsebene durch
     die Leerzeichen am Beginn der Zeile eine andere ist. """
```

Formatierung

Texte, die kursiv dargestellt werden sollen, müssen von einem Asterisk, dem Sternchen (), umschlossen werden. Texte, die fett dargestellt werden sollen, müssen von zwei Asterisken umschlossen werden. Dicktengleiche Texte, werden von mit zwei Gravis umschlossen. Diese dürfen nicht mit dem Doppelgravis verwechselt werden. Ein Gravis wird der Informatik auch Backtick genannt.

[caption=Beispiele zur Formatierung von Texten, label=fig:qdformat]

```
""" Ich bin ein normaler *und ich ein kursiver* Text. """

 ...

""" Ich bin ein normaler **und ich ein fetter** Text. """

 ...

""" Ich bin normaler ``und ich ein dicktengleicher`` Text. """
```

Escape-Sequenz

Insgesamt ein Nachteil von RST ist, dass für die Strukturierung von Texten Zeichen verwendet werden, die selbst reine Zeichen sind. Das hat zwar den Vorteil, dass sich RST in seiner Ausgangsform sehr gut lesen lässt und schnell zu schreiben ist, weil keine größeren Anstrengungen unternommen werden müssen, um eine Formatierung und Strukturierung sicherzustellen. Es hat aber auch den Nachteil, dass die für die Strukturierung verwendeten Zeichen nicht mehr für die eigentlichen Inhalte zu Verfügung stehen und es unbeabsichtigt zu Fehlinterpretationen kommen kann.

Für diesen Fall gibt es die sogenannte Escape-Sequenz (Escape, engl. Flucht, Rettung). Dies ist der umgekehrte Schrägstrich (
, Backslash). Alle Zeichen, die direkt hinter dem umgekehrten Schrägstrich erscheinen, werden nicht hinsichtlich ihrer strukturellen oder formativen Bedeutung interpretiert. Das ist zwar eine Lösung des Problems, die aus der Welt der Programmiersprachen entlehnt und Entwicklern daher sehr vertraut ist, jedoch ist die Gefahr sehr groß, dass an dieser Stelle Fehler geschehen und der Übersichtlichkeit dient diese Lösung auch nicht. Fehler sind insoweit problematisch, dass Docstrings nicht während der Eingabe direkt interpretiert werden und schon gar nicht auf strukturelle Mängel hin untersucht werden. Dies geschieht erst in einem späteren Schritt.

```
""" Ich bin ein normaler \*und ich normaler\* Text.

Ich bin ein normaler \ *und ich kursiver\* Text. """
```

Listing 6.17: Korrekt und Fehlerhaft angewandte Escape-Sequenz

6.3.4.6 Listen

Listen funktionieren in ganz ähnlicher Weise wie Absätze und Zitate. Es existieren Möglichkeiten zur Erzeugung von Aufzählungslisten, nummerierten Listen und Definitionslisten, wobei es mit Ausnahme des menschlichen Interpretationsvermögens keine

Grenzen hinsichtlich deren Länge oder Verschachtelungstiefe gibt. Neue Listen müssen immer in einem neuen Absatz beginnen, also durch eine Leerzeile vom vorherigen Text getrennt sein. Um einen Listeneintrag zu erzeugen gilt grundsätzlich, dass dieser auf einer neuen Zeile beginnen und mit einem der in der folgenden Liste genannten Zeichen beginnen muss, wobei dieses von einem Punkt oder schließenden runden Klammer gefolgt oder von runden Klammern umgeben sein muss.

1 geordnete Liste mit arabsichen Zahlen
A geordnete Liste mit römischen Großbuchstaben
a geordnete Liste mit römischen Kleinbuchstaben
I geordnete Liste mit römischen Zahlen in großer Schreibweise
i geordnete Liste mit römischen Zahlen in kleiner Schreibweise

Aufzählungsliste mit Kreissymbol
– Aufzählungsliste mit Strichsymbol
+ Aufzählungsliste mit Quadratsymbol

Listeneinträge dürfen ebenfalls über Absätze verfügen. Diese Absätze sind mit einer Leerzeile voneinander getrennt und beginnen alle auf der selben Einrückungsebene. Die Verschachtelung von Listen ist möglich, in dem eine neue Liste auf einer tieferen Eirückungsebene begonnen wird.

Die Formalie, dass Listen und die Absätze in Listeneinträgen stehts durch eine Leerzeile getrennt sein müssen, lässt Listen in ihrer Ausgangsform sehr länglich wirken. Das erschwert ihre Wartbarkeit sehr und auch für den Entwickler kann dies ein Problem werden, weil der Quelltext dadurch stark an Länge gewinnt und es schwieriger wird, entscheidende und für den Entwickler gerade notwendige Quelltextpassagen zu finden.

Kritisch anzumerken ist in auch, dass gleich drei Möglichkeiten bestehen, die Typdefinition eines Listeneintrags von seinem Inhalt zu trennen. Das verkompliziert die Erstellung und Lesbarkeit an dieser Stelle, weil ein Entwickler die eine Syntax und ein anderer Entwickler eine andere Syntax präferiert. Es gibt außerdem keine automatische Nummerierung von Listen Elementen. Wenn ein Listenpunkt am Beginn einer Liste eingefügt wird, muss die Listennummerierung umständlich per Hand angepasst werden. Das ist zwar für die Ausgabe von XML-Daten unproblematisch, weil die Nummerierung sehr wahrscheinlich durch einen Automatismus ersetzt wird, jedoch bei der Darstellung von PlainText ein sehr unschöner Umstand. Auch hier wäre eine automatische Nummerierung wünschenswert.

Es ist grundsätzlich zu überlegen, ob Listen im jeweiligen Fall ein geeignetes Mittel zu Darstellung von Inhalten sind. Nicht weil Sie im Ergebnis unpraktisch wären, sondern

weil der Aufwand zur Erstellung und die Pflege von Listen aufwendig und fehleranfällig erscheint.

6.3.4.7 Querverweise

Querverweise stellen in RST eine sehr praktische Möglichkeit dar, durch Nennung einer Struktur eine beim jeweiligen Ausgabemedium eventuell mögliche Verlinkung zu realisieren. Dazu muss der Name der Struktur jeweils von einem einfachen Gravis umgeben sein. Gut an dieser Stelle ist die unkomplizierte Syntax, die das Lesen des Docstrings auch als PlainText nicht behindert und sogar deutlich darauf hinweist, dass die Benennung auf eine Funktion zeigt. Problematisch bei dieser Möglichkeit, wie allerdings bei jedem Querverweis, ist die Tatsache, dass auf ein eventuell nicht mehr existentes Ziel verwiesen wird. Hier bleibt es der Writer Komponente überlassen, eventuell nicht verfügbare Ziele aufzulösen oder den entsprechenden Querverweis zu eliminieren.

6.3.5 Schlussfolgerung

Texte mit einer direkten Formatierung zu versehen, wie das reStructuredText vorsieht, ist hier wie in anderen Situation, mit Vorsicht zu genießen. Das Problem besteht darin, dass eine direkte Formatierung keine Semantik beinhaltet. Der Text wird zwar formatiert, jedoch kann unklar sein, wofür die Formatierung steht, wenn keine klaren Regeln dafür vorgegeben sind. Gerade bei Software-Projekten, die oft von mehreren Entwicklern geschrieben werden, erscheint es schwierig, eine gewisse Konsistenz und damit Eindeutigkeit zu erzeugen. Deshalb sollte hier, wie in XML, festgelegt sein, dass eine bestimmte Formatierung einer bestimmten Semantik zugeordnet ist. Zum Teil geschieht das bereits, z.B. mit Querverweisen. Dabei muss allerdings darauf geachtet werden, dass die Qualität der Dokumentation in einer immer gegebenen PlainText-Darstellung nicht darunter leidet.

Insgesamt stellt RST eine interessante Möglichkeit dar, Docstrings zu strukturieren und Anderen mit einfachen Mitteln zugänglich zu machen. Aufgrund der großen Fülle formativer Möglichkeiten [44], ist es jedoch nahezu unabdingbar sich für ein Projekt auf nur wenige der vielen Möglichkeiten zu begrenzen. Durch die sehr große Anzahl der Gestaltungsmöglichkeiten muss man als Entwickler aufpassen sich nicht zu verlieren. Für den Template-Designer wurden daher mit voller Absicht, nur die obig beschriebenen rudimentären Gestaltungsmöglichkeiten genutzt. Dies geschah genau in der Absicht, sich nicht in den vielen Möglichkeiten zu verlieren und damit Unklarheiten bezüglich der Bedeutung von Auszeichnungen aufkommen zu lassen.

7 Anhang

7.1 Quelltexte

```
<?xml version="1.0" encoding="utf-8"?>

<!--Structured content-->
<!ELEMENT task (goal, condition*, (action | sidegoal)*, result)>

<!ELEMENT goal (#PCDATA)>
<!ELEMENT condition (#PCDATA)>
<!ELEMENT sidegoal (#PCDATA)>
<!ELEMENT action (#PCDATA)>
<!ELEMENT result (#PCDATA)>

<!--Attributes with predefined values-->
<!ATTLIST task
  language (deutsch | english | francais | espanol | italiono |
      portuguesa) "deutsch"
  author CDATA #REQUIRED>
```

Listing 7.1: DTD für Beispiel-XML aus Listing 2.1

```
<!ENTITY % nomath  "#PCDATA|TeXML|cmd|env|group|ctrl|spec|pdf">
<!ENTITY % content "%nomath;|math|dmath">

<!ELEMENT TeXML (%content;)*>
<!ATTLIST TeXML mode (text|math) #IMPLIED
                escape     (0|1) #IMPLIED
                emptylines (0|1) #IMPLIED
                ligatures  (0|1) #IMPLIED
                ws         (0|1) #IMPLIED>

<!ELEMENT env (%content;|opt|parm)*>
<!ATTLIST env name  CDATA #REQUIRED
              begin CDATA #IMPLIED
              end   CDATA #IMPLIED
              start CDATA #IMPLIED
              stop  CDATA #IMPLIED
              nl1   (0|1) #IMPLIED
              nl2   (0|1) #IMPLIED
              nl3   (0|1) #IMPLIED
              nl4   (0|1) #IMPLIED>

<!ELEMENT group (%content;)*>
<!ELEMENT math  (%nomath;)*>
<!ELEMENT dmath (%nomath;)*>

<!ELEMENT cmd    (opt|parm)*>
<!ATTLIST cmd name CDATA #REQUIRED
              nl1  (0|1) #IMPLIED
              nl2  (0|1) #IMPLIED
              gr   (0|1) #IMPLIED>

<!ENTITY % inopt "%content;">

<!ELEMENT opt    (%inopt;)*>
<!ELEMENT parm   (%inopt;)*>

<!ELEMENT ctrl EMPTY>
<!ATTLIST ctrl ch CDATA #REQUIRED>

<!ELEMENT spec   EMPTY>
<!ELEMENT pdf    (#PCDATA)>

<!ATTLIST spec cat CDATA #REQUIRED>
```

Listing 7.2: TeXML DTD

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
                xmlns="http://getfo.sourceforge.net/texml/ns1"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" version="1.0" encoding="utf-8" indent="no"/>

  <xsl:template match="/">
    <TeXML xmlns="http://getfo.sourceforge.net/texml/ns1">
      <cmd name="documentclass" nl2="1">
        <parm>dokumentklasse</parm>
      </cmd>
      <cmd name="usepackage" nl2="1">
        <parm>dokumentstil</parm>
      </cmd>
        <xsl:call-template name="docInfoGoal"/>
        <xsl:call-template name="docInfoAuthor"/>
        <env name="document">
          <xsl:apply-templates/>
        </env>
    </TeXML>
  </xsl:template>

  <xsl:template name="docInfoGoal">
    <cmd name="title" nl2="1">
      <parm><xsl:value-of select="/task/goal"/></parm>
    </cmd>
  </xsl:template>

  <xsl:template name="docInfoAuthor">
    <cmd name="author" nl2="1">
      <parm><xsl:value-of select="/task/@author"/></parm>
    </cmd>
  </xsl:template>

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

  <xsl:template match="//goal">
    <cmd name="section">
      <parm>
        <xsl:call-template name="goaltext"/>
        <xsl:text>: </xsl:text>
        <xsl:value-of select="."/>
      </parm>
    </cmd>
  </xsl:template>

  <xsl:template match="//condition">
    <cmd name="condition" nl2="1">
      <parm><xsl:value-of select="."/></parm>
    </cmd>
  </xsl:template>
```

```
54
55    <xsl:template match="//action">
56      <cmd name="action" nl2="1">
57        <parm><xsl:value-of select="."/></parm>
58      </cmd>
59    </xsl:template>
60
61    <xsl:template match="//sidegoal">
62      <cmd name="sidegoal" nl2="1">
63        <parm><xsl:value-of select="."/></parm>
64      </cmd>
65    </xsl:template>
66
67    <xsl:template match="//result">
68      <cmd name="result" nl2="1">
69        <parm><xsl:value-of select="."/></parm>
70      </cmd>
71    </xsl:template>
72
73    <xsl:template name="goaltext">
74      <xsl:variable name="lang">
75        <xsl:call-template name="goalheading">
76          <xsl:with-param name="param" select="/task/@language"/>
77        </xsl:call-template>
78      </xsl:variable>
79      <xsl:value-of select="$lang"/>
80    </xsl:template>
81
82    <xsl:template name="goalheading">
83      <xsl:param name="param">deutsch</xsl:param>
84      <xsl:if test="$param='deutsch'">Vorhaben</xsl:if>
85      <xsl:if test="$param='english'">Task</xsl:if>
86      <xsl:if test="$param='francais'">Tsche</xsl:if>
87      <xsl:if test="$param='spanish'">Tarea</xsl:if>
88      <xsl:if test="$param='italiono'">Attivita</xsl:if>
89      <xsl:if test="$param='portugese'">Tarefa</xsl:if>
90    </xsl:template>
91
92  </xsl:stylesheet>
```

Listing 7.3: Stylesheet für Transformation nach TeXML

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.bitplant.de/template"
  xmlns="http://www.bitplant.de/template"
  elementFormDefault="unqualified"
  attributeFormDefault="unqualified">

<xs:annotation>
  <xs:documentation>This xml schema defines Bitplant-TemplateXML.
  This xml is used used by Bitplant Template-Designer to manage
  LaTeX-Styles and in the future XSL-FO-Styles. It is thought as easy
  to access format to define simple page layouts used within the
  TeXML publishing process.</xs:documentation>
</xs:annotation>

<xs:element name="designer" type="designerType"/>

<xs:complexType name="designerType">
  <xs:sequence>
    <xs:element name="template" type="templateType" minOccurs="0"
        maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attributeGroup ref="identifyGroup"/>
</xs:complexType>

<xs:element name="template" type="templateType"/>

<xs:complexType name="templateType">
  <xs:sequence>
    <xs:element name="parameter" type="parameterType" minOccurs="1"
        maxOccurs="1"/>
    <xs:element name="page" type="pageType" minOccurs="0" maxOccurs="
        unbounded"/>
  </xs:sequence>
  <xs:attributeGroup ref="identifyGroup"/>
</xs:complexType>

<xs:complexType name="pageType">
  <xs:sequence>
    <xs:element name="parameter" type="parameterType" minOccurs="1"
        maxOccurs="1"/>
    <xs:element name="frame" type="frameType" minOccurs="0" maxOccurs="
        unbounded"/>
  </xs:sequence>
  <xs:attributeGroup ref="identifyGroup"/>
  <xs:attribute name="inherit" type="inheritAttrType" default="enable"/>
</xs:complexType>

<xs:complexType name="frameType">
  <xs:sequence>
    <xs:element name="parameter" type="parameterFrameType" minOccurs="1"
        maxOccurs="1"/>
```

```
47      <xs:element name="content" type="contentType" minOccurs="1" maxOccurs
            ="1"/>
48    </xs:sequence>
49    <xs:attributeGroup ref="identifyGroup"/>
50  </xs:complexType>
51
52  <xs:complexType name="parameterType">
53    <xs:sequence>
54      <xs:element name="description" type="descriptionType" minOccurs="0"
            maxOccurs="1"/>
55      <xs:element name="dimension" type="dimensionType" minOccurs="0"
            maxOccurs="2"/>
56      <xs:element name="position" type="positionType" minOccurs="0"
            maxOccurs="4"/>
57      <xs:element name="paper" type="paperType" minOccurs="0" maxOccurs="3"
            />
58    </xs:sequence>
59  </xs:complexType>
60
61  <xs:complexType name="parameterFrameType">
62    <xs:sequence>
63      <xs:element name="description" type="descriptionType" minOccurs="0"
            maxOccurs="1"/>
64      <xs:element name="dimension" type="dimensionType" minOccurs="0"
            maxOccurs="2"/>
65      <xs:element name="position" type="positionType" minOccurs="0"
            maxOccurs="4"/>
66    </xs:sequence>
67  </xs:complexType>
68
69  <xs:simpleType name="descriptionType">
70    <xs:restriction base="xs:string"/>
71  </xs:simpleType>
72
73  <xs:complexType name="dimensionType">
74    <xs:attribute name="type" type="dimensionAttrType" use="required"/>
75    <xs:attribute name="unit" type="unitType" default="mm"/>
76    <xs:attribute name="value" type="valueDecimalType" use="required"/>
77  </xs:complexType>
78
79  <xs:complexType name="positionType">
80    <xs:attribute name="type" type="positionAttrType" use="required"/>
81    <xs:attribute name="unit" type="unitType" default="mm"/>
82    <xs:attribute name="value" type="valueDecimalType" use="required"/>
83  </xs:complexType>
84
85  <xs:complexType name="paperType">
86    <xs:attribute name="type" type="paperAttrType" use="required"/>
87    <xs:attribute name="value" type="valuePaperType" use="required"/>
88  </xs:complexType>
89
90  <xs:complexType name="contentType">
91    <xs:simpleContent>
```

```
92      <xs:extension base="xs:string">
93        <xs:attribute name="type" type="contentAttrType" use="required"/>
94        <xs:attribute name="angle" type="angleAttrType" default="0"/>
95      </xs:extension>
96    </xs:simpleContent>
97  </xs:complexType>
98
99  <xs:simpleType name="dimensionAttrType">
100   <xs:restriction base="xs:string">
101     <xs:enumeration value="height"/>
102     <xs:enumeration value="width"/>
103   </xs:restriction>
104 </xs:simpleType>
105
106 <xs:simpleType name="positionAttrType">
107   <xs:restriction base="xs:string">
108     <xs:enumeration value="left"/>
109     <xs:enumeration value="right"/>
110     <xs:enumeration value="top"/>
111     <xs:enumeration value="bottom"/>
112   </xs:restriction>
113 </xs:simpleType>
114
115 <xs:simpleType name="paperAttrType">
116   <xs:restriction base="xs:string">
117     <xs:enumeration value="orientation"/>
118     <xs:enumeration value="layout"/>
119     <xs:enumeration value="format"/>
120   </xs:restriction>
121 </xs:simpleType>
122
123 <xs:simpleType name="valuePaperType">
124   <xs:restriction base="xs:string">
125     <xs:enumeration value="portrait"/>
126     <xs:enumeration value="landscape"/>
127
128     <xs:enumeration value="oneside"/>
129     <xs:enumeration value="twoside"/>
130
131     <xs:enumeration value="a4"/>
132     <xs:enumeration value="b4"/>
133     <xs:enumeration value="kiku4"/>
134     <xs:enumeration value="letter"/>
135     <xs:enumeration value="cdsingle"/>
136
137   </xs:restriction>
138 </xs:simpleType>
139
140 <xs:simpleType name="unitType">
141   <xs:restriction base="xs:string">
142     <xs:enumeration value="mm"/>
143     <xs:enumeration value="cm"/>
144     <xs:enumeration value="inch"/>
```

```
145     <xs:enumeration value="pt"/>
146   </xs:restriction>
147 </xs:simpleType>
148 
149 <xs:simpleType name="valueDecimalType">
150   <xs:restriction base="xs:decimal"/>
151 </xs:simpleType>
152 
153 <xs:simpleType name="inheritAttrType">
154   <xs:restriction base="xs:string">
155     <xs:enumeration value="enable"/>
156     <xs:enumeration value="disable"/>
157   </xs:restriction>
158 </xs:simpleType>
159 
160 <xs:simpleType name="contentAttrType">
161   <xs:restriction base="xs:string">
162     <xs:enumeration value="color"/>
163     <xs:enumeration value="image"/>
164     <xs:enumeration value="text"/>
165     <xs:enumeration value="vartext"/>
166   </xs:restriction>
167 </xs:simpleType>
168 
169 <xs:simpleType name="angleAttrType">
170   <xs:restriction base="xs:integer">
171     <xs:enumeration value="0"/>
172     <xs:enumeration value="90"/>
173     <xs:enumeration value="180"/>
174     <xs:enumeration value="270"/>
175   </xs:restriction>
176 </xs:simpleType>
177 
178 <xs:attributeGroup name="identifyGroup">
179   <xs:attribute name="name" type="xs:Name"  use="required"/>
180   <xs:attribute name="id" type="xs:ID" use="optional"/>
181   <xs:attribute name="lang" type="xs:language" use="optional">
182 </xs:attributeGroup>
183 
184 </xs:schema>
```

Listing 7.4: XML Schema von TemplateXML. Dieses Listing beinhaltet aus Platzgründen nicht die eigentlich enthaltene zugehörige Dokumenation und nur eine Auswahl der gestatteten Papierformate. Das vollständige XML Schema befindet sich auf dem beiliegenden Datenträger.

Literaturverzeichnis

[1] Ed. A. Phillips and Ed. M. Davis. Rfc4647 : Matching of language tags. Technical report, RFC Editor, September 2006.

[2] H. Alvestrand. Rfc1766: Tags for the identification of languages. Request for comments, RFC Editor, März 2005.

[3] Artifex. Licensing information. http://www.artifex.com/indexlicense.htm, zuletzt besucht am 22.11.2008, Mai 2008.

[4] Internet Assigned Numbers authority (IANA). language subtag registry. Registry, Internet Assigned Numbers authority (IANA), Oktober 2008.

[5] Prof. Dr. Peter Barth. Python ausnahmen, bibliotheken. http://www.mi.fh-wiesbaden.de/~barth/skripts/vorl/SkriptsprachenPB8.pdf, zuletzt besucht am 4.10.2008, 2005.

[6] Prof. Dr. Peter Barth. Python einführung und datentypen. http://www.mi.fh-wiesbaden.de/~barth/skripts/vorl/SkriptsprachenPB5.pdf, zuletzt besucht am 4.10.2008, 2005.

[7] Prof. Dr. Peter Barth. Python gui mit tkinter. http://www.mi.fh-wiesbaden.de/~barth/skripts/vorl/SkriptsprachenPB9.pdf, zuletzt besucht am 4.10.2008, 2005.

[8] Prof. Dr. Peter Barth. Python gui mit wxwidgets - zusatzinfo. http://www.mi.fh-wiesbaden.de/~barth/skripts/vorl/SkriptsprachenPB9b.pdf, zuletzt besucht am 4.10.2008, 2005.

[9] Prof. Dr. Peter Barth. Python kontrollstrukturen, funktionen, funktionales programmieren. http://www.mi.fh-wiesbaden.de/~barth/skripts/vorl/SkriptsprachenPB6.pdf, zuletzt besucht am 4.10.2008, 2005.

[10] Prof. Dr. Peter Barth. Python module, klassen. http://www.mi.fh-wiesbaden.de/~barth/skripts/vorl/SkriptsprachenPB7.pdf, zuletzt besucht am 4.10.2008, 2005.

[11] Jiri Barton. rest to html conversion. http://www.hosting4u.cz/jbar/rest/rest.html, Mai 2008.

[12] Ján Bodnár. The wxpython tutorial. http://zetcode.com/wxpython/, zuletzt besucht am 11.10.2008, November 2007.

[13] Ján Bodnár. The python tutorial. http://zetcode.com/tutorials/pythontutorial/, zuletzt besucht am 11.10.2008, Juli 2008.

[14] S. Bradner. Rfc2119: Key words for use in rfcs to indicate requirement levels. Request for comments, RFC Editor, März 1997.

[15] James Clark. Xml namespaces. http://www.jclark.com/xml/xmlns.htm, zuletzt besucht am 26.10.2008, Februar 1999.

[16] Sissi Closs. *Single-Source-Publishing – Topicorientierte Strukturierung und DITA*. entwickler.press: Software & Support Verlag GmbH, Dezember 2007.

[17] CollabNet. Subclipse. http://subclipse.tigris.org/, zuletzt besucht am 22.11.2008, November 2008.

[18] CollabNet. Subversion. http://subversion.tigris.org/, zuletzt besucht am 22.11.2008, November 2008.

[19] Stuart Colville. Python: Keychain.py access to the mac osx keychain. https://launchpad.net/keychain.py/, zuletzt besucht am 4.10.2008, Februar 2008.

[20] Roger L. Costello. Extending xml schemas (a collectively developed set of schema design guidelines). http://www.xfront.com/ExtendingSchemas.html, zuletzt besucht am 4.11.2008.

[21] Roger L. Costello. Xml schema country list. http://www.xfront.com/Countries.xsd, zuletzt besucht am 19.10.2008.

[22] Roger L. Costello. Xml schema measureunit list. http://www.xfront.com/CodeList_UnitsOfMeasureCode_XFront.xsd, zuletzt besucht am 19.10.2008.

[23] Comprehensive TeX Archive Network (CTAN). Comprehensive tex archive network (ctan). http://www.tug.org/ctan, zuletzt besucht am 21.11.2008, November 2008.

[24] Refsnes Data. Xml schema tutorial. http://www.w3schools.com/schema/, zuletzt besucht am 26.10.2008.

[25] Guido van Rossum David Goodger. Pep 257: Docstring conventions. Python enhancement proposal, Python Software Foundation, Juni 2007.

[26] Robin Dunn. wxpython documentation - class treectrl. http://www.wxpython.org/docs/api/wx.TreeCtrl-class.html, zuletzt besucht am 11.10.2008.

[27] Robin Dunn. wxpywiki. http://wiki.wxpython.org, zuletzt besucht am 21.11.2008.

[28] Robin Dunn. wxpython. http://www.wxpython.org/, zuletzt besucht am 21.11.2008, November 2008.

[29] Robin Dunn. wxpywiki - treecontrols. http://wiki.wxpython.org/TreeControls, zuletzt besucht am 21.11.2008, März 2008.

[30] International Organization for Standardization. International standard iso 8601:2004(e), third edition – data elements and interchange formats - information interchange - representation of dates and times. Technical report, International Organization for Standardization, Dezember 2004.

[31] International Organization for Standardization. Iso 3166 code lists. Technical report, International Organization for Standardization, März 2007.

[32] International Organization for Standardization. Iso 3166 maintenance agency (iso 3166/ma) - iso's focal point for country codes. Technical report, International Organization for Standardization, 2007.

[33] International Organization for Standardization. Iso 639-2 code list. Technical report, Library of Congress, Oktober 2008.

[34] Eclipse Foundation. Eclipse. http://www.eclipse.org/, zuletzt besucht am 22.11.2008, November 2008.

[35] Eclipse Foundation. Web tools platform (wtp) project. http://www.eclipse.org/webtools/, zuletzt besucht am 22.11.2008, November 2008.

[36] Python Software Foundation. Python software foundation license. http://www.python.org/download/releases/2.6/license/, zuletzt besucht am 21.11.2008, November 2006.

[37] Python Software Foundation. mail.python.org mailing lists. http://mail.python.org/mailman/listinfo, zuletzt besucht am 23.11.2008, November 2008.

[38] Python Software Foundation. Python software foundation. http://www.python.org/psf/, zuletzt besucht am 21.11.2008, November 2008.

[39] Gnu. Gnu general public license – version 2, june 1991. http://www.gnu.org/licenses/gpl-2.0.txt, zuletzt besucht am 22.11.2008, Juni 1991.

[40] Gnu. Diffutils. http://www.gnu.org/software/diffutils/diffutils.html, zuletzt besucht am 21.11.2008, November 2008.

[41] David Goodger. Pep 258: Docutils design specification. Python enhancement proposal, Python Software Foundation, Juni 2007.

[42] David Goodger. Pep 287: restructuredtext docstring format. Python enhancement proposal, Python Software Foundation, Juni 2007.

[43] David Goodger. Pep 256: Docstring processing system framework. Python enhancement proposal, Python Software Foundation, Oktober 2008.

[44] David Goodger. restructuredtext markup specification. Technical specification, DocUtils Project, Oktober 2008.

[45] XML Schema Working Group. Xml schema definition language: W3c xml schema working group and schema specifications. Technical report, Word Wide Web Consortium (W3C), 1999.

[46] XML Schema Working Group. Xml schema requirements. Technical report, Word Wide Web Consortium (W3C), Februar 1999.

[47] XML Schema Working Group. Charter of the xml schema working group. Technical report, Word Wide Web Consortium (W3C), April 2002.

[48] Richard Gruet. Python quick reference. `http://rgruet.free.fr/`, zuletzt besucht am 4.10.2008, April 2008.

[49] Barry Warsaw Guido van Rossum. Pep 8: Style guide for python code. Python enhancement proposal, Python Software Foundation, Juni 2008.

[50] IDEAAlliance. Xsl-fo chef's tools exhibition. `http://www.idealliance.org/proceedings/xml03/slides/xslfoshowcase/`, zuletzt besucht am 21.11.2008, September 2004.

[51] Adobe inc. Adobe - framemaker 8: Xml-editor, wysiwyg-xml-editor. `http://www.adobe.com/de/products/framemaker`, zuletzt besucht am 20.11.2008, Januar 2008.

[52] Apple Inc. Apple help programming guide. Technical report, Apple Inc., Oktober 2007.

[53] Apple Inc. Apple file system overview. Technical report, Apple Inc., Juli 2008.

[54] Apple Inc. Apple human interface guidelines. Technical report, Apple Inc., Juni 2008.

[55] Apple Inc. Introduction to the objective-c 2.0 programming language. `developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/`, zuletzt besucht am 21.11.2008, November 2008.

[56] Sun Microsystems Inc. The source for java developers. `java.sun.com/`, zuletzt besucht am 21.11.2008, November 2008.

[57] Donald Ervin Knuth. *The TeXbook - A Complete User's Guide to Computer typesetting with TeX*. Addison-Wesley Professional, spiralbund edition, Januar 1984.

[58] Anselm Lingnau. *Latex Hacks: Tipps & Techniken für den professionellen Textsatz*. O'Reilly Verlag GmbH & Co.KG, Mai 2007.

[59] Douglas Lovell. Texml: typesetting xml with tex. Tex user group annual meeting presentation paper, IBM Research, August 1999.

[60] Fredrik Lundh. Elementtree overview. `http://effbot.org/zone/element-index.htm`, zuletzt besucht am 22.11.2008.

[61] Fredrik Lundh. The elementtree.elementtree module. `http://effbot.org/zone/pythondoc-elementtree-ElementTree.htm`, zuletzt besucht am 22.11.2008.

[62] Fredrik Lundh. Python imaging library. `http://effbot.org/zone/pil-index.htm`, zuletzt besucht am 3.11.2008.

[63] Fredrik Lundh. Element library functions. `http://effbot.org/zone/element-lib.htm`, zuletzt besucht am 22.11.2008, März 2004.

[64] Fredrik Lundh. The python imaging library handbook. `http://effbot.org/imagingbook/`, zuletzt besucht am 3.11.2008, Mai 2005.

[65] Fredrik Lundh. Elements and element trees. `http://effbot.org/zone/element.htm`, zuletzt besucht am 22.11.2008, Juli 2007.

[66] M. Suignard M. Dürst. Rfc3987: Internationalized resource identifiers (iris). Request for comments, RFC Editor, Januar 2005.

[67] David Ascher Mark Lutz. *Learning Python*. O'Reilly Media, Inc., 2 edition, Dezember 2003.

[68] Robin Dunn Noel Rappin. *wxPython in Action*, volume 1. Manning Publications Co., März 2006.

[69] Uche Ogbuji. Python paradigms for xml. `http://www.idealliance.org/papers/dx_xml03/papers/06-02-03/06-02-03.html`, zuletzt besucht am 4.10.2008, Juni 2003.

[70] Ken Coar Open Source Initiative. The open source definition. `http://www.opensource.org/docs/osd`, zuletzt besucht am 21.11.2008, Juli 2006.

[71] D. Crocker P. Overell. Rfc4234: Augmented bnf for syntax specifications: Abnf. Request for comments, RFC Editor, Oktober 2005.

[72] Oleg Paraschenko. Texml specification. `http://getfo.org/texml/spec.html`, zuletzt besucht am 21.11.2008, Juli 2006.

[73] Dave Pawson. *XSL-FO*. O'Reilly & Associates, Inc., August 2002.

[74] Johannes Ernesti Peter Kaiser. *Python - Das umfassende Handbuch*. Galileo Press, Dezember 2007.

[75] Laurent Pointal. Python quick reference card. http://www.limsi.fr/Individu/pointal/python/pqrc/, zuletzt besucht am 4.10.2008, April 2007.

[76] Oxygen Icon Project. Oxygen. http://www.oxygen-icons.org, zuletzt besucht am 22.11.2008, Januar 2008.

[77] Pydev. Pydev. http://pydev.sourceforge.net/, zuletzt besucht am 22.11.2008, Oktober 2008.

[78] IBM Research. Texml: typesetting xml with tex. http://www.alphaworks.ibm.com/tech/texml, zuletzt besucht am 21.11.2008, Januar 2005.

[79] Ralf Schmitt. wxpywiki - treectrldnd. http://wiki.wxpython.org/TreeCtrlDnD, zuletzt besucht am 21.11.2008, März 2008.

[80] SQLite. Sqlite. http://www.sqlite.org/, zuletzt besucht am 22.11.2008, November 2008.

[81] R. Fielding T. Berners-Lee. Rfc3986: Uniform resource identifier (uri): Generic syntax. Request for comments, RFC Editor, Januar 2005.

[82] TeX User Group (TUG). Tex user group (tug). http://www.tug.org/, zuletzt besucht am 21.11.2008, November 2008.

[83] Frank Mittelbach und Michel Goosens und J.Braams uns D. Carlisle und C. Rowley. *Der LateX Begleiter*. Pearson Studium, 2. überarbeitete und erweiterte auflage (korrigierter nachdruck februar 2007) edition, Oktober 2005.

[84] Eric van der Vlist. Using w3c xml schema. http://www.xml.com/pub/a/2000/11/29/schemas/part1.html, zuletzt besucht am 4.11.2008, Oktober 2001.

[85] World Wide Web Consortium (W3C). Xml schema part 1: Structures second edition. W3c recommendation, World Wide Web Consortium (W3C), Oktober 2004.

[86] World Wide Web Consortium (W3C). xml:id version 1.0. W3c recommendation, World Wide Web Consortium (W3C), September 2005.

[87] World Wide Web Consortium (W3C). Extensible markup language (xml) 1.0 (fourth edition). W3c recommendation, World Wide Web Consortium (W3C), September 2006.

[88] World Wide Web Consortium (W3C). Extensible markup language (xml) 1.1 (second edition). W3c recommendation, World Wide Web Consortium (W3C), September 2006.

[89] World Wide Web Consortium (W3C). Extensible stylesheet language (xsl) version 1.1. W3c recommendation, World Wide Web Consortium (W3C), Dezember 2006.

[90] World Wide Web Consortium (W3C). Namespaces in xml 1.0 (second edition). W3c recommendation, World Wide Web Consortium (W3C), August 2006.

[91] World Wide Web Consortium (W3C). Namespaces in xml 1.1 (second edition). W3c recommendation, World Wide Web Consortium (W3C), Augst 2006.

[92] World Wide Web Consortium (W3C). Cascading style sheets. http://www.w3.org/Style/CSS/, zuletzt besucht am 21.11.2008, November 2008.

[93] World Wide Web Consortium (W3C). Css color module level 3. W3c working draft, World Wide Web Consortium (W3C), Juli 2008.

[94] World Wide Web Consortium (W3C). W3c xml schema definition language (xsd) 1.1 part 1: Structures. W3c working draft, World Wide Web Consortium (W3C), Juni 2008.

[95] World Wide Web Consortium (W3C). World wide web consortium - web standards. http://www.w3.org, zuletzt besucht am 21.11.2008, November 2008.

[96] WxWidgets. wxwidgets – cross-platform gui library. http://www.wxpython.org/, zuletzt besucht am 21.11.2008, October 2008.

[97] Unbekannt (Nickname Xoanan). Standard ways to get union, intersection, difference of lists? http://bytes.com/forum/thread19083.html, zuletzt besucht am 4.10.2008, August 2005.

[98] Sermet Yucel. Ruby - a programmers best friend. ruby-lang.org/, zuletzt besucht am 21.11.2008, Dezember 2004.

[99] Sermet Yucel. What does s̈wing is slowm̈ean? http://www.javalobby.org/articles/swing_slow/index.jsp, zuletzt besucht am 21.11.2008, Dezember 2004.