Einführungsvorträge zum Seminar Dokumentenbeschreibungssprachen im WT 2007

1 Wissensrepräsentation (Christian Szymkowiak) 5
1.1 Wissen 5
1.1.1 Begriffsvereinbarungen 5
1.1.2 cognitive science 5
1.1.3 Menschliches Wissen 6
1.2 Die Darstellung von Wissen 7
1.2.1 Object-Attribute-Value (O-A-V) 7
1.2.2 Genauigkeit 8
1.2.3 Regeln 8
1.2.4 Semantische Netze 8
1.2.5 Rahmen/Schubladen 9
1.3 Sprachen zur Darstellung von Wissen 9
1.3.1 Logikbasierte Sprachen 10
1.3.1.1 Aussagenlogik 10
1.3.1.2 Prädikatenlogik 10
1.3.1.3 Knowledge Interchange Format – KIF 11
1.3.2 Beschreibende Logik 11
1.3.3 Rahmenbasierte Beschreibungssprachen 12
1.3.4 Regelbasierte Beschreibungssprachen 12
1.3.5 Visuelle Beschreibungssprachen 13
1.3.6 Natürliche Sprache als Beschreibungssprache 13
1.4 Ausblick 13
1.4.1 Knowledge Engineering 13
1.4.2 Open Knowledge Base Connectivity 14
1.4.3 Stufen des Wissens 15
2 Ontologien und deren Anwendung (Christian Pauli) 16
2.1 Definitionen (Ontologie) 16
2.2 Aufbau und Aussehen 16
2.3 Eigenschaften 18
2.4 Anwendungen 18
2.4.1 Internetanwendungen 19

2.5 Entwicklung von Ontologien 20
2.5.1 Ontological Engineering 20
2.5.2 Entwicklungssoftware 20
2.5.3 Beschreibungssprachen 21
2.5.4 Methoden 22
2.6 Die Über-Ontologie 22
3 XML (Sebastian Haffner) 24
3.1 XML Sprache - Aufbau 24
3.2 XML strukturieren 26
3.2.1 DTD 26
3.2.2 XML Schema 27
3.3 Namensräume 28
3.4 Adressierung von XML-Elementen 29
3.5 Verarbeitung von XML-Dateien 30
4 RDF/RDFS (Christian Heger) 32
4.1 Einordnung und Abgrenzung 32
4.2 RDF 32
4.2.1 Konzepte 33
4.2.2 Repräsentationen 34
4.2.3 Reifikation 36
4.3 RDF Schema 36
4.3.1 Konzepte 37
4.3.2 Klassen und Eigenschaften 37
4.3.3 Semantik von RDFS 39
4.4 Beispiel 39
4.5 Quellenangaben 40
5 OWL – Web Ontology Language (Sebastian Frenzel) 41
5.1 OWL im Semantic Web layer-cake 41
5.1.1 Anforderungen 41
5.1.2 RDF/RDFS als Grundlage 41
5.1.3 DAML+OIL 42
5.2 Einführung in OWL 42
5.2.1 Die Sprachbeschreibung von OWL Lite 43
5.2.2 Die Sprachbeschreibung von OWL DL/Full 47
5.3 Quellenangaben 50

6 Logik und Folgerungen: Regeln (Blanquett, Geier) 51
6.1 Einleitung 51
6.2 Monotone Regeln 52
6.2.1 Beispiel 52
6.2.2 Syntax 53
6.2.3 Semantik 54
6.3 Nichtmonotone Regeln 56
6.3.1 Beispiel 57
6.3.2 Syntax 59
6.4 Überführung in XML 59
6.4.1 Monotone Regeln in XML 60
6.4.2 Nichtmonotone Regeln in XML 63
6.5 Quelle 65
7 MDA (Spallek, Wloch) 66
7.1 Zielsetzung der MDA 66
7.2 Metamodelle und Modelle 66
7.3 Abstraktionslevels für die Analyse von Systemen 67
7.4 Aufbau der MDA 68
7.5 MDA Metamodelle 69
7.6 Die MOF 69
7.7 UML und Transformationen 71
7.8 XMI 72
7.9 UML Profile und Ontologien 73
7.10 Abbildung der MOF-Elemente auf Java 74
7.10.1 Pakete 74
7.10.2 Klassen 75
7.10.3 Vererbung 76
7.10.4 Attribute und Operationen 77
7.10.5 Assoziationen 77
7.10.6 Einfache und strukturierte Datentypen 78
7.11 Codegeneration mit dem MOmo-Baukasten 79
7.11.1 Der Reader 80
7.11.2 Der ContextGenerator 80
7.11.3 Der TemplateExcecutant 81
7.11.4 Der Schablonenausführer 82

7.12 Fazit 82
8 Modeling Spaces (Peter Wurzler) 83
8.1 Einleitung 83
8.2 Grundlagen 83
8.2.1 Modell 83
8.2.2 Modeling Architecture 83
8.2.3 Einführungsbeispiel 84
8.3 Modeling Spaces 85
8.3.1 Definition 85
8.3.2 Verständnis der realen Welt 86
8.3.3 Arten 87
8.3.4 Beziehungen / Interaktionen 87
8.3.5 Konkretes Beispiel 88
8.3.6 Umwandlungen / Brücken 88
8.4 Anwendung 89
8.4.1 Technical Spaces 89
8.4.2 Nutzen 91
8.5 Quellenangaben 91

1 Wissensrepräsentation (Christian Szymkowiak)

1.1 Wissen

1.1.1 Begriffsvereinbarungen

Im Rahmen dieses Seminars definieren wir Wissen als das Verständnis eines gewissen Themengebietes. Es ist die Grundlage für intelligentes Verhalten und ermöglicht das Erkennen von Zusammenhängen sowie das Finden optimaler Lösungen für Probleme.

Das Ziel der Repräsentation von Wissen ist es einem Computersystem das menschliche Wissen in einer Art zu übergeben, in der es verarbeitet werden kann. Die daraus entstehenden Resultate sollen denen des menschlichen Nachdenkens entsprechen.

Ein Computersystem, welches Wissen ähnlich dem Menschen verarbeiten kann nennt man auch intelligentes System.

1.1.2 cognitive science

Die Wissenschaft, welche sich mit der Studie des Geistes und der Intelligenz befasst nennt man „cognitive science“ oder Wissenschaft über die Erkenntnis. Hierbei handelt es sich um eine interdisziplinäre Wissenschaft. Sie umfasst die Philosophie, Psychologie, künstliche Intelligenz, Neurowissenschaften, Sprachwissenschaften und die Anthropologie. Dabei werden Prozesse und geistige Zustände wie das Denken, Schließen, Erinnern, Lernen, Sprachverständnis, die visuelle und auditive Wahrnehmung, das Bewusstsein sowie Emotionen studiert.

Eine bedeutende Annahme ist, dass der Mensch sein Wissen in mentalen Datenstrukturen repräsentiert, die ähnlich den physischen Datenstrukturen in Computern sind. So existieren im menschlichen Geist zum Beispiel logische Annahmen, Regeln sowie Bilder als Repräsentationen von Wissen. Weiterhin nimmt man an, dass die menschlichen Denkprozesse Computeralgorithmen ähnlich sind.

Zurzeit haben sechs Theorien über die Natur der Darstellung von Wissen im menschlichen Gehirn weit verbreitet.

Formale Logik: Mann nimmt an, dass der Mensch Schlussfolgerungen ziehen kann, weil er gedankliche Konstrukte besitzt, die denen der Prädikaten- oder Aussagenlogik ähnlich sind. Er kann weiterhin mit deduktiven und induktiven Prozeduren auf diesen operieren und so seine Schlüsse ziehen.

Regeln: Große Teile menschlichen Denkens und Argumentierens kann mit Regeln beschrieben werden. Die Menschen folgen gedanklich Regeln und suchen mit Verfahren, welche mit diesen Regeln arbeiten, nach möglichen Problemlösungen. Auf diese Art und Weise entsteht intelligentes Verhalten.

Begriffe: In der cognitive science werden Begriffe als eine Menge typischer Eigenheiten angesehen. Die Begriffe sind einfach Wörter, wie man sie von der Sprache her kennt und drücken in der Regel Beziehungen zwischen Dingen in der Welt aus. Es können so Beziehungen hergestellt und Rangfolgen ausgedrückt werden. Durch das

intelligente Verarbeiten dieser Begriffe können Dinge miteinander verglichen werden und auf allgemeinere Begriffe geschlossen werden.

Analogien: Der Mensch merkt sich verbal oder visuell Situationen, die ihm als Grundlage für ähnliche Situationen dienen können. Mit Verfahren zum Speichern, Wiederfinden und Anpassen dieser kann er sich intelligent verhalten. Computermodelle, die mit Hilfe von Analogien folgern, simulieren diese Prozesse.

Bilder: Der Mensch merkt sich Bilder von Objekten, Gebieten und Situationen. Er ist in der Lage diese zu verarbeiten und zu verändern. Oft eignen sich Bilder besser um Sachen zu beschreiben, die eine längliche Beschreibung in Worten erfordern würden. Bei der Verarbeitung solcher Bilder finden Vorgänge wie Drehen, Spiegeln, Rotation, Vergrößern und Scannen statt.

Neuronale Verknüpfungen: Mentale Repräsentationen von Wissen beinhalten die Verknüpfung von Neuronen, den Verarbeitungseinheiten des Gehirns. Bei manchen geistigen Prozessen weitet sich die Aktivität zwischen ihnen aus, bei anderen werden neue Verbindungen hergestellt oder entfernt. Dadurch entsteht eine Vielzahl intelligenten Verhaltens, wie Entscheidungsfindung und Sprache.

1.1.3 Menschliches Wissen

Psychologen haben eine Reihe von Arten des menschlichen Wissens identifiziert. Aus der Tatsache, dass mehrere Arten des menschlichen Wissens existieren folgern die Forscher, dass der Mensch sein Wissen strukturiert organisiert um damit effektiv Probleme lösen zu können.

Aus den verschieden Arten lassen sich später die entsprechenden Repräsentationstechniken ableiten.

Prozedurales Wissen ist Wissen darüber, wie etwas gemacht wird. Zum Beispiel ein Vorgehen um ein Problem nach einem gewissen Schema lösen zu können.

Deklaratives Wissen beschreibt das, was über ein Thema oder Problem bekannt ist. Das können zum Beispiel Details über Konzepte und Gegenstände sein.

Meta-Wissen ist Wissen über Wissen. Es wird genutzt um zu entscheiden, welches Wissen nun am Besten für die Lösung des Problems geeignet ist. Es wird entschieden, welches Wissen irrelevant ist und gar nicht erst beachtet werden muss bzw. welches Wissen die größten Chancen für eine zufriedenstellende Lösung bietet.

Heuristisches Wissen wird durch Erfahrung erworben. Es hilft Probleme zu vergleichen und effektivere Lösungen zu finden. Eine Problemlösung allein mit Erfahrungswerten verspricht nicht immer Erfolg.

Strukturiertes Wissen beschreibt geistige Modelle sowie die Einordnung von Problemen und Lösungen. Weiterhin beschreibt es Zusammenhänge mit anderen Arten von menschlichem Wissen. Zum Beispiel stellt es einen Zusammenhang mit Prozeduralem Wissen her, wenn eine Gruppe ähnlicher Probleme nach dem gleichen Muster gelöst werden kann. Es drückt Zugehörigkeiten, und Ähnlichkeiten aus.

Ungenaues und unsicheres Wissen beschreibt Probleme, Themen und Situationen bei denen die vorhandenen Informationen nicht für eine zuverlässige Problemlösung ausreichen. Das heißt die Informationen sind unvollständig, nicht zweifelsfrei korrekt oder gar nicht erst vorhanden. Hier findet das subjektive Empfinden des Menschen Ausdruck. Informationen, dass etwas warm, hoch, schnell, oder ein wenig klein sei,

fallen unter diese Kategorie. Anhand solcher Informationen kann nicht einwandfrei auf den tatsächlichen Fakt geschlossen werden.

Weltwissen umfasst das Wissen, was man landläufig als gesunden Menschenverstand kennt. Auf dieses Wissen greift der Mensch zurück, wenn er sich mit nicht ausreichend beschriebenen Problemen konfrontiert sieht oder einfach das präzise Wissen nicht ausreicht. Hier werden dann Analogien gebildet oder es wird von vereinfachten, unvollkommenen Theorien ausgegangen. Zum Beispiel wird ein Bauarbeiter die Grundlagen der Statik nicht in dem Maße wiedergeben können, wie das ein Bauingenieur kann. Er wird aber versuchen anhand der allgemeinen Tatsachen darauf zu schließen.

Ontologisches Wissen fasst das Wissen über ein bestimmtes Thema zusammen und stellt Zusammenhänge zu anderen Themen her. Es beschreibt die Kategorien der Dinge in einem Bereiche und Ausdrücke mit denen darüber gewöhnlich gesprochen wird.

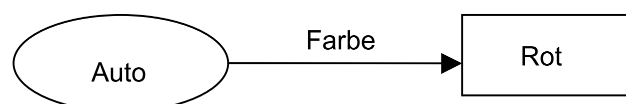
1.2 Die Darstellung von Wissen

Machen wir uns zunächst einmal klar, was die Repräsentation von Wissen leisten soll.

Die Repräsentation soll einen Ersatz für Sachen aus der richtigen Welt liefern. Dieser Ersatz kann die Realität nur unvollständig repräsentieren, wodurch es bei der Verarbeitung später zu falschen Ergebnissen kommen kann. Weiter müssen von Anfang an jeder Repräsentation ontologische Beziehungen zugeordnet werden. Hier muss eine Auswahl getroffen und Beziehungen weggelassen werden, die in der Wirklichkeit zwar vorhanden, für das System aber irrelevant sind. Es muss von vornherein eine Menge möglicher, grundlegender Schlüsse für die Repräsentation geben. Zum Beispiel: „Wenn ein Auto einen Motorschaden hat, dann kann es nicht mehr fahren". Auch hier muss wieder eine Auswahl getroffen werden. Die Darstellung des Wissens muss effizient verarbeitbar sein. Jede Repräsentationstechnik gibt durch sich selbst schon einen Anhalt, wie diese implementiert werden kann. Zuletzt muss sie dem Menschen noch so verständlich sein, dass man sich darüber unterhalten kann. So können Fragen, die während des Sammelns und der Überführung des Wissens in das Computersystem auftreten, noch beantwortet werden. Was kann man damit gut ausdrücken? Wie präzise ist die Repräsentation? Für welche Arten von Wissen ist sie gut nutzbar? Und so weiter.

1.2.1 Object-Attribute-Value (O-A-V)

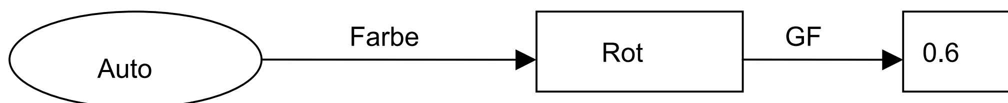
O-A-V-Tripel sind eine Technik, um Fakten über ein Objekt darzustellen. Dabei werden einer Eigenschaft eines Objekts bestimmte Werte zugewiesen. Zum Beispiel kann die Aussage „Das Auto ist Rot" als O-A-V-Tripel wie folgt dargestellt werden.



Da Objekte in aller Regel mehr als nur eine Eigenschaft haben, ergibt sich ein Stern um ein Objekt. Natürlich kann eine Eigenschaft auch mehrere Werte haben. Diese Werte können wiederum selbst Objekte mit Eigenschaften sein.

1.2.2 Genauigkeit

Wie oben bereits erwähnt ist nicht alles Wissen exakt. Um auch unsicheres Wissen darstellen zu können wird dem Wissen ein „Gewissheitsfaktor“ hinzugefügt. Das ist ein diskreter Wert, welcher die Genauigkeit der Aussage wiedergeben soll und aus einem definierten Intervall ausgewählt wird. Im Intervall [0,1] stehen zum Beispiel 0.5 für „wahrscheinlich richtig“, 1.0 für „sicher richtig“ und 0.0 für „unbekannt“. Man kann die O-A-V-Tripel leicht um diesen Faktor erweitern. War das Auto zum Beispiel zu schnell unterwegs um die Farbe mit Sicherheit festzustellen, dann kann die Aussage „Das Auto ist wahrscheinlich rot“ so dargestellt werden.



1.2.3 Regeln

Wissen kann auch durch eine Regel-Struktur repräsentiert werden. Allgemein handelt es sich um Wenn-dann-Konstrukte. Aus einer Vorraussetzung oder der Verknüpfung von mehreren Vorraussetzungen werden Schlüsse, Konsequenzen oder Handlungen abgeleitet. Zum Beispiel:

WENN Die Freundin hat morgen Geburtstag

UND Ich habe kein Geschenk für die Freundin

DANN Sie wird sauer sein.

Die Vorraussetzungen sind für gewöhnlich Fakten, die leicht durch O-A-V-Tripel dargestellt werden können.

Auch hier kann wieder mit den Genauigkeitsfaktoren oder subjektiven Aussagen kombiniert werden. Zum Beispiel:

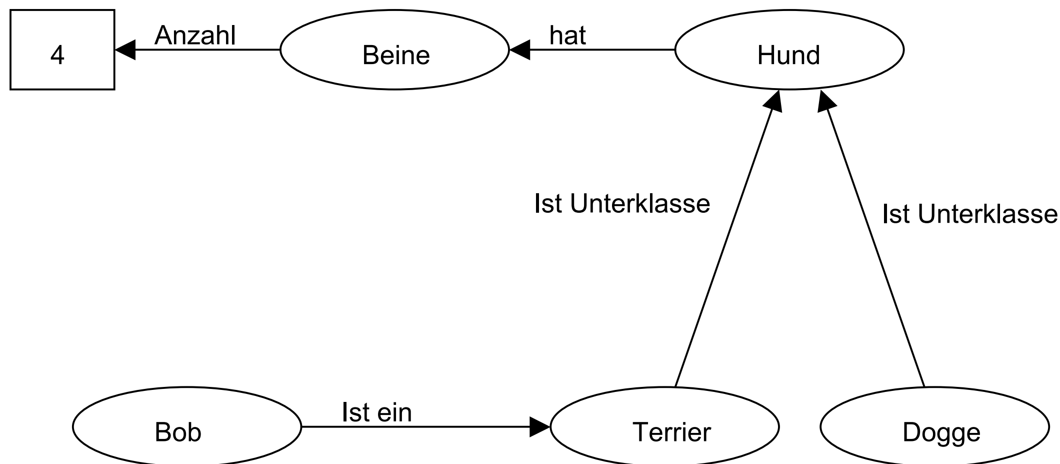
WENN Das Essen ist heiß

DANN Ich sollte ein wenig warten bevor ich es esse.

Regeln, die dasselbe Themengebiet umfassen können in Regelgruppen zusammengefasst werden und durch Techniken, die auf die Repräsentation von ontologischem oder strukturellem Wissens abzielen verwaltet werden.

1.2.4 Semantische Netze

Semantische Netze zielen darauf ab die Wahrnehmung von Sachverhalten, Dingen und Beziehungen darzustellen. Im Wesentlichen ist ein Semantisches Netz ein Graph an dessen Knoten bestimmte Gegenstände oder Themen stehen, welche durch die sie verbindenden Kanten in eine bestimmte Beziehung gesetzt werden.



Die Denkweise beim Erstellen solcher Netze ähnelt der des Objektorientierten Programmierens bzw. einem Brainstorming. Objekte werden mit Pfeilen in eine bestimmte Beziehung zueinander gesetzt. Im Beispiel oben ist Hund die Oberklasse von Terrier und Dogge. Bob ist eine Instanz der Klasse Terrier.

Ein weiterer Vorteil ist die leichte Erweiterbarkeit. Man kann ganz einfach neue Knoten, Relationen und Werte einfügen. Der große Nachteil semantischer Netze ist, dass sie Ausnahmen schlecht repräsentieren und Fehler, besonders in großen Netzen, schlecht zu beseitigen sind. Daher muss man große Sorgfalt beim Aufbau der Netze walten lassen.

1.2.5 Rahmen/Schubladen

Mit der Darstellung von Wissen durch Rahmen lässt sich allgemeingültiges oder klischeehaftes Wissen über ein Objekt darstellen. Auch hier lassen sich wieder Analogien zum Klassenkonzept der objektorientierten Programmierung herstellen. So gibt es zum Beispiel „Klassen-Rahmen“, die gemeinsames Wissen über eine Menge von Objekten zusammenfassen. Für die einzelnen Objekte gibt es „Objekt-Rahmen“, welche die Eigenschaften einer Instanz einer Klasse enthalten. Man kann sich einen Rahmen als ein Formular vorstellen, welches entsprechend ausgefüllt werden muss. Eine typische Position ist für gewöhnlich „Name“.

Rahmen lehnen sich also offensichtlich an die semantischen Netze an. Der Unterschied ist, dass Rahmen auch Prozedurales Wissen repräsentieren. Das kann man sich als eine Prozedur vorstellen, die an einen oder mehrere Positionen gebunden ist und ausgeführt wird, wenn sich eine Eigenschaft ändert oder sie abgerufen wird.

1.3 Sprachen zur Darstellung von Wissen

Nachdem man das menschliche Wissen und die menschliche Denkweise analysiert und abstrahiert hat, ist es nun an der Zeit sich Gedanken über eine mögliche Implementierung als Programmiersprache zu machen.

Ziel des ganzen ist es das menschliche Wissen einem intelligenten System in Form einer Wissensdatenbank so verfügbar zu machen, dass das System damit effizient arbeiten kann.

1.3.1 Logikbasierte Sprachen

Viele andere Repräsentationssprachen bauen auf den logischen Sprachen auf. Ihre Syntax, Semantik und die Schlussregeln sind klar definiert und sie eignen sich daher gut für die Implementierung in Computersysteme. Nachteilig ist, dass alle Aussagen auf Zusicherungen beruhen und sich das Schließen nur auf das Ableiten von Wahrheitswerten oder den Nachweis von Zusicherungen beschränkt. Es ist schwer typisch menschliche Sachen wie Annahmen, Glaube und Zweifel in logischen Sprachen darzustellen.

1.3.1.1 Aussagenlogik

Einzelnen Aussagen werden Variablen zugeordnet. Diese sind entweder wahr oder falsch. Sie können durch logische Operatoren wie „und“, „oder“, „nicht“, „Implikation“ und „Äquivalenz“ miteinander kombiniert werden um so komplexe Regeln und Aussagen zu formulieren.

Sei A := „Die Tochter ist im Garten“, B := „Der Vater ist im Haus“, C := „Der Vater kann die Tochter nicht sehen“. Dann könnte eine Aussage in der Wissensdatenbank so aussehen:

$$A \wedge B \rightarrow C$$

In Worten: „Wenn die Tochter im Garten ist und der Vater im Haus ist, dann kann der Vater die Tochter nicht sehen“.

So können Kombinationen von Aussagen auf ihre Gültigkeit überprüft werden. Schlüsse anhand der Aussage selbst lässt das noch nicht zu. Beim Schließen kommt es nicht auf den Wahrhaltsgehalt der einzelnen Aussage an, sondern auf die Bedeutung. Daher sind Systeme, die mit Aussagenlogik schließen, syntaxbasiert.

1.3.1.2 Prädikatenlogik

In der Prädikatenlogik kommen zu den logischen Operatoren der Aussagenlogik noch der Existenzquantor und der All-Quantor hinzu.

Aussagen werden dadurch gebildet, dass über ein Objekt in Verbindung mit einem Prädikat gewisse Zusicherungen getroffen werden. Die Aussage von oben kann Prädikatenlogisch wie folgt ausgedrückt werden.

$$anOrt(Tochter, Garten) \wedge anOrt(Vater, Haus) \rightarrow \neg kannSehen(Vater, Tochter)$$

Hierbei stellen anOrt(a,b) sowie kannSehen(a,b) die Prädikate dar. Mit Hilfe der Quantoren kann daraus nun zum Beispiel eine allgemeinere Aussage konstruiert werden.

→ Funktionen,

$$\forall x \forall y (anOrt(x, Garten) \wedge anOrt(y, Haus) \rightarrow \neg kannSehen(x, y))$$

In Worten: „Jemand der sich im Garten aufhält kann von jemandem der sich im Haus aufhält nicht gesehen werden“

1.3.1.3 Knowledge Interchange Format – KIF

Das KIF ist eine logische Sprache, welche auf der Prädikatenlogik basiert. Zusätzlich gibt es noch weitere Operatoren, mit denen sich Wissen über Wissen ausdrücken lässt. Dazu die folgenden Beispiele:

(president 820 thomas jefferson) -- repräsentiert einen Datenbankeintrag

(kennt mary ´(president ,?x, ?y, ?z))

Hier wird Wissen über Wissen dargestellt. Der Hochkommaoperator ist auf den inneren Satz angewendet. Das heißt der innere Satz repräsentiert selbst Wissen. Er dient als Argument für die äußere Aussage. Die Fragezeichen zeigen an, dass es sich bei x, y und z um Variablen handelt. Der Kommaoperator, der jeweils davor steht zeigt an, dass die Variablen nicht wörtlich genommen werden sollen. Die Aussage des Satzes ist also dass Mary drei Präsidenten kennt. Ohne den Hochkommaoperator würde sie den Satz an sich kennen.

Die Idee an KIF ist, dass es als Schnittstelle zwischen verschiedenen Repräsentationssprachen dienen kann. So kann jede Sprache (oder zumindest die logischen Sprachen) nach KIF übersetzt werden, und von dort aus in jede andere.

1.3.2 Beschreibende Logik

Die Repräsentation von Wissen mit Hilfe von beschreibender Logik ist von hohem Interesse im Bezug auf Ontologien. Die beschreibenden Systeme bauen auf der Prädikatenlogik auf.

Eine Wissensdatenbank mit beschreibender Logik aufzubauen heißt eine sogenannte „terminological box“ (TBox) und eine sogenannte „assertional box“ (ABox) zu erstellen. Die TBox beinhaltet die Konzepte eines Bereiches. In der ABox werden dann, mir Hilfe des Vokabulars aus der TBox, Zusicherungen über Instanzen der Konzepte getroffen getroffen. Solche Zusicherungen drücken sich in Form von Rollen aus.

Hier ein Beispiel, wie eine TBox in einer solchen Datenbank für familiäre Beziehungen aussehen könnte.

$Frau \equiv Person \sqcap weiblich$

$Mann \equiv Person \sqcap männlich$

$Mutter \equiv Frau \sqcap \exists hatKind.Person$

$Vater \equiv Mann \sqcap \exists hatKind.Person$

$Eltern \equiv Vater \sqcup Mutter$

$MutterMitVielenKindern \equiv Mutter \sqcap \geq 3\ hatKind$

Person, weiblich und männlich sind einzelne Begriffe. Damit werden nun die Begriffe Frau, Mann, Mutter, Vater und Eltern dargestellt.

Eine darauf aufbauende ABox könnte so aussehen:

MutterMitVielenKind(JANE)

hatKind(JANE, KLARA)

Vater(BOB)

hatKind(BOB, KLARA)

Mit dieser Art der Repräsentation von Wissen können auch, in gewissen Grenzen, Schlüsse gezogen werden. So kann anhand der TBox bewertet werden, ob eine Beschreibung zufriedenstellend ist oder ob eine Beschreibung spezieller als eine andere ist. So beinhaltet die Beschreibung von Frau die von Person und ist damit spezieller. Mit der ABox kann anhand der Beschreibung eines Objektes auf sein Konzept geschlossen werden. Aus der Beschreibung „MutterMitVielenKind(JANE)“ folgt, dass JANE eine Mutter ist. Als Ergänzung können auch noch Regeln genutzt werden. Diese können Operatorform haben. zum Beispiel „$C \Rightarrow D$“. Wenn ein Objekt Instanz von C ist, so ist es auch Instanz von D. Sie können auch beschreibende Form haben. Beispielsweise: $\mathbf{K}C \subseteq D$. Der Operator **K** beschränkt die Anwendung des Axioms auf Objekte, für die aus der ABox und TBox folgt, dass sie Ausprägungen von C sind. Solche Regeln können genutzt werden um neue Zusicherungen in die ABox aufzunehmen.

1.3.3 Rahmenbasierte Beschreibungssprachen

Der Aufbau von Rahmenbasierten Sprachen orientiert sich sehr am Konzept der Objektorientierung und liegt dem Menschen daher näher, als zum Beispiel logische Sprachen. Hier ein Beispiel aus der Sprache KM.

```
(every Buy has
        (buyer ((a Agent)))
        (object((a Thing)))
        (seller((a Agent))))
```

Hier wird die Klasse “Buy” eingeführt und durch die Aussagen, dass jeder Kauf einen Käufer, eine Ware und einen Verkäufer hat definiert.

1.3.4 Regelbasierte Beschreibungssprachen

Regelbasierte Sprachen werden sehr häufig im Bereich der künstlichen Intelligenz eingesetzt. Ein Regelbasiertes System ist leicht zu verstehen und es existieren bereits Tools mit denen eine Regeldatenbank erstellt werden kann. Regelbasierte Systeme haben zwei entscheidende Nachteile.

Bei großen Problemen entsteht ein großes Durcheinander in der Regeldatenbank, da es keinen Ordnungsmechanismus gibt. Weiterhin können Schlüsse nicht auf das Objekt, um das es gerade geht, bezogen werden. Man kann nur generelle Schlussfolgerungen ziehen, die dann neue Regeln ergeben.

Daher wird das regelbasierte Konzept auch gerne mit rahmenbasierten Sprachen kombiniert. Das hat den Vorteil, dass Regeln objektspezifisch und generell formuliert werden können. Durch die Rahmen ist so auch eine Ordnung der Regelmenge möglich. Hier ein Beispiel einer solchen Regel-Rahmenimplementierung:

```
(deftemplate buy(slot price))     -- ein Buy-Objekt mit einer Preisangabe
(defrule check-price              -- neue Regel wird definiert
        (buy (price ?x))          -- Wert des Preises wird geholt
        (test (> ?x 100))         -- Vergleich
=>                                -- Ausgabe, wenn die Regel erfüllt ist
```

```
(printout t ?x „kostet mehr als 100“ crlf))
```

Derartige Sprachen werden bereits mit an XML angelehnten Sprachen realisiert. Hier ein Beispiel aus der Sprache „RuleML“.

```
<rulebase label="likesMusic">
    <imp>
        <_head><atom>
            <rel>likes</rel>
            <ind>John</ind>
            <var>x</var>
        </atom></_head>
        <_body><atom>
            <rel>likes</rel>
            <var>x</var>
            <ind>music</ind>
        </atom></_body>
    </imp>
</rulebase>
```

Das Codebeispiel sagt aus, dass wenn jemand Musik mag John ihn mag.

1.3.5 Visuelle Beschreibungssprachen

Sprachen zur Beschreibung von visuellem Wissen existieren so noch nicht. Ansätze existieren zum Beispiel bei UML oder den Benutzeroberflächen, mit denen Wissen zur Überführung in andere Sprachen eingegeben wird. Es soll hier nicht näher darauf eingegangen werden.

1.3.6 Natürliche Sprache als Beschreibungssprache

Der wohl größte Nachteil der natürlichen Sprache als Wissensrepräsentationssprache ist ihre Komplexität. Daher gestaltet sich ihre Verarbeitung mit Computern sehr schwierig. Ein Vorteil ist ihre leichte Verständlichkeit. Ein Ansatz die Vorteile natürlicher Sprache zu nutzen ist sogenannte kontrollierte Sprachen zu verwenden. Dabei wird die Sprache in der Syntax begrenzt, so dass sie leichter von einem Parser verarbeitet werden kann. So kann das Wissen über diesen Zwischenschritt in eine andere Repräsentationssprache übersetzt werden.

1.4 Ausblick

1.4.1 Knowledge Engineering

Das Ziel der Entwicklung von Sprachen zur Repräsentation von Wissen ist es letztendlich Wissen zu sammeln und so zu speichern, dass es effizient verarbeitet wer-

den kann. Den Vorgang des Sammelns von Wissen und es einem intelligenten System verfügbar zu machen nennt man Knowledge Engineering. Diejenigen, die diese Tätigkeit ausüben nennt man Knowledge-Engineers. Das Sammeln von Wissen bringt neben der Suche nach der optimalen Repräsentationstechnik auch noch andere Schwierigkeiten mit sich, denen sich der Knowledge-Engineer bewusst sein muss. So können beispielsweise Missverständnisse beim Übernehmen von Wissen von Experten, Interpretationsprobleme beim Arbeiten mit mehreren verschiedenen Hilfsmitteln oder Wartungsprobleme bei der Datenbank auftreten. Das Wissen, welches gespeichert ist muss stets auf dem neuesten Stand gehalten werden. So kann das Ändern eines Faktums die Änderung von weiteren Datenbankeinträgen nach sich ziehen damit es letztendlich nicht zu Widersprüchen führt.

Knowledge-Engineers arbeiten mit Experten der einzelnen Wissensgebiete zusammen. Sie sammeln das Wissen mit diesen Leuten. Das geschieht auf viele Arten. So sind zum Beispiel Befragungen, die Beobachtung bei bestimmten Tätigkeiten und das Hinterfragen von Handlungen und Denkweisen der Experten durch die Knowledge-Engineers Mittel beim Sammeln und verstehen von Wissen.

Es haben sich über die Zeit einige Methodologien geformt. Die bekannteste ist CommonKADS. Danach wird die Entwicklung von mehreren Modellen für den Bereich des Wissens befürwortet. So gibt es Modelle für Konzepte, für die Organisation und für die Anwendung von Wissen. So sollen die vielen Teilaspekte des Wissens abgedeckt werden. CommonKADS unterscheidet das Wissen weiterhin in vier Kategorien. Wissen über ein bestimmtes Gebiet, schließendes Wissen, Wissen über Vorgänge und strategisches Wissen. So soll das Erstellen von Wissensdatenbanken erleichtert werden.

1.4.2 Open Knowledge Base Connectivity

In aller Regel werden Knowledge Enineers Wissensdatenbanken nur für ein bestimmtes Themengebiet erstellen. Daher wird es unumgänglich sein, dass die verschiedenen entstandenen und im Entstehen begriffenen Datenbanken untereinander kommunizieren und sich austauschen können. Hierbei kann auch wieder eine Vielzahl von Schwierigkeiten auftreten. So müssen zum Beispiel die eventuell unterschiedlichen Repräsentationstechniken in einander überführt werden, ohne dass dabei Wissen verloren geht oder verfälscht wird. Ein Ansatz zur Lösung des Problems heißt „Open Knowledge Base Connectivity“, kurz OKBC. Dabei handelt es sich um ein Protokoll, das den Zugriff auf Wissensdatenbanken regeln soll. Es enthält eine Menge generischer Operationen, welche als ein Interface für den Zugriff auf die Wissensdatenbanken dienen. Jede generische Operation von OKBC erfordert eine konkrete Implementierung in der Programmiersprache der Anwendung. Diese generischen Operationen werden in OKBC unabhängig von Programmiersprachen spezifiziert. Für weit verbreitete Programmiersprachen wie Java, C oder Lisp werden aber auch geeignete Bezüge hergestellt, um die entsprechende Implementierung zu erleichtern.

Man sollte beachten, dass OKBC keine Wissensrepräsentationssprache ist, sondern nur den Austausch von Wissen regelt. Das darunterliegende Wissensmodell ist objektorientiert. Das heißt der Austausch erfolgt in Form von Rahmen, Ausprägungen und Klassen. Mit Hilfe der generischen Operationen wird das Wissen aus einer Wissensdatenbank nach OKBC und zurück übersetzt.

1.4.3 Stufen des Wissens

Zum Zwecke der besseren Übersichtlichkeit wurden von Allen Newell drei sogenannte Wissensstufen eingeführt. Sie spiegeln jeweils eine bestimmte Abstraktionsebene der Wissensrepräsentation wieder und werden im Folgenden kurz erläutert.

Stufe der Implementierung: Hier befindet man sich auf der Ebene der Wissensdatenbank des intelligenten Systems. Wenn auf dieser Ebene von Wissensrepräsentation gesprochen wird, so sind die Datenstrukturen wie sie in der Datenbank implementiert sind gemeint.

Stufe der Logik: Diese Stufe beinhaltet bereits die darunterliegenden Datenstrukturen, deren Beziehungen zueinander und die Repräsentationstechnik in der sie implementiert sind. Aus den Daten werden auf dieser Stufe nun Aussagen formuliert. Einfacher kann man sagen, dass Wissensrepräsentation auf dieser Stufe die zu Sätzen formulierten Daten einer Wissensdatenbank sind.

Stufe des Wissens: Hier wird die Gesamtheit des Wissens eines intelligenten Systems betrachtet. Auf dieser Stufe lässt sich ausdrücken, über welche Bereiche ein intelligentes System etwas weiß, was die Ziele der Verarbeitung des Wissens sind und nach welchen Vorgehensweisen das System diese Ziele erreichen kann.

2 Ontologien und deren Anwendung (Christian Pauli)

2.1 Definitionen (Ontologie)

In folgenden Abschnitt behandele ich nur solche Ontologien, welche der Wissensrepräsentation auf Rechensystemen dienen. Es gibt viele Definitionen vom Begriff *Ontologie*. Die am meisten Zitierte ist: eine Ontologie ist die *Spezifikation einer Konzeptualisierung*. Es beschreibt die Bedeutung dieses Begriffes vermutlich am präzisesten. Eine Konzeptualisierung ist dabei ein abstraktes Abbild der realen Welt. Dieses Abbild repräsentiert eine Wissensbasis und dient immer einem gewissen Zweck. Dieser Zweck entscheidet dann darüber, wie dieses Wissen strukturiert ist. Eine Spezifikation ist eine Beschreibung und beschreibt in diesem Fall, wie die Konzeptualisierung im Detail deklariert ist. Dazu bestimmt sie unter anderem, welche Typen benutzt werden und deren Definitionsbereiche. Dazu benutzt sie eine, von ihr festgelegte, formale Sprache.

Eine andere Definition beschreibt die Ontologie als *eine Menge von Wissen*. Diese Menge enthält Informationen über das Vokabular sowie über die semantischen und logischen Zusammenhänge, des von ihr beschriebenen Sachverhaltes. Aus dieser Definition sticht vor allem hervor, dass eine Ontologie nicht nur Begriffe, sondern auch Zusammenhänge zwischen ihnen beschreibt. Ein Begriff ist hierbei ein Bezeichner für eine Kategorie von Objekten aus dem zu beschreibenden Sachverhalt. Weiterhin können Gesetzmäßigkeiten enthalten sein. Daraus kann man schlussfolgern, dass nicht nur innerhalb einer Ontologie Informationen miteinander vernetzt sind, sondern dass auch verschiedene Ontologien aufeinander verweisen können.

Eine Ontologie ist eine Basisstruktur, um die eine Wissensbasis aufgebaut werden kann lautet eine weitere Definition. In dieser wird eine Ontologie als eine Art Skelett beschrieben, an das sich weiteres Wissen angliedern kann. Dabei wird jedoch zwischen ontologischem und allem anderen Wissen unterschieden. Das ontologische Wissen stellt dabei nur das Fundamentalwissen zum jeweils behandelten Thema dar, alle anderen Informationen beziehen sich nur auf die Ontologie. Eine weitere Definition stellt dabei fest, dass dieses fundamentale Wissen immer von einer gewissen Gruppe als wahr akzeptiert wird, womit es also eine Art Konsens innerhalb dieser Gruppe darstellt.

2.2 Aufbau und Aussehen

Das Aussehen einer Ontologie hängt wesentlich von der Abstraktionsebene ab. Es gibt die Möglichkeit, Ontologien in natürlicher (menschlicher) Sprache auszudrücken, was jedoch die Abarbeitung auf Rechensystemen erheblich erschwert und zudem oft Raum für Interpretationen offen lässt. Aus diesem Grund, werden sie oft als Grafik dargestellt. Doch auch hier variieren die Formen der Darstellung und deren Detailgrad sehr stark. Letztendlich wird jede, auf einem Rechensystem zu nutzende Ontologie in einen Quellcode einer Programmiersprache überführt, welcher von grafischen Editoren oft schon automatisch erzeugt wird. Es folgen Beispiele: *Ein Musiker spielt ein Instrument, nimmt ein Album auf, tritt bei verschiedenen Konzerten auf, welche von Fans besucht werden*. Dieser, durch natürliche Sprache ausgedrückte

Satz, deklariert bereits einige Kernpunkte einer Ontologie, die die Arbeit eines Musikers beschreibt. Dieser einfache Zusammenhang ist nochmals in Bild 1.1 dargestellt. Deutlich zu erkennen sind verschiedene *Begriffe* (Instrument, Album, Event) und *Beziehungen* zwischen ihnen (attends, plays). Die Begriffe werden normalerweise durch *Attribute* näher beschrieben. Ein Musiker hat zum Beispiel immer einen Namen, genauso wie ein Album. Dies ist jedoch in Bild 1.1 nicht der Fall. Da ist die Darstellung in Bild 1.2 schon wesentlich detailreicher: es zeigt ein UML-Klassendiagramm und enthält neben geeigneten Attributen, welche eine Instanz eindeutig identifizieren, auch die zugehörigen Kardinalitäten.

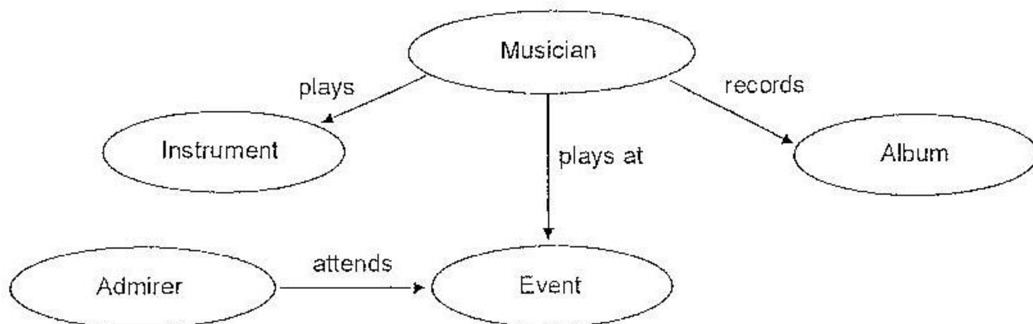


Bild 1.1

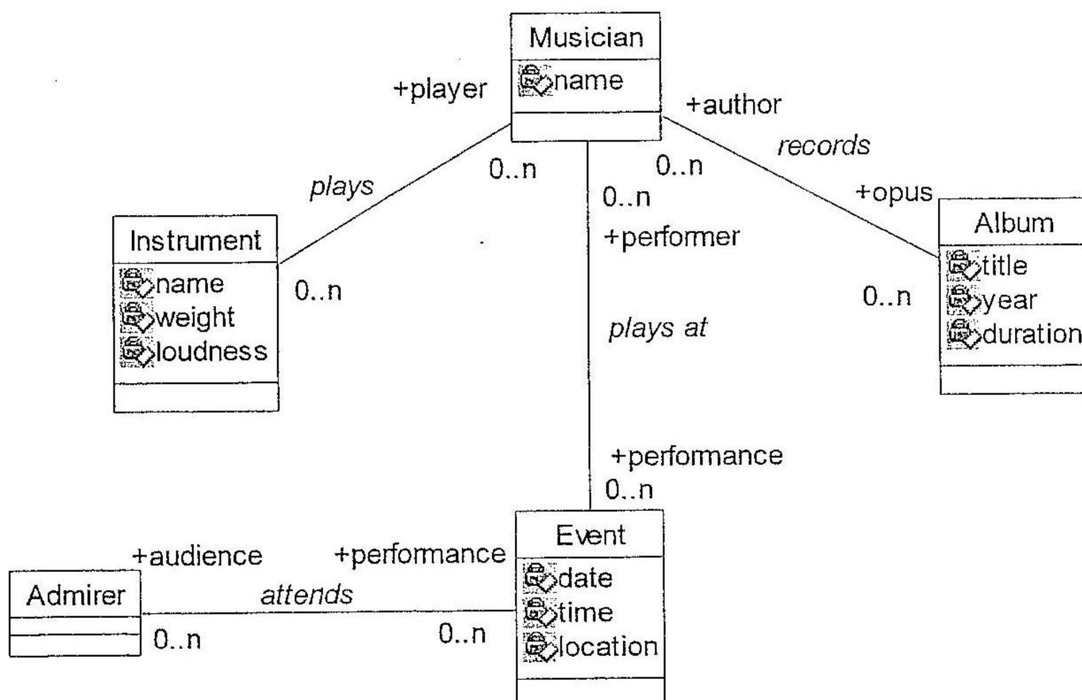


Bild 1.2

2.3 Eigenschaften

Eine Ontologie stellt zu einem Sachverhalt einen Begriffskatalog bereit. Dieser Katalog gibt eindeutig Auskunft über die Bedeutung eines Begriffes, so dass es in dieser Beziehung keinen Spielraum für Interpretationen gibt, so wie das bei menschlicher Sprache durchaus der Fall sein kann. Die Bedeutung eines Begriffes ist dabei unabhängig von Sprache oder den kommunizierenden Elementen.

Weiterhin legt eine Ontologie die Taxonomie, des von ihr dargestellten Sachverhaltes fest. Sie klassifiziert die Begriffe nach bestimmten Kriterien und ordnet sie in Kategorien ein. Diese Kategorien können auch hierarchisch aufgebaut sein. Die Unterteilung der Begriffe sollte dabei eindeutig und, trotz fester Aufteilung, möglichst viele Möglichkeiten offen lassen, also: flexibel sein. Die Taxonomie einer Ontologie ist jedoch noch mehr als eine bloße Kategorisierung der enthaltenen Begriffe, sondern sie stellt weiterhin einen wesentlichen Teil des konzeptionellen Frameworks dar.

Ontologien enthalten nicht nur bestimmte Begriffe und deren Beziehungen untereinander, sondern spezifizieren diese auch sehr ausführlich. Dies geschieht mittels speziellen Ontologiebeschreibungssprachen. Für jeden Begriff werden dazu Attribute definiert. Ontologien können auch Konsistenztests unterstützen, welche sicherstellen, dass Restriktionen von Attributbelegungen (Typ, Wertebereich,…) eingehalten werden. Letztendlich repräsentiert eine Ontologie alles Wissen über einen Sachverhalt auf strukturierte Art und Weise. Damit wird schließlich dessen Struktur und auch das beobachte Verhalten formal beschrieben.

Ein Kernziel bei der Benutzung von Ontologien ist es, Wissen für viele Anwendungen verfügbar und damit auch wieder verwendbar zu machen. Datenbanken könnten sich in Zukunft einfacher realisieren lassen: viele Datenbanken enthalten jeweils nur das gleiche Basiswissen und erweitern sich durch den Zugriff auf andere Datenbanken. Das durch die gemeinsam genutzte Ontologie präsentierte Wissen ist dann quasi eine Art Schablone, die zu erstellende Datenbanken als Grundgerüst verwenden und beliebig erweitern können. Ontologien begünstigen durch eindeutig festgelegte Begriffe die Kommunikation zwischen Menschen oder Softwareapplikationen und eignen sich deshalb als Kommunikationsmittel oder -schnittstelle.

2.4 Anwendungen

Anwendungsmöglichkeiten für Ontologien gibt es reichlich. Sie eignen sich insbesondere für Anwendungen in Bereich des Wissensmanagements. Schwerpunkt sind Softwareanwendungen und -entwicklung, insbesondere bei Internetanwendungen.

Ontologien können dazu dienen, die Kooperation in Entwicklerteams zu verbessern, indem sie eine einheitliche Wissensgrundlage schaffen. Dadurch können verschiedene Sichtweisen auf ein Problem und unterschiedlicher Wissens- oder Erfahrungsstand der Mitarbeiter zwar nicht behoben aber die daraus resultierenden Probleme aufgefangen werden. Eine Ontologie stellt einem solchen Team die Wissensgrundlage bereit, die jeder Mitarbeiter benötigt, um aktiv am Geschehen teilnehmen zu können und auf der jeder Mitarbeiter mit seinem Arbeitsanteil aufsetzen kann. Weiterhin wird auch die Kommunikation innerhalb des Teams erleichtert, da jeder über ein einheitliches Wissen verfügt, ohne jedoch Spezialist in jedem Bereich zu sein. Auch die Kommunikation zwischen Softwareanwendungen wird durch gemeinsam benutzte Ontologien erleichtert.

Bei der Entwicklung von wissensbasierter Software können Ontologien die Bereitstellung und Wiederverwendbarkeit bestimmter Informationen gewährleisten. Diese können dann von vielen Anwendungen als vorgefertigte Bausteine verwendet werden, was die Entwicklungszeit solcher Software erheblich verkürzen kann. Für die gemeinsame Nutzung von Daten eignen sich Ontologien ebenfalls. Nicht nur die Bereitstellung der Daten selbst kann dabei durch Ontologien geschehen, sondern auch die Probleme, welche verschiedene Formate und mangelnde Kompatibilität verschiedener Anwendungen mit sich bringen, können durch gemeinsam genutzte Ontologien gelöst werden. Da das Internet das bedeutendste Medium für Wissensströme geworden ist, sind Internetanwendungen ein Schwerpunkt bei der Benutzung von Ontologien als Datenstruktur. Um vorhandenes Wissen besser vernetzen zu können, beispielsweise durch den Austausch von Webdokumenten, ist es notwendig, dass diese ontologisch kodierte Informationen enthalten. *Briefing Associate* beispielsweise beschreibt ein Verfahren, welches in dem Moment, wenn ein Dokument verfasst wird, eine passende Erläuterung (Markup-Information) automatisch ontologisch kodiert in das Dokument einfügt.

Wie bereits erwähnt, stellen jeweils Ontologien, das von einer gewissen Gruppe akzeptierte Konsenswissen über eine Thematik dar. Ontologien eignen sich dazu dieses Wissen auch verfügbar zu machen. Im Rahmen der Bildung und Forschung können Benutzer dann verlässliche und objektive Informationen über ein Thema erhalten. Ein anderes Problem an dieser Stelle wäre noch das Suchen von gewünschten Informationen (→ Internetanwendungen).

Ein weiteres Anwendungsgebiet von Ontologien wäre der Einsatz als *Datenstruktur für Metadaten*. *Metadaten* sind maschinenlesbare Informationen über elektronische Ressourcen oder andere Dinge. Ein *Metamodell* beschreibt wiederum ein anderes Modell. Diese Eigenschaften charakterisieren ein Metamodell als eine spezielle Ontologie. Meist werden mit Metadaten digitale Dokumente beschrieben, also beispielsweise Internetseiten. Sie geben dabei beispielsweise Auskunft über den Autor, Titel und Zeitpunkt, also über die Semantik einer Seite. Die Darstellungssprachen für Ontologien wären deshalb auch als *Metamodellsprachen* geeignet. Allerdings müssten in diesem Bereich erst noch einheitliche Standards definiert werden.

Internetanwendungen

Eine der wichtigsten Internetanwendungen sind Suchmaschinen. Diese können Ontologien als Datenstruktur nutzen, da jene das strukturierte, vergleichende und individuelle Suchen von Informationen unterstützen. Unmengen an, zu einem Suchbegriff unpassenden Seiten, würden der Vergangenheit angehören, da der Suchalgorithmus anhand der Taxonomie und der begrifflichen Einteilung der Ontologie immer die passende Kategorie finden würde. Falls mehrere Kategorien zum Suchbegriff passen, so könnte der Benutzer aufgefordert werden, zwischen diesen auszuwählen, damit der Suchalgorithmus die Suche präziser fortsetzen kann. Ein Vorraussetzung dafür ist jedoch, dass der Suchalgorithmus die Semantik einer Internetseite kennt. Dies soll zukünftig, im Rahmen des Semantic Web, durch Ontologiebeschreibungssprachen wie OWL oder RDF sichergestellt werden. Rechensysteme sind dann in der Lage die Semantik einer Seite auszulesen. Dazu ein Beispiel: bei der Internetsuche nach dem Begriff „House“ können vielerlei Dinge gemeint sein: der Musikstil *House*, Einrichtungen oder Institutionen die den Begriff *House* im Namen führen, Einrichtungshäuser, Filme, Schauspieler,… Moderne Suchmaschinen liefern all diese Treffer in einer mehr oder weniger ungeordneten Liste, was für den Informationssuchenden meist nicht zufriedenstellend ist. Bei

nicht zufriedenstellend ist. Bei zukünftigen Anwendungen könnte der Anwender zwischen verschiedenen, vom Suchalgorithmus identifizierten, Kategorien auswählen, in denen danach weitergesucht wird. Es würde nicht, wie das aktuell der Fall ist, einfach nur nach bestimmten Schlüsselwörtern gesucht werden.

Mit Ontologien kann auch das Web-Browsing vereinfacht werden: ein Web-Browsing-Tool namens *Magpie* unterstützt schon heute die semantische Interpretation von Internetseiten. Es zeigt einem Benutzer zu einer ausgewählten Ontologie passende Webinhalte. Dies beinhaltet neben Text-Highlighting von in der Ontologie vorkommenden Begriffen auch das Anzeigen verwandter Themen oder Seiten. Dies ermöglicht sehr individuelles Web-Browsing.

FOAF ist ein Internetprojekt zur maschinenlesbaren Darstellung sozialer Netzwerke. Es ist eines der ersten Anwendungen des Semantic Web. Informationen von Personen werden in einem Dokument erfasst. Eine Software kann dieses Dokument dann analysieren, durch Einordnen in eine Ontologie soziale Kontakte feststellen und diese visuell sichtbar machen. Beispiel: ein angemeldeter Benutzer kann feststellen, ob es noch weitere Personen aus seiner Heimatstadt gibt, die zufällig in seiner Firmenabteilung arbeiten, um beispielsweise eine Fahrgemeinschaft zu gründen.

Eine weitere, heute schon verbreitete Anwendung, sind sogenannte *Recommender Systems* (Empfehlungssysteme) im Rahmen von e-commerce Anwendungen. Diese Software beobachtet Benutzer und erstellt mit Hilfe von diversen Daten ein ontologiebasiertes Benutzerprofil. Anhand dieses Profils sucht das Programm dann passende Kaufempfehlungen für den Benutzer. Ein sehr bekanntes *Recommender System* ist das von *Amazon.com*.

2.5 Entwicklung von Ontologien

Ontological Engineering

Das sogenannte *Ontological Engineering* bezeichnet eine Ansammlung von verschiedenen Entwicklungsprinzipien und -prozessen, welche gewährleisten sollen, dass die entstehende Ontologie möglichst leicht zu implementieren, zu warten, wieder verwendbar, portierbar und zu modifizieren ist. Es ist ein iterativer Prozess. Das Verfahren stellt dazu auch ein Konzept zur Entwicklung einer Wissensbasis bereit und unterstützt das Strukturieren und Aneignen von benötigtem Wissen. Die Entwicklung einer Ontologie stellt einen langwierigen, nicht-trivialen Prozess dar, der viel Disziplin und Erfahrung erfordert. Zur Darstellung von Ontologien werden Entwicklungsumgebungen sowie entsprechende Beschreibungssprachen benötigt.

Entwicklungssoftware

Moderne Entwicklungsumgebungen, wie beispielsweise *Protégé*, stellen dem Entwickler für jede Entwicklungsphase die passende Funktionalität bereit und versuchen ihn möglichst zu entlasten. Sie haben in der Regel eine grafische Benutzeroberfläche und stellen einen Ontologieeditor zu Verfügung. Dem Entwickler stehen meist eine Anzahl von Standardformen zur Verfügung und die Software kann möglicherweise syntaktische Fehler erkennen. Bild 1.4 zeigt einen Screenshot von *Protégé*. Zusätzliche Tools können Aufgaben wie das Konvertieren in andere Sprachformate, Vergleich verschiedener Ontologien oder das Zusammenführen von Ontologien erfüllen. Weiterhin gibt es auch Softwareunterstützung für das Aneignen, Organisieren und

Visualisieren von Informationen. Meist werden auch mehrere Beschreibungssprachen unterstützt. Der Entwickler soll jedoch noch weiter entlastet werden: *Ontology Learning* heißt eine Idee, welche zum Ziel hat, einen arbeits- und zeitaufwendigen Bereich des Entwicklungsprozesses wenigstens teilweise zu automatisieren. Dieser Bereich umfasst das automatische Aneignen neuer Informationen und der Integration in die bestehende Ontologie. Bereits bestehende Ontologien sollen also in der Lage sein sich automatisch weiterzuentwickeln. Die Quellen möglicher neuer Informationen wären unter Anderem Internetseiten, Datenbanken und andere Ontologien. Ein von *Maedche* und *Staab* vorgestelltes Framework basiert dabei auf der Verarbeitung natürlicher Sprachen im Zusammenhang mit Data- und Text-Mining. Bei dem Verfahren wird eine bestehende Ontologie durch neues Wissen erweitert und danach bezüglich ihres Nutzens validiert. Ist das Ergebnis, bezogen auf ihren Nutzen, schlechter, so wird die Erweiterung wieder verworfen, andernfalls beibehalten. Dieser Prozess kann beliebig oft wiederholt werden, verläuft jedoch nicht vollautomatisch. Der Entwickler muss regelmäßig in den Programmablauf steuernd eingreifen. Noch aber befinden sich Ontology-Learning-Tools in ihrer Entwicklungsphase.

Beschreibungssprachen

Zur Implementierung von Ontologien wird eine Beschreibungssprache benötigt, von denen es wiederum viele gibt. Einige bekannte sind: KIF, RDF, OIL, OWL. Die meisten dieser Sprachen wurden entwickelt, um Ontologien für das *Semantic Web* darzustellen. Die Letztgenannte ist dabei unter Federführung des W3C entstanden und heute die populärste Sprache dieser Art. Die Vielfalt der Sprachen stellt Entwickler vor Probleme. Zwei in einer unterschiedlichen Sprache implementierte Ontologien sind oft nicht kompatibel und nur mit erheblichem Aufwand zusammenzuführen. Auch bereits vorhandene Übersetzungstools arbeiten nicht immer zufriedenstellend. Eine Lösung dieses Problems, beispielsweise in Form einer neuen universellen Beschreibungssprache, ist nicht in Sicht. Das Bild 1.3 zeigt einen Auszug einer Ontologie, geschrieben in OWL.

```
<owl:Class rdf:ID="Event"/>
<owl:Class rdf:ID="Album"/>
<owl:Class rdf:ID="Instrument"/>
<owl:Class rdf:ID="Musician"/>
<owl:Class rdf:ID="Admirer"/>
<owl:ObjectProperty rdf:ID="author">
        <owl:inverseOf>
                <owl:ObjectProperty rdf:ID="opus"/>
        </owl:inverseOf>
        <rdfs:domain rdf:resource="#Album"/>
        <rdfs:range rdf:resource="#Musician"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="player">
        <rdfs:range rdf:resource="#Musician"/>
        <rdfs:domain rdf:resource="#Instrument"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="loudness">
        <rdf:type
rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
        <rdfs:domain rdf:resource="#Instrument"/>
</owl:ObjectProperty>
```

Bild 1.3

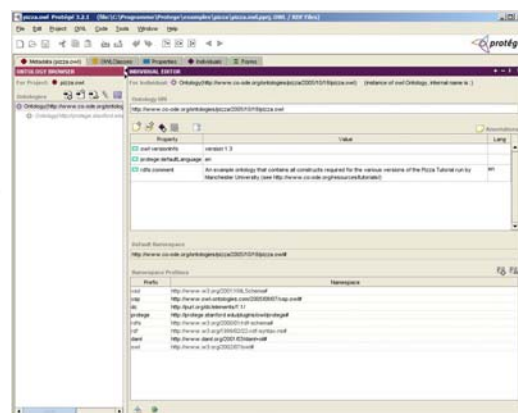


Bild 1.4

2.5.4 Methoden

Eine Entwicklungsmethode für Ontologien beschreibt eine Fülle von Verfahren, Prinzipien und Aktivitäten, welche verwendet werden um eine Ontologie zu entwickeln, zu bewerten und anzuwenden. Eine beste Methode gibt es nicht.

Eine Methode, die den Schwerpunkt auf die Konstruktion von Ontologien legt, schlägt folgende Reihenfolge vor:

1. bestimme die Bereich und den Umfang der Ontologie sowie die beabsichtigte Nutzung
2. untersuche bereits existierende Ontologien, ob diese die gewünschten Anforderungen erfüllen
3. identifiziere wichtige Begriffe
4. definiere Begriffe und eine Hierarchie
5. bestimme Attribute für die Begriffe
6. lege die Definitionsbereiche der Attribute fest
7. erstelle Instanzen der jeweiligen Begriffe und belege deren Attribute mit Werten

Die beschriebenen Schritte sind nicht sehr detailreich. Eine andere, etwas umfangreichere Methode ist das *Methontology Framework*. Es legt zu jeder Phase genaue Verfahren, Standards und Ziele fest. Weiterhin beschäftigt es sich auch mit nebenläufigen Aktivitäten wie Dokumentation, Qualitätssicherung, Bewertung und Produktpflege. Auch auf das Informationsmanagement geht das Framework ein. Es beschreibt verschiedene Techniken zur Informationsgewinnung und wie man daraus eine vorläufige Ontologie entwickelt. Auf die weiteren Zwischenergebnisse, also auf jene Ontologien, wie sie sich am Ende einer jeden Phase ergeben, geht das Framework auch ein. Das Verfahren eignet sich sowohl für die Neuentwicklung als auch für die Weiterentwicklung von bestehenden Ontologien.

Eine andere Kategorie von Methoden nutzt Verfahren, wie sie auch zur Entwicklung von Software eingesetzt werden. Ein Verfahren aus dieser Kategorie schlägt eine analoge Vorgehensweise, wie bei der objektorientierten Softwareentwicklung vor. Die Ähnlichkeiten zwischen beiden Problemstellungen sind unverkennbar: es müssen jeweils ein Konzept, Klassen/Begriffe, eine Hierarchie, Attribute und das Verhalten eines abstrakten Modells festgelegt werden. Auch unterstützen Ontologien Beziehungen wie Vererbung oder Aggregation zwischen Begriffen. Die Möglichkeit Ontologien nach bereits vorgegebenen *Design Patterns* zu konstruieren besteht ebenfalls, wobei eine Ontologie aus einer Ontologienbibliothek im Gegensatz zum *Design Pattern* bereits eine fertige, „gebrauchsfähige“ Ontologie darstellt, was beim *Design Pattern* nicht der Fall ist. Die von den Softwareingenieuren benutzte UML ist auch nichts weiter als eine spezielle Art und Weise eine Ontologie darzustellen. Beide Verfahren haben Vieles gemeinsam, nur haben die einzelnen Prozesse manchmal andere Namen oder die Gewichtung derer ist verschieden.

2.6 Die Über-Ontologie

Eine Ontologie beschreibt ausführlich einen Sachverhalt und legt dabei eine Hierarchie unter den vorhandenen Begriffen fest. Bei Erweiterungen von Ontologien oder,

wenn verschiedene Ontologien verschmelzt werden sollen, kann es nötig sein neue Überbegriffe einzuführen, um vorhandene Begriffe geeignet zu klassifizieren. Die Frage ist hier jedoch, wie weit so etwas gehen kann oder darf. Gibt es also eine Art universellen Wurzelbegriff, aus dem sich allen erdenkbaren anderen Begriffen ableiten lassen oder eine Art *Standard-Über-Ontologie*? In Java erfüllt diese Aufgabe die Klasse *Object*, Protégé dagegen ernennt einen Begriff *Thing* zum Wurzelbegriff. Dies sind jedoch nur technische Lösungen, die eigentliche Fragestellung ist viel komplizierter. Eine Standard-Über-Ontologie müsste alle Themenbereiche abdecken, dürfte nicht zu kompliziert und müsste dennoch jeden Bereich abdecken. Sie müsste also gewissermaßen einen allgemeinen Konsens darstellen, der jedoch enorm schwierig zu finden sein dürfte. Da es aber Bedarf für eine solche Ontologie gibt, hat die IEEE (Institute of Electrical and Electronic Engineers) eine *Standard Upper Ontology Working Group* (SUO WG) gegründet. In ihr forschen Wissenschaftler aus verschiedensten Bereichen nach der *Standard-Über-Ontologie*. Noch befindet sie sich in ihrer Entwicklung, es wird jedoch geschätzt, dass sie zwischen 1000 und 2500 Begriffe mit jeweils bis zu 10 Attributen enthalten wird. Sie wird sich dabei auf generische, abstrakte, beschreibende sowie philosophische Begriffe beschränken und keine themenspezifischen Aspekte enthalten. Als Darstellungssprache soll eine vereinfachte Version von KIF benutzt werden. Sie wird streng nach wissenschaftlichen und weniger nach philosophischen Gesichtspunkten aufgebaut sein, was die Benutzung vereinfachen soll. Der Anwendungsbereich soll enorm breit sein und man hofft, dass sie von der breiten Masse an Nutzern als Basis für weitere speziellere Ontologien verwendet wird. Ein Teil von SUO ist die *Standard-Upper-Merged-Ontology* (SUMO). SUMO ist eine Synthese vieler verschiedener Bereiche oder Module und enthält insgesamt ungefähr 1000 Begriffe. Anwendungsgebiete von SUMO sollen Ontologien sein, welche einen kleinen oder mittelgroßen Bereich abdecken. Sie ist in viele Sprachen übersetzt und frei verfügbar.

3 XML (Sebastian Haffner)

XML (Extensible Markup Language) ist eine Metasprache und ein Standard zur Modellierung von strukturierten Daten.

Im Folgenden wird die Sprache XML beschrieben. In Kapitel 1.1 geht es um den Aufbau der Sprache. Kapitel 1.2 stellt die Möglichkeiten vor, wie einem Nutzer eine Struktur vorgegeben werden kann, um einen einheitlichen Dokumentenaufbau zu erreichen. Das Kapitel „Namensräume“ befasst sich mit einem Problem, was aus der Nutzung von DTDs entstehen kann. In den letzten beiden Kapiteln geht es um die Nutzung und Verarbeitung, so zum Beispiel Ausgabe eines XML-Dokuments oder die Adressierung von gewissen Elementen des XML-Dokuments.

3.1 XML Sprache - Aufbau

Wie in HTML auch, werden Tags zum Aufbau von XML-Dateien genutzt. Der Unterschied liegt darin, dass der Nutzer seine Tags nach seinen Vorstellungen benennen kann, je nach dem wie er es für sein Projekt benötigt. Um jedoch die Wohlgeformtheit eines XML-Dokuments sicherzustellen, müssen gewisse syntaktische Regeln beachtet werden. So darf es nur ein so genanntes Wurzelelement geben, was das gesamte XML-Dokument umschließt. Jedes Tag muss abgeschlossen sein und damit ein Start-Tag und ein End-Tag haben. Ebenfalls darf ein übergeordnetes Tag nicht vor dem ihm untergeordneten Tag geschlossen werden (wie fälschlich im Beispiel <autor><name>Mustermann</autor></name>).

Wie eine HTML-Datei beginnt ein XML-Dokument mit dem Prolog. Darin wird meist definiert, welche XML-Version zur Spezifikation genutzt wird. Wenn es notwendig ist, sind Informationen über die Kodierung des Dokuments enthalten ebenfalls enthalten. Damit ersichtlich ist, dass ein XML-Dokument anhand einer vorgegebenen Struktur entstanden ist, muss im Prolog unbedingt eine DTD oder XML-Schema angegeben werden, und zwar in der Dokumenttyp-Deklaration. Sie kann entweder extern oder intern vorhanden sein. Extern heißt, dass der Ort der DTD oder des XML-Schema definiert ist.

Im Weiteren werden nun die strukturierten Informationen in einem XML-Dokument aufgeführt. Dazu gibt es verschiedene Möglichkeiten, Informationen zu strukturieren. Dadurch, dass man gewisse Tags schachtelt, können Informationen in Form einer Baumstruktur aufgebaut werden.

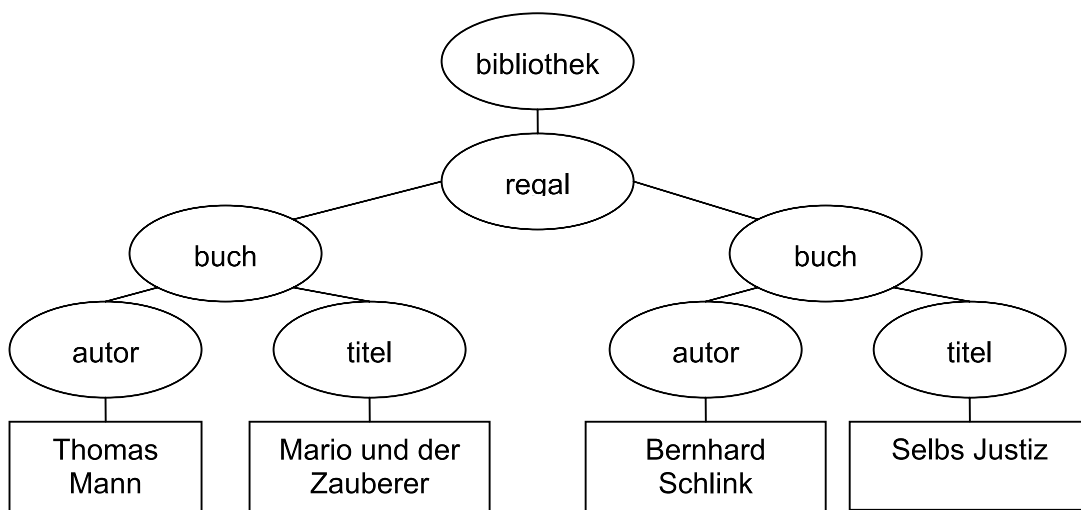
```
<bibliothek>
<regal>
   <buch>
         <titel>Mario der Zauberer</titel>
         <autor>Thomas Mann</autor>
   </buch>
   <buch>
         <titel>Selbs Justiz</titel>
         <autor>Bernhard Schlink</autor>
   </buch>
</regal>
</bibliothek>
```

In dem Beispiel erkennt man den Aufbau einer XML-Datei am Besten. Informationen werden mit so genannten Element-Tags dargestellt (<“name“>…</“name“>). Elemente können auch leer sein, also nur aus einem Tag bestehen (<“name />). Wichtig hierbei ist, dass das erste Zeichen im Tag ein Buchstabe, ein Unterstrich oder ein Doppelpunkt ist und es darf nicht mit „xml“ beginnen. Der Inhalt zwischen dem Start- und End-Tag ist dem Nutzer freigestellt.

Ein leeres Element muss nicht bedeutungslos sein, da man solchen Elementen auch Attribute zuweisen kann. Diese werden dann in dem Empty-Element-Tag oder im Start-Tag definiert (<“element“ name=“…“ alter=“…“/>). Es steht dem Nutzer also frei, ob er seine Informationen mit Hilfe von Element-Tags oder mit Attributen festhält.

Gelegentlich ist es aber auch wünschenswert, in das Dokument beispielsweise Bemerkungen oder Ideen für den weiteren Text einzufügen, die in der formatierten Fassung nicht sichtbar sind. Diesen Text nennt man Kommentar (<?--Kommentar-->). Weiterhin gibt es die Möglichkeit von Verarbeitungsanweisungen, die an gewisse XML-Anwendungen gerichtet sind (<?target instruction ?>).

Ist ein XML-Dokument wohlgeformt, dann kann man die strukturierten Informationen in einem Baummodell darstellen.



3.2 XML strukturieren

Wenn zwei Nutzer getrennt voneinander ein XML-Dokument über das gleiche Thema erstellen sollen, dann werden sich diese Dokumente sicher in vielen Punkten unterscheiden. Das liegt daran, dass der Nutzer bei der Wahl der Tag Namen in XML freie Wahl hat. Wenn nun aber mit diesen Dokumenten gearbeitet werden soll, ist es wünschenswert, dass sie denselben Aufbau und dieselbe Struktur haben. Dafür gibt es nun zwei wesentliche Möglichkeiten: DTDs und XML-Schema, die im Folgenden Näher beschrieben werden. Mit diesen Dokumenttyp-Deklarationen kann man einem Nutzer Vorgaben machen, an welche Struktur er sich beim Aufbau seines XML-Dokuments halten soll.

3.2.1 DTD

Die Komponenten für DTDs können in einer externen Datei oder intern definiert werden. Wenn man die externe Variante nutzt, dann kann man diese für mehrere Dokumente benutzen und ist daher von größerem Nutzen. In DTDs werden Elemente, Attribute von Elementen und Entitäten definiert, so dass ein DTD die Reihenfolge, die Verschachtelung und den Aufbau eines XML-Dokuments bestimmt. Die DTD Syntax hingegen ist kein XML und benötigt daher einen gewissen Mehraufwand an verarbeitenden Programmen. Im Beispiel kann man erkennen, dass ein DTD Konstrukt wie eine Grammatik aufgebaut ist.

```
<! ELEMENT Bibliothek (buch*)>
<! ELEMENT buch (autor+, ISBN)
<! ATTLIST buch
  ISBN ID        #REQUIRED
  ReiheIDREF  #REQUIRED
  Spalte         IDREF  #IMPLIED>
<! ELEMENT Buch (autor+ , auflage)>
<! ELEMENT autor ((name, vorname+) | (vorname+, name))>
<! ELEMENT name (#PCDATA)>
<! ELEMENT vorname (#PCDATA)>
<! ELEMENT auflage EMPTY>
<! ATTLIST auflage
  jahr             CDATA          #REQUIRED
  anzahl                   CDATA          #REQUIRED>
```

In dem Beispiel sind einige Möglichkeiten von DTDs ersichtlich. Elemente werden mit Hilfe von <!ELEMENT …> definiert. In den Klammern können weitere Informationen über das so erstellte Element festgelegt werden. Dazu gehören der Name und die innerhalb des Elements geschachtelten Begriffe, die in den runden Klammern aufgeführt werden. Die Reihenfolge in den Klammern legt auch die Reihenfolge der untergeordneten Elemente fest. Für die Kardinalität von Elementen gibt es drei Ausdrücke: „+" (ein oder mehrere Elemente), „*" (null oder mehrere Elemente) oder „?" (null oder nur ein Element). Kein Kardinalitätsoperator bedeutet, dass es das Element nur einmal gibt. Der Ausdruck „#PCDATA" (Parsed Character Data) steht für irgendeinen Inhalt, den das Element haben muss.

Attribute werden mit Hilfe des <! ATTLIST Elementname…> Befehls zu einem Element hinzugefügt. Eine Attributlisten-Deklaration legt fest, welche Attribute für einen

Elementtyp existieren, von welchem Typ die Attributwerte sind, und ggf. enthält sie einen Vorgabe-Wert (Default). Die Vorgabe Werte sind: #REQUIRED (Das Attribut muss vorkommen), #IMPLIED (Das Attribut kann, muss aber nicht vorkommen), #FI-XED (Jedes Element muss dieses Attribut haben mit dem definierten Wert), „value“ (definiert den Default Wert, den das Attribut hat). Die wesentlichen Attributtypen sind ID, IDREF und CDATA. ID dient dazu, eine eindeutige Identifikation an einem Element anzubringen. Auf so ein Element kann mit einem Attribut vom Typ IDREF verwiesen werden. Attribute mit dem Typ CDATA erwarten eine Zeichenkette als wert, was Zahlen, Buchstaben etc. beinhalten kann.

Dies sind aber nur die Grundbefehle, die nötig sind um eine gewisse XML-Struktur zu definieren. Es soll nur eine kleine Übersicht geben, die DTDs bieten.

3.2.2 XML Schema

XML-Schema bietet bessere Möglichkeiten, XML-Strukturen zu erstellen. XML-Schema nutzt dafür selber die XML-Sprache. Dies ist ein großer Vorteil, denn man muss keine separaten Parser und Editoren schreiben, um es nutzen zu können.

```
<complexType name=”autor”>
 <sequence>
   <element name=”vorname” type=”string” minOccurs=”1” maxOc-
curs=”unbounded”/>
   <element name=”nachname” type=”string”/>
  </sequence>
  <attribute name=”alter” type=”integer” use=”optional”/>
</complexType>
```

In dem Beispiel ist ersichtlich, wie ein XML-Schema aufgebaut ist. Ein Element wird durch <element name=“…“/> definiert. Innerhalb der Klammer kann nun noch festgelegt werden, von welchem Typ das Element ist, sowie dessen Kardinalität. Diese wird durch die Attribute „minOccurs=“x““ und „maxOccurs=“x““ festgelegt. Attribute werden ähnlich mit <attribute name=“…“/> definiert. Auch hier können Typen angegeben werden, sowie, ob das Attribut vorkommen muss oder nicht.

In XML-Schema gibt es wesentlich mehr Möglichkeiten, Datentypen festzulegen. So gibt es wie in anderen Sprachen schon vordefinierte Typen, wie zum Beispiel Strings, Numerische und Datum-Typen. Aber der Nutzer kann sich auch eigene Typen selber erstellen, in dem man die schon vorhandenen Datentypen einschränkt, was im folgenden Beispiel dargestellt ist.

```
<simpleType name=“TagimMonat“>
  <restriction base=“integer“>
    <minInclusive value=”1”/>
    <maxInclusive value=”31”/>
  </restriction>
</simpleType>
```

Die Funktion von Erweiterung und Einschränkung kann man sich wie in der Objektorientierten Programmiersprache vorstellen. Es handelt sich in beiden Fällen um eine gewisse Spezialisierung eines schon vorhandenen Elementes. Die Erweiterung wird mit Hilfe des Ausdrucks <extension base="…"> erreicht. Die „base" gibt an, welches XML-Schema erweitert wird. So wird dann ein neues erweitertes Schema definiert, dass die vorhergehenden Elemente enthält, sowie die neu hinzugefügten. Die Einschränkung erfolgt ähnlich, nur mit der Zeile <restriction base="…">. Einschränkung heißt hier aber nicht das Gegenteil von Erweiterung, dass Elemente oder Attribute gelöscht werden, sondern dass zum Beispiel Kardinalitäten, Attribut Werte oder ähnliches verändert werden.

3.3 Namensräume

Namensräume kennt man eher aus dem Bereich der Objektorientierten Programmierung. In XML ist es eine Möglichkeit, wie man Elmente in XML-Dokumenten bezeichnet, wenn sie aus unterschiedlichen Zusammenhängen (z.B. DTDs) stammen. Im folgenden Beispiel ist ein solcher Zusammenhang ersichtlich. Eine Adresse kann eine Wohnanschrift oder eine Adressierung eines Rechners im Netzwerk sein.

```
<adresse>
  <strasse>Krumme Lanke</strasse>
  <hausnummer>450</hausnummer>
  <plz>98765</plz>
  <ort>Whereever</ort>
</adresse>

<rechner>
  <domain>dieistmeine.de</domain>
  <name>keinzugriff</name>
  <adresse>137.193.138.1</adresse>
</rechner>
```

Will man nun die beiden Adressen in einem Dokument nutzen, so muss sichergestellt werden, dass klar ist, welches Element aus welcher DTD gemeint ist. Dafür wurde ein qualifizierender Name eingeführt, der aus einem Präfix und einem lokalen Namen besteht. Das Präfix bezeichnet den Namensraumteil und der lokale Name besteht aus dem Element- oder Attributnamen, so wie er in der DTD oder dem XML-Schema steht. Beide sind durch einen Doppelpunkt getrennt. Eine Deklaration muss klar machen, dass ein Präfix mit einem bestimmten Namensraum assoziiert ist. Dazu wird das Attribut xmlns benutzt. Im folgenden Beispiel wird dargestellt, dass es sich innerhalb des Adressen-Tags um eine Anschrift handelt.

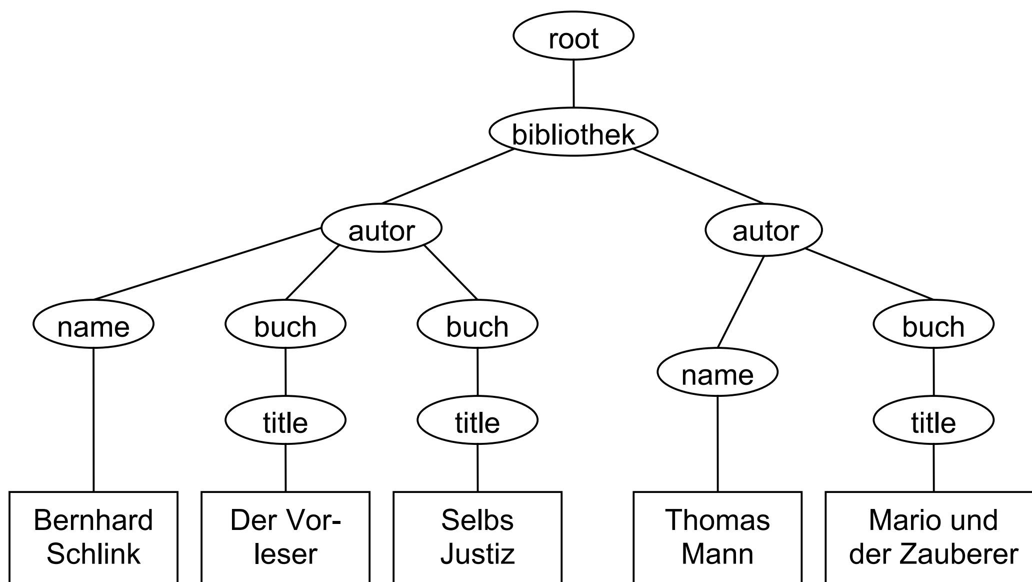
```
<adresse xmlns:anschrift="Ort des DTDs">
  <anschrift:strasse>Krumme Lanke</strasse>
  <anschrift:hausnummer>450</hausnummer>
  <anschrift:plz>98765</plz>
  <anschrift:ort>Whereever</ort>
</adresse>
```

Damit weiß das bearbeitende Programm, dass wenn der Präfix „*anschrift*" davor steht, um welche DTD es sich handelt. Jedoch gilt die Namensraumdeklaration hier nur für den Bereich dieser Adresse. Nutzt man bei der Definition des Namensraumes kein Präfix, so gilt dieser als Default-Namensraum (xmlns="Ort des DTD").

3.4 Adressierung von XML-Elementen

Wie in einer relationalen Datenbank können auch in XML-Dokumenten Teile mit Hilfe von Query Sprachen, wie zum Beispiel XQL, XML-QL oder XQuery herausgefiltert werden. Das zentrale Konzept sind Pfad-Ausdrücke, die angeben, wie man gewisse Knoten erreicht. XPath bietet eine Möglichkeit Teile eines Dokumentes zu adressieren, denn es fasst ein Dokument als Baum beziehungsweise als einen Teilbaum oder Ast auf. Wie in einem Dateisystem lässt sich jedes Bruchstück über einen Pfad oder Verweis wiedergeben.

```
<autor name="Bernhard Schlink">
  <buch title="Der Vorleser"/>
  <buch title="Selbs Justiz"/>
</autor>
<autor name="Thomas Mann">
  <buch title="Mario und der Zauberer">
</autor>
```



Um ein gewissen Einblick in die Möglichkeiten zu bekommen, sind jetzt hier ein paar Beispiele für XPath aufgezählt. Ein Pfad Ausdruck besteht aus einer Serie von Schritten, die durch Slashes getrennt sind. Ein Schritt besteht aus einer Pfadangabe, einem Knotentest und optionalen Filter-Ausdrücken. Dies sind aber nur wenige Beispiele von XPath. Es gibt wesentlich mehr Möglichkeiten.

/bibliothek/autor – Dieser Ausdruck adressiert alle Autoren Elemente. Alternativ kann man bibliothek weglassen (//autor)

/bibliothek/@title="Der Vorleser" – Dieser Ausdruck adressiert alle Titel Knoten, die den Wert „Der Vorleser" haben. Also wird nur der Titel-Knoten adressiert.

//buch [@title="Der Vorleser"] – Im Gegensatz zum Beispiel vorher, werden hier alle Bücher adressiert, die den Titel „Der Vorleser" haben.

/autor [1]/buch [last()] – Hier wird vom ersten Autor das letzte Buch adressiert.

3.5 Verarbeitung von XML-Dateien

Für XML benötigt man, anders als im Falle von HTML, auf jeden Fall Style Sheets. Sonst lässt sich nichts darstellen. Für jeden der Elementtypen in der HTML-DTD wurde festgelegt, wie Instanzen einzelner Elementtypen darzustellen sind. Das ist mit XML-Anwendungen nicht mehr möglich, denn woher soll ein Browser wissen, wie ein Element-Typ in XML dargestellt werden soll. Deswegen werden für das Anzeigen von XML-Dateien Stylesheets benötigt, die auch aus anderen Programmiersprachen kommen können, wie zum Beispiel CSS. Es gibt aber auch die Möglichkeit von XSL (extensible stylesheet language). In ihr sind Transformierungs- und Formatierungsaspekte vereint. Die Transformationssprache XSLT zum Beispiel, kann ein XML-Dokument in ein anderes XML- oder HTML-Dokument (XHTML) transformieren. So wird XSLT genutzt, wenn verschiedene DTDs oder XML-Schemas genutzt werden und mehrere Prozesse mit diesen verschiedenen XML-Dokumenten arbeiten müssen. XSLT kann diese verschiedenen XML-Dokumente in eine einheitliche Struktur bringen, damit alle Prozesse auf der gleichen Struktur arbeiten können.

<u>Eingabe:</u>

```
<autor>
  <name>Thomas Mann</name>
  <buch>Mario der Zauberer</buch>
 </autor>
```

Schablone:

```
<xsl: template match="/autor">
  <html>
    <head><title>Ein Autor</title></head>
    <body>
        <b><xsl: value-of select="name"/></b><br>
        <xsl: value-of select="buch"/><br>
    </body>
  </html>
</xsl: template>
```

Ausgabe:

```
  <html>
    <head><title>Ein Autor</title></head>
    <body>
        <b>Thomas Mann</b><br>
        Mario und der Zauberer<br>
    </body>
  </html>
```

In dem Beispiel wird eine Möglichkeit vorgestellt, mit dessen Hilfe ein XML-Dokument in ein HTML Dokument umgewandelt wird, damit es angezeigt werden kann. XSLT-Dokumente sind selber in XML geschrieben. Die so genannten Schablonen bewirken die gesamte Verarbeitung eines Quelldokumentes zu einem Zieldokument. Sie regeln, welche Elemente aus dem Eingabedokument in die Ausgabe gelangen. Dabei dienen Muster im Attribut „*match*" dazu, zu klären, auf welchen Knoten der Eingabe sich das Template bezieht. Das „value-of" Konstrukt sucht die entsprechenden Informationen aus dem XML Dokument und schreibt dieses an diese Stelle in die Ausgabedatei. Innerhalb von Schablonen kann man andere aufrufen: mit Hilfe von apply-templates. Das kann entweder ein leeres Element sein oder mit Inhalt. Das heißt, man muss nicht für jedes einzelne Element eine Schablone schreiben, sondern alle Kinder eines Elementes werden dann mit dem „apply-template" abgearbeitet.

Dies war die Möglichkeit Elemente zu transformieren und zu formatieren. Für Attribute sieht das ganze ähnlich aus.

```
<xsl:template match="autor">
<person
        vorname="{@vorname}"
        nachname="{@nachname}"/>
</xsl:template>
```

So wie in dem oberen Beispiel beschrieben, können XML-Dokumente auch in neue XML-Dokumente eingearbeitet werden. Anstatt der HTML Tags stehen dann einfach XML-Tags. Das ganze funktioniert ebenfalls mit den „value-of" Funktionen.

4 RDF/RDFS (Christian Heger)

4.1 Einordnung und Abgrenzung

1997 entwickelte das World Wide Web Consortium (W3C) das Resource Description Framework (und wenig später eine „RDF Schema“ oder kurz RDFS genannte Erweiterung) mit dem Ziel, die Informationen des Internets besser maschinenverarbeitbar zu machen. RDF und RDFS sind damit Teil der „Semantic Web layer cake". Darin ordnen sie sich zwischen XML und Ontologien ein:

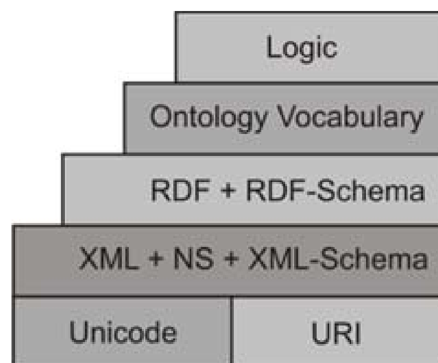


Abbildung 4-1: Die Semantic Web layer cake

RDF ist eine Sprache zur Beschreibung von Semantik. Nun kann Semantik niemals ohne Syntax existieren, in der sie ausgedrückt wird: Deswegen baut RDF auf der XML-Ebene auf (auch wenn wir später sehen werden, dass RDF theoretisch auch vollkommen ohne XML auskäme). Aufbauend auf RDF/RDFS ist die Ontologie-Ebene, in der semantische Zusammenhänge formalisiert werden und schließlich auch formales Schließen erlauben. Die Trennung dieser Ebenen ist eher fließend: RDFS zum Beispiel tendiert bereits in den Bereich der Ontologien hinein.

Nun könnte man einwenden, dass mit XML ja bereits eine Möglichkeit existiert, Daten in Dokumenten maschinenverarbeitbar auszudrücken, und somit für RDF eigentlich keine Notwendigkeit mehr besteht. Dies ist auch richtig. Das Problem ist aber, dass in XML die Information nicht mehr *eindeutig* ausgedrückt wird, d.h. für ein- und dieselbe Information gibt es mehrere Möglichkeiten, sie in XML zu codieren. RDF schafft an dieser Stelle (mehr) Eindeutigkeit, was selbstverständlich für die Computerverarbeitung von höchster Wichtigkeit ist.

4.2 RDF

Wir wollen uns nun eingehender mit dem Resource Description Framework beschäftigen, und einige Grundkonzepte und Besonderheiten dieser Sprache betrachten. Es sei am Rande erwähnt, dass der Begriff Sprache eigentlich ungenau ist, denn strenggenommen handelt es sich bei RDF nicht um eine Sprache, sondern um ein Modell zur Darstellung semantischer Zusammenhänge. Zu einer Sprache wird es erst durch die Anbindung an XML. Die Bezeichnung als Sprache ist jedoch weithin gebräuchlich.

4.2.1 Konzepte

Der Grundbaustein von RDF ist das Statement. Das Statement dient dazu, Information über Objekte auszudrücken. Ein Statement ist ein Tripel, bestehend aus Ressource (resource), Eigenschaft (property) und Wert (value). Solche Tripel sind in der Informatik als OAV-Tripel (Object-Attribute-Value) bekannt. Intuitiv kann man sie mit einem einfachen natürlichsprachlichen Satz der Form Subjekt-Prädikat-Objekt vergleichen.

Im Folgenden werden die drei Bausteine näher erläutert. Betrachten wir dazu die natürlichsprachliche Aussage „Christian Heger wohnt in München." Das zugehörige OAV-Tripel lässt sich mit bloßem Auge erkennen: (Christian Heger, wohnhaft in, München).

1.1.1.1 Ressource (resource)

Etwas informell ausgedrückt ist eine Ressource ein Objekt der Welt, über das eine Aussage gemacht werden soll. Im obigen Beispiel wäre es die Person Christian Heger. Das Objekt muss eindeutig identifizierbar sein, daher besitzt jede Ressource einen Universal Resource Identifier (URI). Dieser kann prinzipiell von nahezu beliebiger Form sein, da RDF jedoch zur Anwendung im Internet gedacht ist liegt die Verwendung von URLs nahe und ist auch in der Praxis bei weitem dominierend. Da jedoch prinzipiell auch andere Formen von URIs möglich sind (im Beispiel wäre etwa die Passnummer denkbar), sei darauf hingewiesen, dass ein URI nicht grundsätzlich auch die Möglichkeit bieten muss, auf die Ressource tatsächlich zuzugreifen!

1.1.1.2 Eigenschaft (property)

Eigenschaften beschreiben eine Ressource. Beispielsweise haben wir bereits die Eigenschaft „wohnt in" betrachtet. Raffinierterweise werden Eigenschaften in RDF ebenfalls als Ressourcen modelliert, insbesondere besitzen sie also einen URI. Dies bietet eine Reihe von Vorteilen. So ermöglicht es etwa, Aussagen über Eigenschaften zu treffen. Dass dies sinnvoll ist, werden wir später anhand von RDFS sehen. Außerdem können Eigenschaften damit ohne Kenntnis der zu beschreibenden Ressource definiert werden. Noch nicht einmal Kenntnis über die *Existenz* der Ressource ist nötig, denn dieses Wissen wird erst dann benötigt, wenn tatsächlich ein Statement getroffen werden soll. Daher kann eine Ressource zusätzliche Eigenschaften erlangen, ohne dabei selbst verändert zu werden (man vergleiche dies mit einer konventionellen objektorientierten Programmiersprache!). Dies ist gerade im Kontext verteilten Wissens, wie es im WWW üblich ist, ein sehr wichtiger Gesichtspunkt.

1.1.1.3 Wert (value)

Der Wert wird der Eigenschaft zugewiesen. Im Beispiel wurde die Eigenschaft „wohnhaft in" mit dem Wert „München" belegt. Ein Wert kann dabei entweder ein Literal sein (d.h. eine Zeichenkette) oder aber eine andere Ressource. Der Fall eines Literals ist, obwohl zweckmäßig, doch von unserem Standpunkt aus eher uninteressant und wird daher im Folgenden vernachlässigt werden. Durch den zweiten Fall jedoch lässt sich ein beliebig dichtes Netz semantischer Querbeziehungen zwischen den Ressourcen aufbauen. Beispielsweise ließe sich der Wert „München" als eigene Ressource modellieren, über welche wiederum Aussagen getroffen werden.

4.2.2 Repräsentationen

Es wurde bereits kurz darauf hingewiesen, dass es sich bei RDF strenggenommen nicht um eine Sprache, sondern um ein Datenmodell handelt. Dies bedingt, dass ein- und dieselbe Aussage auf sehr verschiedene Arten dargestellt werden kann.

1.1.1.4 Darstellung als Tripel

Die Darstellung als OAV-Tripel ist – obwohl streng definitionsgemäß – nur schwer lesbar, mithin ungebräuchlich und daher hier nur der Vollständigkeit halber erwähnt.

1.1.1.5 Darstellung als mathematische Relation

Die Eigenschaft kann aufgefasst werden als mathematische Relation zwischen Ressource und Wert. Auch diese Darstellung hat in der praktischen Anwendung von RDF eigentlich keine Bedeutung. Zur formalen Beschreibung der Semantik von RDF und RDFS jedoch ist diese Darstellung durchaus nützlich, denn solche axiomatischen Definitionen geschehen üblicherweise in der Sprache der Prädikatenlogik.

Auf das Beispiel angewendet erhielte man die wohnt-in-Relation zwischen der Menge der Menschen und der Menge der Städte, und das Tupel (Christian Heger, München) wäre ein Element der Relation.

1.1.1.6 Darstellung als Graph

RDF-Statements können auch als gerichteter Graph mit beschrifteten Kanten dargestellt werden. Solch einen Graph nennt man ein semantisches Netz. Ressourcen werden dabei durch Ellipsen repräsentiert, Literale durch Rechtecke. Eigenschaften werden durch Kanten dargestellt: Genau dann führt eine Kante von einem Knoten zu einem anderen, wenn das Ziel der Wert einer Eigenschaft der Quelle ist. Der Name der Eigenschaft wird an die Kante geschrieben.

Abbildung 4-2: Das Beispiel in Graphendarstellung

Die Graphendarstellung ist die für Menschen mit Abstand am einfachsten verständliche Darstellung und daher die vom W3C für die Entwicklung empfohlene Vorgehensweise. Unglücklicherweise eignen sich Graphen aber nicht besonders für die Verarbeitung und Speicherung durch Maschinen. Hierfür muss der Graph in die im nächsten Abschnitt besprochene XML-Notation überführt werden.

1.1.1.7 Darstellung in XML

Um die Statements besser maschinenverarbeitbar zu machen hat man eine auf XML aufbauende Sprache entwickelt. Dies bietet den Vorteil, dass bestehende Webseiten bequem mit semantischer Information annotiert werden können, ohne dass hierfür neue Infrastruktur (wie etwa ein neues Dokumentenformat o. Ä.) geschaffen werden müsste. Auf die genauen syntaktischen Einzelheiten dieser Sprache soll an dieser Stelle nicht im Detail eingegangen werden – dies würde den gegebenen Rahmen

sprengen und entspricht auch nicht der Zielsetzung dieses Dokuments. Einige allgemeine Beobachtungen zur XML-Darstellung sind aber dennoch angebracht. Um die Lesbarkeit zu steigern, wird dazu im Folgenden der Begriff „RDF“ synonym mit „RDF in XML-Darstellung“ verwendet.

Um die Lesbarkeit zu erhöhen, erlaubt RDF es, mehrere Statements zu einer sogenannten Beschreibung (Description) zusammenzufassen. Die Reihenfolge der Statements innerhalb der Beschreibung spielt dabei jedoch ebenso wenig eine Rolle wie die Reihenfolge der Beschreibungen innerhalb des Dokuments. Descriptions können außerdem ineinandergeschachtelt werden. Die Descriptions der inneren Schachtelungsebenen besitzen dabei ebenfalls globale Sichtbarkeit!

Zur Zusammenfassung mehrerer Ressourcen zu einer Gesamtheit bringt RDF zudem einige rudimentäre Collection-Klassen mit. Eine Collection ist ihrerseits wieder eine Ressource. Damit wird es möglich, Aussagen über mehrere Ressourcen zugleich zu treffen. Neue Funktionalität wird dadurch jedoch nicht eingeführt: Genauso gut könnte die entsprechende Aussage für jedes Element der Collection separat getroffen werden. Es erleichtert nur das Arbeiten mit der Sprache.

Ein wesentlicher Aspekt ist dagegen die Verwendung von Namespaces. Durch Angabe externer Dokumente werden dabei alle in diesem Dokument definierten Elemente importiert, wie beispielsweise Ressourcen, Properties, Statements und Descriptions. Einerseits ist dies schlicht der Bequemlichkeit zuträglich: Statt beispielsweise übersichtlich `rdf:Description` schreiben zu können, müsste sonst die vollständige URI (URL) angegeben werden, und zwar bei jeder Verwendung:

```
http://www.w3.org/1999/02/22-rdf-syntax-ns#Description.
```

Darüber hinaus hat die Verwendung von Namespaces aber noch weiter reichende Konsequenzen. Sie ermöglicht nämlich die verteilte Speicherung von Information, und macht Information damit gewissermaßen wiederverwendbar. Information, die bereits irgendwo beschrieben ist, kann importiert werden und muss nicht erneut definiert werden. Alles andere wäre für eine Internetsprache auch absolut inakzeptabel. Die angestrebte Vision ist ein Netz verteilten Wissens.

In der XML-basierten Form sähe das bisher betrachtete Beispiel so aus:

```
<rdf:RDF
     xmlns:rdf=”http://www.w3.org/1999/02/22-rdf-syntax-ns#”>

<rdf:Description rdf:about=”muenchen”/>

<rdf:Property rdf:ID=”wohntIn”/>

<rdf:Description rdf:about=”cHeger”>
  <wohntIn rdf:resource=”muenchen”/>
</rdf:Description>

</rdf:RDF>
```

In der ersten Zeile wird signalisiert, dass nun RDF-Daten folgen. Dies ist notwendig, da RDF-Annotationen ja innerhalb von HTML-Dokumenten erfolgen können, und da-

her Anfang und Ende von RDF-Bereichen markiert werden müssen. Direkt danach wird der Namespace, der die RDF-Definitionen enthält importiert. Anschließend wird eine Eigenschaft „wohntIn“ und eine Ressource mit URI (!) „muenchen“ und keinen weiteren Attributen erstellt. Schlussendlich wird eine Ressource mit der URI „cHeger“ erstellt, und der Eigenschaft wohntIn der Wert „muenchen“ zugewiesen. Diese Zeile ist das eigentliche Statement. Man beachte dabei, dass der Wert nicht etwa die Zeichenkette „muenchen“ ist, sondern die Ressource mit URI „muenchen“! Dies ist ein wichtiger Unterschied, denn über die Ressource „muenchen“ können ihrerseits Statements existieren. Am Ende wird signalisiert, dass der RDF-Abschnitt nun beendet ist.

Wie man sieht ist auch diese Darstellung für den Menschen noch relativ gut lesbar. Keinesfalls ist sie jedoch so intuitiv erfassbar wie die graphische Darstellung: Der nötige Erklärungsumfang ist deutlich angewachsen.

4.2.3 Reifikation

Der Ausdruck Reifikation (oft auch als Reifizierung bezeichnet) stammt aus dem Lateinischen und bedeutet „zu einer Sache machen“. Die „Sache die gemacht wird“ ist in diesem Falle das Statement, welches objektifiziert wird. Dies erlaubt es, Aussagen über Aussagen zu treffen. Auf RDF übertragen heißt das: Durch Reifikation wird ein Statement zu einer Ressource.

Wir hatten bereits die Aussage „Christian Heger wohnt in München“ betrachtet. Mittels Reifikation ist es nun möglich, eine Aussage *über diese Aussage* zu treffen. Eine solche Aussage wäre zum Beispiel „Christian Szymkowiak glaubt, dass Christian Heger in München wohnt.“ Das ganze sähe dann in RDF so aus:

```
<rdf:Statement rdf:ID=”reifiziertesStatement”>
  <rdf:subject rdf:resource=”cHeger”/>
  <rdf:predicate rdf:resource=”wohntIn”/>
  <rdf:object rdf:resource=”muenchen”/>
</rdf:Statement>


<rdf:Description rdf:about=”szymkowiak”>
  <glaubtDass rdf:Resource=”#reifiziertesStatement”/>
</rdf:Description>
```

Die Deklaration der Eigenschaft glaubtDass wurde hier weggelassen, um das Beispiel nicht unnötig aufzublähen.

4.3 RDF Schema

RDFS ist die auf RDF aufbauende, nächsthöhere Ebene. RDFS dient dazu, das Vokabular zu beschreiben welches in RDF verwendet wird. Das Problem mit dem man sich in RDF relativ schnell konfrontiert sieht ist nämlich folgendes: RDF sieht so gut wie keine Typisierung vor – maximal sind gewisse primitive Datentypen vorgesehen, die aus der XML-Spezifikation „angeliehen“ werden. Dabei kann jedoch kaum von echter Typisierung gesprochen werden: Eigentlich wird nämlich nur eine Art Interpre-

tationsvorschlag gemacht, der Wert an sich ist nach wie vor einfach eine Zeichenkette.

Dies ist aber bei weitem zu wenig um der Struktur von RDF gerecht zu werden, die ja auf Ressourcen beruht, welche zunächst einmal ununterscheidbar sind hinsichtlich ihres Typs. Um dieses Dilemma einmal drastisch zu veranschaulichen: Die Aussage „München wohnt in Christian Heger“ wäre in RDF ein vollkommen legitimes Statement! Allerdings ist es semantisch offensichtlich vollkommen unsinnig. Die Abhilfe ist, zu spezifizieren, dass nur Personen in Städten wohnen können, und nicht umgekehrt. Auf dieser Ebene setzt nun RDFS an.

4.3.1 Konzepte

RDFS basiert auf einem recht simplen Klassenkonzept mit Mehrfachvererbung. Sowohl Ressourcen als auch Eigenschaften werden durch Klassen ausgedrückt. Wie bereits erwähnt sind Eigenschaften in RDF spezielle Ressourcen, es sollte daher nicht verwundern dass die Klasse `rdf:Property` eine Unterklasse von `rdfs:Resource` ist.

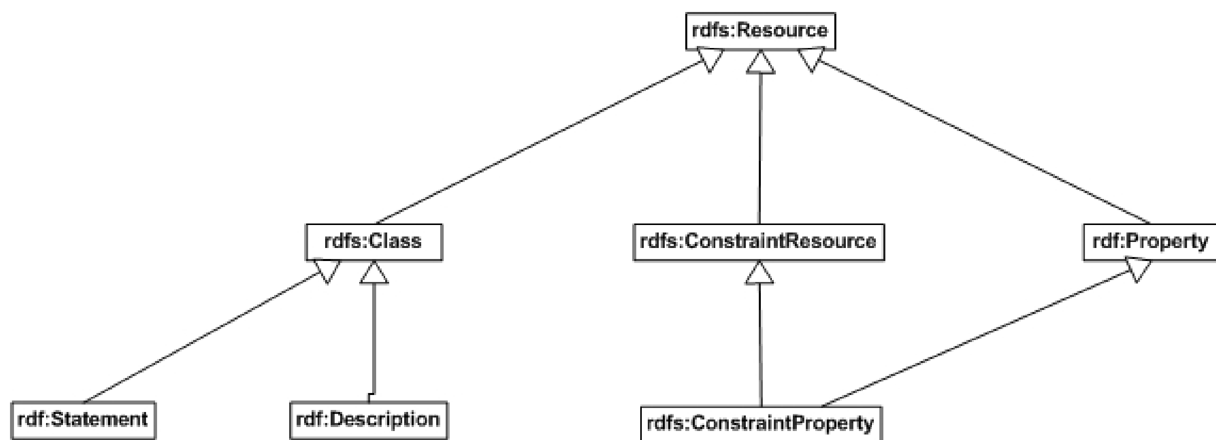


Abbildung 4-3: Klassenhierarchie von RDFS

Wenn man sich dieses Diagramm einmal ansieht, so mag auffallen, dass man es mit einem gerichteten Graphen zu tun hat – als Kantenbeschriftung könnte man implizit „ist Unterklasse von“ annehmen. In der Tat stellt das Vererbungsdiagramm doch semantische Zusammenhänge dar: Nämlich die Vererbungsbeziehung der Klassen untereinander. Daraus folgt, dass man diese Beziehungen in RDF ausdrücken kann. RDFS wird also selbst in RDF beschrieben, indem Klassen als Ressourcen und die (allgemeine) Vererbungsbeziehung durch eine „ist-Unterklasse-von“-Eigenschaft modelliert werden, und die konkreten Verbindungen durch entsprechende Statements hergestellt werden. Insbesondere benötigt RDFS also keine neue Infrastruktur.

4.3.2 Klassen und Eigenschaften

Mittels der RDFS Syntax ist es nun möglich, eigene Klassen und Eigenschaften zu definieren. Eine Unterklassenbeziehung stellt man dabei durch Verwendung der `rdfs:subClassOf`-Eigenschaft her. Analog kann man Eigenschaften definieren. RDFS sieht zudem auch für Eigenschaften eine Hierarchie vor, d.h. man kann Eigenschaften als Nachkommen anderer Eigenschaften deklarieren. Das Schlüsselwort

hierfür ist `rdfs:subPropertyOf`. Dies ist immer dann der Fall, wenn eine Eigenschaft eine andere verfeinert oder präzisiert. Formal ausgedrückt (wir verwenden die Darstellung als mathematische Relation): Wenn P und Q Eigenschaften sind, so ist P Subproperty von Q genau dann, wenn für alle x und y gilt: $P(x, y) \Rightarrow Q(x, y)$. Beispielsweise wäre die Eigenschaft „ist Kind von“ eine Untereigenschaft von „ist verwandt mit“. Dieses Konzept ist konsistent mit der Tatsache, dass auch Eigenschaften Ressourcen sind.

Nachdem man Eigenschaften derart definiert hat, kann man sie selbstverständlich ebenfalls wieder mit Eigenschaften versehen. Zwei spezielle Eigenschaften sind `rdfs:range` und `rdfs:domain`. Mittels dieser beiden Eigenschaften ist es nun möglich, den Definitionsbereich (englisch domain) und Wertebereich (englisch range) von Eigenschaften einzuschränken.

Betrachten wir erneut das Beispiel mit der Verwandtschaftsrelation. In RDF/RDFS modelliert stellt sich das ganze so dar:

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdfs:Class rdf:ID="person">
  <rdfs:comment>
  Eine Klasse um Personen zu modellieren
  </rdfs:comment>
</rdfs:Class>

<rdf:Property rdf:ID="verwandtMit">
  <rdfs:domain rdf:resource="#person"/>
  <rdfs:range rdf:resource="#person"/>
</rdf:Property>

<rdf:Property rdf:ID="kindVon">
  <rdfs:subPropertyOf rdf:resource="#verwandtMit"/>
</rdf:Property>
</rdf:RDF>
```

Zunächst wird eine Klasse Person angelegt. (Exemplarisch wurde an dieser Stelle demonstriert, wie Kommentare eingefügt werden können.) Anschließend definiert man die Eigenschaft verwandtMit und spezifiziert, dass als Definitions- wie auch als Wertebereich nur Instanzen der Klasse (oder Nachkommen der Klasse) Person erlaubt sind. Schließlich wird die Untereigenschaft kindVon definiert. Indem sie als Untereigenschaft von verwandtMit deklariert wird, erbt sie deren Eigenschaften – also insbesondere Definitions- und Wertebereich.

Auch in RDFS ist es beachtenswert, dass Eigenschaften und Klassen weitgehend unabhängig voneinander definiert werden können. Klassen können also auch nachträglich noch zusätzliche Eigenschaften erhalten, ohne dass dazu die Klassendefini-

tion geändert werden oder auch nur bekannt sein müsste. Es muss noch nicht einmal die Existenz der Klasse bekannt sein (dies unterbindet jedoch aus offensichtlichen Gründen die Verwendung als Definitions- oder Wertebereich).

4.3.3 Semantik von RDFS

Auch wenn RDFS in RDF ausgedrückt wird, so gibt es doch einen wesentlichen Unterschied: Während an die semantischen Inhalte die mit RDF ausgedrückt werden von der Sprache selbst keine Anforderungen gestellt werden (lediglich die Bedeutung der Schlüsselworte wie z.B. resource oder property ist fest), ist die Semantik der in RDFS definierten Klassen und Eigenschaften (wie z.B. `rdfs:subClassOf`) fest definiert und verbindlich. Erst durch die Einhaltung dieser Festlegung erlangen RDFS-Dokumente überhaupt eine Bedeutung – denn die Bedeutung der Eigenschaft `rdfs:subClassOf` ist in ihrer Deklaration nirgends festgelegt und kann es auch in der Tat gar nicht sein – dies lässt RDF nicht zu! (Wäre RDF mächtig genug, diese Bedeutung auszudrücken bräuchte man ja auch RDFS gar nicht.) Dadurch, dass die Bedeutung der definierten Elemente aber verbindlich festgelegt ist ergibt sich der Übergang in den Bereich der Ontologien. Daher bezeichnet man RDFS auch als eine (zugegebenermaßen recht primitive) Ontologiebeschreibungssprache. Unter anderem wird es damit für Computer auch möglich, logisch zu schließen: Mit den Definitionen aus dem Verwandtschaftsbeispiel und der Aussage „Christian Heger ist Kind von Gabi Heger“ könnte ein Computer nun folgern, dass auch die Aussage „Christian Heger ist verwandt mit Gabi Heger“ zutrifft.

Eine befriedigende Beschreibung von Ontologien ist jedoch mit RDFS nicht möglich. Beispielsweise beinhaltet RDFS keinerlei Konzept von Äquivalenz bezüglich Klassen oder Eigenschaften. Auch die Möglichkeiten hinsichtlich des formalen Schließens sind eher begrenzt. Daher gibt es, aufbauend auf der RDF/RDFS-Ebene, höhere Ontologiesprachen, um eine bessere Beschreibung zu ermöglichen.

4.4 Beispiel

Es folgt ein etwas ausführlicheres Beispiel, welches sowohl RDF als auch RDFS beinhaltet. Im Wesentlichen setzt es sich aus den bereits besprochenen Beispielen zusammen, und komplettiert diese wo nötig.

```
<rdf:RDF
  xmlns:rdf=”http://www.w3.org/1999/02/22-rdf-syntax-ns#”
  xmlns:rdfs=”http://www.w3.org/2000/01/rdf-schema#”>

<rdfs:Class rdf:ID=”person”/>
<rdfs:Class rdf:ID=”stadt”/>

<rdf:Property rdf:ID=”verwandtMit”>
  <rdfs:domain rdf:resource=”#person”/>
  <rdfs:range rdf:resource=”#person”/>
</rdf:Property>

<rdf:Property rdf:ID=”kindVon”>
  <rdfs:subPropertyOf rdf:resource=”#verwandtMit”/>
```

```
</rdf:Property>

<rdf:Property rdf:ID="wohntIn">
  <rdfs:domain rdf:resource="#person"/>
  <rdfs:range rdf:resource="#stadt"/>
</rdf:Property>


<rdf:Property rdf:ID="glaubtDass">
  <rdfs:domain rdf:resource="#person"/>
  <rdfs:range rdf:resource="&rdf;Statement"/>
</rdf:Property>

<rdf:Description rdf:ID="muenchen">
  <rdf:type rdf:resource="#stadt"/>
</rdf:Description>

<rdf:Description rdf:ID="gHeger>
  <rdf:type rdf:resource="#person"/>
</rdf:Description>

<rdf:Description rdf:ID="cHeger">
  <rdf:type=rdf:resource="#person"/>
  <sohnVon rdf:resource="#gHeger"/>
  <rdf:Statement rdf:ID="reifiziertesStatement">
    <rdf:subject rdf:resource="#cHeger"/>
    <rdf:predicate rdf:resource="#wohntIn"/>
    <rdf:object rdf:resource="#muenchen"/>
</rdf:Description>

<rdf:Description rdf:ID="cSzymkowiak">
  <glaubtDass rdf:resource="#reifiziertesStatement"/>
</rdf:Description>

</rdf:RDF>
```

4.5 Quellenangaben

1. G. Antoniou, F. van Harmelen: „A Semantic Web Primer“, MIT Press 2004, S. 79-107
2. D. Gasevic, D. Djuric, V.Devedzic: “Model Driven Architecture and Ontology Development”, Springer 2006
3. “RDF/XML Syntax Specification (Revised)”, Stand: 17. Januar 2007, http://www.w3.org/TR/rdf-syntax-grammar/

5 OWL – Web Ontology Language (Sebastian Frenzel)

5.1 OWL im Semantic Web layer-cake

Die nächste Stufe des Semantic Web sind Ontologien. Ontologien dienen im Allgemeinen der Wissensrepräsentation. Auch im Semantic Web soll Wissen für Systeme (hier Rechensysteme) nutzbar gemacht werden. Dazu wird eine Sprache benötigt, die ein größeres Ausdrucksspektrum als RDF und RDFS besitzt.

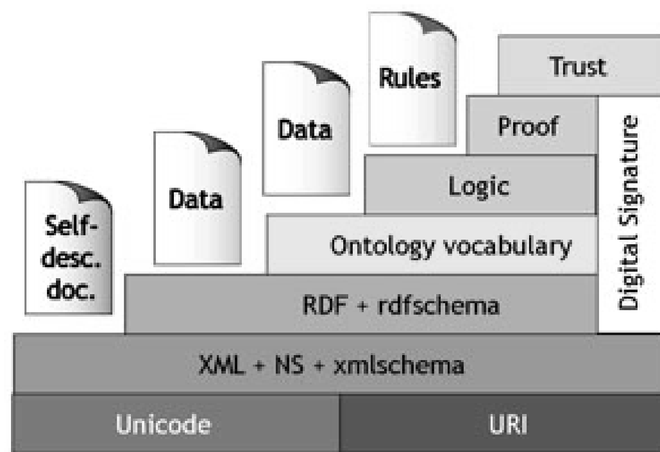


Abbildung 5-1: Semantic Web layer-cake

5.1.1 Anforderungen

Was benötigt eine solche Sprache bzw. welche Voraussetzungen muss sie erfüllen, um Wissen für Rechensysteme effektiv darzustellen? Die wichtigste Grundlage ist eine eindeutige Syntax , denn nur so können die zu verarbeitenden Informationen korrekt ausgewertet werden. Damit verbunden ist eine genaue Interpretation um die richtige Bedeutung aus dieser Analyse zu ziehen. Es soll vermieden werden, dass dieselbe Information von zwei verschiedenen Interpreten unterschiedlich gedeutet wird. Die formale Semantik ist eine Voraussetzung für die Beweisunterstützung (*reasoning support*), die es erlaubt die Ontologien auf Folgerichtigkeit und resultierende Erkenntnisse zu überprüfen. Eine Automatisierung dieses Vorgang ermöglicht eine wesentlich effizientere Prüfung.

5.1.2 RDF/RDFS als Grundlage

Wie man im Semantic Web layer-cake (vgl. Abb.: 5-1) sehen kann soll die Sprache auf dem Resource Description Framework aufbauen, das in seiner Ausdrucksmächtigkeit begrenzt ist. So fehlt es an Möglichkeiten Disjunktion verschiedener Klassen auszudrücken (z.B. rote Kugeln und gelbe Kugeln), Klassen zu vereinen und Schnittstellen zu bilden (z.B. aus roten und gelben Kugeln die Klasse der farbigen Kugeln bilden), Eigenschaften/Einschränkungen mit Kardinalitäten zu versehen (z.B. eine Klasse Behälter in die genau 12 rote Kugeln und 8 gelbe Kugeln passen) und Properties mit Eigenschaften zu versehen (z.B. Symmetrie auf die Property *hat_selbes_Gewicht*). Dies sind nur einige der Möglichkeiten, die von einer solchen Sprache erwartet werden, um wirklich aussagekräftige Ontologien zu erstellen.

5.1.3 DAML+OIL

Der Vorgänger von OWL ist DAML+OIL, der sich aus DAML-ONT (DARPA Agent Markup Language – Ontology) und OIL (Ontology Inference Layer) zusammensetzt. Die Zusammenführung beider Sprachen im Dezember 2000 ergab DAML+OIL. DAML+OIL baut (wie DAML) auf RDF und RDF/S auf. Damit sollte erreicht werden, dass in auf RDF basierenden Anwendungen mit DAML+OIL-ausgezeichneten Seiten so wenige Fehler/Unstimmigkeiten wie möglich auftreten. Die Syntax von DAML+OIL ist daher in RDF/S formuliert. Die Elemente von DAML+OIL sind Klassen, Klassenkonstrukte, Properties (ObjectProperty und DatatypeProperty), Einschränkungen und Instanzen. Die Funktionen dieser Elemente werden im OWL-Kapitel geschildert.

5.2 Einführung in OWL

OWL bietet mehr Ausdrucksmöglichkeiten für Bedeutung und Semantik als XML und RDF(S). OWL ist eine Revision von DAML+OIL, die vom „World Wide Web Consortium“ (W3C) spezifiziert wurde. Aus Sicht des W3C musste die Sprache neu spezifiziert werden, da OWL über mehr Details für Ontologien verfügen sollte. Am Beispiel einer Ontologie über afrikanische Wildtiere, soll im Verlauf dieser Ausarbeitung, die einfache Notation dieser Sprache beschrieben werden. Dieses Beispiel stammt aus „A Semantic Web Primer“ (s. Quellenangaben) und wurde für diese Einführung angepasst und ergänzt. Zuerst eine Übersicht über die geplanten Klassen:

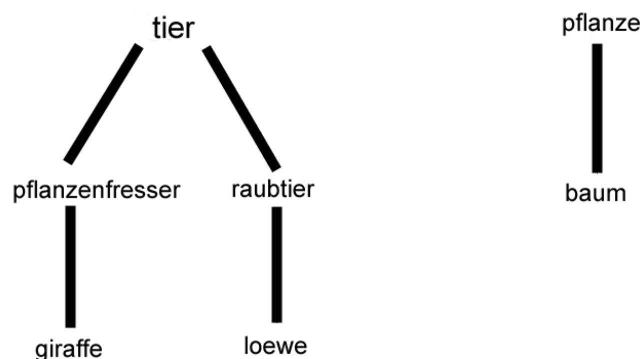


Abbildung 5-2: Die ersten Klassen der Wildtiere

Die drei Sprachen von OWL

OWL unterstützt drei zunehmend ausdrucksstarke Untersprachen, welche aufeinander aufbauen und für ihre speziellen Verwendungen durch unterschiedliche Benutzer ausgewählt werden. Diese Sprachen sind OWL Lite, OWL DL (Description Language) und OWL Full.

OWL Lite unterstützt die üblichen Modellierungselemente wie Klassifikation, Hierarchien und einfache Nebenbedingungen. „Einfache Nebenbedingungen“, da die Kardinalität einer Nebenbedingung nur 1 oder 0 annehmen darf. Man kann also nur Aussagen treffen wie *hat_Kugeln* und *hat_Keine_Kugeln* aber nicht, wie viel davon (Aussagen wie *hat_18_Kugeln* behalten ihre Gültigkeit). Dieses Grundgerüst ist ein-

facher zu handhaben, da durch die geringe formale Komplexität eine Vielzahl von Tools (Editoren, Validator, Reasoner) unterstützt wird.

OWL DL ist für jene Nutzer gedacht, die eine größtmögliche Ausdrucksstärke erreichen wollen und dabei auf Vollständigkeit (alle Schlüsse sind garantiert „berechenbar“) und Entscheidbarkeit (alle „Rechnungen“ in endlicher Zeit durchführbar) Wert legen. Es werden alle Sprachkonstrukte zur Verfügung gestellt, jedoch sind einzelnen Verwendungen bestimmter Einschränkungen auferlegt (zum Beispiel darf eine Klasse nicht Instanz einer anderen Klasse sein). Diese Sprache erhält den Zusatz Beschreibungslogik, da sich mit ihr prädikatenlogische Aussagen erster Stufe treffen lassen.

OWL Full richtet sich an Benutzer, die sich nicht nur die größtmögliche Ausdrucksstärke, sondern auch die syntaktischen Freiheiten von RDF(S) wünschen. Dabei nimmt man Unentscheidbarkeit in Kauf, kann allerdings prädikatenlogische Aussagen höherer Grade zu treffen.

5.2.1 Die Sprachbeschreibung von OWL Lite

Nachfolgend werden die Besonderheiten von OWL Lite erklärt. Dazu werden jeweils die Syntax, die Funktion und ein Beispiel kurz beschreiben. Für die genaue Definition der einzelnen Befehle wird in den Quellenangaben auf die OWL Referenz verwiesen.

Individual: Individuen sind die Instanzen von Klassen. **Class:** Eine Klasse definiert eine Gruppe von Instanzen, die zusammengehören, da sie sich einige Eigenschaften teilen. So sind *Löwe* und *Tiger* Instanzen der Klasse *raubtier.* Klassen können in Hierarchien organisiert werden. Die Syntax dazu ist **rdfs:subClassOf:** Die nachfolgende Klasse ist eine Unterklasse, die vorhergehende Klasse wird zur Oberklasse. So ist *baum* Unterklassen von *pflanze.* In OWL gibt es die Klasse *Thing*, welche die Klasse aller Instanzen und die Oberklasse aller Klassen ist. Das Gegenteil ist die Klasse *Nothing*. Diese Klasse hat keine Instanzen und ist die Unterklasse aller Klassen. Damit haben wir die Klasse *Tier* und *Pflanze* eingeführt, und den *Baum* als Unterklasse der Pflanze.

```
<owl:Class rdf:ID="tier">
 <rdfs:comment>'tier' ist damit eine neue Klasse</rdfs:comment>
</owl:Class>

<owl:Class rdf:ID="pflanze">
</owl:Class>

<owl:Class rdf:ID="baum">
  <rdfs:comment>
   Ein Baum ist eine Pflanze und somit Unterklasse von 'pflanze'
  </rdfs:comment>
 <rdfs:subClassOf rdf:resource="#pflanze"/>
</owl:Class>
```

Abbildung 5-3: Auszug aus der Wildtier-Ontologie

rdf:Property: Eigenschaften können Aussagen zu einer Beziehung zwischen Instanzen machen. Solche Eigenschaften sind zum Beispiel *ist_Teil_von*, *frisst* oder *gefressen_von*. Das sind so genannte ObjectProperties. Damit lassen sich für unsere Wildtiere die ersten Properties einführen und somit eine erste Nahrungskette andeuten (s. Abbildung 5.5). Es können auch Beziehungen zu einer Instanz eines Daten-

typs gemacht werden: *wiegt_kg* ordnet der Klasse ein Gewicht zu (einen Integerwert). Das ist eine DatatypeProperty. Die **owl:ObjectProperty** und **owl:DatatypeProperty** sind damit Unterklassen von rdf:Property. Das bedeutet, dass auch hier Hierarchien existieren, die sich mit **rdfs:subPropertyOf:** bilden lassen. Wenn es die Property *arbeiten_in_selber_Firma* gibt, kann man davon ausgehen, dass *ist_Chef_von* eine Subproperty davon ist. Wenn die Subproperty erfüllt ist, ist auch die übergeordnete Eigenschaft erfüllt.

```
<owl:ObjectProperty rdf:ID="ist_Teil_von"/>
<owl:ObjectProperty rdf:ID="frisst"/>
<owl:ObjectProperty rdf:ID="gefressen_von"/>
```

Abbildung 5-4: Auszug aus der Wildtier-Ontologie

Doch sind diese Eigenschaften längst nicht aussagekräftig, man muss sie beschränken können, da ansonsten Verknüpfungen wie *Giraffe frisst Loewe* zulässig sind.

rdfs:domain: Der Definitionsbereich (die Domäne) einer Property beschränkt die Instanzen, auf die die Property angewendet werden kann. Wird die Property *frisst* auf die Klasse *tier* beschränkt, ist *tier* der Definitionsbereich von *frisst*. So kann man aus *Loewe frisst Giraffe* sehen, dass *Loewe* zur Klasse *tier* gehört.

```
<owl:ObjectProperty rdf:ID="frisst">
 <rdfs:domain rdf:resource="#tier"/>
</owl:ObjectProperty>
```

Abbildung 5-5: Auszug aus der Wildtier-Ontologie

rdfs:range: Dieser Bereich einer Property beschränkt die Instanzen, welche die Property als Wert bekommt. Wird der Wertebereich (Range) der Property *frisst* auf die Klasse *pflanze* beschränkt, ist *pflanze* der Wertebereich von *frisst*. So kann man aus g*iraffe frisst blatt* sehen, dass *blatt* zur Klasse *pflanze* gehört. In unserer Ontologie ist dies aber nicht zielführend, da man mehr als nur Pflanzen fressen darf.

Des Weiteren lassen sich in OWL Lite Aussagen über die Gleichheit von Klassen bzw. Eigenschaften treffen. In unserer kleinen Ontologie könnte man zum Beispiel *raubtier* und *fleischfresser* gleich setzen. Das geschieht mit **equivalentClass:** Wenn zwei Klassen identisch sind, also die selben Instanzen haben, kann man mit diesem Befehl einer Klasse ihrem Äquivalent zuordnen. **equivalentProperty:** Auch Eigenschaften können gleichbedeutend sein, was hier bedeutet, dass die beiden Eigenschaften die gleichen Arten von Instanzen verbinden. Diese Regel gilt auch umgekehrt. Wenn zwei Instanzen identisch sind, so wird dies mit **sameAs:** festgelegt. Es macht Sinn mehrere Instanzen so zu verknüpfen, wenn mit verschiedenen Bezeichnungen, dieselbe Instanz angesprochen werden soll.

```
<owl:Class rdf:ID="fleischfresser">
 <owl:equivalentClass rdf:resource="#raubtier"/>
</owl:Class>

<!-- Die Instanz Sandloewe (der Klasse Loewe), reagiert nun auch auf
     Wuestenloewe -->

<loewe rdf:ID="wuestenloewe">
 <owl:sameAs rdf:resource="#sandloewe" />
</loewe>

<!-- Mit AllDifferent grenze ich die Instanzen (der Einfachheit
     wegen diesmal direkt von der Klasse Raubtier) voneinander
     ab -->

<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <raubtier rdf:about="#loewe" />
    <raubtier rdf:about="#tiger" />
    <raubtier rdf:about="#hyaene" />
  </owl:distinctMembers>
</owl:AllDifferent>
```

Abbildung 5-6: Auszug aus der Wildtier-Ontologie

differentFrom: Sollen sich zweit Instanzen unterscheiden, so muss das mit diesem Befehl gesetzt werden. Zum Beispiel sind die Raubtiere *Loewe* und *Tiger* voneinander verschieden, doch weiß nicht jeder Interpreter, dass dies Bezeichnungen für unterschiedliche Individuen sind. Man kann sie ohne zusätzliche Informationen nicht auseinander halten. Mit **AllDifferent:** wird die Aussage über mehrere Individuen gleichzeitig getroffen. Somit werden die Raubtiere Loewe*, Tiger* und *Hyaenen* in einer Aussage voneinander unterschieden. Setze ich *Loewe* mit differentFrom *Tiger* und Hyaene von diesen beiden Instanzen ab, unterscheiden sich nur *Loewe* und *Tiger*, sowie *Loewe* und *Hyaene*. Mit AllDifferent unterscheiden sich dann auch *Tiger* und *Hyaene*.

Die nachfolgenden Bezeichner betreffen die Properties, um diese zusätzlich mit Informationen zu versehen.

inverseOf: Hiermit wird ausgesagt, dass eine Property das Gegenteil/die Inverse zu einer anderen Property ist. Das bedeutet, dass die inverse Eigenschaft in umgekehrter Richtung gilt, wenn zwei Klassen mit einer Eigenschaft verknüpft sind. Die Eigenschaft *frisst* ist das Gegenteil zu *gefressen_von*.

TransitiveProperty: Transitivität lässt sich auch in OWL Lite für Eigenschaften festlegen. Wenn eine Eigenschaft transitiv ist und die Instanzen A und B, sowie B und C mit dieser Eigenschaft verbunden sind, so sind auch A und C damit verbunden. In OWL Lite/DL dürfen transitive Properties (und deren Subproperties) keine maxCardinality 1 Restriktion haben. Ansonsten werden die beiden Sprachen unentscheidbar.
SymmetricProperty: Wird eine Eigenschaft als symmetrisch bezeichnet, gilt die Eigenschaft in beide Richtungen. Wenn man *hat_gleiche_Fellfarbe_wie* auf *Wuestenloewe* und *Sandloewe* anwendet, dann ist diese Aussage mit S*andloewe hat_selbe_Fellfarbe_wie Wuestenloewe* identisch. Die Eigenschaft und ihre Inverse sind somit gleichbedeutend.

```
<owl:TransitiveProperty rdf:ID="ist_Teil_von"/>

<owl:ObjectProperty rdf:ID="frisst">
 <rdfs:domain rdf:resource="#tier"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="gefressen_von">
 <owl:inverseOf rdf:resource="#frisst"/>
</owl:ObjectProperty>
```

Abbildung 5-7: Auszug aus der Wildtier-Ontologie

FunctionalProperty: Es können Eigenschaften festgelegt werden, die nur einen expliziten oder keinen Wert besitzen dürfen. Diese Eigenschaft ist dann eine FunctionalProperty, sie kann nicht mehr als einen Wert haben oder auch gar keinen Wert. FunctionalProperty ist eine verkürzte Schreibweise für die Festlegung, dass die minimale Kardinalität der Eigenschaft 0 und die maximale Kardinalität 1 ist. Zum Beispiel ist *sieht_Vollmond* eine solche Eigenschaft, da die Erde nur einen *Mond* hat.

InverseFunctionalProperty: Mit dieser Festlegung wird ausgesagt, dass die Inverse zu einer funktionalen Eigenschaft ebenso funktional ist. Zum Beispiel kann ein *US-Buerger* nur eine *Sozialversicherungsnummer* haben. Und jede *Sozialversicherungsnummer* kann maximal einem *US-Buerger* zugeordnet werden, muss aber nicht automatisch zugeordnet sein (es sind niemals alle Nummern vergeben).

Nun werden die Bezeichner erklärt, die Einschränkungen festlegen, wie Properties durch Instanzen verwendet werden dürfen. Die Syntax dafür ist **owl:Restriction,** mit dem **owl:onProperty** wird die eingeschränkte Eigenschaft festgelegt.

allValuesFrom: Mit dieser Anweisung beschränkt man die Instanzen im Wertebereich einer Eigenschaft auf bestimmte Klassen. Also können die Instanzen für die Werte der Property nur aus den erlaubten Klassen kommen. Wenn man also sagen möchte: *Blatt ist_Teil_von Ast* , kann man die Eigenschaft *ist_Teil_von* auf Instanzen der Klassen *Ast* beschränken. Anders ist es bei **someValuesFrom:** Diese Einschränkung sagt aus, dass nicht alle Instanzen des Wertebereichs aus den angegebenen Klassen stammen müssen, aber mindestens eine Instanz. Mit diesen Bezeichnern können wir nun das Futter weiter spezifizieren:

```
<owl:Class rdf:ID="ast">
 <rdfs:comment>
  Aeste sind Teile von Baeumen. Mit allValuesFrom wird festgelegt,
  dass die Eigenschaft 'is_Teil_von' nur auf Instanzen der
  Klasse 'baum'
  zeigt
 </rdfs:comment>
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#ist_Teil_von"/>
   <ow:allValuesFrom rdf:resource="#baum"/>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>

<!-- Blatt wird analog als Teil von Ast definiert-->
```

Abbildung 5-8: Auszug aus der Wildtier-Ontologie

minCardinality: Mit dieser Bezeichnung wird festgelegt, welche Kardinalität eine Eigenschaft mindestens haben muss, also wie viele Werte/Instanzen mindestens im Wertebereich liegen müssen. Dabei wird die Aussage jedoch nur für eine Klasse im Range gemacht. Dieselbe Property kann in einer anderen Klasse eine andere minimale Kardinalität haben. Ein Beispiel: Die Klasse *Vater* ist eine Unterklasse von *Mann*, die wiederum eine Unterklasse von *Mensch* ist. Die Eigenschaft *hat_Nachkommen* erhält für *Mensch* die minimale Kardinalität 0 (ist damit optional) , genau wie *Mann*. *Vater* jedoch erhält die minimale Kardinalität 1, da jeder Vater mindestens einen Nachkommen hat. Anders lässt sich mit **maxCardinality** die maximale Kardinalität festlegen. Es darf also maximal die vorgegebene Anzahl von Werten/ Instanzen im Wertebereich liegen. Für *Mensch, Mann* und *Vater* ist die maximale Kardinalität der Eigenschaft *hat_Vater* also 1. In diesem Fall ist in allen drei Klassen auch die minimale Kardinalität dieser Eigenschaft 1, die beiden Kardinalitätsbeschränkungen sind somit gleich. Dies kann man mit cardinality auch kürzer schreiben. Anmerkung: In OWL Lite sind nur minimale und maximale Kardinalitäten von 0 oder 1 erlaubt.

Wenn wir aber zu unserer Wildtier-Ontologie als Beispiel zurückkehren, haben wir dort noch nicht die Raubtiere und Fleischfresser definiert. Es wäre geschickt, wenn man diese Klassen nur aus der Klasse *tier* und Einschränkungen in deren Futterwahl kombinieren könnte. Raubtiere lassen sich so noch mit **intersectionOf** als Schnittmenge darstellen. In dieser gebildeten Klasse werden die Einschränkungen übernommen:

```
<owl:Class rdf:ID="raubtier">
 <rdfs:comment>
  Raubtiere werden anders definiert, dass diese ja durchaus auch
  Pflanzen essen koennen, und nicht nur andere Tiere. Zuerst wird
  wieder eine neue Klasse aus einer Kombination mittels
  intersectionOf erstellt.
 </rdfs:comment>
 <owl:intersectionOf rdf:parsetype="Collection">
  <owl:Class rdf:about="#tier"/>
   <rdfs:comment>
    Die Einschraenkung wird diesmal mit someValuesFrom
    getroffen. Dies besagt, dass mindestens eine Instanz der
    Klasse 'tier' im Nahrungsangebot des Tieres enthalten sein
    muss. Damit koennen Raubtiere nun Tiere und Pflanzen essen.
   </rdfs:comment>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#frisst"/>
   <owl:someValuesFrom rdf:resource="#tier"/>
  </owl:Restriction>
 </owl:intersectionOf>
</owl:Class>
```

Abbildung 5-9: Auszug aus der Wildtier-Ontologie

Doch für schwierigere Kombinationen oder Aussagen über Mengen ist OWL Lite dann schon nicht mehr geeignet.

5.2.2 Die Sprachbeschreibung von OWL DL/Full

OWL DL stehen die kompletten erweiterten Sprachkonstrukte von OWL Full zur Verfügung. Allerdings unterliegt OWL DL einigen Einschränkungen: OWL DL verlangt Typenunterscheidung. Das bedeutet, dass eine Klasse nicht zeitgleich eine Eigenschaft oder Instanz sein kann und eine Eigenschaft nicht zeitgleich Instanz oder Klasse sein darf. Dies ist nur in OWL Full gestattet, wodurch erlaubt wird, Einschrän-

kungen auf die Sprachelemente von OWL selbst anzuwenden. Des Weiteren muss in OWL DL eine Property entweder ObjectProperty oder DatatypeProperty sein. Die Nutzung beliebiger (positiver) Intergerwerte für Kardinalitätsbeschränkungen ist in beiden Sprachen gestattet.

oneOf: Durch diesen Bezeichner können Enumerationen aufgestellt werden, es werden also die Instanzen aufgelistet, welche die Klasse bilden. Von der Klasse können nur diese Instanzen gebildet werden. Wenn wir also in unserer Ontologie die *Raubtiere* auf eine eine bestimmte Auswahl beschränken wollen, gehen wir so vor:

```
<owl:Class rdf:ID="raubtier">
  ... <!-- s. Oben --->
  <owl:oneOf rdf:parseType="Collection">
    <raubtier rdf:about="#loewe" />
    <raubtier rdf:about="#hyaene" />
    <raubtier rdf:about="#tiger" />
  </owl:oneOf>
</owl:Class>
```

Abbildung 5-10: Auszug aus der Wildtier-Ontologie

hasValue: Von Eigenschaften kann verlangt werden, dass sie eine bestimmte Instanz als Wert annehmen. Damit wird auf die Instanz aus einer Klasse und nicht nur auf die Klasse beschränkt.

Bisher haben wir die Klassen *pflanze* und *tier* definiert, ohne sie richtig zu trennen. Mit **disjointWith:** können Klassen als disjunkt voneinander betrachtet und definiert werden. Also ergänzen wir:

```
<owl:Class rdf:ID="pflanze">
 <rdfs:comment>
  Pflanzen sollen sollen keine gemeinsamen Elemente erhalten, hier
  kommt ein OWL DL/Full Bezeichner: disjointWith
 </rdfs:comment>
 <owl:disjointWith="#tier"/>
</owl:Class>
```

Abbildung 5-11: Auszug aus der Wildtier-Ontologie

Man kann schlussfolgern, dass ein Individuum als Instanz von *pflanze* keine Instanz von *tier* sein kann.

Noch viel mächtiger sind die **Boolean combinations** (boolsche Kombinationen). Mit diesen Bezeichnern kann man Klassen miteinander kombinieren und neue Aussagen treffen. Mit **unionOf** kann man festlegen, dass eine Klasse entweder Elemente der Klasse A oder der Klasse B enthält. **complementOf** sagt aus, dass die Klasse A alle Elemente enthält, die Klasse B nicht enthält.

Die complex classes sollen an dieser Stelle nur kurz beschrieben werden. In OWL Lite wird die Syntax auf einfache Klassennamen eingeschränkt. In OWL Full ist es möglich, diese Einschränkungen um komplexe Klassenbeschreibungen zu erweitern. Als solche wurden aufgeführt: Enumerationen, Einschränkungen von Eigenschaften und die boolschen Kombinationen (inklusive intersectionOf). Letztendlich erlaubt uns OWL DL unsere Ontologie zu vervollständigen (zumindest die *Pflanzenfresser* zu definieren und die in Abbildung 5-2 gezeigten Klassen aufzuführen).

```
<owl:Class rdf:ID="pflanzenfresser">
 <rdfs:comment>
  Pflanzenfressers sind genau die Tiere, die nur Pflanzen und
  Teile von Pflanzen essen. Mit intersectionOf wird die Klasse
  'tier' und einer Einschraenkung kombiniert.
 </rdfs:comment>
 <owl:intersectionOf rdf:parsetype="Collection">
  <owl:Class rdf:about="#tier"/>
  <owl:Restriction>
   <rdfs:comment>
    Die Einschraenkung bezieht sich auf die Property 'frisst'. Mit
    allValuesFrom wird wieder die Zielmenge beschraenkt.
   </rdfs:comment>
   <owl:onProperty rdf:resource="#frisst"/>
   <owl:allValuesFrom>
    <rdfs:comment>
     unionOf sagt, dass die Zielmenge nur aus Pflanzen oder aus
     Instanzen von Klassen bestehen darf, welche Teile von
      Pflanzen sind.
    </rdfs:comment>
    <owl:unionOf rdf:parsetype="Collection">
     <owl:Class rdf:about="#pflanze"/>
     <owl:Restriction>
      <owl:onProperty rdf:resource="#ist_Teil_von"/>
      <owl:allValuesFrom rdf:resource="#pflanze"/>
     </owl:Restriction>
     </owl:unionOf>
   </owl:allValuesFrom>
  </owl:Restriction>
 </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID="giraffe">
 <rdfs:comment>
  Giraffen sind Pflanzenfresser, fressen aber nur Blaetter!
 </rdfs:comment>
 <rdfs:subClassOf rdf:type="#pflanzenfresser"/>
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#frisst"/>
   <owl:allValuesFrom rdf:resource="#blatt"/>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

Abbildung 5-12: Auszug aus der Wildtier-Ontologie

5.3 Quellenangaben

G. Antoniou, F. van Harmelen:
A Semantic Web Primer, MIT Press 2004. S. 109 – 149

D. Gasevic, D. Djuric, V. Devedzic:
Model Driven Architecture and Ontology Development, Springer 2006. S. 79 – 107

OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns

OWL Web Ontology Language Overview – Stand: 10. Februar 2004
http://www.w3.org/TR/owl-features/

OWL Web Ontology Language Guide – Stand: 22. März 2004
http://www.w3.org/TR/owl-guide/

OWL Web Ontology Language Reference – Stand: 12. Oktober 2005
http://www.w3.org/TR/owl-ref/

DAML+OIL (March 2001) Language Release – Stand: 27. März 2001
http://www.daml.org/2001/03/daml+oil-index.html

Description of OIL – Stand: 30. November 2000
http://www.ontoknowledge.org/oil/

6 Logik und Folgerungen: Regeln (Blanquett, Geier)

6.1 Einleitung

Es geht nun darum wie man Wissen, welches im Internet bereitgehalten wird, logisch auswerten kann. Wie ist es möglich aus logisch aufgebautem Wissen neue Folgerungen, also neues Wissen abzuleiten? Als Grundlage dient uns die Prädikatenlogik. Aus folgenden Gründen bietet sich diese an:

- Man kann präzise von einer logischen Aussage auf die andere schließen.
- Die Prädikatenlogik beinhaltet Beweissysteme, mit denen man Aussagen von einer gegebenen Menge von Prämissen ableiten kann.
- Prädikatenlogik ist in dem Sinne einzigartig als dass es ein widerspruchfreies und komplettes Beweissystem bereithält. (Auf einem höheren Level angesiedelte Logiken haben solche Beweissysteme nicht).
- Wegen des Beweissystems kann man jede Aussage zurückverfolgen und erkennen, auf welcher vorher getroffenen logischen Annahme die Aussage beruht.

Die Prädikatenlogik kann mit den bereits vorgestellten Sprachen RDF und OWL abgebildet werden. RDF und OWL eignen sich besonders für das „Semantic Web“, da sie eine Syntax bereitstellen, die den im Web üblichen Tags entspricht. Eine weitere wichtige Begründung ist, dass sie eine sinnvolle Untermenge der Logik darstellen. Eine weitere Untermenge ist die so genannte Horn Logik (auch Regelsystem).

Eine Regel in diesem System hat folgende Syntax:

$A_1, \ldots, A_n \rightarrow B$

Darin sind A_i und B atomare Formeln. Es gibt nun zwei intuitive Wege, wie man diese Regel lesen kann.

1. Wenn bekannt ist, dass $A_1, \ldots A_n$ wahr sind, dann ist B auch wahr. Man spricht dann von deduktiven Regeln.
2. Wenn die Bedingungen $A_1, \ldots A_n$ wahr sind, dann führen wir die Aktion B aus. Bei diesen Regeln spricht man von reagierenden Regeln.

Beide Sichtweisen bieten wichtige Anwendungsmöglichkeiten. Jedoch werden wir im Folgenden nur den deduktiven Ansatz verfolgen. Wir werden die Horn Logik (auch *Regelsystem*) genau betrachten und aufzeigen welche Fragestellungen sich aus ihrem Einsatz ergeben, und gleichzeitig passende Antworten geben.

Allerdings reicht diese Logik nicht für alle Fälle aus. Wir können mit der Horn Logik zum Beispiel folgenden Sachverhalt darstellen: Ein Onlineshop möchte immer dann einen speziellen Rabatt gewähren, wenn der Kunde Geburtstag hat. Wir würden diese Gesetzmäßigkeit wie folgt darstellen:

R1: Wenn Kunde Geburtstag hat, dann Rabatt

R2: Wenn Kunde nicht Geburtstag hat, dann keinen Rabatt

Diese Lösung funktioniert wunderbar, aber nur solange, wie das Geburtsdatum des Kunden auch bekannt ist. Wenn nun aber der Kunde aus Datenschutzgründen sein

Geburtsdatum nicht bekannt geben möchte, dann können diese Regeln nicht angewandt werden, weil die Prämissen (die Voraussetzungen) nicht bekannt sind. Es liegt kein Geburtsdatum vor, also können die Regeln für keinen Vergleich herangezogen werden. Wir müssten die Regeln also wie folgt abändern:

R1: Wenn Kunde Geburtstag hat, dann Rabatt

R2': Wenn kein Geburtstag oder der Geburtstag nicht bekannt ist,
dann keinen Rabatt

Die Prämisse aus Regel 2' kann nicht mit der Prädikatenlogik ausgedrückt werden. Wir benötigen also für solche Fälle ein neues *Regelsystem*. Wir merken uns, dass wir die zuerst genannten beiden Regeln immer dann anwenden können, wenn uns alle Informationen vorliegen (z. B.: Kunde hat Geburtstag, oder aber er hat nicht). Das neue *Regelsystem*, also die beiden zuletzt genannten Regeln, finden da Anwendung, wo die vorliegenden Informationen unvollständig sind.

Prädikatenlogik und die davon abgeleitete Horn Logik ist im folgenden Sinne **monoton**: Wenn wir eine Schlussfolgerung ziehen, dann bleibt sie auch dann bestehen, wenn weitere Informationen hinzukommen.

Betrachten wir nun R2'. Es wird die Schlussfolgerung getroffen, dass der Rabatt nicht gewährt wird, wenn das Geburtsdatum unbekannt ist. Wird dieses aber zu einem späteren Zeitpunkt im Bestellvorgang bekannt, und es stellt sich heraus, dass der Kunde heute Geburtstag hat, dann trifft die Schlussfolgerung nicht mehr zu. Wir sprechen dann von **nichtmonotonen** Regeln, im Unterschied zu den **monotonen** Regeln.

6.2 Monotone Regeln

6.2.1 Beispiel

Wir haben eine fiktive Datenbank, die Informationen über Familienbeziehungen bereithält. Wir nehmen an, es sind Fakten über folgende Prädikate gespeichert:

Mutter(X, Y)	X ist Mutter von Y
Vater(X, Y)	X ist Vater von Y
Männlich(X)	X ist männlich
Weiblich(X)	X ist weiblich

Daraus lassen sich nun weitere Aussagen ableiten, wenn wir die richtigen Regeln anwenden. Als erstes können wir das Prädikat Elternteil definieren.

Mutter(X,Y) → Elternteil(X,Y)

Vater(X,Y) → Elternteil(X,Y)

Und daraus definieren wir jetzt einen Bruder. Ein Bruder ist eine männliche Person, die sich mit jemand anderen ein Elternteil teilt.

Männlich(X) , Elternteil(P,X) , Elternteil(P,Y), unterschiedlich(X,Y) → Bruder(X,Y)

Das Prädikat *unterschiedlich* stellt die Ungleichheit der beiden Personen sicher. Wir gehen davon aus, dass die Unterschiedlichkeit zweier Personen aus der Datenbank ausgelesen werden kann. Natürlich bietet jedes logische System bequemere Wege an, um auf Gleichheit oder Ungleichheit zu testen, doch hier ist es gewusst abstrakt gehalten.

Eine Schwester wird nun wie folgt definiert:

Weiblich(X) , Elternteil(P,X) , Elternteil(P,Y), unterschiedlich(X,Y) → Schwester(X,Y)

Ein Onkel ist der Bruder eines Elternteils:

Bruder(X,P), Elternteil(P,Y) -> Onkel(X,Y)

Eine Großmutter ist die Mutter eines Elternteils

Mutter(X,P), Elternteil(P,Y) -> Großmutter(X,Y)

usw.

So lassen sich aus 4 gegebenen Prämissen weitere Prämissen folgern, und es kann immer genau nachvollzogen werden, auf Grund welcher vorherigen Annahme (siehe Anfang).

6.2.2 Syntax

Beispiel:

treuerKunde(X), Alter(X) > 60 →Rabatt(X)

Alle treuen Kunden über 60 sollen einen Rabatt eingeräumt bekommen. Anhand dieses Beispiels werden wir die Syntax erläutern.

Wir unterscheiden folgende Grundelemente:

- *Variablen*, das sind Platzhalter für Werte: X
- *Konstanten*, das sind feste Werte: 60
- *Prädikate*, welche Beziehungen zwischen Objekten darstellen: treuerKunde, >
- *Funktionen*, welche einen Wert zu einem übergebenen Argument zurückgeben: Alter

Regeln

Regeln sind wie folgt aufgebaut:

$B_1, \dots, B_n \rightarrow A$

A, $B_1, \dots, B_n$ sind atomare Formeln. A ist der Kopf (head) der Regel, und $B_1, \dots, B_n$ sind die Prämissen. Die Menge $\{B_1, \dots, B_n\}$ wird Körper (body) genannt. Die Kommata in der Regel werden Konjunktiv gelesen, also als ein verbindendes „Und“ interpretiert. Wenn B_1 und B_2 und… und B_n wahr sind, dann ist auch A wahr. Zu beachten

ist dass, wie im obigen Beispiel, auch Variablen in den atomaren Formeln auftauchen können.

Fakten

Ein Fakt ist eine atomare Formel, wie z.B. treuerKunde(a345678); dies besagt, dass der Kunde mit der ID a345678 treu ist.

Logische Folgen

Eine logische Folge P umfasst eine endliche Menge von Fakten und Regeln. Die prädikatenlogische Übersetzung pl(P) ist die Menge aller möglichen prädikatenlogischen Darstellungen der Regeln und Fakten in P.

Ziel (Goal)

Ein Ziel kennzeichnet eine Anfrage an eine logische Folge. Es hat die Form

$$B_1, \ldots, B_n \rightarrow$$

Wenn n=0 ist, dann sprechen wir vom *leeren Ziel*.

6.2.3 Semantik

Prädikatenlogische Semantik

Eine logische Folge P sei gegeben. Folgende Anfrage sei zu P gegeben:

$$B_1, \ldots, B_n \rightarrow$$

mit den Variablen $X_1, \ldots, X_k$. Wir beantworten die Anfrage genau dann positiv, wenn, und nur dann wenn

$$pl(P) \models \exists X_1 \ldots \exists X_k (B_1 \wedge \ldots \wedge B_n) \quad (1)$$

Genauso gilt aber umgekehrt auch:

$$pl(P) \cup \{\neg \exists X_1 \ldots \exists X_k (B1 \wedge \ldots \wedge Bn)\} \text{ ist unerfüllbar} \quad (2)$$

Mit anderen Worten: Die Aussage ist genau dann Wahr, wenn die Vereinigung der prädikatenlogischen Interpretation der logischen Folge P vereinigt mit der prädikatenlogischen Interpretation der Anfrage die leere Menge ergibt (Die Aussage also unerfüllbar ist).

Die Interpretation der Prädikatenlogik als solche sei hier kurz wiederholt: Ein prädikatenlogisches Modell ***A*** weist eine eindeutige Bedeutung zu. Ein Modell ***A*** besteht aus

- einem Definitionsbereich def(***A***), der eine nichtleere Menge von Objekten enthält, über die die Formeln aussagen treffen können.
- einem Element aus dem Definitionsbereich von ***A*** für jede Konstante.
- einer konkreten Funktion aus dem Definitionsbereich von ***A*** für jedes Funktionssymbol.
- einer konkreten Relation aus dem Definitionsbereich von ***A*** für jedes Prädikat.

Die Bedeutung der logischen Junktoren $\neg$, $\vee, \wedge, \rightarrow, \forall, \exists$ sind definiert, gemäß ihrer intuitiven Bedeutung. Nämlich als Nicht, Oder, Und, Impliziert, Für alle, Es gibt.

Wenn wir sagen wollen, dass eine Formel im Modell ***A*** wahr ist, schreiben wir $\mathbf{A} \models \varphi$.

Jetzt können wir die beiden Beispiele (1) und (2) erklären. Egal wie wir die in P und der Anfrage auftauchenden Konstanten, Prädikate und Funktionssymbole interpretieren, sobald die prädikatenlogische Interpretation von P wahr ist, ist

$$\exists X_1 \ldots \exists X_k (B_1 \wedge \ldots \wedge B_n)$$

ebenfalls wahr.

Beispiel:

P sei folgende logische Folge

p(a)

p(X) → q(X)

Dazu die Anfrage

q(X) →

q(a) folgt aus pl(P), wegen $\exists$ Xq(X) folgt aus pl(P). Also

ist pl(P) $\cup$ {$\neg\exists$ Xq(X)} unerfüllbar. Wir geben also eine positive Antwort zurück.

Wenn wir jedoch die Anfrage

q(b) →

haben, müssen wir eine negative Antwort zurückgeben, weil q(b) nicht aus pl(P) folgt.

Grundterme und Parametrisierte Terme

Bis jetzt haben wir unser Augenmerk nur auf Anfragen gelegt, die mit ja oder nein beantwortet werden können. Dies reicht jedoch nicht immer aus.

Beispiel:

Wir haben den Fakt

p(a)

und die Anfrage

p(X)

Die Anfrage wird wahr, also mit ja beantwortet. Dies hilft jedoch nicht wirklich weiter. Es erinnert an den Witz, wo man gefragt wird: „Weißt du wie viel Uhr es ist?“, und du schaust auf deine Uhr und antwortest mit „Ja“. In unserem Beispiel wäre eine angemessene Antwort eine Ersetzung (Substitution)

{X/a}

was eine Instantiierung für X darstellt, und eine positive Antwort zurückgibt. Die Konstante a wird Grundterm genannt.

Sind z.B. die Fakten

p(a)

p(b)

gegeben, dann haben wir zwei Grundterme für die gleiche Anfrage. Wir würden also folgende Ersetzung zurückgeben

{X/a}

{X/b}

Um solche Anfragen nützlich zu gestalten, sind Grundterme nicht immer die optimale Antwort. Gehen wir von folgender logischen Folge aus:

add(X, 0, X)

add(X, Y, Z) → add(X,s(Y),s(Z))

Diese logische Folge berechnet eine Addition. Wir lesen s als die „Nachfolger-Funktion", welches den Wert des übergebenen Arguments +1 zurückgibt. Das dritte Argument von add berechnet die Summe der ersten beiden Argumente. Ausgehend von der Anfrage

add(X,s^8(0),Z) →

lassen sich verschiedene Grundterme ermitteln. So z.B.

{X/0, Z/s^8(0)}

{X/s(0), Z/s^9(0)}

{X/s(s(0)), Z/s^{10}(0)}

Der Parametrisierte Term Z = s^8(X) ist jedoch der am allgemeinsten gefasste, um die Anfrage

$\exists X \exists Z$ add(X, s^8(0), Z)

zu erfüllen.

6.3 Nichtmonotone Regeln

Bis jetzt konnten wir davon ausgehen, dass sobald alle Voraussetzungen in einer Regel (also der Body) erfüllt sind, auch der Kopf (Head) als gültig abgeleitet werden kann. Wenn aber neue Informationen hinzukommen, können unter Umständen in anderen Regeln die Prämissen erfüllt werden. In dem Fall kann es zu Konflikten kommen, wenn die Aussagen, die in den Köpfen der Regeln stehen, sich widersprechen oder gegenseitig ausschließen. Um diese besondere Situation handhaben

zu können, bedienen wir uns eines neuen Regelsystems. Dies ist das so genannte nichtmonotone Regelsystem. Darin gelten die Regeln als **anfechtbar**. Das heißt, dass wir, obwohl alle Prämissen der Regel erfüllt sind, nicht automatisch schließen können, dass auch der Kopf gilt, denn dort könnte das Gegenteil des Kopfes einer anderen Regel, die auch erfüllt ist, stehen. Es ist in solchen Fällen zunächst nicht möglich, eine eindeutige Aussage zu treffen, welcher Kopf gilt. Damit ist es für automatisierte Prozesse schwierig, Ausdrücke korrekt auszuwerten.

Dieser Fall ist jedoch in der Praxis weit verbreitet und daher ist es sinnvoll, eine Lösung für dieses Problem zu finden. Dazu werden sowohl im Body als auch im Head zusätzlich Negationen von atomaren Formeln zugelassen, damit die Konflikte auch in der Sprache der Logik ausgedrückt werden können. Ein kleines Beispiel dazu wäre:

$p(X) \rightarrow q(X)$
$r(X) \rightarrow \neg q(X)$

Wenn wir nun annehmen, dass p(X) und r(X) gelten, dann kommt es zu einem Konflikt, da q(X) und ¬q(X) nicht gleichzeitig gelten können.

Um den entstanden Konflikt bei den Regeln aufzulösen, müssen zusätzlich **Prioritätsbedingungen** festgelegt werden. Diese ergeben sich meistens aus dem praktischen Umfeld bei der Anwendung. So kann zum Beispiel eine Regel eine stärkere Gewichtung haben, weil die Quelle verlässlicher oder aktueller ist oder mehr Autorität besitzt (Bundesgericht gegenüber Landesgerichten etc.). Auch kann eine Regel eine allgemeingültige Aussage treffen und eine andere stellt eine Ausnahme dar. Dann ist in solchen Fällen die Ausnahme stärker zu gewichten. Wichtig ist, dass die Prioritätsbedingungen alle Regeln, bei denen es zu Konflikten kommen kann, in eine feste und eindeutige Reihenfolge bringen.

Um Konflikte zwischen Literalen zu beschreiben, gibt es für jedes Literal ein **Konfliktfeld**. Darin ist mindestens die Negation des Literals enthalten und gegebenenfalls noch weitere, die sich aus der Praxis ergeben. Im einfachsten Fall (nur mit der Negation) ergibt sich: $C(L) = \{\neg L\}$.

6.3.1 Beispiel

Ein Beispiel zeigt, wie nichtmonotones Schließen in einer Semantic-Web-Anwendung nutzbringend eingesetzt werden könnte.

> Carlos ist auf der Suche nach einer mindestens 45m² großen Mietwohnung, die wenigstens zwei Schlafzimmer hat. Sollte sich die Wohnung im zweiten Stock oder höher befinden, muss es einen Fahrstuhl geben. Außerdem sollen Haustiere in der Wohnung nicht verboten sein. Er ist bereit, für eine Wohnung, die den Anforderungen entspricht, 400 € Miete zu zahlen. Falls mehrere Wohnungen in Frage kommen, würde er die günstigste nehmen. Sollten zwei akzeptable Wohnungen die gleichen Mietkosten haben, nimmt er die Größere.

Eine Wohnung kann mit Hilfe der folgenden Prädikate beschrieben werden:

size(x, y)	Wohnung x ist y m² groß
bedrooms(x, y)	Wohnung x hat y Schlafzimmer
floor(x, y)	Wohnung x liegt im Stockwerk y
lift(x)	Wohnung x hat einen Fahrstuhl

pets(x) Haustiere sind in der Wohnung x erlaubt
price(x, y) Wohnung x kostet y € Miete
acceptable(x) Wohnung x entspricht Carlos’ Vorstellungen

Nun lassen sich die Bedingungen, die Carlos an eine Wohnung stellt, in Regeln ausdrücken:

r_1: ⇒ acceptable(x)
r_2: bedrooms(x, y), $y<2$ ⇒ ¬acceptable(x)
r_3: size(x, y), $y<45$ ⇒ ¬acceptable(x)
r_4: ¬pets(x) ⇒ ¬acceptable(x)
r_5: floor(x, y), $y>2$, ¬lift(x) ⇒ ¬acceptable(x)
r_6: price(x, y), $y>400$ ⇒ ¬acceptable(x)

Nach r_1 ist jede Wohnung akzeptabel. Die Regeln r_2 bis r_5 stellen die Bedingungen von Carlos dar. Sobald eine dieser Bedingungen zutrifft, kommt die Wohnung nicht mehr in Frage. In diesem Fall kommt es dann zu einem Konflikt zwischen den Regeln, da head(r_2) bis head(r_5) die Negationen zu head(r_1) sind.

Um diesen Konflikt aufzulösen, müssen die Regeln noch gewichtet werden. Dies lässt sich durch die Prioritätsbedingungen erreichen.

$r_2 > r_1$, $r_3 > r_1$, $r_4 > r_1$, $r_5 > r_1$, $r_6 > r_1$

Dabei entspricht r_1 einer allgemeinen Regel, während die anderen die Ausnahmen bilden und daher natürlich stärker zu gewichten sind. In diesem Fall reicht es aus, die Regeln nur gegenüber r_1 zu gewichten, da es zwischen den Regeln r_2 bis r_6 keine Konflikte gibt. Sie führen nämlich alle zum Ablehnen der Wohnung x.

Nun stellen wir uns zwei Wohnungen a und b vor, die durch die Eigenschaften in der folgenden Tabelle charakterisiert werden.

Wohnung	Schlafzimmer	Größe	Haustiere	Stockwerk	Fahrstuhl	Miete
a	2	45	nein	3	ja	400
b	2	50	ja	1	nein	350
c	2	45	ja	4	ja	350

Wir können diese Eigenschaften in der Logik mit den eingeführten Prädikaten beschreiben. Für die Wohnung a ergibt sich:

size(a,45)
bedrooms(a,2)
floor(a,3)
lift(a)
¬pets(a)
price(a,400)

Aus ¬pets(a) folgt nach Regel r_4 ¬acceptable(a), und diese Regel ist stärker gewichtet als r_1. Damit wird die Wohnung a abgelehnt. Die Wohnungen b und c sind akzeptabel, da sie den Anforderungen Carlos’ entsprechen.

Bis hierher haben wir die Auswahl der Wohnungen schon eingeschränkt. Aber wir sind in der Lage, die Auswahl tatsächlich auf nur eine Wohnung zu reduzieren, die dann optimal auf Carlos‘ Anforderungen passt.

Dazu führen wir weitere Prädikate und Regeln ein:

cheapest(x)	Wohnung x hat die niedrigste Miete
largest(x)	Wohnung x ist am Größten
rent(x)	Carlos sollte Wohnung x mieten

r_7: cheapest(x) $\Rightarrow$ rent(x)
r_8: cheapest(x), largest(x) $\Rightarrow$ rent(x)

$r_8 > r_7$

Es gilt nun sowohl cheapest(b) als auch cheapest(c), da beide Wohnungen dieselbe Miete erfordern. Regel r_7 würde in beiden Fällen angewendet werden können, allerdings ergibt sich aus der Praxis ein Konflikt, da Carlos nur eine Wohnung mieten will. Dies kann durch folgendes Konfliktfeld beschrieben werden:

$C(rent(x)) = \{\neg rent(x)\} \cup \{rent(y) \mid y \neq x\}$

Zu einer gemieteten Wohnung x stehen damit auch alle anderen gemieteten Wohnungen im Konflikt. Dieser wird durch die stärker gewichtete Regel r_8 aufgelöst, deren Rumpf durch cheapest(b) und largest(b) erfüllt wird. Dadurch kommt nur rent(b) als Schlussfolgerung in Frage und Carlos hat eine Wohnung gefunden.

6.3.2 Syntax

Um die nichtmonotonen von den monotonen Regeln zu unterscheiden, wird eine andere Notation des Pfeils verwendet.

r_1: p(X) $\Rightarrow$ q(X)
r_2: r(X) $\Rightarrow$ ¬q(X)

Allgemein haben anfechtbare Regeln die Form

r: $L_1,\ldots,L_n \Rightarrow L$

r ist die Bezeichnung der Regel (Label). Die Menge der $L_1,\ldots, L_n$ sind die Voraussetzungen, sie werden auch als Body oder Rumpf bezeichnet, L ist der Head oder Kopf der Regel. In diesem Zusammenhang werden auch die Notationen head(r) (für L), body(r) (für $L_1,\ldots, L_n$) und r für die gesamte Regeln verwendet. $L_1,\ldots, L_n$, L können sowohl positive als auch negative atomare Formeln sein. Ein anfechtbares Logikprogramm ist ein Tripel (F, R, >). Es besteht aus einer Menge F von Tatsachen (Fakten), einer endlichen Menge R von anfechtbaren Regeln und einer Relation >, welche die Prioritätsbedingungen darstellt. Diese Relation kann als Menge von Paaren r > r’, wobei r und r’ Bezeichnung von Regeln aus R sind, geschrieben werden.

6.4 Überführung in XML

Damit Nutzer durch automatisierte Prozesse unterstützt werden können, ist es nötig, die Regeln, wie sie oben definiert wurden, für Computer zugänglich zu machen.

6.4.1 Monotone Regeln in XML

Mit Hilfe der XML Tags <term>, <function>, <var> und <const> können wir **Terme** in einer XML Syntax repräsentieren. Nehmen wir zum Beispiel den Term

f(X,a,g(b,Y))

Mit den oben genannten Tags ergibt sich dazu folgendes

```
<term>
    <function>f</function>
    <term>
        <var>X</var>
    </term>
    <term>
        <const>a</const>
    </term>
    <term>
        <function>g</function>
        <term>
            <const>b</const>
        </term>
        <term>
        <var>Y</var>
        </term>
    </term>
</term>
```

Um **atomare Formeln** in XML darzustellen benötigen wir noch zusätzliche Tags, nämlich <atom> und <predicate>. Die atomare Formel

p(X,a,f(b,Y))

lässt sich dann wie folgt in XML notieren:

```
<atom>
    <predicate>f</predicate>
    <term>
        <var>X</var>
    </term>
    <term>
        <const>a</const>
    </term>
    <term>
        <function>f</function>
        <term>
            <const>b</const>
        </term>
        <term>
        <var>Y</var>
        </term>
    </term>
</atom>
```

Mit dem zusätzlichen Tag <fact> lassen sich auch **Fakten** in XML repräsentieren. Zum Beispiel wird aus

p(a)

folgendes:

```
<fact>
    <atom>
        <predicate>p</predicate>
        <term>
            <const>a</const>
        </term>
    </atom>
</fact>
```

Schauen wir uns die folgende **Regel** an:

$p(X, a), q(Y, b) \rightarrow r(X, Y)$

Sie besteht aus einem Rumpf und einem Kopf. Der Kopf ist ein atomarer Ausdruck und der Rumpf besteht aus einer Liste von atomaren Formeln. Diese kann auch leer sein. Um diese Regel in XML zu repräsentieren, benötigen wir noch die Tags <rule>, <head> und <body>. Damit ergibt sich:

```
<rule>
    <head>
        <atom>
            <predicate>r</predicate>
            <term>
                <var>X</var>
            </term>
            <term>
                <var>Y</var>
            </term>
        </atom>
    </head>
    <body>
        <atom>
            <predicate>p</predicate>
            <term>
                <var>X</var>
            </term>
            <term>
                <const>a</const>
            </term>
        </atom>
        <atom>
            <predicate>q</predicate>
            <term>
                <var>Y</var>
            </term>
```

```
                    <term>
                            <const>b</const>
                    </term>
            </atom>
        </body>
</rule>
```

Zu beachten ist, dass während wir bei den Ausdrücken f(x) und p(a) implizit annehmen, dass zum Beispiel x eine Variable und a eine Konstante ist, müssen wir bei der Umsetzung in XML diese Unterscheidung explizit angeben. Der Computer ist nämlich nicht in der Lage, die allgemein gültigen Konventionen aus der Mathematik zu kennen.

Nachdem wir nun gesehen haben, dass sich Ausdrücke mit Hilfe entsprechender Tags in XML überführen lassen, brauchen wir noch ein Schema, welches den Aufbau der Ausdrücke beschreibt. Damit ist können die eingegebenen Ausdrücke auch verifiziert werden. Ein solches Schema lässt sich als DTD folgendermaßen schreiben:

Ein Programm besteht aus einer ungeordneten Liste von Regeln und Fakten.

```
<!ELEMENT program ( (rule | fact)* ) >
```

Ein Fakt ist genau eine atomare Formel.

```
<!ELEMENT fact (atom) >
```

Eine Regel besteht genau aus einem Kopf und einem Rumpf

```
<!ELEMENT rule (head, body) >
```

Der Kopf ist eine atomare Formel.

```
<!ELEMENT head (atom) >
```

Der Rumpf kann aus beliebig vielen atomaren Formeln bestehen oder leer sein.

```
<!ELEMENT body (atom*) >
```

Eine atomare Formel besteht aus einem Prädikat und einer Liste von Termen.

```
<!ELEMENT atom (predicate, term*) >
```

Ein Term ist entweder eine Konstante, eine Variable oder ein zusammengesetzter Ausdruck von einem Funktionssymbol und einer Liste von Termen. Dadurch lassen sich hier Terme ineinander verschachteln.

```
<!ELEMENT term (const | var | (function, term*) ) >
```

Prädikats- und Funktionssymbole, sowie Variablen und Konstanten sind Strings, die durch die Benutzer angegeben werden. Nach dieser Definition ist es möglich, auch ganze Sätze anzugeben, daher muss hier auf sinnvolle Eingaben geachtet werden.

```
<!ELEMENT predicate (#PCDATA) >
<!ELEMENT function (#PCDATA) >
<!ELEMENT var (#PCDATA) >
<!ELEMENT const (#PCDATA) >
```

Eine Abfrage ist durch eine Liste von atomaren Formeln repräsentiert. Dies stellt den Rumpf von Regeln dar.

```
<!ELEMENT query (atom*) >
```

6.4.2 Nichtmonotone Regeln in XML

Im Gegensatz zu den monotonen Regeln haben nichtmonotone einige syntaktische Unterschiede.

- Es gibt keine Funktionssymbole, dadurch wird die Schachtelung der Ausdrükke nicht so tief wie bei monotonen Regeln.
- Es können auch Negationen von atomaren Formeln in Kopf und Rumpf der Regeln erscheinen.
- Jede Regel hat eine eindeutige Bezeichnung.
- Neben den Regeln und Fakten besteht ein Programm auch aus den Prioritätsbeziehungen.

Schauen wir uns das vorher eingeführte Beispiel mit den Regeln

r_1: $p(X) \Rightarrow q(X)$
r_2: $r(X) \Rightarrow \neg q(X)$

den Fakten

p(a)
q(a)

und der Prioritätsbedingung

$r_1 > r_2$

an.

Dies lässt sich als Logikprogramm folgendermaßen in XML überführen:

```
<program>
	<rule id="r1">
		<head>
			<atom>
				<predicate>q</predicate>
				<var>X</var>
			</atom>
		</head>
		<body>
			<atom>
```

```
                        <predicate>q</predicate>
                        <var>X</var>
                </atom>
        </body>
</rule>

<rule id="r2">
….analog….
</rule>

<fact>
        <atom>
                <predicate>p</predicate>
                <const>a</const>
        </atom>
</fact>
<fact>
        <atom>
                <predicate>q</predicate>
                <const>a</const>
        </atom>
</fact>

<stronger superior="r1" inferior="r2"/>

</program>
```

Hier umschließt das Element program die beiden Regeln, die beiden Fakten und die Prioritätsbeziehung (stronger) zwischen r_1 und r_2.

Das dazugehörige DTD Schema sieht folgendermaßen aus:

Das Root Element program besteht auf einer beliebigen Kombination von Regeln, Fakten und Prioritätsbeziehungen.

```
<!ELEMENT program ( (rule | fact | stronger)* ) >
```

Fakten können entweder atomare Formel oder die Negation einer atomaren Formel sein. Die Negation wird auch als atomare Formel repräsentiert, es gibt nur einen zusätzlichen Tag um die Semantik klar zu stellen.

```
<!ELEMENT fact (atom | neg) >
<!ELEMENT neg (atom) >
```

Eine Regel besteht auf einem Kopf und einem Rumpf. Zusätzlich hat sie noch ein Attribut, welches ihr eine eindeutige ID gibt. Das ist für die Prioritätsbeziehungen wichtig.

```
<!ELEMENT rule (head, body) >
<!ATTLIST rule
        ID #IMPLIED >
```

Der Kopf besteht aus einer atomaren Formel oder deren Negation. Der Rumpf kann aus beliebig vielen atomaren Formeln (oder Negationen) bestehen.

```
<!ELEMENT head (atom | neg) >
<!ELEMENT body ( (atom | neg)* ) >
```

Eine atomare Formel setzt sich aus einem Prädikat (als Bezeichnung) und einer beliebig langen Liste von Variablen und/oder Konstanten zusammen.

```
<!ELEMENT atom (predicate, (var | const)* ) >
```

Das Element stronger ist ein leeres Element und stellt die Prioritätsbeziehungen dar. Dazu hat es zwei Attribute superior und inferior. Diese verweisen auf die eindeutige ID von Regeln und zeigen so die Gewichtung der Regeln untereinander, wobei superior die stärkere Regel, zum Beispiel eine Ausnahme zu einer generellen Regel, referenziert.

```
<!ELEMENT stronger EMPTY >
<!ATTLIST stronger
        superior IDREF #REQUIRED
        inferior IDREF #REQUIRED >
```

Die Elemente predicate, var und const werden durch Eingaben des Nutzers belegt.

```
<!ELEMENT predicate (#PCDATA) >
<!ELEMENT var (#PCDATA) >
<!ELEMENT const (#PCDATA) >
```

Eine Abfrage ist eine Liste von atomaren Formeln. Sie spiegelt damit nur den Rumpf einer Regel wieder.

```
<!ELEMENT query (atom*) >
```

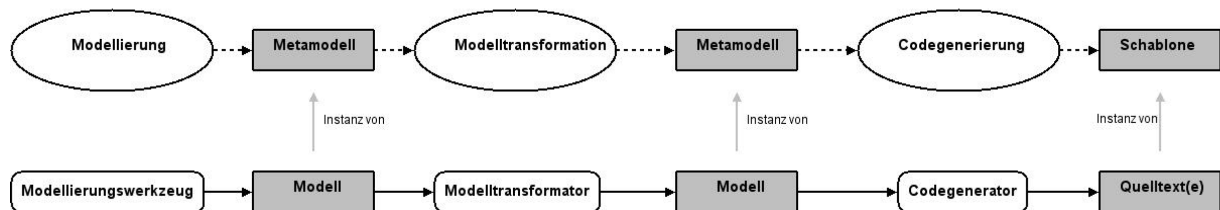
6.5 Quelle

G. Antoniou, F. van Harmelen: A Semantic Web Primer, MIT Press 2004

7 MDA (Spallek, Wloch)

7.1 Zielsetzung der MDA

Softwaresysteme werden zum Einen stets komplizierter, zum Anderen steigt auch ihre Anzahl und die Anzahl der Plattformen, auf denen sie eingesetzt werden. Es wäre ideal, wenn man Programme in Zukunft aus hinreichend genau spezifizierten Modellen erzeugen lassen könnte. In diesem Fall müsste man sich lediglich auf das Programmverhalten konzentrieren, die konkrete Implementierung könnte automatisiert ablaufen. Zeichnung 1 stellt den Ablauf der automatisierten Programmentwicklung dar. Als Basis dient ein Modell, dieses muss passend transformiert werden. Abschließend implementiert der Codegenerator das Programm. Im oberen Strang der Zeichnung erkennt man den Prozess. Dem gegenüber gestellt erkennt man im unteren Strang die jeweiligen Werkzeuge zu den einzelnen Phasen.



Zeichnung 1: Automatisierung der Implementierung

Im Folgenden beschreiben wir die Model Driven Architecture (MDA). Dazu erklären wir zuerst Metamodelle und Modelle und welche Abstraktionsstufen bei der Analyse eines zu modellierenden Systems verwendet werden. Anschließend gehen wir auf die verschiedenen MDA Metamodelle ein und nutzen eines von ihnen zur Erläuterung des Aufbaus der MDA. Die in Zeichnung 1 zu erkennende Modelltransformation wird im Abschnitt 7 thematisiert, die Codegenerierung im Abschnitt 10. Des Weiteren beleuchten wir den Zusammenhang zwischen MOF und UML, beschreiben die Darstellung der Modelle im Rechner sowie ihre Speicherung und Austauschbarkeit mit Hilfe von XMI und gehen kurz auf UML Profile ein, mit denen wir die Ausdrucksmöglichkeiten der Modelle erweitern können.

7.2 Metamodelle und Modelle

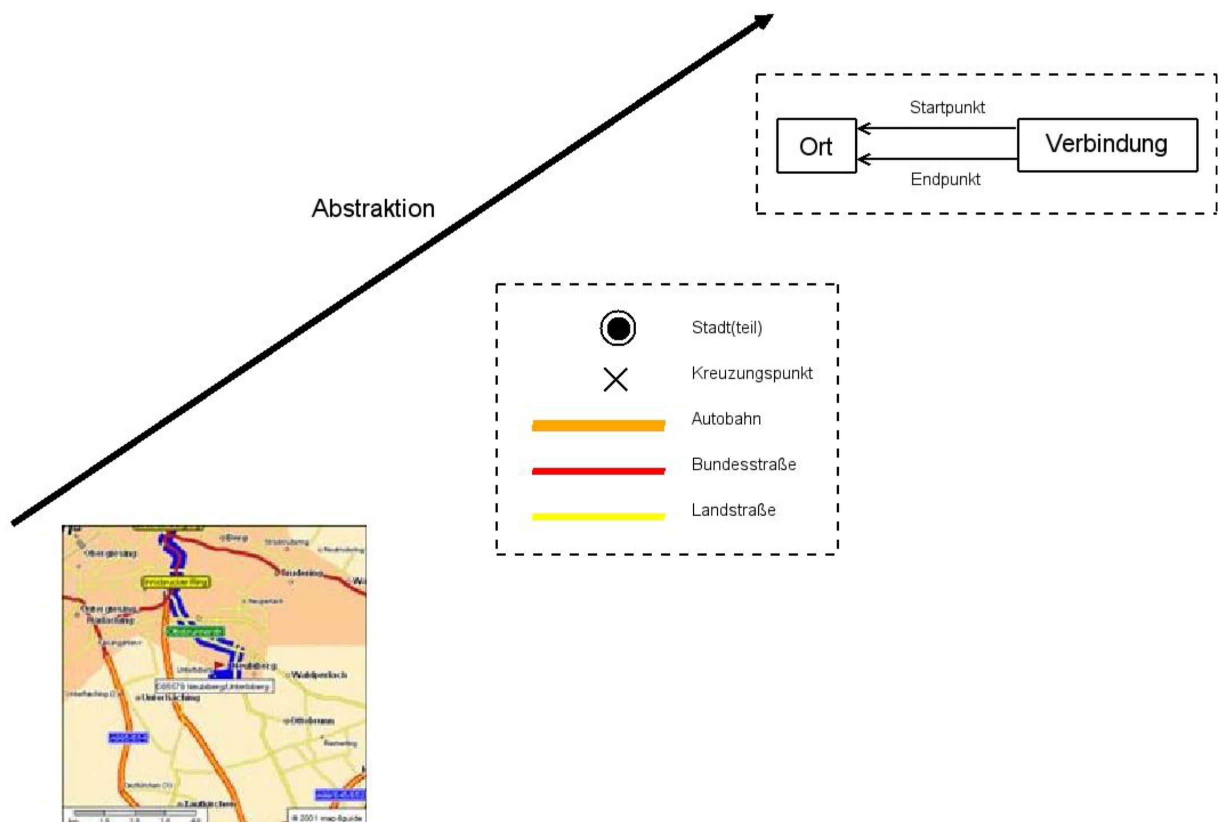
Der Begriff des Modells wird in vielerlei Gebieten jeweils verschieden genutzt. Wir wollen hier einfach sagen, ein Modell ist ein vereinfachtes Abbild der Wirklichkeit. Außerdem gibt es einige Eigenschaften die Modelle erfüllen müssen:

- abstrakt: In einem Modell wird nur ein Wirklichkeitsausschnitt dargestellt und Details wegabstrahiert
- verständlich: Es muss von einer bestimmten Gruppe (Menschen oder Computer) verstanden werden können
- korrekt: Es muss die für die Fragestellung interessanten Eigenschaften korrekt abbilden

- vorhersehbar: Das Verhalten des durch modellierten Systems muss mittels analytischer Methoden oder durch Experimente vorhersehbar sein
- ökonomisch: Analysen und Experimente müssen an dem Modell deutlich preiswerter sein als an dem zu modellierenden realen System

Jede Modellierungssprache besitzt ein Metamodell. Dieses legt fest, welche Elemente diese Sprache besitzt und wie diese sinnvoll miteinander verbunden werden können. Es definiert also Vokabular und Grammatik eben dieser Modellierungssprache. In dieser Sprache kann man Systeme beschreiben, die untersucht werden sollen. Wollte man z.B. eine Straßenkarte zeichnen (ein Modell), würde man im Metamodell beschreiben, dass es Orte gibt, die miteinander über Straßen verbunden werden können. Die konkrete Karte ist also eine Instanz des Metamodells.

Natürlich sind auch Metamodelle Modelle, weshalb auch sie Metamodelle besitzen. D.h. ein Modell A besitzt ein Metamodell B, welches selber ein Modell ist und folglich auch ein Metamodell C besitzt. D.h. C ist das Meta-Meta-Modell von A. Dieses Vorgehen ist nicht auf eine bestimmte Anzahl von Ebenen begrenzt (siehe Zeichnung 2).



Zeichnung 2: Beziehung Modell - Metamodell

7.3 Abstraktionslevels für die Analyse von Systemen

Gehen wir nun davon aus, dass wir ein System zu modellieren haben. Nach einer Befragung / Analyse / etc. haben wir ein deskriptives Modell vor Augen. Dieser grobe Entwurf ist ein *Computational-independent model* (CIM). Es stellt ein Domänenmodell dar, das die im System vorhandenen Elemente sowie die Beziehungen zwischen ihnen zwar darstellt, aber noch keine konkrete Struktur besitzt. Dies erledigen

platform independent models (PIM). PIMs sollen alle Informationen mitbringen, um auf einer angenommenen technologisch unabhängigen Maschine ausgeführt werden zu können. Einem sich auf dieser Ebene befindlichen Element fehlen zur realen Ausführung auf einer existierenden Architektur nur noch die Informationen über die verwendete Plattform. Damit angereicherte Modelle heißen *platform-specific models* (PSM).

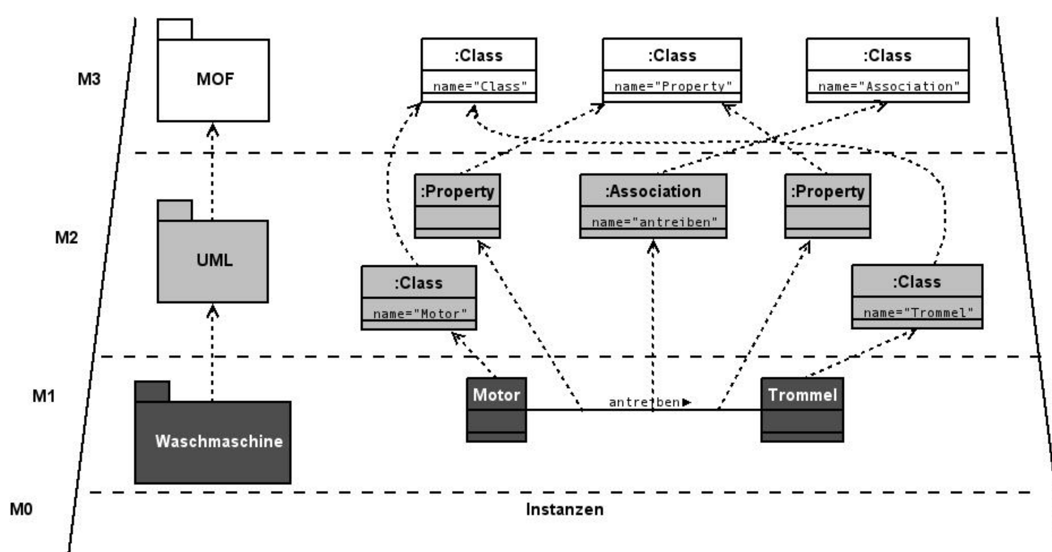
Dazu ein kleines Beispiel zur Veranschaulichung: Nehmen wir an, wir wollten einen Webshop einführen. Wir müssten uns zuerst im Klaren darüber werden, welche Anwendungsfälle auftreten können. Wir geben noch keinerlei konkrete Informationen über das Zusammenspiel der Komponenten wie z.B. die Datenspeicherung an (CIM-Ebene). Nun entscheiden wir uns, diese Daten mit Hilfe einer relationalen Datenbank zu speichern. Zur Modellierung verwenden wir das Relationenschema. Hier ist schon Berechenbarkeit gegeben (PIM-Ebene). Treffen wir nun die Entscheidung, MySQL als Datenbank zu nutzen, müssen wir uns mit seinem SQL-Dialekt beschäftigen. Damit sind wir auf der PSM-Ebene angekommen.

Zwischen den verschiedenen Ebenen finden Transformationen statt, für die ein bestimmtes Transformationswissen benötigt wird. Durch diese Umwandlungen wird das Modell zunehmend konkreter (von CIM zu PIM zu PSM). Die Gegenrichtung ist dabei auch möglich.

Die Idee der MDA ist dabei, nur bis zur PIM-Ebene zu modellieren und die Umsetzung auf die PSM-Ebene (z.B. Java-Code) automatisiert zu realisieren.

7.4 Aufbau der MDA

Der Aufbau der MDA gliedert sich in vier Schichten. In Zeichnung 1 mit M0 bis M4 gekennzeichnet. Allgemein gilt: die Abstraktion nimmt von unten nach oben immer mehr zu.



Zeichnung 3: Aufbau der MDA

Auf der ersten Ebene (M0) werden die Instanzen zusammen mit ihrem Zustand abgebildet. Hier sind also, in diesem Beispiel, konkrete Ausprägungen der Klassen Motor und Trommel angesiedelt. Die Schicht M1 wird als Modellschicht bezeichnet. Hier befindet sich das anwendungsspezifische Modell. Man erkennt die in UML dargestellten beiden Klassen und eine Assoziation „antreiben“ zwischen ihnen.

In der Ebene M2 (oder auch Metamodellschicht) ist wie der Name schon sagt das Metamodell zu unserem UML Diagramm untergebracht. Diese Metamodell definiert die Klassen und Assoziationen des UML Diagramms durch die Klassen Class und Assoziation. Diese werden also in M1 instantiiert.

In der letzten Schicht (M3) wird nun das Metamodell unseres Metamodells von Schicht M2 definiert. Dies ist also das Metametamodell von Schicht M1. Daher wird diese Schicht auch „Meta-Metamodellschicht“ genannt. Das Modell welches in dieser Schicht definiert ist nennt man *Meta Object Facility* (MOF).

7.5 MDA Metamodelle

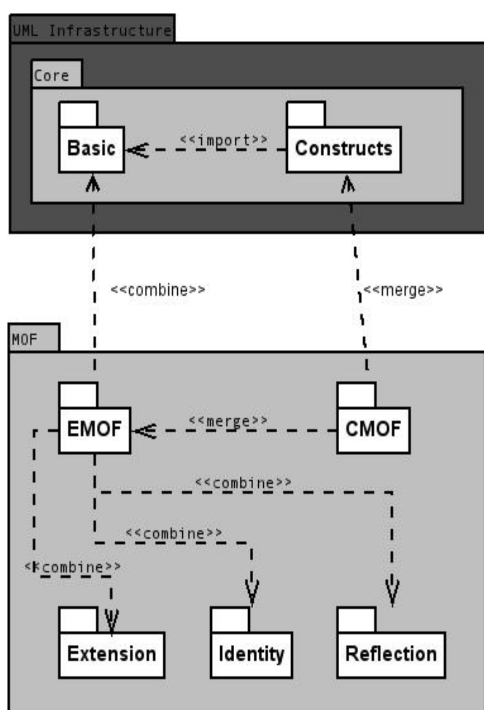
Bisher ist nicht angesprochen worden, welche(s) Metamodell in M2 eigentlich genutzt werden kann. Wir wollen an dieser Stelle die drei bekanntesten ansprechen:

1. Unified Modeling Language (UML)
2. Common Warehouse Metamodel (CWM)
3. Ontology Definition Metamodel (ODM)

Auf die UML wird in 7.6 näher eingegangen. Das CWM ist eine Sammlung verschiedener Metamodelle, welche einen Industriestandard für Integrationstools für *data warehouse* und Geschäftsanalyse darstellen. Die ODM ist eine Sprache, die sich besonders gut zur Modellierung von Ontologien eignet. Da (wie in Zeichnung 3 zu sehen) die UML von der MOF abgeleitet wird, wollen wir nun diese eingehender betrachten.

7.6 Die MOF

Der MOF stellt genau die Komponenten zur Verfügung, die man zur Definition anderer Modellierungssprachen mindestens benötigt. Da andere Modellierungssprachen also in der MOF abgefasst sind, reicht es, wenn ein entsprechendes Modellierungstool diese genau implementiert. Ohne an dieser Stelle alle Elemente der MOF nennen zu können, sei darauf hingewiesen, dass bereits die MOF die Konzepte Datentypen, Pakete, Klassen, Assoziationen eingeführt werden und weitere Sprachen auf dieser Basis erstellt werden können. Die MOF in ihrer heutigen Form (2.0) nutzt Elemente des Core Paketes aus der UML *infrastructure.* Dieses Paket ist aus gemeinsamen Sprachelementen von UML und MOF gebildet.



Zeichnung 4: Beziehung zwischen UML und MOF

Da die MOF eine Basis für die Modellierung darstellt, betrachten wir sie etwas eingehender und wollen dabei auch ihre Ontologie klären. Ein genaues Verständnis der MOF ist notwendig, um später aus in ihr verfassten Modellen Quellcode generieren zu können. Der MOF Standard 2.0 sieht zweierlei Metamodelle vor:

1. Essential MOF (EMOF)
2. Complete MOF (CMOF)

Die EMOF beschränkt sich darauf, die oben angesprochenen Konzepte um Extension, *reflection* und *identity* zu erweitern.

Attribute und Operationen sind ebenfalls schon in der EMOF enthalten und sind sowohl in der *basic* als auch in der *Constructs* definiert.

Nur in der *Constructs* Schicht und damit nur in der CMOF definiert ist die Redefinition von Attributen, sofern der zu redefinierende Typ dem des Neuen entspricht. Ebenso gehören die Teilmengen- und Vereinigungsbeziehungen dazu. Typkonforme Attribute können als Unter- bzw. Obermenge definiert werden. Damit wird jede Instanz eines als Teilmenge angesehenen *Attributes* auch als Instanz des als Obermenge angesehenen *Attributes* geführt. Werden verschiedene Attribute A, B,…X vereinigt, so kann das neue entstandene Attribut jeden Wert aus der Vereinigungsmenge A union B union … union X annehmen.

In der MOF ist zwar das aus der UML Infrastructure übernommene Element Operation enthalten, jedoch beschränkt sich diese darauf, den Auslösemechanismus (incl. Vorbedingung, Nachbedingung, Zwischenbedingung) sowie Typisierung sowie die Parameter zu definieren. Das Operationsverhalten wird nicht genauer spezifiziert. Die eigentliche Spezifikation dieses Verhaltens muss dann später erfolgen.

Vererbung wiederum ist Bestandteil jeder sowohl von EMOF als auch von CMOF.

Was Assoziationen angeht, werden diese aus dem Constructs-Paket übernommen und sind somit nur in der CMOF enthalten. Assoziationen werden durch die Klassen Association und Property beschrieben. Instanzen von Association modellieren die Assoziation als solche, Instanzen von Property stellen die Assoziationsenden dar. Assoziationen können >=2 Enden haben.

Klassen definieren eine Struktur und das Verhalten ihrer Instanzen, wohingegen Datentypen nur eine Menge von Werten enthalten, die sie annehmen können. Sofern diese Datentypen unstrukturiert sind, sind sie auch Bestandteil der EMOF, da sie aus dem Basic-Paket stammen. Strukturierte Datentypen sowie Aufzählungsdatentypen sind jedoch nur Teil des Constructs-Paketes, also nicht in der EMOF, wohl aber in der CMOF enthalten. Strukturierte Datentypen sind eine Menge von einem oder mehreren Feldern. Ihre Inhalte sind Attribute und damit Instanzen der Metaklasse Property. Aufzählungstypen setzen sich aus einer endlichen Anzahl von geordneten Werten zusammen. Der Aufzählungsdatentyp selber ist eine Instanz des Klasse Enumeration, seine Werte sind Instanzen von Enumeration Literal.

Pakete als „Behälter" für logisch zusammengehörige Klassen, können hierarchisch geschachtelt werden. Diese Möglichkeit kommt von Namespaces aus Constructs (deshalb sind Pakete nur Teil der CMOS). Mit Instanzen von ElementImport wird der Namespace gefüllt. Diese Instanzen sind der Import der Modellelemente. Package-

Import unterscheidet sich von Element-Import nur dadurch, dass hier ganze Pakete importiert werden. Pakete können auch zusammengefügt werden, was uns aber nicht weiter interessieren soll.

Kapselung ist wieder sowohl in EMOF als auch in CMOF verfügbar und wird als Kompositionsbeziehung modelliert. Pakete können Modellelemente importieren indem sie eine Referenz auf das zu importierende Element in ihren eigenen Namensraum einfügen. Folglich können die importierten Elemente auch nicht verändert werden, lassen sich aber wiederum importieren.

Man kann durch Redefinition und Spezialisierung den Inhalt zweier Pakete vereinigen, wobei das vereinigte Paket die Modellelemente der beiden Ursprungspakete enthält und diese zu einem Element zusammenfügt.

Der Ausdruck von Bedingungen in sowohl der MOF als auch der UML findet zumeist mittels der OCL (*Object Constraints Language*) ausgedrückt und machen das vormals statische Modell dynamisch.

Warum ist all dies hier derart genau dargestellt worden? Bekanntlich müssen maschinenausführbare Programme hinreichend spezifiziert sein. Nicht anders verhält es sich dabei mit ihren Modellen, die ontologisch sauber darzustellen sind. Atom-Molekül-Stein-Haus ist für einen Computer wesentlich besser „verständlich“ als nur von einem Haus zu sprechen.

7.7 UML und Transformationen

Die UML bietet verschiedene Tools, die sich zum Einem für die Modellierung auf den verschiedenen Abstraktionslevel (CIM, PIM, PSM), zum Anderen zum Integrieren verschiedener Sichten eignen:

- Anwendungsfalldiagramme (use case) zur Darstellung der Interaktion eines Akteurs mit einem System
- Klassendiagramme zur Darstellung von Klassen sowie deren Beziehungen untereinander
- Zustandsdiagramme zur Darstellung des Programmablaufs als endlicher Automat
- Aktivitätsdiagramme zur Darstellungen des Ablaufes eines bestimmten Anwendungsfalles
- Sequenzdiagramme zur Darstellung von Interaktion und Nachrichtenaustausch zwischen Prozessen, Objekten oder Akteuren chronologisch geordnet
- Kollaborationsdiagramme zur Darstellung des gleichen Inhalts wie Sequenzdiagramme aber mit der Betonung der Objektbeziehung und nicht der zeitlichen Abfolge
- Komponentendiagramme zur Darstellung von Subsystemen mit ihren Schnittstellen und zugehörigen Ports
- Verteilungsdiagramm oder Einsatzdiagramm zur Darstellung von physikalischen Knoten auf denen das System arbeitet

Wir machen nun einen Sprung von der MOF zur UML, ohne die dazwischen liegenden Schritte zu erwähnen. Das nähme allerdings ein paar tausend Seite in Anspruch und würde den Rahmen dieser Arbeit sprengen. Der Einsatz der UML für maschinenlesbare Modelle, aus denen dann letztlich sogar Quellcode gewonnen werden soll, mag sehr verwirren. Denn die UML ist als reine Darstellungssprache bekannt. Tatsächlich aber ist sie nun in der Version 2.0 derart spezifiziert, dass man durchaus hinreichend beschriebene Modelle für die Quellcodegenerierung mit ihrer Hilfe erstellen kann.

Dabei wird bei der hier beschriebenen Modelltransformation noch kein Quellcode in der Zielsprache erstellt. Vielmehr wird ein Modell in ein anderes überführt.

Es werden auch immer mehr Tools geschrieben, die UML-Modelle transformieren können, wobei es eben hier verschiedene Ansätze gibt:

- Direkte Programmierung: Hier werden Metamodelle direkt repräsentiert, mit Schnittstellen versehen und können gem. selbst programmierter Transformationsregeln ineinander überführt werden. Dieser Ansatz ist aufgrund der eigenen Erstellung der Regeln überaus flexibel, aber auch sehr zeitaufwendig in der Implementierung

- Deklarative Ansätze: Es werden Relationen zwischen Quell- und Zielelementen erstellt und durch Bedingungen weiter spezifiziert. Z.B. könnte man eine binäre Relation zwischen der Syntax und der Semantik eines Modells erstellen. Dieser Ansatz ist allerdings noch nicht vollständig auf seine Praxistauglichkeit getestet.

- Graphentransformationen: Ist ein Modell visuell beschrieben, bietet sich diese Methode an. Dabei werden regelbasierte Transformationen beliebiger Graphen und Hypergraphen (eine Kante verbindet mehrere Knoten) verwendet.

7.8 XMI

Wir verlassen hier (und im nächsten Abschnitt) kurz den Prozess der uns der eigentlichen Codegenerierung immer näher führt und stellen den Datenaustausch in den Mittelpunkt. Stellt man nun ein Modell in der UML dar, so muss dieses Modell in einem allgemein verständlichen Format gespeichert werden können. Ein in einem Programm X erstelltes Modell muss auch in einem Programm Y lesbar sein können.

Der Gebrauch vom XML zur Speicherung des Modells bietet sich dabei an, da es bereits ein weit verbreiteter Standard ist. Außerdem kann man mittels so auch Objekte serialisieren. Für die entsprechenden XML-Dokumente und die zugehörigen Transformationsregeln hat sich dabei der Name XMI (*XML Metadata Interchange*) eingebürgert. Die XMI muss dabei den Austausch von MOF-Modellen und Ableitungen der MOF ermöglichen, wobei die UML eine besondere Ableitung darstellt. Dazu bedarf es der entsprechenden Schemata.

XMI umfasst also folgende Komponenten:

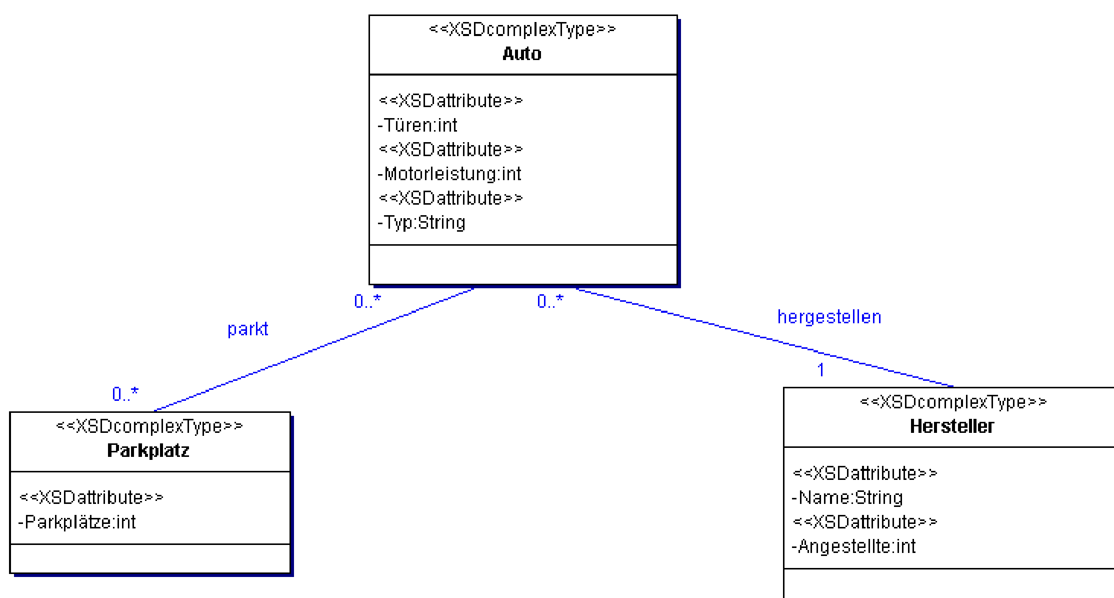
- XML-Schema für die MOF
- XML-Schema für die UML
- XML-Schema für andere Metamodelle
- Konkrete Instanzen der XML-Schemata als Modelle (z.B. ein UML-Klassendiagramm als Instanz des UML-Schemas)
- Möglichkeiten zur Serialisierung von Objekten

7.9 UML Profile und Ontologien

Die Elemente der UML sind begrenzt. Für das Ziel der MDA ist es aber zwingend notwendig, die verwendeten Modelle auch an neue Gegebenheiten anzupassen und somit die Möglichkeit zu erschließen, auch neue Domänen leicht modellieren zu können. Idealerweise kreiert man dazu einfach neue Elemente mit geänderter Semantik. Diese Elemente können aus alten kombiniert werden und mit freiem Text versehen werden. Zur Änderung der Semantik stehen vier Mittel zur Verfügung:

1. Stereotypen sind Unterklassen von UML-Metaklassen mit frei zugeordneter Semantik (bekannt: <<interface>>)
2. Tag-Definitionen führen bestehenden Elementen neue Metaattribute hinzu
3. Tagged Values stellen die eigentlichen Werte zu den Tag-Definitionen dar
4. Constraints dienen dazu, die Semantik der zugewiesenen Elemente zu präzisieren. Dies kann informell mittels natürlicher Sprache oder formell mittels der Objects Constraint Language (OCL) erfolgen.

UML Diagramme sind weitaus übersichtlicher und intuitiver als eine Repräsentation von Modellen mittels XML. Dies gilt besonders bei der Einführung neuer Sprachelemente und Definitionen, weshalb diese von Carlson als Standard für die Systemspezifikation und das Design vorgeschlagen worden sind.



Zeichnung 5: Autoontologie mittels UML

Die hier angegebenen Stereotypen (XSDcomplexType, XSDattribute) dienen dazu, aus diesem UML ein XML Schema entwickeln zu können. Denn ein geeignetes Tool, das die beiden Stereotypen „versteht“, kann daraus leicht ein Schema entwickeln. Mit Hilfe des obigen UML-Diagrammes haben wir also eine Ontologie geschaffen und können dieses Diagramm mittels XMI leicht lesbar machen.

7.10 Abbildung der MOF-Elemente auf Java

Nun kehren wir wieder auf den in Zeichnung 1 beschriebenen Weg zurück und bewegen uns an die Stelle, die direkt vor der Codegenerierung liegt. Wir wollen dazu im Folgenden die Frage klären, wie die Elemente eines MOF-Modells mit Hilfe von Java dargestellt werden können. Eine Darstellung der Elemente der UML wäre ungleich schwieriger zu erreichen, weshalb wir hier mit der MOF Vorlieb nehmen. Die Umgebung, in der wir uns von nun an bewegen wollen, ist Java. Wir können uns das so vorstellen, dass das Tool, in dem wir das Modell idealerweise visuell erkennen können, in Java geschrieben ist. Eine Java angepasste Implementierung ist nötig, da MOF-Modelle gespeichert, geladen und dargestellt werden sollen. Nur wenn die Elemente der MOF in Java repräsentiert werden, kann ein in Java geschriebener Codegenerator überhaupt funktionieren. Denn dieser muss z.B. den Zugriff auf Namen, Attribute und Operationen einer Klasse ermöglichen. Die generierten Komponenten werden somit in Java-Code dargestellt. Zwar existiert mit dem *Java Metadata Interface* (JMI) bereits eine fertige Implementierung, doch zum Einen implementiert diese nur MOF 1.4, zum Anderen soll hier ein Einblick gegeben werden, wie es möglich ist, MOF-Elemente auf Java abzubilden.

7.10.1 Pakete

Das oberste Element eines jeden MOF-Modells ist ein Paket. Instanziiert man also dieses Paket, instanziiert man auch alle Unterlemente. Da mit JMI nicht Pakete, sondern Paketinstanzen referenziert werden, werden bei einem Zugriff auf die enthaltenen Elemente auch wiederum Referenzen auf Objekte (Paket, Klasse, Datentyp – hier alles Objekte) zurückgegeben. Das folgende stellt eine Schnittstelle für die Erzeugung von Paketen dar.

```
public interface <packageName>Package extends RefPackage {

    //für jedes enthaltene Paket
    public <NestedPackageName>Package get<NestedPackageName>();

    //für jede enthaltene Klasse
    public <ClassName>Class get<ClassName>();

    //für jede Assoziation
    public <AssociationName> get<AssociationName>();

    //für jeden enthaltenen strukturierten Datentyp
    public <StructTypeName>DataType get<StructTypeName>();
```

```
        //für jeden enthaltenen Aufzählungstyp
        public <EnumerationTypeName> get<EnumerationTypeName>();
}
```

Zu jedem enthaltenen Paket wird folglich eine get<NestedPackageName>() erstellt. Da ja die JMI nur Objekte referenziert, wird hier ein Verweis auf das Objekt zurückgegeben, das das „Paket“ enthält. Das gilt auch für get<ClassName>, get<StructName> und get<EnumerationTypeName>, welche wiederum eine Referenz auf das Objekt zurückgeben, das eines dieser Elemente repräsentiert.

7.10.2 Klassen

Die Schnittstellen zur konformen Implementierung für eine jede Klasse werden durch einen Namen mit nachgestelltem Class repräsentiert. Diese enthalten dann eine passende create Methode, mit der dann wiederum Instanzen erstellt werden können.

```
public interface <ClassName>Class {

        public <ClassName> create<ClassName>{
                //Erzeugung jedes nicht abgeleiteten Attributes
                <AttributeType> <AttributeName>
        } throws javax.jmi.reflect.JmiException;

        // für jeden enthaltenen strukturierten Datentyp
        public <StructTypeName>DataType get<StructTypeName>()
                throws javax.jmi.reflect.JmiException;
}
```

Mit Hilfe dieser Schnittstelle kann man keine Attribute und Operationen auf Klassenebene erzeugen, was aber der MOF-Spezifikation 2.0 entspricht. Wenn die Klassen aber abgesehen von ihren Attributen „inhaltslos“ sind, warum sollen dann überhaupt Schnittstellen und Repräsentationen von Klassen implementiert werden? Durch dieses Vorgehen erleichtert man später auf der Implementierungsebene hinzuzufügende Klassen-Operationen (z.B. das Abfragen der Anzahl von Instanzen einer Klasse).

Will man Instanzen einer MOF-Klasse in Java darstellen, braucht man dazu eine Schablone, in welcher alle Operationen und Attribute auf Java-Sprachelemente abgebildet werden.

```
public interface <ClassName> extends
        //falls die Klasse keine Superklasse besitzt
        javax.jmi.reflect.RefObject
        /*andernfalls, wird das zu jeder Superklasse gehörige Interface (eins wie dieses) geerbt */
        <SuperClassName>,…
```

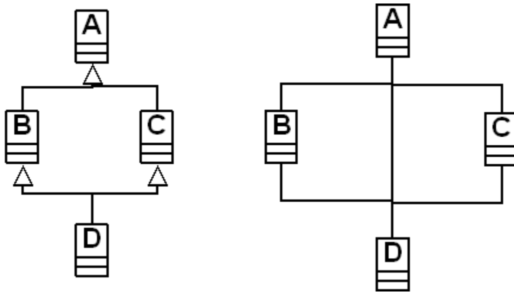
```
{
//für jedes Attribut
        //falls obere Grenze der Multiplizität gleich 1
                public <AttributeType> get<AttributeName>()
                        throws javac.jmi.reflect.JmiException;
                        //falls das Attribut nicht schreibgeschützt ist
                public void <MutatorName> (<AttributeType> newValue)
                        throws javax.jmi.reflect.JmiException;
                //falls obere Grenze der Multiplizität größer 1 und ungeordnet
                public Collection get<AttributeName>()
                        throws javax.jmi.reflect.JmiException;
                // falls obere Grenze der Multiplizität größer 1 und geordnet
                public List get<AttributeName> ()
                        throws javax.jmi.reflect.JmiException;

//nun die Definition der Operationen
        //…ohne Rückgabewert
        public void <OperationName> {
        //… mit Rückgabewert einfach <ReturnType> statt void
        //und
        return <ObjectType> <ObjectName>
        } throws //hier alle Ausnahmen
        … , javax.jmi.relfect.JmiException
```

Wir haben für alle Attribute der Multiplizität 1 einen Getter und einen Setter für jedes nicht-schreibgeschützte Attribut eingefügt. Nicht geordnete Attribute der Multiplizität >1 werden durch den Collection-Getter, die geordneten durch den List-Getter dargestellt. Die Setter für Attribute mit Multiplizität > 1 sind dieselben wie für Attribute der Multiplizität 1, denn <AttributeType> ist ja „flexibel".

7.10.3 Vererbung

Die MOF erlaubt Mehrfachvererbung. Diese ist in Java allerdings nicht darstellbar, weshalb wir dies durch Delegation, Kopieren oder die Kombination der Einfachvererbung ausgleichen müssen. Wir wollen an dieser Stelle aus Platzgründen nur den Mechanismus der Delegation darstellen.

Die Klasse D muss jede Methode der Klassen, von denen sie erbt, hinzugefügt bekommen. Wird nun eine solche, hinzugefügte Methode aufgerufen, „delegiert" sie den Aufruf einfach an die entsprechende durch die Assoziation bestimmte Klasse weiter. Zeichnung 6 veranschaulicht diese Ausführungen.



Zeichnung 6: Delegation statt Mehrfachvererbung

7.10.4 Attribute und Operationen

Attribute und Operationen in MOF-Klassen werden durch Namen, Typ und Multiplizität eindeutig beschrieben. Nehmen wir an, eine Klasse C erbt von den Klassen A und B (in der MOF ist das ja erlaubt) und sowohl A als auch B definieren eine Operation oder ein Attribut gleichen Namens und Typs. Welches Attribut gilt? Da Mehrfachvererbung in Java nicht existiert, gibt es auch kein Sprachkonstrukt, mit dem man diese Frage klären könnte. Damit diese Frage nicht erst zur Laufzeit „beantwortet“ wird, wird stets überprüft, ob ein(e) Attribut/Operation in einer Vererbungskette einmalig ist. Dann ist nichts zu tun. Ist dem nicht der Fall, muss überprüft werden, ob es ein Wurzelattribut gibt, das alle anderen lediglich redefinieren. Das stellt natürlich kein Problem dar. Ist aber all das nicht gegeben, muss das Attribut stets mit dem voll qualifizierten Namen angegeben werden. Bei Operationen ist dies ein wenig vereinfacht, da diese zusätzlich über ihre Signatur erkannt werden können. Es stellt allerdings ein Problem dar, dass der Rückgabewert in der MOF/UML mit zur Signatur gehört, was in Java nicht der Fall ist. Dies ist aber noch nicht das einzige entstehende Problem. Mehrfachvererbung auf Schnittstellenbasis (siehe oben) wird in Java erlaubt. Wird nun ein Attribut oder eine Operation mittels ihres vollqualifizierten Namens angesprochen, bleibt das unqualifizierte Attribut undefiniert. Hier muss sich also getrickst werden, was allerdings einmal mehr aus Platzgründen nicht dargestellt werden soll.

7.10.5 Assoziationen

Die MOF erlaubt es (im Gegensatz zur UML), von einer Assoziation, die auch dieselben Objekte miteinander verbindet, mehrere Instanzen anzufertigen.

```
public interface Link {
        public RefObject getFirstEnd();
        public RefObject getSecondEnd();
}

public interface <AssociationName> extends javax.jmi.relfect.RefAssociation {

public boolean exists (Link link) throws javax.jmi.reflect.JmiException;
```

```
//wenn obere Grenze der Multiplizität des ersten Assoziationselementes gleich 1
public <End1ClassName> get<End1Name> (<End2Type> opposite)
        throws javax.jmi.relfect.JmiException;

//obere Grenze der Multiplizität des 1. Assoziationselementes >1 und ungeordnet
public Collection get <End1Name> (<End2Name> opposite)
        throws javax.jmi.relfection.JmiException;

//… und geordnet

public List get <End1Name> (<End2Name> opposite)

        throws javax.jmi.relfection.JmiException;

//gleiches Vorgehen für das zweite Assoziationsende

public Link add (<End1Type> end1, <End2Type> end2)
        throws javax.jmi.relfect.JmiException;

public boolean remove (Link link) throws javax.jmi.relfection.JmiException;
```

Jede Instanz einer Assoziation wird also durch ein Objekt repräsentiert, welches die Link-Schnittstelle implementiert. Die Eigenschaften der Assoziation können dann mittels der jeweiligen Methoden abgefragt bzw. verändert werden.

7.10.6 Einfache und strukturierte Datentypen

Um alle eingebauten Features von Java nutzen zu können (z.B. ++) werden primitive MOF-Datentypen wo immer möglich auf primitive Java-Datentypen abgebildet.

Da es für strukturierte Datentypen in Java keine Konstrukte wie z.B. die aus Ada bekannten Records existieren, müssen diese durch Objekte dargestellt werden.

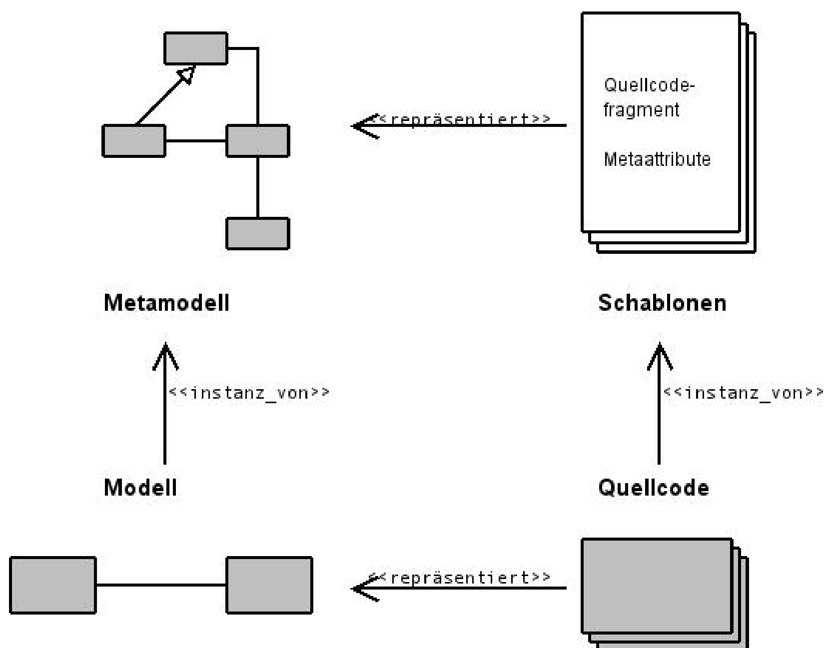
```
public interface <StructTypeName>DataType {
        public <StructTypeName> get<StructTypeName> ()
        throws java.jmi.reflect.JmiException  {
                //für jedes Attribut des Datentyps
                <AttributeName> <AccessorName>
                }
}
```

Diese Schnittstelle stellt die Codegenerierungsschablone für strukturierte Datentypen dar. Es fehlt aber noch eine Schnittstelle, mit deren Hilfe man nun auf die einzelnen Elemente des erzeugte, strukturierten Datentyps zugreifen kann.

```
public interface <StructTypeName> extends javax.jmi.reflect.RefStruct {
        //Abfrage für jedes Element des Datentyps
        public <AttributeName> <AccessorName> ()
        throws javax.jmi.reflect.JmiException;
}
```

7.11 Codegeneration mit dem MOmo-Baukasten

Nun wenden wir uns der eigentlichen Codegenerierung zu und wollen dazu den MOmo-Baukasten betrachten. MOmo steht für MOF-basierte Metamodellierung und der MOmo-Baukasten stellt eine Möglichkeit dar, wie man aus einem MOF-basierten Modell (z.B. der UML) Code generieren kann. Der MOmo-Baukasten verwendet einen schablonenbasierten Ansatz. Nachdem wir nun eine Repräsentation des MOF-Modells in Java besitzen, können wir uns um die konkreten Instanzen (z.B. das MOF-Modell einer Waschmaschine) dieses Modells kümmern. Dabei wird in MOmo jedem Element des Metamodells jeweils eine Codeschablone zugeordnet, welche im Zielcode geschrieben ist. Beim MOmo-Baukasten wird dazu noch ein Kontext erzeugt. Dieser bereitet die entsprechenden Modellelemente so auf, dass sie durch einen Schablonenmechanismus erzeugt werden können. Der Kontext bietet dabei die Möglichkeit, auf die einzelnen Eigenschaften der verschiedenen Modellelemente zuzugreifen. Eine Klassen-Schablone muss beispielsweise wissen, welchen Namen der zu erzeugende Klassenquellcode tragen soll. Betrachtet man Zeichnung 7, dann erkennt man, dass die als Schablonen vorliegenden Quellcodefragmente lediglich das Metamodell repräsentieren. Die Anpassung an das konkret gegebene Modell wird durch den Kontext ermöglicht.



Zeichnung 7: MOmo Baukasten und Schablonen

Dazu besitzt der MOmo-Generator die Steuerkomponente *Main,* welche den Generierungsprozess organisiert. Im Folgenden muss natürlich das entsprechende Modell ausgelesen werden, wofür der *Reader* verwendet wird. Auf das ausgelesene Modell kann man nun mittels der JMI-konformen Schnittstellen zugreifen. Nun muss das ausgelesene Modell noch mittels des *ContextGenerator* in einen Kontext eingefügt werden, sodass die verwendeten Schablonen alle Informationen des Modells auslesen können. Der *TemplateExcecutant* wendet nun die Schablonen auf die jeweiligen Kontexte an, um die gewünschten Artefakte (z.B. Quellcode) zu erzeugen. Noch erwähnt werden soll, dass man hier natürlich auch durch den bereits aus der Modelltransformation bekannten Mechanismus der direkten Programmierung zur automatisierten Codegenerierung nutzen kann.

7.11.1 Der Reader

Aufgabe des Readers ist es, die interne Repräsentation eines MOF-Modells zu erstellen. Dabei greift er auf die im letzten Abschnitt erarbeiteten MOF-Java-Element-Repräsentationen zurück und liefert eine Objektrepräsentation des gelesenen MOF-Modells. Der *Reader* wir durch folgende Schnittstelle beschrieben:

```
//ein Hinweis auf die Herkunft…
package de.unibw_muenchen.momoc.reader


public interface Reader {
        public void setInputPaths(java.util.List pathList);
        public java.util.Collection read();
}
```

Der Methode *setInputPaths* warden die jeweiligen Fragmente des MOF-Modells als Liste übergeben. Mit Hilfe von *read* werden die Elemente eines Eingabedokumentes kombiniert und als eine aus Instanzen der in angegebenen Klassen bestehende Objektrepräsentation zurückgegeben.

7.11.2 Der ContextGenerator

Der Kontextgenerator:

```
package de.unibw_muenchen.momoc.reader


public interface ContextGenerator
        public void init (java.util.Collection metaObjects, String modelName);
        public void generate();
        public java.util.Map getContexts ();
}
```

Die über den Reader eingelesene Objektrepräsentation wird mittels *init* übergeben. *Generate* erzeugt die mittels *getContexts* abfragbaren Kontexte. Diese können z.B. in XML-Dokumenten übergeben werden. Wir werden jetzt allerdings eine andere Form des Kontextzugriffs kennenlernen.

7.11.3 Der TemplateExcecutant

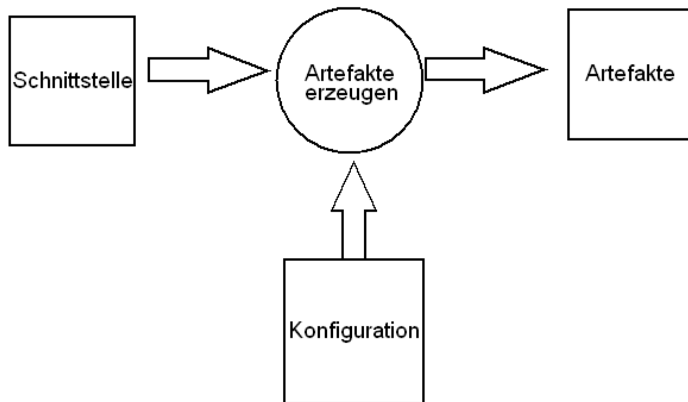
Für die Integration von Schablone und Kontext gibt es verschiedene Möglichkeiten. Man kann dafür z.B. Skriptsprachen verwenden, XSLT, Velocity etc. Wir verwenden im Folgenden Velocity, da es sich syntaktisch sehr eng an Java anlehnt und einfach zu verstehen ist. Mit Hilfe von Velocity wird einer Schablone der entsprechende Kontext hinzugefügt. Man erkennt z.B., dass der Name des Java-Interfaces, dessen Quellcode im Folgenden erstellt wird, dynamisch ist ($name).

```
import java.util.Collection;
import java.util.List;
public interface $name extends
#set ($last = $superclasses.size())
#if ($last == 0)
        javax.jmi.reflect.RefObject
#else
        #foreach ( $class in $superclasses)
                #parse ("GetFullName.vm")
                #if ($velocityCount < $last)
                ,
                #end
        #end
#end
{
#foreach ($attribute in $ownedAttributes)
        #parse ("Attribute.vm")
#end
...
} //Aus und Applaus
```

Nachdem der Name der Schnittstelle mittels $name dynamisch festgelegt worden ist, wird danach der Variablen $last die Anzahl aller Superklassen zugewiesen, indem diese durch *superclasses.size* ausgelesen wird. Gibt es keine Superklasse, erbt *$name* nur von der Standardklasse *RefObject*. Wenn doch, wird mittels *#parse („GetFullName.vm“)* ihr Name ausgelesen und wenn weitere Klassen (die ja durch Interfaces repräsentiert werden – also ist „Mehrfachvererbung“ hier möglich) folgen, wird noch ein Komma hinzugefügt.

Nun folgt der eigentliche Body der Schnittstelle zwischen { und }. Hier werden nun sämtliche Attribute durchiteriert und ausgegeben. Falls es nicht klargeworden sein sollte: Hier wird Java-Code für ein Java-Interface erstellt.

7.11.4 Der Schablonenausführer



Wir können jetzt zwar schon angepasste Codefragmente erstellen, uns fehlt aber noch eine Möglichkeit, den eigentlichen Codegenerierungsprozess zu konfigurieren.

Dies geschieht mittels der Konfiguration. In dieser können Variablen festgelegt sein, die für die Transformation wichtige Regeln enthalten. Darunter fallen:

- Die Art der zu erstellenden Artefakte
- Die Menge an Artefakten, die im aktuellen Durchlauf erstellt werden sollen
- Das Basisverzeichnis
- Die Benennung der verschiedenen Artefakte

7.12 Fazit

Die *Object Management Group* (OMG), die als Schirmherr der MDA auftritt, bietet selber keine Implementierung an, weshalb wir auch eine Implementierung, die an der Universität der Bundeswehr erstellt worden ist, herangezogen haben. Was die Verbreitung der MDA angeht, so steckt diese noch weitestgehend in den Kinderschuhen, was angesichts der vorhandenen Entwicklungstools auch kein Wunder darstellt. Auf http://java.sun.com/products/jmi/ kann man sich über den Entwicklungsstand des JMI informieren, und erfahren, dass immer noch die Spezifikation 1.0 aktuell ist. Es mag dabei auch befremden, dass Stand 05.02.2007 unter „What' s new" immer noch ein Eintrag aus dem Jahre 2002 steht.

8 Modeling Spaces (Peter Wurzler)

8.1 Einleitung

Neuere Trends in der Softwareentwicklung, wie beispielsweise UML oder MDA, beruhen auf Modellen, Metamodellen und Modelltransformationen. Die meisten dieser Bemühungen konzentrieren sich allerdings nur auf bestimmte Vorteile und lassen dabei die Art, wie Modellierung allgemein zu verstehen ist, außer Acht. Wenn über Modelle gesprochen wird, denken Softwareentwickler oft an bestimmte Modelle, zum Beispiel an UML Modelle. Andere glauben, dass dabei wichtige Fragen unbeantwortet bleiben und bisherige Antworten sich zu sehr auf bestimmte Aspekte der Modellierung beziehen. Sie schlagen ein umfassendes Framework (dt.: Gerüst) namens Modeling Spaces (dt.: Modellierungsräume) vor, um Probleme der Modellierung in einheitlicherer Weise zu untersuchen.

Modellierungsräume haben direkte Auswirkungen auf den Softwareentwicklungsprozess und bezeichnen Modellierungsmechanismen, wie sie schon von Semantic Web (OWL, RDF, Ontologien) und MDA (UML, MOF, objektorientiert) her bekannt sind, als Technical Spaces (dt.: Technische Räume).

8.2 Grundlagen

8.2.1 Modell

Für das weitere Verständnis sollte man sich auf eine eindeutige Bedeutung des Begriffs „Modell" festlegen.

Sucht man in einem Duden nach dem Wort „Modell", so findet man verschiedene Definitionen, welche oft nur die häufig genutzten, speziellen Bedeutungen des Wortes wiedergeben. Man findet unter anderem:

- Nachbildung eines Gegenstandes,
- Zeichnung eines Gebäudes,
- Fotomodell (eng.: Model) - Person, die Kleidung bei einer Modenschau trägt.

Keine dieser Definitionen ist generell anwendbar. Allerdings gibt es eine einfachere und allgemeiner gefasste Definition: Ein Modell ist eine vereinfachte Abstraktion der Wirklichkeit.

8.2.2 Modeling Architecture

Das folgende Bild zeigt eine grundsätzliche Modellierungsarchitektur (modeling architecture), hier angelehnt an MDA, und liefert damit die prinzipielle, verallgemeinerte Struktur für jeden Modellierungsraum. Dieser Aufbau besteht aus vier Ebenen (eng: layers), welche verschiedene Abstraktionsstufen der Modelle darstellen. Der Abstraktionsgrad nimmt hierbei mit jeder Ebene zu.

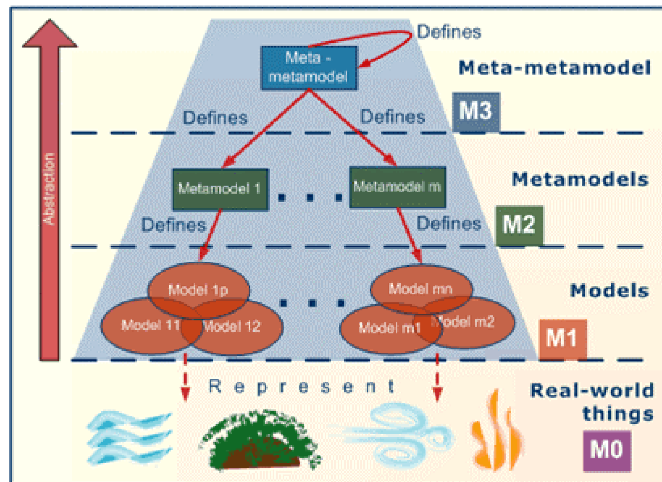


Abbildung 4 - Grundaufbau eines Modellierungsraums mit 4 Ebenen

In solch einem Modell stellt die M0 Ebene die reale Welt dar, die alle möglichen Objekte enthält, welche man durch die Modelle in der M1 Ebene darstellen will. Diese Darstellungen sind mehr oder weniger vereinfacht und abstrakt, abhängig von der Komplexität der verwendeten Modelle.

Die Modelle der M1 Ebene werden wiederum durch Nutzung der Konzepte von Metamodellen in der M2 Ebene definiert. Daher hängt es vom jeweiligen Metamodell ab, wie ausdruckstark die Modelle der M1 Ebene sein können.

Metamodelle werden auch wieder durch Konzepte definiert, welche in nächst höherer Ebene liegen. In diesem Fall also durch Konzepte der M3 Ebene an der Spitze dieser Architektur, welches Meta-Metamodell heißt. Es ist allerdings nichts anderes als ein Metamodell, welches per Konvention dazu genutzt wird, andere Metamodelle und sich selbst zu definieren.

Dieser Aufbau kann aus unbegrenzt vielen Ebenen endlicher Anzahl bestehen, wobei mit jeder Ebene ein weiteres „Meta-“ zum Modellnamen hinzugefügt wird. Auf einer M4 Ebene wäre dies also ein Meta-Meta-Metamodell. Die Anzahl der Modelle pro Ebene ist ebenfalls unbegrenzt aber endlich.

Das Meta-…-Metamodell an der Spitze einer Architektur definiert sich stets selbst und wird bei größeren Architekturen auch zur Abgrenzung als Super-Metamodell bezeichnet. Würde es von anderen Metamodellen definiert, stände es nicht an der Spitze und wäre nur ein normales Metamodell.

Diese Architektur ist verallgemeinert, um nicht nur objektorientierte Modelle und Metamodelle zu umfassen, sondern auch völlig andere Systeme darstellen zu können. So ist dieser Grundaufbau nicht nur auf MOF (Meta Object Facility) oder ähnliches beschränkt, sondern kann auch Ontologien, Semantic Web und sogar nicht-technische Inhalte darstellen.

8.2.3 Einführungsbeispiel

Im Folgenden soll ein kurzes, nicht-technisches Beispiel gegeben werden, um die grundsätzliche Funktion der Modellierungsarchitektur zu demonstrieren.

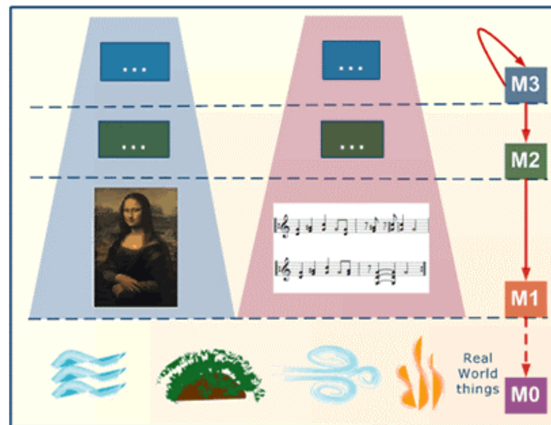


Abbildung 5 - Einführungsbeispiel

In diesem einfachen Beispiel soll es einerseits um das berühmte Gemälde „Mona Lisa“ von Leonardo da Vinci und andererseits um ein Musikstück gehen.

Eine edle Dame der Renaissance und ein Song sind Objekte der realen Welt und befinden sich daher in der Ebene M0. Ein Gemälde und ein Notenblatt sind offensichtlich nur Abstraktionen der realen Objekte und daher deren Modelle in der Ebene M1. Darüber stehen die entsprechenden Metamodelle, welche die vorhergehenden Modelle definieren. Im Fall des Musikstückes war das Modell eine geschriebene Interpretation des Songs und das Metamodell (M2) stellt nun das Konzept dar, was einzelne Noten und Zeichen bedeuten und wie man sie anordnen kann. Das Meta-Metamodell ist nun wiederum ein Konzept zur Definition der Bedeutung von Noten. Auch wenn dieser Modellierungsraum aus Sicht der Musiktheorie sicher nicht perfekt ist, stellt er doch zumindest eine formale Interpretation dar. Komplexer wird die Definition eines Metamodells für ein Gemälde und speziell für ein Meisterwerk der Kunst. Man könnte Linien, Farben und Formen definieren, kann aber nur schwer die menschliche Psychologie und Erfahrung einfließen lassen. Wir nehmen daher nur an, dass entsprechende Metamodelle existieren, aber sehr komplex und indirekt sind.

Ein anderer wichtiger Aspekt ist, dass ein Gemälde und geschriebene Noten zwar an dieser Stelle Modelle sind, aber gleichzeitig auch Objekte der realen Welt.

8.3 Modeling Spaces

8.3.1 Definition

Ein Modellierungsraum ist ein formelles, umfassendes Framework, welches aus einer Modellierungsarchitektur mit einem bestimmten Super-Metamodell besteht, welches die Kernkonzepte festlegt. Innerhalb des Modellierungsraums werden Metamodelle durch ihre Meta-Metamodelle und Modelle durch die Metamodelle definiert. Die Modelle repräsentieren dabei Objekte der realen Welt von einem gewissen Blickwinkel aus, also aus einer gewissen Sicht. Der Kontext ist hierbei der Blickwinkel des Modellierungsraum und abhängig von seinem Super-Metamodell.

8.3.2 Verständnis der realen Welt

Wenn man also Objekte der realen Welt in einem bestimmten Modellierungsraum darstellen will, nutzt man dafür bestimmte Modelle. Wenn man allerdings dieselben Objekte in einem anderen Modellierungsraum modelliert, wird man dafür andere Modelle nutzen und andere Eigenschaften hervorheben, wenn man es abstrahiert. Dies ist abhängig von den vorgegebenen Konzepten des jeweiligen Modellierungsraumes. Dadurch kann es natürlich, abhängig vom Grad der jeweiligen Abstraktion, zu Informationsverlusten bei verschiedenen Modellierungsräumen kommen.

Genauso kann ein Modell des einen Modellierungsraumes auch wieder ein Objekt der realen Welt eines anderen Modellierungsraumes sein, von dem wir ein Modell erstellen können.

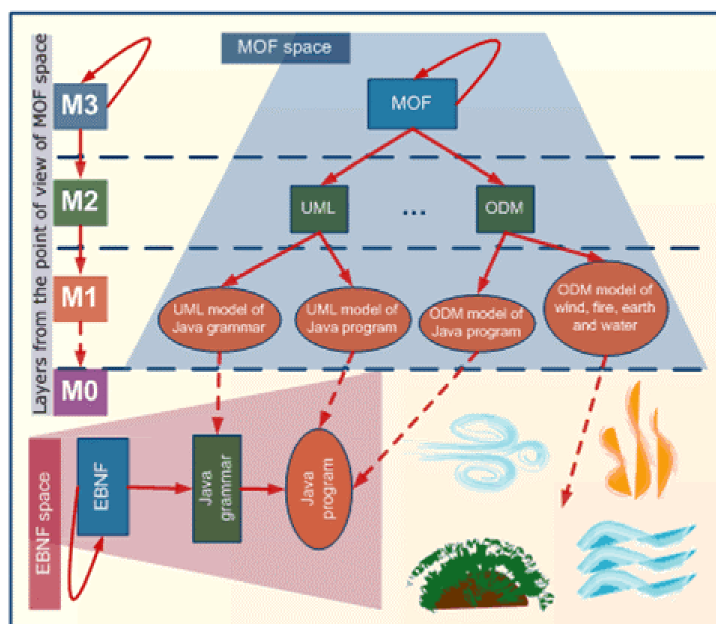


Abbildung 6 - MOF MS sieht den EBNF MS als Sammlung von Objekten der realen Welt an

Das obere Bild (Abbildung 3) zeigt die eben angesprochene Dualität auf, wobei der untere EBNF MS (Extended Backus-Naur Form Modeling Space) gleichzeitig ein Modell der realen Welt und ein Objekt der realen Welt aus der Sicht des MOF MS (Meta-Object Facilty Modeling Space) ist. Interessant ist hierbei, dass jeder Modellierungsraum aus der Sicht des MOF MS ein Teil der realen Welt ist, selbst der MOF MS selbst.

Nur weil die Ebenen über M0 Modelle sind, bedeutet es nicht, dass Meta-Metamodelle, Metamodelle und Modelle Objekte außerhalb der realen Welt sind. Genau betrachtet benutzen wir nur Konventionen, um bestimmte Dinge in entsprechende Ebenen einzuordnen. Im Prinzip liegt alles in der realen Welt, denn es kommt nur auf den auf den entsprechenden Kontext bzw. Blickwinkel an.

Beispielsweise sind Geister zwar nur Fiktion, aber gleichzeitig Objekte der realen Welt. Ansonsten wäre es uns nicht möglich, sie in Filmen, Spielen oder Literatur zu modellieren. Allerdings sind auch Filme, Spiele und Bücher ein Objekt der realen Welt, ansonsten könnten wir sie nicht in den Händen halten.

8.3.3 Arten

Modellierungsräume können auf eine mehr oder weniger abstrakte Art und Weise definiert werden. Man unterscheidet grundlegend zwei verschiedene Arten von Modellierungsräumen, abhängig von deren Funktionen.

1.1.1.1 Konzeptbezogene Räume

Konzeptbezogene Modellierungsräume (Conceptual Modeling Spaces) sind auf die semantischen Aspekte, wie Modelle, Ontologien und mathematische Logik ausgerichtet. Für sie spielt die Technik der Darstellung oder Austausch ihrer Abstraktionen keine Rolle. Ein gutes Beispiel hierfür ist der MOF MS, dessen grundsätzliche Konzepte die Klassen, Assoziationen, Attribute, Packages und deren Relationen sind. Diese werden durch ihre eigenen Konzepte ausgedrückt. Man kann diese zwar als ein UML Diagramm zeichnen, aber eine Gruppe von Boxen und Linien ist kein MOF Modell, sondern eine Zeichnung eines MOF Modells. Also handelt es sich dabei um ein Modell, welches ein MOF Modell als Objekte der realen Welt betrachtet, und von einem anderen Modellierungsraum modelliert wurde.

1.1.1.2 Konkrete Räume

Konkrete Modellierungsräume (Concrete Modeling Spaces) enthalten die nötigen Techniken, um abstrakte Daten verständlich darzustellen und sind auf die syntaktischen Aspekte ausgerichtet. Allgemeine Beispiele für solche Umsetzungen sind Datenbanken und Syntax wie EBNF, welches zwar ein durch Grammatik definierte Syntax besitzt, aber im Grunde keine Semantik. Wenn man ihm den Ausdruck *name = „Peter Wurzler“* gibt, erhält man einen Syntaxbaum, der dem Ausdruck aber keine Bedeutung zuordnen kann. Man benötigt also immer eine externe Interpretation der Bedeutung von abstrakter Syntax.

Auf der einen Seite können konkrete Modellierungsräume also reine Syntax ausgeben, benötigen aber die Semantik ihrer Daten. Auf der anderen Seite können konzeptbezogene Modellierungsräume die Semantik von Daten repräsentieren, diese Informationen aber wiederum nicht darstellen. Es ist offensichtlich, dass man die Möglichkeiten solcher Modellierungsräume kombinieren sollte, um sowohl Semantik als auch Syntax modellieren zu können. Diese Kombination wird in den Technical Spaces angewandt, wobei MDA (Model Driven Architecture) ein gutes Beispiel für die Kombination mehrerer Modellierungsräume darstellt.

8.3.4 Beziehungen / Interaktionen

1.1.1.3 Parallele Räume (Parallel Spaces)

Hierbei stellen zwei verschiedene Modellierungsräume dieselben Objekte der realen Welt dar, aber auf unterschiedliche Art und Weise, weil sie verschiedene Konzepte verwenden. In einem solchen Fall wird die Verbindung zwischen den beiden Modellierungsräume als eine reine Umwandlung untereinander betrachtet. Dabei kann jedes Modell des einen Modellierungsraumes in ein Modell des anderen Modellierungsraumes umgewandelt werden. Es ist oft nötig, komplett von einem Modellierungsraum in einen anderen zu wechseln, was über sogenannte Brücken (Bridges) geschieht.

1.1.1.4 Orthogonale Räume (Orthogonal Spaces)

Hierbei betrachtet ein Modellierungsraum einen anderen Modellierungsraum oder dessen Modelle und Metamodelle als Objekte der realen Welt, also wird ein Modellierungsraum in einem anderen Modellierungsraum dargestellt. Diese Beziehung wird häufig beim sogenannten „round-trip engineering“ (Softwareentwicklungstechnik, sorgt für Konsistenz zwischen Implementierung und Diagrammen) verwendet. Um ein Javaprogramm zu schaffen, könnte man zuerst Use-Cases erstellen, es in Java UML umsetzen, um Klassen und Methoden zu erstellen und anschließend in Java Code umwandeln (vgl. Programmierpraktikum).

8.3.5 Konkretes Beispiel

Abbildung 4 zeigt einige der Modellierungsräume, welche auch in MDA enthalten sind, sowie deren Beziehungen (Relations) untereinander. Außerdem ist noch der ähnlich aufgebaute ECore Modeling Space (Teil des Eclipse Modeling Framework) zu sehen.

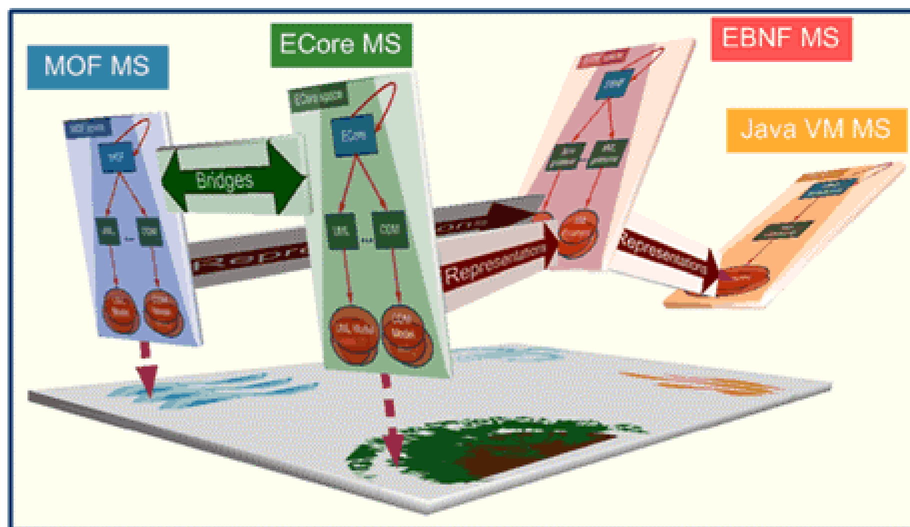


Abbildung 7 - Zwei parallele Conceptual MS, dargestellt durch einen Concrete MS

Der MOF Modeling Space und der ECore Modeling Space sind konzeptbezogene Modellierungsräume, welche unterschiedliche Konzepte nutzen, aber dieselbe reale Welt modellieren. Sie stehen also zueinander parallel und ihre Modelle können über eine Brücke ineinander umgewandelt werden. Allerdings fehlt beiden Modellierungsräumen noch die Möglichkeit, ihre Inhalte darzustellen.

Jeder dieser Modellierungsräume wird von einem weiteren konkreten Modellierungsraum als Teil der realen Welt implementiert, also aus Sicht eines orthogonalen Raums.

8.3.6 Umwandlungen / Brücken

Die Umwandlung zwischen parallelen Modellierungsräumen erfolgt über sogenannte Brücken. Diese Brücken müssen in der Lage sein, Modelle zweier paralleler Modellierungsräume ineinander zu übersetzen. Bisher haben wir aber noch keine direkte Vorstellung von deren Aufbau oder Erscheinungsform.

Diese Umwandlungen sind prinzipiell auch Modelle und sollten daher in einem Modellierungsraum stattfinden, welcher sowohl den Quell- als auch den Ziel-modellierungsraum darstellen kann. Darüber hinaus muss die Umwandlung auch durch einen orthogonalen Modellierungsraum durchgeführt werde, der die beiden parallelen Räume als reale Welt betrachtet.

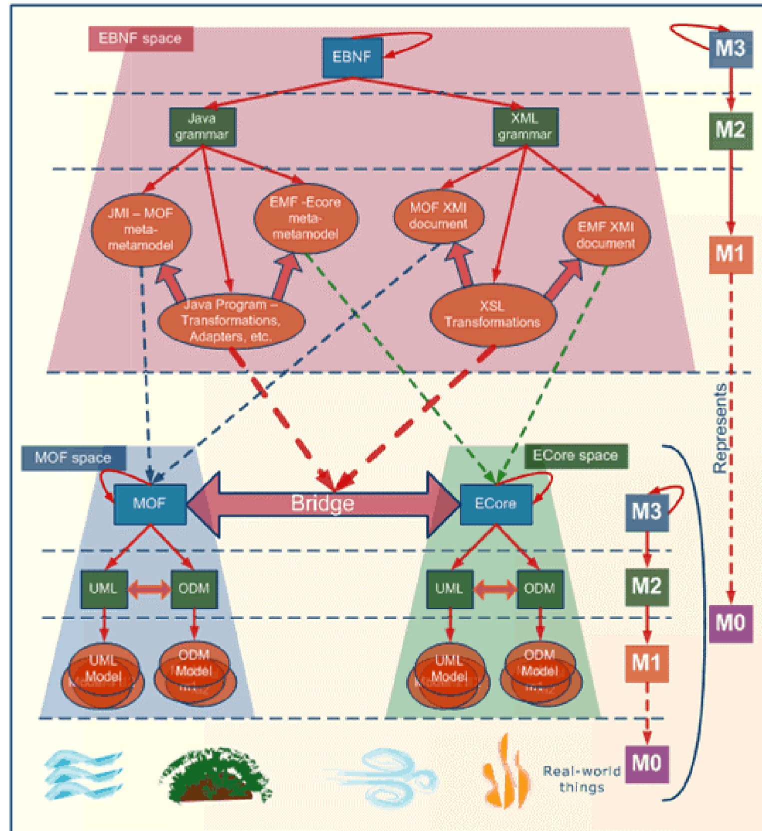


Abbildung 8 - Brücke zwischen parallelen Räumen als konkreter Modellierungsraum

Abbildung 5 zeigt ein Beispiel für einen Modellierungsraum, der wie oben beschrieben als Brück zur Umwandlung zwischen zwei parallelen Räumen fungiert. Konkret sind in dem Beispiel die parallelen Modellierungsräume MOF und ECore zu sehen, welche durch einen orthogonalen EBNF Modellierungsraum repräsentiert werden.

8.4 Anwendung

8.4.1 Technical Spaces

Wie bereits zu Anfang erwähnt, lehnt sich das Konzept der Modellierungsräume (Modeling Spaces) direkt an das Konzept der Technischen Räume (Technical Spaces) an. Als technischen Raum bezeichnet man einen Arbeitskontext, welches sich aus zusätzlichen Konzepten, Wissen, geeigneten Werkzeugen und den benötigten Fähigkeiten zusammensetzt.

Diese unklare Definition liefert allerdings keine präzise Definition für technische Räume, ermöglicht aber es zumindest deren Erkennung, wie beispielsweise den MDA TS, den XML TS, den Java TS und den Ontology Engineering TS. Mithilfe der

Modellierungsräume kann man allerdings eine weit genauere Definition von technischen Räumen angeben.

Ein technischer Raum ist ein Arbeitskontext, welcher verschiedene, untereinander in Beziehung stehende Modellierungsräume beinhaltet. Oft wird ein technischer Raum um einen bestimmten Modellierungsraum herum aufgebaut, wobei die Funktion der anderen Modellierungsräume nur unterstützend (z.B. Implementierung) ist.

Daraus folgt, dass ein technischer Raum aus ein oder mehr Modellierungsräumen besteht und jeder Modellierungsraum Teil von einem oder mehreren technischen Räumen ist. Die Technischen Räume können dabei Modellierungsräume zusammenfassen, welche Gemeinsamkeiten haben oder sich einfach gegenseitig zur Interaktion benötigen.

Falls eine Brücke zwei Modellierungsräume miteinander verbindet, welche in verschiedenen technischen Räumen liegen, so stellt diese Brücke auch eine Verbindung zwischen den entsprechenden technischen Räumen dar.

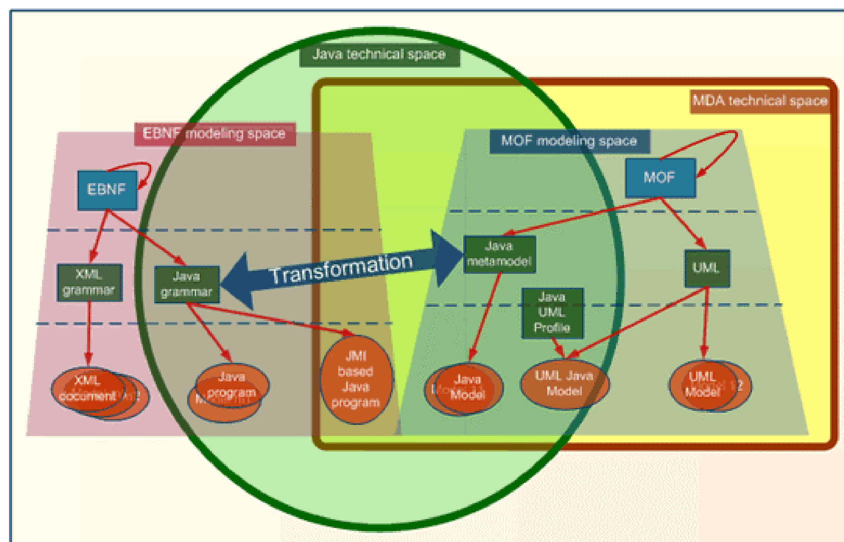


Abbildung 9 - Technische Räume umfassen ein oder mehr Modellierungsräume

1.1.1.5 Model Driven Architecture Technical Space (MDA TS)

Wie in Abbildung 6 zu sehen, ist der MDA TS komplett um den MOF Modellierungsraum aufgebaut, welcher vollständig im Zentrum des MDA TS steht. Allerdings enthält der MDA TS auch teilweise andere Modellierungsräume, wie beispielsweise XML und zur XMI Darstellung und EBNF zur Implementierung der MOF-Modelle in Java über JMI.

1.1.1.6 Java Technical Space (Java TS)

Der Java TS enthält zum Beispiel einen Teil des EBNF Modellierungsraumes, welche mit der Java Grammatik oder Java Programmen verbunden sind, und einen Teil des MOF Modellierungsraumes, welcher für Java Metamodelle und Java UML benötigt wird. Außerdem ist innerhalb des Java TS eine direkte Umwandlung von MOF-Modellen in Java-Code und umgekehrt möglich (Abb.6 - Pfeil „Transformation").

8.4.2 Nutzen

Ingenieure und Entwickler können durch Modellierungsräume ein besseres Verständnis für die vielen, unterschiedlichen Dinge bekommen, welche man modellieren kann. Dieses Framework bietet Softwareentwicklern darüber hinaus eine umfassende begriffliche Basis, da sie die Funktionen verschiedener Modellierungstechniken übersichtlich darstellt, erklärt und zeigt, wie man diese kombinieren kann.

In der Praxis werden oft komplexe und vielfältige Entwicklungstools benutzt, um große Bereiche der Softwaremodellierung abzudecken. Beispielsweise wurden im Programmierpraktikum entsprechende Tools für Javacode, Datenbanken, UML Diagramme und spezielle Technologien wie J2EE genutzt. Durch derartige Tools ist es scheinbar nicht mehr nötig, sich um die Beziehungen zwischen den einzelnen Bereichen zu kümmern und man wählt einfach die entsprechende Funktion für die bevorzugte Entwicklungsumgebung. Dennoch fällt es meist schwer, den Überblick zu behalten, auch wenn entsprechende Tools problemlos zwischen einzelnen Programmieraspekten wechseln können. In manchen Fällen besitzen die vorhandenen Tools auch nicht die nötigen Funktionen. Dann wird es sehr aufwändig, die entsprechenden Umwandlungen per Hand durchzuführen oder sogar eigene Tools für spezielle Fälle zu entwickeln. Gerade in solchen Fällen sind Modellierungsräume eine willkommene Hilfe, um sich komplexe Zusammenhänge zu verdeutlichen und den Überblick zu bewahren.

8.5 Quellenangaben

- D. Djuric, D. Gasevic, V. Devedzic: "The Tao of Modeling Spaces", in *Journal of Object Technology*, Vol. 5. No. 8, November-December 2006, pp. 125-147. http://www.jot.fm/issues/issue_2006_11/article4
- D. Gasevic, D. Djuric, V.Devedzic: “Model Driven Architecture and Ontology Development”, Springer 2006