Hochschule Neubrandenburg
University of Applied Sciences

Studiengang Geoinformatik

Diplomarbeit

Thema: Entwicklung einer grafischen Benutzeroberfläche zum Generieren, Bearbeiten und Ausführen von Transformationsprozessen zwischen Xml - Metadaten und dem internen Metadatenformat des Deutschen Zentrums für Luft- und Raumfahrt

Eingereicht von: Christian Kern - 20.07.2010

Erstprüfer: Prof. Dr. Andreas Wehrenpfennig

Zweitprüfer: Dipl. Ing. Henrike Barkmann

URN: urn:nbn:de:gbv:519-thesis2009-0284-2

Abstract

Diese Diplomarbeit beschreibt die Entwicklung einer grafischen Entwicklungsumgebung zum Generieren und Testen von XSL- Transformationsprozessen. Im Vordergrund steht dabei die Gestaltung der grafischen Nutzeroberfläche bzw. der Nutzerschnittstellen sowie die Visualisierung von XML- Dokumenten und das Erstellen einer Funktionalität zur Verarbeitung dieser Dokumente. Die Funktionalität basiert auf einer Implementierung des Document Object Models, einer Empfehlung des World Wide Web Consortium (W3C).

Die Diplomarbeit wurde im Rahmen einer Projektarbeit im Auftrag des Deutschen Zentrums für Luft- und Raumfahrt (DLR) erstellt. Die Motivation der Arbeit liegt in der Optimierung eines Arbeitsprozesses begründet. Es werden bestimmte Arbeitsabläufe und Strukturen des DLR beschrieben. Die Aufgabenstellung des DLR ist der Ausgangspunkt für den Entwicklungsprozess.

Inhaltsverzeichnis

Abstract **1**

Inhaltsverzeichnis **2**

1 Einleitung **5**

2 Hintergrund **7**

2.1 Produktgenerierung und Archivierung von Nutzlastdaten 9
2.2 Beschreibung des zu optimierenden Arbeitsprozesses 10
2.2.1 XML- Extensible Markup Language 11
2.2.1.1 Struktur 12
2.2.1.2 Eigenschaften 14
2.2.2 XSL- Extensible Stylesheet Language 14
2.2.2.1 Templates 15
2.2.2.2 Variablen und Parameter 16
2.2.2.3 Kontrollstrukturen 16
2.2.3 Funktionen 18
2.2.4 XPath 18
2.2.5 IIF- Item Information File 19
2.3 Merkmale der generierten Metadaten 20
2.4 Transformation der XML- Metadaten durch XSL-T 21

3 Aufgabenstellung **24**

4 Anforderungen **26**

4.1 Transformationsprozess 26

4.2 Benutzeroberfläche 27

4.3 Funktionalität 28

5 Bestimmung des Transformationsprozesses **29**

6 Programmentwurf **30**

6.1 Visualisierung XML/XSL innerhalb der GUI 30

6.2 Gestaltung der Benutzeroberfläche und Schnittstellen 32

6.3 Funktionen 34

6.4 Nutzerhilfe 37

6.5 Entwicklungsumgebung 37

6.6 Verarbeitung XML/XSL in JAVA 38

6.6.1 Visualisierung 38

6.6.2 XML- Parser 38

6.6.3 Document Object Model 39

6.6.4 XSL- Prozessor und Serialisierung von DOM 40

7 Beschreibung der Implementierung **41**

7.1 Programmarchitektur 41

7.2 Verwendete Programmierschnittstellen 43

7.3 Daten und Funktionalität 43

7.3.1 Datenhaltung 43

7.3.2 Speichern und Laden von XML- Dokumenten 44

7.3.3 Generierung des Datenmodells für die Visualisierung 44

7.3.4 Editierfunktionen 47

7.3.4.1 Allgemeiner Aufbau 47

7.3.4.2 Liste der Methoden 49

7.3.5 XSL-Prozessor 52

7.3.6 Fehlerbehandlung 53

7.4 Grafische Benutzeroberfläche 53
7.4.1 Visualisierung XML 55
7.4.2 Verschieben von Knoten innerhalb des JTree 56
7.4.3 Nutzerhilfe 56

8 Validierung des Funktionsumfanges 57
8.1 Abrufen und Visualisieren von XML- Dokumenten 57
8.2 Editierfunktionen und XSL- Prozessor 59

9 Zusammenfassung 62
9.1 Probleme 63
9.2 Ausblick 64

Abbildungsverzeichnis 66

Literaturverzeichnis 67

Anhang 68

1 Einleitung

Diese Diplomarbeit ist im Rahmen eines mehrmonatigen Praktikums im Auftrag des Deutschen Zentrum für Luft und Raumfahrt (DLR) - Standort Neustrelitz entstanden.

Die Motivation dieser Arbeit ist die Optimierung eines Arbeitsprozesses des Fernerkundungsdatenzentrums des Standortes. Dabei werden von Mitarbeitern automatisierte Prozesse generiert, die Rohdaten aus Erdbeoabachtungsmissionen in eine Archivierungsdatenbank übertragen. Die von den Prozessen erzeugten Metadaten über den bearbeiteten Rohdaten müssen dabei in die Metadatenspezifikation der Archivierungsdatenbank überführt werden.

Zur Verarbeitung der Metadaten in die DLR- interne Metadatenspezifikation müssen die Mitarbeiter des DLR innerhalb des automatisierten Ablaufes einen Transformationsprozess einfügen, der die anfallenden Metadaten formatiert und in die internen Metadatenspezifikation überführt. Das Erstellen des Transformationsprozesses gestaltet sich dabei als schwierig, da keine einheitliche und effektive Entwicklungsumgebung zur Verfügung steht.

Im Auftrag des DLR soll nun eine Entwicklungsumgebung in Form einer grafischen Nutzeroberfläche samt Funktionalität entworfen und programmiert werden, mit der ein Nutzer einen Transformationsprozess schnell und effektiv gestalten und testen kann. Somit soll der Arbeitsaufwand verringert werden, der insgesamt bei dem Generieren von automatisierten Prozessen zur Datenarchivierung benötigt wird.

Der erste Teil der Arbeit analysiert den zu optimierenden Arbeitsprozess, zeigt den derzeitigen

Lösungsansatz und beschreibt die verwendeten Technologien. Im Anschluss wird die Aufgabenstellung sowie die daraus resultierenden Anforderungen an die grafische Nutzeroberfläche und deren Funktionalität erklärt.

Der nächste Schritt ist der Entwurf der grafischen Nutzeroberfläche sowie deren Funktionalität. Zudem wird auch beschrieben, auf welche Weise der Transformationsprozess abgebildet werden soll und mit welchen technischen Mitteln der Entwurf realisiert wird. Die Umsetzung des Entwurfs in eine Programmiersprache wird dann bei der Beschreibung der Implementierung besprochen.

Im letzten Teil der Arbeit werden Testszenarien besprochen und Ergebnisse der Arbeit zusammengefasst.

2 Hintergrund

Der zu optimierende Arbeitsprozess betrifft die automatisierte Datenarchivierung und Produktgenerierung des Fernerkundungsdatenzentrums des DLR in Neustrelitz. Die dabei eingehenden Daten, die sog. Nutzlastdaten, stammen aus unterschiedlichen Erdbeobachtungsmissionen. Um den Prozess näher beschreiben zu können muss vorher auf für den Prozess relevanten Komponenten der EDV- Infrastruktur des DLR eingegangen werden. Im Vordergrund steht dabei das Daten- und Informations- Managment System, kurz DIMS.

DIMS ermöglicht das Management großer heterogener digitaler Datenbestände mit Raum-Zeit-Bezug. Das System zeichnet sich durch eine modulare und offene Architektur aus, die flexibel auf neue Anwendungen und Produkttypen reagieren kann. Ein Produkt ist dabei eine logisch zusammenhängende Sammlung von Informationen. Die Funktionalität umfasst u.a.:

- Archivierung von Daten
- Online Zugriff auf Datenbestände
- Produktgenerierung aus Nutzlastdaten

Die Archivierung der Daten wird über eine Komponente des DIMS abgewickelt, der Product Library kurz PL. Die Aufgaben der PL umfassen die konsistente und langfristige Speicherung und Bereitstellung von Daten aus Erdbeobachtungen (Produkte). Der Zugriff erfolgt dabei über eine grafische Benutzeroberfläche oder ein Tool auf Kommandozeilenebene.

Funktionalitäten sind u.a.:

- Übersicht über Produkte anhand von Suchkriterien
- Nutzung einer Object Query language für komplexe Anfragen
- Überführung lokaler Produkte in die PL

Um die automatisierten Prozesse zur Produktgenerierung und Archivierung von Nutzlastdaten zu steuern steht eine weitere Komponente innerhalb des DIMS zur Verfügung, das Processing System Management kurz PSM. Dies ist eine Nutzeroberfläche zum Gestalten von Prozessen. Prozesse lassen sich in Java definieren und werden im DIMS bei bestimmten Ereignissen ausgeführt. [5][8]

2.1 Produktgenerierung und Archivierung von Nutzlastdaten

Abbildung 2.1 beschreibt den Prozess der automatisierten Produktgeneriung- und Archivierung. Ein Mitarbeiter des Fernerkundungsdatenzentrums definiert über das PSM eine Reihe von Prozessen, die ein Produkt aus abgerufenen Nutzlastdaten generieren und in der PL archivieren.

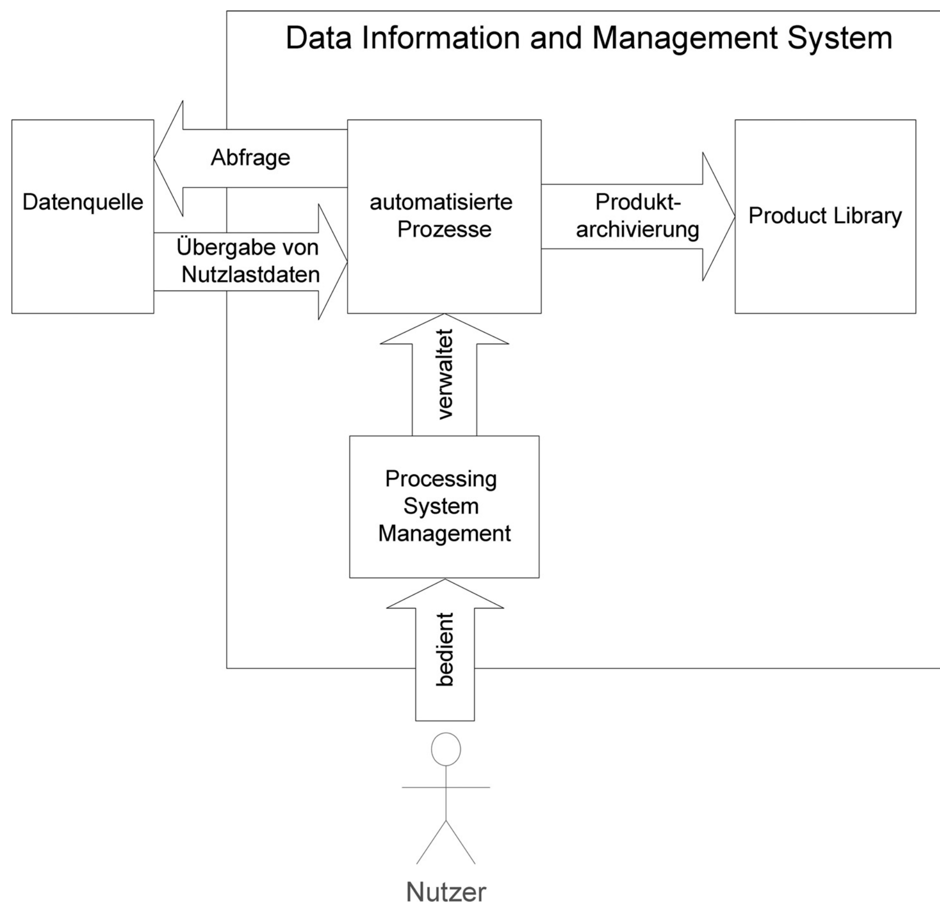

Abbildung 2.1: Datenarchivierung im Data Information and Managment System

Für jede Erdbeobachtungsmission wird individuell eine Reihe von Prozessen innerhalb des DIMS erstellt. Die Prozesse arbeiten abhängig voneinander und sind jeweils für verschiedene

Aufgaben eingesetzt. Diese Aufgaben sind[5][8] :

- Abfrage der Datenströme
- Teilung der Datenströme in logische Einheiten - Generierung eines Produktes
- Generierung von Metadaten über dem jeweiligen Produkt
- Transformation der Metadaten in die Metadatenspezifikation der PL
- Archivierung der Produkte und transformierten Metadaten in die PL

2.2 Beschreibung des zu optimierenden Arbeitsprozesses

Der zu optimierende Arbeitsschritt bei der Generierung der automatisierten Prozesse im DIMS ist das Erstellen des Transformationsprozesses zwischen erzeugten Produktmetadaten und der Metadatenspezifikation der PL, dem Item Information File, kurz IIF.

Die Bedeutung des Transformationsprozesses verdeutlicht Abbildung 2.2, die die automatisierten Prozesse zur Produktgenerierung beschreibt. Nach Abfrage der Nutzlastdaten wird das Produkt generiert. Anschließend wird ein Metadatendokument nach der XML- Spezifikation erstellt. Dieses enthält z.B. allgemeine Daten wie den Missionstyp sowie produktspezifische Daten, wie z.B. die Zeit und den Ort der Beobachtung.

Der nächste Schritt stellt die Transformation der Metadaten mit Hilfe eines XSL Transformationsprozesses dar. Die Metadaten werden aus dem XML- Dokument ausgelesen, wenn nötig formatiert und letztendlich in die Struktur der IIF- Spezifikation eingebettet. Am Ende steht ein Produkt, das mit Hilfe der IIF- Metadaten in die PL überführt werden kann.

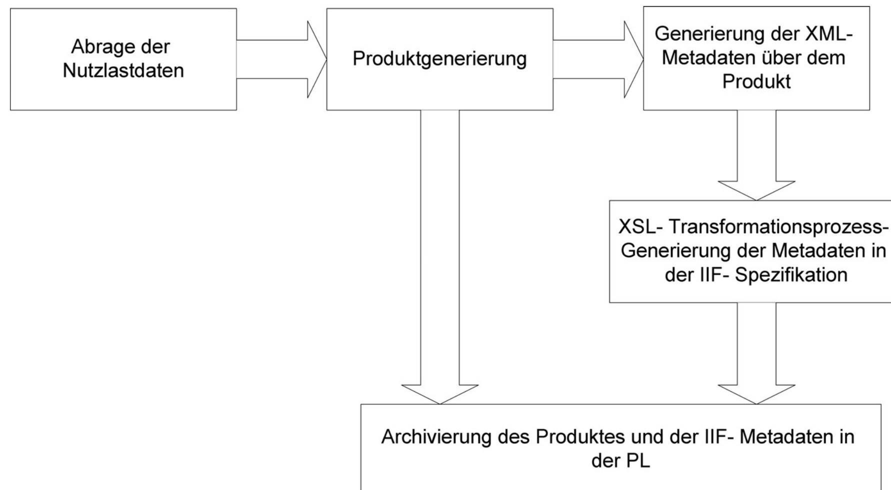

Abbildung 2.2: Automatisierte Prozessreihe im DIMS

Um den Lösungsansatz des DLR näher beschreiben zu können muss an dieser Stelle auf die verwendeten Metadatenspezifikationen XML und IIF sowie die verwendete Transformationssprache XSL-T eingegangen werden. Anschließend werden die Eigenschaften der entstehenden XML- Metadaten näher erläutert. Der letzte Schritt bei der Beschreibung des Arbeitsprozesses ist die Beschreibung der Entwicklung des Transformationsprozesses und die allgemeinen Eigenschaften desselben.

2.2.1 XML- Extensible Markup Language

Die Extensible Markup Language (XML) ist eine Auszeichnungssprache, die zur Speicherung von Daten in hierarchischen Strukturen im Textformat dient. Dabei ist die Namensgebung der Strukturelemente dem Nutzer überlassen, der somit neue Vokabulare definieren kann. Dies ermöglicht eine Vielfalt von auf XML basierenden anwendungsspezifischen Sprachen. Zudem ermöglicht die grundlegende Struktur die Entwicklung von Werkzeugen, die alle auf XML basierenden Sprachen verarbeiten können. Die XML Spezifikation ist eine Empfehlung des World Wide Web Consortiums (W3C). XML wird für den Austausch von Daten zwischen Computerprogrammen eingesetzt. Es ist jedoch auch durch den Menschen lesbar, da es sich

um Textdaten handelt.[1][2]

2.2.1.1 Struktur

Im XML Dokumenten werden Strukturelemente von dem Text, den sie auszeichnen, durch spezielle Zeichen abgegrenzt (<, >, &, "und '). Diese werden als Markup bezeichnet.

Der logische Aufbau einer XML Datei entspricht einer Baumstruktur mit folgenden Baumknoten[1] :

- XML Deklaration

  Eine optionale Deklaration in der ersten Zeile der XML Datei informiert über die Version der XML Spezifikation, die verwendete Zeichenkodierung sowie externe Auszeichnungsdeklerationen (Document Type Defnition).

  ```
  <?xml version="1.0" encoding= "EUC-JP" standalone = "yes" ?>
  ```

- Elemente

  Elemente sind die Hauptbestandteile der XML Struktur. Sie werden durch einen Start-Tag und einen End- Tag begrenzt, der den Inhalt einschließt. Ein Element ohne Inhalt besteht aus nur einem Tag. Die Tags werden durch die Zeichen <und >eingegrenzt. Elemente können jede andere XML Komponente enthalten ausgenommen der DOCTYPE-Deklaration und deren Inhalt. Das bedeudet, dass Elemente beliebig tief ineinander verschachtelt werden können. Das Beispiellisting zeigt das Element <*Person*> mit Start- und Endtag sowie ohne Inhalt nur mit einem Tag.

  ```
  <Person>
         <Name>Max Mustermann</Name>
  </Person>
  <Person/>
  ```

Ein wichtiger Aspekt von Elementen sind Namensräume. Diese werden benutzt um mehrere XML- Sprachen in einem Dokument zu verwenden und um Namen von Elementen eindeutig zu gestalten. So werden Namenskollisionen vermieden. Namensräume werden durch Webadressen (URIs) beschrieben, die Informationen über den Namensraum enthalten. Das folgende Beispiel zeigt die Definition des Namensraumes *namespace* mittels der URI *http://namespace.de*. Anschließend kann der Namensraum im Dokument verwendet werden.

```
<Element xmlns:namespace="http://namespace.de>
   <namespace:Element> Dieses Element ist dem Namensraum namespace zugeordnet
             </namespace:Element>
</Element>
```

- Attribute

Attribute sind Paare aus einem Namen und einem Wert, die innerhalb des Starttags eines Elementes auftreten können. Mehrere Attribute je Element sind möglich, müssen aber unterschiedliche Namen besitzen. Das Beispiel zeigt die Attribute *name* und beruf.

```
<Person name="Max Musterman" beruf="Mustermacher">
     <Geburtsdatum>Elementinhalt der das Geburtsdatum enthält<Geburtsdatum>
</Person>
```

- Textinhalte

Ein Textinhalt ist Text, der innerhalb von Elementen auftritt. Erlaubt sind alle Zeichen der Zeichenkodierung außer Zeichen, die als Trennelemente fungieren.

Auf weitere Komponenten der Spezifikation wie Kommentare oder CDATA wird aufgrund mangelnder Relevanz in dieser Arbeit nicht weiter eingegangen.

2.2.1.2 Eigenschaften

Erfüllt ein XML- Dokument die Regeln der XML- Spezifikation, so kann es als wohlgeformt bezeichnet werden. Dabei müssen verschiedene Bedingungen eingehalten werden[1] :

- Das XML- Dokument besitzt nur ein Wurzelelement, welches in der Hierarchie der logischen Baumstruktur am höchsten steht.

- Jedes Element besitzt einen Start- und End- Tag, Elemente ohne Inhalt können wie beschrieben nur aus einem Tag bestehen.

- Attribute innerhalb eines Elementes haben nicht denselben Namen

- Start- und Endtags eines Elementes müssen auf der selben Hierarchiestufe im Baum stehen.

2.2.2 XSL- Extensible Stylesheet Language

Die extensible Stylesheet Language (XSL) bezeichnet eine Familie von Sprachen, die zur Formatierung und Transformation von XML Dokumenten verwendet werden und die Naviagation in XML Bäumen erleichtern.

XSL Transformations (XSLT) ist eine auf XML basierte Sprache, die zur Transformation der Struktur von XML-Dokumenten dient. Die Transformation wird durch einen XSLT Prozessor durchgeführt. Wie in Abb. 2.3 zu sehen werden dem Prozessor XML Dokumente, XSLT Vorlagen und Parameter übergeben, die zu einem Ausgabedokument verarbeitet werden. Das Ausgabedokument ist wiederum ein XML-Dokument oder ein Textdokument.

Die XSLT-Vorlage, oder auch XSLT-Stylesheet, besteht aus einer Reihe Transformationsregeln

(Templates). Templates beschreiben die für die Transformation ausgewählten XML-Elemente im Quelldokument sowie deren Erscheinung im Ausgabedokument. Dabei können Kontrollstrukturen, Variablen, Sortierfunktionen und andere Hilfswerkzeuge eingebunden werden. Jede XSLT Anweisung im Stylesheet ist durch den Namensraum *xsl* gekennzeichnet.[3][4]

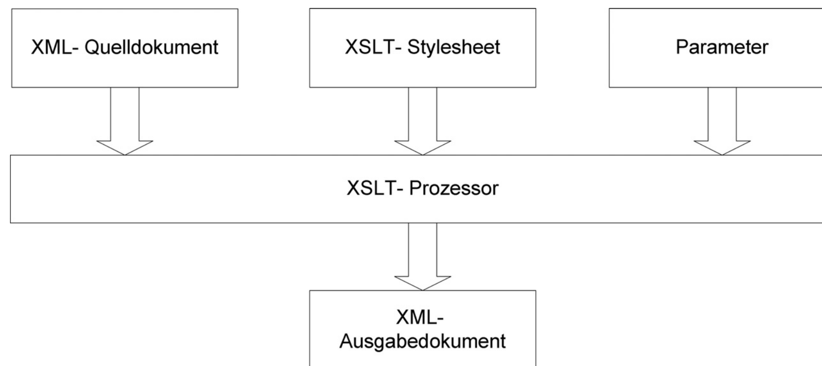

Abbildung 2.3: XSL- Transformation

2.2.2.1 Templates

Templates werden ausgeführt wenn das im Template beschriebene Element im Quelldokument gefunden wurde. Die auf die Bedingung folgende Struktur wird dann in das Ausgabedokument übernommen.

```
<xsl:stylesheet>
   <xsl:template match="//Element">
      <neueStruktur>
          <xsl:apply-templates/>
          <xsl:value-of select="Element"/>
      </neueStruktur>
   </xsl:template>
</xsl:stylesheet>
```

Mit der *match* Anweisung wird die Bedingung festgesetzt. Das Template wird nur beim Finden des Elementes *XmlElement* ausgeführt. *Apply-templates* überträgt das Element und alle darunterliegenden Elemente in die neue Dokumentstruktur. Die *value-of* Anweisung greift nur auf

den Textinhalt eines Elementes zu und überträgt diesen in das Ausgabedokument. Zusätzlich erlaubt die Anweisung auch Zugriff auf den Inhalt von Parametern und Variablen.

Das Stylesheet wird durch die xsl:stylesheet Tags in der XML Datei abgegrenzt.[3]

2.2.2.2 Variablen und Parameter

Neben Stylesheet und Quelldokument können dem Transformationsprozessor auch Parameter übergeben werden. Diese Parameter sind im Stylesheet definiert.

```
<xsl:stylesheet>
   <xsl:param name="ParameterName"/>
</xsl:stylesheet>
```

Es ist zusätzlich möglich innerhalb des Stylesheets Variablen zu deklarieren. Name und Wert werden mit Hilfe von Elementattributen beschrieben. Variablen und Parameter können mit Hilfe des $ Operators aufgerufen werden.[3]

```
<xsl:template match="/">
      <xsl:variable name="VariableName" select="VariableWert"/>
      <xsl:value-of select=\$VariableName/>
</xsl:template>
```

2.2.2.3 Kontrollstrukturen

- For-each

  Die for-each Anweisung wird benutzt um namensgleiche Elemente in einem Quelldokument der Reihe nach abzuarbeiten. In dem Beispiel sucht der Transformationsprozessor alle Elemente mit dem Namen *Element* und gibt den Textinhalt in der *ElementValue* Struktur aus.

```
<xsl:template match="/">
    <xsl:for-each select="Element">
        <ElementValue>
            <xsl:value-of select="Element"/>
        <ElementValue>
    </xsl:for-each>
</xsl:template>
```

- If

  Die If-Anweisung legt eine Bedingung fest, unter der Strukturen in das Ausgabedokument eingefügt werden. Dabei wird die Bedingung in einem Attribut festgelegt.[3]

```
<xsl:template match="/">
    <xsl:variable name="VariableName" select="true"/>
    <xsl:if test="\$VariableName = ’true’">
        <xsl:value-of select="\$VariableName"/>
    </xsl:for-each>
</xsl:template>
```

- Choose

Diese Anweisung ergänzt die If-Kontrollstruktur durch eine weitere Verzweigung nach der Bedingung.[3]

```
<xsl:template match="/">
    <xsl:choose>
        <xsl:when test="Bedingung">
        [...]
        </xsl:when>
        <xsl:otherwise>
        [...]
       </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

2.2.3 Funktionen

XSL- Funktionen können innerhalb der XSL- Anweisungen eingefügt werden. Als Beispiel soll hier nur eine Funktion dienen um das Prinzip verständlich zu machen. Die Funktion *substring* wird innerhalb der Anweisung *xsl:variable* aufgerufen und bei der Transformation ausgeführt. Wie bei den Anweisungen stellt XSL-T eine umfangreiche Auswahl von verschiedenen Funktionen zur Verfügung. [3]

```
<xsl:variable name="testvar2" select="substring(\$testvar1,1,1)"/>
```

2.2.4 XPath

XPath ist eine Abfragesprache, um Bestandteile eines XML Dokumentes zu adressieren. Dabei wird das Dokument als ein Baum gesehen, in dem die Bestandteile die Knoten darstellen.

Ein XPath Ausdruck besteht immer aus einem oder mehreren Lokalisierungsschritten. Ein Lokalisierungsschritt besteht aus der Angabe von Achsen, mit deren Hilfe durch das Dokument ausgehend von einem bestimmten Knoten navigiert werden kann. Zum Beispiel navigiert die Achse *child* zu den Kindelementen des gegenwärtigen Knotens. Der zweite Schritt eines Lokalisierungsschrittes ist der Knotentest. Dabei kann die Elementauswahl einer Achse weiter eingeschränkt werden.

Optional können auch Prädikate verwendet werden, um das Ergebnis der Lokalisierungsschritte weiter einzugrenzen. In Prädikaten können eine Vielzahl von Funktionen wie Zeichenkettenfunktionen, numerische Funktionen oder Knotenmengenfunktionen verwendet werden, um eine Knotenmenge weiter zu filtern. [3]

2.2.5 IIF- Item Information File

Ein Item Informations File (IIF) ist eine auf XML basierte Datei zur Beschreibung von Produkten und deren Komponenten. Dabei werden Produkte und Komponenten jeweils mit dem Element Item beschrieben, das sich wiederum in weitere Elemente gliedert.

Ein IIF sollte dabei üblicherweise aus einem Item bestehen, das das Produkt repräsentiert und zusätzlichen Item-Instanzen die die Komponenten des Produktes darstellen. Die Product Library kann allerdings auch mehrere Produkte in einer Datei verarbeiten.

Das erste Element einer IIF Datei ist dir XML Header. Der Beginn des IIF wird durch den *<IIF>* Tag angezeigt. Es folgen eine beliebige Anzahl von Itemelementen.

Ein Item besteht aus verschiedenen Teilbereichen:

- Administrative Information - *<administration>*

  - beinhaltet den Identifier so wie dessen Typ (z.B. Produkt, Komponente) sowie Stichworte die das Item eindeutig beschreiben Zusätzlich werden Rechte, Zugehörigkeiten und Dauerhaftigkeit festgelegt.

- File Information - *<fileInformation>*

  - beinhaltet Informationen über die physische Representation des beschriebenen Produktes oder Komponenten wie z.B. Pfad und Größe der Datei.

- Insertion Method Specification - *<commandOption>*

  Bestimmt das Verhalten der Product Library, wenn Ergänzungen oder Updates auf Item-Instanzen durchgeführt werden.

- Components Information - *<components>*

  Beinhaltet Verweise auf andere Item-Instanzen, die somit Komponenten des Items sind.

Jede Referenz wird durch ein <*component*> Tag abgegrenzt und durch einen Identifier beschrieben. Das referenzierte Item kann lokal vorhanden oder bereits in der der Product Library enthalten sein.

- Basic Metadata - <*parameters*>

  Beschreibt die allgemeinen Produktparameter wie z.B. die zeitliche und räumliche Einordnung sowie den Herstellungsprozess und Qualitätsparameter.

- Sepecific Metadata - <*specificParameters*>

Spezifische Parameter sind auf das Produkt zugeschnittene Informationen. Die Struktur diese Bereiches ist aufgrund der verschiedenartigen Produkte nicht festgelegt. Informationen werden über <*feature*> Tags gespeichert wobei ein Attribut mit dem Namen *key* den Parameternamen festlegt und das darunterliegende Textelement den Parameterwert trägt.[5]

2.3 Merkmale der generierten Metadaten

Die Metadaten sind Produkt eines Teils der Prozesskette. Die Speicherung erfolgt über eine wohlgeformte XML-Datei, in der das Produkt durch produktspezifisches XML-Vokabular beschrieben wird.

```
<item>
    <Product_Name>S4(CA)</Product_Name>
    <Product_creation_time>07-04-2009 09:48:19</Product_creation_time>
    <creation_data_center>
        <institute>German Aerospace Center, DFD</institute>
        <address>Kalkhorstweg 53, 17235 Neustrelitz, Germany</address>
        <phone>+493981_xxxxx</phone>
        </Principal_Investigator>
          </creation_data_center>
</item>
```

In dem stark gekürzten Beispiel wird das Produkt durch das Element *item* dargestellt und durch weitere Elemente genauer beschrieben.

Die Eigenschaften von Metadatendokumenten können stark variieren. Somit ist es nicht möglich, ein einheitliches Vokabular bzw. eine einheitliche Struktur zu benennen. In Absprache mit dem DLR konnten jedoch Gemeinsamkeiten festgelegt werden. Die Speicherung erfolgt ohne die Verwendung von Namensräumen oder Attributen. Die Metadaten werden durch XML- Elemente klassifiziert und in Elementinhalten als Text gespeichert Außerdem gibt es keine gleichnamigen Elemente auf einer Ebene im XML-Baum.[5]

2.4 Transformation der XML- Metadaten durch XSL-T

Die Lösungsstrategie des DLR zur Umformung von Metadaten in das IIF Format setzt auf die Verwendung eines XSLT- Stylesheets. Das Stylesheet wird in die Prozesskette zur automatisierten Produktgenerierung eingefügt und transformiert mit Hilfe eines ebenso integrierten XSL-Prozessors die vorher generierten XML- Metadaten dynamisch in in die IIF- Spezifikation.

Das Stylesheet wird von den Mitarbeitern mit Hilfe unterschiedlicher Hilfsmittel erstellt. Dies umfasst z.B. den Einsatz von einer Reihe von XML- Editoren sowie verschiedener XSL- Prozessoren. Es besteht keine einheitliche und effiziente Entwicklungsumgebung, die die unterschiedlichen benötigten Funktionen zum Erstellen eines XSL- Transformationsprozesses vereint.

Bei diesen Funktionen handelt es sich zum Einen um die einfache Gestaltung des XSL- Transfortmationsprozesses. Häufig verwendete Anweisungen und Strukturen müssen umständlich eingefügt und editiert werden. Zum Anderen sind mit Funktionalität die verschieden Aufgaben gemeint, für die XLS- Prozessoren eingesetzt werden. Das XSL- Stylesheet muss während des Erstellens auf Gültigkeit überprüft werden. Zudem muss zu Testzwecken die Transformationen mit Parametern ausgeführt werden und das Ergebnis der Transformation angezeigt und gespeichert werden können.

Der Arbeitsprozess beim Erstellen eines Transformationsprozesses auf Basis der XSL- Spezifikation besteht aus folgenden Teilschritten.

- Erstellen eines XSL- Stylesheets

- Einfügen der IIF- Struktur

- Einfügen von XSL- Anweisungen in die IIF Struktur

Häufig genutzte Arbeitsschritte, die ineffizient ausgeführt werden, sind dabei u.a. das Einfügen von dynamischen IIF- feature Elementen sowie dynamischen Abfragen auf das XML- Metadatendokment über die XSL- Anweisung *value-of*.

Durch Analyse von bereits im System verwendeten Tranformationsprozessen konnte der bisher verwendete Funktionsumfang auf Basis der XSL-T Spezifikation wie folgt eingegrenzt werden:

- Parameter
  ```
  <xsl:param name="rid" />
  ```

  Die Parameterfunktion dient der Übergabe von Parameterwerten an den XSLT- Prozessor. Somit können neben dem Quelldokument noch weitere Informationen an den Transformationsprozess übergeben werden. Die Übergabe erfolgt von dem jeweiligen Prozess, der den XSLT- Prozessor aufruft.

- Variablen
  ```
  <xsl:template match="/">
    <xsl:variable name="date" select="//Product_creation_time" />
    <xsl:variable name="Monat" select="substring($date, 4, 3)" />
  ```

  Mit Hilfe der Deklaration von Variablen werden verschiedene Funktionen eingebettet. In diesem Beispiel wird die Variable *date* aus der Quelldatei ausgelesen und durch die Deklaration der Variablen *Monat* weiter formatiert. Variablen dienen hier der strukturellen Übersichtlichkeit da die erwähnten Funktionen auch innerhalb der Struktur der Ausgabedatei verwendet werden kann.

- Value-of

```
<id>
   (rid://
   <xsl:value-of select="$rid" />
   )
</id>
```

Die Funktion *Value-of* wird verwendet, um Daten aus Parameter oder Variablen auszulesen. Zusätzlich können über X-Path Ausdrücke Elemente aus der Quelldatei adressiert und ausgelesen werden. Die ausgelesen Daten werden anschliessend in die Struktur des Zieldokumentes überführt. In dem Beispiel ergibt die Funktion einen Textinhalt unter dem Element *id*, das sich aus dynamischen und nichtdynamsichen Text zusammensetzt.

- If

```
<xsl:if test="$test_dims = '-'">
   <feature key="product_creation_time">
      <xsl:value-of select="$creation_time" />
   </feature>
</xsl:if>
```

Die Kontrollstruktur wird eingesetzt, um in Abhängigkeit einer Bedingung dynamisch eine Struktur in das Zieldokument einzufügen. In der XML- Hierarchie unter dem Anweisung xsl:if stehende Elemente werden bei der Erfüllung der Bedingung vom XSL-Prozessor weiter ausgeführt. Im Beispiel wird die XSL- Variable $test_dimsausgewertet$.

- Output

```
<xsl:output encoding="UTF-8" method="xml" version="1.0"/>
```

Die XSL- Anweisung *xsl:output* definiert das Format des Zieldokumentes. Dies geschieht über eine Reihe von verbindlichen und freiwilligen Optionen, die mit Hilfe von Attributen dargestellt und bei Aufruf an den XSL- Prozessor übergeben werden können. In dem Beispiel wird festgelegt, dass das Zieldokument eine XML- Struktur nach der XML- Spzifikation 1.0 besitzt. Zusätzlich wird die Zeichenkodierung des Zieldokumentes festgelegt.

3 Aufgabenstellung

Die Aufgabenstellung des DLR umfasst das Erstellen einer grafischen Oberfläche mit der ein Nutzer Transformationsprozesse zwischen XML- Metadaten und IIF- Metadaten generieren und validieren kann. Dieser Transformationsprozess muss kompatibel mit den im DIMS verwendeten JAVA- Prozessen sein. Die Transformation muss sich einfach in die Produktgenerierung integrieren lassen. Optional soll der verwendete Transformationsprozess für die Transformation in andere noch nicht näher definierte Metadatenspezifikationen sein.

Diese allgemeine Aufgabenstellung lässt sich in 3 Teilaufgaben zerlegen.

- Bestimmung des Transformationsprozess

  Auf Basis der Analyse des derzeitigen Lösungsansatzes soll eine Möglichkeit benannt werden, wie der Transformationsprozess abgebildet werden kann.

- Erstellen der Funktionalität der grafischen Benutzeroberfläche

  Die Funktionen zum Generieren und Validieren des Transformationsprozesses müssen definiert werden. Der gewählte Funktionsumfang orientiert sich dabei an der Analyse der Lösungsstrategie des DLR. Die Gestaltung der Funktionen soll sich nach den bereits beschriebenen Arbeitsprozessen richten und diese, soweit möglich, den zeitlichen Aufwand betreffend, optimieren.

- Gestaltung der grafischen Nutzeroberfläche

  Der finale Schritt ist das Einbinden der Funktionalität in eine grafische Nutzeroberfläche. Dazu müssen der Transformationsprozess und die XML- Metadaten visualisiert werden. Auf Basis dieser Visualisierung soll der Nutzer die Funktionalität aufrufen.

  Ein weiteres Kriterium für die grafische Nutzeroberfläche ist die intuitive und effiziente Bedienung. Die Funktionalität soll über eine Nutzerhilfe erklärt werden. Die grafische Nutzeroberfläche soll zudem Warnungen und Bestätigungen beim Aufruf von Funktionen an den Nutzer ausgeben.

4 Anforderungen

4.1 Transformationsprozess

Der Transformationsprozess muss über einen unabhängigen Prozessor verfügen, der sich in die bestehenden DIMS- Prozesskette integrieren lässt. Das setzt eine Implementierung des Prozessors in JAVA voraus. Dem Prozessor muss in der Lage sein neben dem XML- Metadatendokument und dem Transformationsprozess auch zusätzliche Parameter entgegenzunehmen. Ergebnis der Verarbeitung soll ein unabhängiges neues Dokument in Dateiform sein.

Der Transformationsprozess muss Anweisungen und Funktionen zur Verfügung stellen, die es ermöglichen, auf Elemente in XML- Dokumenten zuzugreifen, diese auszulesen, zu formatieren und anschließend in einer neuen Metadatenstruktur auszugeben.

Die Anforderungen lassen sich in folgende Punkte gliedern:

- Adressierbarkeit von XML- Elementen

  Elemente innerhalb der XML- Dokumente wie z.B. Text oder Attribute müssen adressierbar sein. Neben dem eigentlichen Transformationsprozess muss also eine Abfragesprache zur Verfügung stehen.

- Abfrage und Manipulation von Daten

  Der Transformationsprozess sollte über einen großen Funktionsumfang verfügen, um adressierte Daten auszulesen und dynamisch manipulieren zu können. Die minimalen Anforderungen sind dabei die im vorherigen Kapitel herausgearbeiteten Funktionen. Ein größerer und somit flexibler Umfang an Funktionen wäre allerdings im Hinblick auf die Flexibilität und Erweiterbarkeit der Entwicklungsumgebung wünschenswert.

- Übersetzung der Daten

  Die ausgelesen und formatierten Daten sollen in eine vom Nutzer definierte Struktur eingefügt werden, die dann als eigenständiges Metadatendokument vom Transformationsprozessor ausgegeben wird. Die dabei definierte Struktur muss im Transformationsprozess eingebettet sein.

4.2 Benutzeroberfläche

Eine Anforderung an die Benutzeroberfläche ist die Visualierung der zu bearbeitenden Daten. Zum Einen muss der Transformationsprozess übersichtlich und möglichst einfach dargestellt werden. Zum Anderen ist es auch notwendig, die zu verwendenden Metadaten anzuzeigen, da der Transformationsprozess auf dieser Basis vom Nutzer gestaltet wird.

Um Zugriff auf die Funktionalität zu erlangen, müssen Schnittstellen zwischen Komponenten der grafische Benutzeroberfläche sowie den Funktionen eingerichtet werden. Diese sollen so gestaltet werden, dass der Zugriff für den Nutzer einfach und effektiv ist.

4.3 Funktionalität

Die Anforderungen an die Funktionalität der Benutzeroberfläche orientieren sich an der Aufgabenstellung des DLR sowie der vorausgegangen Analyse des zu optimierenden Arbeitsprozesses.

Um die Wiederverwendbarkeit zu gewährleisten müssen zunächst Funktionen zum Laden und Speichern des bearbeiteten Transformationsprozesss und der XML- Metadaten innerhalb des Dateissystems eingerichtet werden.

Die Funktionalität muss zudem aus den intern vorgehaltenen Daten ein Datenmodell zur Visualisierung ableiten können, welches den entsprechenden Komponenten der grafischen Nutzeroberfläche übergeben werden kann.

Darüberhinaus müssen Funktionen eingerichtet werden, die den Arbeitsprozess beim Gestalten des Transformationsprozesses wiederspiegeln. Dazu gehören u.a. Editierfunktionen über einem allgemeinen XML- Dokument. Des weiteren soll es möglich sein, die bei der Analyse gezeigten XSL- Anweisungen einzufügen und zu editieren. Zusätzlich sollen auch Funktionen verfügbar sein, oft verwendete Arbeitsschritte abkürzen.

Zur Funktionalität gehört zudem die Verwendung eines Transformationsprozessors. Dieser muss in der Lage sein, Transformationen mit Hilfe des gestalteten Transformationsprozesses und der angezeigten Produkt- Metadaten auszuführen. Die Funktion soll ebenso ein Datenmodell an die grafische Nutzeroberfläche übergeben können, um das Ergebnis der Transformation anzuzeigen. Das Ergebnis soll auch innerhalb des Dateisystem gespeichert werden können. Die Eingabe von Parametern, die dem Prozessor übergeben werden, ist ebenfalls erforderlich.

5 Bestimmung des Transformationsprozesses

Der erste Schritt zur Erfüllung der Aufgabenstellung ist die Bestimmung eines Transformationsprozesses. Dieser Prozess soll die Umwandlung von dem im DIMS erzeugten Metadaten in das IIF-Format und andere Metadatenformate beschreiben können. Dabei wurde letztendlich wieder auf die bereits vom DLR genutzte Lösungsstrategie zurückgegriffen. Um den Prozess abzubilden wird XSL-T als beschreibende Sprache eingesetzt. Dies erleichtert auch den Umstieg auf die grafische Nutzeroberfläche da den meisten Nutzern der Umgang mit XSL bereits vertraut ist.

Mit XSL-T ist es möglich, XML-Strukturelemente im Metadatendokument zu adressieren, auszulesen, zu manipulieren und in einem neuen Format (Text oder XML) dynamisch einzufügen. Zudem können neben Quelldokument und XSL-Stylesheet zusätzlich Parameter in die Transformation übergeben werden. Für die Sprache sind eine Vielzahl von XSL-Prozessoren verfügbar, die mit wenig Aufwand in einen DIMS-Prozess implementierbar sind.

Ein weiterer Vorteil ist die gemeinsame Struktur von Metadaten und XSL-Stylesheet. Beide basieren auf der XML-Syntax, was den Aufwand der Verarbeitung und Visualisierung bei der Implentierung verringert.[6]

6 Programmentwurf

Der Entwurf beschreibt konkrete Entscheidungen hinsichtlich der Gestaltung der Applikation. Dabei wird zunächst gezeigt, wie XML- Dokumente innerhalb der grafischen Benutzeroberfläche visualisiert werden und wie die Schnittstellen zwischen Nutzer und GUI gestaltet sind. Im nächsten Schritt werden die benötigen Funktionen zur Bearbeitung eines XSL- Stylesheets aufgelistet und beschrieben. Zusätzlich wird auch auf den inhaltlichten Entwurf der Nutzerhilfe eingegangen.

Im nächsten Schritt wird aufgezeigt, mit welchen Hilfsmitteln der Entwurf der Funktionalität und der Benutzerobefläche umgesetzt werden soll. Dabei wird sowohl auf die Entwicklungsumgebung eingegangen als auch auf die Programmierschnittstellen, die später verwendet werden sollen.

6.1 Visualisierung XML/XSL innerhalb der GUI

An erster Stelle beim Entwurf der grafischen Benutzeroberfläche steht die Repräsentation der zu verarbeitenden Daten. Diese sollten soweit aufbereitet sein, dass der Nutzer ohne viel Aufwand in den Daten navigieren und wesentliche Eigenschaften der Daten mittels grafischer Hilfsmittel schnell erfassen kann.

Die Darstellung von XML- bzw. XSL- Dokumenten wird durch eine visuelle Baumstruktur rea-

lisiert. Dabei stellt jedes XML- Element einen Knoten dar. Die Baumstruktur hat den Vorteil, dass sie die hierarchischen Strukturen eines XML- Dokumentes exakt wiedergeben kann. Jedes XML- Element kann in die Hierarchie eingeordnet werden. Zwischen allen Elementen bestehen Beziehungen, die sich in einem Baum abbilden lassen. Da die Hierarchie der Elemente durch die Baumstruktur wiedergegeben wird, kann bei der Darstellung auf Markup- Zeichen verzichtet werden. Dies trägt zusätzlich zur Übersicht bei.

Ein weiteres Hilfsmittel ist die farbliche Darstellung der unterschiedlichen Typen von Elementen. Dies hat für den Nutzer den Vorteil, Strukturen schneller wahrzunehmen und erleichtert die Navigation in dem Baum. Abb. 6.1 zeigt ein Beispiel für die Darstellung eines XML-Dokumentes mit Hilfe einer Baumstruktur und farblicher Formatierung in Abhängigkeit des Elementtyps.

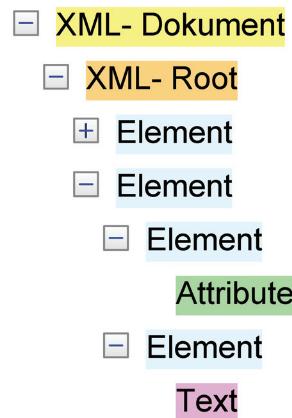

Abbildung 6.1: XML- Visualsierung über Baumstruktur

Die Alternative zu der Darstellung der Daten als Baum ist die reine Textdarstellung. Hier ist es zwar auch möglich über verschiedene Hilfsmittel wie hierarchieschse Darstellung und farbliche Formatierung eine effektive und nutzerfreundlich Repräsentation zu schaffen. Der Nachteil liegt in der teilweise nicht möglichen Trennung von Kindknoten und ihren Elternknoten. Dies ist z.B. der Fall, wenn versucht wird, Attribute und Textinhalten als Knoten auf einer Hierarchieebene darzustellen. Die wird durch das Markup unnötig verkompliziert und erschwert eine übersichtliche und einheitliche Darstellung. Die hierarschische Darstellung im Baum ermöglicht

das Weglassen von Markupzeichen sowie Starttag und Endtag. Nur der Name des Elementes wird dargestellt.

```
<Xmlroot>
    <Element>Text</Element>
    <Element AttributenName="AttributeWert">Text</Element>
</Xmlroot>
```

6.2 Gestaltung der Benutzeroberfläche und Schnittstellen

Die Gestaltung der Benutzeroberfläche erfolgt nach dem WIMP- Konzept, dem derzeit dominierenden Grundkonzept moderner grafischer Benutzerschnittstellen. WIMP bedeutet dabei die Verwendung von Windows, Icons, Menüs und Pointer. Die Verwendung des Konzeptes soll eine intuitive Bedienung des Programmes ermöglichen.[12]

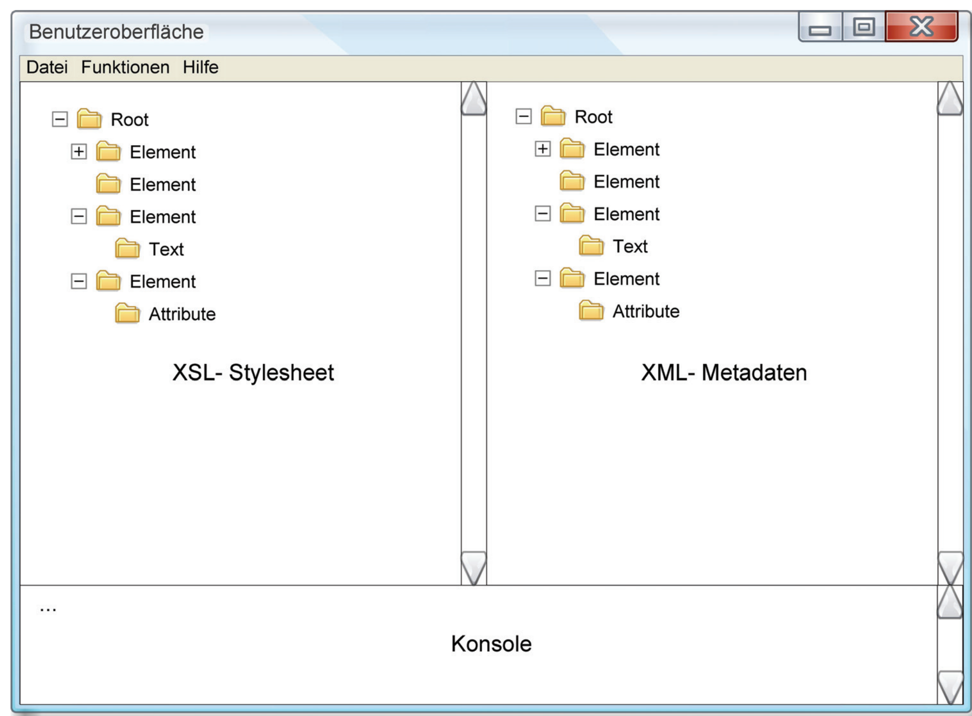

Abbildung 6.2: Entwurf der Benutzeroberfläche

Abb. 6.2 beschreibt den Entwurf der grafischen Nutzeroberfläche. Funktionen wie Laden/Speichern von XML- Dokumenten, die Benutzung des XSL- Prozessors sowie eine Nutzerhilfe sind innerhalb der oberen Menüleiste integriert. Im linken Fenster unter der Menüleiste wird das XSL- Stylesheet visualsiert und bearbeitet. Im rechten Fenster wird das XML- Metadatendokument visualsiert.

Zur Anzeige von Meldungen befindet sich im unteren Teil des Fensters eine Konsole die nur zur Ausgabe dient. Hier sollen Fehlermeldungen angezeigt werden wie sie z.B. im Umgang mit dem XSL- Prozessor bei fehlerhaften XSL- Stylesheets anfallen. Mit Hilfe der Konsole können aber auch Status- und Erfolgsmeldungen sowie Fortschrittsanzeigen ausgegeben werden. Dies steigert die Nutzerfreundlichkeit.

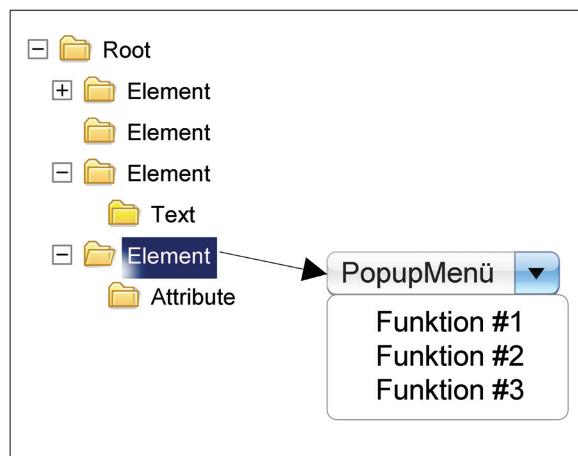

Abbildung 6.3: Zugriff auf die Baumknoten

In Abb. 6.3 ist zu erkennen, wie der Zugriff auf Funktionen über einem XML- Element in den XML- Bäumen erfolgen soll. Über eine Mauseingabe wird eine Element angewählt und ein Popupmenü aufgerufen. In dem Popup- Menü sind Funktionen verfügbar, die sich direkt auf das ausgewählte Element beziehen. Zusätzlich sollen Funktionen nach Auswahl des XML- Elementes auch direkt über eine Tastatureingabe aufgerufen werden können, ohne einen weiteren Dialog über ein Popup- Fenster zu starten. Dies soll die Effizienz bei der Benutzung häufig verwendeter Funktionen steigern.

6.3 Funktionen

Die Funktionen die nötig sind um ein XSL- Stylesheet zu editieren sind:

- Erstellen von XML- Elementen

  Nach Auswahl eines Elementes, soll es möglich sein, einen Kindelement zu erzeugen. Dabei handelt es sich um die XML- Elementtypen Element, Attribut oder Text. Die Eingabe der Elementnamen, Attributenamen und -werte sowie der Textinhalte erfolgt über einen Dialog außerhalb des visualisierten Baumes.

- Editieren von XML- Elementen

  Nach Auswahl eines Baumknotens soll dieser je nach Elementtyp editierbar sein. Bei Elementen soll der Name veränderbar sein, bei Attributen der Attribname und Attributwert und bei Text der Textinhalt.

- Löschen von XML- Elementen

  Elemente, Attribute und Textinhalte sollen nach Auswahl des jeweiligen Knotens löschbar sein. Bei Elementen sollen auch die darunterliegenden Kindsknoten vollständig gelöscht werden.

- Verschieben von XML- Elementen

  Es soll möglich sein, die Reihenfolge von Text und Attributen auf der gleichen Baumebene zu verändern. Dazu muss ein Text- oder Attributeknoten verschiebbar sein. Dies soll eine Funktion ermöglichen, die nach Auswahl zweier Baumknoten einen Tausch beider Knoten vornimmt.

- Kopieren von XML- Teilbäumen

  Diese Funktion soll einen ausgewählten Knoten in eine Zwischenablage kopieren, um ihn unter einem anderen, danach ausgewählten Knoten als Kopie einfügen zu können. Dabei sollen auch alle Kindsknoten bzw. ein Teilbaum mitkopiert werden.

- Einfügen von kopierten XML- Teilbäumen

  Der in der Zwischenablage befindliche Teilbaum kann unter einem ausgewählten Element eingefügt werden.

- Einfügen von XSL- Anweisungen

  Der Umfang der XSL- Anweisungen beinhaltet zunächst nur die vom DLR genutzten Anweisungen (siehe Lösungsansatz des DLR). Nach Auswahl eines Elternknotens ist es möglich, die XSL- Anweisungen *Variable, Parameter, If, Value-Of, Output* einzufügen. Die Eingabe der jeweiligen Parameter der Anweisungen erfolgt über einen Dialog außerhalb des visualisierten Baumes.

Zusätzlich ist es notwendig, verschiedene Funktionen über dem Metadatendokument zu benennen:

- Kopieren von XML- Teilbäumen

  Ein ausgewählter Baumknoten des XML- Metadatendokumentes soll nach Aufruf der Kopierfunktion in eine Zwischenablage kopiert werden. Dies beinhaltet auch den unter dem Knoten liegenden Teilbaum. Dies erlaubt das Kopieren von Teilbäumen aus anderen XSL- Stylesheets und steigert somit die Wiederverwendbarkeit dieser Dokumente.

- Abfragen von Pfaden

  Nach Auswahl eines Baumknotens und Aufruf der Abfragefunktion soll ein XPath- Ausdruck ausgegeben werden, der auf den ausgewählten Baumknoten zeigt. Dies erleichtert dem Nutzer das dynamische Auslesen von Elementen aus dem Metadatendokument wenn dieses durch viele Baumebenen schwer überschaubar ist.

- Einfaches Hinzufügen von Features

  ```
  <feature key="Format"> ASCII </feature>
  ```

  Das Listing zeigt das *feature*- Element, welches in der IIF- Spezifikation sehr häufig verwendet wird. Das Erstellen eines dynamischen Features ist ein Prozess, die aus mehreren

Teilschritten besteht. Zunächst wird ein Element mit dem Namen *feature* erstellt. Dann wird ein Attribute mit dem Attributnamen *key* an dieses Element angehängt. Der Wert des Attributes wird aus dem angewählten Element im Metadatendokument abgeleitet, es ist der Name des Elementes. Der Textinhalt wird anschliessend über eine *value-of* Anweisung dynamisch aus dem Metadatendokument ausgelesen. Dazu wird zunächst der Pfad zu dem Textinhalt des ausgewählten Elementes ermittelt. Das Ergebnis im Stylesheet ist im folgenden Listing zu sehen.

```
<feature key="Format"> ASCII
   <xsl:value-of select = item\Format>
</feature>
```

Die Funktion zum automatischen Erstellen dieser dynamischen Abfragen muss die beschriebenen Arbeitsschritte automatisch bearbeiten.

- Einfaches Hinzufügen von Value-Of

  Nach Auswahl eine Knotens im Metadatenbaum und Aufruf dieser Funktion soll eine dynamische Abfrage des Textinhaltes des Elementes in den Stylesheetbaum übertragen werden. Dies dient wiederum der Effizienz, da es sich wie bei dem Hinzufügen von Features aus den Metadaten um eine häufig genutzte Funktion beim Erstellen eines XSL-Stylesheets handelt.

Der Nutzer soll außerdem innerhalb Benutzeroberfläche auf einen XSL- Prozessor zugreifen können. Dieser soll das XSL- Stylesheet, welches sich in Bearbeitung befindet, auf Fehler prüfen und diese auch ausgeben können. Zusätzlich soll er in der Lage sein eine Transformation auszuführen und die Ergebnisse entweder innerhalb der Benutzeroberfläche anzuzeigen oder in einer externen Datei zu speichern.

Um mit den XML- Dokumenten arbeiten zu können, muss dem Nutzer eine Funktion zu Verfügung gestellt werden, mit der er auf vorhandene Dokumente auf Datenträger zugreiffen kann. Um die Wiederverwendbarkeit zu gewährleisten müssen die erstellten und bearbeitenden XSL- Stylesheets als Dateien abgespeichert werden können. Außerdem soll eine Funktion existieren, die eine neues XSL- Stylesheet generiert.

6.4 Nutzerhilfe

Die Gestaltung der Nutzerhilfe soll durch HMTL erfolgen. Dies erlaubt eine Vielzahl von Mitteln zu Gestaltung der Hilfedokumente und erleichtert die Navigation in ihnen. Der Aufruf der Hilfe erfolgt über die Menüleiste. Die Hilfsdokumente werden in einem neuen Fenster dargestellt, das als eine Art Browser zur Darstellung der HTML- Seiten dient.

Die Hilfsdokumente listen die in der Benutzeroberfläche enthaltenen Funktion auf und beschreiben diese jeweils kurz. Zusätzlich wird auch auf die Auszeichnungsprache XML eingegangen. Dabei soll kurz die Struktur und Komponenten erklärt werden.

Neben XML soll auch die Funktionsweise von XSL kurz dargestellt werden. Dazu gehören die im Rahmen dieser Arbeit relevanten XSL- Anweisungen sowie XSL- Funktionen.

6.5 Entwicklungsumgebung

Die Wahl der Programmiersprache, um den Programmentwurf umzusetzen, fiel auf die objektorientierte Sprache Java. Eine entscheidene Eigenschaft der Sprache ist dabei die Plattformunabhängigkeit der Java-Programme, da sie in einer vorher installierten Java- Laufzeitumgebungen ausgeführt werden.

Ein weiteres Argument ist erforderliche Erweiterbarkeit und Wartbarkeit durch das Personal des DLR. Da dies im Rahmen des DIMS weitgehend mit Java arbeitet, können so Verbesserungen am Programm einfacher und schneller durchgeführt werden.

Ein weiterer Vorteil bietet die umfangreiche Java- API, speziell für die Verarbeitung von XML-Dokumenten. So ist beispielsweise die vom World Wide Web Consortium (W3C) spezifizierte Schnittstelle Document Object Model in der Java- API implementiert. Diese Schnittstelle

ermöglicht die Verarbeitung vom HTML- und XML Dokumenten. Auch das Parsen von XML-Dokumenten und das Ausführen von XSL- Transformationsprozessen wird von der API untersützt.

6.6 Verarbeitung XML/XSL in JAVA

Um den Entwurf der Funktionalität umzusetzen, müssen zunächst Programmierschnittstellen benannt werden, die die grundlegenden Funktionen zur Verfügung stellen. Wie bereits vorgestellt, wird für die interne Speicherung und Verarbeitung der XML- und damit auch der XSL-Dokumente die Schnittstelle Document Object Model verwendet. Desweiteren werden Teile der JAVA- API benannt, die XML- Dokumente parsen und ein Document Object Model erzeugen sowie einen XSL- Prozessor bereitstellen können. Ebenfall muss eine Möglichkeit benannt werden wie die XML- Dokumente visuell aufbereitet werden können.

6.6.1 Visualisierung

Um ein XML- Dokument getreu dem Entwurf visualisieren zu können, wird die Java- Klasse *Jtree* verwendet. Diese kann Baumstrukturen wie ein XML- Dokument wiedergeben. Die Darstellung setzt das Erzeugen eines Datenmodells voraus, welches visualisiert wird. Das wird mit Hilfe der Klasse *DefaultTreeModel* realisiert.

6.6.2 XML- Parser

Um XML- Dokumente zu parsen wird die API *javax.xml.parsers* verwendet. Diese stellt eine*DocumentBuilderFactory* zur Verfügung, die Objekte generiert, die XML- Dokumente parsen und daraus ein Document Object Model erzeugen kann.

6.6.3 Document Object Model

Die DOM- API repräsentiert ein XML- Dokument mit Hilfe einer Baumstruktur. Dabei wird jedes Element des Dokumentes als Knoten (engl.: Node) angesehen. Die verschiedenen Knotentypen sind u.a.

- Dokumentknoten- bildet den gesamten Baum ab
- Dokumentfragmentknoten - bildet Teilbäume ab
- Elementknoten - entspricht einem Element des XML- Dokumentes
- Attributknoten - stellt ein Attribut eines Elementes dar
- Textknoten - stellt den Textinhalt eines Elementes dar

Nach Erzeugung des Document Object Models kann auf verschiedene Funktionen zugegriffen werden. Die API ermöglicht die Navigation zwischen Knoten sowie die Suchanfragen in dem Baum. Es ist außerdem möglich, Information über den verschiedenen Knoten des Baumes auszulesen wie z.B. Elementnamen oder Textinhalte. Zum Funktionsumfang gehört letztendlich auch die Manipulation des im Document Object Model abgebildeten XML- Dokumentes. Baumknoten können geändert, gelöscht und neu generiert werden.

Damit bildet der Funktionsumfang der DOM- API eine Grundlage zu Umsetzung der vorher definierten Anforderungen. Da XSL- Anweisungen auf der XML- Spezifikation basieren ist es möglich, diese mit via DOM zu gestalten.

Für die Implementierung des Entwurfes wird hier die DOM- Implementierung *org.w3c.dom* verwendet. [2]

6.6.4 XSL- Prozessor und Serialisierung von DOM

Um die generierten XSL-T Stylesheets auszuführen, wird der XSL- Prozessor aus *javax.xml.transform* verwendet. Dieser ist in der Lage u.a. auch DOM- Objekte zu verarbeiten und weitere Parameter an den Transformationsprozess anzunehmen. Die Ergebnisse können in u.a. in Dateiform ausgegeben werden. Dies löst gleichzeitig das Problem der Serialisierung der DOM- Objekte in Dateien da der Prozessor auch Transformationen ohne XSL- Stylesheet ausführen kann. Dabei wird eine Kopie des transformierten Objektes erstellt.[2]

7 Beschreibung der Implementierung

Die Beschreibung der Implementierung zeigt die Umsetzung des Entwurfes in Programmcode. Dabei wird auf die jeweiligen Komponenten im Speziellen und auf deren Interaktion mit anderen Komponenten eingegangen.

7.1 Programmarchitektur

Die Applikation besteht aus verschiedenen Teilkomponenten. Eine Komponente ist die die grafische Nutzeroberfläche. Diese kann wiederum in verschiedene Bereiche unterteilt werden:

- Visualisierung

  Die Visualisierungskomponente zeigt die ihr zur Verfügung gestellten Daten an und gibt diese, je nach Inhalt der übergebenen Daten, formatiert aus. Zudem soll der Nutzer über die Visualisierung auch Eingaben durchführen bzw. weitere Funktionen aufrufen können.

- Nutzerdialoge

  Nutzerdialoge sind Menüs oder zusätzliche Eingabemasken, über die der Nutzer Funktionen aufrufen und weitere Informationen angeben kann.

- Eventlistener

Eventlistener reagieren auf bestimmte Aktionen der GUI- Objekte und rufen dann Funktionen auf.

Die andere Hauptkomponente ist die Funktionalität. Sie beinhaltet die Daten eines XML-Dokumentes und stellt verschiedene Funktionen bereit, um mit dem Dokument arbeiten zu können. Diese sind:

- Funktionen im Dateisystem (Laden/Speichern von Dokumenten)
- Rückgabe eines Datenmodells zur Visualisierung
- Manipulation von Dokumenten
- XSL- Prozessorfunktionen

Das Zusammenspiel der Komponenten wird in Abb. 7.1 dargestellt. Der Nutzer tätigt die Eingaben über die Visualisierung oder die anderen Nutzerdialoge. Daraufhin wird der entsprechende Eventlistener aufgerufen, der wiederum die der Eingabe entsprechenden Funktion innerhalb der Funktionalität aufruft. Diese gibt dann das geänderte Datenmodell zurück sowie Fehler- oder Statusmeldungen.

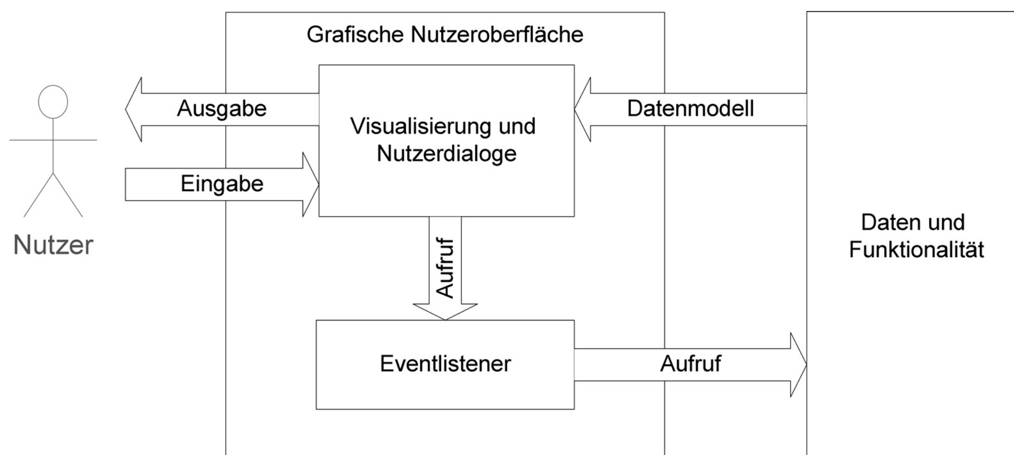

Abbildung 7.1: Zugriffe im Gesamtsystem

7.2 Verwendete Programmierschnittstellen

Alle verwendeten Schnittstellen finden sich in der Java Platform Standard Edition 6 API. Dabei handelt es sich um folgende Programmbibliotheken:

- java.awt, javax.swing

  Beide Bibliotheken stellen APIs zur Erzeugung der grafischen Nutzeroberfläche zur Verfügung.

- org.w3c.dom, javax.xml

  Diese Bibliotheken werden benutzt, um XML- Dokumente zu parsen, zu serialisieren und programmintern zu verarbeiten. Javax.xml stellt zudem einen XSL- Prozessor zur Verfügung.

7.3 Daten und Funktionalität

Die Funktionalität und die verwendeten Datenmodelle der Applikation sind innerhalb der Klasse *XMLDocument* zusammengefaßt. Jede Instanz der Klasse repräsentiert ein XML- Dokument und die Funktionen, die für das Dokument verfügbar sind. Die im Entwurf beschriebenen Funktionen werden durch die jeweiligen Methoden in der Klasse beschrieben.

7.3.1 Datenhaltung

Das XML- Dokument wird mit Hilfe des im Entwurf beschriebenen Document Object Model abbgebildet. Aus dem Document Object Model wird ein weiteres Datenmodell abgeleitet, welches der grafischen Nutzeroberfläche zur Visualisung übergeben wird. Dabei handelt es sich

um das *DefaultTreeModel*. Für beide Datenmodelle stehen die Methoden *getTreeModel()* und *getDocument()* zur Verfügung, um sie außerhalb des Objektes abrufen zu können.

Desweiteren ist in der Klasse ein Zeiger für das *DefaultTreeModell* verfügbar. Dieser Zeiger beschreibt, welcher Knoten im *DefaultTreeModel*- Objekt vom Nutzer zur Bearbeitung ausgewählt wurde. Auf diesen Zeiger greifen alle Methoden zum Editieren des Document Object Models zu. Jede Methode ruft zunächst den Zeiger auf, um den angezeigten Knoten dann zu bearbeiten. Der Zeiger kann von außerhalb über die Methode *setSelectedPath(TreePath selectedTreePath)* gesetzt werden. Wie schon an dem Methodenparameter erkennbar handelt es sich bei dem Zeiger um den Pfad im *DefaultTreeModel*, der zu dem angezeigten Knoten führt.

7.3.2 Speichern und Laden von XML- Dokumenten

Das Auslesen eines XML- Dokumentens aus dem Dateisystem erfolgt über die Methode *getDomFromFile(File file)*. Mit Hilfe des Parsers aus der Klasse *javax.xml.parsers.DocumentBuilder* wird ein Document Object Model aus einem gültigen XML- Dokument erzeugt.

Die Serialisierung wird über die Methode *writeDocumentToFile(File file)* realisiert. Dort wird der XSL- Prozessor aus *javax.xml.transform.Transformer* genutzt, um das Document Object Model zu serialisieren. Dem Prozessor wird dabei kein XSL- Stylesheet übergeben, so dass dieser das Datenmodell unverändert in eine Datei übersetzt.

7.3.3 Generierung des Datenmodells für die Visualisierung

Die rekursive Methode recurseXML in der Klasse *XMLDocument* füllt das *DefaultTreeModel* mit den *Node*- Objekten aus dem Document Objekt Model. Damit wird ein vollständiges Abbild des Document Object Model- Baumes geschaffen, bei dem jeder Element-, Attribut- oder Textknoten als eigenständiger Knoten im *DefaultTreeModel* referenziert wird.

Der Methode wird eine *Node* aus dem Document Object Model, den entsprechenden Knoten im *DefaultTreeModel* und das *DefaulTreeModel* selber übergeben. Das nachfolgende Listing zeigt die Funktionsweise der Methode. Beim Aufruf der Methode werden zunächst sämtliche Kindknoten des übergebenen Knotens in einer *CustomNode* gekapselt. Diese Objekte werden anschließend jeweils über eine *DefaultMutableTreeNode* referenziert und darüber in das *DefaultTreeModel* eingefügt. Bei Elementen funktioniert dies ähnlich nur mit dem Unterschied, dass sich für jedes Kindelement die Methode selber aufruft, um wiederum die Kindelemente zu dem *DefaultTreeModel* hinzuzufügen. Dies geschieht solange bis keine Kindelemente mehr vorhanden sind.

Listing 7.1: Rekursives Aufüllen des TreeModels aus dem Document Object Model

```
private void recurseXml(Node node, DefaultMutableTreeNode parentNode,
                    DefaultTreeModel treeModel)
{
  //Hinzufügen aller Attributknoten
  //[...]

  if(node.hasChildNodes())
  {
    NodeList nodeList = node.getChildNodes();
    for(int i=0; i<=(nodeList.getLength()-1); i++ )
    {
      Node currentNode = nodeList.item(i);
      //Hinzufügen von Textknoten
      //[...]

      //Rekursives Hinzufügen von Elementknoten
      if ((currentNode.getNodeType()==1))
      {
        DefaultMutableTreeNode newNode =
          new DefaultMutableTreeNode(new CustomNode(currentNode));
        treeModel.insertNodeInto(newNode,parentNode,parentNode.getChildCount());
        recurseXml(currentNode,newNode,treeModel);
        }
      }
    }
}
```

Der Grund für die Kapselung der Document Object Model- Objekte ist die textuelle Repräsentation der Objekte in der Visualisierung. Die zur Visualisierung ausgewählte Swing-

Komponente *JTree* greift bei der Darstellung der Knoten auf die im *DefaultMutableTreeNode* referenzierten Objekte zu und bedient sich deren *toString()* Methode, um den Knoten zu beschreiben.

Zur Lösung des Problemes werden die *Node*- Objekte in der Klasse *CustomNode* gekapselt und deren *toString()*- Methode mit einer sinnvolleren Variante überschrieben. Die Klasse wird mit Übergabe eines *Node*- Objektes an den Konstruktor instanziert. Der Abruf einer gekapselten *Node* erfolgt über die Methode *getNode()*.

Abbildung 7.3 verdeutlicht noch einmal die Zusammenhänge zwischen den verschiedenen Datenmodellen. Die Visualisierung in Form des *JTree* greift auf das *DefaultTreeModel* und stellt die verschiedenen *DefaultMutableTreeNode* dar. Die *DefaultMutableTreeNode* wiederum referenziert die jeweilige CustomNode. Diese kapselt das Doccument Object Model in Form des *Node*- Objektes.

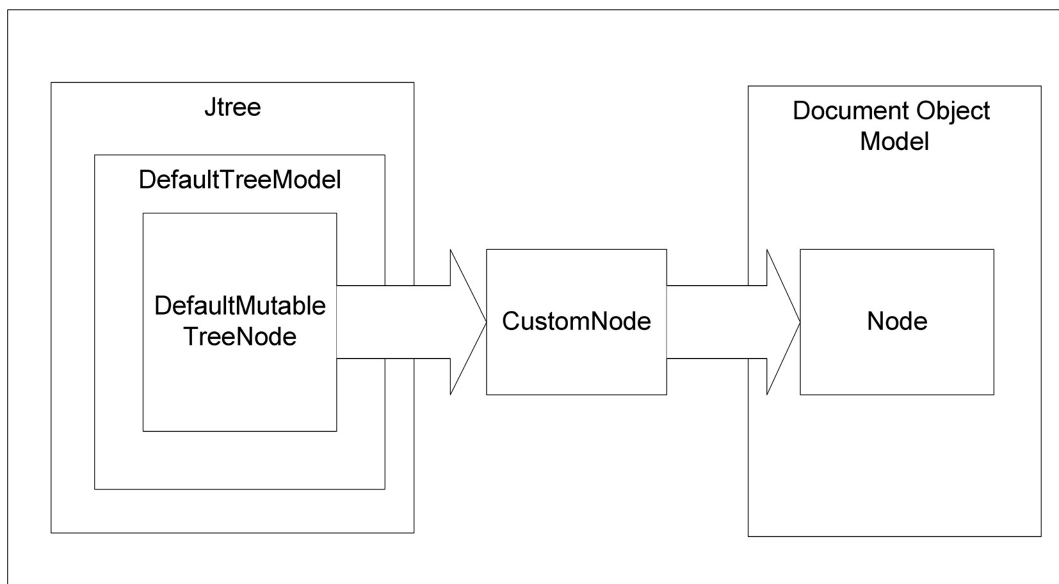

Abbildung 7.2: Visualsierung des Document Object Model

7.3.4 Editierfunktionen

Die Editierfunktionen umfassen sämtliche Funktionen, die Änderungen an dem XML- Dokument in Form des Document Object Models und somit auch des *DefaultTreeModels* vornehmen. Jede Methode repräsentiert dabei eine oder mehrere Funktionen, die im Programmentwurf aufgezählt wurden. Dazu gehören:

- Erstellen eines neuen XSL- Stylesheets
- Löschen von Elementen, Text, Attributen
- Erstellen von Elementen, Text, Attributen
- Ändern von Elementen, Text, Attributen
- Erstellen von XSL- Anweisungen
- Erstellen von IIF- spezifischen Anweisungen
- Kopieren und Einfügen von Elementen, Text, Attributen
- Ändern der Reihenfolge von Elementen und Text
- Erstellen von XPath Ausdrücken

7.3.4.1 Allgemeiner Aufbau

Jede Methode zur Änderung am XML- Dokument ist ähnlich aufgebaut. Zunächst ruft die jeweilige Methode den zu editierenden Knoten aus dem vorher beschriebenen Zeiger ab.

Der nächste Schritt ist das Ändern des Knotens im Document Object Model. Dabei können dem Knoten entweder Kinderknoten hinzugefügt werden, der Knoten selber gelöscht werden

oder der Knoten selber geändert werden. Die Änderungen an den Knoten werden über die Methoden der Interfaces *Node* und *Element* vorgenommen.

Nach erfolgreicher Änderung im Document Object Model wird das *DefaultTreeModel* angepasst. Dazu wird der betreffende Knoten und dessen Kinderknoten im *DefaultTreeModel* gelöscht und mit Hilfe der Methode *recurseXml* aus dem Document Object Model neu eingelesen. So wird sichergestellt, dass beide Datenmodelle synchron bleiben, da das *DefaultTreeModel* nicht einzeln verändert wird sondern nur aus dem geänderten Document Object Model abgeleitet wird.

Wie in Abb. 7.4 zu sehen werden die Methoden aus der grafischen Nutzeroberfläche aufgerufen. Die GUI setzt dabei den Zeiger auf den zu verarbeitenden Knoten und übergibt weitere Daten, die eingefügt werden sollen. Anschließend ändern die Methoden die beiden Datenmodelle Document Object Model und *DefaultTreeModel*.

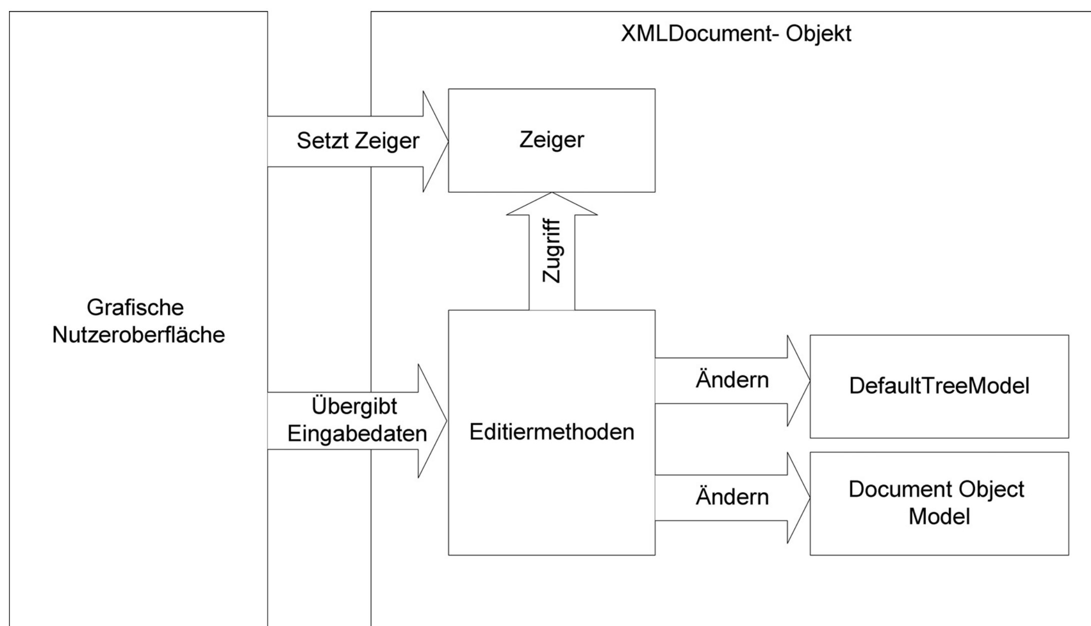

Abbildung 7.3: Aufruf von Editierfunktionen

Das nachfolgende Listing zeigt ein Beispiel für eine Editiermethode. Die Methode *addText* fügt

dem ausgewählten Knoten einen weiteren Textknoten hinzu. Zunächst wird mit Hilfe einiger Hilfsmethoden der vom Zeiger beschriebene Knoten sowie deren Elternknoten abgerufen. Anschließend wird mit Hilfe des DOM- Interface ein neuer Textknoten geschaffen und an den ausgewählten Knoten angehängt. Durch die Hilfsmethode *refreshTreeModel* wird u.a. über die Methode *recurseXML* das *DefaultTreeModel* aktualisiert.

Listing 7.2: Beispiel für eine Editierfunktion

```
public void addText(String textValue)
{
  //[...]
  Node selectedNode = getselectedNode();
  Node parentNode= getselectedParentNode();
        DefaultMutableTreeNode dmtParentNode = getselectedDmtParentNode();

  //Edit Dom
  Node TextNode = document.createTextNode(textValue);
  selectedNode.insertBefore(TextNode, selectedNode.getFirstChild());

  //Edit DefaultTreeModel
  refreshTreeModel(dmtParentNode,parentNode);
  //[...]
}
```

7.3.4.2 Liste der Methoden

Die folgende Liste beinhaltet die Methoden, die zum Ändern des Document Object Models und des *DefaultTreeModel* eingesetzt werden. Zudem werden die Besonderheiten der jeweiligen Methode im Hinblick auf Aufbau und Funktionalität näher erklärt, die über den vorher beschriebenden allgmeinen Aufbau hinausgehen.

- createNewStylesheet()

  Die Methode generiert eine neues Stylesheet. Darin enthalten ist der XSL- Wurzelknoten - *xsl:stylesheet*, Outputoptionen in Form der Anweisung *xsl:output* sowie eine *xsl:template*- Anweisung. Diese XSL- Anweisungen bilden das Grundgerüst für ein XSL-Stylesheet.

- *editAttributeNode(String attributeName, String attributeValue)*

  Die Methode wird eingesetzt, um die Attributknoten eines Elementknotens zu ändern. Die Attribute des Elementknotens werden zunächst in ein Stringarray ausgelesen, über das das betreffende Attribut verändert wird. Anschließend werden alle Attribute des Knotens gelöscht und die geänderten Attribute neu eingefügt. Dies ist dem Umstand geschuldet, dass Attribute im *Node*- Interface nicht direkt geändert werden können.

- *editElementNode(String elementName)*

  Die Methode legt eine Kopie des Elementknotens mit dem übergebenen String als Namen an. Dazu werden die Attribute sowie andere Kinderknoten kopiert. Der neue Elementknoten wird nun mit dem alten Elementknoten vertauscht. Auch hier ist es nicht möglich, den Namen des Elementknotens über das *Node*- bzw. *Element* Interface zu ändern.

- *editTextNode(String textValue)*

  Hier wird der Textwert des Textknotens geändert.

- *deleteNode()*

  Der ausgewählte Knoten wird gelöscht.

- *addElement(String elementName)*

  Die Methode fügt dem ausgewählten Elementknoten ein neues Element mit vorher definierten Namen hinzu. Das Element wird an erster Stelle vor evtl. bereits existierenden Elementen erstellt.

- *addAttribute(String attributeName, String attributeValue)*

  Die Methode fügt dem ausgewählten Elementknoten ein neues Attribut hinzu.

- *addText(String textValue)*

  Die Methode fügt dem ausgewählten Elementknoten einen neuen Textknoten hinzu. Der Textknoten wird an erster Stelle vor evtl. bereits existierenden Textknoten erstellt.

- *addXslNode(String elementName, String[] attributeName, String[] attributeValue*

Dem ausgewählten Elementknoten wird eine XSL- Anweisung angehängt. Dazu wird ein Element mit dem XSL- Namensraum geschaffen. Die Art der Anweisungen sowie deren Parameter werden von der grafischen Oberfläche übergeben.

- *addFeatureAndText(TreePath refTreePath)*

  Die Methode erstellt ein Element mit dem Namen *Feature* und dem Attribut *key* mit dem Namen des ausgewählten übergebenen Knotens Als Kindknoten dieses Elementes wird die XSL- Anweisung *Value-Of* angehängt, die den Textknoten des übergebenen Knotens ausliest. Das erstellte Element wird dann als Kindsknoten an den ausgewählten Elementknoten im Baum des Stylesheets angehängt.

- *addValueOf(TreePath refTreePath)*

  Diese Methode funktioniert ähnlich wie *addFeatureAndText()*. Hier wird allerdings nur die XSL- Anweisung *ValueOf* eingefügt.

- copyNode(TreePath treePath)

  Die Methode kopiert den über den mit *TreePath* beschriebenen Knoten in eine Zwischenablage.

- *pasteNode()*

  Die Methode fügt den Knoten in der Zwischenablage an ein ausgewählten Elementknoten an.

- switchNodes(DefaultMutableTreeNode sourceDmtNode, DefaultMutableTreeNode targetDmtNode)

  Hier werden zwei Knoten, die denselben Elternknoten haben in der Reihenfolge getauscht.

- *String getPathToNode(Node node)*

  Die Methode verändert das Document Object Model nicht. Aus dem übergebenen Knoten wird ein XPath- Ausdruck generiert und dieser als String zurückgegeben.

7.3.5 XSL-Prozessor

Der XSL- Prozessor hat die Aufgabe das zu bearbeitenden XSL- Stylesheet zu einem zu validieren und zum anderen eine Transformation in eine XSL- Datei durchzuführen. Dabei soll der Nutzer über die GUI Parameter eingeben können, die dem Prozessor übergeben werden. Die Funktionen setzen voraus, dass jeweils eine geladenes Stylesheet sowie eine Metadatendatei vorliegen. Für die Transformation wird der XSL- Prozessor *javax.xml.transform.TransformerFactory* verwendet.

- Setzen von Parametern

  Hierfür stehen innerhalb der Klasse *XMLDocument* die beiden Arrays *parameterName* und *parameterValues* zur Verfügung, die von der grafischen Nutzeroberfläche aus verändert werden können. Die beiden Arrays werden bei Verwendung des XSL- Prozessors ausgelesen und als Parameter verwendet.

- Validierung von XSL- Dokumenten

  Die Überprüfung des XSL- Stylesheet auf richtige Syntax erfolgt über die Methode *validateXSL*. Dabei wird die Syntax eines Stylesheets auf Richtigkeit getestet. Dies geschieht über eine *TransformerFactory* die mit einem XSL- Stylesheet ein Transformer- Objekt erstellt. Die dabei anfallenden Fehlermeldungen werden festgehalten und auf Anfrage von außen weitergegeben (siehe Fehlerbehandlung).

- Transformation in eine Datei

  Die Transformation in eine Datei wird ähnlich wie die Validierung über die Methode *transformToFile(Document sourceDocument, File file)* durchgeführt. Der Methode wird dabei ein Document Object Model übergeben, dass vom Prozessor als Quelldokument verwendet wird. Das Ergebnis der Transformation wird in der ebenfalls übergebenen Datei gespeichert.

- Transformation in ein *DefaultTreeModel* zur Visualisierung

Um die Ausgabe des Prozessors innerhalb der GUI darzustellen, wird die Transformation durch die Methode *DefaultTreeModel transformToTreeModel(Document sourceDocument)* durchgeführt. Hier wird allerdings ein *DefaultTreeModel* zurückgegeben, was die Visualisierung der Ergebnisse innerhalb der GUI ermöglicht.

7.3.6 Fehlerbehandlung

In allen beschriebenen Methoden werden evtl. auftretende Fehler beim Benutzen des DOM-Interfaces und und des XSL- Prozessors durch einen *try- catch*- Block abgefangen. Die Fehlermeldungen werden in das Stringarray *message* übertragen und können von dort aus über die Methode *getMessage()* außerhalb des Objektes abgerufen werden und über die Methode *resetMessage* wieder gelöscht werden.

7.4 Grafische Benutzeroberfläche

Die grafische Nutzeroberfläche wird mit Hilfe der Programmierschnittstelle *javax.swing* realisiert. Die GUI wird dabei in folgende Komponenten unterteilt:

- Daten und Funktionalität

  Die grafische Nutzeroberfläche instanziert bei Start zwei Objekte der Klasse *XMLDocument*. Eines für das XSL- Stylesheet und eines für die Produktmetadaten.

- Menüleiste

  Über die Menüleiste erhält der Nutzer Zugang zu den Funktionen Laden bzw. Speichern von XML- Dokumenten, der Bedienung des XSL- Prozessors und der Nutzerhilfe. Dabei werden die *Swing*- Klassen *JMenuBar*, *JMenuItem* sowie *JMenu* genutzt, um eine Menüleiste mit Menüpunkten- und Unterpunkte bereitzustellen.

- XML- Visualisierung

  Wie vorher beschrieben wird die Visualisierung über die Klasse *JTree* vorgenommen. Der Baum wird dabei zunächst in ein *JscrollPane* eingebettet, welches ein vertikales Scrollen des Baumes ermöglicht wenn dieser größer als der Programmframe ist. Da zwei Bäume nebeneinander dargestellt werden sollen, werden beide wiederum in einer *JSplitPane* hinzugefügt. So kann der Nutzer auch die relative Größe der Darstellung beider Bäume ändern.

  Der *JTree* wird mit dem Treemodel aus den *XMLDocument*- Objekten initialisiert.

- Popupmenüs

  Diese Menüs werden durch Mauseingabe im *Jtree* aufgerufen und beinhalten die Editierfunktionen über dem XSL- Stylesheet. Wie die Menüsleiste greifen sie auf die Klassen *JMenu* und *JMenuItem* zurück.

- Dialogframes

  Für die zusätzliche Nutzereingabe über die Tastatur werden weitere Frames eingerichtet, die zur Laufzeit erstellt werden wenn die entsprechende Funktion aufgerufen wurde. Diese Frames beinhalten ein oder mehrere Eingabefelder, die durch die Komponenten *JTextField* oder *JComboBox* realisiert sind. Zur Beschreibung der Funktion wird die Komponente *JLabel* verwendet. Zur Bestätigung oder Abbruch des Dialoges dient die Komponente *JButton*.

  Die Dialogframes rufen z.T. Daten aus den *XMLDocument* Objekten ab.

- Konsole

  Zur Darstellung von Fehlermeldungen, Status- oder Fortschrittsanzeigen wird ein *JTextField* genutzt, welches in ein *JScrollPane* eingebettet ist.

Das Menü, die Jtree- Bäume und die Konsole werden innerhalb des Hauptframes über den Layoutmanager *BorderLayout* angeordnet.

Die Nutzereingaben werden über Eventlistener abgefangen. Diese führen in Abhängigkeit der Eingabe die jeweiligen Methoden in den *XMLDocument*- Objekten aus oder instanzieren einen weiteren Dialogframe. Sie setzen außerdem vor Aufruf der Methoden und Dialoge den Zeiger des jeweiligen *XMLDocument*- Objektes. Dabei wird der ausgewählte Knoten im Jtree abgerufen und an das Objekt übergeben.

Es werden verschiedene Typen von Eventlistenern eingesetzt:

- *Actionlistener*

  Diese werden benutzt, Eingaben auf bestimmten Komponenten in der Menüleiste, den Popupmenüs oder anderen Schaltflächen abzufangen.

- *KeyListener*

  Es werden 2 verschiedene Keylistener auf die beiden *Jtree* angesetzt, so dass Nutzereingaben über Tastatur abgefangen werden können. Dies dient der Implementierung von Hotkeys für den schnelleren Zugriff auf Funktionen.

- *MouseListener*

  Analog zu den Keylistenern werden 2 Mouselistener auf die beiden *Jtree* angesetzt, um Mauseingaben abzufangen. Dies dient dem Aufruf des Popupmenüs zum Abruf von Funktionen über den visualisierten XML- Dokumenten.

- *ComponentListener*

  Der Componentlistener wird eingesetzt, um das Verschieben und das Anpassen der Größe des Hauptframes zu ermöglichen.

7.4.1 Visualisierung XML

Die Visualisierung der XML- Dokumente erfolgt über die Klasse *javax.swing.JTree*. Der *Jtree* wird mit dem Interface *DefaultTreeModel* instanziert, der das Datenmodell des Jtree darstellt.

Dieses Datenmodell wird aus einer Instanz der Klasse *XMLDocument* abgerufen.

Neben der textuellen Repräsentation ist es außerdem erforderlich, eine Möglichkeit zu schaffen grafische Eigenschaften der Knoten des *Jtree* in Abhängigkeit vom Inhalt zu verändern. Dazu wird nach Instanzierung des *Jtree* ein *DefaultTreeCellRenderer* gesetzt, bei dem die Methode *getTreeCellRendererComponent* überschrieben wird. Diese Methode gibt nun in Abhängigkeit vom referenzierten Objekt des Knotens einen veränderten Renderer zurück. Eigenschaften, die so bestimmt werden können, sind u.a. Hintergrundfarbe oder Formatierung der Schrift. Zusätzlich können weitere grafische Elemente wie Icons der grafischen Repräsentation hinzugefügt werden.

7.4.2 Verschieben von Knoten innerhalb des JTree

Das Erzeugen eines XSL- Stylesheets verlangt Gestaltungsmöglichkeiten in Hinblick auf die Reihenfolge von Kindsknoten. Dies wird in der GUI über Drag&Drop realisiert. Dazu wird eine *DropTargetListener* im *Jtree* eingefügt. Dieser prüft zunächst, ob der aufgenommene Knoten auf einen anderen Knoten verschoben werden kann. Dies ist nur möglich wenn beide Knoten denselben Elternknoten haben. Ist dies der Fall, wird der aufgenommene Knoten im Document Object Model gelöscht und vor dem anderen Knoten wieder eingefügt. Dies geschieht über die Methode *switchNodes(DefaultMutableTreeNode sourceDmtNode, DefaultMutableTreeNode targetDmtNode)* in der Klasse *XMLDocument*.

7.4.3 Nutzerhilfe

Die Nutzerhilfe wird über eine Art einfachen Browser innerhalb der grafischen Nutzeroberfläche dargestellt. Dabei wird auf HMTL- Dokumente zurückgegriffen, in denen ein einfaches Navigieren zwischen verschiedenen Themen möglich ist. Zur Darstellung von HTML- Dateien wird die Klasse *JEditorPane* verwendet der eine URL übergeben werden kann.

8 Validierung des Funktionsumfanges

Um den Funktionsumfang des Programmes zu testen wird hier die praktische Arbeit mit der grafischen Nutzeroberfläche anhand von Beispielen beschrieben.

8.1 Abrufen und Visualisieren von XML- Dokumenten

Um die Funktionen der Visualisierung sowie das Laden und Speichern von XML- Dokumenten im Dateissystem zu beschreiben wird ein Beispieldokument mit Hilfe eines externen Editors erstellt:

```
<?xml version="1.0" encoding="UTF-8"?>
<rootElement>
   <Element AttributName="AttributWert"/>
   <Element>
      <Element>Text</Element>
   </Element>
</rootElement>
```

Das Dokument ist ein wohlgeformtes XML- Dokument mit den wesentlichen XML- Komponenten - Elemente, Attribute und Text. Das Dokument wird über die Menüleiste mit dem Menüpunkt - *File - Open Xsl File...* geöffnet und automatisch visualisiert.

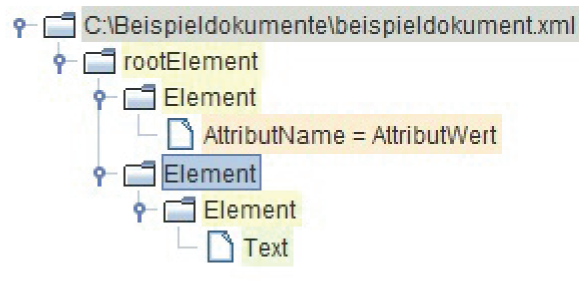

Abbildung 8.1: Visualsierung des XML- Dokumentes

Abb. 8.1 zeigt die Umsetzung der im Entwurf skizzierten Darstellung von XML- Dokumenten. Das Dokument wird durch eine Baumstruktur repräsentiert so dass Hierarchien klar erkennbar sind. Zusätzlich unterstützt die Farbgebung das Erkennen unterschiedlicher Komponenten. Die Struktur des Dokumentes ist somit leicht überschaubar. Abb. 8.2 zeigt ein Xsl- Stylesheet des DLR sowie Produktmetadaten, die über den Menüpunkt *File - Open Product Metadata ...* geladen wurden.

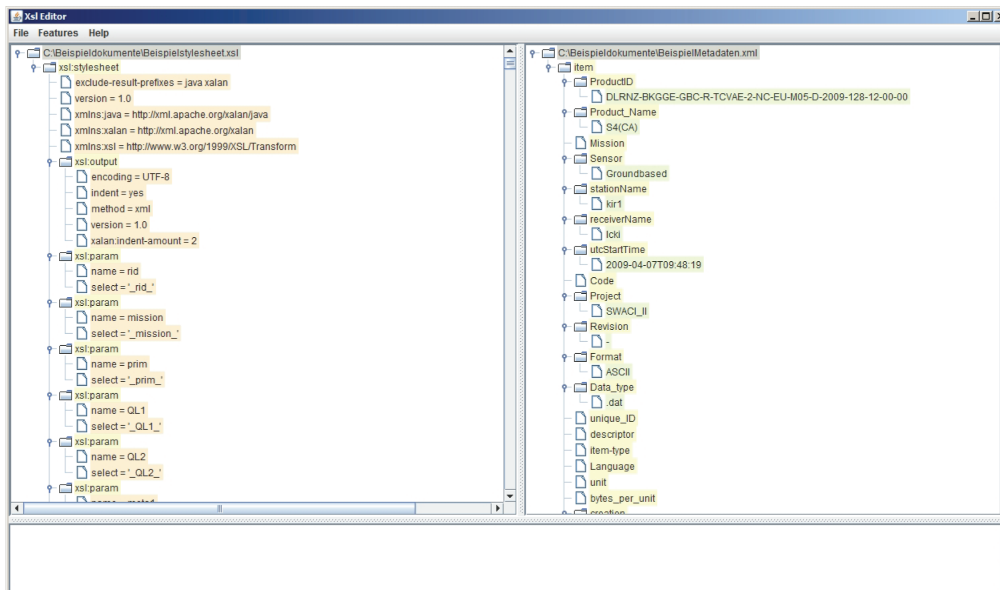

Abbildung 8.2: Entwicklungsumgebung

Das zu bearbeitende Stylesheet ist in der linken Hälfte des Fensters verfügbar, die Metadaten können zur Hilfe in die rechte Hälfte des Fensters eingeblendet werden. Im unteren Teil des Fensters befindet sich zu dem das Textfeld zum Ausgeben von Fehler- oder Statusmeldungen.

8.2 Editierfunktionen und XSL- Prozessor

Um die Editierfunktionen sowie die Verarbeitung von XSL- Stylesheets durch einen XSL-Prozessor zu testen wird über die grafische Nutzeroberfläche eine neues Stylesheet erstellt. In diesem werden die verfügbaren XSL- Anweisungen eingebettet. Als Quelldokument soll das Beispieldokument aus dem letzten Abschnitt dienen.

Zunächst wird über *File - New Xsl- Stylesheet* ein neue Stylesheet erstellt. Dieses beinhaltet das Rootelement *xsl:stylesheet* sowie die Anweisungen *xsl:output* und *xsl:template*. Somit besteht ein Grundgerüst eines XSL- Stylesheets, welches nun erweitert werden kann.

Im nächsten Schritt werden die verfügbaren XSL- Anweisungen exemplarisch eingefügt (siehe Abb. 8.3). Dazu gehören:

- xsl:if
- xsl:variable
- xsl:parameter
- xsl:value-of

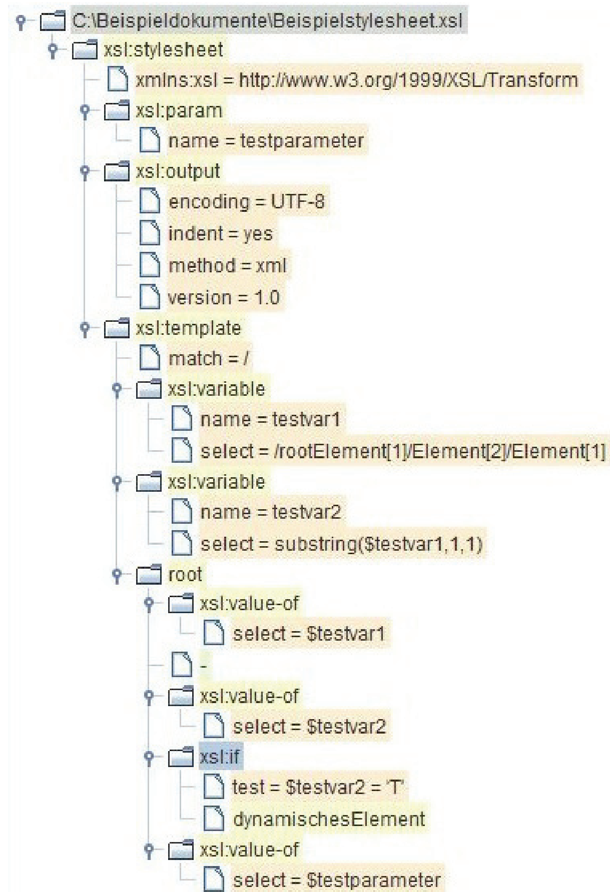

Abbildung 8.3: Beispiel für ein generiertes XSL- Stylesheet

Mit Hilfe der Variablen *testvar1* wird bei der Transformation ein Textinhalt aus dem Quelldokument ausgelesen. Die zweite Variable *testvar2* nutzt die XSL- Funktion *substring*, um den Textinhalt zu manipulieren. Beide Variablen werden dann unter dem *root*- als Text ausgegeben.

Danach folgt die XSL- Anweisung *xsl:if*, die eine Variable ausliest und mit einem Text vergleicht. In Abhängigkeit von diesem Test wird ein weiteres Element eingefügt.

Letztendlich soll auch der über die grafische Nutzeroberfläche eingegebene Parameter ausgegeben werden. Das Ergebnis der Transformation findet sich in Abb. 8.4:

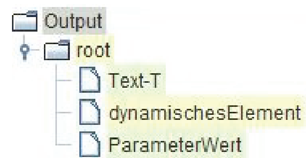

Abbildung 8.4: Ergebnis der Transformation

Das über die grafische Nutzeroberfläche abgespeicherte Ergebnis sieht in einem Texteditor so aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>Text-T<dynamischesElement/>ParameterWert</root>
```

9 Zusammenfassung

Das Ziel dieser Arbeit war der Entwurf und die Umsetzung einer grafischen Nutzeroberfläche und deren Funktionalität, die es den DLR- Mitarbeiter erleichtern soll, Transformationsprozesse zwischen verschiedenen Metadatenspezifikation zu generieren und zu testen.

Die erarbeitete Lösung basiert auf der Transformationssprache XSL-T. Der Funktionsumfang der Spezifikation garantiert eine einfache Erweiterbarkeit der Applikation und deckt alle bereits vorhandenen Anforderungen an einen Transformationsprozess ab. Dies bezieht sich nicht nur auf den Funktionsumfang der Sprache an sich sondern auch auf das Vorhandensein von einfach zu implementierbaren XSL- Prozessoren zum Validieren und Ausführen von Transformationsprozessen.

Im Mittelpunkt der grafischen Nutzeroberfläche steht die Visualisierung der XML- und somit auch XSL- Dokumenten. Hier wurde ein Ansatz gewählt, der auf die Darstellung des XML-Markup sowie bestimmter XML- Strukturen verzichtet. Dazu wurden die Elemente eines XML-Dokumentes in eine Baumstruktur übertragen, so dass die Hierarchie auch ohne Markup erkennbar ist. Desweiteren wurden die Elemente farblich aufbereitet, so dass sie unterscheidbar wurden. Somit wurde eine Visualisierung geschaffen, die für den Nutzer einfacher zu überschauen ist als die Darstellung von XML- Dokumenten in Textform.

Auf Grundlage dieser Visualisierung wurden Schnittstellen geschaffen über die der Nutzer auf die Visualisierung zugreifen kann um Funktionen abzurufen. Ein Beispiel dafür ist das Popupmenü über einem einzelnen selektierten Baumknoten, das über eine Mauseingabe oder Tasta-

tureingabe aufgerufen wird. Aber auch andere Schnittstellen stehen zur Verfügung wie z.B. die Menüleiste. Die Verwendung der Schnittstellen wird zudem in einer Nutzerhilfe erklärt.

Die über die Schnittstellen aufgerufene Funktionalität lässt sich in mehrere Punkte zerlegen. Die Verarbeitung im Dateisystem mittels der Speichern- und Laden- Funktionen garantiert die Wiederverwendbarkeit von XSL- Dokumenten.

Mittels der Editierfunktionen kann ein XSL- Dokument manipuliert werden. Hier ist es möglich, auf wesentliche XML- Elemente zuzugreifen oder in der Analyse benannte XSL- Anweisungen einzufügen. Zudem wurden Funktionen eingeführt, die bestimmte Arbeitsschritte von DLR-Mitarbeitern beim Erstellen eines XSL- Dokumentes beschleunigen.

Zusätzlich kann der Nutzer auf einen XSL- Prozessor zugreifen und diesen nutzen, um XSL-Transformationen zu validieren oder das Ergebnis des Prozesses in einer Datei zu speichern oder anzeigen zu lassen.

Die programmierte Applikation erfüllt somit die vorgegebenen Anforderungen des DLR, da die wesentlichen Arbeitsschritte zum Erstellen eines Transformationsprozesses in einer grafischen Entwicklungsumgebung zusammengefasst sind. Die Umgebung lässt sich zudem auf Basis der verwendeten Technologien (XSL, DOM) einfach und sicher erweitern und kann an zusätzliche Bedürfnisse und zukünftige Anforderungen angepasst werden.

9.1 Probleme

Ein Problem bei der Implementierung ist die Ausgabe von Fehlermeldungen an den Nutzer. Die Fehlermeldungen basieren auf den Schnittstellen des Document Objects Models und des XSL- Prozessors. Diese geben aber keine Informationen über die Position eines möglichen Fehlers im Dokument zurück, sondern nur die Art des Fehlers und die betroffenen Elemente. Ein Workaround gestaltet sich hier schwierig, da in Zukunft evtl. andere Prozessoren benutzt

werden. Für den Nutzer hat dies den Nachteil, dass bei der Validierung von XSL- Dokumenten nicht die Position des Fehlers angezeigt wird und er selber danach suchen muss.

Problematisch ist außerdem das Fehlen einer ausführlichen Dokumentation für den Nutzer. Es wurde nur eine Möglichkeit geschaffen Hilfsdokumente in HTML- Form in der grafischen Nutzeroberfläche aufzurufen. Hier konnte der Entwurf nicht vollständig umgesetzt werden.

9.2 Ausblick

In Hinblick auf die vorher genannten Probleme sollte auch die Fehlerbehandlung sowie -ausgabe überdacht werden. Diese sollte unabhängig von den verwendeten Programmierschnittstellen sein und Fehler genauer lokalisieren können. Zudem sollte eine Dokumentation in Form einer Hilfe geschaffen werden die den Nutzer über die Funktionen des Programmes sowie über die XSL- Spezifikation informiert.

Um die Nutzerfreundlichkeit zu steigern, bedarf es einer Reihe weiterer Funktionen um die die Applikation in der Zukunft erweitert werden könnte. So können von anderen Editoren gewohnte Funktionen wie Suchen und Ersetzen oder das Rückgängigmachen der letzten Nutzeraktion bzw. eine Undelete- Funktion implementiert werden. Zudem können je nach Bedarf auch weitere XSL- Anweisungen verfügbar gemacht werden.

Eine weitere Möglichkeit der Verbesserung ist das Speichern und anschließende Abrufen von Nutzereingaben in den jeweiligen Nutzerdialogen. Das würde die Wiederverwendbarkeit von eingegebenen XSL- Code vergrößern. Der Nutzer könnt z.B. durch eine Autovervollständigungsfunktion bei der Eingabe unterstützt werden.

Eine wichtige Verbesserung wäre außerdem das direkte Editieren auf der Visualisierung und somit das Umgehen der Nutzerdialoge in zusätzlichen Frames. D.h., dass der Nutzer Elemente des Baumes selektiert und anschließend sofort in der Lage ist, den Inhalt des Elementes zu

ändern ohne auf eine weitere Eingabemaske zurückzugreifen zu müssen.

Abbildungsverzeichnis

2.1 Datenarchivierung im Data Information and Managment System 9
2.2 Automatisierte Prozessreihe im DIMS 11
2.3 XSL- Transformation 15

6.1 XML- Visualsierung über Baumstruktur 31
6.2 Entwurf der Benutzeroberfläche 32
6.3 Zugriff auf die Baumknoten 33

7.1 Zugriffe im Gesamtsystem 42
7.2 Visualsierung des Document Object Model 46
7.3 Aufruf von Editierfunktionen 48

8.1 Visualsierung des XML- Dokumentes 58
8.2 Entwicklungsumgebung 58
8.3 Beispiel für ein generiertes XSL- Stylesheet 60
8.4 Ergebnis der Transformation 61

Literaturverzeichnis

[1] SIMON ST. LAURENT,MICHAEL FITZGERALD: *XML kurz & gut.* - O'Reilly 2006

[2] BRETT D. MCLAUGHLIN,JUSTIN EDELSON: *Java & XML* - O'Reilly 2006

[3] EVAN LENZ: *XSL kurz & gut.* - O'Reilly 2006

[4] FRANZ-JOSEF HERPERS, THOMAS J. SEBESTYEN : *XSL- Das Einsteigerseminar* - vmi Buch 2002

[5] S. KIEMLE, A.-K. SCHROEDER LANZ, B. BUCKL: Product Library - User Manual - DLR 2008

[6] HTTP://WWW.COMELIO-MEDIEN.COM/COMELIO-BLOG/XSLT/ALTERNATIVEN_ZU_XSLT - Alternativen zu XSLT

[7] HTTP://WWW.W3SCHOOLS.COM - XML DOM- Turorial, XSLT- Tutorial

[8] HTTP://WWW.DLR.DE - Informationen zur Arbeit und Struktur des DLR

[9] HTTP://WWW.DEVELOPER.COM/JAVA/OTHER/ARTICLE.PHP/10936_3731356_2/DISPLAYING-XML-IN-A-SWING-JTREE.HTM - Darstellung von XML in Javas Jtree

[10] HTTP://WWW.W3.ORG/TR/XSLT - w3c Empfehlung zu XSLT

[11] HTTP://WWW.W3.ORG/TR/REC-XML/ - w3c Empfehlung zu XML

[12] HTTP://DE.WIKIPEDIA.ORG/WIKI/WIMP_(BENUTZERSCHNITTSTELLE) - Konzept für eine Benutzerschnittstelle

Anhang

Als Anhang für diese Arbeit ist eine Daten- CD erstellt worden, auf der die Ergebnisse gespeichert sind. Die Verzeichnisse enthalten folgende Inhalte:

- *src* - Quellcode

  Hier befinden sich alle Klassen der Implementierung. Die Hauptklasse ist die Datei *gui.java*

- *pdf* - Schriftlicher Teil der Diplomarbeit im PDF- Format

- *javadoc* - HTML Dokumentation des Quellcode

  Die Dokumentation kann über die Datei *index.htm* abgerufen werden.

- *Beispieldokumente* - bei der Validierung verwendete XML- und XSL- Dokumente