Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Eine domänenspezifische Sprache zur Transformation von Anwendungszuständen in Webapplikationen

Bachelorarbeit

im Studiengang Informatik

von

Timo Hüther

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr. Rainer Parchmann
Betreuer: M. Sc. Tristan Wehrmaker

Hannover, 18. August 2010

Zusammenfassung

Die modellgetriebene Softwareentwicklung ist eine Technik, mit der es möglich ist, automatisiert aus formalen Modellen lauffähige Software zu erzeugen. Dabei finden domänenspezifische Sprachen Verwendung, die mit entsprechenden Codegeneratoren eingesetzt werden.

Die meisten Anwendungen im World Wide Web haben eine Reihe von Zuständen, die durch Hyperlinks miteinander verbunden sind. In dieser Arbeit wird eine domänenspezifische Sprache für die Domäne Webanwendung definiert und angewendet. Die Sprache ist so gestaltet, dass sie durch eine ebenfalls zu entwickelnde grafische Notation gezeichnet werden kann. Schließlich wird anhand einer Transformation gezeigt, wie die Codegenerierung aussieht.

Abstract

Model Driven Software Development is a technique making it possible to automatically create executable software from formal models. Domain Specific Languages with adequate code generators are used for this.

Most applications in the world wide web have a number of states, which are connected among each other by hyperlinks. In this thesis a Domain-Specific Language for the domain of webapplications will be defined and applied. The language is configured in such a way that it can be drawn by a graphical notation, which wich is developed as well. Finally it will be shown by a transformation how the code generator will appear.

Danksagung

Ich danke allen, die mich beim Erstellen dieser Arbeit unterstützt haben.
Insbesondere danke ich Herrn Tristan Wehrmaker, für die herrvoragende Betreuung.

Akronymverzeichnis

CSS	Cascading Style Sheets
DSL	Domain-Specific Language, Domänenspezifische Sprache
DTD	Document Type Definition
EMF	Eclipse Modeling Framework
FTP	File Transfer Protocol
GEF	Graphical Editing Framework
HATEOAS	Hypermedia As The Engine Of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
MDSD	Model Driven Software Development, Modellgetriebene Softwareentwicklung
MVC	Model View Controller
REST	Representational State Transfer
RSS	Really Simple Syndication
UML	Unified Modeling Language
W3C	World Wide Web Consortium
WebML	Web Modeling Language
WSL	Web State Language
WWW	World Wide Web
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
XSD	XML Schema Definition
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformation

Inhaltsverzeichnis

1. Einleitung **8**
1.1 Motivation 8
1.2 Aufgabe 8
1.3 Struktur dieser Arbeit 9

2. Grundlagen **10**
2.1 Das World Wide Web 10
2.1.1 Representational State Transfer (REST) 10
2.2 Extensible Markup Language (XML) 11
2.2.1 Aufbau von XML-Dokumenten 11
2.2.2 XML als Baumstruktur 12
2.2.3 Wohlgeformtheit und Gültigkeit 12
2.3 XML Schema 13
2.3.1 Aufbau einer XML Schema Definition 13
2.3.2 Einfache Typen 14
2.3.3 Komplexe Typen, Elemente und Attribute 15
2.4 Extensible Stylesheet Language Transformation (XSLT) 15
2.4.1 Funktionsweise eines XSLT Prozessors 15
2.4.2 Struktur einer XSL-Datei 16
2.4.3 XML Path Language (XPath) 17
2.5 Modellgetriebene Softwareentwicklung 18
2.5.1 Ziele und Grundlagen 18
2.5.2 Integration 19
2.5.3 Pretty Printing 20

3. Domänenspezifische Sprache für Webapplikationen **21**
3.1 Anforderungen und Entwurf 21
3.1.1 Modellklassen 22
3.1.2 Zustände 23
3.1.3 Transitionen 23
3.2 WSL Definition als XML Schema 26

4. Entwicklung einer grafischen Notation **28**
4.1 Grafische Darstellung der WSL 29
4.1.1 Grafische Darstellung der Modellklassen 29
4.1.2 Grafische Darstellung der Zustände 30
4.1.3 Grafische Darstellung der Transitionen 31
4.2 ProFLOW 33
4.2.1 Programmierung einer neuen Notation 34
4.2.2 Export als WSL 35

5. Transformation der WSL in eine Webapplikation **36**
5.1 Einführung in das Play Framework 36
5.1.1 Das Play MVC Konzept 36
5.2 Definition einer Transformationsvorschrift 37
5.2.1 Generierung von Modellklassen 38
5.2.2 Generierung von Controllerklassen 39
5.2.3 Generierung von Views 39
5.3 Programmierung eines XSLT Prozessors in Java 40
5.3.1 Validierung der WSL mit einem XML Schema 40
5.3.2 Transformationsdurchführung 40

6. Beispiel **42**

7. Zusammenfassung und Ausblick **45**
7.1 Zusammenfassung 45
7.1.1 Fazit 46
7.2 Ausblick 46

Literaturverzeichnis **48**

Abbildungsverzeichnis **50**

Tabellenverzeichnis **50**

Anhang **51**
A. Inhalt der CD 51
B. Web State Language 51

1. Einleitung

1.1 Motivation

Das World Wide Web hat sich seit der Einführung im August 1991 stark gewandelt. Bestand es zu Anfang größtenteils aus einfachen statischen Webseiten zum Beispiel zur Produktinformation, finden sich heute immer mehr Webanwendungen wie Onlineshops und Verwaltungsanwendungen. Viele klassische Desktopanwendungen wie Tabellenkalkulationen oder Terminplaner werden ins Web verlagert, um eine einfachere Teamarbeit zu ermöglichen und von überall abrufbereit zu sein.

Ziele bei der Entwicklung von (Web-) Anwendungen sind unter anderem die Verbesserung von Qualität, Verkürzung der Entwicklungsdauer, sowie das Wiederverwenden von bereits existierenden Programmteilen. Speziell Webanwendungen die RESTful (siehe Kapitel 2.1) sind und die Bedingungen von Hypermedia As The Engine Of Application State (HATEOAS) erfüllen, haben meinst eine ähnliche Struktur. Dabei gibt es Zustände und Zustandsübergänge. Diese Eigenschaften kann man sich zu Nutze machen, um einen Großteil des zu entwickelnden Programms automatisch zu generieren.

1.2 Aufgabe

Im Rahmen dieser Arbeit soll eine auf XML basierende domänenspezifische Sprache (DSL) entwickelt werden, die in eine Webapplikation transformiert werden kann. Die DSL soll als Zustandsdiagramm grafisch dargestellt und mit einer für diese Aufgabe entworfenen grafischen Notation gezeichnet werden können. Abbildung 1 zeigt die grobe Aufgabenstellung.

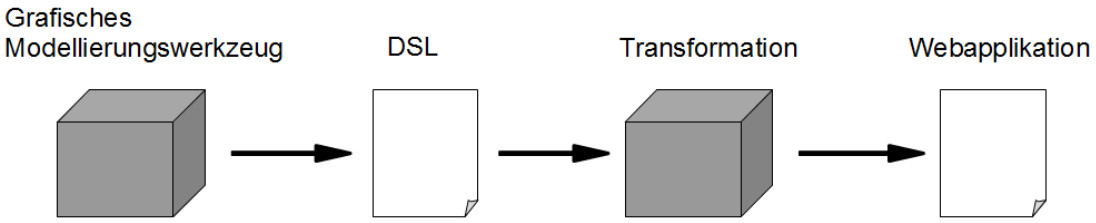


Abbildung 1: Visualisierung der Aufgabenstellung

Die Transformation in die Webapplikation soll nach den Prinzipien der modellgetriebenen Softwareentwicklung (siehe Kapitel 2.5) realisiert werden. Als grafisches Modellierungswerkzeug soll das am Fachgebiet Software Engineering entwickelte Programm ProFLOW[1] verwendet werden. Die Transformation in eine Webapplikation soll mittels XSLT (siehe Kapitel 2.4) geschehen und am Beispiel des Play-Frameworks demonstriert werden. Abbildung 2 ist eine Weiterführung von Abbildung 1 und verfeinert die Aufgabenstellung.

1 http://www.se.uni-hannover.de/forschung/flow/proflow/

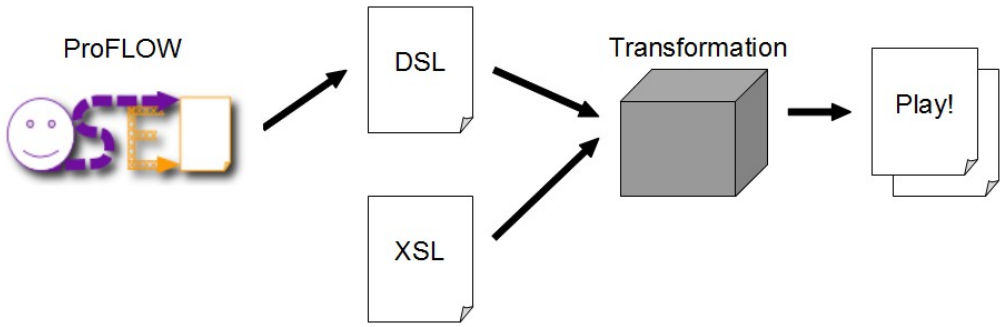


Abbildung 2: Visualisierte Darstellung der verfeinerten Aufgabenstellung

Der Kern der Aufgabe ist also eine DSL zu erarbeiten, die sowohl grafisch dargestellt als auch in eine Webanwendung transformiert werden kann.

1.3 Struktur dieser Arbeit

Diese Arbeit besteht aus insgesamt sieben Kapiteln. Nachdem Kapitel 1 eine Einleitung in die Thematik geboten hat, werden in Kapitel 2 die Grundlagen vorgestellt und erläutert.

In Kapitel 3 geht es dann um den eigentlichen Kern der Arbeit: Die Entwicklung einer domänenspezifischen Sprache und deren Grammatik.

Kapitel 4 und 5 behandeln den praktischen Teil der Aufgabe. Es wird eine grafische Notation entworfen, mit der die DSL gezeichnet werden kann. Außerdem wird ProFLOW so erweitert, dass es in der Lage ist, die DSL zu zeichnen und zu exportieren. Dieser Export wird in Kapitel 5 nachfolgend verwendet, um daraus eine Webanwendung zu erzeugen.

Bei Kapitel 6 handelt es sich um ein Beispiel-Kapitel, in dem gezeigt wird, wie das zuvor Vorgestellte in der Praxis ablaufen kann.

Am Ende der Bachelorarbeit steht mit Kapitel 7 eine Zusammenfassung mit Fazit und ein Ausblick.

2. Grundlagen

Nachdem die Aufgabe und die Beweggründe, diese umzusetzen, erläutert wurden, wird im folgenden Kapitel auf die zu Grunde liegenden Techniken und Standards eingegangen. Zuerst wird gezeigt, wie man sich die Domäne „Webanwendung“ zu Nutze machen kann. Als nächstes werden XML, XML Schema und die XSL Transformation erklärt. Abschließend geht es um die modellgetriebene Softwareentwicklung.

2.1 Das World Wide Web

Das World Wide Web (WWW) ist ein über das Internet abrufbares Hypertext-System. Es besteht aus vielen verschiedenen Webseiten und -anwendungen, die dezentral von Servern zur Verfügung gestellt werden. Um die Inhalte anzeigen zu können, ist ein Client erforderlich (z.B. ein Webbrowser), der die benötigten Daten vom Server abruft und auf dem Bildschirm darstellt. Da das Internet aus vielen verschiedenen Diensten besteht, zum Beispiel Email und das File Transfer Protocol (FTP), ist eine klare sprachliche Trennung sinnvoll. Wenn in dieser Arbeit von Webanwendung oder Webapplikation die Rede ist, ist damit immer eine Anwendung gemeint, die als Schnittstelle einen Browser voraussetzt.

Das WWW basiert auf drei Kernstandards [RR07]:

- Dem Hypertext Transfer Protocol (HTTP) [IETF99],
- der Hypertext Markup Language (HTML) [W3C99] und
- dem Uniform Resource Identifier (URI) [IETF05].

Mittels HTTP kann ein Browser Informationen von einem Webserver anfordern. HTML ist eine Dokumentbeschreibungssprache, die die bereitgestellten Informationen gliedert und über Hyperlinks verknüpft. HTML wird vom World Wide Web Consortium (W3C), dem Gremium zur Standardisierung von Webtechnologien, spezifiziert und weiterentwickelt. Jede Ressource (z.B. Webseite) hat einen eindeutigen URI, sodass sie über Hyperlinks angesteuert werden kann.

Es gibt noch eine Reihe weiterer Standards und Technologien für das Web, wie Cascading Style Sheets (CSS) [W3C98] und JavaScript [ECMA09], die aber für diese Bachelorarbeit nicht von Bedeutung sind.

2.1.1 Representational State Transfer (REST)

Representational State Transfer ist kein Standard oder eine Technologie sondern ein Architekturstil für verteilte Hypermedia-Informationssysteme wie das World Wide Web. Er wurde das erste Mal im Jahr 2000 in der Dissertation von Roy Fielding vorgestellt. Das Web funktioniert nach den REST-Prinzipien, die unter anderem besagen, dass alle Ressourcen eindeutig identifizierbar sind und mehrere Repräsentationen haben können.

Die Grundprinzipien im Einzelnen sind [RR07]:

- Ressourcen mit eindeutiger Identifikation (eindeutig adressierbar),
- Verknüpfungen (Hyperlinks),
- unterschiedliche Repräsentationen und
- zustandslose Kommunikation.

Ressourcen sind dabei alles, was von Interesse ist. Zum Beispiel Personen, Gegenstände oder Orte. Jede Ressource soll eine eindeutige Identifikation haben und somit global über ein URI adressierbar sein. Dies ist auch eine Grundvoraussetzung für Verknüpfungen, die sonst nicht funktionieren würden.

Wenn als Teil der Repräsentation auf eine andere Ressource verwiesen wird, so kann der URI dieser Ressource als Identifikator angegeben werden. Ein Client kann diesem folgen, um zu der verlinkten Ressource zu gelangen. Das Prinzip der Verlinkung wird von Fielding als Axiom für REST unter dem Namen Hypermedia as the Engine of Application State (HATEOAS) genannt [RR07]. Es fordert, dass sich der Client durch Folgen der Hyperlinks durch die Anwendung bewegen können muss. Diese Voraussetzung ist ein Kernstück dieser Arbeit.

Mit unterschiedlicher Repräsentation ist gemeint, dass eine Ressource, zum Beispiel ein Auto, auf verschiedene Weisen repräsentiert werden kann. So ist es denkbar, dass dazu ein Bild und auch ein Datenblatt im HTML Format mit Fahrzeuginformationen existieren.

Zustandslose Kommunikation heißt, dass jede einzelne Anfrage nach einer Repräsentation unabhängig von vorhergehenden Anfragen ist. Eine Anfrage, die an den Server gesendet wird, muss alle Informationen beinhalten, die der Server zum Abarbeiten der Anfrage benötigt. Im konkreten Fall des Webs heißt das, dass der Server keine Informationen über den Anwendungszustand des Clients speichern darf.

2.2 Extensible Markup Language (XML)

XML ist der erste Post-Internet-Standard zur Beschreibung von Datenformaten [Mer03]. Dieser wird ebenfalls wie HTML vom W3C entwickelt [W3C08]. XML wird zunehmend als Schnittstellen-Standard auf allen technischen IT-Ebenen vom Frontend über Anwendungen bis zu Datenbanken eingesetzt [Mer03]. Außerdem ist XML auf keinen Anwendungsbereich begrenzt sondern vielseitig einsetzbar. Bekannte Einsatzgebiete sind Really Simple Syndication (RSS) [RAB09], ein Format zur strukturierten Veröffentlichung von Änderungen auf Webseiten und XHTML [W3C02], eine Neuformulierung von HTML 4.01 [W3C99] in XML. Weitere Einsatzgebiete von XML sind XML Schema (siehe Kapitel 2.3) und XSLT (siehe Kapitel 2.4).

2.2.1 Aufbau von XML-Dokumenten

Ein XML-Dokument besteht aus verschiedenen Elementen. Diese Elemente können Kindelemente oder Geschwisterelemente haben. Folglich kann ein Element auch ein Elternelement haben. Elemente können Attribute haben, können aber auch leer sein [W3C08].

Alle Elemente werden durch Tags dargestellt. Tags sind das Hauptkonstrukt einer Auszeichnungssprache [Ger04]. Attribute werden direkt in die zugehörigen Elementtags geschrieben. Ein XML-Dokument könnte zum Beispiel wie folgt aussehen:

```
<?xml version="1.0"?>
<Literaturverzeichnis>
    <Buch Titel="Modellgetriebene Softwareentwicklung">
          <Autor>Thomas Stahl</Autor>
          <Autor>Markus Völter</Autor>
    </Buch>
    <Buch Titel="XML-Komponenten in der Praxis">
          <Autor>Peter Mertens</Autor>
    </Buch>
</Literaturverzeichnis>
```

Beispiel 1: Ein XML-Dokument

In diesem Beispiel ist `Literaturverzeichnis` das Wurzelelement. Es folgen zwei Elemente `Buch`. Diese haben als Attribut einen `Titel` und als Kindelement `Autor`, welches ein- oder mehrmals vorkommen kann.

2.2.2 XML als Baumstruktur

Neben der einfachen Darstellungen in Code-Form gibt es noch eine andere Darstellungsweise. Der logische Aufbau eines XML-Dokuments entspricht einer Baumstruktur [Kay04b]. Daher wird der oberste Knoten auch als Wurzel bezeichnet. Für Beispiel 1 könnte der dazugehörige Baum wie folgt aussehen:

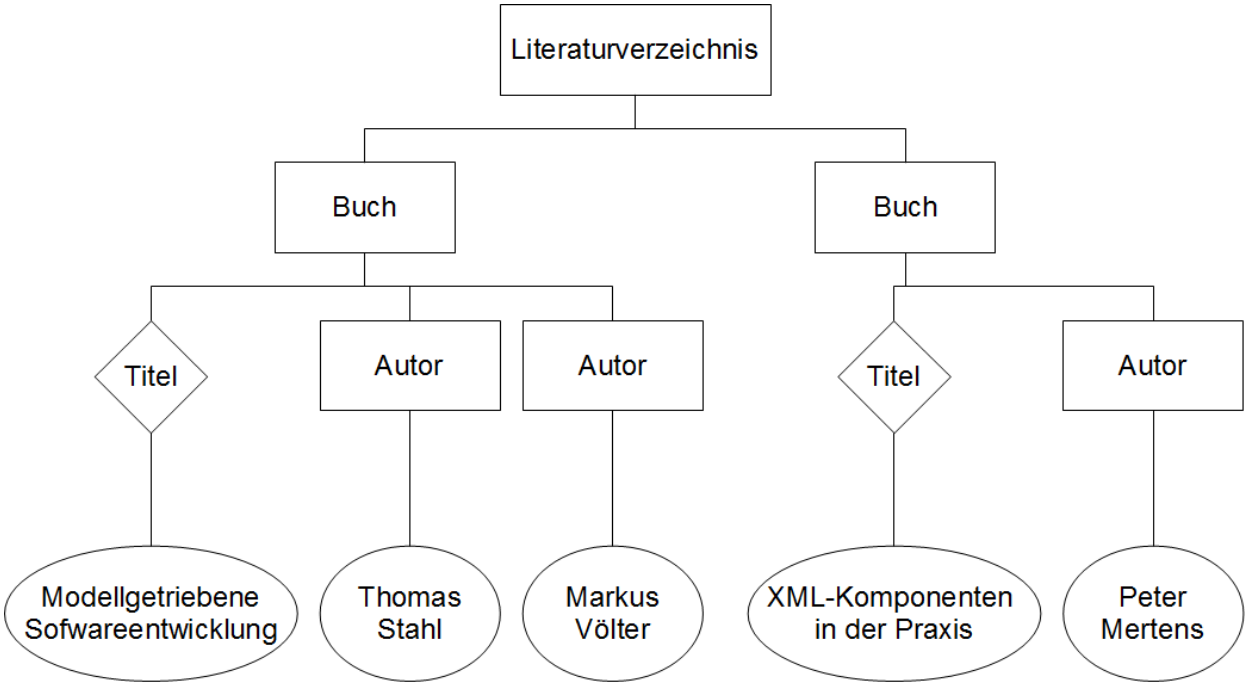


Abbildung 3: XML dargestellt in Baumstruktur

Die Rechtecke in Abbildung 3 entsprechen Elementen, Rauten sind Attribute und Ellipsen stellen Inhalte dar. „Literaturverzeichnis“ ist die Wurzel. Diese Art der Illustration macht deutlich, dass jeder Teil des XML Dokuments, also jedes Element und jeder Inhalt durch einen eindeutigen Pfad erreicht werden kann. Diese Eigenschaft wird in Kapitel 2.4 zur Navigation durch die XML benötigt und dort noch näher erläutert.

2.2.3 Wohlgeformtheit und Gültigkeit

Jedes XML-Dokument sollte die zwei grundlegenden Eigenschaften *Wohlgeformtheit* und *Gültigkeit* mitbringen. Ein Dokument heißt *wohlgeformt* genau dann, wenn [W3C08]:

1. es genau ein Wurzelelement hat,
2. alle Tags, die nicht leer sind, ein schließendes Tag haben,
3. alle Elemente geschlossen werden, bevor das Endtag des Elternelements oder ein Beginntag eines Geschwisterelements erscheint und
4. jedes Element nur Attribute mit unterschiedlichen Namen besitzt.

Wenn ein XML wohlgeformt ist, dann ist es syntaktisch korrekt.

Es ist sinnvoll für XML eine Grammatik mit XML Schema (siehe Kapitel 2.3) festzulegen. Damit kann die Semantik der XML automatisch auf *Gültigkeit* überprüft werden. Ein Dokument ist *gültig*, genau dann, wenn

- es *wohlgeformt* ist,
- den Verweis auf die zu Grunde liegende Grammatik enthält und
- die von der Grammatik beschriebene Semantik einhält.

Das Beispiel 1 (siehe Kapitel 2.2.1) ist *wohlgeformt* aber nicht *gültig*, da es keine zu Grunde liegende Grammatik hat.

2.3 XML Schema

Damit ein XML-Dokument gültig sein kann, ist eine Grammatik erforderlich, die Bedingungen an die XML stellt. Nur wenn all diese Bedingungen für das gesamte Dokument eingehalten werden, heißt es gültig. Hauptsächlich geht es darum festzulegen, welche Elemente in einem Dokument vorkommen dürfen und welche Attribute und Kindelemente diese haben. Mittels XML Schema (XML Schema Definition, XSD) lassen sich auch noch eine Reihe von weiteren Verfeinerungen festlegen. Zum Beispiel lässt sich ein Zahlenbereich definieren, in dem ein Wert liegen muss, oder eine Formatierungsvorschrift legt fest, wie eine Zeichenkette auszusehen hat [W3C04c].

Dieser Abschnitt hat nicht den Anspruch eine vollständige Einleitung in XML Schema zu geben, er soll viel mehr einen grundlegendes Verständnis vermitteln. Alle hier nicht vorgestellten XSD-Elemente, die für die spätere Grammatik verwendet werden, werden in Kapitel 3 genauer erklärt.

Neben XSD gibt es noch weitere Schemasprachen, um Dokumenttypdefinitionen auszudrücken. Die Bekannteste ist die Document Type Definition (DTD). Sie gilt allerdings als veraltet und XML Schema hat einige Vorteile [W3C04a][W3C04b]:

- Es ist erweiterbar und bietet mehr Funktionalität.
- Es wird selbst in XML beschrieben, was zum Beispiel Transformationen ermöglicht.
- Es unterstützt Datentypen und Namensräume.

Weitere hier nicht näher betrachtete Schemasprachen sind RELAX NG[1] und Schematron[2].

2.3.1 Aufbau einer XML Schema Definition

Ein Schema beginnt immer mit einem `schema`-Element. Darunter folgen eine Reihe von Subelementen, wie `element`, `attribute`, `complexType` und `simpleType`. Diese bestimmen letztendlich die Semantik und die Inhalte der XML. Ein Schema für das Beispiel 1 könnte wie in Beispiel 2 gezeigt aussehen.

Wie man sieht, wird zu erst ein Element `Literaturverzeichnis` erzeugt, welches einen neuen Typ `LVType` hat. Da dies kein Standardtyp ist, muss er später im Dokument definiert werden. Darunter sieht man die Definition von `LVType`. Es ist ein komplexer Typ der Buch-Elemente enthält. Diese sind wiederum von einem neuen Datentyp `BuchType`, welcher als Nächstes definiert wird. Er enthält mindestens ein Element `Autor` und ein Attribut `Titel`. Im `element`-Tag von `Buch` und `Autor` sorgt `maxOccurs="unbounded"` dafür, dass beliebig

1 http://relaxng.org/
2 http://www.schematron.com/

viele Elemente des Typs erzeugt werden dürfen. Standardwert sowohl für `maxOccurs` als auch für `minOccurs` ist 1.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Literaturverzeichnis" type="LVType" />
    <xsd:complexType name="LVType">
        <xsd:sequence>
            <xsd:element name="Buch" type="BuchType"
                maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="BuchType">
        <xsd:sequence>
            <xsd:element name="Autor" type="xsd:string"
                maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="Titel" type="xsd:string" />
    </xsd:complexType>
</xsd:schema>
```

Beispiel 2: Ein passendes XML-Schema für Beispiel 1

2.3.2 Einfache Typen

XML Schema stellt eine Reihe von grundlegenden atomaren Datentypen bereit, wie sie zum Teil auch in anderen Sprachen spezifiziert sind. Die Wichtigsten davon sind [W3C04a]:

- `xsd:string`
- `xsd:integer`
- `xsd:float`
- `xsd:boolean`
- `xsd:date`

Diese Typen können durch `simpleType`-Elemente zu neuen Datentypen definiert werden. Das folgende Beispiel zeigt einen neu definierten einfachen Datentyp `Monat`:

```
<xsd:simpleType name="Monat">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Januar" />
        <xsd:enumeration value="Februar" />
        <!-- und so weiter -->
        <xsd:enumeration value="Dezember" />
    </xsd:restriction>
</xsd:simpleType>
```

Beispiel 3: Ein einfacher Datentyp `Monat`

Das `restriction`-Element wird verwendet, um den bestehenden Basistyp anzugeben. Die `enumeration`-Aufzählung kann benutzt werden, um den Typ auf die in der Aufzählung vorkommenden Werte zu beschränken. Einfache Typen können keine Kindelemente oder Attribute haben.

2.3.3 Komplexe Typen, Elemente und Attribute

Neben den einfachen Typen gibt es noch komplexe Typen, die Elemente und Attribute beinhalten können. Ein komplexer Typ kann beispielsweise wie folgt aussehen:

```
<xsd:complexType name="BuchType">
    <xsd:sequence>
        <xsd:element name="Autor" type="xsd:string"
                     maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="Titel" type="xsd:string" />
</xsd:complexType>
```

Beispiel 4: Ein komplexer Datentyp

Man sieht den komplexen Typ `BuchType`, welcher mindestens ein Element `Autor` und das Attribut `Titel` enthält. Denkbar wären auch weitere einfache oder komplexe Typen als Kindelemente und eine Reihe weiterer Regeln an die XML.

2.4 Extensible Stylesheet Language Transformation (XSLT)

Die Extensible Stylesheet Language (XSL) ist eine in XML formulierte Notation zur Definition von Layouts von XML-Dokumenten. XSLT hingegen ist eine Subsprache von XSL und wird zur Übersetzung (Transformation) eines XML-Dokuments in ein anderes XML- oder Text-Dokument verwendet [W3C07a]. Durch die Tatsache, dass XSLT selbst XML ist und sich somit selbst transformieren kann, entstehen interessante Möglichkeiten, da Mehrfach- bzw. Zwischentransformationen möglich sind.

In dieser Arbeit wird XSLT dafür verwendet, aus der domänenspezifischen Sprache eine Webanwendung zu generieren.

2.4.1 Funktionsweise eines XSLT Prozessors

Um ein XML-Dokument in ein anderes Dokument zu übersetzen, ist neben einem XML-Dokument und einer XSL-Beschreibung noch ein XSLT Prozessor nötig, der die Transformation durchführt [Kay04a]. Neben bereits bestehenden XSLT Prozessoren, wie sie in den meisten modernen Webbrowsern implementiert sind, kann man auch selbst welche entwickeln. Abbildung 4 zeigt, wo sich der XSLT Prozessor in dem Ablauf der Transformation wiederfindet.

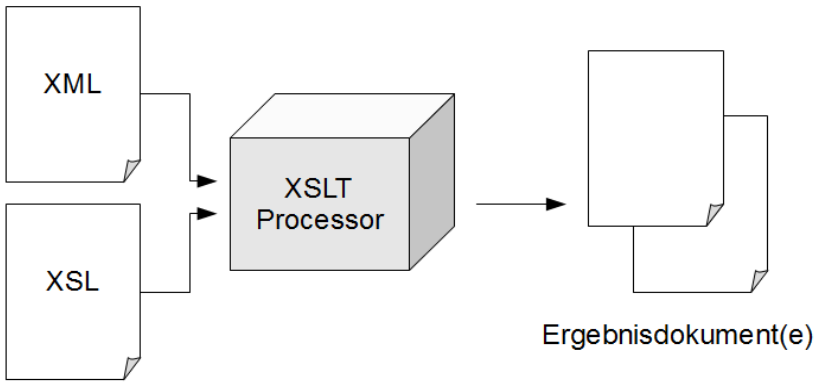


Abbildung 4: Eingliederung eines XSLT Prozessors

Der XSLT Prozessor bekommt die XML und die XSL mit den Transformationsvorschriften übergeben und erzeugt daraus ein oder mehrere Ergebnisdokumente.

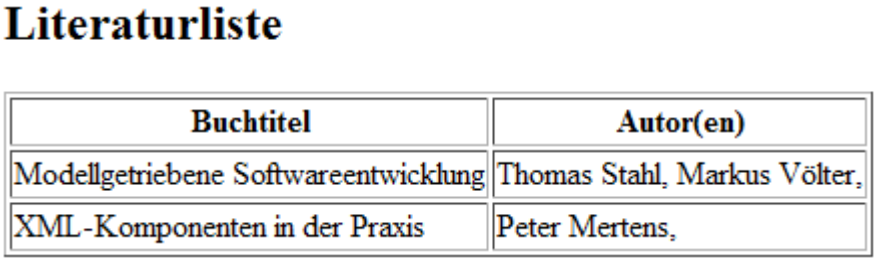
2.4.2 Struktur einer XSL-Datei

Eine XSL-Datei beginnt stets, wie alle XML-Dokumente, mit einem einleitenden XML-Kommando, in dem die Version festgelegt wird. Danach folgt ein `stylesheet`-Tag, welches den eigentlichen Codeblock einleitet. Dort wird auch die Version und der Namespace des XSL festgelegt. Über `template`- und `import`-Elemente lassen sich Transformationsvorschriften in mehrere Dateien gliedern. Jeglicher Text (oder XML/HTML-Code), der nicht XSL ist, wird im späteren Zieldokument direkt übernommen [Kay04a]. Für das Literaturverzeichnis aus Beispiel 1 könnte eine dazu passende XSL-Datei folgendermaßen aussehen:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:template match="/">
      <html>
      <body>
      <h2>Literaturliste</h2>
      <table border="1">
            <tr>
                  <th>Buchtitel</th>
                  <th>Autor(en)</th>
            </tr>
      <xsl:for-each select="Literaturverzeichnis/Buch">
            <tr>
                  <td><xsl:value-of select="@Titel"/></td>
                  <td>
                  <xsl:for-each select="Autor">
                        <xsl:value-of select="text()" />,
                  </xsl:for-each>
                  </td>
            </tr>
      </xsl:for-each>
      </table>
      </body>
      </html>
      </xsl:template>
</xsl:stylesheet>
```

Beispiel 5: Struktur einer XSL-Datei

Unterhalb des `template`-Elements finden sich (X)HTML Anweisungen. Die `for-each`-Anweisung wird für jedes Element „Buch“ im Element „Literaturverzeichnis“ durchlaufen. Das `value-of`-Element selektiert einen Inhalt und gibt diesen aus. Wenn man Beispiel 1 und Beispiel 5 durch einen XSLT Prozessor transformieren lässt, erhält man im Webbrowser folgendes Ergebnis:



Literaturliste

Buchtitel	Autor(en)
Modellgetriebene Softwareentwicklung	Thomas Stahl, Markus Völter,
XML-Komponenten in der Praxis	Peter Mertens,

Abbildung 5: Transformationsergebnis im Webbrowser

Während die Darstellung des Transformationsergebnises von Browser zu Browser leicht variieren kann, ist der zu Grunde liegende, erzeugte Quellcode bei jedem XSLT Prozessor der Gleiche. Aus der gewählten Kombination von Beispiel 1 und Beispiel 5 wird folgendes Resultat erzeugt:

```
 <html><body><h2>Literaturliste</h2><table
border="1"><tr><th>Buchtitel</th><th>Autor(en)</th></tr><tr><td>Modellge
triebene Softwareentwicklung</td><td>Thomas Stahl,
                        Markus Völter,
                        </td></tr><tr><td>XML-Komponenten in der
Praxis</td><td>Peter Mertens,
                        </td></tr></table></body></html>
```

Beispiel 6: Code des Transformationsergebnises

Das Transformationsergebnis ist nicht lesbar formatiert, alle HTML-Tags sind unmittelbar aneinander gehängt. Hierbei handelt es sich um eine Schwachstelle der Transformation. In Kapitel 2.5.3 wird darauf näher eingegangen und es werden mögliche Lösungen diskutiert.

2.4.3 XML Path Language (XPath)

XPath ist eine vom W3C entwickelte Abfragesprache, um Inhalte eines XML-Dokuments zu adressieren [W3C07b]. Es ist Teil von XSLT und wird in verschiedenen XSL-Elementen verwendet. Jedes mal, wenn in irgendeiner Art und Weise auf einen Inhalt zugegriffen werden soll, wird XPath verwendet. Wie in Kapitel 2.2.2 erläutert, lässt sich jede XML in einer Baumstruktur darstellen. Somit ist jedes Element in einer XML über einen eindeutigen Pfad erreichbar. In Beispiel 5 ist es in den `select`-Attributen der `for-each`- und `value-of`-Elemente zu finden.

Ein XPath-Ausdruck setzt sich zusammen aus einem oder mehreren Lokalisierungsschritten, die mit einem Schrägstrich („/“) getrennt werden. Optional gefolgt von einem oder mehreren Prädikaten, die in eckige Klammern („[...]“) geschrieben werden [Kay04b]. Mit diesen Prädikaten lässt sich das Ergebnis weiter einschränken.

Ein Lokalisierungsschritt enthält außerdem sogenannte Achsen. Durch Angabe von Achsen wird ausgehend vom aktuellen Kontextknoten in der Baumstruktur der XML navigiert. Die wichtigsten Achsen sind [Kay04b]:

Achse	adressierte Knoten	Abkürzung
child	direkt untergeordnete Knoten	wird weggelassen
self	der Knoten selbst	.
attribute	Attributknoten	@
parent	der übergeordnete Elternknoten	..

Tabelle 1: Die wichtigsten Achsen in XPath

Ein gültiger XPath-Ausdruck ohne Prädikate könnte also, wie in Beispiel 5 verwendet,

```
 Literaturverzeichnis/Buch/@Titel
```

lauten. Mit Prädikaten hat man die Möglichkeit, eine eingeschränkte Anzahl an Ergebnissen zu erhalten. Zum Beispiel würde

```
 Literaturverzeichnis/Buch[@Titel = 'XML-Komponenten in der
Praxis']/Autor
```

nur den oder die Autoren des Buches mit dem Titel „XML-Komponenten in der Praxis“ auswählen.

2.5 Modellgetriebene Softwareentwicklung

Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) befasst sich mit der Automatisierung der Softwareherstellung. In diesem Abschnitt werden die Ziele, Grundlagen und Verwendung dieser Technik erläutert. In Kapitel 2.5.3 wird außerdem auf das Problem der „sauberen“ Codeerzeugung eingegangen.

2.5.1 Ziele und Grundlagen

Folgende Ziele werden bei MDSD hauptsächlich verfolgt [SV05]:

- Eine Steigerung der Entwicklungsgeschwindigkeit durch Automation. Aus formalen Methoden kann durch Transformation lauffähiger Code erzeugt werden.
- Erstellte Modellierungssprachen und Transformationen können zur Herstellung weiterer Softwaresysteme verwendet werden.
- Bessere Wartbarkeit durch verteilte Implementierungsaspekte. Fehler können im generierten Code an einer Stelle beseitigt werden.
- Gesteigerte Softwarequalität durch den Einsatz formal definierter Modellierungssprachen und automatisierter Transformationen.

Man macht sich dabei zu Nutze, dass alle Anwendungen einer Domäne ähnliche oder sogar gleiche Eigenschaften und Anforderungen haben. Diese kann man maschinell generieren lassen, anstatt sie jedes mal neu zu implementieren. Dafür muss eine für die entsprechende Domäne angepasste Sprache entwickelt werden, die in der Lage ist, in die Zielanwendung transformiert zu werden.

Abbildung 6 zeigt die Zusammenhänge der modellgetriebenen Softwareentwicklung.

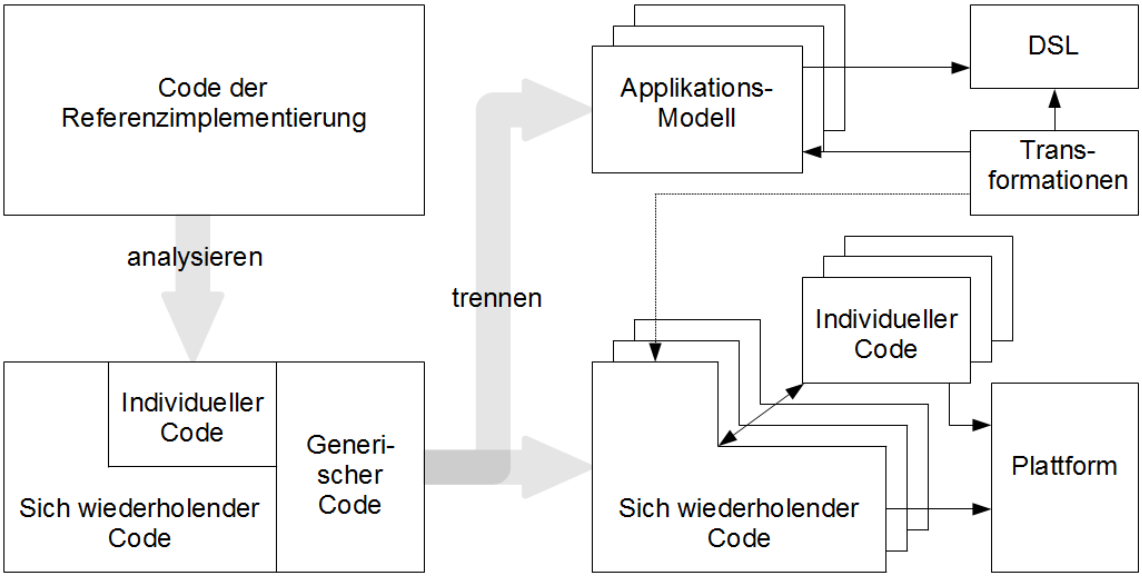


Abbildung 6: Grundidee modellgetriebener Softwareentwicklung nach [SV05]

Zuerst betrachtet man ein bestehendes Programm oder eine Referenzimplementierung. Analysiert man diesen Code, kann man ihn gedanklich in drei Teile teilen. Einen Teil der für alle

späteren Anwendungen identisch ist (Generischer Code), einen Teil der bei allen Programmen einer Domäne ähnlich, aber nicht gleich ist (sich wiederholender Code) und einen Teil mit individuellem Code, der für jede Anwendung unterschiedlich ist. Nun trennt man die Codeteile. Den generischen Code kann man vorerst vernachlässigen [SV05]. Da er für jede Anwendung gleich ist, kann er später recht einfach generiert werden. Individueller Code kann von einem Generator nicht erzeugt werden, er muss vom Entwickler manuell programmiert werden. Die modellgetriebene Softwareentwicklung hat zum Ziel, den sich wiederholenden Code aus dem Modell abzuleiten. Dafür ist eine Transformation nötig, die auf eine DSL zurückgreift.

2.5.2 Integration

Es gibt verschiedene Möglichkeiten die Prinzipien der MDSD zu implementieren. Das größte Augenmerk sollte bei der Implementierung darauf gelegt werden, dass der später generierte Code vom manuell implementierten Code getrennt ist. Sobald die Gefahr besteht, dass manuell geschriebene Programmteile durch die erneute Generierung überschrieben werden könnten, wird MDSD von den Entwicklern gemieden. Das einfachste Vorgehen ist im Generat geschützte Bereiche zu markieren, in denen der manuelle Code eingefügt werden kann. Auf diese Weise arbeiten die meisten UML-Werkzeuge, die die Möglichkeit zur automatischen Codegenerierung bieten [SV05]. Dieses Vorgehen hat einige entscheidende Nachteile:

- Der Generator ist komplexer, da er mit den geschützten Bereichen umgehen muss. Das bedeutet, er muss in der Lage sein sie zu erkennen und zu erhalten.
- In der Praxis geht immer wieder Code aus den geschützten Bereichen verloren.
- Die Trennung von generiertem und manuell erstelltem Code wird aufgeweicht, da beide in derselben Datei/Klasse stehen.

Der zweite Punkt veranlasst zur Ablehnung von MDSD, während der dritte Punkt problematisch ist, weil der Entwickler im generierten Code arbeiten muss.

In objektorientierten Sprachen ist es eine bewährte Technik, das Generat vom manuellen Code, wie in Abbildung 7 gezeigt, zu trennen:

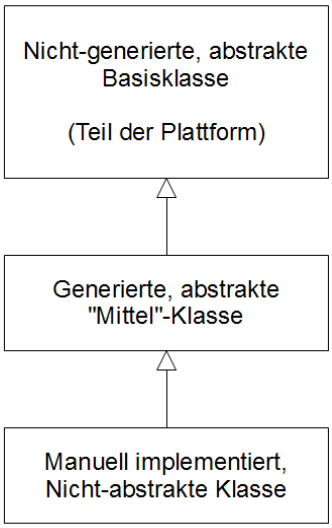


Abbildung 7: Dreischichtige Implementierung

Die Transformation generiert eine – eventuell abstrakte – „Mittel“-Klasse. Die Abstraktion bewirkt dabei, dass sie später vom Entwickler auf jeden Fall implementiert werden muss.

2.5.3 Pretty Printing

In der Softwareentwicklung ist ein wichtiges Qualitätsziel die gute Lesbarkeit von Quelltexten und die damit einhergehende bessere Wartbarkeit von Programmen. Die meisten Compiler sehen großzügig darüber hinweg, wie Code formatiert ist. Es existieren aber auch höhere Programmiersprachen, wie zum Beispiel Python[1], bei denen die Einrückung zur Blockbegrenzung gebraucht wird. Außerdem wird jeder Mensch, der sich den Code einmal ansieht, froh über einen guten Programmierstil und eine lesbare Formatierung sein.

An dieser Stelle treten, wie in Kapitel 2.4.2 gesehen, bei der Transformation mittels XSLT Probleme auf. Der Übersetzer kann Einrückungen, Leerzeichen und Absätze von XSL-Datei und Zieldokument nicht klar trennen. Die Folge: Eine der beiden Seiten der Transformation ist unschön formatiert und damit unleserlich. Gute Wartbarkeit braucht man aber auf beiden Seiten. Will jemand die Transformationsvorschrift bearbeiten, möchte er sich nicht mit schlechten Einrückungen herumschlagen. Man könnte also zu dem Schluss kommen, dass das Generat Abstriche in der Formatierung hat. Doch ist es falsch zu denken, dass niemals ein Entwickler einen Blick in das Generat werfen wird. [SV05]

Abhilfe kann ein Pretty Printer (auch Code Beautifier oder Code Formatter genannt) schaffen. Dieser bekommt den zu formatierenden Quelltext übergeben und erzeugt aus ihm, im besten Fall nach festlegbaren Regeln, einen formatierten und besser lesbaren Code.

1 http://www.python.org/

3. Domänenspezifische Sprache für Webapplikationen

Zu Beginn dieses Kapitels ist es sinnvoll, eine klare Trennung zwischen der DSL, XML und grafischer Notation herauszustellen. Die DSL ist die eigentliche Sprache, die auf der Technologie von XML beruht. Durch die Definition eines XML Schemas wird die DSL eindeutig formuliert. Sie erhält den Namen *Web State Language* (WSL).

Die grafische Notation ist lediglich eine Abbildung der WSL (vergleiche Kapitel 4), um sie zeichnen und daraus exportieren zu können.

XML als Grundlage der Sprache wurde aus verschiedenen Gründen gewählt:

- XML ist ein Standard des W3C [W3C08].
- Es ist weit verbreitet und als Datenaustauschformat anerkannt [Mer03].
- Es ist eine Betriebssystem unabhängige Technologie [Mer03].
- Eine umfangreiche Grammatik lässt sich mit XSD unkompliziert erstellen [W3C04a].
- Es existiert mit XSLT bereits eine Möglichkeit zur Transformation [W3C07a].

Eine Alternative zur XML-Lösung wäre das eigenhändige Definieren von Sprachkonstrukten. Dann müssten allerdings auch eigenständig ein Parser und Transformator entwickelt werden. Neben dem erhöhten Aufwand, sprechen vor allem die oben genannten wegfallenden Vorteile gegen eine Eigenproduktion.

Eine andere Alternative ist Xtext. Dies ist ein Open-Source-Framework, mit dem man sowohl Programmiersprachen als auch domänenspezifische Sprachen entwickeln kann [Xtext10]. Außerdem ist es Teil des Eclipse-Modeling-Frameworks (EMF) [TEF10]. Bei der Sprachentwicklung mit Xtext, werden automatisch ein Codeeditor und EMF-Modelle, die unabhängig von Eclipse verwendet werden können, generiert. Im Funktionsumfang des Editors sind unter anderem Autovervollständigung, Syntax Highlighting und Outline Views enthalten. Mit Xtext lassen sich schlanke Sprachen entwickeln, die ohne die XML-typischen spitzen Klammern und Tags auskommen. Dennoch gab es einige Gründe sich gegen Xtext zu entscheiden:

- Xtext ist kein Standard.
- Man ist weniger flexibel bei der Wahl des grafischen Modellierungswerkzeugs.
- Es existierte zu Beginn der Bachelorarbeit noch keine finale Veröffentlichung von Xtext. Die Version 1.0 erschien erst zur Mitte der Bearbeitungszeit [Xtext10].

3.1 Anforderungen und Entwurf

Generelle Anforderungen an die Sprache lassen sich teilweise direkt aus der Domäne ableiten. Wie in Kapitel 2.1 beschrieben, sollen Webanwendungen erzeugt werden können, die Zustände haben, die über Transitionen erreichbar sind. Diese Domänenanalyse führte dazu, dass beides in die Sprache eingeflossen ist. Anfangs war geplant die Übergänge mit in die Zustände zu modellieren. Hiervon wurde schnell wieder abgewichen, da es die Sprache unnötig kompliziert machen würde. Auch müssen sich Objekte von Webanwendungen in der Sprache wiederfinden. Da diese unabhängig von Zuständen und Transitionen sind, muss für Modellklassen ein eigenes Element eingeführt werden. Als Wurzelelement wird „program“ gewählt. Es ergibt sich folgendes Grundgerüst:

```
<program>
    <models>...</models>
    <states>...</states>
    <transitions>...</transitions>
</program>
```

Grundgerüst der WSL

Models soll Objekte, die in einer Webanwendung vorkommen können enthalten, wie zum Beispiel einen Warenkorb bei einer Shopanwendung. In *States* sollen die Zustände und in *Transitions* die zugehörigen Übergänge angegeben werden.

Da diese Art der Darstellung bei einer großen Anzahl von Sprachkonstrukten unübersichtlich ist, wird nachfolgend eine äquivalente grafische Notation, die respektive kein UML ist, verwendet:

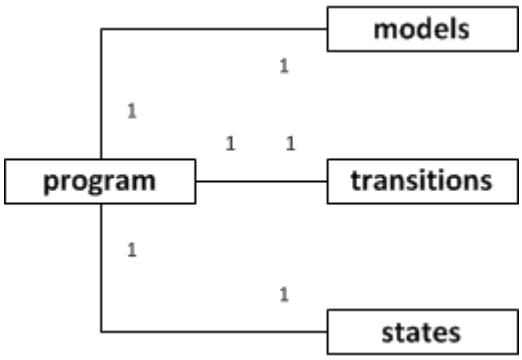


Abbildung 8: Grundgerüst der WSL

Da weitere Entscheidungen die Sprache zu verfeinern sich nicht einfach erdenken lassen, wurde zur zusätzlichen Anforderungserhebung eine Referenzimplementierung einer Webanwendung erstellt und analysiert.

3.1.1 Modellklassen

Die Analyse ergab für Modellklassen, dass

- diese einen Namen haben müssen,
- verschiedene Eigenschaften haben können und
- mehrere Operationen bereitstellen können.

Die Eigenschaften haben ebenfalls einen Namen und sind von einem bestimmten Typ. Optional können sie noch einen Initialwert annehmen und eine Assoziation mit einer anderen Modellklasse haben.

Die Operationen haben auch einen Namen und optional einen Rückgabewert von einem festzulegenden Typ. Außerdem können sie Übergabeparameter entgegennehmen, die wiederum einen Namen und einen Typ haben.

In Abbildung 9 wird das Resultat aus den Überlegungen gezeigt.

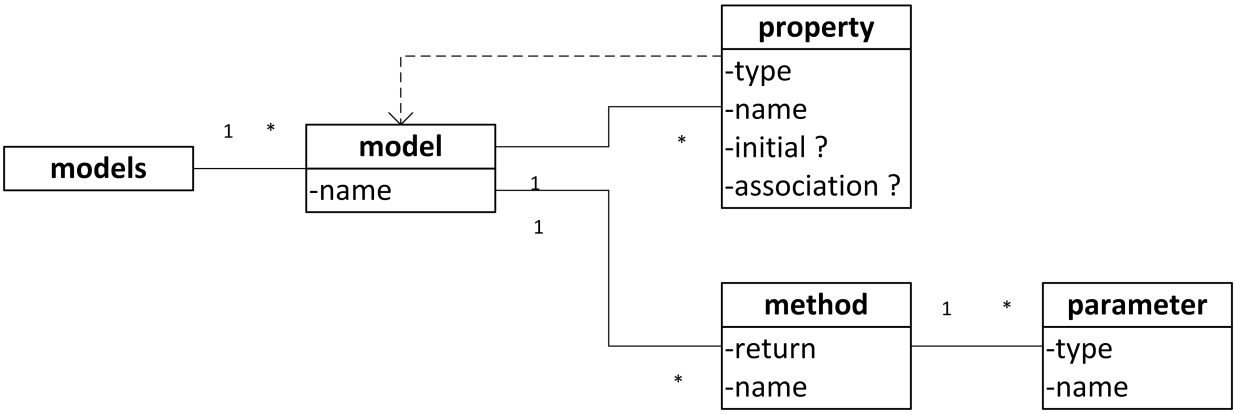


Abbildung 9: Modellklassen in der WSL

Eine „1 zu *“-Beziehung bedeutet, dass beliebig viele dieser Elemente enthalten sein dürfen. Das „?“ bedeutet, dass das damit gekennzeichnete Attribut optional ist. Der gestrichelte Pfeil sagt aus, dass auf dieses Element zugegriffen werden kann. Der Name einer Eigenschaft, kann gleichzeitig der Name einer Modellklasse sein.

3.1.2 Zustände

In der Sprache sollte sich widerspiegeln, dass jeder Zustand

- einen Namen hat und zu einer Gruppe gehört,
- einen Titel hat, der angezeigt wird, wenn man den Zustand erreicht und
- beliebig viele Parameter haben kann.

Es gibt verschiedene Möglichkeiten das oben Genannte umzusetzen. So war anfangs geplant ein *state*-Element zu verwenden, welches als Attribute Name, Titel und Gruppe hat. Dies wäre ohne weiteres möglich gewesen, aber übersichtlicher und weniger redundant ist es, die Gruppenzugehörigkeit in ein eigenes Element zu gliedern. Das Resultat ist in Abbildung 10 zu sehen.

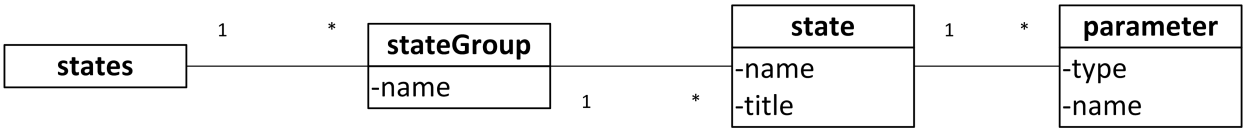


Abbildung 10: Zustände in der WSL

Das *stateGroup*-Element darf beliebig viele *state*-Elemente enthalten, die wiederum beliebig viele *parameter*-Elemente haben dürfen. Die *parameter*-Elemente sind genauso aufgebaut wie bei den Modellklassen.

3.1.3 Transitionen

Die Transitionen stellen den komplexesten Teil der WSL dar. Zum einen gibt es verschiedene Arten von Übergängen. Bei einfachen Übergängen gibt es eine feste Anzahl von Transitionen von einem Zustand in einen Anderen. Es ist aber auch denkbar, dass die Anzahl der Transitionen variabel ist und davon abhängt wie viele Datensätze in der Datenbank vorhanden sind.

Ein Beispiel: Es gibt die zwei Zustände „Notizen anzeigen“ und „Notiz gelöscht“. Bei „Notizen anzeigen“ werden alle in der Datenbank vorhandenen Notizen angezeigt. Neben jeder Notiz befindet sich ein Link zum Löschen. Wenn man diesen anklickt, kommt man in den Zustand „Notiz gelöscht“. Es versteht sich von selbst, dass die zu löschende Notiz (oder zumindest deren eindeutiger Identifikator) in der Transition übertragen werden muss. Also gibt es für jede Notiz eine eigene Transition mit einem entsprechenden Parameter (Liste von Transitionen).

Ein weiterer Grund der erhöhten Komplexität des *transition*-Elements ist, dass es auch Zustände geben kann, die keine eigene Repräsentation haben. Die zwar etwas ausführen, dann aber nichts anzeigen oder ausgeben, sondern in einen nächsten Zustand überleiten. Im Notiz-Beispiel wäre denkbar, dass nachdem eine Notiz gelöscht wurde, direkt wieder die Liste angezeigt wird und man nicht im „Notiz löschen“ Zustand verweilt.

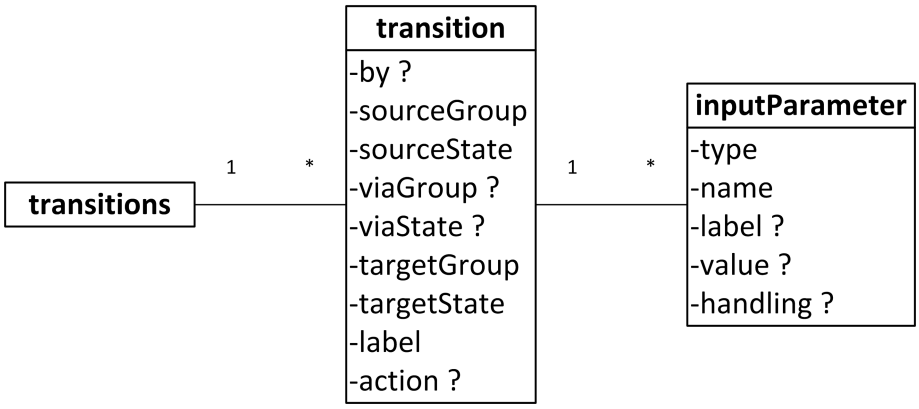


Abbildung 11: Transitionen in der WSL

Das *by*-Element wird als leer angenommen. Deshalb muss es nicht angegeben werden. Wenn es aber angegeben wird, muss es den Namen eines *model*-Elements (siehe Abschnitt 3.1.1) enthalten. Ist es nicht angegeben, handelt es sich um eine einfache Transition. Wird das *by*-Element mit einer Modellklasse versehen, folgt daraus automatisch, dass es eine Liste von Transitionen ist.

Die darunter folgenden sechs Attribute müssen alle einen Zustand referenzieren und zwar jeweils mit Gruppe, da nicht ausgeschlossen ist, dass Zustandsnamen aus verschiedenen Gruppen gleich sind. Sollte es sich, wie oben erläutert, um eine Weiterleitungstransition handeln, sind viaGroup und viaState anzugeben.

Mit dem *label*-Attribut kann die Transition beschriftet werden. Dabei gibt es zwei Fälle:

- Bei einem Link wird es als Linktext verwendet.
- Bei einem Eingabefeld wird der zugehörige Button zum Abschicken des Formulars mit dem *label*-Wert benannt.

Das *action*-Attribut kann die Werte „read“ und „write“ annehmen. Als Standard wird „read“ gesetzt, von daher ist dieses Attribut optional. Diese Unterscheidung ist sinnvoll, da die Datenübertragung mit HTML entweder über „get“ oder über „post“ abgewickelt werden kann. Dabei wird „get“ immer dann verwendet, wenn Daten abgefragt werden und „post“, wenn Daten gespeichert werden sollen. Hat eine Transition den *action*-Wert „read“ wird sie lesende Transition genannt. Analog für „write“ schreibende Transition.

Jede Transition kann ein oder mehrere *inputParameter*-Elemente enthalten. Diese geben an, welche Daten an den Folgezustand weitergegeben werden und in welcher Form diese sind. Der Typ und der Name muss immer angegeben werden. Zusätzlich optional sind eine Beschriftung (*label*) und ein Wert (*value*). Werden diese nicht angegeben, wird eine leere Zeichenkette als

Standardwert für diese beiden Attribute verwendet. Für das *handling*-Attribut stehen die vier Möglichkeiten „visible“, „readonly“, „masked“ und „hidden“ bereit. Die genauen Bedeutungen für diese Werte können Tabelle 2 entnommen werden.

Attributwert	Auswirkung	Sonstiges
visible	Ein Feld zur Dateneingabe wird bereitgestellt	Standardwert
readonly	Es wird ein Datenfeld angezeigt. Der Wert darin kann nicht editiert werden.	
masked	Es wird ein maskiertes Eingabefeld bereit gestellt, wie zum Beispiel bei Passwörtern üblich.	
hidden	Es wird nichts angezeigt, der Wert wird unsichtbar übergeben.	

Tabelle 2: Mögliche Werte des handling-Attributs

Wie oben bereits erwähnt, greift das Transitionselement sowohl auf das Modellelement als auch auf das Zustands- und Zustandsgruppenelement zu. Daher ist Abbildung 11 streng genommen nicht ganz korrekt, da diese Zusammenhänge nicht dargestellt werden. In Abbildung 12 ist dieser Bezug durch gestrichelte Pfeile dargestellt. Des Weiteren kann das `parameter`-Element von *state* und *method* zusammengefasst werden.

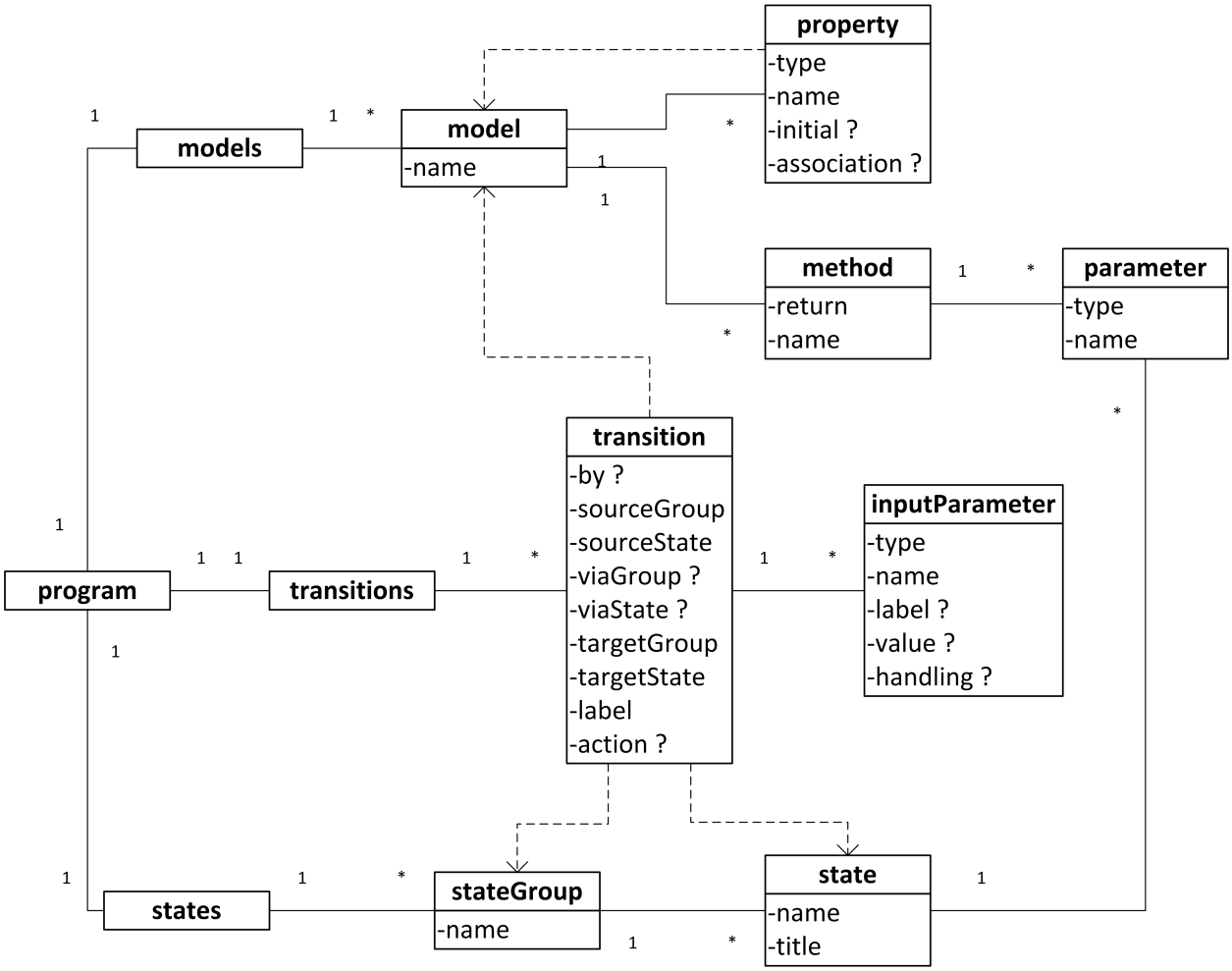


Abbildung 12: Metamodell der WSL

3.2 WSL Definition als XML Schema

Die Überlegungen aus dem vorhergehenden Abschnitt resultierten in einem Metamodell, das für die WSL verschiedene Elemente mit entsprechenden Unterelementen und Attributen vorsieht. Verbal eine Sprache zu beschreiben ist jedoch nicht verbindlich. Erst durch eine Grammatik wird eine Sprache richtig fixiert. Bei Sprachen, die auf XML aufbauen, bietet sich dafür XML Schema an. Wie in Kapitel 2.3 erläutert, lassen sich damit Regeln aufstellen und ein XML-Dokument damit prüfen. Nur wenn alle Regeln erfüllt werden heißt die XML *gültig* und ist somit semantisch korrekt.

Das Wurzelelement `program` wird mit

```
<xsd:element name="program" type="programType" />
```

festgelegt. Ihm wird der Typ `programType` zugeordnet. Diesen gilt es im Folgenden zu definieren. Da dem Wurzelelement immer die drei Elemente models, states und transitions nachgehen, kann programType folgendermaßen beschrieben werden:

```
<xsd:complexType name="programType">
    <xsd:all>
        <xsd:element name="models" type="modelsType" />
        <xsd:element name="states" type="statesType" />
        <xsd:element name="transitions" type="transitionsType" />
    </xsd:all>
</xsd:complexType>
```

Das `all`-Element sagt aus, dass die nachfolgenden Elemente in beliebiger Reihenfolge und höchstens einmal vorkommen dürfen.

Nachfolgend wird auf `modelsType` näher eingegangen. Die vollständige XSD-Datei befindet sich auf der beiliegenden CD.

Auf das `models`-Element in der WSL folgen stets beliebig viele `model`-Elementen. Diese wiederum haben als Attribut einen Namen und beliebig viele Unterelemente `method` und `property`:

```
<xsd:complexType name="modelType">
    <xsd:sequence>
        <xsd:choice maxOccurs="unbounded">
            <xsd:element name="method" type="methodType"
                         minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="property" type="propertyType"
                         minOccurs="0" maxOccurs="unbounded"/>
        </xsd:choice>
    </xsd:sequence>
    <xsd:attribute name="name" type="modelNameType" />
</xsd:complexType>
```

Der Name einer jeden Modellklasse beginnt immer mit einem Großbuchstaben, gefolgt von beliebig vielen weiteren Buchstaben und Zahlen. Sonder- und Leerzeichen sind nicht gestattet. Diese Einschränkungen lassen sich vornehmen, indem man das `modelNameType`-Attribut entsprechend einschränkt:

```
<xsd:simpleType name="modelNameType">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[A-Z]([a-z]|[A-Z]|[0-9])*" />
    </xsd:restriction>
</xsd:simpleType>
```

Interessant sind noch die Eigenschaften einer Modellklasse. Diese haben jeweils als Attribute einen Typ, einen Namen und gegebenenfalls einen Initialwert und eine Assoziation. Dabei darf eine Assoziation nur genau einen von vier Werten annehmen. Einschränken lässt sich diese Bedingung mit

```
<xsd:simpleType name="associationType">
    <xsd:restriction base="xsd:string">
          <xsd:enumeration value="1-1" />
          <xsd:enumeration value="n-1" />
          <xsd:enumeration value="1-n" />
          <xsd:enumeration value="n-m" />
    </xsd:restriction>
</xsd:simpleType>
```

Auf viele weitere Einschränkungen und Definitionen wurde an dieser Stelle nicht weiter eingegangen, da sich die meisten Sprachkonstrukte in der XSD-Datei wiederholen.

4. Entwicklung einer grafischen Notation

Im vorhergehenden Kapitel wurden die Entwurfsentscheidungen und die resultierende Struktur der domänenspezifischen Sprache erläutert. In diesem Kapitel wird sie jetzt praktisch angewandt. Ziel ist es durch eine grafische Notation im Modellierungswerkzeug ProFLOW die WSL darstellen und durch eine Export-Funktion erzeugen zu können. Die WSL wäre, nach der in Kapitel 3 vorgestellten Form, bereits vollständig einsetzbar. Dennoch wird eine grafische Notation entwickelt, die zum Ziel hat:

- eine grafische Repräsentation der WSL zu sein,
- den Entwicklungsprozess von Webanwendungen zu beschleunigen und intuitiver zu machen und
- den Entwurfsprozess von Webanwendungen zu unterstützen.

Anstelle einer Eigenentwicklung könnte man auch auf Bestehendes zurückgreifen. Mit der Unified Modeling Language (UML) [OMG10] existiert eine weit verbreitete Sammlung grafischer Sprachen, die einige Vorteile mitbringen:

- UML ist ein Standard,
- UML ist weit verbreitet und für Benutzer mit Informatikhintergrund intuitiv und
- es existieren Zustands- und Klassendiagramme.

Aber es gibt trotzdem gute Gründe eine eigene Notation zu verwenden. Sonst bestende die Gefahr fehlgeleitet zu werden. Ein Entwickler denkt, dass er versteht was er macht, da er mit vertrauten Notationen arbeitet. Aber das ist falsch, da man die domänenspezifische Sprache und nicht UML verstehen muss.

Weiterhin sollte UML nur für das verwendet werden wofür es gedacht ist, also für die bessere und vereinheitlichte Kommunikation zwischen Menschen. Es wurde nicht geschaffen um Code zu erzeugen [Völ08]. Da UML so gut wie keine textuelle Unterstützung bietet, also das Gezeichnete sich nicht in einer Textnotation verfassen lässt, sollte es bei einer auf Text basierenden DSL, wie sie in dieser Arbeit vorliegt, nicht verwendet werden.

Mit der Web Modeling Language (WebML) [WRG10] existiert eine Modellierungssprache, speziell für die Domäne Webanwendung, die UML erweitert und zum Ziel hat, den Entwicklungsprozess von Webprojekten zu unterstützen [MFV07]. Weiterhin wurde von den Entwicklern von WebML erkannt, dass bei der Entwicklung einer Webanwendung verschiedene Fachbereiche wie zum Beispiel Datenbank- und Oberflächenprogrammierung zum Einsatz kommen. WebML berücksichtigt verschiedene Arten von Spezialisten, die unterschiedliche Rollen annehmen [MM07].

So teilt sich die Modellierung von WebML in ein Struktur-, ein Hypertext-, ein Präsentations- und ein Personalisierungs-Modell [MM07]. Mit Webratio[1] existiert ein kommerzielles Werkzeug zur Erstellung der Modelle und zur automatischen Generierung von Code. Eine kostenlose Version ist ebenfalls verfügbar. Allerdings ist diese Version stark in der Funktionalität eingeschränkt.

WebML wird in dieser Arbeit nicht verwendet, weil es für statische Webseiten ungeeignet ist. Ziel der WSL ist es aber, sowohl dynamisch als auch rein statische Seiten einfach abbilden zu können. Außerdem sind zu WebML bestehende Modellierungswerkzeuge, wie Webratio, nur proprietär.

1 http://www.webratio.com/

Des Weiteren bedeutet eine Eigenentwicklung nicht automatisch, dass man sich nicht an bestehenden Diagrammarten orientieren kann. Ziel ist es, sowohl ein vertrautes Gefühl beim Benutzer zu erzeugen, aber dennoch zu vermitteln, dass es sich bei der Sprache um etwas anderes handelt als er schon kennt.

Weitere Alternativen, auf die hier nicht weiter eingegangen wird, sind zum Beispiel UML-based Web Engineering (UWE) [WEG10] und Object-Oriented Hypermedia (OO-H) [GC03].

4.1 Grafische Darstellung der WSL

Um die Ziele aus dem vorhergehenden Abschnitt zu erreichen, wurde bei dem Entwurf der grafischen Notation auf folgende Dinge geachtet:

- Die Notation soll einfach zu verstehen sein und vertraut wirken.
- Ohne lange Einarbeitungszeiten sollen korrekte WSL-Dokumente erzeugt werden können.

Dabei gibt die WSL selbst alle notwendigen Elemente vor, die die grafische Notation haben soll. Aus den Modellklassen ergibt sich, dass es eine Art Klassendiagramm geben muss. Die Zustände legen nahe, dass man Zustandselemente vorsieht und die Transitionen sind am Besten durch Pfeile zwischen den Zuständen darstellbar.

4.1.1 Grafische Darstellung der Modellklassen

Die Modellklassen haben einen Namen, Eigenschaften und Methoden mit Parametern. Der Bedarf diese Informationen unterzubringen, klingt sehr vertraut aus den Klassendiagrammen die die UML vorsieht:

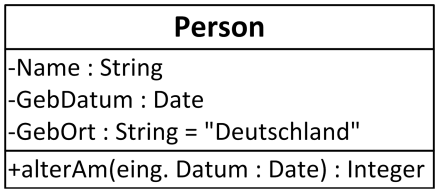


Abbildung 13: UML Klassendiagramm

Abbildung 13 zeigt ein Klassendiagramm in UML. Der Klassenname steht an oberster Stelle, darunter die Attribute und dann die Operationen. Die Attribute haben jeweils einen Namen, einen Datentyp (z.B. String, Integer...) und eine Sichtbarkeit (public, private, protected, package). Optional ist ein Initialwert der direkt hinter den Datentyp geschrieben wird. Die Operationen haben ebenfalls einen Namen, einen Datentyp und eine Sichtbarkeit. Zusätzlich können noch Parameter mit Datentyp ausgedrückt werden. Diese werden in runde Klammern hinter den Methodennamen geschrieben.

Doch nicht alle in der UML vorgesehen Paradigmen des Klassendiagramms werden für diese Bachelorarbeit benötigt. So findet sich in der WSL beispielsweise keinerlei Information über die Sichtbarkeit. Diese ist unabhängig von der Sprache, in der die Webanwendung generiert wird und erst bei der Transformation festzulegen. Auch die weiteren Konstrukte in UML wie Generalisierung und Pakete werden mit Ausnahme der Assoziation nicht benötigt. Daher wäre es unangebracht, das Klassendiagramm der UML exakt zu übernehmen.

Für die grafische Notation dieser Bachelorarbeit reicht das Klassendiagramm, wie in der folgenden Abbildung zu sehen.

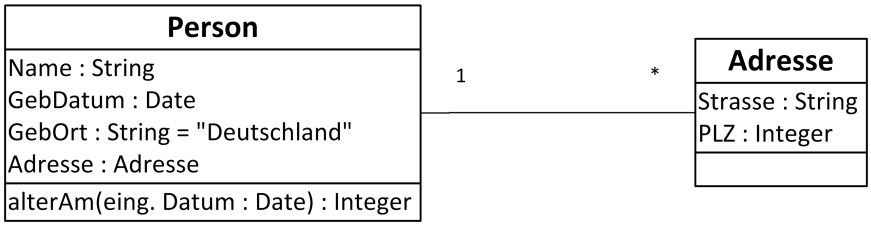


Abbildung 14: Zwei Modellklassen im Klassendiagramm

Aus Abbildung 14 lassen sich alle Informationen ablesen oder ableiten, die für die Modellklassen in der domänenspezifischen Sprache gebraucht werden. In Codeform sähe es wie folgt aus:

```
<model name="Person">
    <property type="String" name="Name" />
    <property type="Date" name="GebDatum" />
    <property type="String" name="GebOrt" initial="Deutschland" />
    <property type="Adresse" name="Adresse" association="1-n" />
    <method return="Integer" name="alterAm">
        <parameter type="Date" name="eing. Datum" />
    </method>
</model>
<model name="Adresse">
    <property type="String" name="Strasse" />
    <property type="Integer" name="PLZ" />
</model>
```

Modellklassen in WSL-Code

Die Klasse „Person“ hat vier Eigenschaften und eine Methode. Wobei die Eigenschaft „GebOrt“ einen Initialwert hat. „Adresse“ ist eine Referenz auf die Klasse „Adresse“. Die „1-*“Assoziation zwischen den Klassen drückt sich im `association`-Attribut des `property`-Elements aus. Die erste Klasse hat eine Methode mit Rückgabewert „Integer“ und einen Parameter vom Typ „Date.“ All dies wird im `method`-Element ausgedrückt. Die zweite Klasse hat keine Methoden, bei ihr gibt es deshalb auch keine `method`-Elemente.

Wie eben gezeigt, ist dieses einfache Klassendiagramm vollkommen ausreichend, um sämtliche Informationen in die Sprache zu übertragen. Es beschränkt sich auf das Wesentliche, nämlich das Darstellen von Modellklassen und lenkt dabei nicht mit zu vielen Möglichkeiten vom eigentlichen Ziel, dem Modellieren von Klassen ab.

4.1.2 Grafische Darstellung der Zustände

Das UML-Zustandsdiagramm ist dafür gedacht endliche Automaten darzustellen und wird benutzt um das Verhalten eines Systems zu spezifizieren [OMG10]. Die Zustände in Webanwendungen stellen kein Verhalten dar und die Übergänge zwischen ihnen werden nicht etwa automatisch, wie bei einem Automaten durchlaufen, sondern erst wenn der Anwender diese auswählt. An diese Übergänge ist weiterhin keine Bedingung gestellt, der Benutzer allein bestimmt welcher Transition er folgt.

Für einen Zustand wird als Darstellungsform ein Kreis ausgewählt. Sein Name soll in der Mitte des Kreises stehen. Ansonsten sollen keine weiteren Informationen im Zustand stehen, um die Übersichtlichkeit zu wahren. Weitere benötigte Informationen sollen erst beim Anwählen des Zustandes in einer gesonderten Sicht zum Vorschein kommen. Zustände als Kreise darzustellen ist aus vielen Zustandsdiagrammen eine vertraute Praxis.

Die Zustände werden in zwei Arten unterteilt:

- **Normale Zustände** haben eine schwarze Linie. Sie werden benutzt, wenn der Zustand eine eigene Darstellung hat, zum Beispiel „Notizen anzeigen“
- **Schwache Zustände** haben eine graue Linie und werden benutzt, wenn der Zustand keine eigene Darstellung hat und direkt nach dem Durchlaufen in einem anderen Zustand übergeht. Beispielsweise könnte der Zustand „Notiz gelöscht“ direkt an „Notizen anzeigen“ weiterleiten.

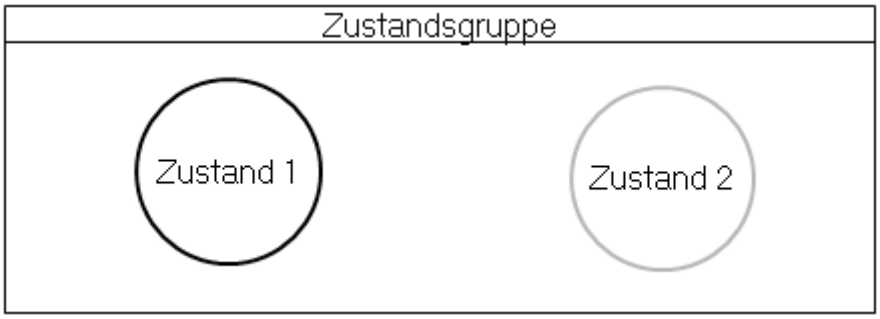


Abbildung 15: Zustandsgruppe mit einem normalen und einem schwachen Zustand

Die WSL sieht weiterhin vor, dass jeder Zustand zu einer Gruppe gehört. Gruppen werden als Rechtecke gezeichnet und mit einem Namen versehen. Alle Zustände, die zu der Gruppe gehören, werden in diesem Rechteck platziert. In Abbildung 15 sieht man zwei Zustände. „Zustand 1“ ist ein normaler und „Zustand 2“ ein schwacher Zustand. Die Gruppe hat den Namen „Zustandsgruppe“ und umrahmt die beiden Zustände. Daraus ergibt sich folgender Code in der Sprache:

```
<stateGroup name=“Zustandsgruppe“>
    <state name=“Zustand 1“ />
    <state name=“Zustand 2“ />
</stateGroup>
```

Zustände in WSL-Code

Über die weiteren Eigenschaften wie einen Titel und eventuelle Parameter lassen sich aus der Darstellung keine Auskünfte gewinnen. Wie oben erläutert, sollen diese Informationen getrennt von der grafischen Repräsentation erfasst werden.

Zuletzt fehlt noch ein Startzustand. Dieser soll ein schwarzer, ausgefüllter Kreis sein, der deutlich kleiner ist als die anderen Zustände. Durch einen Übergang vom Startzustand zu einem normalen Zustand, wird dieser als erster Zustand festgelegt, den der Nutzer sieht, wenn er die Webanwendung startet.

4.1.3 Grafische Darstellung der Transitionen

Die Anforderungen an die Übergänge zwischen den Zuständen lassen sich ebenfalls aus der WSL ableiten. Zur Erinnerung, eine Transition hat unter anderem folgende Eigenschaften:

- `sourceGroup` und `sourceState`
- `viaGroup` und `viaState`
- `targetGroup` und `targetState`
- `label`

Der Quell- und Zielzustand - inklusive Gruppe - lässt sich bei einer Transition, die als gerichtete Kante dargestellt wird, einfach ablesen. Handelt es sich bei dem Zielzustand um einen schwachen Zustand, kann daraus gefolgert werden, dass es nur ein Zwischenzustand ist. Der Zielzustand dieses Zwischenzustands wird als neuer Zielzustand der Transition gesetzt und der Zwischenzustand wird in `viaState` geschrieben.

Das `label`-Attribut, also die Beschriftung der Transition, kann an die Kante an sich geschrieben werden. Dies macht das gesamte Diagramm lesbarer.

Weitere Attribute des `transition`-Elements sind by und `action`. Es gibt insgesamt vier Möglichkeiten für die Belegung dieser Attribute:

- by enthält den Namen einer Modellklasse oder nicht.
- `action` ist entweder read oder write.

Daraus ergeben sich diese vier unterschiedlichen Transitionsarten:

action	by	Art der Transition
read	nicht gesetzt	Einfache Transition, lesender Zugriff
write	nicht gesetzt	Einfache Transition, schreibender Zugriff
read	gesetzt	Liste von Transitionen, lesender Zugriff
write	gesetzt	Liste von Transitionen, schreibender Zugriff

Tabelle 3: Die vier Arten von Transitionen

Nun müssen diese noch unterschiedlich dargestellt werden. Für schreibende Transitionen wird eine durchgezogene Linie festgelegt und für lesende Transitionen einen gestrichelte Linie. Transitionslisten sollen eine deutlich dickere Linie haben. In Abbildung 16 sieht man zwei Zustände, die durch eine einfache, schreibende Transition miteinander verbunden sind und die mit „Transition 1“ beschriftet ist:

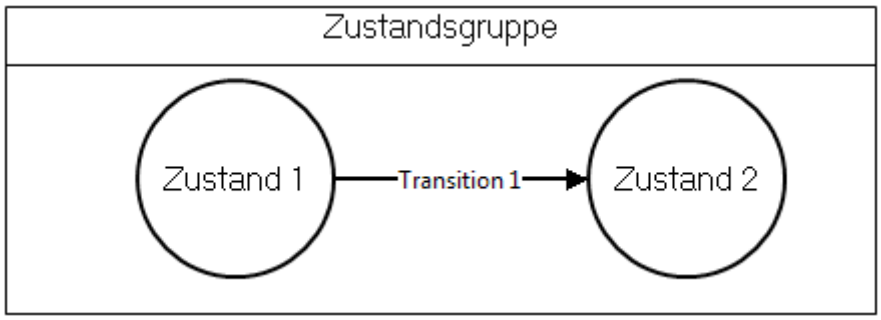


Abbildung 16: Eine einfache, schreibende Transition

Daraus lässt sich der vollständige Code für die Transition in der WSL ermitteln:

```
<transition sourceGroup=“Zustandsgruppe“ sourceState=“Zustand 1“
            targetGroup=“Zustandsgruppe“ targetState=“Zustand 2“
            label=“Transition 1“ action=“write“/>
```

Transition in WSL-Code

Zusammenfassend sind in Tabelle 4 alle Elemente der grafischen Notation mit Darstellungsform aufgelistet.

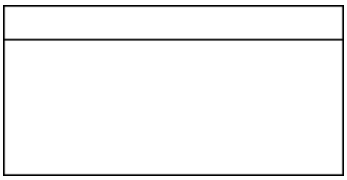
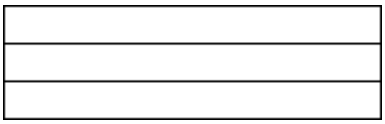
Darstellung	Beschreibung
	Einfache Transition, die einen schreibenden Zugriff auf den Zielzustand durchführt
	Einfache Transition, die einen lesenden Zugriff auf den Zielzustand durchführt
	Liste von Transitionen, die einen schreibenden Zugriff auf den Zielzustand durchführt
	Liste von Transitionen, die einen lesenden Zugriff auf den Zielzustand durchführt
	Startzustand
	Normaler Zustand
	Schwacher Zustand
	Zustandsgruppe
	Modellklasse

Tabelle 4: Übersicht über Transitionen und Elemente

4.2 ProFLOW

ProFLOW ist ein am Fachgebiet Software Engineering entwickeltes Modellierungswerkzeug zur Erstellung und Bearbeitung von FLOW-Modellen, welche einen Forschungsschwerpunkt des Fachgebiets darstellen [SE09]. Dabei wurde es als leicht erweiterbares Modellierungsframework entwickelt. Somit können einfach neue Notationen entwickelt und ausprobiert werden. Des Weiteren gehört zum Funktionsumfang von ProFLOW bereits ein XML-Export und ein Export als Grafik.

ProFLOW ist ein Eclipse-Plugin, dass auf dem Graphical Editing Framework (GEF) basiert. Abbildung 17 zeigt einen Screenshot von ProFLOW. Worin eine große Arbeitsfläche (I), in der Elemente angeordnet und bearbeitet werden können, zu sehen ist. Links daneben befindet sich eine Leiste mit Werkzeugen (II). Man kann Elemente selektieren, einfache Textbeschreibungen hinzufügen oder neue Elemente in die Arbeitsfläche ziehen. In der Outline-View von Eclipse wird das Diagramm dargestellt, um einen Überblick zu behalten (III). Öffnet man die Property-View von Eclipse finden sich dort zu jedem Objekt weiterführende Informationen, die dort auch eingefügt und bearbeitet werden können (IV).

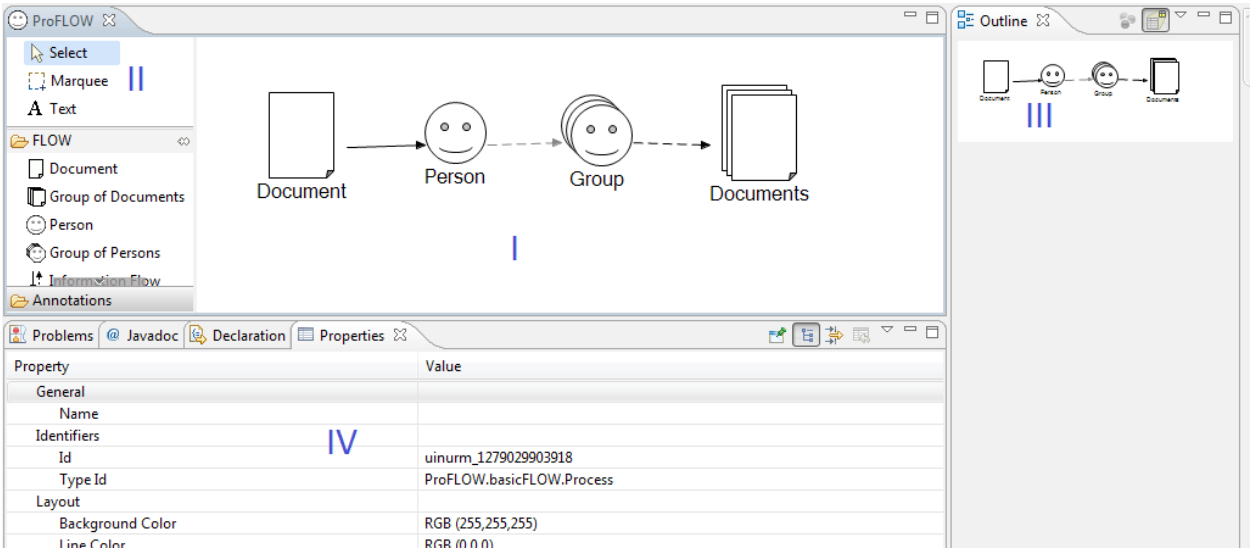


Abbildung 17: ProFLOW

4.2.1 Programmierung einer neuen Notation

ProFLOW folgt dem MVC Entwurfsmuster. Es gibt für jedes Objekt einer Notation eine Modellklasse, eine Viewklasse und eine Controllerklasse. Zusätzlich muss noch eine Plugin-Klasse geschrieben werden, die hauptsächlich dafür da ist, die Verbindung zwischen ProFLOW und grafischer Notation herzustellen.

Im Einzelnen werden folgende Objekte für die WSL benötigt:

- Ein **normales Zustandselement**, welches einen Namen, einen Titel und mehrere Parameter speichern kann.
- Ein **schwaches Zustandselement**, welches das Gleiche speichern muss wie das normale Zustandselement.
- Ein **Startelement**, welches nur einmal existieren darf, damit ausgeschlossen ist, dass mehrere Zustände als Startzustand ausgezeichnet werden können.
- Ein **Gruppenelement für Zustände**, welches einen Namen besitzt.
- Vier **Transitionenarten**, die eine Beschriftung haben und mehrere Parameter speichern können müssen (siehe 4.1.3). Jede Transition verbindet immer genau zwei Zustände miteinander.

Außerdem muss noch ein Diagramm-Element erstellt werden, welches diese Elemente enthält und für den Export als WSL (siehe Abschnitt 4.2.2) eine Klasse bereitstellt, die in der Lage ist, diesen durchzuführen.

Alle in Abschnitt 4.1 überlegten Entwurfsentscheidungen für die grafische Notation konnten in ProFLOW umgesetzt werden, mit einer Ausnahme: Die Modellklassen mussten geändert werden. Kreise, Rechtecke, Ovale und andere zweidimensionale geometrische Figuren sind einfach zu programmieren, komplexere Elemente hingegen, wie etwa ein Klassendiagramm, sieht ProFLOW aber nicht mit einfachen Mitteln vor und sind durch die besondere Struktur schwieriger umzusetzen.

Anstelle eines Klassendiagramms wird für Modellklassen jetzt eine Darstellung als Rechteck mit abgerundeten Ecken gewählt. Der Name der Modellklasse steht in der Mitte des Rechtecks.

Weitere Informationen wie die Eigenschaften und die Operationen finden sich in der Property-View von ProFLOW wieder, wo sie eingegeben und bearbeitet werden können.

Diese Kürzung ist jedoch nicht besonders ausschlaggebend, da der Fokus dieser Arbeit auf dem Darstellen von Zuständen und Transitionen liegt.

4.2.2 Export als WSL

Der bereits zum Funktionsumfang gehörende XML Export wird ausgelöst, wenn man mit der rechten Maustaste auf die Arbeitsfläche klickt und in dem erscheinenden Menü „Export as Plain XML...“ auswählt. Es lag daher nahe, den „Export as WSL...“ an die gleiche Stelle zu setzen.

Am Anfang stand allerdings eine wichtige Überlegung. Da ProFLOW bereits einen XML Export zur Verfügung stellt, wäre es mittels einer XSL Transformation (siehe Kapitel 2.4) möglich gewesen, das Resultat des XML Exports in die WSL zu transformieren. Ein Problem wie in Abschnitt 2.5.3 wäre allerdings die Folge. Deshalb wurde die andere Möglichkeit gewählt, nämlich die Funktionen zur Erstellung von wohlgeformter XML zu verwenden und somit den WSL Code direkt zu erzeugen.

5. Transformation der WSL in eine Webapplikation

Die WSL ist nach Kapitel 4 in der Lage durch eine grafische Notation dargestellt und erzeugt zu werden. Nun folgt der zweite praktische Teil: Sie soll durch eine Transformation in eine Webanwendung umgeformt werden können. Dies wird beispielhaft in der Programmiersprache Java mit dem Play Framework umgesetzt. Genauso denkbar wäre aber jede andere Sprache, die sich für Webanwendungen eignet.

Zuerst werden die Strukturen und das Konzept von Play erläutert. Im zweiten Teil dieses Kapitels geht es um die Transformationsvorschrift. Abschließend im dritten Abschnitt wird ein XSLT Prozessor vorgestellt, der mit Java realisiert wird.

5.1 Einführung in das Play Framework

Bei Play handelt es sich um ein Open Source Web-Framework, welches in Java geschrieben ist und dem MVC Entwurfsmuster folgt [PDT10a]. Wobei dieses im Vergleich zu klassischen Desktopanwendungen geringfügig abgewandelt wurde [PDT10b]. Mit wenigen Einstellungen ist es möglich eine Datenbank zuzuschalten, deren Datenhaltung das Play Framework aber selbstständig übernimmt.

5.1.1 Das Play MVC Konzept

Das MVC Entwurfsmuster, das bei Play eingesetzt wird, unterscheidet sich von dem, welches bei klassischen Desktopanwendungen Verwendung findet. Der Controller wertet die eintreffenden Anfragen des Clients aus und tritt dabei als Moderator zwischen Model und View auf [PDT10b].

In Abbildung 18 kann man den internen Ablauf sehen, wenn eine Ressource angefordert wird. Ein Client ruft über das Internet eine spezielle URI ab (I). Jede URI in Play hat eine eigene Controllermethode, die dadurch ausgeführt wird. Diese Methode ist dann für das weitere Vorgehen zuständig. In den meisten Fällen kommt es zu einer Interaktion mit dem Model (II). Zum Beispiel werden Variablen in Objekten gesetzt. Ist dies der Fall und die Webanwendung mit einer Datenbank verbunden, wird das Model selbstständig die Daten in der Datenbank aktualisieren (III) und gegebenenfalls Informationen an den Controller zurückliefern (IV).

Der Controller ist dann dafür zuständig eine View zu rendern (V). In diesem Schritt werden alle Daten, die die View zum Erzeugen einer Sicht braucht, an sie übergeben. Schließlich wird die erzeugte View, zum Beispiel HTML Code, an den Client über das Internet zurückgeliefert (VI).

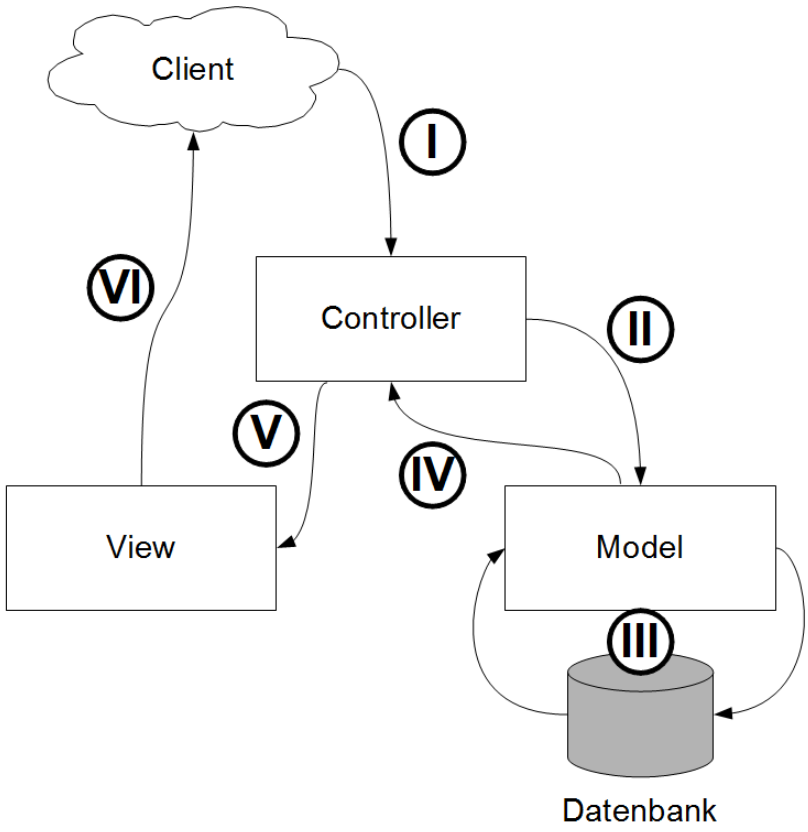


Abbildung 18: Das Play MVC Konzept [PDT10b]

Diese Aufteilung in Model, View und Controller findet sich auch in der Packagestruktur von Play wieder. Beim Anlegen eines neuen Projekts, wird automatisch ein Ordner mit dem Namen „app“ erzeugt, der die Unterordner „models“, „views“ und „controllers“ enthält.

5.2 Definition einer Transformationsvorschrift

Für eine Webanwendung, wie sie in dieser Arbeit behandelt wird, müssen drei Arten von Dateien erzeugt werden: Modell- und Controllerklassen, die jeweils in Java zu verfassen sind, und HTML-Dateien für die Views. Wie man in diesem Kapitel sehen wird, sind die XSLT-Dokumente nicht lesbar eingerückt. Da das Problem des Pretty Printing nicht gelöst werden konnte, wurde entschieden, dass es wichtiger ist, lesbaren Code im generierten Teil zu haben.

Der Übersichtlichkeit halber ist es daher sinnvoll, drei einzelne Transformationsvorschriften zu verfassen, die jeweils einen dieser Bereiche beschreiben. Mit XSLT ist es problemlos möglich, Code in mehrere Dateien aufzuteilen. Diese können dann mit einer `import` Funktion zusammengeführt werden, hier beispielhaft für den Teil, der die Modellklassen erzeugen wird:

```
<xsl:import href="import/model.xsl"/>
<xsl:template match="/">
    <xsl:for-each select="program/models">
          <xsl:apply-templates select="model" />
    </xsl:for-each>
</xsl:template>
```

Aufteilung der Transformationsvorschrift in mehrere Dateien

Seit XSLT 2.0 ist es möglich, mehrere Ergebnisdokumente zu erzeugen [W3C07a]. Möglich macht dies das neu eingeführte `result-document` Element mit dem Attribut `href`. In diesem Attribut gibt man einen Dateipfad an. Bei der Transformation erstellt der XSLT Prozessor dann eine Datei in diesem Pfad und schreibt die Ergebnisse hinein.

Für die Generierung von Modellklassen sähe es zum Beisiel folgendermaßen aus:

```
<xsl:template match="model">
<xsl:result-document href="app/models/{@name}.java">
    <!-- Inhalt der Datei -->
</xsl:result-document>
</xsl:template>
```

Erzeugen von mehreren Ergebnisdokumenten am Beispiel Modellklassen

Das `template`-Element gibt vor, welches Element der WSL betrachtet wird. In diesem Fall entspricht jedes `model`-Element in der WSL einem Durchlauf im XSLT. Das `result-document`-Element sorgt dafür, dass jeder Durchlauf in eine eigene Datei geschrieben wird. Die geschweiften Klammern im Dateipfad kennzeichnen einen Platzhalter. Dieser wird mit der Variable `@name`, die den Namen des aktuellen `model`-Elements enthält, gefüllt.

Im Folgenden wird auf jede der drei XSLT-Instanzen genauer eingegangen. Es wird dabei jeweils der Code betrachtet, der im `result-document` Element steht. Dabei wird nicht auf jedes Detail der Transformationsvorschriften eingegangen. Die vollständigen XSL-Dateien befinden sich auf der CD.

5.2.1 Generierung von Modellklassen

Wie in Kapitel 2.5 erklärt, lassen sich wiederholende Programmteile sehr einfach generieren. Jede Java-Modellklasse im Play Framework beginnt gleich. Erst wird der Klasse ein Paket zugeordnet, dann folgt der Import von Bibliotheken. Danach geht es an den Klassenkopf, der mit Ausnahme des Klassennamens immer

```
public class <classname> extends Model{
```

lautet.

In Abschnitt 2.5.2 wurde erläutert, dass die qualifizierteste Art nach den Techniken der modellgetriebenen Softwareentwicklung vorzugehen ist, jeweils zwei Klassen zu Erzeugen. Eine Klasse für generierten und eine für individuellen Code. Der XSLT Prozessor kann wie folgt dazu angewiesen werden:

```
<xsl:result-document href="app/models/{@name}Gen.java">
package models;
@javax.persistence.Entity
public class <xsl:value-of select="@name" />Gen extends Model { … }
</xsl:result-document>

<xsl:result-document href="app/models/{@name}.java">
package models;
@javax.persistence.Entity
public class <xsl:value-of select="@name" /> extends <xsl:value-of
select="@name" />Gen { }
</xsl:result-document>
```

Erzeugen einer Klasse für generierten und einer Klasse für individuellen Code

Das obere `result-document` Element erzeugt eine generierte Klasse, die im Namen den Zusatz „Gen“ trägt. In ihr werden sämtliche Attribute und Methoden generiert, die aus der WSL extrahiert werden können, angedeutet durch „…“.

Das zweite `result-document` Element generiert eine Klasse, in der später vom Entwickler individueller Code mit der Geschäftslogik geschrieben werden muss. Diese Klasse erbt von der generierten Klasse und damit alle in der WSL festgelegten Eigenschaften und Operationen, kann diese aber bei Bedarf überschreiben.

5.2.2 Generierung von Controllerklassen

Eine Controllerklasse im Play Framework enthält eine oder mehrere Methoden. Jede dieser Methoden stellt eine so genannte *action* dar, die Programmcode ausführt und eine View rendert. Es ist aber auch möglich, dass keine View gerendert wird und stattdessen eine weitere Controllermethode aufgerufen wird. Es ist üblich, den Controllerklassen Namen in der Mehrzahl der zugehörigen Modellklassen zu geben.

Ein Beispiel: Es gibt die Modellklasse „Note“. Eine Controllerklasse sollte dann „Notes“ heißen und könnte die Methoden „update“ und „delete“ enthalten, die für das Ändern bzw. Löschen von Notizen verantwortlich wären.

Auf die WSL bezogen bedeutet das, dass für jede Zustandsgruppe eine eigene Controllerklasse und für jeden Zustand eine eigene Controllermethode generiert werden muss. In XSLT sieht es wie folgt aus:

```
<xsl:result-document href="app/controllers/{@name}Gen.java">
public class <xsl:value-of select="@name" />Gen extends Controller {
<xsl:for-each select="state">
    public static void <xsl:value-of select="@name" /> () {
          render();
    }
</xsl:for-each>
}
</xsl:result-document>
```

Generieren von Controllerklassen und -methoden

Dieser Code wird für jedes `stateGroup`-Element in der WSL durchlaufen und erzeugt dabei eine generierte Klasse mit dem Namen der Zustandsgruppe und dem Zusatz „Gen“. Die `for-each` Schleife durchläuft jeden Zustand der Gruppe und generiert dabei jeweils eine Methode.

Sollte der Zustand von ihm ausgehende Listen von Transitionen enthalten, muss noch gewährleistet werden, dass alle Daten, die die Sicht zum rendern braucht, an sie übergeben werden.

Zusätzlich muss jeweils noch eine Controllerklasse für individuellen Code erzeugt werden, dies funktioniert analog zu den Modellklassen.

5.2.3 Generierung von Views

Views sind im Play Framework keine Javaklassen, sondern zum Beispiel HTML-, XML- oder Textdateien. Deshalb tritt bei der Transformation mit XSLT eine kleine Schwierigkeit auf. Da der Prozessor nicht unterscheiden kann, ob ein eingelesenes Tag zur Transformationsvorschrift oder zur Zielsprache gehört, müssen alle reservierte Sonderzeichen, insbesondere spitze Klammern, speziell codiert werden.

Des Weiteren macht es keinen Sinn zwei Dateien, eine für generierten und eine für individuellen Code, zu erzeugen, da bei HTML keine Objektorientierung vorliegt.

Für jede Controllerklasse im Play Framework gibt es einen eigenen Viewordner und für jede Controllermethode eine HTML-Datei im dazugehörigen Ordner. Die Views werden also erzeugt indem, ähnlich wie bei der Erzeugung der Controllerklassen, über alle *state*-Elemente in der WSL iteriert wird.

Jede Viewdatei muss wiederum, für jede einfache ausgehende Transition des Zustands, einen Hyperlink auf den Zielzustand setzen. Bei Transitionslisten muss eine Liste von Hyperlinks erzeugt werden.

5.3 Programmierung eines XSLT Prozessors in Java

Nun geht es daran die Webanwendung zu generieren. Dafür soll die domänenspezifische Sprache, die in Kapitel 3 definiert wurde, mittels XSLT (siehe Kapitel 2.4) in eine Play-Anwendung transformiert werden. Zuerst wird das in Kapitel 3.2 erstellte XML Schema verwendet, um sicher zu gehen, dass eine gültige DSL vorliegt. Anschließend wird erklärt wie die Transformation durchgeführt wird.

Zu diesem Zweck wird ein XSLT Prozessor in der Programmiersprache Java geschrieben, der diese Aufgaben durchführt. In den folgenden zwei Abschnitten wird erklärt, wie das Programm in Java umgesetzt werden kann. Auf der CD befindet sich der Quellcode des vollständigen Prozessors.

5.3.1 Validierung der WSL mit einem XML Schema

Eine XML-Validierung durchzuführen, ist in Java eine einfache Angelegenheit. Wenn man bereits ein XML Schema geschrieben hat, kann man auf die Unterstützung der `javax.xml` Bibliothek zurückgreifen, um eine Validierung durchzuführen.

Zuerst liest man eine existierende XML-Instanz über eine `DocumentBuilderFactory` und einen `DocumentBuilder` in ein `Document` ein. Auf diesem `Document` wird später die Validierung durchgeführt. Vorher liest man über eine `SchemaFactory` und eine `Schema` Klasse die XSD-Datei ein. Nun kann man einen Validator instanziieren und die XML validieren.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(xmlFile);

SchemaFactory schemaFactory =

    SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Source schemaFile = new StreamSource(xsdFile);
Schema schema = schemaFactory.newSchema(schemaFile);

Validator validator = schema.newValidator();
validator.validate(new DOMSource(doc));
```

XML Validierung in Java

Bereichern sollte man das Programm noch mit einer Textausgabe, die ausgibt, dass die Validierung erfolgreich war oder fehlgeschlagen ist.

5.3.2 Transformationsdurchführung

Im Gegensatz zur Validierung gibt es bei der Transformation einen Haken: Java unterstützt derzeit nur XSLT 1.0. Das vom W3C empfohlene XSLT 2.0 erweitert XSLT 1.0 aber um wichtige Komponenten, wie mehrere Ergebnisdokumente, die für diese Arbeit benötigt werden. Abhilfe schafft eine Open Source Java-Bibliothek mit dem Namen Saxon [Kay10].

Nachdem diese Bibliothek installiert wurde, kann man alle von Java bekannten Transformationsbefehle verwenden, mit dem Unterschied, dass sie XSLT 2.0 ausführen können.

Zuerst müssen die WSL- und die XSLT-Datei (siehe Kapitel 5.2) als `Source` eingelesen werden. Anschließend erzeugt man eine neue `TransformatorFactory` und einen neuen `Transformator`. Dieser kann dann mit der `transform()` Methode die Transformation durchführen und die Ergebnisdokumente im Dateisystem speichern.

```
Source xmlSource = new StreamSource(xmlFile);
Source xsltSource = new StreamSource(xsltFile);

TransformerFactory transFact = TransformerFactory.newInstance();
Transformer trans = transFact.newTransformer(xsltSource);

trans.transform(xmlSource, new StreamResult(System.out));
```

XSL Transformation in Java

Auch hier sollte man das Programm noch mit einer Textausgabe bereichern, die ausgibt, dass die Transformation erfolgreich war oder fehlgeschlagen ist.

6. Beispiel

In diesem Kapitel wird ein Beispiel vorgeführt, an dem deutlich werden soll, wie der Ablauf aussehen kann, wenn man mit der in ProFLOW erstellten Notation, der Web State Language und dem Play Framework eine Webanwendung generieren will. Ziel des Beispiels ist es, das Grundgerüst einer Anwendung zur Notizverwaltung zu erhalten. Notizen sollen dabei angezeigt, hinzugefügt und gelöscht werden können.

Zuerst überlegt man sich, welche Zustände für ein solches Programm benötigt werden. Man wird ziemlich schnell feststellen, dass man mit einer Zustandsgruppe mit dem Namen „Notes“ auskommt. In dieser Gruppe befinden sich vier Zustände. Das sind zum einen zwei normale Zustände „list“ und „newNote“. Der Startzustand zeigt auf „list“, weil es der erste Zustand ist, in den man nach dem Starten der Anwendung gelangen soll. Zum Anderen gibt es noch die zwei schwachen Zustände „deleted“ und „added“.

An Modellklassen braucht man in diesem einfachen Beispiel nur eine zum Verwalten von Notizen, die hier „Note“ genannt wird. In der Property-View wird für sie eine Eigenschaft mit dem Namen „content“ vom Type „String“ festgelegt.

Nun überlegt man sich die Transitionen. Sie sollen die Zustände sinnvoll verbinden. Von „list“ muss es die Möglichkeit geben, neue Notizen hinzuzufügen („add note“) und bereits vorhandene zu löschen („delete note“). Dafür ist eine Transitionsliste am Besten geeignet. Für jede Notiz die existiert, soll also ein „Löschlink“ angelegt werden. Damit das später funktioniert, muss für diese Transition ein Übergabeparameter definiert werden. In der Property-View wird deshalb ein Parameter mit Namen „id“ vom Typ „int“ eingetragen. Als Beschriftung (label) erhält er „content“. Das bewirkt, dass der Notizinhalt später angezeigt wird.

Ist man im Zustand „newNote“ gibt man die Notiz ein und kann auf „add“ klicken, um sie zu speichern. Danach kommt man in den schwachen Zustand „added“, der wieder auf „newNote“ leitet. Dies soll man beliebig oft wiederholen können oder mit „back“ in den Zustand „list“ zurück gelangen. Alle Transitionen, außer „delete note“ (siehe oben), kommen ohne Übergabeparameter aus.

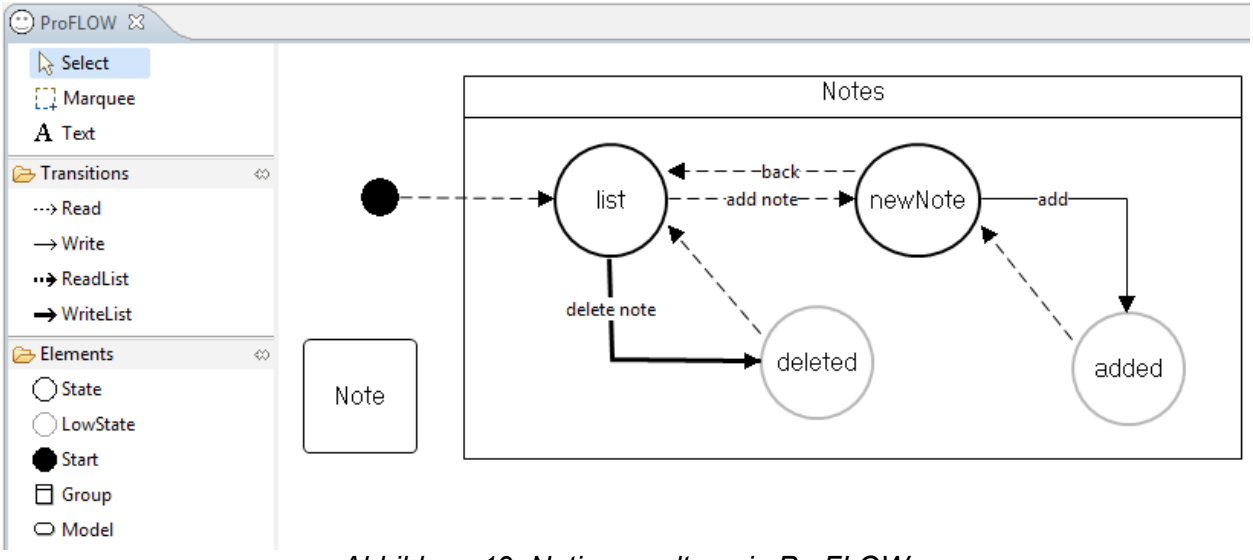


Abbildung 19: Notizverwaltung in ProFLOW

Abbildung 19 zeigt, wie die Notizverwaltung in ProFLOW gezeichnet aussehen kann. Nun kann man sich noch Titel für die normalen Zustände überlegen. Diese werden bei der Transformation in Überschriften umgewandelt. Für „list“ wird „show notes“ und für „newNote“ „add a new note“ gewählt.

Hat man alle Einstellungen wie oben beschrieben getätigt, kann man die WSL exportieren. Dafür klickt man mit der rechten Maustaste auf die Arbeitsfläche von ProFLOW und wählt im erscheinenden Kontextmenü „Export as WSL...“ aus. Ein Fenster erscheint, in dem man den Namen und Dateipfad der zu erzeugenden Datei festlegen kann. Der WSLExporter erzeugt dann folgendes Resultat:

```
<program>
    <models>
        <model name="Note">
            <property name="content" type="String"/>
        </model>
    </models>
    <states>
        <stateGroup name="Notes">
            <state name="newNote" title="add a new note"/>
            <state name="deleted" title="">
                <parameter name="id" type="int"/>
            </state>
            <state name="added" title=""/>
            <state name="list" title="show notes"/>
        </stateGroup>
    </states>
    <transitions>
        <transition action="read"
                    sourceGroup="start" sourceState="start"
                    targetGroup="Notes" targetState="list"/>
        <transition action="read" label="add note"
                    sourceGroup="Notes" sourceState="list"
                    targetGroup="Notes" targetState="newNote"/>
        <transition action="write" by="Note" label="delete note"
                    sourceGroup="Notes" sourceState="list"
                    targetGroup="Notes" targetState="list"
                    viaGroup="Notes" viaState="deleted">
            <inputParameter handling="hidden" label="content"
                        name="id" type="int" value="id"/>
        </transition>
        <transition action="read" label="back"
                    sourceGroup="Notes" sourceState="newNote"
                    targetGroup="Notes" targetState="list"/>
        <transition action="write" label="add"
                    sourceGroup="Notes" sourceState="newNote"
                    targetGroup="Notes" targetState="newNote"
                    viaGroup="Notes" viaState="added"/>
    </transitions>
</program>
```

Ergebnis des WSL Exports

Nun hat man die WSL in einer eigenen Datei. Der nächste Schritt ist es, die WSL in eine Play Anwendung zu transformieren. Dafür wird der in Kapitel 5 beschriebene XSLT Prozessor verwendet. Dieser muss mit zwei Parametern aufgerufen werden: Der erste Parameter ist der Pfad des WSL-Dokuments, der zweite Parameter der Pfad zur XSLT Transformationsvorschrift. Optional als dritter Parameter, kann der Pfad zur XSD-Datei übergeben werden. Wird dieser nicht gesetzt, wird die WSL nicht auf *Gültigkeit* überprüft. Der Aufruf in diesem Beispiel lautet also:

```
java Transform notes.xml transform.xsl wsl.xsd
```

Nach kurzer Bearbeitungszeit erscheint in der Ausgabe

```
XML validation [successful]
XSL transformation [successful]
```

und ein neuer Ordner mit dem Namen „app“ wurde im Verzeichnis des Transformators erzeugt. Er enthält die drei Unterordner „controllers“, „models“ und „views“.

Nun muss ein neues Play Projekt mit dem Namen „Notes“ angelegt werden. Bei den Konfigurationen des Projekts muss eine Aktivierung der Datenbankunterstützung erfolgen. Anschließend, können die generierten Dateien in das Projekt kopiert und das Programm bereits ausgeführt werden.

Die Geschäftslogik fehlt jedoch noch. Es können also noch keine Notizen hinzugefügt oder gelöscht werden. Für das Löschen und Hinzufügen müssen die entsprechenden Controllermethoden überschrieben werden. Das Anzeigen funktioniert bereits durch die Transitionsliste, die jede Notiz mit einem Löschlink auflistet. Um das zu testen müssen vorerst Datensätze für die Modellklasse angelegt werden. In diesem Beispiel werden zwei Notizen angelegt, die als Inhalt (content) „Notiz eins“ und „Notiz zwei“ enthalten. Führt man die Anwendung anschließend aus, zeigt der Browser folgendes Ergebnis an:

Abbildung 20: Ergebnis des Beispiels

Wie man sieht, gelangt man nach dem Starten der Anwendung, in den Zustand „list“. Dort werden alle Transitionen angezeigt. Einmal die einfache Transition „add note“, die zum Hinzufügen von neuen Notizen gedacht ist und die Transitionsliste „delete note“ (dargestellt als Buttons), zum Löschen von Notizen.

Wenn man nun das Löschen implementieren möchte, muss man individuellen Code schreiben. In diesem einfachen Beispiel muss die Controllermethode „deleted“ implementiert werden:

```
public static void deleted(int id){
    Note.findById((long) id).delete();
    Notes.list();
}
```

Damit wird die generierte Methode „deleted“ überschrieben und bekommt individuelle Logik. Nachdem Löschen leitet sie wieder in den Zustand „list“ um.

Die nächsten Schritte wären das Implementieren des Hinzufügens und das Anpassen des Designs. Auf Beides wird an dieser Stelle verzichtet.

7. Zusammenfassung und Ausblick

Dieses Kapitel fasst die Ergebnisse und Erkenntnisse der Bachelorarbeit zusammen und gibt einen Ausblick auf mögliche Erweiterungen der WSL, der zugehörigen grafischen Notation und der Transformation.

7.1 Zusammenfassung

Das Ziel dieser Arbeit war es, eine domänenspezifische Sprache zu entwickeln, die nach den Techniken der modellgetriebenen Softwareentwicklung in eine Webanwendung im Play Framework transformiert werden kann. Zusätzlich sollte das grafische Modellierungswerkzeug ProFLOW so erweitert werden, dass die grafische Notation der Sprache gezeichnet werden kann.

Im 2. Kapitel wurden die Grundlagen erklärt. Zuerst ging es um die Domäne: Anwendungen im World Wide Web und deren Eigenschaften. Da die Sprache in XML verfasst sein sollte, wurde anschließend in XML und XML Schema eingeführt. Danach ging es um die XSL Transformation, mit der man XML Dateien in ein beliebiges anderes XML oder Textformat umwandeln kann. Das Kapitel schloss mit einem Abschnitt über modellgetriebene Softwareentwicklung, denn bei der Generierung der Webanwendung in der Zielsprache sollten die Paradigmen der MDSD angewandt werden.

Das Kapitel 3 beschäftigte sich mit dem Kern dieser Bachelorarbeit. Im ersten Teil des Kapitels ging es um das Erarbeiten einer domänenspezifischen Sprache, die *Web State Language* (WSL) genannt wurde. Es wurden prinzipielle Anforderungen an eine solche Sprache diskutiert. Dabei wurde erkannt, dass es sinnvoll ist, die Sprache in drei Stränge zu gliedern: Modellklassen, Zuständen und Transitionen. Weitere Sprachelemente wurden aus Referenzimplementierungen abgeleitet. Im zweiten Teil des Kapitels ging es darum, die festgesetzten Sprachelemente in XML Schema zu verfassen und somit eine Grammatik für die WSL festzulegen.

Nachdem die WSL feststand, ging es im 4. Kapitel darum, eine grafische Notation zu entwickeln, damit die Sprache gezeichnet werden konnte. Zuerst wurde theoretisch überlegt, welche Anforderungen an eine solche Notation zu stellen sind. Im zweiten Teil wurden diese Überlegungen in ProFLOW umgesetzt. Dabei wurden geringfügige Abstriche in der Funktionalität vorgenommen, weil sich nicht alle in der Theorie erdachten Elemente unkompliziert umsetzen ließen.

Bei Kapitel 5 lag der Fokus auf dem Play Framework und der Transformationsvorschrift. Zuerst wurde erklärt wie Play aufgebaut ist, welche Eigenschaften es hat und wie man Webanwendungen damit programmiert. Anschließend ging es um die XSL-Dateien, die die Transformationsvorschriften für die Umwandlung von WSL in eine Webanwendung beinhalten. Der letzte Teil des Kapitels beschrieb, wie ein XSLT Prozessor in Java aussehen kann und was man dabei beachten muss. Dabei konnte das Problem des Pretty Printing aus zeitlichen Gründen nicht gelöst werden und es wurde entschieden, dass es wichtiger ist, ein lesbar formatiertes Zieldokument zu erhalten.

Um das Erarbeitete praxisnah noch einmal zu verinnerlichen, folgte mit dem 6. Kapitel ein Beispiel anhand einer Notizverwaltung. Darin wurden alle zuvor erläuterten Aspekte der Reihe nach durchgeführt, wie es auch in der Praxis der Fall sein würde. Ergebnis war ein funktionierendes Grundgerüst einer Webanwendung.

7.1.1 Fazit

Die modellgetriebene Softwareentwicklung ist eine sehr interessante Methode, Software nach dem Prinzip einer Fertigungsstraße zu entwickeln. Ist die Domäne festgelegt, eine DSL entworfen und ein Generator erst einmal geschrieben, lassen sich funktionsfähige Grundgerüste von Anwendungen sehr schnell und einfach erzeugen. Verwendet man zusätzlich noch ein grafisches Modellierungswerkzeug, kann der Softwareentwicklungsprozess bereits in der Entwurfsphase oder sogar schon in der Anforderungsphase sinnvoll unterstützt werden.

Doch muss man genau abwägen, ob sich der erhöhte Aufwand lohnt. Möchte man nur eine oder zwei Anwendungen in einer Domäne erstellen, sollte man auf MDSD verzichten und klassische Softwareentwicklungsmethoden bevorzugen. Sobald aber eine Reihe von Anwendungen in der selben Domäne zu erstellen sind, lohnt sich der Einsatz von modellgetriebenen Ansätzen.

Eine domänenspezifische Sprache zu entwerfen, ist anspruchsvoller als man zu Anfang denkt. An vielen Stellen muss man abwägen, ob bestimmte Elemente in die Sprache aufgenommen werden oder ob man zugunsten einer einfachen DSL auf sie verzichtet und sich auf das Wesentliche konzentriert. Abstriche und Erweiterungen muss man während des ganzen Entwicklungsprozesses von DSL und Generator durchführen. Manchmal merkt man erst zu einem späteren Zeitpunkt, dass die anfangs überlegten Sprachkonstrukte sich nicht eignen oder das wichtige Elemente fehlen.

Zusammenfassend lässt sich sagen, dass man die WSL für kleine bis mittelgroße Webapplikationen sehr gut verwenden kann. Die grafische Notation anzuwenden, ist einfach und weitgehend intuitiv. Sind allerdings zu viele Zustände und Transitionen anzulegen, kann das Diagramm sehr unübersichtlich und wenig hilfreich werden. In einem solchen Fall sollte man überlegen, die Anwendung in mehrere Teile aufzugliedern und später per Hand zusammenzufügen.

7.2 Ausblick

Derzeit ist das XML Schema, welches die Grammatik der WSL darstellt, noch nicht auf das Äußerste ausgereizt. Es gibt zum Beispiel die Möglichkeit in XSD Schlüssel zu definieren, anhand derer man festlegen kann, dass als Wert nur ein bereits vorgekommener Wert verwendet werden darf. Dies wäre für die WSL zum Beispiel bei den Transitionen interessant, da dort Modellklassen eingetragen werden, aber aktuell nicht überprüft wird, ob diese überhaupt existieren.

Wie im Beispiel 6 gesehen müssen momentan rudimentäre Funktionen, wie das Löschen und Erstellen von Objekten per Hand implementiert werden. Da diese aber immer sehr ähnlich sind, wäre es denkbar sie ebenfalls mit generieren zu lassen. Dafür müsste die WSL erweitert werden, da sie derzeit keine passenden Sprachmittel zur Verfügung stellt.

Mit der Transformationsvorschrift vom XSLT Prozessor werden aktuell nur der reine Quellcode mit wenigen Kommentaren erzeugt. Langfristig gesehen, ist es sinnvoll die Transformationsvorschrift so zu erweitern, dass aussagekräftige Kommentare mit generiert werden. Sind Kommentare erst einmal in einem erhöhten Umfang enthalten, könnte es sich zusätzlich anbieten eine Dokumentation automatisch mit zu generieren. Javadoc wäre so ein Software-Dokumentationswerkzeug, das aus Java-Quelltexten automatisch Dokumentationsdateien erstellt.

Des Weiteren enthält das Play Framework eine vollständig integrierte Umgebung für automatisierte Tests. Um die Qualität der Software zu steigern, wäre es denkbar Testfälle vom Generator mit erzeugen zu lassen. Dabei muss aber unbedingt bedacht werden, dass man sich

nicht ausschließlich auf die generierten Testfälle verlässt und beschränkt. Sie müssen stets von einem Tester überprüft und gegebenenfalls ergänzt werden.

Mit XML als Basis für die Web State Language, hat man bewusst auf eine variable und einfach transformierbare Grundtechnik gesetzt. Dies könnte man in Zukunft ausnutzen um Webanwendungen auch in anderen Programmiersprachen erzeugen zu können. Auch ist man nicht auf das Modellierungswerkzeug ProFLOW beschränkt, sondern könnte ein eigenes schreiben oder bestehende OpenSource Werkzeuge so anpassen, dass sie die grafische Notation zeichnen und die WSL exportieren können.

Offen bleibt auch die Frage, ob weitere Konzepte, wie zum Beispiel eine Art Sequenzdiagramm, die grafische Notation und die WSL so bereichern könnten, dass noch effektiver Software erstellt werden kann.

Eine Alternative zum Erstellen einer domänenspezifische Sprache mit XML, wäre das in Kapitel 3 diskutierte Xtext. Mit Xtext können sowohl Programmiersprachen als auch domänenspezifische Sprachen entwickeln werden. Das hätte den Vorteil, dass man zu seiner DSL noch einen Codeeditor mit großer Funktionalität, wie Syntaxhervorhebung und Autovervollständigung bekäme. Auch wäre die WSL von den XML-typischen spitzen Klammern befreit und wäre unter Umständen wesentlich leichter lesbar.

Literaturverzeichnis

ECMA09 ECMA: *ECMAScript Language Specification*, 2009,
Link: http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf

GC03 J. Gómez, C. Cachero, *OO-H Method: Extending UML to Model Web Interfaces*,
Universität Alicante, Spanien, 2003

Ger04 V. Geroimenko: *Dictionary of XML Technologies and the Semantic Web*,
Springer, London, 2004

IETF05 IETF: *Uniform Resource Identifier (URI): Generic Syntax*, 2005,
Link: http://tools.ietf.org/html/rfc3986

IETF99 IETF: *Hypertext Transfer Protocol -- HTTP/1.1*, 1999,
Link: http://tools.ietf.org/html/rfc2616

Kay04a M. Kay: *XSLT 2.0 Programmer's Reference Third Edition*,
Wiley, Indianapolis, IN, USA, 2004

Kay04b M. Kay: *XPath 2.0 Programmer's Reference*,
Wiley, 2004, Indianapolis, IN, USA

Kay10 M. Kay: *SAXON The XSLT and XQuery Processor*, 2010,
Link: http://saxon.sourceforge.net/

Mer03 P. Mertens: *XML-Komponenten in der Praxis*,
Springer, Berlin, Heidelberg, 2003

MFV07 N. Moreno, P. Fraternali, A. Vallecillo, *WebML Modeling in UML*,
IET, 2007

MM07 R. Meo, M. Matera, *Designing and Mining Web Applications*,
Idea Group, Italien, 2007

OMG10 OMG: *UML 2.3*, 2010,
Link: http://www.omg.org/spec/UML/2.3/

PDT10a The Play Developers Team: *Play framework overview*, 2010,
Link: http://www.playframework.org/documentation/1.0.3/overview

PDT10b The Play Developers Team: *The main concepts*, 2010,
Link: http://www.playframework.org/documentation/1.0.3/main

RAB09 RSS Advisory Board: *RSS 2.0 Specification*, 2009,
Link: http://www.rssboard.org/rss-specification

RR07 L. Richardson, S. Ruby: *RESTful Web Services*,
O'Reilly, Sebastopol, CA, USA, 2007

SE09 Fachgebiet Software Engineering: *ProFLOW*, 2009,
Link: http://www.se.uni-hannover.de/forschung/flow/proflow/

SV05 T. Stahl, M. Völter: *Modellgetriebene Softwareentwicklung*,
Dpunkt, Heidelberg, 2005

TEF10 The Eclipse Foundation: *Eclipse Modeling Framework Project (EMF)*, 2010,
Link: http://www.eclipse.org/modeling/emf/

Völ08 M. Völter: *Architecture as Language*, 2008,
Veröffentlichung: InfoQ 01/2008 (http://www.infoq.com/)

W3C02 W3C: *XHTML™ 1.0 The Extensible HyperText Markup Language*, 2002,
Link: http://www.w3.org/TR/2002/REC-xhtml1-20020801/

W3C04a W3C: *XML Schema Part 0: Primer Second Edition*, 2004,
Link: http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/

W3C04b W3C: *XML Schema Part 1: Structures Second Edition*, 2004,
Link: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

W3C04c W3C: *XML Schema Part 2: Datatypes Second Edition*, 2004,
Link: http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

W3C07a W3C: *XSL Transformation (XSLT) Version 2.0*, 2007,
Link: http://www.w3.org/TR/2007/REC-xslt20-20070123/

W3C07b W3C: *XML Path Language (XPath) 2.0*, 2007,
Link: http://www.w3.org/TR/2007/REC-xpath20-20070123/

W3C08 W3C: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 2008,
Link: http://www.w3.org/TR/2008/REC-xml-20081126/

W3C98 W3C: *Cascading Style Sheets, Level 2*, 1998,
Link: http://www.edition-w3c.de/TR/1998/REC-CSS2-19980512/

W3C99 W3C: *HTML 4.01 Specification*, 1999,
Link: http://www.w3.org/TR/1999/PR-html40-19990824/

WEG10 Web Engineering Group: *UWE - UML-based web engineering*, 2010,
Link: http://uwe.pst.ifi.lmu.de/

WRG10 WebML Research Group: *The Web Modeling Language*, 2010,
Link: http://www.webml.org/

Xtext10 Xtext-Developers: *Xtext User Guide*, 2010,
Link: http://www.eclipse.org/Xtext/documentation/1_0_0/xtext.html

Abbildungsverzeichnis

Abbildung 1: Visualisierung der Aufgabenstellung 9
Abbildung 2: Visualisierte Darstellung der verfeinerten Aufgabenstellung 9
Abbildung 3: XML dargestellt in Baumstruktur 14
Abbildung 4: Eingliederung eines XSLT Prozessors 17
Abbildung 5: Transformationsergebnis im Webbrowser 18
Abbildung 6: Grundidee modellgetriebener Softwareentwicklung nach [SV05] 20
Abbildung 7: Dreischichtige Implementierung 21
Abbildung 8: Grundgerüst der WSL 24
Abbildung 9: Modellklassen in der WSL 25
Abbildung 10: Zustände in der WSL 25
Abbildung 11: Transitionen in der WSL 26
Abbildung 12: Metamodell der WSL 27
Abbildung 13: UML Klassendiagramm 31
Abbildung 14: Zwei Modellklassen im Klassendiagramm 32
Abbildung 15: Zustandsgruppe mit einem normalen und einem schwachen Zustand 33
Abbildung 16: Eine einfache, schreibende Transition 34
Abbildung 17: ProFLOW 36
Abbildung 18: Das Play MVC Konzept [PDT10b] 40
Abbildung 19: Notizverwaltung in ProFLOW 45
Abbildung 20: Ergebnis des Beispiels 47

Tabellenverzeichnis

Tabelle 1: Die wichtigsten Achsen in XPath 19
Tabelle 2: Mögliche Werte des handling-Attributs 27
Tabelle 3: Die vier Arten von Transitionen 34
Tabelle 4: Übersicht über Transitionen und Elemente 35

Anhang

A. Inhalt der CD

Auf der beiliegenden CD zu dieser Arbeit, findet sich folgende Struktur:

Verzeichnis	Inhalt
/Arbeit	Diese Arbeit im PDF-Format
/Beispiel	Die Ergebnisse des Beispiels (Kapitel 6)
/ProFLOW	Quellcode des Plugins zur grafischen Notation der WSL
/XML Schema	Die XML Schema Definition der WSL
/XSLT	Die XSL Transformationsvorschrift zur Generierung von Play Anwendungen
/XSLT Prozessor	Quellcode des XSLT Prozessors

B. Web State Language

An dieser Stelle befinden sich noch einmal alle Elemente und Attribute der WSL mit Beschreibung und möglichen Werten. Dabei haben folgende Zeichen die Bedeutung:

- * Beliebig viele Elemente
- ? Höchstens einmal
- @ Attribut
- _ Leere Zeichenkette
- u Unterstrichen = Standardwert

Element / Attribut	Beschreibung	Werte
program	Wurzelelement	
models	Liste der Modellklassen	
model *	Eine Modellklasse	
@name	Name der Modellklasse	<ModelName>
property *	Eine Eigenschaft der Klasse	
@type	Typ der Eigenschaft	String/Integer/<ModelName>/...
@name	Name der Eigenschaft	<PropertyName>
@initial ?	Initialwert der Eigenschaft	<String>/<Integer>/...
@association ?	Assoziationstyp der Eigenschaft	1-n/n-1/1-1/n-m
method *	Eine Methode der Klasse	
@return	Rückgabewert der Methode	Void/String/Integer/<ModelName>/...

@name	Name der Methode	<MethodName>
parameter *	Parameter der Methode	
@type	Typ des Parameters	String/Integer/<ModelName>/...
@name	Name des Parameters	<ParameterName>
states	Liste der Zustände	
stateGroup *	Eine Zustandsgruppe	
@name	Name der Zustandsgruppe	<StateGroupName>
state *	Ein Zustand	
@name	Name des Zustands	<StateName>
@title	Titel des Zustands	<StateTitle>
parameter *	Parameter des Zustands	
@type	Typ des Parameters	String/Integer/<ModelName>/...
@name	Name des Parameters	<ParameterName>
transitions	Liste der Transitionen	
transition *	Eine Transition	
@by ?	Referenzierte Modellklasse	_/<ModelName>
@sourceGroup	Gruppe des Ursprungszustands	<StateGroupName>
@sourceState	Name des Ursprungszustands	<StateName>
@viaGroup ?	Gruppe des Zwischenzustands	<StateGroupName>
@viaState ?	Name des Zwischenzustands	<StateName>
@targetGroup	Gruppe des Zielzustands	<StateGroupName>
@targetState	Name des Zielzustands	<StateName>
@label	Linktext	<TransitionLabel>
@action ?	Art der Verlinkung	read/write
inputParameter *	Ein Eingabeparameter	
@type	Typ des Parameters	String/Integer/<ModelName>/...
@name	Name des Parameters	<InputParameterName>
@label ?	Beschriftung des Parameters	_/<InputParameterLabel>
@value ?	Wert des Parameters	_/<String>/<Integer>/...
@handling ?	Art des Eingabeparameters	visible/readonly/masked/hidden

Erklärung der Selbstständigkeit

Ich versichere, die vorliegende Bachelorarbeit mit dem Titel *Eine domänenspezifische Sprache zur Transformation von Anwendungszuständen in Webapplikationen* selbstständig, ohne fremde Hilfe und nur unter Verwendung der von mir aufgeführten Quellen und Hilfsmittel angefertigt zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem Prüfungsamt vorgelegen.

Hannover, 18. August 2010

_____

Timo Hüther