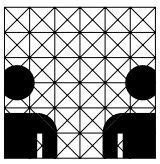
Studienarbeit

Konfiguration von Fachwerten mit XML-Definitionen

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Vogt-Kölln-Straße 30
D-22527 Hamburg



9. Juli 2001

Joachim Sauer

Matr.# 4920432
Grot Sahl 72
D-22559 Hamburg

6sauer@informatik.uni-hamburg.de

Betreuung: Prof. Dr.-Ing. Heinz Züllighoven

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Kontext und Motivation

Bei heutigen modernen Anwendungssystemen werden viele Daten mit der Umgebung ausgetauscht, sei es von System zu System oder durch Ein- und Ausgabe in Masken und Formularen. Dabei spielt die Datenintegrität eine entscheidende Rolle. Durch Kapselung der internen Struktur und wohldefinierte Zugriffsoperationen können Datentypen geschaffen werden, die von außen nur sinnvoll manipuliert werden können und jederzeit einen konsistenten Zustand haben. Wenn die Erzeugungs- und Zugriffsoperationen fachlich und nicht technisch motiviert sind, erhält man Fachwerte – fachliche Datentypen.

Am Arbeitsbereich Softwaretechnik des Fachbereichs Informatik der Universität Hamburg ist das JWAM-Rahmenwerk [JWAM 01] entstanden, ein in der Programmiersprache Java entwickeltes objektorientiertes Rahmenwerk für Anwendungsentwicklung nach dem Werkzeug- und Materialansatz (siehe [Züllighoven 98]). Darin integriert ist das Konzept der Fachwerte, das die obigen Kriterien erfüllt. Ein eigenes Rahmenwerk für Fachwerte wurde nötig, da in gängigen objektorientierten Sprachen (wie auch in Java) neue Datentypen nur als Objekte und nicht als Werte entworfen werden können. Die momentane Implementation hat jedoch einige Nachteile, die ich analysieren werde und in einem neuen Konzept vermeiden möchte.

1.2 Zielsetzung

In dieser Arbeit möchte ich untersuchen, wie sich neue Fachwert-Klassen einfacher als bisher erzeugen lassen. Den Schlüssel dazu stellt die neue Technologie XML Schema dar, mit der sich Restriktionen für XML-Dokumente festlegen lassen. Diese Restriktionen können auch die im Dokument verwendeten Datentypen einschränken. Meine Aufgabe ist nun, zu untersuchen, inwieweit sich diese Datentyprestriktionen für die Validierung und Konfiguration von Fachwert-Typen nutzen lassen.

Hauptziel soll sein, das Schreiben von Fachwert-Klassen für Anwendungsentwickler zu vereinfachen, was zum einen die dafür benötigte Zeit und zum anderen die Komplexität des dahinterstehenden Konzepts anbelangt.

1.3 Inhalt der Arbeit

Im ersten Abschnitt werde ich das Fachwertkonzept erläutern. In einer Einführung werden die generellen Unterschiede zwischen Werten und Objekten und vorgegebenen und benutzerdefinierten Werten in objektorientierten Programmiersprachen thematisiert, der Begriff Fachwert einführt und grundlegende Anforderungen an diese aufführt. Anschließend werde ich verschiedene Realisierungsalternativen diskutieren, das momentan im Rahmenwerk

JWAM umgesetzte Konzept vorstellen und Probleme dieser Implementation herausarbeiten.

Thema des darauffolgenden Abschnitts ist XML, die Extensible Markup Language. Zuerst werde ich die Grundkonzepte und wichtige zugehörige Standards erörtern. Danach beschäftige ich mich intensiv mit XML Schema, einer aktuellen Entwicklung zur Einschränkung und Validierung von XML-Dokumenten, der eine herausragende Bedeutung in dieser Arbeit zukommt. Da ich als Grundlage das JWAM-Rahmenwerk für die Programmiersprache Java benutze, untersuche ich dann die Bearbeitungsmöglichkeiten von XML in Java, wobei ich verschiedene wichtige Standards und Bibliotheken vorstellen werde.

Nach dieser Klärung des Handwerkszeuges werde ich im nächsten Abschnitt ein Konzept zur Konfiguration von Fachwerten aus XML-Beschreibungen vorstellen, Alternativen und kritische Punkte erörtern.

Dieses Konzept setze ich im folgenden Abschnitt prototypisch um. Anhand von Entwurfsmustern und Schnittstellenbeschreibungen erläutere ich von mir getroffene Entwurfsentscheidungen. Beispiele zeigen die Verwendungsmöglichkeiten. Es folgt ein Schlußwort mit einer Zusammenfassung, einer Bewertung und einem Ausblick auf Verbesserungsmöglichkeiten und voraussichtliche zukünftige Entwicklungen.

Im Anhang sind die vollständigen öffentlichen Schnittstellen der neu erstellten Klassen zu finden, im Glossar technische Abkürzungen aufgeführt. Die Arbeit wird abgeschlossen durch ein Literaturverzeichnis.

1.4 Danksagung

Ich danke Herrn Professor Heinz Züllighoven für seine Betreuung dieser Studienarbeit und für seine inhaltlich sehr guten und innovativen Vorlesungen, Projekte und Seminare, die die Grundlage dieser Arbeit bilden. Henning Wolf danke ich für seine intensive Betreuung und viele Anregungen und Hilfen bei der Konzeption dieser Arbeit und Entwurf und Codierung des Prototypen.

2 Fachwerte

2.1 Konzept

Fachwerte sind aus der Notwendigkeit entstanden, einen wertorientierten, referentiell transparenten Programmierstil auch in objektorientierten Sprachen für Subsysteme zu realisieren. Fachliche Werte sollen sich wie Werte verhalten, also, wie unten ausgeführt wird, z.B. unveränderlich sein. Diese Freiheit von Seiteneffekten macht die Programmierung und den Umgang mit ihnen sehr viel angenehmer.

Ich werde zunächst untersuchen, wie sich Werte von Objekten unterscheiden und welche Möglichkeiten Programmierer in gängigen objektorientierten Sprachen besitzen, eigene Werte zu definieren. Daraus ergeben sich Anforderungen an Fachwerte, die ich anschließend darstellen werde. Verschiedene Realisierungsalternativen können in Betracht gezogen werden. Die konkrete Umsetzung im JWAM-Rahmenwerk werde ich genauer betrachten und einige ihrer Probleme herausarbeiten.

2.1.1 Werte und Objekte

In [Züllighoven 98, S. 57] wird ausgeführt, welche Unterschiede zwischen Werten und Objekten bestehen:

Wert	**Objekt**
zeit- und ortlos	existent in Raum und Zeit
abstrakt, ohne Identität, nur Gleichheit	konkret, mit Identität und Gleichheit
definiert oder undefiniert, aber nicht veränderlich	veränderbar bei Wahrung der Identität
kann nicht gemeinsam benutzt werden	kann über Verweise gemeinsam genutzt werden

Objekte werden erzeugt und existieren dann bis zu ihrer Löschung im Speicher, während Werte die Konzepte Ort und Zeit nicht kennen, keinen Zustand und keinen Anfang und Ende besitzen. Daher haben Werte auch keine Identität, können nur auf Gleichheit untersucht werden. Es ist sinnlos, von meiner „42“ und seiner „42“ zu reden, entscheidend ist die Feststellung, daß beide Werte gleich sind. Da Werte abstrakt sind, können sie auf verschiedene Art und Weise dargestellt werden. Beispielsweise bezeichnen „42“, „Zweiundvierzig“ und „XLII“ denselben Wert.

Werte können nicht verändert werden. Dies widerspricht vielleicht der allgemeinen Vorstellung des Verfahrens bei arithmetischen Operationen, beispielsweise bei Addition von 23 zu 69: $69 + 23 = 92$. Dabei stellen jedoch 69, 23 und 92 jeweils eigene Werte dar. Dies unterscheidet sich vom Variablenkonzept in imperativen Programmiersprachen, bei dem mit Behältern,

realisiert durch Speicherstellen, gearbeitet wird, die ihren Inhalt ändern können.

Dagegen können sich Objekte sehr wohl ändern, indem sich ihr interner Zustand ändert. Man denke z.B. an ein Kontoobjekt, bei dem sich durch Aufruf der Einzahlungsoperation der Kontostand erhöht oder an ein Autoobjekt, das nach Aufruf der Fahrenoperation einen höheren Kilometerstand aufweist. Die Identität des Objektes ändert sich dadurch nicht.

Mehrere Benutzer können gleichzeitig an einem Objekt arbeiten, indem sie jeweils eine Referenz darauf halten, über die sie auf das Objekt zugreifen können. Die gleichzeitige Benutzung von Objekten ist für die meisten Systeme wesentlich, kann aber zu Problemen führen, falls Änderungen am Objekt durch einen Benutzer den anderen verborgen bleiben (das sogenannte *Alias-Problem*). Werte können nicht zusammen benutzt werden, jede Nennung desselben Wertes ist unabhängig von den anderen.

Ein Umgang mit Werten in Programmiersprachen begründet einen **wertorientierten Programmierstil**, bei dem Ausdrücke *referentiell transparent* ausgewertet werden. Dies heißt, daß die Bedeutung eines Ausdrucks sein Wert ist und seine Auswertung keine Nebeneffekte verursacht, daß jeder Ausdruck durch seine Teilausdrücke vollständig definiert ist und daß der Austausch von wertgleichen Teilausdrücken den Wert des gesamten Ausdrucks nicht ändert, analog zu mathematischen Umformungen.

Objekte bilden – natürlicherweise – die Grundlage von objektorientierten Sprachen. Wertorientierte Subsysteme können die oben erläuterten Unterschiede von Werten vorteilhaft für eine sauberere, konzeptionell einfachere Programmierung nutzen.

2.1.2 Benutzerdefinierte Werte

Die in den gängigen objektorientierten Sprachen vorhandenen Werttypen reichen nicht aus, um die aus Anwendungsbereichen motivierten Werte sauber umzusetzen. Werttypen existieren hier meist als primitive Datentypen, die direkt auf Speicherformate der zugrundeliegenden Rechnerarchitektur (wie z.B. bei C++) oder einer virtuellen Maschine (wie z.B. bei Java) abgebildet werden. Es finden sich vorzeichenbehaftete und -lose Ganzzahlen mit verschiedenen Wertebereichen, Gleitkommazahlen in mehreren Genauigkeiten, boolesche Werte, Zeichen und Zeichenketten in bestimmten Zeichensätzen und einige speziellere mehr.

Leider lassen sich keine neuen Werte definieren, wie z.B. in rein funktionalen Sprachen wie Miranda. Dies hat dazu geführt, daß vielfach auf die vorhandenen primitiven Werte zurückgegriffen wurde und durch Konventionen versucht wurde sicherzustellen, daß sie immer einen sinnvollen Wert besitzen. So kann beispielsweise der Kilometerstand eines Autos in der Programmiersprache Java als vorzeichenbehafteter ganzzahliger Integerwert modelliert

werden (Java stellt nur vorzeichenbehaftete Ganzzahlwerte zur Verfügung). Alle Zuweisungen an die Kilometerstand-Variable müssen darauf achten, daß deren Wert nicht negativ wird.

Das ist eine potentielle Fehlerquelle und erschwert die Programmierung. Außerdem besitzen solche vordefinierten Werttypen nur generische Operationen, die nicht fachlich motiviert sind. So kann sich der Kilometerstand durch Fahren oder Rücksetzen verändern, was bei Implementation durch eine Ganzzahl nur durch Addition einer positiven Zahl bzw. Zuweisung von 0 umgesetzt werden kann.

All dies führte zur Entwicklung von Fachwerten. Hier die Definition nach [Züllighoven 98, S. 62]:

Definition Fachwert:

Ein Fachwert ist ein benutzerdefinierter Wert. Er repräsentiert Werte im Anwendungsbereich.

Ein Fachwert[a] ist ein Werttyp mit definierter Wertemenge und festgelegten Operationen. Seine innere Repräsentation ist verborgen.

Fachwerte werden in objektorientierten Sprachen als Klassen definiert. Wichtig ist, daß Exemplare eines Fachwerts immer Wertsemantik besitzen, d.h. daß ein einmal gesetzter Wert nicht mehr verändert werden kann.

[a] Hier sollte meiner Ansicht nach eher vom Fachwert-*Typ* gesprochen werden. Ich werde die Unterscheidung zwischen Fachwert-Typ und Fachwert als einem seiner konkreten Exemplare in dieser Arbeit berücksichtigen.

In [Bleek et al. 99a] werden weitere Vorteile der Verwendung von Fachwerten genannt: Werttypen sind sinnvoll beim Austausch in verteilten Systemen, weil Objektreferenzen nicht über Systemgrenzen verfolgt werden müssen, wenn Fachwerte lokal gehalten werden. Performanceverringernde Sperrmechanismen werden vermieden, da Fachwerte prinzipiell unveränderbar sind. Außerdem ist beim Ablegen in Datenbanken für Fachwerte keine Prüfung auf zyklische Objektreferenzen nötig.

2.1.3 Anforderungen an Fachwerte

Welche generellen Anforderungen müssen nun an Fachwerte gestellt werden? [Müller 99] hat die wichtigsten herausgearbeitet, hier von mir zusammengefaßt:

Wertebereich: Fachwert-Typen sollten einen klar definierten und von außen abfragbaren Wertebereich besitzen. Auch undefinierte Fachwerte

können sinnvoll sein, z.B. als Defaultwert für noch nicht initialisierte Variablen.

Konstruktoren: Geeignete Konstruktoren erzeugen neue Fachwert-Objekte aus wohlgeformten, gültigen Repräsentationen. Die Gültigkeit muß dabei vorher prüfbar sein.

Definition als Klassen: Fachwerte können in objektorientierten Sprachen nur als Klassen definiert werden, die aber Wertsemantik besitzen müssen. Sie müssen unabhängig vom Kontext und unveränderbar sein.

Zugriffsoperationen: Über geeignete Operationen soll auf die Repräsentation des Fachwerts und ggf. seiner Teilkomponenten zugegriffen werden können. Erstrebenswert ist neben einer an der internen Repräsentation orientierten Abbildung auch eine als Zeichenkette, die beispielsweise bei der Darstellung in generischen Ein-/Ausgabemasken verwendet werden kann.

Fachliche Operationen: Es sollen sinnvolle Operationen zum Umgang mit dem Fachwert definiert werden können. Vor- und Nachbedingungen regeln den Zugriff. Vorbedingungen können über sondierende Operationen geprüft werden. Fachliche Operationen sollen sich wie Funktionen verhalten, d.h. keine Nebeneffekte besitzen. Operationen, die für alle Fachwerte sinnvoll sind, sind Gleichheitstest und (bei Fachwerten mit Ordnungsrelation) Vergleichsoperationen wie größer und kleiner.

2.2 Realisierungsalternativen

Bei der Umsetzung des oben erläuterten Konzeptes in objektorientierten Programmiersprachen ergeben sich zwei Hauptschwierigkeiten:

1. Prinzipiell sind Objekte veränderlich. Das bricht mit der Wertsemantik.
2. Wird jedesmal ein neuer Fachwert erzeugt, kann dies zu Speicherplatzproblemen führen.

Glücklicherweise lassen sich beide durch Anwendung geeigneter Entwurfsmuster in den Griff bekommen.

2.2.1 ad 1: Problem Wertsemantik

Fachwert-Objekte sollen sich nach Wertsemantik verhalten, wie oben ausführlich diskutiert. Hierzu gibt es zwei Ansätze: unveränderliche und veränderliche Fachwert-Objekte.

Bei den **unveränderlichen Fachwert-Objekten** werden Veränderungen verhindert, indem nur sondierende und keine verändernden Operationen

zugelassen werden. Diese Lösung ist in allen Programmiersprachen umsetzbar. Probleme liegen im Speicherplatzverbrauch und in der eher umständlichen Handhabung, die bei Veränderungen an Fachwerten jedesmal einen neuen Bezeichner erfordert.

Das Problem des bei naiver Implementation hohen Speicherplatzverbrauches läßt sich effizient lösen (s.u.), die umständliche Handhabung ist unvermeidbar.

Die Handhabung bei **veränderlichen Fachwert-Objekten** ist angenehmer, da die Fachwert-Objekte über alle sinnvollen verändernden Operationen verfügen. Zur Vermeidung von daraus entstehenden Problemen gibt es hauptsächlich zwei Möglichkeiten: Der Benutzer von veränderlichen Fachwert-Objekten muß sich an die Programmierkonvention halten, vor Aufruf einer verändernden Operation eine Kopie des Fachwert-Objektes zu erstellen (*„copy-on-write“*). Dies ist fehleranfällig, da die Einhaltung dieser Konvention nicht überprüft werden kann.

Die andere Möglichkeit besteht deshalb darin, das Kopieren der Fachwert-Objekte den Fachwert-Klassen zu überlassen. Es bietet sich der Einsatz des *Body/Handle-Musters* an, auf das ich hier nicht weiter eingehen will. Eine Erläuterung dieses Musters findet sich bei [Bäumer et al. 98, S. 20].

2.2.2 ad 2: Problem Speicherplatzverbrauch

Bei unveränderlichen Fachwerten, bei dem für jeden Fachwert ein neues Fachwert-Objekt erzeugt wird, wird sehr viel Speicherplatz benötigt. Dieses Problem läßt sich lösen, indem unter Einsatz von Entwurfsmustern sichergestellt wird, daß nur benötigte Werte angelegt und vorgehalten werden und diese gemeinsam genutzt werden können.

Beim Entwurfsmuster **„Abstrakte Fabrik“** (siehe [Gamma et al. 95, S. 87]) ist vorgesehen, daß Objekte, hier konkrete Fachwert-Objekte, nur an einer Stelle, nämlich der zugehörigen Fabrik, erzeugt werden können. Die Fachwertfabrik stellt ebenfalls sicher, daß nicht mehr benötigte, d.h. anderswo referenzierte, Fachwert-Objekte entfernt werden.

Das andere hier anwendbare Entwurfsmuster ist das der **„Fliegengewichte“** (siehe [Gamma et al. 95, S. 195]), das erlaubt, dieselben Fachwerte an mehreren Stellen gemeinsam zu nutzen. Dabei enthält das Fachwert-Objekt den *intrinsischen Zustand*, der für alle Kontexte gleich ist, also z.B. den Wert, den Typ, die erzeugende Fabrik u.ä., während kontextabhängige Informationen, der *extrinsische Zustand*, im benutzenden Objekt gespeichert ist. Die Anwendung dieses Musters sorgt besonders bei Fachwert-Typen, die meist gehäuft vorkommen, wie z.B. Wochentagen, für eine große Speicherplatzeinsparung. Falls eine Mehrfachverwendung von Fachwerten aber eher unwahrscheinlich ist, wie z.B. bei Geldbeträgen mit DM und Pfennigen, ist der Nutzen deutlich geringer.

Beide Entwurfsmuster werden kombiniert, indem die Fabrik nur für den ersten neuen Fachwert ein eigenes Objekt erzeugt und danach Verweise auf dieses herausgibt, falls derselbe Wert nocheinmal verlangt wird. So ist sichergestellt, daß ein Fachwert-Objekt für denselben Wert nicht mehrmals instantiiert wird.

2.3 Umsetzung in JWAM

Fachwerte sind in Java als Fliegengewichte realisiert. Fachwert-Objekte werden von Fabriken erzeugt, die als Singletons implementiert sind (d.h. es existiert höchstens ein Exemplar pro Fachwert-Typ, auf das global zugegriffen werden kann, siehe [Gamma et al. 95, S. 127]). Fachwert-Typ und zugehörige Fachwertfabrik werden durch jeweils eine Klasse realisiert, wobei die Fachwertfabrik-Klasse eine innere Klasse der Fachwert-Klasse ist.

Erzeugte Fachwert-Objekte werden in einem Fachwertuniversum gehalten (engl. DomainValueUniverse), das persistent gemacht werden kann.

Die Basis-Funktionalität von Fachwerten wird im Interface `DomainValue`[1] definiert. Dazu existiert eine Standardimplementation in der abstrakten Klasse `DomainValueImpl`. Darauf aufbauend enthält das Rahmenwerk einige häufig vorkommende Fachwert-Typen und beispielhafte Implementationen komplexerer Fachwert-Typen. In Anwendungsprogrammen können weitere spezielle Fachwert-Typen definiert werden. Siehe Klassendiagramm, Abbildung 1.

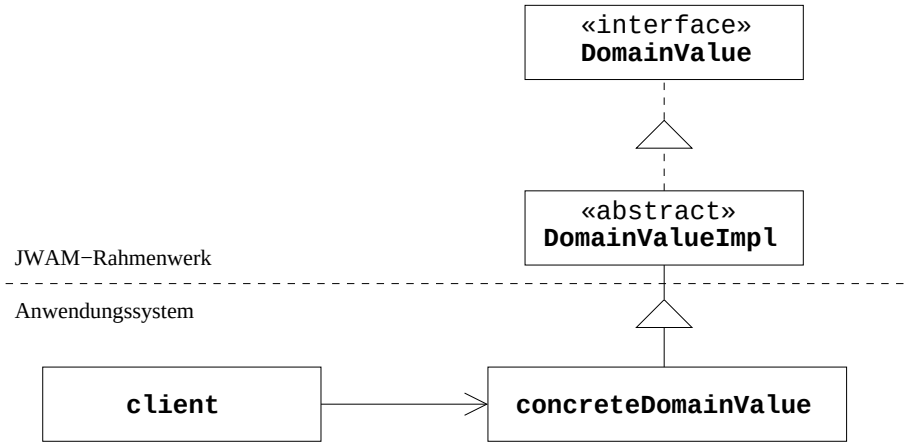


Abbildung 1: Klassendiagramm des bestehenden Fachwertrahmenwerks, in Anlehnung an [Müller 99, S. 37]

[1] Wie im gesamten JWAM-Rahmenwerk sind Schnittstellen-, Klassen-, Methoden- und Parameternamen sowie Kommentare in Englisch gehalten. Damit einher geht eine unschöne Vermischung von Deutsch und Englisch, die sich leider auch in dieser Arbeit nicht vermeiden läßt.

Das Fachwertrahmenwerk hat sich im Laufe der Entwicklung von JWAM verändert, einiges wurde angepaßt, neue Methoden wurden integriert. Im folgenden beschreibe ich den Stand von JWAM 1.6.0, der momentan aktuellen Version:

Die **DomainValue**-Schnittstelle besitzt Methoden, um eine Zeichenkettenrepräsentation des Fachwerts abzufragen (`String toString()`), um einen Text mit den Daten des Fachwerts zu erhalten (`String[] toStrings()`), einen Fachwert auf Gültigkeit zu prüfen (`boolean isDefined()`), bei Ungültigkeit die ungültige Zeichenkette, die zur Erzeugung verwendet wurde, abzufragen (`String invalidCreationString()`) und die zugehörige Fabrik zu erhalten (`DomainValue.Factory factory()`). Alle Operationen sind – wie das gesamte Rahmenwerk – nach dem Vertragsmodell (siehe [Meyer 97], „Design by Contract“) mit Vor- und Nachbedingungen versehen, die ein Benutzungsprotokoll vorgeben.

In der **DomainValue.Factory**-Schnittstelle gibt es Operationen, um abzufragen, ob ein Fachwert aus einer Zeichenkette erzeugt werden kann (`boolean canCreateFromString()`), ob das auch aus einer ungültigen Zeichenkette möglich ist (`boolean canCreateFromInvalidString()`) und ob ein ungültiger Wert vorhanden ist (`boolean hasUndefinedValue()`). Dieser kann genauso erzeugt werden (`DomainValue undefinedValue()`) wie ein Standardwert (`DomainValue defaultValue()`) und natürlich ein gültiger Wert (`DomainValue value(String s)`). Mit `boolean isValid(String s)` kann man abfragen, ob eine Zeichenkette eine gültige Repräsentation darstellt und sich mit `String invalidityExplanation(String s)` im negativen Fall die Erklärung dafür geben lassen.

In der Klasse **DomainValueImpl** werden diese Methoden sowie zusätzlich sinnvolle weitere implementiert. In der Fabrikklasse **DomainValueImpl.Factory** gibt es für einige Methoden Standardimplementationen, die durch protected Einschub-Operationen in Unterklassen angepaßt werden können. Oft ist dies der Fall, wenn ungültige und gültige Fachwerte unterschiedlich behandelt werden sollen.

Desweiteren gibt es Methoden, die das DomainValueUniverse, in dem Fachwerte gespeichert werden, setzen und manipulieren sowie die Möglichkeit, sich mit `Class type()` den Fachwert-Typ, den die Fabrik produzieren kann, zurückgeben zu lassen.

In **konkreten Fachwert-Klassen**, die als Unterklassen von DomainValueImpl realisiert werden, kommen hauptsächlich fachliche Methoden hinzu sowie Variablen, die die interne Repräsentation des Fachwerts speichern. In den Fabriken wird die Methode `DomainValue.Factory instance()` implementiert, die eine Referenz auf das Singleton-Objekt der Fabrik zurückgibt. Dazu Konstruktoren, realisiert als Methoden, die primitive Typen oder

andere Fachwert-Typen als Parameter haben, daraus Fachwerte dieser Klassen machen und vorher die Gültigkeit der Parameter prüfen.

2.4 Probleme des bestehenden Konzeptes in JWAM

Ich habe das momentan implementierte Fachwertmodell analysiert. Dazu habe ich einige Fachwert-Typen selber implementiert, auf JWAM basierende Anwendungssysteme betrachtet und mit erfahrenen JWAM-Entwicklern diskutiert.

Folgende Probleme und Unzulänglichkeiten habe ich festgestellt:

Unnötig viel Aufwand

Die Konstruktion eines Fachwert-Typs ist recht umständlich und erfordert sehr viel neuerstellten Code. Der Entwickler ist mit Dingen belastet, die nicht wesentlich für den neuen Fachwert-Typ sind. Daraus folgt, daß in der Praxis Code für Fachwert-Typen meist aus dem Quelltext eines anderen Fachwert-Typs mittels copy & paste übernommen und anschließend abgeändert wird. Dies führt zu typischen copy & paste-Fehlern durch Übersehen einiger Stellen, an denen Änderungen notwendig wären. Während einige dieser Fehler (wie z.B. falsche Bezeichner) zur Übersetzungszeit erkannt werden können, bleiben andere in der Regel unentdeckt, wie z.B. fehlerhafte Algorithmen und veraltete Kommentare.

Kapselung der Umwandlung von Zeichenketten in Fachwerte

Fachwerte werden über Konstruktoren aus primitiven Typen erzeugt. In sehr vielen Fällen wird es sich dabei um Zeichenketten handeln, insbesondere wenn die Eingabe in generischen Eingabemasken für Fachwerte oder webbasiert über HTML-Formulare geschieht, die nur Zeichenketten zulassen. Die Erfahrung hat gezeigt, daß ein sehr hoher Anteil aller Fachwerte (etwa 95%) auf einer String-Repräsentation beruht.

Wie oben ausgeführt, erfolgt die Erzeugung mit den beiden Methoden `isValid()` und `value()`. Diese Methoden können nun algorithmisch anspruchsvollen Code beinhalten. In Umgebungen, in denen Java nicht zur Verfügung steht, können sie nicht genutzt werden.

Fachwerte können daher beispielsweise nicht im Web bei der Eingabe beim Client geprüft werden, falls nur HTML und Javascript als Techniken zur Verfügung stehen. Eine farbliche Anzeige der Korrektheit während des Tippens oder eine vollständige Korrektheitsprüfung vor Absenden des Formulars ist dann nicht realisierbar. Das führt dazu, daß die Eingaben im Serverprogramm nicht einfach übernommen werden können, sondern noch einmal überprüft und eventuell zurückgewiesen werden müssen, was zu vermeiden ist.

Ein Fachwert pro Fachwert-Klasse

Typischerweise werden in einem Projekt viele Fachwert-Typen benutzt. Bei der momentanen Realisierung muß für jeden Fachwert-Typ eine eigene Klasse erstellt werden. Dies führt zu vielen Fachwert-Klassen, die nur getrennt gewartet werden können.

Einige Fachwert-Typen sind sich ähnlich oder ergeben sich durch Einschränkung oder Kombination von anderen Fachwert-Typen, z.B. Beträge in DM, in Euro und in Dollar. Hier wäre es sinnvoll, diese in einer Klasse zusammenzufassen. Dies ist in JWAM nicht einfach möglich, da bisher implizit davon ausgegangen wurde, daß ein spezieller Fachwert-Typ durch die dazugehörige Klasse eindeutig identifizierbar ist.

Umständliche Abbildung auf Datenbanken

Wenn Objekte in herkömmlichen, weit verbreiteten Relationalen Datenbanken gespeichert werden, werden sie dort entweder als Binary Large Object (BLOB) abgelegt, also im Ganzen serialisiert ohne Berücksichtigung ihrer Struktur, oder auf Tabellen und Felder abgebildet. Dieses ist im ersten Fall ineffizient und verhindert ein Suchen nach einzelnen Objektmerkmalen, im zweiten Fall aufwendig zu programmieren, fehlerträchtig und schwierig zu ändern. Da Fachwerte als Objekte realisiert werden, sind sie davon betroffen.

Falls nun Fachwerte in einem Format definiert werden würden, das sich direkt in den Datenbanken sichern ließe, könnten diese Probleme vermieden werden. Hier bieten sich Zeichenketten oder besser XML-basierte Daten an. In [Awizen 00] wird gezeigt, wie eine solche Abbildung und Speicherung von XML-Dokumenten aussehen kann. Eine weitere Verbreitung von XML-basierten Datenbanken, an denen momentan intensiv geforscht wird, würde die Vorteile noch hervorheben.

Kein Standard für Fachwerte

Fachwerte bieten sich für viele Projekte an. Es existiert jedoch kein rahmenwerksübergreifendes Standardformat für sie. Fachwert-Typen, die beispielsweise für JWAM geschrieben wurden, lassen sich nicht einfach in das JValue-Rahmenwerk[2] integrieren. Gäbe es hier einen Standard, könnten Fachwert-Typen in einem unabhängigen Format geschrieben werden, das einen Import in verschiedene Rahmenwerke erlaubt und Portierungen erleichtert.

Die oben aufgeführten Punkte verhindern eine effiziente Programmierung und Verwendung von Fachwerten. Ich werde analysieren, wie sie vereinfacht

[2]Das JValue-Rahmenwerk ist ein Open-Source Projekt unter der Leitung von Dirk Riehle, das zum JWAM-DomainValue Rahmenwerk vergleichbare Funktionalität bieten soll, momentan aber noch unvollständig und schlecht dokumentiert ist. Siehe [JValue 01].

werden kann. Dies geschieht auf Basis von XML, das ich deshalb im folgenden zunächst vorstelle, bevor ich verschiedene Realisierungsalternativen betrachte.

3 XML

3.1 Grundlagen von XML

Die Extensible Markup Language, kurz XML, ist ein Standard für die Erstellung und Verarbeitung von Dokumenten, der aus der SGML, der Standard Generalized Markup Language, abgeleitet und für das World Wide Web vereinfacht worden ist. Er beruht auf Vorschlägen des World Wide Web Consortiums W^{3}C. Dieses ist ein internationales Industriekonsortium, das im Oktober 1994 von Tim Berners-Lee gegründet wurde und momentan mehr als 500 Mitglieder umfaßt. Es möchte die technische Entwicklung des WWW anführen, Interoperabilität sicherstellen und ein Offenes Forum für Diskussionen bieten (vgl. die Eigendarstellung unter [W3C 00]).

Die Kernspezifikation von XML 1.0 ist als W^{3}C Recommendation[3] in [W3XML 00] zu finden. Aktuell ist die zweite Ausgabe vom 6. Oktober 2000, die Fehlerbeseitigungen und Klarstellungen der ersten Ausgabe vom 10. Februar 1998 umfaßt.

XML ist eine textbasierte Metasprache zur Definition eigener Dokumentenstrukturen und ihr entsprechenden Dokumenten. Im Gegensatz beispielsweise zu HTML existieren keine vordefinierten Markup-Tags. Der Entwickler hat vollständige Kontrolle darüber, wie die Struktur aussieht und welche Elemente, Attribute u.ä. er definieren kann. Um XML-Dokumente zwischen verschiedenen Partnern austauschen zu können, sollten sie einer gemeinsamen Form unterliegen, die festgelegt werden muß. In einem XML-Dokument ist nur die logische Struktur beschrieben, Darstellungsanweisungen sind nicht enthalten. Falls eine Darstellung in anderer als einer automatisch erzeugbaren, generischen hierarchischen Form gewünscht ist, müssen die Formatierungsanweisungen zusätzlich vorliegen.

Man sieht, daß für eine sinnvolle Anwendung von XML mehrere Teile zusammenspielen müssen, die verschiedenen Standards gehorchen. Die wichtigsten sind im einzelnen:

Das XML-Dokument: Hierin sind die Dokumentdaten enthalten, untergliedert in hierarchisch strukturierte Markup-Elemente, die Zeichendaten enthalten können. Diese machen den eigentlichen Inhalt des Dokuments aus. Außerdem vorhanden sein können bzw. müssen ein Prolog, der die verwendete XML-Version und Zeichensatzcodierung angibt, Anweisungen an verarbeitende Programme (sogenannte processing instructions), Kommentare u.a.

[3] Der Dokumenterstellungsprozeß des W^{3}C folgt einer festgelegten Struktur vom ersten Working Draft bis zur Recommendation. Diese stellt das Endresultat dar, über das innerhalb und außerhalb des W^{3}C Einigkeit bestehen soll und das zur weiten Verbreitung vorgesehen ist. Siehe die detaillierte Beschreibung aller Dokumenttypen unter `http://www.w3.org/Consortium/Process-20010208/tr`.

Wenn sich ein XML-Dokument an bestimmte Regeln hält, z.B. die richtige Schachtelung von Elementen einhält und Attributwerte immer in Anführungszeichen faßt, ist es *wohlgeformt*. Dies ist Grundvoraussetzung für eine maschinelle Verarbeitung. Formal definiert sind die Regeln als Produktionsregeln in einer einfachen Erweiterten Backus-Naur-Form (EBNF) in der Spezifikation.

Elemente in XML-Dokumenten beginnen mit einem Start-Tag, das aus „<“, dem Namen und „>“ besteht und enden mit dem Ende-Tag „</“ *Name* „>“. Zwischen Start- und Ende-Tag kann der eigentliche Inhalt stehen. Inhaltslose Elemente können alternativ auch mit dem zusammengefaßten Tag „<“ *Name* „/>“ geschrieben werden. Attribute zu einem Element stehen hinter dessen Namen im Start-Tag. Sie bestehen aus Attributnamen und dem Attributwert in Anführungszeichen, getrennt durch ein Gleichheitszeichen. Kommentare werden zwischen „<!--“ und „-->“ eingeschlossen. Desweiteren gibt es sog. CDATA-Abschnitte, die vom XML-Prozessor nicht interpretiert werden, eingeschlossen in „<![CDATA[“ und „]]>“. Eine gute, verständliche Zusammenfassung vieler Möglichkeiten von XML findet sich in [Eckstein 00].

Ein Beispiel für ein wohlgeformtes XML-Dokument ist das folgende:

```
<?xml version="1.0"?>
<Person>
  <Vorname> Joachim </Vorname>
  <Nachname> Sauer </Nachname>
  <Status verheiratet="Nein"/>
  <!-- erweiterbar mit anderen Daten -->
</Person>
```

Dokumentenstrukturbeschreibung: Aus Gründen der Portabilität, Fehlervermeidung und Selbsterklärung können die in dem XML-Dokument erlaubten Teile und deren Struktur durch Regeln beschränkt werden. Zur Definition dieser Regeln gibt es zwei Standards:

Dokumenttypdefinitionen (DTD) sind in der originalen XML-Spezifikation beschrieben und stellen somit die am frühesten verfügbare Möglichkeit dar, XML-Dokumente durch Regeln zu beschränken. Ein DTD-Regelsatz kann auf eine ganze Menge von gleichartigen XML-Dokumenten angewendet werden, indem er in einer eigenen Datei abgelegt wird, die von den entsprechenden XML-Dokumenten referenziert wird. Alternativ dazu kann ein XML-Dokument seine eigene DTD direkt enthalten. Ein wohlgeformtes XML-Dokument, das eine DTD angibt und sich an deren Regeln hält, wird als *gültig* bezeichnet.

DTDs haben einige Nachteile, die zu der Entwicklung von XML Sche-

ma geführt haben. So sind DTDs nicht in XML geschrieben sondern in einer proprietären Sprache, was zu einem höheren Lernaufwand führt und andere Editierwerkzeuge erfordert. Einige Konstrukte in DTDs sind unintuitiv und unnötig kompliziert. Datentypen können nicht sehr detailliert beschrieben werden, es stehen nur recht allgemeine Typen zur Verfügung, die in eigenen DTDs vom Autor nicht weiter spezialisiert werden können.

Trotz dieser Nachteile haben DTDs heutzutage eine weite Verbreitung erlangt. Es ist aber zu erwarten, daß sie durch **XML Schemas** abgelöst werden, die im Gegensatz zu DTDs wohlgeformte und gültige XML-Dokumente sind. Ein wohlgeformtes XML-Dokument, das einem XML Schema genügt, wird als *Schema gültig* bezeichnet. Da XML Schema für diese Arbeit eine sehr wichtige Bedeutung hat, gehe ich im nächsten Abschnitt ausführlich auf diesen Standard ein.

Stylesheets: Ein wichtiges Konzept von XML ist die strikte Trennung von Inhalt und Darstellung. Mit einem Stylesheet kann die Formatierung der Dokumentelemente für verschiedene Ausgabemedien festgelegt werden. Dies erlaubt, aus demselben XML-Dokument verschiedene Ansichten beispielsweise für HTML-Browser, WAP-Handys, PDF-Reader und zum Ausdruck über PostScript zu erzeugen.

Einen Standard für Stylesheets stellen die **Cascading Style Sheets (CSS)** dar. Sie erlauben, strukturierte Dokumente (wie HTML- und XML-Dokumente) mit Stilinformationen zu versehen, z.B. zu Schriftarten, Farben und Abständen. Die Bezeichnung ist darauf zurückzuführen, daß ein Dokument mehrere CSS-Stilinformationen besitzen kann, beispielsweise ein CSS in einer externen Datei, mehrere direkt ins Dokument eingebunden und ein persönliches CSS des Lesers, die dann kaskadiert werden. CSS ist in zwei W^{3}C Recommendations in zwei aufeinander aufbauenden Leveln beschrieben, an CSS 3 wird momentan gearbeitet.

Die modernere, weitaus mächtigere Alternative ist die **Extensible Stylesheet Language (XSL)**, deren Spezifikation noch nicht abgeschlossen ist. Mit Hilfe von sogenannten Formatierungsobjekten wird in einem von einem XSL-Dokument gesteuerten Prozeß, der sich XSLT (XSL Transformations) nennt, aus einem XML-Dokument eine neue Darstellung. XSL baut auf CSS auf, erweitert deren Möglichkeiten aber beträchtlich, u.a. im Bereich Seitenbeschreibung und Layout. Die Spezifikation ist nicht einfach zu verstehen und änderte sich in der Vergangenheit häufig. Der Bereich ist ein großes Forschungsgebiet.

Namensräume: Sie dienen dazu, Namenskollisionen von Bezeichnern innerhalb von XML-, XSL- und XML Schema-Dokumenten zu vermeiden bzw. auflösen zu können. Dies ist wichtig, um sicherzustellen, daß

Dokumente modular verwendet werden können. Namen werden durch einen Präfix qualifiziert, der einem URI[4] zugeordnet ist, womit Eindeutigkeit der Namen sichergestellt wird. Das Konzept stellt eine wichtige Erweiterung gegenüber der ursprünglichen XML-Definition dar.

XPath: Die XML Pfadsprache spezifiziert, wie bestimmte Bestandteile eines XML-Dokuments addressiert werden können, z.B. das erste Element, dessen Name mit „WAM“ beginnt und das ein Attribut besitzt, das „Version“ heißt. Außerdem bietet es Möglichkeiten zum Musterabgleich. XPath wird insbesondere bei der XSL Transformation und XPointer (s.u.) verwendet. Es ist eine eigenständige W^{3}C Recommendation, um eine einheitliche Syntax und Semantik in diesen Spezifikationen sicherzustellen.

XLink und XPointer: Mit **XPointer** können Stellen und Bereiche im selben Dokument oder in fremden Dokumenten referenziert werden. Das können Elemente sein, die ein ID-Attribut haben, aber auch durch relative Lokalisierungsausdrücke beschriebene Stellen oder ganze Spannweiten.

Mit **XLink**, der XML Linking Language, kann man Hyperlinks basierend auf XPointer in XML-Dokumente einbauen. Dabei ist man nicht auf unidirektionale Verweise (wie aus HTML bekannt) beschränkt, sondern kann auch erweiterte Verweise verwenden, mit denen man u.a. mehrere Stellen gleichzeitig referenzieren kann.

Einige der obigen Spezifikationen sind momentan noch nicht fertiggestellt. Teilweise gibt es immer noch gravierende Änderungen. Daher sind auch noch nicht viele geschweige denn vollständige Implementationen verfügbar.

3.2 XML Schema

XML Schema wurde unter Leitung des W^{3}C entwickelt, um eine einfachere und flexiblere Möglichkeit zu schaffen, XML-Dokumente zu beschränken. Die wichtigsten Dokumente des W^{3}C zu XML Schema sind ein nichtnormativer Primer [XMLS0 01], der eine verständliche Einführung bieten soll, und die normativen Teile Structures [XMLS1 01] und Datatypes [XMLS2 01]. Alle drei sind W^{3}C Recommendations seit Anfang Mai 2001.[5]

XML Schemas haben den Vorteil, daß sie selber wohlgeformte und (durch eine DTD oder selbst durch ein XML Schema beschränkte) gültige XML-Dokumente darstellen, was zur leichteren Erlernbarkeit und zur Verarbeitung

[4] Uniform Resource Identifier (URI) dienen dazu, Ressourcen im Internet zu identifizieren, entweder durch Angabe des Ortes (klassische URL) oder durch ihren Namen oder andere Attribute.

[5] Von den W^{3}C XML Schema Dokumenten gibt es noch keine offizielle deutsche Übersetzung. Die im folgenden an einigen Stellen verwendete stammt von mir.

mit schon vorhandenen XML-Werkzeugen beiträgt. Eine wichtige Neuerung gegenüber DTDs ist, daß Deklarationen aus anderen XML Schema Dokumenten einfach miteingebunden werden können, was die Verständlichkeit und Wartbarkeit gegenüber den oft sehr großen eindokumentrigen DTDs deutlich erhöht.

3.2.1 Grundlegende Konstrukte

Eine XML Schema Definition besteht hauptsächlich aus Element- und Attributdeklarationen sowie Typdefinitionen. In diesem Abschnitt werde ich die beiden ersten sowie einige weitere Möglichkeiten erläutern. Auf die für diese Arbeit sehr wichtigen Typdefinitionen und Datentypen werde ich dann in den nächsten Abschnitten detailliert eingehen.

Elemente

Elemente werden mit dem „element“ Element deklariert. Mit dem Attribut „name“ kann ihnen ein Name zugewiesen werden, mit „type“ ein Typ. Der Elementtyp unterscheidet sich grundsätzlich darin, ob Attribute und verschachtelte Elemente zugelassen sind oder nicht. Im ersten Fall nennt man den Elementtyp komplex (**complexType**), im anderen einfach (**simpleType**).

Beispiel: <**element name**="Vorname" **type**="string"/> deklariert ein Element namens Vorname vom einfachen Typ string.

Um die Häufigkeit des Auftauchens von Elementen festzulegen, lassen sich die **Occurence Constraints** minOccurs und maxOccurs angeben. Mit minOccurs wird vorgegeben, wie oft ein Element *mindestens* erscheinen muß, mit maxOccurs entsprechend, wie oft es *höchstens* erscheinen darf. Der Vorgabewert ist für beide Attribute 1. Mit dem Wert „unbounded“ für maxOccurs gibt man an, daß keine obere Schranke existiert.

Beispiel: Um zu erreichen, daß ein Element mindestens zweimal auftreten muß, würde man minOccurs="2" und maxOccurs="unbounded" festlegen.

Mit dem Attribut „default“ kann ein Standardwert für ein Element vorgegeben werden. Alternativ kann mit „fixed“ der Wert festgelegt werden.

Attribute

Attribute besitzen immer einen einfachen Typ. Sie werden mit dem Element „attribute“ deklariert und können ebenfalls Name und Typ besitzen.

Beispiel: <**attribute name**="verheiratet" **type**="string"/> deklariert ein Attribut namens verheiratet vom Typ string.

Generell darf ein Element dasselbe Attribut höchstens einmal besitzen. Das Auftreten von Attributen kann darüber hinaus genauer durch das Attribut „use“ festgelegt werden. Dieses kann die Werte „required“ (Attribut *muß*

vorkommen), „optional“ (Attribut *kann* vorkommen; dies ist der Standardwert) und „prohibited“ (Attribut *darf nicht* vorkommen) annehmen.

Wie für Elemente sind auch bei Attributdeklarationen die Attribute „default“ und „fixed“ möglich.

Es ist möglich, Attribute in sogenannte **Attributgruppen** zusammenzufassen. Ein Vorteil dieser Vorgehensweise besteht darin, diese dann leicht an anderer Stelle wiederverwenden zu können.

Anmerkungen

Anmerkungen dienen zur Erläuterung von Schema Teilen für menschliche und maschinelle Leser. Sie werden innerhalb eines „annotation“ Elementes geschrieben. Für Menschenlesbarkeit ist das Subelement „documentation“ gedacht. Hier sollte man mit dem Attribut „lang“ die verwendete Sprache angeben. Das Subelement „appInfo“ ist dagegen für Werkzeuge, Stylesheets oder andere Applikationen vorgesehen.

Beispiel:

```
<annotation>
  <documentation lang="de">
    Dies ist eine deutsche Anmerkung.
  </documentation>
  <documentation lang="en">
    And this one is in English.
  </documentation>
</annotation>
```

Ersetzungsgruppen und abstrakte Elemente

Mit dem Mechanismus der **substitution groups** ist es möglich, das objektorientierte Konzept der Spezialisierung auf Dokumentteile zu übertragen. Elemente können für andere substituiert werden. Dabei müssen sie den gleichen Typ des Oberelementes oder einen abgeleiteten besitzen. Für ein Element lassen sich beliebig viele substituierbare Elemente definieren.

Elemente lassen sich auch als abstrakt auszeichnen. Dafür wird das Attribut „abstract“ auf „true“ gesetzt. Dies ist für Oberelemente bei Ersetzungsgruppen sinnvoll, die dann – ähnlich wie abstrakte Klassen in der OOP – nicht direkt verwendet werden können.

Ein Beispiel für Ersetzungsgruppen und abstrakte Elemente gebe ich in Abschnitt 5.1, Seite 33f.

3.2.2 Typdefinitionen

Die Definition von Datentypen durch Schema Designer kann anonym oder benannt erfolgen. Bei der **anonymen** Typdefinition erhält der Typ keinen eigenen Bezeichner. Er erscheint zwischen Start- und Endetag des Elementes oder Attributs, für das er definiert wird. **Benannte** Typen haben den Vorteil, daß sie an anderer Stelle wieder verwendet werden können. Außerdem kann die getrennte Definition von Elementen, Attributen und Typen zur Übersichtlichkeit beitragen. Hierbei erhält der Typ mit dem Attribut „name“ einen Bezeichner zugewiesen, der dann mit dem Attribut „type“ referenziert werden kann.

Wie oben bei der Erläuterung von Elementen angeschnitten, werden einfache von komplexen Typen unterschieden. **SimpleTypes** können keine Elemente oder Attribute verschachtelt enthalten. Sie werden durch Ableitung vorhandener einfacher Typen definiert, entweder durch Einschränkung, durch Bildung einer Liste oder durch Vereinigung. In Abschnitt 3.2.3 auf Seite 22f gebe ich dafür Beispiele an.

ComplexTypes können Attribute besitzen und andere Elemente beinhalten. Ein komplexer Typ kann auf Basis von vorhandenen Typen durch Einschränkung (engl. restriction) oder Erweiterung (engl. extension) mit dem Element *complexContent* definiert werden.

Bei einer Gruppe von Subelementen kann detailliert festgelegt werden, welche Elemente in welcher Reihenfolge erscheinen dürfen: Elemente, die in einer *sequence*-Gruppe sind, müssen in dieser Reihenfolge auftauchen. Elemente in einer *all*-Gruppe dürfen in beliebiger Reihenfolge erscheinen, aber nicht mehr als einmal. Von den Elementen einer *choice*-Gruppe darf nur eines auftauchen.

Mit sogenannten **Inhaltsmodellen** (engl. Element Content) können einige Besonderheiten erreicht werden:

- Falls man einem einfachen Typ ein Attribut hinzufügen möchte, kann man einen komplexen Typ mit *simpleContent* definieren.
- Wenn bei einem complexType das Attribut *„mixed“* auf „true“ gesetzt wird, können Zeichendaten auch zwischen Subelementen auftauchen.
- Elemente *ohne Inhalt* (beispielsweise sinnvoll, falls nur Attribute verwendet werden sollen) können ebenfalls definiert werden.

Für die formale Spezifikation, ausführliche Erläuterungen und Beispiele siehe [XMLS1 01].

3.2.3 Datentypen

In XML Schema wird der Begriff Datentyp so definiert ([XMLS2 01, Abschnitt 2.1]):

Definition Datatype (dt. Datentyp):

In this specification, a datatype is a 3-tuple, consisting of a) a set of distinct values, called its value space, b) a set of lexical representations, called its lexical space, and c) a set of facets that characterize properties of the value space, individual values or lexical items.

Der *Wertebereich* eines Datentypen kann dabei intensional durch Axiome, extensional durch Aufzählung aller möglichen Werte oder durch Beschränkung oder Erweiterung vorhandener Datentypen definiert werden.

Der *lexikalische Raum* bestimmt, wie Datentypen repräsentiert werden können. Da ein Wert dabei möglicherweise durch mehrere Literale dargestellt werden kann, wird eine kanonische Darstellung als bijektive Abbildung zwischen Werte- und Darstellungsbereich definiert.

Aspekte bestimmen Eigenschaften des Wertebereiches ([XMLS2 01, Abschnitt 2.4]):

Definition Facet (dt. Aspekt):

A facet is a single defining aspect of a value space. Generally speaking, each facet characterizes a value space along independent axes or dimensions.

Durch eine Synthese von verschiedenen Aspekten kann der Wertebereich eines Datentypen definiert werden. Es werden *fundamentale* von *nichtfundamentalen* oder *beschränkenden* Aspekten unterschieden.

Fundamentale Aspekte beschreiben alle Werte eines Wertebereiches semantisch. Folgende gibt es:

- Gleichheit (engl. equal)
- Ordnung (ordered)
- Schranken (bounded)
- Kardinalität (cardinality)
- numerisch (numeric)

Beschränkende Aspekte sind optional. Sie können den Wertebereich weiter einschränken. Es gibt unter anderem:

- Länge (exakt, Minimum, Maximum)
- Muster (als reguärer Ausdruck)
- Aufzählung (Angabe aller erlaubten Werte)

- Schranken (obere und untere, inklusive oder exklusive Werte)
- Stelligkeit (bei numerischen Werten)
- Nachkommastellen (bei Gleitkommazahlen)

Datentypen werden weiterhin nach unterschliedlichen Kriterien eingeteilt in atomare Typen, Listen und Vereinigungen, primitive und abgeleitete Typen sowie nach Herkunft vordefiniert oder benutzerdefiniert.

Atomare Typen, Listen und Vereinigungen

Atomare Typen (atomic datatypes) haben Werte, die nicht mehr weiter unterteilbar sind. Werte von **Listen** (list datatypes) bestehen aus einer (möglicherweise leeren) Sequenz von Werten eines atomaren Datentyps. **Vereinigungen** (unions) vereinigen den Wertebereich und seine Repräsentation von mehreren Datentypen zu einem neuen.

Primitive und abgeleitete Typen

Primitive Datentypen (primitive datatypes) existieren von Beginn an, während **abgeleitete Datentypen** (derived datatypes) durch andere Datentypen definiert sind. Die Ableitung kann entweder durch Einschränkung eines bestehenden Datentyps erfolgen, indem beschränkende Aspekte angegeben werden, oder durch Listenbildung mit einem bestehenden Datentyp als Typ der Listenelemente oder durch Vereinigung der Wertemengen mehrere Typen.

Vordefinierte Datentypen

Vordefinierte Datentypen sind die in der XML Schema Spezifikation, Teil Datentypen ([XMLS2 01]) vorgegebenen. Es gibt primitive und abgeleitete vordefinierte Datentypen. Dies sind u.a.:

Primitive Datentypen: string, boolean, float, double, decimal, duration, date, time

Abgeleitete Datentypen: language, Name, integer, nonPositiveInteger, negativeInteger, long, int, short, byte, ID

Auf diese sind jeweils passende Aspekte anwendbar. So z.B. length, minLength, maxLength, pattern, enumeration und whiteSpace bei string oder totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive und minExclusive bei den Ganzzahltypen wie integer. Eine vollständige Liste aller vordefinierten Datentypen und der erlaubten Aspekte ist in [XMLS2 01, Abschnitt 3] zu finden.

Zusätzlich gibt es den Urtyp *anyType*, der an der Spitze der Datentyphierarchie steht und jeden beliebigen einfachen oder komplexen Typ umfaßt. Ein Untertyp davon ist *anySimpleType*, der alle einfachen Typen enthält.

Benutzerdefinierte Datentypen

Schema Designer können eigene Datentypen definieren. Ein benutzerdefinierter Datentyp kann aus vorgegebenen Typen auf drei Arten abgeleitet werden (s.o.):

1. Durch Einschränkung des Wertebereichs durch zusätzliche Aspekte mit dem Element „restriction".

   Beispiel: Definition eines Betrags durch Einschränkung des vorhandenen Datentyps der Dezimalzahlen auf zwei Nachkommastellen

   ```
   <simpleType name="betrag">
     <restriction base="decimal">
       <fractionDigits="2"/>
     </restriction>
   </simpleType>
   ```

2. Erzeugung einer Liste mit einem atomaren Datentyp als Elementtyp mit dem Element „list".

   Beispiel: Liste von booleschen Werten

   ```
   <simpleType name="boolescheListe">
     <list itemType="boolean"/>
   </simpleType>
   ```

3. Als Vereinigung vorhandener Typen mit dem Element „union".

   Beispiel: Bankangabe *entweder* als Bankleitzahl *oder* mit ausgeschriebenem Banknamen

   ```
   <simpleType name="bank">
     <union memberTypes="BLZ bankname"/>
   </simpleType>
   ```

3.3 XML und Java

XML und verwandte Standards sind unabhängig von einer konkreten Programmiersprache. Für Java existieren mittlerweile viele Möglichkeiten zur Verarbeitung von XML-Daten, wohl mehr als für jede andere Sprache. Ich

möchte hier einige wichtige Rahmenwerke betrachten (SAX, DOM, JAXP und JDOM) und XML-Parser für Java vorstellen.

3.3.1 SAX, das Simple API for XML

SAX definiert ein Rahmenwerk für ein ereignisorientiertes Parsen von XML-Daten. Während mit einem SAX-Parser ein Dokument geparst wird, werden den Daten entsprechende Ereignisse erzeugt, für die man sich registrieren kann. Es gibt folgende Ereignis-Handler Schnittstellen:

- ContentHandler
- ErrorHandler
- DTDHandler

Die *ContentHandler*-Schnittstelle ist die wichtigste. Hiermit werden Benachrichtigungen erzeugt, die mit dem Inhalt des XML-Dokumentes zu tun haben, wie dem Beginn und Ende des Parsens, dem Beginn und Ende eines Elementes, dem Auftreten von Zeichen und von Processing-Instruktionen.

Über die *ErrorHandler*-Schnittstelle werden Probleme beim Parsen mitgeteilt, unterschieden in Warnungen, Fehler und fatale Fehler, die ein weiteres Parsen verhindern.

Der *DTDHandler* definiert Benachrichtigungen, die beim Parsen der DTD auftreten können.

Die erste Version von SAX wurde 1998 in einer privaten Initiative entwickelt. Sie ist genauso wie die mittlerweile aktuelle zweite Version, SAX2 vom Mai 2000, public domain (siehe [Megginson 00]) und de facto Standard.

SAX gibt nur die Schnittstellen vor, ist also – wie die Autoren betonen – *kein XML Parser*. Eine Java Implementierung dieser Schnittstellen ist unter [Megginson 00] erhältlich. Es gibt verschiedene Parser für Java, u.a. Xerces von Apache, den XML Parser von Oracle und XML4J von IBM, die sich an die SAX-Spezifikation halten und mit SAX angesprochen werden können. Für Details siehe Abschnitt 3.3.5, Seite 26.

Die Benutzung des SAX-Rahmenwerks ist recht einfach. Das Parsen ist effizient, da es sequentiell abläuft und keine großen Strukturen im Speicher gehalten werden müssen. Nachteilig ist die ungenügende Beachtung der hierarchischen Struktur von Dokumenten, die meist ein Verwalten der Hierarchiestufe im Klienten (z.B. über einen Stack) nötig macht. SAX ist ein Modell zum reinen Zugriff auf XML-Daten. XML-Dokumente können mit ihm nicht verändert oder erstellt werden.

3.3.2 DOM, das Document Object Model

Das Document Object Model definiert eine baumbasierte, objektorientierte Repräsentation von XML-Dokumenten. Beim Parsen eines XML-Doku-

mentes wird ein vollständiger, strukturgleicher Baum im Speicher erzeugt. Die einzelnen Dokumentteile bilden dabei Knoten (Nodes), die nach dem Kompositum-Entwurfsmuster (siehe [Gamma et al. 95, S. 163]) zusammengesetzt sind (siehe Abbildung 2): Ein Dokument ist ein Knoten, der wiederum andere Knoten wie Elemente und Processing Instructions beinhalten kann; ein Elementknoten kann u.a. Attributknoten, andere Elementknoten und Textknoten enthalten usw.

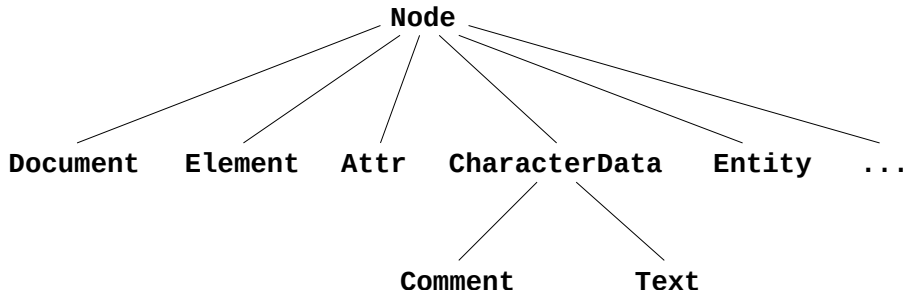


Abbildung 2: Hierarchie im DOM-Modell

Alle Knoten bieten generische und je nach Knotentyp spezielle Operationen, mit denen der Inhalt ausgelesen, durchsucht und verändert werden kann. So bietet z.B. ein Elementknoten die Möglichkeit, Attribute nach Namen zu suchen, zu setzen und zu löschen und alle Unterelemente herauszugeben. Dabei wird der Elementtext nicht als Inhalt des Elementknotens gesehen sondern als zusammengesetzte Werte aller Textknoten, die Kinder des Elementknotens sind. Dies ist eine folgerichtige und stringente Umsetzung des Kompositumkonzepts, aber wenig intuitiv. Neue Knoten sind dokumentspezifisch. Daher müssen sie zentral am Dokumentknoten erzeugt werden und können dann an passende Knoten angehängt werden.

Nachdem ein Dokumentbaum auf diese Art und Weise bearbeitet wurde, kann er wieder in ein flaches, textuelles XML-Dokument umgewandelt werden, indem der Baum geeignet XML-konform serialisiert wird.

DOM stellt ein allgemeines objektorientiertes Modell dar, dessen API programmiersprachenunabhängig in einer CORBA-IDL beschrieben ist. Es ist daher nicht auf JAVA beschränkt. DOM wird von der Document Object Model Working Group des W^{3}C entwickelt (siehe [DOMWG 01]). Es ist in aufeinander aufbauende Level unterteilt. DOM Level 1 und 2 sind offizielle W^{3}C Recommendations, Level 3 befindet sich in Arbeit, die Anforderungen sind schon spezifiziert und es existieren einige Dokumente im Working Draft Status. Level 1 beschreibt die grundlegende Funktionaliät und die Navigation innerhalb eines Dokumentes. Darauf aufbauend detailliert Level 2 den Umgang mit spezifischen Inhaltsmodellen wie beispielsweise XML, HTML und CSS und bringt u.a. Namensräume und ein Ereignismodell ein. Level 3

soll einige Lücken von Level 2 schließen und weitere Verbesserungen bringen.

Nachteilig am Document Object Model ist die erforderliche Erzeugung des Baumes, die bei großen Dokumenten signifikant Zeit und Speicher benötigt. Von Vorteil ist das Baummodell, das als Standard-Datenmodell in der Informatik seit geraumer Zeit gut verstanden ist und für das effiziente Algorithmen existieren, und die einfache Möglichkeit, XML-Dokumente zu ändern bzw. neu zu erstellen.

Das baumorientierte Modell von DOM und das ereignisorientierte Vorgehen von SAX schließen sich gegenseitig nicht aus sondern können gut kombiniert werden. Durch Auswertung der generierten Ereignisse beim Parsen eines XML-Dokumentes kann ein Dokumentenbaum erstellt werden. Andererseits ist es möglich, bei der Arbeit mit dem Baum Ereignisse zu erzeugen.

3.3.3 JAXP, das Java API for XML Processing[6]

Das, verglichen mit SAX und DOM, sehr kleine JAXP-API von Sun zielt darauf ab, eine zusätzliche Abstraktionsebene über die benutzten Parser einzuführen. Zusätzlich sind in der neuesten Version auch von konkreten Implementierungen abstrahierende XSLT-basierte Transformationen von XML-Dokumenten möglich.

Bei naiver Verwendung von DOM und SAX muß der benutzte Parser explizit durch Import und Referenz der speziellen Parserklasse integriert werden. Dies hat eine unerwünschte starke Kopplung zwischen Parser und Anwendung zur Folge, die einen Austausch des verwendeten Parsers erschwert. Mit JAXP wird dieses vermieden, indem der Parser durch eine Fabrik erzeugt wird. Das entsprechende Package `javax.xml.parsers` besteht aus nur vier abstrakten Klassen, einer Exception und einem Error. Konkrete Implementierungen der Klassen definieren einen Default-Parser, der im Standardfall benutzt wird. Von diesem Standard kann durch Systemeinstellungen (`javax.xml.parsers.SAXParserFactory` für SAX und entsprechend `javax.xml.parsers.DocumentBuilderFactory` für DOM) abgewichen werden, die entweder beim Aufruf oder durch `System.setProperty()` im Java-Code gesetzt werden können. Normalerweise beinhalten XML-Parser für Java eine Implementierung von JAXP, bei der sie selbst als Standardparser eingestellt sind.

Lange Zeit hinkte die Entwicklung von JAXP etwas hinterher, mittlerweile ist es in der Version 1.1 vom Februar 2001 aber mit Unterstützung von SAX2 und DOM Level 2 auf einem aktuellen Stand. Während es momentan noch als optionales Package erhältlich ist, soll es in die Version 1.4 der Java 2 Platform Standard Edition (J2SE) eingehen.

[6] In der Version 1.0 wurde JAXP noch als Java API for XML *Parsing* angegeben.

3.3.4 JDOM[7]

DOM und SAX sind Spezifikationen, die nicht für eine spezielle Programmiersprache entwickelt wurden.

Das Open Source Projekt JDOM unter der Leitung von Jason Hunter und Brett McLaughlin entwickelt ein speziell für Java optimiertes und seinem Programmiermodell angepaßtes Rahmenwerk (siehe [JDOM 01]). Das soll erreicht werden durch ein für Java-Programmierer natürliches API (z.B. durch Nutzung von Java Collections statt proprietärer Behälter und Java-Klassen zur Ein- und Ausgabe), das performant, robust und leichtgewichtig sein soll. JDOM bietet Schnittstellen zu SAX und DOM an. JDOM Dokumente können aus DOM Bäumen und SAX Ereignissen erstellt werden und in diese Formate exportiert werden. Die Erstellung von JDOM Dokumenten aus reinen XML-Dateien und der Export in diese ist ebenfalls möglich.

Momentan aktuell ist die Beta 6 Version vom Februar 2001. Die fertige JDOM 1.0 Version ist für August angepeilt.

Meiner Einschätzung nach ist das JDOM-API einfacher und intuitiver zu benutzen. Es bleibt aber abzuwarten, in wieweit es in der Java-XML Gemeinde Unterstützung findet. Ausführliche Beispiele zur Benutzung von JDOM sind in [Laughlin 00a] zu finden.

3.3.5 XML-Parser

Für Java gibt es eine Reihe von Parsern für XML-Daten. Sie unterscheiden sich in der Unterstützung der oben vorgestellten Spezifikationen und APIs und in ihrer Mächtigkeit. Wichtig ist die Unterscheidung in *validierende* und *nicht-validierende* Parser, da nur bei den ersteren ein XML-Dokument auf Gültigkeit in Bezug auf eine DTD oder ein XML Schema geprüft wird.

Einige der momentan am weitesten entwickelten und meist benutzten Pakete sind:

- Apache Xerces
- IBM XML4J
- Oracle XML Parser
- Sun Microsystems Project X

Von diesen macht Xerces den aktuellsten, stabilsten und vollständigsten Eindruck. Die neueste Version Xerces 1.4.0 vom 22. Mai 2001 unterstützt die meisten vorgestellten Rahmenwerke, inklusive SAX in der Version 2 und DOM Level 1 und 2. Am wichtigsten für diese Arbeit ist, daß auch die sehr aktuelle XML Schema Recommendation schon weitgehend berücksichtigt ist.

[7] JDOM ist *kein* Akronym, um den Warenzeichenbestimmungen von Sun Microsystems zu entsprechen.

Das unter `http://xml.apache.org/xerces-j/` erhältliche Paket steht unter der Apache Software License, die Modifikationen und Verteilung recht frei zuläßt (`http://xml.apache.org/LICENSE`).

Ebenfalls im Apache XML Projekt wird an Xerces 2 gearbeitet, einem Parser, der einfach, unkompliziert und modular aufgebaut sein soll. Das entsprechende Vorhaben steht allerdings noch am Anfang der Entwicklung.

4 Problemlösungskonzept

4.1 Anforderungen

Nach der Vorstellung des Fachwertkonzepts sowie der Techniken XML und XML Schema, stelle ich nun einige Anforderungen an ein verbessertes Fachwertkonzept fest. Diese orientieren sich an den identifizierten Problemen des alten Konzepts (siehe Abschnitt 2.4, S. 10).

Weniger Aufwand bei der Programmierung

Die Arbeit für den Programmierer soll verringert werden. Dies soll hauptsächlich dadurch erreicht werden, daß er nur die für den Fachwert relevanten Teile zu programmieren hat und wenig Overhead. Relevant ist in dieser Hinsicht bei fast allen Fachwerten nur die Prüfung, ob eine übergebene Repräsentation einen gültigen Fachwert darstellt oder nicht. Da die Repräsentation bei den allermeisten Fachwerten auf einem String basiert, enthalten die anderen technischen Operationen keine spezifischen Teile.

Den wenigsten Aufwand würde es also für den Anwendungsentwickler darstellen, wenn er nur die Gültigkeit einer Zeichenkette angeben würde, in möglichst einfacher Notation. Hinzu kommen in einigen Fällen fachliche Operationen für diese Fachwerte, die zusätzlich anzugeben sind. Speziellere Fachwerte, also solche, die nicht auf einer Zeichenkette beruhen oder mehrere Parameter zur Konstruktion benötigen, sollten auch nicht viel mehr Aufwand erfordern.

Eine schnellere Programmierung ist auch sehr wichtig für das *Prototyping*, das in einem evolutionären Entwicklungsprozeß mit Autor-Kritiker-Zyklen eine große Rolle spielt (vgl. [Züllighoven 98, S. 637ff]). Hier wurde in der Vergangenheit oft auf Fachwerte verzichtet, weil die Erstellung von Fachwert-Typen zu aufwendig war.

Kompatibilität zum bisherigen Rahmenwerk

Da viele Fachwert-Typen nach dem alten Konzept existieren, ist es wichtig, das alte Rahmenwerk beizubehalten. Anwendungsentwickler sollten nicht gezwungen werden, gleich umzulernen. Es sollte dem Anwendungsprogrammierer überlassen bleiben, ob er Fachwert-Typen nach dem neuen oder alten Konzept erstellt oder schon vorhandene integriert.

Das neue Konzept sollte wenig an der bestehenden Schnittstelle der Fachwert-Klasse ändern. Anwendungen sollen neue Fachwerte transparent, ohne Kenntnis des speziellen Untertyps, integrieren können. Eine Erweiterung der Schnittstelle, die die neuen Möglichkeiten reflektiert, wäre aber möglich und sinnvoll.

Gleichartige Fachwert-Typen zusammenfassen

Verwandte Fachwert-Typen sollten in einer Klasse zusammengefaßt werden können. Dies würde die Übersichtlichkeit erhöhen. Neue Fachwert-Typen sollten einfach aus anderen Fachwert-Typen durch Zusammenfassung, Erweiterung oder Einschränkung ableitbar sein. Dafür wäre es sinnvoll, wenn schon wichtige Fachwert-Typen vordefiniert wären, auf die sich aufbauen ließe.

Unterstützung offener Standards

Grundlage der neuen Fachwerte sollte kein proprietäres Konstrukt sein, sondern ein Standard, der von mehreren Rahmenwerken unterstützt werden könnte. Dieser Standard sollte möglichst weit verbreitet sein und einfach zu implementieren.

Unabhängigkeit von Java

Es sollte möglich sein, die Korrektheit eines gegebenen Fachwerts überprüfen zu können, ohne auf Java angewiesen zu sein, z.B. im Web durch JavaScript. Dafür wäre es sinnvoll, wenn die gewählte Konstruktion einfach auf andere Plattformen übertragbar wäre und Prüfroutinen und Fachwertrepräsentation nicht an Java gebunden wären.

Vorteile erhalten

Vielleicht wirken die aufgelisteten Probleme des in JWAM realisierten Fachwert-Konzeptes und die Änderungswünsche so, als ob der Sinn von Fachwerten in Frage gestellt werden würde. Dies ist nicht der Fall. Fachwerte bieten viele Vorteile, wie in Abschnitt 2 ausgeführt, und diese komen auch in der momentanen Implementation zum tragen. Das möchte ich im neuen Konzept erhalten.

4.2 Grundidee

Im folgenden möchte ich meine Idee für ein neues Fachwert-Konzept, das die obigen Anforderungen erfüllt, in ihren Grundzügen erläutern.

Das Konzept beruht auf der Verwendung von XML Schemas zur Definition von Fachwert-Typen. Um diese habe ich ein Rahmenwerk gebaut, das auf dem vorhandenen aufbaut und dieses mit XML-spezifischen Möglichkeiten erweitert.

Eine wichtige Grundsatzfrage dabei war, ob Java-Datentypen nach XML gewandelt werden, um dort anhand von XML Schemas durch XML-Parser geprüft zu werden, oder ob XML Schema Dateien in Java-Objekte umgewandelt werden und Fachwerte dann in Java mit diesen Objekten geprüft werden.

Ich habe mich für die erste Variante entschieden, da sie weniger Aufwand bedeutet, weil vorhandene XML-Parser benutzt werden können. Diese werden momentan stetig weiterentwickelt, so daß das Fachwert-Rahmenwerk von allen Verbesserungen profitieren kann. Bei einer Prüfung durch Java-Klassen müßte man alle Prüfroutinen selber programmieren und sich – aufgrund der Komplexität der XML Schema Spezifikation – auf einige Mechanismen beschränken. Außerdem ist eine der Anforderungen an das neue Fachwert-Rahmenwerk die Unabhängigkeit von der Programmiersprache Java, die mit der zweiten Variante naturgemäß nicht realisierbar wäre.

Die zweite Option hat allerdings den großen Vorteil, daß die Fachwert-Typ Definitionen nur einmal eingelesen werden. Dies führt ebenso wie die weniger aufwendige Prüfung innerhalb Javas zu Performance-Vorsprüngen. Brett McLaughlin hat in einer JavaWorld-Artikelserie (siehe [Laughlin 00b]) Vorschläge hierfür gemacht und eine erste Fassung eines prototypischen Rahmenwerks vorgestellt. Siehe auch den Ausblick auf JAXB auf Seite 47.

4.2.1 Verwendung von XML Schemas

XML Schemas bieten sich für die Umsetzung der Lösungsidee an, da sie eine Möglichkeit darstellen, Fachwertdefinitionen in einer festgelegten formalen Sprache zu erstellen und diese persistent zu speichern. XML-Dateien können mit allgemeinen oder speziellen Editoren erstellt werden. Für XML vorhandene Parser können für die Validierung genutzt werden.

Im folgenden werde ich näher beleuchten, wofür ich XML Schema verwende.

Formale Definition von Fachwerten

Wie in Abschnitt 3.2.3 ausgeführt, stellen XML Schemas eine Sprache zur Verfügung, die eine detaillierte Definition von Typen erlaubt. Sie läßt sich leicht nutzen, um damit Definitionen von stringbasierten Fachwerten zu erstellen. In meiner Untersuchung hat sich ergeben, daß die allermeisten Fachwerte stringbasiert sind und die anderen so umgeschrieben werden können, daß sie auf Strings arbeiten. Daher stellt dies keine wesentliche Einschränkung dar.

Ich habe ein Prototyp-Schema entworfen, das als Basis für eigene Fachwertdefinitionen dient. In diesem können alle Möglichkeiten, die XML Schemas bietet, für die formale Definition von Fachwerten genutzt werden. Oft reicht schon die Angabe eines regulären Ausdrucks, der auf den Fachwertstring mittels Musterabgleich zutreffen muß. Auch komplexere, zusammengesetzte Fachwerte können so konstruiert werden.

Von besonderem Vorteil ist auch die Möglichkeit der Konstruktion neuer Fachwerte aus vorhandenen durch Einschränkung, Erweiterung oder Kombination.

Persistente Speicherung der Konfiguration

Die Fachwertdefinitionen liegen in XML Schemas in Textform vor. Sie können so mit jedem Texteditor leicht erstellt werden. Ein besonderer Mechanismus zur Persistenzmachung, wie z.B. eine Serialisierung von Klassen, ist somit nicht notwendig. Auch der Austausch von Fachwert-Typ Definitionen beschränkt sich somit auf die Übertragung einer Textdatei. Dies erleichtert z.B. die Überprüfung von Eingaben bei Webmasken.

Außerdem können Änderungen an den Fachwert-Typ Definitionen vorgenommen werden, ohne daß das gesamte Anwendungssystem neu kompiliert werden muß. Dies ist sicherlich ein Vorteil. Man sollte aber beachten, daß Fachwerte nur durch Änderung einer Textdatei ihr Verhalten ändern können, was zu schwer zu findenden Fehlern führen kann.

Es ist möglich, in einer XML Schema Datei mehrere Fachwert-Typen zu definieren. So können verwandte Fachwert-Typen zusammengefaßt definiert werden.

Benutzung vorhandener Tools und Parser

XML stellt eine relativ neue Technik dar, XML Schemas eine sehr aktuelle. Trotzdem existieren schon etliche Tools und Parser. Editoren können zur Erstellung der XML Schema Dateien benutzt werden. Viele bieten eine Baumansicht der Hierarchie und Syntax Highlighting. Wichtiger ist die Möglichkeit, vorhandene Parser für das Einlesen und die Validierung nutzen zu können. Dies erspart eine Ausprogrammierung der Überprüfung von Fachwerten und ermöglicht es, neuen Entwicklungen auf dem Gebiet von XML durch Einsatz einer neuen Parser-Version Rechnung zu tragen.

Andererseits kann durch Benutzung eines fremden Parsers auch eine Abhängigkeit entstehen. In meiner Implementation habe ich deshalb darauf geachtet, daß durch entsprechende Abstraktionsmechanismen von der verwendeten Technologie diese Gefahr gebannt wird. Die Benutzung eines XML Parsers wird so optional. Ebenso möglich ist die Verwendung selber geschriebener Java-Klassen.

4.2.2 Design des Fachwert-Rahmenwerks

Das neue Fachwert-Rahmenwerk, im folgenden **XMLDomainValue** genannt, erweitert das vorhandene DomainValue-Rahmenwerk. Alle XML-basierten Fachwerte sind durch Vererbung auch herkömmliche DomainValues.

Zur schnellen Erstellung neuer Fachwert-Typen gibt es nicht nur die Möglichkeit, neue Fachwert-Typen durch Unterklassenbildung zu definieren, sondern auch generische Fachwerte. Bei diesen muß keine eigene Klasse erstellt werden. Es reicht aus, wenn die XML Schema Definition des Datentyps vorliegt.

Alle XML-basierten Fachwerte besitzen eine gemeinsame abstrakte Oberklasse, die die Fachwert-Repräsentation aufnimmt und generische Operationen zum Umgang mit ihnen anbietet. Einige davon sind zur direkten Verwendung in konkreten Fachwerten vorgesehen, andere erfordern je nach Fachwert-Typ noch spezielle Parameter und sind daher protected.

Die Typdefinitionen nach der XML Schema Spezifikation erfolgen in einer Textdatei, die durch Fachwert-Typ Entwickler erstellt wird. Für konkrete Fachwerte ist es nicht sinnvoll, jedesmal eine eigene XML Datei zu erstellen. Deshalb sehe ich einen Wrapper vor, der die XML-Daten kapselt und komfortable Operationen zu deren Erstellung bereitstellt.

Zur Prüfung der Gültigkeit von Fachwert-Repräsentationen, d.h. der Übereinstimmung der XML-Daten mit dem Fachwert-Typ XML Schema, verwende ich eigene Validator-Klassen. Um diese austauschbar zu machen, werden sie durch ein Interface gekapselt.

Ein wichtiger Punkt ist, daß bei generischen Fachwerten und bei mehreren Fachwert-Typen in einer Klasse die Fachwert-Klasse nicht mehr eindeutig den Typ identifiziert, wovon bisher implizit ausgegangen wurde. Diese weniger strenge Typisierung ist bei Einsatz von *„Type Tagging“*, zur Unterscheidung verschiedener Typen aufgrund von Zeichenketten, unvermeidbar. Bei einer speziellen Art von generischen XML Fachwerten, die auf dvXMLGenericTemplate (siehe S. 40) basieren, ist der spezielle Typ eines Fachwerts anhand seiner zugehörigen Fachwertfabrik erkennbar, was vielfach ebenfalls eine strenge Typisierung ermöglicht, z.B. in Präsentationsformen (siehe Abschnitt 6.2 auf S. 43).

Im nächsten Abschnitt werde ich genauer beschreiben, wie Fachwert-Typ Definitionen auf XML Schema abgebildet werden, und das XMLDomainValue-Rahmenwerk und seine Klassen detailliert vorstellen.

5 Implementation

5.1 Format der XML Schema Definitionen

Eine Fachwert-Typ Definition mit XML Schema soll zum einen formal korrekt sein, d.h. wohlgeformt und gültig, und die möglichen Ausprägungen des Fachwert-Typs eindeutig festlegend, zum anderen überschaubar und einfach zu nutzen und auch mit Blick auf den Quelltext zu verstehen.

Es sollte möglich sein, mehrere Fachwert-Typen in einer Datei zu implementieren und auch mehrere Dateien für unterschiedliche Fachwert-Typen zu verwenden. Ich sehe eine Standarddatei „default.xsd“ für generische XML Fachwerte vor und eine eigene Datei pro speziellem XML Fachwert, die konventionsgemäß den Dateinamen „<fachwertname>.xsd“ tragen sollten.

Jede XML-Datei, die einen konkreten Fachwert beschreibt, soll genau einen Fachwert enthalten. In den XML Schema Dateien sehe ich deshalb ein Wurzelelement „domainvalue“ vor. Dieses ist mit dem Attribut „abstract“ als abstrakt deklariert, so daß es in einer Fachwertbeschreibung nicht direkt verwendet werden darf. Es muß ersetzt werden durch einen speziellen Fachwert-Typ. Jeder spezielle Fachwert-Typ muß in der Substitutionsgruppe für das Element „domainvalue“ liegen. Durch diese Konstruktion kann eine XML Schema Datei mehrere Fachwert-Typ Definitionen enthalten, von denen ein spezieller Fachwert genau eine benutzt.

Jeder Fachwert-Typ kann ein simpler oder ein zusammengesetzter Typ sein. Alle Typen und Untertypen müssen stringbasiert sein und können beliebige Einschränkungen enthalten.

5.1.1 Generische Fachwert-Typen

Zur schnellen Erstellung von Fachwert-Typen dienen generische Fachwert-Typen. Dies sind alles einfache Datentypen, die auf dem vorgegebenen primitiven Datentyp string beruhen. Alle Definitionen von generischen Fachwert-Typen werden in der XML Schema Datei „default.xsd“ zusammengefaßt.

Diese kann beispielsweise wie folgt aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns='http://www.w3.org/2001/XMLSchema'>

  <element name="domainvalue" type="string"
           abstract="true"/>

  <element name="amount"
           substitutionGroup="domainvalue">
    <simpleType>
      <restriction base="string">
```

```
        <pattern value="[1-9][0-9]*\.\d{2}"/>
      </restriction>
    </simpleType>
  </element>

  <element name="accountName"
           substitutionGroup="domainvalue">
    <simpleType>
      <restriction base="string">
        <pattern value="\d[a-z]{2,7}"/>
      </restriction>
    </simpleType>
  </element>

</schema>
```

In dieser XML Schema Datei werden zwei Fachwert-Typen definiert, ein Typ für einen Betragswert, amount, und ein Typ, der einen Accountnamen am FB Informatik der Universität Hamburg modelliert.

Der Typ *amount* ist ein einfacher Typ, der über einen regulären Ausdruck so eingeschränkt wird, daß entsprechende Fachwerte mit einer Ziffer zwischen eins und neun beginnen müssen, gefolgt von beliebig vielen Ziffern, dem Dezimalpunkt und genau zwei Nachkommastellen. Zu regulären Ausdrücken siehe [Friedl 97].
Ein Beispielwert könnte so aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<amount xmlns:xsi=
          "http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="default.xsd">
    295.00
</amount>
```

Der Typ *accountName* ist ebenfalls ein einfacher Typ. Werte beginnen mit einer Ziffer, gefolgt von mindestens zwei und höchstens sieben Buchstaben. Dies entspricht der Konvention hier am Fachbereich für reguläre studentische Accounts. Mitarbeiter und Nebenfächler bleiben unberücksichtigt. Ein Beispielwert:

```
<?xml version="1.0" encoding="UTF-8"?>
<accountName xmlns:xsi=
          "http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="default.xsd">
    6sauer
</accountName>
```

5.1.2 Spezielle einfache stringbasierte Typen

Das Format für konkrete Fachwert-Typen, die auf einer Zeichenketten-Repräsentation beruhen, entspricht dem oben bei generischen Fachwert-Typen beschriebenen.

Anders ist in diesem Fall, daß die XML Schema Datei fachlich zusammengehörige Fachwert-Typen enthalten soll.

5.1.3 Zusammengesetzte Typen

Zusammengesetzte Fachwert-Typen lassen sich nicht aus einer einzigen Zeichenketten-Repräsentation erstellen. Ihre XML Schema Datei besitzt ebenfalls ein abstraktes „domainvalue“-Wurzelelement. Dieses ist jetzt jedoch von einem komplexen Typ, genauso wie der eigentliche Fachwert-Typ.

Beispiel: Eine Adresse mit Postleitzahl und Ort

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns='http://www.w3.org/2001/XMLSchema'>

  <element name="domainvalue" type="dvType"
           abstract="true"/>

  <complexType name="dvType"/>

  <element name="address"
           substitutionGroup="domainvalue">
    <complexType>
      <complexContent>
        <restriction base="dvType">
          <sequence>

            <element name="zip">
              <simpleType>
                <restriction base="string">
                  <pattern value="\d{5}"/>
                </restriction>
              </simpleType>
            </element>

            <element name="place" type="string"/>

          </sequence>
        </restriction>
```

```
        </complexContent>
      </complexType>
    </element>
</schema>
```

Der Typ des Elementes „domainvalue“ ist ein beliebiger komplexer Typ. Daher würde sich hier die Verwendung des *anyType* anbieten (siehe Abschnitt „Vordefinierte Datentypen“ auf S. 21f). Leider funktioniert dies mit der von mir verwendeten Version des Apache Xerces-Parsers nicht. Daher definiere ich als Typ des Wurzelelementes den komplexen Typ „dvType“, den ich nicht weiter beschränke.

Der Fachwert-Typ Adresse beschränkt diesen Typ mit complexContent aus einer Sequenz von Postleitzahl und Ort, die daher in genau dieser Reihenfolge auftauchen müssen. Beide Komponenten sind einfache Typen. Die Postleitzahl besteht aus genau fünf Dezimalziffern, der Ort kann eine beliebige Zeichenkette sein.

Ein dementsprechender Wert kann so aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<address xmlns:xsi=
           "http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="address.xsd">
    <zip>
        22527
    </zip>
    <place>
        Hamburg
    </place>
</address>
```

Anmerkung: Da ich solche zusammengesetzten Fachwert-Typen zulassen wollte, habe ich sie als *Elemente* modelliert, weil diese im Gegensatz zu Attributen Subelemente enthalten können. Ansonsten wäre es auch möglich, *Attribute* zur Typdefinition zu verwenden. Dieser Ansatz wird im schon erwähnten Prototypen von Brett McLaughlin gewählt (siehe [Laughlin 00b]).

5.2 Erweiterung des Fachwertrahmenwerks

5.2.1 Überblick über die Struktur

Das XMLDomainValue-Rahmenwerk besteht aus folgenden Schnittstellen und Klassen:

- abstract class XMLDomainValue
- class XMLWrapper
- interface XMLValidator
- class XMLValidatorXerces
- class dvXMLGeneric
- class dvXMLGenericTemplate

Hinzu kommen Beispielimplementationen von Fachwert-Typen und Testklassen.

In diesem Abschnitt möchte ich die grundlegende Struktur des XMLDomainValue-Rahmenwerks erläutern. Im nächsten Abschnitt werden alle Klassen detaillierter betrachtet. Im Anhang A sind die öffentlichen Schnittstellen zu finden.

Ein Überblick über die eigentlichen Fachwert-Typ Klassen ist in einem Klassendiagramm zu finden, Abbildung 3.

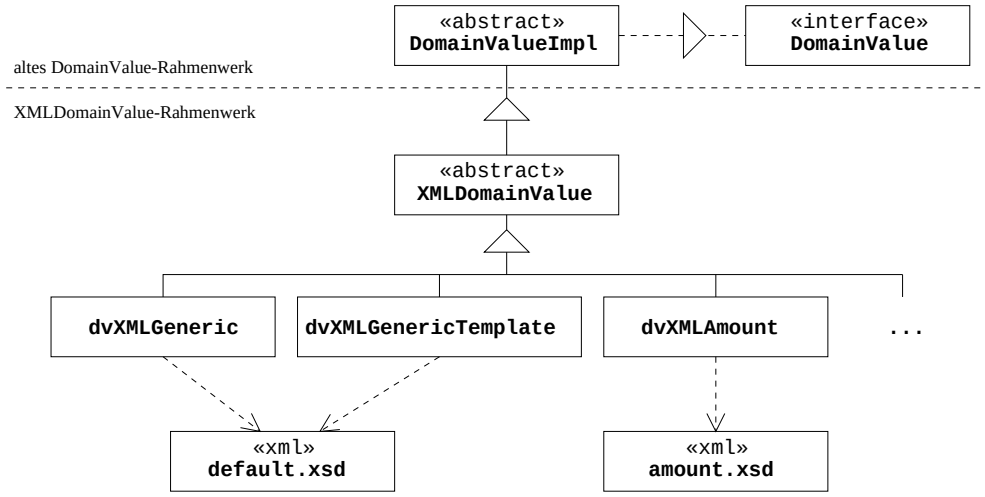


Abbildung 3: Klassendiagramm XMLDomainValue

Daraus geht hervor, daß die abstrakte Klasse **XMLDomainValue** als Unterklasse von DomainValueImpl an das bestehende Rahmenwerk angebunden ist. Die in der Oberklasse vorgegebenen Methoden sind semantisch korrekt implementiert. Außerdem werden weitere XML-spezifische Operationen zur Verfügung gestellt und die XML-Fachwert Repräsentation gespeichert. Da die Schnittstelle der Klasse im Vergleich zur Oberklasse nur um

wenige Methoden erweitert wird, habe ich keine Trennung in Interface und implementierende Klasse vorgenommen.

Wie im alten Fachwert-Rahmenwerk eingeführt, ist die Fachwertfabrik als statische innere Klasse implementiert. Diese Konstruktion ist auch in allen Unterklassen von XMLDomainValue realisiert, so daß sich zusätzlich zu der hier dargestellten Fachwert-Typ Hierarchie eine analoge für die zugehörigen Fabriken ergibt.

Die Unterklassen von XMLDomainValue sind konkrete Fachwert-Typ Klassen, nach einer Konvention benannt mit dem Präfix „dvXML“. Es gibt zwei generische Fachwert-Typen und einige konkrete Fachwert-Typen, beispielsweise dvXMLAmount. Alle diese Klassen arbeiten auf jeweils einer XML Schema-Datei. Die Dateien sind für die richtige Funktion des Rahmenwerks unerläßlich und müssen daher wie die kompilierten Class-Dateien in jeder Installation vorliegen. In meiner prototypischen Realisierung habe ich die Schema-Dateien immer im Arbeitsverzeichnis abgelegt. Bei einer Integration ins JWAM-Rahmenwerk können andere Konventionen und Zugriffsmechanismen, z.B. über das Environment oder über System Properties, sinnvoll sein.

Die beiden generischen Fachwert-Klassen sind für die schnelle Entwicklung von Fachwert-Typen gedacht, ohne daß eigene Klassen geschrieben werden müssen. Bei **dvXMLGeneric** ist bei den Fabrikoperationen zusätzlich zum Wert auch immer der gewünschte Typname mit anzugeben. In manchen Fällen ist die Verwendung von **dvXMLGenericTemplate** praktischer. Hier wird beim Zugriff auf das Fabrikexemplar eine für den entsprechenden Fachwert-Typ spezialisierte zurückgegeben. Dies hat den Vorteil, daß die Fabrikoperationen ohne zusätzlichen Typname-Parameter auskommen und damit kompatibel zu einigen Komponenten wie Präsentationsformen sind (siehe auch Abschnitt 6.2).

Der Zusammenhang zwischen den Hilfsklassen ist dem Klassendiagramm in Abbildung 4 zu entnehmen.

Damit die Klassen des XMLDomainValue-Rahmenwerks nicht neben den XML Schema Fachwert-Typ Definitionen auch noch mit tatsächlichen XML-Dateien für konkrete Fachwerte umgehen müssen, kapselt die Klasse **XMLWrapper** diese. Sie stellt Möglichkeiten zu ihrer Erzeugung und sinnvolle Zugriffsoperationen bereit. Die Klasse wird von den meisten anderen Klassen des Rahmenwerks benutzt.

Die eigentlichen Parsing-Operationen sollen so gut wie möglich gekapselt und damit austauschbar gemacht werden. Hierfür git es das Interface **XMLValidator**, das die vom Rahmenwerk benötigten Methoden enthält. Konkrete Implementationen können diese Anforderungen dann auf verschiedene Art und Weise, beispielsweise bzgl. der verwendeten Technologie, der Geschwindigkeit und des Speicherplatzverbrauches, erfüllen. Dieses Konstrukt ist eine Umsetzung des Entwurfsmusters Strategie (siehe [Gamma et al. 95,

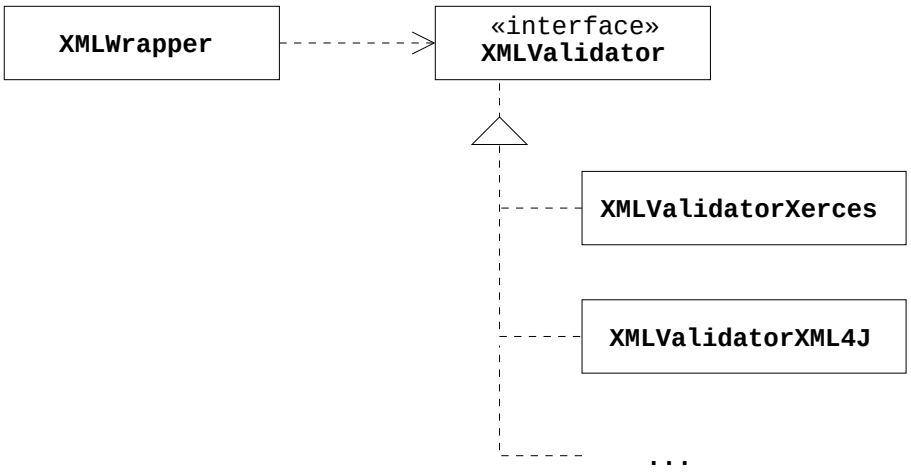


Abbildung 4: Klassendiagramm XMLWrapper, XMLValidator

S. 315]). Eine konkrete Implementierung ist **XMLValidatorXerces**, die auf dem in Abschnitt 3.3.5 vorgestellten Xerces Parser des Apache XML Projektes beruht. Weitere Möglichkeiten wären z.B. die Verwendung des IBM Parsers XML4J oder selber geschriebene Parsing-Klassen.

5.2.2 Die Klassen im einzelnen

XMLDomainValue

Abstrakte Unterklasse von DomainValueImpl und Oberklasse aller Fachwert-Typen, die auf XML beruhen.

Eines der Hauptziele des neuen Fachwert-Rahmenwerks ist, die Fachwert-Typ Programmierung zu vereinfachen. Daher habe ich versucht, soviel Funktionalität wie möglich schon in der Oberklasse zu realisieren. Dabei war es sehr hilfreich, daß alle Fachwert-Typen auf einer XML-Repräsentation beruhen. So kann diese der Oberklasse im Konstruktor übergeben werden, hier gespeichert und in Operationen verwendet werden. Diese sind `value()`, die den Fachwert als String zurückliefert, `typename()`, die den Typ des Fachwert zurückgibt und `toStringImpl()`, die als Einschuboperation für `toString()` dafür sorgt, daß der Fachwert korrekt ausgegeben wird. Der Rückgabeparameter der geschützten Methode `xmlDescription()` ist ein XMLWrapper mit der Fachwertrepräsentation. Er kann von konkreten Fachwert-Typ Klassen zur Implementation eigener Zugriffsmethoden genutzt werden.

In der zugehörigen Fabrik, die ebenfalls abstrakt ist, werden einige als sinnvoll erscheinende Voreinstellungen getroffen, die in Unterklassen aber überschrieben werden können. So wird festgelegt, daß Fachwerte nicht aus einer String-Repräsentation erstellt werden können, damit natürlich auch

nicht aus einer ungültigen String-Repräsentation, und daß ein ungültiger Fachwert existiert. Außerdem wird vorgegeben, daß alle XML-Fachwerte mit `canCreateValues(String typename)` gefragt werden können, ob sie Werte des übergebenen Typs erstellen können.

Weiterhin sind in der Fabrik einige geschützte Operationen implementiert, die von Unterklassen mit entsprechenden Parametern aufgerufen werden können und so nicht in jeder Unterklasse einzeln implementiert werden müssen. Dies sind `canCreateValues(String typename, Sting filename)`, `invalidityExplanation(String typename, String filename, String value)` und `isValid(String typename, String filename, String value)`.

dvXMLGeneric

Unterklasse von XMLDomainValue, realisiert generische Fachwerte durch zusätzliche Typparameter in Methoden.

Ohne eigene Fachwertklasse lassen sich mit dieser Klasse neue Fachwert-Typen benutzen. Die verwendete Fabrik ist in der Lage, eine Vielzahl von Fachwert-Typen zu erstellen. Bei den Fabrikmethoden ist dabei zusätzlich zum Wert auch der Typname als String anzugeben. Dies ist der Fall bei den Methoden `isValid()`, `invalidityExplanation()` und `value()`.

Da immer der Typname zusätzlich übergeben werden muß, können nicht die in DomainValueImpl vorgegebenen Methoden benutzt werden, die nur den Wert als Parameter erfordern. Außerdem ist keine Typsicherheit mehr gegeben. Wenn eine Variable von einem generischen Fachwert-Typ ist, kann ihr konkreter Fachwert-Typ nicht vorgegeben werden. Dies ist der Preis der Flexibilität der Fabrik.

dvXMLGenericTemplate

Unterklasse von XMLDomainValue, realisiert generische Fachwerte durch spezialisierte Fabriken.

Diese Klasse stellt eine alternative Realisierung von generischen Fachwerten dar. Der Nachteil von dvXMLGeneric, daß Fachwerte nicht nur aus einer String-Repräsentation erstellt werden können, sondern immer noch einen zusätzlichen Typ-Parameter erfordern, was dazu führt, daß einige Komponenten wie Präsentationsformen nicht einfach benutzt werden können, ist hier vermieden. Dies wird erreicht durch spezialisierte Fabriken. Die Fabrikklasse hält in einer Hashmap mehrere spezialisierte Fachwertfabriken vorrätig, eine für jeden Typ. Der `instance()`-Methode, die eine konkrete Fabrik zurückliefert, wird der gewünschte Typ übergeben. Dies ähnelt dem Template-Mechanismus in C++, aus dem sich der Klassenname ableitet.

Die spezialisierte Fabrik bietet dann Methoden an, die nur den Wert erfordern. Einer Variable vom generischen Fachwert-Typ kann auch hier nicht der konkrete Fachwert-Typ vorgegeben werden. Falls aber nur Fachwerte

einer bestimmten Fabrik zugelassen sind, wie es in einigen Beispielen der Fall ist, siehe Darstellung in Präsentationsformen in Abschnitt 6.2, kann die Typsicherheit gewährleistet werden.

Als ein Nachteil bleibt, daß die `instance()`-Methode mit Typparameter nicht dem Standard ohne Parameter entspricht. Dies spielt eine Rolle, wenn die Fabrik eines bestehenden Fachwerts gesucht wird. Hierzu gibt es die in DomainValueImpl definierte Methode `factory()`, die mittels Reflection die `instance()`-Methode der zugehörigen Fabrikklasse aufruft. In der dvXML-GenericTemplate Klasse habe ich diese Methode überschrieben, so daß sie jetzt korrekt den Typparameter übergibt.

XMLWrapper

Klasse, die einen Wrapper um XML-Dateien darstellt, mit Erzeugungs- und vielfältigen Zugriffsmethoden.

Bei der Erzeugung wird dem Konstruktor entweder eine SAX InputSource übergeben, aus der das XML-Dokument gewonnen wird, oder die drei Parameter Typname, Dateiname des dazugehörigen XML Schemas, und Wert des Fachwerts, aus denen das XML-Dokument zusammengestellt wird.

Mit der Methode `isWellformed` kann abgefragt werden, ob dieses wohlgeformt ist, mit `dom()` erhält man einen DOM-Baum des Dokuments. Die Methode `type()` liefert den Typ des im Dokument enthaltenen Fachwerts zurück, die Methode `value()` den Wert. Bei zusammengesetzten Fachwerten kann der Wert nicht automatisch extrahiert werden. Daher muß er bei diesen Fachwerten in der Fachwertklasse aus dem DOM-Baum ausgelesen werden.

Zur Realisierung dieser Operationen benötigt der XMLWrapper Zugriff auf einen XML Parser. In der momentanen prototypischen Implementation hält er selber eine Referenz auf einen konkreten XMLValidator, die er mit der Methode `validator()` herausgibt. Bei Integration in das JWAM-Rahmenwerk könnte der Validator auch im Environment registriert und von dort abfragbar sein.

XMLValidator

Interface, das alle benötigten Parsing-Operationen zusammenfaßt.

Die Schnittstelle enthält die Methode `check(XMLWrapper xml)`, die prüft, ob der übergebene XMLWrapper Schema gültig ist, und die vergleichbare Methode `checkWithReport(XMLWrapper xml)`, die zusätzlich in einem String alle evtl. gefundenen Warnungen und Fehler mitgibt. Außerdem die Methode `contains(String filename, String dvname)`, die prüft, ob in dem XML Schema, das unter dem übergebenen Dateinamen abgespeichert ist, der Fachwert-Typ namens dvname definiert ist, und die wichtige Methode

`parse(InputSource is)`, die eine SAX InputSource parst und ein DOM-Dokument zurückliefert.

Außerdem sind die Methoden `newDocument()`, die ein neues, leeres, DOM-Dokument zurückgibt, und `writeFile(Document doc)`, die das übergebene Dokument serialisiert auf die Standardausgabe ausgibt, enthalten.

XMLValidatorXerces

Implementiert das XMLValidator-Interface, verwendet den Xerces Parser für das Parsing.

Diese Klasse ist als Singleton implementiert. Da die Erzeugung eines Parsers eine komplexe Operation ist, ist es aus Performancegründen sinnvoll, nur ein Exemplar für alle Parsing-Operationen zu verwenden. Damit sich diese nicht gegenseitig stören und falsche Ergebnisse liefern, habe ich den Validator zustandslos implementiert. Für jede Operation sind alle benötigten Parameter zu übergeben. Operationen haben für den Parser keine Nebeneffekte. Mit der Methode `instance()` erhält man das Singleton-Objekt. Intern wird geprüft, ob es schon angelegt wurde, dann wird diese Referenz herausgegeben, ansonsten wird es instantiiert.

In der Klasse verwende ich JAXP zur Abstraktion von der speziellen Parserklasse, SAX und DOM für das Parsen und letzteres auch für Manipulationen am Dokument. JDOM benutze ich nicht, da abzuwarten bleibt, ob es sich in der Entwicklergemeinde durchsetzen kann oder nicht.

Eine private innere Klasse ist der Errorhandler, der die Callbacks für Warnungen, Fehler und schwere Fehler anbietet und daraus einen Fehlerbericht erstellt. Mit der Methode `parsingWithoutError()` kann abgefragt werden, ob Fehler aufgetreten sind, mit der Methode `errorlog()` kann man den Fehlerbericht erhalten.

6 Verwendung des Rahmenwerks

6.1 Zusammengesetzte Fachwert-Typen

Am Beispiel des aus Postleitzahl und Ort bestehenden Fachwert-Typs Adresse möchte ich zeigen, wie sich zusammengesetzte Fachwert-Typen realisieren lassen.

Bei Verwendung des alten Fachwert-Rahmenwerks würde man drei Fachwert-Typen definieren müssen, in drei verschiedenen Klassen: Einen Typ Postleitzahl, einen Typ Ort und den Typ Adresse, der aus den beiden Teilen zusammengesetzt ist. Es wäre auch möglich, für Postleitzahl und Ort einfache Datentypen wie Ganzzahl und String zu verwenden, dies wäre allerdings konzeptuell nicht sauber.

Mit XML-basierten Fachwert-Typen läßt sich dies einfacher realisieren. Das zur Typdefinition verwendete XML Schema ist auf Seite 35 abgedruckt. Postleitzahl und Ort sind als einfache Datentypen definiert, die Adresse als Sequenz aus beiden. In der Fabrik der Fachwert-Typ Klasse dvXMLAddress benötigen die entsprechenden Operationen zwei Parameter, Postleitzahl und Ort. Der kapselnde XMLWrapper kann nicht mehr mit der convenience-Operation erzeugt werden, die nur auf einen einfachen Wert ausgelegt ist. Daher muß die XML-Datei selber konstruiert werden, im Beispiel realisiert durch einfache Stringkonkatenation.

Am Fachwert selber sind Postleitzahl und Ort getrennt abfragbar. Hier muß jeweils entsprechender Code geschrieben werden, der die geforderten Daten aus dem DOM-Baum des Fachwerts ausliest, ein – wenn auch zugegebenermaßen etwas kompliziert erscheinender – Einzeiler. Die Standard-`value()`-Methode liefert eine Konkatenation von Postleitzahl und Ort zurück.

Wie man sieht, ist die Konstruktion etwas aufwendiger als bei einfachen Typen, insbesondere da noch keine Hilfsoperationen definiert sind, die Teile automatisieren (siehe den Ausblick in Abschnitt 7.3). Da jedoch alle Komponenten als Datentypen in *einem* XML Schema definiert werden können, ist die Konstruktion einfacher und übersichtlicher als beim alten Rahmenwerk.

6.2 Einsatz in Präsentationsformen

Wie mehrfach erwähnt, spielt die Ein- und Ausgabe von Fachwerten eine große Rolle. Bei heutigen GUI-basierten Systemen, wie sie am Arbeitsplatz anzutreffen sind, geschieht dies in Formularen. Fachwerte können und sollen direkt in den Präsentations- und Interaktionsklassen verwendet werden, da sie aufgrund ihrer Wertsemantik kontextunabhängig sind und das Prinzip der Trennung von Handhabung und Präsentation nicht verletzen (vgl. [Züllighoven 98, S. 263ff]).

Die Ausgabe ist unkritisch, da jeder Fachwert eine Zeichenkettenreprä-

sentation von sich liefern kann, die in entsprechende Felder ausgegeben werden kann. Interessanter wird es bei der Eingabe. Es ist sinnvoll, dem Anwender schon während des Tippens eine Rückmeldung zu geben, ob seine Eingabe syntaktisch richtig ist, also einen gültigen Fachwert darstellt. Nach Ende der Bearbeitung muß die Gültigkeit der eingegebenen Repräsentation geprüft und daraus ein Fachwert-Objekt erstellt werden.

Im JWAM-Rahmenwerk ist das Konzept der Präsentationsformen und Interaktionstypen realisiert (für eine ausführliche Darstellung möchte ich auf [Bleek et al. 99b] verweisen). Präsentationsformen dienen zur Darstellung von Daten, und Interaktionsformen bestimmen die möglichen Umgangsformen. So kann z.B. ein bestimmter Wert eines Aufzählungstyps aus einer Präsentationsform mit einer Liste aller möglichen Werte oder aus einer Präsentationsform mit Radioboxes ausgewählt werden. Der Interaktionstyp, 1-aus-n-Auswahl, ist jeweils derselbe.

Ich möchte hier zeigen, wie XML-basierte Fachwerte in dieses System eingebunden werden können. Am einfachsten ist der Einsatz von Klassen für generische Fachwerte. Der XML Fachwert-Typ dvXMLGenericTemplate wurde so konstruiert, daß er damit verwendet werden kann. Als Präsentationsform kann **pfJDomainValueFillInField** verwendet werden. Beim Erzeugen dieses Widgets für ein Formular kann die Fabrikklasse der zugehörigen Fachwerte mit angegeben werden. Hier würde man eine mit dem Template-Mechanismus spezialisierte Fabrik übergeben. Der dazu passende Interaktionstyp ist **ifGenericDomainValueFillIn**, das – der Name weist darauf hin – zur Eingabe generischer Fachwerte gedacht ist. Dies reicht aus, um XML Fachwerte in Formularen zu präsentieren. Natürlich können auch eigene Präsentationsformen für spezielle XML Fachwert-Typen geschrieben werden.

Als ein Anwendungsbeispiel möchte ich auf das im JWAM-Kontext bekannte Cashier Tool zurückgreifen. Es ist ein Werkzeug zur Bearbeitung von Konto-Daten und zur Durchführung von Abbuchungen und Einzahlungen, siehe Abbildung 5.

Ich habe hier den Gleitkommazahl-Wert, der den Betrag (engl. amount) repräsentiert, durch einen Fachwert vom Typ amount ausgetauscht, der unter Verwendung der Klasse dvXMLGenericTemplate realisiert ist. Dessen XML Schema Typdefinition entspricht der auf Seite 34. Ich verwende die Präsentationsform pfJDomainValueFillInField, der ich die mit dem Typnamen „amount“ parametrisierte Fachwertfabrik übergebe. Wie oben beschrieben, ist die zugehörige Interaktionsform vom Typ ifGenericDomainValueFillIn.

Bei Eingabe eines Betrages signalisiert jetzt die Farbe der Eingabezeichen, ob der Fachwert gültig ist. Schwarz bedeutet in Ordnung, rot Fehler. Erst nach Eingabe eines korrekten Fachwerts mit Dezimalpunkt und zwei Nachkommastellen wechselt die Farbe auf schwarz.

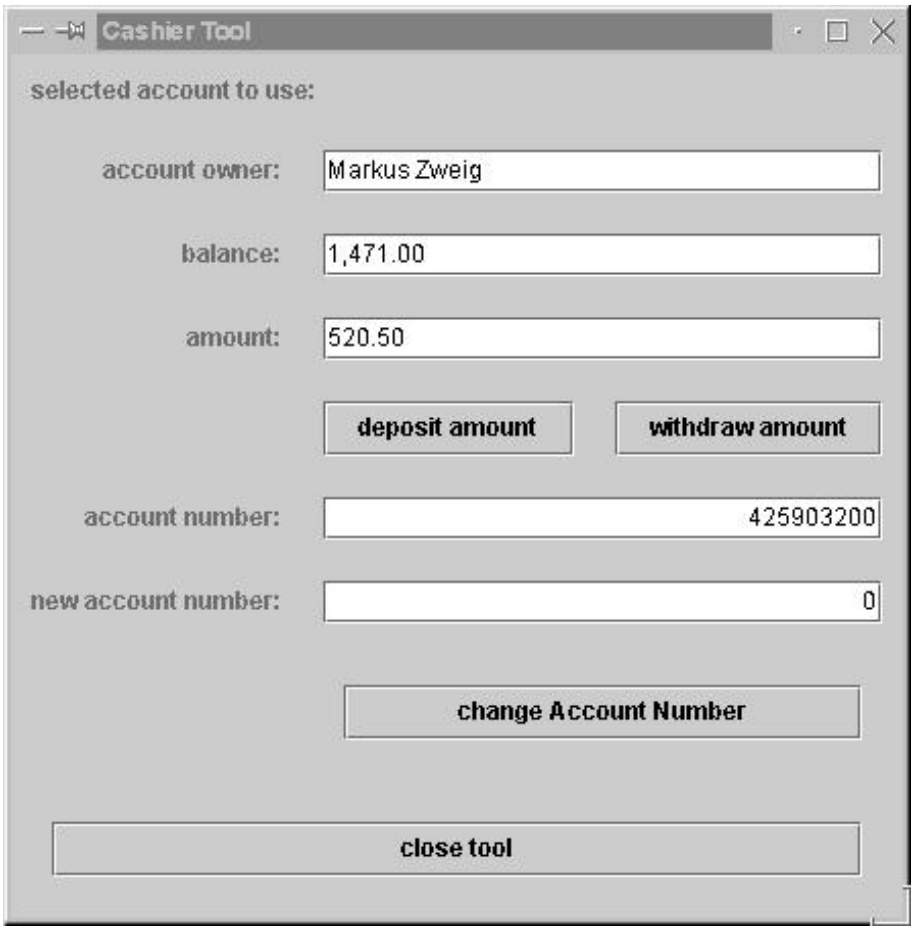


Abbildung 5: Screenshot des Anwendungsbeispiels CashierTool Werkzeug

7 Schlußwort

7.1 Zusammenfassung

In dieser Arbeit habe ich ein Rahmenwerk zur einfachen Erstellung von fachlichen Datentypen aus XML Schema Beschreibungen für das JWAM-Rahmenwerk entwickelt. Ich habe erläutert, was Fachwerte sind, welche Vorteile sie besitzen und wie sie realisierbar sind. Die Umsetzung in JWAM habe ich näher betrachtet und einige Nachteile herausgearbeitet.

Danach habe ich mich dem Datenaustauschformat XML und verwandten Standards zugewandt. Die Strukturbeschreibungssprache XML Schema mit ihren Möglichkeiten, Datentypen zu definieren, habe ich genau beschrieben. Anschließend habe ich Anforderungen an ein besseres Fachwert-Rahmenwerk herausgearbeitet und Ideen für ein solches entwickelt. Dieses habe ich dann prototypisch umgesetzt, die Struktur, wichtige Entwurfsentscheidungen und die einzelnen Klassen erläutert.

Ich habe beschrieben, wie sich XML Fachwerte einsetzen lassen, wie auch zusammengesetzte Fachwert-Typen realisierbar sind und wie der Einsatz in Präsentationsformen aussehen kann.

7.2 Bewertung

Ein Hauptziel war die weniger aufwendige Erstellung von neuen Fachwert-Typen. Durch die Einführung von generischen Fachwert-Typen habe ich dieses gut erreichen können. Nachteilig bei ihrem Einsatz wirkt sich die fehlende Typsicherheit aus. Daher sind generische Fachwert-Typen besonders für Prototypen interessant, in denen sonst keine Fachwerte eingesetzt werden würden. Spezielle Fachwert-Typ Klassen lassen sich ebenfalls erstellen, doch ist hierbei der Aufwand deutlich größer als bei generischen Fachwerten. Dies liegt daran, daß einige Teile nicht vordefinierbar sind. So muß z.B. jedesmal die spezielle Fachwertfabrik-Klasse geschrieben werden. Sie muß das Exemplar der konkreten Fabrik aufbewahren und in einer `instance()`-Methode herausgeben.

Ein großer Nachteil ist die geringe Geschwindigkeit des neuen Rahmenwerks, die im Konzept begründet liegt. Jedesmal wenn ein neuer Fachwert erzeugt wird, wird die Korrektheit seiner Repräsentation durch Validierung mit Hilfe eines XML Schemas festgestellt. Der entsprechende Parsing-Vorgang ist aufwendig. Daher eignet sich das XMLDomainValue-Rahmenwerk nicht für Anwendungen, in denen viele Fachwerte in kurzer Zeit erzeugt werden müssen.

Mit Blick auf die Einschränkungen der alten Realisierung des Fachwert-Rahmenwerks und die aufgestellten Anforderungen an ein neues Konzept, glaube ich, daß das vorliegende XMLDomainValue-Rahmenwerk die alten Probleme gut lösen kann. Die Zukunft wird zeigen, ob es in der Praxis seinen Zweck erfüllt.

7.3 Ausblick

In Anbetracht der sehr aktuellen verwendeten Technologie sind für die Zukunft einige Fortschritte zu erwarten. Java und XML werden immer weiter zusammenwachsen. Nach Auskunft von Sun Microsystems wird die Version 1.4 der J2SE, die für das vierte Quartal 2001 angekündigt ist, weitere Möglichkeiten des Zugriffs auf XML-Daten beinhalten, u.a. wird das JAXP Rahmenwerk ein Bestandteil sein.

Zu analysieren wird sein, inwieweit neue Techniken und APIs in das Fachwert-Rahmenwerk integriert werden können oder alternative Möglichkeiten darstellen. Bei Sun wird momentan an JAXB, der Java Architecture for XML Binding, gearbeitet, mit der sich XML Schema Dateien in Klassen umwandeln lassen können[8]. Anhand dieser Klassen sollen dann XML Dokumente auf Schema Gültigkeit untersucht werden können, ohne daß jedesmal ein SAX- oder DOM-basierter Parser die XML Schema Datei einlesen muß. Dieses Verfahren könnte die Geschwindigkeit des XMLDomainValue-Rahmenwerks deutlich steigern.

Eine andere Technik ist die der automatischen Generierung von Fachwert-Typ Klassen. Da viele Fachwerte einen standardisierten Aufbau haben, insbesondere in Systemen in einem eher technischen Umfeld, reichen wenige Parameter zu ihrer Beschreibung aus. Aus diesen kann dann Quelltext für Fachwert-Typ Klassen generiert werden. Auch hier können XML und verwandte Technologien genutzt werden, z.B. durch Beschreibung der speziellen Fachwertparameter in XML Dokumenten und Umwandlung dieser in Quelltexte durch einen XSLT Prozeß. Am Fachbereich Informatik der Universität Hamburg läuft eine Diplomarbeit zu fachlichen Werten und ihrer Implementierung in objektorientierten Programmiersprachen, die sich auch mit dieser Thematik beschäftigt.

Im Detail läßt sich das vorliegende Rahmenwerk ebenfalls verbessern. So würde es sich anbieten, Möglichkeiten zu schaffen, XML Schema Dateien komfortabler zu erstellen und zu bearbeiten. Momentan müssen sie noch von Hand editiert werden. Vorstellbar wären Softwarewerkzeuge oder -methoden, die diese Arbeit kapseln und Syntaxfehler vermeiden helfen. Man könnte auch über eine bessere Unterstützung zusammengesetzter Fachwert-Typen nachdenken, z.B. durch vordefinierte Methoden, die Arrays von Strings (den einzelnen Fachwertkomponenten entsprechend) als Parameter verarbeiten können.

Es wird interessant zu beobachten sein, inwieweit sich das Fachwert-Konzept in der Entwicklergemeinde durchsetzen wird. Vielleicht wird ja irgendwann in zukünftigen objektorientierten Programmiersprachen ein um

[8] Die Homepage des Projektes ist unter `http://java.sun.com/xml/jaxb/` zu finden. Bei Abschluß dieser Arbeit aktuell war Working Draft Version 0.21 vom 30. Mai 2001.

fachliche Datentypen erweiterbares Typsystem integriert werden, das heutige Fachwert-Rahmenwerke überflüssig macht.

Anhang

A Schnittstellen des Rahmenwerks

Hier sind die vollständigen öffentlichen Schnittstellen der von mir programmierten Klassen aufgelistet sowie wichtige Methoden der Oberklassen, mit * markiert.

XMLDomainValue

```
public abstract class XMLDomainValue
    extends DomainValueImpl {
  public boolean equals(Object value); *
  public DomainValue.Factory factory(); *
  public String invalidCreationString(); *
  public boolean isDefined(); *
  public String toString(); *
  public String[] toStrings(); *
  public String typename();
  public String value();
}
```

XMLDomainValue.Factory

```
public abstract static class XMLDomainValue.Factory
    extends DomainValueImpl.Factory {
  public boolean canCreateFromInvalidString(); *
  public boolean canCreateFromString(); *
  public abstract boolean canCreateValues(String typename);
  public DomainValue defaultValue(); *
  public boolean hasUndefinedValue();
  public String invalidityExplanation(String s); *
  public boolean isValid(String s); *
  public Class type(); *
  public abstract DomainValue undefinedValue();
  public DomainValue value(String s); *
}
```

XMLWrapper

```
public class XMLWrapper {
  public XMLWrapper(InputSource is);
  public XMLWrapper(String typename, String filename,
                    String value);
```

```
  public Document dom();
  public boolean isWellformed();
  public String type();
  public XMLValidator validator();
  public String value();
}
```

XMLValidator

```
public interface XMLValidator {
  public boolean check(XMLWrapper xml);
  public String checkWithReport(XMLWrapper xml);
  public boolean contains(String filename, String dvname);
  public Document newDocument();
  public Document parse(InputSource is);
  public void writeFile(Document doc);
}
```

XMLValidatorXerces

```
public class XMLValidatorXerces
    implements XMLValidator {
  public boolean check(XMLWrapper xml);
  public String checkWithReport(XMLWrapper xml);
  public boolean contains(String filename, String dvname);
  public static XMLValidator instance();
  public Document newDocument();
  public Document parse(InputSource is);
  public void writeFile(Document doc);
}
```

dvXMLGeneric

```
public class dvXMLGeneric
    extends XMLDomainValue {
  public DomainValue.Factory factory(); *
  public String typename(); *
  public String value(); *
}
```

dvXMLGeneric.Factory

```
public static class dvXMLGeneric.Factory
    extends XMLDomainValue.Factory {
  public boolean canCreateValues(String typename);
  public static dvXMLGeneric.Factory instance();
  public String invalidityExplanation(String typename,
                                      String value);
  public boolean isValid(String typename, String value);
  public DomainValue undefinedValue();
  public dvXMLGeneric value(String typename,
                            String value);
}
```

dvXMLGenericTemplate

```
public class dvXMLGenericTemplate
    extends XMLDomainValue {
  public DomainValue.Factory factory();
  public String invalidCreationString();
  public String typename(); *
  public String value(); *
}
```

dvXMLGenericTemplate.Factory

```
public static class dvXMLGenericTemplage.Factory
    extends XMLDomainValue.Factory {
  public boolean canCreateFromInvalidString();
  public boolean canCreateFromString();
  public boolean canCreateValues(String typename);
  public static dvXMLGenericTemplate.Factory
    instance(String typename);
  public boolean isValid(String s); *
  public DomainValue undefinedValue();
  public DomainValue value(String s); *
}
```

B Glossar

API	Application Programming Interface
BLOB	Binary Large Object
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheets
DTD	Document Type Declaration
DOM	Document Object Model
EBNF	Erweiterte Backus-Naur Form
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDL	Interface Definition Language
J2SE	Java 2 Platform Standard Edition
JAXB	Java Architecture for XML Binding
JAXP	Java API for XML Processing
JWAM	Java WAM
OOP	Objektorientierte Programmierung
PDF	Portable Document Format
SAX	Simple API for XML
SGML	Standard Generalized Markup Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WAM	Werkzeug, Automat, Material
WAP	Wireless Application Protocol
W^3C	World Wide Web Consortium
WWW	World Wide Web
XLink	XML Linking Language
XPath	XML Path Language
XPointer	XML Pointer Language
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations

Literatur

[Awizen 00] M. Awizen: *Einsatz von XML-Schemata zum generischen Zugriff auf objektorientierte Persistenzsysteme*, Diplomarbeit am Fachbereich Informatik, Universität Hamburg, 2000

[Bäumer et al. 98] D. Bäumer et al.: *Values in Object Systems*, Ubilab Technical Report 98.10.1, Zürich 1998

[Bleek et al. 99a] W.-G. Bleek et al.: *Frameworkbasierte Anwendungsentwicklung (Teil 4): Fachwerte*, in: OBJEKTSpektrum 5/99

[Bleek et al. 99b] W.-G. Bleek et al.: *Frameworkbasierte Anwendungsentwicklung (Teil 3): Die Anbindung von Benutzungsoberflächen und Entwicklungsumgebungen an Frameworks*, in: OBJEKTSpektrum 3/99

[DOMWG 01] W^{3}C Working Group: *Document Object Model (DOM)*, Juni 2001: `http://www.w3.org/DOM/`

[Eckstein 00] R. Eckstein: *XML - kurz & gut*, O'Reilly & Associates Inc., Köln 2000

[Friedl 97] J. Friedl: *Mastering Regular Expressions*, O'Reilly & Associates Inc., Sebastopol 1997

[Gamma et al. 95] E. Gamma et al.: *Design-Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[JDOM 01] JDOM Rahmenwerk, 2001: `http://www.jdom.org/`

[JValue 01] JValue Rahmenwerk, Mai 2001: `http://www.jvalue.org/`

[JWAM 01] JWAM Rahmenwerk, Juni 2001: `http://www.jwam.de/`

[Laughlin 00a] B. McLaughlin: *Java and XML*, O'Reilly & Associates Inc., Sebastopol 2000

[Laughlin 00b] B. McLaughlin: *Validation with Java and XML schema, Parts 1–4*, `http://www.javaworld.com/javaworld/`, September bis Dezember 2000

[Megginson 00] D. Megginson: *SAX 2.0: The Simple API for XML*, `http://megginson.com/SAX/`

[Meyer 97] B. Meyer: *Object-Oriented Software Construction*, Prentice-Hall, New York, London, 1997

[Müller 99] K. Müller: *Konzeption und Umsetzung eines Fachwertkonzeptes*, Studienarbeit am Fachbereich Informatik, Universität Hamburg, 1999

[W3C 00] I. Jacobs: *About the World Wide Web Consortium*, März 2000: `http://www.w3.org/Consortium/`

[W3XML 00] *XML 1.0 (Second Edition)*, W^{3}C Recommendation 6. Oktober 2000: `http://www.w3.org/TR/2000/REC-xml-20001006/`

[XMLS0 01] *XML Schema Part 0: Primer*, W^{3}C Recommendation 2. Mai 2001: `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`

[XMLS1 01] *XML Schema Part 1: Structures*, W^{3}C Recommendation 2. Mai 2001: `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/`

[XMLS2 01] *XML Schema Part 2: Datatypes*, W^{3}C Recommendation 2. Mai 2001: `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/`

[Züllighoven 98] H. Züllighoven: *Das objektorientierte Konstruktionshandbuch*, dpunkt.verlag, 1998