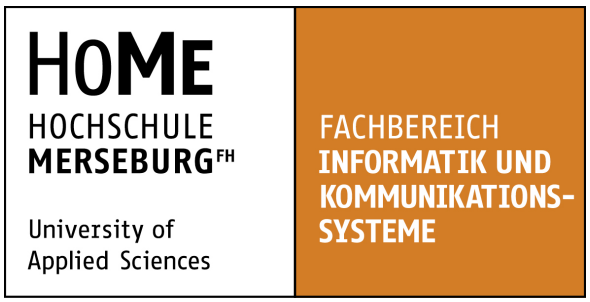
Masterthesis

Regelgestützte und interaktive Formatvorlagenzuweisung für XML-basierte Dokumentformate unter Verwendung von XProc, XSLT und JavaScript



Hendrik Schwede

Studiengang: Technische Redaktion und Wissenskommunikation

Fachbereich: Informatik und Kommunikationssysteme

Hochschule Merseburg

Diese Abschlussarbeit dient zur Erlangung des akademischen Grades

Master of Arts

Erstprüfer:	Dr. rer. nat. Thomas Meinike
Zweitprüfer:	Dipl.-Phys. Gerrit Imsieke
Matrikel-Nr.:	17028

Merseburg, d. 17.12.2015

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit mit dem Titel *Regelgestützte und interaktive Formatvorlagenzuweisung für XML-basierte Dokumentformate unter Verwendung von XProc, XSLT und JavaScript* selbständig verfasst habe, dass ich sie zuvor an keiner anderen Hochschule und in keinem anderen Studiengang als Prüfungsleistung eingereicht habe und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht.

Hendrik Schwede

Merseburg, d. 17.12.2015

Kurzfassung

Die Arbeit beschäftigt sich mit der Entwicklung einer Web-Anwendung zur regelgestützten Zuweisung von Formatvorlagen für XML-basierte Dokumentformate. Zur Erstellung der Zuweisungsregeln und Durchführung einer Vorlagenzuweisung werden die Programmierspache JavaScript und die XML-Verarbeitungstechnologien XProc und XSLT verwendet.

This thesis is about the developement of a web application for rule based allocation of document styles for XML document formats. To create the allocation rules the programming language JavaScript has been used while the xml languages XProc und XSLT have been used to perform the mapping process via xml pipeline processing and xml transformations.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcode-Verzeichnis

Tabellenverzeichnis

Nomenklatur

Akronyme/Abkürzungen

CSSa	Cascading Style Sheet as attribute properties
CSS	Cascading Style Sheets
HTML	Hypertext Markup Language
IDML	InDesign Markup Language
ODF	Opden Document Format
RDF	Ressource Description Framework
SVN	Apache Subversion
UFC	Universal Format Container
WordML	Wordprocessing Markup Language
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
XMP	Extensible Metadata Platform
XProc	XML Processing
XSLT	Extensible Stylesheet Language Transformation

Kapitel 1

Einleitung

1.1 Idee und Werkzeuge

Die Firma *le-tex publishing services GmbH* aus Leipzig entwickelte eine Konzeptidee zur Umstrukturierung von Dokumenten. Es sollte für die Autoren ihrer Verlagskunden möglich sein, alte Word-Dokumente und InDesign-Dateien mit neuen vorgegebenen Formatvorlagen für Zeichen, Absätze, Tabellen und Textrahmen zu versehen. Der Grundgedanke dabei ist, den Kunden möglichst viel technisches Verständnis über die bei *le-tex* verwendeten XML-Technologien zu ersparen und ihnen stattdessen selbst über intuitive Wege den Formatvorlagenersatz zu ermöglichen.

Als XML-Dienstleister kann diese Herausforderung bisher nur unter Einsatz spezifischer Lösungswege für jedes Kundenprojekt realisiert werden, ohne eine interaktive Bearbeitung bereit zu stellen. Mit Hilfe des firmeneigenen Frameworks *transpect* zur Konvertierung und Prüfung von Dokumenten werden XProc- und XSLT-basierte Konverter-Module zur Bearbeitung von XML-Strukturen miteinander verknüpft, um als Verarbeitungskette im Kundenprojekt die gewünschten Ergebnisse zu erreichen[1].

Das Ziel der Idee ist es, ein Produkt zu verwirklichen, das als universelles Projekt allen Kunden zu Verfügung steht. Dazu soll ein neuer Konverter zur Zuweisung von Formatvor-

[1] Vgl: [13], Abschnitt: *Das Open-Source-Framework für die Konvertierung und Prüfung von Dokumenten*, Absatz 3

lagen entwickelt werden, der später in Form eines Kunden-Projektes vorliegen soll. Die Fertigung eines solchen Projektes kann dazu unter Zuhilfenahme weiterer Konverter-Module aus *transpect* erfolgen, die aus einem Repository bezogen werden.[2] Das bietet den Vorteil, dass durch die Einbindung bereits bestehender XML-Werzeuge Entwicklungszeit eingespart werden und die Wartung der Projekte über eine Versionskontrolle erfolgen kann.
Das Projekt trägt den Namen *Stylemapper*, der eine Wortschöpfung aus den englischen Worten *style*[3] und *mapping*[4] ist. In dieser Software soll mittels einer grafischen Nutzeroberfläche eine interaktive Nutzung des Konverters gewährleistet werden, die dem Nutzer bereits aus verschiedenen Text-Editoren vertraut ist. Als geeigneten Lösungsansatz für einen Prototypen wurde als Frontend eine browserbasierte Anwendung unter Verwendung von HTML, CSS und JavaScript und als Backend die Verarbeitungsketten in Form von XProc-Pipelines und XLS-Stylesheets mit einem Zugang über eine HTTP-API in Betracht gezogen. Dabei nimmt das Frontend die Manuskripte oder Satzdaten vom Anwender entgegen und schickt sie zum Backend, worin diese eine HTML-konvertierte Version der jeweiligen Datei zurückliefert. Mit Hilfe des Stylemappers erstellt er Instruktionen unter Zuhilfenahme des HTML-Dokumentes und leitet diese ebenfalls an das Backend. Als Ergebnis erhält der Anwender einen Link zu einer durch die Instruktionen modifizierten Ausgangsdatei und zu den Instruktionen selbst.
Die Eingangsformate für die Stylemapper-Konvertierungen und die Entwicklung der Konverter bei der Firma le-tex liegen bei den XML-basierten Word- und InDesign-Formaten, da diese nach der Aussage des Geschäftsführers und Verantwortlichen im Bereich *Business Development* Gerrit Imsieke am gebräuchlichsten im Verlagsumfeld sind. Andere Dokumenteformate wie beispielsweise OpenDocument werden dagegen nicht berücksichtigt, weil sie von Verlagsautoren kaum verwendet werden.

[2] Vgl.: [14], Abschnitt: *About le-tex transpect*, Absatz 7
[3] style, zu deutsch *Gestaltung*
[4] mapping, zu deutsch *Zuordnung*

1.2 Aufbau der Arbeit

Theoretische Grundlagen

Im folgenden Kapitel 2 auf der Seite 4 werden die Grundlagen der relevanten XML-Formate WordML, IDML, Hub und CSSa beschrieben. Danach wird auf die XML-Verarbeitungstechnologien XProc und XSLT sowie deren Nutzung unter dem Calabash- und dem Saxon-Prozessor über eine Makefile eingegangen. Im Anschluss folgt die Beschreibung der Programmierschnittstelle des Backends und der JavaScript-Mechanismen zur Manipulation des DOMs im Frontend. Abschließend werden die theoretischen Arbeitsabläufe des Front- und des Backends erläutert.

Umsetzung

Die Entwicklungsebenen des Stylemappers sind im Kapitel 3 auf der Seite 35 beschrieben. Dabei werden zu Beginn die Anforderungen für das Front- und Backend herausgearbeitet und die Projektstruktur des Prototypen erläutert. Schlussendlich werden beide Workflows inklusive der Backend-Pipelines und der Frontent-Elemente anhand schematischer Darstellungen und Screenshots erklärt. Als Unterstützung für einen tieferen Einblick in die Programmstruktur dienen dabei auszugsweise Quellcodes aus der Programmstruktur.

Auswertung

Im Abschlusskapitel 4 auf der Seite 73 werden für eine Beispiel-Formatvorlagenzuweisung das Quell-, das Template- und das Ergebnis-Dokument sowie die eingesetzten Regeln dargestellt und erläutert. Im Anschluss folgen Ausführungen über offengelassene bzw. unvollendete Anforderungen des Prototypen. Abschließend steht für diese Arbeit ein Ausblick auf die zukünftige Weiterentwicklung des Stylemappers.

Kapitel 2

Theoretische Grundlagen

2.1 Dokumentformate

2.1.1 WordProcessing Markup Language

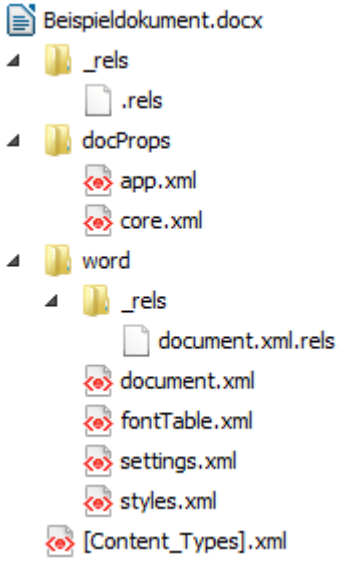
Die XML-Spezifikation zum Word-Dokument trägt den Namen *Word Processing Markup Language* (*WordprocessingML* oder kurz *WordML*). Sie ist Auszeichnungssprache für Textdokumente und Bestandteil der durch Microsoft initiierten Formatspezifikation *Office Open XML (OOXML)* für Büroanwendungen. Unter der Bezeichnung *ISO/IEC-29500-1* ist sie im Verzeichnis der Internationalen Organisation für Normung aufgeführt und ist dort im Kapitel beschrieben. Neben der Spezifikation für Textdokumente beinhaltet diese Norm weitere Bestimmungen für Formate von Büroanwendungen, wie Tabellen (*SpreadsheetML*), Grafiken (*DrawingML*) oder Präsentationsfolien (*PresentationML*)[1].

Das Konzept eines WordML-Dokumentes basiert auf dem Zusammenspiel einzelner XML-Dateien, welche nach verschiedenen Aufgabengebieten modularisiert sind. Sie sind in einem auch als Container bezeichneten ZIP-Archiv verpackt und werden durch ihn mit der Dateierweiterung *.docx* als Word-Dokument ausgezeichnet[2]. In der Abbildung 2.1 ist die Ordnerstruktur anhand eines einfachen *Hallo-Welt!*-Beispieldokumentes dargestellt.

[1] Vgl.:[25], Seite 8 Kapitel *Introduction*

[2] Vgl.: [32], Kapitel *Anatomy of a WordProcessingML File*, Abschnitt *Package Structure*

Abbildung 2.1 Ordnerstruktur eines WordML-Containers



Im Wurzelverzeichnis des Containers befindet sich zunächst die *[Content_Types].xml*. Darin sind sämtliche Datentyp-Beschreibungen der einzelnen Module aufgelistet[3]. Neben diesem XML-Dokument existieren die Ordner *_rels*, *docProps* und *word*. Der *_rels*-Ordner beinhaltet ein gleichnamiges XML-Dokument, welches die Beziehungen der Dateien innerhalb und –nutzungsabhängig– der Dateien außerhalb des Archivs herstellt[4]. Im *docProps*-Ordner befindet sich die *app.xml* und die *core.xml*. In der *app.xml* befinden sich die Basiseigenschaften des Textdokumentes, wie zum Beispiel die Angabe der Seitenanzahl, Wortanzahl, Absätze, die Anwendung, womit das Dokument erstellt wurde und dessen Versionsnummer. Wogegen in der *core.xml* die Metadaten des Dokumentes gespeichert sind[5]. Das Hauptaugenmerk für die späteren XML-Operationen liegt auf dem *word*-Ordner. Er beinhaltet alle Komponenten, die das Textdokument visuell und inhaltlich ausmachen. Das Kernstück bildet dazu die *document.xml*. Darin befinden sich der Textinhalt des Dokumentes, Blockelemente und Inline-Elemente.[6]. Desweiteren liegen in diesem Verzeichnis die *style.xml*, die *fontTable.xml* und die

[3] Vgl.: [32], Kapitel *Anatomy of a WordProcessingML File*, Abschnitt *Content_Types*

[4] Vgl.: [32], Kapitel *Anatomy of a WordProcessingML File*, Abschnitt *Relationships*

[5] Vgl.: [32], Kapitel *Anatomy of a WordProcessingML File*, Abschnitt *Parts Shared by Other OOXML Documents*

[6] Vgl.: [25], S.195, Kapitel *Anatomy of a WordProcessingML File*, Abschnitt 17.3 *Paragraphs and Rich Formatting*

settings.xml. Die Formatvorlagen sind in der *styles.xml* gespeichert, wogegen die verwendeten Schriftsätze in der *fontTable.xml* hinterlegt sind. In der *settings.xml* stehen Einstellungen über das Verhalten des Textdokumentes. Darunter fallen zum Beispiel die Rechtschreibkorrektur, der Schreibschutz und das Speichern von Bearbeitungsverläufen. Neben diesen XML-Modulen können auch weitere generiert werden. Sobald Nummerierungen, Fuß- & Kopfzeilen, Fußnoten, Kommentare oder Glossare erzeugt sind, werden diese separaten XML-Module gespeichert. Eine im Unterordner *_rels* existierende *document.xml.rels*-Datei sorgt, wie bereits im übergeordneten Verzeichnis dafür, dass die Beziehungen unter den Dateien gewährleistet sind. Das bedeutet, dass die *document.xml* dadurch die notwendigen Informationen erhält, welche in den Modulen ausgelagert sind[7]. In den folgenden Ausführungen wird speziell auf die Beziehung zwischen dem Inhalt(*document.xml*) und den Formatvorlagen(*styles.xml*) eingegangen, da sich das später dargestellte Verarbeitungsprinzip auf diese Region fokussiert.

Quellcode 2.1 *w:document* - Inhalt des Textdokumentes

```
<w:document>
    <w:body>
        <w:p>
            <w:pPr>
                <w:pStyle w:val="Normal"/>
            </w:pPr>
            <w:r>
                <w:rPr/>
                <w:t>Hallo Welt!</w:t>
            </w:r>
            <w:r>
                <w:rPr>
                    <w:rStyle w:val="Beispiel"/>
                </w:rPr>
                <w:t>Welt</w:t>
            </w:r>
        </w:p>
        ...
    <w:body>
<w:document>
```

[7] Vgl.: [32], Kapitel *Anatomy of a WordProcessingML File*, Abschnitt *Parts Specific to WordprocessingML Documents*

Im Quellcode 2.1 ist der Aufbauprinzip des Inhaltes beschrieben. Das Kürzel *w* stellt dabei den Namenraum für die WordML-Format[8] dar. Innerhalb des *body*-Elementes, welches den Dokumentkörper des Dokumentes darstellt, wird in zwei Kindelementen unterschieden. Zum einen das *p*-Element, welches ein Absatz repräsentiert und zum anderen das *r*-Element, welches einen darin als Textknoten befindlichen Textabschnitt darstellt. Ein solcher Abschnitt wird auch als *run* bezeichnet und verkörpert ein Inline-Element, welches fernab vom Absatz ebenfalls individuell formatiert werden kann. Im Falle des Beispieldokumentes, geschieht die Formatierung über eine Absatzformatvorlage mit der Bezeichnung *Normal*, welche über das Attribute *val* des *pStyle*-Elementes im Element *pPr* referenziert ist. In diesem Falle steht der Elementname für die Absatzeigenschaften in englischer Fassung für *paragraph properties*. Die Textinformation, wie im Beispiel der Gruß *Hallo Welt!*, steht innerhalb der *runs* jeweils in einem *t*-Element. Für die Formatierung des zweiten *runs* wird eine Zeichenformatvorlage verwendet. Diese ist in den *run*-Eigenschaften unter dem *rStyle*-Element äquivalent zur Absatzformatierung integriert.

Quellcode 2.2 *w:style* - Aufbau einer Absatzformatvorlage

```
<w:style w:type="paragraph" w:styleId="Normal">
    <w:name w:val="Normal"/>
    <w:pPr>
        <w:suppressAutoHyphens w:val="false"/>
    </w:pPr>
    <w:rPr>
        <w:rFonts w:ascii="Liberation Serif" w:hAnsi="
           Liberation Serif"/>
        <w:color w:val="auto"/>
        <w:sz w:val="24"/>
        <w:szCs w:val="24"/>
        <w:lang w:val="de-DE"/>
    </w:rPr>
</w:style>
```

Wie eine Absatzformatvorlage im XML-Modul *styles.xml* aussehen kann, ist im Quellcode 2.2 dargestellt. Im *style*-Element sind über die Attribute *type* und *styleId*, der Vorlagentyp, sowie die ID der Vorlage hinterlegt. Als Kindelemente besitzt es ein *name*-, ein *pPr*- und ein *rPr*-Element. Zuletzt genanntes steht –ähnlich der *pPr*-Bezeichnung– für die Zeichen-

[8] WordML-Namensraum: *xmlns:w=ïttp://schemas.openxmlformats.org/wordprocessingml/2006/main¨*

eigenschaften *run properties*. Innerhalb des *pPr*-Elements befinden sich nur absatzspezifische Eigenschaften, wie im oberen Beispiel angegeben. Hier findet man beispielsweise ein *suppressAutoHyphen*-Element mit dem Attributswert *false*, was bedeutet, dass die Silbentrennung deaktiviert sein soll. In den Zeicheneigenschaften befinden sich hingegen die Eigenschaften für die Schriftart(*rFonts*), die Schriftgröße(*sz*), die Schriftgröße von komplexen Schriftarten(*szCs*) und die Sprachen-Bezeichnung(*lang*). Die einheitenlose Größe für die Schrift ist an dieser Stelle in sogenannten Half-Points bzw. Halbpunkten angegeben. Aus diesem Beispiel ist also zu entnehmen, dass es sich bei der Schriftgröße dieser Absatzformatvorlage um *12pt* handelt. Im Abschnitt 2.4.2 auf Seite 30 ist –aufbauend auf dieser WordML-Struktur– das Formatvorlagenmapping erklärt.

2.1.2 InDesign Markup Languange

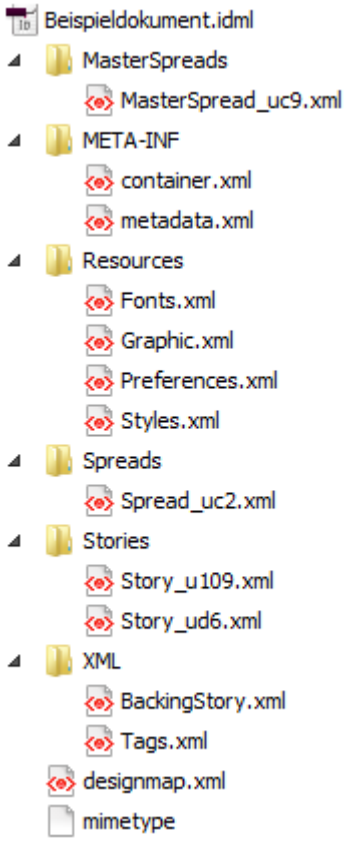
Das Textsatz- und Layout-Programm *InDesign* der *Adobe Systems* bietet ab der CS4-Version die Möglichkeit seine Satzdaten rein XML-basiert zu repräsentieren. Dazu wurde eine eigene Auszeichnungssprache(IDML[9]) spezifiziert, welche die Satzdatei nach dem *Document Object Model*-Prinzip beschreibt.[10]

Ähnlich wie *docx*-Container befinden sich funktional voneinander getrennte XML-Dateien in einem ZIP-Archiv. Der Zugriff und der Aufbau der Struktur ist wiederum nach einer eigenen Container-Technologie, dem *Universal Format Container(UFC*, festgelegt[11]. Bezüglich der Semantik wurde darauf Wert gelegt, dass die Elemente und Attribute im Falle von Wortgruppen über Binnenversalien verfügen bzw. in CamelCase-Schreibweise ausformuliert sind. Dadurch soll gewährleistet werden, dass sich die Architektur der XML-Strukturen einfacher lesen und verstehen lassen.[12] In der folgenden Abbildung 2.2 ist eine Übersicht über den Aufbau eines IDML-Containers dargestellt.

Der IDML-Container besteht aus einer *MIMETYPE*-Datei, einer *designmap.xml* und den Ordnern *Ressources*, *Spreads*, *Stories*, *MasterSpreads*, *XML*, und *META-INF*. Eine zunächst

[9] IDML – InDesign Markup Language
[10] Vgl.: [8], S.10, Kapitel 2 *Introduction*
[11] Vgl.: [8], S.10, Kapitel 8 *IDML Document Structure*, Abschnitt 8.1 *IDML Package File Organization*
[12] Vgl.: [8], S.11, Kapitel 4 *INX, IDML, and InDesign Scripting*

Abbildung 2.2 Ordnerstruktur eines IDML-Containers



irrelevante Position für die Formatvorlagenzuweisung haben die beiden letzten Ordner. In *XML* befinden sich die *Backingstories.xml* und *Tags.xml*. Die jeweils darin existierenden Dateien haben die Aufgaben, Inhalte und erstellte Tags, die noch keinem Layout-Element zugewiesen sind, zu speichern[13]. Der *META-INF*-Ordner beinhaltet zum einen die *container.xml*, welche für den *UFC*-Standard notwendig ist. Zum anderen befindet sich in diesem Ordner die *metadata.xml* zur Speicherung von Metadaten, die in Form des XMP[14]-Standards vorliegt[15]. Die im Wurzelverzeichnis liegende *mimetype*[16]–Datei beinhaltet wiederum die

[13] Vgl.: [8], S.20, Kapitel 8 *IDML Document Structure*, Abschnitt 8.1 *IDML Package File Organization*, Unterabschnitt 8.1.7 *XML Folder*

[14] XMP – Extensible Metadata Platform

[15] Vgl.: [8], S.20, Kapitel 8 *IDML Document Structure*, Abschnitt 8.1 *IDML Package File Organization*, Unterabschnitt 8.1.8 *XML Folder*

[16] MIME – Multipurpose Internet Mail Extension

Typdefinition zum IDML–Packet[17]. Die *designmap.xml* stellt die Beziehungen der funktional separierten XML-Module (*Ressources*), wie Formatvorlagen (*Styles.xml*, Schriftarten (*Fonts.xml*, Grafiken (*Graphics.xml* oder Dokumenteinstellungen (*Preferences.xml*), zum Textinhalt herstellt. Des Weiteren regelt es die Reihenfolge der Dokumentseiten[18]. Diese werden als *Spreads* deklariert und befinden sich –wie die Seitenvorlagen (*Master Spreads*)– in dem gleichnamigen Ordner. Bei der Generierung der Seiten und Seitenvorlagen wird zur Namensgebung eine Kombination aus der Art der Seite und einer eindeutigen Bezeichnung bestehend aus Buchstaben und Ziffern verwendet. Im folgenden Abschnitt wird der grundlegende Zusammenhang zwischen Seiten, Inhalt und Formatvorlagen erläutert. Dazu wird anhand eines Beispieldokumentes die Beziehung zwischen *designmap.xml*, den jeweiligen *Spreads* und den Textsegmenten (*Stories*) erläutert. Die Layout-Konzeption in InDesign bietet beispielsweise gegenüber einem Manuskript, wie das eines Word-Dokumentes, komplexere Möglichkeiten, um Text und Layout intelligenter zu verknüpfen. Das bedeutet, dass man statt ausschließlich über Blockelementdeklarationen zu formatieren, Textboxen erstellen und diese miteinander verbinden kann. Dadurch ergeben sie mehr Möglichkeiten, Text flexibel und seitenübergreifend zu setzen. Das Verständnis gegenüber dem Dreiergespann, bestehend aus Seiten, Vorlagen und Textsegmenten ist essentiell, um eine Vorlagenzuweisung durchführen zu können.

Zum besseren Verständnis werden Element- und Attributdarstellungen in den Codebeispielen vorerst vereinfacht. Im folgenden Quellcode 2.3 ist das wesentliche Gerüst des Beispieldokumentes aufgeführt. Zum einen beinhaltet es die vorher aufgezählten Ressourcen und zum anderen die Muster- und Dokumentseiten.

[17] Vgl.: [8], S.18, Kapitel 8 *IDML Document Structure*, Abschnitt 8.1 *IDML Package File Organization*, Unterabschnitt 8.1.1 *MIMETYPE*

[18] Vgl.: [8], S.19, Kapitel 8 *IDML Document Structure*, Abschnitt 8.1 *IDML Package File Organization*, Unterabschnitt 8.1.2 *designmap.xml*

Quellcode 2.3 *designmap.xml*

```
<Document>
    <idPkg:Graphic src="Resources/Graphic.xml"/>
     <idPkg:Fonts src="Resources/Fonts.xml"/>
     <idPkg:Styles src="Resources/Styles.xml"/>
     <idPkg:Preferences src="Resources/Preferences.xml"/>
     <idPkg:MasterSpread src="MasterSpreads/MasterSpread\_uc9.
        xml"/>
     <idPkg:Spread src="Spreads/Spread\_uc2.xml"/>
</Document>
```

Weiter geht es mit der Dokumentseite. Diese ist im Quellcode 2.4 aufgeführt und besitzt im Wurzelelement *Story* ein gleichnamiges Kindelement, welches mit dem Attribut *Self* die ID *uc2* in sich birgt.

Quellcode 2.4 *Spread_uc2.xml*

```
<idPkg:Spread>
     <Spread Self="uc2">
          <TextFrame Self="ue8" ParentStory="ud6"
             PreviousTextFrame="n" NextTextFrame="u100">
     </Spread>
</idPkg:Spread>
```

Das darin stehende Kindelement *TextFrame* repräsentiert einen Textrahmen, der über das Attribut *ParentStory* mit einem Textsegment verknüpt ist. Die Reihenfolge der Textrahmen wird über eine Vorgänger-Nachfolger-Beziehung (*PreviousTextFrame, NextTextFrame*) realisiert.

Quellcode 2.5 *Story_ud6.xml*

```
<idPkg:Story>
     <Story Self="ud6" StoryTitle="$ID/">
          <ParagraphStyleRange AppliedParagraphStyle="
             ParagraphStyle/Headline 2">
               <CharacterStyleRange AppliedCharacterStyle="
                  CharacterStyle/$ID/[No character style]">
                    <Content>Hallo Welt!</Content>
               </CharacterStyleRange>
          </ParagraphStyleRange>
     </Story>
</idPkg:Story>
```

Das Textsegment ist im oberen Quellcode 2.5 dargestellt. Es ist in den ersten beiden Ebenen wie eine Dokumentseite aufgebaut. Das Element *ParagraphStyle* stellt den Container

für einen Absatz dar, in dem für sämtliche Inhalte, sofern sie nicht von Zeichenformaten überschrieben werden, eine Absatzformatvorlage gilt. Über das Attribut *AppliedCharacterStyle* wird die Absatzformatvorlage in der *Styles.xml* referenziert. Der Textinhalt befindet sich wiederum in einem *CharacterStyleRange*-Element, welches als Zeichenformat eines äquivalenten Containers für eine Zeichenformatvorlage steht. Es besitzt aber im Beispielfall keine zugewiesene Vorlage, weswegen der Wert *[no character style]* im Referenzattribut hinterlegt ist. Zu guter Letzt folgt im Quellcode 2.6 der Aufbau einer Absatzformatvorlage.

Quellcode 2.6 *Styles.xml*

```
<idPkg:Styles>
     <RootParagraphStyleGroup Self="u76">
          <ParagraphStyle Self="ParagraphStyle/Headline 2" Name=
             "Headline 2" NextStyle="ParagraphStyle/Headline 2"
           FillColor="Color/Black" FontStyle="Italic" PointSize=
              "14">
               <Properties>
                    <BasedOn type="object">ParagraphStyle/
                       Headline 1</BasedOn>
                    <AppliedFont type="string"> Arial </
                       AppliedFont>
               </Properties>
          </ParagraphStyle>
     </RootParagraphStyleGroup>
</idPkg:Styles>
```

Innerhalb des Wurzelelements befinden sich die Absatzformatvorlagen wiederum im Kindelement *RootParagraphStyleGroup* jeweils als *ParagraphStyle*-Element. Als *Self*-ID besitzt es die Referenzierung bestehend aus der Vorlagenart (*Paragraph*) und dem Namen (*Headline 2*, welche im Quellcode 2.5 auf Seite 11 im *AppliedParagraph*-Attribut zu finden ist. Wenn die Eigenschaften einfache Schlüssel-Wert-Paare sind, kommen sie in Form von Attributen im Hauptelement vor. Sind sie jedoch komplexer aufgebaut, beinhalten sie zusätzliche Informationen, wie z.B. Datentypangaben oder werden durch Kindelemente im *Property*-Element dargestellt. Im oberen Beispiel sind die Vorlagenvererbung(<*BasedOn*>) und die Schriftartenfestlegung(<*AppliedFonts*>) aufgeführt.

2.1.3 le-tex Hub-Format

Das le–tex Hub–Format basiert auf der DocBook Version 5.1 und ist ein Hauptbestandteil bei der Konvertierung von XML–Formaten. Wie der Name *Hub*[19] bereits vermuten lässt, wird dieses XML–Format als Übergangs- bzw. Ausgangsformat für einen Konvertierungsprozess bzw. einem komplexeren Workflow verwendet. Dabei besteht der Hauptunterschied zum herkömmlichen DocBook-Aufbau in der Baumstrukturtiefe. Während das ursprüngliche Format eine hierarchische Dokumentstruktur besitzt, die in Kapiteln, Abschnitten und weiteren Subkategorien verschachtelt ist, wird im Hub–Format lediglich mit einfachen Blockelementen, wie Absätzen, Listen oder zum Beispiel Abbildungen gearbeitet.[20]
Im folgenden Quellcode 2.7 ist die vereinfachte Struktur dieses Formates dargestellt, das aus dem Beispieldokument des Abschnittes 2.1 auf Seite 6 erzeugt wurde.

Quellcode 2.7 Hub-Übersicht

```
<hub>
   <info>
      <keywordset role="hub">
           <keyword role="source-type">docx</keyword>
      </keywordset>
      <css:rules>
           <css:rule/>
      </css:rules>
   </info>
   <para>Hallo<phrase>Welt</phrase>!</para>
</hub>
```

Darin ist neben dem Blockelement eines Absatzes (*para*) das Element *info* mit den Kindelementen *keywordset* und *css:rules* enthalten. Die Keywordsets sind Bestandteil für die Konvertierung des Hub-Dokumentes. Das heißt, dass über das *role*-Attribut der Name des Keywords definiert und über einen Textknoten Informationen über den Verlauf zur Hub-Konvertierung mitführen, welche evtl. für weitere Prozesse benötigt werden könnten. Über das Keyword *source-type* wird beispielsweise dem Konverter mitgegeben, dass es sich bei dem Quelldokument um ein *docx* handelt.

[19] Hub, zu deutsch *Knotenpunkt*
[20] Vgl.: [26], Dokumentation *le-tex Hub Format*

Im unteren Quellcode 2.8 ist das *rules*-Element, dessen Kindelemente Absatzvorlagen repräsentieren.

Quellcode 2.8 *Absatzformatvorlagen im Hub-Format*

```
<css:rules>
   <css:rule layout-type="para"
             css:font-family="Liberation Serif"
             css:color="black"
             css:font-size="12pt"
             xml:lang="de"
             name="Normal"/>
</css:rules>
```

Das Attribut *layout-type* legt den Vorlagentyp fest. Die Eigenschaften werden in Form von CSS-Attributen dargestellt. Der Absatz des Hub-Dokumentes ist im unteren Quellcode 2.9 zu sehen.

Quellcode 2.9 Absatz im Hub-Format

```
<para srcpath="word/document.xml?xpath=/w:document[1]/w:body[1]/
   w:p[1]" role="Normal"> Hallo
         <phrase role="" srcpath="word/document.xml?xpath=/
            w:document[1]/w:body[1]/w:p[1]/w:r[2]"> Welt
         </phrase>
</para>
```

Innerhalb des Absatzes, werden die separat formatierten Inline-Elemente in *phrase*-Elemente gesetzt. Das *srcpath*-Attribut gibt dazu jedem Block- und Inline-Element die genaue Position der Elemente im Quelldokument mit. Die hierarchische Struktur wird über der Kombination aus einer relativen Pfadangabe zur XML-Datei des Inhaltes und einer XPath-Selektion umgesetzt.

2.1.4 Cascading Style Sheet properties as attributes (CSSa)

Beim Hub-Format wird für die Formatierung die CSS-Syntax verwendet. Dazu werden in Formatvorlagen im *rule*-Element bzw. Ad-Hoc-Formatierungen am Block- oder Inline-Element CSS3-Eigenschaften in Form von Attributen gesetzt. Deshalb spricht man auch von *Cascading Style Sheet properties as attributes(CSSa)*[21]. Dieses standardisierte Konzept dient

[21] Vgl.: [7], S.3, Kapitel*CSSa Specification*

als Schnittstelle für die Formatierung anderer Dokumentformate in weiteren Konvertierungen. Im Quellcode 2.8 auf Seite 14 ist ein Beispiel für eine Absatzformatvorlage dargestellt, welche aus dem *docx*-Beispieldokument erzeugt wurde und aus dessen Eigenschaften CSS-Attribute generiert wurden.

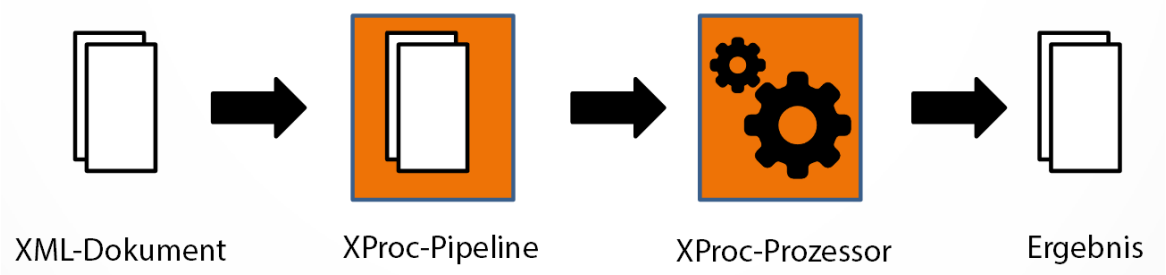
2.2 XML-Technologien

2.2.1 XProc–Pipeline Language

Grundlagen

XProc ist eine Technologie zur Verarbeitung von XML-Dokumenten in einer Prozesskette. Sie wurde von Norman Walsh entworfen, gehört seit Mai 2010 zum W3C-Standard und wird zusammen mit Alex Milowski und Henry Thompson weiterentwickelt. Das Prinzip des XProc-Workflows ist in der Abbildung 2.3 skizziert.

Abbildung 2.3 Das XProc-Prinzip



In einer sogenannten Pipeline werden einzelne Operationen aneinandergereiht. Dabei ist es möglich, ein oder mehrere XML-Dokumente als Eingangselement festzulegen, ohne dabei zwangsläufig am Ausgang der Pipeline ein Ergebnis zu erhalten, dass in gleicher Relation steht.[22]. Das XML-Dokument wird an die Pipeline übergeben und von einem XProc-Prozessor verarbeitet. Das Endresultat erscheint am Ausgang der Pipeline, während

[22] Vgl.: [21], Kapitel *Abstract*

ebenfalls Zwischenresultate im Prozess ausgegeben werden können[23]. Für den Zugriff auf XML-Inhalte wird als Auszeichnungssprache XPath verwendet[24].

Die Operationen, die nacheinander ausgeführt werden, bezeichnet man als Schritte. Sie verfügen über Funktionen, um XML-Struktur verändern zu können. Die Verbindung der Pipeline mit einem Schritt bzw. der Schritte untereinander erfolgt über sogenannte Ports. Jede Pipeline und jeder Schritt verfügt über mindestens einen Eingangs- und einen Ausgangsport, sowie die Möglichkeit, weitere Parameter anzugeben. Dabei setzt die Konvention voraus, dass jeder Ausgangsport entweder zu einem nächsten Eingangsport oder mindestens einem Platzhalter zugewiesen ist. Ein ausgeglichenes Verhältnis von Ein- und Ausgangsport ist nicht notwendig. Aus dem Regelwerk der Port-Konventionen leiten sich 3 unterschiedliche Arten von Schritten ab. Man unterscheidet prinzipiell zwischen einer einfachen Operation(Atomarer Schritt), mehreren Operationen (*zusammengesetzter Schritt*) und konditionale Operationen (*Mehrfach-Schritt*)[25]. Bei einem atomaren Schritt handelt es sich um die Anwendung einer Funktion auf die XML-Struktur[26]. Während in einem zusammengesetzten Schritt eine Subpipepline existiert[27]. Darin befinden sich wiederum ein oder mehrere Unterschritte, womit mehrere Funktionen nacheinander angewendet werden können. Der Mehrfach-Schritt ist ebenso im Stande mehrere Schritte in sich zu beherbergen. Der Unterschied zum zusammengesetzten Schritt besteht dabei jedoch in der Abarbeitung. Über eine Fallentscheidung bzw. einen Mehrfachzweig wird bestimmt, welche der Schritte bzw. Schrittfolgen abgearbeitet werden soll[28]. Im unteren Quellcode 2.10 ist das Beispiel einer Pipeline dargestellt.

Quellcode 2.10 Beispiel einer XProc-Pipeline

```
<p:declare-step>
    <p:input port="source">
     <p:document href="item_collection.xml"/>
    </p:input>
    <p:output port="result"/>
    <p:for-each>
        <p:iteration-source select="//item"/>
```

[23] Vgl.: [17], Kapitel 2 *Prozessoren*
[24] Vgl.: [21], Kapitel 2 *Pipeline Concepts*, Abschnitt 2.6 *XPaths in XProc*, Absatz 1
[25] Vgl.: [21], Kapitel 2 *Pipeline Concepts*, Abschnitt 2.1 *Steps*
[26] Vgl.: [17], Kapitel 4 *Steps*, Abschnitt *Atomic Steps*
[27] Vgl.: [21], Kapitel 2 *Pipeline Concepts*, Abschnitt 2.2 *Inputs and Outputs*, Absatz 9
[28] Vgl.: [17], Kapitel 4 *Steps*, Abschnitt *Multi-Container Steps*

```
        <p:rename match="//item/price" new-name="cost"/>
    </p:for-each>
    <p:wrap-sequence wrapper="item_collection"/>
</p:declare-step>
```

Das Wurzelelement *declare-step* beinhaltet zu Beginn Kindelemente *input* und *output*, welche über das Attribut *port* die Ein- und Ausgänge deklarieren. Am Eingang ist das Beispieldokument über das *href*-Attribut des Elementes *document* referenziert. In diesem Beispiel geschieht die Einbindung des XML-Dokumentes statisch. Bei der Verwendung des Stylemappers geschieht es jedoch dynamisch. Das heißt, dass die notwendigen Eingänge in der Pipeline deklariert werden und ein Dokument erst über den Aufruf des Prozessors zur Verarbeitung eingespeist wird. Im Anschluss an die Port-Deklarationen erfolgt der Schritt *for-each*, der Elemente aus dem Dokument iterativ verarbeitet. Über das Kindelement*iteration source* wird das Element *item* ausgewählt und dessen Kindelement wiederum in einem weiteren Schritt (*rename*) umbenannt. Am Ausgang des Schrittes finden sich die *item*-Elemente sequentiell angeordnet. Das bedeutet, dass das Wurzelelement über diesen Schritt eliminiert wurde. Deshalb wird im darauffolgenden Schritt *wrap-sequence* ein neues Element erzeugt, dessen Start- und End-Tag um die Sequenz gelegt wird. Dadurch soll gewährleistet werden, dass die ursprüngliche Struktur am Ausgang der Pipeline, trotz des bestimmten operativen Eingriffs, erhalten bleibt. Über eine Import-Funktion ist es möglich externe Pipelines zu integrieren und entweder komplett als einzigen zusammengesetzten Schritt oder auch die internen Schritte separat zu verwenden.

Calabash-Prozessor

Calabash ist ein Prozessor zur Verarbeitung der XProc-Dokumente des Stylemappers, der ebenfalls von Norman Walsh entwickelt wurde. Er basiert auf *Java 1.7* und nutzt für XSL-Transformationen den integrierten Prozessor *Saxon 9he*[29], der im Abschnitt 2.2.2 auf der Seite 24 beleuchtet wird.

Quellcode 2.11 Aufruf des Calabash-Prozessors

```
java com.xmlcalabash.drivers.Main pipeline.xpl
```

[29] Vgl.: [35], Abschnitt *Prerequisites*

Im oberen Quellcode 2.11 auf Seite 17 ist ein einfacher Aufruf des Calabash zur Verarbeitung einer Pipeline dargestellt. Über den Befehl *java* wird der Prozessor über die Klasse *com.xmlcalabash.dri- vers.Main* aufgerufen. Nachfolgend ist eine relative Pfadangabe zur Pipeline aufgeführt, worin ein Dokument statisch referenziert und das Resultat lediglich in der Console ausgegeben wird. Um die Ein- und Ausgänge flexibel zu nutzen bzw. die Ergebnisse zu speichern oder Dokumente über den Aufruf einzubinden, bedarf es weitere Parameter. Diese finden im Falle des Stylemappers in einem komplexeren Konstrukt zum Aufruf von Calabash in einer sogenannten Makefile ihren Platz.

Makefile

Über diese Datei wird der gesamte Programmaufruf mit allen Anwendungsbedingungen und datei- sowie verzeichnisregulierenden Einstellungen formuliert. Sie gehört zu dem Werkzeug *make*, mit dem Programme generiert oder softwareseitige Arbeitsabläufe automatisiert werden können[30]. Diese Technologie macht sich der transpect-Server zu nutze, um über die Makefile den Datenverkehr der Konverter-Projekte, einschließlich des Stylemapper-Projektes zu steuern.

Quellcode 2.12 Beispiel einer Makefile

```
#Makefile
OUT_DIR = /output/$(notdir $(IN_FILE))
CALABASH = /programs/calabash.sh

preprocess: -mkdir $(OUT_DIR)

stylemapper: $(CALABASH) \
                /pipelines/pipeline.xpl \
                out_dir_uri=$(OUT_DIR)

conversion: preprocess stylemapper

#Aufruf der Makefile
make conversion IN_FILE=Beispieldokument.xml
```

Im Quellcode 2.12 ist eine stark vereinfachte Version der Makefile des Stylemappers aufgeführt. Mit deren Hilfe sollen alle Parameter in einem Aufruf zusammengefasst und für einzel-

[30] Vgl.: [33], S.1, Kapitel 1 *Overview of make*, Absatz 2

ne Teilaufrufe bereitgestellt werden. Über so genannte *Targets*[31] werden Regeln gegliedert, die prinzipiell als Teilaufrufe zu betrachten sind und über den Befehl *make* ansteuerbar gemacht werden[32]. In der Beispiel-Makefile sind drei Regeln dargestellt. Die *preprocess*-Regel beinhaltet den Aufruf zur Erzeugung eines Ausgabeverzeichnisses. Das Zweite (*stylemapper*) beinhaltet einen Calabash-Aufruf. Als Zusammenführung dieser beiden Aufrufe dient die dritte Regel (*conversion*). Über die Angabe der beiden Regelnamen werden diese abgearbeitet und als Voraussetzung für die eigenen Regelinhalte verwendet. Im Vorfeld finden in den ersten beiden Zeilen Variablendeklarationen für 2 Pfadangaben statt. Dabei wird in der ersten Zeile in der *OUT_DIR*-Variable der Pfad für die Haupt- und Teilresultate der Pipeline gespeichert. Der Parameter *IN_FILE*, der beim Aufruf der Makefile angegeben wird, liefert den Pfad des Quelldokumentes. Syntaktisch wird er mit vorangestelltem $ und in Klammern ausgedrückt. Ebenso verhält es sich bei Funktionsaufrufen. Der Identifikator steht zu Beginn des Ausdrucks und weitere Parameter werden ihm nachgestellt. Im Falle der Pfadangabe wird über die Funktion *notdir* der Quellpfad, der über *IN_FILE* mitgegeben wird, bis zum letzten Ausdruck abgeschnitten und isoliert zurückgegeben. Am Ende des Quellcodes 2.12 ist der Aufruf der *conversion*-Regel dargestellt. Im zweiten Argument wird der gewünschte Targetname angegeben, dessen Regel verarbeitet werden soll. Anschließend wird über das dritte Argument *IN_FILE* das Quelldokument referenziert.

2.2.2 XSLT–Extensible Stylesheet Language Transformation

Grundlagen

XSLT ist als Version 2.0 eine seit 2007 vom W3C empfohlene Technologie für XML-Strukturen. Als Auszeichnungssprache dient XSL[33], die seit 2006 vom W3C ebenfalls empfohlen wird. Die Einsatzfähigkeit beläuft sich dabei auf die Verarbeitung sämtlicher, wohlgeformter XML-Strukturen und ist wie im komplexeren Falle des Stylemappers dzau im

[31] *Target*, zu deutsch *Ziel*
[32] Vgl.:[33], Kapitel 2: *An Introduction to Makefiles*, Abschnitt 2.1:*What a Rule looks like*
[33] XSL - Extensible Stylesheet Language

Stande, XML-Formate zu konvertieren [34]. In den folgenden Ausführungen werden grundsätzliche Aufgaben und Funktionen von XSLT dargestellt.

Das Konzept beruht auf der Verwendung von XSL-Dokumenten, sogenannten *Stylesheets*, die mit Mustervorlagen -*Template-Regeln* genannt- XML-Strukturen verarbeiten. Eine *Template-Regel* besitzt als Selektor das Attribut *match*, das über eine XPath-Adressierung eine Sequenz von XML-Knoten oder atomaren Werten zurückgibt und für das Filtern der jeweiligen XML-Baumstrukturen, kurz Bäume, notwendig ist. Um die Transformationsergebnisse explizit zu beschreiben, empfiehlt es sich, die Bäume über vier verschiedene Stadien zu definieren. Die Unterteilung erfolgt in Quellbaum (*source tree*), temporärer Baum (*temporary tree*), resultierender Baum (*result tree*), finaler resultierender Baum (*final result tree*). Der Quellbaum ist der Baum im Quelldokument, während der temporäre Baum als Zwischenergebnis innerhalb einer Template-Regel zu verstehen ist. Der resultierende Baum ist wiederum das Ergebnis am Ende des Template-Regel-Durchlaufes. Zuletzt wird der final resultierende Baum als Gesamtergebnis des Stylesheets gesehen.[35]

Die Resultate der Template-Regeln können anhand dieses Befehls-Repertoires neu generierte oder veränderte Baumstrukturen sein, die über Reproduktionsprozesse entstehen (siehe Quellcode 2.13 auf Seite 22). Äquivalent zu XProc, wird für den Zugriff auf XML-Strukturen XPath verwendet[36].

Innerhalb der Template-Regeln können Parameterschnittstellen, XSLT-eigene Befehle (*instructions*), erweiterte Befehle (*extension instructions*) und Elemente, die unbehandelt ins Ergebnis geführt werden, stehen[37]. XSLT-eigene Befehle sind über Elemente repräsentiert, die den *xsl*-Namensraum besitzen und Operationen ausführen. Sie können in sechs Kategorien unterteilt werden. Dazu gehören als erstes Befehl-Elemente zur Erzeugung von Knoten. Dazu zählen beispielsweise *value-of* für Textknoten, *attribute* für Attributknoten und *element* für Elementknoten. Zur zweiten Kategorie zählt der Befehl *sequence*, womit über einen XPath-Ausdruck eine Sequenz von Knoten oder Werten erzeugt werden kann. Für iterati-

[34] Vgl.:[9], Kapitel 1: *Introduction*, Abschnitt 1.1: *What is XSLT?*, Absatz 4
[35] Vgl.: [9], Kapitel 2: *Concepts*, Abschnitt 2.1: *Terminology*
[36] Vgl.: [9], Kapitel 2: *Concepts*, Abschnitt 2.2: *Notation*, Unterabschnitt: *Example: Syntax Notation*
[37] Vgl.: [9], Kapitel 2: *Concepts*, Abschnitt 2.6: *Executing a Transformation*, Absatz 1

ve und konditionale Verarbeitung der dritten Kategorie gibt es zum Beispiel *choose* oder *for-each*. Über *choose* ist es möglich Mehrfachbedingungen zu erstellen. Mit *for-each* ist wiederum eine einfache Iteration durch jedes im Fokus stehende Kindelement durchführbar. Dagegen können über die Befehle *variable* und *param* Variablen deklariert werden. Die fünfte Kategorie steht für Befehle, die zum Aufruf weiterer Template-Regeln fungieren. Stellvertretend werden dazu mittels *apply-templates* über eine XPath-Selektion alle passenden Template-Regeln für die Knoten innerhalb der gerade zu verarbeitenden Baumstruktur aufgerufen. Im Unterschied dazu sind auch Einzelaufrufe über Namensreferenzen durch den Befehl *call-template* möglich. Zur letzten Kategorie zählen alle weiteren Befehle, die individuelle Funktionen mit sich bringen. Beispielsweise Datentypkonvertierungen (*number*) oder Stringüberprüfungen (*analyze-string*)[38].

Erweiterte Befehle sind Elemente die über Namensräume verfügen, die nicht zum XSL-Namensraum gehören und die Möglichkeit bieten, den Funktionsumfang von XSLT zu erweitern[39]. Des Weiteren bietet XSLT die Möglichkeit, Template-Regeln aus externen Stylesheets zu importieren. Dazu muss zu Beginn des Stylesheets über den Import-Befehl das Stylesheet referenziert werden.

Die letztgenannten relevanten Werkzeuge in dieser Ausführung stellen Funktionen dar. XSLT und XPath verfügen über eine Reihe von Kernfunktionen, die innerhalb von Attributwerten aufrufbar sind. Weiterhin bietet XSLT die Möglichkeit, eigene Funktionen über den Befehl *function* zu erstellen. Dabei können Parameter und Variablen definiert und XSLT-Befehle für den Algorithmus verwendet werden. Im späteren Verlauf dieses Kapitels werden Funktionen in den Quellcodes in Erscheinung treten.

Anhand eines einfachen Beispiels wird in den folgenden Ausführungen der Aufbau eines XSL-Stylesheets erklärt. Dabei handelt es sich um ein Szenario, das die Erstellung einer eigenen Funktion, sowie deren Anwendung und die einer Kernfunktion in sich birgt. Man nehme an, dass das Eingangsdokument ein Wurzelelement *items* besitzt und aus ein oder mehreren *item*-Elementen besteht, welche das Kindelement *value* in sich bergen. Dieses

[38] Vgl.: [9], Kapitel 2: *Concepts*, Abschnitt 2.6: *Executing a Transformation*, Absatz 10
[39] Vgl.: [9], Kapitel 18: *Extensibility and Fallback*, Absatz 2

beinhaltet wiederum als Kindelement einen Textknoten mit einem beliebigen Zahlenwert. Im Quellcode 2.13 beginnt die Beschreibung zum Inhalt des Stylesheets.

Quellcode 2.13 Copy-Template eines Beispiels-Stylesheets

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                    xmlns:func="www.functions.org/functions">
    <xsl:template match="*|@*">
        <xsl:copy>
            <xsl:apply-templates select="node()|@*"></xsl:apply-
               templates>
        </xsl:copy>
    </xsl:template>
   ...
</xsl:stylesheet>
```

Im Wurzelelement befindet sich neben dem eigenen XSL-Namensraum ein weiterer für die eindeutige Zuordnung der erstellten Funktion, die im weiteren Verlauf dargestellt wird. Über das Element *template* wird eine Template-Regel erstellt, die durch ihren Inhalt zur Reproduktion der Quellbaumstruktur dient.

Die Wildcard-Selektoren für Element(*) und Attributknoten(@*) im *match*-Attribut sorgen dafür, dass alle Strukturen angesprochen werden, soweit keine höher priorisierte Template-Regel vorhanden ist, die diese ablöst. Innerhalb eines Stylesheets kann es passieren, dass zwei oder mehrere Template-Regeln auf gemeinsame Knoten passen. Dadurch kommt es zu einem Konflikt, der dazu führt, dass der Prozessor nicht genau weiß, welche Template-Regel angewandt werden soll. Die Priorisierung ist über 2 Bestimmungen realisiert. Zuerst wird die *Import Präzedenz* geprüft[40]. Diese Festlegung besagt, dass Template-Regeln innerhalb eines Stylesheets gegenüber denen importierter Stylesheets Vorrang haben. Falls mehrere Importe existieren, sind die Template-Regeln der jeweils zuletzt importierten Stylesheets vorrangig[41]. Danach werden die Prioritäten der einzelnen Tempalte-Regeln miteinander verglichen. Sie befinden sich im Attribut *priority* des *template*-Elementes. Angewendet wird die Template-Regel mit höchstem Prioritätswert.

[40] Vgl.:[9], Kapitel 6: *Template Rules*, Abschnitt: 6.4: *Conflict Resolution for Template Rules*, Absatz 1

[41] Vgl.:[9], Kapitel 3: *Stylesheet Structure*, Abschnitt: 3.10: *Combining Stylesheet Modules*, Unterabschnitt 3.10.3: *Stylesheet import*

Über den Befehl *copy* wird gewährleistet, dass das Wurzelelement am resultierenden Baum bestehen bleibt bzw. in das Ergebnis kopiert wird. Mit Hilfe des *select*-Attributes im *apply-templates*-Element werden ebenfalls über Wildcards *node()* und @ sämtliche Kindelemente und Attribute angesprochen, deren Template-Regeln an dieser Stelle aufgerufen werden sollen.

Quellcode 2.14 Deklaration und Aufruf von Funktionen

```
<xsl:function name="func:addition">
    <xsl:param name="a"></xsl:param>
    <xsl:param name="b"></xsl:param>
    <xsl:value-of select="$a + $b"></xsl:value-of>
</xsl:function>

<xsl:template match="item">
  <xsl:call-template name="add-one">
    <xsl:with-param name="value" select="number(item/value/text
       ())"></xsl:with-param>
  </xsl:call-template>
</xsl:template>

<xsl:template name="add-one">
  <xsl:param name="value"></xsl:param>
  <xsl:copy>
    <xsl:attribute name="value">
      <xsl:value-of select="func:addition($value, 1)"></
         xsl:value-of>
    </xsl:attribute>
  </xsl:copy>
</xsl:template>
```

Im Quellcode 2.14 auf Seite 23 ist der Inhalt des Stylesheets weitergeführt. Über das *function*-Element wird der Befehl zur Funktionsdeklaration initiiert. Die Funktion soll dabei eine einfache Addition aus zwei Variablen durchführen. Per *param*-Element wird die Variable als Parameter deklariert und das Ergebnis über das Element *value-of* zurückgeliefert. Die eigentliche Operation findet hierbei im Ausgabeattribut *select* wieder, welche die Variablen über die Dollarschreibweise referenziert.

Im Anschluss daran folgt eine Template-Regel für *item*-Elemente. Sie beinhaltet über das Element *call-template* den Aufruf-Befehl einer Template-Regel mit dem Namen *add-one*. Mit ihr soll als Ergebnis das Kindelement *value* des *item*-Elements eliminiert und als gleichnamiges

Attribut angefügt werden. Dazu soll über die vorher deklarierte Funktion der Textknoten als Zahlenwert mit der Zahl 1 addiert und als Attributwert beigefügt werden.
Dem Aufruf wird für die Referenzierung auf den *item*-Knoten in der Template-Regel *add-one* über das Hilfselement *with-param* ein Parameter mitgegeben. Dieses besitzt den Textknoten des *value*-Elements, der durch eine XPath-Adressierung initialisiert wird. Zusätzlich gewährleistet die XPath-Funktion *number()*, dass dieser Textknoten als numerischer Datentyp interpretiert wird. Dieser wird dann im *param*-Element gespeichert und steht für weitere Aufgaben zur Verfügung. Das *copy*-Elementes reproduziert das *item*-Element und der Befehl *attribute* erzeugt einen neuen Attributknoten. Ihm wird der Wert mit Hilfe des *value-of*-Elementes übergeben, in dessen *select*-Attribut der Funktionsaufruf von *func:addition* stattfindet. Beide Summanden, der Parameterwert und die Zahl *1* werden wiederum als Parameter der Funktion übergeben. Die Funktion gibt das Ergebnis in das *select*-Attribut zurück und steht im resultierenden Baum als Attributwert des *value-Attributes*.

Saxon-Prozessor

Der Saxon-Prozessor ist eine XSLT-Implementierung von Michael Kay(Saxonica), die auf Java[42] basiert. Er ist unter einer kostenlosen Version(*Home Edition*) mit allen grundlegenden XSLT 2.0-Funktionen und in zwei kommerziellen Versionen(*Professional* und *Enterprise Edition*) mit erweiterten Features(z.B. integrierten Validierungen oder Unterstützung von XSLT 3.0), erhältlich[43]. Für die Anwendung des Stylemappers wird die *Home Edition* verwendet. Der Prozessor wird über den Calabash-Prozessor aufgerufen sobald eine XSL-Transformation durchgeführt werden muss. Ein manueller Aufruf ist nicht notwendig.

[42] Mindestanforderung: Java JDK 1.5 oder Java 5.0
[43] Vgl.: [30], Abschnitt: *Our Products*, Unterabschnitt: *Server Processing*

2.3 Web-Technologien

2.3.1 Programmierschnittstelle (API) für HTTP-An/Abfragen

Wie im Kapitel 1.1 auf Seite 1 erläutert, befindet sich der Stylemapper als *transpect*-Projekt als Konverter auf dem Webserver. Deshalb erfolgt die Kommunikation des Nutzers mit dem Webserver über HTTP-Anfragen. Um die Programmierschnittstelle (*API*[44]) des Konverters vollständig nutzen zu können, muss sich der Nutzer im Vorfeld authentifizieren. Das geschieht über eine Standard-HTTP-Authentifizierung (*Nutzername:Password*)[45] die für jede Anfrage an den Konverter benötigt[46] wird.

Die API umfasst 7 Funktionen. Darunter befindet sich das Hochladen von Dateien, das Abfragen des Konvertierungsstatus, das Anfordern der Ergebnisliste in Form von Download-Links, der direkte Download des Ergebnisses, das Ausführen einer Aktion im Vorfeld der Konvertierung, das Anfordern von allen bisher hochgeladenen Dateien, inklusive ihrer Konvertierungstatus und zuletzt das Löschen der Daten einer bestimmten Konvertierung[47].

Im folgenden Quellcode 2.15 ist eine Beispiel-HTTP-Anfrage zum Hochladen einer Datei für den Stylemapper dargestellt.

Quellcode 2.15 Upload-HTTP-Request einer Beispieldatei

```
https://transpect.le-tex.de/api/get_status?input_file=Beispiel.
    docx&type=stylemapper
```

Die URL setzt sich aus der Basis-URL (*https://transpect.le-tex.de/api/*), dem Namen der jeweiligen Methode (*get_status*) und zusätzlicher Parameter zusammen. Über *input_file* wird die hochzuladende Datei referenziert und mit Hilfe von *type* der Konverter zur Verarbeitung (*stylemapper*) angegeben.

[44] API – Application Programming Interface
[45] Vgl.: [5], Seite 6, Kapitel 2: *Basic Authentication Scheme*
[46] Vgl.: [6], Abschnitt: *Basics*
[47] Vgl.: [6], Abschnitt: *API Documentation*

2.3.2 DOM-Manipulation mit JavaScript

Um im Web-Browser interaktiv agieren zu können und dem Nutzer das Gefühl zu geben, eine Anwendung vor sich zu haben, ist die Verwendung von JavaScript unabdingbar. Es handelt sich dabei um eine objektorientierte Programmiersprache zur Dynamisierung von Web-Inhalten. Sie ist neben HTML und CSS einer der drei Grundelemente für die Webseitenerstellung. Die Aufgabengebiete der Komponenten sind unterteilt in die Darstellung (CSS), den Inhalt (HTML) und das Verhalten einer Webseite (JavaScript).[48]

JavaScript ist unter dem Namen *ECMAScript* in *ECMA-262* und *ISO/IEC 16262* standardisiert und wird im Einsatz als Web-Technologie, abgesehen von serverseitigen einigen Ausnahmen, wie zum Beispiel *V8*(NodeJS, MongoDB), clientseitig ausgeführt[49]. Diese Programmiersprache beinhaltet den Umgang mit Objektklassen, den daraus erzeugbaren Objekten sowie deren Erweiterbarkeit durch Vererbung und der Verwendung von Arrays und Funktionen.In den folgenden Ausführungen liegt der Fokus auf der Verarbeitung von statischen Inhalten im Web-Browser.

Beim Öffnen eines HTML-Dokumentes analysiert der Parser des Web-Browsers das Dokument und erstellt davon ein Abbild, welches die Darstellung im Browser repräsentiert. Dieses Abbild nennt man *Document Object Model*(DOM)[50]. Dabei handelt es sich um ein Verzeichnis, das in einer Baumstruktur vorliegt und mit JavaScript manipuliert werden kann[51].

Das *window*-Objekt repräsentiert das Browser-Fenster und stellt das Top-Level-Element im DOM dar. Über dieses und das sich darin befindende *document*-Objekt verschafft sich JavaScript Zugang zum HTML-Dokument und der DOM-API[52]. Jedes Element im Dokument wird über einen Element-Knoten referenziert. Besitzen diese Elemente ein ID-, oder Klassenattribut, können sie mit Hilfe der Methode *getElementById()* des *document*-Objektes einzeln angesteuert oder mit *getElementsByClassName()* über eine gemeinsame Klasse in ein

[48] Vgl.: [2], Seite 1, Kapitel 1: *Introduction*, Absatz 1
[49] Vgl.: [2], Seite 307, Kapitel 13: *JavaScript in Web Browsers*, Absatz 1
[50] Vgl.: [4], Seite 229, Kapitel 13: *The Document Model*, Absatz 1
[51] Vgl.: [4], Seite 229, Kapitel 13: *The Document Model*, Absatz 2
[52] Vgl.: [2], Seite 307, Kapitel 13: *The Document Model*, Abschnitt 13.1: *Client-Side JavaScript*, Absatz 1

Array gespeichert werden. Diese Elemente liegen ebenfalls in Form von Objekten vor, die über verschiedene Eigenschaften und Methoden verfügen[53]. Im Quellcode 2.16 sind einige Beispiele zum Umgang mit Objekten, sowie die Anwendung deren Methoden dargestellt.

Quellcode 2.16 Umgang mit Objekten in JavaScript

```
var p_obj = document.getElementById('intro'),
    new_span = document.createElement('span');
p_obj.style.color = 'green';
new_span.innerHTML = 'neuer Text';
p_obj.appendChild(new_span);
```

In den ersten beiden Zeilen befinden sich Variablendeklarationen. Bei der ersten wird ein Element aus dem HTML-Dokument in Form eines Objektes übergeben, das auf den Parameter der Methode *getElementByID()* des *document*-Objektes passt. Im Gegensatz zur ersten Deklaration wird mit der Methode *createElement()* ein neues Element -in diesem Fall ein *span*-Element- erzeugt, welches zunächst nur im Variablenspeicher liegt. Danach erfolgt ein Zugriff auf die *style*-Eigenschaft, welche über Getter- und Setter-Methoden für den Umgang mit CSS-Eigenschaften verfügen. Im Beispiel erfolgt über die Methode *fontColor()* das Ändern der Zeichenfarbe für den Textknoten innerhalb des *p*-Elementes. Den Wert der Zeichenfarbe würde man wiederum erhalten, wenn die Methode ohne einen Parameter aufgerufen würde. Im Anschluss daran wird für das *span*, neuer Inhalt erzeugt, der vom Datentyp *String* sein muss. Die *innerHTML()*-Methode sorgt nämlich dafür, dass der vorherige Inhalt des jeweiligen Elementes damit überschrieben wird. Als letzter Bestandteil des Beispiels ist die Integration eines Elementes in ein anderes dargestellt. Durch die Methode *appendChild()* wird das *span*-Element als Kindelement zum Elementknoten *p* angehängt.

Neben dem DOM-Element-Zugriffen bietet die API sogenannte *Event Handler* an, womit Änderungen von DOM-Elementen bzw. Maus- oder Eingabe-Ereignisse[54] auf dem HTML-Dokument überwacht und in die gegebenenfalls eingegriffen werden können. Sie sind als Eigenschaft von allen DOM-Objekten(*window*, *document*) vertreten. Durch sie ist es möglich, über eine Funktion weitere Algorithmen auszuführen und somit das Verhalten der jeweiligen

[53] Vgl.: [2], Seite 308, Kapitel 13: *The Document Model*, Abschnitt 13.1: *Client-Side JavaScript*, Absatz 6

[54] event, zu deutsch *Ereignis*

Elemente zu verändern, sobald ein Event Handler aktiv wird[55]. Im folgenden Quellcode 2.17 ist die Initialisierung eines Event Handlers als Überwachungsinstanz(Event Listener) über die Methode *addEventListener()* dargestellt.

Quellcode 2.17 Beispiel eines Mausklick-Event Handlers

```
function feedback(ele){
          ele.style.opacity = '0.4';
          window.setTimeOut(function(){
               ele.style.opacity = '1';
          }, 3000);
}
var li_array = document.getElementsByClassName('menu');
for (var i=0; i<li_array.length;i++){
     this.addEventListener('click', feedback(event.target),
        false);
};
```

Im Vorfeld findet die Funktionsdeklaration statt, die später vom Event Handler genutzt wird. Sie soll dem Ziel eine bestimmte Transparenz verleihen, die sich nach 3 Sekunden Wartezeit wieder auflöst. Dazu bekommt sie einen Eingangsparameter *ele*, die für das zu übergebende Element-Objekt steht. Innerhalb der Funktion erfolgt nun der Zugriff auf die CSS-Eigenschaft *opacity*[56], die als Deckkraft eine Wertänderung von 1 runter auf 0.4 erfährt. Im Anschluss daran erfolgt die *window*-Methode *setTimeOut()*, die eine anonyme Funktion nach einer bestimmten Zeitverzögerung –angegeben in Millisekunden– aufruft.

Nach der Deklaration werden Listenelemente aus einem fiktiven Listenmenü über den gemeinsamen Klassennamen *menu* in einem Array gespeichert. Durch eine For-Schleife wird für sie iterativ ein Event Handler erzeugt. Der erste Parameter der Methode stellt den Typ des Ereignisses dar. Hierbei handelt es sich um ein einfaches Mausklick-Ereignis. Als zweiter Parameter folgt eine Anweisung in Form der selbst erstellten *feedback()*-Funktion. Der dritte Parameter stellt einen *boolean*-Wert dar, der für das sogenannte *Capturing*[57] zuständig ist. Würde der Wert auf *true* gesetzt, löst sich dieses Event auch aus, wenn innerhalb der Elemente weitere Kindelemente bestünden, die ebenfalls mit einem Event Handler ausgestattet sind[58].

[55] Vgl.: [2], Seite 309, Kapitel 13: *The Document Model*, Abschnitt 13.1: *Client-Side JavaScript*, Absatz 1

[56] opacity, zu deutsch *Deckkraft*

[57] capturing, zu deutsch *Aufnahme*

[58] Vgl.: [34], Kapitel 1: *Document Object Model Events*, Abschnitt 1.2: *Description of event flow*, Unterabschnitt 1.2.2: *Event capture*, Absatz 2

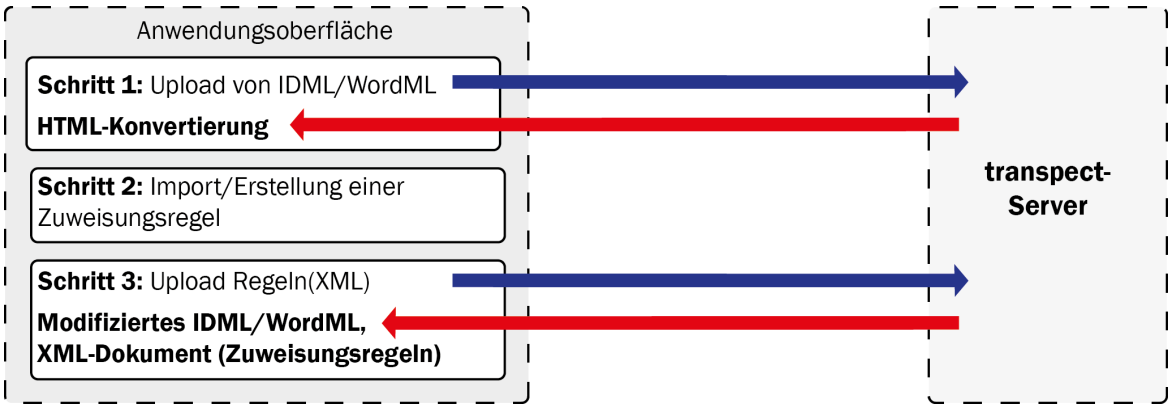
Mit reinem JavaScript auf statische Inhalte zugreifen, kann an vielen Stellen aufwendig sein. Gerade wenn es sich um selektive Elementzugriffe handelt[59]. Hierbei bieten die Methoden der DOM-API nur die Selektoren für Elementknoten über IDs, Klassenattribute oder Element-Tags. Dafür wird die auf allen gängigen Browsern[60][61] lauffähige JavaScript-Bibliothek JQuery[62] hinzugezogen.

2.4 Prinzip der Vorlagenzuweisung

2.4.1 Workflow des Frontends

In diesem Abschnitt wird der Workflow zur Nutzung des Stylemappers über die Anwendungsoberfläche erläutert. Der Nutzer hat im Browser ein leeres Dokumentblatt vor sich und rechts daneben den ersten von vier Menüpunkten geöffnet. Darin sind die drei Arbeitsschritte in Form von Upload-Formularen oder Menüpunkt-Querverweisen zu sehen, deren Durchführung nötig ist, um ein modifiziertes Dokument mit neuen Zielformaten zu erhalten. In der Abbildung 2.4 sind dazu die Schritte schematisch dargestellt.

Abbildung 2.4 Frontend-Workflow



[59] Vgl.: [23], Kapitel: *Using JQuery Core*, Abschnitt: *The JQuery Object*, Unterabschnitt: *The JQuery Object*, Absatz 1

[60] Internet Explorer(9+), Firefox, Chrome, Opera, Safari(5.1+)

[61] Vgl.:[22], Abschnitt: *Browser Support*, Unterabschnitt: *Current Active Support*

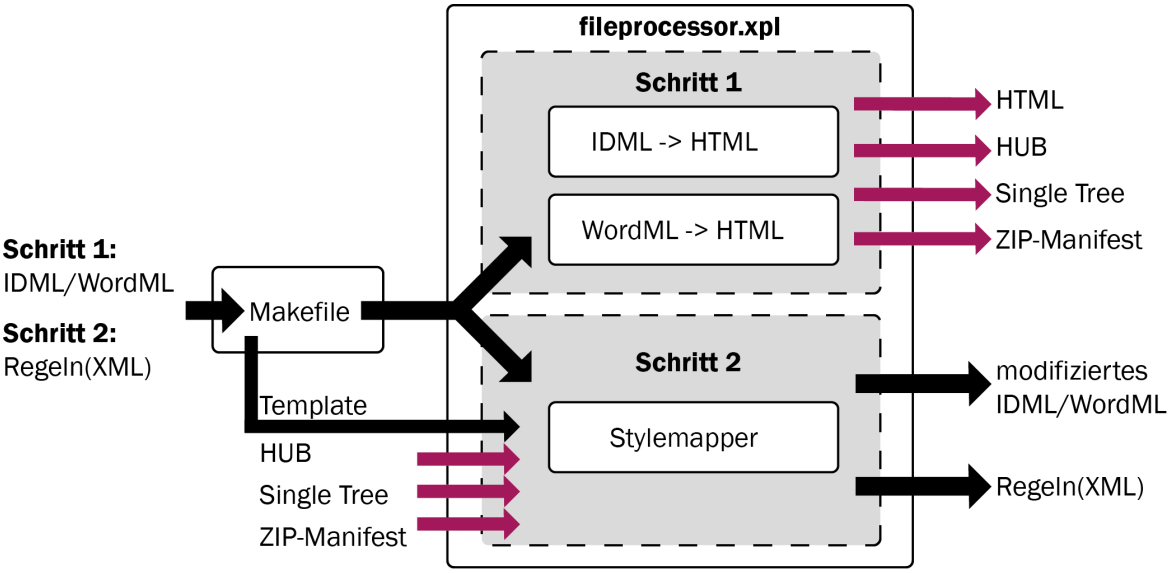
[62] Version 2.1.3

Zunächst wird ein Dokument(IDML- oder WordML-Format) auf den transpect-Server hochgeladen und als ein HTML konvertiertes Dokument in das leere Dokumentblatt zurückgegeben. Im zweiten Schritt müssen ein oder mehrere Zuweisungsregeln, sogenannte *mapping rules* erstellt oder bereits vorliegend als XML-Struktur importiert werden. Zum Import von Regeln existiert ein zweiter Menüpunkt(*Mapping Rules*), der sie zusätzlich in einer tabellarisch angeordneten Liste zusammenfasst. Im Abschnitt 3.1 auf Seite 35 wird auf die Funktionen der einzelnen Bereiche des Stylemappers genauer eingegangen.
Für die Erstellung einer Regel sind die beiden letzten Menüpunkte zuständig. Über den *Mapping Editor* können über Formulare Regeln erstellt werden, wogegen im Menüpunkt *Inspector* Informationen über Formatierungen einsehbar sind, die mit Hilfe des Inspizierens per Mausklick im Dokument bezogen werden. Diese Eigenschaften sind für die Erstellung der Regel nutzbar, da sie zum Mapping Editor hinzugefügt werden können. Der Abschnitt 2.4.3 auf Seite 32 gibt wiederum einen genaueren Einblick in den Aufbau und die Funktionsweise einer Regel. Als dritter und letzter Schritt folgt das Hochladen der Regeln. Nach der Verarbeitung des Dokumentes mit den Regeln über die XProc-Pipelines im Backend, werden Download-Links zur modifizierten InDesign-, oder Word-Datei sowie einem XML-Dokument mit den sich darin befindenden mapping rules generiert. Der detaillierte Aufbau der Anwendungsoberfläche ist im Abschnitt 3.3.1 auf der Seite 59 erläutert.

2.4.2 Workflow des Backends

Wie die Verarbeitung der Dokumente und Regeln über die Backend-Pipelines des Stylemappers funktioniert, wird in diesem Abschnitt erklärt. Zu Beginn ist das Schema des Workflows auf der folgenden Abbildung 2.5 auf der Seite 31 repräsentiert. Dieser setzt sich grundsätzlich aus zwei Schritten zusammen, die aus dem zweiten(Hochladen des Dokumentes) und dritten Schritt(Hochladen der Regeln) des Frontends herrühren.
Ein Problem dabei ist, dass die Konverter-API die Konvertierung eines Dokumentes über einen einzigen Schritt vorsieht und das Ergebnis in einem eigenen Verzeichnis speichert. Das bedeutet, dass die Konvertierungsschritte zunächst unabhängig voneinander sind und somit keine Beziehungen zu einander herstellen können. Das ist jedoch unabdingbar, da ein zweiter

Abbildung 2.5 Backend-Workflow



Schritt notwendig ist, um die Regeln für das vorher hochgeladene Dokument zur Anwendung zu übergeben. Die Verknüpfung der beiden Schritte erfolgt über eine Referenz-ID, die beim ersten Schritt erzeugt und dem Regel-XML-Dokument für den zweiten Schritt übergeben wird. Die genaue Verfahrensweise, wird im Abschnitt 3.2.2 auf der Seite 38 erläutert.

Zu Beginn des ersten Schrittes wird das jeweilige Eingangsdokument durch die Konverter API über eine Makefile in einem make-Aufruf übergeben. Sie differenziert die Formate und entscheidet, welches Template-Dokument für die IDML bzw. WordML-Konvertierung verwendet wird. Danach beginnt die Prozessierung des Dokumentes mit dem Calabash-Aufruf der XProc-Pipepline *fileprocessor.xpl*. Je nach Art des Formates konvertieren zwei importierte Pipelines die Eingangsdokumente zu HTML. Darunter werden 3 Zwischenergebnisse erzeugt, die für den zweiten Schritt verwendet werden. Zuerst das sogenannte *Single Tree*-Dokument, welches die Gesamtheit der ganzen XML-Container-Module des Eingangsdokumentes darstellt. Dann folgt das Hub-Dokument, welches als Ergebnis aus dem jeweiligen Hub-Konverter hervorgegangen ist. Das letzte Zwischenergebnis ist das *ZIP-Manifest*. Dabei handelt es sich um eine XML-Struktur, die den Verzeichnispfad vom Eingangsdokument speichert und für den abschließenden Verpackungsschritt in der Pipeline benötigt wird.

Im zweiten Schritt werden die Regeln in Form eines XML-Dokumentes ebenfalls an den transpect-Konverter übergeben. Es durchquert die Makefile-Instanz zur *fileprocessor.xpl*, woraufhin es an den Eingang der *stylemapper.xpl* gelegt wird. In dieser Pipeline finden die eigentlichen Schritte zur Formatvorlagenzuweisung statt. Sie benötigt neben dem XML-Dokument mit den Regeln, die Zwischenergebnisse aus dem ersten Schritt(Hub-, Single Tree-Dokument, ZIP-Manifest) sowie das Template-Dokument als Eingansparameter. Das Resultat ist eine modifizierte Kopie des Eingangsdokumentes aus dem ersten Schritt und zusätzlich die Regeln in der unbehandelten XML-Struktur. Eine ausführliche Erklärung der Prozesse im Inneren der *stylemapper.xpl* folgt im Abschnitt 3.2.

2.4.3 Regeln zur Vorlagenzuweisung

Eine Regel dient zur Erfassung von Dokumentelementen, wie Absätze, Inline-Elemente und hoch- oder tiefgestellte Zeichen unter Filterung der Regeleigenschaften. Für eine Konvertierung können ein oder mehrere Regeln verwendet werden. Sie stehen als *mapping* –Kurzform für *mapping rules*– im Wurzelelement *mapping-set* und beinhalten als Kindelemente ein oder mehrere Eigenschaften in Form von *prop*-Elementen. In der Tabelle 2.1 auf der Seite 33 sind diese Elemente mit den dazugehörigen Attributen dargestellt.
Das Wurzelelement *mapping-set* besitzt die Attribute *name* und *date* als Meta-Information. Die Regel –verkörpert durch das Kindelement *mapping*– besitzt hingegen Attribute, die aktiv der Konvertierung dienen. Nach der Angabe des Namens steht das *priority*-Attribut. Damit wird die Priorität der Regel festgelegt, die sich mit dem höchsten Wert gegen konkurrierende Regeln durchsetzt. Das *target-style*-Attribut gibt über seinen Wert den Namen eines Zielformates aus den Template-Formatvorlagen an. Darüber wird im späteren Verlauf der Konvertierung die neue Vorlage zugewiesen. Des Weiteren gibt das *target-type*-Attribut darüber Auskunft um welches Dokumentelement es sich handelt. In der derzeitigen Version des Stylemappers bewegt sich die Vorlagenzuweisung jedoch nur um Absatzformate. Als letztes Attribut steht *remove-adhoc*. Dieses gibt an welche Ad-Hoc-Formatierungen aus den Dokumentelementen gelöscht werden sollen. Dabei besteht die Möglichkeit, einzelne CSS-Eigenschaften, die in der Tabelle unter dem Platzhalter *layout props* zusammenge-

Tabelle 2.1 Aufbau einer Regel

Element	Attribut
mapping-set	name
	date
mapping	name
	priority
	target-style
	[targe-type] = 'para', 'inline', 'supscript', 'subscript'
	[remove-adhoc] = '#props', '#all', *layout-props*
prop	name
	value
	min-value
	max-value
	color-h,-s,-l
	color-min-h,-s,-l
	color-max-h,-s,-l
	background-color-h,-s,-l
	background-color-min-h,-s,-l
	background-color-max-h,-s,-l

fasst sind oder mit den Schlüsselwörtern *#all* alle Formatierungen sowie mit *#props* die Formatierungen, die den Namen der Regel-Eigenschaften entsprechen, zu löschen.
In den Regeln befinden befinden sich wiederum Elemente, die deren Filterbedingungen beinhalten. Sie werden jeweils durch ein *prop*-Element verkörpert, deren Charakteristika ebenfalls über Attribute definiert ist. Diese Filterbedingungen setzen sich aus einer CSS-Eigenschaft zusammen, die entweder einen Einzelwert (*value*), einen Wertebereich -bestehend aus einem Minimalwert (*min-value*) und einem Maximalwert(*max-value*) oder einen regulären Ausdruck filtern. Als Werte sind je nach CSS-Eigenschaft Größen bzw. Abstände in Form von Punkten (*pt*), individuelle String- oder Farbwerte zulässig. Lediglich bei den individuellen Strings lässt sich vorerst kein Wertebereich darstellen, was den Eintrag unzulässig macht.
Wenn nach einer Farbeigenschaft gefiltert werden soll, werden durch eine clientseitige Farbkonvertierung aus hexadezimalen oder RGB-Farbwerten HSL[63]-Werte erzeugt, die in separate Wertattribute wie beispielsweise für die Schriftfarbe jeweils in*color-h*, *color-s* und *color-l*) eingeteilt werden. Als Abschlussattribut steht das *relevant*-Attribut, welches die Wirksamkeit

[63] Farbwerte, bestehend aus Farbton(Hue), Sättigung(Saturation) und Helligkeit(Lightness)

der Eigenschaft bei der Konvertierung festlegt. Ihm wird dazu ein Wahrheitswert(*true*, *false*) übergeben.

Kapitel 3

Umsetzung

3.1 Anforderungen

3.1.1 XProc-Modul(Pipelines und XSLT-Schritte)

Die Anforderungen des Backends liegen zum einen bei der Konvertierung des Quelldokumentes in das HTML-Format und zum anderen in der Formatvorlagenzuweisung unter Anwendung von Zuweisungsregeln, worunter mehrere Teilanforderungen anfallen. Dazu gehören die Konvertierung von WordML- und IDML-Formaten in das Hub-Format und die Transformation des Quell- und Template-Dokumentcontainers in jeweils ein XML-Dokument(*Single-Tree*). Des Weiteren müssen diese Dokumente für Weiterverarbeitung gespeichert werden.
Zu der Anwendung der Formatvorlagenzuweisung gehören die Erzeugung eines XSL-Stylesheets, welches den Dokumentinhalt auf Regelinhalte prüft und die Durchfühung weiterer Transformationen die zum Ersatz der Formatvorlagen des Quelldokumentes mit denen aus dem Template-Dokument führen sowie die Verknüpfung der Dokumentelemente mit den Vorlagen.

3.1.2 Web-Anwendung

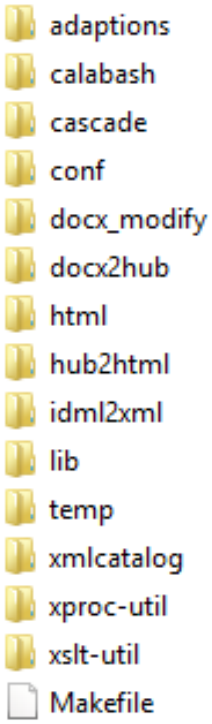
Die Web-Anwendung besitzt in ihrem Anforderungskatalog folgende Schwerpunkte. Zunächst sollte ein Formular bereitstehen, womit eine WordML- bzw. IDML-Datei hochgeladen werden kann. Danach muss über ein HTTP-Request das Resultat in Form einer HTML-Konvertierung vom transpect-Server angefordert und in die Oberfläche integriert werden können. Anschließend sollte es möglich sein, die Absätze und einzelnen Textabschnitte per Mausklick zu inspizieren und die Information über die Formatierungen zu erhalten. Die Namen und Werte der Formatierungen sollten wiederum dem Regel-Editor zur Verfügung stehen, indem sie als Regeleigenschaften instrumentalisiert werden können. Eine weitere Anforderung ist das Erstellen und Importieren von Zuweisungsregeln. Zur Erstellung gehört das Formular für die Regel, sowie die Erweiterung um eine weitere Formular-Maske für die Eigenschaften. Alle vorliegenden Zuweisungsregeln sollten in einer Liste veranschaulicht werden. Dabei sollte es möglich sein, die Anordnung derer per Drag and Drop-Mechanismus in der Rangfolge zu verändern, was gleichzeitig die Priorität konfiguriert. Des weiteren sollte der Nutzer die Möglichkeit besitzen, die Zuweisungsregeln bearbeiten und löschen zu können. Zusätzlich sollte es eine Funktion geben, die eine Vorschau aus den jeweiligen Regeln erzeugt, die eine Markierung für jedes passende Dokumentelement erzeugt und diese wieder unsichtbar werden lässt. Zuletzt sollten die Zuweisungsregeln über einen Upload-Mechanismus zum transpect-Server hochgeladen werden können und das Ergebnis des Backends als Download Link zur Verfügung stehen.

3.2 Implementierung des Workflows im Backend

3.2.1 Verzeichnis des Stylemapper-Projektes

In den folgenden Ausführungen steht die Beleuchtung der Projektstruktur des Stylemappers, sowie die Vertiefung der Prozesse in der *stylemapper.xpl* im Mittelpunkt.
Die Abbildung 3.1 zeigt das Projekt-Verzeichnis, welches alle Komponenten zur Nutzung des Stylemappers beinhaltet. Dazu gehören die projekteigenen Ordner *adaptions*, *conf*, *html*,

Abbildung 3.1 Backend-Verzeichnis des Stylemapper-Projektes



temp, *xmlcatalog* und *lib*, sowie die Makefile-Datei. Wogegen zu den extern bezogenen Daten die Module aus dem *externals*-, *calabash*-, *cascade*-, *docx_modify*-, *docx2hub*-, *hub2html*-, *idml2xml*-, *xproc-util*- und dem *xsl-util*-Ordner des *SVN*[1]-Repositories zählen. Bei den *SVN externals* handelt es sich um bereits bestehende Konverter-Module, deren Pipelines und Stylesheets für unterschiedliche Konvertierschritte benötigt werden.

Aber zunächst zum des Stylemapper-Projektes: Im *adations*-Ordner lagern die Dateien, die benötigt werden, um Pipeline- und XSLT-Strukturen der *externals* zu überschreiben, ohne diese verändern oder kopieren zu müssen. Der *conf*-Ordner besitzt hingegen ein *transprect-conf.xml*-Dokument, in dem die Einstellungen über den Ablauf von Konvertierungs-Prozessen eines Projektes aufgeführt sind. Die Existenz dieser Konfigurationsdatei ist derzeit für ein transpect-Projekt unter der Nutzung der Überschreibungen aus dem *adaptations*-Ordner vorgegeben. Für den Stylemapper ist diese jedoch nicht von Relevanz. Im *html*-Ordner befinden sich die HTML, CSS und JavaScript sowie zusätzliche JavaScript-Bibliotheken und im *temp*-Ordner die Template-Dokumente. Eine *catalog.xml* beinhaltet sämtliche XML-

[1] SVN - Apache Subversion, eine Software zur Versionsverwaltung

Kataloge der Konverter, einschließlich dem des Stylemappers. Diese Kataloge gewährleisten eine Referenzierung des lokalen Verzeichnispfades, hin zu einer universellen Pfad-Struktur. Dadurch kann bei der Nutzung von Online-Ressourcen auf lokale Daten zurückgegriffen werden, die über den Katalog referenziert sind[2]. Das Kernstück des Projektes bildet der *lib*-Ordner. Darin befinden sich XProc,- XSLT,- und Validierungsdokumente des Stylemappers. Nun zu den extern bezogenen Konverter-Modulen. Der *calabash*-Ordner beinhaltet die gesamte Calabash-Software einschließlich des Saxon Prozessors. Der *cascade*-Ordner liefert ein Modul, das wiederum den Modulen zur Dokumentkonvertierung zu Hub und HTML zur Verfügung steht. Darin befinden sich u.a. Mechanismen, die das Einbinden von Dokumenten über Parameterzugaben erweitern und über Fallbacks absichern können[3]. Die Ordner *docx2hub* und *idml2xml* stehen für die Module zur Konvertierung von WordML-Dokumenten und IDML-Dokumenten zum Hub-Format. Zusätzlich hilft das *docx_modify*-Modul beim Manipulieren der WordML-Dokumente[4]. Die letzten beiden Module stellen die *xproc-util*- und *xslt-util*-Ordner dar. Sie liefern Bibliotheken, um zum Einen das Zwischenspeichern von XProc-Schritten für das Debugging[5] und zum Anderen XSLT Funktionen zum Umrechnen verschiedener Größen oder Farbwerten[6].

3.2.2 fileprocessor.xpl-, idml2html.xpl- und docx2html.xpl-Pipeline

In den folgenden zwei Abschnitten werden die Verarbeitungsmechanismen der Backend-Pipelines beschrieben, deren Konzept bereits in der Abbildung 2.5 des Abschnitts 2.4.2 auf Seite 30 erläutert wird. Dabei handelt es sich um die Haupt-Pipeline *fileprocessor.xpl*, sowie deren inneren Pipelines *docx2html.xpl*, *idml2html.xpl* und der *stylemapper.xpl*. Bei der Erklärung der Verarbeitungsprozesse werden die Pipelines und Stylesheets auszugsweise und gemessen an ihrer Rolle und Wichtigkeit in den Quellcodes aufgeführt.

[2] Vgl.: [28], Kapitel: *Editing Documents*, Abschnitt: *Working with XML Catalogs*, Absatz 1

[3] Vgl.: [12], Absatz 1

[4] Vgl.: [13], Abschnitt: *LE-TEX TRANSPECT*, Unterabschnitt: *Detaillierte Liste der Module und Standalone-Werkzeuge*, Tabellenabschnitt: *Container->Container*

[5] Vgl.: [13], Abschnitt: *LE-TEX TRANSPECT*, Unterabschnitt: *Detaillierte Liste der Module und Standalone-Werkzeuge*, Tabellenabschnitt: *XProc-Kernbibliotheken*

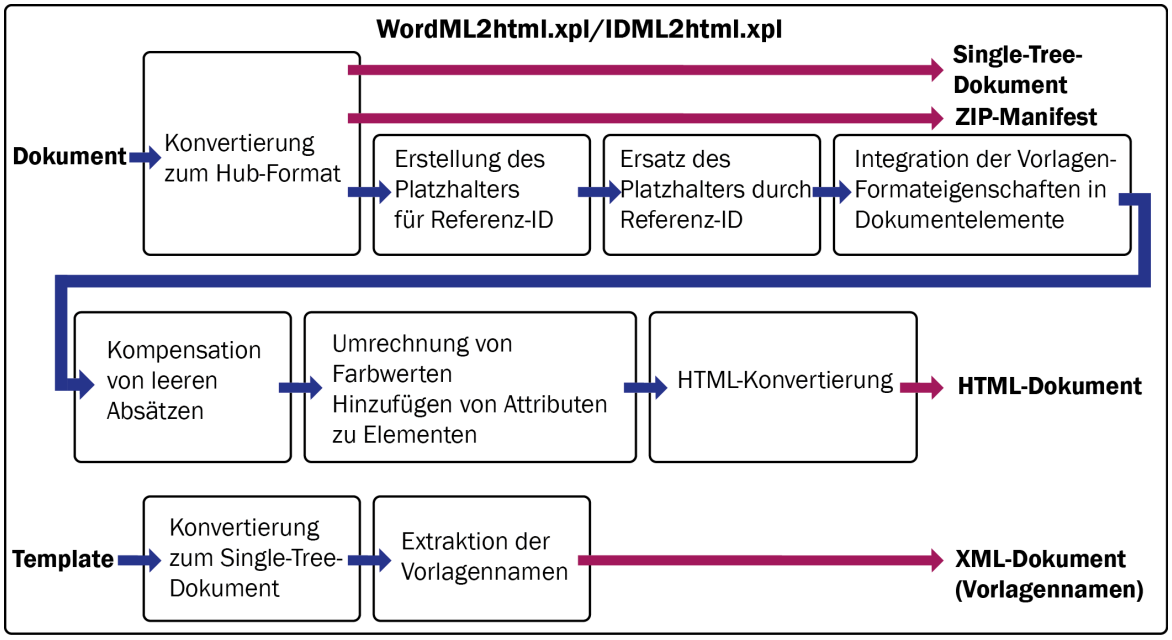
[6] Vgl.: [13], Abschnitt: *LE-TEX TRANSPECT*, Unterabschnitt: *Detaillierte Liste der Module und Standalone-Werkzeuge*, Tabellenabschnitt: *CSS<->CSSa, Hub<->HTML*, xslt-util als letex-util benannt

Quellcode 3.1 Formatunterscheidung mit XPath-Funktion *ends-with()*

```
<p:choose name="distinction">
    <p:when test="ends-with($file, '.docx')">
      <docx2hub:html>
        <p:with-option name="debug" select="$debug"/>
        <p:with-option name="debug-dir-uri" select="$debug-dir-
            uri"/>
        <p:with-option name="file" select="$file"/>
        <p:with-option name="template" select="$template"/>
        <p:with-option name="temp-dir-uri" select="$temp-dir-uri
            "/>
      </docx2hub:html>
    </p:when>
    ...
```

Im Quellcode 3.1 ist ein Teil der Formatunterscheidung des Eingangsdokumentes dargestellt. Über den Schritt *choose* wird eine Fallunterscheidung initiiert, deren Bedingung die Abfrage nach einer bestimmten Dateiendung ist. Sie ist durch ein *when*-Element repräsentiert, welches über die XPath-Funktion *ends-with()* das Ende des Dateipfades vom Eingangsdokument als String in der *$file*-Variable liegt und nach *docx*-, *idml*- oder *xml*-Endung prüft. Bei Zutreffen eines Falles werden die Kindelemente von *when* verarbeitet. Handelt es sich bei dem Eingangsdokument um ein WordML-Dokument(.docx), wird der Aufruf der *docx2html.xpl*-Pipeline durchgeführt. Dabei werden Parameter zur Nutzung der Debug-Option zur Weitergabe des Debug- und temporären Verzeichnisses sowie des Template- und Eingangsdokumentes übergeben. Äquivalent würde die *idml2html.xpl* durch die Dateiendung *.idml* aufgerufen. Vom Grundaufbau sind beide HTML-Pipelines gleich aufgebaut. In der Abbildung 3.2 auf der Seite 40 ist der Workflow schematisch dargestellt.

Abbildung 3.2 Workflow der *docx2html.xpl*- und *idml2html.xpl*-Pipeline



Zunächst wird das Eingangsdokument in beiden Pipelines mit Hilfe der jeweiligen Konverter (*idml2xml.xpl*, *wml2hub.xpl*) in das Hub-Format umgewandelt. Danach folgt die Einrichtung der Referenz-ID, die im zweiten Schritt zur Identifikation mit dem ersten Schritt benötigt wird. Im Quellcode 3.2 sind dazu zwei XProc-Schritte aus der *docx2html.xpl* aufgeführt.

Quellcode 3.2 Referenz-ID

```
<p:insert name="add-episode-keyword" match="/*/dbk:info"
   position="last-child">
  <p:input port="insertion">
    <p:inline>
      <keywordset role="stylemapper" xmlns="http://docbook.org
         /ns/docbook">
        <keyword role="episode"><placeholder/></keyword>
      </keywordset>
    </p:inline>
  </p:input>
</p:insert>

<p:string-replace match="/*/dbk:info/dbk:keywordset[@role ='
   stylemapper']/dbk:keyword[@role='episode']/dbk:placeholder"
   >
  <p:with-option name="replace" select="concat('''', p:system-
     property('p:episode'),'_docx','''')"></p:with-option>
</p:string-replace>
```

Zuerst kommt der *insert*-Schritt. Dieser filtert über das *match*-Attribut das *info*-Element aus dem Hub-Dokument heraus und fügt nach dem letzten Kindelement, dessen Festlegung durch das Attribut *position* bestimmt wurde, über das *input*-Element neuen Inhalt ein. Das darin liegende Pipeline Element *inline* erzeugt wiederum einen Dokumentknoten, der den Inhalt transportiert. In das *info*-Element wird ein weiteres *keyword-set* angelegt, welches im Kindelement *keyword* die Referenz-ID im nächsten Schritt geliefert bekommt. Diese beiden Schritte müssen voneinander getrennt ausgeführt werden, da das Kernstück der Referenz-ID über eine XPath-Funktion bezogen wird, welche nicht innerhalb einer Knotenstruktur eines Dokumentes aufrufbar ist. Deshalb ist dafür zunächst ein Platzhalterelement *placeholder* vorgesehen, das später ersetzt werden kann. Die Antwort auf die Frage, wieso die Rolle des Schlüsselwort-Elementes *episode* ist, folgt im nächsten XProc-Schritt. Dieser lautet *sting-replace* und ersetzt das Ziel durch einen String, der die Referenz-ID darstellen soll. Sie setzt sich aus einer systematisch generierten ID und der Dateiendung des Eingangsformates, des *_docx*, zusammen. Diese ID wird bei jedem Calabash-Aufruf erzeugt und ist durch die erweiterte XPath-Funktion *p:system-property('p:episode')* abrufbar[7]. Die *concat()*-Funktion setzt letztendlich diese sogenannte Episoden-ID mit der Dateierweiterung zu einem String zusammen. Im Anschluss folgt die Übergabe der CSS-Eigenschaften aus den Formatvorlagen (*rules*) in die jeweiligen Dokument-Elemente(*para*,*phrase*, *subscript*, *supscript*). Das ist notwendig, um später die Vorlagenzuweisung über das generierte XSLT-Stylesheet aus den Regeln zu realisieren. Im Quellcode 3.3 findet sich ein Auszug aus dem Stylesheet des dafür zuständigen XSLT-Schrittes.

Quellcode 3.3 Hinzufügen der CSS-Eigenschaften aus Vorlagen(XSLT)

```
<xsl:key name="rule-by-role" match="css:rule" use="@name"/>
<xsl:template match="*[@role]" priority="3">
    <xsl:copy>
      <xsl:copy-of select="key('rule-by-role', @role)/(@css:*|
          @xml:lang)"/>
      <xsl:apply-templates select="@*|node()"/>
      </xsl:copy>
</xsl:template>
```

[7] Vgl.: [21], Kapitel 2 *Pipeline Concepts*, Abschnitt 2.7 *XPath Extension Functions*, Unterabschnitt: 2.7.1 *System Properties*

Über die *key*-Funktion wird die im *role*-Attribut referenzierte Formatvorlage über den Namen zum *rule*-Element getunnelt. Über die Template-Regel werden alle Elemente gefiltert, die eine Formatvorlage besitzen. Durch die Anwendung der Schlüsselfunktion wird auf ein *rule*-Element mit einem übereinstimmenden Wert des *role*-Attributes referenziert und deren Eigenschaften, inklusive des *lang*-Attributes in das jeweilige Dokumentelement kopiert. Danach folgt ein weiterer Schritt, der zur Kompensation von leeren Absätzen dient. In dem dazu verwendeten Stylesheet werden Absätze gefiltert, die entweder keinen Textknoten oder nur Leerzeichen beinhalten. Der Abstand des Vorgänger- und Nachfolger-Absatzes, der kompensiert werden muss, errechnet sich aus 4 Komponenten: Dem oberen und unteren Zeilenabstand, der Schriftgröße und der Zeilenhöhe. Diese Kompensation wird dem Nachfolgeelement als neuen Zeilenabstand übergeben. Im Quellcode 3.4 ist das Template dargestellt, welches auf die Absätze passt, deren Vorgängerabsatz keine weiteren Kindelemente besitzt und keinen Textknoten beinhaltet.

Quellcode 3.4 Template-Regel zum Orten eines leeren Absatzes

```
<xsl:template match="para[preceding-sibling::para[1][not(./
    node()) and not(./text())]]">
  <xsl:variable name="pre-sib" select="preceding-sibling::para
     [1]" as="element(para)?"/>
  <xsl:copy>
    <xsl:attribute name="orig:margin-top" select="@css:margin-
       top"/>
    <xsl:apply-templates select="@*|node()">
      <xsl:with-param name="pre" select="$pre-sib" tunnel="yes
         "/>
    </xsl:apply-templates>
  </xsl:copy>
</xsl:template>
```

Die Filterbedingungen werden über die *not()*-Funktion realisiert. Sofern der vorhergehende Element- und der Textknoten des aktuellen nicht existiert, gibt Sie den Wahrheitswert *wahr* zurück. In der Variable *pre-sib* wird das Element des Vorgängerabsatzes gespeichert und dem Parameter *pre* des Template-Aufrufes *apply-templates* mitgegeben. Dazu wird der XPath-Ausdruck *preceding-sibling::* verwendet, welcher den Vorgängerknoten zu dem Knoten bildet, der nach den beiden Doppelpunkten folgt. Als Rücksicherung wird der alte obere Zeilenabstand des gefilterten Absatzes in das neu erzeugte, gleichnamige Attribut *margin-top*

unter dem zur namentlichen Trennung ebenfalls neu erzeugten Namensraum *orig* gespeichert. Für die Berechnung wurde eine Funktion erstellt, die den Zahlenwert aus dem jeweiligen Attribut filtert. Sie findet in der nächsten Template-Regel, die auf den oberen Zeilenabstand passt, ihre Anwendung. Die Umsetzung dazu befindet sich im Quellcode 3.5.

Quellcode 3.5 Template-Regel für das Attribut der Abstandskompensation

```
<xsl:template match="@css:margin-top">
  <xsl:param name="pre" as="element(para)?" tunnel="yes"/>
  <xsl:variable name="line-height" select=" css:length-to-
     double($pre/@css:line-height)*css:length-to-double($pre/
     @css:font-size)"/>
    <xsl:attribute name="{name()}"
        select="concat(
                string(
                sum((css:length-to-double(.),
                css:length-to-double($pre/@css:margin-bottom),
                css:length-to-double($pre/@css:margin-top),
                css:length-to-double($pre/@css:font-size),
                $line-height)) ),'pt')"/>
</xsl:template>
```

Im XSL-Element *variable* wird der tatsächliche Abstand der Zeilenhöhe(*line-height* aus der Multiplikation des Verhältnisses mit der Schriftgröße(*font-size*) ermittelt und in der Variable *line-height* gespeichert. Dafür wird die eigens erstellte Funktion *length-to-double()* herangezogen. Im Anschluss wird ein Element zur Erzeugung bzw. Beibehaltung des *margin-top*-Attributes eingesetzt, welches über die XPath-Funktion *name()* den Namen des aktuellen Knotens, einschließlich des Namensraumes zurückgibt. Dessen Abstandswert ergibt sich aus der Summe der vier Komponenten des leeren Vorgängerabsatzes und dem eigenen oberen Abstand des aktuellen Absatzes. Sie wird zusammen mit der Einheit *pt* zu einem String zusammengefügt. Nach dem Kompensationsschritt, folgt die Umrechnung der Farbeigenschaften und das Hinzufügen der Einzelwerte zu den Absätzen durch Attribute. Das Stylesheet, welches für die Transformation verwendet wird, befindet sich im Anhang B auf der Seite 82. Darin sind vier Funktionen enthalten. Zum ersten eine zum Potenzieren einer Zahl, die für die zweite Funktion zur Konvertierung von hexadezimalen zu dezimalen Zahlen verwendet wird. Diese Funktion steht wiederum für die dritte Funktion zur Verfügung, welche die dezimalen Zahlen in die passende RGB-Syntax umsetzt. Die vierte und letzte

Funktion nutzt die 3 RGB-Farbwerte zur Umwandlung zu den HSL-Farbwerten. Dazu nutzt sie im Falle von eingängigen hexadezimalen Farbwerten die vorher beschriebenen Funktionen, um einen einheitlichen RGB-Farbwert zur Verarbeitung zu besitzen. Dieser liegt als String in der Form *rgb(255, 255, 255)*[8] vor, deren Zahlenzeichen über die XPath-Funktion *replace()* mit dem regulären Ausdruck im Filterargument isoliert, indem die Zeichenkette *rgb(* und die abschließende Klammer gelöscht werden. Sie werden anschließend mit Hilfe der Funktion *tokenize()* getrennt durch die Kommas in eine Sequenz gesetzt. Nach der Isolierung der Zahlen erfolgt die Anwendung eines Algorithmus zur Umrechnung in den Winkel des Farbtons, dem Sättigungs- und dem Helligkeitswert. Im Anschluss wird das resultierende Hub-Dokument über den XProc-Schritt *store* gespeichert, der im Quellcode 3.6 aufgeführt ist.

Quellcode 3.6 Speichern des Hub-Dokumentes

```
<p:store name="save-hub">
    <p:with-option name="href" select="concat($main-uri, '/',
       p:system-property('p:episode'),'/', p:system-property('
       p:episode'), '_hub.xml')"/>
</p:store>
```

Bei diesem Schritt bestimmt der Parameter *href* den Zielpfad. Er setzt sich aus dem Ausgabeverzeichnis über die Variable *$main-uri* sowie dem Ordner mit der Episodennummer und der Kombination des Format-Typs und der Referenz-ID zusammen. Da dieser Pfad vorher noch nicht existiert, wird dieser neu angelegt. Das bedeutet, dass neben den Ordnern der beiden Konvertierungsschritte, ein dritter besteht, der die notwendigen Dokumente für den zweiten Schritt beinhaltet. Darin erfolgt parallel ebenfalls die Speicherung des ZIP-Manifestes und des Single Trees, die als Zwischenergebnisse aus der Hub-Konvertierung bezogen werden auf die gleiche Art und Weise. Einziger Unterschied ist hierbei jedoch der Bezug der Daten für den Speichervorgang. Während das Hub-Format als Resultat des letzten Schrittes gespeichert wird, müssen die beiden anderen Dokumente über den XProc-Schritt *identity* reproduziert werden, um zu gewährleisten, dass der Eingang des Speicherschrittes mit einem Ausgang verknüpft ist. Im Quellcode 3.7 auf der Seite 44 ist der *identity*-Schritt aufgeführt.

[8] Beispielwert für die Farbe *weiß*

Quellcode 3.7 Reproduktion eines Zwischenergebnisses

```
<p:identity>
  <p:input port="source">
    <p:pipe port="zip-manifest" step="docx2hub"></p:pipe>
  </p:input>
</p:identity>
```

Um das Zwischenergebnis aus dem Hub-Konverter zu beziehen, muss man den jeweiligen Ausgangsport der externen Pipeline abgreifen. Durch das *pipe*-Element kann beispielsweise das Resultat aus dem Port *zip-manifest* der *wml2hub.xpl*-Pipeline, deren Aufruf den Namen *docx2hub* trägt, als Eingang des *identity*-Schrittes festgelegt werden. Der Ausgang dieses Schrittes steht nun dem Speichervorgang zur Verfügung. Im Anschluss daran erfolgt die Konvertierung des Hub-Dokumentes zum HTML-Format mit Hilfe der externen *hub2html.xpl*-Pipeline. Das Ergebnis wird ebenfalls per *store* gespeichert, welches jedoch nicht im Zusatzordner, sondern im Ordner des ersten Upload-Schrittes liegt. Der Client kann das HTML-Dokument nun über einen Callback-Request anfordern und für die Integration in die Web-Anwendung nutzen. Ähnlich verhält es sich bei der Bereitstellung der Template-Formatvorlagennamen für den Regel-Editor im Fontend. Dafür wird gegen Ende der *doc2html.xpl* das Template-Dokument über den XSLT-Schritt *single-tree* aus der externen *docx2hub*-Pipeline zu einem Single Tree-Dokument konvertiert, wessen Vorlagennamen durch eine weitere XSL-Transformation extrahiert wird. Ein Auszug aus dem betreffenden Stylesheet ist im Quellcode 3.8 dargestellt.

Quellcode 3.8 Auszug aus dem Schritt zur Extraktion von Formatvorlagen

```
<xsl:template match="@*[not(local-name() = 'styleId')]|
  ParagraphStyle/@*[not(local-name() ='Name')]|
  *">
</xsl:template>
```

Darin steht eine Template-Regel, die für die WordML,- als auch für die IDML-Single Tree-Dokumente alle irrelevanten Elemente und Attribute entfernt. Die Filterkriterien sind hierbei alle Elemente, sowie alle Attribute, die nicht die Referenznamen der Formatvorlagen beinhalten. Template-Regeln mit höherer Priorität sorgen dafür, dass das Wurzelelement der Formatvorlagen sowie die einzelne Elemente der Formatvorlagen samt ihrer Referenz-

Attribute im Ergebnis bestehen bleiben. Diese resultierende Struktur wird im Anschluss gespeichert und ebenfalls über einen Callback-Request bezogen.
Nach dem Durchlauf der gesamten Pipeline, stehen das Hub-, Single Tree- und ZIP-Manifest-Dokument für den zweiten Konverter-Schritt zur Verfügung.

3.2.3 stylemapper.xpl-Pipeline

Dieser Abschnitt behandelt den zweiten Konvertier-Schritt im Prozess der Formatvorlagenzuweisung, der sich hauptsächlich in der *stylemapper-xpl* abspielt. Um diese Pipeline anzusteuern, muss ein XML-Dokument mit Regeln zum *transpect*-Konverter hochgeladen werden. Es wird durch die bereits im Abschnitt 3.2.2 auf der Seite 38 beschriebene Datei-Differenzierung an die Stylemapper-Pipeline weitergeleitet. Für ihre Nutzung bedarf es noch zusätzlich die drei Dokumente aus dem ersten Konvertier-Schritt. Bevor diese geladen werden erfolgt ein Zwischenschritt zur Normalisierung des Dateipfades. Mit Hilfe der importierten *file-uri.xpl*-Pipeline aus dem *xproc-util*-Modul wird der relative Dateipfad oder die URL zu einer Datei-URI normalisiert und im Ergebnisdokument in einem *local-href*-Attribut hinterlegt.[9]. Im folgenden Quellcode 3.9 ist das Laden des Single Trees sowie der Aufruf der externen Pipeline dargestellt.

Quellcode 3.9 Laden des Single Trees (fileprocessor.xpl)

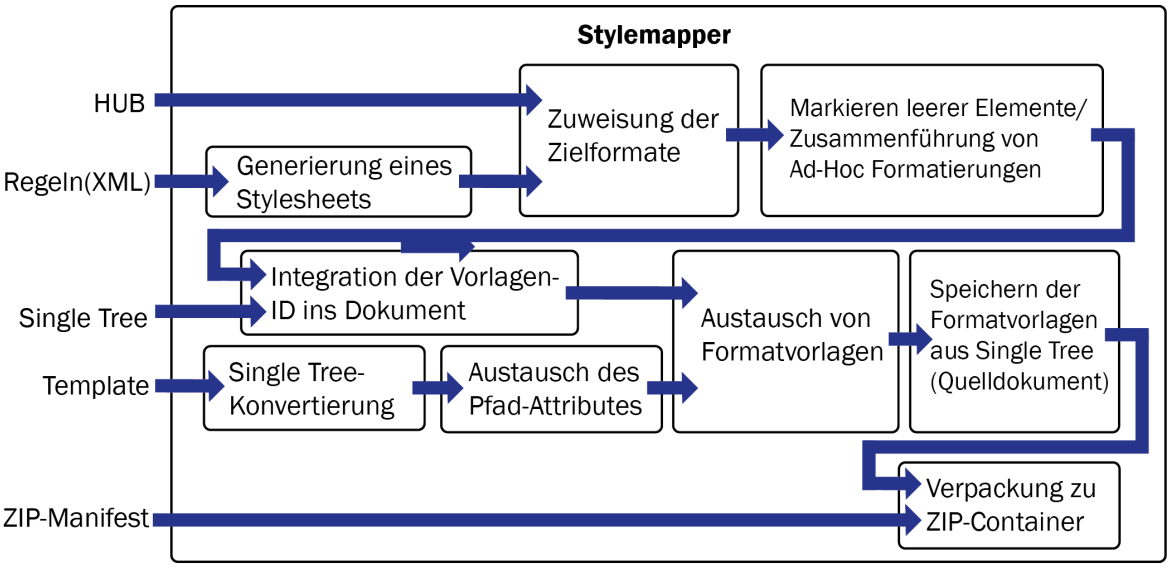
```
<p:variable name="episode"
        select="replace(tokenize($file, '/')[last()],
                        '(_docx|_idml)\.xml', '')">
</p:variable>
<transpect:file-uri name="single-tree-uri">
  <p:with-option name="filename" select="concat($main-uri,'/',$
     episode,'/', $episode, '_single_tree.xml')"></p:with-option
     >
</transpect:file-uri>

<p:load name="load-single-tree">
  <p:with-option name="href" select="/*/@local-href"></p:with-
     option>
</p:load>
```

[9] Vgl.: [11], Absatz 1 und Abschnitt: *Output format*

Die Referenzierung auf die Dokumente erfolgt über die Episoden-Nummer, die im ersten Konvertier-Schritt generiert wurde. Das XML-Dokument, welches die Regeln inne hat, bekommt seinen Dateinamen über die Referenz-ID, die aus dem als *stylemapper* benannten *keyword-set*-Element des konvertierten HTML-Dokumentes bezogen wird. In der Variable *$episode*, wird die Episode-Nummer des vorherigen Calabash-Aufrufes aus dem ersten Schritt gespeichert, um den Pfad für den *file-uri*-Schritt zusammenzusetzen. Aus dem Resultat bezieht der *load*-Schritt über das Attribut *local-href* den normalisierten Pfad, um -wie im Quellcode dargestellt- das Single Tree-Dokument zu laden. Das Hub-Dokument und das ZIP-Manifest werden äquivalent behandelt und stehen nun für die Eingangsports der *stylemapper.xpl* zur Verfügung. Der Workflow dieser Pipeline ist in der Abbildung 3.3 schematisch dargestellt. Zu Beginn erfolgt eine XSL-Transformation, welche die Zuweisungs-

Abbildung 3.3 Workflow der Stylemapper-Pipeline



regeln verwendet, um ein neues XSL-Stylesheet zu erzeugen. Darin befinden sich Templates, deren Filterkriterien aus den Eigenschaften der Regeln generiert wurden. Über sie bekommt das jeweilige Dokumentelement eine neue Formatvorlagen-Referenz zugewiesen. Da dieser XSLT-Schritt der *stylemapper.xpl* eine der wichtigen Stationen im Worflow ist, wird deren Inhalt durch die folgenden Quellcodes genauer betrachtet. Das Prinzip ein Stylesheet zu generieren, beruht dabei auf dem Aufbau einer Stylesheet-Struktur, die während des Transfor-

mationsprozesses vom Compiler ignoriert wird und erst danach eine lesbare XSL-Syntax für die weitere Verwendung aufweist. Um dieses Prinzip umzusetzen, wurde für die Elemente des Stylesheet-Gerüstes ein neuer Namensraum mit dem Präfix *xslout* verwendet. Im Quellcode 3.10 ist die Namensraum-Deklaration sowie das XSL-Element *namespace-alias* angegeben, um dafür zu sorgen, dass das Dummy-Präfix nach der Kompilierung dem XSL-Namensraum angehört.

Quellcode 3.10 Wurzelelement der *mapping2xsl.xsl*

```
<xsl:stylesheet xmlns:xsl="... " xmlns:xs="..."
  xmlns:xslout="foo"
  version="2.0">
  <xsl:namespace-alias stylesheet-prefix="xslout" result-prefix=
     "xsl"/>
```

Das Dummy-Präfix bedeutet, dass der Namensraum *foo* dieses Stylesheet-Gerüstes willkürlich festgelegt ist, um einen Platz freizuhalten, der nach der Transformation mit dem XSL-Namenraum ersetzt wird. Danach folgt das erste Template, welches auf das Wurzelelement der Zuweisungsregeln passt und ein Identity-Template-Gerüst erzeugt, sowie weitere Template-Regeln für die einzelnen Zuweisungsregeln aufruft. Des Weiteren ist in diesem Gerüst eine Import-Funktion für das Stylesheet *lengths.xsl* aus der *xslt-util*-Bibliothek integriert, welches XSLT-Funktionen zum Umrechnen von verschiedenen Schrift,- bzw. Abstandsgrößen liefert. Im Quellcode 3.11 ist das Stylesheet-Gerüst sowie Template-Aufrufe dargestellt.

Quellcode 3.11 Stylesheet-Generierung: Wurzelelement

```
<xsl:template match="mapping-set">
   <xslout:stylesheet
     xmlns:xsl=".." xmlns:rel=".." xmlns:w=".." xmlns:css=".."
     xmlns:letex="http://www.le-tex.de/namespace"
     xmlns:docx2hub="http://www.le-tex.de/namespace/docx2hub"
     xmlns:sm="http://www.le-tex.de/namespace/stylemapper"
     exclude-result-prefixes="letex" version="2.0">
     <xslout:import href="http://transpect.le-tex.de/xslt-util/
        lengths/lengths.xsl"/>
     <xsl:apply-templates select="*"/>
     <xslout:template match="*|@*">
       <xslout:copy>
           <xslout:apply-templates select="@*|node()"/>
       </xslout:copy>
```

```
      </xslout:template>
   </xslout:stylesheet>
</xsl:template>
```

Im *xslout:stylesheet*-Element sind neben der Namenräume von WordML, CSS, XSL zusätzlich die Namenräume für eigene XSL-Funktionen des Stylemappers, le-tex eigene XSLT-Funktionen sowie für den Aufruf der Hub-Konverter-Pipeline enthalten. Das Attribut *exclude-result-prefixes* stellt sicher, dass nach der Anwendung des generierten Stylesheets der Namensraum mit dem Präfix *letex* wieder entfernt wird. Danach folgt das Template, das die einzelnen Zuweisungsregeln zu einem neuen Template-Regel-Gerüst verarbeitet und im Quellcode 3.12 beschrieben steht.

Quellcode 3.12 Stylesheet-Generierung: Zuweisungs-Template

```
<xsl:template match="mapping">
    <xsl:variable name="predicates" as="xs:string+">
      <xsl:apply-templates select="@target-type, prop"/>
    </xsl:variable>
    <xslout:template match="*{$predicates}" priority="{./
       @priority}">
      <xslout:copy>
        <xslout:attribute name="role" select="'{@target-style}'"
           />
        <xslout:attribute name="layout-type" select="'{@target-
           type}'"/>
        <xslout:attribute name="mapping-priority" select="./
           @priority"/>
        <xslout:attribute name="mapping-rules" select="'{prop/
           @name}'"/>
        <xslout:attribute name="sm:mapping-name" select="'{@name
           }'"/>
        <xsl:if test="@remove-adhoc">
          <xslout:attribute name="sm:remove-adhoc"
            select="'{if (@remove-adhoc = '#props')
                     then sm:prop-names-to-remove-adhoc-att(
                        prop)
                     else @remove-adhoc}'"/>
        </xsl:if>
        <xslout:apply-templates select="@* except @role, node()"
           />
      </xslout:copy>
    </xslout:template>
</xsl:template>
```

Darin wird zunächst die Variable(*predicates*) angelegt, die die Ergebnisse aus dem anschließenden Template des @*target-type*-Attributes und der *prop*-Elemente aus der Zuweisungsregel bezieht. Das nachfolgende *xslout:template* nutzt diese Variable, um die sich darin aneinanderreihenden Filterkriterien in das *match*-Attribut des Template-Gerüstes zu legen. Des Weiteren verfügt es über ein *priority*-Attribut, das die Priorität des Templates-Gerüstes festlegt. Da die Priorisierung der Templates äquivalent zu den Zuweisungsregeln ist, bekommen sie die Prioriäten aus den *mapping*-Elementen. Mit *xslout:copy* und den darin befindlichen *xslout:attribute* und *xslout:apply-templates* wird die Vorraussetzung geschaffen, die Attribute aus den *prop*-Elementen der Zuweisungsregeln bei der Anwendung des generierten Stylesheets in die Dokumentelemente des Hub-Dokumentes zu kopieren. Danach folgt eine Abfrage auf die Existenz des Attributes für das Löschen von Ad-Hoc-Formatierungen. Wenn das *remove-adhoc*-Attribut vorhanden ist, wird geprüft, ob es den Schlüssel-Wert *#props* besitzt. Trifft dieser Fall ein, so werden über die eigen erstellte XSLT-Funktion *prop-names-to-remove-adhoc-att()* sämtliche Eigenschaftsnamen einer Zuweisungsregel in das *remove-adhoc*-Attribut des Template-Gerüstes gelegt. Falls nicht, wird der ursprüngliche *remove-adhoc*-Wert genutzt. Im Anschluss daran folgen Templates für das Attribut @*target-type*, das die Filterkriterien für den Typ des Dokumentelementes liefert. Im folgenden Quellcode 3.13 auf der Seite 50 ist das Template für einen Absatz (*para*) aufgeführt.

Quellcode 3.13 *target-type* und *delete-props*-Templates der *mapping2xsl.xsl*

```
<xsl:template match="@target-type[. = 'para']">
    <xsl:text>[name() = ('para')]</xsl:text>
</xsl:template>
```

Über das XSL-Element *text* wird ein Textknoten erzeugt, der als Ergebnis in die *$predicates*-Variable überführt wird. Das Template für den Typ des Inline-Elementes ist hinsichtlich dieses Templates identisch aufgebaut. Danach folgt das Template für die *prop*-Elemente der Zuweisungsregel. Sie beinhaltet -ähnlich wie das *mapping*-Template- eine Variablendeklaration, in die weitere Filterkriterien eingebaut werden. Diese Kriterien setzen sich aus den Ergebnissen der Templates zusammen, die für die Attribute der Regel-Eigenschaften angewandt werden. Zur Veranschaulichung des Aufbaus ist im Quellcode 3.14 auf der Seite 51 dieses Template aufgeführt.

Quellcode 3.14 Stylesheet-Generierung: Template für die Regel-Eigenschaften

```
<xsl:template match="prop">
 <xsl:variable name="conditions" as="xs:string*">
   <xsl:choose>
     <xsl:when test="matches(./@name, 'color') or matches(./
        @name, 'background-color')">
       <xsl:apply-templates select="@regex, @color-h, @min-color
          -h..."/>
     </xsl:when>
     <xsl:otherwise>
       <xsl:apply-templates select="@value, @min-value, @max-
          value, @regex"/>
     </xsl:otherwise>
   </xsl:choose>
 </xsl:variable>
 <xsl:sequence
   select="string-join(
                   for $c in $conditions return concat('[', $c,
                      ']'),
                    '')"
 />
</xsl:template>
```

Die Ergebnisse der Template-Aufrufe werden mit Hilfe der *string-join* zu einem String transformiert. Dabei sind die einzelnen Iterations-Instanzen(*c* in *conditions*) im String mit eckigen Klammern voneinander abgegrenzt, um der Syntax der Filterkriterien gerecht zu werden. Innerhalb der Deklaration befindet sich die Fallunterscheidung. Handelt es sich bei der Regel-Eigenschaft um eine Farbe, in dem Falle Schriftfarbe oder Hintergrundfarbe, werden lediglich die Templates für die einzelnen Attribute der konvertierten Farbwerte und *regex*-Attribut aufgerufen. Ist die Eigenschaft keine Farbe, erfolgt der Template-Aufruf der regulären Attribute. Wegen der Vielzahl der möglichen Farb-Attribute einer *prop* ist der Quellcode zur besseren Übersicht gekürzt worden. Sie sind in der Tabelle 4.1 auf Seite 74 nachzulesen. Letztendlich wird über das XSL-Element *sequence* die String-Kombination als Ergebnis des Templates ausgegeben. Im Anschluss folgen weitere Attribut-Templates , die dem Aufruf des *prop*-Templates folgen und die Erzeugung der Filterkriterien unterstützen. Die Abarbeitung derselben bestimmt sich hinsichtlich der Rangfolge zum einen über die Höhe ihrer Prioritäten und zum anderen in Bezug auf den Modus des spezifischen Aufrufes über ihr *mode*-Attribut. Im folgenden Quellcode 3.15 ist das Haupt-Template dargestellt, welches für

alle Attribute einen String zusammensetzt, der über weitere individuel instanzierte Templates zusammengesetzt wird.

Quellcode 3.15 Allgemeines Attribut-Template der *mapping2xsl.xsl*

```
<xsl:template match="@value | @min-value | @max-value | @regex |
    @color-h | @color-s | @color-l ..." priority="2" as="
   xs:string?">
  <xsl:variable name="prelim" as="item()*">
    <xsl:next-match/>
  </xsl:variable>
  <xsl:sequence select="string-join($prelim, '')"/>
</xsl:template>
```

Dazu wird eine Variable definiert, die über die Funktion *nextmatch* diese Templates aufruft. Anschließend wird das jeweilige Template-Resultat in String-Form über *xsl:sequence* als Ergebnis in das *prop*-Template weitergetragen. Als Beispiel für die String-Zusammensetzung wird im Quellcode 3.16 ein Template für das *value*-Attribut aufgezeigt, dessen Ergebnis in der Variable *prelim* gespeichert wird.

Quellcode 3.16 Beispiel für ein attributspezifisches Template der *mapping2xsl.xsl*

```
<xsl:template match="@value">
  <xsl:apply-templates select="../@name" mode="value"/>
  <xsl:choose>
    <xsl:when test="matches(., '^\d+pt$')">
      <xsl:text> = </xsl:text>
      <xsl:sequence select="'letex:length-to-unitless-twip'"/>
      <xsl:text>('</xsl:text>
      <xsl:value-of select="."/>
      <xsl:text>')</xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text> = '</xsl:text>
      <xsl:value-of select="."/>
      <xsl:text>'</xsl:text>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Zu Beginn des Template-Inhaltes erscheint ein weiteres *apply-templates*-Element, das im Modus *value* die vorletzte Template-Instanz aufruft. Dieses Template differenziert den Wert nach dem Eigenschaftsnamen von Abständen bzw. Schriftgrößen, einheitenlosen Zahlenwerten oder einfachen Zeichenketten. Darüber werden Templates über den Modus *na-*

me2expression aufgerufen, die als letzte Instanz dafür sorgen, dass CSS-Eigenschaftsnamen ihr Namensraum-Präfix angefügt bekommen und der jeweilige *prop*-Wert als Parameter gegebenenfalls einer Umrechnungsfunktion zugestellt und die Zeichenketten in einfache Anführungszeichen gesetzt werden können. Das gewährleistet nach der *prop*-Erstellung die Anwendung der XPath-Filter im erzeugten Stylesheet auf die Dokumentelemente im Hub-Dokument und damit den Vergleich dieser Elemente mit der Zuweisungsregel. Darunter wird der restliche Inhalt des Filterkriteriums generiert. Zurück zum Beispiel-Template: Nach dem Template-Aufruf folgt eine Fallunterscheidung nach dem Attributswert *pt*, die beim Eintreten den Namen der Umrechnungsfunktion *lenght-to-unitless-twip* in das Template-Ergebnis integriert. Diese Funktion soll später die Einheit *pt* in *twip* umrechnen, um im Falle unterschiedlicher Einheiten mit der Universalgröße operieren zu können.

Über das *text*-Element wird für das Ergebnis der aufgerufenen Templates der Wertvergleich mit einem = und Anführungszeichen erstellt, die den Wert des Attributes umfassen, der über das *value-of*-Element bezogen wird. Würde es sich bei dem Filterattribut um einen Extremwert handeln, dann würde dementsprechend ein >- oder <-Zeichen statt dem = verwendet werden. Für das Attribut eines regulären Ausdruckes würde sich der Textknoten aus der Anwendung der XSL-Funktion *matches* auf dieses zusammensetzen. Zum Abschluss der Beschreibung der Stylesheet-Erzeugung soll dieses Beispielergebnis aus einer Zuweisungsregel im Quellcode 3.17 dienen.

Quellcode 3.17 Beispiel einer Stylesheet-Erzeugung

```
<!-- Zuweisungsregel-->
<mapping name="small-fonts" priority="2" target-style="header2"
   target-type="para">
  <prop name="font-size" min-value="12pt" max-value="20pt"
     relevant="true"/>
</mapping>
<!-- generiertes XSL-Template-->
<xsl:stylesheet xmlns:css="..." xmlns:xsl="..." xmlns:letex="...
   "
                    ...
                exclude-result-prefixes="xs letex docx2hub"
                version="2.0">
 <xsl:template match="*[name() = ('para')]
     [letex:length-to-unitless-twip(@css:font-size) &gt;=
        letex:length-to-unitless-twip('12pt')]
```

```
    [letex:length-to-unitless-twip(@css:font-size) &lt;=
       letex:length-to-unitless-twip('20pt')]" priority="2">
     <xsl:copy>
        <xsl:attribute name="role" select="'header2'"/>
        <xsl:attribute name="layout-type" select="'para'"/>
        <xsl:attribute name="mapping-priority" select="'2'"/>
        <xsl:apply-templates select="@* except @role, node()"/>
     </xsl:copy>
  </xsl:template>
```

Das resultierende Stylesheet wird im nächsten Pipeline-Schritt dafür genutzt, die Transformation des Hub-Resultates durchzuführen. Darin werden die Zielformate namentlich als Referenz an die regelkonformen Absätze im Hub-Dokument übertragen. Das bedeutet, dass den *para*-Elementen ein neues *role*-Attribut hinzugefügt wird, welches als Wert den neuen Formatvorlagennamen besitzt. Zusätzlich werden Attribute hinzugefügt, die Auskunft über die Priorität der Regel und dem gefilterten Typ des Dokumentelementes geben. Anschließend werden im nächsten XSLT-Schritt, das im Anhang A.1 auf der Seite 80 zu finden ist, die leeren Absätze, deren Nachfolger einer Zuweisungsregel unterliegen und die wiederum als Eigenschaft *margin-top* (= oberer Zeilenabstand) besitzt, mit dem Attribut *delete* markiert. Dadurch wird sichergestellt, dass die Abstandskompensation nur erfolgt, wenn die Formatvorlagenzuweisung aktiv ist.

Das darauffolgende Template steht für Inline-Elemente, deren zu löschende Ad-Hoc-Formatierungen um die des umschließenden Absatzes erweitert wird. Dazu wird in diesem Template über das XSL-Element *call-template* ein Aufruf durchgeführt, der explizit das Template mit dem Namen *merge-remove-adhocs* anspricht. Dabei wird ihm als Parameter – sofern vorhanden– das *remove-adhoc*-Attribut des Absatzes und des jeweiligen Inline-Elementes übergeben, welche in einer Variable mit Hilfe der *tokenize()*-Funktion sequentiell separiert und doppelt vorkommende Format-Namen per *distinct-values()* entfernt werden.

In der nachfolgenden Transformation werden die markierten Dokumentelemente sowie die Ad-Hoc-Formatierungen gelöscht und die neuen Formatvorlagen-Namen, die den Hub-Elementen über die Zuweisungsregeln verliehen wurden, übergeben. Für diese Transformation werden das Hub-Resultat aus dem vorherigen XSLT-Schritt und das bereitstehende Quelldokument als Single Tree-Dokument aus dem ersten Konvertierungsschritt benötigt. Im

Quellcode 3.18 befinden sich zwei Variablen und zwei Key-Funktionen, die als Grundlage dafür dienen. Mit ihrer Hilfe kann der Referenz-Schlüssel, das srcpath-Attribut, realisiert werden, um die Dokumentelemente aus dem Hub- und Single Tree-Dokument in Beziehung zu bringen.

Quellcode 3.18 Integration der Vorlagen-ID: Variablen und Key-Funktionen

```
<xsl:variable name="hub-doc" select="collection()[2]" as="
    document-node()"/>
<xsl:variable name="source-dir-uri" as="xs:string"
   select="$hub-doc/dbk:hub/dbk:info/dbk:keywordset[@role='hub']/
      dbk:keyword[@role='source-dir-uri']"/>

<xsl:key name="hub-element-by-srcpath" match="*[@srcpath]" use="
    concat($source-dir-uri, @srcpath)"/>
<xsl:key name="marked-element-by-srcpath" match="*[@sm:action =
    'delete'][@srcpath]" use="concat($source-dir-uri, @srcpath)"
    />
```

Die erste Variable steht für das Hub-Dokument, welches am zweiten Eingangsport durch die XPath-Funktion *collection()* abgegriffen wird. Da die Dokumentelemente des Hub-Dokumentes im *srcpath*-Attribut –im Gegensatz zum Single Tree-Dokument– nur über relative Pfade verfügen, ist es notwendig ihn zu einem Absoluten zu erweitern. Dazu wird in der zweiten Variable *source-dir-uri* der Quellpfad des Dokumentes aus dem *keyword*-Element mit dem gleichnamigen *role*-Attributswert gespeichert. Anschließend wird die Key-Funktion *hub-element-by-srcpath* konfiguriert, sodass sämtliche Elemente mit dem *srcpath*-Attribut -bestehend aus einer Stringkombination aus Quellpfad und relativem Pfad- ausgewählt werden.

Eine zweite Key-Funktion sorgt dafür, dass die Elemente ausgewählt werden, die im vorherigen XSLT-Schritt mit dem *delete*-Attribut markiert wurden. Dabei wird ebenfalls der zusammengesetzte Pfad verwendet. Im nachfolgenden Template, welches im Quellcode 3.19 dargestellt ist, wird mit Hilfe der ersten Funktion den Absätzen eines WordML-Dokumentes die ID der neuen Formatvorlage übermittelt. Um das Dokument, in der die Key-Funktion wirken soll, eindeutig zu bestimmen, steht die Variable *$hub-doc* als dritter Parameter stellvertretend für das Hub-Dokument.

Quellcode 3.19 Integration der Vorlagen-ID: Template eines Absatzes

```
<xsl:template match="w:p[key('hub-element-by-srcpath', @srcpath,
    $hub-doc)[@layout-type = 'para']]" priority="2">
  <xsl:copy>
    <xsl:apply-templates select="@*"/>
    <w:pPr>
      <w:pStyle w:val="{key('hub-element-by-srcpath', @srcpath,
          $hub-doc)/@role}"/>
      <xsl:apply-templates select="w:pPr/* except w:pPr/w:pStyle
          "/>
    </w:pPr>
    <xsl:apply-templates select="* except w:pPr"/>
  </xsl:copy>
</xsl:template>
```

Dazu werden zunächst im <copy>-Element über den Aufruf *apply-templates* des Identity-Templates alle vorhandenen Attribute samt Absatzelement reproduziert. Im Anschluss wird ein XML-Gerüst erstellt, das die Absatzeigenschaften (*pPr*) repräsentiert. Darin wird die ID der Formatvorlage im *val*-Attribute des Elementes *pStyle* mit Hilfe des *role*-Attributes aus dem Hub-Dokument übergeben. Anschließend erfolgt die Reproduktion der restlichen Elemente dieses und des Elternelementes *p*. Für IDML-Dokumente beruht die ID-Zuweisung auf dem gleichen Prinzip. Der Unterschied dabei in der Baumstruktur der Dokumentelemente. Die Vorlagen-ID wird dabei im *AppliedParagraphStyle*-Attribut des Absatzelementes *ParagraphStyleRange* übergeben. Um eine Ad-Hoc-Formatierung zu löschen, wird zunächst über ein Template –dargestellt im Quellcode 3.20– auf den Bereich zugegriffen, der diese beinhaltet. In einem WordML-Dokument sind diese Formatierungen in einem den *run*-Eigenschaften *rPr* als attributbehaftete Kindelemente vertreten.

Quellcode 3.20 Integration der Vorlagen-ID: Template der *run*-Eigenschaften

```
<xsl:template match="w:p[@srcpath]//w:rPr">
  <xsl:copy>
    <xsl:apply-templates select="@*, node()">
      <xsl:with-param name="corresponding-hub-element" as="
          element(*)?"
        select="key('hub-element-by-srcpath', ancestor::*[2]/
          @srcpath, $hub-doc)" tunnel="yes"/>
    </xsl:apply-templates>
  </xsl:copy>
</xsl:template>
```

Das Prinzip beim Entfernen bestimmter Ad-Hoc-Formatierungen basiert zum einen auf dem Übersetzen der Formatierungen der Dokumentelemente aus dem Single Tree in die CSS-Syntax des Hub-Dokumentes. Zum anderen das Löschen der übersetzten Formatierungen, die über die Werte des *remove-adoc*-Attributes herausgefiltert werden. In der Praxis erfolgt ein allgemeiner Template-Aufruf, der als Parameter das Dokumentelement aus dem Absatz übergeben bekommt, um sein *remove-adhoc*-Attribut nutzen zu können. Die Pfadübergabe des *srcpath*-Attributes vom Absatzelement an die Key-Funktion erfolgt mit dem XPath-Ausdruck *ancestor::*[10], der es ermöglicht, übergeordnete Elementknoten auszuwählen. Dieser Parameter wird dem Template, welches im Quellcode 3.21 aufgeführt ist, für die Formatierungen eines *runs* mitgegeben.

Quellcode 3.21 Integration der Vorlagen-ID: Template der der *run*-Formatierungen

```
<xsl:template match="w:rFonts|w:color|w:sz|w:b..."> 
  <xsl:param name="corresponding-hub-element" as="element(*)?"
      tunnel="yes"/>
  <xsl:variable name="prop" as="xs:string">
    <xsl:apply-templates select="." mode="w2css"/>
  </xsl:variable>
  <xsl:if test="not(tokenize($corresponding-hub-element/
     @sm:remove-adhoc, '\s+') = ($prop, '#all'))">
    <xsl:next-match/>
  </xsl:if>
</xsl:template>
```

In der Variable *prop* werden die Templates für die einzelnen Formatierungen aufgerufen, die über das *sequence*-Element die CSS-Übersetzung zurückgeben. Durch das XSL-Element *if* wird eine Abfrage durchgeführt, die nach der Existenz der einzelnen Werte oder dem Schlüsselwert zum Löschen aller Ad-Hoc-Formatierungen prüft und bei Nichteintreten des Falles über das Identity-Template reproduziert bzw. erhalten bleibt. Ansonsten würde die Formatierung durch das momentane Template gelöscht werden, da es ein leeres Ergebnis herbeiführt. Im Pipeline-Verlauf steht das Resultat der letzten Transformation dem nächsten Schritt entgegen, bei dem der Ersatz der Formatvorlagen des Quelldokuments mit denen des Single Tree-Dokumentes erfolgt. Dafür wird im Vorfeld das Template-Dokument mit Hilfe des *docx2hub*- bzw. *idml2xml*-Moduls zu einem Single Tree-Dokument transformiert und

[10] ancestor, zu deutsch *Vorfahre*

das *srcpath*-Attribut des Formatvorlagen-Wurzelelements durch das des Äquivalents aus dem Single Tree des Quelldokumentes. In der Pipeline wird dabei ähnlich der *fileprocessor.xpl*-Pipline durch ein *choose*-Element nach den beiden Dokumentformaten IDML und WordML unterschieden, weil der Austausch über unterschiedliche XML-Strukturen erfolgt. Da für diesen Prozess der Pfad des Quelldokument-Containers benötigt wird, findet im Vorfeld der Fallunterscheidung die Deklaration der Variable *file-base-uri* mit dieser Information statt. Sie findet im *viewport*-Schritt, der im folgenden Quellcode 3.22 dargetstellt ist, nach der jeweiligen Single-Tree-Konvertierung des Templates Anwendung.

Quellcode 3.22 Austausch der Formatvorlagen: Initialisierung des Dokumentpfades

```
<p:viewport match="/w:root/*[@xml:base]" name="add-file-base">
  <p:add-attribute attribute-name="xml:base" match="/*[@xml:base
     ]">
    <p:with-option name="attribute-value" select="concat($file-
       base-uri, replace(/*/@xml:base, '^.+\.tmp/', ''))"/>
  </p:add-attribute>
</p:viewport>
```

Mit Hilfe dieses Schrittes ist es möglich alle XML-Module des Template-Dokument-Containers bzw. Kindelemente des Wurzelelementes im Single-Tree-Dokument, die über ein *base*-Attribut verfügen , isoliert zu betrachten. Ihnen wird per *add-attribut*-Element ein neu erzeugtes Attribut mit selbem Namen angefügt, das zur Überschreibung des alten Quellpfadattributes führt. Dabei besteht der neue Pfad aus der Kombination des Quellpfades der *file-base-uri*-Variable und der relativen Pfad der XML-Module, der aus dem absoluten Pfad des *base*-Attributes extrahiert wurde. Im Anschluss folgt der XProc-Schritt *replace*, der mit dem Ersetzen des *styles*-Elementes für WordML-Dokumente bzw. *Styles* für IDML-Dokumente durch die äquavalenten Elemente mit zuvor modifizierten Pfadattributen den Austausch der Formatvorlagen im Single-Tree-Dokument durchführt. Im nachfolgenden *viewport*-Schritt werden das Wurzelelement der Formatvorlagen und der bearbeitete Dokument-Inhalt ausgewählt und mit dem XSL-Elemente *store* über das Quellpfadattribut als XML-Dokument gespeichert, welches die ursprünglichen XML-Module im Container überschreibt. Der letzte Schritt der Formatvorlagezuweisung im Backend stellt die Verpackung des Quelldokument-

Ordners zu einem ZIP-Container mit adäquater Dateierweiterung dar. Der XProc-Schritt *zip* stammt aus der Calabash-Bibliothek für Erweiterungen und ist im Quellcode 3.23 dargestellt.

Quellcode 3.23 Verpackung zum ZIP-Container

```
<cx:zip compression-method="deflated" compression-level="default
   " command="create" name="zip" cx:depends-on="store-modified">
   <p:with-option name="href" select="replace($file-base-uri, '(
      docx|idml)\.tmp/?.+', 'mod.$1')"/>
   <p:input port="source">
     <p:empty/>
   </p:input>
   <p:input port="manifest">
     <p:pipe port="zip-manifest" step="stylemapper"/>
   </p:input>
</cx:zip>
```

Dieser Schritt benötigt neben dem Single-Tree-Dokument das ZIP-Manifest, um die ursprüngliche Verzeichnisstruktur innerhalb des Containers beizubehalten. Der Dateiname des Containers wird dahingehend modifiziert, damit die formatspezifische Dateierweiterung *.docx* bzw. *.idml* sowie die temporäre *.tmp*-Erweiterung entfernt werden kann. In selben Atemzug werden diese durch die Erweiterung *.mod* [11] und durch die jeweilige formatspezifische Dateierweiterung ersetzt. Nach Durchührung dieses Schrittes liegt das Dokument im ursprünglichen Container-Format vor, womit die Vorlagenzuweisung beendet ist. Die *stylemapper*-Pipeline empfängt am Ausgang das Ergebnis des Verpackungsschrittes in Form einer XML-Struktur, die Auskunft über den Container hinsichtlich des Verarbeitungsdatums und der einzelnen Dateipfade gibt. Er wird darin durch das Wurzelelement *zipfile* und den verarbeiteten Dateien, die jeweils durch das Kindelement *file* repräsentiert sind, dargestellt.

3.3 Implementierung des Workflows im Frontend

3.3.1 Konzeption und Layout der Web-Anwendung

Die folgenden zwei Abschnitte beschäftigen sich mit dem Aufbau der Anwendungsoberfläche sowie ihrer Funktionsweise. Zum einen werden in diesem Abschnitt das Layout,

[11] Abkürzung für *Modifikation*

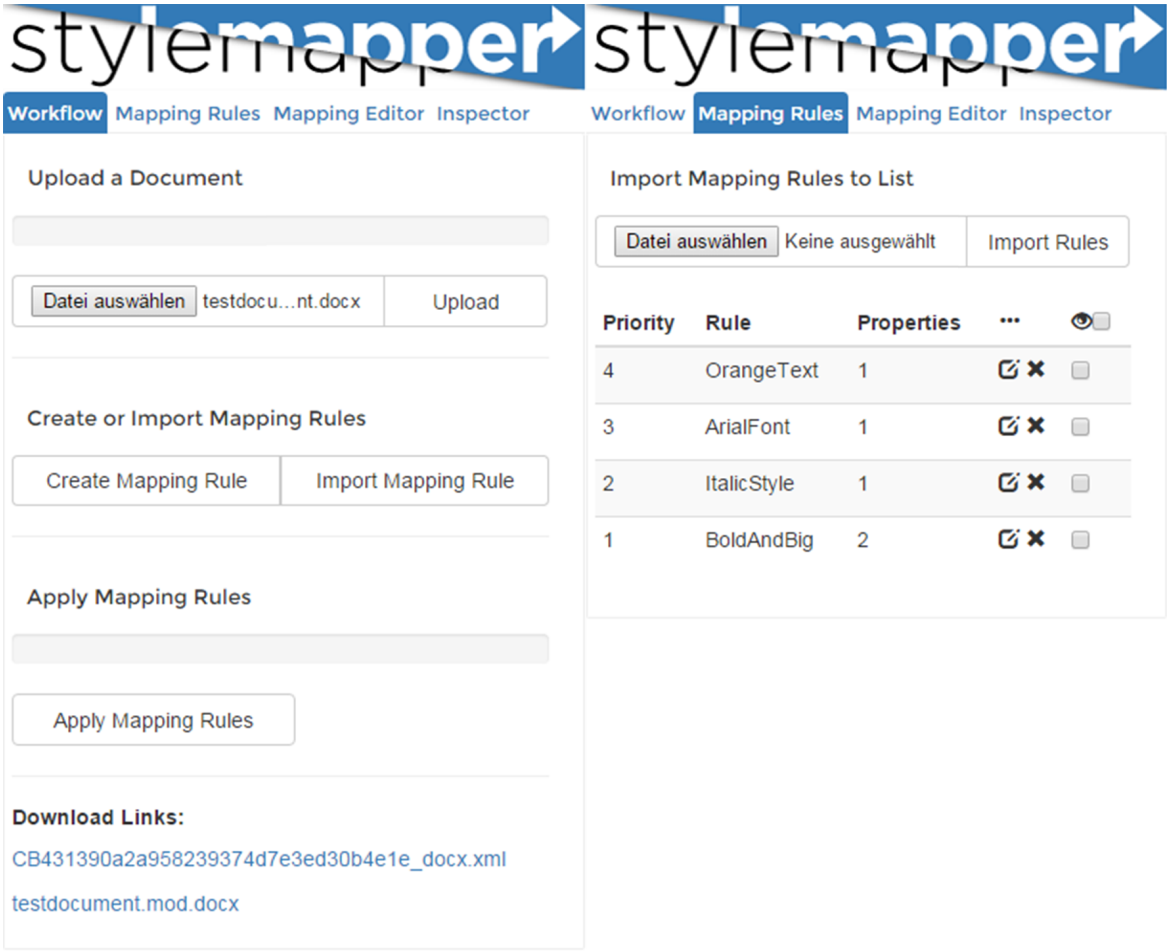
sowie das Anwendungskonzept unter Verwendung von HTML und CSS und zum anderen ausgewählte JavaScript-Mechanismen näher erläutert, die das funktionale Fundament bilden. Zur Gestaltung von Buttons, Listen, Containern und zur Realisierung interaktiver Elemente, wie beispielsweise Fortschrittsbalken, wurde das Framework Bootstrap v3.3.5 verwendet[12]. Die Abbildung 3.4 zeigt eine schematische Darstellung des Aufbaus. Die Oberfläche gestaltet

Abbildung 3.4 Layout der Browser-Anwendung

sich im Wesentlichen aus dem Platzhalter der HMTL-Konvertierung des Quelldokumentes, der –angefangen von der linken Seite– das Dreiviertel des Stylemappers umfasst und dem verbleibenden Drittel, welches auf der rechten Seite den Bereich für das Logo, die Menüpunkte und deren Inhalte bereithält. Zu den Menüpunkten zählen –wie bereits im Abschnitt 2.4.1 erwähnt– *Worflow*, *Mapping Rules*, *Mapping Editor* und *Inspector*. Der Workflow-Bereich stellt dem Benutzer den Handlungsablauf zur Verfügung, der gleichzeitig beiden Upload-Schritte für die Prozesskette im Backend und Verweise zu den übrigen Funktionen der anderen Abschnitte aufzeigt. In der Abbildung 3.5 sind die Inhalte der ersten beiden Menüpunkte dargestellt. Zunächst sind auf der linken Seite im Bereich *Workflow* die 3 Schritte zur Durchführung einer Vorlagenzuweisung aufgelistet. Sie tragen als Titel jeweils die erforderlichen Handlungsanweisungen *Upload a Document*, *Create or Import Mapping Rules* und *Apply Mapping Rules* die aus dem englischen ins deutsche übersetzt *Lade ein Dokument hoch*, *Erstelle oder Importiere eine Zuweisungsregel* und *Wende die Zuweisungsregel an*. Für den ersten Schritt ist durch Formular-Element *form* und dem Kindelement *input* eine Dateiauswahl realisiert, deren Zieldatei per *submit*-Button über einen HTTP-Request hochgeladen werden kann. Als Feedback liegt darüber ein Fortschrittsbalken, der die Statusabfragen des Konvertierungsprozesses in Form prozentualer Anteile darstellt.

[12] Vgl.: [31], Abschnitt *What is Bootstrap?*, Absatz 1

Abbildung 3.5 Browser-Anwendung: Menüpunkte *Workflow* und *Mapping Rules*

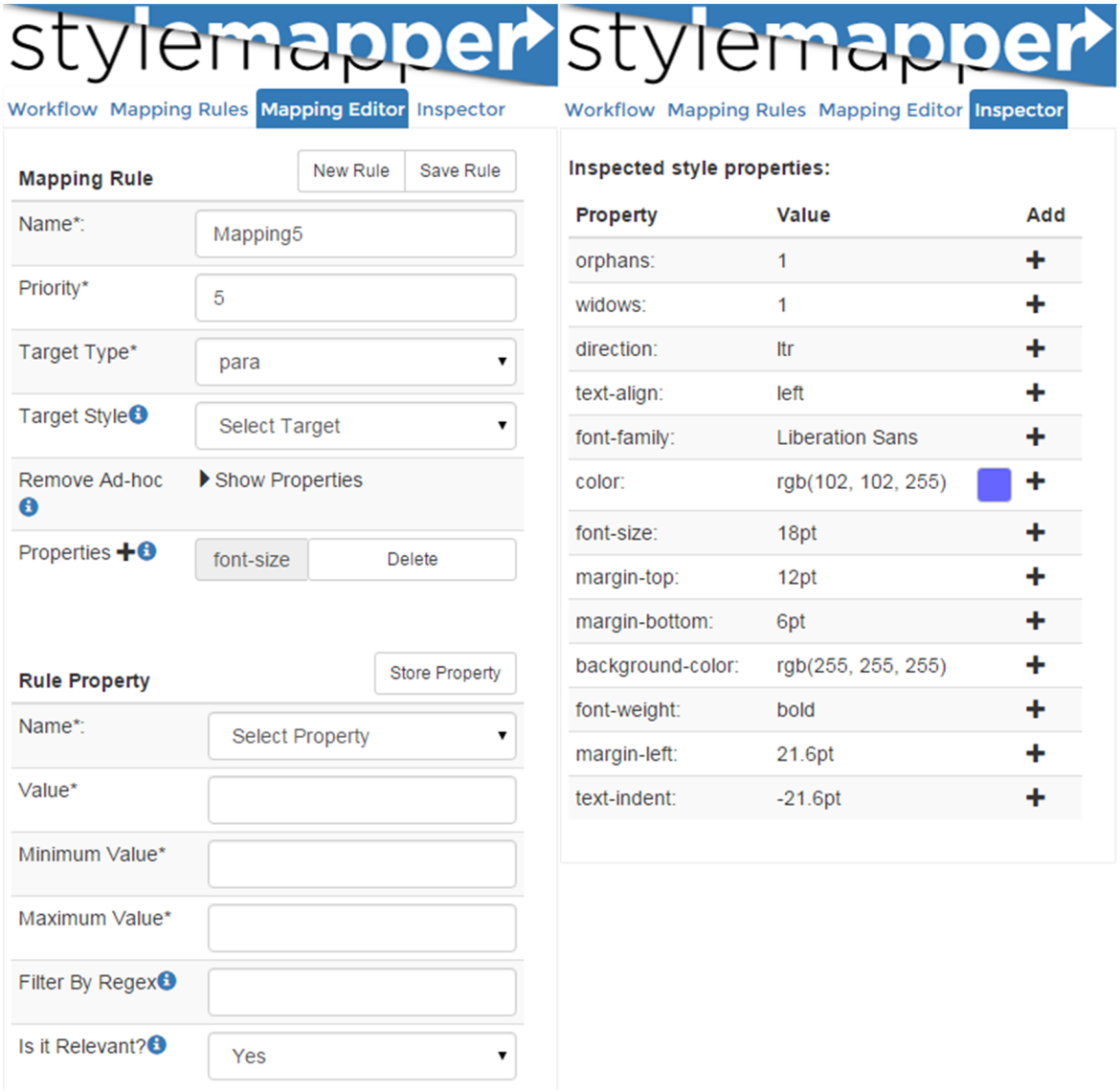


Der zweite Schritt gibt dem Nutzer die Auswahl die Zuweisungsregeln zu erstellen oder zu importieren. Über die entsprechenden Buttons wird er zu den Punkten *Mapping Editor* oder *Mapping Rules*-Punkt verwiesen. Innerhalb des dritten Schrittes wird mit Hilfe eines Buttons ein HTTP-Request durchgeführt, der die Knotenstruktur der Zuweisungsregel in Form eines XML-Dokumentes an den transpect-Server hochlädt und den zweiten und finalen Schritt zur Konvertierung im Backend auslöst. Die Rückgabe der Ergebnisse erfolgt über angeforderte Links, die ebenfalls im *Worflow*-Inhalt generiert werden. Um dem Nutzer einen reibungslosen Arbeitsablauf zu ermöglichen und die Bedienfreundlichkeit zu erhöhen, wurde auf eine sukzessive Funktionserweiterung gesetzt. Das bedeutet, das nach dem Aufruf des Stylemappers die 3 Menüpunkte nach *Workflow*, sowie all seine Teilinhalte außer dem Upload-Formular ausgegraut und unwirksam sind. Erst nach dem ein Dokument hochgeladen, konvertiert und

als HTML-Ansicht erschienen ist, werden der zweite Teilschritt und die anderen Menüpunkte vollständig sicht- und nutzbar. Der dritte Teilschritt wird erst dann zu Verfügung stehen, wenn mindestens eine Zuweisungsregel in der Liste existiert. Auf der rechten Seite ist der Menüpunkt *Mapping Rules* ausgewählt und dessen Inhalt zu sehen. Hier hat der Nutzer die Möglichkeit Regeln zu importieren, die –auch über den Editor erstellte Regeln– in einer Liste dargestellt werden. Sie besitzt eine tabellarische Struktur, bestehend aus den Spalten für die Priorität, den Regelnamen, die Anzahl der Regel-Eigenschaften, die Bearbeitungs- und Lösch-Operation, sowie die Vorschaufunktion. Per Drag&Drop können die Regeleinträge vertikal verschoben werden und dadurch ihre Priorität ändern. Die Bearbeitungs- und Lösch-Funktion ist über das Anklicken des Blatt-Stift-Symbols bzw. des Kreuzchen-Symbols aus dem Bootstrap-Repertoire ausführbar. Zum Bearbeiten gelangt man zum Menüpunkt *Mapping Editor* mit dem die Informationen der jeweiligen Zuweisungsregel in das Formular übertragen wird. Zur Betrachtung der Zuweisungsregel erscheint via MouseOver-Ereignis ein Container im Footer-Bereich, der die Namen der Eigenschaften beinhaltet. Um für eine oder mehrere Zuweisungsregeln eine Vorschau über die konformen Dokumentelemente zu erlangen muss die jeweilige Checkbox in der letzten Spalte aktiviert werden. Für eine Gesamtauswahl gilt die Checkbox neben dem Augen-Symbol in der Kopfzeile. Die beiden Menü-Inhalte für den *Mapping Editor* und *Inspector* sind in der Abbildung 3.6 aufgeführt. Der Inhalt des Mapping Editors und Inspectors sind stets Veränderungen ausgesetzt. Deshalb werden sie per JavaScript dynamisch erstellt. Im Mapping Editor wird zunächst ein Formular gestaltet, welches über Tabellen- und Input-Elemente realisiert wurde. Da es nicht für Server-Anfragen verwendet wird kann deshalb auf das *form*-Element verzichtet werden. Die Eingabefelder spiegeln den Aufbau einer Zuweisungsregel wieder und sind, sofern sie über eine *-Markierung verfügen, zur Regelerstellung verpflichtend. Über den Button *New Rule* steht das Formular dem Nutzer zur Verfügung, welches nach Betätigung einen vorgefertigten Namen erzeugt. Dieser setzt sich aus dem Wort *Mapping*[13] und einer Nummer, die sich aus der vorhandenen Regelanzahl um eins addiert ergibt, zusammen. Die Priorität wird ebenfalls automatisch anhand der Anzahl generiert und lässt nach der Erstellung die

[13] mapping, zu deutsch *Zuordnung* – im Kontext verwendete Kurzform einer Zuweisungsregel

Abbildung 3.6 Browser-Anwendung: Menüpunkte *Mapping Editor* und *Inspector*



Zuweisungsregel gegenüber allen vorhandenen Regeln im Konkurrenzfall durchsetzen. Der Zielvorlagentyp und die Zielvorlage aus dem Template-Dokument lassen sich per selektiver Auswahl über das *select*-Element auswählen. Die zu löschenden Ad-Hoc-Formatierungen sind über Checkboxen auswählbar, die durch das Anklicken des DropDown-Menüpunktes *Show Properties* sichtbar werden. Die Auswahlmöglichkeiten ergeben sich dabei aus den im Dokument-vorhandenen Formatierungen. Im Feld *Properties* sind die bisherig angehängten Eigenschaften dargestellt. Deren Inhalte sind ebenfalls per MouseOver-Ereignis anzusehen

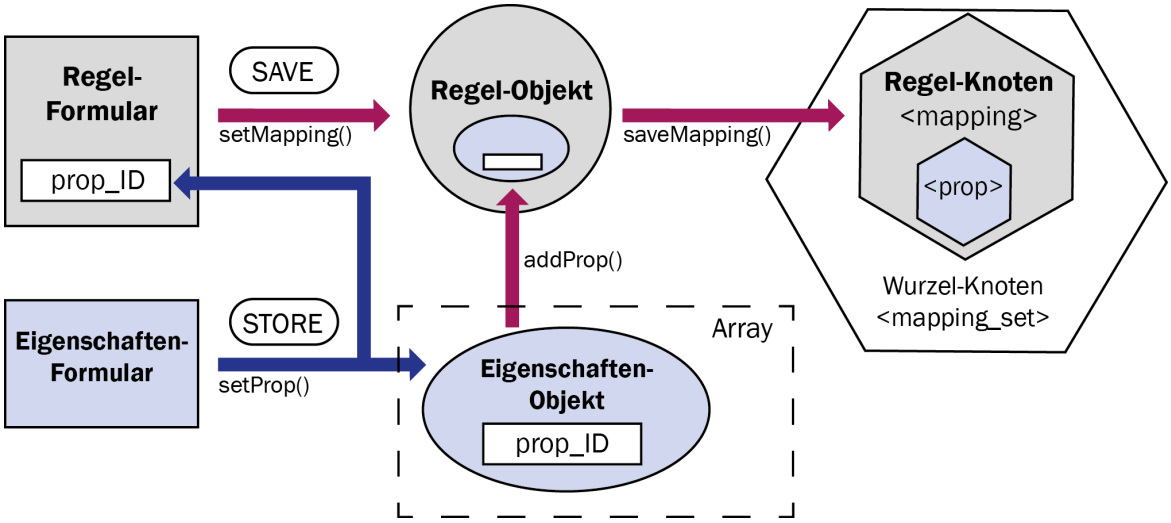
und per *Delete*-Button entfernbar. Über einen Mausklick auf das Plus-Symbol wird das Formular um die Felder zum Erstellen einer Eigenschaft erweitert. Diese Felder setzen sich ebenfalls aus ihren *prop*-Attributen zusammen. Der Name wird über ein Auswahlmenü konfiguriert, dessen Repertoire sich aus den Formatierungen zusammensetzt, die auch für das *remove*-Adhoc-Element eingesetzt werden. Die Eintragung der Eigenschaftswerte, mit Ausnahme von Farben, verläuft über Texteingabefelder. Sie stehen in Beziehung zueinander, die dafür sorgt, dass entweder ein absoluter Wert(*Value*), ein Wertebereich, oder ein regulärer Ausdruck eingegeben werden kann. Dazu wird über einen EventListener erfasst, ob sich innerhalb des jeweiligen Eingabefeldes eine Zeichenkette befindet. In diesem Fall würden alle Anderen Eingabefelder inaktiv geschaltet und ausgegraut. Nach Löschen dieser Zeichenkette stehen wieder alle Optionen zu Werteeingabe offen. Zuletzt erfolgt die Relevanz-Angabe über ein weiteres Auswahlmenü. Mit Hilfe des *Store Property*-Buttons wird die Eigenschaft der Zuweisungsregel beigegeben. Über den *Save Button* wird diese letztendlich gespeichert und zur Liste im Bereich *Mapping Rules* hinzugefügt. Im Bereich *Inspector* werden die Formatierungen des Dokumentelements angezeigt, die per Mausklick in die HTML-Konvertierung des Quelldokumentes inspiziert werden können. In einer tabellarischen Übersicht sind diese angeordnet und lassen ihren Namen und Wert per Plus-Symbol in der dritten Spalte als Eigenschaft einer Zuweisungsregel zuweisen. Dabei wird der Mapping Editor geöffnet und die beiden Attribute in die passenden Eingabefelder übernommen.

3.3.2 Umsetzung der funktionaler Grundanforderungen mit JavaScript

Speicherung und Bearbeitung von Zuweisungsregeln

Dieser Abschnitt umfasst die Speicherung und Bearbeitung von Zuweisungsregeln auf der Client-Seite, die für den zweiten Schritt des Backend-Workflows benötigt wird. In der Abbildung 3.7 auf der Seite 65 ist der Speichervorgang dargestellt, der mit Hilfe einer Kombination aus mehreren JavaScript-Funktionen aus der *stylemapper.js* im Ordner *html* umgesetzt wird. Zunächst trägt der Nutzer die notwendigen Informationen in die Formularfelder ein. Um eine Eigenschaft hinzuzufügen, wird das Eigenschaften-Formular aufgerufen, ausgefüllt und der

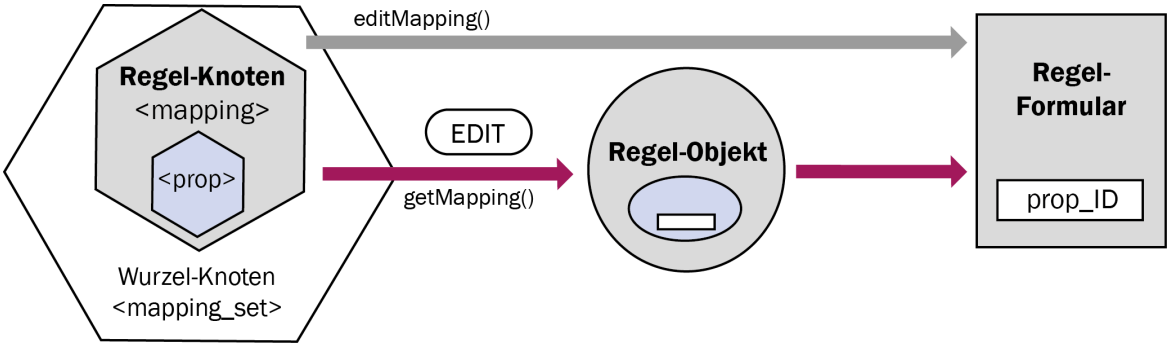
Abbildung 3.7 Speicherung einer Zuweisungsregel



STORE-Button gedrückt. Über die Funktion *setProp()* wird daraus ein Objekt erstellt, das in einem Array gelagert und deren ID im Formular hinterlegt wird. Die folgenden Funktionen sind im Anhang C vollständig dargestellt. Dabei wird ein neues Objekt aus der im Vorfeld erstellten *Prop*-Klasse erzeugt, welches über alle Attribute verfügt, die in der Tabelle 2.1 des Abschnittes 2.4.2 auf der Seite 33 aufgeführt sind. Ihnen werden die Werte aus den Eingabefeldern mit Hilfe der JQuery-Methoden *$()* und *val()* übergeben. Über die erste Methode wird durch einen ID-Selektor das Eingabefeld identifiziert und mit der zweiten auf den Wert zugegriffen. Falls es sich bei der Eigenschaft um eine Farbe handelt, werden werden die jeweiligen Farbwerte mit einer Konvertierungsfunktion in das HSL-Format umgewandelt und die Einzelwerte in die passenden Attribute gespeichert. Des Weiteren wird über die Funktion *guid()* eine ID erzeugt, die dem Objekt als zusätzliche Eigenschaft übergeben wird. Darauf erfolgt die Erzeugung eines Listeneintrages, welches den Namen und einen ID- referenzierten Hyperlink für den Löschvorgang beinhaltet. Das Eigenschaften-Objekt wird abschließend in ein globales Array gelegt, was als Zwischenspeicher für sämtliche Eigenschaften dient. Betätigt der Nutzer nun den *SAVE*-Button, wird die Funktion *setMapping()* ausgelöst. Darüber wird, wie bei den Eigenschaften, aus dem Regel-Formularinhalt ein Objekt erzeugt. Anschließend wird dem Array-Attribut *props* des Objektes mit Hilfe der Funktion *addProps()* die Eigenschaften-Objekte angefügt, indem die Listenelemente

per JQuery-Methode *find()* in einem Array zusammengefasst und anhand der hinterlegten Eigenschaften-IDs identifiziert werden. Das vollständige Regel-Objekt wird nun der JavaScript-Funktion *saveMapping()* übergeben, um das Informationskonstrukt als Knoten im DOM zu integrieren. Neben dem Regel-Objekt befindet sich der Parameter *override*. Dabei handelt es sich um einen Wahrheitswert dem die Funktion *checkMapping()* übergeben wird, durch die eine Überschreibung stattfinden kann. Die Gelegenheit eine Überschreibung einzuleiten bietet sich während des Speicherprozesses im Falle der gleichnamigen Präsenz einer Zuweisungsregel und einer Bestätigung durch den Nutzer. Im Vorfeld des Funktionsaufrufes wurde gemäß des XML-Aufbaus der Zuweisungsregeln das Wurzelelement *mapping_set* über die Methode *createElement()* im DOM erzeugt, welches in der globalen Variable referenziert ist. Um darin weiteres Mapping-Element mit den dazugehörigen Eigenschaft-Elementen zu speichern, wurde in der Funktion ebenfalls diese Methode verwendet. Deren Attribute werden jeweils in neuen Attributknoten gespeichert, die per *setAttribute()* erzeugt und an die Elemenknoten angefügt werden. Bei der Bearbeitung einer Zuweisungsregel erfolgt der Weg umgekehrt, wie in der Abbildung 3.8 schematisch dargestellt ist. Klickt der Nutzer auf

Abbildung 3.8 Bearbeitung einer Zuweisungsregel



das in einem *span*-Element integrierten Stift-Dokument-Symbol, welches in der Abbildung als *EDIT*-Button vertreten ist, wird die Funktion *editMapping()* aufgerufen. Dazu wird ihr der Name der Zuweisungsregel über das *name*-Attribut des *span*-Elementes als Parameter übergeben. Die Zuweisungsregel wird mit Hilfe der Funktion *getMapping()*, die ebenfalls den Namen übergeben bekommt, als Regel-Objekt repräsentiert, welches die Information aus dem Regel-Elementknoten *mapping* bezieht. Über die Funktion *createMappingTable()* wird

die HTML-Struktur für das Formular erzeugt, worin die Information an die Eingabefelder sowie die Eigenschaften-ID in neu erzeugten Listeneinträgen übergeben werden.

Hochladung von Dokumenten und Zuweisungsregeln zum transpect-Server

Das Hochladen von Dateien über die Web-Anwendung erfolgt über eine POST-Anfrage. Diese wird mit Hilfe der Ajax-Technologie realisiert, die über die JQuery-Methode *$.ajax()* genutzt wird. Ajax bedeutet *Asynchones JavaScript und XML* und bietet den Vorteil, statt einer regulären HTTP-Anfrage, die ein neues Rendern der gesamten Oberfläche zur Folge hätte, über JavaScript-Mechanismen mit dem Server im Hintergrund zu kommunizieren. Dadurch kann ein Datenaustausch zwischen Client und Server geschehen und sich clientseitiger Inhalt dynamisch ändern, ohne dass ein Neuladen der HTML-Oberfläche erfolgen muss.[14] Dazu wird in einem HTML-Formular über ein *input*-Element ein Dokument referenziert, welches der Nutzer über den Datei-Explorer auswählen muss. Mittels der *submit()*-Methode, die beim Betätigen des Buttons innerhalb des Formulares, aufgerufen wird, bereitet eine anonyme Funktion das Informationspaket vor der Absendung vor und ruft anschließend die *ajax()*-Methode auf. Im Quellcode 3.24 ist die *submit*-Methode aufgeführt.

Quellcode 3.24 Absenden eines Formulares

```
$("form#docxfile").submit(function(evt){
      var filename = document.getElementById('upload-doc').value.
         replace(/^.*\\/, "");
    formData = new FormData($(this)[0]);
    formData.append('type','stylemapper');
    $.ajax({
      url: 'https://transpect.le-tex.de/api/upload_file',
      type: 'POST',
      data: formData,
      async:true,
      success: function (data){
          var callbackURI = data['callback_uri'];
          initDocxStatusRequest(callbackURI);
      },
      "beforeSend": function(xhr){
          xhr.setRequestHeader("Authorization",
             basicHTTPAuthString(username, password));
      },
```

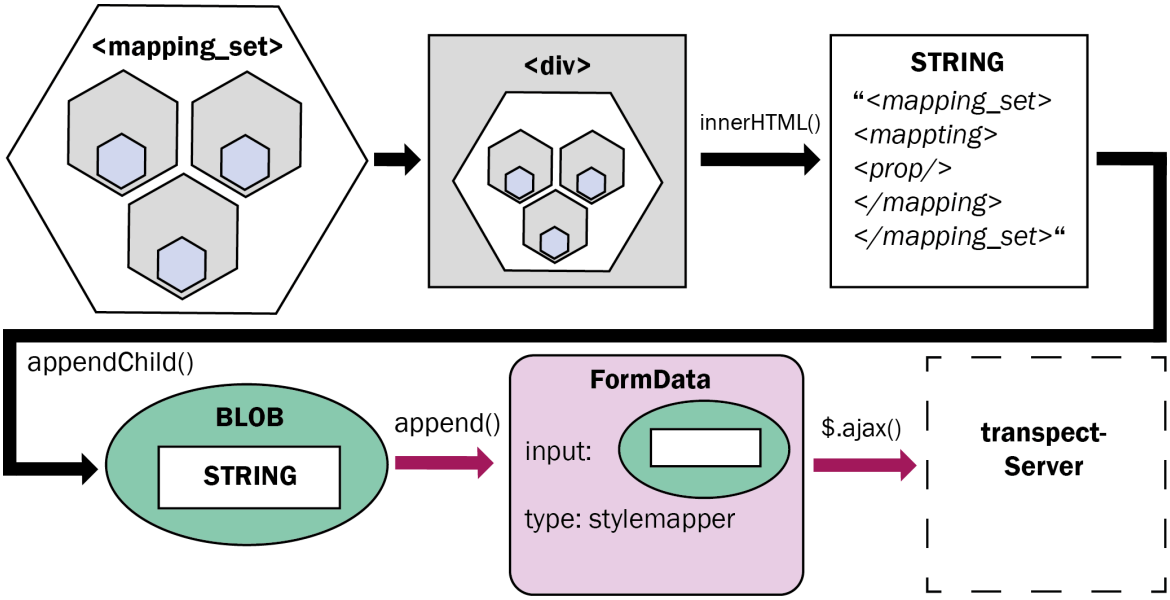
[14] Vgl.: [19], Kapitel 1: Einführung in Ajax, Abschnitt *Das XHttpRequest-Object*

```
        cache: false,
        contentType:false,
        processData:false
    })
    return false
})
```

Zunächst wird in der Variable *filename* der Dateiname gespeichert, der aus dem beschneiden des Dateipfades aus dem *input*-Element mit Hilfe der *replace()*-Funktion resultiert. Dieser wird für die Zielpfade für den Bezug der HTML-Konvertierung und der Zielvorlagennamen benötigt. In der *formData*-Variable wird ein *FormData*-Objekt erzeugt, dass den Inhalt des Formulars per *$(this)[0]* übergibt. Des Weiteren wird ihm der *type*-Parameter mit dem Wert *stylemapper* über die Methode *append()* mitgegeben, der gemäß der Konverter-API eine Voraussetzung für die Kommunikation für den transpect-Server ist. Im Anschluss folgt der eigentliche Ajax-Aufruf *ajax()*. Darin liegen die URL für den Datei-Upload des Konverters, der Anfragentyp *POST* und der Formularinhalt *formData* eingebunden im Parameter *data*. Über den *success*-Parameter kann eine Callback-Funktion über einen anonymen Funkionsaufruf realisiert werden, deren Rückgabe-Information im *data*-Objekt mitgeliefert wird. Die URI für die Statusabfragen werden über den Schlüssel *callback_uri* bezogen und in der gleichnamigen Variable platziert. Diese Adressierung wird benötigt, um Statusabfragen in Form weiterer Ajax-Anfragen über die Funktion *initDocxStatusRequests* durchzuführen. Der Name bezieht sich jedoch nicht nur auf die Kompatibilität mit einem Dokument-Format, sondern neben WordML auch auf IDML-Datein und ist somit entwicklungshistorischer Natur. Letzter relevanter Parameter ist *beforeSend*. Die Konverter-API setzt bei jeder Anfrage eine Nutzerauthentifizierung in der Form eines identischen Paares von dem Nutzernamen und dem Passwort voraus. Über diesen Parameter wird im Vorfeld der eigentlichen Anfrage eine weitere über ein XHttpRequest-Objekt(xhr) ausgeführt, dem die Authentifizierungsinformationen über die Methode *setRequestHeader* in den Nachrichtenkopf übergeben wird. Nachdem diese Anfrage vom transpect-Server erfolgreich entgegen genommen wurde, erfolgt das Hochladen des Dokumentes über die POST-Anfrage. Das Hochladen der Zuweisungsvorlagen geschieht im Prinzip ebenfalls über eine POST-Anfrage. Jedoch muss im Vorfeld das Datenpaket für die im DOM gespeicherte Baumstruktur geschnürt werden. In der Abbildung ist der

Workflow für den Upload-Prozess dargetstellt. Im Quellcode C.6 des Anhanges C auf Seite

Abbildung 3.9 Hochladung einer Zuweisungsregel



90 ist zusätzlich die Funktion *sendMapping()* sichtbar, welche den Arbeitsablauf des Hochladens realisiert. Zunächst wird das Wurzelelement *mapping-set* mit den darin befindlichen Zuweisungsregeln in einen *div*-Container gelegt, der mit Hilfe der *innerHTML()*-Methode den Inhalt des Containers zu einem String verarbeitet. Dieser wird gemäß seiner Syntax als Element eines Arrays einem neu erzeugten Blob-Objekt übergeben, das eine Pseudo-Datei repräsentiert, die Rohdaten speichert[15] und an den *input*-Parameter des FormData-Objektes angehängt wird. Da dieses Datenpaket noch über keinen Dateinamen verfügt, muss dieser als dritter Parameter der *append()*-Methode bestimmt werden. Wie bereits im Abschnitt 3.2.2 angegeben ist, muss der Dateiname aus der Referenz-ID *episode* bestehen, die in die gleichnamige Variable gespeichert wird. Sie ist als *meta*-Element der HTML-Konvertierung des Quelldokumentes mit dem *name*-Attributswert *episode* hinterlegt. Zusammen mit der Dateiendung *.xml*, die als String an die Referenz-ID angefügt wurde, bildet diese Kombination den Dateinamen. Die weitere Einrichtung des FormData-Objektes und die Ajax-Anfrage gestaltet sich nach dem gleichen Inhalt des vorherigen Dokument-Uploads.

[15] Vgl.: [1], Kapitel 1: *Introduction*, Absatz 2

Inspektion von Dokumentelementen

Nachdem das Quelldokument hochgeladen und die HTML-Konvertierung als Teilresultat in die HTML-Oberfläche integriert wurde, wird über den Aufruf der *initStyleInspector()*-Funktion das Inspizieren der Dokumentelemente initialisiert. Diese Funktion ist im Quellcode 3.25 dargestellt.

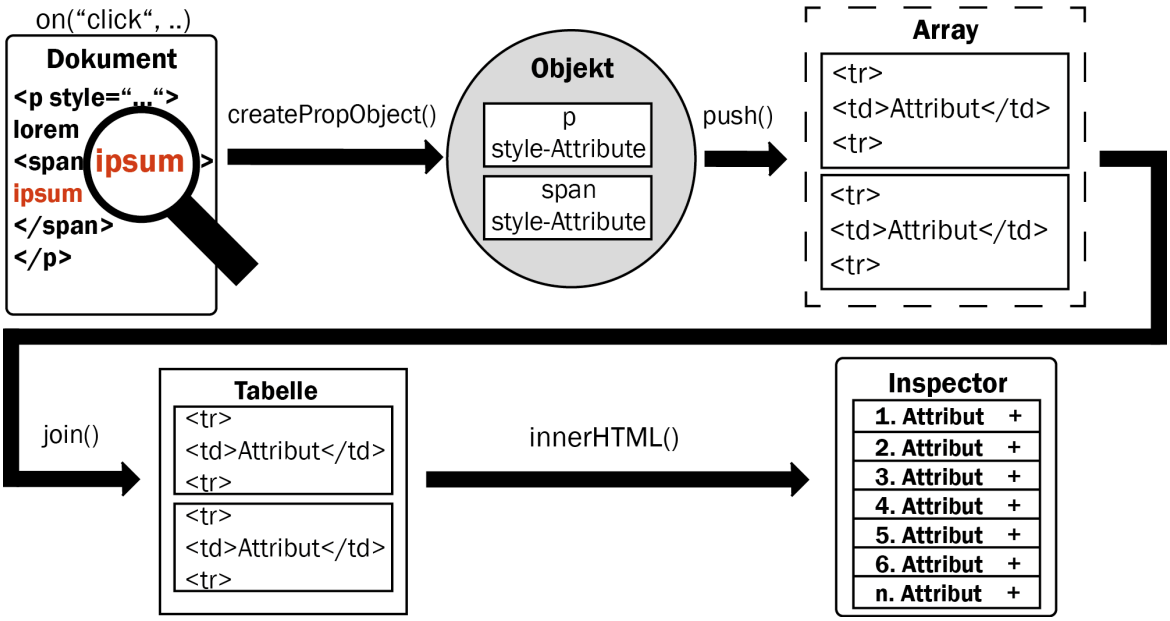
Quellcode 3.25 Initialisierung der Inspektion

```
function initInspector(){
    $('#sm-page > *').on('click',function(event) {
        viewTarget(event.target);
    })
}
```

Über den Wildcard-Selektor * werden alle Kindelemente des *div*-Containers mit der ID *sm-page*, der die HTML-Konvertierung des Quelldokumentes beinhaltet, ausgewählt. Durch die Methode *on* mit dem Parameter *click* wird Eventlistener für ein Mausklick-Ereignis auf das JQuery-Objekt angewandt. Dieses Ereignis ruft eine anonyme Funktion auf, die wiederum einen weiteren Funktionsaufruf von *viewTarget* zur Folge hat. In der folgenden Abbildung 3.10 ist nach dem $().on("click"..)-Symbol der Aufbau der Funktion schematisch aufgeführt.

Zunächst wird das Zielelement, welches mit dem Mauszeiger angeklickt wurde über die Methode *target* aus dem *event*-Objekt bezogen und der *viewTarget*-Funktion übergeben. Zu Beginn dieser Funktion findet ein weiterer Funktionsaufruf von *createPropObject()* statt, worüber die CSS-Eigenschaften des aktuellen Elements sowie die der Elternelemente für Inline(*span*)- oder Absatz(*p*)-Elemente als Eigenschaften eines gemeinsamen Objektes gespeichert werden. Diese CSS-Eigenschaften liegen in Form von Schlüssel-Wert-Paaren im *style*-Attribut der Dokumentelemente vor. Anschließend erfolgt eine iterative Verarbeitung der Objekt-Eigenschaften in einzelne Zeichenketten, die Tabellenzeilenstrukturen darstellen, welche ebenfalls ein *span*-Element mit einem Plus-Symbol für das Hinzufügen zum Mapping Editor beinhalten. Diese Strings werden mit Hilfe der *push()*-Methode in ein neu erzeugtes Array gesetzt. Danach erfolgt die Erzeugung eines weiteren Strings, der das Gerüst einer HTML-Tabelle beinhaltet, worin über die Methode *join()* alle Elemente hintereinander als

Abbildung 3.10 Inspektion eines Dokumentelementes



ein gemeinsamer String ausgegeben und mit ihm zusammengefasst werden. Dieser String wird über die Methode *innerHTML()* dem *div*-Container des Inspektor-Inhaltes übertragen und vom Parser als eine HTML-Tabelle mit Informationen über die CSS Eigenschaften des angeklickten Dokumentelementes gelesen.

Vorschau von regelkonformen Absätzen

Die Vorschau der greifenden Zuweisungsregeln im HTML-Dokument wurde im Prototyp zunächst über die Absätze realisiert. Durch die Aktivierung der jeweiligen Checkbox des Listeneintrags, der im Bereich *Mapping Rules* zu sehen und in der Abbildung 3.5 auf der Seite 61 dargestellt ist, wird eine Funktion aufgerufen, die den Algorithmus für eine farbliche Markierung der Absätze bewirkt. Dazu werden als Suchkriterien CSS-Klassennamen verwendet. Mit Hilfe der JQuery-Methode *find()* können die Eigenschaft-Namen und Werte in einer Kombination zu einem Klassennamen zusammengesetzt und aneinandergereiht als Selektor genutzt werden. Als Beispiel dafür wäre die Kombination einer Schriftgröße von 12pt der Klassenname *fs_12*. Die Voraussetzung für diese Auswahlmethode ist jedoch, dass die Stringkombination der einzelnen Formatierungen als CSS-Klassenreferenzen bereits

in den Dokumentelementen bzw. den Absätzen vorhanden sind. Um das zu gewährleisten, musste in den HTML-Konvertierungsprozess mit Hilfe einer Stylesheet-Überschreibung, die bereits im Abschnitt 3.2.1 auf der Seite 36 erwähnt ist, eingegriffen werden.
Zurück zum Algorithmus: Die Generierung der Klassennamen erfolgt nur bei dem regulären Ausdruck und dem absoluten Werten bzw. Farbwerten einer Regel-Eigenschaft. Ausgeschlossen ist davon der Wertebereich, da dieser separat geprüft werden muss. Nach der Erzeugung der Klassennamen, werden die darauf passenden Absätze in einem Array gespeichert. Im Anschluss erfolgt eine Bedingung, die prüft, ob innerhalb einer Zuweisungsregel Eigenschaften existieren, die einen Wertebereich umfassen. Ist das der Fall, erfolgt ein weiterer Funktionsaufruf, der einen Vergleich der Formatierungswerte mit den Extremwerten der Regel-Eigenschaften iterativ über die Absatzelemente vollzieht. Sofern alle Regel-betreffenden Formatierungen im zuständigen Wertebereich liegen, wird der jeweilige Absatz in einem neuen Array gespeichert und als Ergebnis zurückgegeben und steht dem Visualisierungsprozess zur Verfügung. Falls die Zuweisungsregel jedoch frei von Wertebereichen ist, werden die bereits herausgefilterten Elemente aus dem HTML-Dokument für die Visualisierungen verwendet. Die Markierung der Absätze im HTML-Dokument erfolgt anschließend über farblich transparente *div*-Container iterativ über das Array erzeugt werden. Deren Position sowie deren Breite und Höhe ergibt sich aus den äquivalenten CSS-Eigenschaften der Absätze.

Kapitel 4

Auswertung

4.1 Ergebnis

In dieser Arbeit wurde versucht die Entwicklung eines *transpect*-Projektes in Form einer Web-Anwendung aus theoretischer Sicht darzustellen und deren praktische Umsetzung zu erläutern. Alle verwendeten Quellcodes und Screenshots wurden aus dem aktuellen Entwicklungsstand bezogen und funktionieren gemäß ihrer Aufgaben. Nach der Durchführung einiger Testläufe im Anschluss der Entwicklung erwies sich die Zuweisung von IDML- bzw. WordML-Formatvorlagen als erfolgreich. Genauer betrachtet deckt er in Bezug auf die Konvertierungen im Backend und auf die Nutzung der browserbasierten Anwendung alle Anforderungen ab, die im Abschnitt 3.1 auf der Seite 35 aufgeführt sind. Mit ihm ist es möglich, Formatvorlagen unter Verwendung von Regeln zwischen XML-basierten Dokumentformaten auszutauschen. Dabei bezieht sich die Zuweisung vorerst aber nur auf Absatzformatvorlagen, was im Abschnitt 4.2 näher erläutert wird.
Der folgende Abschnitt präsentiert das Ergebnis der Formatvorlagenzuweisung eines Beispiel-Word-Dokumentes aus der Sicht eines Authors. Dabei werden nur relevante Ausschnitte der Dokumente gezeigt. Zur Darstellung des Quell- und Template-Dokumentes dient die Abbildung 4.1. Sie ist zusammen mit den verwendeten Zuweisungsregeln auf der Seite 74 aufgeführt, die mit Hilfe der browserbasierten Anwendung erstellt wurden und sich in der Tabelle 4.1 befinden.

Abbildung 4.1 Beispiel: Quell- und Template-Dokument

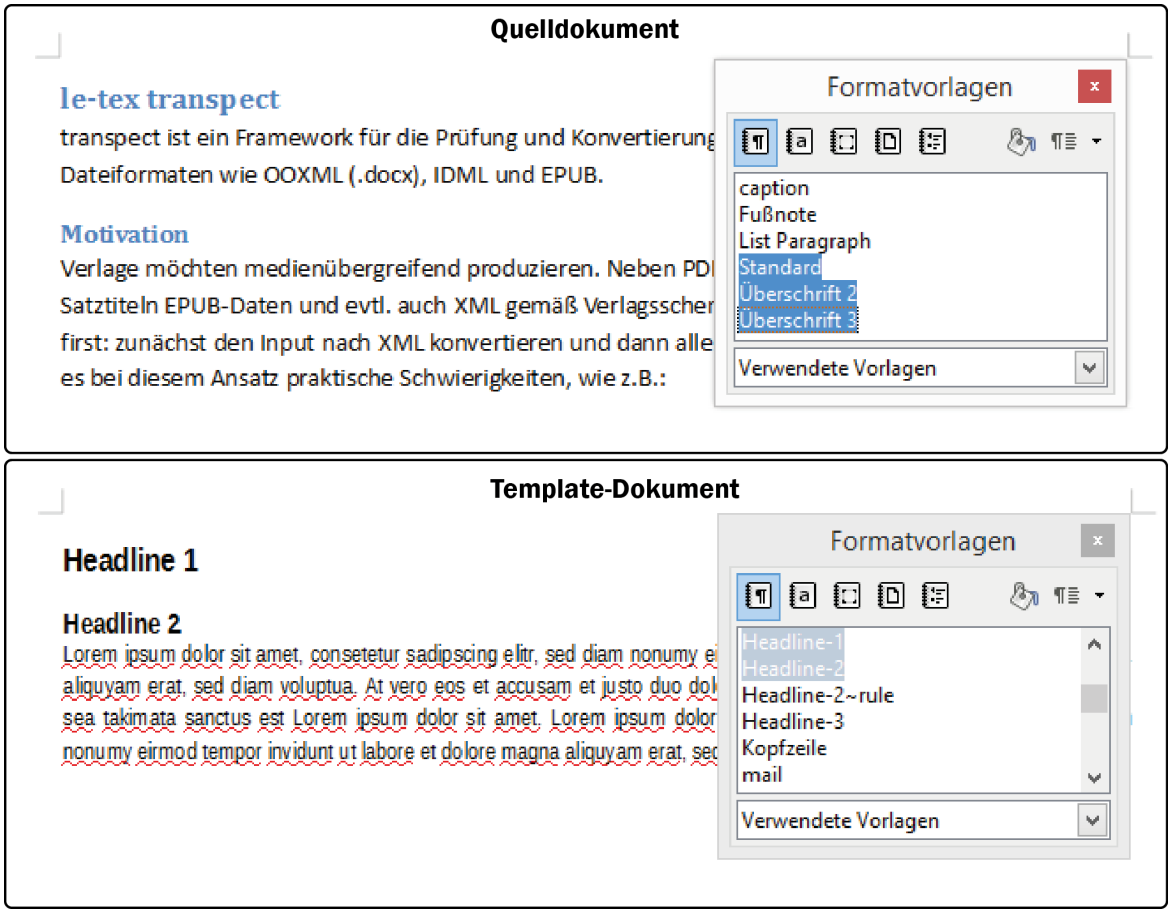
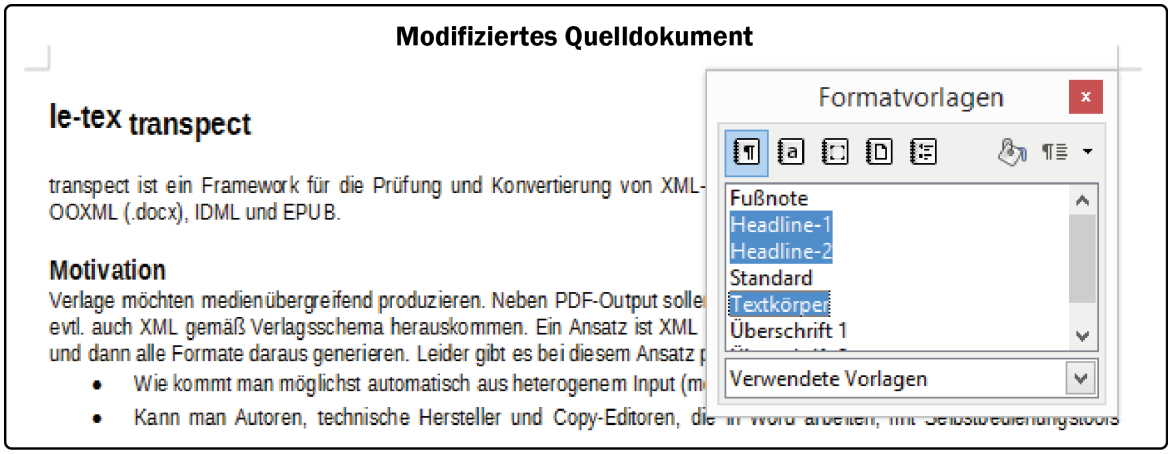


Tabelle 4.1 Beispiel: Zuweisungsregeln

Zuweisungsregel	Priorität	Zielelement	Zielvorlage	Eigenschaft	Wert
Regel 1	3	Absatz	Headline-1	Schriftgröße	13pt
				Schriftstärke	fett
Regel 2	2	Absatz	Headline-2	Schriftgröße	9pt – 11pt
Regel 3	1	Absatz	Textkrper	Schriftgröße	14pt
				Schriftfarbe	rgb(54, 95, 145)[1]

Darin befinden sich drei Regeln, die den Absätzen der beiden Überschriften und ihrer nachfolgenden Fließtexte über die Schriftstärke-, Schriftgröße- und Farbeigenschaften neue Formatvorlagen aus dem Tempalte-Dokument zuweisen. Das Konvertierungsergebnis der Backend-Pipelines unter Verwendung der Zuweisungsregeln ist eine modifizierte Version des Quelldokumentes, die in der Abbildung 4.2 auf der Seite 75 zu sehen ist. Darauf ist zu

Abbildung 4.2 Beispiel: Modifiziertes Quelldokument



erkennen, dass die Formatvorlagen über den *target-style*-Attributen erfolgreich angewendet wurden. Die Überschrift *le-tex transpect* bekommt die Formatvorlage *Headline-1* und die Überschrift *Motivation* die Vorlage *Headline-2*. Als Absatzformatierung der beiden Fließtexte wird die *Textkörper*-Vorlage verwendet.

4.2 Offene Anforderungen und Weiterentwicklung

Der Grund für die Beschränkung der Formatvorlagenzuweisungen auf Absätze war der enge Zeitrahmen, der für die Integration der Zuweisung von Zeichenformatvorlagen in die Verarbeitungsmechanismen zu eng war. Anfänglich wurden die Dokumentelemente, die über eigene Zeichenformatierungen verfügen, in die Zuweisungskriterien der Regeln eingebunden. Aber einige Zeit später stellte sich heraus, dass die Absatzformatvorlagenzuweisung gegenüber den Inline-Elementen der WordML-Dokumente (*runs*) sowie der Indesign-Satzdaten (*CharacterStyleRanges*) für den erstmaligen Durchlauf des Workflows ausreichend ist. Daraus abgeleitet existiert im Frontend bisher keine differenzierte Einteilung in Absatz- und Zeichenformatvorlagen.

Weiterhin steckt die Vorschaufunktion der Web-Anwendung noch in den Kinderschuhen. Die Klassenvergabe für die Dokumentelement-Formatierungen der HTML-Konvertierung muss weiter ausgebaut werden, damit die Zuweisungsregel möglichst genau auf die Elemente

anwendbar ist. Für die Inspektion der Dokumentelemente bedarf es nach der Integration der Zeichenformatvorlagenzuweisung in den Stylemapper eine differenzierte Darstellung von Inline- und Absatzelementen. Für einen vollständigen Zuweisungsumfang für die Formatvorlagen muss der Übersetzungskatalog der Formatierungskriterien in den XSL-Stylesheets für die Regeln um weitere IDML- und WordML-Formatierungen erweitert werden. Nach aktuellem Stand sind bisher für die Kriterien die Formatierungen der Abstands- und die Schriftformate verfügbar, wozu ebenfalls die Erweiterung der Farbformatierungen zum HSL-Format zählt. Derzeit sind ausschließlich die Schrift- und die Hintergrundfarbe integriert.
Für die Zukunft gibt es neben der Umsetzung der noch offenen Anforderungen Pläne für die Weiterentwicklung. Da die Firma le-tex publishing services GmbH auch im Bereich Ebook-Produktion tätig ist und das dort verwendete Open-Source-Format Epub auf XHTML und CSS basiert[2], wird eine Erweiterung der Formatvorlagenzuweisung für dieses Dokument-Format in Erwägung gezogen.
Abschließend folgt ein Vorschlag zur Verbesserung der Web-Anwendung. Um dem Nutzer einen Leitfaden zur Nutzung zu geben, wird über eine integrierte Hilfe auf der Basis von JavaScript nachgedacht. Dabei sollte er vor, während und nach der Durchführung des Workflows eine Bedienhilfe via Overlay-Elemente aktivieren können. Das hat den Vorteil, dass er nicht aus dem Anwendungsprozess herausgerissen und auf eine externe Ebene, wie beispielsweise eine Bedienungsanleitung geführt werden, sondern mitten im Geschehen bleiben würde.

[2]Vgl.: [3], Kapitel: *What Is Epub 3?*, Abschnitt: *EPUB 3 in a Nutshell* , Seite 4, Absatz 2

Quellenverzeichnis

[1] Jonas Sicking Arun Ranganathan. *File API*. Apr. 2015. URL: http://www.w3.org/TR/FileAPI/ (besucht am 02. 12. 2015).

[2] David Flanagan. *JavScript: The Definite Guide*. Sebastopol, USA: O'Reilly Media, Inc., März 2011.

[3] Matt Garrish. *What Is EPUB3?* Sebastopol, USA: O'Reilly Media, Inc., 2011.

[4] Marijn Haverbeke. *Eloquent JavaScript A Modern Introduction to Programming*. San Francisco, USA: No Starch Press, 2015.

[5] *HTTP Authentication: Basic and Digest Access Authentication*. 1999. URL: http://tools.ietf.org/html/rfc2617 (besucht am 05. 11. 2015).

[6] Gerrit Imsieke. *Access via Converter API*. URL: http://transpect.le-tex.de/api/ (besucht am 05. 11. 2015).

[7] Gerrit Imsieke. *Conveying Layout Information with CSSa*. Jan. 2013. URL: http://publishinggeekly.com/wp-content/uploads/2013/01/CSSa.pdf (besucht am 27. 10. 2015).

[8] Adobe Systems Incorporated. *IDML File Format Specification*. Juni 2010. URL: https://www.adobe.com/content/dam/Adobe/en/devnet/indesign/cs55-docs/IDML/idml-specification.pdf.

[9] Michael Kay. *XSL Transformations (XSLT) Version 2.0*. Jan. 2007. URL: http://www.w3.org/TR/xslt20/ (besucht am 02. 11. 2015).

[10] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference*. Indianapolis, USA: Wrox, Mai 2008.

[11] o.V. – le-tex publishing services GmbH. *file-uri.xpl - Documentation*. 2015. URL: https://subversion.le-tex.de/common/xproc-util/file-uri/file-uri.xpl (besucht am 20. 11. 2015).

[12] o.V. – le-tex publishing services GmbH. *load-cascaded.xpl – Documentation*. 2015. URL: https://github.com/transpect/cascade/blob/master/xpl/load-cascaded.xpl (besucht am 14. 11. 2015).

[13] o.V. – le-tex publishing services GmbH. *load-cascaded.xpl - Documentation*. 2015. URL: https://www.le-tex.de/de/transpect.html (besucht am 16. 11. 2015).

[14] o.V. – le-tex publishing services GmbH. *transpect Setup Manual*. URL: https://subversion.le-tex.de/common/transpect-demo/content/le-tex/setup-manual/en/out/xhtml/transpect-setup.xhtml (besucht am 15. 12. 2015).

[15] Sal Mangano. *XSLT Kochbuch*. Köln, Deutschland: O'Reilly Verlag, 2006.

[16] Tobias Klevenz Manuel Montero Pineda. *Einführungen in unsere XML-Technologien*. Jan. 2015. URL: http://www.data2type.de/ (besucht am 24. 06. 2015).

[17] Tobias Klevenz Manuel Montero Pineda. *Übersicht zu XProc*. Jan. 2015. URL: http://www.data2type.de/xml-xslt-xslfo/xproc (besucht am 27. 10. 2015).

[18] Tobias Klevenz Manuel Montero Pineda. *XML-Technologien / XSLT / Referenz 2.0*. Jan. 2015. URL: http://www.data2type.de/xml-xslt-xslfo/xslt/xslt-referenz/ (besucht am 24. 06. 2015).

[19] Brett McLaughlin. *Ajax meistern*. 2005. URL: http://www.oreilly.de/artikel/ajax1/index.html (besucht am 01. 12. 2015).

[20] Henry S. Thompson Norman Walsh Alex Milowski. *XProc 2.0: An XML Pipeline Language, W3C First Public Working Draft 18 December 2014*. Dez. 2014. URL: http://www.w3.org/TR/xproc20/ (besucht am 20. 06. 2015).

[21] Henry S. Thompson Norman Walsh Alex Milowski. *XProc: An XML Pipeline Language*. Mai 2013. URL: http://www.w3.org/TR/xproc/ (besucht am 24. 06. 2015).

[22] o.V. *Browser Support*. 2015. URL: http://jquery.com/browser-support/ (besucht am 10. 11. 2015).

[23] o.V. *Browser Support*. 2015. URL: http://learn.jquery.com/ (besucht am 10. 11. 2015).

[24] o.V. *CSS properties as XML attributes*. Juni 2015. URL: https://github.com/le-tex/CSSa/blob/master/css.rng.

[25] o.V. *ISO/IEC 29500-1*. Sep. 2012. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c061750_ISO_IEC_29500-1_2012.zip (besucht am 20. 06. 2015).

[26] o.V. *le-tex Hub Format*. Sep. 2015. URL: http://www.le-tex.de/resource/schema/hub/1.1/hub.rng.

[27] o.V. *Makefile*. Jan. 2015. URL: https://wiki.ubuntuusers.de/makefile (besucht am 30. 10. 2015).

[28] o.V. *Oxygen XML Editor 17.1*. 2015. URL: http://www.oxygenxml.com/doc/versions/17.1/ug-editor/index.html (besucht am 14. 11. 2015).

[29] o.V. *Saxon*. URL: http://www.saxonica.com/documentation/ (besucht am 05. 11. 2015).

[30] o.V. *Saxon*. URL: http://www.saxonica.com/products/products.xml (besucht am 05. 11. 2015).

[31] Keith Pijanowski. *Web Development - Building Responsive Web Sites with Bootstrap*. Juni 2015. URL: https://msdn.microsoft.com/en-us/magazine/mt147241.aspx (besucht am 27. 11. 2015).

[32] Daniel Dick of Rochester. *office open xml-wordproccessing*. Jan. 2014. URL: http://officeopenxml.com/WPstyle.php (besucht am 18. 06. 2015).

[33] Michael Spivak. *GNU Make*. Boston, USA: Free Software Foundation, 2014.

[34] Netscape Communications Corp. Tom Pixley. *Document Object Model (DOM) Level 2 Events Specification*. Nov. 2000. URL: http://www.w3.org/TR/DOM-Level-2-Events (besucht am 09. 11. 2015).

[35] Norman Walsh. *Documentation*. Mai 2015. URL: http://xmlcalabash.com/docs/ (besucht am 28. 10. 2015).

Anhang A

XSL-Stylesheet zur Markierung von leeren Absätzen

Quellcode A.1 Stylesheet *mark_para.xsl*

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:letex="http://www.le-tex.de/namespace"
  xmlns:sm="http://www.le-tex.de/namespace/stylemapper"
  xmlns:w="http://schemas.openxmlformats.org/wordprocessingml
     /2006/main"
  xmlns:dbk="http://docbook.org/ns/docbook"
  xmlns:css="http://www.w3.org/1996/css"
  xpath-default-namespace="http://docbook.org/ns/docbook"
  exclude-result-prefixes="xs"
  version="2.0">
  <xsl:template match="@*|*">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
  <xsl:function name="letex:contains-token" as="xs:boolean">
    <xsl:param name="tokens" as="xs:string?"/>
    <xsl:param name="token" as="xs:string?"/>
    <xsl:sequence select="tokenize($tokens, '\s+') = $token"/>
  </xsl:function>
  <xsl:template match="para[not(normalize-space())]
                           [
                              following-sibling::*[normalize-
                                 space()][1]
```

```
                                /self::para[letex:contains-token(
                                   @mapping-rules, 'margin-top')]
                             ]">
    <xsl:copy>
      <xsl:attribute name="sm:action" select="'delete'"/>
      <xsl:apply-templates select="@*,node()" mode="#current"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="*[local-name() = ('phrase', 'subscript',
     'superscript')]
                         [@srcpath]">
    <xsl:copy>
      <xsl:call-template name="merge-remove-adhocs">
        <xsl:with-param name="from-para" select="ancestor::para
           [1]/@sm:remove-adhoc" as="attribute(sm:remove-adhoc)?
           "/>
        <xsl:with-param name="from-phrase" select="@sm:remove-
           adhoc" as="attribute(sm:remove-adhoc)?"/>
      </xsl:call-template>
      <xsl:apply-templates select="@* except @sm:remove-adhoc,
         node()" mode="#current"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template name="merge-remove-adhocs" as="attribute(
     sm:remove-adhoc)?">
    <xsl:param name="from-para" as="attribute(sm:remove-adhoc)?"
       />
    <xsl:param name="from-phrase" as="attribute(sm:remove-adhoc)
       ?"/>
    <xsl:variable name="tmp" as="xs:string*"
      select="distinct-values(
                 (
                    tokenize($from-para, '\s+'),
                    tokenize($from-phrase, '\s+')
                 )
              )"/>
    <xsl:if test="exists($tmp)">
      <xsl:attribute name="sm:remove-adhoc" separator=" "
        select="if ($tmp = '#all')
                then '#all'
                else $tmp"/>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

Anhang B

Funktion zur HSL-Farbkonvertierung

Quellcode B.1 Auszug aus der *add_attributes.xsl*: Funktion zur HSL-Farbkonvertierung

```
<xsl:function name="letex:color2hsl" as="xs:string+">
    <xsl:param name="in"></xsl:param>
    <xsl:message select="'INPUTRGB',$in"></xsl:message>
    <xsl:variable name="rgb"  as="xs:string+">
       <xsl:choose>
           <xsl:when test="starts-with($in,'rgb')">
               <xsl:variable name="rgb_v"  as="xs:string+"
                  select="tokenize(
                   replace($in, '^rgb\(|\)|\s', ''), ','
                   )"/>
               <xsl:sequence select="$rgb_v"></xsl:sequence>
           </xsl:when>
           <xsl:when test="starts-with($in,'#')">
               <xsl:variable name="rgb_v"  as="xs:string+"
                  select="tokenize(replace(letex:hex2rgb($in),
                  '^rgb\(|\)|\s', ''), ',')"></xsl:variable>
               <xsl:sequence select="$rgb_v"></xsl:sequence>
               <xsl:message select="'RGBV_____VALUE',
                  letex:hex2rgb($in)"></xsl:message>
           </xsl:when>
           <xsl:when test="$in[. = 'black']">
               <xsl:variable name="rgb_v" select="('0','0','0')
                  " as="xs:string+"></xsl:variable>
               <xsl:sequence select="$rgb_v"></xsl:sequence>
               <xsl:message select="'RGBV_____VALUE_____black
                  ', $rgb_v"></xsl:message>
           </xsl:when>
       </xsl:choose>
   </xsl:variable>
    <xsl:message select="'RGBVALUE', $rgb"></xsl:message>
```

```
    <xsl:message select="'hahahahahah', string-join(for $v in $
        rgb return string(number($v)), ' ')"></xsl:message>
    <xsl:variable name="rgb_num" select="for $v in $rgb return
        number($v) div 255"></xsl:variable>

    <xsl:variable name="r" select="$rgb_num[1]"/>
    <xsl:variable name="g" select="$rgb_num[2]"/>
    <xsl:variable name="b" select="$rgb_num[3]"/>
    <xsl:variable name="min" select="min(($r, $g, $b))"/>
    <xsl:variable name="max" select="max(($r, $g, $b))"/>
    <xsl:variable name="d" select="$max - $min"/>
    <xsl:variable name="l" select="($max + $min) div 2"/>
    <xsl:variable name="h">
        <xsl:choose>
            <xsl:when test="$max eq $min">
                <xsl:value-of select="0"></xsl:value-of>
            </xsl:when>
            <xsl:when test="$r eq $max">
                <xsl:choose>
                    <xsl:when test="$g &lt; $b">
                        <xsl:variable name="h_v" select="(($g
                            - $b) div $d) + 6"></xsl:variable>
                        <xsl:value-of select="$h_v"></
                            xsl:value-of>
                    </xsl:when>
                    <xsl:otherwise>
                        <xsl:variable name="h_v" select="(($g -
                            $b) div $d) + 0"></xsl:variable>
                        <xsl:message select="' R = MAX', $h_v"><
                            /xsl:message>
                        <xsl:value-of select="$h_v"></xsl:value-
                            of>
                    </xsl:otherwise>
                </xsl:choose>
            </xsl:when>
            <xsl:when test="$g eq $max">
                <xsl:variable name="h_v" select="(($b - $r) div
                    $d) + 2"></xsl:variable>
                <xsl:message select="' G = MAX', $h_v"></
                    xsl:message>
                <xsl:value-of select="$h_v"></xsl:value-of>
            </xsl:when>
            <xsl:when test="$b eq $max">
                <xsl:variable name="h_v" select="(($r - $g) div
                    $d) + 4"></xsl:variable>
                <xsl:message select="' b = MAX', $h_v"></
                    xsl:message>
                <xsl:value-of select="$h_v"></xsl:value-of>
            </xsl:when>
```

```
        </xsl:choose>
    </xsl:variable>
    <xsl:variable name="s">
        <xsl:choose>
            <xsl:when test="$max eq $min">
                <xsl:value-of select="0"></xsl:value-of>
            </xsl:when>
            <xsl:otherwise>
               <xsl:choose>
                    <xsl:when test="$l &gt; 0.5">
                        <xsl:variable name="s_v" select="$d div
                           (2 - $max -$min)"></xsl:variable>
                        <xsl:value-of select="$s_v"></xsl:value-
                           of>
                    </xsl:when>
                    <xsl:otherwise>
                        <xsl:variable name="s_v" select="$d div
                           ($max + $min)"></xsl:variable>
                        <xsl:value-of select="$s_v"></xsl:value-
                           of>
                    </xsl:otherwise>
                </xsl:choose>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:variable>
        <xsl:sequence select="(xs:string(round($h div 6 * 360)),
            xs:string(round($s * 100)), xs:string(round($l *
          100)))"></xsl:sequence>
</xsl:function>

<xsl:template match="para|css:rule" priority="2">
    <xsl:copy>
        <xsl:choose>
            <xsl:when test="./@css:color">
                <xsl:attribute name="color-h" select="
                   letex:color2hsl(@css:color)[1]"></
                   xsl:attribute>
                <xsl:attribute name="color-s" select="
                   letex:color2hsl(@css:color)[2]"></
                   xsl:attribute>
                <xsl:attribute name="color-l" select="
                   letex:color2hsl(@css:color)[3]"></
                   xsl:attribute>
        </xsl:when>
            <xsl:when test="./@css:background-color">
                <xsl:attribute name="background-color-h" select
                   ="letex:color2hsl(@css:background-color)[1]"
                   ></xsl:attribute>
```

```
                    <xsl:attribute name="background-color-s" select
                        ="letex:color2hsl(@css:background-color)[2]"
                        ></xsl:attribute>
                    <xsl:attribute name="background-color-l" select
                        ="letex:color2hsl(@css:background-color)[3]"
                        ></xsl:attribute>
                </xsl:when>
    </xsl:choose>
        <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
</xsl:template>
```

Anhang C

JavaScript-Funktionen für Zuweisungsregeln

Quellcode C.1 Speicherung einer Zuweisungsregel: Funktion *setProp()*

```
function setProp(){
     var prop1 = new Prop();
     prop1.name =  $('select#pname').val();
     prop1.relevant =  $('#prelevant').val();
     prop1.id = guid();
     if (prop1.name == 'color' || prop1.name == 'background-
        color'){
            var name = prop1.name.replace(/-/, '');
            if ($('#pregex').val() != ""){
             prop1.regex =  $('#pregex').val();
    }
    else if ($('#r2').prop('checked') == true || $('#pvalue').
       val() == null || $('#pvalue').val() == 'undefined'){
          prop1.minvalue =  $('#pmin-value').val();
          prop1.maxvalue =  $('#pmax-value').val();
          var rgb_min_arr = rgb2array(prop1.minvalue),
          rgb_max_arr = rgb2array(prop1.maxvalue),
          hsl_min_arr = rgb2Hsl(rgb_min_arr[0],rgb_min_arr[1],
             rgb_min_arr[2]),
          hsl_max_arr = rgb2Hsl(rgb_max_arr[0],rgb_max_arr[1],
             rgb_max_arr[2]);
          var minh = name+'minh',
          mins = name+'mins',
          minl = name+'minl',
          maxh = name+'maxh',
          maxs = name+'maxs',
```

```
            maxl = name+'maxl';
            prop1[minh] = hsl_min_arr[0];
            prop1[mins] = hsl_min_arr[1];
            prop1[minl] = hsl_min_arr[2];
            prop1[maxh] = hsl_max_arr[0];
            prop1[maxs] = hsl_max_arr[1];
            prop1[maxl] = hsl_max_arr[2];
    }
    else if ($('#r1').prop('checked') == true){
          prop1.value =  $('#pvalue').val();
          var rgb_value_arr = rgb2array(prop1.value),
          hsl_value_arr = rgb2Hsl(rgb_value_arr[0],rgb_value_arr
             [1],rgb_value_arr[2]),
          h = name+'h',
          s = name+'s',
          l = name+'l';
          prop1[h] = hsl_value_arr[0];
          prop1[s] = hsl_value_arr[1];
          prop1[l] = hsl_value_arr[2];
     }
     }
     else{
          prop1.value =  $('#pvalue').val();
          prop1.regex =  $('#pregex').val();
          prop1.minvalue =  $('#pmin-value').val();
          prop1.maxvalue =  $('#pmax-value').val();
     }
     if (checkProp(prop1) === true) {
        prop_arr.push(prop1);
        var li = document.createElement('li');
        li.setAttribute('id', prop1.id);
        li.setAttribute('name', prop1.name);
        li.innerHTML = "<div class='input-group'><span class='
           input-group-addon'>"+prop1.name+'</span>'+"<div class
           ='btn-group btn-group-justified'><a role='button' data
           -id='"+ prop1.id+"' class='btn btn-sm btn-default
           delete-prop'>Delete</a></div></div>" ;
       $('#prop-table').remove();
       $('#properties').append(li);
       $('.edit-prop').on('click', function(){
         editProp(event.target.getAttribute('data-id'));
       })
       $('.delete-prop').on('click', function(event){
         deleteProp(event.target.getAttribute('data-id'));
       })
       }
}
```

Quellcode C.2 Speicherung einer Zuweisungsregel: Funktion *setMapping()*

```
function setMapping(){
  map_obj = new Mapping();
  map_obj.name = document.getElementById('name').value;
  map_obj.priority = document.getElementById('priority').value;
  map_obj.targetstyle = document.getElementById('target-style').
     value;
  map_obj.targettype = document.getElementById('target-type').
     value;
  var adhoc_arr = [];
  adhoc_arr = $('#remove-adhoc > li').find('input:checked');
  var val_arr = [];
  adhoc_arr.each( function (){
      val_arr.push(this.value)
  })
  adhoc_string = val_arr.join(" ");
  map_obj.removeadhoc = adhoc_string;
  return map_obj
}
```

Quellcode C.3 Speicherung einer Zuweisungsregel: Funktion *addProps(map_obj)*

```
function addProps(map_obj){
 var proplinks = $('#properties').find('li[name]');
 for (var i=0; i < proplinks.length; i++){
   var prop = getPropById(proplinks[i].id);
   console.log(proplinks[i]);
   map_obj.props.push(prop);
 }
}
```

Quellcode C.4 Speicherung einer Zuweisungsregel: Funktion *saveMapping(map_obj, override)*

```
function saveMapping(map_obj, override){
    if (checkMapping(map_obj, override) === true){
    var mapping = document.createElement('mapping');
    mapping.setAttribute('name', map_obj.name);
    mapping.setAttribute('priority', map_obj.priority);
    mapping.setAttribute('target-type', map_obj.targettype);
    mapping.setAttribute('target-style', map_obj.targetstyle);
    mapping.setAttribute('remove-adhoc', map_obj.removeadhoc);
    $.each(map_obj.props, function(){
    var property = document.createElement('prop');
    property.setAttribute('name', this.name);
    property.setAttribute('relevant', this.relevant);
    property.setAttribute('regex', this.regex);
```

```
    var name = this.name.replace(/-/, '');
    if (this.name == 'color' || this.name == 'background-color')
       {
        if (((this.value == '') || (this.value == null)) && ((
           this.regex == '') || (this.regex == null))){
        property.setAttribute('max-value', this.maxvalue);
        property.setAttribute('min-value', this.minvalue);
        property.setAttribute(this.name+'-min-h', this[name+'
           minh']);
        property.setAttribute(this.name+'-min-s', this[name+'
           mins']);
        property.setAttribute(this.name+'-min-l', this[name+'
           minl']);
        property.setAttribute(this.name+'-max-h', this[name+'
           maxh']);
        property.setAttribute(this.name+'-max-s', this[name+'
           maxs']);
        property.setAttribute(this.name+'-max-l', this[name+'
           maxl']);
        console.log('propertyyyy', property);
        }
        else{
        property.setAttribute('value', this.value);
        property.setAttribute(this.name+'-h', this[this.name+'h'
           ]);
        property.setAttribute(this.name+'-s', this[this.name+'s'
           ]);
        property.setAttribute(this.name+'-l', this[this.name+'l'
           ]);
        }
    }
    else{
        property.setAttribute('value', this.value)
        property.setAttribute('max-value', this.maxvalue)
        property.setAttribute('min-value', this.minvalue)
    }
    mapping.appendChild(property)
    });
        createSuccess('Mapping rule saved!');
        mapping_set.appendChild(mapping);
        sortByPriority()
        showMaps();
        showContent('mrules');
        $('#mapping > tbody').addClass('disabled06');
        createMappingTable();
    }
}
```

Quellcode C.5 Speicherung einer Zuweisungsregel: Funktion *editMapping(name)*

```
createMappingTable();
 $('#mapping').removeClass('disabled06');
 var map_obj = getMapping(name);
 $('#properties').children('a').remove();
 $('#name').val(map_obj.name);
 $('#priority').val(map_obj.priority);
 $('#target-style').val(map_obj.targetstyle);
 $('#target-type').val(map_obj.targettype);
 var adhoc_arr = map_obj.removeadhoc.split(" ");
 for (var i=0; i < cssstyles.length; i++){
     for ( var j=0; j < adhoc_arr.length; j++){
         if (cssstyles[i] == adhoc_arr[j]) {
              console.log(cssstyles[i]);
             console.log($("#mapping > li > input[name="+
                cssstyles[i]+"]"));
             $("input[name="+cssstyles[i]+"]").attr('checked',
                true)
         }
     }
 }
 for (var i=0; i < map_obj.props.length ; i++){
  var li = document.createElement('li');
  var icon_edit = document.createElement('span');
  var icon_delete = document.createElement('span');
  li.setAttribute('id', map_obj.props[i].id);
  li.setAttribute('name', map_obj.props[i].name);
  li.innerHTML = "<div class='input-group'><span class='input-
     group-addon'>"+map_obj.props[i].name+'</span>'+"<div class
     ='btn-group btn-group-justified'><a role='button' data-id
     ='"+map_obj.props[i].id+"' class='btn btn-sm btn-default
     delete-prop'>Delete</button></div></div>" ;
  prop_arr.push(map_obj.props[i]);
  $('#properties').append(li);
 }
 $('.edit-prop').on('click', function(event){
   editProp(event.target.getAttribute('data-id'));
 })
 $('.delete-prop').on('click', function(event){
   deleteProp(event.target.getAttribute('data-id'));
 })
 $('#mapping > tbody').removeClass('disabled06');
}
```

Quellcode C.6 Hochladung von Zuweisungsregeln: Funktion *sendMapping()*

```
function sendMapping(){
```

```
  objbox = document.createElement('div');
  objbox.appendChild(mapping_set)
  stringbox = objbox.innerHTML
  episode = $('meta[name=episode]').attr('content');
  var string_data = [stringbox];
  var blob = new Blob(string_data, {type: 'text/plain'});
  var fd = new FormData();
  fd.append('input_file', blob, episode+'.xml');
  fd.append('type', "stylemapper");
  fd.append('add_params', "");
$.ajax({
    type: 'POST',
    url: 'https://transpect.le-tex.de/api/upload_file',
    async:true,
    data: fd,
    processData: false,
    contentType: false,
    "beforeSend": function(xhr){
      xhr.setRequestHeader("Authorization", basicHTTPAuthString(
         username, password));
    },
    success: function(data){
      var callbackuri = data["callback_uri"];
      initMappingStatusRequest(callbackuri);
    }
})};
```