Konzepte und Ansätze zur Unterstützung der Implementierung fachlicher Werte in objektorientierten Programmiersprachen am Beispiel des Java-Rahmenwerks JWAM

15. April 2002

Martin Fürter und
Annika Spreckelmeyer

Diplomarbeit

Arbeitsbereich Softwaretechnik
Fachbereich Informatik
Universität Hamburg

Betreuung:
Prof. Dr.-Ing. Heinz Züllighoven
Prof. Dr. Winfried Lamersdorf

Inhaltsverzeichnis

1 Einleitung 1

1.1 Aufgabenstellung 2

1.2 Inhaltsangabe 2

2 Erläuterung und Definition grundlegender Begriffe 5

2.1 Objekte und Werte 5

2.1.1 Objekte 5

2.1.2 Das Aliasproblem 6

2.1.3 Werte 8

2.1.4 Referentielle Transparenz 9

2.2 Kategorien von Typen 10

2.3 Speichermodelle 10

2.3.1 Referenzsemantik / Zeigersemantik 11

2.3.2 Speichersemantik 12

2.3.3 Wertsemantik 12

2.4 Der Begriff Fachwert 13

2.5 Zusammenfassung 14

3 Werttypen in objektorientierten Programmiersprachen 15

3.1 Typen und Wertsemantik 15

3.1.1 Primitive Typen 15

3.1.2 Benutzerdefinierte Typen 16

3.2 Festlegen von Wertebereichen 17

3.2.1 Zusicherungen durch Klasseninvarianten 17

3.2.2 Typen mit Aufzählungscharakter 19

3.3 Verhindern des Aliasproblems durch Objekte mit Speichersemantik 21

3.3.1 Java und Smalltalk: keine Objekte mit Speichersemantik 22

3.3.2 C++: Eingebettete versus referenzierte Objekte 23

3.3.3 Eiffel: expanded types 23

3.4 Sicherstellen der Unveränderlichkeit 25

3.4.1 Programmiertechniken zur Realisierung unveränderlicher Objekte 26

3.4.2 Konstante Attribute und Klassen 28

3.5 Optimierungstechniken für den Einsatz von unveränderlichen Objekten 33

3.5.1 Duale Klassen 33

3.5.2 Die Entwurfsmuster „Fliegengewicht“ und „Fabrik“ 34

3.6 Intuitive Benutzbarkeit 34

3.6.1 Überladen von Operatoren 35

3.6.2 Die Trennung von Wert und Verhalten 36

3.7 Schlußbetrachtungen 36

4 Möglichkeiten zur Unterstützung der Deklaration von Werttypen 39

4.1 Spracherweiterung durch Einbettung von Konzepten funktionaler Sprachen 39

4.1.1 Das Typkonzept in funktionalen Programmiersprachen 39

4.1.2 Die Programmiersprache Pizza 43

4.2 Typspezifikationen mittels XML-Schema 45

4.2.1 Die Beschreibungssprache XML 46

4.2.2 Verwendung von XML-Schema 47

4.2.3 Wertebereichsvalidierung mittels XML-Schema 49

4.3 Einsatz einer Klassenarchitektur 51

4.3.1 Rahmenwerke allgemein 51

4.3.2 Das JWAM-Rahmenwerk und seine Umsetzung des Fachwertkonzepts 51

4.3.3 Erfahrungen von Benutzern beim Erstellen von Fachwerten mit dem JWAM-Rahmenwerk 55

4.4 Zusammenfassung 58

5 Konzeption eines Systems zur Fachwerttypentwicklung 59

5.1 Unterstützbare Konzepte zur Deklaration von Werttypen in objektorientierten Programmiersprachen 59

5.1.1 Aufzählungstypen 60

5.1.2 Einschränkung des Wertebereichs 60

5.1.3 Undefinierter Wert 61

5.1.4 Einfaches Definieren von Konstruktoren und Verwendung von Pattern Matching 62

5.1.5 Angabe einer Stringrepräsentation 62

5.1.6 Kennzeichnung von Werttypen 63

5.1.7 Sicherung der Unveränderlichkeit 64

5.1.8 Einfache Entwicklung von Werttypen 64

5.1.9 Zusammenfassung 65

5.2 Ansatz der dynamischen XML-Verifikation 68

5.3 Entwurf eines eigenen Lösungsansatzes 69

5.3.1 Einsatz von Fachwerttypbeschreibungen 70

5.3.2 Generierung von JWAM-Fachwertklassen 70

5.3.3 Einsatz eines Werkzeugs 71

6 Aufbau von Fachwerttypbeschreibungen 73

6.1 Arten von Fachwerttypen 73

6.2 Aufbau der einfachen und komplexen Fachwerttypen 74

6.2.1 Attribute 75

6.2.2 Wertebereichseinschränkungen 77

6.2.3 Stringrepräsentation 79

6.2.4 Standardwert und spezielle Werte 83

6.3 Aufbau der aufzählbaren Fachwerttypen 85

6.3.1 Attribut 85

6.3.2 Wertebereich ... 85
6.3.3 Stringrepräsentation ... 86
6.3.4 Sortierung ... 87
6.4 Erweiterungen von Fachwerttypbeschreibungen ... 88
6.5 Aufbau von Testklassen ... 89
6.6 Zusammenfassung ... 92

7 Beschreibung des Fachwertwerkzeugs ... 93
7.1 Aufbau des Systems ... 93
7.2 Umgang mit dem Fachwertwerkzeug ... 95
7.2.1 Interaktion mit dem DomainValueTool ... 95
7.2.2 Interaktion mit dem ComplexDomainValueEditor ... 95
7.2.3 Interaktion mit dem SimpleDomainValueEditor ... 97
7.2.4 Interaktion mit dem EnumerableDomainValueEditor ... 98
7.2.5 Interaktion mit TestValueListEditor und TestValueEditor ... 99
7.2.6 Interaktion mit dem CodeGenerator ... 99
7.3 Beschreibung des Mechanismus zur Codegenerierung ... 100
7.3.1 Die Erzeugung des XML-Dokuments ... 101
7.3.2 Verabeitung von XML-Dokumenten mit Hilfe von XSLT ... 102
7.3.3 Die Codeschablonen: Grundgerüste für Fachwerttypen und Testklassen ... 103
7.3.4 Schwierigkeiten im Umgang mit XSLT-Stylesheets ... 108
7.4 Zusammenfassung ... 111

8 Auswertung ... 113

9 Zusammenfassung und Ausblick ... 117

10 Anhang ... 119
10.1 XML-Fachwerttypbeschreibung am Beispiel des komplexen Fachwerttyps „dvUhrzeit“ ... 119
10.2 Document Type Definition (DTD) zur Beschreibung von einfachen und komplexen Fachwerttypen ... 121

11 Literaturverzeichnis ... 125

Abbildungsverzeichnis

Abbildung 2-1: Das Aliasproblem....7
Abbildung 2-2: Zuweisung nach Referenzsemantik....11
Abbildung 2-3: Zuweisung nach Speichersemantik....12
Abbildung 2-4: Zuweisung nach Wertsemantik....12
Abbildung 4-1: Fachwert-Rahmenwerk....53
Abbildung 5-1: Generierung von JWAM-Fachwertklassen....71
Abbildung 5-2: Einsatz eines Werkzeugs zur Generierung von....72
Abbildung 7-1: Komponenten für die Erzeugung von JWAM-Fachwerttypen....93
Abbildung 7-2: Komponenten für die Erzeugung von Testklassen....93
Abbildung 7-3: DomainValueTool....95
Abbildung 7-4: ComplexDomainValueEditor....96
Abbildung 7-5: AttributeEditor....96
Abbildung 7-6: SimpleDomainValueEditor....97
Abbildung 7-7: EnumerableDomainValueEditor....98
Abbildung 7-8: TestValueEditor....99
Abbildung 7-9: CodeGenerator....100
Abbildung 7-10: Codegenerierung (Schritt 1)....101
Abbildung 7-11: Codegenerierung (Schritt 2)....103

Tabellenverzeichnis

Tabelle 4-1: XML-Schema-Datentypen....47
Tabelle 4-2: Einschränkungsmöglichkeiten bei XML-Schema....48
Tabelle 5-1: Unterstützte Konzepte bei verschiedenen Lösungsansätzen....65
Tabelle 6-1: Wertebereichseinschränkungen....78
Tabelle 8-1: Unterstützung der Wertkonzepte durch das Fachwertwerkzeug....115

1 Einleitung

Bei der Entwicklung komplexer Softwaresysteme müssen sowohl anwendungsnahe Gegenstände und Konzepte als auch abstrakte Größen modelliert werden. Im Bereich von Softwareanwendungen für Banken existiert beispielsweise das Konzept *Konto*, welches einen internen Zustand aufweist, der unter anderem durch das aktuelle Saldo bestimmt ist. Ein Beispiel für eine abstrakte Größe wäre eine *Kontonummer* zur Identifikation eines einzelnen bestimmten Kontos.
Anwendungsnahe Gegenstände und Konzepte, welche wie das Konzept *Konto* einen internen Zustand aufweisen, lassen sich in Programmiersprachen gut als Objekte darstellen, während sich für abstrakte Größen wie *Kontonummer* eine Wertrepräsentation anbietet. Die genaue Unterscheidung dieser Konzepte *Wert* und *Objekt* wurde in einem grundlegenden Artikel herausgearbeitet (vgl. [MAC82]): Während Objekte einen veränderlichen Zustand besitzen und von mehreren Klienten referenziert werden können, sind Werte unveränderlich und frei von Seiteneffekten. Werte können aufgrund ihrer Eigenschaften zur Korrektheit von Programmen beitragen und stellen daher ein wichtiges Modellierungskonzept dar.
Nach [MAC82] kann die Anerkennung der Wert/Objekt-Unterscheidung eine wertvolle Hilfe beim Umgang mit der Komplexität von Programmen sein. In diesem Zusammenhang wird gefordert, daß in Programmiersprachen beide Konzepte unterstützt werden. Im Widerspruch dazu ist die Unterstützung des Wertkonzepts nicht in allen Programmiersprachen ausreichend vorgesehen: oftmals fehlen ausgereifte Konzepte zur Definition benutzerdefinierter anwendungsspezifischer Werttypen. Solche Sprachmittel sind jedoch von besonderer Bedeutung, da Programmiersprachen stets nur eine begrenzte Anzahl von Basistypen bereitstellen und daher für die meisten Anwendungsfälle keine fachlich motivierten Datentypen bieten.
Wie bereits zu Beginn angedeutet, könnte beispielsweise im Bankenbereich ein Werttyp *Kontonummer* zur Identifikation eines Kontos sinnvoll sein. Da ein solcher Werttyp jedoch speziell auf den betreffenden Anwendungsbreich zugeschnitten ist, kann nicht erwartet werden, daß alle Programmiersprachen diesen speziellen Werttyp als vordefinierten Werttyp bereitstellen. Daher ist es wichtig, daß Programmiersprachen ausreichende Sprachmittel zum Definieren solcher benutzerdefinierter, anwendungsspezifischer Werttypen bereitstellen. In bezug auf solche Sprachmittel zeigt sich allerdings ein deutlicher Unterschied zwischen Programmiersprachen des funktionalen und des objektorientierten Paradigmas: während die Erstellung von Werttypen in funktionalen Sprachen trivial ist, erweist sich diese Aufgabe in objektorientierten Sprachen als schwierig.

1.1 Aufgabenstellung

Im vorigen Abschnitt wurde angedeutet, daß Werttypen in Programmiersprachen von besonderer Bedeutung sind. Im Rahmen dieser Diplomarbeit ist zu klären, ob Werttypen auch in objektorientierten Programmiersprachen definiert werden können, inwiefern dabei Schwierigkeiten auftreten können und inwiefern die Werteigenschaften solcher benutzerdefinierter Werttypen in objektorientierten Sprachen sichergestellt sind.
Ziel dieser Diplomarbeit ist es, eine Möglichkeit zu finden, die Deklaration von Werttypen in objektorientierten Sprachen zu unterstützen.
Um einen geeigneten Lösungsansatz entwickeln zu können, müssen zum einen die Konzepte herausgearbeitet werden, die im Zusammenhang mit Werttypen wichtig sind. Zum anderen werden bereits vorhandene Ansätze zur Deklaration von Typen mit Wertcharakter untersucht. Besondere Aufmerksamkeit soll dabei der Möglichkeit gewidmet werden, ein Rahmenwerk einzusetzen. In bezug auf die Problematik der Erstellung von Werttypen in objektorientierten Programmiersprachen ist insbesondere das sogenannte JWAM-Rahmenwerk von Interesse, welches am Fachbereich Informatik der Universität Hamburg entwickelt wurde; denn dieses Rahmenwerk bietet eine explizite Unterstützung für die Entwicklung benutzerdefinierter Werttypen im Umfeld der objektorientierten Sprache Java.
Es soll am Beispiel dieses Rahmenwerks gezeigt werden, wie eine werkzeugbasierte Unterstützung für die Entwicklung von Werttypen in einer objektorientierten Programmiersprache aussehen kann.

1.2 Inhaltsangabe

Um die im vorigen Abschnitt genannte Aufgabenstellung zu erfüllen, werden im Rahmen dieser Diplomarbeit die folgenden Schritte unternommen:
Zunächst wird dargestellt, wodurch sich die Konzepte *Wert* und *Objekt* unterscheiden und warum diese Unterscheidung für die Korrektheit von Programmen wichtig ist (vgl. Kapitel 2).

Daraufhin erfolgt die Untersuchung, auf welche Weise benutzerdefinierte Werttypen in objektorientierten Programmiersprachen definiert werden können, welche Schwierigkeiten dabei auftreten und welche Programmiertechniken beim Umgang mit dieser Problematik eingesetzt werden können (vgl. Kapitel 3).

Anschließend werden Lösungsansätze vorgestellt, welche Möglichkeiten zur Unterstützung von Werttypdeklarationen aufzeigen. Dabei wird unter anderem dargestellt, welche Möglichkeiten funktionale Programmiersprachen für die Erstellung von Werttypen bieten.
Des weiteren wird insbesondere auf den Vorschlag der Einführung eines Fachwertkonzepts eingegangen (vgl. [ZÜL98], [BÄU98]). Dieses Fachwertkonzept wurde in dem an der Universität Hamburg entwickelten JWAM-Rahmenwerk umgesetzt (vgl. [MÜL99]). Im Hinblick auf diese Umsetzung wird untersucht, wie sich die Arbeit

mit dem JWAM-Rahmenwerk gestaltet und ob weitergehende Hilfestellungen für die Fachwerttyp-Entwicklung mit dem JWAM-Rahmenwerk sinnvoll erscheinen (vgl. Kapitel 4).

Die in Kapitel 4 vorgestellten Ansätze werden im folgenden dahingehend betrachtet, inwiefern sie sich zur Umsetzung derjenigen Konzepte eignen, die für die Erstellung von Werttypen in objektorientierten Sprachen wünschenswert sind. Die resultierenden Erkenntnisse werden daraufhin zur Konzeption eines Systems genutzt, welches die Entwicklung von Werttypen vereinfachen soll (vgl. Kapitel 5).

Um das in Kapitel 5 entworfene System realisieren zu können, muß zunächst herausgearbeitet werden, wie Fachwerttypen aufgebaut sind und wie sich ihr Aufbau beschreiben läßt (vgl. Kapitel 6).

Daraufhin wird beschrieben, wie die im vorigen Kapitel entwickelten Fachwerttypbeschreibungen mit Hilfe eines Fachwertwerkzeugs in JWAM-Fachwerttypen umgesetzt werden können. In diesem Kapitel wird der genaue Aufbau und die Handhabung des entwickelten Systems beschrieben (vgl. Kapitel 7).

Im Anschluß an die Entwicklung dieses Werkzeugs ist in einer Auswertung zu untersuchen, inwiefern es verbesserte Möglichkeiten bereitstellt, um in einer objektorientierten Programmiersprache Werttypen zu definieren (vgl. Kapitel 8).

2 Erläuterung und Definition grundlegender Begriffe

Um die Thematik dieser Diplomarbeit in den folgenden Kapiteln besser darstellen zu können, werden im folgenden zunächst einige zentrale Begriffe und Konzepte geklärt. Dazu zählt zum einen die Unterscheidung zwischen den Konzepten *Objekt* und *Wert* und die Erläuterung der damit zusammenhängenden Begriffe *referentielle Transparenz* und *Aliasproblem*. Weiterhin wird eine Kategorisierung von Datentypen dargestellt; und es erfolgt die Erläuterung von Speichermodellen und die Einordnung des Begriffs *Fachwert*.

2.1 Objekte und Werte

Bei der programmiersprachlichen Modellierung können Elemente des zu modellierenden Systems entweder als Objekt oder als Wert realisiert werden. Beide Begriffe stehen für grundlegende Konzepte der Softwareentwicklung. Im folgenden werden diese Konzepte *Objekt* und *Wert* dargestellt, ebenso wie das *Aliasproblem* im Umgang mit Objekten und die Werteigenschaft der *referentiellen Transparenz*.

2.1.1 Objekte

Das Konzept der Objekte unterscheidet sich grundlegend vom Wertkonzept, welches in Abschnitt 2.1.3 vorgestellt wird. Die Kenntnis der Unterschiede beider Konzepte ist von großer Bedeutung, zumal in objektorientierten Programmiersprachen häufig Objekte für die Modellierung von Werten verwendet werden.
Der Begriff *Objekt* beinhaltet nach [MAC82]:

- Objekte repräsentieren Elemente des zu modellierenden Systems.
- Objekte haben einen internen Zustand. Dieser setzt sich zusammen aus der Summe der internen Eigenschaften und Attribute zu einem bestimmten Zeitpunkt.
- Objekte sind veränderlich, denn der interne Zustand eines Objektes kann sich mit der Zeit ändern. Objekte sind im Gegensatz zu Werten nicht zeitlos.
- Da Objekte nicht zeitlos sind, können sie zerstört und erzeugt werden.
- Objekte besitzen eine Identität. Die Identität eines Objekts ist unabhängig von seinen internen Eigenschaften oder Attributen, das heißt, unabhängig von seinem internen Zustand. Programmiertechnisch wird die Identität durch den Speicherort des Objekts bestimmt.
- Wenn sich Objekte verändern, bleibt ihre Identität erhalten. Ändert man beispielsweise alle Elemente einer Array-Variablen, so handelt es sich immer noch um dasselbe Objekt, da sich der Speicherort nicht ändert.
- Von Objekten können Kopien angefertigt werden, welche sich durch ihre Identität unterscheiden.
- Objekte können gemeinsam benutzt werden. Änderungen an einem gemeinsam benutzten Objekt wirken sich auf alle Klienten aus, welche dieses Objekt benutzen.

In den meisten gängigen objektorientierten Programmiersprachen werden zur Beschreibung von Objekten Klassen eingesetzt. Solche Klassen spezifizieren, welche Attribute und welche Methoden Objekte dieser Klasse besitzen sollen.
Objekte werden in der Regel mit Referenzsemantik verwendet (vgl. Abschnitt 2.3.1). Mit diesem Speichermodell ist allerdings das im folgenden beschriebene Aliasproblem verbunden.

2.1.2 Das Aliasproblem

Ähnlich wie ein und dieselbe Person unter mehreren Namen bekannt sein kann, können Objekte über mehrere Referenzen zugreifbar sein. Diese Namen beziehungsweise Referenzen werden als *Alias* bezeichnet (vgl. [LOU94]). Ein Alias für ein Objekt entsteht zur Laufzeit dadurch, daß mehreren Variablen Referenzen auf dasselbe Objekt zugewiesen werden. Das referenzierte Objekt kann dadurch von verschiedenen Seiten geändert werden. Dabei kann das Problem entstehen, daß eine Änderung über einen bestimmten Namen erfolgt, ohne zu beachten, daß sich hinter einem anderen Namen dasselbe Objekt verbirgt. Es wird also nicht offensichtlich, daß auch der andere Name nunmehr ein geändertes Objekt bezeichnet. Bezeichnen Referenzen fälschlicherweise dasselbe Objekt, so kann dies unerwünschte Änderungen am Zustand dieses Objekts zur Folge haben. Dies wird als *Aliasproblem* bezeichnet und im folgenden am Beispiel der Java-Klasse *Person* erläutert:

```
public class Person
{
   private int alter;

   public Person(int a)
   {
      alter = a;
   }

   public void aendereAlter (int neuesAlter)
   {
      alter = neuesAlter;
   }
}
```

Seien nun *p* und *mueller* zwei Variablen vom Typ *Referenz auf Person*:

```
Person p;
Person mueller;
```

Nach der Zuweisung

```
p = new Person(25);
```

bezeichnet der Name beziehungsweise die Referenz *p* ein Objekt vom Typ Person, dessen Attribut *alter* den Wert *25* hat.

Nach der folgenden Zuweisung

```
mueller = p;
```

kann dieses Personenobjekt auch durch die Referenz *mueller* angesprochen werden:

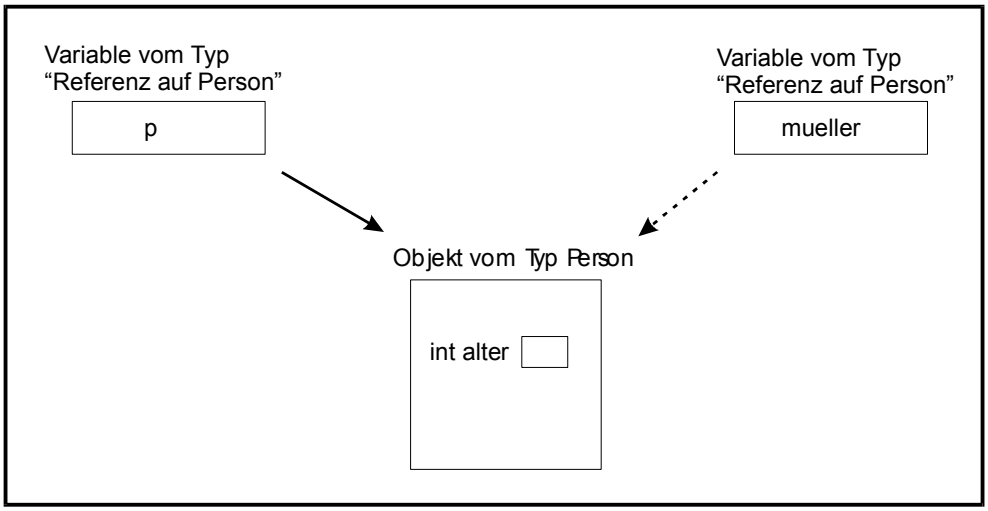

Abbildung 2-1: Das Aliasproblem

Es kann also das Alter der Person *p* geändert werden, ohne diese Referenz *p* zu verwenden:

```
mueller.aendereAlter(80);
```

Das Objekt, welches sich hinter der Referenz *p* verbirgt, wird auf diese Weise geändert, ohne daß dies im Programmcode auf den ersten Blick offensichtlich wird. Dies kann zu Fehlern führen, da sich der Programmierer stets bewußt sein muß, daß sich die Referenzen *p* und *mueller* auf dasselbe Objekt beziehen. Ansonsten könnte es im obigen Beispiel passieren, daß der Programmierer fälschlicherweise davon ausgeht, daß sich hinter der Referenz *p* weiterhin ein Person-Objekt mit dem Alter *25* verbirgt. Solche Fehler können schwer auffindbar sein, denn sie sind nicht an der Stelle zu finden, an der sie sich auswirken.

2.1.3 Werte

In diesem Abschnitt wird zunächst das Konzept der Werte vorgestellt, wobei noch nicht auf deren programmiertechnische Realisierung eingegangen wird. Dies erfolgt im weiteren Verlauf der Arbeit.
Der Begriff *Wert* beinhaltet nach [MAC82]:

- Werte sind zeitlose Abstraktionen: sie können weder erzeugt noch zerstört werden. Beispielsweise stellt die Zahl *2* das Konzept eines Paares dar, wobei von der Art des Paares abstrahiert wird. Die Zahl *2* kann nicht zerstört werden, da sie nicht an die Existenz eines bestimmten Paares gebunden ist.
- Werte haben keinen Zustand und sind daher unveränderlich.
- Werte sind referentiell transparent (vgl. Abschnitt 2.1.4).
- Werte sind applikativ: es können Funktionen auf Werte angewendet werden, um neue Werte zu erhalten, wobei die Funktionen diese Werte nicht ändern.
- Werte können nicht gezählt werden. Es gibt keine Kopien von Werten. So existiert beispielsweise das Konzept des Paares, repräsentiert durch die Zahl *2*, nur ein einziges Mal. Es macht keinen Sinn, mehrere Kopien desselben Konzepts zu unterscheiden.

In objektorientierten Programmiersprachen werden zur Darstellung von Werten häufig Objekte verwendet. Problematisch ist dabei, daß eine der wichtigsten Eigenschaften von Werten, nämlich die Unveränderlichkeit, nicht zu den Eigenschaften von Objekten gehört. In Abschnitt 3.4 wird untersucht, welche Möglichkeiten es dennoch gibt, die Unveränderlichkeit von Objekten sicherzustellen.

Solche unveränderlichen Objekte, im folgenden als Wertobjekte bezeichnet, bieten an vielen Stellen Vorteile (vgl. [BÄU98]):
Zum einen müssen Wertobjekte bei nebenläufiger Bearbeitung (*Multithreading*) nicht vom zugreifenden Prozeß reserviert werden. Da Wertobjekte unveränderlich sind, können mehrere Prozesse gleichzeitig auf ein Wertobjekt zugreifen, ohne daß Inkonsistenzen auftreten. Dies wird auch als *thread-safe* bezeichnet (vgl. [VEN98]).
Zum anderen bieten Wertobjekte den Vorteil, daß sie in verteilten Systemen ohne Synchronisationsmaßnahmen in den Adreßraum des Prozesses kopiert werden können, der dieses Wertobjekt benötigt. Dadurch werden entfernte Aufrufe auf unveränderlichen Objekten vermieden.
Diese Vorteile von Werttypen können allerdings nur dann genutzt werden, wenn Werttypen von anderen Typen eindeutig unterschieden werden können. Eine solche Kennzeichnung von Wertobjekten findet sich beispielsweise in dem CORBA Standard (ab Version 2.3), welcher eine plattform- und programmiersprachenunabhängige Umgebung für verteilte Objekte darstellt. Mit dem in diesem Standard enthaltenen Schlüsselwort *valuetype* können Objekte gekennzeichnet werden, die stets als Wert (*by value*) übergeben werden sollen (vgl. [SIE00])
Zudem tragen Wertobjekte, unabhängig von deren Kennzeichnung, zur Korrektheit von Programmen bei. Denn bei Wertobjekten kann nicht das Aliasproblem (vgl. Abschnitt 2.1.2) auftreten, wie dies bei Verwendung von veränderlichen Objekten

der Fall sein kann. Das Aliasproblem kann bei Werten deswegen nicht auftreten, weil sie unveränderlich sind. Diese Unveränderlichkeit steht im Zusammenhang mit der Eigenschaft der referentiellen Transparenz, welche im folgenden Abschnitt dargestellt wird.

2.1.4 Referentielle Transparenz

Zur Darstellung von Werten werden in der Mathematik Ausdrücke verwendet, welche auch Namen von unbekannten Größen enthalten können, wie beispielsweise der Ausdruck "*gradCelsius * 9/5 + 32*". Dabei ist *gradCelsius* ein Name, der innerhalb eines bestimmten Kontextes stets denselben Wert bezeichnet. Obwohl solche Namen als *Variablen* bezeichnet werden, gilt in der Mathematik, daß sie sich innerhalb eines bestimmten Kontextes nicht verändern. Dadurch wird die sogenannte *referentielle Transparenz* gewährleistet, auch bezeichnet als Bezugstransparenz (vgl. [BIR92]), welche wie folgt definiert wird (vgl. [MEY97]):

> Ein Ausdruck *e* ist referentiell transparent, wenn es möglich ist, jeden seiner Subausdrücke mit dessen Wert zu ersetzen, ohne den Wert von *e* zu ändern.

Sei *e* beispielsweise der Ausdruck "*seitenlänge * seitenlänge/2*", und *seitenlänge* habe den Wert *34*, so könnte statt *seitenlänge* stets *34* geschrieben werden, ohne den Wert des Ausdrucks *e* zu ändern.
Nur wenn die referentielle Transparenz gewährleistet ist, sind auch die mathematischen Grundlagen gesichert, wie beispielsweise die Gleichheit der beiden Ausdrücke "*x+x*" und "*2*x*" (vgl. [MEY97]). Im folgenden Beispiel wird *x* durch die Funktion *seitenlänge()* ersetzt:

```
int laenge = 100;

int seitenlaenge()
{
   laenge = laenge + 100;
   return laenge;
}
```

Dann würde der Ausdruck "*seitenlänge() + seitenlänge()*" den Wert *200 + 300*, also *500*, bezeichnen. Der Ausdruck "*2*seitenlänge()*" dagegen würde den Wert *2*200*, also *400*, bezeichnen. Diese unerwartete Ungleichheit der Werte der beiden Ausdrücke entsteht dadurch, daß die Funktion *seitenlänge()* als Seiteneffekt den Wert der Variablen *länge* ändert. Dadurch geht die referentielle Transparenz verloren, denn im Ausdruck "*seitenlänge() + seitenlänge()*" bezeichnen die beiden Operanden nicht denselben Wert. Solche Fehler können unter Umständen schwer auffindbar sein und könnten durch unveränderliche Objekte vermieden werden.

2.2 Kategorien von Typen

Werte und Objekte können mittels sogenannter *Datentypen* klassifiziert werden. In der Literatur werden Datentypen klassischerweise als Einheit aus Wertebereich und Operationen betrachtet (vgl. [HOA72]).
Im folgenden wird eine Kategorisierung von Datentypen aufgestellt (vgl. [MEE94]), die im weiteren Verlauf der Arbeit dazu genutzt werden kann, verschiedene Datentypen aufgrund ihres Aufbaus voneinander abzugrenzen.

- **Primitive Typen,** auch **elementare Typen** genannt, bezeichnen grundlegende Datentypen, die nicht weiter reduziert werden können. Hierzu können Typen wie *Boolean*, *State*, *Enumerated*, *Character*, *Ordinal*, *Date-and-time*, *Integer*, *Rational*, *Scaled*, *Real*, *Complex* und *Void* gezählt werden (vgl. [MEE94]). Primitive Typen bilden die Grundbausteine für die Datentypen der im folgenden aufgeführten Kategorien.

- Ein **Subtyp** ist ein Typ, der einen Basistyp in bestimmter Weise ändert, beispielsweise durch die Auswahl oder den Ausschluß von Werten, möglich ist aber auch eine Erweiterung des Wertebereichs.

- **Generierte Typen** sind Typen, die mittels Zeiger-, Funktions- oder Auswahldeklaration aus anderen Typen abgeleitet wurden.

- Ein **aggregierter Typ** besteht aus mehreren Komponenten, das heißt, sein Wertebereich ergibt sich aus dem kombinierten Wertebereich dieser Komponenten. Die Komponenten können dabei beliebige Typen haben, also auch wiederum aggregierte Datentypen. Der Wertebereich eines aggregierten Typs kann ähnlich wie bei den Subtypen durch Eigenschaften und Einschränkungen angepaßt werden, welche für die Komponenten festgelegt werden. Diese Eigenschaften und Einschränkungen können sich auf einzelne Komponenten oder auf Kombinationen von Komponenten beziehen.

Über diese Kategorisierung hinaus werden benutzerdefinierte und eingebaute Typen unterschieden. Während eingebaute Typen im Sprachumfang einer Programmiersprache enthalten sind, werden benutzerdefinierte Typen vom Entwickler selbst definiert. Bei den benutzerdefinierten Typen handelt es sich meist um aggregierte Datentypen, die in objektorientierten Programmiersprachen durch Klassen realisiert werden.
Die Semantik der verschiedenen Datentypen steht in Zusammenhang mit der Art der Speicherung ihrer Exemplare: Datentypen können unterteilt werden in Typen mit Wertsemantik (Werttypen) und Typen mit Referenzsemantik (Objekttypen). Dies wird im folgenden Abschnitt näher erläutert.

2.3 Speichermodelle

Der Umgang mit Variablen wird davon beeinflußt, welches Speichermodell zu dem Typ der Variablen gehört. Unterschieden werden dabei *Referenzsemantik*, auch

Zeigersemantik genannt, sowie *Speichersemantik* und *Wertsemantik*. Um die Unterschiede dieser drei Speichermodelle zu verdeutlichen, wird das folgende Beispiel verwendet (vgl. [STA95]):
Seien *A* und *B* zwei Variablen mit den Anfangswerten *2* und *3*, und es erfolge eine Zuweisung der Form "*A = B*".

2.3.1 Referenzsemantik / Zeigersemantik

Referenz- bzw. Zeigersemantik wird im Zusammenhang mit Objekttypen verwendet. Dabei bezieht sich die Zuweisung "*A = B*" auf die Speicherstellen und führt daher dazu, daß *A* dieselbe Speicherstelle bezeichnet wie *B* (vgl. Abbildung 2-2):

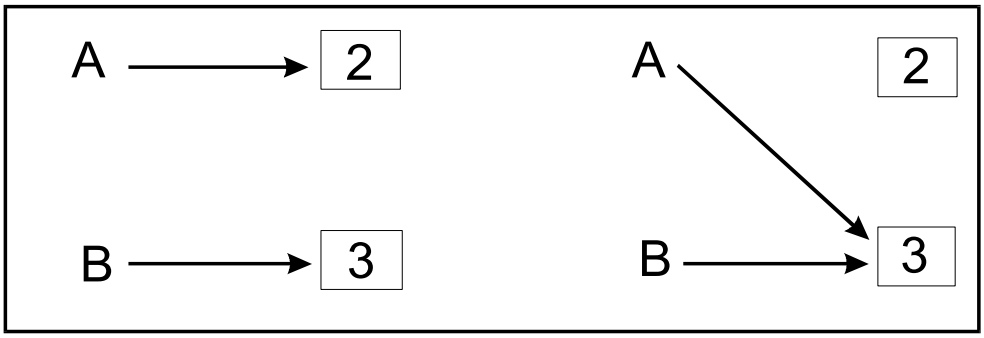

Abbildung 2-2: Zuweisung nach Referenzsemantik

Nach dieser Zuweisung wirken sich Änderungen an *A* auch auf *B* aus: nach einer weiteren Zuweisung wie beispielsweise "*A = 14*" beinhaltet der Wert der gemeinsamen Speicherstelle von *A* und *B* den Wert *14*, und der ursprüngliche Wert von *B* ist verloren, obwohl im Zusammenhang mit *B* keine verändernde Operation durchgeführt wurde. Die Referenzsemantik kann sich daher problematisch auswirken, falls eine solche Änderung unbeabsichtigt erfolgt. Ein solche Situation wird als Aliasproblem bezeichnet (vgl. Abschnitt 2.1.2). Es gibt jedoch durchaus Situationen, in denen gemeinsam referenzierte Objekte zum Informationsaustausch genutzt werden, wie beispielsweise bei der Modellierung einer Kartenvorverkaufstelle. Dabei könnte die Liste mit den bereits verkauften Sitzplätzen einer Theatervorstellung durch ein Objekt repräsentiert werden, welches von allen Theaterkassen referenziert wird. Wird an einer Theaterkasse eine weitere Karte verkauft, so muß dies für alle anderen Verkaufsstellen sichtbar werden, damit es nicht zu Doppelbelegungen kommt.
Allgemein lassen sich durch Referenzsemantik gut Situationen modellieren, bei denen mehrere Objekte ein anderes Objekt gemeinsam kennen und benutzen. Diese Beziehung wird auch als Aggregation bezeichnet (vgl. [WAH98]). Liegt hingegen der Fall vor, daß ein Objekt *A* nur von einem einzigen anderen Objekt *B* benutzt wird, so entspricht dies eher dem physikalischen Enthaltensein von Objekt *A* in Objekt *B,* auch als Komposition bezeichnet (vgl. WAH98]). In einem solchen Fall sollte zur Modellierung die im folgenden Abschnitt beschriebene Speichersemantik verwendet werden.

2.3.2 Speichersemantik

Neben der Referenz- bzw. Zeigersemantik können Objekttypen auch der Speichersemantik folgen. Wird Speichersemantik verwendet, so bezieht sich die Zuweisung "*A=B*" auf die Inhalte der Speicherstellen und führt daher dazu, daß in der von A bezeichneten Speicherstelle eine Kopie des Inhalts der von B bezeichneten Speicherstelle abgelegt wird:

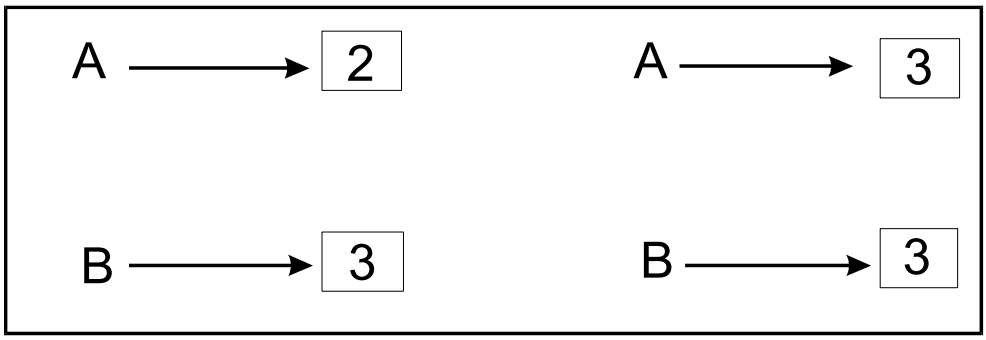

Abbildung 2-3: Zuweisung nach Speichersemantik

Wird nun eine Änderung am Speicherinhalt von *A* durchgeführt, so ändert sich der Inhalt von *B* nicht, da ja nur die Kopie geändert wird. Nach einer Zuweisung der Form "*A = 14*" würde *B* daher weiterhin den Wert *3* bezeichnen.
Bei Verwendung von Speichersemantik kann also im Gegensatz zur Referenzsemantik kein Aliasproblem auftreten. Allerdings können auch keine Objekte zur Kommunikation verwendet werden, und zudem führt die Anfertigung von Kopien zu einem erhöhten Speicherplatzbedarf.

2.3.3 Wertsemantik

Während sich die Speichersemantik auf die Inhalte von Speicherstellen bezieht, wird bei Wertsemantik von dem konkreten Ort der Speicherung abstrahiert. Bei Wertsemantik werden die Operanden von Funktionen als zeit- und ortslose Werte betrachtet, welche sich nicht verändern können. Die Zuweisung *A = B* würde sich daher folgendermaßen auswirken:

Abbildung 2-4: Zuweisung nach Wertsemantik

Auf den ersten Blick scheint dies der Zeigersemantik zu ähneln, jedoch besteht ein entscheidender Unterschied: In diesem Fall kann der Wert *3* nicht verändert werden und es kann daher kein Aliasproblem auftreten. Wenn bei der Zeigersemantik die Unveränderlichkeit der Inhalte der referenzierten Speicherstellen gewährleistet ist, gleicht das Verhalten der Zeigersemantik dem der Wertsemantik.
Der Unterschied zur Speichersemantik besteht darin, daß bei der Zuweisung "*A=B*" keine Kopie des Wertes *3* erstellt wird. Während bei der Speichersemantik zwei identische Wertkopien in zwei verschiedenen Speicherstellen existieren, verwenden bei der Wertsemantik beide Variablen denselben Wert, wobei von dessen Speicherort abstrahiert wird.
Sowohl bei Speicher- als auch bei Wertsemantik kann kein Aliasproblem auftreten: dies wird bei Speichersemantik durch die Anfertigung von Kopien und bei Wertsemantik durch die Unveränderlichkeit der Werte erreicht.

2.4 Der Begriff Fachwert

Objektorientierte Programmiersprachen bieten zwar einige Standarddatentypen, welche jedoch zur Modellierung von fachlichen Werten nicht ausreichen. So könnte man zwar eine Kontonummer als Integer darstellen, jedoch darf sie im Gegensatz zu diesem Standardtyp keinen negativen Wert annehmen. Zudem gelten die arithmetischen Operationen für Integer-Werte nicht unbedingt auch für Kontonummern: es ist beispielsweise nicht sinnvoll, zwei Kontonummern zu multiplizieren. Um die Menge der Standard-Datentypen dem Anwendungskontext entsprechend zu erweitern, werden daher sogenannte Fachwerte erstellt.
Diese werden in objektorientierten Sprachen als Klassen realisiert, allerdings muß dabei darauf geachtet werden, daß sich die Exemplare dieser Klassen wie Werte verhalten, das heißt, sie müssen der Wertsemantik folgen.

Die Definition des Begriffes *Fachwert* lautet nach [ZÜL98] :

- Fachwerte sind benutzerdefinierte Werte, welche im jeweiligen Anwendungskontext gebraucht werden.
- Fachwerte sind Werttypen mit definierter Wertemenge und festgelegten Operationen, deren innere Repräsentation verborgen bleibt.

Im weiteren Verlauf dieser Arbeit werden die Werttypen des zweiten Punkts der oben angeführten Definition als *Fachwerttypen* bezeichnet, während der Begriff *Fachwert* für ein Exemplar eines solchen Fachwerttyps verwendet wird.

Entsprechend des zweiten Punkts der Definition ist es mittels Fachwerttypen möglich, neue benutzerdefinierte Werttypen einzuführen und deren Operationen vorzugeben. Fachwerttypen erlauben eine detailliertere Typprüfung sowie einen kontrollierten Umgang mit den Exemplaren dieser Typen und tragen daher zur

Programmkorrektheit bei. Beispielsweise kann ein neuer Typ *Kontonummer* definiert werden, dessen Exemplare im Gegensatz zu denen des primitiven Typs *Integer* nur Operationen zulassen, die fachlich für Kontonummern von Bedeutung sind.
Zudem können Fachwerttypen eingesetzt werden, um eine feinere Typisierung zu erreichen. Dies dient dazu, Programme leichter lesbar zu machen. Beispielsweise ist es für die Verständlichkeit von Programmtexten hilfreich, wenn statt des Datentyps *String* genauere Typbezeichnungen eingesetzt werden, wie etwa *Vorname* oder *Kontoinhaber*.

2.5 Zusammenfassung

In diesem Kapitel wurde zunächst die Unterscheidung zwischen den Konzepten *Wert* und *Objekt* herausgearbeitet: Objekte haben einen zur Laufzeit änderbaren Zustand, während Werte zeitlose Abstraktionen darstellen und keinen Zustand besitzen.
In Zusammenhang mit den beiden Konzepten wurden verschiedene Speichermodelle erläutert. Es wurde dargestellt, daß Objekte der Referenzsemantik folgen, so daß bei ihrer Verwendung das Aliasproblem auftreten kann; Werte hingegen folgen der Wertsemantik und besitzen die Eigenschaft der referentiellen Transparenz.
Das Modellieren mit Werttypen kann daher zur Korrektheit von Programmen beitragen. Somit ist es bei der Entwicklung von Softwaresystemen wichtig, für das jeweilige Fachgebiet Werttypen erstellen zu können. Diese werden Fachwerte beziehungsweise Fachwerttypen genannt.
Im folgenden Kapitel ist zu untersuchen, welche Möglichkeiten objektorientierte Programmiersprachen zur Definition solcher Werttypen bieten.

3 Werttypen in objektorientierten Programmiersprachen

Objektorientierte Programmiersprachen gehören zu den imperativen Sprachen. Die imperativen Sprachen zeichnen sich unter anderem durch die Existenz veränderbarer Variablen aus. In diesen Sprachen ist es daher möglich, Prozeduren zu definieren, welche den Zustand von globalen Variablen verändern können.
Mittels solcher Prozeduren läßt sich das in Abschnitt 2.1.1 beschriebene Konzept der Objekte implementieren, welche die Basis der Modellierung in objektorientierten Sprachen bilden. Wie in Abschnitt 2.1.3 beschrieben, ist es jedoch wichtig, neben Objekten auch mit Werten modellieren zu können. Daher wird im folgenden untersucht, inwiefern Sprachkonstrukte von objektorientierten Programmiersprachen zur Realisierung von Werttypen beitragen können (vgl. Abschnitt 3.1 bis 3.4). Des weiteren werden Programmiertechniken vorgestellt, welche bei der Realisierung der Unveränderlichkeit von Werten den Speicherplatzverbrauch optimieren können (vgl. Abschnitt 3.5). Abschließend wird darauf eingegangen, welche Techniken zur intuitiven Benutzbarkeit benutzerdefinierter Werte beitragen können (vgl. Abschnitt 3.6).
Aufgrund der großen Menge unterschiedlicher objektorientierter Programmiersprachen ist es erforderlich, sich bei der Untersuchung der angebotenen Sprachkonzepte auf einige Sprachen zu beschränken. Bei den betrachteten Sprachen handelt es sich um Java, C++ und Eiffel. Das Hauptaugenmerk liegt dabei auf Java, weil das JWAM-Rahmenwerk in dieser Sprache implementiert ist (vgl. Abschnitt 4.3.2). C++ wird aufgrund der großen Verbreitung als relevant angesehen und Eiffel wegen dessen sauberer Konzeption objektorientierter Mechanismen. Am Rande wird außerdem noch auf die Sprache Smalltalk eingegangen, welche deswegen interessant erscheint, weil in dieser Sprache alle Einheiten Objekte sind. Beispielsweise werden in Smalltalk auch Klassen als Objekte betrachtet.
Soweit nicht anders angegeben, basieren die folgenden Betrachtungen zu Java auf [GOS96] und die Betrachtungen im Zusammenhang mit C++ auf [STR00]. Für Eiffel bildet [MEY97] die Grundlage und für Smalltalk [GOL83].

3.1 Typen und Wertsemantik

In diesem Abschnitt wird dargestellt, welche Arten von Typen in objektorientierten Sprachen angeboten werden und inwiefern diese Typen Wertsemantik aufweisen. Dabei wird unterschieden zwischen Typen, die von den Sprachen bereitgestellt werden (vgl. Abschnitt 3.1.1) und solchen, die selbst definiert werden können (vgl. Abschnitt 3.1.2).

3.1.1 Primitive Typen

Objektorientierte Programmiersprachen bieten in der Regel eine Auswahl von primitiven Typen an. Diese primitiven Typen sind je nach Programmierspache auf unterschiedliche Weise definiert. Die Typinformationen können zum einen durch eine Klassendefinition gegeben sein. Zum anderen können für einige primitive Typen

Schlüsselwörter existieren, deren Typinformation im eingesetzten Compiler verankert sind. Die Exemplare der mit Hilfe von Schlüsselwörtern realisierten Typen weisen generell Wertsemantik auf. Solche Schlüsselwörter sind oft in sogenannten *hybriden Sprachen* zu finden, welche aus einer nicht objektorientierten Sprache entwickelt wurden. Ein Beispiel dafür ist die Sprache C++: sie bietet Schlüsselwörter für primitive Typen wie Ganzzahlen, Wahrheitswerte, Gleitkommazahlen und Zeichen.
Die Sprache Java weist dieselben Schlüsselwörter für primitive Typen auf wie C++, da Java syntaktisch stark an C++ angelehnt ist. Darüber hinaus existieren in Java für diese primitiven Typen sogenannte *Wrapper-Klassen*: Diese Klassen speichern lediglich einen Wert des zugrundeliegenden Typs und stellen Methoden zum Umgang mit diesem Wert bereit (vgl. [GOS96]). Diese Wrapper-Klassen bewahren dadurch die Wertsemantik, daß sie keine ändernden Operationen besitzen. Zusätzlich existiert in Java eine Klasse *String*, die eine Zeichenkette speichert und ebenfalls unveränderlich ist. Für den Zeichenkettentyp gibt es kein eigenes Schlüsselwort. Daran zeigt sich, daß in Java nicht nur Schlüsselwörter zur Definition primitiver Typen eingesetzt werden, sondern auch unveränderliche Klassen. Dasselbe gilt für die Sprache C++.
In Eiffel und Smalltalk werden alle Typen durch Klassen definiert. Die primitiven Typen sind dabei mit Hilfe von Klassen ohne veränderliche Operationen implementiert und werden in Standardklassenbibliotheken bereitgestellt. Unterschiede bestehen bei diesen beiden Sprachen durch die verwendeten Speichermodelle: in Eiffel werden primitive Typen wie *Integer*, *Real*, *Double*, *Boolean* und *Character* mittels sogenannter *expanded types* definiert, deren Exemplare Speichersemantik aufweisen (vgl. Abschnitt 3.3.3). In Smalltalk hingegen besitzen Exemplare primitiver Typen Referenzsemantik, ebenso wie die anderen Typen der Sprache.
Zusammenfassend läßt sich sagen, daß alle untersuchten Programmiersprachen nur eine begrenzte Anzahl primitiver Werttypen bereitstellen. Da diese in vielen Fällen nicht ausreichen, ist die Möglichkeit zur Erzeugung neuer, benutzerdefinierter Werttypen von großer Bedeutung. Im folgenden soll betrachtet werden, wie in den verschiedenen Programmiersprachen neue Typen erstellt werden können und ob es die Möglichkeit gibt, neue Werttypen einzuführen.

3.1.2 Benutzerdefinierte Typen

In den meisten objektorientierten Programmiersprachen steht für die Definition von neuen benutzerdefinierte Typen das Klassenkonstrukt bereit. Durch eine Klasse wird der Aufbau einer bestimmten Objektart beschrieben. Sie umfaßt zum einen die Attribute, welche die Eigenschaften repräsentieren, die auf alle Objekte dieser Klasse zutreffen; die Belegung dieser Attribute mit konkreten Werten stellt dann den Zustand eines einzelnen Objekts dar. Zum anderen werden bei einer Klassendefinition Methoden definiert, mittels derer auf dem Zustand der einzelnen Objekte operiert werden kann.
Objektorientierte Sprachen sind vom Design her auf die Modellierung mit veränderlichen Objekten ausgerichtet. Daher sind in Klassendefinitionen nicht nur Funktionen zu finden, die den Zustand von Exemplaren des Typs unverändert lassen, sondern auch Prozeduren, die den Zustand von Exemplaren des Typs ändern.

Da die objektorientierten Sprachen auf die Möglichkeit von Zustandsänderungen fokussieren, wird in diesen Sprachen die Modellierung mit unveränderlichen Typen vernachlässigt: Es ist nicht möglich, in Klassen das Vorhandensein von Prozeduren zu verhindern. In objektorientierten Programmiersprachen wird dementsprechend kein spezielles Schlüsselwort bereitgestellt, welches für bestimmte benutzerdefinierte Typen Zustandsänderungen der Exemplare verhindern könnte. Das bedeutet, in objektorientierten Sprachen ist es nicht möglich, einen benutzerdefinierten Typ als Werttypen zu kennzeichnen und gleichzeitig durch diese Kennzeichnung dessen Wertsemantik sicherzustellen.
Bei der Definition von Klassen können allerdings verschiedene Techniken angewendet werden, um die Exemplare dieser Klassen Werte repräsentieren zu lassen. Diese Ansätze werden in den folgenden Abschnitten dargestellt.

3.2 Festlegen von Wertebereichen

Ein Typ umfaßt eine Menge von Werten sowie Operationen, die auf diesen Werten ausgeführt werden können. Bei den primitiven Typen ist der Wertebereich durch die Hardware oder die Implementation des Compilers vorgegeben. Wird beispielsweise eine Ganzzahl (*Integer*) von einem Compiler mit *32 bits* implementiert, können an eine Variable vom Typ *Integer* Werte zwischen $-2^{31}-1$ und 2^{31} zugewiesen werden.
Bei benutzerdefinierten Typen ist der Wertebereich in der Regel das Kreuzprodukt der Wertebereiche der Attribute. Je nach Anwendung kann eine Eingrenzung dieses Wertebereichs sinnvoll sein. Wird beispielsweise ein Typ *Uhrzeit* mit zwei Integerfeldern für Stunden und Minuten definiert, so sollte der Wertebereich der Stundenzahl zwischen *0* und *23* sowie der der Minutenzahl zwischen *0* und *59* liegen. Solche Einschränkungen können in Eiffel durch die sogenannten Klasseninvarianten festgelegt werden (vgl. Abschnitt 3.2.1).
Einen Spezialfall stellen solche Wertebereiche dar, deren Elemente explizit aufgeführt werden können. Typen mit solchen Wertebereichen werden Aufzählungstypen genannt und werden in einigen Sprachen explizit durch Sprachmittel unterstützt (vgl. Abschnitt 3.2.2).

3.2.1 Zusicherungen durch Klasseninvarianten

Ein wesentliches Merkmal der Sprache Eiffel ist das sogenannte *Vertragsmodell* (vgl. [MEY97]), welches in keiner der anderen untersuchten Sprachen vorhanden ist. Mittels des Vertragsmodells wird die Korrektheit von Programmen zur Laufzeit geprüft. Ein Vertrag ist eine formale Vereinbarung zwischen einer Klasse und ihren Klienten. Ein solcher Vertrag wird anhand von Vor- und Nachbedingungen für die Methoden der Klasse definiert. Die Vorbedingungen geben an, welche Eigenschaften vor dem Ausführen einer Methode gelten müssen. Gilt eine dieser Bedingungen nicht, so hat der Klient den Vertrag verletzt. Sofern alle Vorbedingungen erfüllt sind, muß eine Methode sicherstellen, daß nach ihrer Abarbeitung die Nachbedingungen erfüllt sind, sonst hat die Klasse den Vertrag verletzt.

Zur Veranschaulichung, wie das Vertragsmodell eingesetzt werden kann, soll das nachfolgende Beispiel dienen:

```
class UHRZEIT
feature
  setze_minuten (min: INTEGER) is
    require
      gueltige_mintenzahl: ist_minutenzahl_gueltig(min)
    do
      minuten := min
    ensure
      minuten_gesetzt: min = minuten
      gueltige_minutenanzahl: ist_minutenanzahl_gueltig(minuten)
    end

  springe_minute_weiter is
    require
      gueltige_mintenzahl: ist_minutenzahl_gueltig(minuten)
    do
      minuten := (minuten+1)\\AnzahlMinuten
    ensure
      gueltige_minutenzahl: ist_minutenzahl_gueltig(minuten)
    end

  ist_minutenzahl_gueltig(min: INTEGER): BOOLEAN is
    do
      Result :=  min >= KleinsteGueltigeMinutenzahl and
                 min <= GroessteGueltigeMinutenzahl
    end

... -- weitere features

end -- class UHRZEIT
```

Eine Vorbedingung für das Setzen der Minutenzahl ist hier beispielsweise, daß der übergebene Parameter eine gültige Minutenzahl bezeichnet. Der Wertebereich des Attributs *minutenzahl* wird in diesem Beispiel vor und nach jedem Methodenaufruf geprüft, da sichergestellt werden muß, daß *minutenzahl* durch die Abarbeitung der Methode keinen ungültigen Wert erhalten hat.
Um solche Bedingungen zu definieren, die übergreifend für alle öffentlichen Methoden eines Exemplars einer Klasse gültig sein müssen, können in Eiffel *Klasseninvarianten* angegeben werden. Klasseninvarianten müssen für alle öffentlichen Methoden sowohl als Vorbedingung als auch als Nachbedingung gelten. Durch das Spezifizieren von Klasseninvarianten müssen die Eigenschaften, die immer gelten sollen, nicht mehr als Vor- und Nachbedingung jeder einzelnen Methode eingetragen werden. Es genügt, die Invarianten am Ende der Klassendeklaration anzugeben. Im obigen Beispiel kann daher die Bedingung *ist_minutenzahl_gültig* aus den Vor- und Nachbedingungen der Methoden gelöscht und statt dessen als Klasseninvariante realisiert werden, indem folgendes am Ende der Klassendefinition eingefügt wird:

```
invariant ist_minutenzahl_gültig(minuten)
```

Mittels Invarianten kann also gewährleistet werden, daß bei einer Klasse, die eine Uhrzeit repräsentiert, das Attribut *minutenzahl* nur Werte zwischen *0* und *59* annimmt. Das bedeutet, daß durch die Benutzung von Invarianten der Wertebereich von einzelnen Attributen festgelegt werden kann. Damit kann beispielsweise ein Integerwert auf einen bestimmten Wertebereich beschränkt werden. Dies ist ein häufiger Anwendungsfall von benutzerdefinierten Werten, etwa zur Definition von Prozentzahlen, die nur Werte zwischen *0* und *100* annehmen dürfen, oder für Typen, die Quantitäten angeben und daher nicht negativ sein dürfen.
Ein Problem der Klasseninvarianten stellt zudem ihr Aufgabenbereich dar: Generell sind Invarianten nur zur Sicherstellung der Programmkorrektheit gedacht und nicht als Modellierungsmittel zur Modellierung von Werten. Das heißt, daß durch Klasseninvarianten nicht garantiert werden kann, daß ein bestimmter Wertebereich eingehalten wird. Durch das Prüfen einer Klasseninvariante oder einer Nachbedingung kann lediglich zur Laufzeit das Auffinden von Fehlern im Programm erleichtert werden, indem bei einer Wertebereichsverletzung mit einer Ausnahme (*Exception*) reagiert wird.

3.2.2 Typen mit Aufzählungscharakter

Aufzählungstypen stellen innerhalb der benutzerdefinierten Typen einen Spezialfall dar. Die Wertemenge von Aufzählungstypen wird explizit angegeben. Beispielsweise könnte ein Aufzählungstyp *Primärfarbe* die Wertemenge {*rot, grün, blau*} haben. Zudem gehören nach der Definition des Typbegriffs zu einem Aufzählungstyp die Menge der Operationen, welche auf diesem explizit angegebenen Wertebereich ausgeführt werden können.
Ein Vorteil von Aufzählungstypen liegt darin, daß sie zur Vermeidung von Codierungen beitragen, wodurch die Lesbarkeit von Programmen verbessert wird (vgl. [MEY97]). Die Schreibweise "*farbe = rot*" ist beispielsweise leichter verständlich als "*farbe = 0*", denn bei dieser Codierung der Farben in Zahlen muß dem Leser des Programmtextes immer bewußt sein, welche Zahl für welche Farbe steht. Ein weiterer Vorteil von Aufzählungstypen besteht darin, daß sich Variablen eines Aufzählungstyps effizient speichern lassen, nämlich durch eine Ganzzahl: Da jeder Wert eines Aufzählungstyps durch seine Position in der Aufzählung gekennzeichnet ist, braucht eine Variable eines Aufzählungstyps nur diese Position zu speichern.
Im Gegensatz zu Java und Smalltalk, die keine spezielle Unterstützung für Werte mit Aufzählungscharakter aufweisen, bietet C++ die Möglichkeit, fachliche Werte mit Aufzählungscharakter mit Hilfe des Schlüsselwortes *enum* zu definieren. In Eiffel kann für diesen Zweck das Schlüsselwort *unique* verwendet werden. Diese Schlüsselwörter werden im folgenden daraufhin untersucht, ob sie zur Definition von echten Aufzählungstypen verwendet werden können.

3.2.2.1 C++: Das Schlüsselwort enum

Mit Hilfe des Schlüsselwortes *enum* können in C++ beliebigen Bezeichnern statisch Integerwerte zugeordnet werden. Eine Aufzählung wird dabei folgendermaßen definiert:

```
enum aufzählungsname {Bezeichner1, Bezeichner2,...,
                      BezeichnerN};
```

Standardmäßig werden den Bezeichnern in diesem Fall die ganzen Zahlen beginnend mit *0* zugewiesen. Alternativ können den Bezeichnern aber auch beliebige ganzzahlige Werte direkt bei der Initialisierung zugewiesen werden. Wird einem der Bezeichner kein spezieller Wert zugewiesen, so erhält er den ganzzahligen Nachfolger jenes Wertes, welcher dem in der Aufzählung vorherigen Bezeichner zugeordnet wurde. In der Aufzählung

```
enum aufzählungsname{Bezeichner1 = 14, Bezeichner2,
                     Bezeichner3 = 20};
```

würde das Element *Bezeichner2* daher dem Wert *15* entsprechen.
Der Name der Aufzählung, hier *aufzählungsname*, bezeichnet einen neuen benutzerdefinierten Typ mit Wertsemantik, denn bei Aufzählungen kann auf die einzelnen Elemente nicht via Adresse zugegriffen werden (vgl. [AND98]). Zur Erläuterung der Bedeutung des Schlüsselwortes *enum* für die Implementation von Werten diene folgendes Beispiel einer Aufzählung von Wochentagen:

```
enum wochentag {montag, dienstag, mittwoch, donnerstag,
                freitag, samstag, sonntag};
```

Wird im folgenden eine Variable des Typs *wochentag* definiert, so kann dieser Variablen eines der Aufzählungselemente zugewiesen werden.:

```
wochentag meinWochentag;
meinWochentag = montag;
...
```

Einer Variablen mit Aufzählungstyp kann zur Laufzeit dynamisch ein anderer Wert aus der Aufzählung zugewiesen werden:

```
...
meinWochentag = dienstag;
...
```

Die Werte der einzelnen Aufzählungselemente dagegen sind statisch und bleiben konstant: *montag* ist in diesem Beispiel eine Konstante mit dem Wert *0*, *dienstag* eine Konstante mit dem Wert *1* und so weiter. Dadurch erfüllt eine mit Hilfe von *enum* deklarierte Aufzählung die wichtige Werteigenschaft der Unveränderlichkeit. Allerdings können für Aufzählungstypen in C++ keine beliebigen Methoden definiert werden. Auf die Werte eines Aufzählungstyps können lediglich die Integerope-

ratoren angewendet werden, deren Verhalten vom Entwickler durch Überladen angepaßt werden kann (vgl. Abschnitt 3.6.1).
Es ist zu beachten, daß Integeroperationen auf Aufzählungstypen oftmals nicht sinnvoll sind. Beispielsweise ergibt die Addition zweier Werte des Typs *wochentag* ("*montag* + *freitag*") auf fachlicher Ebene keinen Sinn.
Dagegen könnte es eventuell wünschenswert sein, auf Werte des Typs *wochentag* eine Methode der Art *gibTageBis(wochentag tag2)* anwenden zu können. Eine solche Methode kann auf Aufzählungstypen in C++ jedoch nicht definiert werden. Diese fehlende Möglichkeit zum Definieren fachlicher Operationen stehen im Widerspruch zur Definition des Typbegriffs, wonach zu einem Typ auch die auf der Wertemenge ausführbaren Operationen gehören. Daher ist das Schlüsselwort *enum* in C++ für die Implementation von Werten mit Aufzählungscharakter zwar in Ansätzen geeignet, jedoch nicht zur Definition echter Aufzählungstypen.

3.2.2.2 Eiffel: das Schlüsselwort **unique**

In Eiffel existiert die Möglichkeit, mittels des *unique*-Schlüsselworts Integerkonstanten zu definieren, die dann als Werte benutzt werden können:

```
Rot, Grün, Blau: INTEGER is unique
```

Das *unique*-Schlüsselwort stellt dabei sicher, daß den Konstanten unterschiedliche Integerwerte zugewiesen werden. Welche Werte dies sind, wird allerdings nicht zugesichert, außer daß gilt: "*Rot* < *Grün* < *Blau*".
Durch eine solche Deklaration ist zudem noch kein Typ eingeführt worden. Eine Variable *meineFarbe* müßte vom Typ *Integer* sein.

```
meineFarbe: Integer
meineFabe = Rot
```

Da zuvor die Konstanten *Rot, Grün* und *Blau* deklariert wurden, ist nun die Zuweisung "*meineFarbe* = *Rot*" möglich geworden. Da der Typ von *meineFarbe* allerdings *Integer* ist, wäre genauso "*meineFarbe* = *4711*" oder "*meineFarbe* = *Rot* + *1*" möglich gewesen. Das widerspricht jedoch dem intuitiven Umgang mit Farbwerten. Es bleibt daher festzuhalten, daß es in Eiffel kein eigenes Konstrukt gibt, um Aufzählungstypen zu definieren, so daß Aufzählungstypen im Bedarfsfall durch Klassen nachgebildet werden müssen.

3.3 Verhindern des Aliasproblems durch Objekte mit Speichersemantik

Ein schwerwiegendes Problem bei der Implementierung von Werten mittels Objekten ist, daß Objekte üblicherweise Referenzsemantik aufweisen. Wenn die referenzierten Objekte veränderlich sind, kann dadurch das Aliasproblem auftreten (vgl. Abschnitt 2.1.2). Dieses Aliasproblem kann vermieden werden, wenn Objekte nach Speichersemantik verwendet werden, also bei Zuweisungen eine Kopie der zugewiesenen Speicherstelle erstellt wird (vgl. Abschnitt 2.3.2).

Die Speichersemantik kann im Zusammenhang mit benutzerdefinierten Werten auf zweierlei Weise eingesetzt werden:

1.) Durch die Implementierung von Wertobjekten mit Speichersemantik wird sichergestellt, daß diese Wertobjekte bei Zuweisungen an Variablen kopiert werden. Dies hat zur Folge, daß ein Objekt stets nur einer einzigen Variablen zugeordnet ist. Andere Variablen sind von dieser Änderung nicht betroffen, daher besteht kein Aliasproblem. Allerdings wird durch die Verwendung der Speichersemantik nicht die referentielle Transparenz gewährleistet, wenn die Klasse des Objekts verändernde Operationen erlaubt.

2.) Es kann möglich sein, daß Wertobjekte Attribute eines Objekttyps haben. Dies ist beispielsweise der Fall, wenn ein Objekt eines vorgegebenen veränderlichen Typs unveränderlich gemacht werden soll, indem es in einem unveränderlichen Werttyp gekapselt wird.
In diesem Zusammenhang ist es sinnvoll, daß diese prinzipiell veränderlichen Attribute Speichersemantik aufweisen. Dann wird durch automatisches Kopieren sichergestellt, daß es keine Referenzen geben kann, die von außerhalb auf diese veränderlichen Objekte zeigen. Das hat zur Folge, daß die Attribute nicht von außerhalb geändert werden können. Der Entwickler muß sich dann nicht mehr selbst um das Klonen dieser Objekte kümmern, was ansonsten notwendig wäre (vgl. Abschnitt 3.4.1).

In den folgenden Abschnitten wird dargestellt, welche Möglichkeiten von den untersuchten Sprachen bereitgestellt werden, um solche Objekte mit Speichersemantik zu implementieren.

3.3.1 Java und Smalltalk: keine Objekte mit Speichersemantik

Java sieht bei der Speicherung von Variablen eine Zweiteilung vor (vgl. [HEN97]): eingebaute, durch Schlüsselwörter gegebene Typen sind Werttypen und werden grundsätzlich direkt gespeichert, das heißt, sie folgen der Speichersemantik. Exemplare von Klassentypen weisen hingegen stets Referenzsemantik auf.
Dies ist im Standardfall auch erwünscht: Referenzieren mehrere Benutzer ein Objekt, so wirken sich Änderungen dieses Objekts automatisch auf alle diese Benutzer aus. Diese Kommunikationsmöglichkeit kann in vielen Anwendungsfällen sinnvoll eingesetzt werden. Werte hingegen können nicht verändert werden und schließen damit eine solche Kommunikationsmöglichkeit aus. Sie brauchen daher auch nicht über Referenzen zugreifbar zu sein.
Letzteres gilt zwar eigentlich auch für benutzerdefinierte Werte, jedoch können sie in Java nur über Objekte realisiert werden und weisen dadurch standardmäßig Referenzsemantik auf. Es gibt in Java kein Sprachmittel, welches die Definition von Objekten mit Speichersemantik erlauben würde.
Smalltalk hat bezüglich der Speicherung von Variablen ein einheitliches Konzept. Alle Variablen und Attribute von Klassen werden als Zeiger auf ein Objekt realisiert (vgl. [GOL83]). Auch in Smalltalk gibt es keine Mechanismen zur Definition von Objekten mit Speichersemantik.

3.3.2 C++: Eingebettete versus referenzierte Objekte

Während in Java Objekte nur über Referenzen an Variablen gebunden werden können, bietet C++ die Möglichkeit, Objekte direkt einzubinden, so wie dies auch mit Werten primitiver Datentypen geschehen würde. Das heißt, in C++ können Objekte auch via Speichersemantik zugegriffen werden. Wie dies in C++ erreicht werden kann, soll im folgenden Beispiel gezeigt werden.
Sei *Person* eine Klasse, welche ein Attribut *meineAdresse* enthält. Dieses Attribut kann entweder den Typ *Zeiger auf Adresse* oder den Typ *Adresse* haben:

a)
meineAdresse enthält einen **Zeiger auf ein Objekt** vom Typ Adresse

```
class Person
{
  Adresse *meineAdresse;
}
```

b)
meineAdresse enthält ein **Objekt** vom Typ Adresse

```
class Person
{
  Adresse meineAdresse;
}
```

Beispiel 3-1: Eingebettete versus referenzierte Objekte

In Beispiel 3-1 a) weist das Attribut *meineAdresse* Referenzsemantik auf. Es kann daher von mehreren Objekten des Typs *Person* zugegriffen werden, wodurch Aliasprobleme entstehen können.
In Beispiel 3-1 b) hingegen ist das Adreßobjekt direkt eingebunden und folgt damit der Speichersemantik. Es kann daher nicht von anderen Objekten referenziert werden. Dies scheint auf den ersten Blick das Aliasproblem zu verhindern. Allerdings bestehen in diesem Zusammenhang zwei Probleme:
Erstens bietet C++ die Möglichkeit, mit Hilfe des Adreßoperators die Speicheradresse dieses direkt eingebunden Objekts zu erhalten, wodurch es anschließend doch gemeinsam genutzt werden könnte. Es gibt in C++ keine Sprachmittel, um diesen Zugriff via Referenz auf ein Objekt zu verhindern.
Zweitens kann die Speichersemantik nicht für Objekte eines bestimmten Typs festgelegt werden, sondern nur für einzelne Variablen, welche zu diesem Zweck ohne Zeiger- und Referenzoperatoren deklariert werden.
Es gibt daher in C++ kein Sprachmittel, um mit Hilfe von Objekten mit Speichersemantik Wertobjekte zu simulieren.

3.3.3 Eiffel: expanded types

Eiffel bietet im Gegensatz zu C++ eine sicherere Möglichkeit, um Objekte mit Speichersemantik zu definieren, denn in Eiffel kann bei Klassendefinitionen angegeben werden, ob Variablen dieser Klasse ein Objekt oder eine Referenz auf ein Objekt speichern. Beim Entwurf der Sprache wurde einerseits darauf Wert gelegt, daß primitive Typen wie *Integer* und *Boolean* als Werte, benutzerdefinierte Typen wie *Punkt* oder *Buch* hingegen über Referenzen zugreifbar sind. Andererseits war es ein Ziel, der Sprache ein einheitliches Typsystem zu geben, derart, daß alle Typen der

Sprache durch ein einheitliches Konstrukt darstellbar sind. Daher werden in Eiffel alle Typen der Sprache durch das Klassenkonstrukt definiert. Um auf primitive Typen wie Integer und Real nicht über eine Referenz zugreifen zu müssen, wurden in Eiffel sogenannte *expanded types* eingeführt. Neben den vordefinierten primitiven Typen können mit Hilfe von *expanded types* auch benutzerdefinierte Objekte definiert werden, die Speichersemantik aufweisen.
Expanded types können auf zwei unterschiedlichen Arten eingesetzt werden:

1. Eine ganze Klasse kann als *expanded type* definiert werden, indem das Schlüsselworts *expanded* vor die Klassendefinition einfügt wird (siehe Beispiel 3-2 b). Das hat zur Folge, daß Variablen dieser Klasse immer Objekte speichern und nicht Referenzen.

2. Eine Variable einer Klasse *A* kann als *expanded type* deklariert werden, indem das Schlüsselwort *expanded* bei der Typdeklaration vor dem Typ eingefügt wird, beispielsweise *x: expanded A*. Die Variable speichert dann ein Objekt und keine Referenz. Ob die Klasse *A* selbst als *expanded* deklariert ist oder nicht, spielt keine Rolle.

a)
```
class PUNKT1 feature
  ...
end
```

b)
```
expanded class PUNKT2 feature
  ...
end
```

Beispiel 3-2: Klassendefinitionen in Eiffel

Um in *x* eine Referenz auf ein Objekt *Punkt* zu speichern, müßte dieses folgendermaßen deklariert werden:
- x: PUNKT1

Jede der folgenden Deklarationen würde dazu führen, daß *x* ein Objekt speichert.
- x: **expanded** PUNKT1
- x: PUNKT2
- x: **expanded** PUNKT2

Die Deklaration der gesamten Klasse als *expanded* wie in Beispiel 3-2 b) gewährleistet, daß alle Objekte dieser Klasse Speichersemantik aufweisen. Eiffel bietet damit im Gegensatz zu C++ ein Sprachmittel, welches für die Deklaration von Typen eingesetzt werden kann, deren Exemplare zur Verhinderung des Aliasproblems Speichersemantik aufweisen. Die Form aus Beispiel 3-2 a) ist hierzu weniger geeignet, da es hierbei dem Klienten der Klasse obliegt zu entscheiden, ob ein Objekt eines Typs Referenzsemantik aufweisen soll oder nicht. Dies entspräche der Situation, die in C++ gegeben ist.
Wie in C++ kann auch in Eiffel das Auftreten von Aliasproblemen für Attribute von Werttypen verhindert werden. Hierzu müssen alle Attribute als *expanded* deklariert sein, so daß sie allesamt Speichersemantik aufweisen.

Der Mechanismus der *expanded types* in Eiffel eignet sich demnach sehr gut, um Wertobjekte durch Objekte mit Speichersemantik anzunähern. Wenn die als *expanded* deklarierten Klassen zudem keine verändernden Operationen definieren, verhalten sich deren Exemplare wie die Werte der durch Schlüsselwörter gegebenen Werttypen in Java und C++. In Eiffel sind beispielsweise primitive Werttypen wie *Integer* und *Boolean* auf ebendiese Art definiert, nämlich durch *expanded types* ohne verändernde Operationen. Leider gibt es keine Möglichkeit, verändernde Operationen bei *expanded types* durch den Compiler auszuschließen. Das Problem liegt darin, daß nicht alle verändernden Anweisungen verboten werden können, da es sinnvoll sein kann, temporär verändernde Operationen zuzulassen, die vor der Abarbeitung den ursprünglichen Zustand wiederherstellen. Um sicherzustellen, daß eine Operation keine Veränderung auslöst, müßte der Compiler jedoch alle verändernden Anweisungen vermeiden, denn er kann nicht entscheiden, ob tatsächlich eine Veränderung stattgefunden hat oder ob die Änderung vor Beendigung der Operation wieder rückgängig gemacht wurde.

3.4 Sicherstellen der Unveränderlichkeit

Objekte mit Speichersemantik, die im vorigen Abschnitt beleuchtet wurden, lösen zwar das Aliasproblem. Derartige Objekte sind aber veränderlich, und das widerspricht dem Wertkonzept, da Werte unveränderlich und referentiell transparent sein sollen (vgl. Abschnitte 2.1.1 und 2.1.4).
Ein anderer Ansatz zur Implementierung von Werttypen als die Verwendung von Objekten mit Speichersemantik ist der Einsatz von unveränderlichen Objekten. Denn wird die Unveränderlichkeit garantiert, so wird dadurch zum einen wie bei Objekten mit Speichersemantik das Aliasproblem verhindert, und zudem die referentielle Transparenz gewährleistet. Das Aliasproblem, also eine nicht bemerkte Änderung eines mehrfach referenzierten Objekts, kann bei unveränderlichen Objekten naturgemäß nicht auftreten. Die referentielle Transparenz weisen unveränderliche Objekte deswegen auf, da sich ihr Zustand nicht ändern kann und sie daher stets denselben Wert bezeichnen. Unveränderliche Objekte können auf zweierlei Weise realisiert werden (vgl. [ZÜL98]):

1. *Objekte ohne verändernde Operationen (unveränderliche Objekte, immutable objects)*
   Die Unveränderlichkeit von Objekten wird dabei dadurch erreicht, daß die Attribute dieser Objekte nicht öffentlich zugreifbar sind und zudem keinerlei verändernde Operationen auf diesen Objekten definiert werden.

2. *Objekte, welche beim Aufruf verändernder Operationen kopiert werden (copy-on-write objects)*
   Diese Art von Objekten wird bei [ZÜL98] als *veränderliche Objekte* bezeichnet, da sie im Gegensatz zu den unveränderlichen Objekten verändernde Operationen aufweisen können. Wird eine solche verändernde Operation aufgerufen, so wird das Originalobjekt kopiert und die Änderung auf dieser Kopie durchgeführt. Die verändernde Operation liefert als Ergebnis diese geänderte Kopie zurück, während das Originalobjekt unverändert bleibt.

In folgenden wird zunächst dargestellt, wie die unveränderlichen Objekte des oben genannten ersten Ansatzes am Beispiel von Java realisiert werden können. Anschließend wird darauf eingegangen, inwiefern konstante Attribute für unveränderliche Objekte sinnvoll sein können.

3.4.1 Programmiertechniken zur Realisierung unveränderlicher Objekte

Im folgenden wird am Beispiel von Java dargestellt, welche Schritte zur Definition benutzerdefinierter unveränderlicher Objekte notwendig sind (vgl. [DAV98]).
Zunächst stellt sich die Frage nach einer Kennzeichnung unveränderlicher Klassen (vgl. Abschnitt 2.1.3). Es wäre zweckmäßig, daß alle unveränderlichen Klassen einen gemeinsamen Typ haben, beispielsweise *Immutable*. Dazu müßten diese Klassen von einem entsprechenden Java-Interface *Immutable* erben. Die Standardbibliothek von Java bietet jedoch kein solches Interface für die Eigenschaft der Unveränderlichkeit. Ohne ein solches Interface bleibt die Unveränderlichkeit einer Klasse eine Konvention und wird höchstens in der Klassendokumentation sichtbar. Als Lösung für dieses Problem kann ein Programmierer für seine eigenen Klassen ein Interface *Immutable* einführen, welches den Benutzern dieser Klassen erlaubt, sie zur Laufzeit mit Hilfe von *instanceof* auf Unveränderlichkeit zu prüfen Es muß sichergestellt sein, daß als unveränderlich deklarierte Objekte auch tatsächlich unveränderlich sind; denn wenn sich der Programmierer darauf zu Unrecht verläßt, kann es zu Fehlern kommen, beispielsweise zu einer Verletzung der referentiellen Transparenz.
Im folgenden wird dargestellt, welche Maßnahmen zur Sicherstellung der Unveränderlichkeit getroffen werden müssen.

Ein erster Schritt zur Erzeugung von unveränderlichen Objekten ist das Vermeiden von Methoden zum Setzen von Attributen: die Werte der Attribute dürfen nur durch den Konstruktor gesetzt werden. Außerdem müssen alle Attribute als *private* deklariert sein, damit weder Subklassen noch andere Klassen im selben Package auf die Attribute zugreifen können.
In einem zweiten Schritt muß der Umgang mit Rückgabewerten festgelegt werden. Rückgabewerte von Methoden müssen geklont werden, falls sie ein veränderliches Attribut zurückgeben. Dies wird beispielsweise in folgendem Fall notwendig (vgl. [DAV98]):
Sei eine Klasse, deren Objekte unveränderlich sein sollen, definiert durch

```
public class Zahlenliste
{
   private int[] eintraege;
   ...
   public int [] gibEintraege()
   {
      return eintraege;
   }
}
```

Mittels Aufruf der Methode *gibEintraege* kann nun der Inhalt von Objekten dieser Klasse von anderen Objekten referenziert werden:

```
int[] zahlen = meineZahlenliste.gibEintraege();
```

Nach einer solchen Anweisung wirken sich Änderungen an der Variablen *zahlen* auch auf das Attribut *eintraege* des Objekts *meineZahlenliste* aus. Um dieses Ändern der als unveränderlich vorgesehenen Klasse zu verhindern, muß bei der Rückgabe eines Arrays oder eines anderen veränderlichen Attributs ein Klonen erfolgen. Die Methode *gibEintraege* muß also in diesem Fall folgendermaßen aussehen:

```
public int[] gibEintraege()
{
   return (int[]) eintraege.clone();
}
```

Zudem muß ein Klonen erfolgen, wenn veränderliche Objekte einem Konstruktor übergeben werden. Dies läßt sich an folgendem Beispiel zeigen:

```
public class Zahlenliste
{
   private int[] eintraege;

   public Zahlenliste(int[] eintraege)
   {
      this.eintraege = eintraege;
    }
}
```

An anderer Stelle kann ein Array *zahlen* angelegt und dem Konstruktor übergeben werden:

```
int[] zahlen = {1,2,3};
Zahlenliste meineZahlenliste= new Zahlenliste(zahlen);
```

Der als Parameter übergebene Array kann nun von außerhalb geändert werden:

```
zahlen[1] = 0;
```

Diese Änderung wirkt sich auch auf den in dem Objekt *meineZahlenliste* gespeicherten Array aus, das bedeutet: die Objekte der Klasse *Zahlenliste* sind nicht wirklich unveränderlich. Um diese Änderung von außerhalb auszuschließen, müßte der Konstruktor folgendermaßen korrigiert werden:

```
public Zahlenliste (int[] eintraege)
{
   this.eintraege = (int[])eintraege.clone();
}
```

Allerdings ist zu beachten, daß das Klonen nicht immer unproblematisch ist: viele veränderliche Klassen der Java-Standardbibliothek, wie beispielsweise *Vector*, unterstützen kein sicheres Klonen. In diesen Fällen kann eine Änderung des Origi-

nals immer noch Einfluß auf das geklonte Objekt haben und umgekehrt (vgl. [DAV98]). Dies ist dann der Fall, wenn beim Klonen keine tiefe, sondern nur eine flache Kopie angelegt wird. Wird etwa ein Objekt der Java-Klasse *Vector* geklont, welches eine Reihe von Referenzen enthält, so werden nur diese Referenzen kopiert und nicht die referenzierten Objekte. Die Kopie des *Vector*-Objekts enthält dadurch dieselben Referenzen wie das Original und kann über diese Referenzen das ursprüngliche *Vector*-Objekt verändern.
Ist in einer Klasse die Methode zum Klonen nicht sicher, so muß der Programmierer entweder eine eigene sichere Methode zum Klonen implementieren oder Objekte dieser Klassen als Parameter in Konstruktoren und als Rückgabewerte von Zugriffsmethoden vermeiden.

3.4.2 Konstante Attribute und Klassen

In diesem Abschnitt wird untersucht, welche Sprachmittel in den untersuchten Sprachen für die Implementierung unveränderlicher Objekte genutzt werden können. Der Einsatz solcher Sprachmittel würde gegenüber den im vorigen Abschnitt beschriebenen Programmierkonventionen den Vorteil haben, daß der Compiler überprüfen könnte, ob verändernden Operationen stattfinden, um gegebenenfalls mit einer Fehlermeldung zu reagieren.
Dazu ist es erforderlich, angeben zu können, daß sich Attribute einer Klasse nach der Initialisierung nicht mehr verändern dürfen. Zu diesem Zweck können möglicherweise Konstanten verwendet werden. Generell sind dabei zwei verschiedene Einsatzmöglichkeiten denkbar: Zum einen könnten alle Attribute einer Klasse als konstant deklariert werden. Denn wenn sichergestellt werden könnte, daß alle Attribute einer Klasse unveränderlich sind, wäre damit die gesamte Klasse unveränderlich. Zum anderen ist zu untersuchen, ob Objekte oder Klassen direkt als konstant deklariert werden können.

3.4.2.1 Java: Das Schlüsselwort final

Sowohl Klassen- als auch Exemplarvariablen können in Java als *final* deklariert werden, ebenso wie Methoden oder Klassen. Die Initialisierung eines als *final* deklarierten Attributs muß stattfinden, bevor es benutzt wird. Dies kann auf drei verschiedene Arten erfolgen (vgl. [GOS96]): entweder statisch bei der Deklaration oder in einem Initialisierungblock oder, wenn die Initialisierung dynamisch stattfinden soll, in jedem Konstruktor der Klasse.

```
class PunktAufOrdinate
{
   final int x = 0;
   final int y;

   public PunktAufOrdinate (int y) { this.y = y;}
   ....
}
```

Einem als *final* deklarierten Attribut kann nach der Initialisierung kein neuer Wert mehr zugewiesen werden: dies würde bereits zur Compilezeit zu einer Fehlermel-

dung führen. Ein als *final* deklariertes Attribut ist also unveränderlich und enthält daher stets den bei der Initialisierung zugewiesenen Wert. Besteht dieser Wert aus einer Referenz auf ein Objekt, so bleibt dementsprechend diese Referenz unverändert und verweist daher immer auf dasselbe Objekt.
Der Zustand dieses referenzierten Objekts kann allerdings veränderbar sein, falls auf diesem Objekt entsprechende Operationen definiert sind (Beispiel aus [DAV98]):

```
final Punkt ursprung = new Punkt(0,0);
   ...
   void eineMethode()
   {
      ursprung.x = 3;  // Änderung des Punktes 'ursprung'!
   }
```

Ähnlich verhält es sich mit referenzierten Arrays: enthält ein als *final* deklariertes Attribut eine Referenz auf ein Array, so bedeutet dies nur, daß stets dasselbe Array referenziert wird – dessen Inhalte jedoch können durch Operationen veränderbar sein (vgl. [GOS96]).

Die Verwendung des Schlüsselwort *final* garantiert also nicht in jedem Fall die Unveränderlichkeit von Objekten, es kann jedoch in den folgenden beiden Fällen sinnvoll eingesetzt werden:

1. Sind alle Attribute einer Klasse als *final* deklariert, und alle Attribute haben entweder primitive Datentypen oder sind Referenzen auf unveränderliche Objekte, so ist diese Klasse unveränderlich. Ist allerdings eines der Attribute eine Referenz auf ein veränderliches Objekt, so ist die Klasse nicht als unveränderlich anzusehen.

2. Wird eine Klasse so implementiert, daß sie die Eigenschaft der Unveränderlichkeit aufweist, so kann sie als *final* deklariert werden, um auf diese Weise die Ableitung von Unterklassen zu verhindern. Ist nämlich eine unveränderliche Klasse nicht als *final* deklariert, so ist diese Eigenschaft bei abgeleiteten Klassen nicht gesichert, da sie zusätzliche verändernde Methoden erhalten können (vgl. [DAV98]). Klienten können sich daher bei Objekten vom Typ einer unveränderlichen Klasse nur dann auf die Eigenschaft der Unveränderlichkeit verlassen, wenn diese Klasse als *final* deklariert ist.

3.4.2.2 C++: Das Schlüsselwort const

Das Schlüsselwort *const* kann ähnlich wie das Schlüsselwort *final* in Java dazu eingesetzt werden, die Attribute von Klassen als unveränderlich zu deklarieren. Es kann allerdings nicht dazu verwendet werden, eine gesamte Klasse als konstant zu markieren. Werden Attribute mit dem Schlüsselwort *const* deklariert, so müssen sie in der Elementinitialisierungsliste des Konstruktors initialisiert werden (vgl. [AND98]):

```
class Punkt
{
  private:
    const int x,y;
  public:
    Punkt (int wert1, int wert2) : x (wert1), y (wert2) {}
}
```

Anschließend können den Attributen keine anderen Werte mehr zugewiesen werden. Das Schlüsselwort *const* kann nicht nur bei Attributen mit einfachen Wert- und Objekttypen eingesetzt werden, sondern auch bei Attributen mit Zeigertypen. Dabei gibt es zwei Möglichkeiten: Zum einen kann das Schlüsselwort wie *final* in Java dazu verwendet werden, daß der Zeiger stets auf dieselbe Speicheradresse verweist (vgl. [RED94]):

```
int i = 10;
int j = 12;
int* const ptr = &i
ptr = &j    // ist nicht erlaubt
```

Dabei besteht wie in Java das Problem, daß das referenzierte Objekt änderbar ist. Der Vorteil von *const* gegenüber *final* liegt in der zweiten Einsatzmöglichkeit dieses Schlüsselwortes bei Zeigertypen: Es kann verhindern, daß Zeiger zum Ändern des referenzierten Objekts verwendet werden können, indem das Schlüsselwort bei der Zeigerdeklaration vorangestellt wird. Diese beiden Einsatzmöglichkeiten von *const* können auch kombiniert werden, so daß einerseits der Zeiger stets auf denselben Speicherplatz zeigt und darüber hinaus dieser Zeiger auch nicht zum Ändern des referenzierten Objekts verwendet werden kann:

```
class Beispielklasse
{
  private:
    const Punkt * const zeiger_auf_punkt ;

  public:
    // (Konstruktor ....)

    void eine_prozedur()
    {
      zeiger_auf_punkt->erhoeheXKoordinate(); // nicht erlaubt
    }
}
```

Durch eine solche Deklaration des Attributs *zeiger_auf_punkt* wird sichergestellt, daß das referenzierte Objekt nicht über diesen Namen verändert werden kann. Allerdings kann die Eigenschaft der Konstantheit durch eine explizite Typumwandlung aufgehoben werden. So könnte der Zeiger des obigen Beispiels in einen normalen Zeiger umgewandelt werden, wonach er doch zum Verändern des referenzierten Objekts verwendet werden könnte:

```
((Punkt*) zeiger_auf_punkt) ->erhoeheXKoordinate();
```

Diese spezielle Art der expliziten Typumwandlung, welche noch ein Erbe aus der imperativen Sprache *C* ist, wird aus Sicherheitsgründen allgemein als problematisch angesehen. Es werden für explizite Typumwandlungen sicherere Methoden vorgeschlagen, wie die Benutzung vordefinierter Funktionen (vgl. [STR00]).

Zum Ändern des Objekts, welches vom Attribut *zeiger_auf_punkt* referenziert wird, könnte ferner ein Zeiger von außerhalb der Klasse *Beispielklasse* verwendet werden. Diese zweite Änderungsmöglichkeit kann aber dadurch verhindert werden, daß das Objekt nur von der zugehörigen Klasse referenziert wird, das heißt, es muß bei der Übergabe geklont werden (vgl. Abschnitt 3.4.1). Bei Java würde dieses Klonen nicht ausreichen, da die Deklarierung von Referenzen als *final* in Java nicht die Änderung der referenzierten Objekte durch die eigene Klasse verhindern.

Zusammenfassend ist festzustellen, daß mit Hilfe des Schlüsselworts *const* zwar ein zusätzlicher Schutz vor Änderungen von Attributen zu erzielen ist, welcher jedoch umgangen werden kann. Daher kann dieses Schlüsselwort die Unveränderlichkeit eines Attributs nicht garantieren.

3.4.2.3 Eiffel: Konstanten und Klasseninvarianten

Ähnlich wie Java und C++ bietet Eiffel eine Möglichkeit, Konstanten zu definieren. Es können sowohl Konstanten von Basistypen als auch Konstanten von Klassentypen definiert werden. Zum Definieren von Konstanten gibt es in Eiffel allerdings kein eigenes Schlüsselwort. Statt dessen werden Konstanten als Methoden implementiert, die immer denselben Wert liefern. Bei Konstanten- bzw. Methodendeklarationen wie im folgenden Beispiel wird demnach immer der Wert *0* zurückgeliefert, wenn auf das Attribut *Ok* zugegriffen wird. Daher würde bei der Zuweisung "*fehler_code = Ok*" der Variablen *fehler_code* der Integerwert *0* zugewiesen.

```
class DATEI feature
  -- Konstanten
  Ok: INTEGER is 0
  Fehler_beim_Öffnen: INTEGER is 1

  -- Variable
  fehler_code: INTEGER;
  ...
end
```

Da die derart definierten Konstanten keine Attribute darstellen, ist es nicht möglich, ihnen variabel einen Wert zuzuweisen, auf daß das Attribut danach immer diesen Wert bezeichne. Der Rückgabewert muß vielmehr wie im obigen Beispiel bereits im Programmcode festgelegt werden. Um unveränderliche Objekte anhand von unveränderlichen Attributen implementieren zu können, müssen die Attribute aber flexibel innerhalb eines Konstruktors initialisierbar sein.
Zudem liefert die konstante Methode zwar stets dasselbe Ergebnis - ist der Rückgabewert jedoch ein Objekt, so kann es anschließend geändert werden und ist daher nicht wirklich konstant. Aus diesen Gründen eignet sich das Konstanten-Konzept von Eiffel nicht für die Implementierung unveränderlicher Attribute: Es kann lediglich dazu eingesetzt werden, um *symbolische Konstanten* bereitzustellen, also einem bestimmten Wert einen bedeutungsvollen Namen zu geben.
Interessanter ist in diesem Zusammenhang das Vertragsmodell (vgl. Abschnitt 3.2.1). Ein Vertrag legt für eine Methode die Vor- und Nachbedingungen fest. Es könnte dabei festgelegt werden, daß alle Attribute nach Ausführung dieser Methode denselben Wert haben wie vorher.
Ein solcher Vertrag müßte bei Objekten, die unveränderlich sein sollen, für alle Methoden gelten. Deswegen wäre es ideal, hierfür das Konstrukt der Klasseninvarianten einsetzen zu können, da sich diese automatisch auf alle Methoden einer Klasse beziehen. Allerdings gibt es keine vordefinierte Invariante, die garantieren könnte, daß die Methoden keine Attribute verändern. Es ist auch nicht möglich, eine solche Invariante selbst zu definieren, weil es bei der Definition einer Invarianten unmöglich ist, sich auf den Zustand von Attributen nach und vor der Ausführung einer Methode gleichzeitig zu beziehen. Theoretisch könnte dieses Problem durch die Verwendung des Eiffel-Schlüsselworts *old* gelöst werden, durch welches in einer Nachbedingung auf den ursprünglichen Wert eines Attributs zum Zeitpunkt des Methodeneintritts zugegriffen werden kann. Um zu gewährleisten, daß sich kein Attribut durch das Ausführen einer Methode ändert, müßte für jedes Attribut als Nachbedingung erscheinen: "*ensure attribute = old attribute*". Bei Referenzen würde dieser Vergleich allerdings zum Vergleich der Referenzen führen. Da aber deren Inhalte gleich sein müssen, muß die Nachbedingung bei Referenzen: *ensure equal(attribute, old attribute)* lauten. Da das Schlüsselwort *old* nur bei Nachbedingungen erlaubt ist, ist es in Eiffel aber nicht möglich, diese Bedingungen als Invarianten zu deklarieren.
Die einzige Möglichkeit, Änderungen an Attributen zu überwachen, besteht also darin, für jede einzelne Methode die entsprechenden Nachbedingungen festzulegen. Dabei kann eine solche Änderung nicht verhindert werden, sondern sie führt nur zu einer Fehlermeldung (vgl. Abschnitt 3.2.1).

Zusammengefaßt läßt sich feststellen, daß von den betrachteten Sprachen nur Java und C++ Schlüsselwörter aufweisen, die zur Unveränderlichkeit von Objekten beitragen können. Es erscheint jedoch generell wenig sinnvoll, vom Programmierer per Programmierkonvention zu verlangen, daß alle Attribute als konstant deklariert werden müssen - nur um die andere Programmierkonvention, welche den Einsatz verändernder Operationen verbietet, zu überprüfen. Jedoch könnte die Verwendung von konstanten Attributen eventuell sinnvoll sein, wenn der Code für eine unverän-

derliche Klasse automatisch erzeugt wird, denn in diesem Fall würden konstante Attribute ohne Mehraufwand für den Programmierer vor dem manuellen Eintragen verändernder Operationen schützen.

3.5 Optimierungstechniken für den Einsatz von unveränderlichen Objekten

Sowohl die Verwendung unveränderlicher Objekte als auch die Verwendung von Copy-on-write-Objekten (vgl. Abschnitt 3.4) für die Implementierung benutzerdefinierter Werte hat zur Folge, daß jede Änderung zur Erzeugung eines neuen Objekts führt. Diese wiederholte Erzeugung von Objekten kann sich jedoch auf die Effizienz auswirken. Zu den Kosten für ein Objekt trägt nicht nur die Reservierung des Speicherplatzes bei, sondern auch die notwendige Wiederbereitstellung des Speichers nach der Zerstörung des Objekts und der Umstand, daß jedes Objekt während jeder Speicherbereinigung (*Garbage Collection*) überprüft werden muß (vgl. [DAV98]). Im folgenden werden Techniken vorgestellt, mit deren Hilfe die Erzeugung von Objekten auf das notwendige Mindestmaß beschränkt werden kann: die Verwendung *dualer Klassen* (vgl. [DAV98]) und der Einsatz der Entwurfsmuster *Fliegengewicht* und *Fabrik* (vgl. [GAM95]).

3.5.1 Duale Klassen

Beim Ansatz der Verwendung von dualen Klassen wird zunächst eine Basisklasse implementiert, von welcher zwei Klassen erben: eine unveränderliche Klasse für die Realisierung von Werten und eine veränderliche Klasse zur Modellierung von Objekten. In der veränderlichen Klasse sind Methoden zum Setzen und Verändern von Attributen enthalten, während die unveränderliche Klasse diese nicht enthält. Die Bereitstellung einer veränderlichen Klasse ist sinnvoll, wenn eine unveränderliche Klasse viele Funktionen besitzt, die ein verändertes Objekt gleicher Klasse zurückliefern. Solche Funktionen können auch als verändernde Prozeduren implementiert werden, wodurch der Aufwand der Objekterzeugung gespart wird - allerdings auf Kosten der Wertsemantik.
Ein Beispiel für duale Klassen sind in Java die Klassen *String* und *StringBuffer*: Instanzen der Klasse *String* haben einen unveränderlichen Wert, daher führt ein Zusammenfügen zweier Zeichenketten mit Hilfe des Konkatenationsoperators dazu, daß für die Repräsentation dieser Verbindung ein neues *String*-Objekt erzeugt wird. Die beiden ursprünglichen Zeichenketten existieren unabhängig davon unverändert weiter. Dies bedeutet, daß bei Verwendung von Objekten der Klasse *String* relativ viele neue Objekte angelegt werden. Im Gegensatz dazu sind Exemplare der Klasse *StringBuffer* veränderlich, so daß beim Hinzufügen einer neuen Zeichenkette zu einer bereits existierenden kein neues *String*-Objekt angelegt, sondern das ursprüngliche entsprechend geändert wird. Dadurch wird die Erzeugung neuer Objekte deutlich reduziert, allerdings geht der ursprüngliche Wert des geänderten Strings dabei verloren.
Die Klasse *StringBuffer* bietet sich daher für Anwendungen an, wo die Eigenschaft der Unveränderlichkeit nicht benötigt wird, die Klasse *String* sollte dagegen nur bei Anwendungen eingesetzt werden, wo diese Eigenschaft von Bedeutung ist. Mit

solchen dualen Klassen kann die Erzeugung von Objekten auf das notwendige Minimum reduziert und zudem je nach Anwendungsfall entschieden werden, ob die Geschwindigkeit der entscheidende Faktor ist oder die Sicherheit, welche mit Hilfe eines unveränderlichen Objekts erreicht werden kann (vgl. [DAV98]).

3.5.2 Die Entwurfsmuster „Fliegengewicht“ und „Fabrik“

Im Zusammenhang mit der wiederholten Erzeugung von Objekten kann es zu unverhältnismäßig hohem Speicherplatzbedarf kommen. Sei beispielsweise *Prozentzahl* ein Werttyp, der ganze Zahlen zwischen *0* und *100* repräsentiert, so wären zur Darstellung dieses Wertebereichs prinzipiell 100 Objekte ausreichend. Existieren allerdings in einem System viele Variablen des Typs *Prozentzahl*, so werden weitaus mehr als 100 *Prozentzahl*-Objekte benötigt, da jede dieser Variablen ein eigenes Objekt enthält – dadurch können mehrfache Kopien des gleichen Werts existieren. Da Werte unveränderlich sind, ist es semantisch nicht von Bedeutung, ob mehrere Klienten denselben Wert verwenden oder jeweils Kopien dieses Wertes. Daher kann es programmiertechnisch durchaus sinnvoll sein, Werte gemeinsam zu benutzen, um dadurch Speicherplatz zu sparen.
Ein entsprechender Lösungsansatz ist der Einsatz von *gemeinsam benutzten Objekten (shared objects*): dabei werden Objekte mit Hilfe des Fliegengewicht-Musters (vgl. [GAM95]) mehrfach verwendet. Grundprinzip ist dabei, daß zur Repräsentation eines bestimmten Werts nur ein einziges Objekt notwendig ist. Wird derselbe Wert an zwei Stellen benötigt, so kann das entsprechende Objekt von diesen beiden Stellen gemeinsam verwendet werden – da das Objekt unveränderlich ist, kann kein Aliasproblem auftreten. Auf diese Weise kann die Anzahl der notwendigen Objekte deutlich reduziert werden.
Das Fliegengewichtmuster stellt durch die Verwendung einer sogenannten Fabrik (*Factory*) sicher, daß für jeden Wert nur ein einziges entsprechendes Objekt angelegt wird. Dies geschieht dadurch, daß nur die Fabrik Objekte anlegen kann und zudem Kenntnis über die Menge der bereits angelegten Wertobjekte hat. Wird von der Fabrik ein Objekt für einen bestimmten Wert angefordert, so überprüft sie zunächst, ob für diesen Wert bereits ein Objekt angelegt wurde. Ist dies der Fall, so liefert die Fabrik das bereits existierende Objekt zurück, ansonsten erzeugt sie das gewünschte Objekt neu. Die Verwendung des Fliegengewichtmusters bietet sich besonders bei benutzerdefinierten Werttypen an, deren Wertebereich begrenzt ist, wie beispielsweise bei Wochentagen (vgl. [ZÜL98]). In solchen Fällen ist der Speicherplatzgewinn besonders hoch, da nur wenige Objekte erzeugt werden müssen. Beinhaltet ein benutzerdefinierter Werttyp dagegen sehr viele verschiedene Werte, so müssen im Extremfall auch ebenso viele Objekte erzeugt werden und der Speicherplatzgewinn kann dementsprechend gering ausfallen.

3.6 Intuitive Benutzbarkeit

Werden benutzerdefinierte Werte mit Hilfe von Objekten implementiert, so führt dies zu einer unnatürlichen Notation von manchen Operationen. Als Beispiel diene eine Methode zum Addieren zweier benutzerdefinierter Werte *a* und *b*: natürliche, intuitive Schreibweisen wären "*a+b*" oder "*addiere(a,b)*". Sind *a* und *b* jedoch mit Hilfe von Objekten realisiert, so muß statt dessen geschrieben werden

"*a.addiere(b)*". Diese Notation erweckt den Anschein, daß zum einen eine Änderung des Wertes *a* erfolgt, und daß zum anderen ein aktives Verhalten dieses Wertes vorliegt. Da Werte jedoch nicht veränderlich sein dürfen und zudem als passiv betrachtet werden müssen, erscheint diese Notation unnatürlich. Dazu trägt auch bei, daß die Gleichberechtigung der beiden Operanden *a* und *b* nicht deutlich wird: "*a.addiere(b)*" erscheint nicht wie eine Operation mit zwei Operanden, sondern eher wie ein Auftrag an das Objekt *a*, einen einzelnen Operanden, nämlich *b*, zu addieren (vgl. [PER99]).
In diesem Abschnitt werden zwei Ansätze vorgestellt, die zu einer natürlicheren Benutzbarkeit von benutzerdefinierten Werten führen können: das Überladen von Operatoren sowie die Trennung von Wert und Verhalten in Kombination mit geschachtelten Klassen.

3.6.1 Überladen von Operatoren

Für eingebaute primitive Datentypen von Programmiersprachen existieren in der Regel Operatoren für den intuitiven Umgang mit Werten dieser Typen. Beispielsweise können Berechnungen mit Ganzzahlen mit den Infixoperatoren "+", "*", "<" usw. durchgeführt werden.
Bei der Definition eines benutzerdefinierten Datentyps müssen zusätzlich zu seiner Struktur auch die Operationen definiert werden, welche auf Werte dieses Typs anwendbar sein sollen. C++ und Eiffel bieten dabei die Möglichkeit, Operatoren zu überladen: dies bedeutet, den Operatoren werden im Zusammenhang mit benutzerdefinierten Typen andere Bedeutungen zugeordnet als im Zusammenhang mit Werten der eingebauten primitiven Datentypen. Das Ziel dabei ist, den Umgang mit Werten benutzerdefinierter Typen möglichst ebenso intuitiv zu gestalten wie mit Werten der primitiven Datentypen.

Um beispielsweise zwei Daten zu vergleichen, kann folgende Methode definiert werden (Beispiele in C++-Notation):

```
Datum datum1, datum2;
...
if (datum1.frueher_als(datum2)) ...
```

Vereinfachend könnte der Operator "<" überladen werden, so daß folgende Notation möglich wäre:

```
Datum datum1, datum2;
....
if(datum1 < datum2) ....
```

Der Nutzen des Operator-Überladens wird besonders an komplizierteren Ausdrücken deutlich. Werden beispielsweise für Bruchzahlen benutzerdefinierte Werttypen verwendet, so würde eine Rechnung mit zwei Variablen *zahl1* und *zahl2* dieses Typs folgendermaßen aussehen:

```
zahl1.mult(10).minus(zahl2.mult(7)).div(20)
```

Das Überladen von Operatoren könnte dagegen zu einer deutlich verständlicheren Notation führen:

```
(10*zahl1-7*zahl2)/20
```

Ein solches Überladen von Operatoren ist in Java nicht möglich: in Java müssen daher alle Operationen auf Werten mit benutzerdefinierten Datentypen mit Hilfe von Methoden nachgebildet werden. Da das Überladen von Operatoren zur intuitiveren Anwendbarkeit benutzerdefinierter Datentypen beitragen kann, bieten C++ und Eiffel in dieser Hinsicht eine bessere Unterstützung benutzerdefinierter Werte als Java.

3.6.2 Die Trennung von Wert und Verhalten

Bei diesem Ansatz zur besseren Benutzbarkeit benutzerdefinierter Werte wird die Implementierung des Werts von der Implementierung seines Verhaltens getrennt, welche jeweils in einer eigenen Klasse erfolgt (vgl. [PER99]). Von der Verhaltensklasse existiert nur ein Exemplar, während von der Wertklasse ein Exemplar pro benötigtem Wert existiert. In Java kann die Verhaltensklasse als statische Klasse implementiert werden, welche nur statische Attribute und Funktionen enthält. Diese statischen Funktionen können unabhängig von Objekten der Klasse aufgerufen werden. Sind *a* und *b* beispielsweise Vektoren, so wird es durch den Einsatz einer Verhaltensklasse *VektorOperationen* möglich, statt "*a.addiere(b)*" zu schreiben: "*VektorOperationen.addiere(a, b)*".
Zudem bietet Java die Möglichkeit, die beiden miteinander in Zusammenhang stehenden Klassen mittels sogenannter *inner classes* ineinander zu schachteln, so daß die Implementierung eines benutzerdefinierten Werts nach außen hin in einer einzigen Einheit erfolgen kann.
Der Vorteil dieses gesamten Ansatzes liegt darin, daß sich für Klienten Werte, Variablen und Operationen auf dieselbe Weise verhalten wie sprachinterne Werttypen.
Allerdings stellt sich die Frage, ob eine Klasse, welche keine Exemplare hat, in Übereinstimmung mit dem objektorientierten Paradigma steht (vgl. [PER99]) – denn solch eine Klasse dient lediglich dem Zusammenfassen von Funktionen und entspricht daher eher dem Strukturierungskonzept von imperativen Programmiersprachen.

3.7 Schlußbetrachtungen

Die Realisierung von benutzerdefinierten Werttypen kann in objektorientierten Programmiersprachen wie C++, Java oder Eiffel nur mit Hilfe von Klassen erfolgen. Diese Repräsentation von Werten durch Objekte führt jedoch dazu, daß die Werte einige der Eigenschaften von Objekten übernehmen, welche im Widerspruch zu den geforderten Werteigenschaften stehen.
Dies hat zur Folge, daß diese Werteigenschaften, wie beispielsweise die Unveränderlichkeit, die Vermeidung des Aliasproblems und die intuitive Benutzbarkeit, explizit durch den Entwickler sichergestellt werden müssen. Die untersuchten Programmiersprachen bieten auf Ebene der Sprachmittel allesamt keine überzeugende Lösung für

dieses Problem. Zwar können in einigen Sprachen Objekte mit Speichersemantik eingesetzt werden, wodurch das Aliasproblem verhindert wird (vgl. Abschnitt 3.3). Die für Werte geforderte referentielle Transparenz läßt sich dadurch allerdings nicht sicherstellen, denn dafür müssen Objekte, die Werte repräsentieren sollen, generell unveränderlich sein. Es gibt in den untersuchten objektorientierten Sprachen jedoch keine Möglichkeit, diese Unveränderlichkeit in jedem Fall zu garantieren (vgl. Abschnitt 3.4). Zudem existiert in den untersuchten Sprachen kein spezielles Sprachkonstrukt für die Kennzeichnung von Typen als Werttypen. Daher kann ein Compiler keine Optimierungstechniken wie die gemeinsame Benutzung von Werten einsetzen.
Zusammenfassend ist festzustellen, daß bei der Implementierung benutzerdefinierter Werte in den untersuchten objektorientierten Sprachen nur schwer die vollständige Wahrung aller Werteigenschaften zu erreichen ist und zudem ein hohes Fachwissen über diese Eigenschaften sowie über die technische Wirkungsweise der jeweiligen verfügbaren Sprachkonstrukte erfordert. Dadurch ergeben sich zahlreiche Fehlermöglichkeiten, welche die Erstellung benutzerdefinierter Werte in den untersuchten Programmiersprachen als problematisch und zeitaufwendig erscheinen lassen.

4 Möglichkeiten zur Unterstützung der Deklaration von Werttypen

In der Literatur finden sich einige interessante Ansätze im Zusammenhang mit der Deklaration benutzerdefinierter Werttypen. Dazu gehören die Techniken, die in diesem Kapitel beschrieben werden: die Kombination von funktionalem mit objektorientierten Paradigma, die Verwendung von XML-Schema-Datentypen und der Einsatz von Rahmenwerken.

4.1 Spracherweiterung durch Einbettung von Konzepten funktionaler Sprachen

Während Objekttypen die Basis der objektorientierten Programmiersprachen bilden, wird in funktionalen Sprachen ausschließlich mit Werttypen gearbeitet. Daher sind in funktionalen Programmiersprachen alle Typbildungsmechanismen auf die Erstellung von Werttypen ausgelegt. Im Gegensatz zu objektorientierten Sprachen ist es also in funktionalen Programmiersprachen auf einfache Weise möglich, Werttypen zu definieren. Dies wird im folgenden näher erläutert (vgl. Abschnitt 4.1.1). Anschließend wird anhand der Programmiersprache *Pizza* dargestellt, inwiefern die Einbindung funktionaler Konzepte in objektorientierte Sprachen die Möglichkeiten zur Definition von Werttypen in objektorientierten Programmiersprachen verbessern kann (vgl. Abschnitt 4.1.2).

4.1.1 Das Typkonzept in funktionalen Programmiersprachen

Funktionale Programmiersprachen gehören zur Familie der deklarativen Sprachen. Die deklarativen Sprachen erlauben die einfache und klare Darstellung von Algorithmen und verfolgen dabei das Ideal, das Programmieren durch ein einfaches Vorgeben des Problems zu ersetzen (vgl. [PAU91]).
Funktionale Programmiersprachen basieren auf der Auswertung von Ausdrücken, wobei ein Ausdruck einen Wert repräsentiert. Mit Hilfe von Funktionen kann der Programmierer angeben, auf welche Weise diese Auswertung erfolgen soll, wobei die Funktionen nach mathematischen Prinzipien formuliert werden. Ergebnis der Auswertung ist die einfachste Form des Ausdrucks und damit die einfachste Darstellung des Wertes, den der Ausdruck bezeichnet, sofern der Ausdruck einen wohldefinierten Wert besitzt (vgl. [BIR92]). Eine Auswertung endet also mit einem kanonischen Resultat und hat ansonsten keine Auswirkungen. In funktionalen Sprachen können keine Seiteneffekte auftreten, da es keine veränderlichen Variablen gibt: in funktionalen Sprachen können Werte direkt definiert und in Ausdrücken verwendet werden, wobei vom Speicherort dieser Werte abstrahiert wird. Durch die Abwesenheit von änderbaren Variablen bezeichnet ein Ausdruck immer denselben Wert. Diese Eigenschaft, die als *referentielle Transparenz* (vgl. Abschnitt 2.1.4) bezeichnet wird, gilt als das charakteristische Merkmal einer funktionalen Sprache.
Als Konsequenz aus den oben beschriebenen Eigenschaften der funktionalen Sprachen ergibt sich, daß alle Typen in diesen Sprachen Werttypen sind. Für jeden dieser

Werttypen in funktionalen Sprachen wird angenommen, daß ein sogenannter *undefinierter Wert* in der Wertemenge des Typs enthalten ist - ohne daß dafür spezielle Maßnahmen bei der Definition des Typs getroffen werden müßten. Dieser undefinierte Wert, welcher für die mathematische Korrektheit von Funktionen notwendig ist, wird im folgenden Abschnitt erläutert (vgl. Abschnitt 4.1.1.1).
Da es in den funktionalen Sprachen im Gegensatz zu den objektorientierten Sprachen einfach ist, neue Werttypen zu definieren, sind diese funktionalen Typdefinitionsmechanismen von besonderem Interesse; sie werden daher im anschließenden Abschnitt untersucht (vgl. Abschnitt 4.1.1.2).

4.1.1.1 Der undefinierte Wert

Der *undefinierte Wert* ist bei Datentypen in funktionalen Sprachen stets Teil der Wertemenge. Unter dem undefinierten Wert wird ein einziger besonderer Wert verstanden, welcher zur einfacheren mathematischen Behandlung von Funktionen dient. Nur wenn der undefinierte Wert zur Wertemenge gehört, liefert eine Funktion immer einen Wert als Ergebnis, das heißt: nur dann bezeichnet jeder Ausdruck einen Wert. Ohne den undefinierten Wert müßte beispielsweise bei der Funktion *dividiere(a,b)* die Einschränkung gelten, daß *b* nicht *0* sein darf. Ist der undefinierte Wert jedoch in der Wertemenge enthalten, so läßt sich sagen, daß auch der Ausdruck *dividiere(1,0)* einen Wert bezeichnet: Wird der undefinierte Wert zum Beispiel durch das Zeichen "⊥" dargestellt, so gilt "*dividiere(1,0)* = ⊥" (vgl. [BIR92]). Das bedeutet allerdings nicht, daß diese Berechnung in einer funktionalen Sprache zu einer Ausgabe wie "<*undefiniert*>" führt. Statt dessen kann mit einer Fehlermeldung abgebrochen, oder die Berechnung unendlich fortgesetzt werden (vgl. [BIR98]).
Wenn der undefinierte Wert dem Wertebereich zugerechnet wird, ist zu beachten, daß dann auch der Wert von Ausdrücken definiert sein muß, die den undefinierten Wert enthalten (vgl. [BIR92]). Dazu müssen alle Funktionen auch dann wohldefinierte Ergebnisse haben, wenn der undefinierte Wert als Parameter auftritt, beispielsweise gilt "⊥ * *2* = ⊥" oder "*first (3, ⊥) = 3*". Ist in einem solchen Fall das Ergebnis einer Funktion stets der undefinierte Wert, so wird die Funktion als *strikt* bezeichnet. In vielen funktionalen Programmiersprachen können auch nicht-strikte Funktionen wie *first* definiert werden, weil Ausdrücke mittels so genannter *lazy-evaluation* ausgewertet werden. Die Auswertungsregel *lazy-evaluation* gibt vor, daß immer der äußerste Ausdruck zuerst ausgewertet wird, so daß es nicht zur Auswertung des beinhalteten undefinierten Werts kommen muß.

4.1.1.2 Algebraische Datentypen

Funktionale Sprachen bieten im allgemeinen eine Auswahl der in Abschnitt 2.2 angeführten primitiven Typen, auch elementare Wertetypen genannt (vgl. [BIR92]), als eingebaute Typen an, um daraus benutzerdefinierte Typen ableiten zu können. Diese Auswahl beinhaltet in allen gängigen funktionalen Sprachen Typen für Zahlen, Wahrheitswerte und Zeichen. Zudem werden in funktionalen Sprachen mittels Tupel- und Listentypen einfache aggregierte Typen bereitgestellt. Diese Tupel- und Listentypen bilden in Kombination mit den primitiven Datentypen die Basis zur Definition neuer benutzerdefinierter Typen, welche *algebraische Datentypen* ge-

nannt werden. Im folgenden wird dargestellt, wie solche algebraischen Datentypen definiert werden können.

Algebraische Datentypen bestehen ähnlich wie Tupel aus einer festen Anzahl von Feldern, welche unterschiedlichen Typs sein können. Im Unterschied zu Tupeln und Listen werden für algebraische Datentypen eigene Konstruktoren definiert, anstatt vordefinierte Repräsentationen zu verwenden. Bei der Erstellung algebraischer Typen können sowohl einfache als auch sogenannte *alternative Konstruktoren* eingesetzt werden. Zudem können Aufzählungstypen definiert werden. Diese Möglichkeiten werden im folgenden genauer dargestellt (vgl. [REA89], [PLA93] und [HOL91]). Die verwendeten Beispiele sind in den folgenden Abschnitten in der Notation der Programmiersprache *Miranda* angegeben.

Einfache algebraische Typen

Einfache algebraische Typen weisen einen einzigen Konstruktor auf, welcher in der Regel ein oder mehrere Argumente hat. Ein solcher Konstruktor dient dazu, Werte des Typs zu erzeugen. Ein Konstruktor ohne Argumente wird als Konstante bezeichnet.
Ein Beispiel für einen einfachen algebraischen Typ:

```
schulklasse ::= Schulklasse num char [char]
```

Werte dieses Typs setzen sich zusammen aus einer Zahl, welche die Klassenstufe angibt, einem Zeichen zur Unterscheidung der Klassen gleicher Stufe, und dem Namen des Klassenlehrers, welcher als Zeichenkette repräsentiert wird. Mit Hilfe des Konstruktors *Schulklasse* läßt sich beispielsweise aus den Argumenten *10*, '*b*' und "*Meyer*" ein Wert des Typs *schulklasse* erzeugen: *Schulklasse 10 'b' "Meyer"*. Diese Notation dient nicht nur zum Erzeugen, sondern auch zum Ausgeben des betreffenden Werts.
Der Konstruktorname hat außerdem eine typunterscheidende Funktion. Konstruktornamen müssen immer derart vergeben werden, daß sie für einen Wert eindeutig bestimmen, welchen Typ er hat. Deswegen dürfen keine Typen übereinstimmende Konstruktoren besitzen.

Typen mit alternativen Konstruktoren

Eine Typdefinition kann mehrere Konstruktoren einführen, welche sich auch in bezug auf ihre Argumente unterscheiden können. Dies erlaubt die Erzeugung von Werten, die sich in ihrer Repräsentation unterscheiden, jedoch trotzdem zum selben Typ gehören.
Ein Beispiel für einen algebraischen Typ mit alternativen Konstruktoren:

```
person ::= Kind [char] num schulklasse |
           Erwachsener [char] num [char]
```

Werte vom Typ *person* sind also entweder konstruierbar durch den Konstruktor *Kind*, welcher den Namen, das Alter und die Schulklasse als Argumente aufweist,

oder durch den Konstruktor *Erwachsener*, welcher statt dem Argument *schulklasse* eine Zeichenkette zur Angabe des Berufs enthält.
Bei der Verwendung von Typen mit alternativen Konstruktoren müssen bei Funktionen, die als Argument einen Wert eines solchen Typs erhalten, alle Konstruktoren dieses Typs berücksichtigt werden. So muß eine Funktion zum Bestimmen des Namens einer Person, die einen Wert vom Typ *person* erhält, folgende Fallunterscheidung enthalten:

```
nameVon (Kind name alter schulklasse) = name
nameVon (Erwachsener name alter beruf) = name
```

Dieses Beispiel illustriert zudem das sogenannte *Pattern Matching*, welcher ein wichtiges Konzept bei der Funktionsdefinition in funktionalen Sprachen darstellt. Ein Muster ist aus Konstruktoren und formalen Parametern zusammengesetzt und erlaubt es, auf die einzelnen Komponenten eines strukturierten Werts zuzugreifen (vgl. [HOL91]). Die Struktur eines übergebenen Werts wird dazu mit dem definierten Muster abgeglichen. Wird beispielsweise die Funktion *nameVon* mit dem Parameter *Kind "Kevin" 10 "4b"* aufgerufen, so wird dieser Parameter mit dem Muster *Kind name alter schulklasse* abgeglichen, so daß der lokale Parameter *name* mit dem Wert *"Kevin"* in Verbindung gebracht wird. Dadurch kann der Wert "*Kevin*" als Ergebnis der Funktion *nameVon* zurückgeliefert werden.

Aufzählungstypen

Aufzählungstypen sind Typen, deren Werte explizit aufgezählt werden. Solche Typen können mit Hilfe der oben genannten alternativen Konstruktoren definiert werden. Die Werte des Aufzählungstyps werden durch Konstruktoren ohne Argumente angegeben, wie beispielsweise beim Aufzählungstyp *wochentag*:

```
wochentag ::= Mo | Di | Mi | Do | Fr | Sa | So
```

Die Werte eines Aufzählungstyps können in der Fallunterscheidung von Mustern (*pattern matching*) eingesetzt werden:

```
istWochenende (Sa) = true
istWochenende (Mo) = false
```

Darüber hinaus können die Werte eines Aufzählungstyps auf Gleichheit getestet werden. Andere Relationen sind auf Aufzählungstypen nicht definiert, insbesondere keine Ordnungsrelation: daher kann keine Aussage darüber getroffen werden, ob beispielsweise "*Mo*<*Mi*" gilt.

Zusammenfassung

Die in diesem Abschnitt kurz dargestellten Typdefinitionsmechanismen funktionaler Sprachen reichen bereits aus, um die meisten benutzerdefinierten Werttypen zu modellieren. Beispielsweise ist es auf einfache Weise möglich, mit Hilfe von Konstruktoren eine Folge von Typen zu einem neuen Werttyp zusammenzusetzen.

Zudem können Aufzählungstypen ohne viel Aufwand durch die Benutzung von alternativen Konstruktoren definiert werden.
Darüberhinaus können in funktionalen Programmiersprachen mit Hilfe fortgeschrittener Typkonzepte wie dem der *rekursiven* und der *polymorphen* Typen auch Typen definiert werden, die komplexe Datenstrukturen von variablen Elementen repräsentieren.
Zusammenfassend läßt sich also sagen, daß funktionale Programmiersprachen die Möglichkeit bieten, neue Werttypen durch algebraische Datentypen auf einfache Weise zu konstruieren. Da den im Rahmen dieser Diplomarbeit untersuchten objektorientierten Sprachen ein solches intuitives Sprachmittel zur Konstruktion benutzerdefinierter Werttypen fehlt, liegt der Vorschlag nahe, das Konzept der algebraischen Datentypen in objektorientierte Sprachen zu integrieren. Dieser Ansatz wurde bei der Programmiersprache Pizza verfolgt, welche im folgenden Abschnitt vorgestellt wird.

4.1.2 Die Programmiersprache Pizza

Die Programmiersprache *Pizza* ist eine funktionale Erweiterung der objektorientierten Sprache Java (vgl. [ODE97]). Sie erweitert Java um das Konzept des parametrisierten Polymorphismus sowie um zwei Konzepte aus funktionalen Sprachen: Funktionen höherer Ordnung und algebraische Datentypen.
Zur Definition algebraischer Datentypen wird in Pizza eine Klasse erstellt. Die Konstruktoren des Typs werden dann mittels *Case*-Anweisungen und die Funktionen mit Hilfe von P*attern Matching* realisiert. Ein Datentyp *Uhrzeit* könnte in Pizza beispielsweise folgendermaßen aussehen:

```
class Uhrzeit
{
   case Euro_UhrzeitKonstruktor (int stunde, int minute);
   case US_UhrzeitKonstruktor (int hour, int min, String ampm);

   int stunden_spaeter(int anzahl)
   {
      switch (this)
      {
         case Euro_UhrzeitKonstruktor (stunde, minute) :
            return Euro_UhrzeitKonstruktor((stunde + anzahl)%24,
              minute);

         case US_UhrzeitKonstruktor (hour, min, ampm) :
            String ap = ampm;
            if ((hour+anzahl)\12)%2 == 1)
            {
              ap = (ampm == "am"? "pm": "am");
            }
            return US_UhrzeitKonstruktor(((
              (hour + anzahl)-1)%12)+1, min, ap)
      }
   }
}
```

Objekte vom Typ *Uhrzeit* können dann auf intuitive Weise mit Hilfe von Konstruktoren definiert werden:

```
Uhrzeit u1 = Euro_UhrzeitKonstruktor(5, 23);
Uhrzeit u2 = US_UhrzeitKonstruktor (5, 23, "am");
```

Der Pizza-Code wird zunächst in Java-Code transformiert, woraus anschließend Code für die *Java Virtual Machine* (*JVM*) erzeugt werden kann. Die Erzeugung von Code für die JVM bietet sich deswegen an, weil die JVM auf vielen Plattformen verfügbar ist und zudem der Zugriff auf die umfangreichen Java-Bibliotheken möglich wird. Der Zwischenschritt der Erzeugung von Java-Code wurde eingeführt, damit die Mechanismen von Pizza verständlicher werden.
Bei der Übersetzung wird zum einen eine Oberklasse erzeugt und zum anderen für jeden Konstruktor eine eigene Klasse abgeleitet. Dabei wird bei den Operationen der Oberklasse mit Hilfe einer Markierung (engl.: *tag*) festgestellt, mit welchem Konstruktor das aktuelle Objekt angelegt wurde; das heißt, zu welcher Unterklasse das aktuelle Objekt gehört. Solche Markierungen sind im folgenden Beispiel *euro_tag* und *us_tag*.
Der Pizza-Code des vorigen Beispiels wird also in folgenden Java-Code übersetzt:

```
class Uhrzeit
{
   final int euro_tag = 0;
   final int us_tag = 1;
   int tag;  // dieses Attribut enthält eins der
             // beiden oben definierten tags

   Uhrzeit stunden_spaeter (int anzahl)
   {
      switch(this.tag)
      {
         case euro_tag:
            int stunde = ((Euro_UhrzeitKonstruktor)this).stunde;
            int minute = ((Euro_UhrzeitKonstruktor)this).minute;
            return new Euro_UhrzeitKonstruktor(
               (stunde + anzahl)%24, minute);

         case us_tag:
            int hour = ((US_UhrzeitKonstruktor).this).hour;
            int min = ((US_UhrzeitKonstruktor)this).min;
            String ampm = ((US_UhrzeitKonstruktor).this).ampm;
            String ap = ampm;
            if ((hour+anzahl)\12)%2 == 1)
            {
               ap = (ampm == "am"? "pm": "am");
            }
            return new US_UhrzeitKonstruktor(((
               (hour + anzahl)-1)%12)+1, min, ap)
      }
   }
}
```

```
// Die beiden Unterklassen zur Repräsentation der Konstruktoren:

class Euro_UhrzeitKonstruktor extends Uhrzeit
{
   int stunde;
   int minute;

   Euro_UhrzeitKonstruktor(int stunde, int minute)
   {
      this.tag = euro_tag;   // kennzeichnet dieses Objekt
                             // als Euro_Uhrzeit-Objekt
      this.stunde = stunde;
      this.minute = minute;
   }
}

class US_UhrzeitKonstruktor extends Uhrzeit
{
   int hour,
   int min:
   String ampm;

   US_UhrzeitKonstruktor(int hour, int min, String ampm)
   {
      this.tag = us_tag;    // kennzeichnet dieses Objekt
                            // als US_Uhrzeit-Objekt
      this.hour = hour;
      this.min = min;
   }
}
```

Wie am obigen Beispiel zu erkennen ist, enthalten auch die in Java übersetzten Klassen keine verändernden Methoden, so daß deren Objekte Wertcharakter besitzen.
Der grundlegende Unterschied von algebraischen Datentypen in Pizza und normalen Java-Klassen besteht darin, daß keine Exemplarvariablen definiert werden. Der Zustand von Exemplaren algebraischer Datentypen kann über die *case*-Anweisungen zugegriffen, aber nicht verändert werden. Innerhalb der *case*-Anweisungen wird das *Pattern Matching* realisiert, indem die Attributwerte des betreffenden Konstruktors an lokale Variablen zugewiesen werden. Eine Änderung auf einer Variablen im Quellcode des algebraischen Datentyps bewirkt daher lediglich eine Änderung auf einer lokalen Kopie des Attributs. Allerdings ist die Unveränderlichkeit aus zwei Gründen dennoch nicht sichergestellt. Zum einen könnte ein Attribut einen Referenztyp haben, was zur Folge hat, daß die lokale Kopie auf dasselbe Objekt wie das Attribut verweist. Zum anderen ist es möglich, direkt auf die Attribute der Unterklassen schreibend zuzugreifen, da diese weder als *final* noch als *private* deklariert sind.

4.2 Typspezifikationen mittels XML-Schema

Eine spezielle Art, Datentypen zu spezifizieren, bietet die Spezifikationssprache *XML-Schema*. Diese erlaubt es, Datentypen für den Einsatz in XML-Dokumenten zu definieren. Auf diesen Datentypen können keine Methoden definiert werden. Daher

sind sie frei von veränderlichen Aspekten und haben somit Wertcharakter. Durch das Fehlen von Methoden sind die XML-Schema-Datentypen allerdings keine Typen im Sinne der klassischen Typdefinition.
In diesem Abschnitt soll erläutert werden, welche grundlegenden Mechanismen die Sprache XML-Schema besitzt, um Datentypen zu definieren (Abschnitt 4.2.2) und inwiefern XML-Schema verwendet werden kann, um Werttypen in Programmiersprachen abzubilden (Abschnitt 4.2.3).
Zunächst müssen allerdings die Grundlagen von XML beschrieben werden, da sich XML Schema auf XML bezieht und in XML notiert wird.

4.2.1 Die Beschreibungssprache XML

Die Beschreibungssprache *XML* (*Extensible Markup Language*) ist eine Sprache zum Austausch von strukturierten, textbasierten Daten zwischen Softwaresystemen, wobei die Daten von ihrer Repräsentation getrennt werden (vgl. [KAY00]). Dies ermöglicht es einem empfangenden Softwaresystem, die Informationen entsprechend der lokalen Anforderungen weiterzuverarbeiten und zu repräsentieren.
XML entwickelte sich, ebenso wie HTML, aus der *Standard Generalized Markup Language* (*SGML*). Im Unterschied zu HTML können in XML eigene Markierungen (engl.: *tags*) definiert werden, wodurch sich diese Sprache vielseitig einsetzen läßt (vgl. [MAR99]). Die erste Version des XML-Standards wurde im Februar 1998 von dem *World Wide Web Consortium* (*W3C*) herausgegeben; einem internationalen Firmenkonsortium, welches offene Standards für das Netz entwickelt, um die Entwicklung des Netzes in verschiedene konkurrierende Richtungen zu verhindern (vgl. [MAR99]).

Die Sprache XML kann zum Strukturieren von Daten verwendet werden. Dazu werden Elemente definiert, die durch Markierungen begrenzt werden, welche in spitzen Klammern notiert werden. Die Start- und Endmarkierungen eines Elements müssen denselben Namen haben, wobei dem Namen der Endmarkierung ein Schrägstrich vorangestellt wird.
Elemente können hierarchisch ineinander geschachtelt werden und Attribute besitzen. Das folgende Beispiel zeigt ein Element namens *Adresse*, welches die Elemente *Strasse*, *PLZ*, und *Ort* enthält. Das Element *Adresse* besitzt zudem das Attribut *format*.

```
<ADRESSE format="deutsch">
  <STRASSE>Hauptstraße 111</STRASSE>
  <PLZ>22222</PLZ>
  <ORT>Hamburg</ORT>
</ADRESSE>
```

Mittels solcher Elemente können Daten Begriffe zugeordnet werden. Die Daten aus obigem Beispiel können so von einem Programm als Adresse erkannt werden. Über den genauen Aufbau von XML-Dokumenten gibt die Spezifikation des *World Wide Web Consortium* Aufschluß (vgl. [XML00]).

4.2.2 Verwendung von XML-Schema

Die XML-Spezifikation des *World Wide Web Consortium* beschreibt lediglich den syntaktischen Aufbau von XML-Dokumenten. Sie gibt hingegen nicht vor, welche Elemente in Dokumenten vorkommen dürfen und müssen, wie beispielsweise die *Hypertext Markup Language* (HTML). Ein Dokument, das dem rein syntaktischen Aufbau entspricht, wird *wohlgeformt* genannt.
Darüber hinaus wird ein Dokument *gültig* genannt, wenn es einer ihm zugeordneten *Document Type Definition* (DTD) entspricht. Eine DTD gibt an, welche Elemente in einem Dokument auftreten müssen und beschreibt den Aufbau und Inhalt der einzelnen Elemente. Durch DTDs kann also beispielsweise bestimmt werden, daß ein Dokument ein Element *Adresse* beinhalten muß, welches wiederum die Elemente *Strasse, PLZ* und *Ort* umfaßt.
Mittels einer *Document Type Definition* läßt sich allerdings der Inhalt von Elementen teilweise nur ungenau spezifizieren. Beispielsweise kann der Inhalt des Elements *PLZ* lediglich als Zeichenkette (#PCDATA) definiert werden. Es ist nicht möglich, den Inhalt eines Elements als Zahl oder als fünfstellige Ziffernfolge zu definieren. Um solche Definitionen vornehmen zu können, wurde die Sprache XML-Schema eingeführt. XML-Schema kann wie die *Document Type Definitions* zur Gültigkeitsprüfung von XML-Dokumenten benutzt werden. Sie bietet aber über Spezifizierung des Aufbaus von Elementen hinaus Mechanismen zur Datentypspezifizierung. Diese sollen im folgenden ansatzweise beschrieben werden (vgl. [XML01]).

XML-Schema definiert primitive Datentypen, die abgeleitet und zur Definition von einfachen und komplexen Datentypen verwendet werden können. Die wichtigsten der von XML-Schema definierten primitiven Datentypen sind in Tabelle 4-1 dargestellt.

XML-Schema-Typ	Bedeutung
String	Zeichenketten
Boolean	Wahrheitswerte
Decimal	Dezimalzahlen
Float	Gleitkommazahl einfacher Präzision
Double	Gleitkommazahl doppelter Präzision
Duration	Zeitdauer
Time	Uhrzeit
date	Datum

Tabelle 4-1: XML-Schema-Datentypen

Darauf aufbauend gibt es in XML-Schema weitere eingebaute Datentypen, die von anderen Datentypen abgeleitet sind und deren Wertebereich einschränken. Die wichtigsten sind die eingebauten Datentypen, die von *decimal* abgeleitet sind, so wie *integer* (Ganzzahlen ohne beschränkten Wertebereich)*, int* (entspricht einer Ganzzahl mit 32 Bit) und *long* (einspricht einer Ganzzahl mit 64 Bit).
Im Vergleich mit der Typkategorisierung aus Abschnitt 2.2 fällt auf, daß XML-Schema keine eingebauten Datentypen für komplexe oder rationale Zahlen vorgesieht. Der Grund dafür könnte sein, daß diese Typen bei den weit verbreiteten Pro-

grammiersprachen eine untergeordnete Rolle spielen, da der Anwendungsbereich selten mathematisch ausgerichtet ist.
Ferner ist zu beachten, daß das Ableiten von Werttypen bei XML-Schema-Datentypen kein Problem darstellt, da diese Datentypen nicht mit Methoden verknüpft sind. Es besteht daher nicht die Gefahr, daß Methoden eines unveränderlichen Typ von Unterklassen mit verändernden Methoden überschrieben werden (vgl. Abschnitt 3.4.2.1). Der abgeleitete Typ gibt lediglich den Wertebereich vor, der anschließend weiter eingeschränkt werden kann.

Zur Definition eines einfachen Datentyps wird das Element *simpleType* benutzt. Dieses kann dann auf verschiedene Weise beschränkt werden. Von besonderem Interesse ist dabei das Element *restriction*. Damit werden zwei Effekte erzielt: Erstens kann ein Basistyp angegeben werden, der dem neu zu definierenden Typ zugrunde liegt. Zweitens können innerhalb des *restriction* Elements weitere Unterelemente benutzt werden, um den Wertebereich des Basistyps einzuschränken. Beispielsweise kann ein Minimumwert mittels des Elements *minInclusive* angegeben werden:

```
<simpleType name="prozent">
 <restriction base="decimal">
  <fractionDigits value="2"/>
  <minInclusive value="0.00"/>
  <maxInclusive value="100.00"/>
 </restriction>
</simpleType>
```

Zum Beschränken der Wertebereiche können folgende Aspekte (*Facets*) aus Tabelle 4-2 benutzt werden.

Aspekt	Beschreibung
Length	Exakte Länge
MinLength	Minimale Länge
MaxLength	Maximale Länge
Pattern	Kontextfreies Muster
Enumeration	Aufzählung zulässiger Werte
WhiteSpace	Gibt an, wie Leerzeichen vor der Validierung normalisiert werden. Zulässige Werte: *preserve*, *replace*, *collapse*
MaxInclusive	Höchstwert, angegebener Wert eingeschlossen
MaxExclusive	Höchstwert, angegebener Wert ausgeschlossen
MinInclusive	Tiefstwert, angegebener Wert eingeschlossen
MinExclusive	Tiefstwert, angegebener Wert ausgeschlossen
TotalDigits	Maximale Anzahl an Stellen bei Dezimalzahlen
FractionDigits	Maximale Anzahl an Nachkommastellen bei Dezimalzahlen

Tabelle 4-2: Einschränkungsmöglichkeiten bei XML-Schema

Je nach Basistyp sind unterschiedliche Einschränkungen möglich. Beispielsweise sind beim Basistyp *decimal* unter anderen die Einschränkungen *pattern* oder *length* nicht möglich, während beim Basistyp *string* die Einschränkungen *maxInclusive* und *fractionDigits* nicht benutzt werden können.

Neben den zuvor besprochenen einfachen Typen können auch komplexe Typen (*complexTypes*) definiert werden, welche insbesondere dazu dienen, Elemente zu definieren, die Attribute oder Subelemente besitzen. Das Element *Adresse* (vgl. Abschnitt 4.2.1) kann beispielsweise folgendermaßen als komplexer Typ definiert werden:

```
<complexType name="Adresse">
  <sequence>
    <element name="strasse" type="string"/>
    <element name="plz" type="plzTyp"/>
    <element name="ort" type="string"/>
  </sequence>
  <attribute name="format" type="formatTyp"/>
</complexType>
```

Für jedes Subelement muß neben dem Namen dessen Typ angegeben werden, welcher wiederum einfach oder komplex sein kann.

4.2.3 Wertebereichsvalidierung mittels XML-Schema

XML-Schema Dateien geben keine programmiersprachlichen Typen an, sondern spezifizieren Aufbau und Inhalt von Elementen in XML-Dokumenten. Es ist jedoch möglich, XML-Schema-Datentypen auf programmiersprachliche Typen abzubilden. Ein Ansatz, der in diese Richtung zielt, soll in diesem Abschnitt beschrieben werden (vgl. [MCL00]). Dieser Ansatz widmet sich der Problematik der Datenvalidierung, welche beispielsweise benötigt wird, um die Gültigkeit von Einträgen auf Formularen prüfen. Solche Gültigkeitsüberprüfungen von Daten werden oftmals gänzlich ignoriert oder durch entsprechende Routinen auf nicht wiederverwendbare Weise für den konkreten Anwendungsfall programmiert. Letzteres führt zu einer beträchtlichen Codeduplizierung, wenn ein Formular große Mengen zu prüfender Felder aufweist. Ferner wirden bei dieser Lösung die Prüfroutinen nicht von der Anwendungslogik getrennt, wodurch die Lesbarkeit von Programmen beeinträchtigt wird.
Die in [MCL00] vorgeschlagene Lösung benutzt XML-Schema-Datentypen um anzugeben, welche Werte in einem Feld eines Formulars gültig sein sollen. Dadurch werden die Angaben über einzuhaltende Wertebereiche in separate Dateien ausgelagert, so daß im Quellcode der Anwendungslogik keine Prüfroutinen mehr auftreten müssen. XML-Schema kann prinzipiell auf zwei unterschiedliche Arten zur Gültigkeitsprüfung verwendet werden:

1.) indem ein XML-Dokument überprüft wird, welches zur Laufzeit aus dem zu überprüfenden Wert erzeugt wurde.
2.) indem dynamisch ein Objekt erzeugt wird, das zur Gültigkeitsprüfung eines Werts benutzt werden kann.

In [MCL00] wird der zweite Ansatz gewählt. Dabei wird auf Basis einer Datentypspezifikation in XML-Schema ein *Constraint*-Objekt generiert, das einen einfachen Zugriff auf den Inhalt des XML-Schema-Datentyps bereitstellt. *Constraint*-Objekte repräsentieren also die Einschränkungen eines XML-Schema-Datentyps. Die Methoden der *Constraint*-Objekte beschränken sich auf gib- und setze-Methoden für die einzelnen Einschränkungen und den Basistyp, also beispielsweise *setMinInclusive(double)* und *getMinInclusive()*. Mit Hilfe dieses *Constraint*-Objekts können Werte auf ihre Gültigkeit überprüft werden. Für diesen Ansatz werden neben einer *Constraint*-Klasse die folgenden Klassen benötigt:

- *SchemaParser*
  Der *SchemaParser* initialisiert ein *Constraint*-Objekt. Dazu parst er ein XML-Schema-Dokument und ruft dessen setze-Methoden auf. Wenn beispielsweise ein XML-Schema-Datentyp ein Element namens *maxInclusive* enthält, dessen Wert *100* ist, so wird bei dem entsprechenden *Constraint*-Objekt die Methode *setMaxInclusive(100)* aufgerufen.

- *Validator*
  Das Herz eines *Validator*-Objekts ist die *isValid*-Methode. Diese prüft, ob eine gegebene Zeichenkette einem *Constraint*-Objekt entspricht. Dazu prüft sie zuerst, ob der String in den vom *Constraint*-Objekt vorgegebenen Datentyp umgewandelt werden kann. Danach wird geprüft, ob keine der festgelegten Einschränkungen verletzt sind.

- *DataConverter*
  Da die XML-Schema-Datentyen nicht dieselben sind wie die Java-Datentypen, wird der *DataConverter* benötigt, der die XML-Schema-Datentypen in Java-Datentypen umsetzt. Dazu stellt dieser eine Methode bereit, die den im XML-Schema-Dokument gespeicherten Typ als Zeichenkette übergeben bekommt und den entsprechenden Java-Typ als Zeichenkette zurückgibt.

Zusammenfassend betrachtet basiert der Ansatz auf der Idee, einen XML-Schema-Datentyp in ein *Constraint*-Objekt zum Überprüfen von Werten umzuwandeln. Da ein solches Objekt durch den *SchemaParser* zur Laufzeit erstellt wird, kann bei diesem Ansatz der Wertebereich eines Formularfeldes angepaßt werden, ohne die Anwendung neu kompilieren zu müssen. Es genügt, das zugehörige XML-Schema-Dokument zu ändern.
Wenn auf diese dynamische Anpaßbarkeit nicht besonderer Wert gelegt wird, ist noch eine andere Umsetzung des XML-Schema-Datentyps denkbar: Aus diesem kann statisch eine Klasse generiert werden, die die nötigen Einschränkungen kapselt und eine Methode zum Prüfen eines Werts anbietet.
Solche Klassen, die Wertebereichseinschränkungen und Methoden zur Gültigkeitsprüfung enthalten, werden auch in dem Ansatz verwendet, welcher im Verlauf des folgenden Abschnitts beschrieben wird.

4.3 Einsatz einer Klassenarchitektur

Um die Modellierung von Werttypen in objektorientierten Sprachen zu ermöglichen, kann neben der Einbettung funktionaler Konzepte und der Verwendung von XML-Schema auch eine Klassenarchitektur eingesetzt werden.
Dies hat den Vorteil, daß zur Implementierung von Werttypen keine neue Programmiersprache erlernt werden muß. Bei Verwendung objektorientierter Sprachen müssen allerdings die in Kapitel 3 beschriebenen Probleme beachtet werden. Einige der dort dargestellten Aspekte der Implementierung von Werttypen, wie die Speicherplatzersparnis durch Benutzung des *Fliegengewicht*-Musters (vgl. Abschnitt 3.5.2), können wiederverwendbar gemacht werden, indem sie in ein Rahmenwerk (vgl. Abschnitt 4.3.1) integriert werden. Dieser Ansatz wurde bereits in dem JWAM-Rahmenwerk realisiert (vgl. Abschnitt 4.3.2). Es wird untersucht, welche Erfahrungen sich für Benutzer dieses Rahmenwerks bei der Entwicklung von Fachwerten ergeben (vgl. Abschnitt 4.3.2).

4.3.1 Rahmenwerke allgemein

Ein Rahmenwerk ist „eine Architektur aus Klassenhierarchien, die eine allgemeine generische Lösung für ähnliche Probleme in einem bestimmten Kontext vorgibt“ (vgl. [OBJ99]). Die Anwendungsentwicklung mittels eines Rahmenwerks erfolgt durch die Ableitung und Anpassung der vorgegebenen Klassen. Dabei werden der vom Rahmenwerk vorgegebene Kontrollfluß und das Systemdesign übernommen.
Durch Rahmenwerke werden einige klassische Probleme der Softwareentwicklung adressiert (vgl. [FAY97]). Sie vermeiden zum einen hohe Kosten, indem sie wiederverwendbare Komponenten bereitstellen. Dadurch müssen Kernkonzepte nicht ständig neu entwickelt werden. Zum anderen kann es durch Einsatz eines Rahmenwerkes vermieden werden, komplexe, effiziente und portable Systeme ganz neu entwickeln zu müssen.

4.3.2 Das JWAM-Rahmenwerk und seine Umsetzung des Fachwertkonzepts

Das im Rahmen dieser Diplomarbeit betrachtete JWAM-Rahmenwerk wurde am Arbeitsbereich Softwaretechnik der Universität Hamburg entwickelt (vgl. [ZÜL98]). Es dient zur Entwicklung von Anwendungen in Java (J) nach den Prinzipien des Werkzeug/Automaten/Material-Ansatzes (WAM). Der WAM-Ansatz zielt darauf ab, Experten unterschiedlicher Anwendungsbereiche bei ihrer eigenverantwortlichen Tätigkeit zu unterstützen. Das JWAM-Rahmenwerk unterstützt daher jene Anteile, die jenseits eines konkreten Anwendungsbereichs liegen (vgl. [OBJ99]).
Im JWAM-Rahmenwerk werden Oberklassen für die Konstruktion von Materialien und Werkzeugen bereitgestellt. Zudem gibt es Oberklassen für die Erstellung sogenannter Fachwerttypen (vgl. Abschnitt 2.4), da bei der Entwicklung nach dem WAM-Ansatz die Modellierung von anwendungsspezifischen Werttypen einen hohen Stellenwert hat. Die Umsetzung des Fachwertkonzepts im JWAM-Rahmenwerk (vgl. [MÜL99]) wird im folgenden genauer dargestellt. Die Beschreibung richtet sich nach der Version 1.6.0 des JWAM-Rahmenwerks.

Das Konzept der Fachwerttypen im JWAM-Rahmenwerk wurde aufgrund von bestimmten Zielsetzungen entwickelt. Auf folgende Aspekte wurde dabei besonderer Wert gelegt:

- **Bereitstellung von Konstruktoren**
  Für Fachwerte ist es wichtig, daß sie geeignet erzeugt werden können. Ebenso wie bei den Werttypen der funktionalen Programmiersprachen (vgl. Abschnitt 4.1.1) ist es sinnvoll, Fachwerte direkt aus ihren Attributen zu erzeugen. Hierzu wird ein Konstruktor bereitgestellt, der für jedes Attribut des zugehörigen Fachwerttyps einen Parameter enthält. Zudem ist es von Vorteil, einen Fachwert aus einer Zeichenkette erzeugen zu können. Ein solcher Konstruktor wird in der Regel bereitgestellt, damit der Inhalt von Textfeldern auf einfache Weise in Fachwerte umgewandelt werden kann.

- **Eingrenzung des Wertebereichs**
  Da ein Typ wesentlich durch seinen Wertebereich geprägt ist, muß eindeutig definiert sein, welche Werte für einen gegebenen Fachwerttyp gültig sind. Zu diesem Zweck stellen Fachwerttypen im JWAM-Rahmenwerk eine Funktion zum Überprüfen der Gültigkeit eines Fachwerts bereit. Für jeden Konstruktor eines Fachwerttyps muß eine entsprechende Funktion zur Gültigkeitsüberprüfung definiert sein, die vor dem Erzeugen eines Fachwerts ausgeführt wird.

- **Zugriff auf den beinhalteten Wert**
  Da der innere Zustand eines Fachwerts verborgen bleiben soll (vgl. Abschnitt 2.4), müssen zur Benutzung von Fachwerten Methoden zum Zugriff auf deren gespeicherten Wert definiert sein. Dieser Zugriff kann zum einen über sogenannte *Selektoren* erfolgen, welche die Werte der einzelnen Attribute zurückliefern. Zum anderen wird zur Darstellung des gesamten Fachwerts, beispielsweise in Fenstersystemen, eine Methode benötigt, die den Fachwert als Zeichenkette ausgibt. Eine solche Methode muß in jedem Fachwerttyp definiert sein.

- **Definition fachlicher Funktionen**
  Um einen Fachwert im vorgesehenen fachlichen Kontext verwenden zu können, muß es möglich sein, entsprechende fachliche Funktionen zu definieren. Solche fachlichen Funktionen sind speziell auf das Anwendungsgebiet des jeweilgen Fachwerttyps ausgerichtet. Beispielsweise könnte ein Fachwerttyp *dvUhrzeit* die fachliche Funktion *stundeSpaeter()* enthalten – für einen Fachwerttyp wie *dvKontonummer* würde eine solche Methode hingegen keinen Sinn machen.

- **Verhalten nach Wertsemantik**
  Variablen eines Fachwerttyps sollten Wertsemantik besitzen und die referentielle Transparenz sicherstellen. Im JWAM-Rahmenwerk wird dazu der Ansatz der unveränderlichen Objekte (vgl. Abschnitt 3.4) gewählt. Daß keine verändernden Methoden bei Fachwerttypen definiert werden, muß dabei durch entsprechende Programmierkonventionen gewährleistet werden. Das Problem des hohen Spei-

cherbedarfs von unveränderlichen Fachwert-Objekten wird im JWAM-Rahmenwerk durch das Einsatz des *Fliegengewicht*-Musters entschärft (vgl. Abschnitt 3.5.2).

- **Undefinierte Werte**
  Fachwerttypen sollten einen undefinierten Wert besitzen, welcher den undefinierten Werten bei funktionalen Programmiersprachen ähnelt (vgl. Abschnitt 4.1.1.1). Dieser wurde eingeführt, weil die Wertemenge fachlicher Werttypen in der Regel begrenzt ist. Wenn ein Benutzer beispielsweise in einem Eingabefeld eine Zeichenkette angibt, die keinen gültigen Wert des geforderten Fachwerttyps darstellt, kann kein entsprechender Fachwert initialisiert werden. Zur Kennzeichnung eines solchen Ausnahmefalls können verschiedene *Ausnahmewerte* verwendet werden (vgl. [CUN95]). Im JWAM-Rahmenwerk wird zur Kennzeichnung eines solchen Ausnahmefalls der sogenannte *undefinierte Wert* benutzt. Ausnahmewerte unterscheiden sich von anderen Werten dadurch, daß der Einsatz dieses Wertes besondere Verhaltensweisen zur Folge haben kann. Methoden, die auf dem undefinierten Wert ausgeführt werden, führen entweder zu Fehlermeldungen oder liefern wiederum den undefinierten Wert zurück. Als Ausnahme gilt eine sondierende Funktion, wie *isNil()* (vgl. [CUN95]).

Zur Entwicklung von Fachwerttypen stehen im JWAM-Rahmenwerk folgende vordefinierte Klassen bereit (vgl. Abbildung 4-1):

- **DomainValue**
  Allgemeines Interface für Fachwerttypen
- **DomainValueImpl**
  Basisimplementation des DomainValue-Interface

- **DvEnumerable**
  Allgemeines Interface für aufzählbare Fachwerttypen
- **DvEnumerableImpl**
  Basisimplementation des DvEnumerable-Interface

Fehler! Keine gültige Verknüpfung.

Abbildung 4-1: Fachwert-Rahmenwerk

Die zwei Typen *DomainValue* und *DvEnumerable* sind als Java-Interfaces implementiert. Die meisten Methoden des Interface werden von den abstrakten Klassen *DomainValueImpl* und *dvEnumerableImpl* implementiert. Diese abstrakten Klassen dienen bei neu zu definierenden Fachwerttypen als Oberklassen und bieten die eigentliche Unterstützung bei der Entwicklung von konkreten Fachwerttypen.
Die Typen *dvEnumerable* und *dvEnumerableImpl* werden bei der Definition von aufzählbaren Fachwerttypen eingesetzt, welche über spezielle Methoden verfügen, wie beispielsweise über eine Methode zum Liefern aller gültigen Fachwerte. Eine solche Funktionalität ist bei aufzählbaren Fachwerttypen sinnvoll, da diese häufig in Auswahllisten eingesetzt werden, welche alle auswählbaren Werte anzeigen müssen. Um in einem solchen Fall Werte anzugeben, die nicht in der Auswahlliste angezeigt

werden, ist es bei aufzählbaren Fachwerttypen zudem von Vorteil, eine Methode zum Hinzufügen neuer Werte zum Wertebereich vorzusehen. Dadurch ändert sich dann der Wertebereich eines Aufzählungstyps zur Laufzeit.
Zur Erstellung von Fachwerttypen, die keine Aufzählungstypen sind, wird als Oberklasse in der Regel *DomainValueImpl* benutzt. Entsprechend der Ziele beim Entwurf des Fachwert-Rahmenwerks stellt diese Oberklasse vorgefertigte Lösungen für folgende Punkte bereit:

- **Fachwertfabrik zum Erstellen von Fachwerten**
  Das JWAM-Rahmenwerk implementiert das Fliegengewichtmuster, um den Speicheraufwand für Fachwerte zu minimieren. Die Basisfunktionalität der zugehörigen Fachwertfabrik, welche durch die Klasse *DomainValueImpl* bereitgestellt wird, umfaßt beispielsweise das Registrieren von bisher unregistrierten Fachwerten sowie den Zugriff auf bereits registrierte Fachwerte.

- **Umgang mit dem undefinierten Wert**
  Einige Methoden müssen spezielle Ergebnisse liefern, wenn sie auf einem undefinierten Wert aufgerufen werden. Diese Methoden werden durch Schablonenmethoden (vgl. [GAM95]) realisiert, die den korrekten Umgang mit undefinierten Werten sicherstellen. Beispielsweise muß die Methode zur Ausgabe eines Fachwerts als Zeichenkette (*toString*) bei undefinierten Werten die Repräsentation des undefinierten Werts zurücklieferen. In allen anderen Fällen wird die Methode *toStringImpl* aufgerufen, die in allen konkreten Unterklassen definiert sein muß.

- **Implementierung wichtiger Methoden auf Basis der Stringrepräsentation**
  Jeder Fachwerttyp besitzt eine Methode zum Darstellen seiner Exemplare als Zeichenkette. Anhand dieser Methode können beispielsweise die Methoden zum Vergleich zweier Fachwerte (*compareTo*) und zum Prüfen der Gleichheit (*equals*) bereits in der Oberklasse für Fachwerttypen implementiert werden.

Die oben beschriebenen Funktionalitäten werden vom Fachwert-Rahmenwerk für alle Fachwerttypen bereitgestellt, die von *DomainValueImpl* erben. Ein neu zu entwickelnder, konkreter Fachwerttyp muß darüber hinaus noch gegebenenfalls folgende Methoden implementieren: die Methode zum Liefern des Fachwerts als String (*toStringImpl*), eine Validierungsfunktion (*isValidImpl*), Methoden zum Erzeugen eines neuen Fachwerts und fachliche Methoden.

Zusammengefaßt erleichtert die Bereitstellung des Fachwert-Rahmenwerks im JWAM-Rahmenwerk das Entwickeln fachlicher Werttypen. Es wird ein Rahmen für die Entwicklung von Typen mit wertspezifischen Eigenschaften vorgegeben. Dazu gehören der undefinierte Wert, die Eingrenzung des Wertebereichs und das Prüfen der Gleichheit zweier Werte.
In bezug auf die Werteigenschaften der Unveränderlichkeit und der referentiellen Transparenz ist anzumerken, daß sie nur durch Programmierkonventionen sichergestellt werden: Theoretisch können in JWAM-Fachwerttypen Methoden definiert werden, die den Zustand der JWAM-Fachwerte verändern. Allerdings wird der

Umgang mit unveränderlichen Objekten dadurch unterstützt, daß der hohe Speicherbedarf, der bei ihrer Verwendung entsteht, durch den Einsatz des Fliegengewichtmusters beschränkt wird.
Im folgenden Abschnitt wird dargestellt, wie sich für Entwickler von Fachwerttypen die Arbeit mit dem JWAM-Rahmenwerk darstellt.

4.3.3 Erfahrungen von Benutzern beim Erstellen von Fachwerten mit dem JWAM-Rahmenwerk

Um herauszufinden, inwiefern das JWAM-Rahmenwerk Entwickler bei der Erstellung von Fachwerttypen unterstützt, wurden zwei Interviews mit unterschiedlich erfahrenen Entwicklern geführt. Zum einen wurde ein Entwickler befragt, der selbst am Design des JWAM-Rahmenwerks beteiligt war und daher genaue Kenntnisse über dieses Rahmenwerk hat. Die Aussagen über unerfahrene Entwickler basieren auf dem Interview mit einem Entwickler, der im Zuge einer Lehrveranstaltung mit dem JWAM-Rahmenwerk in Kontakt kam und daher nur über Grundkenntnisse im Umgang mit diesem Rahmenwerk verfügt.
Vor der Beschreibung der persönlichen Erfahrungen der beiden Entwickler wird zunächst anhand der Informationen aus beiden Interviews dargestellt, welche Schritte zur Erstellung von Fachwerttypen bei Verwendung des JWAM-Rahmenwerks notwendig sind.

4.3.3.1 Vorgehen bei der Fachwerttyperstellung mit Hilfe des JWAM-Rahmenwerks

Beim Programmieren mit dem JWAM-Rahmenwerk werden Fachwertklassen in der Regel nicht komplett neu erstellt, sondern aus Kopien vorhandener Klassen angefertigt. Besonders der im Rahmenwerk integrierte Fachwerttyp *dvString* eignet sich als Vorlage, da dieser intern einfach realisiert ist: er enthält keine fachlichen Operationen, welche nur für *dvString* verwendet werden könnten und deswegen beim Umwandeln in einen neuen Fachwert gelöscht werden müßten. Zudem wurden andere Fachwertklassen, die nicht Teil des Rahmenwerks sind, möglicherweise für eine andere Version des Rahmenwerks geschrieben, so daß bei ihrer Verwendung als Vorlage umfangreiche Änderungen notwendig werden könnten.
Trotzdem wird in der Praxis oftmals nicht unbedingt *dvString* gewählt, sondern eine Fachwertklasse aus demselben Package, in welchem der neue Fachwerttyp stehen soll. Dies hat zum einen den Vorteil, daß dann die Angabe des Packagenamens im Programmcode bereits korrekt ist, und zum anderen muß keine Zeit für die Suche nach einer Vorlage in der Verzeichnisstruktur des JWAM-Rahmenwerks investiert werden.
Für Aufzählungen, welche mit Hilfe der Oberklasse *dvEnumerable* realisiert werden, gibt es keine Rahmenwerk-Klasse, die sich als Vorlage eignen würde. Statt dessen wird eine bereits existierende Aufzählungsklasse aus einem Projekt verwendet.

Nach der Auswahl der Vorlage wird die ausgesuchte Fachwertklasse kopiert und dem neuen Fachwerttyp entsprechend umbenannt. Steht die neue Fachwertklasse in einem anderen Package als die Vorlage, so muß der im Quellcode eingetragene Name des Packages entsprechend angepaßt werden.

Als nächstes wird im Quellcode der Kopie eine Ersetzung durchgeführt, wobei jedes Vorkommen des alten Fachwerttypnamens in die neue Bezeichnung umgewandelt wird. Danach muß überprüft werden, ob die Ersetzung überall in der richtigen Weise durchgeführt wurde und an welchen Stellen eventuell fälschlicherweise ersetzt wurde. Gibt es zu viele falsche Ersetzungen, muß gegebenenfalls mit einer neuen Kopie von vorne angefangen werden.

Jeder Fachwerttyp besitzt die folgenden drei Methoden, welche gegebenenfalls angepaßt werden müssen:

1. **Die Methode *valueImpl***
   Diese Methode sucht aus einem String mittels Parsing die Informationen heraus, welche für die Erzeugung eines Fachwerts benötigt werden. Diese Umwandlung kann entweder innerhalb der Fabrik oder in der Fachwertklasse erfolgen.
2. **Die Validierungsfunktion *isValidImpl***
   Diese Methode überprüft, ob eine Stringrepräsentation einen gültigen Fachwert repräsentiert.

3. **Die Methode *toStringImpl***
   Diese Methode wandelt einen Fachwert in die entsprechende Stringrepräsentation um.

Zusätzlich müssen noch gegebenenfalls die fachlichen Methoden geändert beziehungsweise neue fachliche Methoden hinzugefügt oder überflüssige Methoden gelöscht werden. Dabei ist zu beachten, daß nicht nur in der Fachwertklasse, sondern auch in der darin enthaltenen Fabrikklasse fachliche Operationen definiert werden können.

Neben der Erstellung der eigentlichen Fachwertklasse gehören zur Konstruktion eines Fachwerttyps noch die Entwicklung weiterer Klassen. Dazu zählt zum einen die Testklasse, anhand derer die Funktionalität eines neu entwickelten Fachwerttyps überprüft werden kann. Zum anderen kann es bei besonderen Anforderungen an die Darstellungsweise des neuen Fachwerttyps notwendig werden, ein spezielles Oberflächenobjekt (*Widget*) für die Darstellung auf der Benutzungsoberfläche zu erstellen. Solche Oberflächenobjekte werden im JWAM-Rahmenwerk *Präsentationsformen* genannt. Während bei strukturierten Fachwerttypen, welche aus anderen Fachwerttypen zusammengesetzt sind, stets ein entsprechendes Oberflächenobjekt erstellt werden muß, reicht es bei nicht strukturierten Fachwerttypen aus, eine passende existierende Präsentationsform auszuwählen.

4.3.3.2 Die Arbeit mit dem JWAM-Rahmenwerk aus Sicht eines erfahrenen Entwicklers

Der erfahrene Entwickler arbeitet an Projekten, für welche er jeweils zu Beginn eine hohe Anzahl an Fachwerttypen erstellen muß, wobei er dem oben beschriebenen Vorgehen folgt. Da Fachwerttypen in vielen Aspekten oftmals nur wenig voneinander abweichen, unterscheidet sich die Implementierung verschiedener Fachwerttypen

nur in wenigen variablen Punkten. Da also für die Erstellung der für ein Projekt benötigten Fachwerttypen wiederholt sehr ähnliche Vorgänge durchgeführt werden müssen, entsteht für den Entwickler ein lästiger und potentiell vermeidbarer Arbeitsaufwand.
Eventuelle Fehler enstehen bei dieser routinemäßigen Fachwerttyperstellung durch erfahrene Entwickler eher durch Vergeßlichkeit oder Unachtsamkeit als durch Unwissen. Beispielsweise muß bei der im vorigen Abschnitt beschriebenen Namensersetzung darauf geachtet werden, daß in der Kopie der Fachwerttypvorlage keine falschen Ersetzungen erfolgen. Des weiteren wird leicht übersehen, wenn der Package-Name geändert werden muß.
Zudem müssen bei der Änderung der drei Methoden *isValidImpl*, *toStringImpl* und *StringImpl* die Konsistenzbedingungen zwischen diesen drei Methoden, dem *Hashcode* und der Methode *equals* beachtet werden, da es sonst zu schwer auffindbaren Fehlern kommen kann. Die einzuhaltenden Konsistenzbedingungen werden in der zum Fachwerttyp gehörenden Testklasse überprüft. Eine dieser Konsistenzbedingungen lautet beispielsweise: wird ein Fachwert in einen String umwandelt und dieser String wiederum in einen Fachwert, dann muß dieser neue Fachwert der gleiche sein wie der ursprüngliche.
Neben den bisher beschriebenen möglichen Fehlerquellen entsteht zusätzlicher lästiger Arbeitsaufwand bei Einführung einer neuen Rahmerwerksversion. In einem solchen Fall müssen ältere Klassen, die migriert werden sollen, gegebenenfalls geändert und angepaßt werden – und es kann passieren, daß alle Fachwerttypen neu geschrieben werden müssen. Was in einem solchen Fall geändert werden muß, ist nicht voraussehbar.

4.3.3.3 Die Arbeit mit dem JWAM-Rahmenwerk aus Sicht eines unerfahrenen Entwicklers

Für den unerfahrenen Entwickler liegen die Schwierigkeiten bei der Arbeit mit dem JWAM-Rahmenwerk in der mangelnden Kenntnis über die Zusammenhänge und Abläufe im Rahmenwerk. Die Dokumentation des Rahmenwerks und die enthaltene Anleitung zur Konstruktion von Fachwerttypen reichen nicht aus, um die Einzelheiten und genauen Zusammenhänge bei der Fachwerttypentwicklung zu durschauen. Häufig ist es zur Einarbeitung in die JWAM-Fachwerttyp-Entwicklung notwendig, Quellcode-Beispiele durchzuarbeiten, anhand derer der allgemeine Aufbau von Fachwerttypen zu erkennen ist.
Weitere Schwierigkeiten ergeben sich für den unerfahrenen Entwickler durch den komplizierten Klassenbaum des Rahmenwerks. Der Entwickler benötigt genaue Kenntnisse über den Aufbau des JWAM-Rahmenwerks, um die gesuchten Vorlagen und einzubindenden Klassen schnell finden zu können. Des weiteren sind Kenntnisse über den WAM-Ansatz für die erfolgreiche Arbeit mit dem JWAM-Rahmenwerk notwendig, da dieser die Grundlage für dieses Rahmenwerk bildet. Vor der Einarbeitung in das JWAM-Rahmenwerk steht also zunächst die Einarbeitung in die Prinzipien des WAM-Ansatzes.
Die Entwicklung eines Fachwerttyps im JWAM-Rahmenwerk erfolgt derzeit über die Verwendung der Kopie eines bereits existierenden Fachwerttyps. Dabei stellen sich für einen unerfahrenen Entwickler beispielsweise folgende Fragen:

- Welche Klasse sollte als Vorlage dienen?
- Welche Methoden des Codebeispiels können unverändert übernommen werden?
- Welche Methoden müssen angepaßt werden?
- Welche Methoden können wegfallen?
  Vom Löschen von Methoden nimmt der unerfahrene Entwickler möglichst Abstand, da er nicht genau weiß, ob sie für die Rahmenwerk-Funktionalität erhalten bleiben müssen. Dies kann dazu führen, daß nicht notwendige Methoden in der neuen Klasse verbleiben.
- Welche fachlichen Methoden müssen neu hinzugefügt werden?
- Welche Methode wird benutzt, um auf den Wert des Fachwertes zuzugreifen?

Problematisch bei der Anpassung der Fachwerttyp-Vorlage ist für den unerfahrenen Entwickler die Vielzahl der Methoden in den Fachwertklassen des Rahmenwerks. Beispielsweise besitzt die Fachwertklasse *dvString* mehrere *value*-Methoden: *value*, *valueOf* und *valueImpl*. Wenn diese Klasse als Vorlage benutzt wird, ist es für den unerfahrenen Entwickler nicht sofort offensichtlich, welche dieser Methoden angepaßt werden müssen.
Zudem ist es möglich, daß einige wichtige Aspekte der Fachwerttypentwicklung vom unerfahrenen Entwickler nicht berücksichtigt werden, so daß auf diese Weise mögliche Fehlerquellen entstehen. Es ist beispielsweise möglich, daß ihm die Notwendigkeit der Einhaltung der Konsistenzbedingungen nicht bewußt ist. Von den drei im vorigen Abschnitt erwähnten Methoden *toStringImpl, isValidImpl* und *valueImpl* werden dann eventuell nur eine oder zwei angepaßt, obwohl alle drei geändert werden müssen, um die Konsistenz zu wahren. Ebenso vergißt ein unerfahrener Entwickler eventuell die Überprüfung der Konsistenzbedingungen in der zum Fachwert gehörenden Testklasse.

4.4 Zusammenfassung

Es wurde dargestellt, daß es verschiedene Möglichkeiten gibt, die Deklaration von Werttypen in einer Sprache zu ermöglichen.
Pizza bietet das zusätzliche Sprachkonstrukt der algebraischen Datentypen, deren Deklaration von der Definition normaler Klassen abweicht.
XML-Schema bietet einfache Möglichkeiten, den Wertebereich von Datentypen zu spezifizieren. Durch eine geeignete Zusammenführung der XML-Schema-Datentypen mit einer Programmiersprache kann die Basis für Werttypen gelegt werden.
Im JWAM-Rahmenwerk wird der Ansatz verfolgt, durch geeignete Oberklassen und vorgegebene Funktionalitäten ein Werttypkonzept in die objektorientierte Sprache Java einzubauen, um Werttypen mittels normaler Java-Klassen modellieren zu können.
Im folgenden wird aufbauend auf den in diesem Kapitel dargestellten Lösungsansätzen ein eigener Ansatz entwickelt. In diesem Zusammenhang werden auch die positiven und negativen Aspekte der hier vorgestellten Ansätze beleuchtet.

5 Konzeption eines Systems zur Fachwerttyp-entwicklung

In diesem Kapitel wird zunächst herausgearbeitet, welche der in den vorangegangenen Kapiteln erwähnten Konzepte für die Erstellung von Werttypen in objektorientierten Sprachen wünschenswert wären und inwiefern die in Kapitel 4 vorgestellten Ansätze sich für die Umsetzung dieser Konzepte eignen. Auf Basis dieser Betrachtungen soll die Grundstruktur eines Systems entworfen werden, welches für eine möglichst einfache Entwicklung von Werttypen verwendet werden kann.

5.1 Unterstützbare Konzepte zur Deklaration von Werttypen in objektorientierten Programmiersprachen

In Kapitel 3 wurde dargestellt, welche Schwierigkeiten in objektorientierten Sprachen bei der Definition von Werttypen auftreten. In Abschnitt 4.1.1 wurde vorgestellt, inwiefern funktionale Sprachen in bezug auf die Deklaration von Werttypen Vorteile bieten.
In diesem Abschnitt sollen die Konzepte, die in den genannten Abschnitten erwähnt wurden, im Hinblick darauf betrachtet werden, welche zur Unterstützung der Deklaration von Werttypen in objektorientierten Sprachen dienen können. Zu den betrachteten Konzepten gehören zum einen jene, die sich auf die Deklaration von Werttypen beziehen. Diese lassen sich unterteilen in Konzepte im Zusammenhang mit dem Angeben des Definitionsbereichs und Konzepte zur Definition von Funktionen, die den Umgang mit Exemplaren dieser Werttypen festlegen:

Konzepte zum Angeben des Definitionsbereichs:
- Aufzählungstypen
- Einschränkung des Wertebereichs
- Undefinierter Wert

Konzepte zur Definition von Funktionen:
- Einfache Konstruktoren und Pattern Matching
- Angabe der Stringrepräsentation

Neben den oben genannten Konzepten sind noch die folgenden von Interesse:
- Kennzeichnung von Werttypen
- Sicherung der Unveränderlichkeit

Zusätzliches Kriterium:
- Einfache Entwicklung von Werttypen

Im weiteren Verlauf werden diese Konzepte jeweils zunächst noch einmal kurz erläutert und anschließend wird herausgearbeitet, inwiefern die in Kapitel 4 vorgestellten Lösungsansätze diese Konzepte unterstützen.

Bei einem dieser Lösungsansätze, XML-Schema, können nur diejenigen Konzepte bewertet werden, welche sich auf die Angabe des Wertebereichs beziehen. Die anderen Konzepte spielen in dieser Sprache keine Rolle, weil XML-Schema-Datentypen keine Methoden enthalten und daher keine programmiersprachlichen Typen sind. Alle Konzepte, die sich auf das Laufzeitverhalten und das Definieren von Methoden oder Konstruktoren beziehen, sind also im Zusammenhang mit XML-Schema nicht bewertbar.

5.1.1 Aufzählungstypen

Dieses Konzept dient zum Definieren von Typen mit einem Definitionsbereich, der nur wenige Werte umfaßt. Grundprinzip bei der Definition solcher Typen ist es, die Werte der Wertemenge einzeln anzugeben. Darüberhinaus sollte kein hoher Arbeitsaufwand bei der Deklaration notwendig sein: Grundlegende Funktionalitäten, die für alle Aufzählungstypen wichtig sind, wie etwa der Zugriff auf die Elemente des Wertebereichs, sollten nicht selbst geschrieben werden müssen.

In Pizza ist kein Aufzählungstypkonzept vorgesehen, das heißt, es gibt kein separates Schlüsselwort zum Definieren von Aufzählungstypen. Allerdings könnten Aufzählungstypen ähnlich wie in funktionalen Programmiersprachen durch die Angabe von alternativen Konstruktoren nachgebildet werden, so daß jeder Konstruktor ein Aufzählungselement darstellt. Die derart definierten Aufzählungstypen würden aber aufgrund des Fehlens eines eigenständigen Aufzählungstypkonzepts keine Aufzählungstypfunktionalitäten bereitstellen. Diese müßten bei Bedarf selbst implementiert werden.
XML-Schema bietet eine Einschränkung *Enumeration*, mittels derer der Wertebereich eines Typs durch die Aufzählung der erlaubten Werte angegeben werden kann. Allerdings werden Aufzählungstypen wie andere Typen auch als *simpleType* definiert, das heißt, Aufzählungstypen bilden in XML-Schema keine eigene Klasse. Das Fehlen einer solchen eigenen Klasse ist in XML-Schema allerdings nicht negativ zu werten, da für XML-Schema-Datentypen keine Funktionalität festgelegt werden kann. In XML-Schema würde also ein eigener *enumerationType* keinen Vorteil bieten, da ein er keine speziell für Aufzählungstypen definierten Funktionen enthalten kann.
Das JWAM-Rahmenwerk unterstützt aufzählbare Fachwerttypen, indem es eine entsprechende Oberklasse bereitstellt (*dvEnumerable*). Diese besitzt auch geeignete Funktionalitäten, die speziell für Aufzählungstypen erwünscht sind (vgl. Abschnitt 4.3.2). Allerdings erfordert die Erstellung von Aufzählungstypen im JWAM-Rahmenwerk relativ hohen Arbeitsaufwand. Beispielsweise müssen Zugriffsfunktionen auf die einzelnen Aufzählungselemente vom Entwickler selbst geschrieben werden.

5.1.2 Einschränkung des Wertebereichs

Bei vielen Typen ist der Definitionsbereich zu komplex, um ihn mittels einer Aufzählung der einzelnen Werte angeben zu können. Zum Angeben der Definitionsbereiche solcher Typen sollte es möglich sein, einen Basistyp festzulegen und darauf geeignete Einschränkungen vorzunehmen.

In *Pizza* gibt es keine entsprechende Funktionalität. Jeder Wert ist gültig, solange ein Konstruktor mit Attributwerten des korrekten Typs aufgerufen wird. Eine Einschränkung der gültigen Attributwerte ist nicht möglich. Dies wird am Beispiel des Werttyps *Uhrzeit* (vgl. Abschnitt 4.1) deutlich:

```
Euro_UhrzeitKonstruktor (int hour, int min);
```

Der Konstruktor *Euro_Uhrzeit* kann mit sämtlichen *int*-Werten aufgerufen werden, so daß auch Werte wie *13:99* erzeugt werden können, obwohl solche Werte keine Uhrzeit darstellen. Es ist in Pizza nicht vorgesehen, diese Konstruktoren mit Funktionalität zu versehen, so daß beispielsweise ungültige Werte für Uhrzeiten ausgeschlossen werden könnten.
In der Sprache XML-Schema sind Datentypen gänzlich durch ihren Wertebereich definiert. Die Angabe eines Wertebereichs erfolgt dabei hauptsächlich durch die Einschränkung des Wertebereichs eines Basistyps. Zu diesem Zweck stellt XML-Schema vielfältige Möglichkeiten bereit, um Wertebereiche einzuschränken.
Fachwerttypen im JWAM ermöglichen die Einschränkung des Wertebereichs durch die Implementierung der Methode *isValid*, welche vor der Erstellung eines neuen Fachwerts aufgerufen wird. Diese Überprüfung wird zur Laufzeit vorgenommen und verhindert beispielsweise, daß eine Uhrzeit wie *13:99* erzeugt wird.

5.1.3 Undefinierter Wert

Werttypen sollten einen undefinierten Wert aufweisen. Dieser sollte entweder wie bei funktionalen Sprachen in die Sprache integriert sein, oder es sollte eine Möglichkeit geben, mittels geeigneter Methoden einen undefinierten Wert beziehungsweise Ausnahmewerte zu definieren.
Beim Fehlen von undefinierten Werten müßte eine Fehlersituation zu einer der folgenden Reaktionen führen: zum einen könnte ein festgelegter Wert zurückgeliefert werden, wobei das Problem bestehen kann, daß dieser in den gültigen Wertebereich fällt - dann wäre nicht mehr zu erkennen, daß eine Fehlersituation vorliegt. Zum anderen könnte eine *Exception* ausgelöst werden, welche zu einem Programmabbruch führen kann, wenn sie nicht abgefangen wird.

In der Programmiersprache Pizza gehört der undefinierte Wert nicht automatisch zur Wertemenge eines neu definierten Werttyps. Die Sprache stellt auch keine Mittel bereit, um explizit einen undefinierten Wert zurückzuliefern. Ausnahmefälle können in Pizza daher nur in einer der beiden oben genannten Weisen behandelt werden.
Bei Datentypen, die mit XML-Schema definiert werden, ist ebenfalls nicht automatisch ein undefinierter Wert enthalten. Der Wertebereich eines Datentyps könnte allerdings bei Bedarf durch eine Vereinigung (*union*) aus dem gewünschten Wertebereich und einem zusätzlichen undefinierten Wert angegeben werden. Dies ist allerdings nur dann sinnvoll möglich, wenn dem undefinierte Wert eine Repräsentation gegeben werden kann, die nicht einem Wert des eigentlichen Wertebereichs entspricht.
Das JWAM-Rahmenwerk sieht für jeden Fachwerttyp den undefinierten Wert als Teil der Wertemenge vor. Zu diesem Zweck existiert in den Klassendefinitionen von

Fachwerttypen eine Methode, die den undefinierten Wert explizit zurückliefert. Darüber hinaus ist dieser undefinierte Wert derart in das Rahmenwerk eingebunden, daß er automatisch als Wert von Eingabefeldern zurückgeliefert werden kann. Dies ist beispielsweise dann sinnvoll, wenn in einem Eingabefeld etwas eingegeben wurde, was nicht in den Wertebereich des mit diesem Eingabefeld verbundenen Werttyps fällt.

5.1.4 Einfaches Definieren von Konstruktoren und Verwendung von Pattern Matching

In funktionalen Programmiersprachen kann ein neuer aggregierter Typ auf einfache Weise definiert werden, indem ein Konstruktorname angegeben wird sowie die Typen, aus denen der neue Typ zusammengesetzt sein soll. Der Zugriff auf die Komponenten eines solchen aggregierten Typs kann mit Hilfe von Pattern Matching erfolgen (vgl. Abschnitt 4.1.1.2).
In objektorientierten Sprachen hingegen müssen für die einzelnen Komponenten eines aggregierten Typs sowohl Attribute definiert und benannt werden als auch Methoden zum Zugriff auf diese Attribute. In den Konstruktoren müssen diese Attribute entsprechend der übergebenen Konstruktor-Parameter gesetzt werden.
Es wäre zum einen wünschenswert, wenn zum Definieren von Werttypen auch in objektorientierten Sprachen nur die Typen angegeben werden müßten, aus denen er zusammengesetzt ist. Zum anderen sollte es nicht notwendig sein, Zugriffsfunktionen selbst definieren zu müssen. Zu beachten ist hierbei, daß es keine Lösung sein kann, alle Attribute einer Klasse als *public* zu definieren, denn öffentliche Attribute können nicht nur gelesen, sondern auch von außen geändert werden, was im Widerspruch zur Unveränderlichkeit von Werttypen stehen würde.

In der Java-Erweiterung Pizza können Konstruktoren wie in funktionalen Sprachen auf einfache Weise definiert werden. Innerhalb einer Klassendefinition müssen dazu lediglich der Name des Konstruktors und dessen Argumente angegeben werden. Daraus werden dann bei der Übersetzung nach Java automatisch die Attribute und die aufwendigeren Java-Konstruktoren erzeugt. Auch die Methoden für den Zugriff auf die Attribute müssen in Pizza nicht manuell geschrieben werden, denn Pizza stellt einen Pattern Matching Mechanismus bereit.
Das JWAM-Rahmenwerk bietet im Gegensatz zu Pizza keine Möglichkeiten, um Typen durch einfache Konstruktoren zu definieren. Die Fachwerttypen im JWAM-Rahmenwerk müssen statt dessen wie normale Klassen implementiert werden, also mit separaten Attributen und Konstruktoren, die den Attributen Werte zuweisen. Auch Pattern Matching ist nicht möglich. Statt dessen müssen manuell Zugriffsfunktionen zum Zugriff auf die Attribute definiert werden.

5.1.5 Angabe einer Stringrepräsentation

Es muß für jeden Wert eines Werttyps möglich sein, diesen auszugeben. Bei algebraischen Datentypen in funktionalen Programmiersprachen erfolgt dies mit der gleichen Zeichenkette, die auch zum Erzeugen des entsprechenden Werts benutzt wurde. Die Repräsentation eines Werts setzt sich also wie dessen Deklaration aus

dem Kontruktor und den Werten für die einzelnen Komponenten zusammen, beispielsweise *Schulklasse 10 'b' "Meyer"*.
Bei objektorientierten Sprachen hingegen muß für einen Werttyp eine Stringrepräsentation angegeben werden können, welche jedem Wert dieses Typs eine geeignete Zeichenkette zuordnet. Ähnlich wie bei funktionalen Sprachen sollte die Stringrepräsentation eines Werts derjenigen entsprechen, aus der er auch erzeugt werden kann.

Im JWAM-Rahmenwerk ist in jedem Fachwerttyp eine Methode *toStringImpl* vorgesehen, welche die Stringrepräsentation festlegt. Diese Methode muß für jeden Fachwerttyp speziell angepaßt werden. Eine weitere Methode, die im Zusammenhang mit der Stringrepräsentation steht, ist die Methode *valueImpl*. Sie dient als Konstruktor zum Erzeugen eines neuen Fachwerts aus einer Stringrepräsentation. Obwohl diese beiden Methoden in Zusammenhang stehen, müssen sie separat implementiert werden, wobei allerdings deren Kopplung vom Entwickler beachtet werden muß.
Pizza stellt keine besondere Funktionalität zum Liefern einer Stringrepräsentation oder zum Erzeugen eines Werts aus einer Stringrepräsentation bereit. Zwar läßt sich wie in jeder objektorientierten Sprache solche Funktionen manuell implementieren, aber es ist nicht sichergestellt, daß sie auf jeden Fall implementiert werden.

5.1.6 Kennzeichnung von Werttypen

Exemplare von Werttypen sollten als solche erkannt und behandelt werden können (vgl. Abschnitt 2.1.3). Dazu müßten Werttypen als solche gekennzeichnet sein. Eine derartige Kennzeichnung könnte beispielsweise dadurch erfolgen, daß bei der Deklaration von Werttypen ein Schlüsselwort, wie etwa *value*, verwendet wird. Da in den objektorientierten Programmiersprachen kein derartiges Schlüsselwort existiert (vgl. Abschnitt 3.1.2), wäre es wünschenswert, wenn die Kennzeichnung von Werttypen auf andere Weise erfolgen könnte.

Die Programmiersprache *Pizza* bietet zwar die Möglichkeit, Werttypen zu erstellen. Jedoch sind diese Typen nicht von anderen Typen zu unterscheiden, da sowohl Objekttypen als auch Werttypen mit dem *class*-Konstrukt definiert werden.
Die Sprache XML-Schema definiert ebenfalls Datentypen, jedoch sind diese nicht direkt in einer Programmiersprache einsetzbar. Ob daraus in einer Programmiersprache neue Typen generiert werden und ob sie dort als Werttypen gekennzeichnet werden können, hängt von der Umsetzung ab. Beispielsweise werden beim in Abschnitt 4.2.3 beschriebenen Vorgehen keine Typen generiert. Die XML-Schema-Datentypen werden lediglich zur Laufzeitüberprüfung genutzt. Bei einer Generierung von Klassen aus XML-Schema-Datentypen würden hingegen neue Typen erstellt werden. Ob diese Typen dann als Werttypen gekennzeichnet sind, hängt von deren Implementierung ab. Das Konzept „Kennzeichnung von Werttypen“ läßt sich also in bezug auf XML-Schema ohne Zusammenhang mit einem Umsetzungsmechanismus nicht bewerten.
Im JWAM-Rahmenwerk können Werttypen deklariert werden. Diese erben alle von der Klasse *DomainValue* und sind daher als Werttypen erkennbar.

5.1.7 Sicherung der Unveränderlichkeit

Wenn Werttypen anhand von Referenztypen implementiert werden, so müssen die Objekte dieser Referenztypen unveränderlich sein, um die Wertsemantik zu wahren (vgl. Abschnitt 3.4). Es sollte nicht möglich sein, Werttypen zu definieren, die diese Eigenschaft der Unveränderlichkeit nicht aufweisen.
Um dieses Ziel zu erreichen, sind zwei Ansätze denkbar: entweder wird wie in funktionalen Sprachen das Konzept von Zustandsänderungen generell ausgeschlossen, oder der Compiler müßte Zustandsänderungen bei Werttypen erkennen und als Fehler melden können. Wie am Beispiel der objektorientierten Sprache Eiffel erläutert wurde, ist die Erkennung von Zustandsänderungen durch den Compiler problematisch (vgl. Abschnitt 3.3.3) und wird daher durch objektorientierte Sprachen nicht unterstützt. Wünschenswert wäre jedoch, daß auch objektorientierte Sprachen eine der beiden genannten Möglichkeit bieten, die Unveränderlichkeit von Werttypen zu garantieren.

In *Pizza* ist innerhalb von Funktionen der direkte Zugriff und somit die Veränderung von Attributen nicht vorgesehen. Statt dessen erfolgt der Zugriff mittels *Pattern Matching*, wodurch die Werte der Attribute von algebraischen Datentypen nur gelesen, aber nicht verändert werden können. Allerdings ist anzumerken, daß die Änderung von Attributen in Pizza nicht ausgeschlossen ist. Um die Unveränderlichkeit zu garantieren, müßten die erzeugten Attribute bei der automatischen Generierung von Java-Klassen als *final* deklariert sein (vgl. Abschnitt 3.4.2.1), was allerdings nicht der Fall ist.
Im JWAM-Rahmenwerk wird die Implementierung unveränderlicher Java-Klassen durch das Fachwert-Rahmenwerk unterstützt: Werttypen, die nach den Richtlinien dieses Fachwert-Rahmenwerks erstellt werden, weisen keine verändernden Operationen auf, so daß die Exemplare dieser Werttypen unveränderlich sind. Allerdings beruht die Einhaltung dieser Richtlinien nur auf Programmierkonventionen und kann daher nicht garantiert werden.

5.1.8 Einfache Entwicklung von Werttypen

Die Entwicklung von Werttypen sollte grundsätzlich einfach sein. Als einfach wird die Entwicklung betrachtet, wenn sich der zeitliche und programmiertechnische Aufwand bei der Entwicklung in vertretbaren Grenzen hält. Als kontraproduktiv wird daher die Einführung einer neuen Notation erachtet, denn das Erlernen einer neuen Notation würde wiederum einigen Aufwand erfordern. Ein weiteres Kriterium für die Vereinfachung ist es, daß auch für unerfahrene Entwickler die Erstellung von Werttypen erleichtert wird.

Die Java-Erweiterung *Pizza* bietet zwar die Möglichkeit, Werttypen zu definieren, jedoch muß der Entwickler dafür neben einer neuen Notation auch noch ein neues Programmierparadigma anwenden, nämlich das der funktionalen Programmierung.
Bei Verwendung von XML-Schema-Datentypen wird der Entwicklungsaufwand von Werttypen dadurch reduziert, daß Wertebereichseinschränkungen auf einfache Weise darstellbar sind. Die Erzeugung von XML-Schema-Datentypen erfordert zwar

Kenntnis über diese neue Notation, jedoch ist es denkbar, die Generierung dieser Datentypen durch ein Werkzeug zu unterstützen.
Für die Arbeit mit dem JWAM-Rahmenwerk muß zum einen die Sprache Java bekannt sein. Außerdem muß ein Entwickler, der Werttypen mit Hilfe des JWAM erstellen will, mit diesem Rahmenwerk vertraut sein. Zur Vereinfachung der Erstellung von Werttypen mit dem JWAM-Rahmenwerk werden diese in der Regel nicht gänzlich neu entwickelt (vgl. Abschnitt 4.3.3): Sowohl der erfahrene als auch der unerfahrene Entwickler greifen bei der Erstellung eines neuen Fachwerttyps auf den Code eines bereits existierenden zurück und passen diesen an. Der Grund hierfür ist, daß sich der Quellcode von Fachwerttypen in großen Teilen ähnelt. Da für Projekte in der Regel eine hohe Anzahl von Fachwerttypen erstellt werden muß, führt diese Redundanz zu ständigen Wiederholungen derselben Arbeitsschritte. Zudem sind genaue Kenntnisse über das Fachwert-Rahmenwerk notwendig, um die richtigen Arbeitsschritte zur Fachwerttypentwicklung durchführen zu können.

5.1.9 Zusammenfassung

Zusammengefaßt ergibt sich die folgende Bewertung der einzelnen Ansätze in bezug auf die oben betrachteten Konzepte, die für die Erstellung von Werttypen in objektorientierten Sprachen wünschenswert wären (vgl. Tabelle 5-1).
In der folgenden Tabelle sind in der Spalte XML-Schema einige Felder als nicht bewertbar gekennzeichnet, weil in XML-Schema nur diejenigen Konzepte bewertet werden können, welche sich auf die Angabe des Wertebereichs beziehen.

	Pizza	**XML-Schema**	**JWAM-Rahmenwerk**
Aufzählungstypen	+	+	+
Einschränkungen des Wertebereichs	-	++	+
Undefinierter Wert	O	O	++
Einfache Konstruktoren / Pattern Matching	++	nb	-
Angabe der Stringrepräsentation	O	nb	+
Erstellung neuer Typen / Kennzeichnung von Werttypen	+	nb	++
Sicherung der Unveränderlichkeit	+	nb	+
Einfache Entwicklung von Werttypen	O	+	O

++	sehr gute Unterstützung
+	gute Unterstützung
O	neutral
-	keine Unterstützung
nb	nicht bewertbar

Tabelle 5-1: Unterstützte Konzepte bei verschiedenen Lösungsansätzen

Wie sich anhand des oben angeführten Vergleichs erkennen läßt, unterstützt das JWAM-Rahmenwerk bereits die meisten der Konzepte, die für die Erstellung von Werttypen in objektorientierten Sprachen als wünschenswert betrachtet werden. Es liegt daher nahe, dieses Rahmenwerk als Basis für ein System zur Unterstützung der Werttypentwicklung zu wählen.
Das zu entwickelnde System sollte die Stärken des JWAM-Rahmenwerks wahren und, wenn möglich, weitere Unterstützung bieten. Als unterstützenswert werden dabei prinzipiell alle Konzepte betrachtet, die in Tabelle Tabelle 5-1 nicht als „sehr gut geeignet“ beurteilt sind. Insbesondere die Konzepte *Einfache Konstruktoren / Pattern Matching* und *Einfache Entwicklung von Werttypen* scheinen sich für eine Unterstützung anzubieten.
Im folgenden wird erläutert und begründet, welche Konzepte für die Entwicklung eines Systems zur Unterstützung der Fachwerttypentwicklung ausgewählt werden.

Als Schwerpunkte werden die folgenden Bereiche betrachtet:

- **Einfache Entwicklung von Werttypen**
  Die Entwicklung von JWAM-Fachwerttypen ist nicht trivial. Wie in Abschnitt 4.3.3 gezeigt, stellt die Entwicklung mit Hilfe von Rahmenwerken insbesondere unerfahrene Entwickler vor Schwierigkeiten, während für erfahrene Entwickler der verhältnismäßig hohe Arbeitsaufwand bei der Entwicklung von Werttypen ins Gewicht fällt. Wie sich in der Tabelle 5-1 erkennen läßt, ist diese Problematik einer derjenigen Bereiche, in denen Unterstützung besonders wichtig erscheint. Daher ist die Vereinfachung der Entwicklung einer der Hauptschwerpunkte beim Entwurf des zu implementierenden Systems.

- **Wertebereichseinschränkungen**
  Ein Werttyp setzt sich zusammen aus der Deklaration seines Wertebereichs und der Methoden, die sein Verhalten bestimmen. In vielen Fällen sind keine speziellen Methoden notwendig, so daß der Hauptaspekt bei der Deklaration von Werttypen auf der Angabe des Wertebereichs liegt. Mit der Angabe des Wertebereichs könnten also solche Werttypen, die keine fachlichen Methoden benötigen, fast vollständig spezifiziert werden (siehe auch folgenden Punkt *Stringrepräsentation*). Daher erscheint die Unterstützung der Angabe des Wertebereichs als ein wichtiger Schwerpunkt bei der Entwicklung eines Systems zur Fachwerttyperstellung. Wie aus Tabelle 5-1 ersichtlich, kann die Sprache XML-Schema in dieser Hinsicht möglicherweise einige Anregungen bieten.

- **Stringrepräsentation**
  Die Zeichenkette, aus der ein Wert erzeugt werden kann, muß mit derjenigen übereinstimmen, die zur Repräsentation dieses Werts dient. Im JWAM-Rahmenwerk werden für diese beiden Aufgaben die Methoden *toStringImpl* und *valueImpl* eingesetzt (vgl. Abschnitt 5.1.5). Das zugrundeliegenden Aufbauprinzip der Stringrepräsentation muß also zweimal in unterschiedlicher Weise in unterschiedlichen Methoden in eine Implementierung umgesetzt werden. Es wäre jedoch wünschenswert, wenn dem Zusammenhang dieser beiden Methoden da-

durch Rechnung getragen werden könnte, daß das zugrundeliegende Aufbauprinzip der entsprechenden Stringrepräsentation nur einmal angegeben werden muß.
Die Angabe eines solchen Aufbauprinzips wird auch deswegen als wichtig erachtet, weil diese neben der Wertebereichsangabe den einzigen weiteren Aspekt darstellt, der zur Spezifizierung von Werttypen ohne fachliche Methoden notwendig ist.

Zusätzlich zu den oben genannten Bereichen ist eine Unterstützung bei folgenden Konzepten wünschenswert:

- **Unveränderlichkeit**
  Die Unveränderlichkeit von JWAM-Fachwerttypen ist nicht garantiert. JWAM-Fachwerte sind nur dann unveränderlich, wenn sie korrekt implementiert werden. Die Korrektheit von JWAM-Fachwerttypen wird zwar per Programmierkonvention gefordert, kann aber vom Rahmenwerk nicht garantiert werden.
  Da die Eigenschaft der Unveränderlichkeit als grundlegend für Werttypen betrachtet wird, wäre es wünschenswert, wenn die Unveränderlichkeit von JWAM-Fachwerttypen in jedem Fall garantiert werden könnte.

- **Aufzählungstypen**
  Wie bereits weiter oben erwähnt, sollen Wertebereichsangaben einen Schwerpunkt bei der Entwicklung des Systems darstellen. Da Aufzählungstypen eine spezielle Art der Wertebereichsangabe darstellen, spielen sie im weiteren Verlauf der Arbeit eine untergeordnete Rolle.

Neben den bisher betrachteten Bereichen wäre eine Unterstützung theoretisch auch im folgenden Bereich interessant:

- **Einfache Konstruktoren und Pattern Matching**
  In bezug auf die einfache Implementierung von Konstruktoren und Pattern Matching stellt das JWAM-Rahmenwerk keine Funktionalität bereit (vgl. Abschnitt 5.1.4).
  Obwohl diese Konzepte nicht geeignet unterstützt werden, können einige deren Aspekte durch andere Konzepte ersetzt werden, die ähnliches leisten. Statt Pattern Matching für den Zugriff auf die Attribute eines Typs zu verwenden, können für alle Attribute eines Typs Zugriffsfunktionen definiert werden. Statt einem Entwickler die Definition einfacher Konstruktoren zu ermöglichen, könnten übliche Konstruktoren aus gegebenen Attributen automatisch generiert werden.
  Das Konzept der einfachen Konstruktoren und des Pattern Matching wird daher nicht im Mittelpunkt stehen, wohl aber die Forderung, daß Zugriffsfunktionen für Attribute und Konstruktoren automatisch bereitstehen.

Die genannten Bereiche bieten Ansatzpunkte für die Entwicklung eines Systems, welches die Erstellung von Werttypen mit dem JWAM-Rahmenwerk unterstützen kann. Im weiteren Verlauf dieser Arbeit soll ein solches System entworfen und implementiert werden.

Dazu wird zunächst ein Ansatz betrachtet, der sich auf die Kombination von XML-Schema und JWAM-Rahmenwerk bezieht, denn wie sich aus Tabelle 5-1 ergibt, scheinen sich diese beiden Ansätze gut zu ergänzen. So könnte XML-Schema im Bereich Wertebereichseinschränkungen und Vereinfachung der Entwicklung von Werttypen möglicherweise gute Ansatzpunkte zur Unterstützung der Entwicklung von JWAM-Fachwerttypen bieten.

5.2 Ansatz der dynamischen XML-Verifikation

Der Lösungsansatz, der in diesem Abschnitt betrachtet wird, wurde im Rahmen einer Studienarbeit am Fachbereich Informatik der Universität Hamburg entwickelt (vgl. [SAU01]). Dabei wurde das JWAM-Rahmenwerk um das sogenannte XMLDomainValue-Rahmenwerk erweitert, welches im folgenden erläutert wird.

Das XMLDomainValue-Rahmenwerk vereinfacht die Definition von Fachwerttypen, indem es die Möglichkeiten der Datentypspezifikation von XML-Schema ausnutzt (vgl. Abschnitt 4.2.2). Das XMLDomainValue-Rahmenwerk ist über Vererbung an das bestehende Fachwert-Rahmenwerk angebunden. Dadurch können die bestehenden Fachwertklassen weiterhin genutzt werden und Fachwerttypen auch weiterhin so wie bisher erstellt werden. Zu einem Fachwerttyp im *XMLDomainValue*-Rahmenwerk gehören stets zwei Komponenten:

1. **Ein XML-Schema-Datentyp**
   Der XML-Schema-Datentyp spezifiziert die Wertemenge des neu zu erstellenden Fachwerttyps. Der Wertebereich kann dabei durch logische Einschränkungen wie *MinInclusive* und *MinExclusive* oder durch Angabe der gültigen Repräsentationen mittels regulärer Ausdrücke festgelegt werden.

2. **Ein XML-Fachwerttyp**
   Ein XML-Fachwerttyp ist eine konkrete Subklasse der abstrakten Klasse *XMLDomainValue*. Diese Typen sollen bei der Anwendungsentwicklung die konventionellen Fachwerttypen ersetzen. Die Exemplare der XML-Fachwerttypen werden im folgenden als XML-Fachwerte bezeichnet.

Möchte ein Entwickler einen neuen Fachwerttyp definieren, muß er in jedem Fall einen neuen XML-Schema-Datentyp entwickeln. Der XML-Fachwerttyp muß hingegen nicht immer neu entwickelt werden, da für Fachwerttypen in vielen Fällen die vordefinierten generischen Klassen *dvXMLGeneric* oder *dvXMLGenericTemplate* verwendet werden können. Dies ist dann möglich, wenn ein zu entwickelnder Fachwerttyp lediglich eine Wertebereichsprüfung beinhalten soll und keine spezielle Funktionalität, wie etwa fachliche Methoden. In solchen Fällen wird die Entwicklung von neuen Fachwerttypen stark vereinfacht.
Die Wertebereichsvalidierung im XMLDomainValue-Rahmenwerk funktioniert folgendermaßen: Wird während der Ausführung eines Programms ein XML-Fachwert instantiiert, so wird dieser zunächst in ein XML-Dokument umgesetzt. Die Überprüfung der Gültigkeit dieses XML-Dokuments und damit des XML-Fachwerts erfolgt mit Hilfe eines Prüfmechanismus, welcher durch eigene Parsing-Klassen oder

durch einen validierenden XML-Parser implementiert werden kann. Überprüft wird, ob das XML-Dokument in bezug auf den zugehörigen XML-Schema-Datentyp *gültig* ist (vgl. Abschnitt 4.2.2), das heißt es wird geprüft, ob die übergebenen Werte in den im XML-Schema-Datentyp definierten Wertebereich fallen.

Es ist anzumerken, daß die im XMLDomainValue-Rahmenwerk notwendigen XML-Schema-Datentypen manuell erstellt werden müssen, so daß einem Entwickler die XML-Schema-Notation bekannt sein muß. Es wäre aber durchaus denkbar, zur Vereinfachung der Erstellung von XML-Schema-Datentypen ein Werkzeug einzusetzen. So existieren bereits mehrere professionelle Entwicklungsumgebungen, die einen XML-Schema-Editor beinhalten, wie beispielsweise *XML Spy* (vgl. [SPY01] und *Turbo XML™* (vgl. [TUR01]). Diese XML-Schema-Editoren sind allerdings nicht speziell auf die Erstellung von XML-Schema-Datentypen für das XMLDomainValue-Rahmenwerk zugeschnitten.
Bezüglich der Leistung des XMLDomainValue-Rahmenwerks wird in [SAU01] angemerkt, daß es nicht für Anwendungen geeignet ist, bei denen zur Laufzeit viele Fachwerte in kurzer Zeit erzeugt werden müssen, da der Aufwand der dynamischen Validierung hoch ist.

Zusammengefaßt läßt sich sagen, daß das XMLDomainValue-Rahmenwerk auf die Möglichkeit zur vereinfachten Angabe von Wertebereichen mittels XML-Schema-Datentypen konzentriert ist. Das Benutzen von XML-Schema-Datentypen für die Spezifikation hat den Vorteil, daß XML-Schema einen offiziellen Standard darstellt, wodurch die Austauschbarkeit mit anderen Programmen ermöglicht wird.
Allerdings lassen sich durch XML-Schema-Datentypen nur Wertebereiche spezifieren. Wie in Abschnitt 5.1.9 dargestellt wurde, sind bei der Werttypentwicklung neben der Wertebereichsangabe jedoch noch weitere Konzepte von Interesse. Dazu gehört unter anderem die Angabe einer Stringrepräsentation und die Benennung einzelner Komponenten eines Typs, um Zugriff auf diese zu ermöglichen. Zum Angeben solcher Informationen innerhalb einer Typdeklaration reichen XML-Schema-Datentypen nicht aus.
Im folgenden wird ein anderes System zur Unterstützung der Fachwerttypentwicklung im JWAM-Rahmenwerk entworfen, welches auch diese weiteren Konzepte berücksichtigen soll.

5.3 Entwurf eines eigenen Lösungsansatzes

Bei der Entwicklung eines Systems zur Erstellung von JWAM-Fachwerttypen sind die in Abschnitt 5.1.9 gewonnenen Erkenntnisse zu berücksichtigen. Dort wurde herausgearbeitet, daß das JWAM-Rahmenwerk an folgenden Stellen unterstützt werden könnte:

1. Vereinfachung der Entwicklung
2. Wertebereichseinschränkungen
3. Kopplung von Konstruktor und Stringrepräsentation
4. Sicherung der Unveränderlichkeit

Die Anforderung *Vereinfachung der Entwicklung* läßt sich folgendermaßen genauer spezifizieren (vgl. Abschnitt 5.1.8):

- Vermeidung einer neuen Notation
- weniger Aufwand bei der Programmierung, Zeitersparnis
- Unterstützung unerfahrener Entwickler

Neben den bisher genannten Ansatzpunkten ist noch ein weiterer von Interesse. Dieser ergibt sich aus der Tatsache, daß fachliche Werte in allen in Kapitel 3 untersuchten objektorientierten Programmiersprachen nicht unterstützt werden. Es wäre daher wünschenswert, wenn durch das zu entwickelnde System Fachwerttypen für verschiedene objektorientierte Sprachen und Rahmenwerke erzeugt werden könnten.

In den folgenden Abschnitten wird ein System zur Fachwerterstellung im JWAM-Rahmenwerk entworfen, welches die oben aufgestellten Anforderungen berücksichtigt.

5.3.1 Einsatz von Fachwerttypbeschreibungen

Eine der wesentlichen Anforderungen an ein System zur Fachwerttypentwicklung ist die Vereinfachung der Entwicklung. Ein wichtiger Aspekt, um dieses Ziel zu erreichen, stellt die Abstraktion vom Rahmenwerk dar. Ein Entwickler soll sich bei der Erstellung von Fachwerttypen nicht um diejenigen Aspekte kümmern müssen, die bei allen Fachwerttypen gleich sind. Zur Erstellung von Fachwerttypen wäre es sinnvoll, nur diejenigen Anteile anzugeben zu müssen, die für einen Fachwerttyp relevant sind. Diese relevanten Informationen sollen in einer geeigneten Notation dargestellt werden, welche im folgenden als *Fachwerttypbeschreibung* bezeichnet wird.

Als Notation eignet sich die Sprache XML, da sie für die Beschreibung strukturierter Daten konzipiert ist (vgl. Abschnitt 4.2.1). Die Spezifikation XML-Schema ist ein Beispiel dafür, wie Wertebereiche von Datentypen mittels XML spezifiziert werden können (vgl. Abschnitt 4.2.2). Da auch bei den Fachwerttypen die Angabe des Wertebereichs zu den relevanten Anteilen gehört, wäre es denkbar, XML-Schema als Basis der Fachwerttypbeschreibung zu benutzen. Wie in den folgenden Abschnitten erläutert werden wird, gehören zu Fachwerttypbeschreibungen jedoch nicht nur Wertebereichseinschränkungen, sondern noch weitere Elemente. Diese Elemente, wie beispielsweise die Stringrepräsentation (vgl. Abschnitt 6.2.3), lassen sich nicht mit Hilfe von XML-Schema beschreiben. Daher bietet es sich an, proprietäre XML-Elemente zur Beschreibung der Fachwerttypen zu verwenden.
Der genaue Aufbau dieser XML-Fachwerttypbeschreibungen wird in Kapitel 6 erarbeitet.

5.3.2 Generierung von JWAM-Fachwertklassen

XML-Fachwerttypbeschreibungen können zwar genutzt werden, um Fachwerttypen zu spezifizieren. Allerdings werden diese Beschreibungen noch keine programmier-

sprachlichen Typen eingeführt. Zu diesem Zweck soll das zu entwickelnde System aus einer Fachwerttypbeschreibung einen JWAM-Fachwerttyp generieren können.
Der Einsatz von Fachwerttypbeschreibungen und automatischer Codegenerierung kann außerdem bei unerfahrenen Entwicklern zur Vermeidung von Fehlern beitragen. Flüchtigkeitsfehler sowie durch Unkenntnis ausgelöste Fehler bei der Programmierung werden verhindert. Gleichzeitig wird auch für erfahrene Entwickler der Schreibaufwand auf ein Minimum beschränkt.
Bei der Generierung von Fachwerttyp-Klassen bietet sich ein Mechanismus an, der mit speziellen Dateien arbeitet, welche im folgenden als *Codeschablonen* bezeichnet werden. Solche Schablonen enthalten zum einen Quellcode, der bei der Generierung direkt übernommen wird und zum anderen Markierungen, welche das Einsetzen von variablen Code-Anteilen ermöglichen. Dieser Mechanismus kann für die Generierung von Fachwerttypen gut eingesetzt werden, weil auch diese viele statische Anteile erhalten. Die variablen Anteile können mit Hilfe der Informationen aus der Fachwerttypbeschreibung generiert werden.
Zusammengefaßt könnte ein System, das JWAM-Fachwertklassen mit Hilfe von Fachwerttypbeschreibungen und Codeschablonen generiert, folgendermaßen aussehen:

Fehler! Keine gültige Verknüpfung.

Abbildung 5-1: Generierung von JWAM-Fachwertklassen

Generell ist zu beachten, daß nur solche Codeanteile erzeugt werden können, welche begrenzt variabel sind. Nicht alle denkbaren Fachwerttypen sind vollständig generierbar, da die fachlichen Methoden von Fachwerttypen prinzipiell beliebig sein können. Wäre die Generierung aller denkbaren fachlichen Methoden angestrebt, so müßte das System komplexe Möglichkeiten zum Angeben solcher fachlicher Methoden bereitstellen - und dies würde die Handhabbarkeit des zu entwickelnden Systems in Frage stellen.
In den Fällen, wo kein vollständiger JWAM-Fachwerttyp generiert werden kann, muß der Entwickler die fehlende Funktionalität manuell ergänzen. Zudem kann es notwendig sein, jene Teile des generierten Codes anzupassen, die nur vereinfacht generiert werden konnten. In diesem Zusammenhang ist es wichtig, daß der generierte Code gut lesbar ist, um dessen manuelle Weiterbearbeitung zu erleichtern.

5.3.3 Einsatz eines Werkzeugs

Durch die Benutzung von XML-Fachwerttypbeschreibungen und einem Mechanismus, der daraus JWAM-Fachwerttypen generiert, ist es für einen Entwickler möglich, zur Entwicklung eines Fachwerttyps lediglich dessen relevanten Anteile angeben zu müssen. Bei einer solchen Lösung bleiben allerdings noch zwei Probleme: zum einen wird der Entwickler mit einer neuen Notation zur Spezifizierung von Fachwerttypen konfrontiert, zum anderen fehlt noch die Möglichkeit, den Codegenerierungsmechanismus auf einfache Weise zu aktivieren. Damit ein Entwickler nicht direkt mit den XML-Fachwerttypbeschreibungen umgehen muß, soll daher ein Werkzeug eingesetzt werden, womit diese Fachwerttypbeschreibungen erzeugt werden können. Dieses Werkzeug soll eine grafische Benutzungsschnittstelle zum

Erstellen der XML-Fachwerttypbeschreibungen bereitstellen. Es soll außerdem dazu benutzt werden können, die Codegenerierung anzustoßen (vgl. Abbildung 5-2).
Fehler! Keine gültige Verknüpfung.

Abbildung 5-2: Einsatz eines Werkzeugs zur Generierung von JWAM-Fachwertklassen

Damit die Vereinfachung der Fachwertentwicklung auch tatsächlich erreicht werden kann, muß allerdings beim Entwurf eines solchen Werkzeugs auf eine möglichst einfache Benutzbarkeit geachtet werden, um den Einarbeitungsaufwand zu minimieren. Zur Einfachheit des Werkzeugs trägt auch bei, wenn einfache Fachwerttypen möglichst einfach modellierbar sind. Zu diesem Zweck sollen sogenannte *komplexe* und *einfache Fachwerttypen* unterschieden werden (vgl. Abschnitt 6.1). Dadurch wird der Anforderung der Zeitersparnis Rechnung getragen.
Auch die Anforderung, unerfahrene Entwickler zu unterstützen, kann durch ein Werkzeug erfüllt werden. Dies wird zum einen dadurch erreicht, daß bei bestimmten Angaben nur sinvolle Werte zugelassenen sind. Zum anderen werden die Haupteigenschaften der JWAM-Fachwerttypen verbildlicht, so daß auf einen Blick klar wird, welche Aspekte von Fachwerttypen anzupassen sind.
Da das Werkzeug zur Erstellung von Fachwerttypbeschreibungen und deren Umsetzung in JWAM-Quellcode dient, können diese Fachwerttypbeschreibungen als Basis des Fachwertwerkzeugs angesehen werden. Vor der Beschreibung der Implementation des Fachwertwerkzeugs wird daher im folgenden Kapitel herausgearbeitet, wie diese Fachwerttypbeschreibungen aufgebaut sind.

6 Aufbau von Fachwerttypbeschreibungen

Im vorigen Kapitel wurde eine grundsätzliche Lösungsidee für ein System zur Fachwerttyperstellung entwickelt, die im wesentlichen darauf beruht, Fachwerttypbeschreibungen für die Konstruktion von Fachwerttypen zu benutzen. In diesem Kapitel soll die genaue Struktur dieser Fachwerttypbeschreibungen herausgearbeitet werden. Die Basis der folgenden Betrachtungen bilden zum einen Beispiele für JWAM-Fachwerttypen und zum anderen die in Abschnitt 5.1 aufgestellten Konzepte, insbesondere die Aspekte Wertebereichsangaben, Stringrepräsentation und Aufzählungstypen.

6.1 Arten von Fachwerttypen

Die zu erstellenden Fachwerttypen lassen sich in mehrere Arten unterteilen. Wie in Abschnitt 4.3.2 erläutert, sind einige der JWAM-Fachwerttypen Subtypen von *dvEnumerable*, andere hingegen von *DomainValueImpl*. Anstatt allerdings nur zwei Arten von Fachwerttypen zu unterscheiden, wie sich das anhand der Oberklassen anbieten würde, wird im folgenden von drei Arten von Fachwerttypen ausgegangen. Es bietet sich nämlich an, die von *DomainValueImpl* erbenden Fachwerttypen noch einmal aufzuteilen. Eine solche Aufteilung kann anhand der Kategorisierung in Abschnitt 2.2 motiviert werden, in welcher *aggregierte Typen* und *Subtypen* unterschieden werden: die aggregierten Typen bestehen aus mehreren Komponenten, während die Subtypen keine Komponenten aufweisen, sondern einen Basistyp spezialisieren. Eine ähnliche Aufteilung findet sich auch in XML-Schema, wo komplexe und einfache Typen unterschieden werden: komplexe Typen können im Gegensatz zu den einfachen Typen Subelemente und Element-Attribute haben.
In Anlehnung an diese Aufteilung wird im folgenden zwischen *komplexen* und *einfachen* Fachwerttypen unterschieden: die komplexen Fachwerttypen sind aus Attributen zusammengesetzt, während die einfachen Fachwerttypen den Subtypen entsprechen und daher konzeptuell keine Attribute aufweisen.
In bezug auf diese Aufteilung ist allerdings anzumerken, daß sich die komplexen Fachwerttypen bezüglich der Implementierung kaum von den einfachen Fachwerttypen unterscheiden: Obwohl ein einfacher Fachwerttyp konzeptuell keine Attribute aufweist, wird er programmiertechnisch durch eine Klasse mit einem einzigen Attribut realisiert, wobei der Typ dieses Attributs den Basistyp darstellt. Der Grund für die Einführung eines solchen Attributs ist, daß einfache Typen in Programmiersprachen nicht durch Ableiten eines Basistyps realisiert werden können. So ist es beispielsweise in Java nicht möglich, einen Subtyp der Klasse String oder der Wrapperklassen wie *Integer* und *Float* zu bilden.
Ein einfacher Fachwerttyp stellt also einen komplexen Fachwerttyp mit einem einzigen Attribut dar und könnte auch als solcher modelliert werden. Es ist allerdings sinnvoll, die einfachen Fachwerttypen von den komplexen abzugrenzen, da dadurch das Konzept des Basistyps aufrecht erhalten und von dem einzelnen Attribut abstrahiert werden kann. Zusätzlich vereinfacht sich für einen Entwickler das Erzeugen eines einfachen Fachwerttyps, da das Erstellen des einzelnen Attributs

sowie einige weitere Arbeitsschritte automatisiert werden können (vgl. Abschnitt 7.2.3).
Neben den einfachen und komplexen Typen gibt es noch die Klasse von Fachwerttypen, die ein Subtyp von *dvEnumerable* sind. Sie werden im weiteren als *aufzählbare Fachwerttypen* bezeichnet. Auch sie haben konzeptuell keine Attribute, ähneln also insofern den einfachen Fachwerttypen. Entsprechend werden Aufzählungen in der XML-Schema-Spezifikation auch als *simpleTypes* definiert, wobei jeder Aufzählungswert durch ein *enumeration*-Element angegeben wird. Obwohl Aufzählungstypen in dieser Arbeit keinen Schwerpunkt bilden, werden sie im folgenden gesondert behandelt, da sie im JWAM-Rahmenwerk einen speziellen Supertyp und Aufbau besitzen (vgl. Abschnitt 4.3.2).
Für den Entwurf unseres Ansatzes sollen also folgende Arten von Typen unterstützt werden:

- einfache Fachwerttypen
- komplexe Fachwerttypen
- aufzählbare Fachwerttypen

In den folgenden Abschnitten wird herausgearbeitet, welche Struktur diese einzelnen Arten von Fachwerttypen jeweils aufweisen und wie sie sich mit Hilfe von XML-Fachwerttypbeschreibungen formal beschreiben lassen.

6.2 Aufbau der einfachen und komplexen Fachwerttypen

Wie in Abschnitt 4.3.2 herausgearbeitet wurde, bietet die Einbindung des Wertkonzepts in das JWAM-Rahmenwerk unter anderem Lösungen für folgende Konzepte:

1. Standardwert
2. Stringrepräsentation
3. Wertebereichseinschränkungen

In diesem Abschnitt soll untersucht werden, anhand welcher Methoden diese Konzepte in einfachen und komplexen JWAM-Fachwerttypen umgesetzt werden. Anschließend wird herausgearbeitet, welche Informationen zur Generierung dieser Methoden gebraucht werden und wie sich diese Informationen in XML-Notation darstellen lassen. Die dabei verwendeten Beispiele stammen aus dem Fachwerttyp *dvQuantity*, der im JWAM-Rahmenwerk enthalten ist. Teilweise wurde der Quellcode dieses Fachwerttyps leicht angepaßt, um einige Konzepte besser verdeutlichen zu können.
Anhand dieser Überlegungen wird herausgearbeitet, wie die vom Fachwertwerkzeug zu verwendenden XML-Fachwerttypbeschreibungen aufgebaut sein müssen. Ein vollständiges Beispiel für eine komplexe XML-Fachwerttypbeschreibung findet sich im Anhang.

In jedem Fall muß eine Fachwerttypbeschreibung die Information über den Namen des beschriebenen Fachwerttyps enthalten. Der Name wird unter anderem benötigt, um die Klassendefinition für die entsprechende JWAM-Klasse erstellen zu können.

Eine Fachwerttypbeschreibung für den Fachwerttyp *dvQuantity* enthält daher folgendes Element:

```
<NAME>dvQuantity<\NAME>
```

Die weiteren Elemente von XML-Beschreibungen einfacher und komplexer Fachwerttypen werden in den folgenden Abschnitten herausgearbeitet.

6.2.1 Attribute

Sowohl einfache als auch komplexe Fachwerttypen verfügen über Attribute, die die Werte der Fachwertexemplare repräsentieren. Einfache Fachwerttypen haben nur ein einziges Attribut, welches im allgemeinen den Namen *value* trägt. Komplexe Fachwerttypen können beliebig viele Attribute haben, die jeweils durch einen Namen und einen Typ gekennzeichnet sind. Gemeinsam ist beiden Arten von Fachwerttypen, daß ihre Attribute Zugriffsfunktionen und Wertebereichseinschränkungen aufweisen können. Zusammengefaßt müssen also folgende Informationen für die Generierung und den Umgang mit den Attributen vorliegen: Name, Typ, Einschränkungen und die Angabe, ob eine Zugriffsfunktion für das jeweilige Attribut generiert werden soll.

Der Beispiel-Fachwerttyp *dvQuantity* ist ein komplexer Fachwerttyp mit zwei Attributen:

```
/**
 * the value
 */
private int _value;

/**
 * the unit
 */
private String _unit;
```

Für das Attribut namens *_unit* existiert eine Zugriffsfunktion, welche den Wert des Attributs zurückliefert:

```
/**
   * returns the unit of the quantity
   * @require isDefined()
   * @ensure result != null
   * @since <= 1.5.1
   */
  public String unit ()
   {
     Contract.require(isDefined(),this, "domain value is defined");
     return _unit;
   };
```

Um die oben angeführten Attributdefinitionen und Zugriffsfunktionen generieren zu können, sind Informationen über die Namen und Typen der Attribute notwendig

sowie darüber, ob eine Zugriffsfunktion für ein Attribut erzeugt werden soll oder nicht. Diese Informationen lassen sich folgendermaßen in XML-Notation darstellten:

```
<ATTRIBUT zugriff="nein">
  <ATTRIBUT_NAME>value</ATTRIBUT_NAME>
  <ATTRIBUT_TYP>int</ATTRIBUT_TYP>
  <ATTRIBUT_EINSCHRAENKUNGEN>
     (potentielle Einschränkungen, vgl. Abschnitt 6.2.2)
  </ATTRIBUT_EINSCHRAENKUNGEN>
</ATTRIBUT>

<ATTRIBUT zugriff="ja">
  <ATTRIBUT_NAME>unit</ATTRIBUT_NAME>
  <ATTRIBUT_TYP>String</ATTRIBUT_TYP>
  <ATTRIBUT_EINSCHRAENKUNGEN>
     (potentielle Einschränkungen, vgl. Abschnitt 6.2.2)
  </ATTRIBUT_EINSCHRAENKUNGEN>
</ATTRIBUT>
```

Beim Vergleich dieser XML-Notation mit dem zuvor angeführten Quellcode fällt auf, daß sich die Attributnamen unterscheiden. Im Quellcode wird vor den Namen des Attributs (*value*) ein Unterstrich eingefügt (*_value*). Dies ist eine Programmierkonvention im JWAM-Rahmenwerk, die dazu dient, die Attribute eines Fachwerttyps optisch von anderen für die Programmierung notwendigen Variablen abzugrenzen. Diese Kennzeichnung ist also durch eine Programmierkonvention bedingt und wird daher nicht in der formalen Fachwerttypbeschreibung sichtbar, denn diese soll ja gerade von den programmiertechnischen Aspekten von Fachwerttypen abstrahieren.
Die Attribute des Fachwerttyps *dvQuantity* weisen keine Wertebereichseinschränkungen auf, daher sind in der obigen XML-Fachwerttypbeschreibung von *dvQuantity* keine entsprechenden Elemente vorhanden. Generell jedoch können diese Einschränkungen in der XML-Fachwertbeschreibung innerhalb der Attributdefinition angegeben werden, daher wurden sie im obigen Beispiel an der entsprechenden Stelle angedeutet. Die Möglichkeiten, den Wertebereich einzelner Attribute zu beschränken, werden in Abschnitt 6.2.2 näher erläutert.
Neben dem Namen und den Wertebereichseinschränkungen muß für ein Attribut auch dessen Typ angegeben werden. Dabei stellt sich die Frage, welche Typen für die Attribute unterstützt werden sollen. Da es nicht sinnvoll ist, prinzipiell veränderliche Objekttypen in unveränderlichen Werttypen einzusetzen, sollten für die Modellierung von Fachwerttypen grundsätzlich Werttypen als Attributtypen eingesetzt werden. Daher sollen als Attributtypen zum einen die primitiven Java-Datentypen unterstützt werden, wie beispielsweise *int* und *float.* Zum anderen sollen JWAM-Fachwerttypen unterstützt werden. Allerdings sind nur diejenigen Fachwerttypen als Attributtypen unterstützbar, welche aus einer Zeichenkette initialisiert werden können, da diese Eigenschaft an verschiedenen Stellen gefordert wird. Beispielsweise müssen beim Erzeugen eines neuen Fachwerts aus einer Zeichenkette alle Attribute aus den entsprechenden Abschnitten dieser Zeichenkette erzeugbar sein. Auch beim Erzeugen von Standardwerden und speziellen Werten ist dies der Fall.

Objekttypen wie *Point* und Arrays sowie Fachwerttypen, deren Exemplare nicht aus einer Zeichenkette erzeugbar sind, werden nicht als Attributtyp unterstützt. Sinnvoll wäre allerdings noch die Unterstützung von unveränderlichen Typen, die in Java eingebaut sind, wie *BigDecimal*. Darauf wurde aber bislang verzichtet, da mit der Anzahl der unterstützten Typen auch die Komplexität der Codegenerierung deutlich steigt.

6.2.2 Wertebereichseinschränkungen

Die Attribute von Fachwerttypen können in ihrem Wertebereich eingeschränkt sein. Ein Fachwerttyp wie *dvQuantity* könnte beispielsweise eine Einschränkung für das Attribut *value* enthalten, welche in folgender Form notiert sein könnte:

```
/**
 * Is the given string a valid string representation?
 * @require s != null
 * @require canCreateFromString()
 * @since <= 1.5.1
 */
protected boolean isValidImpl (String s)
{
   Contract.require(s != null, this, "s not null");
   Contract.require(canCreateFromString(), this,
      "can create a domain value from a String representation");

   boolean result = false;

   StringTokenizer token = new StringTokenizer(s);
   int value = Integer.valueOf(token.nextToken()).intValue();

   if (value < 1)
      result = false;

   return result;
}
```

Um solche Wertebereichseinschränkungen generieren zu können, müssen Fachwerttypbeschreibungen die entsprechenden Informationen enthalten. Allerdings gibt es prinzipiell eine große Anzahl von möglichen Einschränkungen, es können daher nicht alle denkbaren Einschränkungsmöglichkeiten unterstützt werden. Bei der Auswahl der zu unterstützenden Wertebereichseinschränkungen bietet XML-Schema eine gute Orientierungshilfe, da Wertebereichsvalidierung in diese Sprache eine entscheidende Rolle spielt (vgl. Abschnitt 4.2.2).
In Anlehnung an die Einschränkungsmöglichkeiten bei XML-Schema sollen daher die in Tabelle 6-1 aufgeführten Wertebereichseinschränkungen unterstützt werden. Da nicht alle Einschränkungen auf alle Typen angewendet werden können, ist in der Tabelle zudem angegeben, für welche Attributtypen die Einschränkungen jeweils eingesetzt werden können.

Einschränkung	Verwendbar für
Minimum, Maximum	int, long, float, double
Länge, minimale Länge, maximale Länge	String
Anzahl Nachkommastellen	float, double
Anzahl signifikanter Stellen	float, double
Aufzählung	String
Muster	String

Tabelle 6-1: Wertebereichseinschränkungen

Die Angaben *Minimum* und *Maximum* sind wichtige Einschränkungsmöglichkeiten, mit denen anwendungsübergreifend Zahlenbereiche spezifiziert werden können. Beispielsweise können durch diese Einschränkungen Attribute definiert werden, die nur positive Zahlen zulassen, indem als Minimum der Wert Null angegeben wird.
Die Angaben zur Länge können für die Einschränkung von Zeichenketten verwendet werden. Dies erlaubt beispielsweise die Definition von Attributen, welche Eingaben für längenbegrenzte Eingabefelder repräsentieren.
Die Einschränkung *Anzahl Nachkommastellen* eignet sich beispielsweise für die Definition von Attributen zur Repräsentation von Geldbeträgen, denn diese weisen üblicherweise zwei Nachkommastellen auf.
Eine relativ spezielle Einschränkung stellt die Angabe der *Anzahl signifikanter Stellen* dar, die insbesondere im Zusammenhang mit Messungen benötigt wird. Sie gibt die Anzahl der Ziffern ab der ersten von Null verschiedenen Ziffer an und dient damit zur Angabe der Genauigkeit von Meßergebnissen. Diese Einschränkung war ursprünglich für die Aufnahme in den XML-Schema-Standard vorgesehen, ist in der aktuellen Version dieses Standards allerdings nicht enthalten.
Zur Wertebereichseinschränkung *Aufzählung* ist anzumerken, daß diese nicht dazu dient, einen aufzählbaren Fachwerttyp zu definieren. Statt dessen bezieht sich diese Einschränkung auf den Wertebereich eines Attributs: Sie wird benutzt, um die möglichen Werte genau vorzugeben, die ein Attribut annehmen kann. Dies kann beispielsweise zur Definition eines Attributs verwendet werden, welches die zu einer Uhrzeit gehörige Tageszeit repräsentiert, wie etwa *am*/*pm* im englischen Uhrzeitformat.
Die Einschränkung *Muster* ermöglicht es, die für ein Attribut zulässigen Werte durch einen regulären Ausdruck zu spezifizieren. Ein solches Muster könnte beispielsweise der reguläre Ausdruck *\d{5}* sein, welcher fünfstellige Zahlenkombinationen repräsentiert. Dieses Muster kann beispielsweise zur Definition von deutschen Postleitzahlen verwendet werden. Es ist anzumerken, daß Kenntnisse über die Notation zur Angabe eines Musters nicht vorausgesetzt werden können. Es ist daher dafür zu sorgen, daß im Zusammenhang mit der Einschränkung *Muster* Informationen über die zugehörige Notation zur Verfügung stehen, beispielsweise in Form eines entsprechenden Hilfetextes.

Da sich Wertebereichseinschränkungen jeweils auf ein Attribut beziehen, werden sie in der Fachwerttypbeschreibung innerhalb des Elements *Attribut* angegeben:

```
<ATTRIBUT zugriff="nein">
  <ATTRIBUT_NAME>value</ATTRIBUT_NAME>
  <ATTRIBUT_TYP>int</ATTRIBUT_TYP>
  <ATTRIBUT_EINSCHRAENKUNGEN>
    <MINIMUM inclusive="true">1</MINIMUM>
  </ATTRIBUT_EINSCHRAENKUNGEN>
</ATTRIBUT>
```

Innerhalb des Elements `<ATTRIBUT_EINSCHRAENKUNGEN>` können auch mehrere Einschränkungen definiert werden; jedoch nur solche, die für den Typ des Attributs gelten (vgl. Tabelle 6-1).

6.2.3 Stringrepräsentation

Für Fachwerttypen ist es wichtig, daß sie eine Stringrepräsentation besitzen, um Werte erzeugen und darstellen zu können. Allerdings sind nicht alle JWAM-Fachwerttypen aus einer Stringrepräsentation erzeugbar. Dazu zählen Fachwerttypen, die eine Sammlung von Werten darstellen, so daß es keinen Sinn macht, diese Wertesammlung aus einer einzigen Zeichenkette zu erzeugen. Werte solcher Typen werden statt dessen aus den Werten der einzelnen Attribute erzeugt.

Fachwerttypen sollen nicht nur aus einer Zeichenkette erzeugbar sein, sondern auch als Zeichenkette ausgegeben werden können. Zudem soll eine an einen Fachwerttyp übergebene Zeichenkette daraufhin geprüft werden können, ob sie eine gültige Fachwertrepräsentation darstellt.
Es wäre wünschenswert, wenn die Angabe der Stringrepräsentation ausreichen würde, um den Quellcode für die drei oben genannten Funktionalitäten generieren zu können (vgl. Abschnitt 5.1.5). Auf die einzelnen Funktionalitäten wird im folgenden nacheinander eingegangen.

- **Erzeugung eines Fachwerts aus einem String**
  Die Methode *valueImpl* nutzt die Informationen über die Stringrepräsentation eines Fachwerts, um aus einem übergebenen String die Parameter herauszufiltern, welche für die Erzeugung des Fachwerts benötigt werden. Diese Parameter werden dann dem Konstruktor übergeben, welcher den entsprechenden Fachwert zurückliefert.
  Die Methode *valueImpl* kann jedoch nur dann zur Erzeugung eines Fachwerts verwendet werden, wenn die Exemplare des Fachwerttyps aus einer Zeichenkette erzeugbar sind. Diese Information muß daher in der XML-Fachwerttypbeschreibung enthalten sein, etwa in der folgenden Form:

  ```
  <AUS_STRING_ERZEUGBAR>true</AUS_STRING_ERZEUGBAR>
  ```

  Der als Beispiel gewählte Fachwerttyp *dvQuantity* ist aus einer Zeichenkette erzeugbar und enthält daher die Methode *valueImpl.* Dieser Methode wird eine Zeichenkette übergeben, aus welcher sich die Werte für die Attribute *value* und *unit*

herausfiltern lassen, um anschließend daraus den entsprechenden Fachwert zu erzeugen:

```
/**
 * Returns the quantity domain value with the given string
 * representation.
 * @require repr != null
 * @require isValid(s)
 * @require canCreateFromString()
 * @ensure result != null
 * @since <= 1.5.1
 */
protected DomainValue valueImpl (String s)
{
   Contract.require(s != null, this, "s not null");
   Contract.require(canCreateFromString(), this,
      "can create a domain value from a stringrepresentation");
   Contract.require(isValid(s), this, "s is valid: " + s);

   StringTokenizer token = new StringTokenizer(s, " ");

   int value = Integer.valueOf(token.nextToken());  // (*)
   String unit = token.nextToken();
   return value(value, unit);
}
```

Um die Parameter aus dem gegebenen String herausfiltern zu können, muß bekannt sein, welcher Teil der Zeichenkette zu welchem Parameter gehört. Zu diesem Zweck werden in der Regel Trennzeichen verwendet. Im obigen Beispiel dient das Leerzeichen als Trennzeichen, welches dem *StringTokenizer* übergeben werden muß.
Um die Methode *valueImpl* generieren zu können, sind also folgende Informationen notwendig: zum einen die Angabe darüber, welche Attributewerte bei der Stringrepräsentation eines Fachwerts eine Rolle spielen und welche Trennzeichen verwendet werden, um diese Attributwerte voneinander abzugrenzen. Zum anderen muß der jeweilige Typ dieser Attribute bekannt sein, um entscheiden zu können, wie eine Zeichenkette in ein Attribut des betreffenden Typs umgewandelt werden kann. So muß im obigen Beispiel bekannt sein, daß es sich beim Attribut *value* um ein Attribut vom Typ *int* handelt, um die mit "(*)" markierte Zeile generieren zu können.
In der XML-Beschreibung eines Fachwerttyps könnten die notwendigen Informationen folgendermaßen repräsentiert werden:

```
<STRINGREPRAESENTATION>
   <ATTRIBUT_NAME>value/ATTRIBUT_NAME>
   <TRENNZEICHEN> </TRENNZEICHEN>
  <ATTRIBUT_NAME>unit</ATTRIBUT_NAME>
</STRINGREPRAESENTATION>
```

Die Informationen über die Typen der Attribute sind in den Attribut-Elementen enthalten, so daß sie bei der Stringrepräsentation nicht noch einmal explizit aufge-

führt werden müssen. Dazu müssen die angegebenen Attribute im XML-Dokument definiert sein, so wie in Abschnitt 6.2.1 beschrieben. Die Information über den Typ eines Attributs wird innerhalb der Stringrepräsentation über den Attributnamen referenziert.
Da bekannt sein muß, an welcher Stelle der Stringrepräsentation sich der Wert für welches Attribut befindet, muß die Stringrepräsentation in der XML-Fachwerttypbeschreibung zusätzlich Informationen über die Reihenfolge der Attribute enthalten. Diese Information ergibt sich implizit aus der Reihenfolge, in der die Attribute im Element *Stringrepräsentation* angegeben sind.

Im folgenden wird untersucht, ob die bisher aufgestellte Beschreibung der Stringrepräsentation genügend Informationen enthält, um auch die anderen beiden Methoden zu generieren, welche im Zusammenhang mit der Stringrepräsentation stehen.

- **Validierung eines Fachwerts**
  Die Methode *isValidImpl* testet, ob eine übergebene Zeichenkette einen gültigen Fachwert repräsentiert. Dazu muß überprüft werden, ob diese Zeichenkette derart aufgeteilt werden kann, daß die einzelnen Abschnitte in jene Parameter umgewandelt werden können, die für die Erzeugung des entsprechenden Fachwerts notwendig sind.

```
/**
 * Is the given string a valid string representation?
 * @require s != null
 * @require canCreateFromString()
 * @since <= 1.5.1
 */
protected boolean isValidImpl (String s)
{
  Contract.require(s != null, this, "s not null");
  Contract.require(canCreateFromString(), this,
    "can create a domain value from a stringrepresentation");

  boolean result = true;
  StringTokenizer token = new StringTokenizer(s, " ");
  if(token.countTokens() != 2){ result = false; }
  else
  {
    try
    {
      Integer.valueOf(token.nextToken());
    }
    catch(NumberFormatException ex)
    {
      // First value is not a number
      result = false;
    }
  }
  return result;
}
```

Ebenso wie in der Methode *valueImpl* müssen bei *isValidImpl* die Attribute mit Hilfe der Trennzeichen herausgefiltert werden und anschließend muß geprüft werden, ob sie in den betreffenden Attributtyp konvertiert werden können. Der einzige Unterschied zu *valueImpl* besteht darin, daß in der Methode *isValidImpl* kein neuer Fachwert erzeugt wird. Um die Gültigkeitsprüfung generieren zu können sind daher dieselben Informationen nötig wie für die Generierung von *valueImpl*: Attribute, Attributtypen, Reihenfolge der Attribute und Trennzeichen.
Für die Generierung der Methode *isValidImpl* können darüber hinaus noch Informationen über eventuelle Einschränkungen der Attribute notwendig sein. Diese Informationen sind in den Attributdefinitionen enthalten und können über den Attributnamen referenziert werden.

- **Ausgabe eines Fachwerts**
  Die Methode *toStringImpl* dient dazu, die Stringrepräsentation eines Fachwertes zu liefern. Da diese Methode in der Oberklasse für Fachwerttypen abstrakt ist, muß sie von jedem konkreten Fachwerttyp implementiert werden. In dem als Beispiel gewählten Fachwerttyp *dvQuantity* sieht diese Methode folgendermaßen aus:

```
/**
 * the string-representation
 * @require isDefined()
 * @ensure result != null
 * @since <= 1.5.1
 */
protected String toStringImpl ()
{
   Contract.require(isDefined(), this,
      "this domain value is defined");
   return _value + " " + _unit;
}
```

  Da das Ausgabeformat für einen Fachwert dem Format entsprechen muß, das für die Erzeugung eines Fachwerts verwendet wird, werden für den Rückgabewert von *toStringImpl* dieselben Informationen benötigt, die schon bei der Erzeugung eines Strings relevant waren: die Namen und Typen der verwendeten Attribute, sowie die Trennzeichen zwischen den Attributen.

Zusammengefaßt läßt sich sagen, daß die Informationen über Attribute, Trennzeichen und deren Reihenfolge ausreichen, um die oben angeführten Methoden generieren zu können. Die zuvor entwickelte Beschreibung der Stringrepräsentation deckt diese Informationen ab. Sie unterstützt allerdings nur einen bestimmten Standardfall: die Aneinanderreihung verschiedener Attribute, welche mit Trennzeichen voneinander abgegrenzt werden, wobei diese Reihe einen String als Präfix und einen String als Suffix aufweisen kann.
Denkbar wären noch weitere Arten von Stringrepräsentationen, wie beispielsweise Zusammensetzungen von Attributen ohne Trennzeichen. So können etwa Benutzernamen für Computersysteme mit Hilfe einer Zahl und einer Zeichenkette repräsen-

tiert werden („5sprecke“), wobei die unterschiedlichen Typen der beiden Komponenten es ermöglichen, sie voneinander zu trennen. In vielen anderen Fällen ist jedoch ohne Trennzeichen keine eindeutige Abgrenzung der einzelnen Komponenten einer Stringrepräsentation möglich. Beispielsweise wäre eine trennzeichenlose Anneinanderreihung von einem Attribut mit Typ *float* und einem mit Typ *int* problematisch: eine Stringrepräsentation wie *1,0573* etwa könnte sowohl aus den Zahlen *1,05* und *73* zusammengesetzt worden sein als auch aus *1,057* und *3*. Eine solche Stringrepräsentation wäre also nicht eindeutig und damit problematisch.
Ein weiterer Speziallfall von Stringrepräsentationen sind beispielsweise solche, die mit Abfragen auf Attribute vom Typ *boolean* arbeiten. Dabei könnte es beispielsweise vom Typ eines Boolean-Attributs *istDM* abhängen, ob einem Betrag der String *DM* oder *Euro* angehängt wird.
Da im Rahmen dieser Arbeit nicht alle Spezialfälle abgedeckt werden können, wird nur der am häufigsten auftretende Standardfall der Stringrepräsentation unterstützt, welcher mit Trennzeichen arbeitet und keine Abfragen auf Attribute vom Typ *boolean* ermöglicht.

6.2.4 Standardwert und spezielle Werte

Als Standardwert wird ein besonderes Exemplar eines Fachwerttyps bezeichnet, für welches vom Entwickler ein ganz bestimmter Wert festgelegt wird. Dieser Standardwert kann beispielsweise eingesetzt werden, wenn in Feldern einer Anwendung bereits beim Start ein Vorschlag für einen Eintrag gegeben werden soll. In JWAM-Fachwerttypen wird der Standardwert mit Hilfe der Methode *defaultValue()* geliefert:

```
/**
   * Returns a default domain value used as the starting point
   *  for editing etc.
   * @ensure result != null
   * @since <= 1.5.1
   */
  public DomainValue defaultValue ()
  {
    return valueOf("0 [X]");
  }
```

Im Beispiel verwendet die Methode *defaultValue* zur Erzeugung des Standardwerts dessen Stringrepräsentation. Alternativ könnte der Standardwert auch aus festgelegten Parametern erzeugt werden. Anzahl und Typ dieser Parameter sind abhängig von den Attributen des Fachwerttyps, die konkreten Werte der Parameter müssen durch den Entwickler fest vorgegeben werden. Im obigen Beispiel würde dies wie folgt aussehen:

```
return value(0, "[X]");
```

Diese Alternative bietet den Vorteil, daß Standardwerte auch für Fachwerttypen angegeben werden können, die nicht aus einer Zeichenkette erzeugbar sind. Daher wird der Standardwert in der XML-Beschreibung mit Hilfe seiner Attribute beschrieben. Dabei werden Name und der konkrete Wert des jeweiligen Attributs

angegeben. Informationen zum Attributtyp sind über die Referenzierung des Attributnamens zu erhalten: dieser verweist auf die Definition des entsprechenden Attributs (vgl. Abschnitt 6.2.1), welche unter anderem die Information über den Attributtyp enthält. Die XML-Beschreibung des Standardwerts sieht demnach folgendermaßen aus:

```
<STANDARDWERT>
  <ATTRIBUT_WERT_PAAR>
    <ATTRIBUT_NAME>value</ATTRIBUT_NAME>
    <ATTRIBUT_WERT>0</ATTRIBUT_WERT>
  </ATTRIBUT_WERT_PAAR>
  <ATTRIBUT_WERT_PAAR>
    <ATTRIBUT_NAME>unit</ATTRIBUT_NAME>
     <ATTRIBUT_WERT>[X]</ATTRIBUT_WERT>
  </ATTRIBUT_WERT_PAAR>
</STANDARDWERT>
```

Spezielle Werte bezeichnen ähnlich wie der Standardwert ganz bestimmte, vom Entwickler festgelegte Exemplare eines Fachwerttyps. Im Unterschied zum Standardwert kann der Entwickler mehrere spezielle Werte festlegen und sie zur Unterscheidung mit Namen versehen. Spezielle Werte werden eingesetzt, um stets wiederkehrende Fachwertexemplare vorzudefinieren. Bei dem Fachwerttyp *dvQuantity* wäre beispielsweise wäre ein spezieller Wert denkbar, der den Zeitraum von 10 Jahren bezeichnet:

```
/**
   * Returns the special value "decade"
   * @ensure result != null
   */
  public DomainValue decade()
  {
    return value(10, "Jahre");
  }
```

In der XML-Fachwerttypbeschreibung können die speziellen Werte sehr ähnlich wie der Standardwert repräsentiert werden:

```
<SPEZIELLE_WERTE>
  <SPEZIELLER_WERT>
    <SPEZIELLER_WERT_NAME>decade</SPEZIELLER_WERT_NAME>
    <ATTRIBUT_WERT_PAAR>
      <ATTRIBUT_NAME>value</ATTRIBUT_NAME>
      <ATTRIBUT_WERT>10</ATTRIBUT_WERT>
    </ATTRIBUT_WERT_PAAR>
    <ATTRIBUT_WERT_PAAR>
      <ATTRIBUT_NAME>unit</ATTRIBUT_NAME>
      <ATTRIBUT_WERT>Jahre</ATTRIBUT_WERT>
    </ATTRIBUT_WERT_PAAR>
   </SPEZIELLER_WERT>
</SPEZIELLE_WERTE>
```

Da es mehrere spezielle Werte geben kann, werden diese von einem überliegendem Element *spezielle_werte* zusammengefaßt. Dies gilt auch, wenn wie im obigen Beispiel nur ein spezieller Wert existiert.
Auch bei den speziellen Werten dient der Attributname zur Referenzierung des Attribut-Elements, welches unter anderem die Information über den Attributtyp enthält.

6.3 Aufbau der aufzählbaren Fachwerttypen

Als *aufzählbare Fachwerttypen* werden im JWAM-Rahmenwerk jene Fachwerttypen bezeichnet, welche von der Oberklasse *dvEnumerable* erben. Diese Fachwerttypen haben als Basistyp im allgemeinen einen Zeichenkettentyp (*dvString* oder *String*). Ihr Wertebereich wird nicht durch Einschränkugen wie *Minimum* und *Maximum* (vgl. Abschnitt 6.2.2) definiert, sondern durch eine Aufzählung der enthaltenen Werte.
Der Aufbau aufzählbarer Fachwerttypen wird in den folgenden Abschnitten anhand des Beispiels *dvEmployeePosition* dargestellt. Dieser aufzählbare Fachwerttyp ist im JWAM-Rahmenwerk enthalten und umfaßt die Werte *Manager* und *Team member*.

6.3.1 Attribut

Aufzählbare Fachwerttypen haben nur ein einziges Attribut, welches standardmäßig den Namen *value* trägt. In einem konkreten Fachwert repräsentiert dieses Attribut ein Element der Aufzählung, welche durch den Fachwerttyp definiert wird. Beim Aufzählungstyp *dvEmployeePosition* kann das Attribut *value* also die Werte *Manager* oder *Team member* annehmen.

6.3.2 Wertebereich

Wie bereits oben erläutert, wird der Wertebereich von aufzählbaren Fachwerttypen dadurch angegeben, daß alle zugehörigen Werte explizit aufgeführt werden.
Es werden bei aufzählbaren Fachwerttypen zwei Arten unterschieden: solche, deren Wertebereich bereits bei der Definition des Typs endgültig festgelegt wird und solche, deren Wertebereich zur Laufzeit dynamisch erweitert werden kann. Der Fachwerttyp *dvEmployeePosition* ist nicht dynamisch erweiterbar. Dies wird in der XML-Beschreibung folgendermaßen gekennzeichnet:

```
<DYNAMISCH_ERWEITERBAR>false</DYNAMISCH_ERWEITERBAR>
```

Bei solchen statischen Fachwerttypen wird für jedes Element der Aufzählung eine Methode zum Erstellen eines entsprechenden Fachwerts definiert:

```
/**
 * @return the enumeration value 'Manager'
 */
public dvEmployeePosition managerValue()
{
   DomainValue result = new dvEmployeePosition(this, MANAGER);
   return (dvEmployeePosition) registerAndReturnValue(result,
    result);
}
```

Eine entsprechende Methode existiert auch für das zweite Element der Aufzählung, *Team member*.
Für die Angabe der Aufzählungswerte sind also die Informationen über den Namen des Fachwerttyps und die Elemente der Aufzählung notwendig. Eine XML-Fachwerttbeschreibung für aufzählbare Fachwerttypen enthält daher die folgenden Elemente:

```
<NAME>dvEmployeePosition</NAME>

<WERT>Manager</WERT>
<WERT>Team member</WERT>
```

Bei Fachwerttypen, die dynamisch erweiterbar sind, können zur Laufzeit neue Aufzählungselemente hinzugefügt werden. Für derart hinzugefügte Elemente wird keine direkte Zugriffsoperation definiert. Der Zugriff auf ein solches Element kann nur über eine Methode erfolgen, die ein Objekt zum Itererieren über alle Elemente der Aufzählung zurückliefert.

6.3.3 Stringrepräsentation

Wie in Abschnitt 6.2.3 dargestellt, wird die Stringrepräsentation bei einfachen und komplexen Fachwerttypen für die Generierung von drei verschiedenen Methoden genutzt: *valueImpl*, *isValidImpl* und *toStringImpl*. Im folgenden werden diese drei Methoden im Zusammenhang mit aufzählbaren Fachwerttypen betrachtet.

- **valueImpl** und **isValidImpl**
  Diese beiden Methoden zum Erzeugen eines Fachwerts aus einer Zeichenkette und zur Gültigkeitsprüfung einer Fachwertrepräsentation müssen bei Aufzählungstypen oftmals gar nicht implementiert werden, weil die Erzeugbarkeit aus einer Zeichenkette keine wesentliche Rolle spielt. Dies ist bei solchen Aufzählungstypen der Fall, die nur mit Hilfe von Auswahllisten der Benutzungsoberfläche und nicht durch frei wählbare Zeichenketten generiert werden.
  Für den Fall, daß Aufzählungstypen doch aus einer Zeichenkette erzeugbar sein sollen, können die beiden Methoden *valueImpl* und *isValidImpl* in Abhängigkeit der Zeichenketten der einzelnen Aufzählungselemente implementiert werden. Bei dem Fachwerttyp *dvEmployeePosition* würde eine Zeichenkette beispielsweise genau dann einen gültigen Fachwert repräsentieren, wenn es sich um eine der beiden Zeichenketten "Manager" oder "Team member" handelt, da diese die beiden einzigen Aufzählungselemente repräsentieren. Ebenso kann aus diesen Zeichenketten der entsprechende Wert erzeugt werden. Zur Generierung der Methoden *valueImpl* und *isValidImpl* wird bei einem Aufzählungstyp demnach nur die Information benötigt, welche Aufzählungselemente dieser Typ umfaßt.

- **toStringImpl**
  Diese Methode ist in vereinfachter Form auch in aufzählbaren Fachwerttypen enthalten. Da Werte eines aufzählbaren Fachwerttyps einen einfachen Aufbau besitzen, können sie von der Methode *toStringImpl* ohne weiteren Aufwand als

Zeichenkette ausgegeben werden. Dazu wird einfach die Stringrepräsentation des Attributs zurückgeliefert, welches das Aufzählungselement speichert:

```
/**
 * Returns the string representation of this
 * employee position.
 */
protected String toStringImpl()
{
  return _value.toString();
}
```

Es werden also bei aufzählbaren Fachwerttypen keine speziellen Informationen über die Stringrepräsentation benötigt, daher muß dieses Element auch in der XML-Fachwerttypbeschreibung nicht enthalten sein.

6.3.4 Sortierung

Die Elemente einer Aufzählung können nach verschiedenen Kriterien sortiert sein, beispielsweise alphabetisch oder nach semantischen Gesichtspunkten. Die Sortierung gibt für aufzählbare Fachwerttypen die Reihenfolge an, in der ein Iterator-Objekt die Elemente der Aufzählung durchlaufen soll.

```
/**
 * Returns a new iterator for this enumerable factory.
 * @ensure result != null
 */
public EnumerableIterator iterator ()
{
  return new EnumerableIterator(new dvEnumerable[]{
    (dvEmployeePosition) undefinedValue(),
    managerValue(), teamMemberValue()});
}
```

Der Entwickler kann nur dann eine bestimmte Reihenfolge der Aufzählungselemente angeben, wenn sich die Anzahl der Elemente nicht mehr verändert, das heißt, wenn der aufzählbare Fachwerttyp nicht dynamisch erweiterbar ist. Diese Sortierung wird in der XML-Beschreibung durch die Reihenfolge der Elemente repräsentiert. Für das obige Beispiel muß daher in der XML-Beschreibung das Aufzählungselement *Manager* vor dem Element *Team member* angegeben sein:

```
<WERT>Manager</WERT>
<WERT>Team member</WERT>
```

Im Quellcodebeispiel ist zusätzlich zu diesen beiden Werten noch der undefinierte Wert aufgeführt. Dieser muß in der Fachwerttypbeschreibung allerdings nicht angegeben werden, da er stets zum Wertebereich gehört und somit automatisch eingefügt werden kann. In der Sortierung steht er immer als erstes Element.

Eine direkte Angabe der Reihenfolge durch die Fachwerttypbeschreibung wie im obigen Beispiel ist bei dynamisch erweiterbaren Aufzählungstypen nicht möglich. Bei dynamisch erweiterbaren Aufzählungstypen ist statt dessen nur die Angabe von allgemeinen Sortierungskriterien denkbar, welche auf die jeweils aktuellen Aufzählungselemente anzuwenden sind. So könnten beispielsweise die bereits bei der Definition des Fachwerttyps festgelegten Elemente gemeinsam mit den zur Laufzeit hinzugefügten Elementen alphabetisch sortiert werden. Solche Sortierungskriterien können beim Fachwertwerkzeug nicht angegeben werden. Standardmäßig wird bei dynamisch erweiterbaren Aufzählungstypen die alphabetische Sortierung verwendet. Wird eine andere Sortierung erwünscht, so muß diese nachträglich im generierten Code manuell implementiert werden.

6.4 Erweiterungen von Fachwerttypbeschreibungen

Für die Generierung von JWAM-Fachwerten werden neben den fachwertspezifischen Informationen noch weitere Informationen benötigt. Dazu zählen zum einen programmiersprachenabhängige Aspekte: so ist beispielsweise die Angabe des Fachwerttyp-Packages notwendig, da das JWAM-Rahmenwerk auf Java basiert. Ohne diese Angabe wäre der erzeugte JWAM-Fachwertcode unter Umständen nicht kompilierbar, ohne zuvor manuell das Package im Quellcode einzutragen. Dies gilt auch für Packages, welche gegebenenfalls im Zusammenhang mit Attributen benötigt werden, die einen Fachwerttyp besitzen: Denn enthält ein Fachwerttyp solche Attribute, wie beispielsweise *dvString*, so müssen die entsprechenden Packages im Quellcode vermerkt sein, um den Fachwertcode kompilieren zu können.
Eine Reihe weiterer Informationen sind JWAM-spezifisch: im JWAM-Rahmenwerk ist es üblich, Fachwertklassen mit Kommentaren zu Autor, Copyright und Version der zugehörigen Anwendung zu versehen. Das Fachwertwerkzeug unterstützt diese Angaben, um dem Benutzer Schreibarbeit zu ersparen; der erzeugte JWAM-Fachwertcode ist jedoch auch beim Fehlen dieser JWAM-spezifischen Informationen kompilierbar.
Die Angaben *Fachwert-Package*, *Autor des Fachwerts*, *Copyright* und *Version* werden im folgenden als *Projektoptionen* bezeichnet, da sie bei allen JWAM-Fachwerttypen übereinstimmen, welche in einem bestimmten fachlichen Zusammenhang entwickelt wurden. Die Angaben über Packages, die für die Unterstützung von Attributtypen benötigt werden, beziehen sich hingegen stets nur auf einen einzigen konkreten Fachwerttyp. Diese zusätzlichen Package-Informationen können gemeinsam mit den Projektoptionen als Erweiterungen für Fachwerttypbeschreibungen aufgefaßt werden, so daß die in den vorangegangenen Abschnitten entwickelten Fachwerttypbeschreibungen folgendermaßen ergänzt werden:

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

<ERWEITERUNGEN>
   <-- Projektoptionen -->
   <PACKAGENAME>domainvalue</PACKAGENAME>
   <AUTOR>Annika Spreckelmeyer, Martin Fuerter</AUTOR>
   <COPYRIGHT>AS&MF</COPYRIGHT>
   <VERSION>1.1</VERSION>

   <!-- Packages für Attributtypen -->
   <IMPORTS>
      <IMPORT> ... </IMPORT>
      ...
   </IMPORTS>
</ERWEITERUNGEN>

<FACHWERTTYP>
   [... vgl. Abschnitte 6.1 und 6.3]
</FACHWERTTYP>

</ERWEITERTER_FACHWERTTYP>
```

Solche Fachwertbeschreibungen, deren vollständige formale Definition im Anhang zu finden ist, bilden die Grundlage für das Fachwertwerkzeug, welches im Kapitel 7 vorgestellt wird.

6.5 Aufbau von Testklassen

Bei Anwendungen, die mit dem JWAM-Rahmenwerk entwickelt werden, wird für jede Klasse eine zugehörige Testklasse entwickelt, mittels derer die implementierten Klassen auf korrektes Verhalten geprüft werden können. Auch für Fachwerttypen werden solche Testklassen erstellt, deren Entwicklung einigen Schreibaufwand erfordert. Daher soll das im Rahmen dieser Diplomarbeit zu entwickelnde Fachwertwerkzeug nicht nur Fachwerttypen, sondern auch die zugehörigen Testklassen generieren können. Im folgenden soll untersucht werden, welche Elemente solche Testklassen enthalten und wie deren Generierung unterstützt werden kann.
Zum Generieren einer Testklasse wird zunächst der Name der Testklasse benötigt. Um die Diskussion der Testklassen zu verdeutlichen, wird eine Testklasse zum Fachwerttyp *dvQuantity* betrachtet:

```
<NAME>dvQuantity_Test</NAME>
```

Testklassen enthalten Tests für alle Methoden, die im zugehörigen Fachwerttyp enthalten sind. Dazu werden zuvor Testwerte definiert, für die verschiedene Eigenschaften und Methoden geprüft werden können:

```
dvQuantity q1 = dvQuantity.Factory.valueOf("43 Aepfel");
dvQuantity q2 = dvQuantity.Factory.valueOf("2 Aepfel");
dvQuantity q3 = dvQuantity.Factory.valueOf("3 Birnen");
```

Die Testklassen aller Fachwerttypen enthalten unter anderem folgende Tests:

- **Tests für die definierten Zugriffsfunktionen**
  Im Beispiel *dvQuantity* existiert ein solcher Test beispielsweise für die Zugriffsfunktion auf das Attribut *value*:
  ```
  assert(q2.value() == 2);
  ```

- **Tests der fachlichen Methoden**
  Ein Test der fachlichen Methode *multiply* kann beispielsweise folgendermaßen aussehen:
  ```
  assert(dvQuantity.Factory.multiply(q1, q2) == 6.0);
  ```

- **Tests der Validierungsfunktionen**
  Es wird geprüft, ob gültige Repräsentationen korrekterweise als gültig und ungültige als ungültig erkannt werden:
  ```
  assert(dvQuantity.Factory.isValidValue("43 Aepfel"));
  assert(!dvQuantity.Factory.isValidValue("43x Birnen"));
  ```

Um die oben genannten Tests generieren zu können, sind folgende Informationen notwendig:

1. **Der Name der getesteten Klasse**
   Dieser Name wird zum Referenzieren derjenigen Fachwertfabrik verwendet, welche für die Erzeugung der Testwerte, im obigen Beispiel *q1* bis *q3*, benötigt wird. Die XML-Beschreibung einer Testklase muß also folgende Zeile enthalten:

   ```
   <GETESTETE_KLASSE>dvQuantity</GETESTETE_KLASSE>
   ```

2. **Die Stringrepräsentation**
   Zum Testen der Validierungsfunktionen werden mehrere gültige und ungültige Fachwert-Repräsentationen des entsprechenden Fachwerttyps benötigt. Dabei wird mindestens eine gültige Fachwert-Repräsentation benötigt, aus der ein Testwert erzeugt werden kann. Solche Testwerte lassen sich folgendermaßen mit Hilfe von XML-Elementen darstellen:

   ```
   <TESTWERT gueltig="ja">
     <STRINGREP>43 Aepfel</STRINGREP>
   </TESTWERT>
   <TESTWERT gueltig="nein">
     <STRINGREP>43x Birnen</STRINGREP>
   </TESTWERT>
   ```

3. **Informationen über Zugriffsfunktionen**
   Zum Generieren der Tests eventueller Zugriffsfunktionen wird die Angabe benötigt, für welche Attribute eine Zugriffsfunktion definiert ist und welchen Wert diese Zugriffsfunktion für einen gegebenen Fachwert liefern muß. Für den Fachwert *q2* (*„2 Aepfel“*) muß die Zugriffsfunktion für das Attribut *value* beispielsweise den Wert *2* zurückliefern, wie im obigen Test `assert(q2.value() == 2)`.

Neben dem Wert wird der Name der Zugriffsfunktion benötigt, welcher sich direkt aus dem Namen des Attributs ableiten läßt und daher nicht eigens im XML-Dokument enthalten sein muß.
Die Informationen, die zum Generieren von Tests für Zugriffsfunktionen notwendig sind, können in XML-Notation folgendermaßen dargestellt werden:

```
<TESTWERT gueltig="ja">
  <STRINGREP einaus="gleich">2 Aepfel</STRINGREP>
  <ATTRIBUT zugriff="ja">
    <ATTRIBUT_NAME>value</ATTRIBUT_NAME>
    <ATTRIBUT_TYP>int</ATTRIBUT_TYP>
    <ATTRIBUT_WERT>2</ATTRIBUT_WERT>
  </ATTRIBUT>
</TESTWERT>
```

Der Typ des zurückgelieferten Werts muß angegeben werden, damit der Attributwert als Literal des erforderlichen Typs dargestellt werden kann. Beispielsweise müßten bei der Generierung eines String-Literals die einschließenden Anführungszeichen generiert werden.

4. **Fachliche Methoden**
   Das Testen von fachlichen Methoden kann nicht unterstützt werden, weil diese Methoden nicht zum allgemeinen Aufbau von Fachwerttypen gehören, sondern speziell zu konkreten Fachwerttypen. Sie sind daher auch nicht Teil der Fachwerttypbeschreibung und werden somit nicht durch das Fachwertwerkzeug unterstützt.

In bezug auf die Angabe der Stringrepräsentation tritt ein Sonderfall auf, wenn die Exemplare eines Fachwerttyps nicht aus Zeichenketten erzeugt werden können. Die Testwerte müssen dann aus Initialisierungswerten für die Attribute erzeugt werden. Statt gültiger und ungültiger Stringrepräsentationen werden in einem solchen Fall Repräsentationen benutzt, die im folgenden *Attributrepräsentationen* genannt werden sollen. Eine solche Attributerepräsentation ist gegeben, wenn allen Attributen eines Fachwerttyps ein Attributwert zugeordnet ist:

```
<TESTWERT>
  <ATTRIBUTREP>
    <ATTRIBUT zugriff="ja">
      <ATTRIBUT_NAME>value</ATTRIBUT_NAME>
      <ATTRIBUT_TYP>int</ATTRIBUT_TYP>
      <ATTRIBUT_WERT>43</ATTRIBUT_WERT>
    </ATTRIBUT>

    <ATTRIBUT zugriff="nein">
      <ATTRIBUT_NAME>unit</ATTRIBUT_NAME>
      <ATTRIBUT_TYP>String</ATTRIBUT_TYP>
      <ATTRIBUT_WERT>Aepfel</ATTRIBUT_WERT>
    </ATTRIBUT>
  </ATTRIBUTREP>
</TESTWERT>
```

Eine Attributrepräsentation kann auch zusätzlich zu einer Stringrepräsentation angegeben werden.

Aus den gegebenen Angaben für Testwerte können einige zusätzliche Tests generiert werden, welche häufig in Testklassen für Fachwerttypen zu finden sind. Diese prüfen zum einen, ob die Stringrepräsentation eines Fachwerts derjenigen entspricht, aus der der Fachwert erzeugt wurde. Wurde ein Fachwert beispielsweise aus der Zeichenkette „*3 Birnen*“ erzeugt, so muß die Methode *toString* genau diese Zeichenkette zurückliefern. Im Beispiel *dvQuantity* wurde der Wert *q3* aus der Zeichenkette *„3 Birnen“* erzeugt. Es muß also folgender Test erfolgreich durchführbar sein:

```
assertEquals(q3.toString(), "3 Birnen");
```

Wenn ein Fachwert sowohl aus einer Zeichenkette als auch aus gültigen Initalisierungswerten für die Attribute erzeugt werden kann, muß zudem geprüft werden, ob diese beiden Möglichkeiten zu demselben Fachwert führen. Im Beispiel *dvQuantity* sieht ein solcher Testfall folgendermaßen aus:

```
assertEquals(q1,
  dvQuantity.Factory.instance().value(43,"Aepfel"));
```

6.6 Zusammenfassung

In diesem Kapitel wurde zunächst herausgearbeitet, daß drei Arten von Fachwerttypen unterschieden werden können: einfache, komplexe und aufzählbare Fachwerttypen. Anschließend wurde für jede dieser Arten von Fachwerttypen der jeweilige Aufbau analysiert und eine XML-Beschreibung zur Repräsentation dieses Aufbaus entwickelt. Die formale Beschreibung, welche Elemente diese XML-Beschreibungen enthalten, kann mit Hilfe einer sogenannten DTD (*Document Type Definition*) erfolgen, welche im Anhang zu finden ist.
Neben den XML-Beschreibungen für verschiedene Arten von Fachwerttypen wurde auch eine XML-Beschreibung für die Repräsentation von Testklassen entwickelt, da diese Testklassen im JWAM-Rahmenwerk stets für Fachwerttypen erstellt werden müssen.
Im folgenden Kapitel wird ein Werkzeug beschrieben, das zum Erstellen und Verarbeiten der in diesem Kapitel entwickelten XML-Beschreibungen eingesetzt werden kann.

7 Beschreibung des Fachwertwerkzeugs

In diesem Kapitel wird die Implementierung eines Fachwertwerkzeug beschrieben, welches Möglichkeiten bietet, die in Kapitel 6 entwickelten XML-Beschreibungen zu erstellen und aus diesen die entsprechenden JWAM-Fachwertklassen zu generieren.

7.1 Aufbau des Systems

Zur Erläuterung des Systemaufbaus diene das folgende Szenario: Will ein Entwickler einen neuen Fachwerttyp konstruieren, so muß er sich als erstes überlegen, zu welcher der drei Arten von Fachwerttypen der geplante Typ zählt, also ob ein einfacher, ein komplexer oder ein aufzählbarer Fachwerttyp erzeugt werden soll. In den weiteren Arbeitsschritten werden dann die Eigenschaften des zu konstruierenden Fachwerttyps festgelegt. Die dafür anzugebenden Informationen stehen im Zusammenhang mit der Art des zu erstellenden Fachwerttyps. Im Anschluß an die Definition des Fachwerttyps kann der Entwickler die Generierung des zugehörigen JWAM-Quellcodes veranlassen. Außerdem kann er eine zum Fachwerttyp gehörige Testklasse erstellen. Dieses Szenario wird durch das im folgenden vorgestellte Fachwertwerkzeug unterstützt.

Fehler! Keine gültige Verknüpfung.

Abbildung 7-1: Komponenten für die Erzeugung von JWAM-Fachwerttypen

Die Basis des Fachwertwerkzeugs bildet das entsprechend benannte *DomainValueTool*. Es verwaltet eine Liste von Fachwerttypbeschreibungen, welche unterschiedlicher Art sein können: einfach, komplex oder aufzählbar (vgl. Abschnitt 6.1). Jede dieser Arten wird durch ein Material repräsentiert und durch eine Unterklasse von *DomainValueDescription* implementiert. Diese Materialien werden als *SimpleDomainValue, ComplexDomainValue* und *EnumerableDomainValue* bezeichnet (vgl. Abbildung 7-1). Die in der Abbildung angedeutete Vererbungsbeziehung zwischen *SimpleDomainValue* und *ComplexDomainValue* erklärt sich daraus, daß ein einfacher Fachwerttyp eine spezielle Ausprägung eines komplexen Fachwerttyps darstellt: Ein einfacher Fachwerttyp hat im Unterschied zu komplexen Fachwerttypen stets nur ein einziges Attribut.
Aus einer Fachwerttypbeschreibung (*DomainValueDescription*) erzeugt der Codegenerator einen JWAM-Fachwerttyp, der der jeweiligen konkreten Unterklasse von *DomainValueDescription* entspricht. Dabei fließen auch die Informationen über das aktuelle Projekt mit ein, welche der Entwickler mit Hilfe der Komponente *ProjectOptionsEditor* angeben kann.

Auch für die Generierung der Testklassen wird der Codegenerator eingesetzt (vgl. Abbildung 7-2).

Fehler! Keine gültige Verknüpfung.

Abbildung 7-2: Komponenten für die Erzeugung von Testklassen

Der Entwickler kann entscheiden, ob nur eine Fachwertklasse oder auch die zugehörige Testklasse generiert werden soll. Für die Generierung der Testklassse verwendet der Codegenerator das Material *TestValueList*, welches eine Liste von Testwerten repräsentiert. Diese Liste kann mit Hilfe des *TestValueListEditors* bearbeitet werden (vgl. Abschnitt 7.2.5)

Die in den beiden obigen Abbildungen angeführten Werkzeugkomponenten des Fachwertwerkzeugs werden im folgenden kurz vorgestellt. Die genaue Beschreibung der Interaktion mit diesen Komponenten erfolgt in Abschnitt 7.2.

1. **DomainValueTool**
   Die Basis des Fachwertwerkzeugs bildet das entsprechend benannte *DomainValueTool*. Es ermöglicht die Erzeugung von Fachwerttypen der drei verschiedenen Arten und dient zur Verwaltung aller Fachwerttypen eines Projekts.

2. **SimpleDomainValueEditor**
   Dieser Editor dient zur Erzeugung und Bearbeitung einfacher Fachwerttypen, welche nur ein einziges Attribut aufweisen.

3. **ComplexDomainValueEditor**
   Ein komplexer Fachwerttyp kann mehrere Attribute aufweisen. Ein solcher Fachwerttyp kann nicht mit dem *SimpleDomainValueEditor* bearbeitet werden, statt dessen wird dafür der *ComplexDomainValueEditor* verwendet.

4. **EnumerableDomainValueEditor**
   Da sich die aufzählbaren Fachwerttypen sowohl von den einfachen als auch von den komplexen Fachwerttypen unterscheiden, werden sie mit Hilfe eines eigenen Subwerkzeugs angelegt: dem *EnumerableDomainValueEditor*.

5. **TestValueListEditor und TestValueEditor**
   Es ist im JWAM-Rahmenwerk üblich, das Verhalten neu erzeugter Fachwerttypen mit Hilfe von Testklassen zu überprüfen. Daher bietet das Fachwertwerkzeug auch die Erzeugung von Quellcode-Rahmen für diese Testklassen an. Die dafür notwendigen Testwerte kann der Entwickler mit Hilfe des Subwerkzeugs *TestValueListEditor* anlegen und mit dem Subwerkzeug *TestValueEditor* bearbeiten.

6. **CodeGenerator**
   Der Codegenerator kann sowohl zur Erzeugung der Fachwertklassen als auch für die Erzeugung der Testklassen eingesetzt werden. Dafür erzeugt der Codegenerator aus den Angaben, die in einem der Editoren gemacht wurden, das entsprechenden XML-Dokument. Dieses wird anschließend dazu benutzt, den zugehörigen Java-Quellcode zu erzeugen. Diese Schritte werden ausführlich in Abschnitt 7.3 dargestellt.

7.2 Umgang mit dem Fachwertwerkzeug

Wie bereits im vorhergehenden Abschnitt beschrieben, setzt sich das Fachwertwerkzeug aus mehreren Komponenten zusammen. Die Hauptkomponenten bilden das *DomainValueTool* zur Verwaltung aller Fachwerttypen eines Projekts und die drei Subwerkzeuge, welche zum Generieren der drei verschiedenen Arten von Fachwerttypen dienen. In den folgenden Abschnitten wird der Umgang mit den wichtigsten Komponenten des Fachwertwerkzeugs erläutert.

7.2.1 Interaktion mit dem DomainValueTool

Das *DomainValueTool* enthält eine Liste aller bereits definierten Fachwerttypen. In Abbildung 7-3 sind dies beispielsweise die Typen *dvUhrzeit* und *dvProzent* :

Fehler! Keine gültige Verknüpfung.

Abbildung 7-3: DomainValueTool

Die in der Liste enthaltenen Fachwerttypen werden als zu einem Projekt zugehörig betrachtet. Über die Projektoptionen (vgl. Abschnitt 6.4) hat der Entwickler die Möglichkeit, bestimmte Informationen für alle Fachwerttypen des aktuellen Projekts festzulegen. Solche Projekte können gespeichert und später erneut eingeladen werden, um neue Fachwerttypen hinzuzufügen oder um existierende Fachwerttypen zu bearbeiten und neu zu generieren.
Will der Entwickler einen der bereits in der Liste des *DomainValueTool* existierenden Fachwerttypen bearbeiten, so wird automatisch je nach Art des Fachwerttyps der entsprechende Editor geöffnet: der *SimpleDomainValueEditor* für einfache Fachwerttypen, der *ComplexDomainValueEditor* für komplexe Fachwerttypen und der *EnumerableDomainValueEditor* für aufzählbare Fachwerttypen.
Außer zum Bearbeiten können die Fachwerttypen in der Liste auch ausgewählt werden, um den entsprechenden JWAM-Quellcode und gegebenenfalls die zugehörige Testklasse zu generieren. Diese Aufgaben werden an den Codegenerator (vgl. Abschnitt 7.3) delegiert.
Will der Entwickler keinen der bereits existierenden Fachwerttypen bearbeiten, sondern einen neuen Fachwerttyp anlegen, so muß er zwischen den drei möglichen Arten von Fachwerttypen auswählen und öffnet damit den entsprechenden Editor. Das weitere Vorgehen bei der Fachwertentwicklung hängt dann von der Art des zu erstellenden Fachwerttyps ab und wird in den entsprechenden Abschnitten (vgl. Abschnitt 7.2.2 bis 7.2.4) beschrieben. Da es sich bei den einfachen Fachwerttypen um eine Spezialisierung der komplexen Fachwerttypen handelt, wird im folgenden Abschnitt zunächst das Werkzeug für den allgemeineren der beiden Fälle vorgestellt: das Werkzeug zum Erstellen komplexer Fachwerttypen.

7.2.2 Interaktion mit dem ComplexDomainValueEditor

Da komplexe Fachwerttypen mehrere Attribute enthalten können, besitzt der *ComplexDomainValueEditor* eine Liste zur Verwaltung dieser Attribute. Außerdem verfügt der *ComplexDomainValueEditor* über eine Liste mit speziellen Werten und maximal einem Standardwert sowie über eine Möglichkeit zur Angabe der Stringrep-

räsentation. Für das Beispiel des Fachwerttyps *dvUhrzeit* könnten die Angaben im geöffneten *ComplexDomainValueEditor* folgendermaßen aussehen:

Fehler! Keine gültige Verknüpfung.
Abbildung 7-4: ComplexDomainValueEditor

Die Oberflächenelemente zum Angeben der Stringrepräsentation passen sich beim *ComplexDomainValueEditor* dynamisch an die Anzahl der Attribute an und erfordern die Angabe von Trennzeichen zwischen diesen Attributen. Wird eine Stringrepräsentation angegeben, ohne Eintragungen für die Trennzeichen zwischen den Attributen vorzunehmen, so werden als Trennzeichen automatisch Leerzeichen verwendet. Eine Stringrepräsentation kann optional um ein Präfix und ein Suffix ergänzt werden.

Zum Anlegen eines Attributs muß der Benutzer zunächst nur dessen Namen und Typ angeben. Anschließend kann das Attribut in der Liste ausgewählt und bearbeitet werden, wobei weitere Einstellungen für das Attribut vorgenommen werden können. Im Beispiel *dvUhrzeit* könnte auf diese Weise die Eigenschaften für das Attribut *minuten* festgelegt werden:

Fehler! Keine gültige Verknüpfung.
Abbildung 7-5: AttributeEditor

Zu den Informationen, die im *AttributeEditor* eingetragen werden können, gehört zum einen die Angabe, ob bei der späteren Codegenerierung eine Methode zum Zugriff auf dieses Attribut angelegt werden soll. Zum anderen können Angaben über die Einschränkung des durch den Typ vorgegebenen Wertebereichs gemacht werden sowie Angaben zum Package des Attributtyps. Letzteres ist von Bedeutung, wenn es sich beim Attributtyp nicht um einen der vorgegebenen Java-Datentypen (*int*, *long*, *float*, *double*, *String*, *byte* oder *char*) handelt, sondern um einen benutzerdefinierten oder durch das JWAM bereitgestellten Fachwerttyp. Die Angabe des Packages für solche Attributtypen wird durch das Fachwertwerkzeug für ausgewählte Fachwerttypen unterstützt: Für diese wird bei Angabe des Typnamens das Package automatisch eingetragen. Die Liste der ausgewählten Fachwerttypen beinhaltet die im JWAM-Rahmenwerk definierten Fachwerttypen und kann von Entwickler um weitere Typen erweitert werden.
Die Angaben über Einschränkungen des Wertebereichs unterscheiden sich je nach Art der gewählten Wertebereichseinschränkung: der Benutzer kann entweder eine Aufzählung des Wertebereichs vornehmen, indem er die einzelnen Elemente des Wertebereichs explizit benennt und in eine entsprechende Liste einträgt; oder der Entwickler kann einschränkende Angaben machen. Wie in Abschnitt 6.2.2 dargestellt, sind nicht alle Wertebereichseinschränkungen auf alle Attributtypen anwendbar. Die für den gewählten Attributtyp nicht anwendbaren Einschränkungen werden dementsprechend auf der Benutzungsoberfläche deaktiviert. In Abbildung 7-5 sind daher nur die Felder für *Minimum* und *Maximum* aktiv.

7.2.3 Interaktion mit dem SimpleDomainValueEditor

Die Auswertung von bereits existierenden Fachwerttypen ergab, daß häufig Fachwerttypen erstellt werden, die nur ein einziges Attribut aufweisen. Da die Sonderbehandlung von einfachen Fachwerttypen einige Vorteile bei der Interaktion mit dem Fachwertwerkzeug bietet, werden einfache Fachwerttypen nicht mit dem gleichen Subwerkzeug wie die komplexen Fachwerttypen bearbeitet. Im folgenden werden die daraus resultierenden Vorteile erläutert.

Prinzipiell wäre es möglich gewesen, auf die separate Behandlung von einfachen Fachwerttypen zu verzichten, weil sie vollständig durch komplexe Fachwerttypen modellierbar sind. Um einen einfachen Fachwerttyp zu modellieren, wären dann jedoch drei Schritte notwendig: zum einen müßte das Attribut definiert werden. Zudem müßte in den meisten Fällen das Attribut anschließend bei der Stringrepräsentation ausgewählt und das Kästchen zum Generieren einer Zugriffsfunktion angehakt werden; denn einfache Fachwerttypen sind in der Regel aus einer Zeichenkette erzeugbar und besitzen eine Zugriffsfunktion auf das gekapselte Attribut. Diese Arbeitsschritte werden durch das Definieren eines einfachen Fachwerttyps automatisch vorgenommen: Das einzige Attribut des einfachen Fachwerttyps, standardmäßig *value* genannt, ist dabei schon vordefiniert und in der Stringrepräsentation bereits ausgewählt (vgl. Abbildung 7-6). Außerdem wird die Zugriffsfunktion auf das einzige Attribut für jeden einfachen Fachwerttyp automatisch generiert.

Fehler! Keine gültige Verknüpfung.
Abbildung 7-6: SimpleDomainValueEditor

Das Attribut *value* dient dazu, den Wert eines einfachen Fachwerts zu repräsentieren. Der Typ dieses Attributs entspricht dem Basistyp des Fachwerttyps. Dieser Basistyp wird vom Entwickler angegeben. Handelt es sich dabei um einen Klassentyp, dessen Klasse sich nicht im standardmäßig eingebundenen Package java.lang befindet, so muß für die korrekte Erzeugung des Fachwertcodes das Package des Basistyps angegeben werden. Diese Eintragung wird ebenso wie beim *AttributeEditor* für ausgewählte Fachwerttypen automatisch vorgenommen (vgl. Abschnitt 7.2.2).

Aufgrund der Annahme, daß die meisten der einfachen Fachwerttypen aus einer Zeichenkette erzeugbar sein sollen, ist das entsprechende Kästchen (*can create from string*) standardmäßig ausgewählt, ebenso wie das Attribut *value* in der Stringrepräsentation. Der Entwickler kann die Stringrepräsentation ferner um ein Präfix und ein Suffix ergänzen. In Abbildung 7-6 wird in der Stringrepräsentation des einfachen Fachwerttyps *dvProzent* beispielsweise das Suffix "%" festgelegt.

Der Wertebereich des Basistyps kann ähnlich wie die Wertebereiche der Attribute beim *ComplexDomainValueEditor* eingeschränkt werden. Die Möglichkeit, den Wertebereich durch eine Aufzählung einzuschränken ist hiervon ausgenommen, da dieser Fall durch aufzählbare Fachwerttypen abgedeckt wird.

Weitere Angaben, die für einen einfachen Fachwerttypen gemacht werden können, sind Informationen über spezielle Werte (vgl. Abschnitt 6.2.4). Dabei können bestimmten Werten, die das Attribut *value* annehmen kann, besondere Namen zuge-

wiesen werden. Auch ein Standardwert ist möglich. Zudem können Name und Package des Fachwerttyps angegeben werden.

7.2.4 Interaktion mit dem EnumerableDomainValueEditor

Zur Definition aufzählbarer, auf Zeichenketten basierender Fachwerttypen (vgl. Abschnitt 6.3) genügt ein Editor, welcher die Angabe der Elemente der Aufzählung und gegebenenfalls ihre Sortierung ermöglicht. Das Beispiel des aufzählbaren Fachwerttyps *dvEmployeePosition* mit den zwei Aufzählungselementen *Manager* und *Team member* könnte im *EnumerableDomainValueEditor* folgendermaßen angegeben werden:

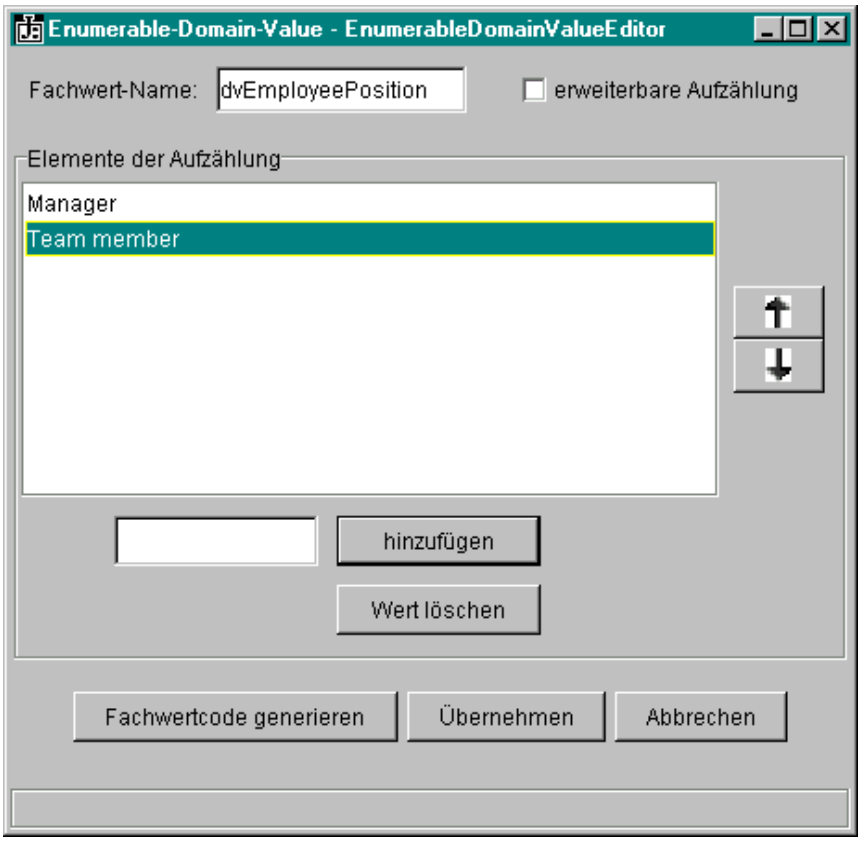

Abbildung 7-7: EnumerableDomainValueEditor

Die Angabe der Sortierung ist nur dann sinnvoll, wenn es sich um einen nicht dynamisch erweiterbaren Aufzählungs-Fachwerttyp handelt (vgl. Abschnitt 6.3.4). Der Benutzer entscheidet sich daher bei der Definition eines neuen aufzählbaren Fachwerttyps, ob der Wertebereich dynamisch erweiterbar sein soll oder nicht und kann nur dann mit eine Sortierung festlegen, wenn der Fachwerttyp als *nicht dynamisch erweiterbar* gekennzeichnet wird. Daher sind die Schaltflächen mit den Pfeilen, welche zur Angabe der Sortierung dienen, nur in diesem Fall aktiv.

7.2.5 Interaktion mit TestValueListEditor und TestValueEditor

Da es im JWAM-Rahmenwerk üblich ist, das Verhalten neu erzeugter Fachwerttypen mit Hilfe von Testklassen zu überprüfen, bietet das Fachwertwerkzeug auch die Erzeugung von Quellcode-Rahmen für diese Testklassen an.
Testklassen enthalten mehrere Tests, mit denen die Methoden, Konsistenzbedingungen und die Funktionalität der getesteten Klasse geprüft werden. Einige dieser Tests können vom Fachwertwerkzeug generiert werden (vgl. Abschnitt 6.5). Für diese Generierung werden Testwerte verwendet, welche der Entwickler vorgibt. Diese Testwerte werden vom *TestValueListEditor* in einer Liste verwaltet, welche das Löschen von Testwerten, das Hinzufügen neuer Testwerte und das Bearbeiten bereits existierender Testwerte ermöglicht. Für das Anlegen und Editieren von Testwerten wird der *TestValueEditor* aufgerufen. Ein Testwert für den Fachwerttyp *dvUhrzeit* könnte beispielsweise folgendermaßen aussehen:

Fehler! Keine gültige Verknüpfung.
Abbildung 7-8: TestValueEditor

In diesem Editor können die Werte für die einzelnen Attribute des zugehörigen Fachwerttyps sowie die Stringrepräsentation festgelegt werden. Für jeden Testwert muß zudem angegeben werden, ob er einen gültigen Fachwert des betreffenden Fachwerttyps darstellen soll.
Je nachdem, welche Angaben für einen Testwert gemacht werden, können verschiedene Tests generiert werden. Der wichtigste Test ist der Test zum Prüfen, ob eine gegebene Zeichenkette einen gültigen Fachwert darstellt. Zusätzlich können Werte für die einzelnen Attribute eingetragen werden. Dadurch können zum einen Zugriffsfunktionen getestet werden: Mittels der Angaben aus Abbildung 7-8 kann beispielsweise geprüft werden, ob das Attribut *stunden* des Fachwerts "*10:15 am*" tatsächlich *10* ist. Wenn für alle Attribute ein Wert angegeben ist, kann zusätzlich ein Test generiert werden, der überprüft, ob diese Attributwerte einen gültigen Fachwert repräsentieren.

7.2.6 Interaktion mit dem CodeGenerator

Der Codegenerator kann sowohl von dem *DomainValueTool* als auch von den drei untergeordneten Fachwerttyp-Editoren aufgerufen werden. Auf der Benutzungsoberfläche des Codegenerators kann angegeben werden, ob sowohl der ausgewählte Fachwerttyp als auch die zugehörige Testklasse erzeugt werden sollen oder ob nur eine dieser beiden Klassen zu generieren ist:

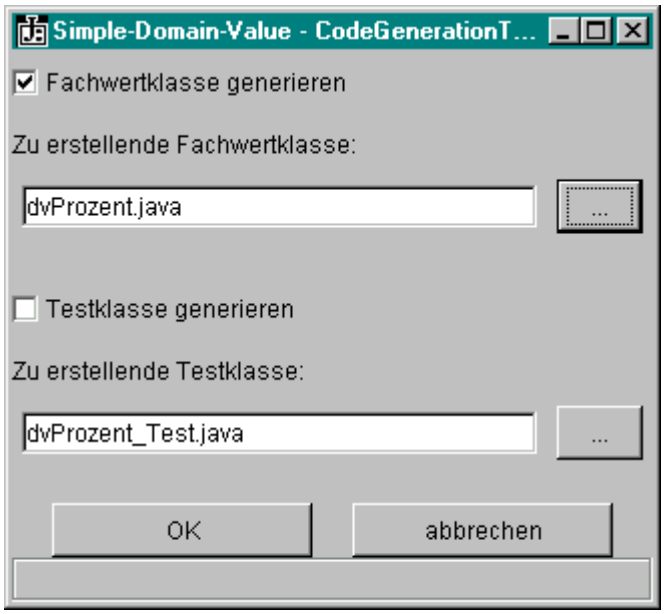

Abbildung 7-9: CodeGenerator

Da bei Java-Klassen der Klassenname stets mit dem Dateinamen übereinstimmen muß, sind die Namen der Fachwertklasse und der Testklasse bereits vorgegeben. Sie entsprechen dem Namen des betreffenden Fachwerttyps, kombiniert mit der Standardkennzeichnung von Javadateien; wobei die Testklasse mit Hilfe des Suffixes "_Test" von der Fachwertklasse unterschieden wird. Für den Fachwerttyp *dvUhrzeit* werden also die Dateien *dvUhrzeit.java* und gegebenenfalls *dvUhrzeit_Test.java* generiert. Der Speicherort dieser Dateien muß vom Entwickler selbst angegeben werden, wofür eine Schaltfläche für den Dateiauswahldialog zur Verfügung steht. Wird kein Speicherort angegeben, so wie in der obigen Abbildung dargestellt, dann werden die Dateien in dem Verzeichnis angelegt, aus welchem das Fachwertwerkzeug gestartet wurde.
Die Codegenerierung, die vom Codegenerator angestoßen wird, wird im folgenden Abschnitt dargestellt.

7.3 Beschreibung des Mechanismus zur Codegenerierung

Der in diesem Abschnitt vorgestellte Mechanismus zur Codegenerierung wird sowohl zur Generierung von Fachwertklassen, als auch zur Generierung von Testklassen eingesetzt. Im folgenden wird der Mechanismus am Beispiel der Fachwertklassengenerierung erläutert.
Es werden zunächst die Techniken vorgestellt, die das Fachwertwerkzeug für die Generierung des JWAM-Fachwertcodes verwendet. Dabei wird aufgezeigt, inwiefern sich diese Techniken für die gestellten Aufgaben eignen und wo Schwierigkeiten auftreten. Zudem wird die verwendete Codeschablone dargestellt sowie der Weg von der strukturellen Fachwerttypbeschreibung hin zur konkreten JWAM-Fachwertklasse.

Die Codegenerierung läßt sich in zwei Arbeitsschritte aufteilen, welche in den folgenden beiden Abschnitten erläutert werden: zunächst muß ein XML-Dokument zur Beschreibung eines Fachwerttyps erstellt werden, welches im nächsten Schritt zur Erzeugung von JWAM-Quellcode genutzt werden kann.

7.3.1 Die Erzeugung des XML-Dokuments

Der erste Schritt der Codegenerierung besteht darin, die fachwerttypspezifischen Informationen, die auf der Benutzungsoberfläche des Werkzeugs eingegeben wurden, in ein entsprechendes XML-Dokument umzusetzen (vgl. Abbildung 7-10).

Fehler! Keine gültige Verknüpfung.
Abbildung 7-10: Codegenerierung (Schritt 1)

Der Entwickler erstellt und bearbeitet mit dem Werkzeug ein Material, welches einen Fachwerttyp repräsentiert. Dieses Material wird durch eine der drei Unterklassen vom Typ *DomainValueDescription* implementiert: *SimpleDomainValue*, *ComplexDomainValue* oder *EnumerableDomainValue* (vgl. Abbildung 7-1).
Die Funktionalität zum Erzeugen der XML-Fachwerttypbeschreibung wird von der Klasse *XMLBuilder* bereitgestellt, welche ein Teil des Codegenerators ist. Dieser *XMLBuilder* generiert aus den Daten eines Materials vom Typ *DomainValueDescription* ein XML-Dokument.

Zum Generieren eines einfachen Fachwerttyps *dvProzent* könnte der Entwickler beispielsweise folgende Informationen auf der Benutzungsoberfläche des Werkzeugs angeben (vgl. Abbildung 7-6):

- Name: *dvProzent*
- Basistyp: *int*
- Einschränkungen: Minimum: *0,* Maximum: *100*
- Stringrepräsentation: *value %* (*value* bezeichnet dabei das Attribut)
- Name des speziellen Werts: *total*
- Attributwert zur Erzeugung des speziellen Werts: *100*
- Attributwert zur Erzeugung des Standardwerts: *0*

Der *XMLBuilder* würde aus diesen Informationen das folgende XML-Dokument generieren:

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

  <ERWEITERUNGEN>
    <PACKAGENAME>domainvalue</PACKAGENAME>
    <AUTOR>Annika Spreckelmeyer, Martin Fuerter</AUTOR>
    <COPYRIGHT>AS&MF</COPYRIGHT>
    <VERSION>1.1</VERSION>
  </ERWEITERUNGEN>
```

```
<FACHWERTTYP>

  <NAME>dvProzent</NAME>

  <ATTRIBUT zugriff="ja">
    <ATTRIBUT_NAME>value</ATTRIBUT_NAME>
    <ATTRIBUT_TYP>int</ATTRIBUT_TYP>
    <ATTRIBUT_EINSCHRAENKUNGEN>
      <MAXIMUM inclusive="true">100</MAXIMUM>
      <MINIMUM inclusive="true">0</MINIMUM>
    </ATTRIBUT_EINSCHRAENKUNGEN>
  </ATTRIBUT>

  <AUS_STRING_ERZEUGBAR>true</ AUS_STRING_ERZEUGBAR>

  <STRINGREPRESENTATION>
    <ATTRIBUT_NAME>value</ATTRIBUT_NAME>
    <TRENNZEICHEN>%</TRENNZEICHEN>
  </STRINGREPRESENTATION>
  <SPEZIELLE_WERTE>
    <SPEZIELLER_WERT>
      <SPEZIELLER_WERT_NAME>total</SPEZIELLER_WERT_NAME>
      <ATTRIBUT_WERT_PAAR>
        <ATTRIBUT_NAME>value</ATTRIBUT_NAME>
        <ATTRIBUT_WERT>100</ATTRIBUT_WERT>
      </ATTRIBUT_WERT_PAAR>
    </SPEZIELLER_WERT>
  </SPEZIELLE_WERTE>

  <STANDARDWERT>
    <ATTRIBUT_WERT_PAAR>
      <ATTRIBUT_NAME>value</ATTRIBUT_NAME>
      <ATTRIBUT_WERT>0</ATTRIBUT_WERT>
    </ATTRIBUT_WERT_PAAR>
  </STANDARDWERT>

</FACHWERTTYP>
</ERWEITERTER_FACHWERTTYP>
```

Solche XML-Dokumente werden im zweiten Schritt der Codegenerierung in entsprechende JWAM-Fachwertklassen umgesetzt. Dies wird im folgenden Abschnitt näher erläutert.

7.3.2 Verabeitung von XML-Dokumenten mit Hilfe von XSLT

Zur Verarbeitung von XML-Dokumenten kann der Standard XSLT (*eXtensible Stylesheet Language: Transformations*) verwendet werden. Dies ist ein vom *World Wide Web Consortium* (W3C) veröffentlichter Standard, welcher für die Transformation von XML-Dokumenten in andere Dokumente verwendet wird (vgl. [KAY00]). Die Zieldokumente können beispielsweise HTML- oder XML-Dokumente sein oder andere textbasierte Formate aufweisen. Im Verlauf der Transformation wird zunächst mit Hilfe eines Parsers eine Baumstruktur aus dem XML-Dokument erzeugt. Dabei wird jeder einzelne Bestandteil des XML-Dokuments, also beispielsweise Elemente und Attribute, in einen Knoten der Baumstruktur umgesetzt. Anschließend kann mit

Hilfe von XSLT innerhalb dieser Baumstruktur navigiert werden, um einzelne Knoten auszuwählen und zu manipulieren. Das ursprüngliche XML-Dokument bleibt dabei unangetastet. Die Navigationsanweisungen werden in einem XSLT-Stylesheet definiert, welches wiederum eine Baumstruktur aufweist. Der XSLT-Prozessor verwendet dieses XSLT-Stylesheet, um aus der Baumstruktur des XML-Dokuments das Zieldokument zu erzeugen – und auch dieses Zieldokument hat ebenso wie die anderen Dokumente eine Baumstruktur.
Dieser XSLT-Mechanismus wird im Fachwertwerkzeug zur Erzeugung des JWAM-Codes genutzt: In diesem zweiten Schritt der Codegenerierung wird ein XSLT-Prozessor eingesetzt, welcher mit Hilfe eines XML-Stylesheet aus dem XML-Dokument den JWAM-Code erzeugt (vgl. Abbildung 7-11).

Fehler! Keine gültige Verknüpfung.
Abbildung 7-11: Codegenerierung (Schritt 2)

Im folgenden wird der Aufbau und die Funktionsweise des XSLT-Stylesheet näher erläutert.

7.3.3 Die Codeschablonen: Grundgerüste für Fachwerttypen und Testklassen

Im Fachwertwerkzeug werden XSLT-Stylesheets auch als Codeschablonen bezeichnet, da sie als Vorlage für den zu generierenden JWAM-Code dienen. Das Fachwertwerkzeug verwendet drei verschiedene Codeschablonen: eine für die Erzeugung von Testklassen (*testClass.xsl*), eine für die Erzeugung von aufzählbaren Fachwerttypen (*enumValue.xsl*) und eine weitere für die Generierung von komplexen und einfachen Fachwerttypen (*domainValue.xsl*). Die einfachen und komplexen Fachwerttypen können aufgrund ihrer Ähnlichkeiten durch eine gemeinsame Codeschablone erzeugt werden (vgl. Abschnitt 5.1).
Die Codeschablonen des Fachwertwerkzeugs wurden jeweils aus einer entsprechenden JWAM-Fachwertklasse entwickelt. Beispielsweise diente eine komplexe Fachwertklasse als Grundlage für die Codeschablone *domainValue.xsl*. Diese Codeschablone wird im weiteren Verlauf dieses Abschnitts beispielhaft vorgestellt und erläutert.
Der Quellcode eines JWAM-Fachwerttyps enthält zum einen statische Anteile, die bei allen JWAM-Fachwerttypen gleich sind, und zum anderen variable Anteile, die speziell für diesen Fachwerttyp gelten. Die Codeschablone zum Erzeugen eines JWAM-Fachwerttyps weist dementsprechend statische und variable Anteile auf. Jene Code-Anteile, welche allen Fachwerttypen gemeinsam sind, bilden die statischen Anteile der Codeschablone. Darin eingebettet finden sich XSLT-Anweisungen, welche zur Generierung der variablen Codeabschnitte eingesetzt werden. In einer Codeschablone wird also JWAM-Quellcode mit XSLT-Anweisungen kombiniert. Zum besseren Verständnis werden diese XSLT-Anweisungen in den späteren Code-Beispielen grau hinterlegt.
Um anstelle der XSLT-Anweisungen die variablen Codeanteile einfügen zu können, müssen die dafür notwendigen Informationen mit Hilfe des XSLT-Prozessors aus der XML-Fachwerttypbeschreibung extrahiert werden können. Für diese Navigation

innerhalb der XML-Fachwerttypbeschreibung werden folgenden Mechanismen benötigt:

1. **Auswahl einzelner Knoten und Auslesen deren Inhalts**
   Dies dient zum einfachen Einfügen von Informationen aus der XML-Fachwerttypbeschreibung in den Quellcode. Dazu gehören beispielsweise die Namen des Fachwerttyps und der Attribute.

2. **Abfragen nach Existenz oder Inhalt eines Knotens**
   Dies ist insbesondere wichtig, um unterschiedlichen Code generieren zu können für
   a) unterschiedliche Attributtypen
   b) unterschiedliche Einschränkungen dieser Typen.

3. **Schleifen zum wiederholten Erzeugen von stark ähnlichen Codepassagen**
   Dies wird beispielsweise zum Generieren von Zugriffsfunktionen für die Attribute benötigt, denn diese Zugriffsfunktionen unterscheiden sich nur in Attributname und Attributtyp.

Um zu verdeutlichen, wie diese Mechanismen in XSLT angeboten werden und wie sie zum Erzeugen von JWAM-Fachwerttypen eingesetzt werden können, werden im folgenden einige Ausschnitte der Codeschablone *domainValue.xsl* vorgestellt und erläutert. In diesen Beispielen sind die jeweils zu erläuternden XSLT-Anweisungen durch Fettdruck gekennzeichnet. Im Anschluß an diese Ausschnitte der Codeschablone wird jeweils verdeutlicht, wie der JWAM-Code aussieht, der mit ihrer Hilfe erzeugt werden kann. Die Grundlage für die folgenden Abschnitte bildet das Beispiel des einfachen Fachwerttyps *dvProzent*, dessen XML-Fachwerttypbeschreibung in Abschnitt 7.3.1 zu finden ist.

7.3.3.1 Schleifen und Auswahl von Knoten

Zur Deklaration der Attribute werden in der Codeschablone *domainValue.xsl* folgende Zeilen verwendet:

```
//
// private attributes
//
<xsl:for-each select="ATTRIBUT">
private <xsl:value-of select="ATTRIBUT_TYP"/> _<xsl:value-of
select="ATTRIBUT_NAME"/>;
</xsl:for-each>
```

Die Attribute des Fachwerts sind in der XML-Fachwerttypbeschreibung mit Hilfe der Kennzeichnung `<ATTRIBUT>` versehen. Für jedes dieser *Attribut*-Elemente, welche im XML-Dokument zu finden sind, werden die Anweisungen zwischen `<xsl:for-each select = "ATTRIBUT">` und `</xsl:for-each>` ausgeführt. Innerhalb des Blocks `<xsl:for-each ..>` wird auf den Knoten *ATTRIBUT_TYP* zuge-

griffen und dessen Inhalt anstelle der Anweisung `<xsl:value-of select = "ATTRIBUT_TYP"/>` eingefügt.
Der resultierende JWAM-Code für das Beispiel des einfachen Fachwerttyps *dvProzent* aus Abschnitt 7.3.1 würde demnach folgendermaßen aussehen:

```
//
// private attributes
//

private int _value;
```

Da der Fachwerttyp *dvProzent* nur ein einziges Attribut aufweist, wird die Schleife zur Attributerzeugung in diesem Beispiel nur einmal durchlaufen.

7.3.3.2 Fallunterscheidungen

Inhalte von Knoten können als Testbedingungen bei Fallunterscheidungen eingesetzt werden. Eine solche Fallunterscheidung findet sich in der Codeschablone *domainValue.xsl* in dem Abschnitt für die Generierung der Methode *valueImpl*, welche zur Erzeugung eines Fachwerts aus einer Zeichenkette dient:

```
[...]
<xsl:choose>
   <xsl:when test="ATTRIBUT_TYP='int'">
   <xsl:value-of
   select="ATTRIBUT_NAME"/> = Integer.valueOf(<xsl:value-of
   select="$parse-variable"/>).intValue();</xsl:when>

   <xsl:when test="ATTRIBUT_TYP='long'">
   <xsl:value-of
   select="ATTRIBUT_NAME"/> = Long.valueOf(<xsl:value-of
   select="$parse-variable"/>).longValue();</xsl:when>

   <xsl:when test="ATTRIBUT_TYP='float'">
   <xsl:value-of
   select="ATTRIBUT_NAME"/> = _decimalFormat_<xsl:value-of
   select="ATTRIBUT_NAME"/>.parse(<xsl:value-of
   select="$parse-variable"/>).floatValue();</xsl:when>
   [...]
</xsl:choose>
[...]
```

Ein Block, der mit *<xsl:choose>* eingeleitet wird, beinhaltet eine Fallunterscheidung. Das obige Beispiel eines solchen Blocks bezieht sich auf das einzige Attribut des Fachwerttyps *dvProzent*: je nach Typ dieses Attributs wird vom XSLT-Prozessor unterschiedlicher Code in den statischen Anteil der Codeschablone eingefügt. Die Information über den Typ des Attributs erhält der XSLT-Prozessor aus der XML-Fachwerttypbeschreibung, wo diese Information im Element `<ATTRIBUT_TYP>` zu finden ist. Da es sich bei dem einzelnen Attribut von *dvProzent* um ein Attribut vom Typ *int* handelt, wird aus dem obigen Codeschablonen-Ausschnitt die folgende Zeile erzeugt:

```
value = Integer.valueOf(parsed_value).intValue();
```

Im obigen Beispiel dient die Fallunterscheidung dazu, je nach Inhalt eines bestimmten Elements verschiedenen Code zu generieren. Darüber hinaus kann mittels der Fallunterscheidung bei XSLT herausgefunden werden, ob ein gegebenes Element in einer XML-Fachwerttypbeschreibung definiert ist, woraufhin für dieses Element der entsprechende Quellcode generiert werden kann.
Dieser Mechanismus wird in der Codeschablone *domainValue.xsl* im Abschnitt für die Generierung der Methode *isAttributeSettingValid* verwendet. Diese Methode dient dazu festzustellen, ob aus den übergebenen Parametern ein gültiger Fachwert erstellt werden könnte. Da die Attribute des Fachwerts mit diesen Parametern initialisiert werden, müssen die Parameter die angegebenen Einschränkungen für die Attribute einhalten.

```
/**
 * do the given parameters represent a valid
 * domainvalue <xsl:value-of select="NAME"/>?
 */
public boolean isAttributeSettingValid(<xsl:for-each
   select="ATTRIBUT">
   <xsl:value-of
      select="ATTRIBUT_TYP"/><xsl:text>&#x20;</xsl:text>
   <xsl:value-of select="ATTRIBUT_NAME"/>
   <xsl:if test="position()!=last()">,</xsl:if> </xsl:for-each>)
{  <xsl:for-each select="ATTRIBUT">
   <xsl:for-each select="ATTRIBUT_EINSCHRAENKUNGEN/*">

   <xsl:variable name="attribut-name"
   select="ancestor::ATTRIBUT/ATTRIBUT_NAME"/>

   <xsl:choose>
   <xsl:when test="name()='MINIMUM'">
   if(!(<xsl:value-of select="$attribut-name"/>&gt;<xsl:if
      test="@inclusive='true'">=</xsl:if> <xsl:value-of
      select="."/>))
      return false;
   </xsl:when>
   <xsl:when test="name()='MAXIMUM'">
   if(!(<xsl:value-of select="$attribut-name"/>&lt;<xsl:if
      test="@inclusive='true'">=</xsl:if><xsl:value-of
      select="."/>))
      return false;
   </xsl:when>
   <xsl:when test="name()='LAENGE'">
   if(!(<xsl:value-of
      select="$attribut-name"/>.length() == <xsl:value-of
      select="."/>))
      return false;
   </xsl:when>
      [...]
   </xsl:choose>
   </xsl:for-each><!-- ATTRIBUT_EINSCHRAENKUNGEN -->
   </xsl:for-each><!-- ATTRIBUT -->
   return true;
}
```

Um für alle Einschränkungen den geeigneten Code erzeugen zu können, wird der Mechanismus der Fallunterscheidung mit einer Schleife kombiniert, die über alle definierten Einschränkungen iteriert.
In dem oben angeführten Beispiel werden die Subelemente des Elements `<ATTRIBUT_EINSCHRAENKUNGEN>` nacheinander abgearbeitet und es wird jeweils geprüft, um welche Art Einschränkung es sich handelt. Je nach Name des abgearbeiteten Elements wird dann der entsprechende Quelltextabschnitt des Fachwerttyps generiert.
Im Beispiel des Fachwerttyps *dvProzent* existieren zwei Einschränkungen: der Wertebereich des Attributs *value* soll ein Minimum von *0* und ein Maximum von *100* haben. Der resultierende JWAM-Code würde daher folgendermaßen aussehen:

```
/**
 * do the given parameters represent a valid domainvalue
 * dvProzent?
 */
public boolean isAttributeSettingValid(int value)
{
   if(!(value<=100))
      return false;

   if(!(value>=0))
      return false;

   return true;
}
```

Die in diesem Abschnitt dargestellten Techniken eignen sich gut, um Quellcode zu generieren, der nicht zu stark variabel ist. Beispielsweise können ähnliche Codepassagen wiederholt für verschiedene Parameter generiert werden. Dieses Schema wird auch auf Codepassagen angewendet, die optimaler implementiert werden könnten, wenn sie manuell geschrieben würden. So werden beispielsweise beim Parsen der Attribute einer Stringrepräsentation die Trennzeichen stets nacheinander abgearbeitet, auch wenn alle diese Trennzeichen gleich sind und daher eventuell eine Schleife eingesetzt werden könnte. Der vom Fachwertwerkzeug generierte Quellcode muß also bei Bedarf per Hand optimiert werden.
Gänzlich versagen muß der Codeerzeugungsmechanismus bei der Generierung von speziellem Quellcode. Beispielsweise ist es nicht möglich, fachliche Methoden zu generieren.
Neben diesen Grenzen, die der Codegenerierung innewohnen, hat sich die Benutzung von XSLT-Stylesheets als Codeschablonen teilweise als problematisch erwiesen; denn Änderungen an den XSLT-Stylesheets können sich schwierig gestalten. Solche Änderungen könnten an den Codeschablonen notwendig werden, wenn sich bei einer neuen Version des JWAM-Rahmenwerks die Implementierung von Fachwerttypen verändert. Auf diese Problematik der Änderbarkeit wird im folgenden Abschnitt näher eingegangen.

7.3.4 Schwierigkeiten im Umgang mit XSLT-Stylesheets

In dem im Rahmen dieser Diplomarbeit entwickelten Fachwertwerkzeug werden XML-Dokumente zur Beschreibung von Fachwerttypen und damit zum Speichern von Informationen eingesetzt. Die Sprache XSLT ist dazu ausgelegt, ein solches XML-Dokument in ein anderes XML-Dokument zu transformieren (vgl. [KAY00]). Allerdings kann XSLT auch dazu eingesetzt werden, beliebige andere Textformate zu erzeugen, wobei sich XML-ähnliche Formate wie HTML anbieten. Diese Fähigkeit, XML-Dokumente in eine andere Repräsentation umzuwandeln wird im Rahmen des Fachwertwerkzeuges genutzt, um Quellcode für Fachwerttypen zu generieren. Dazu werden die im XML-Dokument enthaltenen Informationen extrahiert und in eine Codeschablone eingefügt (vgl. Abschnitt 7.3.3). Diese Codeschablone enthält große Teile statischen Quelltextes, welcher zur besseren Lesbarkeit und Editierbarkeit bestimmten Formatierungsrichtlinien genügen muß. Damit im Zieldokument alle Zeilenumbrüche und -einschübe korrekt sind, müssen diese im XSLT-Stylesheet beachtet werden. Da dies jedoch oftmals nicht vereinbar ist mit einer sinnvollen Formatierung des XSLT-Stylesheet selbst, muß eine schlecht formatierte und dadurch unübersichtliche Codeschablone in Kauf genommen werden. Anpassungen und Korrekturen der Codeschablone werden dadurch stark erschwert, was an folgendem Ausschnitt der Codeschablone *domainValue.xsl* illustriert werden soll:

```
/**
 * the string-representation
 * @require isDefined()
 * @ensure result != null
 */
protected String toStringImpl ()
{
   Contract.require(isDefined(), this,
      "this domain value is defined");

   <xsl:if
   test="AUS_STRING_ERZEUGBAR='true'">return ""<xsl:for-each
    select="STRINGREPRESENTATION/*"> + <xsl:if test =
     "name()='ATTRIBUT'"><xsl:if test="key('float-or-double',
         ATTRIBUT_NAME)">Factory._decimalFormat_<xsl:value-of
select="ATTRIBUT_NAME"/>.precisionFormat(</xsl:if>_<xsl:value-of
select="ATTRIBUT_NAME"/>
      <xsl:if
      test="key('float-or-double', ATTRIBUT_NAME)">)</xsl:if>
    <xsl:if test = "name()='TRENNZEICHEN'">"<xsl:value-of
      select="."/>"</xsl:if>
    </xsl:for-each>;</xsl:if>
    <xsl:if test="AUS_STRING_ERZEUGBAR='false'">return <xsl:for-each
      select="ATTRIBUT"> "<xsl:value-of
      select="ATTRIBUT_NAME"/>" + ": " + _<xsl:value-of
      select="ATTRIBUT_NAME"/>
      <xsl:if test="position()!=last()"> + ", " + </xsl:if>
     </xsl:for-each>;</xsl:if>
  }
```

Die Übersichtlichkeit des Stylesheets wird zum einen dadurch beschränkt, daß einige Code-Anteile optional sind. Im obigen Stylesheet-Ausschnitt werden bestimmte Stellen beispielsweise nur dann in den Code eingefügt, wenn der Fachwert aus einer Zeichenkette erzeugbar ist.
Zur Erzeugung einer einzigen Zeile Code können unter Umständen größere Blöcke von mehreren Fallunterscheidungen notwendig sein. Dies führt häufig dazu, daß die Struktur des zu erzeugenden JWAM-Codes in der Codeschablone nicht mehr deutlich zu erkennen ist. So läßt sich anhand der Codeschablone oftmals nur schwer abschätzen, wie der erzeugte JWAM-Code aussehen wird. Beispielsweise ist in dem obigen Ausschnitt der Codeschablone nicht zu erkennen, daß die umfangreichen XSLT-Anweisungen für das Beispiel des einfachen Fachwerts *dvProzent* folgenden kurzen JWAM-Code erzeugen würde:

```
/**
 * the string-representation
 * @require isDefined()
 * @ensure result != null
 */
protected String toStringImpl ()
{
  Contract.require(isDefined(), this,
    "this domain value is defined");

  return "" + _value + "%";
}
```

An dem ausgewählten Ausschnitt der Codeschablone fällt zudem auf, daß die Verschachtelung der XSLT-Anweisungen nicht erkennbar ist. Beispielsweise können *if*-Anweisungen nicht so formatiert werden, daß die eingeschlossenen Blöcke sofort erkennbar sind. Dies hängt damit zusammen, daß bei der Formatierung die Textzwischenräume besonders beachtet werden müssen. In einem XSLT-Stylesheet gilt: Steht zwischen zwei XSLT-Anweisungen (`<xsl: ...>`) kein sichtbares Zeichen, so werden auch keine unsichtbaren Zeichen wie *Carriage Return* oder Leerzeichen berücksichtigt. Sobald jedoch mindestens ein sichtbares Zeichen zwischen zwei Anweisungen steht, werden sämtliche Textzwischenräume beachtet. Dies wirkt sich auf die Formatierungsmöglichkeiten des XSLT-Stylesheets aus. Soll beispielsweise durch einen größeren Block von XSLT-Anweisungen eine einzelne Codezeile generiert werden, so dürfen die Zeilen in diesem XSLT-Block nur an ganz bestimmten Stellen getrennt werden: entweder zwischen zwei direkt aufeinanderfolgenden XSLT-Anweisungen oder innerhalb einer solchen Anweisung. Erschwerend kann sich dabei auswirken, daß XSLT-Anweisungen relativ lang sein können und teilweise mehr als eine halbe Zeile in Anspruch nehmen. Es ist nur sehr eingeschränkt möglich, gleichzeitig auch noch auf die Schachtelung der XSLT-Anweisungen Rücksicht zu nehmen. Das XSLT-Stylesheet kann also nicht beliebig formatiert werden.

Anhand der folgenden Zeilen soll verdeutlicht werden, wieviel lesbarer die Codeschablone wäre, wenn die Hierarchie der XSLT-Anweisungen besser sichtbar gemacht werden könnte:

```
/**
 * the string-representation
 * @require isDefined()
 * @ensure result != null
 */
protected String toStringImpl ()
{
   Contract.require(isDefined(), this,
      "this domain value is defined");

   <xsl:if test="AUS_STRING_ERZEUGBAR='true'">return ""
      <xsl:for-each select="STRINGREPRESENTATION/*"> +
         <xsl:if test = "name()='ATTRIBUT'">
            <xsl:if test="key('float-or-double', ATTRIBUT_NAME)">
               Factory._decimalFormat_<xsl:value-of
               select="ATTRIBUT_NAME"/>.precisionFormat(
            </xsl:if>
            _<xsl:value-of select="ATTRIBUT_NAME"/>
            <xsl:if test="key('float-or-double', ATTRIBUT_NAME)">
               )
            </xsl:if>
         </xsl:if>
         <xsl:if test = "name()='TRENNZEICHEN'">
            "<xsl:value-of select="."/>"
         </xsl:if>
      </xsl:for-each>
      ;
   </xsl:if>
   <xsl:if test="AUS_STRING_ERZEUGBAR='false'">
      return
      <xsl:for-each select="ATTRIBUT">
         "<xsl:value-of select="ATTRIBUT_NAME"/>" + ": " +
         _<xsl:value-of select="ATTRIBUT_NAME"/>
         <xsl:if test="position()!=last()">
            + ", " +
         </xsl:if>
      </xsl:for-each>
      ;
   </xsl:if>
 }
```

Würde die obige Codeschablone tatsächlich zur Generierung von Code eingesetzt, so würde der generierte Code eine schlecht lesbare Formatierung aufweisen. Dies würde jedoch die manuelle Weiterbearbeitung des generierten Codes erschweren, welche bei vielen Fachwerttypen notwendig sein kann. Ein denkbarer Lösungsansatz für dieses Problem wäre es, einen Mechanismus einzusetzen, welcher den generierten Code in eine lesbare Form transformiert.
Allerdings ist zu beachten, daß auch eine bestmöglichst formatierte Codeschablone noch nicht wirklich gut lesbar ist. Der Grund dafür liegt, wie bereits erläutert, in dem großen Umfang der XSLT-Anweisungen.

7.4 Zusammenfassung

In diesem Kapitel wurde ein System entwickelt, welches die in Kapitel 6 entwickelten XML-Fachwerttypbeschreibungen automatisch generieren und in JWAM-Quellcode umsetzen kann. Als Ansatz für die Verarbeitung der XML-Beschreibungen wurde der Einsatz von Codeschablonen gewählt, in denen XSLT-Anweisungen mit JWAM-Quellcode kombiniert werden. Es wurde verdeutlicht, welche Schwierigkeiten im Umgang mit den teilweise sehr unübersichtlichen Codeschablonen auftreten und wie dieses Problem entschärft werden könnte.

8 Auswertung

Im Verlauf dieser Arbeit wurde herausgearbeitet, welche Konzepte im Zusammenhang mit Werttypen wünschenswert sind und daß sie in Programmiersprachen nicht adäquat unterstützt werden. In diesem Kapitel soll herausgearbeitet werden, inwieweit das Fachwertwerkzeug diese für Werttypen wünschenswerten Konzepte (vgl. Abschnitt 5.1) umsetzt.

Keine besondere Unterstützung bietet das Fachwertwerkzeug für das Konzept *einfache Konstruktoren / Pattern Matching*.
Allerdings bietet das Fachwertwerkzeug in bezug auf diese Konzepte zumindest eine leichte Verbesserung gegenüber dem JWAM-Rahmenwerk, indem zwei alternative Konzepte unterstützt werden (vgl. Abschnitt 5.1.9): das Konzept der einfachen Konstruktoren wird beim Fachwertwerkzeug durch die automatische Generierung von Konstruktoren ersetzt. Der Zugriff auf Attribute von komplexen Fachwerttypen wird nicht durch Pattern Matching ermöglicht, sondern dadurch, daß für die einzelnen Attribute Zugriffsmethoden generiert werden können.

Gute Unterstützung bietet das Fachwertwerkzeug für die Konzepte *Angabe der Stringrepräsentation* und *Sicherung der Unveränderlichkeit*:
Die Angabe einer einfachen Stringrepräsentation wird durch entsprechende Eingabefelder auf der Werkzeugoberfläche unterstützt, so daß einfache Stringrepräsentationen vom Entwickler nicht mehr selbst implementiert werden müssen. Ein Vorteil des Fachwertwerkzeugs ist, daß daraus sowohl die Funktionalität für die Ausgabe eines Werts, als auch die Funktionalität für das Einlesen eines Fachwerts erzeugt werden kann. Dies spiegelt den Zusammenhang wider, der zwischen diesen beiden Funktionalitäten besteht (vgl. Abschnitt 5.1.5). Komplexe Stringrepräsentationen allerdings, welche nicht nur durch Trennzeichen abgetrennte Felder enthalten, erfordern noch immer die manuelle Erstellung durch den Entwickler.
Die Sicherung der für Werttypen notwendigen Unveränderlichkeit wird dadurch verbessert, daß die vom Werkzeug automatisch generierten Fachwerttypen garantiert keine verändernden Operationen enthalten. Bei einfach aufgebauten Fachwerttypen, die vollständig vom Fachwertwerkzeug generiert werden können, wird dadurch die Unveränderlichkeit gesichert. Allerdings können die automatisch generierten JWAM-Fachwertklassen prinzipiell nach ihrer Generierung durch den Entwickler manuell verändert werden. Dies könnte theoretisch dazu führen, daß diese Typen durch Hinzufügen verändernder Methoden nicht mehr die Eigenschaft der Unveränderlichkeit aufweisen. Das Fachwertwerkzeug kann daher nur für die automatisch generierten Fachwerttypen die Unveränderlichkeit garantieren, jedoch nicht für manuell weiterentwickelte Fachwerttypen.

Sehr gute Unterstützung bietet das Fachwertwerkzeug für die Konzepte *Aufzählungstypen*, *Einschränkungen des Wertebereichs*, *einfache Entwicklung von Werttypen*, *undefinierter Wert* und *Erstellung neuer Typen / Kennzeichung von Werttypen*:

Dabei ist anzumerken, daß die Konzepte, die vom JWAM-Rahmenwerk bereits sehr gut berücksichtigt werden, auch bei Verwendung des Fachwertwerkzeugs unterstützt werden. Dazu zählen der *undefinierte Wert* und die *Erstellung neuer Typen* inklusive der Möglichkeit, Werttypen als solche kennzeichnen zu können. Der Grund dafür ist, daß das Fachwertwerkzeug reine JWAM-Fachwerttypen erzeugt, so daß die im JWAM-Rahmenwerk umgesetzten Konzepte in diesen Fachwerttypen unverändert enthalten sind.
Die Erstellung von Aufzählungstypen wird dadurch erleichtert, daß es zu deren Spezifizierung ausreicht, die einzelnen Aufzählungswerte anzugeben. Es ist nicht mehr nötig, Funktionalitäten wie Sortierung der Elemente oder Zugriffsfunktionen für die einzelnen Aufzählungselemente manuell zu implementieren.
Gängige Einschränkungen des Wertebereichs werden auf der Oberfläche unterstützt, so daß die Erstellung der entsprechenden Quellcodeanteile dem Entwickler abgenommen wird. Auch hierbei gilt allerdings: komplexere Wertebereichseinschränkungen müssen nach wie vor manuell vorgenommen werden. Es können beispielsweise keine Wertebereichseinschränkungen generiert werden, welche über die unterstützten Standardfälle, wie etwa das Angeben eines Minimums, hinausgehen. Soll der Wertebereich eines Fachwerttyps beispielsweise nur ungerade Zahlen aufweisen, so muß der Entwickler diese Einschränkungen manuell in den generierten Code eintragen.
Die einfache Entwicklung von Werttypen wird insbesondere dadurch unterstützt, daß die manuelle Implementierung von Fachwerttypen teilweise ganz entfällt, neue Notationen vermieden und unerfahrene Entwickler unterstützt werden. So können zum einen fehlerhafte oder unvollständige Eingaben abgefangen werden, so daß der generierte JWAM-Code keine fehlerhaften Anteile enthält, und zum anderen können Techniken eingesetzt werden wie beispielsweise das Ausgrauen von Feldern, die in bestimmten Situationen nicht sinnvoll sind. Nicht sinnvoll ist etwa die Kombination der Einschränkungen *Minimum* und *Maximum* mit dem Datentyp *String*, denn diese Kombination ist in XML-Schema nicht zulässig. Das Fachwertwerkzeug bietet die Möglichkeit, diese Information durch Ausgrauen der Felder *Maximum* und *Minimum* für den Entwickler sichtbar zu machen. Generell läßt sich sagen, daß ein Werkzeug einen unerfahrenen Entwickler recht gut unterstützen kann, weil es seine Aufmerksamkeit auf die wesentlichen Aspekte der Fachwerttyp-Entwicklung richten kann. So könnte es beispielsweise sein, daß ein unerfahrener Entwickler sich ohne die Hilfestellungen des Werkzeugs nicht bewußt wäre, welche Angaben für die Konstruktion eines Fachwerttyps notwendig oder möglich sind, wie etwa die Angabe des Wertebereichs oder das Definieren eines Standardwerts.

Die im obigen Abschnitt vorgestellte Bewertung des Fachwertwerkzeugs läßt sich in der folgenden Tabelle zusammenfassen:

	Fachwert-Werkzeug
Aufzählungstypen	++
Einschränkungen des Wertebereichs	++
Undefinierter Wert	++
Einfache Konstruktoren / Pattern Matching	O
Angabe der Stringrepräsentation	+/++
Erstellung neuer Typen / Kennzeichnung von Werttypen	++
Sicherung der Unveränderlichkeit	+/++
Einfache Entwicklung von Werttypen	++

++	sehr gute Unterstützung
+	gute Unterstützung
O	neutral
-	keine Unterstützung

Tabelle 8-1: Unterstützung der Wertkonzepte durch das Fachwertwerkzeug

Da das Fachwertwerkzeug die meisten der für Fachwerttypen geforderten Konzepte unterstützt, stellt es eine gute konzeptuelle Lösung für das Problem dar, Werttypen in einer Sprache zu implementieren, die für diesen Zweck keine besonderen Sprachmittel bereitstellt. Allerdings zeigen sich im praktischen Einsatz auch einige Nachteile, welche im folgenden aufgeführt werden.

Jene Daten, die zur Erstellung eines bestimmten Fachwerttyps auf der Oberfläche des Fachwertwerkzeugs eingegeben wurden, können wiederholt in das Fachwertwerkzeug eingeladen und bearbeitet werden. Wird aus einer so geänderten Fachwerttypbeschreibung erneut eine JWAM-Fachwertklasse generiert, so wird dabei die zuvor generierte Fachwertklasse gleichen Namens überschrieben, so daß eventuelle manuelle Änderungen verloren gehen. Der Grund dafür liegt darin, daß es nicht möglich ist, JWAM-Fachwertklassen in das Fachwertwerkzeug einzuladen. Werden also die vom Fachwertwerkzeug automatisch generierten Fachwerttypen im Anschluß an die Generierung manuell verändert, so können diese Änderungen vom Fachwertwerkzeug nicht berücksichtigt werden. Es ist nicht möglich, solche manuell bearbeiteten Fachwerttypen mit dem Fachwertwerkzeug weiterzubearbeiten.
Eine weitere Beschränkung des implementierten Fachwertwerkzeugs ist, daß es derzeit nur zum Generieren von JWAM-Fachwertklassen eingesetzt werden kann. Theoretisch allerdings sollten die Fachwerttypbeschreibungen zur Generierung von beliebigem Zielcode verwendbar sein. Diese Anforderung der Programmiersprachenunabhängigkeit wurde nicht im vollen Umfang verfolgt, da im Rahmen dieser

Diplomarbeit exemplarisch Quellcode für JWAM-Fachwerttypen erzeugt werden sollte. Die Fachwerttypbeschreibungen wurden daher teilweise auf die Generierung von JWAM-Fachwerttypen ausgerichtet. Dies wirkt sich insbesondere darin aus, daß als Attributtypen direkt Java-Typen angegeben werden. Dadurch wird zwar die Arbeit für Entwickler vereinfacht, die mit der Programmiersprache Java vertraut sind, allerdings wird eventuell die Umsetzung in andere Programmiersprachen erschwert.
Weitere Überlegungen sind darüber anzustellen, welche Konsequenzen potentielle Änderungen des JWAM-Rahmenwerks auf das Fachwertwerkzeug haben würden. Um die Aktualität des Fachwertwerkzeugs sicherzustellen, müßte es Änderungen bei der Struktur der JWAM-Fachwerttypen nachvollziehen. Während die grundlegenden Prinzipien von Fachwerttypen theoretisch fundiert sind und daher keinen großen Änderungen unterworfen sein sollten, trifft dies nicht auf einzelne Methoden, deren Namen oder genaue Implementierung zu. Es ist zu erwarten, daß sowohl die Benutzungsschnittstelle des Werkzeugs als auch der Aufbau der Fachwerttypbeschreibungen bei Veränderungen der JWAM-Fachwerttypen unverändert weiterbenutzt werden können. Angepaßt werden müßten jedoch die durch XSLT-Stylesheets realisierten Codeschablonen, welche den genauen Aufbau von JWAM-Fachwertklassen widerspiegeln - und wie herausgearbeitet wurde, sind Änderungen von XSLT-Stylesheets nicht trivial.

Generell ist das Fachwertwerkzeug für Fachwerttypen begrenzter Komplexität gut einsetzbar. Einige Anteile können hingegen nicht oder nur ansatzweise generiert werden: dies gilt für Kommentare und fachliche Methoden, aber auch für komplizierte Wertebereichseinschränkungen oder Stringrepräsentationen.
Eine Unterstützung der oben genannten Fälle würde zur Folge haben, daß das Fachwertwerkzeug eine große Menge an komplexen Informationen vom Entwickler abfragen müßte. Dadurch würde die Entwicklung von Fachwerttypen jedoch wiederum sehr aufwendig. Für einen erfahrenen Entwickler lassen sich komplexe Funktionalitäten wie beispielsweise fachliche Methoden zudem am besten direkt in einer Programmiersprache ausdrücken.

Als Fazit läßt sich festhalten, daß ein werkzeugbasierter Ansatz einige konzeptuelle Verbesserungen für die Werttypentwicklung bieten kann. So werden die meisten der für Fachwerttypen geforderten Konzepte vom Fachwertwerkzeug gut unterstützt, wobei einige dieser Konzepte durch die Verwendung eines für Fachwerttypen geeigneten Rahmenwerks bereitgestellt werden.
Im praktischen Einsatz erweist sich das Fachwertwerkzeug speziell bei der Erstellung von Fachwerttypen mit einfachem Aufbau als hilfreich, denn in diesen Fällen kann das Werkzeug oftmals vollständigen Code erzeugen.

9 Zusammenfassung

In dieser Diplomarbeit wurde zunächst aufgezeigt, warum es wichtig ist, daß eine Programmiersprache sowohl Mechanismen zur Definition von Objekttypen als auch solche zur Definition von Werttypen bereitstellt. Eine Untersuchung verschiedener objektorientierter Sprachen ergab jedoch, daß diese keine Sprachmittel zum Modellieren von Werttypen zur Verfügung stellen.
Es wurden verschiedene Lösungsansätze betrachtet, die sich mit der Einführung von Werttypen in objektorientierte Sprachen beschäftigen. In diesem Zusammenhang wurden die Java-Erweiterung *Pizza*, ein Ansatz mit Hilfe von XML-Schema und die Fachwertklassen des JWAM-Rahmenwerks betrachtet.
Im Anschluß daran wurden die Konzepte herausgearbeitet, die im Zusammenhang mit Werttypen wichtig sind. Dabei stellte sich heraus, daß das JWAM-Rahmenwerk viele Konzepte unterstützt, die für die Entwicklung von Werttypen erforderlich sind. Allerdings ist die Definition von Werttypen im JWAM-Rahmenwerk rein objektorientiert und ist daher nicht so einfach wie erwünscht.
Es wurde daher ein System entwickelt und implementiert, welches die einfache Erstellung von Werttypen im objektorientierten Umfeld des JWAM-Rahmenwerks ermöglicht und dabei die für Werttypen wünschenswerten Konzepte berücksichtigt.
Das konzipierte System benutzt dazu Fachwerttypbeschreibungen, welche durch ein grafisches Werkzeug erzeugt werden können. Die eigentlichen Werttypen in Form von JWAM-Fachwerttypen werden daraus mittels eines automatischen Codegenerierungsmechanismus erstellt.
In der Auswertung wurde aufgezeigt, daß durch ein solches Werkzeug das Definieren von Werttypen konzeptuell verbessert werden kann. So ist es beispielsweise möglich, eine Stringrepräsentation anzugeben, welche sowohl zum Erzeugen als auch zum Ausgeben eines Werts benutzt werden kann. Ferner können Werttypen durch Angabe ihres Wertebereichs definiert werden.

Zu den in der Auswertung erwähnten, bisher noch ungelösten Problemen des Fachwertwerkzeugs zählt die Programmiersprachenabhängigkeit: derzeit kann das Fachwertwerkzeug nur JWAM-Fachwerttypen generieren. Um die Programmiersprachenunabhängigkeit zu verbessern, wäre es denkbar, statt der Java-Typbezeichner allgemeine Typbezeichner einzusetzen, die dann bei der Codegenerierung in Java-Typen übersetzt werden. Solche allgemeinen Typbezeichner werden beispielsweise von XML-Schema und von der *Interface Definition Language* (*IDL*) (vgl. [COR01]) verwendet.
In bezug auf die Programmiersprachenunabhängigkeit der entworfenen XML-Fachwerttypbeschreibungen bleibt zudem noch zu zeigen, daß diese tatsächlich auch in andere Zielcodes umgesetzt werden könnten.

10 Anhang

In diesem Anhang findet sich eine komplette XML-Fachwerttypbeschreibung für das Beispiel des komplexen Fachwerttyps *dvUhrzeit* sowie die formale Beschreibung des Aufbaus von einfachen und komplexen Fachwerttypen in Form einer DTD (*DocumentTypeDefinition*).

10.1 XML-Fachwerttypbeschreibung am Beispiel des komplexen Fachwerttyps „dvUhrzeit“

In Kapitel 5 wurde herausgearbeitet, welche Informationen für die Generierung von Fachwerttypen benötigt werden und wie sie sich mit Hilfe von XML-Elementen darstellen lassen. Für die Generierung von Fachwerttypen speziell für das JWAM-Rahmenwerk sind noch weitere Informationen sinnvoll; diese Projektoptionen wurden in Abschnitt 6.4 erläutert.
Um zu verdeutlichen, wie das Ergebnis dieser Überlegungen insgesamt aussieht, wird im folgenden ein komplettes Beispiel für eine XML-Fachwerttypbeschreibung vorgestellt. Als Beispiel dient dabei der Fachwerttyp *dvUhrzeit*. Dieser bezieht sich auf das englische Uhrzeitformat und enthält daher neben den beiden Attributen s*tunde* und *minute* noch das Attribut *am_pm* für die Angabe der Tageszeit. Während das Attribut *minute* genau wie beim deutschen Uhrzeitformat Werte zwischen *0* und *59* annehmen kann, liegt der Wertebereich des Attributs *stunde* zwischen *1* und *12*. Das Attribut *am_pm* kann nur zwei Werte annehmen: Weist es den Wert *am* auf, so bezieht sich die angegebene Uhrzeit auf die Zeit vor 12 Uhr mittags; die Angabe *pm* hingegen weist auf den Zeitbereich nach 12 Uhr mittags hin.

Das Fachwertwerkzeug stellt den Beispiel-Fachwerttyp *dvUhrzeit* mit Hilfe folgender XML-Fachwerttypbeschreibung dar:

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

<ERWEITERUNGEN>
   <PACKAGENAME>domainvalue</PACKAGENAME>
   <AUTOR>Annika Spreckelmeyer, Martin Fuerter</AUTOR>
   <COPYRIGHT>AS&MF</COPYRIGHT>
   <VERSION>1.1</VERSION>
</ERWEITERUNGEN>

<FACHWERTTYP>
<NAME>dvUhrzeit</NAME>

<ATTRIBUT zugriff="nein">
   <ATTRIBUT_NAME>stunden</ATTRIBUT_NAME>
   <ATTRIBUT_TYP>int</ATTRIBUT_TYP>
   <ATTRIBUT_EINSCHRAENKUNGEN>
      <MAXIMUM inclusive="true">12</MAXIMUM>
      <MINIMUM inclusive="true">1</MINIMUM>
   </ATTRIBUT_EINSCHRAENKUNGEN>
</ATTRIBUT>
```

```
<ATTRIBUT zugriff="nein">
  <ATTRIBUT_NAME>am_pm</ATTRIBUT_NAME>
  <ATTRIBUT_TYP>String</ATTRIBUT_TYP>
  <ATTRIBUT_EINSCHRAENKUNGEN>
    <AUFZAEHLUNG>
      <WERT>am</WERT>
      <WERT>pm</WERT>
    </AUFZAEHLUNG>
  </ATTRIBUT_EINSCHRAENKUNGEN>
</ATTRIBUT>

<ATTRIBUT zugriff="nein">
  <ATTRIBUT_NAME>minuten</ATTRIBUT_NAME>
  <ATTRIBUT_TYP>int</ATTRIBUT_TYP>
  <ATTRIBUT_EINSCHRAENKUNGEN>
    <MAXIMUM inclusive="true">59</MAXIMUM>
    <MINIMUM inclusive="true">0</MINIMUM>
  </ATTRIBUT_EINSCHRAENKUNGEN>
</ATTRIBUT>

<AUS_STRING_ERZEUGBAR>true</AUS_STRING_ERZEUGBAR >

<STRINGREPRESENTATION>
  <ATTRIBUT_NAME>stunden</ATTRIBUT_NAME>
  <TRENNZEICHEN>:</TRENNZEICHEN>
  <ATTRIBUT_NAME>minuten</ATTRIBUT_NAME>
  <TRENNZEICHEN> </TRENNZEICHEN>
  <ATTRIBUT_NAME>am_pm</ATTRIBUT_NAME>
</STRINGREPRESENTATION>

<SPEZIELLE_WERTE>
  <SPEZIELLER_WERT>
    <SPEZIELLER_WERT_NAME>Mittag</SPEZIELLER_WERT_NAME>
    <ATTRIBUT_WERT_PAAR>
      <ATTRIBUT_NAME>stunden</ATTRIBUT_NAME>
      <ATTRIBUT_WERT>12</ATTRIBUT_WERT>
      </ATTRIBUT_WERT_PAAR>
    <ATTRIBUT_WERT_PAAR>
      <ATTRIBUT_NAME>am_pm</ATTRIBUT_NAME>
      <ATTRIBUT_WERT>pm</ATTRIBUT_WERT>
    </ATTRIBUT_WERT_PAAR>
    <ATTRIBUT_WERT_PAAR>
      <ATTRIBUT_NAME>minuten</ATTRIBUT_NAME>
      <ATTRIBUT_WERT>0</ATTRIBUT_WERT>
    </ATTRIBUT_WERT_PAAR>
  </SPEZIELLER_WERT>
</SPEZIELLE_WERTE>
```

```
<STANDARDWERT>
   <ATTRIBUT_WERT_PAAR>
      <ATTRIBUT_NAME>stunde</ATTRIBUT_NAME>
      <ATTRIBUT_WERT>8</ATTRIBUT_WERT>
   </ATTRIBUT_WERT_PAAR>
   <ATTRIBUT_WERT_PAAR>
      <ATTRIBUT_NAME>am_pm</ATTRIBUT_NAME>
      <ATTRIBUT_WERT>am</ATTRIBUT_WERT>
   </ATTRIBUT_WERT_PAAR>
   <ATTRIBUT_WERT_PAAR>
      <ATTRIBUT_NAME>minute</ATTRIBUT_NAME>
      <ATTRIBUT_WERT>15</ATTRIBUT_WERT>
   </ATTRIBUT_WERT_PAAR>
</STANDARDWERT>

</FACHWERTTYP>
</ERWEITERTER_FACHWERTTYP>
```

10.2 Document Type Definition (DTD) zur Beschreibung von einfachen und komplexen Fachwerttypen

Im folgenden wird die DTD vorgestellt, welche den formalen Aufbau von XML-Fachwerttypbeschreibungen für einfache und komplexe Fachwerttypen spezifiziert.

Die DTD beginnt mit der Angabe, welcher Dokumenttyp definiert werden soll. In diesem Fall soll der Aufbau einer XML-Fachwerttypbeschreibung definiert werden. Dazu wird das höchste Element der Hierarchie der XML-Fachwerttypbeschreibung angegeben (*Erweiterter_Fachwerttyp)*.

```
<!DOCTYPE ERWEITERTER_FACHWERTTYP [
```

Ein erweiterter Fachwerttyp enthält einen Block von Erweiterungen und einen Fachwerttyp:

```
<!ELEMENT ERWEITERTER_FACHWERTTYP (ERWEITERUNGEN, FACHWERTTYP)>
```

Die Erweiterungen enthalten jeweils Packagename, Autor, Copyright und Version:

```
<!ELEMENT ERWEITERUNGEN (PACKAGENAME, AUTOR, COPYRIGHT, VERSION)>
```

Die Elemente Packagename, Autor, Copyright und Version enthalten jeweils Daten, die nicht weiter spezifiziert werden können (pcdata):

```
<!ELEMENT PACKAGENAME (#PCDATA)>
<!ELEMENT AUTOR (#PCDATA)>
<!ELEMENT COPYRIGHT (#PCDATA)>
<!ELEMENT VERSION (#PCDATA)>
```

Ein Fachwerttyp enthält einen Namen und mindestens ein Attribut, sowie die Information darüber, ob Werte dieses Fachwerttyps aus einem String erzeugt werden können und gegebenenfalls die Angabe der Stringrepräsentation. Außerdem kann ein Fachwerttyp eine Liste der speziellen Werte enthalten und/oder einen Standardwert.

```
<!ELEMENT FACHWERTTYP (NAME, ATTRIBUT+, AUS_STRING_ERZEUGBAR,
   STRINGREPRAESENTATION?, SPEZIELLE_WERTE?, STANDARDWERT?)>
```

Der Name des Fachwerts besteht aus Daten, die nicht weiter spezifiziert werden können (pcdata):

```
<!ELEMENT NAME (#PCDATA)>
```

Ein Attribut eines Fachwerttyps setzt sich zusammen aus einem Attributnamen, einem Attributtyp und eventuell einer Liste von Attribut-Einschränkungen. Außerdem wird durch ein DTD-Attribut namens *zugriff* angegeben, ob für dieses Attribut eine Zugriffsfunktion generiert werden soll. Fehlt diese Angabe, so entspricht dies der Angabe *zugriff = nein*.

```
<!ELEMENT ATTRIBUT (ATTRIBUT_NAME, ATTRIBUT_TYP,
  ATTRIBUT_EINSCHRAENKUNGEN?)>
<!ATTLIST ATTRIBUT zugriff CDATA "nein">
```

Der Name sowie der Typ eines Attributs bestehen aus Daten, die nicht weiter spezifiziert werden können (pcdata):

```
<!ELEMENT ATTRIBUT_NAME (#PCDATA)>
<!ELEMENT ATTRIBUT_TYP (#PCDATA)>
```

Die Liste der Attribut-Einschränkungen kann die folgenden Einschränkungen enthalten: *Maximum*, *Minimum*, *Länge*, *Minimale Länge*, *Maximale Länge*, *Muster*, *Anzahl der Nachkommastellen*, *Präzision*, *Aufzählung*.

```
<!ELEMENT ATTIRBUT_EINSCHRAENKUNGEN (MINIMUM?, MAXIMUM?, LAENGE?,
  MINIMALE_LAENGE?, MAXIMALE_LAENGE?, MUSTER?,
  ANZAHL_NACHKOMMASTELLEN?, PRAEZISION?, AUFZÄHLUNG?)>
```

Die Einschränkungen Minimum und Maximum bestehen jeweils aus Daten, die nicht weiter spezifiziert werden können (pcdata) und haben jeweils ein DTD-Attribut *inclusive*, welches definiert, ob der angegebene Wert zum Wertebereich gehört oder nicht. Fehlt diese Angabe, so entspricht dies der Angabe *inclusive = false.*

```
<!ELEMENT MINIMUM (#PCDATA)>
<!ATTLIST MINIMUM inclusive CDATA "false">
<!ELEMENT MAXIMUM (#PCDATA)>
<!ATTLIST MAXIMUM inclusive CDATA "false">
```

Die Einschränkungen *Länge*, *Mininmale_Länge*, *Maximale_Länge*, *Muster*, *Anzahl der Nachkommastellen* und *Präzision* bestehen jeweils aus Daten, die nicht weiter spezifiziert werden können (pcdata):

```
<!ELEMENT LAENGE(#PCDATA)>
<!ELEMENT MINIMALE_LAENGE (#PCDATA)>
<!ELEMENT MAXIMALE_LAENGE (#PCDATA)>
<!ELEMENT MUSTER (#PCDATA)>
<!ELEMENT ANZAHL_NACHKOMMASTELLEN (#PCDATA)>
<!ELEMENT PRAEZISION (#PCDATA)>
```

Die Einschränkung Aufzählung enthält mindestens einen Wert, welcher Daten enthält, die nicht genauer spezifiziert werden können (pcdata).

```
<!ELEMENT AUFZAEHLUNG (WERT+)>
<!ELEMENT WERT (#PCDATA)>
```

Die Angabe darüber, ob ein Wert des Fachwerttyps aus einem String erzeugbar ist, besteht aus Daten, die nicht weiter spezifiziert werden können (pcdata):

```
<!ELEMENT AUS_STRING_ERZEUGBAR (#PCDATA)>
```

Die Stringrepräsentation enthält entweder mehrere Attribute, getrennt durch jeweils ein Trennzeichen, oder nur ein Attribut. Bei beiden Möglichkeiten kann vor dem ersten und nach dem letzten Attribut ein Trennzeichen stehen, welches als Präfix beziehungsweise Suffix bezeichnet werden kann.

```
<!ELEMENT STRINGREPRAESENTATION (TRENNZEICHEN?, ATTRIBUT,
   (TRENNZEICHENEN, ATTRIBUT)*, TRENNZEICHEN?)>
<!ELEMENT TRENNZEICHEN (#PCDATA)>
```

Die Liste der speziellen Werte enthält mindestens einen speziellen Wert:

```
<!ELEMENT SPEZIELLE_WERTE (SPEZIELLER_WERT+)>
```

Ein spezieller Wert hat einen Namen und ein Attribut-Wert-Paar für jedes Attribut, welches für den Fachwerttyp definiert ist:

```
<!ELEMENT SPEZIELLER_WERT (SPEZIELLER_WERT_NAME,
   ATTRIBUT_WERT_PAAR+)>
<!ELEMENT SPEZIELLER_WERT_NAME (#PCDATA)>
```

Ein AttributWertPaar enthält den Namen des Attributs und einen Wert für dieses Attribut:

```
<!ELEMENT ATTRIBUT_WERT_PAAR (ATTRIBUT_NAME, ATTRIBUT_WERT)>
<!ELEMENT ATTRIBUT_WERT (#PCDATA)>
```

Ein Standardwert enthält für jedes Attribut, welches für den Fachwerttyp definiert ist, mindestens ein Attribut-Wert-Paar. Ein Standardwert ähnelt demnach einem speziellen Wert, hat aber keinen Namen.

```
<!ELEMENT STANDARDWERT (ATTRIBUT_WERT_PAAR+)>
```

Ende der DTD:

```
]>
```

11 Literaturverzeichnis

[AND98] Anderson, Paul; Anderson, Gail:
„Navigating C++ and Object-Oriented Design“
Prentice Hall, 1998

[BÄU98] Bäumer, Dirk; Riehle, Dirk; Siberski, Wolf; Lilienthal, Carola; Megert, Daniel; Sylla, Karl-Heinz; Züllighoven, Heinz :
„Values in Object Systems“
Ubilab Technical Report 98.10.1

[BIR92] Bird, Richard; Wadler, Philip:
„Einführung in die funktionale Programmierung“
Prentice Hall, 1992

[BIR98] Bird, Richard:
„Introduction to Functional Programming Using Haskell“
Prentice Hall, 1998

[COR01] OMG Management Group
„The Common Object Request Broker: Architecture and Specification, Revision 2.5“
September 2001
ftp://ftp.omg.org/pub/docs/formal/01-09-34.pdf

[CUN95] Cunningham, Ward:
„The CHECKS Pattern Language of Information Integrity“
In: Coplien, J.O.; Schmidt, D.C.:
„Pattern Languages of Program Design“,
Reading, Massachusetts:Addison-Wesley, 1995

[DAV98] Davis, Mark:
„Durable APIs in Java - Immutables“
Java Report 4(4), April 1999, 70-77

[FAY97] Fayad, M.E.; Schmidt, D.C.:
„Object-oriented Application Frameworks“
In Communications of the ACM 1997, 40, 10

[GAM95] Gamma, Erich et al.:
„Design Patterns – Elements of Reusable Object-Oriented Software“
Addison Wesley, 1995

[GOL83] Goldberg, A.; Robson, D.:
„Smalltalk-80 The Language and its Implementation“
Addison Wesley, 1983

[GOS96] Gosling, James; Joy, Bill; Steele, Guy:
„The Java Language Specification“
Addison Wesley, 1996

[HEN97] Hendrich, Norman:
„Java für Fortgeschrittene “
Springer Verlag, 1997

[HOA72] Hoare, C.A.R.:
„Notes on Data Structuring“
In: Dahl, Dijkstra, Hoare: „Structured Programming“
Academic Press, 1972

[HOL91] Holyer, Ian:
„Functional Programming with Miranda“
UCL Press Limited, 1993

[KAY00] Kay, Michael:
„XSLT – Programmers Reference“
Wrox Press Ltd, 2000

[LOU94] Louden, Kenneth C.:
„Programmiersprachen “
International Thomson Publishing, 1994

[MAC82] MacLennan, B.J.:
„Values and Objects in Programming Languages“
ACM SIGPLAN Notices, Vol.17, No.12, Dec. 1982

[MAR99] Maruyama, Hiroshi; Tamura, Kent; Uramoto, Naohiko:
„XML and Java – Developing Web Applications“
Addison Wesley Longmanl, Inc. 1999

[MCL00] McLaughlin, Brett:
„Validation with Java and XML schema“, parts 1-4
September bis Dezember 2000
http://www.javaworld.com/javaworld/

[MEE94] Meek, Brian:
„A taxonomy of datatypes“
ACM SIGPLAN Notices, Volume 29 No.9, September 1994

[MEY97] Meyer, Bertrand:
„Object-oriented Software Construction“, 2nd edition
Prentice Hall, 1997

[MÜL99] Müller, Klaus:
„Konzeption und Umsetzung eines Fachwertkonzeptes“
Studienarbeit, Fachbereich Informatik der Universität Hamburg, 1999

[OBJ99] Gryczan, Guido; Lilienthal, Carola; Lippert, Martin;
Roock, Stefan; Wolf, Henning; Züllighoven, Heinz:
„Frameworkbasierte Anwendungsentwicklung (Teil 1)“
ObjektSpektrum 1/99, S. 90 – 99

[ODE97] Odersky, Martin; Wadler, Philip:
„Pizza into Java: Translating theory into practice“
In Proc. 24th ACM Symp. Principles of Programming Languages, pp. 146-159, 1997

[PAU91] Paulsen, Lawrence C.:
„ML for the Working Programmer“
Cambridge University Press, 1991

[PER99] Perry, Nigel:
„The Humble Fraction“
SIGCSE Bulletin, Vol 31, No. 4, Dezember 1999

[PLA93] Plasmeijer, Rinus; van Eekelen, Marko:
„Functional Programming and Parallel Graph Rewriting“
Addison-Wesley, 1993

[REA89] Reade, Chris:
„Elements of Functional Programming“
Addison-Wesley, 1989

[RED94] Reddy, Saveen; Wise, G. Bowden:
„An Introduction to C++“
September 1994
ACM Crossroads Student Magazine
http://www.acm.org/crossroads/xrds1-1/ovp.html

[SAU01] Sauer, Joachim:
„Konfiguration von Fachwerten mit XML-Definitionen“
Studienarbeit, Fachbereich Informatik der Universität Hamburg,
Juli 2001

[SIE00] Siegel, Jon.:
„CORBA 3 - Fundamentals and Programming“, 2nd edition
Wiley, 2000

[SPY01] Altova GmbH:
„XML Spy 4.0 Brochure“
http://www.xmlspy.com/download/40/Brochure40.pdf

[STA95] Stansifer, Ryan:
„Theorie und Entwicklung von Programmiersprachen“
Prentice Hall, 1995

[STR00] Stroustrup, Bjarne:
„The C++ Programming Language“
Addison Wesley, 2000

[TUR01] TIBCO Extensibility
„Turbo XML™ - Overview“
http://www.tibco.com/products/extensibility/solutions/turbo_xml_overview_v2_2.pdf

[VEN98] Venner, Bill:
„Design for Thread Safety“
JavaWorld, August 1998
http://www.javaworld.com/javaworld/jw-08-1998/jw-08-techniques_p.html

[WAH98] Wahl, Günter:
„UML kompakt“
OBJEKTspektrum 2/1998
http://www.sigs.de/publications/docs/obsp/umlkompt/umlkompt.htm

[XML00] World Wide Web Consortium (W3C):
„Extensible Markup Language (XML) 1.0 (Second Edition) – W3C Recommendation“
6 October 2000
http://www.w3.org/TR/2000/REC-xml-2001006

[XML01] World Wide Web Consortium (W3C):
„XML Schema - W3C Recommendation“, Parts 0-2
http://www.w3.org/TR/xmlschema-0/
http://www.w3.org/TR/xmlschema-1/
http://www.w3.org/TR/xmlschema-2/

[ZÜL98] Heinz Züllighoven et al.:
„Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz“
Heidelberg: dpunkt-Verlag, 1998

Erklärung

Die Diplomarbeit wurde von mir selbständig in Zusammenarbeit mit Annika Spreckelmeyer durchgeführt. Dabei habe ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Ich zeichne mich für die Kapitel 2, 4, 7, 8 und 9 verantwortlich.

Hamburg, den 15. April 2002

_____

Martin Fürter

Die Diplomarbeit wurde von mir selbständig in Zusammenarbeit mit Martin Fürter durchgeführt. Dabei habe ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Ich zeichne mich für die Kapitel 1, 3, 5, und 6 verantwortlich.

Hamburg, den 15. April 2002

_____

Annika Spreckelmeyer