Bewertung des Jena-Frameworks beim Einsatz von Ontologien

Baccalaureatsarbeit
Sommersemester 2003
Universität Hamburg
Fachbereich Informatik

Hauke Loock
An der Eilwettern 4
21723 Hollern-Twielenfleth
Matrikelnummer: 5336035

Betreuung: Dr. Ralf Klischewski

Inhalt

1. Einleitung

Diese Arbeit entstand im Rahmen des Projektes „Software, Wissen und Organisation“ im Wintersemester 2002/03 und Sommersemester 2003 im Arbeitsbereich Softwaretechnik an der Universität Hamburg unter Betreuung von Herrn Dr. Ralf Klischewski und Herrn Martti Jeenicke. In dem Projekt ging es um die Kontextualisierung von Informationen, die über das Internet angeboten werden und der vom Benutzer eingegebenen Daten. Im Internet vorhandene Informationen sollen mit Daten, die der Benutzer während seiner Internet-Recherche (bei einem Anbieter) angibt, in einen Zusammenhang gesetzt, also zu einem Kontext verknüpft, werden. Zudem soll durch die Speicherung der Benutzereingaben die Wiederverwendbarkeit in demselben oder einem anderen Kontext (während einer Sitzung oder später in einer neuen Sitzung) gefördert werden. Dies wurde am Beispiel eines (fiktiven) Umzugs und der Direkten Bürger Informations Services (DiBIS, http://dibis.dufa.de/) der Stadt Hamburg betrachtet.

Zur Realisierung der Verknüpfung von Daten und Benutzerinformationen standen uns das Resource Description Framework (RDF), eine im Zuge des Ansatzes des Semantic Web entstandene Sprache zur Modellierung von Informationen vom World Wide Web Consortium (W3C), und Topic Maps, ein ISO Standard entwickelt vom TopicMaps.org Konsortium, zur Auswahl. Im Semantic Web wird versucht dem Web zusätzlich zur Verknüpfungsstruktur (durch Hyperlinks) auch Bedeutung (Semantik) zu verleihen, damit Softwareagenten die Informationen „verstehen“ und dadurch effizienter benutzen können. Diese Struktur wird vielfach auch als Ontologie bezeichnet; auf den Begriff der Ontologie gehe ich in Kapitel drei genauer ein. Wir haben uns schließlich für eine Erweiterung von RDF bzw. RDF Schema, nämlich DAML+OIL, entschieden. Zur Implementierung haben wir das Jena Semantic Web Toolkit, ein Java Rahmenwerk, das RDF und DAML+OIL implementiert, benutzt; im Folgenden werde ich dieses als Jena-Framework bezeichnen.

Unsere Ontologie, die zur Navigation auf einer Webseite und der Speicherung der Benutzerdaten in Instanzen der in der Ontologie angelegten Begrifflichkeiten dienen soll, haben wir an Hand von Szenarien, die im Projekt geschrieben wurden, erstellt. Bei der Umsetzung dieser (grafischen) Ontologie in eine vom Computer verarbeitbare Form, nämlich in Java-Code, der DAML+OIL Syntax implementiert, mit Hilfe des Jena-Frameworks, traten einige Probleme zu Tage. Beim Erstellen der Ontologie war noch nicht klar, welche der verfügbaren Modellierungssprachen wir benutzen wollen. Daher wurde eine einfache grafische Notation gewählt. Auch stellte sich heraus, dass es Probleme bei der Modellierung der Ontologie gab, und auch der Umgang mit dem Jena-Framework stellte uns vor einige Herausforderungen.

In meiner Arbeit möchte ich den Einsatz des Jena-Frameworks in unserem Projekt betrachten. Dazu möchte ich darstellen, welche Probleme im Laufe unseres Projektes mit dem Jena-Framework auftraten und untersuchen was die Ursachen gewesen sein könnten. Das Ziel hierbei ist es, festzustellen, wie gut sich das Jena-Framework für den Umgang mit einer Ontologie in DAML+OIL Syntax eignet.
Beginnen werde ich damit, das Vorgehen in dem Projekt „Software, Wissen und Organisation“ zu beschreiben. Im Projekt haben wir über selbst erstellte Szenarien und das Herausarbeiten von Infoobjekten eine eigene Ontologie für den Bereich „Umzug“ erstellt. Diese Ontologie haben wir dann mit Hilfe des Jena-Frameworks versucht zu implementieren. Außerdem beschreibe ich in diesem Kapitel die Architektur unseres Prototypen und den Umgang mit der Ontologie im Prototypen.
Um zu einer Bewertung des Jena-Frameworks beim Einsatz von Ontologien zu gelangen, werde ich im dritten Kapitel zunächst darstellen, was eine Ontologie ist und wozu man diese

benutzen kann. Im Anschluss daran stelle ich die Modellierungssprachen, mit denen wir uns im Projekt beschäftigt und von denen wir dann schließlich eine ausgewählt und benutzt haben, kurz vor. Dies sind RDF, RDF Schema und DAML+OIL. Bei ersteren geht es mir nur darum, die Grundlagen kurz vorzustellen, da DAML+OIL, das wir benutzt haben, auf diesen aufbaut und sie erweitert. Auf DAML+OIL werde ich etwas genauer eingehen, da dieses die Grundlage unserer Arbeit bildet.
Darauf folgt ein Kapitel, in dem ich die Grundzüge des Jena-Frameworks erläutere. Das Jena-Framework bildete im Projekt die Grundlage für unsere technische Realisierung der Ontologie und soll in meiner Arbeit als konkretes Beispiel für die möglicherweise auftretenden Probleme beim Einsatz von Ontologien dienen.
Das folgende Kapitel widmet sich schließlich der angestrebten Bewertung des Jena-Frameworks. In diesem Abschnitt werde ich versuchen die Probleme, die sich im Laufe der Arbeit im Projekt ergaben, und die auf Defizite des Jena-Frameworks oder auf Verständnisprobleme unsererseits zurückzuführen sind, aufzuzeigen. Ich möchte dabei herausarbeiten, inwiefern sich das Jena-Framework dazu eignet eine Ontologie, die der DAML+OIL Spezifikation folgt, zu implementieren. Dies soll am Beispiel unseres Projektes geschehen. Ich möchte dabei versuchen, die Möglichkeiten der DAML+OIL Spezifikation den Möglichkeiten des Jena-Frameworks an Beispielen gegenüber zu stellen.
Abschließend gebe ich noch eine Zusammenfassung und einen kurzen Ausblick auf Jena2, die Weiterentwicklung des Jena-Frameworks, und OWL (Web Ontology Language), eine Sprache für die DAML+OIL als Ausgangspunkt diente und auf deren Entwicklung DAML+OIL bedeutenden Einfluss hatte.

2. Das Projekt

In diesem Kapitel möchte ich das Projekt „Software, Wissen und Organisation“ vorstellen. Dabei beschreibe ich die Zielsetzung des Projektes und unser Vorgehen bei der Entwicklung des Prototypen für ein Web-Basiertes Informationssystem.

2.1 Zielsetzung des Projektes

Die Zielsetzung des Projektes war es einen Prototypen eines Web-Basierten Informationsservices zu entwickeln, der Kontextinformationen des Benutzers sowohl für das Zusammenstellen der gesuchten Informationen benutzen kann als auch in der Lage ist diese für spätere Sitzungen zu speichern. Unter *Kontextinformationen* verstehen wir die Informationen, die der Benutzer während seiner Sitzung durch die Eingabe von Texten oder das Auswählen bestimmter Links dem Informationsservice zur Verfügung stellt. Eine weitere Funktion, die bei einer Weiterentwicklung des Prototypen hinzugefügt werden könnte, ist die Möglichkeit Informationen an andere Web-Basierte Informationsservices weiterzugeben und die Zusammenarbeit zwischen diesen zu unterstützen.
Als Anwendungsbeispiel dienten uns die Direkten Bürger Informations Services (DiBIS, http://dibis.dufa.de/) der Stadt Hamburg, die zum Zeitpunkt unseres Projektes noch keinen Gebrauch von Kontextinformationen machten.
Im Laufe des Projektes machten wir uns dabei auch mit Technologien, die aktuell im Zuge des Semantic Web oder allgemein bei der Entwicklung von Web-Anwendungen zum Einsatz kommen, vertraut. Unter diesen Punkt fielen unter anderem RDF, DAML+OIL, Topic Maps und Servlets.

2.2 Kontextualisierung

Unser Hauptanliegen war die Kontextualisierung von Web-Basierten Informationsservices. Unter *Kontextualisierung* verstehen wir die Nutzung von Informationen, die der Nutzer in das System eingibt, zur Bestimmung der Ausgabe. Durch die Kontextualisierung sollen der Komfort und die Effizienz des Informationsservices für den Benutzer verbessert werden.
Der Kontext wird während einer Session aus den Eingaben von Text oder die Auswahl bestimmter Links durch einen Nutzers aufgebaut. Dadurch, dass dem Informationsservice dieser Kontext zur Verfügung steht, können die angezeigten Informationen besser auf den Benutzer zugeschnitten werden. Es soll verhindert werden, dass dem Benutzer Informationen angezeigt werden, die in dem Kontext nicht von Bedeutung sind. Außerdem soll es dem Benutzer möglich sein jederzeit die über ihn bzw. von ihm gespeicherten Informationen einzusehen und gegebenenfalls zu verändern oder zu löschen. Hierdurch soll unter anderem das Vertrauen der Benutzer in den Informationsservice aufgebaut werden.
Eine Verbesserung des Komforts wird außerdem dadurch erreicht, dass Daten, die der Benutzer schon einmal eingegeben hat, später nicht noch einmal eingegeben werden müssen, sondern die dem System schon bekannten Daten automatisch in die entsprechenden Felder eingesetzt werden.

2.3 Das Vorgehen im Projekt

Im Projekt haben wir damit begonnen, uns einen Überblick über unser Anwendungsbeispiel, die Direkten Bürger Informations Services, zu verschaffen. Dazu hatten wir diverse Vorträge, in denen uns über den Entwicklungsstand zum damaligen Zeitpunkt und über mögliche Weiterentwicklungen berichtet wurde. Des Weiteren haben wir mit einer fiktiven Problemstellung selbst ausprobiert, wozu die bisherigen DiBIS in der Lage waren. Wir kamen zu folgendem Ergebnis: Die DiBIS lieferten zu diesem Zeitpunkt zu einem Anliegen das jeweilige zuständige Amt mit Öffnungszeiten und Zufahrtsmöglichkeiten. Es wurden keine Kontextinformationen genutzt, eventuell wurde die Straße abgefragt und damit das zuständige Amt ermittelt. Es bestand zwar die Möglichkeit Formulare herunter zu laden, aber das Ausfüllen und online Weiterleiten der Formulare war nicht möglich. Aus dem Vortrag erfuhren wir, dass die Daten bisher nur sehr einfach strukturiert waren.
Um uns darüber klar zu werden, welche Abläufe in einem Web-Basierten Informationsservice in verschiedenen Lebenslagen vorkommen können, haben wir verschiedene Szenarios zu diesen verfasst.
Ein *Szenario* ist ein Prosatext in der Sprache des Anwendungsbereiches. In ihnen werden die Arbeitssituationen des Anwendungsbereiches, der Zweck bestimmter Handlungen und die dafür benötigten Mittel beschrieben [Zue01]. Szenarios dienen dem Entwickler dazu, den Anwendungsbereich zu verstehen und die wichtigen Konzepte des Anwendungsbereiches zu identifizieren.
Unter einer *Lebenslage* verstehen wir den Zustand inklusive aller Begleitumstände, in dem sich eine Person gerade befindet. Dies kann zum Beispiel die Lebenslage Hochzeit sein oder, wie für unser Projekt als Beispiel gewählt, ein Umzug. Aus einer Lebenslage ergeben sich auch alle Handlungen, die von einem aktuellen Zustand aus ausgeführt werden können.
An Hand des Szenarios haben wir dann dessen verschiedene Infoobjekte identifiziert. *Infoobjekte* sind die Informationen, die im Szenario über die Hauptperson, auf die sich die Lebenslage bezieht, vorhanden sind und die Informationen, die

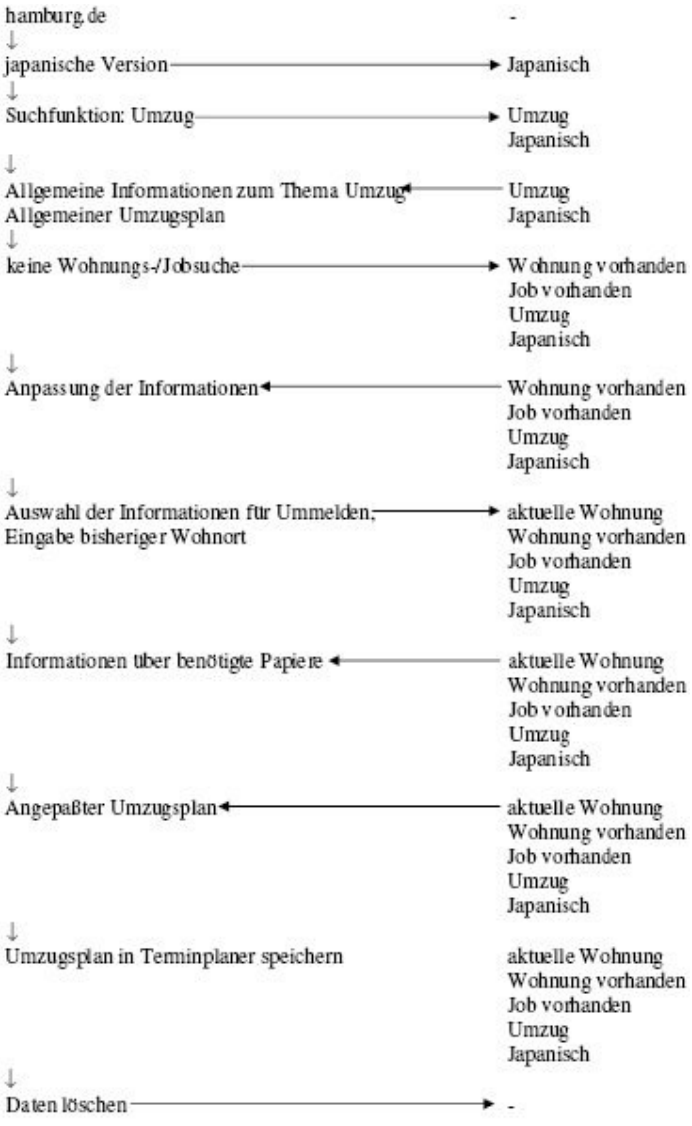


Abbildung 2.1: Der Lebenszyklus von Infoobjekten

vorher schon im System vorhanden sind. Das sind die Informationen, die während einer Sitzung im Informationsservice über den Benutzer anfallen (z. B. Name und Adresse) und die Informationen, die das Informationssystem dem Benutzer liefert (z. B. Öffnungszeiten eines Amtes).
Für ein Informationssystem ist es natürlich auch von Bedeutung was während einer Sitzung mit den Infoobjekten weiter geschieht. Dazu haben wir untersucht, wie die Infoobjekte während der Sitzung zu- oder abnehmen und welchen Einfluss sie auf die angezeigten Informationen haben. In Abbildung 2.1 ist ein Beispiel für die Infoobjekte und deren Lebenszyklus in unserem Umzugsbeispiel zu sehen.
Auf der linken Seite der Abbildung ist der Ablauf der Sitzung zu sehen. Hier sind die Benutzereingaben und die entsprechenden Reaktionen des Informationsservices aufgeführt. Auf der rechten Seite der Abbildung befinden sich die Infoobjekte, also die Informationen über den Benutzer, die sich während der Sitzung ansammeln. Jeder Aktion des Benutzers und jeder Reaktion des Systems sind dabei die zu dem Zeitpunkt vorhandenen Infoobjekte zugeordnet.

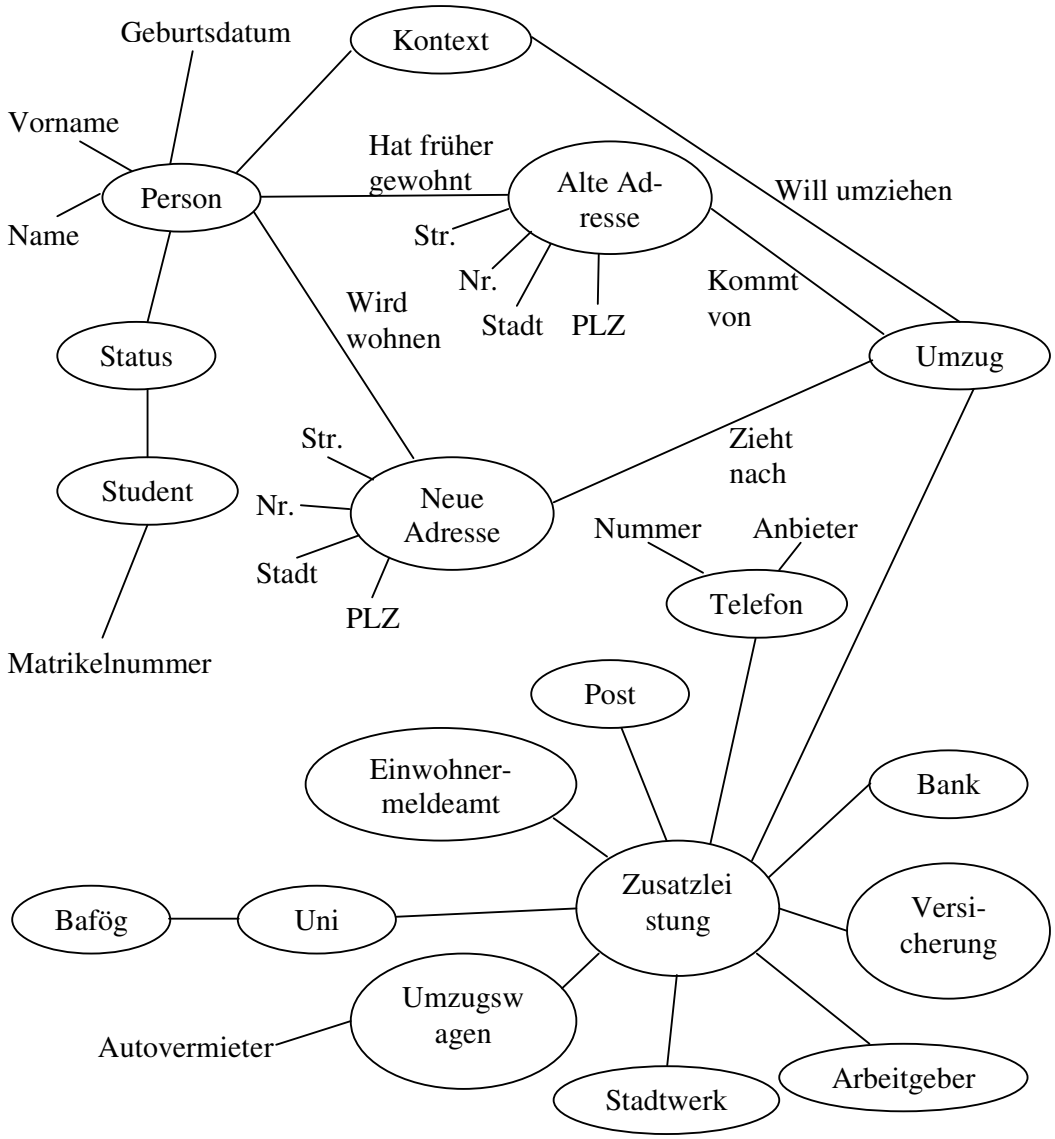


Abbildung 2.2: Ein erster Versuch für ein fachliches Modell des Anwendungsbereiches

Ein Pfeil von links nach rechts bedeutet, dass der Benutzer Informationen über sich Preis gibt, die dem System dann bekannt sind. Diese Informationen können vom System dazu benutzt werden bestimmte Informationen auszuwählen oder andere auszublenden. Dies wird durch einen Pfeil von rechts nach links symbolisiert. Durch diese Nutzung der Infoobjekte durch das Informationssystem wird die Kontextualisierung erreicht.
Nachdem wir den Lebenszyklus von Infoobjekten und die Beziehungen zwischen diesen und den angezeigten Informationen untersucht hatten, haben wir uns daran gemacht aus den gewonnen Erkenntnissen ein grafisches Modell für die Lebenslage aus unserem Szenario zu entwickeln. Dieses erste Modell ist in Abbildung 2.2 dargestellt.
Dieses grafische Modell der Lebenslage diente uns später als Ausgangspunkt für die Erstellung einer Ontologie für unseren Informationsservice. Zunächst mussten wir uns jedoch darüber Gedanken machen, wie wir unser fachliches Modell erstellen wollten.
Zur Auswahl für die Erstellung dieses Modells standen uns Topic Maps und RDF zur Verfügung. Details zu RDF sind in Abschnitt 3.3 zu finden und zu Topic Maps in Abschnitt 3.7.
Um eine Entscheidung darüber treffen zu können, welche dieser Modellierungsmöglichkeiten wir für die Erstellung unserer Ontologie benutzen wollten, haben wir uns über beide einen Überblick verschafft und versuchsweise einen Ausschnitt des oben gezeigten Modells (in Abbildung 2.2) in RDF bzw. eine Topic Map übertragen (siehe Abbildung 2.3).
Im Beispiel für die Topic Map (links in Abbildung 2.3) stehen die Rechtecke für die Topics und die Verbindungen zwischen ihnen für die Assoziationen. Diese sind mit den Rollen und dem Namen der Assoziation beschriftet. Die Topic Map liefert ein Schema dafür, wie (in diesem Fall) eine Person angelegt werden soll.
Im RDF Beispiel (Abbildung 2.3 rechts) stehen die Ellipsen für ein Subjekt, die Rechtecke für ein Objekt und die Pfeile zwischen ihnen für das Prädikat. In diesem Fall wird eine konkrete Person angelegt (und kein Schema), was dadurch symbolisiert werden soll, dass als Beschriftung Name X, Vorname X usw. gewählt wurde.

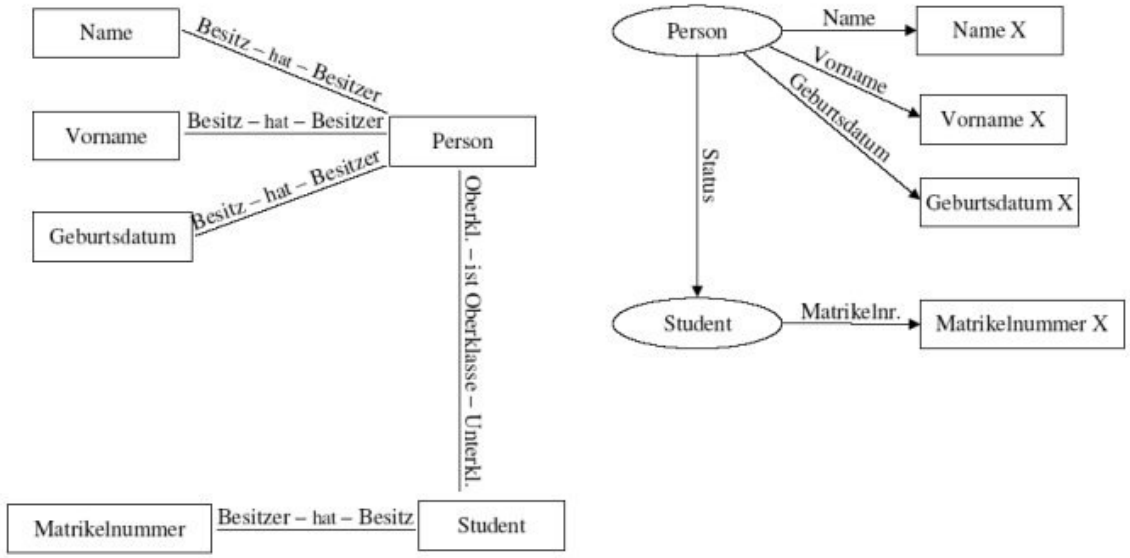


Abbildung 2.3: Ein Ausschnitt unseres Szenarios als Topic Map (links) und RDF (rechts)

Uns schien RDF besser geeignet, da es schon etablierter als Topic Maps war und Unterstützung durch das World Wide Web Consortium (www.w3c.org) erfährt.
Nachdem feststand, welche Modellierungsmethode wir benutzen wollten, mussten wir ein geeignetes Framework auswählen. Die Entscheidung fiel auf das Jena-Framework (Version

1.6.0), da es die gewünschten Funktionalitäten aufwies und es zusätzlich zu den Java-Dokumentationen auch ein Tutorial gab.
Die für den Ontologie-Entwurf zuständige Gruppe entwarf unterdessen eine große Ontologie für die Lebenslage Umzug auf der Basis unserer bisherigen Entwürfe, die dann in unserem Prototypen implementiert werden sollte. Diese Ontologie wurde schon komplett in RDF entworfen. Es wurde dabei allerdings auch festgestellt, dass uns das nötige Wissen über die fachlichen Hintergründe fehlt.
Damit die Ontologie im fertigen System auch vom Benutzer an seine Bedürfnisse anpassbar ist, ohne dass er spezielle Informatikkenntnisse besitzt, sollte es ein einfach zu bedienendes Tool zum Editieren der Ontologie geben. Dieses Tool haben wir aber nicht weiter behandelt, da dafür im Projekt keine Zeit mehr war.
Nun ging es darum festzulegen, wie unser System aufgebaut sein sollte. Unser Entwurf für die Architektur sah folgendermaßen aus:

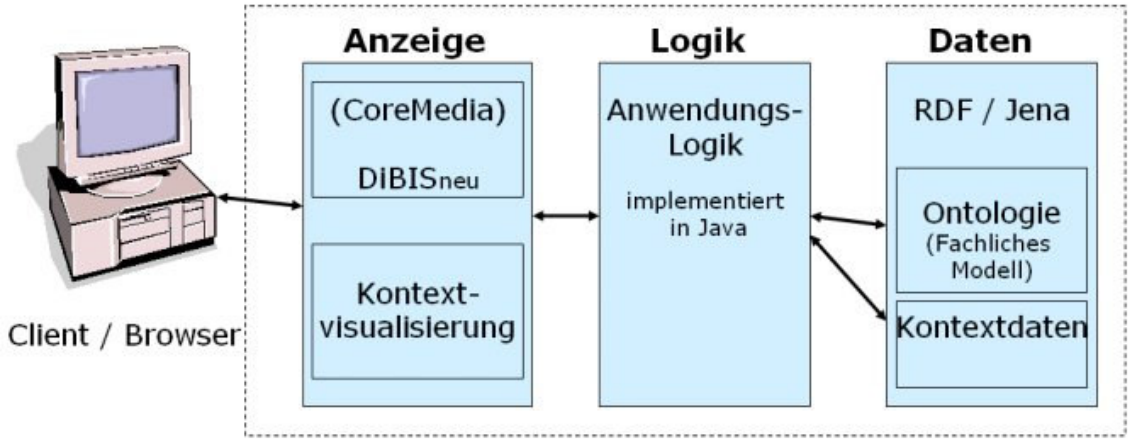


Abbildung 2.4: Die Architektur unseres Prototypen

Die Architektur besteht aus den drei Schichten Anzeige, Logik und Daten.
Die Anzeige-Schicht ist für die Erstellung der HTML-Seiten aus den von der Logik gelieferten Informationselementen und die Visualisierung der gespeicherten Kontextinformationen des Benutzers, also der Infoobjekte, zuständig.
Die Logik-Schicht ist dafür zuständig, mit Hilfe der Benutzereingaben und der Ontologie, die Informationen, die angezeigt werden sollen, auszuwählen.
In der Daten-Schicht werden sowohl die Ontologie als auch die Kontextdaten des Benutzers während der Sitzung gespeichert.
Eine Anfrage läuft also folgendermaßen ab: Der Benutzer startet in seinem Browser die Anfrage durch Auswahl eines angezeigten Links oder Eingabe von Daten. Die Anfrage wird dann an die Logik-Schicht weitergeleitet. Diese benutzt dann die Ontologie und eventuell schon eingegebene Kontextdaten, um die Anfrage des Benutzers auszuwerten und die gewünschten Informationen auszuwählen oder gegebenenfalls weitere Daten vom Benutzer abzufragen. Die berechneten Informationen oder Rückfragen werden daraufhin an die Anzeige-Schicht weitergegeben, die diese zu einer neuen HTML-Seite zusammensetzt, die dann vom Browser des Benutzers dargestellt wird. Dort beginnt der Prozess wieder von vorn.
Unabhängig von den angezeigten kontextualisierten Informationen werden dem Benutzer immer die Informationen über ihn, die dem System zum jeweiligen Zeitpunkt bekannt sind, angezeigt (siehe auch [Dei03]).
Das Zusammenspiel der Ontologie und der Kontextinformationen ist in Abbildung 2.5 dargestellt und funktioniert wie folgt:

Die Ontologie stellt ein Schema für eine Lebenslage dar. In ihr ist die grundsätzliche Struktur der Lebenslage gespeichert, aber noch keine konkrete Ausprägung der Lebenslage für eine Person behandelt.
Während einer Sitzung mit einem Informationssystem, das die Ontologie benutzt, wird die Ontologie in Teilen oder im Ganzen instanziiert. Das bedeutet, dass von den betroffenen Elementen der Ontologie eine Kopie angelegt und mit einem konkreten Wert belegt wird (erster Pfeil in Abbildung 2.5). Mit Benutzereingaben instanziiert werden z. B. Elemente wie Name, Geburtstag oder Adresse, wobei die Adresse selbst wieder aus den Elementen Strasse, Postleitzahl usw. besteht. Allerdings werden nicht alle Elemente der Ontologie mit Benutzereingaben belegt. Für manche Elemente wird die Instanz erzeugt, wenn der Benutzer dieses Element auswählt, damit das System weiß, dass dieses Element für den Benutzer relevant ist. Diese Elemente erhalten als Wert dann z. B. einen Zeitstempel. Ein solches Element könnte z. B. Telefon ummelden sein, das nur von Interesse ist, wenn der Benutzer ein Telefon hat und sonst auch keine Informationen über dieses Thema angezeigt werden müssen. Diese Elemente dienen also dazu, die Informationen, die der Informationsservice bereitstellt, in den Kontext einzubinden.
In einer Sitzung wachsen die Informationen über den Benutzer mit jeder Eingabe oder Auswahl, die er tätigt, immer mehr an oder werden durch ihn verändert (zweiter Pfeil in Abbildung 2.5); dies entspricht dem Lebenszyklus von Infoobjekten in Abbildung 2.1. Dem Benutzer soll es hierbei auch möglich sein schon getätigte Eingaben wieder zu ändern oder zu löschen.
Schließlich kann der Benutzer die Daten, die er im System belassen hat, abspeichern oder an andere Dienste zur weiteren Bearbeitung weitergeben. Dabei sollen Idealerweise nur die Elemente abgespeichert werden, die instanziiert sind (rechte Seite von Abbildung 2.5). Beim nächsten Aufruf des Informationssystems durch denselben Benutzer werden dann die gespeicherten Daten wieder mit dem Schema in Verbindung gebracht. Diese Funktionen wurden allerdings für unseren Prototypen noch außer Acht gelassen.

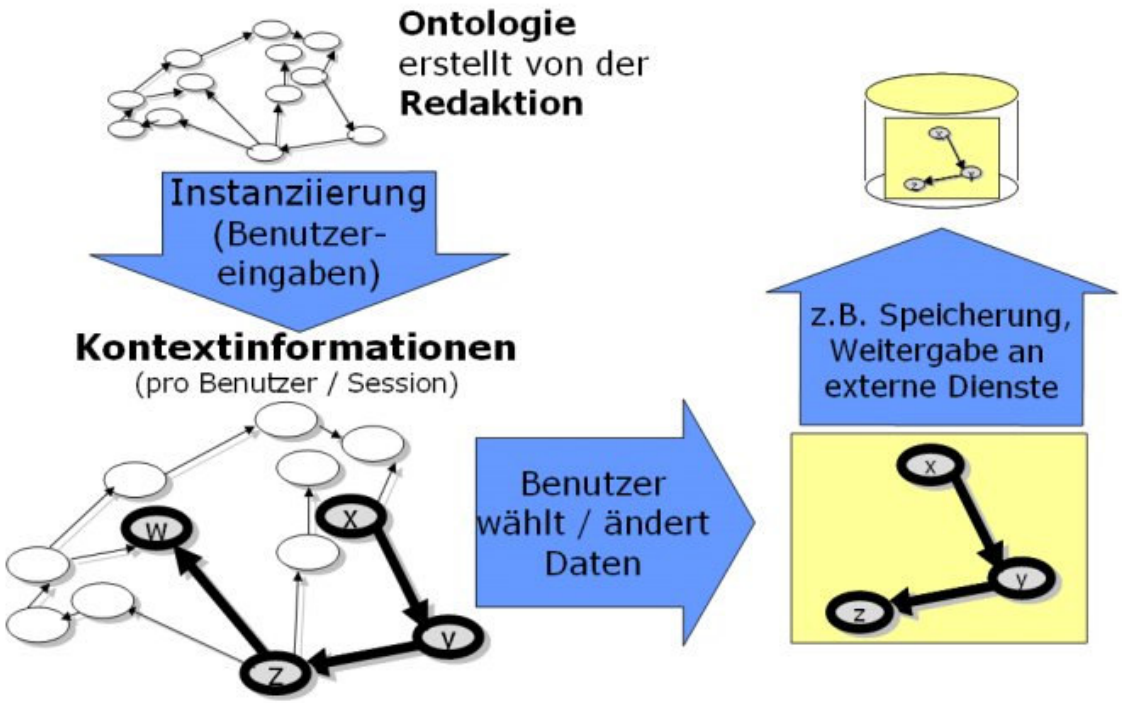


Abbildung 2.5: Der Umgang mit der Ontologie und den Kontextinformationen im Prototypen

Daraus, dass unser Entwurf für den Umgang mit den Kontextinformationen ein Schema aus verschiedenen Klassen, die nur bei Bedarf instanziiert werden, vorsieht, ergab sich die Notwendigkeit RDF Schema anstatt einfachem RDF zu benutzen, da das Anlegen von Klassen erst mit RDF Schema möglich wird (siehe Abschnitt 3.4). Da das Jena-Framework eine direkte Unterstützung von DAML+OIL (siehe Abschnitt 3.5), was ein nochmals erweitertes RDF Schema ist, anbot, entschlossen wir uns dies zu verwenden.
Da die Architektur unseres Prototypen nun entworfen war, machten sich die verschiedenen Gruppen daran ihren Teil des Prototypen zu implementieren und die Schnittstellen untereinander zu vereinbaren. Die Aufgabe meiner Gruppe bestand darin, die Ontologie in DAML+OIL zu implementieren und Methoden für die Suche in der Ontologie und die Manipulation der Ontologie bereitzustellen. Dies sollte, wie gesagt, mit dem Jena-Framework erreicht werden. Auf die Probleme, die dabei im Umgang mit dem Jena-Framework im Laufe des Einsatzes im Projekt auftraten, werde ich im fünften Kapitel näher eingehen.

3. Der Begriff Ontologie und Modellierungssprachen zur Erstellung von Ontologien

In diesem Abschnitt meiner Arbeit werde ich zunächst darlegen, was unter einer Ontologie zu verstehen ist. Darauf folgend werde ich Standards, die man bei der Erstellung von Ontologien benutzt, vorstellen. Diese wären das Resource Description Framework (RDF), RDF Schema (RDFS), die DARPA Agent Markup Language + Ontology Inference Layer (DAML+OIL) Spezifikation und Topic Maps. RDF, RDFS und DAML+OIL werden vom World Wide Web Consortium (W3C) unterstützt, wohingegen Topic Maps ein ISO-Standard ist.

3.1 Ontologie – Ein Erklärungsversuch

Der Begriff *Ontologie* leitet sich vom griechischen onta (das Seiende) und logos (Lehre) her und bedeutet „die Lehre vom Seienden". In der Philosophie ist die Ontologie ein Teilgebiet der Metaphysik, welche sich mit der Seinsstruktur der Wirklichkeit und den Gründen des Seins beschäftigt. Die Ontologie selbst beschäftigt sich mit Fragen nach der Natur des Seins (nature of existence) und der Struktur der Realität.
Die Geschichte der Metaphysik bzw. der Frage nach dem Sein lässt sich bis in die Antike zu Aristoteles oder Platon zurückverfolgen. Später beschäftigten sich Philosophen wie Descartes, Kant, Leibniz und viele andere mit diesem Thema. Dementsprechend gibt es viele voneinander abweichende Ansichten dazu, auf die ich hier aber nicht näher eingehen möchte, da dies den Rahmen der Arbeit sprengen würde. [Ber72], [Div03]
Der Begriff Ontologie wurde von Entwicklern auf dem Gebiet der Künstlichen Intelligenz und der Web-Entwicklung für die Informatik übernommen und angepasst. Gruber bezeichnet eine Ontologie auch als „explicit specification of a conceptualization" [Gru93].
Für die Informatik besteht eine Ontologie aus einem Vokabular, das eine Domäne/ein Universum beschreibt, und aus Assoziationen, die die Beziehungen der Terme des Vokabulars beschreiben. Außerdem werden für das Vokabular die gewünschten Bedeutungen explizit vorgegeben. Dies geschieht üblicherweise als logische Ausdrücke oder in einer Logik-basierten Sprache (Terme aus dem Vokabular haben die Form von Prädikaten und Relationen). Mit Hilfe der Logik kann man in der Ontologie aus den gegebenen Informationen weitere implizit vorhandene Informationen ableiten; dies ist ein Aspekt bei der Benutzung von Ontologien für das Semantic Web [Ber01]. Außerdem können auch Informationen darüber vorhanden sein, wie verschiedene Ontologien miteinander in Zusammenhang stehen. [Div03]
Eine Ontologie dient in der Informatik dazu eine gemeinsame Begriffsbildung über eine Domäne zu schaffen. Ontologien können zu diesem Zweck von Menschen oder Softwareagenten benutzt werden [Hef03]. Es können Ontologien in ihrer Struktur unterschiedlich stark ausgeprägt sein. Der einfachste Fall ist eine Ontologie, die Begriffe, die durch ihre Beziehungen zu einer Hierarchie angeordnet sind, enthält. Eine komplexere Ontologie enthält zusätzlich noch komplexere Beziehungen, Beziehungen zwischen Konzepten und genauere Angaben über beabsichtigte Interpretationen. [Div03]
Im Rahmen dieser Arbeit ist unter einer Ontologie der zweite Punkt, also die Sicht der Informatik auf eine Ontologie, zu verstehen.
Derzeit sind Ontologien besonders für das sich entwickelnde Semantic Web von Bedeutung. Das Ziel des Semantic Web ist es Informationen im Internet so zu strukturieren, dass Computer bzw. Softwareagenten sie „verstehen" können. Ein Softwareagent, der eine Webseite besucht, soll dann nicht nur erkennen können, dass ein Link vorhanden ist, sondern auch „wis-

sen“, was die semantische Bedeutung hinter diesem Link ist. Diese semantische Struktur soll mit Hilfe von Ontologien hergestellt werden. [Ber01]

3.2 Überblick über Modellierungssprachen

Zunächst möchte ich einen kurzen Überblick über die Modellierungssprachen, mit denen wir uns im Laufe unseres Projektes beschäftigt haben, geben.
In Abbildung 3.1 ist die historische Entwicklung von Technologien für das Semantic Web dargestellt. Im Rahmen dieser Arbeit geht es dabei nur um RDFS (RDF Schema) und RDF (Resource Description Framework), die sich in der Schicht „Data“ in der Abbildung wiederfinden lassen, und um DAML (DARPA Agent Markup Language), OIL (Ontology Inference Layer) und OWL (Web Ontology Language) aus der Schicht „Ontology“. Die Schichten in dieser Abbildung spiegeln zugleich auch ein Schichtenmodell für das Semantic Web wieder [Ber03], das aus Datenmodellen (Data), Strukturdefinitionen (Ontology) und Logik (Rules/Query, Proof) gebildet wird.

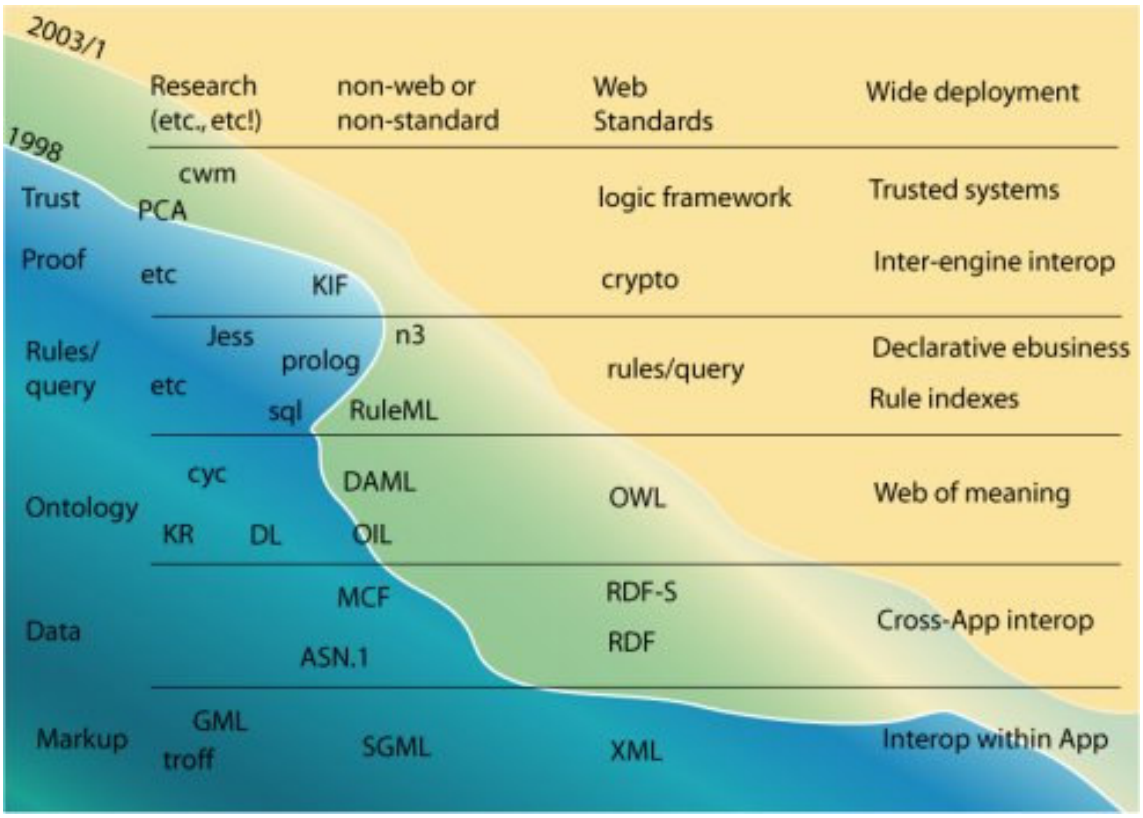


Abbildung 3.1: Historische Entwicklung von Semantic Web Technologien [Ber03]

Der Zusammenhang zwischen einigen der Modellierungssprachen ist in Abbildung 3.2 näher dargestellt. Diese Abbildung gibt einen Überblick über das Schichtenmodell von Sprachen, die zur Modellierung von Informationen im Internet benutzt werden.

Die unterste Schicht bilden HTML und XML. Ursprünglich wurde das World Wide Web nur mit HTML, das einen Standard zur Strukturierung von Dokumenten mit Hilfe von Formatierungsanweisungen, den Tags, die von einem Browser interpretiert werden können, bietet, und der in HTML-Dokumenten enthaltenen Linkstruktur aufgebaut. XML bietet gegenüber HTML eine größere Flexibilität dadurch, dass die Bedeutung der Tags, im Gegensatz zu HTML, nicht vordefiniert ist, sondern selbst definiert werden kann.

Das Resource Description Framework (RDF) ermöglicht es in einem einfachen Datenmodell Beziehungen zu definieren und benutzt zur Repräsentation (u. a.) XML; eine detailliertere Beschreibung folgt unter Abschnitt 3.3.

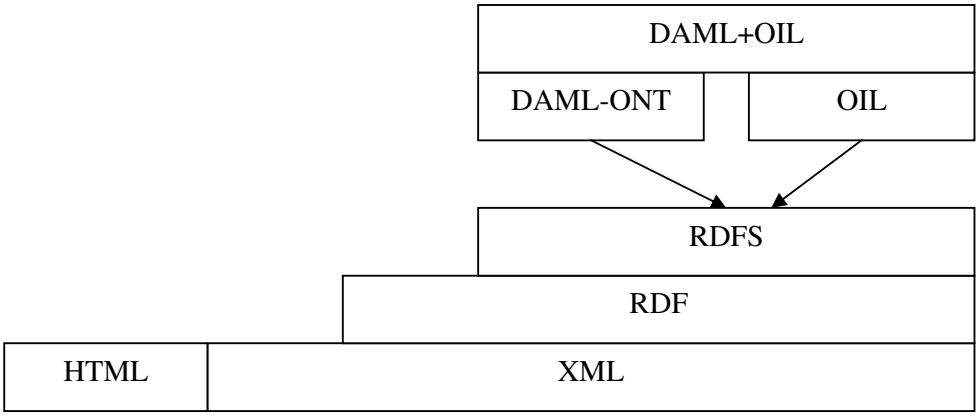


Abbildung 3.2: Schichtenmodell für Sprachen für das Web (in Anlehnung an[Fen00])

RDF Schema (RDFS) baut auf RDF auf und erweitert es um die Möglichkeit Klassen zu definieren und definiert einige Standardklassen; eine detailliertere Beschreibung folgt unter Abschnitt 3.4.
RDFS wiederum war der Ausgangspunkt für die Modellierungssprachen DAML Ontology (DAML-ONT) und Ontology Inference Layer (OIL). Diese beiden Ansätze sind sich relativ ähnlich und wurden zu Darpa Agent Markup Language + Ontology Inference Layer (DAML+OIL) zusammengefasst. Das Ziel dieser Sprachen ist es die grundlegenden Modellierungsmöglichkeiten zum Erstellen einer Ontologie (siehe Abschnitt 3.1) zu liefern. DAML+OIL wird unter Abschnitt 3.5 genauer vorgestellt.

In Abbildung 3.3 ist dargestellt, welche Zusammenhänge zwischen RDF, RDF Schema und weiteren Sprachen, die auf diesen aufsetzen, bestehen:

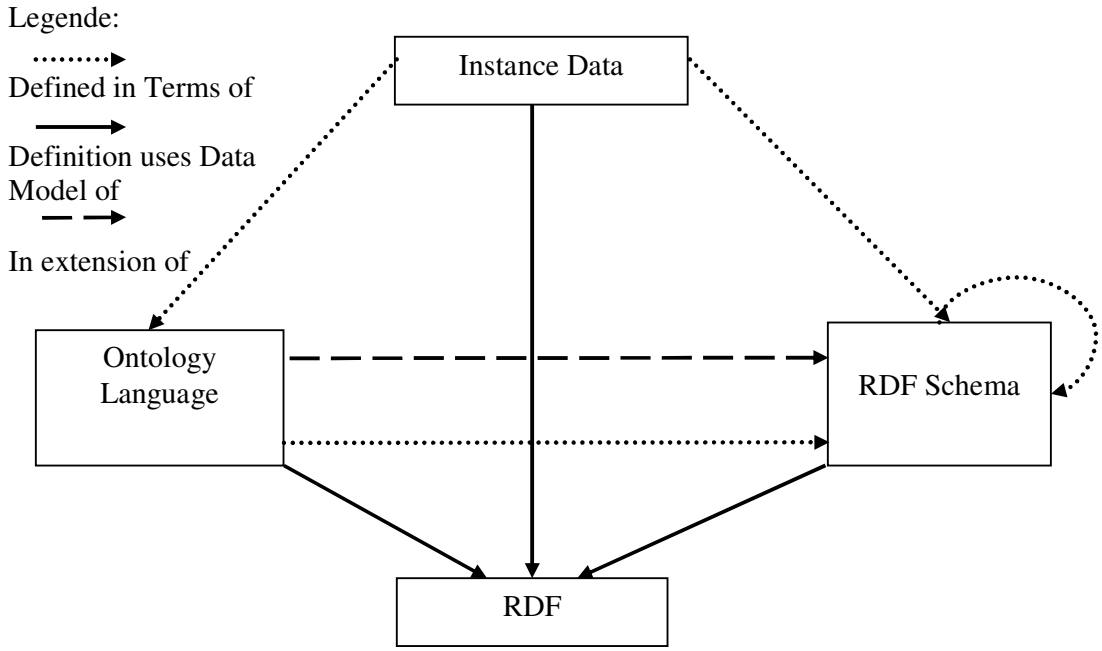


Abbildung 3.3: Prinzipieller Zusammenhang zwischen RDF, RDF Schema und Sprachen, die darauf aufbauen [Bol01]

RDF liefert also das Datenmodell und bildet die Grundlage von Modellierungssprachen für Ontologien (in Abbildung 3.3 als Ontology Language bezeichnet). Ontologiesprachen (Ontology Languages), z. B. DAML+OIL oder OWL, sind also selbst in RDF Schema definiert und erweitern es um weitere Modellierungsmöglichkeiten.

Nun werde ich Modellierungssprachen kurz vorstellen, die man zur Konstruktion von Ontologien benutzen kann bzw. die für das Semantic Web von Bedeutung sind. Beginnen werde ich mit dem *Resource Description Framework* (RDF).

3.3 RDF

RDF ist eine Empfehlung des World Wide Web Consortium (W3C) für die Modellierung von Metadaten zur Beschreibung von Ressourcen im Internet. Beim Design von RDF wollten die Entwickler u. a. Folgendes erreichen [Kly03]:

- Es sollte ein einfaches Datenmodell, das für Anwendungen einfach zu manipulieren und zu verarbeiten ist, geschaffen werden.
- Es gibt eine formale Spezifikation der Semantik (siehe [Hay03]).
- Das Vokabular ist erweiterbar und eindeutig identifizierbar (URIs).
- Die Austauschbarkeit unter verschiedenen Anwendern/Anwendungen ist durch die Möglichkeit RDF in XML zu repräsentieren gegeben.

Das Ziel von RDF ist es automatische Prozesse im Internet, wie z. B. Suche, durch den Aufbau einer Struktur aus Metadaten, zu verbessern und die Zusammenarbeit von Softwareagenten zu unterstützen. Zur Spezifikation des World Wide Web Consortium (W3C) für RDF siehe [Las99].
Das grundlegende Datenmodell von RDF besteht aus den drei Objekttypen Ressource, Property und Statement.
Eine *Ressource* (Resource) kann theoretisch jedes Objekt sein; die einzige Bedingung ist, dass es durch einen *Unified Resource Identifier* (URI) eindeutig identifiziert ist. Da im Konzept der URIs Erweiterbarkeit vorgesehen ist, kann alles eine URI besitzen und somit auch eine RDF Ressource sein.
Eine *Property* ist eine Eigenschaft einer Ressource und wird benutzt, um diese näher zu beschreiben. Eine Property ist selbst auch eine Ressource, woraus folgt, dass sie wiederum durch weitere Properties beschrieben werden kann.
Die Grundstruktur von RDF besteht aus einem Tripel, dessen Bestandteile *Subjekt* (subject), *Prädikat* (predicate) und *Objekt* (object) heißen; dieses Konstrukt wird als *Statement* bezeichnet. Diese Grundstruktur bildet einen gerichteten Graphen, wobei das Subjekt und das Objekt als Knoten fungieren und das Prädikat als Kante mit Richtung vom Subjekt zum Objekt. Das Subjekt eines Statements ist eine Ressource und das Prädikat eine Property. Ein Objekt kann hierbei entweder eine Ressource oder ein *Literal*, also ein Text, oder auch das Subjekt eines anderen Statements sein. Das Objekt eines Statements wird auch als *Wert* (value) der entsprechenden Property bezeichnet. Subjekt, Prädikat und Objekt sind also die Bezeichnungen für die Objekttypen Ressource und Property, die in einem Statement eine bestimmte Rolle einnehmen.
Mehrere Statements können zu einer komplexeren Struktur zusammengefasst werden.
Die Gesamtheit aller angelegten Statements heißt *Modell* (Model). Auch ein einzelnes Statement bildet schon ein RDF Modell.
Durch die Aggregation von Statements können komplexe Strukturen aufgebaut werden, die aber, auf Grund der Einfachheit der Grundstruktur Statement, gut verarbeitet werden können,

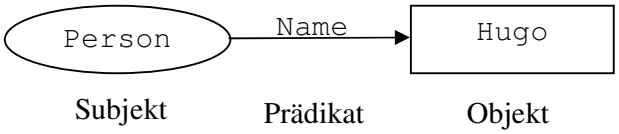
wobei etablierte Techniken aus der Graphentheorie zum Einsatz kommen können. Außerdem wird so eine einfache Erweiterbarkeit eines RDF Modells gewährleistet.
RDF bietet verschiedene syntaktische Möglichkeiten solch ein Statement darzustellen, z. B.: in Tripel-Notation, als Graph oder in XML-Syntax. Die gewählte Darstellungsform hat dabei keinen Einfluss auf die Bedeutung der RDF Statements.

Im Folgenden ein kleines Beispiel zur Verdeutlichung dieser Zusammenhänge:
Das Beispiel ist die Aussage, dass eine Person den Namen Hugo hat. „Person“ ist demnach das Subjekt, „Name“ das Prädikat und „Hugo“ das Objekt. In diesem Fall ist das Objekt ein Literal.

Das Beispiel in *Tripel-Notation* nach dem Muster <Subjekt> HAT <Prädikat> <Objekt>:

`Person` HAT `Name Hugo`

Dasselbe Beispiel als *Graph*:



Und in *XML-Syntax*:

```
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:       xmlns="http://ein.Beispiel.Namespace/">
4:             <rdf:Description rdf:about="Person">
5:             <Name>Hugo</Name>
6:      </rdf:Description>
7: </rdf:RDF>
```

Die XML-Namespaces in den Zeilen zwei und drei sorgen für eine eindeutige Namenszuordnung und bilden zusammen mit den Ressourcen und Properties durch Verknüpfung die URIs. In Zeile vier folgt das Subjekt und in Zeile fünf das Prädikat und das Objekt.
In einer komplexeren Struktur kann ein Subjekt natürlich auch mehrere Prädikate haben und das Objekt kann gleichzeitig Subjekt eines anderen Statements sein.

Als weitere Modellierungsmöglichkeiten bietet RDF die Möglichkeit drei verschiedene Arten von *Containern* anzulegen. Zur Auswahl stehen: eine *Bag*, das ist eine ungeordnete Liste von Ressourcen oder Literalen, eine *Sequence*, das ist eine geordnete Liste von Ressourcen oder Literalen, oder eine *Alternative*, das ist eine Liste von Ressourcen oder Literalen, die Alternativen für ein Objekt darstellen. Diese Container sind in der Kapazität nicht beschränkt und erlauben Duplikate.
Außerdem kann ein RDF Statement selbst wieder ein Objekt eines Statements sein, z. B. für eine Aussage der Art: Person A sagt, dass Person B der Autor von Buch C ist. Diese Modellierungsmöglichkeit heißt *Reification*.

Weitere Modellierungsmöglichkeiten werden RDF durch RDF Schema hinzugefügt, auf das ich im Folgenden eingehen werde.

3.4 RDFS

RDF Schema (RDFS) [Bri03] ist eine Erweiterung von RDF, die dazu dient Vokabulare zu erstellen. Ein *Vokabular* ist eine Sammlung von vorgefertigten Ressourcen und Properties; man kann sich ein Vokabular auch als eine Art Wörterbuch vorstellen, das Ausdrücke, die in RDF Statements benutzt werden können, und deren Bedeutung und Beschränkungen für die Benutzung vorgibt. Ein Beispiel für ein solches Vokabular ist Dublin Core (http://dublincore.org/), eine Spezifikation für Metadaten, wie sie in einer Bibliothek benutzt werden könnten. Zum Zweck der Erstellung solcher Vokabulare bietet RDFS noch weitere, über reines RDF hinausgehende, Modellierungsmöglichkeiten.
RDF Schema bietet die Möglichkeit Klassen zu definieren und Instanzen dieser Klassen anzulegen. Der Zusammenhang zwischen Klassen und Properties in RDF Schema hat eine gewisse Ähnlichkeit zum Typsystem in Objektorientierten Programmiersprachen [Bri03], wie z. B. Java. Der Unterschied ist, dass in einer Objektorientierten Programmiersprache die Klasse im Vordergrund steht und dieser Attribute (Analogon in RDF: Properties) mit Typen zugeordnet werden. Z. B. wird ein Klasse `Buch` definiert, die ein Attribut `Autor` vom Typ `Person` erhält. Im RDF Schema dagegen steht das Attribut bzw. die Property im Vordergrund. Hier wird eine Property genommen und es wird definiert auf welche Klassen sie angewendet werden darf (*domain*, vergleichbar dem mathematischen Definitionsbereich einer Abbildung) und aus welchen Klassen die Werte der Property stammen dürfen (*range*, vergleichbar dem mathematischen Wertebereich). Für das Beispiel von oben würde man also für die Property `Autor` die domain `Buch` und die range `Person` definieren. Zudem kann man eine Klasse als Unterklasse (*subClassOf*) einer anderen Klasse definieren; dies ist eine Property. Eine Unterklasse hat alle Eigenschaften der Oberklasse und zusätzlich eigene Eigenschaften; hierdurch wird Vererbung realisiert. Eine Instanz ist eine konkrete Ausprägung von Klassen oder Properties, also z. B. der Name eines Autors und eines von ihm geschriebenen Buches wären Instanzen der Klassen `Person` bzw. `Buch` und bilden zusammen mit der Property `Autor` eine Instanz dieser.
Des Weiteren bietet RDF Schema eigene Klassen für die grundlegenden RDF Objekttypen, wie Resource, Literal, Class, Statement oder Container, und Properties, wie die schon erwähnten domain, range und subClassOf und einige Properties, die in erster Linie zur Dokumentation oder zum Verknüpfen von verschiedenen Vokabularen gedacht sind.
Durch Benutzen der XML-Namespaces ist es möglich Definitionen aus verschiedenen Vokabularen in einem RDF Modell zu benutzen aber gleichzeitig eindeutig festzulegen, welche Definition benutzt wurde. Dies ist nötig, falls es in verschiedenen Vokabularen Definitionen zu denselben Ressourcen gibt.

Der wesentliche Unterschied zwischen RDF und RDF Schema ist, dass in RDF Modellen die Ressourcen und Properties konkret festgelegt werden. Es gibt jedoch in RDF keine Möglichkeit Klassen oder Properties zu deklarieren, ohne eine konkrete Ausprägung anzulegen. Zu diesem Zweck wird RDF Schema benötigt [Sim02]. Das bedeutet, dass wenn man eine bestimmte Property in RDF modellieren möchte, diese auch in jedem Fall im RDF Modell existiert. In einem RDF Modell, das RDF Schema benutzt, ist es jedoch möglich, diese Property zu modellieren, ohne, dass sie sofort auch im Modell existiert. Dadurch ist aber die Möglichkeit gegeben später eine Ausprägung dieser Property im Modell zu erschaffen.

3.5 DAML+OIL

Nun komme ich zu *DARPA Agent Markup Language + Ontology Inference Layer* [Con01], kurz DAML+OIL. DAML+OIL ist eine Erweiterung von RDFS, die aus der Vereinigung zweier früherer Ansätze RDFS um komplexere Modellierungsmöglichkeiten zu erweitern hervorging, nämlich DAML-ONT und OIL (siehe Abbildung 3.2). DAML-ONT ist aus einem von der US-Regierung unterstützten Projekt zur Entwicklung einer Sprache zur Nutzung von Ontologien für das Semantic Web hervorgegangen. Etwa zeitgleich entstand OIL, eine Sprache, die große Ähnlichkeit zu DAML-ONT hat. Ein wichtiger Unterschied ist, dass bei OIL größerer Wert auf eine formale Beschreibung der Semantik von Ausdrücken (beeinflusst aus dem Gebiet der Logik) gelegt wurde und effiziente Schlussverfahren vorhanden sind.
DAML+OIL ist ein Vokabular (siehe Abschnitt 3.4) aus Properties und Klassen, das dem RDF Schema folgende Möglichkeiten hinzufügt:

- Properties mit Kardinalitäten zu versehen
- XML Schema Datentypen zu benutzen
- Beschränkte Listen anzulegen
- Mengenoperationen (wie z. B. Vereinigung, Schnitt) zu benutzen

Eine Klasse in DAML+OIL ist eine Subklasse der Klasse aus dem RDF Schema. Im Unterschied zu RDFS bietet DAML+OIL zur Definition von Klassen aber auch die Möglichkeit dieses durch direktes Aufzählen der Mitglieder oder durch Angeben von Vereinigungen oder Schnitten oder Verknüpfungen von diesen zu tun. In einer DAML+OIL Klassendefinition kann man außerdem angeben, ob die Klasse zu einer anderen Klasse disjunkt oder äquivalent sein soll. Jedes dieser Elemente kann mehrfach in einer Klassendefinition vorkommen.
Die Properties werden in DAML+OIL nach *ObjectProperty* und *DatatypeProperty* unterschieden. DatatypeProperties setzen Klassen mit einfachen Datentypen in Beziehung. DAML+OIL benutzt für diese einfachen Datentypen Definitionen aus XML Schema. ObjectProperties setzen Klassen miteinander in Beziehung.
Neben den schon aus RDFS bekannten Angaben für domain und range einer Property, bietet DAML+OIL die Möglichkeit Properties mit Kardinalitätsbeschränkungen zu versehen. Zu diesem Zweck wird eine anonyme Klasse, eine *Restriction*, definiert, in der angegeben wird auf welche Property sich die Beschränkung bezieht und wie genau die Beschränkung aussieht, also eine untere oder eine obere Beschränkung oder der genaue Wert der Anzahl der Werte, die eine Property erhalten darf, und ob die Werte der Property einer bestimmten Klasse angehören müssen. Die Klasse, die in der domain der zu Beschränkenden Property vorkommt und für die diese Beschränkungen gelten sollen (also das Subjekt eines Statements), wird als Unterklasse der Restriction definiert. Wenn z. B. einer `Person` eine eindeutige `Identifikationsnummer` mit dem Typ `ID-Nummer` zugeordnet werden soll, würde man eine Restriction erzeugen, die sich auf die Property `Identifikationsnummer` bezieht, eine Kardinalitätsbeschränkung auf eins enthält und angibt, dass die Werte dieser Property aus der Klasse `ID-Nummer` stammen müssen. Von dieser Restriction würde man die Klasse `Person` mittels subClassOf ableiten. Eine auf diese Weise erzeugte Beschränkung ist eine lokale Beschränkung und gilt nur für die Klasse, die von ihr abgeleitet ist.
Eine andere Möglichkeit Properties zu beschränken ist, die Einschränkungen direkt in der Definition der Property anzugeben. Hier ist es möglich anzugeben, ob die Property von einer anderen abgeleitet wurde (*subPropertyOf*), welche domain bzw. range gilt (wie bei RDFS; siehe oben), ob die Property einzigartig (*UniqueProperty*), unzweideutig (*UnambiguousProperty*) oder transitiv (*TransitiveProperty*) ist. Bei einer einzigartigen Property bestimmt das Subjekt eindeutig das Objekt, d. h., dass es zu jedem Subjekt höchstens ein Objekt gibt; dies

entspricht einer oben genannten Kardinalitätsbeschränkung auf einen Wert von eins. Ein Beispiel hierfür ist, dass zu einem Kind (Subjekt) eindeutig eine Mutter (Objekt) gehört (Subjet: Kind; Prädikat: hatMutter; Objekt: Mutter). Bei einer unzweideutigen Property bestimmt das Objekt eindeutig das Subjekt, d. h., dass es zu jedem Objekt höchstens ein Subjekt gibt; Unzweideutigkeit ist das Inverse zur Einzigartigkeit (siehe Abbildung 3.4).

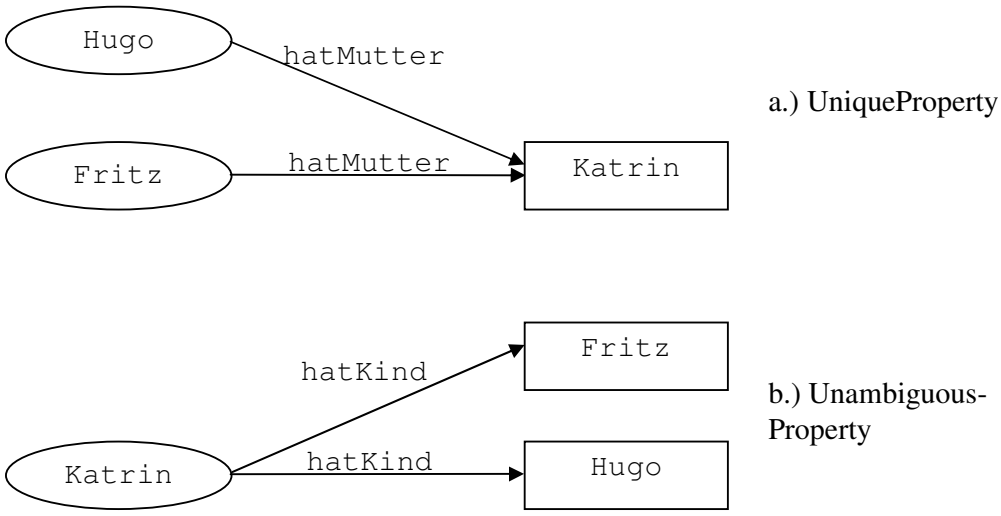


Abbildung 3.4: UniqueProperty und UnambiguousProperty

Für das Beispiel bedeutet dies, dass eine Mutter (Subjekt) mehrer Kinder (Objekte) haben kann, aber es zu jedem Kind (Objekt) nur eine Mutter (Subjekt) gibt (Subjekt: Mutter; Prädikat: hatKind; Objekt: Kind). Diese Art von Beschränkungen ist global und bewirkt, dass jede Klasse, auf die die Property angewendet werden soll, die Beschränkungen erfüllen muss. Für das Beispiel, das ich für die Restriction angeführt habe, hieße dies, dass man die Eindeutigkeit auch dadurch erreichen könnte, dass man die Property `Identifikationsnummer` als unzweideutig definierte (UnambiguousProperty). Dies hätte zur Folge, dass jede Klasse, auf die die Property `Identifikationsnummer` angewendet wird, eine eindeutige `ID-Nummer` haben muss, wohingegen sich dies in der ersten Möglichkeit nur auf die Klasse `Person` bezöge und es bei anderen Klassen noch möglich wäre verschiedenen Subjekten dieselbe `ID-Nummer` zuzuordnen (auch wenn dies zugegebenermaßen nicht besonders sinnvoll wäre).
Dass eine Property transitiv ist bedeutet, dass, wenn die Property A mit B und B mit C verbindet, dann verbindet sie auch A mit C, z. B. wenn Hugo der Vorfahre von Karl ist und Karl der Vorfahre von Wilhelm, dann ist Hugo auch Vorfahre von Wilhelm.
Anders als in RDF bzw. RDFS ist es in DAML+OIL möglich eine beschränkte Liste anzulegen; diese ist vom Typ *Collection*. Die Struktur einer solchen Liste lässt sich rekursiv aus den Elementen *first*, *rest* und *nil* zusammensetzen, wobei nil das Ende der Liste kennzeichnet (siehe Abbildung 3.5). Dadurch ist gewährleistet, dass die Liste beschränkt ist und keine Elemente hinzugefügt werden können, ohne die Struktur der Liste zu verändern bzw. eine vorhandenes Element zu überschreiben. Diese Liste wird bei der Definition einer Klasse durch Aufzählen der Instanzen (*Enumeration*) benutzt.

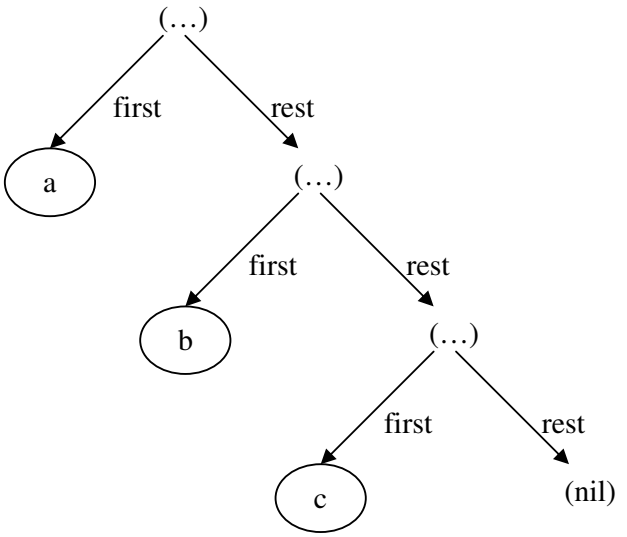


Abbildung 3.5: Struktur einer beschränkten Liste in DAML+OIL [Oue03]

3.6 Überblick über RDFS und DAML+OIL

Die folgende Tabelle gibt einen Überblick über einige Konzepte von RDFS und DAML+OIL und stellt diese einander gegenüber (basierend auf [Gil00]). Daraus wird deutlich, welche Konzepte DAML+OIL bietet, die im RDF Schema noch nicht vorhanden waren. Außerdem wird deutlich, dass DAML+OIL teilweise auf die RDF Schema Syntax zurückgreift, wodurch es möglich wird, dass Agenten, die nur mit RDF Schema umgehen können, Teile von in DAML+OIL geschriebenen Spezifikationen verstehen können.
Das Präfix „rdfs“ bedeutet, dass diese Konzepte durch das RDF Schema definiert werden, diejenigen mit dem Präfix „daml“ dagegen durch die DAML+OIL Spezifikation.

	RDFS	**DAML+OIL**
Klassen	rdfs:Class	daml:Class ist eine Unterklasse von rdfs:Class
Properties	rdfs:Property	daml:ObjectProperty daml:DatatypeProperty sind Unterklassen von rdfs:Property
Vererbung	rdfs:subClassOf rdfs:subPropertyOf	benutzt RDFS
Range	Global: rdfs:range	Global: rdfs:range Lokal: daml:Restriciton
Domain	Global: rdfs:domain	Global: rdfs:domain Lokal: indirekt; über das Ableiten einer Klasse von einer Restriction wird diese Klasse zur einzigen domain für diese Beschränkung

Kardinalitätsbeschränkungen	-	Global: daml:UniqueProperty ist eine Unterklasse von rdfs:Property Lokal: daml:Restriction
Einfache Datentypen	-	benutzt XML Schema Datentypen
Beschränkte Listen	-	daml:collection
Mengenoperationen (werden benutzt, um Klassen über schon vorhandene Klassen zu definieren)	-	daml:disjointWith daml:disjointUnionOf daml:intersectionOf daml:unionOf daml:complementOf

3.7 Topic Maps

Topic Maps dienen dem Navigieren in großen Datenmengen und sind etwa vergleichbar mit einem Index in Büchern; Topic Maps werden auch als das GPS (Global Positioning System) des Web bezeichnet [Pep00]. Topic Maps sind durch den ISO-Standard 13250 [ISO99] standardisiert. Grundsätzlich besteht eine Topic Map aus *Themen* (Topics), *Assoziationen* (Associations) und *Vorkommensangaben* (Occurrences). Als Thema kann praktisch alles dienen, z. B. Gegenstände, Konzepte etc. Die Themen werden durch die Assoziationen untereinander verknüpft und eventuell durch Vorkommensangaben ergänzt. Das entstehende Beziehungsgeflecht bildet die Topic Map. Im Gegensatz zum Datenmodell von RDF sind die Beziehungen in Topic Maps ungerichtet. Jedes Thema übernimmt daher in einer Assoziation eine bestimmte Rolle. In einer Assoziation „Hugo – benutzen – Informationsservice“ hat z. B. Hugo die Rolle „Benutzer“ und der Informationsservice die Rolle „Dienst“.
In einer Topic Map ist es auch möglich Klassen und Instanzen dieser Klassen anzulegen. Dazu werden zwei Themen über die Assoziation instanceOf in Beziehung gesetzt, wobei ein Thema die Rolle der Klasse und das andere die Rolle der Instanz übernimmt (siehe Abbildung 3.6).

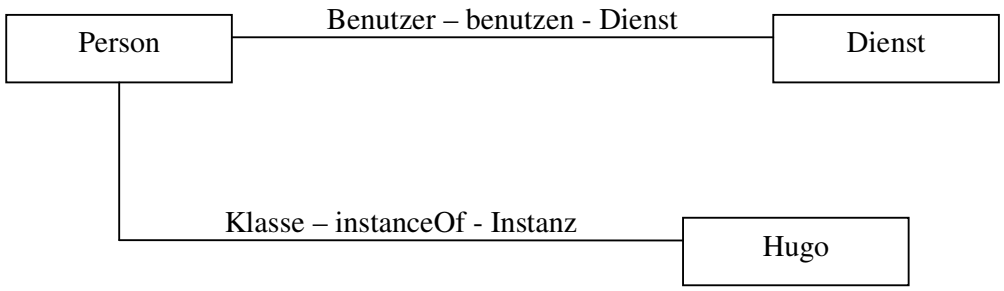


Abbildung 3.6: Ausschnitt aus einer Topic Map

4. Das Jena-Framework

In diesem Kapitel möchte ich die Grundzüge des Jena-Frameworks darstellen. Das Jena-Framework implementiert die Modellierungssprachen RDF, RDFS und DAML+OIL, die ich im vorhergehenden Kapitel vorgestellt habe.

4.1 Überblick über das Jena-Framework

Das Jena-Framework ist eine von Hewlett Packard entwickelte Schnittstelle in der Programmiersprache Java um das RDF Datenmodell (siehe Abschnitt 3.3) in einer Anwendung verwenden zu können.

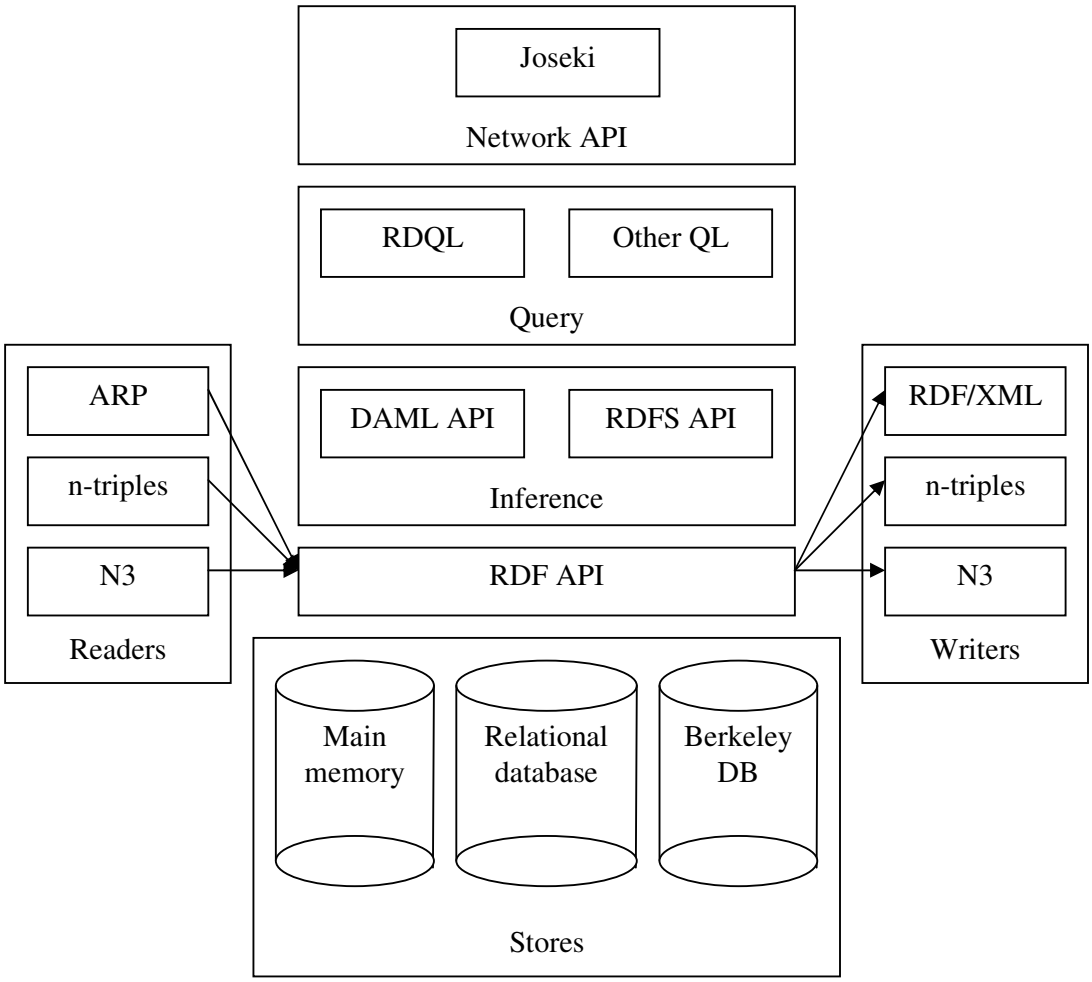


Abbildung 4.1: Die Jena Architektur [McB02A]

Das Kernstück ist die RDF API (Application Programming Interface), die die Möglichkeit bietet RDF Graphen bzw. Modelle zu erstellen und zu manipulieren. Jena bietet eine Anzahl von Java Interfaces und entsprechenden Implementationen mit deren Hilfe RDF Ressourcen, Literale, Properties, Statements und Container repräsentiert werden können (siehe Abbildung 4.2); das RDF Datenmodell wird hierbei mit Javaklassen nachgebildet.

Außerdem enthält das Jena-Framework verschiedene Mechanismen zum Einlesen und Ausgeben von RDF Datenstrukturen (siehe Abbildung 4.1: Readers, Writers). ARP ist ein in Jena integrierter Parser, der RDF Modelle in XML Repräsentation einlesen kann. N-triples und N3 sind alternative Repräsentationsformen, mit denen das Jena-Framework ebenfalls umgehen kann.
Das Jena-Framework bietet verschiedene Möglichkeiten die RDF Modelle zu speichern. Eine Möglichkeit ist die Modelle im Hauptspeicher abzulegen; dies ist die während der Verarbeitung von RDF Modellen genutzte Methode. Zur dauerhaften Speicherung gibt es die Möglichkeiten der Speicherung in verschiedenen Datenbanken (siehe Abbildung 4.1, Stores).
RDF Schema (Abbildung 4.1, Inference) ist in Jena als Vokabular, also als eine Sammlung vordefinierter Ressourcen und Properties. Im Jena-Framework sind in dem Vokabular nur die Namen der Ressourcen und Properties vordefiniert. Für DAML+OIL (Abbildung 4.1, Inference) gibt es eigene Interfaces, die von den Interfaces für RDF abgeleitet sind; auf die DAML+OIL Implementation gehe ich später genauer ein.
Des Weiteren bietet Jena eine Abfragesprache für RDF (Abbildung 4.1, Query), die sich an SQL orientiert, und Unterstützung für den Einsatz im Netz (Abbildung 4.1, Network API).

4.2 RDF im Jena-Framework

Das Jena-Framework bietet Interfaces für die Bestandteile des RDF Datenmodells an. Zu diesen Interfaces gibt es auch vorgefertigte Klassen, die diese implementieren. Die Beziehungen zwischen diesen Interfaces sind im folgenden Klassendiagramm dargestellt.

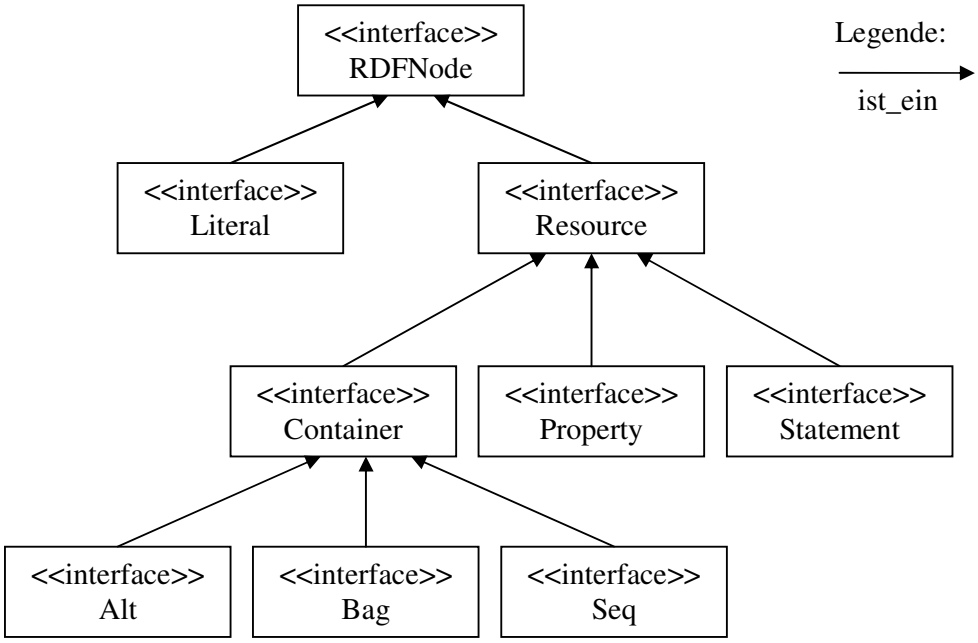


Abbildung 4.2: Klassendiagramm der Klassen für RDF des Jena-Toolkits [Ver03]

Die oberste Position in der Hierarchie nimmt das Interface *RDFNode* ein. Diese Klasse fasst Literale und Ressourcen, die beide als Objekt in einem Statement dienen können, zusammen.
Das Interface *Literal* bzw. dessen Implementation dient der Repräsentation von Literalen im RDF Modell zur Laufzeit. Das Jena-Framework bietet die Möglichkeit, dass ein Literal nicht nur ein String sein kann, sondern auch als Boolean, Byte, Character, Integer, Double, Long, Float oder Objekt, das Oberklasse einer beliebigen Klasse ist, interpretiert werden kann. Klas-

sen, die dieses Interface implementieren können im RDF Datenmodell die Stellung eines Objekts einnehmen.
Das Interface *Resource* bzw. dessen Implementation dient der Repräsentation von Ressourcen des RDF Datenmodells zur Laufzeit. Es werden Methoden zur Abfrage und Manipulation der Properties und zum Abfragen des Namens bzw. der URI angeboten. Klassen, die dieses Interface implementieren können im RDF Datenmodell die Stellung eines Subjekts oder Objekts einnehmen.
Konform zum RDF Datenmodell sind die Interfaces *Container*, *Property* bzw. *Statement* vom Interface Resource abgeleitet, da diese auch im RDF Datenmodell spezielle Ressourcen sind.
Für Statements bestehen die Möglichkeiten Subjekt, Objekt oder Prädikat aus dem Statement zurückgeben zu lassen, das Objekt oder die Property zu bearbeiten, zwei Statements auf Gleichheit zu Prüfen oder das Statement aus dem Modell zu löschen. Klassen, die das Interface Statement implementieren können im RDF Datenmodell die Stellung eines Objekts einnehmen.
Bei Containern hat man die Möglichkeit etwas einzufügen oder auf Vorhandensein zu prüfen. Klassen, die das Interface Container implementieren können im RDF Datenmodell die Stellung eines Objekts einnehmen.
Klassen, die das Interface Property implementieren nehmen im RDF Datenmodell die Position des Prädikats ein.
Mit Hilfe dieser Mittel ist man in der Lage das RDF Datenmodell nachzubilden und RDF Modelle zur Laufzeit anzulegen, zu verändern und mit der RDF Query Language (RDQL) zu befragen.

4.3 DAML+OIL im Jena-Framework

Eine weitere Sammlung von Interfaces fügt dem Jena-Framework die Modellierungsmöglichkeiten von DAML+OIL hinzu.
Jena bietet auch für die Modellierungsmöglichkeiten von DAML+OIL jeweils eigene Interfaces und entsprechende Implementierungen an. DAMLCommon dient hierbei als Oberklasse für alle DAML+OIL Ressourcen und bietet Methoden, die für diese gemeinsam benutzt werden können, wie das Bearbeiten von Properties oder Zugriff auf allgemeine Properties wie den Bezeichner, den Kommentar oder der Verweis auf äquivalente Ressourcen. Dieses Interface selbst ist wiederum eine Unterklasse des Interfaces Resource des RDF Datenmodells. Daher können alle DAML+OIL Ressourcen auch als RDF Ressourcen verwendet werden, verlieren dabei allerdings die Möglichkeit des Zugriffs auf die durch DAML+OIL hinzugefügten Properties. Entsprechendes gilt für DAMLProperty, die Unterklasse von RDF Property ist. DAMLProperty bietet Zugriff auf die in der DAML+OIL Spezifikation vorgesehenen Eigenschaften einer Property einzigartig zu sein oder einer Property, von der diese abgeleitet wurde, anzugeben (siehe Abschnitt 3.5). Auch für die in DAML+OIL vorgesehene Unterscheidung in ObjectProperty und DatatypeProperty bietet Jena jeweils ein eigenes Interface. DatatypeProperties können selbst kein Subjekt in einem weiteren Statement und daher auch nicht transitiv sein. ObjectProperties können die Eigenschaften unzweideutig und transitiv besitzen und selbst wieder Subjekt eines Statements sein.
DAMLClassExpression dient dazu Klassen als Aufzählungen, Schnitte oder Vereinigungen zu definieren, wie es die DAML+OIL Spezifikation vorsieht.
DAMLRestriction ist die Java Entsprechung zur Restriction der DAML+OIL Spezifikation und bietet die Möglichkeit Kardinalitätsbeschränkungen und Beschränkungen auf bestimmte Klassen zu definieren.

DAMLInstance und DAMLDataInstance sind die Java Repräsentationen einer Instanz einer Klasse bzw. einer Instanz eines DAMLDatatype. DAMLDatatype ist die Java Repräsentation der einfachen Datentypen, die in DAML+OIL durch XML Schema definiert werden.
DAMLList ist die Java Repräsentation der DAML+OIL Collection. Es werden Methoden angeboten, um Elemente in die Liste einzufügen oder zu entfernen und um das erste Element oder die Restliste zurückzugeben. Außerdem ist es möglich ein Element an einer bestimmten Stelle zurückgeben zu lassen oder die Länge der Liste abzufragen.
Das Element DAMLOntology enthält Informationen, wie z. B. Version oder Kommentar, zur aktuellen Ontologie.

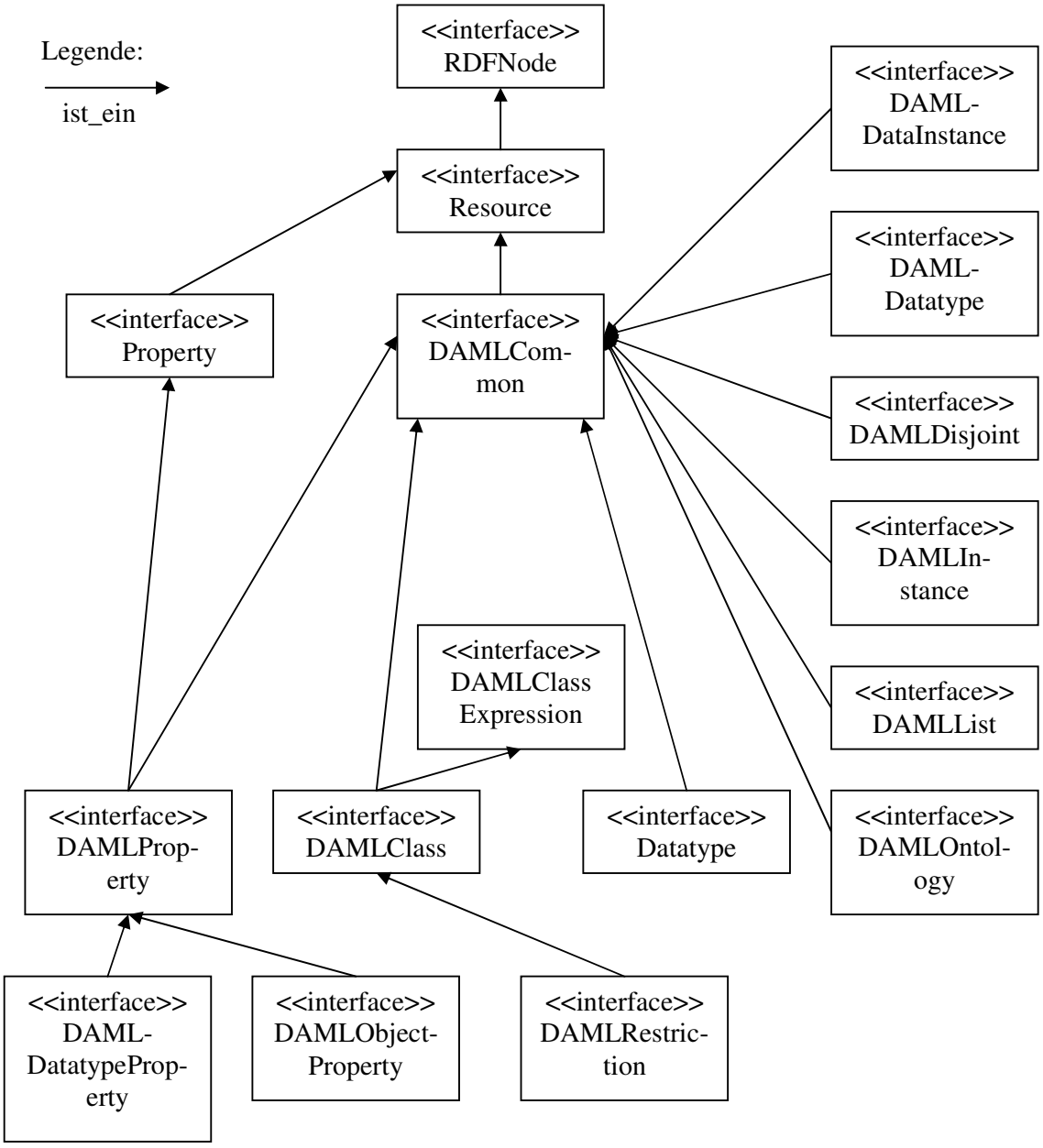


Abbildung 4.3: Klassendiagramm der Klassen für DAML+OIL des Jena-Toolkits

5. Das Jena-Framework im Einsatz

In diesem Kapitel möchte ich darstellen, welche Probleme im Umgang mit dem Jena-Framework auftraten und worauf diese zurückzuführen waren. Außerdem möchte ich herausarbeiten, welche Verbesserungsmöglichkeiten an unserer Ontologie es speziell im Hinblick auf DAML+OIL noch gibt und ob diese mit dem Jena-Framework realisierbar wären.

Nachdem die Entscheidung für das Jena-Framework im Projekt gefallen war, begannen wir damit uns in dieses einzuarbeiten. Dies geschah mit Hilfe der Java-Dokumentation und eines Tutorials [McB02B], das eine kurze Einführung in die Grundlegenden Funktionen des Jena-Frameworks für die RDF Implementation bietet. Da das RDF Datenmodell sehr einfach ist und wir auch nur einen ersten Einblick in das Jena-Framework erhalten und dann mit DAML+OIL weitermachen wollten, traten hier noch keine Probleme auf. Das Anlegen, Ausgeben und Einlesen von RDF Modellen ist in dem Tutorial gut nachvollziehbar beschrieben und lieferte auch die gewünschten Ergebnisse.
Da unsere Ontologie jedoch in DAML+OIL realisiert werden sollte, arbeiteten wir uns als nächstes in dessen Implementation im Jena-Framework ein. Auch dies geschah an Hand eines Tutorials. Der Vorteil von DAML+OIL gegenüber RDF ist, dass man in DAML+OIL Properties oder Ressourcen nicht sofort konkret anlegen muss, sondern diese erst in einem Schema deklariert werden. RDF dagegen arbeitet nur mit Instanzen, d. h., dass jede Property, die man benutzen möchte in einer konkreten Ausprägung vorliegen muss. Es müssen also auch Elemente, die erst während einer Sitzung einen Wert bekommen sollen, schon vorher mit einem Standardwert angelegt werden. DAML+OIL dagegen ermöglicht hier das deklarieren von Klassen, von denen erst später eine Instanz angelegt wird.

5.1 Probleme im Zusammenhang mit der Programmiersprache

Beim Erstellen von Ontologien mit dem Jena-Framework muss man beachten, dass die Reihenfolge, in der die Elemente der Ontologie erstellt werden, eine Rolle spielt. In einer in XML Syntax geschriebenen Ontologie spielt die Reihenfolge der Elemente keine Rolle und auch das Einlesen einer solchen Ontologie mit dem Jena-Framework bereitet keine Probleme. Wenn man allerdings mit den Methoden des Jena-Frameworks Elemente zu der Ontologie hinzufügen möchte, die untereinander in Beziehungen stehen, muss man darauf achten, dass die Elemente, zu dem Zeitpunkt, an dem sie benutzt werden, auch schon existieren. Ein Beispiel hierfür ist das Anlegen einer Klassenhierarchie. In einer in XML geschriebenen Ontologie ist es unerheblich, in welcher Reihenfolge die Klassen deklariert werden. Mit dem Jena-Framework muss man die Oberklasse anlegen, bevor man die Unterklassen deklarieren kann. Dies hängt natürlich mit der sequentiellen Ausführung der Programmiersprache zusammen und dieser Fehler kann leicht vermieden werden, wenn man dies bedenkt. Dieser Fehler kann jedoch schwierig zu entdecken sein, vor allem, wenn man einen Teil der Ontologie aus einer Datei einliest und einen Teil mit den Jena-Framework Methoden erzeugt, da man als Fehlermeldung die Java Nullpointer Exception erhält, die noch viele weitere Ursachen haben kann, weshalb ich ihn hier aufführe.
Ein weiteres Problem, das im Zusammenhang mit der Programmiersprache auftreten kann, ist, dass sich eine Methode in der Programmiersprache korrekt anwenden lässt, aber die DAML+OIL Syntax nicht beachtet. Dies hängt damit zusammen, dass einige Methoden im Jena-Framework für verschiedene Aufgaben wieder verwendet werden. Deshalb erlauben sie als Parameter eine Klasse, die in der Klassenhierarchie besonders weit oben eingeordnet ist, wie z. B. DAMLCommon oder Resource (vergleiche Abbildung 4.3). Dadurch ist zwar eine

hohe Wiederverwendbarkeit der Klasse gegeben, allerdings ist es dadurch auch möglich Properties dort einzusetzen, wo nach der DAML+OIL Spezifikation nur Klassen erlaubt sind, oder umgekehrt Klassen, wo nur Properties erlaubt sind. Dieses Problem ließe sich dadurch beheben, dem Jena-Framework selbst speziellere Funktionen hinzuzufügen, oder eine Logik zu programmieren, die überprüft, in welchem Fall welcher Parameter erlaubt ist.

5.2 Probleme durch die Klassenhierarchien im Jena-Framework

Ein weiteres Problem, das beim Umgang mit der DAML+OIL Implementation des Jena-Frameworks auftrat, ist, dass hier davon ausgegangen wird, dass die Ontologie schon fertig vorliegt, z. B. aus einem Verzeichnis für Ontologien im Internet (http://www.daml.org/ontologies/). Wir wollten unsere Ontologie jedoch selbst erstellen und dafür das Jena-Framework benutzen. Durch die Annahme, dass die Ontologie schon fertig vorliegen solle, wurde in dem Tutorial weniger darauf eingegangen, wie Ontologien in DAML+OIL im Jena-Framework angelegt werden. Probleme bereiteten uns dabei anfangs das Anlegen von Instanzen zu gegebenen Klassen und der Zugriff auf schon vorhandene Elemente der Ontologie. Beim Zugriff auf Elemente in der Ontologie muss man beachten, dass alle DAML+OIL Elemente über die Methode getDAMLValue() zugreifbar sind. Zurück erhält man ein DAMLCommen Objekt, das zur weiteren Bearbeitung noch in die richtige Klasse umgewandelt werden muss. Außerdem muss man beim Zugriff auf die Elemente beachten, dass man dazu nicht die RDF Methoden benutzt. Diese liefern zwar auch das gewünschte Element zurück, da alle DAML+OIL Elemente Unterklassen von RDFNode bzw. Resource (siehe Abbildung 4.3) sind und ein DAMLModel Unterklasse der Klasse Model für RDF ist, allerdings werden auf diese Weise (mit der Methode getProperty()) zugegriffene DAML+OIL Properties dann zu RDF Properties. Diese lassen sich auch nicht wieder in DAML+OIL Properties umwandeln, da die Vererbungsstruktur auf der Ebene der Implementationen eine andere ist als auf Ebene der Interfaces (siehe Abbildung 5.1). Andererseits lassen sich jedoch als DAML+OIL Properties zugegriffene Properties auf Ebene der Interfaces in RDF Properties umwandeln. Durch diese zwei Ebenen der Klassenhierarchien (Interfaces und Implementationen) wird die Mehrfachvererbung erreicht, so erbt z. B. die DAMLProperty auf Ebene der Interfaces sowohl von der RDF Property, als auch von dem Wurzelinterface der DAML+OIL Klassenhierarchie, dem Interface DAMLCommon (vergleiche Abbildung 4.3). Allerdings verursachte diese zweischichtige Modellierung auch das ein oder andere Mal Verwirrung bei uns, da manchmal nicht sofort klar war, ob nun eine Implementation oder ein Interface benutzt werden musste. Bei genauerer Betrachtung zeigte sich, dass die Klassen für die Implementationen nie direkt benutzt werden. Konstrukte der DAML+OIL Spezifikation, wie Klassen oder Properties, werden im Jena-Framework nie direkt mittels Konstruktor erzeugt, sonder nur über die Methoden der Klasse DAMLModel in diesem Model erzeugt. Die einzige Ausnahme hiervon bildet das Model selbst, das explizit durch `DAMLModel model = new DAMLModelImpl();` erzeugt werden muss.

Auch zwischen der Hierarchie, die in der DAML+OIL Spezifikation definiert ist, und der Implementation im Jena-Framework gibt es Unterschiede. Zum Beispiel ist in der DAML+OIL Spezifikation ein Datatype auch eine Klasse, im Jena-Framework ist dies jedoch nicht so; dort ist DAMLDatatype eine Unterklasse von DAMLCommon. Die Klasse DAML-Datatype bzw. Datatype in der DAML+OIL Spezifikation ist die Klasse, die einfache Datentypen, die DAML+OIL aus XML Schema benutzt, aufnimmt. Bei uns sollte sie den Datentyp String aufnehmen.

Ein daraus resultierendes Problem entstand uns bei der Suche nach allen Klassen im Wertebereich einer Property. Dazu benutzten wir einen Iterator, der zu einer gegebenen Property alle Klassen deren Instanzen im Wertebereich dieser Property liegen, enthält. Wir versuchten dann

die gefundenen Klassen als DAMLClass weiter zu verarbeiten, was nach der DAML+OIL Spezifikation möglich sein müsste, durch das Jena-Framework jedoch nicht zugelassen wird. Der Irrtum, dem wir hier aufgesessen waren, war, dass wir davon ausgingen, dass die Methode getRangeClasses() auch tatsächlich nur Objekte vom Typ DAMLClass zurückliefert, wie es der Name vermuten lässt. Dies ist jedoch nicht so; stattdessen liefert sie einen Iterator mit Objekten vom Typ Object zurück. Darunter fällt natürlich auch die Klasse DAMLDatatype. Aufgrund der oben gezeigten Hierarchie (Abbildung 4.3) lässt sich diese nicht als DAMLClass behandeln, so dass dieser Versuch unsererseits scheitern musste. Eine einfache Lösung für dieses Problem wäre gewesen, den Inhalt des Iterators als DAMLCommon weiterzuverarbeiten. Diesen Fehler zu vermeiden hätte uns mehr Zeit für die Einarbeitung in das Jena-Framework und eine ausführlichere Dokumentation geholfen.

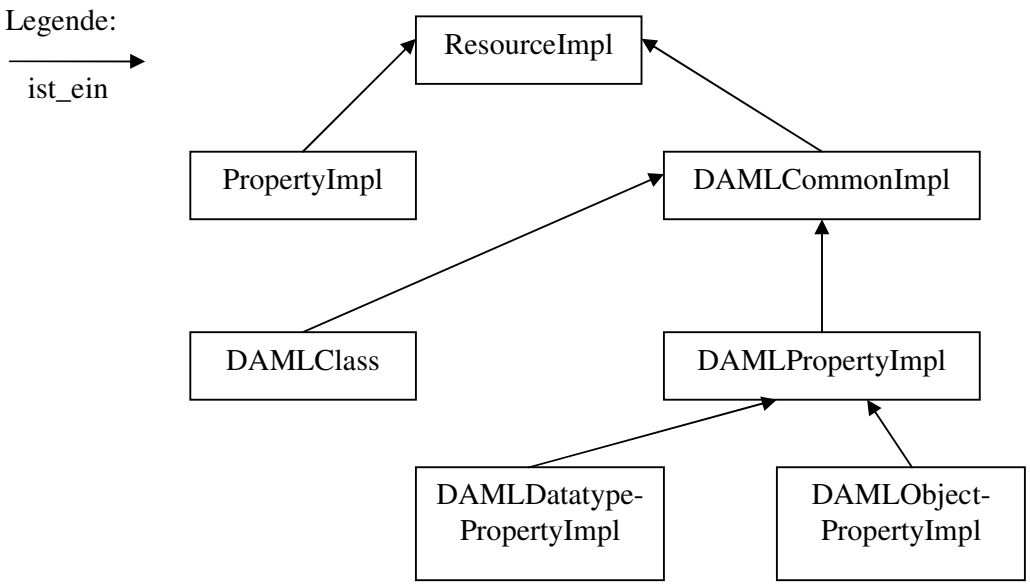


Abbildung 5.1: Ausschnitt aus der Klassenhierarchie der RDF und DAML+OIL Klassen des Jena-Toolkits (vergleiche Interfaces in Abbildung 4.2 und 4.3)

5.3 Probleme durch fehlende Logik bzw. fehlende Unterstützung für automatisches Schließen

Ein wichtiger Punkt beim Entwurf von DAML+OIL war zu erreichen, dass Softwareagenten das Ableiten von Informationen aus den gegebenen Daten ermöglicht wird. Dieser Ableitungsmechanismus ist im Jena-Framework nur ansatzweise implementiert. Das Jena-Framework beherrscht das Ableiten der Unterklassen zu einer gegebenen Klasse, d. h. die Methode getInstances(), zum Auflisten aller Instanzen einer Klasse, findet auch die Instanzen der Unterklassen. Selbiges gilt für die Properties und deren übergeordnete Properties, die auf eine gegebene Klasse anwendbar sind. In unserem Beispiel könnte man dies z. B. benutzen, wenn man einmal die Klasse der Männer und die der Frauen einzeln ansprechen möchte und ein anderes Mal als Klasse der Personen im Ganzen.
Was das Jena-Framework nicht bietet ist eine Überprüfung der Veränderungen an der Ontologie zur Laufzeit. Es ist problemlos möglich einer Instanz eine Property, die eigentlich einzigartig (siehe Abschnitt 3.5) sein sollte, mehrfach zuzuordnen. Zudem ist es möglich in Instanzen von Properties Instanzen von Klassen zu benutzen, die nicht zum Werte- bzw. Definitionsbereich der Property gehören. Dies gilt sowohl für das Erzeugen bzw. Verändern von Ontologien mit den Methoden des Jena-Frameworks als auch das Einlesen einer bestehenden

Ontologie. Das Problem der mehrfach erzeugbaren einzigartigen Property ließe sich sehr einfach dadurch lösen, dass man eine Methode schreibt, die überprüft, ob eine Property einzigartig ist, und gegebenenfalls statt der Methode addProperty() die Methode setPropertyValue() benutzt. Die Methode setPropertyValue() überschreibt eine schon vorhandene Property der bearbeiteten Property, so dass sie immer nur einmal existiert. Für die Typprüfung ist es nötig bei jedem Anlegen einer Property den Definitions- und den Wertebereich zu überprüfen und die Änderung gegebenenfalls abzuweisen. Da es in keiner Ontologie zu derartigen Inkonsistenzen kommen sollte, wäre es vorteilhaft diese Konsistenzprüfungen direkt dem Jena-Framework hinzuzufügen und nicht erst in der Anwendungslogik zu realisieren.
Da im Jena-Framework bisher kaum Unterstützung für das automatische Schließen vorhanden ist, beschränkt sich der Vorteil von DAML+OIL gegenüber RDF auf die Möglichkeit Klassen anzulegen und Properties mit Definitions- und Wertebereich und Kardinalitätsbeschränkungen auszustatten. Allerdings muss die Überprüfung der Definitions- und Wertebereiche und Kardinalitätsbeschränkungen, wie schon erwähnt, selbst implementiert werden. Dazu kann man schon im Jena-Framework vorhandene Methoden für den Zugriff auf diese Eigenschaften verwenden. Auch das Anlegen von Restrictions (siehe Abschnitt 3.5) ist mit den Methoden des Jena-Frameworks zwar möglich, beinhaltet aber keine automatische Überprüfung der Einschränkungen. Außerdem gibt es keine Möglichkeit an einer von einer Restriction abgeleiteten Klasse die vererbten Einschränkungen abzurufen; diese Werte abzurufen ist nur an der Restriction selbst möglich. Um dies zu ermöglichen müsste man also eine Methode schreiben, die zu einer Klasse alle Oberklassen, die vom Typ Restriction sind, ermittelt und die Einschränkungen von dort abruft. Zusätzlich muss gegebenenfalls die lokale, durch eine Restriction definierte, Beschränkung mit der globalen, durch den Definitions- bzw. Wertebereich der Property definierten, Beschränkung in Einklang gebracht werden.
Auch die Möglichkeit Klassen durch Aufzählen ihrer Mitglieder oder durch Mengenvereinigungen zu erzeugen ist im Jena-Framework nur teilweise implementiert. Es ist zwar möglich Ontologien, die diese Modellierungsmöglichkeiten benutzen einzulesen, aber das Jena-Framework kann nur bedingt mit Instanzen dieser Klassen umgehen [Car02]. Zum Beispiel ist das Einlesen einer Klasse, die durch die Aufzählung ihrer Instanzen definiert ist, aus einer bestehenden Ontologie möglich, in wie weit sich die entsprechende Jena Implementation schon verwenden lässt, ging aus der Dokumentation jedoch nicht hervor.

5.4 Weiterführende Modellierung mit DAML+OIL

Eine Möglichkeit der Modellierung, die für unsere Ontologie nützlich sein könnte, ist, dass man Properties zwischen Klassen anlegen kann. Dies könnte für eine genauere Situationsbezogene Ablaufsteuerung benutzt werden. Hier ein kleines Beispiel:

Es gibt die Klassen A, B, C, D, E und F und die Properties 1, 2, 3, 4, 5, 6 und 7. Die Definitions- und Wertebereiche sind wie folgt definiert:
Property 1: Definitionsbereich: A, Wertebereich: B,
Property 2: Definitionsbereich: A, Wertebereich: C,
Property 3: Definitionsbereich: B, Wertebereich: D,
Property 4: Definitionsbereich: C, Wertebereich: D,
Property 5: Definitionsbereich: D, Wertebereich: F,
Property 6: Definitionsbereich: D, Wertebereich: E,
Property 7: Definitionsbereich: E, Wertebereich: F.

Zusätzlich gibt es ein Schema, das genauer definiert, wann eine Property angewendet werden darf (Abbildung 5.2). Dieses Schema entsteht durch das Anlegen von Properties zwischen Klassen.

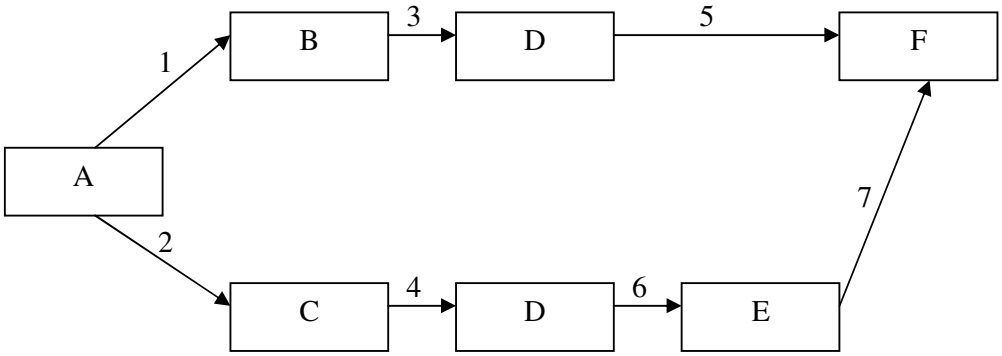


Abbildung 5.2: Properties zwischen Klassen

Durch das entstehende Schema und entsprechende Logik, die dieses auswerten kann, wird es möglich zu bestimmen, in welcher Situation eine Property angewandt werden kann. Im Beispiel soll es möglich sein, auf zwei verschiedenen Wegen die Klasse D zu erreichen. Von dem Weg, auf dem Klasse D erreicht wird, soll es abhängen, welche der Properties mit Klasse D als Definitionsbereich danach angewendet werden kann, d. h., dass Property 5 nur angewendet werden kann, wenn Klasse D auf dem Weg über die Properties 1 und 3 erreicht wurde und Property 6 nur, wenn Klasse D auf dem Weg über die Properties 2 und 4 erreicht wurde. Wenn man hierzu nur die Deklarationen der Properties benutzt, wäre es möglich, unabhängig vom Weg, auf Klasse D sowohl Property 5 als auch Property 6 anzuwenden. Letztere Möglichkeit wird in unserem Prototypen benutzt.
Eine mögliche Anwendung hierfür ist zum Beispiel der Umzug innerhalb Hamburgs (Klasse B in Bild 5.2) und der Umzug von außerhalb nach Hamburg (Klasse C in Bild 5.2). In beiden Situationen gibt es eine Abfrage der alten Adresse (Klasse D in Bild 5.2), aber nur bei einem Umzug von Außerhalb wird auch das Bundesland abgefragt (Klasse E in Bild 5.2). Schließlich vereinigen sich beide Wege wieder, z. B. in einem Formular (Klasse F in Bild 5.2).

Das Jena-Framework eignet sich also gut dazu einfache Ontologien einzulesen oder zu erstellen und zu bearbeiten. Defizite gibt es auf dem Gebiet des automatischen Schließens, d. h. dem Ableiten von implizit in der Ontologie enthaltenen Informationen. Diesem Defizit soll aber in Jena2 (siehe Abschnitt 6.3) abgeholfen werden.

5.5 Überblick über die aufgeführten Probleme

Problem oder nicht genutzte Möglichkeit	**Lösungsansatz**
Die Reihenfolge der Befehle des Jena-Frameworks in der Programmiersprache zum Anlegen der Ontologie kann zu Fehlern führen (siehe Abschnitt 5.1)	Besonders bei Ontologien, die zum Teil eingelesen und zum Teil mit den Methoden des Jena-Frameworks erzeugt werden, muss man sich klar machen, in welcher Reihenfolge die Elemente der Ontologie erzeugt werden und ob benötigte Abhängigkeiten immer erfüllt sind.

Die Anwendung bestimmter Methoden kann in der Programmiersprache zulässig sein, aber ein in der DAML+OIL Spezifikation unzulässiges Ergebnis erzeugen (Abschnitt 5.1).	Man muss selbst darauf achten, dass man in Methoden nur Parameter benutzt, die auch bezüglich der DAML+OIL Spezifikation zulässig sind, oder eine Logik programmieren, die dies übernimmt.
Es gibt unterschiede zwischen den Hierarchien der Interfaces und der Implementationen im Jena-Framework (Abschnitt 5.2).	Eine genauere Dokumentation für das Jena-Framework und gründliche Einarbeitung helfen dieses Problem zu erkennen.
Es gibt Unterschiede zwischen der Hierarchie der DAML+OIL Spezifikation und den Hierarchien im Jena-Framework(Abschnitt 5.2).	Eine genauere Dokumentation für das Jena-Framework und gründliche Einarbeitung in das Jena-Framework und die DAML+OIL Spezifikation helfen dieses Problem zu erkennen.
Es wird nicht überprüft, ob eine Property einzigartig ist (Abschnitt 5.3).	Logik programmieren, die diese Überprüfung vornimmt.
Aufzählungen, Mengenvereinigungen und Restrictions sind im Jena-Framework nur teilweise implementiert (Anschnitt 5.3).	Implementation von eigener Logik, die mit diesen Konstrukten umgehen kann.
Die Möglichkeit Properties zwischen Klassen anzulegen wurde nicht genutzt (Abschnitt 5.4).	Entwurf eines Schemas, dass diese Möglichkeit für eine genauere Ablaufsteuerung benutzt.

6. Zusammenfassung und Ausblick

In diesem Kapitel möchte ich eine Zusammenfassung der Erkenntnisse aus der Arbeit mit dem Jena-Framework und einen kurzen Ausblick auf Jena2, den Nachfolger des Jena-Frameworks, und die Web Ontology Language (OWL) [McG03], eine von DAML+OIL ausgehend entwickelte Modellierungssprache für Ontologien, geben.

6.1 Zusammenfassung

Als Ergebnis aus der Arbeit mit dem Jena-Framework lässt sich sagen, dass sich das Jena-Framework in der von uns verwendeten Version 1.6.0 gut dazu verwenden lässt mit Ontologien umzugehen, die die komplexeren Modellierungsmöglichkeiten (z. B. Restriction, Aufzählung; siehe Kapitel 5) von DAML+OIL nicht benötigen. Im Jena-Framework ist noch keine Unterstützung für das automatische Schließen implementiert. Dadurch bleiben auch einige der komplexeren Modellierungsmöglichkeiten von DAML+OIL ungenutzt. Die Logik im Jena-Framework beschränkt sich momentan auf den Umgang mit Klassen- und Property-Hierarchien und das Auffinden von Äquivalenzen. Logik für die komplexeren Modellierungsmöglichkeiten, die DAML+OIL bietet, muss im Jena-Framework noch selbst implementiert werden. In Jena2 soll dies jedoch geändert und ein stärkeres Gewicht auf die Logik gelegt werden.
Einige der Probleme, die bei uns im Umgang mit dem Jena-Framework auftraten, lassen sich darauf zurückführen, dass das Jena-Framework noch nicht über eine ausführliche Dokumentation, sondern nur über ein einführendes Tutorial [Car02] und eine Dokumentation der Java Klassen verfügt. Hierdurch haben wir auch manche Modellierungsmöglichkeiten, die das Jena-Framework bietet, übersehen und teilweise war aus dem Tutorial oder der Dokumentation der Java Klassen auch nicht klar ersichtlich, wie die Methoden anzuwenden sind. Dabei spielt es natürlich auch eine Rolle, dass sowohl das Jena-Framework als auch DAML+OIL sich noch in der Entwicklung befindet und wir uns somit auf Neuland bewegten.
Auch die Modellierungssprache DAML+OIL ist komplexer, als es uns auf den ersten Blick erschien. Hier hätte uns mehr Zeit für die Einarbeitung sicher geholfen, die Konzepte von DAML+OIL besser zu verstehen und einzusetzen.
Für unsere Zwecke, nämlich die Erstellung eines Prototypen mit einer einfachen Ontologie, erscheint das Jena-Framework in der benutzten Version durchaus geeignet. Da es uns in erster Linie darauf ankam Klassen anlegen zu können und es in dem Entwurf unserer Ontologie keine komplexeren DAML+OIL Konstrukte gab, standen uns auch von Seiten des Jena-Frameworks die benötigten Methoden zur Verfügung. Erst im Laufe der Arbeit am Prototypen kam der Wunsch danach auf, auch die komplexeren Modellierungsmöglichkeiten von DAML+OIL zu benutzen. Hier traten dann auch erst die genannten Defizite des Jena-Frameworks auf. Für eine weitere Entwicklung des Prototypen, mit einer komplexeren Ontologie als der unseren, bietet sich ein Umstieg auf Jena2 (siehe Abschnitt 6.3) an, das die komplexeren Modellierungsmöglichkeiten für das automatische Schließen ausnutzt und dazu anstatt DAML+OIL die Web Ontology Language (OWL, siehe Abschnitt 6.2) benutzt. Da die Entwicklung der Web Ontology Language DAML+OIL als Ausgangspunkt hatte, sollten auch bei der Umstellung der Ontologie von DAML+OIL auf OWL keine größeren Probleme auftreten.

6.2 OWL

Die *Web Ontology Language* (OWL) ist eine Sprache, die, wie RDFS und DAML+OIL, dazu dient, die Bedeutung von Ausdrücken und deren Beziehungen explizit in einem Vokabular festzulegen und damit eine Ontologie zu erzeugen. Für die Entwicklung von OWL dienten die Erfahrungen, die während der Entwicklung und des Einsatzes von DAML+OIL gemacht wurden, als Ausgangspunkt. Daher haben beide Modellierungssprachen ähnliche Funktionalitäten und OWL kann als Revision von DAML+OIL betrachtet werden.
OWL besteht aus drei Untersprachen: OWL Lite, OWL DL und OWL Full.

OWL Lite ist die einfachste und eingeschränkteste dieser drei Untersprachen. Es wird benutzt, wenn man in erster Linie eine Hierarchie und nur einfache Definitions- und Wertebereichs-Einschränkungen benötigt. Es ist z. B. für Kardinalitätsbeschränkungen nur möglich diesen die Werte Null oder Eins zu geben. OWL Lite bietet weniger Ausdrucksmöglichkeiten als OWL DL oder OWL Full, ist dafür aber auch weniger Komplex und es ist daher auch einfacher Software, die diese Sprache unterstützt, zu entwerfen.

OWL DL bietet viele Ausdrucksmöglichkeiten und gewährleistet dabei die Vollständigkeit und die Entscheidbarkeit, d. h., dass alle aus gegebenen Prämissen ableitbaren Folgerungen auch tatsächlich abgeleitet werden und dass dieser Prozess terminiert, d. h. in endlicher Zeit berechnet wird. In OWL DL können alle Möglichkeiten von OWL genutzt werden, allerdings unterliegen einige Einschränkungen, um die Vollständigkeit und die Entscheidbarkeit gewährleisten zu können. Das DL im Namen ergibt sich aus dem Zusammenhang mit description logics, die die formale Grundlage für OWL bilden. Description logics sind eine Familie von Sprachen zur Wissensrepräsentation, die besonders auf dem Gebiet der künstlichen Intelligenz untersucht werden.

OWL Full bietet die kompletten Modellierungsmöglichkeiten von OWL ohne Einschränkungen, aber auch ohne Garantien für die Berechenbarkeit.

Jede dieser Untersprachen ist eine Erweiterung des einfacheren Vorgängers. Dies wiederum bedeutet, dass OWL Lite Ontologien auch OWL DL Ontologien und OWL DL Ontologien auch OWL Full Ontologien sind; die Umkehrung gilt allerdings nicht.
OWL Full kann als eine Erweiterung von RDF angesehen werden, wohingegen dies für OWL Lite und OWL DL nur eingeschränkt gilt. Dies hängt damit zusammen, dass aufgrund der Einschränkungen in OWL Lite und OWL DL nicht alle RDF Dokumente OWL Lite oder OWL DL Dokumente sind. Es sind jedoch alle RDF Dokumente auch OWL Full Dokumente.
Bei dem Entwurf von OWL wurden im Vergleich zu DAML+OIL einige Modellierungsmöglichkeiten hinzugefügt und wenige entfernt, so dass die Ausdrucksmöglichkeiten von OWL Full in etwa vergleichbar mit denen von DAML+OIL sind.

6.3 Jena2

Jena2 ist der Nachfolger des in unserem Projekt genutzten Jena-Frameworks. Außer der schon aus dem Jena-Framework bekannten Unterstützung von RDF und DAML+OIL unterstützt Jena2 auch OWL.
In Jena2 wurde versucht, die Java-Klassen von den Modellierungssprachen zu trennen. Es gibt nun für jede Modellierungssprache ein Profil, das festlegt, welche Modellierungsmöglichkeiten erlaubt sind. Dadurch können Klassen, die in verschiedenen Modellierungssprachen

gleich sind, für diese gemeinsam benutzt werden und es werden nicht mehr für jede Modellierungssprache eigene Java-Klassen benötigt. Mit dieser Umstellung hat sich auch die Klassenhierarchie im Vergleich zum Vorgänger stark verändert.
Jena2 bietet im Vergleich zum Vorgänger eine verbesserte Unterstützung für automatisches Schließen. Realisiert wurde ein solcher Ableitungsmechanismus auf Basis der Sprache OWL Lite. Die Unterstützung des automatischen Schließens bildet nun einen wichtigen Teil von Jena2.
Zu diesem Zeitpunkt befindet sich Jena2 immer noch in der Entwicklung und es sind noch keine vollständigen Dokumentationen vorhanden. Weitere Informationen finden sich auf der Homepage des Jena2 Projekts: http://jena.sourceforge.net/index.html.

Literatur

[Ber01] Tim Berners-Lee, James Hendler, Ora Lassila, *Scientific American: The Semantic Web*, http://www.scientificamerican.com/print_version.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21, 17.05.2001

[Ber03] Tim Berners-Lee, *Standards, Semantics and Survival*, http://www.w3.org/2003/Talks/01-siia-tbl/Overview.html, 2003

[Ber72] Bertelsmann, Das moderne Lexikon, *Ontologie*, Band 13, Seite 388, 1972

[Bol01] Harold Boley, Stefan Decker, Michael Sintek, *Tutorial on Knowledge Markup and Resource Semantics*, http://www.dfki.uni-kl.de/km/kmrs/, 20.08.2001

[Bri03] Dan Brickley, R. V. Guha, *RDF Vocabulary Description Language 1.0: RDF Schema*, http://www.w3.org/TR/rdf-schema/, 23.01.2003

[Car02] Jeremy Carroll, *Jena Tutorial: DAML+OIL – Ontology Description*, http://www.hpl.hp.com/semweb/doc/tutorial/DAML/index.html, 04.2002

[Con01] Dean Connoly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, Lynn Andrea Stein, *DAML+OIL (March 2001) Reference Description*, http://www.w3.org/TR/2001/NOTE-daml+oil-reference-20011218, 18.12.2001

[Dei03] Ulf Deichmann, Fabian Müller, Christoph Kurek, *Visualisierung von Kontextinformationen in WebInfoServices*, Projektbericht am Arbeitsbereich Softwaretechnik der Universität Hamburg, 07.04.2003

[Div03] Diverse, *What is Ontology? Definitions by leading philosophers*, http://www.formalontology.it/section_4.htm, 2003

[Fen00] Dieter Fensel, Ora Lassila, Frank van Harmelen, Ian Horrocks, James Hendler, Deborah L. McGuinness, *The Semantic Web and its languages*, http://www.computer.org/intelligent/ex2000/pdf/x6067.pdf, 12.2000

[Gil00] Yolanda Gil, Varun Ratnakar, *A Comparison of (Semantic) Markup Languages*, http://trellis.semanticweb.org/expect/web/semanticweb/flairs02.pdf, 2000

[Gru93] Thomas R. Gruber, *Towards Principles for the Design of Ontologies Used for Knowledge Sharing*, ftp://ftp.ksl.stanford.edu/pub/KSL_Reports/KSL-93-04.ps, 23.08.1993

[Hay03] Patrick Hayes, *RDF Semantics*, http://www.w3.org/TR/rdf-mt/, 23.01.03

[Hef03] Jeff Heflin, *Web Ontology Language (OWL) Use Cases and Requirements*, http://www.w3.org/TR/webont-req/, 31.03.2003

[ISO99] ISO, *ISO/IEC 13250 – Topic Maps*, http://www.y12.doe.gov/sgml/sc34/document/0129.pdf, 1999

[Kly03] Graham Klyne, Jeremy J. Carroll, *Resource Description Framework (RDF): Concepts and Abstract Syntax*, http://www.w3.org/TR/rdf-concepts/, 23.01.2003

[Las99] Ora Lassila, Ralph R. Swick, *Resource Description Framework (RDF) Model and Syntax Specification*, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/, 22.02.1999

[McB02A] Brian McBride, *Jena: A Semantic Web Toolkit*, http://dsonline.computer.org/0211/f/wp6jena.htm, 11.2002

[McB02B] Brian McBride, *An Introduction to RDF and the Jena RDF API*, http://www.hpl.hp.com/semweb/doc/tutorial/RDF_API/index.html, 2002

[McG03] Deborah L. McGuinnes, Frank van Harmelen, *OWL Web Ontology Language Overview*, http://www.w3.org/TR/owl-features/, 18.08.2003

[Oue03] Roxane Ouellet, Uche Ogbuji, *Introduction to DAML: Part I*, http://www.xml.com/lpt/a/2002/01/30/daml1.html, 1998-2003

[Pep00] Steve Pepper, *The TAO of Topic Maps*, http://www.ontopia.net/topicmaps/materials/tao.html, 2000

[Sim02] Kiril Simov, *DAML+OIL Syntax Overview and Analysis*, http://www.sirma.bg/OntoText/publications/daml-oil-syntax.htm, 05.04.2002

[Ver03] Joe Verzulli, *Using the Jena API to Process RDF*, http://www.xml.com/lpt/a/2001/05/23/jena.html, 1998-2003

[Zue01] Heinz Züllighoven, *Praktische Informatik 2 (Skript)*, Seite 1.12, Sommersemester 2001