Integration von Ontologien in objektorientierte Programmiersprachen als verteilte und dynamische Datenmodelle

Masterthesis

eingereicht zur Erlangung des akademischen Grades

Master of Science

Erster Prüfer Prof. Dr. Kristian Fischer
Zweiter Prüfer Prof. Christian Noss

Fachhochschule Köln
Cologne University of Applied Sciences

Fakultät für Informatik und Ingenieurswissenschaften
Studiengang Medieninformatik Master

Lars Brillert (04.05.1985, Berg. Gladbach)

Gummersbach, März 2010

The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.

Tim Berners-Lee, 2001

Zusammenfassung

Immer mehr Teilbereiche des Semantic Web sind in den letzten Jahren erfolgreich umgesetzt geworden. Ebenso wird bei der Bearbeitung von komplexen Problemräumen mittlerweile oft auf semantische Modelle zurückgegriffen, um eine flexible Beschreibung der Domäne zu erstellen. Werkzeuge, welche die Entwicklung von Anwendungen, die auf semantischen Modellen basieren, unterstützen sind bislang jedoch nur in begrenztem Maße verfügbar. Insbesondere die Verarbeitung von verteilten und dynamischen Modellen ist mit keinem der derzeit verfügbaren Produkte vollständig zu realisieren.

Diese Arbeit untersucht die Möglichkeiten zur Integration von semantischen Modellen in objektorientierte Programmiersprachen. Es werden bestehende Ansätze analysiert und ein formales Modell der Integration erstellt. Das formale Modell wird in Form eines prototypischen Rahmenwerks in der Programmiersprache Ruby implementiert und validiert.

Abstract

Many aspects, that were part of Tim Berners-Lee's vision of the semantic web, are now in use. As well in the context of complex semistructured problemspaces, semantic models were used to gather a flexible description of the domain of interest. But tools for supporting the development based on semantic models are incomplete and too complex for pushing semantic applications beyond their limits. Especially the processing of distributed and dynamic models in an object-oriented manner is not covered by any of the products presently available.

This thesis examines the possibilities of integrating ontologies into object-oriented programming languages. Based on the analysis of existing approaches a formal model will be established. In order to validate the model, a reference implementation for the programming language Ruby will be developed.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

1.1. Motivation

Seit Tim Berners-Lee im Jahr 2001 seine Vision des semantischen Webs [1] beschrieben hat, haben sich zum einen die Techniken der semantischen Verarbeitung weiterentwickelt, aber auch das Web als selbstständiges Ökosystem hat sich von damals bis heute stark verändert.

Auf der Seite der Semantik sind Entwicklungsumgebungen gewachsen, die helfen sollen Ontologien aufzubauen und bestehende Datenbestände in semantische Datenmodelle zu überführen [2]. Zusätzlich wurden die Ideen von Tim Berners-Lee in der Linked Data Initiative[1] [3, 4] aufgegriffen und, mit den heute verfügbaren Techniken, begonnen diese umzusetzen. Mit großem Erfolg wie sich zeigte, denn das Linked Data Web enthält ungefähr 4,7 Milliarden Tripel (Stand: März 2009; Mai 2007: 1 Milliarde Tripel). Immer mehr Unternehmen setzen semantische Techniken ein [5] um spezielle Problemstellungen zu lösen. Mit Google und Yahoo haben auch zwei der größten Suchmaschinenanbieter damit begonnen semantische Metadaten in ihre Suchindexe aufzunehmen und für Suchanfragen auszuwerten.

Nach wie vor gilt die Entwicklung von semantischen Webanwendungen, also Anwendungen, die über das Netz zugreifbar sind und deren Datenmodell größtenteils aus semantischen Modellen besteht, als unangemessen kompliziert, da es an geeigneten Werkzeugen und Framworks mangelt, welche die Entwicklung unterstützen. Im Gegensatz dazu haben sich für klassische Anwendungsarchitekturen auf Basis von relationalen Datenbanken für fast jede Programmiersprache Frameworks entwickelt, welche die Entwicklung durch die Bereitstellung von typischen immer wiederkehrenden Funktionen vereinfachen. Dazu gehört insbesondere der Zugriff auf die angebundene Datenbank, der durch die Frameworks fast vollständig abstrahiert wird und es so dem Entwickler

[1] http://linkeddata.org

ermöglicht durchgängig objektorientiert zu arbeiten.

Der Durchbruch des Semantic Web hängt jedoch auch mit der breiten Verfügbarkeit von Angeboten zusammen, die dessen Fähigkeiten aufgreifen und benutzen. Um dies zu erreichen, sollten sich semantische Webanwendungen mit den um semantische Datenmodelle erweiterten etablierten Werkzeugen der klassischen Web-Entwicklung umsetzen lassen, um die Vorteile beider Welten miteinander zu verbinden.

1.2. Fragestellung

Die vorliegende Arbeit wird sich mit der Frage beschäftigen, inwieweit sich semantische Modelle unter Beachtung der Aspekte Dynamik, Schemaeinflüsse und Verteiltheit in objektorientierte Programmiersprachen integrieren lassen.

Dynamik Bei einem semantischen Modell handelt es sich um eine dynamische Datenquelle. In erster Linie verändern sich die Aussagen zu einzelnen Individuen. Diese Änderungen des Datenbestands müssen auch zur Laufzeit erkannt und verfügbar gemacht werden. Dies gilt insbesondere auch für die zugrundeliegenden Schemata, die einer zeitlichen Änderung unterworfen sind.

Schemaeinflüsse Anders als bei der Benutzung von relationalen Daten, bei denen die Eigenschaften direkt mit einem Objekt verwoben sind, entsteht im semantischen Modell durch das Reasoning eine weitere Annotationsebene. Mit Hilfe des Schemas kann bestimmt werden, warum Ressourcen mit bestimmten Eigenschaften verbunden sind. Auf der Benutzerebene kann man dieses Wissen unter anderem dafür einsetzen, die „Rezeptionstiefe“ zu erhöhen; neben den reinen Informationen lässt sich auch deren Herkunft erklären.

Verteiltheit Das Semantic Web, oft auch als „Web der Daten“ bezeichnet, propagiert die Vernetzung und Wiederverwendung von dezentralem Wissen. Die Operationalisierung dieser Ideen, also der strukturierte Zugriff auf Wissen außerhalb des eigenen „Hoheitsgebiets“ ist bislang nicht abschließend gelungen. Gerade deshalb und auch weil andere Ansätze diese Thematik gar nicht adressieren, sollte dieser Aspekt angesprochen werden, um zumindest ein paar grundsätzliche Lösungsansätze zu skizzieren.

1.3. Aufbau

Die Arbeit beginnt mit der Rekapitulation des Paradigmas der Objektorientierung (2.1) und dessen Bedeutung sowie Implikation auf Programmiersprachen und Programmierer. Im weiteren werden einige ausgewählte Architekturmuster (2.2) für interaktive Systeme erläutert und der Begriff des Frameworks (2.3), insbesondere in Bezug auf die Objektorientierung beleuchtet.

Ein weiterer Grundstein dieser Arbeit ist die semantische Modellierung von Daten. Kapitel 3 liefert die Definition dieses Begriffs und verdeutlicht die Bedeutung dieses Konzepts für bestehende und zukünftige Informationstechnische Systeme im Allgemeinen und im Besonderen von webbasierten Anwendungen.

Auf Basis der Erkenntnisse der vorhergehenden vergleicht Kapitel 4 die Paradigmen semantische Modellierung und Objektorientung, um Rückschlüsse auf die Integration von semantischen Daten in objektorientierte Programmiersprachen zu gewinnen. Die Ergebnisse des Vergleichs werden in ein formales Modell überführt werden, das dazu dienen soll, eine allgemeingültige Integration zu beschreiben, die als Grundlage zur Apdation in verschiedenen Programmiersprachen dienen soll.

Im Kapitel 5 wird das formale Modell in der Programmiersprache Ruby implementiert. Neben der umfangreichen Auseinandersetzung mit den erfolgten Designentscheidungen wird auch ein Anwendungsszenario (5.5) implementiert, um die Einsatzmöglichkeiten plastisch aufzuzeigen.

2. Grundlagen

2.1. Objektorientierung

Objektoerientierung beschreibt ein Paradigma, mit dem ein System als Zusammenhang aus mehreren Objekten beschrieben werden kann. Der Begriff des Objekts ist über mehrere Quellen divergent definiert. In der Definition der IEEE wird ein Objekt als Kapselung von Daten und Diensten zur Manipulation dieser Daten charakterisiert.

> „**object**. (...) (3) An encapsulation of data and services that manipulate that data. See also: **object-oriented design**.“ [6]

Der Begriff *objektorientiertes Design*, welcher in der IEEE-Defition verwendet wird, ist wie folgt definiert:

> „ **object-oriented design**. A software development technique in which a system or component is expressed in terms of objects and connections between those objects (...)“ [6]

Wie in der Definition des IEEE wird der Begriff der Objektorientierung im Kontext dieser Arbeit dem Bereich der objektorientierten Programmierung zugeordnet, auch wenn sich Objektorientierung auch in andere Bereiche übertragen lässt. In der objektorientierten Programmierung wird insofern versucht, durch die Verwendung von Objekten die Komplexität der entwickelten Softwaresystemen zu verringern, um die Entwicklung und Wartung zu vereinfachen [7].

Die Anzahl an objektorientierten Programmiersprachen ist scheinbar unendlich lang und die Unterscheidungsmerkmale der einzelnen Sprachen sind in vielen Fällen nur sehr gering ausgeprägt. Ein grundsätzliches Merkmal ist beispielsweise die vertikale Integration des objektorientierten Gedankens; während die Programmiersprache Ruby ausschließlich mit Objekten arbeitet, gibt es auch Sprachen, die unterschiedliche Paradigmen vermischen z.B. Objective C 2.0. Grundsätzlich ist zudem der Trend zu erkennen, dass sich ehemals rein pro-

zedurale Sprachen langsam in Richtung Objektorientierung entwickeln. Unabhängig davon, wie spezifische Details innerhalb einer Sprache umgesetzt werden, existiert eine Obermenge an Eigenschaften, die der gesamten Menge aller objektorientierten Programmiersprachen gemein sind.

Klassen kommt im Kontext der objektorientierten Programmierung die Aufgabe zu, eine einheitliche Schnittstelle für Objekte gleichen Typs zu definieren. In der Regel können instanziierte Objekte nur zu einer direkten Klasse gehören, so spricht [8] von Klassen als Schablonen, aus denen die Objekte „herausgestanzt“ werden, d.h. das Verhalten, der Zustand und die Identität [9] von Objekten wird initial durch ihre Klasse vorgegeben. Über Konzepte wie *Schnittstellen* oder *abstrakte* Klassen lässt sich in einigen Sprachen diese strikte Kopplung zwischen Objekt und nur einer Klasse reduzieren.

Vererbung erlaubt das Ableiten von Klassen von anderen Klassen. Dies bedeutet, dass eine *Unterklasse*, die von einer *Basisklasse* abgeleitet wird, dessen Eigenschaften und Methoden erben kann. So lassen sich Klassenhierarchien definieren, in denen allgemeinere Eigenschaften lediglich einmal spezifiziert werden müssen; spezialisiertere Klassen können hierauf zurückgreifen. Vererbung hilft in erster Linie der Strukturierung einer Softwarearchitektur durch die Vermeidung von doppelten Funktionen.

Datenkapselung spielt in der objektorientierten Programmierung, im Gegensatz zur strukturierten Programmierung [10], eine wichtige Rolle. Durch die Kapselung soll der ungefilterte Zugriff auf den Zustand eines Objekts verhindert werden. Anstatt direkt mit den internen Daten zu arbeiten, werden einheitliche Schnittstellen bereitgestellt, um die Daten zu manipulieren. Im einfachsten Fall werden die Eigenschaften über *getter/setter*-Methoden öffentlich zugänglich gemacht, gerade der schreibende Zugriff kann aber spezifischer ausgeprägt werden, um beispielsweise Konsistenzprüfungen der Eingabedaten vorzunehmen oder direkt mehrere Attribute zu verändern.

Polymorphie bedeutet, dass der Typ des Objekts zur Laufzeit über die aufzurufende Methode entscheidet. Dabei ist Polymorphie eng mit den Konzepten Vererbung und Schnittstellen verbunden, mit denen sich die Flexibilität der Programme steigern lässt, da Objektreferenzen benutzt werden können ohne einen expliziten Typ festlegen zu müssen. Durch den Polymorphismus von Klassen wird die Flexibilität weiter gesteigert, da

Methoden ausgeführt werden können ohne zuvor genau zu wissen, welche Methode letztendlich aufgerufen wird. Wichtig ist die Abgrenzung zur Überladung und Redefinition von Methoden, die zwar ein ähnliches Verhalten produzieren, jedoch bereits zur Übersetzungszeit fest gebunden werden.

2.2. Architekturmuster

Architekturmuster werden in der Softwareentwicklung eingesetzt, um die grundsätzliche Organisation und Interaktion zwischen einzelnen Komponenten zu beschreiben und liefern daher nur abstrakte Empfehlungen für den Aufbau von Softwareprodukten. Architekturmuster können in verschiedene Kategorien eingeteilt werden: *Verteilte Systeme*, *Adaptive Systeme* und *Interaktive Systeme*. Im Kontext dieser Arbeit sind jedoch lediglich Muster für interaktive Systeme von Interesse.

2.2.1. Model View Controller

Das Model-View-Controller Paradigma [11] (MVC) ist ein Architekturstil für interaktive Systeme, bei denen das Datenmodell (*Model*), die Benutzersicht auf das System (*View*) und die Anwendungslogik (*Controller*) von einander abgetrennt sind und über definierte Schnittstellen miteinander kommunizieren können.

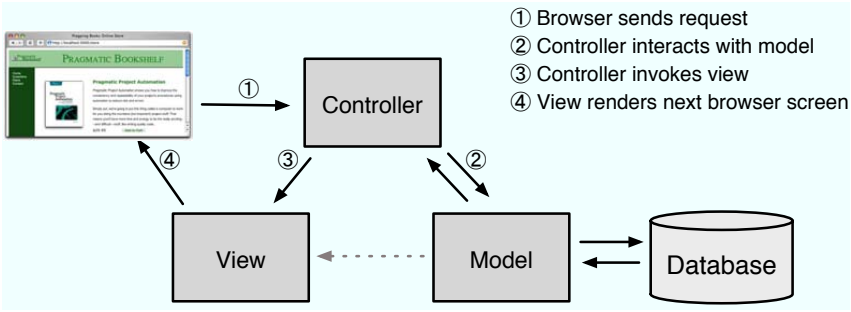

Abbildung 2.1.: Model-View-Controller [12]

In Abbildung 2.1 ist das Zusammenspiel der einzelnen Komponenten im Kontext einer „klassischen“ Webanwendung dargestellt. Der *Controller* interpretiert die Eingaben der *View* und gibt Veränderungen an diese weiter. Ebenso ist er verantwortlich für die Verwaltung des *Model*, d.h. z.B. die Durchführung

der CRUD-Methoden. Die *View* ist dafür zuständig, die vom Controller empfangenen Informationen auf einem entsprechenden Ausgabegerät darzustellen oder an ein solches zu vermitteln. Das *Model* enthält das Verhalten und den Zustand der Anwendung und reagiert auf Aufrufe des *Controllers* indem es die entsprechenden Informationen bereitstellt.

Durch die Entkopplung der einzelnen Komponenten lässt sich ein Softwareprodukt, welches nach MVC entwickelt wird, leicht zwischen mehreren Entwicklern aufteilen, die dann jeweils unabhängig[1] voneinander *Controller, Modell* oder *View* entwickeln. Zusätzlich erhöht MVC die Wiederverwendbarkeit und Wartbarkeit der Anwendung: So kann beispielsweise die Implementierung des Modells umgestellt werden, ohne dass dies Einfluss auf den zugehörigen Controller hat.

2.2.2. Presentation Abstraction Control

Der Presentation-Abstraction-Control [13] Ansatz geht in die selbe Richtung wie MVC, allerdings wird durch ihn zusätzlich auch die Problematik der Unterteilung einer Anwendung in einzelne Funktionskomponenten adressiert. Hierzu wird das System in die drei Komponenten *Presentation, Abstraction* und *Control* aufgeteilt, welche zu *Agenten* zusammengefasst werden, die Teilfunktionen des Systems abbilden. Die Agenten werden in eine hierarchische Ordnung gebracht. Jede Anwendung besitzt einen *Top-Level*-Agenten, der grundlegende Aufgaben der Anwendung abhandelt und mehrere *Bottom-Level* Agenten, die den Teilfunktionen des Systems entsprechen. Zur besseren Organisation können zusätzlich *Intermediate-Level* Agenten eingeführt werden, die mehrere Teilfunktionen zusammenfassen und an den Wurzelagenten anbinden.

Die Agenten können sich, wie erwähnt, aus den Komponenten: *Presentation*, welche die gesamte Ein/Ausgabe verwaltet, *Abstraction*, welche das Datenmodell des Agenten repräsentiert und *Control*, welche die Steuerung und die Kommunikation zu den anderen Agenten implementiert, zusammensetzen. Jeder Agent muss eine *Control*-Komponente besitzen, die beiden anderen sind optional nach Bedarf zu implementieren.

[1] Vorausgesetzt, dass zuvor Kommunikationsschnittstellen definiert worden sind.

2.3. Framework

Die Wiederverwendung von bestehenden Komponenten und Architekturen gilt als wesentlicher Aspekt der Softwareentwicklung, um den Entwicklungsprozess definierter und schneller zu gestalten. Eine Möglichkeit hierzu stellt die Verwendung von Frameworks dar, die bestehende Funktionen kapseln und dem Entwickler strukturiert zur Verfügung stellen.

> „**Software-Framework**: Ein Software-Framework stellt typischerweise ein halbfertiges Architekturgerüst für einen (komplexen) Anwendungsbereich dar, das auf die Bedürfnisse und Anforderungen einer konkreten Anwendung aus diesem Anwendungsbereich angepasst werden kann.“ [14]

Frameworks sind in der Regel sehr spezifisch auf eine Domäne zugeschnitten, da ihre Entwicklung in vielen Fällen auf eine langjährige Entwicklungstätigkeit in der entsprechenden Domäne zurückgeht. Aber auch der umgekehrte Fall ist möglich, in [15, 16] wurde mit COSIMA ein Framework aus einem rein konzeptuellen Hintergrund spezifiziert und implementiert.

Charakteristisch für Frameworks ist der starke Eingriff in zum einen die Anwendungsarchitektur, die zumeist vollständig durch das Framework vorgegeben wird und zum anderen das Entwicklungsvorgehen, da der Kontrollfluss der Anwendung umgekehrt wird. Um dennoch eine flexible Plattform darzustellen, auf der Anwendungen entwickelt werden können, lassen sich Frameworks über Variationspunkte anpassen.

2.3.1. Umkehrung des Kontrollflusses

In der „klassischen“ Anwendungsentwicklung wird wiederverwendbarer Kode üblicherweise in Klassenbibliotheken ausgelagert und von der Anwendung an den benötigten Stellen aufgerufen. Dieses Aufrufen von Methoden wird *Call-Down*-Prinzip genannt und wird in Abbildung 2.2a dargestellt. Wird eine Anwendung auf Basis eines Frameworks entwickelt, verschiebt sich das *Call-Down*-Prinzip zu einem *Callback*-Prinzip (vgl. Abbildung 2.2). Hierbei liegt die Kontrolle der Ausführungsreihenfolge nicht mehr bei der Anwendung, sondern wird durch das Framework gesteuert. Die Grundarchitektur wird von Framework vorgegeben, anwendungsspezifische Teile werden dann durch das Framework eingebunden.

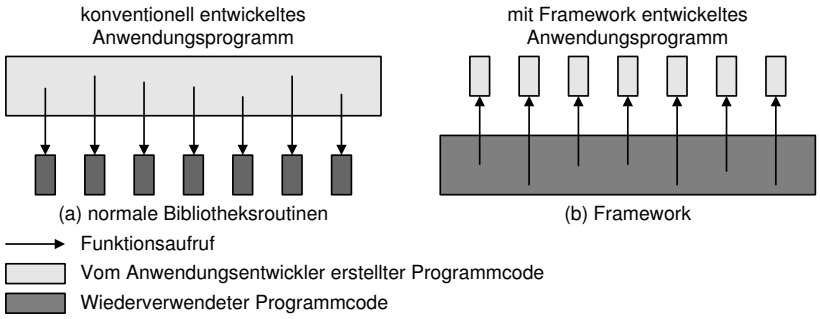

Abbildung 2.2.: Darstellung des umgekehrten Kontrollflusses [14]

2.3.2. Vorgabe einer Anwendungsarchitektur

Durch die Umkehrung des Kontrollflusses kann das Framework die grundlegende Anwendungsarchitektur gezielt vorgegeben. Durch das Framework wird somit ein „Skelett“ [17] vorgegeben, in das bei der Anwendungsentwicklung die domänenspezifischen Teilkomponenten eingehängt werden, so dass sich die Entwickler auf die Details der Anwendung konzentrieren können, anstatt sich mit Fragen über die Strukturierung der Anwendung aufzuhalten.

Rahmenwerke stehen hier auf einem schmalen Grad, zum einen soll möglichst viel Funktionalität durch das Framework abstrahiert werden, auf der anderen Seite sollte das Framework generisch genug gestaltet sein, um möglichst unterschiedliche Anwendungsfälle der betrachteten Domäne umsetzen zu können.

2.3.3. Anpassbarkeit durch Variationspunkte

Bei Frameworks handelt es sich um einen „unvollständigen Entwurf und eine unvollständige Implementierung für Anwendungen in einer speziellen Domäne bzw. einem Problembereich“ [18]. Um dieser Definition gerecht zu werden, muss das Framework Schnittstellen bieten, an denen es um anwendungsspezifische Details erweitert werden kann.

Über Variationspunkte lassen sich Frameworks um die benötigten Aspekte erweitern. Nach dem *offen/geschlossen* - Prinzip sollen über diese Variationspunkte lediglich Erweiterungen, aber keine Modifikationen zugelassen werden, um die Integrität des Frameworks nicht zu gefährden. Da die Anzahl der möglichen Ausprägungen der Variationspunkte nicht zuvor spezifiziert ist, wird auch von *offenen* Variationspunkten [14] gesprochen. Sie stellen für bestimmte Funktionen Rümpfe, bereit die dann durch die jeweilige Anwendung

ausgeformt werden.

2.3.4. objektorientierte Web-Frameworks

Mit Rails[2] (Ruby), Django[3] (Python) und Catalyst[4] (Perl) und vielen weiteren stehen für die meisten Programmiersprachen Frameworks für die Entwicklung von Webanwendungen zur Verfügung. Auch wenn es zwischen den einzelnen Frameworks Unterschiede, gibt wie bestimmte Detailfragen gelöst wurden, so folgen die meisten Frameworks grundsätzlich den gleichen Annahmen.

Als Defacto-Standardarchitekturstil hat sich das MVC-Paradigma entwickelt. Hier kommt insbesondere dem *Modell* eine wesentliche Bedeutung zu, welche die weite Akzeptanz und Verbreitung von Webframeworks erst möglich gemacht hat. Modere Webframeworks sind auf die Zusammenarbeit mit relationalen Datenbanken ausgelegt, durch die konsequente Nutzung von ORM wurde es erstmals möglich, Webanwendungen nach objektorientierten Maßstäben zu entwickeln und die komplette Datenhaltung für den Entwickler zu abstrahieren.

[2] http://www.rubyonrails.org/
[3] http://www.djangoproject.com/
[4] http://www.catalystframework.org/

3. Semantische Modellierung

3.1. Semantic Web

Der Begriff des Semantic Web wurde in erster Linie durch den visionären Aufsatz von Tim Berners-Lee [1] im Jahre 2001 geprägt. Berners-Lee skizziert ein „Internet der Daten“, das von Maschinen verstanden, verarbeitet und zu Neuem integriert werden kann.

Das Semantic Web beschreibt allerdings kein „neues“ Internet, sondern soll eine Erweiterung zum bestehenden dokumentorientierten Internet darstellen und auf dessen Strukturen und Techniken aufsetzen. Anders ausgedrückt kann das Semantic Web auch als eine alternative Sichtweise auf das World Wide Web verstanden werden, in diesem Fall eine Sichtweise, die überwiegend von Maschinen genutzt werden wird. Prinzipiell kann dieses Vorgehen mit dem Aufkommen von RSS/ATOM Feeds für Webseiten verglichen werden. Auch hier wird der gleiche Inhalt in eine andere Repräsentation überführt, um definierte Tätigkeiten zu unterstützen, in diesem Fall die *automatisierte* Auslieferung von Veränderungen auf einer abonnierten Webseite.

Trotz der Anlehnung an das bestehende World Wide Web liegt dem Semantic Web ein eigenständiger Architektur-Stack [19, 20] zugrunde, der gemeinhin als *Semantic Web Tower* (siehe Abbildung 3.1) betitelt wird und die einzelnen Technikschichten beschreibt, auf denen das Semantic Web aufbaut. Das Schichten-Modell lässt sich zusätzlich in drei Kategorien teilen: *Base*, *Knowledge* und *Security*.

Base definiert die Kerntechniken, welche die Grundlage für das Semantic Web bilden. Hier finden sich überwiegend Techniken, die auch jetzt schon im Internet eingesetzt werden. XML mit den Zusätzen Namespaces und Schema als Austauschformat für Daten sowie URI/IRI als Adressierungsschema für Ressourcen sind keine Unbekannten. Unicode als Zeichensatz wird im Internet mit zunehmender Tendenz verwendet, um Inkompati-

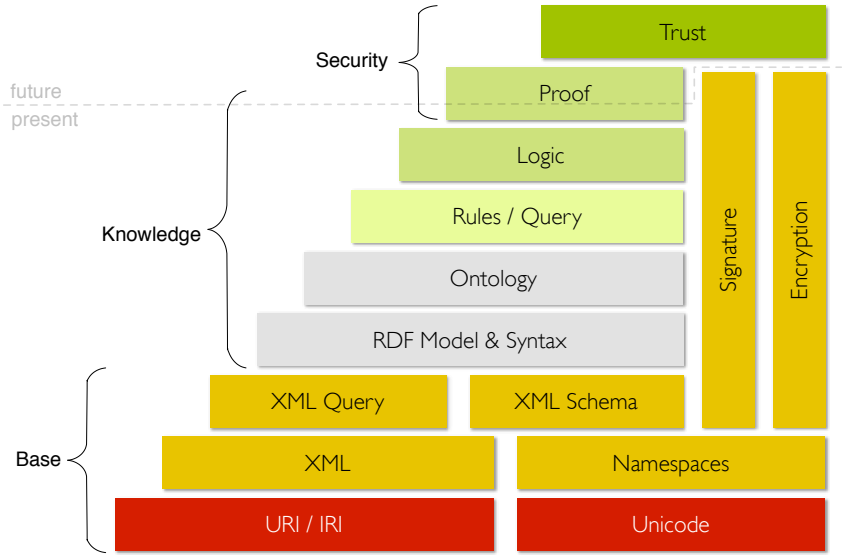

Abbildung 3.1.: Semantic Web Tower: Schichtenmodell des Semantic Web, modifiziert nach [21]

bilitäten zu begegnen, die früher durchaus an der Tagesordnung waren.

Knowledge definiert die Techniken und Standards, die eigens für das Semantic Web entworfen wurden. Diese Metaschicht beginnt mit RDF als grundlegendes Rahmenwerk zur Beschreibung von beliebigen Ressourcen und deren Relationen untereinander. Auf RDF (3.3.1) aufbauend definiert die Ontologieschicht weitere Sprachformate, aber auch invariant Ontologien/Vokabulare (3.3.2), um Wissen im Semantic Web zu modellieren. Die darüberliegenden Schichten Rules, Query und Logic regeln zum einen den Zugriff auf semantische Datenbestände (SPARQL) und zum anderen die Erschließung von implizitem Wissen über Regeln oder Logikkonstrukte.

Security Neben den expliziten Sicherheitsmaßnahmen *Signaturen* und *Verschlüsselung*, die sich über die vorangegangenen Schichten erstrecken, dienen die zwei obersten Schichten *Proof* und *Trust* auch dem Sicherheitsgedanken. Die Proof-Schicht übernimmt die technische Verifikation von Wissen innerhalb des Semantic Web, mit Hilfe von logischen Operationen lassen sich z.B. einzelne Fakten auf ihre Richtigkeit überprüfen oder zumindest Kollisionen mit anderen Definitionen auffinden. Allein aus dem Grundgedanken des Web kann jeder jederzeit über jedes Thema eine Aussage treffen; im Kontext der automatischen Verarbeitung von Inhalten sind Strukturen, die die „Glaubwürdigkeit“ einer Aussage bewerten, ein wesentlicher Aspekt um die Integrität des Semantic Web zu

gewährleisten. Die *Trust*-Ebene, als oberste Schicht des Semantik Web, soll es ermöglichen, Vertrauensstrukturen im Semantic Web zu etablieren, um die Herkunft und Echtheit von Informationen verifizieren zu können.

Zum derzeitigen Zeitpunkt sind fast alle Schichten des Semantic Web verfügbar. Im wesentlichen sind es die Sicherheitsaspekte, die noch nicht spezifiziert und implementiert sind. Die vertikalen, mehrere Schichten umfassenden, Elemente Signierung und Verschlüsselung könnten beispielsweise mit Hilfe von XML Signature/Encryption [22, 23] umgesetzt werden, die Umsetzung der Proof- und Trustebene ist hingegen zeitlich noch nicht festzustellen. Trotz der momentan noch fehlenden Sicherheitsaspekte existiert bereits eine Umsatzung des Semantic Web, das auf den existierenden Standards aufbaut; nämlich das Web of Data, welches in erster Linie durch die Linking Open Data Initiative vorangetrieben wird. Auch wenn aufgrund der unterschiedlichen Namensgebung auf den ersten Blick die Zusammengehörigkeit nicht direkt ersichtlich ist, handelt es sich hierbei um die Umsetzung der theoretischen Gedanken des Semantic Web, in erster Instanz allerdings beschränkt auf die Verknüpfung und Wiederverwendung von Wissensbeständen. Die weiterführenden Ideen von Berners-Lee zum intelligenten Agenten sind weiterhin noch in der Zukunft verortet.

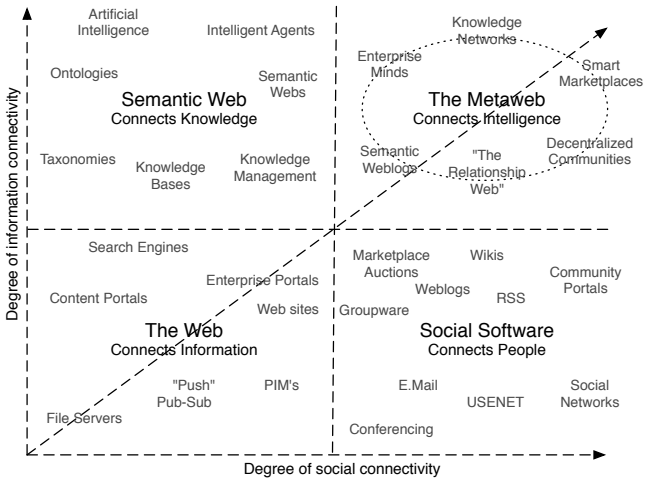

Abbildung 3.2.: Das Metaweb: Semantic Web und Social Web wiedervereint

Nebenläufig zu den Implementierungversuchen des Semantic Web wird schon dessen Weiterentwicklung diskutiert. Während das Semantic Web eine Erweiterung des syntaktischen Webs um die semantische Ebene darstellt, wird in [24, 25] die weitere Entwicklung des Internets prognostiziert, die sich in wesent-

lichen Teilen an dem Begriff des Metawebs[1] von Nova Spivac aufhängt. Dem Metaweb (Abbildung 3.2) liegt die Annahme zugrunde, dass die Seitenarme verstärkter sozialer Verknüpfung (social web) und verstärkter informationeller Verknüpfung (semantic web) zusammengeführt werden. Was hier von Spivac „Metaweb“ genannt wird, findet sich an anderen Stellen auch unter den Titeln *Web 3.0* oder *social semantic Web*, kürzlich hat sich auch *future internet* als Begriff für diese Entwicklung herausgebildet. Bekanntlich sind Namen „Schall und Rauch“, hinter all diesen Begriffen verbirgt sich die natürliche Evolution des Internets.

3.2. Semantisches Datenmodell

Traditionelle Datenmodelle bestehen in der Regel aus zwei voneinander separierten Teilen: dem Schema der Daten und den eigentlichen Daten. Ziel der semantischen Modellierung ist es, diese beiden Teile zu einem Datenmodell zu verbinden.

> „This is the essence of semantic data modelling: flexible schemas where the relationships are described by the data itself.“ [26]

3.2.1. Tabellarische Daten

Die wohl häufigste Form, in der Daten dargestellt werden, ist eine Tabelle ganz gleich welchen Bereich man betrachtet, Tabellen sind allgegenwärtig z.B. Speisekarten, Fahrpläne, Stundenpläne oder Öffnungszeiten. Alle diese Daten werden in tabellarischer Form dargestellt.

#	Name	Verkaufte Portionen
48	Pasta Genovese	6,5
50	Pizza Salami	4,1
92	Tomatensuppe	12

	Name	Preis
48	Pasta Genovese	6,5
50	Pizza Salami	4,1
92	Tomatensuppe	12

Abbildung 3.3.: Ein Datensatz und zwei unterschiedliche Interpretationen

Ein essentielles Merkmal tabellarischer Daten ist die Trennung von Inhalt und Bedeutung. Abbildung 3.3 zeigt eine Tabelle, welche zwar die gleichen Daten darstellt, allerdings mit unterschiedlicher Bedeutung der Daten. Während im

[1] http://novaspivack.typepad.com/nova_spivacks_weblog/2004/03/the_metaweb_is_.html

ersten Beispiel die rechte Spalte für den Preis eines Gerichts steht, wird dieser Zahl im zweiten Bespiel die Menge der verkauften Portionen zugeordnet.

So simpel das Beispiel aus Abbildung 3.3 auch auf den ersten Blick erscheint, wird dadurch ein wesentlicher Punkt deutlich. Durch die Trennung von Daten und Bedeutung ist es zwar möglich diese beiden Teile unabhängig voneinander zu übertragen, allerdings lassen sich die Daten nur korrekt interpretieren, wenn auch die Verknüpfung zu ihrer Bedeutung gegeben ist. Daraus resultiert, dass ein Datensatz ohne zugehörige Bedeutungserklärung wertlos ist.

3.2.2. Relationale Daten

Für rein tabellarische Daten, wie sie in Abschnitt 3.2.1 vorgestellt wurden, sind für *flache* Datensätze durchaus ausreichend. Besteht allerdings der Bedarf unter einer Spalte mehr als eine Information unterzubringen (Abbildung 3.4), werden Tabellen in der Regel schnell unübersichtlich. Um die Übersichtlichkeit der Daten wiederherzustellen kann eine Normalisierung des Schemas vorgenommen werden. Die Spalte, welche die multiplen Informationen enthält, wird hierbei in eine eigenständige Tabelle ausgegliedert und über *künstliche* Schlüssel mit der Elterntabelle verknüpft.

#	Name	Verkaufte Portionen
48	Pasta Genovese	morgens: 2,3; mittags: 10,3; abends: 7,8
50	Pizza Salami	morgens: 1,0; mittags: 11,1; abends: 8,9
92	Tomatensuppe	morgens: 4,8; mittags: 5,2; abends: 10,8

Abbildung 3.4.: Denormalisiertes Schema

id#	Zeitraum	Verkaufte Portionen
1	morgens	2,3
1	mittags	11,1
1	abends	10,8
...	...	...

#	Name	Verkaufte Portionen
48	Pasta Genovese	1
50	Pizza Salami	2
92	Tomatensuppe	3

Abbildung 3.5.: Normalisiertes Schema

Das abschließend entstandene Schema der Daten wird *relationales* Datenschema genannt und ist in Abbildung 3.5 dargestellt. Diese Form eines Datenmodells stellt die in der Softwareentwicklung vorherrschendste Art dar, wie Daten abgebildet und gespeichert werden. Da es sich bei relationalen Daten nach wie vor um tabellarische Daten handelt, gilt auch für diese Datenmodelle die strickte Trennung von eigentlichen Daten und deren Struktur bzw. Semantik.

3.2.3. Verbindung von Struktur und Daten

Das Datenschema des vorherigen Abschnitts 3.2.2 ist nicht flexibel, wenn beispielsweise neue Eigenschaften zu den Mahlzeiten hinzugefügt werden sollen. Hierzu müssten neue Tabellen und Relationen eingeführt werden. Abbildung 3.6 zeigt einen Ansatz, in dem auf die Modellierung von einzelnen Konzepten verzichtet wurde und die Eigenschaften als *key/value* Paare dargestellt werden. Soll das Modell nun um weitere Eigenschaften ergänzt werden, sind dazu lediglich Änderungen am Datenbestand, nicht aber am Schema notwendig.

Meal

#	Name
48	Pasta Genovese
50	Pizza Salami
92	Tomatensuppe

Field

ID	name
1	portionen morgens
2	portionen mittags
3	portionen abends

Properties

mealID	fieldID	value
48	1	2,3
48	2	10,3
48	3	7,8
50	1	1,0
50	2	11,1
50	3	8,9
92	1	4,8
92	2	5,2
92	3	10,8

Abbildung 3.6.: Überführung

Nach wie vor handelt es sich aber um ein relationales Datenschema, welches darauf beruht, die Relationen über Schlüsselbeziehungen herzustellen. Um die Semantik der Beziehungen jedoch explizit zu machen, ist es erforderlich die Schlüsselbeziehungen aufzulösen. In Abbildung 3.7 wird dargestellt wie ein entsprechendes Datenmodell aussieht. Das gesamte Datenmodell wurde denormalisiert und in eine flache Form überführt, in der lediglich *key/value* Paare vorhanden sind. Aus Sicht der relationalen Datenmodellierung ist dies vollkommen ungenügend, da ein solches Datenschema den Sinn einer relationalen Datenbank verkehrt und deren Performance sehr negativ beeinflusst. Aus der Sicht der semantischen Datenmodellierung ist jedoch genau der gewollte Status erreicht, denn die Beschreibung der Daten ist nun Teil der Daten geworden.

Ein semantisches Datenmodell lässt sich zwar auch in einer relationalen Datenbank anlegen, dies bringt jedoch einige Einschränkungen, unter anderem bezüglich der Geschwindigkeit und Zugreifbarkeit, mit sich, da dem Modell eine gänzlich andere Art der Modellierung zugrunde liegt.

meal	field	value
Pasta Genovese	portionen morgens	2,3
Pasta Genovese	portionen mittags	10,3
Pasta Genovese	portionen abends	7,8
Pizza Salami	portionen morgens	1,0
Pizza Salami	portionen mittags	11,1
Pizza Salami	portionen abends	8,9
Tomatensuppe	portionen morgens	4,8
Tomatensuppe	portionen mittags	5,2
Tomatensuppe	portionen abends	10,8

Abbildung 3.7.: Denormalisiertes Schema

3.3. Sprachen des Semantic Web

3.3.1. Resource Description Framework

Das Resource Description Framework [27], kurz RDF, geht zurück auf die im Jahr 2001 gestartete Semantic Web Initiative des W3C. RDF ist keine Modellierungsprache im klassischen Sinne, sondern vielmehr ein konzeptionelles Werkzeug, um beliebige Ressourcen zu beschreiben und untereinander in Relation zu setzen.

Ein RDF-Modell setzt sich aus mehreren sogenannten Fakten zusammen. Ein Fakt wird in der Regel auch als Tripel bezeichnet und setzt sich aus einem Subjekt, einem Prädikat und einem Objekt zusammen. Ein einzelnes Tripel kann auch als unvollständiger, gerichteter Graph aufgefasst werden, das Subjekt und Objekt werden als Knoten interpretiert und das Prädikat als gerichtete Kante zwischen Subjekt und Objekt. Der Gedanke des Graphen kommt jedoch umso besser zur Geltung, je mehr Tripel existieren, deren Elemente mit denen anderer Tripel verknüpft werden.

RDF/XML

Die bekannteste und auch vom W3C als Empfehlung verabschiedete Serialisierung von RDF verwendet XML. Häufig wird deshalb auch irrtümlicher Weise RDF mit RDF/XML gleichgesetzt, obwohl sie differente Aspekte bezeichnen. XML, also ein baum-orientiertes Format, zur Serialisierung von RDF, einem graph-orientierten Format, einzusetzen, erscheint auf den ersten Blick vielleicht ungünstig, die Vorteile (Sprachunabhägigkeit), die XML als etabliertes Austauschformat bietet, wiegen dies aber auf.

```
1 <rdf:Description rdf:about="#76619000">
2     <hatLichtfarbe rdf:resource="#RED"/>
3     <rdf:type rdf:resource="#Leuchte"/>
4 </rdf:Description>
5
6 <Leuchte rdf:about="#76619000">
7     <hatLichtfarbe rdf:resource="#RED"/>
8 </Leuchte>
```

Listing 3.1: Beispiel einer RDF/XML Serialisierung

Da durch RDF-Graphen nur selten Baumstrukturen erzeugt werden, haben RDF/XML Dokumente in der Regel nur eine sehr geringe Hierarchie, Ausnahmen sind hier vor allem Schemadefinitionen in Verbindung mit Klassenrestriktionen. Das Kernelement von RDF/XML ist `rdf:Description`, welches die im `about`-Attribut angegeben Ressource beschreibt. In Listing 3.1, Zeile 1-4 wird exemplarisch die Leuchte `#7661900` definiert; die Lichtfarbe ist rot und die Ressource gehört zur Klasse „Leuchte“. Um die Lesbarkeit zu erhöhen, lassen sich oft verwendete Konstrukte wie das Zuweisen zu Klassen verkürzen, indem anstelle von `rdf:Description` der gewünschte Klassenname verwendet wird (3.1, Zeile 6-8), dies impliziert das Statement aus der dritten Zeile. Ein vollständiger Überblick auf RDF/XML findet sich in der Sprachspezifikation des W3C [28].

N3 & N–Triple

Der Grundgedanke hinter N3 [29] besteht in der Etablierung einer Alternative zu RDF/XML, die wesentlich kompakter und einfacher zu lesen ist. Damit ist N3 in erster Linie eine Serialisierung, die von Menschen sehr viel schneller rezipiert werden kann als RDF/XML, aber auch als Austauschformat konnte sich N3 behaupten, da der Overhead in Vergleich zu RDF/XML minimal ausfällt.

Shorthand	stands for
a	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
=	<http://www.w3.org/2002/07/owl#sameAs>
=>	<http://www.w3.org/2000/10/swap/log#implies>
<=	<http://www.w3.org/2000/10/swap/log#implies> but in the inverse direction

Abbildung 3.8.: Übersicht einiger Verkürzungen von N3 [29]

```
1 # N3
2 ex:76619000 ex:hatLichtfarbe ex:RED;
3             a ex:Leuchte.
4
5 # N-Triple
6 ex:76619000 ex:hatLichtfarbe ex:RED.
7 ex:76619000 rdf:type ex:Leuchte.
```

Listing 3.2: Beispiel einer N3 und N-Triple Serialisierung

Zur weiteren Erhöhung der Lesbarkeit von N3-Dokumenten wurden so genannte *Shorthands* (Abbildung 3.8) eingeführt, die besonders häufig verwendete Axiome durch *sprechende* Namen abkürzen.

N–Triple [30] ist eine Untermenge von N3. Sämtliche Verkürzungen und Vereinfachungen von N3 wurden gestrichen, so dass eine N–Triple Datei lediglich sämtliche Triple zeilenweise auflistet. In erster Linie werden N–Triple Serialierungen als Testfälle für RDF-Parser verwendet, daher auch die enorme Reduktion der Komplexität.

In Listing 3.2 ist derselbe Graph serialisiert, dem auch Listing 3.1 zugrunde liegt. Die Zeile 5-7 zeigen die Serialisierung im N–Triple Format, im direkten Vergleich lässt sich die unterschiedliche Komplexität gut erkennen.

RDFa

RDFa [31] nimmt eine gewisse Sonderrolle unter den Serialierungsformaten ein, denn im Gegensatz zu den anderen Formaten entsteht bei RDFa keine eigenständige Datei. Anstelle dessen werden die RDF-Tripel innerhalb einer (X)HTML-Datei ausgezeichnet. Mit Hilfe von GRDDL[2] und XSLT[3] oder sprachspezifischen Implementierungen[4] lassen sich die Daten dann aus dem HTML-Gerüst extrahieren und weiterverarbeiten. Um nach wie vor ein valides XHTML-Dokument zu erhalten, muss mit `XHTML 1.1 + RDFa` ein besonderer Dokumententyp im Kopf angegeben werden. Die Verwendung von RDFa eignet sich insbesondere dafür, um dem Browser zusätzliche Meta-Informationen mitzuteilen. So können beispielsweise Kontaktdaten oder Eventinformationen vom Browser interpretiert und dem Benutzer gesondert dargestellt und zur

[2] http://www.w3.org/2004/01/rdxh/spec
[3] http://www.w3.org/TR/xslt
[4] http://rdfa.info/rdfa-implementations/

Interaktion bereitgestellt werden. Grundsätzlich lässt sich RDFa mit Mikroformaten[5] [32, 33] vergleichen, nur stellt RDFa nicht die Vokabulare, sondern das Werkzeug, um diese mit HTML zu verknüpfen.

3.3.2. Vokabulare & Ontologien

Zwar stellt RDF ein geeignetes Werkzeug zur Verfügung, um beliebige Ressourcen zu beschreiben, allerdings wird durch den offenen Charakter von RDF zunächst einmal der „Wildwuchs“ an Daten gefördert. Dem Semantic Web liegt zwar wie dem „klassischen“ Web die Maxime zugrunde, dass Jeder jederzeit alles sagen kann [8], das Semantic Web beruht allerdings auf der Tatsache, dass Daten aus der gleiche Domäne mit den gleichen Worten beschrieben werden. Aus dieser Not heraus haben sich bereits für die unterschiedlichsten Domänen Vokabulare entwickelt, die von Datenprovidern genutzt werden sollten, wenn sie ihre Daten beschreiben. Ein Vokabular ist hierbei eine Sammlung von Konzepten und Eigenschaften einer Domäne, um deren Ressourcen und Beziehungen untereinander zu beschreiben [34]. Aus dieser Definition lässt sich ebenso herleiten, dass Vokabulare im Kontext der semantischen Modellierung auch als Ontologien interpretiert werden können. Nach Gruber [35] ist eine Ontologie eine explizite formale Spezifikation einer (gemeinsamen) Konzeptualisierung.

RDF Schema

RDF Schema [36] (RDFS) wurde 2004 als W3C Recommendation verabschiedet und ist die erste Sprache zu Definition von Vokabularen im Kontext von RDF. Korrekterweise muss RDFS nicht als Definitionssprache deklariert werden, sondern ebenfalls als Vokabular. Denn bei RDFS handelt es sich um ein ein RDF definiertes Vokabular zur Definition von weiteren Vokabularen.

Zu den wichtigsten Konstrukten, die in RDFS definiert wurden, zählt vor allem das Konzept von Klassen `rdfs:Class` und Klassenhierarchien (`rdfs:subClassOf`). Somit lassen sich erstmals Ressourcen zuvor definierten Konzepten zuordnen. Mit dem ebenfalls neu eingeführten dedizierten Konzept `rdfs:Property` wird es möglich, Ressourcen bzw. Prädikate explizit als solche zu definieren und

[5] http://microformats.org

mit zusätzlichen weiteren Informationen wie `rdfs:domain` und `rdfs:range` die Subjekt- bzw. Objektseite des Prädikats zu charakterisieren.

Daneben wurden mit RDFS viele weitere Konstrukte eingeführt, die zur Dokumentation genutzt werden können, z.B. `rdfs:label`, `rdfs:comment` usw. aber auch Konstrukte, um Ressourcen mit anderen Datenquellen in Verbindung zu setzen. Elemente wie Container, Listen oder Aufzählungen, welche die Strukturierung von Inhalten vereinfachen sollten, sind zwar in RDFS enthalten, im Kontext von *Linking Open Data* (LOD) wird von deren Benutzung allerdings abgeraten [34], da in den meisten Fällen diese Konstrukte sich nicht korrekt mit Sparql abfragen lassen und häufig eine mehrfache Anwendung des gleichen Prädikats ausreichend ist, um die gleiche Aussagen treffen zu können.

OWL – Web Ontology Language

OWL 2 [37, 38] ist wie RDFS eine Vokabular- bzw. Ontologiesprache, die in der zweiten Version (OWL 2) 2009 vom W3C als Empfehlung verabschiedet wurde. Die Ursprungsempfehlung von OWL [39] aus dem Jahre 2004 wird durch die neue Empfehlung jedoch nicht ersetzt, sondern erweitert diese an einigen Punkten.

Wie RDFS ist OWL in RDF formuliert und erweitert den Sprachraum von RDF Schema um neue Konstrukte. Die Beziehung zwischen beiden Sprachen wird in der Regel über Vererbungen hergestellt, so gilt beispielsweise `owl:Class rdfs:subClassOf rdfs:Class`. Der wesentliche Unterschied zwischen beiden Sprachen ist die Mächtigkeit und Ausdrucksstärke. Mit Hilfe von OWL lassen sich sehr viel diffizilere Ontologien erstellen als es mit RDFS der Fall ist. Hinzu kommt die erweiterte Entscheidbarkeit von OWL in Gegensatz zu RDFS, die es ermöglicht, mit Hilfe eines Reasoners neues implizites Wissen in der Ontologie zu erschließen. Wie schon bei OWL beschreibt der OWL 2 Standard nicht eine Sprache, sondern mehrere Unterklassen, die sich in Ausdrucksstärke und Entscheidbarkeit unterscheiden.

OWL 2 EL/RL/QL Das Ziel der Sprachklasse OWL Lite, einen einfach und effizient zu implementierenden Teil von OWL darzustellen, wurde komplett verfehlt [40]. Zum einen wurde OWL Lite ähnlich komplex wie OWL DL definiert und durch die komplizierte Syntax war es kaum möglich, seine Stärken korrekt auszuspielen. Viele OWL Lite Ontologien entsprechen nicht geplant, sondern eher zufällig, diesem Standard. In OWL 2

wurde darum die OWL Lite Sprachklasse durch mehrere einfache Sprachprofile [41] ersetzt, die jeweils unterschiedliche Anwendungskontexte unterstützen sollen. OWL 2 EL eignet sich für Ontologien, in denen große Mengen Klassen und Eigenschaften modelliert werden müssen. OWL 2 RL hingegen sollte für Anwendungsfälle eingesetzt werden, in denen viele Instanzen erzeugt werden.

OWL 2 DL ist die Weiterentwicklung von OWL DL. Diese Sprachspezifikation ist äquivalent zur Prädikatenlogik erster Ordnung und damit voll endscheidbar. Anders als OWL DL basiert OWL 2 DL nicht mehr auf der Bescheibungslogik $SHOIN$, sondern auf $SROIQ$. Durch diesen Wechsel werden eine Reihe von neuen logischen Konstruktionen ermöglicht, wie z.B. qualifizierte Zahlenrestriktionen, Rollenaxiome und erweiterte Negationsmöglichkeiten. Neben den Veränderungen der logischen Grundlagen zählt zu den wesentlichen Neuerungen die Erweiterung der Metamodellierungsmöglichkeiten. OWL 2 DL bietet Unterstützung für die einfachste Form der Metamodellierung, das so genannte *Punning*[6]. Das bedeutet, dass die Namen von Klassen, Rollen und Instanzen nicht disjukt sein müssen, gleichzeitig aber keine logische Beziehung zwischen Ressourcen mit gleichlautenden Namen aber unterschiedlichen Typs besteht. So lassen sich z.B. Klassen auch als Objekte einer Instanzrelation verwenden.

OWL 2 Full stellt die Weiterentwicklung von OWL Full dar. Insbesondere im Bereich Metamodellierung wurden der Sprachspezifikation einige neue OWL2-Konstrukte hinzugefügt. Nach wie vor ist OWL 2 Full nicht endscheidbar, d.h. die logische Konsistenz der Spezifikation lässt sich aufgrund der Ausdrucksstärke nicht beweisen. Das Einsatzgebiet von OWL 2 Full ist momentan eher auf konzeptionelle Modellierungsarbeiten beschränkt, da bislang noch keine Inferenzmaschinen existieren, die automatische Ableitungen ermöglichen. Durch die Veränderung der Sprachspezifikationen insbesondere durch die Einführung von Punning, können viele bestehende OWL Full Ontologien nun auch als OWL 2 DL interpretiert werden.

[6] http://www.w3.org/2007/OWL/wiki/Punning

3.4. Anwendungsszenarien

3.4.1. Semantische Suche

Das Suchen oder vielmehr das Finden von Informationen wird zu einer zunehmend komplexeren Tätigkeit. Die Informationsmenge im Web wächst jeden Tag weiter und gerade sehr spezifische und hochintegrierte Fragestellungen lassen sich mit den heutigen Suchmaschinen teils nur schwer beantworten. Eine mögliche Lösung oder zumindest ein Mittel zur Hilfestellung bei der Informationssuche sind semantische Suchmaschinen.

Neben den Vorteilen, die klassische Suchmaschinen bieten, nämlich das Durchsuchen von enormen Informationsmengen, wird das Auffinden von Informationen durch weitere Funktionen unterstützt:

Sprachunabhängigkeit Da die Bedeutung von Informationen explizit modelliert wird, sind Suchanfragen an semantische Suchmaschinen nicht auf die Sprache der Anfrage beschränkt. Es ist ebenso möglich, Informationen in die Ergebnismenge zu integrieren, die in einer anderen Sprache vorliegen. Im Idealfall geht diese Integration sogar noch einen Schritt weiter und es wird vor der Präsentation eine Übersetzung in die Ausgangssprache vorgenommen.

Fuzzy-Search Entgegen der reinen Stichwort-Suche, die in traditionellen Suchverfahren einsetzt wird, erlauben semantische Suchen auch so genannte *unscharfe* Suchen in Informationsbeständen. Hierbei wird nicht nur der eingegebene Begriff verwendet um Informationen zu finden, sondern auch z.b. Synonyme des Suchbegriffs. Mit Hilfe von Thesauren lassen sich zudem Begriffe genauer charakterisieren und so kann der Suchbegriff entweder weiter spezifiziert (narrow terms) oder expandiert (broader terms) werden.

Kontext-Sensivität Semantische Suchtechnik erlaubt es außerdem, aus den gewonnenen Ergebnissen den Suchkontext zu extrahieren und diesen zur Präzisierung der gefundenen Informationen einzusetzen. Wenn z.B. nach dem Begriff „Jaguar“ im Kontext von Tieren gesucht wurde, dann sind Ergebnisse im Kontext Autos oder Betriebssysteme zu vernachlässigen.

Ähnlichkeits-Erkennung Ähnlich wie bei der *unscharfen* Suche können semantische Suchmaschinen die gefundenen Dokumente untersuchen und

auf Basis der identifizierten Daten Dokumente suchen, die dem bereits gefundenen ähnlich sind. Hierdurch wird eine fast vollständige Abstraktion des eingegebenen Suchbegriffes erreicht und auch Informationen gefunden, die zwar relevant, aber mit dem ursprünglichen Suchbegriff nicht explizit verbunden sind.

3.4.2. Linking Open Data

Ein weiteres Anwendungsszenario des semantischen Web ist das Aggregieren, Verarbeiten und Darstellen von verteilten Datenbeständen. Dies kommt der Vision vom Tim Berners-Lee sehr nahe. Denn statt eine Anwendung monolithisch auf einen Datenbestand zu gründen, liegt der Linking Open Data (LOD) Initiative die Idee zugrunde, bestehende Daten mit den eigenen Daten zu verknüpfen und von den so entstehenden Synergien zu profitieren.

Das Linking Open Data Web besteht zur Zeit[7] aus etwa 4,7 Milliarden Tripeln [4, 42], die sich u.a. aus den Gebieten Life Science, Publication, Media, Geoinformation zusammensetzen.

Interoperabilität Mit Hilfe des Linking Open Data Webs lassen sich in kürzester Zeit Anwendungen konzipieren, die unterschiedlichste Datenbestände miteinander verbinden. Anstatt, wie bisher, auf proprietäre APIs [43] angewiesen zu sein, um auf die Daten von anderen Instanzen zugreifen zu können, können nun Daten auf standardisierte Art und Weise zwischen Anwendungen ausgetauscht werden. Zudem sind die Daten selbstbeschreibend, da Schema und Daten zusammen kodiert werden. So können auch völlig unbekannte Datenbestände integriert werden.

Inhaltliche Bereicherung Auf Basis von *Linking Open Data* ist es möglich, Informationen mit weiterem Wissen zu verknüpfen und so die eigenen Daten mit neuen Informationen anzureichern. Dies können z.B. andere Dokumente sein, die sich mit einen ähnlichen Thema beschäftigen, oder Informationen, die Aspekte aus einem Dokument vertieft darstellen.

Komplementäre Sichtweisen Anstatt das Wissen einer Anwendung aus einer monolithischen Quelle zu beziehen, erlaubt das Data Web, auf einfache Weise multiple Quellen anzubinden. Durch die Kopplung unterschiedlicher Wissenensrepräsentationen wird eine vielschichtigere Sicht

[7] März 2009

auf unterschiedliche Fakten ermöglicht. Ganz grundsätzlich lassen sich somit Konstrukte erzeugen, die sich mit RDF Refications vergleichen lassen, denn von der Verwendung von Refications [34] wird aufgrund von Schwierigkeiten in Verbindung mit SPARQl abgeraten.

3.4.3. Semantische Widgets

Als eine Mischung aus semantischer Suche und *LOD* können so genannte „semantische Widgets“ verstanden werden. Dies sind kleine HTML-Elemente, die sich in Webseiten integrieren lassen. Die Anzahl an Variationen von Widgets und die daraus resultierenden Möglichkeiten sind vielfältig und gehen von der Anzeige von weiteren, zum betrachteten Dokument, relevanten Informationen (semantische Suche) über das Nutzen externer Datenquellen bis zum Veröffentlichen der eigenen Daten (Linking Open Data).

Der wesentliche Vorteil von semantischen Widgets ist das Partizipieren am Semantic Web, ohne selbst große programmatische Schritte unternehmen zu müssen. Die eigene Arbeit beschränkt sich auf das einmalige Einstellen von ein paar grundlegenden Parametern und dem Einbinden des entsprechenden Widget-Codes. Das vielmals von Kritikern hervorgebrachte Argument, das *Semantic Web* könne sich nicht durchsetzen, da die technische Hürde, aufgrund der vielen begleitenden Techniken, zu hoch sei, um das Semantic Web für jeden begreifbar und zugänglich zu machen, ist mindestens seit dem Aufkommen von semantischen Widgets hinfällig.

3.4.4. Semantische Werbung

Auch im Bereich der Online-Werbung lassen sich semantische Techniken einsetzen, um einen Vorteil klassischen Verfahren gegenüber zu erhalten. Aufgrund der expliziten Auszeichnung der Semantik lassen sich genauere Vorschläge generieren, welche Werbung zu dem aktuell betrachteten Kontext passen könnte.

Abbildung 3.9[8] zeigt die Unterschiede zwischen traditionellen Finden von Treffern auf Basis von Stichworten und dem Finden von angepassteren Ergebnissen mit Hilfe der *Semantischen Signatur*. Bei der „traditionellen“ Suche werden lediglich die Stichwörter mit einer Datenbank verglichen und die Ergebnisse

[8] http://www.textwise.com

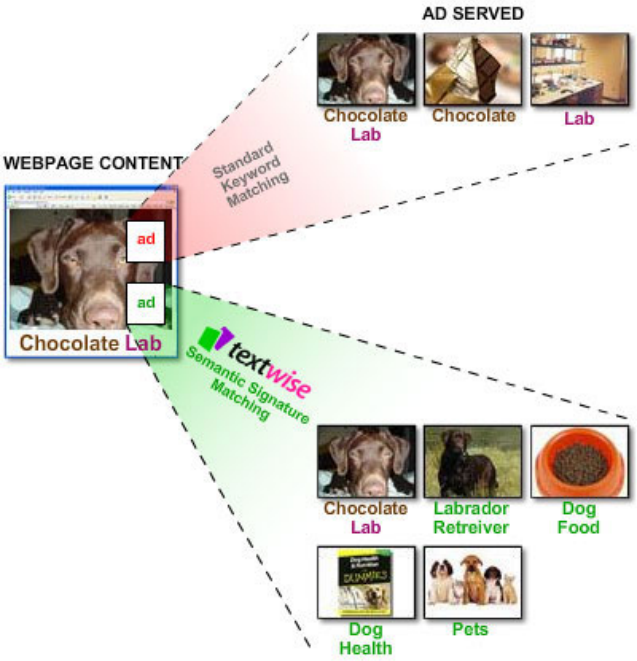

Abbildung 3.9.: Einsatz von Semantik für kontextorientierte Werbung

angezeigt, hat der betrachtete Begriff jedoch mehrere Bedeutungen, so können diese nicht differenziert werden und es kommt auch Werbung zur Anzeige, die nicht zu dem ursprünglichen Material passt. Wird semantische Technik für die Bereitstellung von Werbung verwendet, kann neben dem Schlagwort auch der zugehörige Kontext untersucht werden. Kommt es nun zu einer Mehrdeutigkeit, lassen sich diese über den Kontext auflösen. Zusätzlich können dann auch verwandte Begriffe zur Auswahl geeigneter Werbung herangezogen werden, da die genaue Semantik bekannt ist.

4. Formales Modell der Integration

Es existieren bereits Ansätze, die sich mit der Integration semantischer Modelle in Programmiersprachen beschäftigen. Quasthoff et al. entwickeln in [44] ein Vorgehensmuster, das sehr stark auf die Verwendung von Java Annotations fixiert ist. Die Arbeit von Bartalos und Bielikova [45] verfolgt einen generischeren Ansatz, vernachlässigt aber die Verteiltheit von Datenmodellen. In [46] und [47] werden vielversprechende Ideen aufgezeigt, die zum Teil in Kapitel 5 aufgegriffen wurden. Alle Ansätze betrachten jedoch immer nur einen kleinen Teil des Problemraums, so dass in der Regel Insellösungen beschrieben werden, die in der Praxis auf Probleme stoßen.

4.1. Ontologie – Objekt - Mapping

Oren [48] liefert eine Gegenüberstellung von Objekt-orientierten Sprachen und RDF Schema in Hinsicht auf die Unterschiede bezüglich Klassen- und Objektbegriff. Die folgenden Abschnitte widmen sich den einzelnen Aspekten im Detail.

4.1.1. Klassenzugehörigkeit

Die Beziehung zwischen Objekt und Klasse ist in jeder Programmiersprache eine *n* zu *1* Beziehung. Es können zwar mehrere Instanzen einer Klasse existieren, allerdings kann jede Instanz nur zu einer einzigen Klasse gehören. Dem zugrunde liegt das Template-Verhalten einer Klasse. Eine Klasse definiert eine Vorlage für ein Objekt, eine Instanz wird nun aus dieser Vorlage heraus „gestanzt“. Eine Instanz im semantischen Modell ist aber per se noch nicht zwangsläufig an eine Klasse gebunden und kann auch Instanz mehrerer Klas-

capability	object-oriented languages	RDF Schema
class membership	each object is member of exactly one class	resources can have multiple types
class inheritance	classes can only inherit behaviour from at most one superclass	classes can inherit from multiple superclasses
object conformance	instance structure must strictly adhere to class	class denitions are not exhaustive and do not constrain the structure of their instances
semi-structured data	class denitions are strict, prescribe all instance properties and are required for all instances	instances might deviate from the class denition or appear without any schema information
runtime evolution	class denitions typically cannot evolve during runtime	data is integrated from heterogeneous sources with varying structures where both schema and data may evolve at runtime

Tabelle 4.1.: Unterschiede zwischen Objektorientierung und semantischer Modellierung

sen werden. Das Paradigma der „Vorlage“ passt in diesem Kontext somit nicht mehr. Zur Offenlegung der Divergenz zwischen Klassen im Objekt-orientierten und im semantischen Modell wird in [47] der Begriff des *intensional sets* geprägt, wenn es um die Integration von semantischen Modellen in objektorientierte Sprachen geht.

4.1.2. Klassenvererbung

Entgegen der verallgemeinerten Aussage von Oren [48] (siehe Tabelle 4.1) existieren eine ganze Reihe von Programmiersprachen, die *multiple inheritance* unterstützen, bei den dynamischen Scriptsprachen z.B. Python oder Perl und C++ bei den statischen Programmiersprachen, um eine kleine Auswahl zu nennen. Ungeachtet dessen muss zunächst betrachtet werden, ob das Fehlen von *multiple inheritance* überhaupt ein Problem bei der Integration in objektorientierte Sprachen darstellt.

In einem semantischen Modell werden Konzepte, Instanzen und Eigenschaften, sowie die Relationen zwischen diesen Ressourcen modelliert. Das Modell bewegt sich vollständig auf einer deskriptiven Ebene, d.h. es wird kein Verhal-

ten der Ressourcen modelliert. Dies ist auch ein wesentlicher Unterschied von Instanzen im objektorientierten, die sich über *Verhalten*, *Zustand* und *Identität* charakterisieren lassen, wohingegen eine Instanz im semantischen Modell lediglich durch *Zustand* und *Identität* charakterisiert wird.

Unter der Annahme, dass das semantische Modell in einer anfragbaren Form[1] zur Verfügung steht und damit unter Beachtung des *Semantic Web Towers* die Inferenz-Bildung im Modell bereits erfolgt ist, relativiert sich die aus der *multiple inheritance* entstehende Problematik. Die Integration des semantischen Modells heißt im Kontext dieser Arbeit: „Objektorientierter Zugriff auf Ressourcen im Sinne von RDF“, daraus folgt, dass die Instanzen einer programmatischen Klasse die Instanzen einer semantischen Klasse repräsentieren. Die Vererbung, bzw. genauer das *Reasoning*, ist zu diesem Zeitpunkt schon durchgeführt. Im Gegensatz zur objektorientierten Vererbung, wo dieser Prozess Teil der Laufzeitumgebung ist.

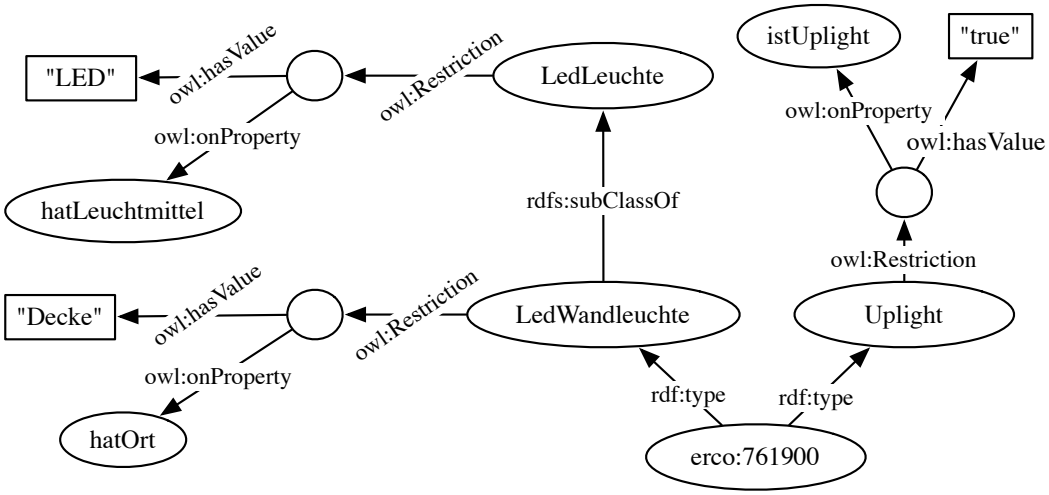

Abbildung 4.1.: Vereinfachtes semantisches Modell

Abbildung 4.1 zeigt ein einfaches semantisches Modell, in dem drei Klassen definiert wurden, die jeweils eine Eigenschaft an ihre Instanzen vererben. Im Falle von *LedLeuchte* ist dies *hatLeuchtmittel* = *„LED“*, für *LedWandleuchte* *hatOrt* = Wand und *Uplight istUplight* = *„true“*. Die Instanz *7619000* wird über *rdf:type* nur mit den beiden Klassen *Uplight* und *LedWandleuchte* verbunden. In Abbildung 4.2 werden die Eigenschaften der Instanz dargestellt, nachdem das *Reasoning* durchgeführt wurde. Die Integration der Ressourcen in das objektorientierte Schema findet dann auf Basis dieser angereicherten

[1] In der Regel ist dies ein *triple-store*, welcher mit Hilfe von *SPARQL* abgefragt werden kann. Die Implementierung ist an dieser Stelle letztendlich jedoch zweitrangig.

Ressourcen statt. Die Vererbung von Eigenschaften findet nicht auf Ebene der Programmiersprache statt, um eine Duplizierung der Funktionalität des *Reasoners* zu vermeiden.

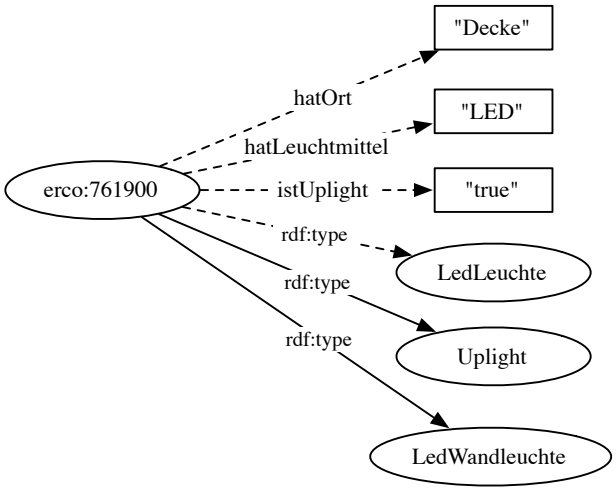

Abbildung 4.2.: Eigenschaften der Ressource *7619000* nach Abschluss des *Reasoning*, gestrichelte Attribute beschreiben *inferred*, also durch Reasoning hinzugefügte, Statements und durchgehende *asserted* (explizit formulierte) Statements (siehe auch Abbildung 4.1)

Zwar greift die vorgelagerte Vererbung für Instanzen, die bereits zur Laufzeit existieren, nicht jedoch bei Instanzen, die erst während der Laufzeit erzeugt werden. Bei einem hochdynamischen System, das transparent auf den Daten im *triple-store* arbeitet, werden die Eigenschaften in dem Moment für die Ressource sichtbar, wenn das Tripel, das die Typzuordnung über *rdf:type* beschreibt, in den Datenspeicher geschrieben wird. Die Instanz ist in diesem Fall direkt mit dem semantischen Modell verbunden und immer auf dem aktuellsten Stand. Es besteht also auch hier nicht der Bedarf, die Vererbung im Kontext der Programmiersprache durchzuführen.

In Systemen die, z.B. aus Gründen der Transaktionssicherheit, nicht direkt jede Änderung auf dem *triple-store* durchführen, muss somit gewährleistet sein, dass neue Objekte mit den Eigenschaften „versorgt“ werden, die ihnen aufgrund ihres *Typs* zur Verfügung stehen.

Sollen die Klassen des semantischen Modells zusätzlich programmatisch noch um *Verhalten* erweitert werden, spielt das Thema Vererbung jedoch wieder eine Rolle, da der Fokus nun in der Programmiersprache liegt und diese somit auch für die Vererbung von Verhalten zuständig ist. Wird multiple Verer-

bung von der Programmiersprache allerdings nicht unterstützt, kann ein solches Verhalten z.B. mittels *Metaprogramming* implementiert werden. Hierbei muss trotzdem drauf geachtet werden, so wenig Funktionalität des *Reasoners* nachzubilden wie möglich.

4.1.3. Objektkonformität

Wie schon in den vorherigen Abschnitten verdeutlicht, wird der Begriff der Klasse zwar ambivalent genutzt, hat aber semantisch eine vollkommen andere Bedeutung. Im objektorientierten Kontext steht eine Klasse für einen Bauplan einer Reihe von ähnlichen Objekten.

> „In object-oriented programming, a class is a construct that is used as a blueprint (or template) to create objects of that class.“ [49]

Dem gegenüber wird die Klasse im semantischen Modell einfach als „Menge“ mehrerer Instanzen beschrieben.

> „Associated with each class is a set, called the class extension of the class, which is the set of the instances of the class.“ [36]

Obwohl der „Mengen“-Gedanke auch auf OO-Klassen zutrifft, resultieren weitergehende Konsequenzen aus der unterschiedlichen Definition was eine Klasse ausmacht.

Im Objektorientierten können keine Objekte existieren, die nicht Mitglied irgendeiner Klasse sind und ein Objekt gehört immer zu nur einer Klasse[2]. Da die Klasse den Bauplan für Objekte bereitstellt, bestimmt die Klasse die Charakteristik der Objekte.

Die Klasse kann also als Schablone gesehen werden, mit der die Instanzen heraus „gestanzt“ werden. In Ontologien besteht diese strikte Bindung zwischen Instanzen und Klasse nicht. Anders als im Objektorientierten wird hier eine Instanz mit einer oder mehreren Klassen verbunden. Es können sogar Ressourcen existieren, die keiner expliziten[3] Klasse zugeordnet sind. Anders als im Objektorientierten gibt die Klasse also nicht die Struktur ihrer Mitglieder vor, daraus

[2] Es gibt auch Programmiersprachen, in denen Objekte Instanzen mehrerer Klassen sein können, aber Instanzen gehören immer nur zu einer Klasse.

[3] Im Kontext von OWL existiert die Klasse OWL:Thing, die Klasse aller Objekte. Das Reasoning fügt alle Instanzen einer Ontologie dieser *Über*-Klasse hinzu, auch wenn es kein Statement gibt, was eine Instanz irgendeiner Klasse zuordnet. In RDF-Schema ist keine solche Oberklasse definiert.

folgt im Umkehrschluss, dass allein aufgrund der Klassenzugehörigkeit nicht auf die Eigenschaften der Instanz geschlossen werden kann. Abbildung 4.3 zeigt noch einmal die unterschiedliche Interpretation.

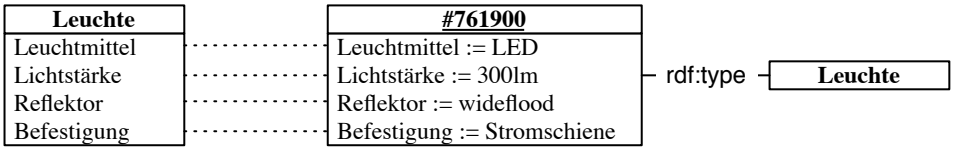

Abbildung 4.3.: Während im OO die Klasse einer Objekts dessen Struktur vorgibt, d.h. deren Eigenschaften vorgibt, besteht im semantischen kein direkter Zusammenhang zwischen den Eigenschaften der Instanz und der Klassenzugehörigkeit. Die Ressource im semantischen Modell ist eigenständig und die Eigenschaften werden direkt an die jeweilige Ressource gebunden.

Die Aussage, dass die Klasse einer Instanz, deren Struktur nicht bestimmt ist jedoch nicht allgemeingültig für alle Anwendungsfälle im Bereich der semantischen Modellierung. Tatsächlich besteht die Möglichkeit, auch Instanzen durch ihre Klassenzugehörigkeit zu beeinflussen. Abbildung 4.1 zeigt eine einfache Ontologie mit erweiterten Klassendefinitionen. Durch die Verwendung von *Restrictions* lassen sich weitere Aussagen über eine Klasse treffen, die auch ihre Instanzen beeinflussen. Im Beispiel aus Abb. 4.1 werden allen Instanzen der Klasse LedLeuchte die Eigenschaften *hatLeuchtmittel* mit dem Wert *LED* zugeordnet, ähnliches gilt auch für die weiteren Klassen-Definitionen. Allerdings unterscheidet sich dieses Verhalten immer noch von dem im Objektorientierten. Dort wird nur die Struktur der Instanzen vorgegeben, also welche Eigenschaften eine Instanz besitzt. Das hier dargestellte Verhalten geht weiter und legt die Eigenschaft und den Wert der Eigenschaft für eine Instanz fest, dies kann am ehesten mit dem Konzept der Klassenkonstanten verglichen werden. Zudem sind solche Modellierungskonstrukte nur in OWL-Ontologien verfügbar und bedürfen der Inferenz-Bildung mit einem Reasoner bevor sie *sichtbar* werden.

4.1.4. Semistrukturierte Daten

Dieser Aspekt steht sehr nah im Zusammenhang mit dem zuvor Beschriebenen. Die Ressourcen im semantischen Modell folgen keinem strukturierten Schema. Zwar ist es möglich, Ressourcen auf Basis von strukturiertem Wissen zu beschreiben, aber die eine Ressource an sich folgt lediglich dem Beschreibungsschema: *Subjekt*, *Prädikat* und *Objekt*.

Es existieren jedoch zwei Modellierungskonstrukte, die irrtümlich dafür gehalten werden können, dieses Verhalten zu erzeugen. In [8] werden einige typische Modellierungsfehler vorgestellt, die bevorzugt von Entwicklern begangen werden, die aus dem objektorientierten Umfeld stammen. Mit Hilfe der Konstrukte *domain* und *range* lassen sich das *Subjekt* und das *Objekt* eines *Prädikats* typisieren. Bei falscher Interpretation von Listing 4.1 könnte angenommen werden, dass sich die Instanzen der Klasse *MonochromeLeuchte* durch das Prädikat *hatLichtfarbe* beschreiben lassen.

```
1  <owl:Class rdf:about="#MonochromeLeuchte"/>
2
3  <owl:ObjectProperty rdf:about="#hatLichtfarbe">
4      <rdfs:range rdf:resource="#Lichtfarbe"/>
5      <rdfs:domain rdf:resource="#MonochromeLeuchte"/>
6  </owl:ObjectProperty>
7
8  <rdf:Description rdf:about="#76619000">
9      <hatLichtfarbe rdf:resource="#RED"/>
10 </rdf:Description>
11
12 <MonochromeLeuchte rdf:about="#87892000"/>
```

Listing 4.1: Wissen aus der Domäne Leuchten

Die Semantik von *domain* und *range* dient jedoch nicht dazu die Instanzen einer Klasse mit Constraints zu belegen. Im Reasoning-Prozess werden diese Attribute nur dazu genutzt Typzuordnungen vorzunehmen. Zeile 8-10 in Listing 4.1 beschreibt eine Ressource, die das Prädikat *hatLichtfarbe* verwendet. In Kombination mit der Prädikatdeklaration (Zeile 3-6) wird hieraus abgeleitet, dass *7661900* vom Typ *MonochromeLeuchte* ist. Im Gegenzug dazu wird in Zeile 12 eine Leuchte beschrieben, die explizit als monochrome Leuchte ausgewiesen wird. Die Definition von *hatLichtfarbe* hat auf dieses Statement keinen Einfluss. Obwohl *8789000* kein Attribut *hatLichtfarbe* verwendet, ist diese Ressource konsistent vom Typ *MonochromeLeuchte*.

4.1.5. Laufzeitevolution

Aufgrund der verteilten Architektur eines semantischen Modells ergeben sich vollkommen andere Voraussetzungen als bei der Verwendung von „gewöhnlichen“ Klassen. Die Definition einer Klasse verändert sich im objektorientierten in der Regel nicht zur Laufzeit einer Anwendung. Im dem Fall, in dem die Klassen-

definitionen jedoch auf einem semantischen Modell realisiert werden, können solche Änderungen auftreten.

Die Vision des Semantic Web propagiert die Wiederverwendung von Wissen und den Einsatz von dezentralen Wissensmodellierungen. Damit einher geht jedoch auch die Notwenigkeit Änderungen an externen Datenbeständen wahrzunehmen und entsprechend zu reagieren.

4.1.6. Open World Assumption

Neben den Differenzen zwischen Objektorientierung und semantischer Modellierung, die in den vorangegangenen Abschnitten betrachtet wurden, bedarf es noch weiterer Aspekte, die von Oren[48] noch nicht benannt wurden. Während dem objektorientierten ein geschlossenes Weltbild zugrunde liegt, handelt es sich im semantischen Modell um ein offenes Weltbild. Diese Eigenschaft wird unter anderem *Open-* oder *Closed World Assumption* genannt und spielt eine bedeutende Rolle, insbesondere für die Inferenzbildung in der semantischen Modellierung.

```
1 <ex:Person> a <owl:Class>.
2 <ex:Mortal> rdfs:subClassOf <ex:Person>.
3
4 <ex:Plato> a <ex:Person>.
5 <ex:Sokrates> a <ex:Mortal>.
```

Listing 4.2: Einfaches Modell von Plato; Sokrates und der Sterblichkeit

Die wahr/falsch Dichotomie [47] von Programmiersprachen gilt nicht für semantische Modelle. Alle Fakten, die nicht bekannt sind, evaluieren in Programmiersprachen aufgrund der *closed world assumption* zu *falsch/false*. Listing 4.2 zeigt ein einfaches Modell: *Mortal* ist die Sub-Klasse von *Person*, *Plato* ist vom Typ Person und *Sokrates* vom Typ Mortal. Das Statement `Sokrates in Person` evaluiert für die üblichen Programmiersprachen zu *false*, da keine explizite Verbindung zwischen Sokrates und *Person* besteht. Wird dem Modell jedoch ein offenes Weltbild zugrunde gelegt, kann die Frage, ob Sokrates eine Person ist, weder mit *true* oder *false* beantwortet werden. Denn prinzipiell könnte irgendwo in einem anderen Modell diese Information existieren. Wenn jedoch weder true noch false zutreffen, kann diese Frage nur mit *unknown* beantwortet werden, es somit nicht bestimmbar ob Sokrates eine Person ist oder nicht. Um die Frage nach dem Typ von Sokrates abschließend beantworten zu

können, müsste das Modell a) von einem Reasoner berechnet werden, aufgrund der Sub-Klassen-Beziehung wird Sokrates dann auch Teil von Person oder b) es müsste ein explizites Statement über die Zugehörigkeit von Sokrates getroffen werden. Entweder `<ex:Sokrates> a <ex:Person>` oder `<ex:Sokrates> not a <ex:Person>`. Nur wenn solch ein explizites Statement vorliegt greift das typische true/false wie es in Programmiersprachen üblich ist.

4.2. Bestehende Ansätze

Die hier vorgestellten Ansätze Deep Semantics und ActiveRDF sind als exemplarische Instanzen zu sehen. Es wurden diese beiden Ansätze ausgewählt, da sie recht gegensätzliche Verfahren einsetzen um zu einen ähnlichen Ziel zu gelangen.

4.2.1. Deep Semantics

Die Arbeit von Mainz [50] „Deep Integration of the OWL ontology language into Ruby using metaprogramming“ stellt einen Ansatz dar wie sich semantische Modelle in die objektorientierte Programmiersprache Ruby integrieren lassen.

Der Entwicklung von Deep Semantics lag der Bedarf nach einem hochintegrierten Zugriffsschemata auf Ontologien zugrunde, die von API wie Jena und OWLAPI nicht geboten werden. Ursprünglich sollte das Projekt auch einen Beitrag zum Ontologieeditor *Ontoverse* liefern, bis dato hat sich allerdings vor allem die Nutzung in webbasierten Ontologie-Anwendungen etabliert.

Architektur

Die Architektur von Deep Semantics basiert auf dem *Builder*-Pattern. So wird die nötige Entkopplung zwischen den einzelnen Komponenten erreicht, um flexibel auf neue Anforderungen reagieren zu können oder komfortabel die Funktionalität anpassen zu können. Bei Deep Semantics ist diese Flexibilität besonders wichtig, da die derzeitige Implementierung auf Ontologien vom Format OWL lite beschränkt ist. Diese Beschränkung wird jedoch nur durch die Komponente „Deep Integration Builder for OWL Lite“ (siehe Abbildung 4.4) hervorgerufen. So ist es dennoch ein Leichtes, ohne die übrigen Komponenten

verändern zu müssen, die Unterstützung von OWL DL Ontologien herbeizuführen, etwa durch eine Komponente „Deep Integration Builder for OWL DL“.

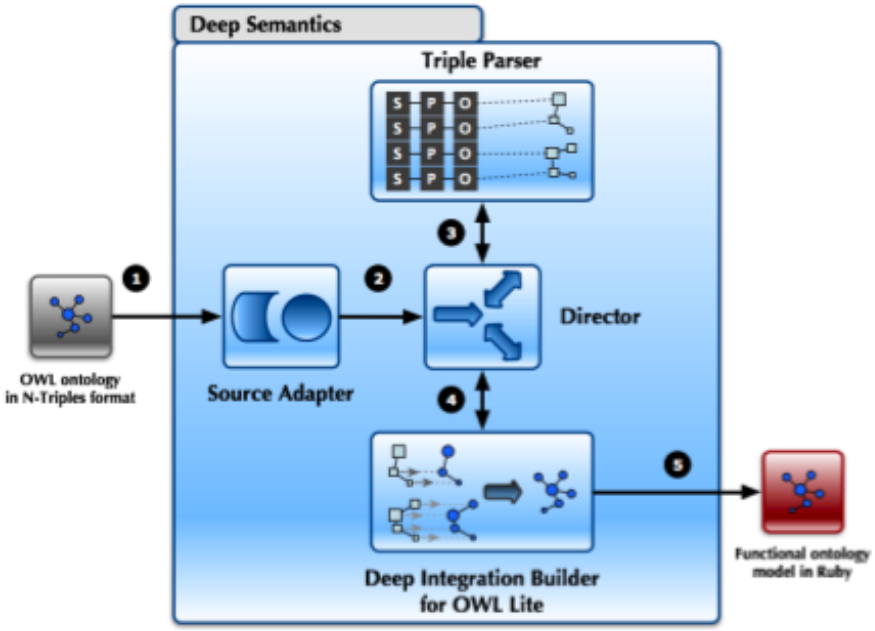

Abbildung 4.4.: Architektur von Deep Semantics [50]

Im Ganzen setzt sich Deep Semantics aus den Komponenten *Source Adapter*, *Triple Parser*, *Director* und *Integration Builder* zusammen.

Source Adapter Der *Source Adapter* dient zum Einlesen einer Ontologie in das Deep Semantics Framework. Grundsätzlich werden zwei Quellen unterschieden aus denen eine Ontologie bezogen werden kann. Zum einen aus einer Textdatei und zum anderen aus einer Datenbank. Der Begriff der Datenbank wird nicht genauer spezifiziert, so ist nicht klar, ob hiermit in erster Linie ein triple-store oder relationale Datenbanken gemeint sind. „For the current version if Deep Semantics [...] two different *Source Adapter* are available – for N-Triple files and N-Triples stored in a database table.“ [50], diese Bestandsaufnahme über den Source Adapter legt allerdings nahe, dass die Nutzung von relationalen Datenbanken intendiert ist.

Zusätzlich zu der Art, wie die Triple dargereicht werden, spielt zudem deren Serialisierung eine weitere Rolle bei der Anbindung an Deep Semantics. Zur Zeit wird lediglich Unterstützung für *N-Triple* geboten. Warum die Datenanbindung nicht mit Hilfe von bestehende Bibliothe-

ken realisiert wurde, ergibt sich aus der Arbeit von Mainz nicht. Bei der Verwendung einer solchen Bibliothek könnte man „out-of-the-box“ direkt die gängigsten Formate wie *XML/RDF*, *N3* und *N-Triple* nutzen.

Triple Parser Der Triple Parser dient der Überführung des vom Source Adapter generierten Stroms an Tripel-Daten in eine abstraktere Datenstruktur, welche zur weiteren Aufbereitung verwendet wird. Ebenso wird in dieser Komponente eine Reduktion der Datenmenge vorgenommen. Tripel, die sich auf SWRL-Elemente beziehen, sowie Tripel, deren Subjekt oder Objekt *owl:Nothing* oder *owl:Thing* ist, werden in einem vorgeschobenen Prozess aus der Datenmenge entfernt, da sie keinen substanziellen Beitrag für die Funktionalität von Deep Semantics liefern.

Nachdem das Modell reduziert wurde, werden in der Folge unterschiedliche Muster aus den Daten herausgesucht und für die weitere Verarbeitung abgespeichert. Unter anderem werden alle Properties (Data und Object), alle Klassen, alle SubClassen, alle Equivalenzbeziehungen und Ontologieeigenschaften als einzelne Mengen aus dem kompletten Datenbestand gewonnen. Alle bis zu diesem Punkt verarbeiteten Tripel beziehen sich auf das terminologische Wissen der Ontologie, die so genannten TBoxes.

Das Instanz-Wissen (ABoxes) wird getrennt vom terminologischen Wissen extrahiert. Hierbei ist wichtig anzumerken, dass in Deep Semantics nur Instanzen verarbeitet werden, die über das Prädikat *rdf:type* einer Klasse zugeordnet wurden. Dieses Verhalten wird jedoch häufiger in Object-Triple-Mappern angetroffen, da sich „anonyme“ Instanzen nur schwer integrieren lassen. Neben direkt getypten Instanzen werden allerdings auch *owl:sameAs* und *owl:differentFrom* ausgewertet. Das Prädikat *rdf:seeAlso*, welches das Ober-Prädikat zu *owl:sameAs* darstellt, wird allerdings nicht von der Implementierung erfasst.

Die gesammelten A- und TBoxes werden in eine propriätere Objektstruktur eingefügt (Template Values) und zur weiteren Verarbeitung an die *Director*-Instanz zurückgegeben.

Director Der *Director* ist die zentrale Komponente in Deep Semantics und auch die einzige Komponente mit der ein Entwickler direkt in Interaktion tritt. Von hier wird der gesamte Ablauf zum Aufbau des integrierten Modells gesteuert. Nach dem eine Instanz des Direktors erzeugt wurde,

wird mit einem passenden Source Adapter eine Ontologie geladen. Das entstandene *triple set* wird dann an den Triple Parser übergeben und verarbeitet. Das Resultat der Aufbereitung wird wiederum vom Director an den Integration Builder geleitet, der aus diesen Daten ein funktionales Ruby Modul erzeugt, welches im Anschluss vom Entwickler genutzt werden kann.

Integration Builder Der *Integration Builder* ist wie schon der Source Adapter, eine Komponente, die unterschiedliche Implementierungen vorsieht. Deep Semantics beinhaltet zur Zeit einen Integration Builder, der sich auf die Lite Variante von OWL bezieht. Andere Ontologiesprachen wie OWL DL, OWL Full oder RDFS werden noch nicht unterstützt.

Die Aufgabe des Integration Builders ist es, die Ontologie in ein Ruby-Modul umzusetzen. Hierfür bekommt der Builder als Eingabeparameter die vom Parser aufbereiteten Tripel. Für alle Klassen, die in der Ontologie gefunden wurden, werden auch Ruby-Klassen angelegt. Die Objekt- und Dateneigenschaften werden auf Membervariablen abgebildet. Zur Umsetzung des Modells in ein „ausführbares“ Ruby-Modul setzt Deep Semantics auf die Meta-Programmiereigenschaften wie *mod_eval* oder *class_eval*, die es ermöglich beliebige Zeichenketten als Ruby-Programmcode zu interpretieren.

4.2.2. ActiveRDF

Das Ruby-Modul *ActiveRDF*[51] wurde von Eyal Oren entwickelt. An der Namensgebung kann man bereits ablesen, welche Funktionalität durch dieses Paket zur Verfügung gestellt werden soll. Die Namensgleichheit zu *ActiveRecord* kommt nicht von ungefähr, das Ziel von Oren war es, den Zugriff auf RDF(S)-Daten so einfach zu gestalten wie der Zugriff auf relationale Datenbanken durch ActiveRecord unterstützt wird.

Aufbau

ActiveRDF liegt einer vierschichtigen Architektur zugrunde. Jede dieser Schichten kapselt jeweils eine bestimmte Funktionalität. Abbildung 4.5 zeigt die genaue Schichtung der Komponenten und die Interaktion zwischen den Schichten

aus der Sicht des Datenzugriffs aus der Applikation und aus Sicht der Datenquelle, wenn Daten für die Anwendung bereitgestellt werden müssen.

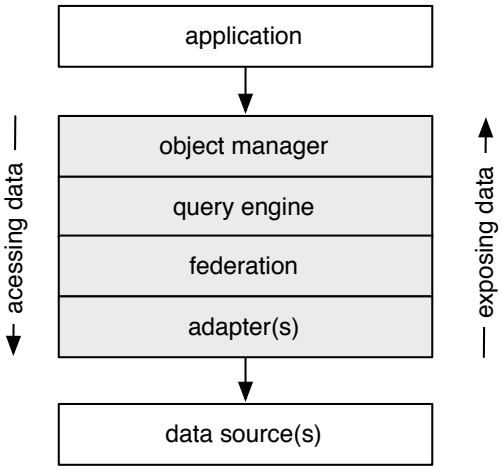

Abbildung 4.5.: Architektur von ActiveRDF [51]

object manager Der `ObjectManager` ist die Highlevel-API, um mit Ruby RDF-Daten zu manipulieren. Hier werden RDFS-Klassen auf Ruby-Klassen, Resourcen auf Ruby-Objekte und RDF-Properties auf Eigenschaften von Objekten abgebildet. Das Design der Schnittstelle orientiert sich nah an ActiveRecord, dessen Funktionalität mit Hilfe von ActiveRDF auf RDF-Daten projiziert werden soll. Durch die Abbildung von Eigenschaften auf die entsprechenden Ressourcen wird, wie auch bei ActiveRecord, eine domänenspezifische Sprache [52] (DSL) generiert, die dem Entwickler die Arbeit mit den Objekten vereinfachen soll.

query engine Mit Hilfe der `QueryEngine` lassen sich Abfragen an die Datenquellen der Anwendung konstruieren. Die Anfragen werden dabei zuerst in einer abstrakteren Form formuliert und werden erst zum Zeitpunkt in eine ausführbare Query überführt. So wird der Kode der Anwendung von der Anfragesprache, welche von der Datenquelle implementiert wird, unabhängig. Ebenso steht dem Entwickler eine einheitliche Möglichkeit zur Verfügung, um Anfragen zu formulieren ohne sich Gedanken über die zugrunde liegende Datenquelle machen zu müssen. Die `QueryEngine` wird natürlich auch intern vom `ObjectManager` verwendet, um benötigte Daten aus dem Datenspeicher zu beziehen.

Federation Manager Da ActiveRDF Unterstützung für multiple Datenquellen anbietet, wird auch eine Einheit benötigt, die alle Datenquellen einer Anwendung verwaltet. Diese Aufgabe wird durch den `Federation`

`Manager` übernommen. Wenn eine Anwendung mehrere Datenquellen verwendet, muss sichergestellt sein, dass alle Anfragen auch von allen Datenquellen beantwortet werden und die Ergebnisse zusammengefügt werden, bevor sie an die nächst höhere Schicht weitergegeben werden. ActiveRDF bietet keine besonders ausgefeilte Strategie, um die unterschiedlichen Datenquellen miteinander zu verbinden, jede Query wird lediglich an jede registrierte Datenquelle gestellt und die Ergebnisse zu einem Graphen zusammengefügt. Für zukünftige Versionen sind an dieser Stelle intelligentere Algorithmen geplant, um die Last auf die Datenspeicher zu verringern.

Adapter(s) Ein Vorteil von ActiveRDF ist, dass die Erweiterung per se nicht auf nur eine Art von Datenspeicher beschränkt ist. Es bestehen zahlreiche Möglichkeiten RDF-Daten einzubinden. Zur Zeit gibt es Adapter, mit denen sich Daten aus Sesame[4], Jena[5], Redland[6], Reddy[7], Rdflite und SPARQL-Endpoint[8] beziehen lassen. Die Adapter dienen dazu, die abstrakten Anfragen, die mit Hilfe der *QueryEngine* erzeugt werden, auf die Eigenschaften wie Anfragesprache, Format der Ergebnisse etc. der jeweiligen Datenspeicher abzubilden.

Benutzung

Wie schon erwähnt orientiert sich die Benutzung von ActiveRDF stark an der von ActiveRecord. Dies ermöglicht zum einen einen schnellen Umstieg und zum anderen lässt sich der Kode auch verstehen, wenn keine Erfahrung mit ActiveRDF vorhanden ist, die Konstrukten und die DSL von ActiveRecord aber bekannt sind.

```
1 require 'active_rdf'
2
3 # we add an existing SPARQL database as datasource
4 ConnectionPool.add_data_source(:type => :sparql, :results
      => :sparql_xml, :url => "http://m3pe.org:8080/
     repositories/test-people")
5
6 # we register a short-hand notation for the namespace
     used in this test data
```

[4]http://openrdf.org
[5]jena.sourceforge.net/
[6]http://librdf.org
[7]http://github.com/tommorris/reddy/
[8]http://www.w3.org/TR/rdf-sparql-protocol/

```
7  Namespace.register :test, 'http://activerdf.org/test/'
8  ObjectManager.construct_classes
9
10 all_resources = RDFS::Resource.find_all
11 all_people = TEST::Person.find_by_test::age(27)
12
13 # print all the people, and their friends
14 all_people.each do |person|
15   puts "#{person} has #{person.test::eye} eyes"
16 end
```

Listing 4.3: Initialisierung und Benutzung von ActiveRDF

Listing 4.3 zeigt exemplarisch auf, wie ActiveRDF benutzt werden kann. Prinzipiell wird immer damit begonnen (Zeile. 4) die Datenquellen in einem Pool von Verbindungen zu registrieren. Auch die Registrierung von Namensräumen gehört zu den Initialisierungsmaßmahmen, die vor der Benutzung durchzuführen sind. Die Instruktion `ObjectManager.construct_classes` dient dazu ,alle Klassen, die in den Datenquellen definiert sind, mit Hilfe von Metaprogramming in Ruby-Klassen zu überführen. Wenn die Initialisierung abgeschlossen ist, lassen sich z.B. mit der Methode `find_by_test::age(27)` alle Instanzen der Klasse Person extrahieren, deren Prädikat `http://activerdf.org/test/age` mit dem Wert `27` ausgeprägt ist. Der Zugriff auf die Eigenschaften einer Ressource wird mit Methodenaufrufen nach dem Muster `<Namensraum>::<Prädikatname>` ermöglicht. Handelt es sich bei dem Prädikat um eine Objekteigenschaft, so wird keine automatische Auflösung auf die Ressource vorgenommen, sondern lediglich die URI bereitgestellt.

4.3. Zusammenfassung

Auf Basis der in Abschnitt 4.1 diskutierten Differenzen und Gemeinsamkeiten zwischen semantischen und objektorientierten Modell und den betrachteten Ansätzen zur Integration sollen im folgenden die Anforderungen beschrieben werden, welche bei der Integration beider Paradigmen beachtet werden sollten.

4.3.1. Verwaltung der Ressourcen

Im Prinzip verfolgen alle bestehenden Ansätze die Idee, die Klassen des semantischen Modells auf Klassen der Programmiersprache abzubilden. Dies hat allerdings zum einen zur Folge, dass Ressourcen, die keiner Klasse angehören nicht abgebildet werden können und zum anderen sind die Vorstellungen davon was eine Klasse ausmacht in beiden Paradigmen teils so widersprüchlich, dass die Art der Abbildung dem semantischen Modell nicht gerecht werden kann.

Es ist daher sehr viel sinnvoller alle Instanzen des semantischen Modells zentral mit Hilfe einer generischen Klasse zu verwalten. Zusätzlich sollte aber auch die Möglichkeit bestehen einzelne Klassen auf Klassen der Programmiersprache abzubilden, diese übernehmen dann allerdings mehr eine Mediatorrolle, indem sie den Zugriff auf Ressourcen gleichen Typs kapseln.

4.3.2. Auflösung und Integration externer Ressourcen

Ein wesentlicher Kritikpunkt an den bestehenden Ansätzen besteht darin, dass Objekt-Eigenschaften, also Verknüpfungen zwischen zwei Ressourcen, nicht selbstständig von den Systemen aufgelöst werden. Dadurch, dass einige Systeme Daten aus mehreren Datenquellen beziehen können, wird der Verteiltheit von sematischen Daten zumindest grundsätzlich Rechnung getragen. Es ist jedoch nicht so, dass auch Ressourcen integriert werden können, die nicht von den registrierten Datenquellen abgedeckt werden. Die Anwendung operiert somit auf einem eigenen Datensilo [53] und ist nicht mehr direkter Konsument und Produzent des semantischen Webs.

Ein Rahmenwerk zur Integration muss somit, neben der Unterstützung von mehreren Datenquellen, auch in der Lage sein selbstständig unbekannte Ressourcen aufzulösen. Hierfür sollten die Empfehlungen der Linked Data Initiative implementiert werden, wie z.B. in [54].

4.3.3. Lazy Loading

Der Begriff *Lazy Loading*[55] beschreibt ein Design Pattern der Softwareentwicklung. Durch den Einsatz von *lazy loading* wird versucht, die Laufzeit von

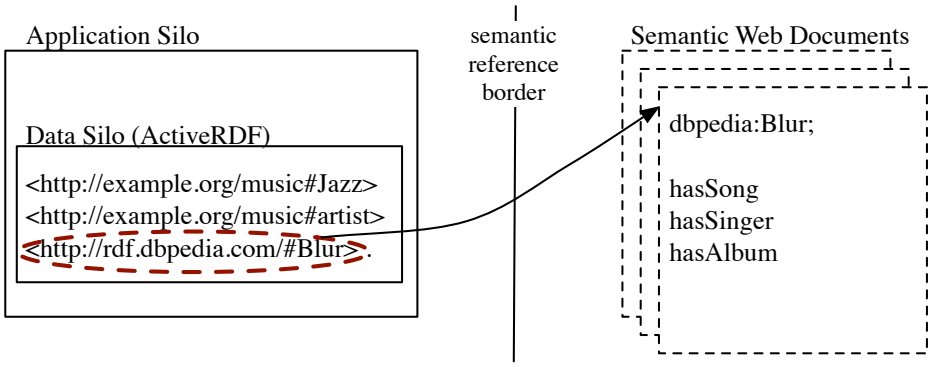

Abbildung 4.6.: Eine Schwierigkeit, die viele Bibliotheken, Frameworks oder Softwareprodukte rund um das Semantic Web betreffen, ist die Beschränkung, externe Ressourcen nicht selbstständig auflösen zu können. Zwar können im Modell Aussagen zu „externen“ URIs, extern bedeutet in diesem Kontext, dass diese URIs zwar Objekt eines Triples sind, aber keine weiteren Informationen im Datenspeicher vorliegen, getroffen werden. Es ist allerdings nicht möglich, die Informationen, die hinter einer externen URI stehen, zu extrahieren und das Wissen explizit verfügbar zu machen. Hierdurch entstehen Datensilos (kleine Teilgraphen), die nicht Teil des eigentlichen Semantic Webs sind.

Anwendungen zu optimieren, indem Ressourcen erst zu dem Zeitpunkt initialisiert werden, an dem sie auch tatsächlich benötigt werden. Im konkreten Fall der Integration von semantischen Ressourcen würde dies z.B. bedeuten, dass die Objekte eines Triples erst abgerufen werden, wenn die jeweilige Eigenschaft im Programmablauf verwendet wird.

Gerade im Umgang mit Ressourcen aus einem semantischen Modell eignet sich diese Technik der Softwareentwicklung, da die Ressourcen bisweilen große Mengen an Eigenschaften besitzen, die in den wenigsten Fällen alle verwendet werden. Würden alle Eigenschaften bei der Instanziierung der Ressource geladen, würde sich dies negativ auf die Performanz auswirken. Vergleichbar zum *Lazy Loading*-Pattern ist auch das *Proxy*-Pattern, welches nach einem ähnlichen Prinzip funktioniert.

4.3.4. Rechtzeitig plazierte Inferenzbildung

Bevor, oder spätestens wenn auf das semantische Modell zugegriffen wird, sollte ein Reasoning die impliziten Aussagen im Modell berechnet haben. Dies ist

insbesondere wichtig um Klassenzuordnungen korrekt vornehmen zu können, wenn Hierarchiekonstrukte genutzt wurden. Hier ergibt sich eine Problematik, wenn mehrere Datenquellen verwendet werden. Denn auch wenn die Datenspeicher schon die Inferenzen berechnet haben, müsste ein zusätzliches Reasoning über die gesammelte Tripelmenge druchgeführt werden, da sich hier wieder neue Aussagen ergeben könnten.

In der Regel sollte es für die meisten Fälle ausreichen wenn ein RDFS-Reasoner eingesetzt wird, wie z.B. innerhalb von Sesame. Für komplexere Modelle, die auch OWL Konstrukte verwenden muss dann allerdings ein OWL-(DL)-Reasonder herangezogen werden.

4.3.5. Erhaltung der Semantik

Diese Anforderung klingt auf den ersten Blick vielleicht absurd, ein Blick auf bestehende Implementierungen (siehe 4.2) zeigt allerdings, dass diese Forderung explizit gemacht werden sollte.

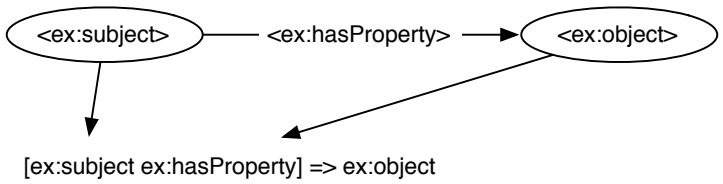

Abbildung 4.7.: Reduzierung der Wissenstiefe durch Triple-Object-Mapping

Fast alle bestehenden Ansätze zur Bewältigung der Integration von Ontologien in objektorientierte Sprachen nehmen eine Verkürzung der Informationstiefe vor. Das etablierte Vorgehen sieht in der Regel vor, dass die Eigenschaften einer Ressource auf die Eigenschaften einer entsprechenden Instanz abgebildet werden. Betrachtet man Beziehung zwischen Ressource, Eigenschaft und Ausprägung jedoch genauer, so wird offenbar, dass der Teil der *Eigenschaft* bei dieser Art des Mappings zwischen den Paradigmen verloren geht. Die Rezeptionstiefe wird an dieser Stelle somit künstlich herabgesetzt, da Informationen ausgeblendet werden, die grundsätzlich zur Verfügung stünden. Abbildung 4.7 zeigt den Zusammenhang schematisch auf, ausgehend von dem Tripel `<ex:subject> <ex:hasProperty> <ex:object>` wird eine Transformation vorgenommen, die in einer Instanz `ex:subject` resultiert, die mit der Instanz `<ex:object` verbunden ist. Das Bindeglied zwischen beiden wurde

reduziert auf eine benannte Zugriffsschnittstelle `<ex:hasProperty>`. Im semantischen Modell handelt es sich bei `<ex:hasProperty>` allerdings um eine eigene Ressource, die auch eigene Eigenschaften besitzt, dieses Wissen geht allerdings verloren, denn `<ex:hasProperty>` lässt sich bei dieser Art der Abbildung nicht mehr als Instanz adressieren. Besonders gravierend ist dieser Einschnitt, da dieses zusätzliche Wissen über Beziehungen zwischen Ressourcen einer der Mehrwerte von semantischen Modellen ist. Die Integration in die Objektorientierung geht hier einen Schritt zu weit.

4.3.6. Kleinster gemeinsamer Teiler

Ein weiterer Aspekt, der Beachtung finden muss und sich teils tiefgreifend auf die Integration auswirkt, ist die Aufrechterhaltung der Kompatibilität. Es stellt sich nämlich die Frage, ob das OTM die Modellierung beeinflussen soll und darf oder ob jede Modellierung durch das OTM unterstützt werden soll. Quasthoff setzt bei seiner Implementierung [56] einige Modellierungkonstrukte voraus, um die Ontologie integrieren zu können. Betrachtet man das OTM jedoch nur als Werkzeug zum Zugriff auf beliebige Wissensmodelle, so wird klar, dass das Werkzeug nicht das Design des Werkstückes beeinflussen sollte. Aus dieser Betrachtungsweise resultiert natürlich, dass die Integration nur auf die grundlegendsten Modellierungskonzepte aufbauen kann und alle erweiterten Möglichkeiten nur optional eingebunden werden können. Im Konkreten bedeutet das, dass unter anderem folgende Konstrukte nicht genutzt werden können, um Rückschlüsse für die Integration zu ziehen:

- `rdfs:label`
- Die explizite Kennzeichnung von Ressourcen als Property mit `rdfs:Property`, `owl:DatatypeProperty` oder `owl:ObjectProperty`
- `domain` und `range` als erweiterte Eigenschaften eines Properties

Für die Integration muss jedoch auf die zuvor genannten Informationen nicht verzichtet werden, diese dürfen allerdings lediglich als zusätzliche Parameter verwendet werden. Die Basis-Integration baut somit nur auf wenigen Modellierungskonzepten auf, nämlich:

- `rdf:type` als Zuordnung von Ressourcen zu einer Klasse
- `rdfs:Class` in Kombination mit `rdf:type`, um Ressourcen als Klasse zu markieren.

- Jegliche Informationen, die aus einem Tripel gewonnen werden können.

Wenn die Integration auf diese Weise gestaltet wird, erlangt man eine wesentlich größere Kompatibilität, d.h. im Idealfall kann jede beliebige Ontologie als Datenmodell herangezogen werden. Diese Freiheit besteht nicht, wenn Ontologien bestimmte Modellierungskriterien erfüllen müssen, um integriert werden zu können.

5. Prototypische Implementierung

Das folgende Kapitel widmet sich der Ruby-Erweiterung „SemanticRecord“, die eine prototypische Implementierung des in den vorhergehenden Kapiteln entwickelten formalen Modells zur Integration von Ontologien in objektorientierte Programmiersprachen darstellt. Neben Details zur Architektur und genauen Implementierung wird die Erweiterung auf ihre Validität hin überprüft.

5.1. Architektur

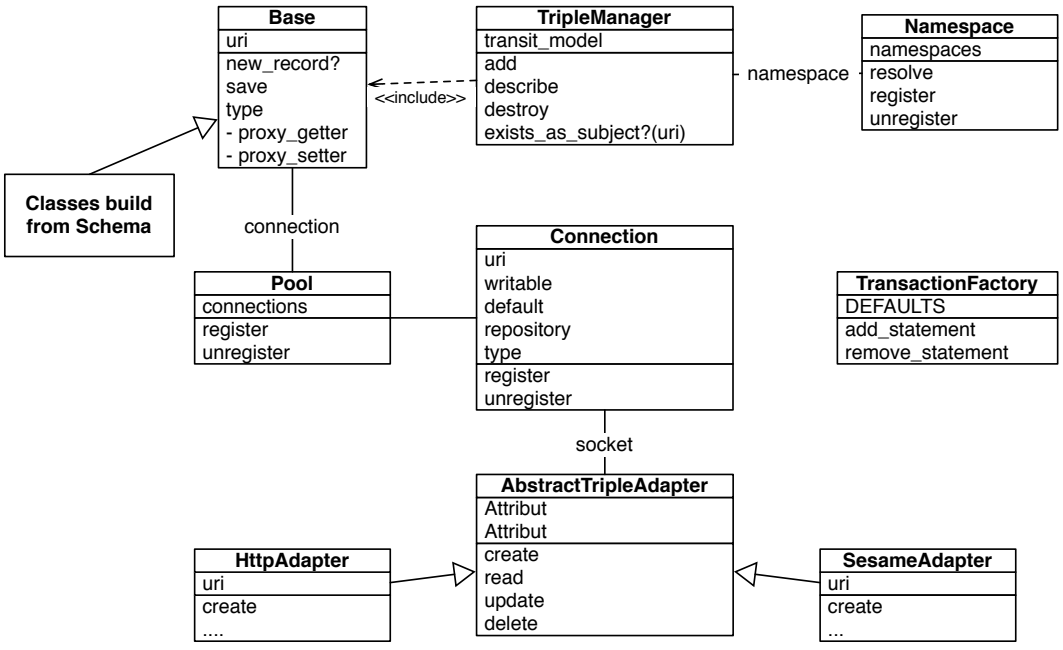

Abbildung 5.1.: Klassendiagramm

5.1.1. Triple Manager

Der Triple Manager ist das Kernstück von Semantic Record. Alle tripel-bezogenen Operationen werden durch den `TripleManager` transparent auf das `Transit-`

`Model` oder auf die entsprechenden Datenquellen verteilt. Die komplette Triple-Logik wurde von `SemanticRecord::Base` ausgelagert, um das Anwendungsmodell flexibel zu halten. Bislang ist das `TransitModel` auf Basis von *Redland* implementiert, ggfs. kann es notwendig sein hier in anderen Kontexten auf eine andere Implementierung zurückzugreifen, wenn z.B. JRuby verwendet wird, kann ein Wechsel auf *Jena* oder *Sesame* sinnvoll sein. Durch die Ausgliederung lässt sich diese Komponente nun transparent austauschen, solange die entsprechenden Schnittstellen bedient werden.

5.1.2. Connection Pool

Für die Verwaltung der einzelnen Datenquellen ist der Connection Pool verantwortlich. Die Methoden `register` und `unregister` dienen dem Hinzufügen bzw. dem Entfernen von Datenquellen. Sparql-Abfragen, die vom Triple Manager durchgereicht wurden, werden über alle registrierten Datenquellen verteilt und die Ergebnisse im Transit Modell zusammengeführt. Semantic Record kann prinzipiell alle Typen von Datenquellen unterstützen, die das Sparql-Protocol implementieren und `CONSTRUCT`-Queries beantworten können.

Ein `Connection`-Objekt wird über eine Reihe von Attributen definiert:

uri Bezeichnet die URI, unter der die Datenquelle erreichbar ist. In den meisten Fällen wird es sich hierbei um HTTP-URLs handeln, jedoch sind auch andere Protokolle denkbar.

writable Semantische Webanwendungen beziehen ihre Daten zu großen Teilen aus Quellen, die außerhalb des administrativen Umfelds liegen, somit ist es üblich, dass auf die Quellen lediglich Lese-, aber kein Schreibrecht besteht. Dieses Attribut spezifiziert, ob die Anwendung neue Triple in den Datenspeicher schreiben darf.

default Innerhalb einer Anwendung muss mindestens eine Verbindung existieren, die als `default` gekennzeichnet ist. Tripel, die im Anwendungsablauf gespeichert werden sollen, die sich aber auf eine schreibgeschützte Verbindung beziehen, werden in diesem Speicher abgelegt. Dementsprechend muss die `default`-Verbindung zwingend Schreibrechte besitzen.

type Das `type`-Attribut legt den Typ der Verbindung fest. Alle möglichen Typen müssen die abstrakte Klasse `AbstractTripleAdapter` implementieren, um zu garantieren, dass die benötigten Methoden bereit gestellt

werden. Nach der Registrierung einer Datenquelle wird ein Objekt des Typs `<type>Adapter` mit den Attributen (`uri, writable, default`) erzeugt.

5.1.3. AbstractTripleAdapter & dessen Unterklassen

Die Klasse `AbstractTripleAdapter` definiert eine Reihe von Methoden, die von spezifisch ausgeprägten `TripleAdaptern` implementiert werden müssen, um innerhalb von Semantic Record eingesetzt werden zu können. Da Ruby das Konzept von abstrakten Klassen nicht unterstützt, wird das entsprechende Verhalten einer abstrakten Klasse durch `AbstractTripleAdapter` nachgebildet. Die Methoden `create`, `read`, `update` und `delete` (CRUD) werden zwar definiert, jedoch nicht ausimplementiert, anstatt dessen werfen die Methoden einen `NotYetImplemeted`-Fehler.

Neue Adaptertypen müssen von der Klasse `AbstractTripleAdapter` abgeleitet werden. Um prinzipiell sicher zustellen, dass die entsprechenden Methoden angeboten werden und sich so verhalten wie angenommen. Diese Art der Klassengestaltung ist zwar nicht notwendig, bietet für Entwickler allerdings genauere Hinweise, wie ein neuer Adapter zu implementieren ist. Zum derzeitigen Implementierungsstand werden zwei Adapter unterstützt, `SesameAdapter` zur Einbindung von Sesame triple-stores und `HttpAdapter` für die Integration von Sparql-Endpunkten.

5.1.4. SemanticRecord::Base

Neben dem *Triple Manager* stellt `SemanticRecord::Base` die Essenz von Semantic Record dar. Ein Entwickler, der eine Anwendung auf Basis von Semantic Record implementiert, wird in der Regel nur mit dieser Klasse in Berührung kommen. `Base` kapselt zum einen die Verbindungsverwaltung vom `ConnectionPool` und regelt die Abstraktion der Ressourcen-Eigenschaften. Aus Sicht der semantischen Modellierung ist `base` das Äquivalent zu `owl:Thing`, der Klasse die alle weiteren Klassen und Ressourcen unter sich vereint. Mit der Methode `find` können alle zur Klasse gehörigen Ressourcen *gefunden* werden. Im Fall von `Base`, und dementsprechend `owl:Thing`, sind dies alle Instanzen, die in den registrierten Datenquellen vorhanden sind. Bei der praktischen Anwendung ist es dann üblich, weitere Klassen aus der Anwendungsdomäne zu

definieren, die von Base abgeleitet werden, diese Klassen entsprechen dann dem Typ `rdfs:Class` bzw. `owl:Class`.

Objekte vom Typ `SemanticRecord::Base` oder deren Unterklassen sind zum einen natürlich gewöhnliche Instanzen der jeweiligen Klasse, allerdings repräsentieren sie auch Instanzen die jeweiligen Klasse im semantischen Modell. Gesetzt des Falles, es existiert eine Klasse `Leuchte`, die von `SemanticRecord::Base` abgeleitet wurde, so referenziert ein Objekt, das mit der URI `http://erco.com/products#76619000` angelegt wurde, auf eine Ressource mit eben dieser URI. Der Zugriff auf die Eigenschaften erfolgt nach dem Schema `<Namensraum>_<Prädikatname>`, die übliche Schreibweise mit „:“ als Trennzeichen wird von Ruby nicht unterstützt, so dass auf dieses Format ausgewichen werden musste. Die Namensräume müssen bei der Initialisierung, oder zur Laufzeit im Modul `Namespace` registriert werden, die Namensräume `owl, rdf, rdfs` sind jedoch bereits vorkonfiguriert. Unter der Annahme, eine Variable `p` sei mit einem Objekt vom Typ `Leuchte` assoziiert, so liefert die Methodenaufruf `g.rdf_type` eine Liste aller Klassen im semantischen Modell mit denen `g` in Relation steht, projiziert auf die Syntax von Sparql entspricht diese Abfrage dem Triple-Muster `<g> rdf:type ?class_name`.

5.1.5. Support

Unter Support sind Klassen definiert, die keinen direkten Bezug zu Semantic Record haben, deren Funktionalität jedoch wesentlich ist. Ebenso ist es denkbar, diese Module auch in anderen Anwendungen einzusetzen.

Namespaces ist ein Verzeichnis, das verwendet wird, um XML Namespaces zu registrieren und wieder aufzulösen. Diese Funktionalität wird benötigt, um innerhalb von `Base` die Methodenaufrufe nach dem Schema `<Namensraum>_<Prädikatname>` auf die entsprechende URI aufzulösen.

TransactionFactory ist eine Wrapper-Klasse für das Sesame *Transaction Format*, um schnell und einfach Transaktionsdokumente zu erstellen. Mit Hilfe dieser Transaktionsdokumente lassen sich mehrere Einfüge- und Löschoperationen zusammenfassen, um die Konsistenz des Datenbestand nicht durch fehlerhafte oder abgebrochene Operationen zu gefährden. Ebenso lässt sich mit einem Transaktionsdokument die Update-Funktionalität imitieren.

5.2. Umsetzung

5.2.1. Proaktives Zugriffsverhalten

In Kapitel 4.3.3 wurde propagiert, dass für den Zugriff auf die Eigenschaften von Ressourcen das Lazy-Loading-Pattern angewendet werden soll. Dies ist insofern sinnvoll, als dass die Menge von Eigenschaften, die eine Ressource besitzt, generell sehr groß sein kann. Betrachtet man exemplarisch einige Ressourcen aus dem Modell der DBpedia[1], so ergibt sich ungefähr eine durchschnittliche Prädikatanzahl von 40 pro Ressource. Die Wahrscheinlichkeit, dass alle diese Eigenschaften in einer Anwendung zur gleichen Zeit zum Tragen kommen, lässt sich aus Erfahrung jedoch als relativ gering abschätzen. Daraus würde folgen, dass wenn beim Überführen einer Ressource ins Objektorientierte Modell auch alle ihre Relationen direkt instanziiert werden würden, viele dieser Objekte niemals verwendet werden würden.

Generell bietet dieser Sachverhalt es an, die entsprechenden Eigenschaften erst zur Laufzeit aufzulösen, wenn sie auch tatsächlich verwendet werden. Für ActiveRDF [57] wurde von den Entwicklern genau dieser Weg gewählt. In der Untersuchung von mainz [50] wird jedoch deutlich, dass sich das dynamische Nachladen der Inhalte negativ auf die Laufzeit auswirken kann. Die Laufzeiten wurden auf einem 2Ghz Dual-Core AMD Opteron System mit 2GB RAM ermittelt und beziehen sich auf die Dauer, die benötigt wurde, um alle Ressourcen einer Ontologie zu finden, hierzu wurde zum einen die BIO2Me[2] Ontologie und zum anderen die IKEN[3] Ontologie verwendet.

In Abbildung 5.2 zeigen sich deutlich die Laufzeitunterschiede zwischen den einzelnen Frameworks im Vergleich. Während Deep Semantic, Jena und OWLAPI mit maximal 3 Sekunden Laufzeit aufwarten, fällt ActiveRDF mit 60 bzw. 35 Sekunden signifikant aus dem Rahmen. Dieser Unterschied wird in erster Linie damit beantwortet, dass die ersten drei Frameworks *eager*-loading einsetzen und ActiveRDF auf *lazy* loading setzt. Es darf aber nicht außer Acht gelassen werden, dass die exzellenten Werte der anderen Frameworks auf Kosten der Dynamik gekauft werden. Denn diese Lösungen können auf Änderungen im Datenmodell wesentlich inflexibler reagieren als ActiveRDF.

[1] http://dbpedia.org

[2] BIO2ME ontology consisting of 213 named classes, 41 object properties, 29 datatype properties and 343 individuals [50]

[3] IKEN ontology comprising 366 named classes, 86 object properties, 23 datatype properties and 859 individuals http://i-ken.de

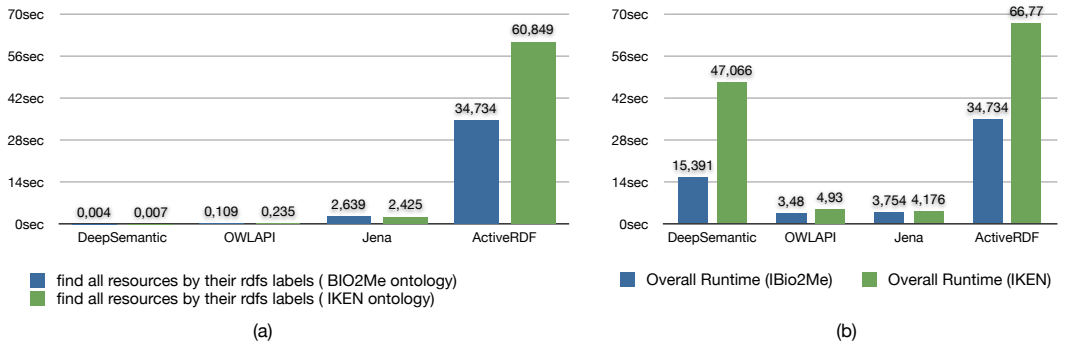

Abbildung 5.2.: (a) Laufzeitenvergleich für das Finden von zehn Ressourcen anhand vorgegebener Label. (b) Akkumulierte Laufzeit für das Finden und Vor- und Nachberechnungen. nach [50]

Damit Semantic Record die Anforderungen bezüglich Dynamik und Verteiltheit erfüllen kann und sich die Laufzeiten in einem akzeptablen Rahmen halten, bedarf es einem differenzierteren Ansatz zum Zugriff auf die Eigenschaften von Ressourcen. Die Gratwanderung zwischen Geschwindigkeit, Flexibilität und Dynamik wird bei der Entwicklung durch ein Konzept mit dem Namen *Transit Model* versucht Rechnung zu tragen. Vereinfacht dargestellt handelt es sich bei einem Transit Model um einen Datenspeicher, der zwischen originären Datenspeicher und Anwendungslogik eingeschoben wird. Grundsätzlich könnte dieses Konzept auch als Cache bezeichnet werden, welcher die später benötigten Informationen performant vorhalten kann. Zusätzlich hierzu sind dem Transit Model allerdings noch weitere Funktionen zugeordnet, die von gewöhnlichen Caching-Systemen nicht geboten werden.

Das Transit Model agiert als Mediator zwischen Datenspeicher und Applikationslogik, d.h. fast alle SPARQL-Abfragen laufen nicht mehr gegen externe Datenspeicher, sondern gegen das lokale Transit Model. Die einzige Ausnahme bildet hier der *Initial Call*. Wenn ein Objekt neu erzeugt wird, so wird an die registrierten Datenspeicher eine *DESCRIBE*-Anfrage gesendet und das Ergebnis im Transit Model gespeichert. Wenn nun auf ein Prädikat der Ressource zugegriffen wird, ist lediglich das Transit Model für die Beantwortung der Frage zuständig. Der Geschwindigkeitsvorteil ergibt sich aus der Antwortzeit, die der Datenspeicher braucht um ein Ergebnis liefern zu können. Bei der Verwendung des Ruby Gems `ruby-sesame` ergibt sich eine durchschnittliche Dauer von 0,06 Sekunden pro Anfrage, wenn sich der Datenspeicher im gleichen physikalischen Netz befindet. Wird ein Datenspeicher „irgendwo“ im Netz angesprochen, so erhöht sich die Antwortzeit auf durchschnittlich 0,1ms. Der

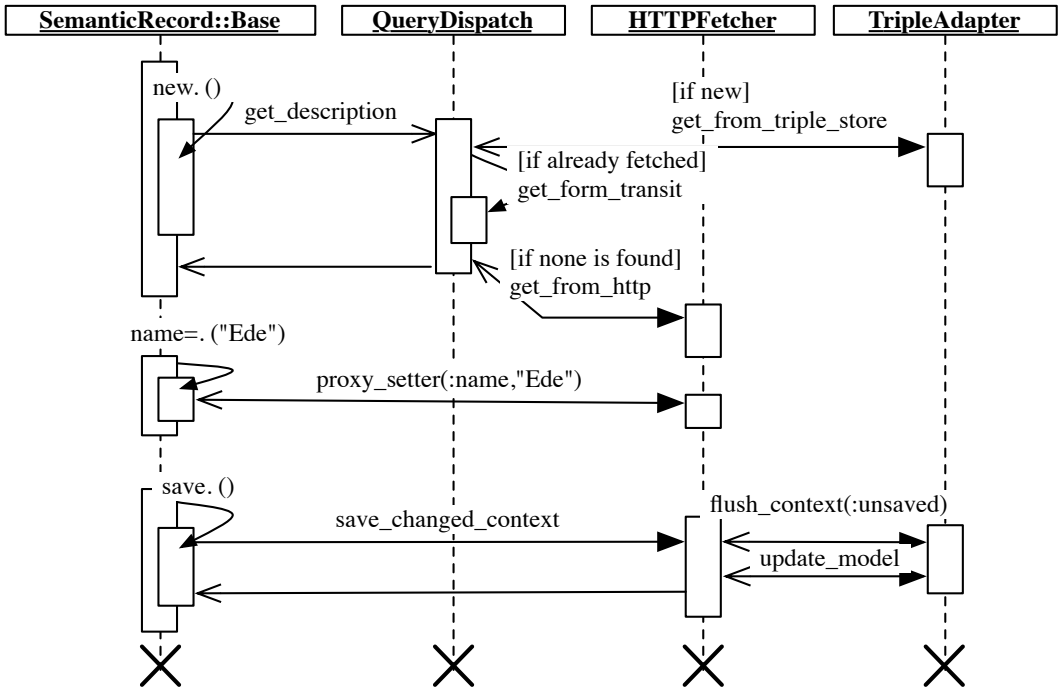

Abbildung 5.3.: Sequenzdiagramm

Zugriff auf einen lokalen `Redland`[4] Speicher liegt dagegen bei etwa 0,005ms. Beim reinen Vergleich der Antwortzeiten lohnt sich der Einsatz eines Transit Models, wenn auf mindestens zwei Prädikate einer Ressource zugegriffen wird. Werden weniger Prädikate angesprochen, so liegt der Zeitaufwand immerhin auf dem gleichen Level wie beim direkten Zugriff auf den Datenspeicher. Die gemessenen Zeiten sind natürlich nicht repräsentativ und unterliegen systembedingten Schwankungen, der grundsätzlich vorhandene Trend ist dennoch erkennbar und rechtfertigt die Entscheidung für das so genannte Transit Model.

Das Sequenzdiagramm, Abbildung 5.3, zeigt noch einmal die Funktion des Transit Modells auf. Wenn ein neues Objekt erzeugt werden soll, wird eine Anfrage an den *Query Dispatcher* gesendet und überprüft, ob im Transit Modell bereits Tripel vorhanden sind, in denen die URI des zu erzeugenden Objekts als Subjekt verwendet wird. Werden keine Daten im Transit Modell gefunden, wird der *Triple Adapter* angefragt, die benötigten Informationen aus dem registrierten Datenquellen zu aggregieren und dem Transit Modell hinzuzufügen. Der Aufruf von Methoden, die sich auf die Prädikate der Ressource beziehen, werden über so genannte *proxy_getter* und *proxy_setter* abgefangen und an das Transit Modell weitergeleitet. Änderungen am Zustand des Objekts werden

[4] http://librdf.org

zuerst im Transit Modell gespeichert. Die persistente Speicherung im Triple Store wird durch den Aufruf von `save()` initiiert. Bei dieser Gelegenheit werden zusätzlich die Tripel im Transit Modell aktualisiert.

5.2.2. Einbettung mehrerer Datenquellen

Anders als bei klassischen Webanwendungen, die zumeist auf relationalen Datenbanken basieren, existieren bei semantischen Webanwendungen keine monolithischen Datensilos. Semantische beziehen ihre Daten in der Regel aus mehreren Datenquellen, die sich in vielen Fällen nicht oder nur indirekt im Einflussbereich der Anwendung befinden. Die einzige Gemeinsamkeit zwischen den Datenquellen ist, dass sie ihre Daten entweder über das SPARQL-Protocol [58, 59] oder HTTP-Content-Negotiation [60, 61, 34] bereitstellen.

In der Praxis resultiert daraus die Verteilung der Informationen, die sich auf eine spezifische Ressource beziehen. So existiert beispielsweise eine Datenquelle, die GEO-Informationen über eine Ressource bereithält und eine weitere Quelle, die meteorologische Informationen zur Verfügung stellt. Um ein vollständiges[5] Bild dieser Ressource geben zu können, bedeutet dies in der Konsequenz, dass Semantic Record mehrere Datenquellen anbinden, verwalten und integrieren muss. Da die einzelnen Informationsstücke als RDF Graph codiert sind, können sie wesentlich einfacher zusammengeführt werden als zum Beispiel Baumstrukturen. Innerhalb von Semantic Record wird das Zusammenführen der einzelnen Datenquellen zu einem kompletten Modell ebenfalls vom Transit Model durchgeführt. Das Transit Model hierfür einzusetzen liegt insofern auf der Hand, da aus Architektursicht alle Anfragen vom Transit Model beantwortet werden. Davor steht das Befüllen des Transit Models, anstatt nun die Informationen zu einer Ressource nur aus einem Datenspeicher zu beziehen, werden einfach alle registrierten Datenspeicher zu Informationen über die spezifische Ressource befragt und die gesammelten Informationen im Transit Model gespeichert.

Die Verwaltung der einzelnen Datenquellen ist Teil des so genannten `Connection Pools`. Dieses Modul dient dazu bekannte Datenquellen unter Angabe ihrer URL und weiteren Metadaten zu registrieren und ein typenspezifisches Verbindungsobjekt (`socket`) anzulegen, welches sich von der Klasse `AbstractTriple-Adapter` ableitet. Mit diesem Verbindungsobjekt werden unterschiedliche Typen von Datenquellen abstrahiert, um Semantic Record später einen einheitli-

[5] vollständig im Sinne aller bekannten und erreichbaren Informationsquellen

chen Zugriff auf die Datenquelle zu ermöglichen.

Sobald alle Informationen im Transit Model aggregiert worden sind, bietet sich als nächster Schritt an, dieses Modell durch einen Reasoner weiter anzureichern. Bislang besteht jedoch noch keine Möglichkeit einen Reasoner in Redland einzubinden, um so eine entsprechende Funktionalität zu implementieren.

5.2.3. Integration von Linked Data

Alle bislang vorgestellten Verfahren und Ansätze basieren auf der Annahme, dass die Anwendungsdaten in einem oder mehreren Triple-Stores angelegt sind. Semantische Webanwendungen auf dieser Ebene zu entwickeln, kann zwar aufgrund der Vorzüge, die ein semantisches Modell gegenüber einer entity-relationalen Modellierung bietet, berechtigt sein, allerdings wird somit ein wesentlicher Teilaspekt des Semantic Webs ausgespart, der die Essenz, aber auch den streitbarsten Teil des Semantic Web bedeutet. Da alle Ressourcen in einem RDF Graphen über eine URI eindeutig identifiziert werden können, ergibt sich im Idealfall durch das Zusammenfügen aller existierenden RDF Graphen der Welt ein globaler RDF Graph, prinzipiell genau das Konstrukt, welches von Tim Berners-Lee als „The Semantic Web“ beschrieben wurde. Triple-Stores ermöglichen zwar einfachen und strukturierten Zugriff auf RDF Daten, hierfür müssen aber die kleinen Teilgraphen des Semantic Web isoliert werden. In der Konsequenz entstehen somit die gleichen Datensilos, die auch bei datenbankorientierten Anwendungen entstehen, lediglich die Art der Datenhaltung hat sich geändert. In Kapitel 4 wurde dieses Phänomen als *Semantic Reference Border* vorgestellt, also das Unvermögen der Triple-Stores über den eigenen „Tellerrand“ zu blicken.

Die Arbeit von Hartig et al. [54] ist ein vielversprechender Ansatz um das benannte Problem zu adressieren. Im wesentlichen beruht der Ansatz auf dem in [34] beschriebenen Verfahren, um RDF Daten mittels HTTP-Content-Negotiation verfügbar zu machen. Hierzu werden in einem ersten Schritt alle URIs, die in einer Sparql-Anfrage vorhanden sind über HTTP abgerufen. Die Ergebnisse werden in einem lokalen Tripelspeicher abgelegt und dann die ursprüngliche Anfrage auf diesen Datenspeicher ausgeführt. Dieses Verfahren funktioniert allerdings nur wenn hinter den URIs auch Inhalte in einer entsprechenden Repräsentation vorgehalten werden. Dies wird allerdings erst seit

dem Aufkommen der Linking Open Data Initiative propagiert, in früheren Aufsätzen des W3C wurde noch nicht gefordert, dass hinter den URIs auch Beschreibungsdokumente der Ressourcen zu finden sein müssen.

Auch Semantic Record wird diese Möglichkeit nutzen, um entweder weitere Informationen zu einer bekannten Ressource zu finden oder auch um Informationen über unbekannte Ressourcen zu aggregieren. In Abbildung 5.3 ist der Funktionsablauf schematisch dargestellt. Im Beispiel wird ein Objekt erzeugt, im ersten Schritt werden wie gewohnt die registrierten Datenspeicher abgefragt. Zusätzlich zu den Informationen, die in den Datenspeichern gefunden wurden, wird ein HTTP-Request auf die URI der Ressource abgesetzt, der HTTP-Accept Header wird hierbei auf `application/xml+rdf` gesetzt, um der Gegenstelle zu signalisieren, anstatt der HTML-Variante den Inhalt als XML/RDF auszuliefern. Die so gesammelten Daten werden dann ebenfalls im Transit Modell gespeichert.

5.3. Verwendung

Die Verwendung von SemanticRecord geht grundsätzlich immer auf mehrere Schritte zurück: 1) Deklarieren der Datenquellen 2) Registrieren von anwendungsspezifischen Namensräumen 3) Initialisieren einer `Base-URI` für die Anwendung 4) Anlegen von domänenspezifischen Klassen als Modelle der Anwendung. Darauf folgt die Verwendung von Klassen und Objekten wie bei Webanwendungen, die auf relationalen Datenbeständen aufbauen.

```
1 SemanticRecord::Pool.register( {:uri => "http://localhost
    :8080/openrdf-sesame",:type => :sesame, :default =>
    true, :writable => true,:repository => "erco" } )
2 SemanticRecord::Pool.register( {:uri => "http://products.
    erco.com/light_knowledge",:type => :sparql, :default
    => false, :writable => false,:repository => "light" }
    )
3
4 Namespaces.register( {:product => "http://knowledge.erco.
    com/products#" })
5 Namespaces.register( {:light => "http://knowledge.erco.
    com/light#" })
6
7 SemanticRecord::Base.base = "http://knowledge.erco.com/#"
8
9 class Leuchte < SemanticRecord::Base
```

```
10   self.base=
11 end
```

Listing 5.1: Initialisierungsprozess von SemanticRecord

Listing 5.1 zeigt die Deklaration der Datenquellen. Im gegeben Beispiel werden ein Sesame triple-store und ein generischer Sparql-Endpunkt angebunden. Neben der „programmatischen“ Anbindung der Datenquellen können die Parameter auch in einer YML-Datei abgelegt werden, die mit dem Aufruf `SemanticRecord::Pool.load` geladen wird. Darauf folgend beschreiben die Zeilen 4 und 5 das Vorgehen zum Hinzufügen der anwendungsspezifischen Namensräume, hier werden `products` und `light` definiert. Einige Datenspeicher bieten Methoden, um die intern verwendeten Namensräume zu extrahieren, aufgrund von möglichen Überscheidungen in unterschiedlichen Datenquellen wurde bei der Konzeption von SemanticRecord auf diese Funktionalität bewusst verzichtet und stattdessen auf die manuelle Registrierung gesetzt. Die Anweisung in Zeile 7 setzt die `Base-URI` der Anwendung. Diese Base-URI wird im Anwendungsverlauf verwendet um URIs für neue Individuen zu vergeben. Mit den Direktiven `base=` und `uri=` können diese Defaultwerte jedoch überschrieben werden. In Zeile 9-11 wird letztlich eine neue Klasse angelegt, die als semantisches Modell[6] der Anwendung dient.

In Listing 5.2 wird die Verwendung von SemanticRecord in der Ruby interactive shell (irb) dargestellt, die Rückgaben wurden hier allerdings zur Erhöhung der Lesbarkeit auf die wesentlichen Elemente verkürzt. Mit der Anweisung in Zeile 1 lassen sich alle Instanzen der Klasse `Leuchte` abrufen, d.h. alle Instanzen vom Typ (rdf:type) `http://knowledge.erco.com/#Leuchte` (vgl. Listing 5.1). Die Semantik von `rdf:type` schreibt vor, dass sich die Objektseite einer Beziehung auf eine Klasse bezieht. Dieses Verhalten wird auch von SemanticRecord widergespiegelt, der Methodenaufruf (`rdf_type`) liefert zwar eine Liste von Objekten des Typs `SemanticRecord::Base`, diese Objekte verfügen jedoch selbst auch wieder zusätzlich über die entsprechenden Klassenmethoden.

```
1 >> p = Leuchte.find
2 [sadada]
3 >> p[0].rdf_type
4 []
5 >> p[0].light_has_luminous_flux
6 "1000"^^xsd:integer
```

[6] Modell im Sinne des MVC Paradigmas

```
7  >> pred = p[0].predicate(:light_has_luminous_flux)
8  ["http://knowledge.erco.com/light#"^^SemanticRecord::
       Property]
9  >> pred.context
10 ["http://products.erco.com/light_knowledge"]
```

Listing 5.2: Verwendung von SemanticRecord innerhalb von irb

Zeile 3 zeigt wiederum den Zugriff auf ein einfaches Prädikat der ausgewählten Instanz. Die präsentierten Anwendungsfälle unterscheiden sich bislang nicht wesentlich vom Einsatz von `ActiveRecord` oder `ActiveResource`. Diese syntaktische Nähe zu etablierten *Datenadapter* ist jedoch eines der Designziele von SemanticRecord, dem Entwickler soll der Zugriff auf semantische Daten in ähnlichem Maße erleichtert werden, wie es diverse Frameworks für relationale Daten vorgemacht haben. Aus dem adressierten Kontext ergeben sich allerdings auch neue Funktionen, die in bisherigen Ansätzen nicht realisierbar oder teils auch nicht notwendig waren. Hierzu gehört der direkte Zugriff auf Eigenschaften einer Ressource als eigenständiges Objekt (Zeile 7). Mit der Methode `predicate`, über die jede Instanz vom Typ `SemanticRecord::Base` verfügt, lässt sich ein spezifisches Prädikat unter Angabe der URI oder verkürzten Schreibweise als Objekt vom Typ `SemanticRecord::Property` instanziieren.

Mit Hilfe dieses Objekts können nun auf weitere Informationen über das entsprechende Prädikat zugegriffen werden. Hierbei kommt dem Kontext des Prädikats eine besondere Bedeutung zu. Während in klassischen Webanwendungen klar ist, aus welchem Datenbestand die Daten herrühren, so kann eine Information innerhalb einer semantischen Webanwendungen aus quasi jeder assoziierten Datenverbindung stammen. Damit sich die Eigenschaften einer Ressource zu ihrem Ursprung nachverfolgen lassen, bietet die Klasse `SemanticRecord::Property` die Methode `context` (Zeile 9+10), die Informationen darüber bereitstellt, aus welcher Datenquelle die derzeit betrachtete Information stammt. Natürlich kann eine Information auch aus mehreren Datenquellen bezogen werden, dann kann dieses Wissen wiederum dazu verwendet werden, um die *Echtheit* einer Information zu bestimmen. Zusätzlich lassen sich über das `Property`-Objekt auch elementare Informationen wie z.B. `domain` und `range` eines Prädikats abfragen.

Der Zugriff auf die Metainformationen der Eigenschaften eines Objekts bietet bereits eine Reihe von neuen Möglichkeiten in der Verarbeitung und der Repräsentation von Ressourcen. Die wesentliche Neuerung im Umgang mit Objekten bezieht sich allerdings auf die Manipulation der Metainformationen.

Relationale Systeme erlauben nur den Zugriff auf die eigentlichen Daten eines Modells, ggfs. lassen sich auch noch einige Informationen des zugrunde liegenden Schemas abrufen, aber eine Veränderung des Schemas ist nicht möglich. Dabei ist die Abwesenheit einer solchen Funktionalität nicht technisch bedingt, denn modere Data Definition Languages (DDL) wie SQL bieten sehr wohl auch die Möglichkeit, die Struktur eines Datenbankschemas auch zur Laufzeit zu verändern. Das grundsätzliche Problem ist daher vielmehr konzeptueller Natur.

Wie in Abschnitt 3.2 dargestellt besteht in relationalen Datenmodellen eine Trennung zwischen den Daten und der Struktur der Daten. Diese Disparität hat dazu geführt, dass Anwendungen stark in Hinblick auf das zugrunde liegende Datenschema entwickelt wurden. Per se ist diese Verknüpfung zwischen Datenschema und Anwendungslogik nicht als negativ zu bewerten, da a) die Modellierung der Schemata in den meisten Fällen im gleichen Kontext wie die Entwicklung der Anwendung stattfindet, b) eine Anwendung in den meisten Fällen nur auf einem Schema beruht und c) die Änderungsrate eines relationalen Datenschemas gering ausfällt, zumindest unter der Voraussetzung, dass die Modellierung gewissenhaft und vollständig erfolgte. Hieraus folgt die Annahme, dass Veränderungen des Datenschemas kontrolliert und nicht spontan eintreten und somit die Anpassung der Anwendungslogik zu einem Teil des Migrationsprozesses des Schemas werden kann.

Für semantische Anwendungen gelten jedoch andere Prämissen, zum einen basieren die Anwendungen in der Regel auf mehreren extern definierten Schemata und zum anderen können sich gerade diese extern angebundenen Schemata in wesentlich fluktuativeren Zuständen befinden als es für relationale Schemata üblich ist. Das heißt, Veränderungen des Datenschemas von semantischen Anwendungen treten unkontrolliert und spontan ein. Besteht nun eine starre Abhängigkeitsbeziehung zwischen Datenschema und Anwendungslogik, können Veränderungen die Funktionsfähigkeit der Anwendung und die Konsistenz der generierten Daten beeinträchtigen. In semantischen Modellen existiert keine Trennung zwischen Schema und Daten. Der nächste Schritt ist also die Entkopplung der Anwendungslogik von der Datenstruktur.

In Abbildung 5.4 ist exemplarisch eine Anwendung dargestellt, die zur Verwaltung von Kontakten dienen soll. Neben den üblichen Informationen wie Name, Anschrift, Telefon usw. werden auch Informationen über die sozialen Kontakte einer Person erfasst. Um die Bedienung zu vereinfachen, werden bei der

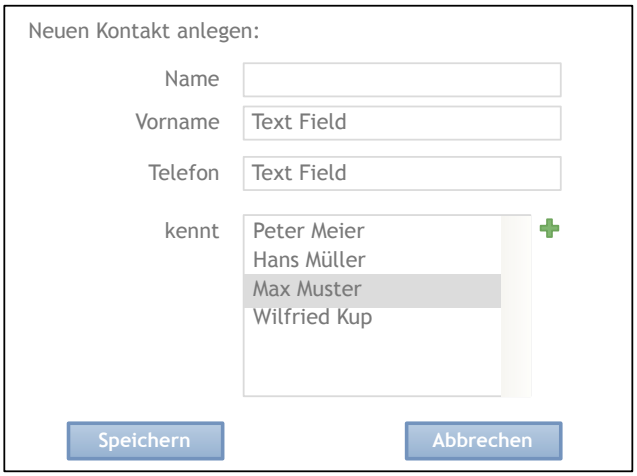

Abbildung 5.4.: Beispielhafte Oberfläche eines Formulars

Erzeugung dieser Beziehungen bereits alle im System vorhandenen Personen als Liste dargestellt, aus der die entsprechenden Kontakte ausgewählt werden können. Auf der Seite der Anwendungslogik bedeutet dies, dass für die Belegung eines Attributs `knows` eine Liste aller im System bekannten `Person`en bereitgestellt werden muss. Diese Information über diese Verknüpfung wird aber nicht automatisch hergestellt, sondern wird starr im Kode der Anwendung hinterlegt. Wenn die Semantik der Beziehung `knows` jedoch im Nachhinein verändert wird, z.B. wenn nur firmeninterne Kontakte abgebildet werden sollen, im System aber auch Kunden als Personen modelliert sind, muss, um die Konsistenz der Anwendung wiederherzustellen, der Anwendungskode angepasst werden. Da über SemanticRecord neben den Attributen auch deren Metainformationen zugänglich sind, lassen sich Anwendungen entwickeln, die auf eine starre Verbindung zwischen Anwendungskode und Datenschema verzichten können. Listing 5.3 zeigt in den ersten Zeilen wie die Informationen für das Formular in einer klassischen Rails-Anwendung bereitgestellt würden. In Zeile 5 wird die gleiche Aufgabe durchgeführt, allerdings wird bei der Verwendung von SemanticRecord das Schema mitbetrachtet und für das Formular die Instanzen des Typs ausgewählt, welcher in der Definition des Prädikats `knows` spezifiziert wurde. Ändert sich nun die Schemadefinition, wird die Veränderung direkt im Kode reflektiert und Anwendung und Schema sind und bleiben konsistent.

```
1 # relational rails way:
2 options_for_select = Person.find(:all)
3
4 # semantic way:
5 options_for_select = @person.knows.range.find(:all)
```

Listing 5.3: ActiveRecord und SemanticRecord im Vergleich

Durch die lose Kopplung zwischen Anwendung und Schema, die mit SemanticRecord erreicht, wird kommt in den Anwendungen ein wesentlich höherer Teil an generischem Kode zum Einsatz. Hierdurch wird auf der einen Seite die Wartbarkeit der Anwendung verbessert und auf der anderen Seite lassen sich somit zusätzliche Entwicklungsschritte automatisieren, was die Entwicklungsdauer positiv beeinflusst.

Ein weiteres Anwendungsgebiet von SemanticRecord, das sich hierdurch erschließt, ist die Entwicklung von Prototypen. Ähnlich wie in der Arbeit von Schwabe und Nunes [62, 63], die mit SHMD eine Methode vorstellten, die es ermöglicht *rapid prototyps* auf Basis semantischer Modelle zu generieren. Mit Hilfe von SemanticRecord lassen durch die Integration von Schemainformationen auch Anwendungen realisieren, die sich *aus sich heraus* weiterentwickeln lassen.

5.4. Benchmarking

5.4.1. Vergleich zweier Implementierungsalternativen

In Abschnitt 5.2.1 wurde die Entscheidung für die TransitModel Komponente dargelegt. Insbesondere war der Aspekt der Zugriffsgeschwindigkeit ausschlaggebend für die Einführung dieser Komponente. Tests mit ActiveRDF haben gezeigt, dass das *lazy-loading* Paradigma zwar die Flexibilität erhöht, die Geschwindigkeit jedoch dazu unverhältnismäßig einschränkt. Im folgenden werden zwei unterschiedliche Implementierungen von SemanticRecord, eine ohne und die andere mit TransitModel, gegeneinander gestellt. Als Grundlage für diesen Laufzeittest dienen die RSpec-Modul-Spezifikationen [64], mit denen das Verhalten von SemanticRecord getestet wird.

- Auffinden spezifischer Ressourcen auf Basis ihrer URI
- Auffinden von Ressourcen einer bestimmten Klasse
- Zugriff und Manipulation von Ressourceneingeschaften

Da die SemanticRecord Implementierung keine Unterstützung für multiple Datenquellen und URI *dereferencing*, bietet wurde für die Tests nur eine Da-

tenquelle verwendet. Abbildung 5.5 zeigt ist das Ergebnis der Laufzeituntersuchung. Die entsprechenden Tests wurden 20 Mal in Folge ausgeführt und der Median der ermittelten Werte bestimmt. Bei den verwendeten Demodaten handelt es sich zum einen um eine Ausspielung aus dem *Semantic Media Wiki* [65, 66] der Medieninformatik[7] an der Fachhochschule Köln, dieser Auszug enthält 9735 Tripeln. Der zweite verwendete Datensatz[8] modelliert sehr rudimentär die Domäne Musik und besteht aus 53 Tripeln. Um zu einem Anhaltspunkt zu gelangen wieweit sich der „Standort“ des verwendeten Triplestores auswirkt, wurden die Testdaten einmal auf einer lokalen und einmal auf einer externen Sesame-Instanz vorgehalten.

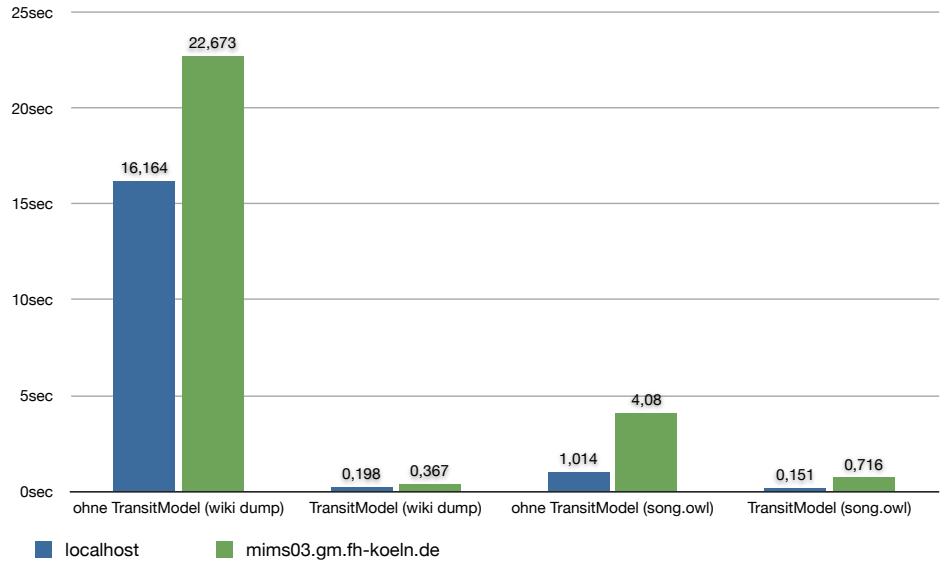

Abbildung 5.5.: Ausführungsdauer der RSpec-Tests von SemanticRecord im Vergleich: lokaler und externer Triplestore, sowie Implmentierung mit und ohne TransitModel

Die Ergebnisse der Untersuchung sind in Abbildung 5.5 dargestellt. Ein signifikanter Ausreißer ergibt sich für SemanticRecord ohne TransitModel in Kombination mit dem Wiki-Datensatz. Hierbei spielt es jedoch keine große Rolle, ob der Datenspeicher lokal oder extern angebunden wurde. Die Ausführung der Testumgebung beträgt hier etwa das 130-fache als der vergleichbare Aufbau mit TransitModel. Die Betrachtung des anderen Datenbestands lässt feststellen, das auch hier die Laufzeit der Implementierung ohne TransitModel über der mit TransitModel liegt; die Ausführungszeit beträgt hier etwa das 5fache, 0.2 Sekunden zu 1 Sekunde. Für alle Testreihen gilt, wie zu erwarten, dass

[7] http://medieninformatik.fh-koeln.de/miwiki/
[8] http://jena.s3.amazonaws.com/ruby/song.owl

die Laufzeit bei der Verwendung des externen Datenspeichers zunimmt, was letztendlich mit der Latenz des Netzes zu erklären ist. Abgesehen von den erweiterten Funktionen des TransitModel trägt diese Komponente im wesentlichen positiv zur Laufzeit von SemanticRecord bei.

5.4.2. Vergleich der Gesamtlaufzeit

Der vorangegangene Abschnitt untersuchte das verallgemeinerte Laufzeitverhalten über eine Reihe von typischen Operationen. Aus Gründen der Vergleichbarkeit mit der Kompagnon-Implementierung ohne Transit Modell wurde dem Aspekt der Verteiltheit der Datenquellen keine Rechnung getragen. Da es allerdings ein wesentliches Charakteristikum von semantischen Anwendungen, ist das Wissen dezentral zu organisieren, soll auch dieser Aspekt in Hinblick auf die Laufzeit beleuchtet werden.

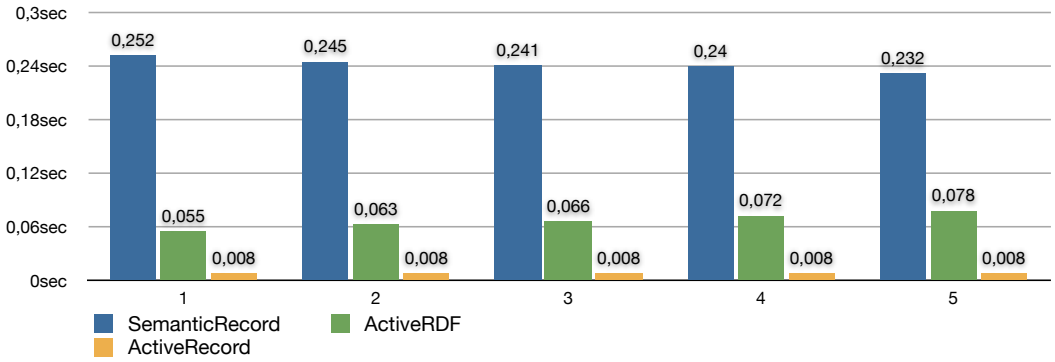

Abbildung 5.6.: Laufzeitenvergleich für das Finden von 155 Instanzen verteilt auf 1-5 Datenspeicher

Als Datengrundlage für diese Untersuchung wird wie zuvor ein Auszug aus dem Semantic Media Wiki der Medieninformatik verwendet. Dieser Datenbestand wird sukzessive gleichmäßig von einem Datenspeicher auf fünf verteilt. Die Laufzeit wird über die `find`-Methode bestimmt, insgesamt sind im Datenbestand 155 Individuen des Typs *Student* enthalten, es wird die Zeit gemessen bis alle Individuen gefunden sind und als Objekt zurückgegeben wurden. Die Ergebnisse dieser Untersuchung sind in Abbildung 5.6 dargestellt, hier zeigt sich jedoch deutlich, das bezüglich der Laufzeit SemanticRecord noch deutlich hinter ActiveRDF zurücksteht, der Verhältnis zwischen beiden Implementierung ist ungefähr 1 zu 4. Bemerkenswert ist allerdings, dass die Laufzeit vom SemanticRecord konstant abnimmt, während die Laufzeit von ActiveRDF hingegen bei zunehmender Verteilung des Datenbestands ansteigt. Dennoch liegt

die die Laufzeit bedeutend unter der von SemanticRecord, in umfangreicheren Tests müssten nun die Ursachen herausgearbeitet werden, um passende Optimierungsarbeiten durchzuführen. Als Referenzwert wurde auch die Laufzeit von ActiveRecord angegeben, die benötigt wird, um 155 Instanzen aus einer Datenbank zu initiieren: 0,008 Sekunden.

5.5. Beispielanwendung

Die vorherigen Abschnitte haben sich theoretisch mit der Erweiterung SemanticRecord auseinandergesetzt. Im folgenden Abschnitt wird beispielhaft eine semantische Webanwendung implementiert, um die Möglichkeiten, die SemanticRecord bietet, direkt im Kontext darzustellen. Die Beispielanwendung wird mit Hilfe des Webframeworks Ruby on Rails entwickelt und ist somit grundsätzlich nach dem MVC-Paradigma aufgebaut. In diesem Fall wird das **M**odell der Anwendung durch SemanticRecord abstrahiert in Abbildung 5.7 wird die Verbindung zwischen semantischem Modell, SemanticRecord und der Anwendung schematisch dargestellt.

Auf die explizite Darstellung der Anwendung wird im Rahmen dieser Arbeit verzichtet. Weitergehende Informationen können über die Projektseite[9] abgerufen werden.

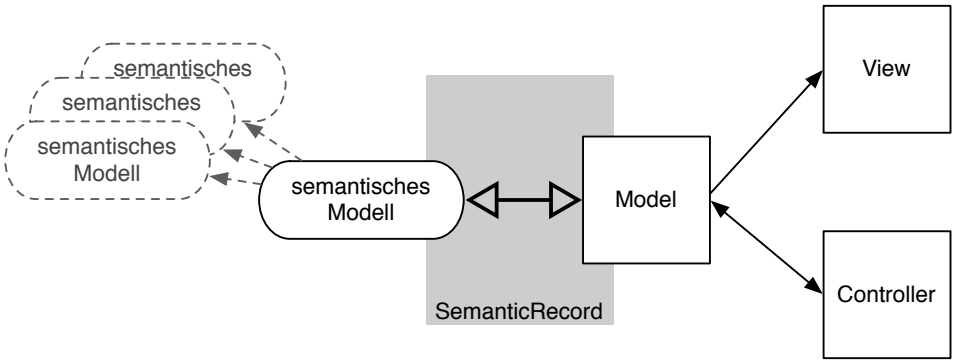

Abbildung 5.7.: Vereinfachte Darstellung der Verortung von SemanticRecord im MVC-Paradigma.

[9] http://semanticrecord.aufnahme.com

5.5.1. Anwendungsszenario

Da es sich bei SemanticRecord um eine Softwarekomponente handelt, die in erster Linie die Entwicklung von semantischen Anwendungen vereinfachen soll, handelt es sich bei dem Stakeholder des Anwendungsszenarios um einen Softwareentwickler. Es soll untersucht werden, ob eine skizzierte Anwendung durch den Einsatz von SemanticRecord realisiert werden kann.

> Die Medieninformatik der FH Köln möchte auf ihrer Internetpräsenz gerne eine Liste der Projekte veröffentlichen, um Studieninteressierten einen besseren Eindruck davon geben zu können, was einen beim Studium der Medieninformatik ungefähr erwartet.
>
> Da im Semantic Media Wiki bereits die aktuellen und abgeschlossenen Projekte von den Studierenden dokumentiert werden, sollen diese Informationen für die Webseite weiterverwendet werden, um doppelte oder inkonsistente Einträge zu vermeiden, trotzdem soll es möglich sein, auch manuell über eine Eingabemaske neue Projekte zu erstellen. Eine der Kernfunktionen der neuen Projektliste soll aber die Verknüpfung der Projekte mit den von der ACM spezifizierten Forschungsgebieten sein, um im Kontext eines Projekts Verweise auf andere Projekte oder Forschungsarbeiten geben zu können, die im gleichen Forschungsgebiet angesiedelt sind. Um die Verknüpfung zu anderen Arbeiten herzustellen, sollen die Projektinformationen von anderen Hochschulen verwendet werden, die ihre Daten ebenfalls im Sinne von *Linked Data* veröffentlichen.

In dem beschriebenen Anwendungsszenario sind die *high-level* Designziele von SemanticRecord kodiert:

Verteiltheit Um die gewünschten Funktionen zu realisieren, müssen Daten aus mehreren Quellen aggregiert werden. Projekte aus dem SMW, Wissen über die Forschungsgebietskategorisierung der ACM und Wissen über Projekte an anderen Hochschulen. (vgl. Abbildung 5.8)

Dynamik Jede der verwendeten Datenquellen kann sich zu jeder Zeit verändern, dies gilt insbesondere für neue Projekte oder Metainformationen, die manuell hinzugefügt werden

Schemaeinflüsse Da die verwendeten Daten ihre Schemabeschreibungen direkt mitliefern, sollten diese auch verwendet werden um schema-agnostischen

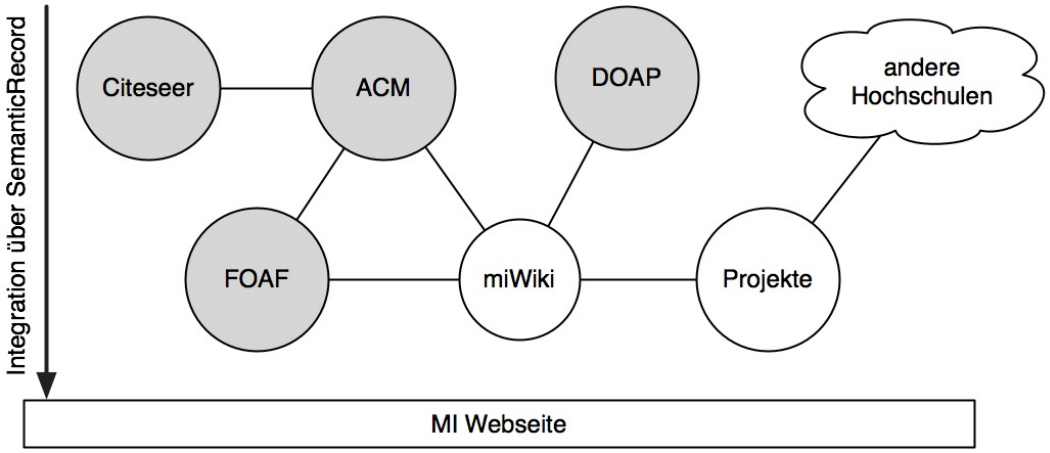

Abbildung 5.8.: Für die Anwendung werden mehrere Datenquellen verwendet. Grau hinterlegte Datenquellen gehören zum Linked Data Netz, Knoten ohne Hintergrund entsprechen den Tripelspeichern der Medieninformatik, abstrahiert wurden alle weiteren Datenquellen anderer Hochschulen dargestellt. SemanticRecord kapselt die verteilten Datenquellen und stellt sie dem Anwendungsentwickler über eine wohldefinierte API zur Verfügung.

Anwendungskode zu entwickeln und dem Benutzer Metainformationen über die betrachteten Ressourcen zur Verfügung zu stellen.

5.5.2. User Stories

User Stories sind in erster Linie den Methoden der agilen Softwareentwicklung zuzuordnen und dienen zwar ähnlichen Zwecken wie *Usecases*, sind aber wesentlich strukturierter aufgebaut und folgen festen Regeln. User Stories dienen dem exakten Festhalten über den aktuellen Zustand des Systems, die ausgeführten Aktionen und deren Resultat in Bezug auf einzelne Funktionen der Anwendung. Das Nutzungsszenario aus 5.5.1 lässt sich grob in vier Teiltätigkeiten aufbrechen: Anzeigen aller Projekte, Einzelansicht eines Projekts, Bearbeitung eines Projekts und Anlegen eines neuen Projekts. Die Teiltätigkeiten können zudem noch auf zwei unterschiedliche Nutzertypen verteilt werden. Die Funktionen zur reinen Ansicht von Projekten werden in der Regel durch die „normalen“ Benutzer wahrgenommen, die bearbeitenden Funktionen jedoch vom Administrator der Seite. Die hier verwendeten von User Stories richten sich nach dem *Behavior Driven Development*-Ansatz. [64]. Das Besondere an diesen User Stories ist, dass mit Hilfe vom Cucumber[10] die User Stories für

[10]http://cukes.info

automatisierte Softwaretests verwendet werden können.

```
In order to get a information
As a user
I want to see an products

  Scenario: View overview
    Given I am on the main page
    And the system contains 3 projects
    When I click on "Projects"
    Then I should see a table with 3 row

  Scenario: Inspect project
    Given I am on the products page
    And a project with "Semantic␣Web" title
    When I click on the link "Semantic␣Web"
    Then I should be taken to the detailed page
    And I should see "Semantic␣Web"
```

Listing 5.4: User Story: Projektübersicht und Detailansicht

Die Benutzergeschichte aus 5.4 beschreibt die Tätigkeiten eines gewöhnlichen Benutzers auf der Suche nach Informationen. Im ersten Szenario wird das Aufrufen der Übersichtsseite beschrieben und das zweite Szenario beschreibt die Detailansicht eines Projekts.

```
In order to manage projects
As a admin
I want to edit and create projects

  Scenario: Edit project
    Given I am logged in
    And a Project with "Metaprogramming" rdfs:label
    When I fill in rdfs:label with "Metaprogramming␣with␣
       Ruby"
    And I click on "Save␣changes"
    Then I should see "Metaprogramming␣with␣Ruby"

  Scenario: New project
    Given I am logged in
    And I am on the new project page
    When I fill in rdfs:label with "Semantische␣
       Modellierung"
    And I press "submit"
    Then the project should be saved
    And I should be taken to the projects page
```

Listing 5.5: User Story: Verwaltung von Projekten

In 5.4 werden schließlich die Tätigkeiten des Administrators beschrieben. Zum einen das Bearbeiten von bestehenden Projekten und das Erstellen von neuen Projekten. Die User Stories beschreiben die Benutzung und die Anwendung zwar auf einer abstrakten Ebene, aber es werden genug Details abgebildet, um die Implementierung zu unterstützen. Die spezifischen Ausprägungen einzelner Funktionen sollten allerdings mit Hilfe von RSpec- oder Unittests[11] modelliert werden. Im Rahmen dieser Arbeit wird aufgrund des prototypischen Charakters hierauf verzichtet. Im Anhang finden sich noch weitere User Stories, die den Rahmen für die hier Dargestellten beschreiben.

5.5.3. Umsetzung

Modell

Die Anwendung zur Verwaltung der Projekte besteht aus nur einem Modell, das den zu verwaltenden Projekten entspricht. In Listing 5.6 ist die Definition des Modells dargestellt. Wie unschwer zu erkennen ist, besteht die reine Definition aus nur wenigen Zeilen. Die Klasse leitet von `SemanticRecord::Base` ab und definiert mit der setter-Methode `base=` die URI, die in Kombination mit dem Namen der Klasse, die URI ergibt, welche die Klasse im semantischen Modell beschreibt. Diese kann alternativ auch direkt über `uri=` spezifiziert werden, falls die Namen voneinander abweichend sind. Die Klasse `Project` stellt nun eine Schnittstelle zum verteilten semantischen Modell der Anwendung bereit, hierüber können die Individuen der Klasse als Ruby-Objekte instanziert werden, die dann im weiteren Verlauf von der Anwendung manipuliert werden.

```
1 class Project < SemanticRecord::Base
2   self.base = 'http://mims04.gm.fh-koeln.de'
3
4   def mark_as_done
5     # do stuff
6   end
7 end
```

Listing 5.6: Projekt Modell der Beispielanwendung

[11] Dies gilt für Ruby-Projekte, ähnliche Test-Frameworks existieren aber auch für viele andere Programmiersprachen

Für den Fall, in dem das semantische Modell noch um eigenes Verhalten erweitert werden soll, lassen sich der Definition noch weitere Methoden hinzufügen, wie beispielsweise in Zeile 4-6 durch den Funktionsrumpf angedeutet wird. Es ist nicht notwendig, alle für alle Klassen des semantischen Modells, die in der Anwendung verwendet werden, auch eigene Ruby-Klassen zu erstellen. Die entsprechenden Bezüge könnten auch „von Hand“ über Sparql-Anfragen hergestellt werden. Für die essentiellen Klassen der Anwendung ist dies jedoch nicht anzuraten, da so auf viele der Abstraktionen von SemanticRecord verzichtet werden muss und sich keine spezifischen Methoden auf den Klassen definieren lassen.

Controller

Der Controller stellt nach MVC die „Schaltzentrale“ der Anwendung dar, hier werden Model und View miteinander verknüpft. Für die Beispielanwendung kommt, um den Namenkonventionen von Rails gerecht zu werden, der `ProjectsController` zum Einsatz. Da die Anwendung nach dem RESTful Paradigma gestaltet wird, finden sich hier in der Regel nur die typischen *CRUD*-Methoden.

```
1  class ProjectsController < ApplicationController
2
3    def index
4      @projects = Project.find
5    end
6
7    def create
8      @project = Project.new(params[:uri])
9      params.each do |param,value|
10       @project.send(:param,value)
11     end
12
13     respond_to do |format|
14       if @project.save
15         format.html
16         format.js
17       else
18         redirect_to :index
19       end
20     end
```

Listing 5.7: ProjectsController

Listing 5.7 zeigt einen Ausschnitt der `ProjectsController`-Klasse, welcher die Methoden `index` und `create` darstellt. Die `index`-Methode dient zur Vorbereitung der entsprechenden *index*-View, in der alle Projekte dargestellt werden. Hier werden mit `Project.find` die Projekte aus dem Modell abgerufen und der View über die globale Variable `@projects` bereitgestellt. Der Aufruf von `create` wird durch einen `POST`-Request ausgelöst und initiiert das Erstellen eines neuen Project-Objekts. Im `params`-Hash sind die übergebenden Variablen enthalten, die nun dem neu erstellten Objekt zugeordnet werden, im Anschluss wird das Objekt gespeichert und damit in die entsprechenden Triplestores übertragen.

Views

Die View beschreibt die Schnittstelle zum Benutzer der Anwendung. Da dies somit der Punkt ist, an dem die Welt des Entwicklers auf die des Benutzers trifft, handelt es sich auch um eine Art „Problemknoten“. Vielfach existiert die ungerechtfertigte Meinung, dass semantische Anwendungen im Gegensatz zu konventionellen aufgrund der Semantik doch auch andersartige Oberflächen besitzen müssten, denen man die Semantik direkt ansehen könne. Oft wird gefragt: „Und wo sehe ich jetzt die Semantik?“ Die Antwort auf diese Frage ist trivial und gegebenenfalls auch enttäuschend, aber die Semantik lässt sich in den seltensten Fällen einer Anwendung „ansehen“.

Die Vorteile, die semantische Anwendungen bieten, sind in der Regel nicht sichtbar, sondern verbergen sich im Inneren und beeinflussen das Verhalten der Anwendung. Und das Verhalten der Anwendung wird nicht unwesentlich von den zugrunde liegenden Daten bestimmt, im Falle von semantischen Anwendungen ein verteiltes, verknüpftes und dynamischen Datenmodell, welches sich im Vergleich zu relationalen Datenmodellen eher dazu eignet, Anwendungen zu entwickeln, die dem Benutzer nicht „dumm“ vorkommen. Es gelten also zum einen die üblichen Prinzipien und Regeln des Usability-Engeneering aber auch der intelligente Umgang mit den bereitgestellten Daten.

6. Fazit

In dieser Arbeit wurde die mögliche Integration von Ontologien als verteilte und dynamische Datenmodelle in objektorientierten Programmiersprachen behandelt. Bei der Untersuchung bereits bestehender Ansätze zeigte sich deutlich, dass alle Ansätze mit mindestens einer von drei konzeptionellen Schwierigkeiten zu kämpfen haben.

1. Es wird bei der Betrachtung des Problemraums keine Unterscheidung zwischen *application* und *domain* Modell vorgenommen. Das bedeutet, dass neben dem Datenmodell auch versucht wird, Anwendungsklassen aus dem semantischen Modell abzuleiten. Da es sich bei Ontologien allerdings um rein deskriptive Modelle handelt, wenn neueste Ansätze wie z.B. [67] ausgeklammert werden, ist dies eine Anforderung, gar nicht erfüllt werden kann.

2. Bei einer Reihe von Ansätzen wird zwar eine Integration als Datenmodell erreicht, auf konzeptioneller Ebene wird hier jedoch lediglich eine monolithische relationale Datenbank durch einen monolithischen Triplestore ersetzt. So wird zwar der programmatische und objektorientierte Zugriff auf ein semantisches Datenmodell erleichtert, aber der wesentliche Aspekt, dass ein semantisches Modell eine verteilte und dynamische Wissensrepräsentation darstellt, ausgeklammert.

3. Einer Vielzahl der Ansätze gelingt es nicht, den Paradigmenwechsel zwischen relationalen und semantischen Datenmodell transparent durchzuführen. Der Kernaspekt, der beide Paradigmen unterscheidet, ist die Verknüpfung von Schema und Daten in einem Modell. Die vorliegenden Implementierungen reduzieren die Informationstiefe des Modells aber dahingehend, dass auf Schemainformationen nicht programmatisch zugegriffen werden kann, was dazu führt, dass die vorhandene Differenzierung beider Paradigmen wirkungslos wird.

Als ein Ansatz, der die genannten Anforderungen Dynamik, Verteiltheit und Schemaeinflüsse zusammenbringt, wurde die Erweiterung *SemanticRecord* konzipiert und implementiert, welche die Programmiersprache Ruby dahingehend erweitert, den Zugriff auf semantische Modelle durch ein Framework zu abstrahieren und zu vereinfachen. Eine erste Verifikation der Softwarekomponente wurde bereits durch die Implementierung eines Anwendungsszenarios durchgeführt und kommt zu dem Ergebnis, dass *SemanticRecord* die Anforderungen erfüllt um Ontologien als dynamische und verteilte Datenmodelle im Programmiersprachen zu integrieren. Dieses Ergebnis gilt beschränkt auf den Fall der Programmiersprache Ruby. Implementierungen des zugrunde liegenden Modells in anderen Sprachen wie Python oder Java könnten in einer anschließenden Arbeit durchgeführt werden, um so eine ganzheitliche Validierung des formalen Modells zu ermöglichen oder nötige Modifikationen und Beschränkungen zu identifizieren.

Die derzeitige Implementierung von *SemanticRecord* lässt sich trotz des grundsätzlich prototypischen Charakters bereits in weiteren Projekten einsetzen, beispielsweise wird in [68] ein Empfehlungs-Framework konzipiert, welches SemanticRecord verwendet. Aufgrund der Modularchitektur ließen sich in weiteren Arbeiten einzelne Aspekte herausgreifen und weiterentwickeln. Am augenscheinlich dringendsten erscheint die Entwicklung einer Reasoning-Komponente für das Transit Model, um das Datenmodell um implizite Aussagen zu erweitern. Ebenso sollte die Implementierung auf mögliche Optimierungspotentiale hinsichtlich der Performance untersucht werden.

SemanticRecord wurde als Werkzeug für spezifische Problemszenarien entwickelt, dies sind in erster Linie Rapid Prototyping und die Entwicklung von generischen schemaungebundenen Anwendungen. Es ist wichtig, am Ende dieser Arbeit festzuhalten, dass Anwendungskontexte existieren, in denen ein Zurückgreifen auf klassische Entwicklungsmethoden, Frameworks und Werkzeuge durchaus sinnvoll und notwendig ist. Von *SemanticRecord* werden die Fälle adressiert, in denen der Einsatz von semantischen Modellen einen Mehrwert bieten, welcher bislang aber nicht wahrgenommen werden konnte, da die technischen Voraussetzungen nicht gegeben waren. Interessanterweise hat sich auch gezeigt, dass SemanticRecord neben relationalen Datenbanken auch sehr viele Anwendungsgebiete der dokument-orientierten Datenbanken, wie Couch DB oder Mongo DB, adressiert. Dies ist ein Themenkomplex der ebenfalls in weiteren Arbeiten untersucht werden sollte.

Literaturverzeichnis

[1] Tim Berners-Lee, James Hendler und Ora Lassila. The semantic Web. *Scientific American*, 2001. `http://www.sciam.com/article.cfm?id=the-semantic-web` . Abruf: 01.03.2010 .

[2] Chris Bizer und Andy Seaborne. D2RQ-treating non-RDF databases as virtual RDF graphs. *Proceedings of the 3rd International Semantic Web Conference*, 2004.

[3] Christian Bizer, Tom Heath, Danny Ayers und Yves Raimond. Interlinking open data on the web. *Proceedings of the 4th European Semantic Web Conference*, 2007.

[4] Christian Bizer, Tom Heath und Tim Berners-Lee. Linked data–the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[5] Jorge Cardoso. The semantic web vision: Where are we? *IEEE Intelligent systems*, 22(5):84–88, 2007.

[6] IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE*, Std 610.12, 1990.

[7] Bernhard Lahres und Gregor Raýman. *Praxisbuch Objektorientierung: von den Grundlagen zur Umsetzung*. Gallileo Press, 2006.

[8] Dean Allemang und James A. Hendler. Semantic web for the working ontologist: modeling in RDF, RDFS and OWL. *Morgan Kaufmann*, 2008.

[9] Patrick Grässle, Henriette Baumann und Philippe Baumann. UML 2.0 projektorientiert. *Gallileo Press*, 2004.

[10] Ole-Johan Dahl, Edsger Wybe Dijkstra und Charles Antony Richard Hoare. *Structured programming*. Academic Press, 1972.

[11] Trygve Reenskaug. The original MVC reports. 1979.

[12] David Thomas, David Heinemeier Hansson und Leon Breedt. *Agile web development with rails*. Pragmatic Bookshelf, 2006.

[13] Joelle Coutaz. PAC: an object oriented model for implementing user interfaces. *ACM SIGCHI Bulletin*, 19(2):37–41, 1987.

[14] Ansgar Scherp und Susanne Boll. *Framework-Entwurf*. Handbuch der Software-Architektur. dpunkt.verlag, 2006. 397–419 pp.

[15] Dirk Breuer. Konzeption, prototypische Realisierung und szenariobasierte Validierung einer dienstorientierten Multimediaarchitektur. 2009.

[16] Dirk Breuer, Sebastian Conen und Mathias Richter. Konzeption eines verteilten, dienst-orienterten Rahmenwerkes für Medienverarbeitung. *Institutsbericht. Fakultät für Informatik und Ingenieurwissenschaften, FH Köln*, 2008.

[17] Ralph Johnson und Brian Foote. Designing reusable classes. *Journal of object-oriented programming*, 1988.

[18] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. Addison-Wesley Professional, 2000.

[19] Tim Berners-Lee. Stack of expressive power. `http://www.w3.org/2003/Talks/0922-rsoc-tbl/slide30-0.html` . Abruf: 01.03.2010 .

[20] Aurona Gerber, Alta van der Merwe und Andries Barnard. A functional semantic web architecture. In *Lecture Notes in Computer Science*, volume 5021, Seiten 273–287, 2008.

[21] Harry Halpin. Sense and Reference on the Web. 2009.

[22] Mark Bartel, John Boyer, Barb Fox, Brian LaMacchia und Ed Simon. XML Signature Syntax and Processing (Second Edition). *W3C Recommendation*, 2008. `http://www.w3.org/TR/xmldsig-core/` . Abruf: 01.03.2010 .

[23] Takeshi Imamura, Blair Dillaway und Ed Simon. XML Encryption Syntax and Processing. *W3C Recommendation*, 2002. `http://www.w3.org/TR/xmlenc-core/` . Abruf: 01.03.2010 .

[24] Sebastian Schaffert. Semantic social software: Semantically enabled social software or socially enabled semantic web. *Proceedings of the SEMANTICS 2006 conference*, Seiten 99–112, 2006.

[25] Sebastian Schaffert, Andreas Gruber und Rupert Westenthaler. A semantic wiki for collaborative knowledge formation. 2006.

[26] Toby Segaran, Colin Evans und Jamie Taylor. *Programming the Semantic Web*. O'Reilly, 2009.

[27] Graham Klyne und Jeremy Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. *W3C Recommendation*, 2004. `http://www.w3.org/TR/rdf-concepts/` . Abruf: 01.03.2010 .

[28] Dave Beckett und Brian McBride. RDF/XML Syntax Specification (Revised). *W3C Recommendation*, 2004. `http://www.w3.org/TR/rdf-syntax-grammar/` . Abruf: 01.03.2010 .

[29] Tim Berners-Lee. Notation 3 - An readable language for data on the Web. *W3C Design Issues*, 2006. `http://www.w3.org/DesignIssues/Notation3` . Abruf: 01.03.2010 .

[30] Jan Grant, Dave Beckett und Brian McBride. RDF Test Cases. *W3C Recommendation*, 2004. `http://www.w3.org/TR/rdf-testcases/%23ntriples` . Abruf: 01.03.2010 .

[31] Ben Adida und Mark Birbeck. RDFa Primer - Bridging the Human and Data Webs. *W3C Editors' Draft*, 2008. `http://www.w3.org/TR/xhtml-rdfa-primer/` . Abruf: 01.03.2010 .

[32] Rohit Khare und Tantek Çelik. Microformats: a pragmatic path to the semantic web. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, Seiten 865–866. ACM, 2006.

[33] John Allsop. *Microformats: empowering your markup for Web 2.0*. Apress, 2007.

[34] Chris Bizer, Richard Cyganiak und Tom Heath. How to Publish Linked Data on the Web. 2007. `http://www4.wiwiss.fu-berlin.de/bizer/pub/LinkedDataTutorial/` . Abruf: 01.03.2010 .

[35] Thomas Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5:199, 1993.

[36] Dan Brickley, Ramanathan Guha und Brian McBride. RDF Vocabulary Description Language 1.0: RDF Schema. *W3C Recommendation*, 2004. `http://www.w3.org/TR/rdf-schema/` . Abruf: 01.03.2010 .

[37] Jie Bao, Diego Calvanese und Bernardo Cuenca Grau. OWL 2 Web Ontology Language. *W3C Recommendation*, 2009. `http://www.w3.org/TR/2009/REC-owl2-overview-20091027/` . Abruf: 01.03.2010 .

[38] Sebastian Rudolph, Peter F Patel-Schneider, Bijan Parsia, Markus Krotzsch und Pascal Hitzler. OWL 2 Web Ontology Language Primer. *W3C Recommendation*, 2009. `http://www.w3.org/TR/owl2-primer/` . Abruf: 01.03.2010 .

[39] Deborah L McGuinness und Frank van Harmelen. OWL Web Ontology Language. *W3C Recommendation*, 2004. `http://www.w3.org/TR/owl-features/` . Abruf: 01.03.2010 .

[40] Pascal Hitzler, Sebastian Rudolph und Markus Krotzsch. *Foundations of Semantic Web Technologies*. CRC Press, 2009.

[41] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue und Carsten Lutz. OWL 2 Web Ontology Language Profiles. *W3C Recommendation*, 2009. `http://www.w3.org/TR/owl2-profiles/` . Abruf: 01.03.2010 .

[42] Li Ding und Tim Finin. Characterizing the semantic web on the web. In *Proceedings of the International Semantic Web Conference*, 2006.

[43] Christian Bizer, Richard Cyganiak und Tobias Gauß. The RDF Book Mashup: from Web APIs to a web of data. *3rd Workshop on Scripting for the Semantic Web*, 2007.

[44] Matthias Quasthoff und Christoph Meinel. Design Pattern for Object Triple Mapping. In *International Conference on Services Computing*. IEEE Computer Society, 2008.

[45] Peter Bartalos und Mária Bielikova. An approach to object-ontology mapping. *IIT. SRC–Student Research Conference*, 2007.

[46] Kimio Kuramitsu. A Map-based Integration of Ontologies into an Object-Oriented Programming Language. In *Artificial Intelligence in Theory and Practice II*. Springer Boston, 2008.

[47] Denny Vrandecic, Soren Auer, Christian Bizer und Libby Miller. Deep integration of scripting languages and semantic web technologies. *1st International Workshop on Scripting for the Semantic Web*, 135:1613–0073, 2005.

[48] Eyal Oren. *Algorithms and Components for Application Development on the Semantic Web*. PhD thesis, National University of Ireland, Galway, 2007.

[49] Frederic P. Miller, Agnes F. Vandome und John McBrewster. *Class (Computer Science): Object- Oriented Programming, Blueprint, Template, Object (computer Science), Data Type, Concept, Attribute (computing), Source Code, Cohesion (computer Science), Metadata, Metaobject*. Alphascript Publishing, 2009.

[50] Dominic Mainz. *Deep integration of the OWL ontology language into Ruby using metaprogramming*. PhD thesis, Heinrich-Heine-Universität, Düsseldorf, 2009.

[51] Eyal Oren, Benjamin Heitmann und Stefan Decker. ActiveRDF: Embedding Semantic Web data into object-oriented languages. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(3):191–202, 2008.

[52] Arie van Deursen, Paul Klint und Joost Visser. Domain-Specific Languages: An Annotated Bibliography. 2000. `http://homepages.cwi.nl/~arie/papers/dslbib/` . Abruf: 01.03.2010 .

[53] Wernher Behrendt und Georg Günther. Content Islands. *Proceedings of the 5th Int. Conference on Semantic Systems*, 2009.

[54] Olaf Hartig, Christian Bizer und Johann-Christoph Freytag. Executing SPARQL Queries over the Web of Linked Data. In *Proceedings of the 8th International Semantic Web Conference*, 2009.

[55] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, 2003.

[56] Matthias Quasthoff und Christoph Meinel. Semantic Web Admission Free–Obtaining RDF and OWL Data from Application Source Code. *Proceedings of the 4th International Workshop on Semantic Web*, 2008.

[57] Eyal Oren, Armin Haller, Manfred Hauswirth und Benjamin Heitmann. A flexible integration framework for Semantic Web 2.0 applications. *IEEE Software*, 24(5):64–71, 2007.

[58] Kendall Grant Clark, Lee Feigenbaum und Elias Torres. SPARQL Protocol for RDF. *W3C Recommendation*, 2008. `http://www.w3.org/TR/rdf-sparql-protocol/` . Abruf: 01.03.2010 .

[59] Eric Prudhommeaux und Andy Seaborne. SPARQL Query Language for RDF. *W3C Recommendation*, 2008. `http://www.w3.org/TR/rdf-sparql-query/` . Abruf: 01.03.2010 .

[60] Michael Hausenblas. Exploiting Linked Data For Building Web Applications. *IEEE Internet Computing*, Seiten 80–85, 2009.

[61] R Fielding, J Gettys, J Mogul, H Frystyk, L Masinter, P Leach und Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. 1999. `http://www.w3.org/Protocols/rfc2616/rfc2616.html` . Abruf: 01.03.2010 .

[62] Fernanda Lima und Daniel Schwabe. Application modeling for the semantic Web. *First Latin American Web Congress*, 2003.

[63] Demetrius Arraes Nunes und Daniel Schwabe. Rapid prototyping of web applications combining domain specific languages and model driven design. *Proceedings of the 6th international conference on Web engineering*, 2006.

[64] David Chelimski, Dave Astels, Bryan Helmkamp, Zach Dennis, Dan North und Aslak Hellesoy. *The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends*. Pragmatic Bookshelf, 2009.

[65] Sebastian Schaffert, Francois Bry, Joachim Baumeister und Malte Kiesel. Semantic Wiki. *Informatik-Spektrum*, 2007.

[66] Max Völkel, Markus Krötzsch, Denny Vrandecic und Heiko Haller. Semantic Wikipedia. *Proceedings of the 15th international conference on World Wide Web*, 2006.

[67] Bernhard Schandl. Functions over RDF Language Elements. *Proceedings of the 8th International Semantic Web Conference*, 2009.

[68] Stephan Pavlovic. Konzeption, prototypische Realisierung und Evaluation einer Frameworkarchitektur für ontologiebasierten Empfehlungssysteme. 2010.

A. Zusätzliche Listings

Hier finden sich die Listings der wichtigsten Komponenten von SemanticRecord, die aufgrund ihres Umfangs nicht im laufenden Text untergebracht wurden. Der komplette Quellcode ist im Internet unter:

`http://github.com/kangguru/semantic_record`

veröffentlicht.

Listing A.1: SemanticRecord Base

```
1  require 'rubygems'
2  require 'ruby-sesame'
3  require 'json'
4
5  module SemanticRecord
6
7    class Base
8        attr_reader :uri
9
10       class << self
11         attr_accessor :base, :connection,:rdf_type,:uri
12       end
13
14       def initialize(uri)
15         @uri = uri
16         @presaved_attributes = {}
17
18         TripleManager.describe(uri)
19
20         if self.new_record?
21          self.rdf_type = self.class
22         end
23       end
24
25       def new_record?
26         exists = TripleManager.exists_as_subject?(self.uri)
27
28         !exists
29       end
30
31       def type
32         proxy_getter(:rdf_type)
33       end
```

```
34
35        def attribute(p_uri)
36          TripleManager.property_for(uri,p_uri)
37        end
38
39        def attributes
40          TripleManager.properties_for(uri)
41        end
42
43        def destroy!
44           remove =
45           "<transaction>
46␣␣␣␣␣␣␣␣␣␣<remove>
47␣␣␣␣␣␣␣␣␣␣␣␣<uri>#{uri}</uri>
48␣␣␣␣␣␣␣␣␣␣␣␣<null/>
49␣␣␣␣␣␣␣␣␣␣␣␣<null/>
50␣␣␣␣␣␣␣␣␣␣</remove>
51␣␣␣␣␣␣␣␣␣␣</transaction>"
52          connection.add!(remove, "application/x-rdftransaction")
53        end
54
55        def save
56          if new_record?
57            begin
58              @presaved_attributes.each {|predicate,value|
59                value.each do |val|
60                     TripleManager.add(uri,predicate,val)
61                end
62
63                @presaved_attributes.delete(predicate)
64              }
65            rescue
66              return false
67            end
68          else
69            begin
70
71              TripleManager.update(uri,@presaved_attributes)
72              @presaved_attributes = {}
73            rescue ArgumentError
74              puts $!
75              return false
76            end
77          end
78          true
79        end
80
81        def method_missing(mth,*args)
82          if mth.id2name.end_with?("=")
83            proxy_setter(mth,*args)
84          else
85            proxy_getter(mth,*args)
86          end
87        end
```

```
88
89        def self.inherited(receiver)
90          receiver.base = self.base
91          receiver.rdf_type = "http://http://www.w3.org/2000/01/rdf-
                schema#Class"
92        end
93
94        def self.count
95          TripleManager.count
96        end
97
98        def self.rdf_type
99          @rdf_type ||= "http://www.w3.org/2002/07/owl#Thing"
100       end
101
102       def self.uri
103         @uri ||= "#{self.base}#{self}"
104       end
105
106       def self.base
107         @base ||= "http://example.org/"
108       end
109
110       def self.find_by_uri
111
112       end
113
114       def self.find_by_sparql(query)
115         TripleManager.get_by_sparql(query,true)
116       end
117
118       def self.find
119         # if self isn't an inherited form of this
120         # class then return all existing instances
121         if self == SemanticRecord
122           s = "?nil"
123         else
124           uri = URI.parse self.uri #"#{self.base}#{self}"
125           if uri.absolute && uri.path
126             s = "<#{uri.to_s}>"
127           else
128             raise ArgumentError, "base␣uri␣seems␣to␣be␣invalid"
129           end
130         end
131         instances_response = TripleManager.get_subjects(s)
132
133       end
134
135       protected
136
137       def proxy_setter(mth,*args)
138         predicate = mth.to_sym.expand#(mth)
139         @presaved_attributes[predicate] = args.flatten
140       end
```

```
141
142       def proxy_getter(mth,*args)
143
144         predicate = mth.to_sym.expand#(mth)
145
146         if @presaved_attributes.has_key?(predicate)
147            value_response = @presaved_attributes[predicate]
148         else
149            value_response = TripleManager.get_objects(uri,predicate,*
                args)
150         end
151
152       end
153
154     def expand(name)
155         ns, predicate = name.id2name.split("_",2)
156         if predicate.blank?
157           raise Namespaces::NoPredicateError, "no␣valid␣predicate␣
                defined"
158         end
159         namespace = Namespaces.resolve(ns)
160
161         predicate = predicate.chomp("=")
162
163         return namespace + predicate
164       end
165     end
166
167
168 end
```

Listing A.2: TripleManager

```
1  require 'rdf/redland'
2  require 'curb'
3
4  module TripleManager
5
6    @@transit_model = Redland::Model.new( Redland::MemoryStore.new )
7
8    def self.describe(uri)
9      unless exists_as_subject?(uri)
10       count = @@transit_model.size
11       populate_model_with_result( "CONSTRUCT␣{<#{uri}>␣?p␣?o}␣WHERE␣
             {<#{uri}>␣?p␣?o}" )
12       if count == @@transit_model.size
13           populate_model_with_result_from_http(uri)
14       end
15     end
16   end
17
18   def self.property_for(s,p)
19     unless exists_as_subject?( p )
20       populate_model_with_result_from_http( p )
21     end
```

```
22
23      resource = @@transit_model.get_resource( Redland::Uri.new(p) )
24      value = get_objects(s,p)
25
26      return {:value => value,
27              :range => resource.try(:get_property, Redland::Uri.new("
                    http://www.w3.org/2000/01/rdf-schema#range") ),
28              :domain => resource.try(:get_property, Redland::Uri.new("
                    http://www.w3.org/2000/01/rdf-schema#domain") )
29      }
30    end
31
32    def self.properties_for(s)
33      query_string = "SELECT␣?result␣WHERE␣{<#{s}>␣?result␣?object}"
34
35      get_by_sparql(query_string)
36    end
37
38    def self.populate_model_with_result_from_http(uri)
39      curl = Curl::Easy.new(uri)
40      curl.headers["Accept"] = "application/rdf+xml"
41      curl.follow_location = true
42      curl.connect_timeout = 2
43      curl.max_redirects = 5
44      begin
45        body = curl.perform
46        # only process responce if content-type matches
47        if !!(curl.content_type =~ /application\/rdf\+xml/)
48          parser = Redland::Parser.new
49          parser.parse_string_into_model(@@transit_model,curl.body_str,
                  Redland::Uri.new( uri ))
50        end
51      rescue
52
53      end
54    end
55
56    def self.attribute(uri)
57      @@transit_model.get_resource(uri)
58    end
59
60    def self.populate_model_with_result(query)
61      q = query
62      parser = Redland::Parser.new
63      SemanticRecord::Pool.connections.each do |connection|
64        content = connection.socket.query(q,:result_type => RubySesame::
              DATA_TYPES[:RDFXML],:infer => true )
65        parser.parse_string_into_model(@@transit_model,content,Redland::
              Uri.new( "http://example.org/" ))
66      end
67    end
68
69    def self.get_by_sparql(query_string,with_population=false)
70      query = Redland::Query.new(query_string)
```

```
71      result = @@transit_model.query_execute(query)
72
73      if result.size == 0
74        parser = SparqlParser.new
75        query_object = parser.parse(query_string)
76        bindings = "#{query_object.query_part.bindings} ?p ?o"
77        where_clause = query_object.query_part.where.group_graph_pattern
              .text_value.insert(1,"#{bindings}.")
78
79        construct_query = "CONSTRUCT {#{bindings}} WHERE #{where_clause}
              "
80
81        populate_model_with_result(construct_query)
82      end
83      query = Redland::Query.new(query_string)
84      result = @@transit_model.query_execute(query)
85
86
87      returning [] do |res|
88        while !result.finished?
89          value = result.binding_value_by_name("result")
90          if value.resource?
91            res << SemanticRecord::Base.new(value.uri.to_s)
92          else
93            res << value.to_s
94          end
95          result.next
96        end
97      end
98    end
99
100   def self.get_subjects(s)
101     query_string = "SELECT ?result WHERE {?result <http://www.w3.org
          /1999/02/22-rdf-syntax-ns#type> #{s}}"
102     get_by_sparql(query_string,false)
103   end
104
105   def self.get_objects(subject,predicate,*args)
106
107     unless args.empty?
108       filter = "FILTER ( lang(?result) = '#{args[0][:lang]}' )"
109     else
110       filter  = nil
111     end
112
113     query_string = "SELECT ?result WHERE {<#{subject}> <#{predicate}>
          ?result #{filter} }"
114
115     get_by_sparql(query_string)
116   end
117
118   def self.add(subject,predicate,object,context=nil)
119     s,p,o = build(subject,predicate,object)
120     @@transit_model.add(s,p,o,context)
```

```
121   end
122
123   def self.update(subject,attributes)
124     attributes.each do |predicate,objects|
125       sub,pre = build(subject,predicate,nil)
126       @@transit_model.find(sub,pre).each do |removable_statement|
127         @@transit_model.delete_statement(removable_statement)
128       end
129       objects.each do |object|
130         s,p,o = build(subject,predicate,object)
131         @@transit_model.add(s,p,o)
132       end
133     end
134   end
135
136   def self.count
137     @@transit_model.size
138   end
139
140   def self.exists_as_subject?(uri)
141     query = Redland::Query.new("SELECT␣?result␣WHERE␣{<#{uri}>␣?
          property␣?result}␣LIMIT␣1")
142     result = @@transit_model.query_execute(query).size
143     result > 0 ? true : false
144   end
145
146   protected
147
148   def self.build(subject,predicate,object)
149     s = Redland::Resource.new( subject )
150     p = Redland::Resource.new( predicate )
151
152     if object.kind_of?(SemanticRecord) || (object.kind_of?(Class) &&
          object.respond_to?(:uri) )
153       o = Redland::Resource.new( object.uri )
154     else
155       o = object.to_s
156     end
157
158     return s,p,o
159   end
160 end
```

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsgbehörde vorgelegen.

Lars Brillert — Köln, den 10. März 2010