Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Erwin Lang

Analyse von Lösungsansätzen zur semiautomatischen Generierung von RDF-Annotation mit Hilfe von Textmining

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Erwin Lang

Analyse von Lösungsansätzen zur semiautomatischen Generierung von RDF-Annotation mit Hilfe von Textmining

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Neitzke
Zweitgutachter: Prof. Dr. Kai von Luck

Eingereicht am: 13. Oktober 2014

Erwin Lang

Thema der Arbeit
Analyse von Lösungsansätzen zur semiautomatischen Generierung von RDF-Annotation mit Hilfe von Textmining

Stichworte
Semantic Web, RDF, UIMA, Text Mining, Maschinelles Lernen

Kurzzusammenfassung
Das Internet beinhaltet heutzutage immer mehr Informationen. Mit Hilfe des Semantic Webs und semantischen Annotationen können diese Informationen für Computer verständlich gemacht werden. Dies ermöglicht beispielsweise eine semantische Suche und eine einfache Assoziation von Inhalten. Jedoch ist die Generierung von solchen Annotationen von Hand sehr aufwändig. Um diesen Vorgang zu automatisieren gibt es eine Reihe von Ansätzen wie der Mustererkennung oder dem maschinellen Lernen. In dieser Arbeit werden diese Ansätze einander gegenübergestellt und ein Ansatz am Beispiel eines Bloggingportals umgesetzt.

Erwin Lang

Title of the paper
Analysis of methods for semiautomatic Generation of RDF-Annotations with Textmining

Keywords
Semantic Web, RDF, UIMA, Text Mining, Maschine Learning

Abstract
The Internet contains more and more information nowadays. With the Semantic Web and semantic annotations this information can be understood by computers. This allows for semantic search as well as easy association of content. The generation of semantic annoations by hand however requires much effort. To automate this there are methods like pattern matching and machine learning. In this thesis these methods will be compared and one of them will be implemented as an example for a blog.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

1.1. Motivation

In den letzten Jahren ist die Menge der verfügbaren Informationen durch das Internet rasant angestiegen. Doch hat sich nicht nur die Masse an Informationen verändert, durch das Aufkommen des Web 2.0 ändert sich auch die Wahrnehmung durch den Benutzer. Während früher noch starre Dokumente das Internet ausmachten ist heute besonders die Interaktivität und Individualität für den Nutzer von Bedeutung. Ein Benutzer möchte genau die Informationen angeboten kriegen welche für ihn interessant sind. Dennoch basiert die grundlegende Struktur des Internets auf den Dokumenten und nicht auf den eigentlichen Inhalten. Suchmaschinen wie Google sind lediglich in der Lage auf eine Frage hin einen Verweis auf den Ort zu liefern wo sich die Antwort voraussichtlich befindet. Viel sinnvoller wäre es jedoch, wenn eine Suchmaschine gleich selbst die Antwort liefern könnte.

Es erscheint dementsprechend sinnvoll, wenn man eine Struktur anbieten kann welche die Semantik des Internets in maschinenlesbarer Form darstellt. Dies bezeichnet man als Semantic Web. Im Semantic Web werden Inhalte mit semantischen Informationen angereichert, meist in der Form von RDF-Annotationen. Durch den Einsatz des Semantic Webs können Informationen durch Computer verwertet und kombiniert werden. Dadurch lässt sich die Qualität für den Nutzer verbessern, da nur wirklich relevante Inhalte angezeigt werden.

Solche semantischen Netze werden in kleinerem Rahmen bereits vielfach eingesetzt. So hat *BBC.co.uk* ihre Webseite mit umfangreichen semantischen Informationen erweitert, was zu einer Verbesserung von Suche und Navigation für die Nutzer führte. Auch die amerikanische Elektronikkaufhauskette *Best Buy* erreichte durch semantische Annotationen ihrer Produkte etwa 30% mehr Traffic. Gleichzeitig können durch die Verknüpfung der Artikel sinnvolle Kaufempfehlungen mit geringem Aufwand erstellt werden.

Das zentrale Problem des Semantic Webs heute ist allerdings, dass es nicht realistisch ist

das gesamte Internet manuell mit semantischen Informationen anzureichern. Aus diesem Grund ist es notwendig Methoden zu finden um diesen Prozess soweit wie möglich zu automatisieren. In dieser Arbeit sollen verschiedene Ansätze einer solchen automatischen Generierung von semantischen Annotationen analysiert werden.

1.2. Zielsetzung

Im Verlauf dieser Arbeit soll untersucht werden auf welche Weise und in welcher Qualität die Erstellung von RDF-Annotationen für das Semantic Web durch Methoden des Text Minings unterstützt werden kann. Insbesondere soll betrachtet werden ob und unter welchen Umständen eine Automatisierung zur Erzeugung der Annotationen erreicht werden kann und welche Probleme zu lösen sind.

Um eine solche Analyse möglichst praxisnah durchführen zu können wird ein Szenario für ein Bloggingportal in der Domäne der Softwareentwicklung definiert und ein Prototyp für das gewählte Szenario implementiert. In diesem Prototyp sollen neue Blogeinträge hinzugefügt und automatisch annotiert werden. Es ist das Ziel diese Implementierung so allgemein wie möglich zu halten um die Ergebnisse auch auf weitere Domänen übertragen zu können. Die Realisierung und das Testen wird mit Hilfe des *UIMA Frameworks* geschehen, welches in dieser Arbeit nur grob beschrieben wird.

Um die für das Szenario gewählten Anforderungen umsetzen zu können sollen zunächst verschiedene aus der aktuellen Forschung bekannten Varianten zur Generierung von RDF-Annoatationen betrachtet und bewertet werden. In dieser Arbeit sollen einige Ansätze auf Basis von Mustern, sowie maschinellem Lernen, vorgestellt werden. Dabei soll herausgearbeitet werden welcher Ansatz für eine Umsetzung auf Basis der Anforderungen am besten geeignet ist. Nach der Implementierung des Prototyps wird dieser nach diversen festgelegten Kriterien wie Qualität, Automatisierbarkeit und Generalisierbarkeit getestet und bewertet.

Das Hauptaugenmerk dieser Arbeit liegt darauf zu untersuchen ob ein praktikabler Ansatz zur automatischen oder semiautomatischen Annoatation von Webinhalten existiert. Des Weiteren soll der aktuelle Stand der Forschung beschrieben und nachvollzogen werden.

1.3. Gliederung

Die Arbeit ist inklusive der Einleitung in insgesamt 6 Kapitel aufgeteilt. Auf die Einleitung folgen zunächst die Grundlagen der Arbeit in Kapitel 2. Dort soll das Themengebiet näher vorgestellt werden. Es wird auf die Definition, den Nutzen, und den Aufbau des Semantic Webs eingegangen. Des Weiteren wird das Resource Description Framework (RDF) beschrieben, welches für die semantischen Annotationen verwendet wird. Auch wird der Begriff der Ontologie erläutert. Im letzten Teil des Kapitels wird noch auf Text Mining als zugrundeliegende Technik eingegangen.

Die Analyse in Kapitel 3 widmet sich vor allem dem aktuellen Stand der Forschung. Zunächst wird jedoch das Szenario und die Anforderungen für den zu implementierenden Prototypen beschrieben um einen praxisbezogenen Kontext für die Analyse zu liefern. Danach werden typische Probleme im Bezug auf automatische semantische Annotationen dargestellt und die möglichen Lösungen konkret vorgestellt. Als Ergebnis der Analyse soll anhand der Anforderungen ein Ansatz herausgearbeitet werden welcher im Weiteren implementiert wird.

Kapitel 4 beschäftigt sich mit der Realisierung des Prototypen. Es werden die verwendeten Frameworks und Hilfsmittel vorgestellt. Danach wird die Struktur des Projektes erläutert. Der Kern des Kapitels ist die Beschreibung der UIMA-Pipeline welche den gesamten Weg vom Text eines Blogeintrages bis hin zu den fertigen RDF-Annotationen aufzeigt. Hier wird insbesondere auf die verwendeten *Analysis Engines* eingegangen. Im letzten Teil des Kapitels geht es um eine Erweiterung auf Basis der beschriebenen Pipeline welche den Automatisierungsgrad weiter erhöht und das System lernfähig macht.

Im darauf folgenden Kapitel 5 wird der implementierte Prototyp nun anhand der bereits in Kapitel 3 vorgestellten Anforderungen bewertet. Dabei wird vor allen Dingen auf die Qualität der Annotationen geachtet und welche Art von Semantik das System in der Lage ist zu erkennen. Für mögliche Problemfälle werden einige Lösungsansätze skizziert.

Am Ende der Arbeit werden in Kapitel 6 die gewonnen Erkenntnisse zusammengefasst und ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

2. Grundlagen

In diesem Kapitel sollen die Begriffe und Techniken näher erläutert werden welche für das Verständnis dieser Arbeit von Nöten sind. Zunächst wird das Konzept des Semantic Webs näher erklärt sowie RDF und RDFS. Anschließend sollen Ontologien als mögliche Wissensrepräsentationen genauer besprochen werden. Im letzten Teil des Kapitels wird noch auf die Funktion von Text Mining eingegangen.

2.1. Semantic Web

Das Konzept des Semantic Web wurde nach Berners-Lee u. a. (2001) als Web definiert, in dem die Informationen nicht nur für Menschen verständlich, sondern auch für Maschinen lesbar vorliegen. Es ist also essentiell eine Erweiterung zu dem bereits vorhandenen Internet. Es wird daher in Anspielung auf den Begriff Web 2.0 auch oft von dem semantischem Web als Web 3.0 gesprochen. Der Nutzen einer solchen Erweiterung ist vielfältig. Vor allem lassen sich für den Benutzer nicht wichtige Informationen leicht herausfiltern. Um eine solche Entwicklung zu erreichen ist es nötig von dem aktuellen Internet, einem *Web of Documents*, zu einem *Web of Data* zu gelangen. Betrachtet man das Internet stellt man schnell fest, dass die Informationen in der Regel in Freitexten abgelegt sind welche nicht von Maschinen Interpretiert werden können. Es gibt keine Struktur zwischen den einzelnen Dokumenten und das Thema ist nicht einheitlich deklariert. In einem *Web of Data* hingegen sind Dokumente anhand ihrer Themen zusammenhängend und der Inhalt liegt, beispielsweise durch Annotationen, in maschinenlesbarer Form vor. Das Internet soll nicht länger nach rudimentären Dokumenten geordnet, sondern anhand des Inhalts sortiert zugänglich gemacht werden.

Da es Dokumenten im Internet an semantischen Informationen mangelt sind Suchmaschinen wie Google darauf angewiesen HTML-Dokumente nach Stichworten zu durchsuchen. Sinnvoller wäre es wenn eine Suchmaschine die tatsächliche Semantik einer Webseite verstehen könnte, da sich damit wesentlich komplexere Suchen durchführen lassen können. Dies bezeichnet man als Semantische Suche (vgl. Tabelle 2.1), welche nur ein Anwendungsfall des Semantic Webs ist. Ein weiterer Anwendungsfall sind Querverweise zwischen Inhalten im Semantic Web.

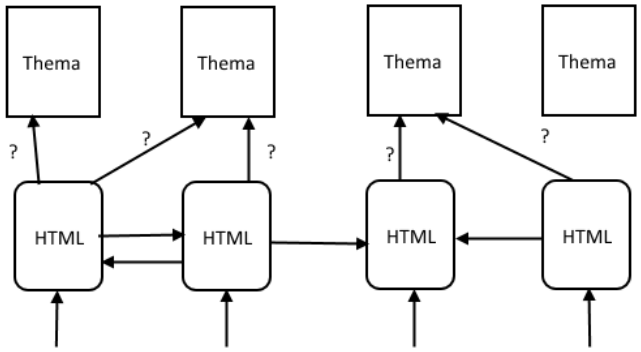

Abbildung 2.1.: Web of Documents

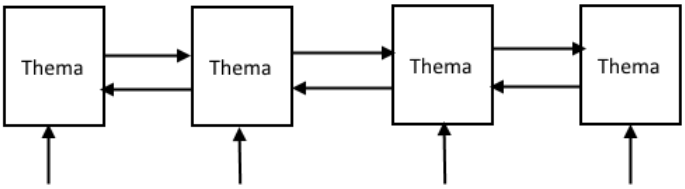

Abbildung 2.2.: Web of Data

Da alle Informationen zusammenhängend sind ist es weitaus einfacher neue relevante Inhalte zu finden. Das Semantic Web kann also auch zur Orientierung im Internet genutzt werden.

Anfrage
Städte in Deutschland mit hoher Durchschnittstemperatur
Sänger wie Elvis
Der schnellste Weg von Berlin nach Hamburg
Anzahl der WM-Finalspiele an denen Deutschland teilgenommen hat

Tabelle 2.1.: Beispiele für Anfragen bei semantischer Suche

2.1.1. Schichtenmodell

Nach Berners-Lee u. a. (2001) lassen sich die Technologien, die für die Umsetzung eines Semantic Web wichtig sind, anhand eines Schichtenmodells einteilen. Im folgenden sollen die einzelnen Schichten erläutert werden. Auf einige Aspekte wie RDF wird im Verlauf dieses Kapitels noch genauer eingegangen.

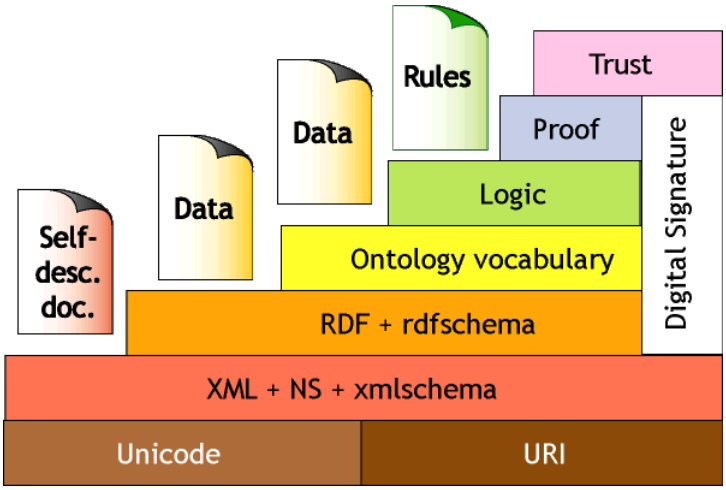

Abbildung 2.3.: Schichtenmodell des Semantic Web

- *Unicode/URI*: Diese Schicht beschreibt die Standards zur Darstellung der Ressourcen. Unicode ist ein Standard für die Anzeige von Zeichen und URI beschreibt den Aufbau der Verlinkungen.
- *XML + Namespace + Schema*: Diese Schicht beschreibt den Aufbau auf der nächsthöheren Ebene. XML ist eine vom W3C standardisierte Auszeichnungssprache welche die Struktur des Dokuments beschreibt.
- *RDF + RDFS*: Das Ressource Description Framework ist eine auf XML basierende Sprache um Aussagen über Ressourcen zu machen. Auf diese Art können semantische Informationen einer Ressource angehängt werden (siehe 2.1.2).
- *Ontology vocabulary*: Auf der Ontologieschicht wird die Semantik zwischen den Konzepten definiert. Die Ontologie stellt die Wissensbasis dar und bietet ein gemeinsames Vokabular mit klar festgelegter Semantik an (siehe 2.1.3).
- *Logic*: Mit Hilfe von Ontologien und RDF sind die Informationen maschinenlesbar abgelegt. Auf der Logikschicht werden nun Abfragen auf Basis von Prädikatenlogik ermöglicht. Dadurch lässt sich beispielsweise die semantische Suche realisieren.
- *Proof*: Auf dieser Schicht wird die Konsistenz und Nachvollziehbarkeit sichergestellt. Dies ist sowohl für den Nutzer als auch die Softwareagenten wichtig die das semantische Web nutzen.

- *Digital Signature*: Neben den vorgestellten Schichten existiert eine auf XML basierende digitale Signatur um eine Authentifizierung zu gewährleisten. Dies ist wichtig für die Nachvollziehbarkeit der semantischen Aussagen. Es soll zum Beispiel gewährleistet werden, dass Ontologien nicht durch fremde Softwareagenten manipuliert werden können.

- *Trust*: Die höchste Schicht im Semantic Web ist die Trust, oder Vertrauensschicht. Durch die vielen Informationen im Internet und da keine prinzipielle Aussage über die Richtigkeit dieser Informationen gemacht werden kann ist eine solche Schicht notwendig. Es ist klar, dass Dinge die in einer Ontologie stehen oder auf einer Webseite semantisch annotiert wurden nicht automatisch wahr sein müssen. Es kann aber auch zwei Quellen mit gegensätzlichen Informationen geben. Die Fragestellung welcher Quelle man in einem solchen Fall vertrauen sollte zu beantworten ist Aufgabe dieser Schicht.

2.1.2. Resource Description Framework

Das *Resource Description Framework* oder auch RDF wurde ursprünglich vom W3C entwickelt um Metainformationen jeglicher Art an Ressourcen anhängen zu können. Inzwischen ist RDF der Standard um Informationen im Semantic Web zu notieren. RDF besteht im wesentlichen aus drei Teilen welche in folgendem Abschnitt erläutert werden.

2.1.2.1. RDF Modell

Im RDF Modell werden alle Informationen anhand eines Graphen dargestellt welcher sich auf die Ressourcen, Eigenschaften und Werte bezieht. Dadurch kann jegliche Informationen in RDF anhand eines Tripels mit *{Ressource, Eigenschaft, Wert}*, also einer Menge mit drei Elementen beschrieben werden.

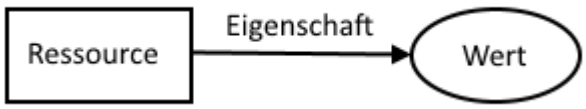

Abbildung 2.4.: Graphische Repräsentation von RDF aus Beckett (2004)

RDF verwendet eindeutige URIs um die Elemente in einem solchen Tripel zu beschreiben. Dies hat eine Reihe von Vorteilen. So lassen sich zum Beispiel neue, ursprünglich vielleicht gar nicht geplante Informationen mit bereits vorhandenen verbinden in dem schlicht auf eine URI verlinkt wird. Hat eine Bücherhandlung beispielsweise Autor und Titel im RDF-Format

abgelegt, so kann ganz einfach der Verlag hinzugefügt werden in dem auf die Ressource Buch verlinkt wird mit einer neuen Eigenschaft Verlag (siehe Abbildung 2.5). Auch können auf diese Weise sehr leicht Synonyme verarbeitet werden. Alle identischen Begriffe müssen lediglich auf die gleiche Ressource verlinken.

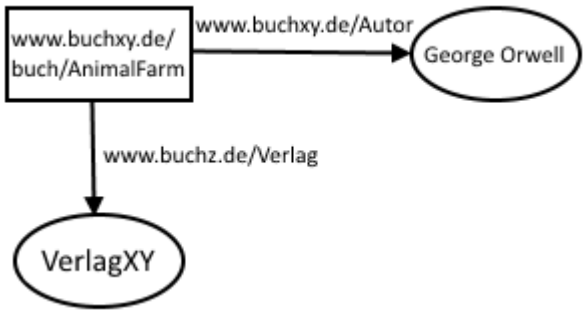

Abbildung 2.5.: Beispiel für RDF Informationen

Da das RDF-Modell so sehr darauf ausgelegt ist, dass jeder zusätzliche Informationen zu einem bereits bekannten Konzept hinzufügen kann wird nun auch klar wieso die Trust-Schicht im Schichtenmodell so essentiell ist Beckett (2004), Gandon und Schreiber (2014).

2.1.2.2. RDF Syntax

Wie bereits erwähnt wird RDF im XML Format geschrieben. Nun soll auf die wichtigsten Syntaxelemente von RDF eingegangen werden um einen Einblick in den Umfang der möglichen Informationen die dargestellt werden können zu bieten.

```
<?xml version="1.0"?>
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:bxy="http://www.buchxy.de/"
        xmlns:bz="http://www.buchz.de/">

        <rdf:Description
        rdf:about="http://www.buchxy.de/buch/AnimalFarm">
                <bxy:Autor>George Orwell</bxy:Autor>
                <bz:Verlag>VerlagXY</bz:Verlag>
        </rdf:Description>
```

```
12 </rdf:RDF>
```

Listing 2.1: RDF Beispiel im XML-Format

An dem Beispiel 2.1 sieht man das zunächst der Namensraum von RDF importiert werden muss. Zusätzliche Namensräume können noch eingebunden werden um die Notation übersichtlicher zu machen. Im folgenden nun eine Übersicht der wichtigsten Elemente von RDF.

Element	Beschreibung
<rdf:RDF>	Das Wurzelelement eines RDF-Dokuments
<rdf:Description>	Umschließt eine konkrete Ressource wie z.B. eine Webseite die beschrieben wird
<rdf:Bag>	Eine unsortierte Liste von Werten
<rdf:Seq>	Eine geordnete Liste von Werten
<rdf:Alt>	Eine Liste von Alternativen von der nur eine gewählt werden kann

Tabelle 2.2.: Übersicht von RDF Sprachelementen

Bag, Seq und Alt sind Container, vergleichbar mit Listen aus der Programmierung, von RDf welche benutzt werden um Gruppen von Ressourcen zu beschreiben. Auf diese Weise können auch Mengenbeziehungen modelliert werden. Grundsätzlich legen die Container nicht fest ob noch weitere, nicht aufgeführte Elemente, dazugehören können oder nicht. Um den Container explizit als unveränderbar zu definieren kann das Attribut *rdf:parseType="Collection"* verwendet werden Beckett (2004), Gandon und Schreiber (2014).

Da die Syntax XML-typisch sehr viel Overhead hat lassen sich die Eigenschaften und Werte der Tripel in RDF nicht nur als Elemente, sondern auch verkürzt als Attribute von Description notieren. Beide Varianten sind semantisch gleich.

```
1 <rdf:Description
2         rdf:about="http://www.buchxy.de/buch/AnimalFarm"
3         ns:Autor="George Orwell" cd:Verlag="Verlagxy"/>
4 </rdf:RDF>
```

Listing 2.2: RDF Beispiel mit verkürzter Notation

Um eine Verlinkung auf eine externe Ressource einzutragen wird das Attribut *rdf:resource* benutzt. Nur auf diese Weise lassen sich viele RDF-Dokumente miteinander verknüpfen und ein semantisches Netz bilden welches traversiert werden kann. nun folgt noch einmal das

bekannte Beispiel, wobei diesmal der Verlag des Buches als externe Ressource deklariert ist.

```
<rdf:Description
        rdf:about="http://www.buchxy.de/buch/AnimalFarm">
        <ns:Autor>George Orwell</ns:Autor>
        <cd:Verlag rdf:resource="www.verlagxy.de"/>
</rdf:Description>
```

Listing 2.3: RDF Beispiel mit externem Link

2.1.2.3. RDF Schema

Der dritte Teil von RDF ist das Schema (RDFS). Das Schema wird ähnlich wie das Schema aus XML dafür verwendet um die Struktur der Daten zu definieren. Dabei ist die Syntax im Grunde genommen die gleiche wie die von RDF. Der einzige Unterschied ist, dass anstatt konkreter Ressourcen nun die möglichen Ressourcenklassen beschrieben werden. RDFS bietet Mittel an um unter anderem die folgenden Dinge festzulegen.

- Das Grundvokabular der Domäne. Die Semantik der Begriffe wird durch das Schema festgelegt. Mehrdeutigkeiten werden so verhindert.
- Die möglichen Klassen und Eigenschaften und ihre Beziehung zueinander.
- Der Wertebereich von Eigenschaften.
- Einfache Klassenhierarchien, ähnlich wie in objektorientierten Programmiersprachen.

```
<?xml version="1.0"?>
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

        <rdfs:Class rdf:ID="Person" />

        <rdfs:Class rdf:ID="Autor">
                <rdfs:subClassOf rdf:resource="#Person"/>
        </rdfs:Class>
</rdf:RDF>
```

Listing 2.4: RDFS Beispiel mit Unterklassen

Mit Hilfe von RDFS wird also der Rahmen für die Domäne festgelegt. Das Schema dient als eine Art Wissensbasis für RDF. Es ergibt sich daher, dass mit RDFS auch einfache Ontologien modelliert werden können. Eine weitere verbreitete Sprache zur Gestaltung von Ontologien ist OWL (Web Onthology Language) welche auf Basis von RDF/S komplexe Ontologien beschreiben kann. Für die Realisierung in dieser Arbeit genügt jedoch RDFS als Beschreibungssprache Guha und Brickley (2014).

2.1.3. Ontologien

Im vorherigen Abschnitt wurde bereits darauf eingegangen, dass mit RDFS auch einfache Ontologien beschrieben werden können. Nun soll erläutert werden, was Ontologien genau sind und was ihre Bedeutung im Kontext des Semantic Webs ist. Der Begriff der Ontologie stammt ursprünglich aus der Philosophie und beschäftigt sich dort mit der Frage nach Existenz von Objekten bzw. Entitäten. In der Informatik bezeichnet eine Ontologie nach Gruber (2009): „..defines a set of representational primitives with which to model a domain of knowledge or discourse." Eine Ontologie dient also dazu Strukturen zu definieren mit denen Konzepte modelliert werden können. Daher spricht man bei Ontologien auch oft von „specification of conceptionalisation." Ein Konzept im Sinne einer Ontologie ist ein Phänomen aus der Realität welches modelliert werden soll. Mit einer Ontologie wird also ein Plan vorgegeben welche Entitäten und Beziehungen zur Modellierung von gewissen Konzepten verwendet werden können. Eine Ontologie besteht prinzipiell aus den folgenden Elementen.

- *Classes* sind die bereits erwähnten Konzepte aus der Domäne. Diese entsprechen auch der in Abschnitt 2.1.2.3 vorgestellten Klassen von RDFS. Diese sind hierarchisch aufgebaut und können voneinander erben. Dabei erbt eine Klasse immer alle Eigenschaften ihrer Oberklasse, vergleichbar mit objektorientierten Programmiersprachen. Außerdem können Klassen als abstrakt definiert werden wenn es von Ihnen keine Instanzen geben kann.
- *Slots* beschreiben die möglichen Attribute die zu einer Klasse gehören. Diese können auch weitere Klassen beinhalten, sowie Listen von Objekten. Slots dienen dazu die Klasse genauer zu beschreiben.
- *Instances* sind die konkreten Objekte, die durch Klassen beschrieben werden. Ein Beispiel wäre eine Klasse *Person* mit einer Instanz *Max Mustermann*.

Die Elemente einer Ontologie erinnern stark an die einer Programmiersprache. Einer der Vorteile von Ontologien ist allerdings, dass diese unabhängig von konkreten Implementie-

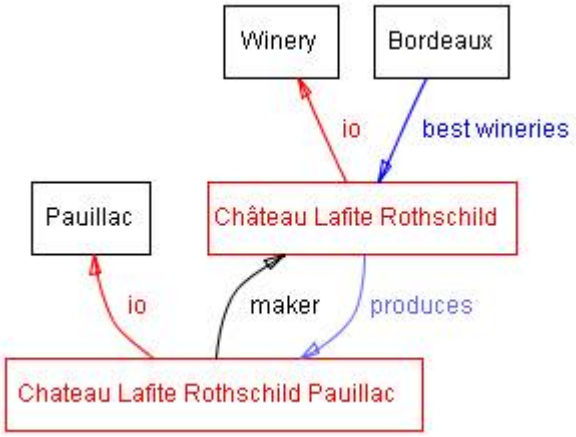

Abbildung 2.6.: Beispielontologie aus Noy und Mcguinness (2001). Klassen sind schwarz, Instanzen rot, und Beziehungen blau dargestellt

rungsdetails sind. Weitere Gründe für den Einsatz von Ontologien im Semantic Web sind. (Gruber (2009), Blumauer und Tassilo (2006), Noy und Mcguinness (2001))

- Wissen einer Domäne zu teilen. Dies ist wohl der Hauptgrund um Ontologien zu entwickeln. Eine einmal fertiggestellte Ontologie für eine Domäne kann von anderen eingebunden werden. Auf diese Weise entsteht das Semantic Web welches von Software Agenten verwendet werden kann um semantische Informationen zu erhalten.
- Eine allgemeine und einheitliche Form um Wissen darzustellen in einer Form deren Semantik für Maschinen verständlich ist.
- Wiederverwertbarkeit von Information in verschiedenen Domänen. Ein typisches Beispiel hier sind Einheiten wie Zeit. Diese werden in einer Reihe von Domänen benötigt. Daher wäre es einfacher wenn eine fertige Ontologie für alle anderen zur Verfügung stehen würde.
- Explizite Darstellung aller Annahmen über die Domäne. In einem typischen Programm ist oft nicht klar welche Grundannahmen über Konzepte getroffen werden. Eine Ontologie visualisiert dies deutlich und Annahmen können durch ändern der Ontologie angepasst werden.
- Trennung vom Domänenwissen und sonstiger Logik.

2.2. Text Mining

Um das Semantic Web umzusetzen müssen die Informationen welche in den RDF-Dokumenten oder Ontologien festgehalten werden zunächst gefunden werden. Eine manuelle Gestaltung aller RDF-Dokumente ist offensichtlich nicht realistisch. Daher beschäftigt sich diese Arbeit mit Ansätzen um den Prozess der Generierung von semantischen Annotationen zu automatisieren. Um eine solche Automatisierung umsetzen zu können müssen die Informationen zunächst aus anderen Quellen extrahiert werden. Diese Quelle ist im Falle des Semantic Webs normalerweise die Webseite, die mit Metainformationen erweitert werden soll. Es müssen folglich semistrukturierte HTML-Seiten analysiert werden können.

Mit der Problemstellung Informationen aus un- oder semistrukturierten Texten zu entnehmen beschäftigt sich das Text Mining. Im Rahmen dieser Arbeit soll untersucht werden, wie mit Text Mining Methoden aus unstrukturierten Blogeinträgen RDF-Annotationen erstellt werden können. Auf die konkreten Techniken wird hierbei in Kapitel 3.4 eingegangen. Mit Hilfe von Text Mining lassen sich jedoch nicht nur Informationen in Texten finden und extrahieren. Es können die folgenden vier grundlegenden Aufgabenbereiche unterschieden werden (Hotho u. a. (2005), Berry (2003)).

- *Classification*, die Frage zu welcher von einer Reihe von vorgegebenen Kategorien ein Text gehört. Ein typisches Beispiel sind Mailfilter welche bestimmen ob eine neue E-Mail in den Spamordner verschoben werden soll.
- *Clustering* ist Eine Sortierung von Texten anhand von Ähnlichkeiten in Clustern. Man kann sich dies wie eine Classification vorstellen, bei der die Kategorien nicht bekannt sind bzw. nicht benannt werden.
- *Information Retrieval* ist der Bereich mit dem sich diese Arbeit vorwiegend beschäftigt. Er beinhaltet das Auffinden von bekannten Daten in unstrukturierten Texten. Dies wird beispielsweise auch von Suchmaschinen verwendet um relevante Ergebnisse für den Nutzer zu finden.
- *Information Extraction*. Im Gegensatz zu Information Retrieval ist hier das Ziel neue Zusammenhänge in einem Text zu finden und sichtbar zu machen.

3. Analyse

In folgendem Kapitel sollen Lösungsansätze aus der aktuellen Forschung vorgestellt und ein Ansatz für den umzusetzenden Prototyp ausgewählt werden. Dazu wird das Szenario für ein Bloggingportal im IT-Kontext umrissen und die Anforderungen werden entsprechend festgelegt.

Anschließend werden die typischen Probleme und Herausforderungen die zur der Generierung von RDF-Annotationen gelöst werden müssen beschrieben. Danach folgt eine Analyse der gängigsten Ansätze und eine Bewertung im Bezug auf die gewählten Anforderungen.

Um der Analyse eine möglichst große Praxisnähe zu verleihen werden anschließend noch einige derzeit in Entwicklung befindliche Systeme zur Erzeugung von RDF-Annotationen näher erläutert um die Ansätze mit der praktischen Anwendung verknüpfen zu können.

Am Ende der Analyse soll ein Lösungsansatz für die Umsetzung im Prototyp ausgewählt werden.

3.1. Szenario des Prototypen

Für die praktische Entwicklung in dieser Arbeit wurde ein Portal um Blogeinträge Verfassen und Veröffentlichen zu können ausgewählt. Dies erlaubt einen praktischen Einblick in die Umsetzbarkeit und den Nutzen von einem System, welches RDF-Annotationen generiert.

Um den Aufwand angemessen zu halten spezialisiert sich das System auf den Bereich der Informatik, sowie Artikel in englischer Sprache. Dadurch ist es möglich auch mit Ansätzen welche Domänenspezifisches Wissen erfordern zu arbeiten. Prinzipiell soll das System möglichst so arbeiten wie es auch bereits verbreitete Bloggingsysteme tun. Der Benutzer soll im Backend des Systems einen neuen Eintrag Verfassen und veröffentlichen können. Anschließend sollen automatisch RDF-Annotationen zu diesem Eintrag erzeugt werden. Sollte eine vollautomatische Generierung nicht umsetzbar sein muss ein weiterer Schritt eingefügt werden um den

Benutzer bei der Generierung eingreifen zu lassen (Siehe Abbildung 3.1). Durch diesen Ansatz lässt sich auch näher betrachten wie viel zusätzlicher Aufwand für den Benutzer realistisch zumutbar ist.

Im Frontend sollen nun der veröffentlichte Artikel und die generierten Annotationen dargestellt werden. Optional werden mit Hilfe der erzeugten RDF-Annotationen für den Artikel relevante Zusatzinformationen aus dem Internet gefunden und angezeigt. Durch den Nutzen dieser Informationen können weitere Aussagen über die Qualität des Systems gemacht werden.

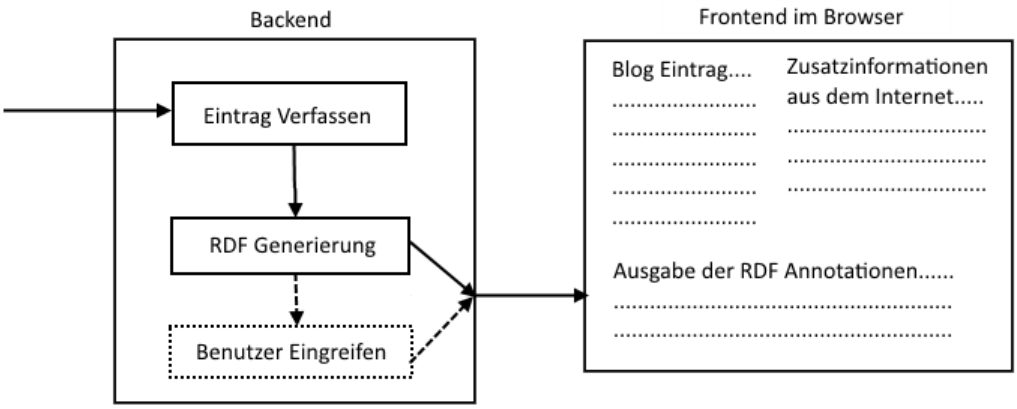

Abbildung 3.1.: Skizze des grundlegenden Ablaufs im Prototypen

3.2. Anforderungen

Nun da das Konzept für das umzusetzende System näher geklärt ist lassen sich die Anforderungen bestimmen. Es ist zu beachten, dass nur eine kleine Anzahl an Anforderungen sich rein objektiv bewerten lassen. So kann beispielsweise nur schwer in harten Zahlen ausgedrückt werden wie gut die Automatisierung eines Systems wirklich ist wenn verschiedene Systeme unterschiedliche Aspekte automatisieren. Eine subjektive Aussage ob eine Umsetzung mehr oder weniger Aufwand automatisiert ist aber durchaus möglich. Aus diesem Grund sind die hier aufgeführten Maßstäbe kaum voll oder gar nicht zu erfüllen sondern eher als Faktor anzusehen. Eine Ausnahme ist hier die Qualität die auf Basis von *Precision*, *Recall* und *F-Score* eingeschätzt werden kann. Auf die konkrete Bewertung eines Systems wird in Kapitel 5 dieser Arbeit näher eingegangen.

Qualität

Die Qualität der Annotationen ist wichtig um dem System einen tatsächlichen Nutzen zu verleihen. Bei einer sehr geringen Qualität können die RDF-Annotationen nicht verwendet werden um die Informationen tatsächlich zu verknüpfen und maschinenlesbar zu machen.

Die Qualität eines Annotationssystems kann auf die gleiche Weise bewertet werden wie die von anderen Information Extraction Systemen. Die beiden Werte *Precision* und *Recall* können als Indikatoren für die Brauchbarkeit verwendet werden. *Precision* bezeichnet hierbei die Korrektheit des Systems, indem die korrekten Annotation mit allen vom System erzeugten Annotationen in Verhältnis gesetzt werden.

$$Precision = \frac{Korrekte\ Annotationen}{Falsche\ Annotationen\ +\ Korrekte\ Annotationen} \tag{3.1}$$

Recall hingegen ist das Verhältnis von den korrekten Annotationen zu allen korrekten Annotationen in dem Dokument. Es wird hierbei also die Vollständigkeit des Systems beschrieben. Als Maßstab für die Gesamtheit der möglichen Annotationen wird in der Regel das Dokument manuell von Hand annotiert und das Ergebnis mit dem des Verfahrens verglichen.

$$Recall = \frac{Korrekte\ Annotationen}{Alle\ korrekten\ Annotationen} \tag{3.2}$$

Precision und *Recall* sind allerdings voneinander abhängig. es kann beispielsweise *Precision* auf Kosten von *Recall* geopfert werden, in dem alle Annotationen welche möglich wären produziert werden. Dies würde zwangsweise dazu führen, dass auch alle korrekten Annotationen gefunden wurden, der *Recall-wert* wäre daher maximal während der *Precision-wert* abfällt. Andersrum kann auch *Recall* zu Gunsten von *Precision* geopfert werden, da das System nur eine einzige Annotation finden könnte welche aber mit sehr hoher Wahrscheinlichkeit auch korrekt ist. Hier wäre der *Precision-wert* maximal und der *Recall-wert* minimal. Aus diesem Grund wird ein weiterer Wert *F-measure* oder auch *F-score* verwendet welcher beide Werte in Verhältnis zueinander setzt.

$$F - measure = \frac{2 * Recall * Precision}{Recall + Precision} \tag{3.3}$$

Es ergibt sich, dass es für die Wahl der Methodik für die Generierung der Annotationen von

großer Bedeutung ist nicht nur auf Charton u. a. (2011) und *Recall* sondern auch auf den daraus resultierenden *F-measure* zu achten. Diese Werte lassen sich als Indikatoren für die Gesamtqualität eines Annotationssystems verwenden.

Automatisierung

Der Grad der Automatisierung des Systems ist von wesentlicher Bedeutung für die praktische Anwendung. Sollte der Aufwand für den Benutzer zu groß werden so wird das System nicht akzeptiert. Gleichzeitig sollte bedacht werden, dass RDF-Annotationen und das semantische Web erst ab einer kritischen Masse einen wirklichen Mehrwert bringen können, da die Verknüpfung der extrahierten Informationen ausreichend stark sein muss. Daraus folgt, dass auch bereits vorhandene Informationen annotiert werden sollten damit neue Systeme nutzbar werden. Da die bereits vorhandenen Informationen im Internet bei weitem zu groß sind um sie manuell zu annotieren ist es von besonderer Bedeutung Ansätze zu finden die einen möglichst großen Automatisierungsgrad versprechen.

Komplexität

Die Laufzeitkomplexität der Erzeugung der RDF-Annotationen sollte gewisse Werte nicht überschreiten. Da bei dem im Szenario vorgestellten System bei der Veröffentlichung eines Blogeintrages die Annotationen sofort erzeugt werden müssen. Die Zeit zwischen dem Absenden des Artikels und dem Erscheinen mit Annotationen sollte auf einem durchschnittlichen Computer keinesfalls zu lange dauern. Als vorläufiger Richtwert sollte ein Artikel mit etwa 1000 Worten innerhalb von maximal 30 Sekunden annotiert werden können.

Generalisierbarkeit

Als Generalisierbarkeit versteht man die Verallgemeinerung der Vorgehensweise auf weitere Domänen. Ist die Methodik von der Domäne abhängig kann diese kaum für größere Systeme verwendet werden. Für den in dieser Arbeit besprochenen Prototypen ist der Grad der Generalisierbarkeit nicht relevant, in der Realität ist eine solche Trennung allerdings notwendig um Annotationen für die vorhandene Domänenvielfalt erzeugen zu können. Im Idealfall ist das Domänenwissen des Lösungsansatzes von allem anderen gekapselt, beispielsweise durch eine verwendete Ontologie.

Unabhängigkeit von weiteren Systemen

Es ist wünschenswert, dass das System nicht von fremden Ressourcen abhängig ist. So ist es Beispielsweise möglich mit Hilfe von Artikeln aus der Wikipedia nach Zusammenhängen

zwischen Entitäten zu suchen. Allerdings bedeutet dies auch, dass keine Nutzung der Plattform möglich ist wenn der Zugriff auf Wikipedia nicht vorhanden ist. Eine Abhängigkeit wäre zu tolerieren, wenn andere Anforderungen wie Qualität und Automatisierung maßgeblich davon profitieren.

Unterstützung verschiedener Sprachen
Eine optionale Anforderung ist die Unterstützung von einer Vielfalt von Sprachen. Auch wenn der umzusetzende Prototyp sich auf das Englische beschränkt so kann eine solche Einschränkung sich im weltweiten Kontext des Internets stark auf die Nutzbarkeit eines Systems auswirken.

3.3. Problembeschreibung

Im folgenden werden einige der Grundlegenden Probleme und Herausforderung näher beschrieben welche bei dem Entwurf und der Auswahl des Lösungsansatzes berücksichtigt werden. Auf die Lösung der hier genannten Problemstellungen wird in Kapitel 3.4 näher eingegangen.

3.3.1. Konstruktion von Ontologien

Um komplexe Beziehungen aus einem zugrundeliegenden Dokument erkennen zu können ist es notwendig das bereits bekannte Wissen über die Domäne zur Verfügung zu haben. Um dies darzustellen werden Ontologien verwendet welche in Beschreibungssprachen wie RDFS oder OWL formuliert werden können. Eine manuelle Erzeugung von derartigen Ontologien ist jedoch sehr zeitaufwendig und mit viel Aufwand verbunden. Auf die manuelle Konstruktion von Ontologien in dieser Arbeit nicht weiter eingegangen.

Bei einer automatischen oder semiautomatischen Generierung müssen zunächst die notwendigen Informationen gewonnen werden. laut Ciravegna u. a. (2004) kann dies durch eine grundlegende manuell erzeugte Ontologie welche nach und nach durch Techniken aus der Statistik, dem maschinellen Lernen oder natürlichen Sprachverarbeitung weitere Informationen dazugewinnt und wächst erreicht werden. Es resultiert daher eine Ontologie welche durch ihre Verwendung neue Fakten über die Domäne lernt. Diesen Ansatz bezeichnet man als *Ontology Learning*.

Nach Zouaq und Nkambou (2010) *Ontology Learning* muss jedoch die Konsistenz der resultierenden Ontologie überwacht werden. Falsche Informationen die aus einem Text stammen

würden zunächst als Fakt hingenommen werden und so die Wissensbasis unbrauchbar machen. Auch Widersprüche müssen explizit gefunden und verhindert werden. Die Qualität der Texte während der Lernphase einer Ontologie ist daher von wesentlicher Bedeutung, dennoch ist es weiter notwendig die Ontologie mit den Fakten aus der realen Domäne abzugleichen.

Problematisch bei den derzeit in der Forschung verwendeten Ansätzen ist der Mangel an Modularisierung um die vielen verschiedenen Methodiken zur Extraktion von Informationen zu kombinieren und einander Gegenüberzustellen. Des weiteren fehlt es an einem formalen Testkonzept um die Qualität einer Ontologie angemessen bewerten zu können. Weiterhin ist zu beachten, dass sich Wissen, insbesondere im Kontext des Semantic Webs ständig verändert und aus diesem Grund eine einmal erzeugte Ontologie durchgehend anpassbar bleiben sollte. Gleichzeitig muss dazu jedoch auch ein Weg gefunden werden um Inkonsistenzen und Duplikate in der Ontologie zu verhindern.

3.3.2. Named Entity Recognition

Der Zweck von Annotationen ist es weitere Informationen zu einer vorhandenen Quelle hinzuzufügen. Laut Charton u. a. (2011) lassen sich zwei Arten von Informationen finden. Zum einen kann zu jeder Entität ein Label mit einer Beschreibung hinzugefügt werden. Als Entitäten werden Beispielsweise Eigennamen, Personen und Orte betrachtet. Die Zuordnung von Labeln zu Entitäten kann als Klassifikationsaufgabe aufgefasst werden, da für eine gegebene Entität eine passende Klasse aus einer vorher bekannten Menge gesucht wird. Trifft in einem Text zum Beispiel ein Name Max Mustermann auf, so wird ihm vom System das Label *Person* zugewiesen. Er wurde also der Klasse *Person* zugeordnet. Eine solche Zuordnung wird als *Named Entity Recognition* bezeichnet.
Das Problem dieses Ansatzes alleine zeigt sich, wenn zusätzlich zu einem Label noch weitere Eigenschaften einer Entität annotiert werden sollen. Zwar wäre es am Beispiel von *Person* noch möglich das Geschlecht in die Klassifizierung aufzunehmen, in dem die Klassen um *Person.maennlich* und *Person.weiblich* erweitert werden, allerdings stößt dieser Ansatz schnell an seine Grenzen wenn auch Attribute wie das Geburtsdatum verwendet werden sollen. Des weiteren besteht das inhärente Problem, dass die möglichen Klassen von vornherein bekannt sein müssen um eine Zuordnung vornehmen zu können. Es wird daher Wissen über die Domäne benötigt bevor ein Text annotiert werden kann.

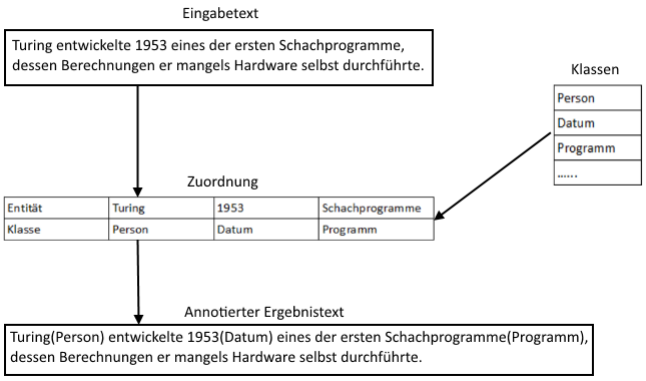

Abbildung 3.2.: Beispiel Named Entity Recognition mit Text aus der Wikipedia

3.3.3. Word Sense Disambiguation

Um eine gefundene Entität um komplexere Informationen zu erweitern wird daher vorgeschlagen die Entität mit einer externen Wissensbasis zu verknüpfen. Wurde mit Hilfe der *Named Entity Recognition* eine Klasse gefunden wird ein Link erzeugt, welcher die Entität eindeutig zuordnet. Ein solcher Link kann als URI aufgebaut werden wie *Person:MaxMustermann* oder *Stadt:Paris*.

Um eine Named Entity Recognition erfolgreich durchführen zu können, müssen Mehrdeutigkeiten korrekt gedeutet werden können. Abbildung 3.3 zeigt mögliche Klassen für die Entität *Paris*. Eine solche Erkennung kann zum Beispiel durch statistische Mittel oder die Betrachtung des Kontextes der Entität vorgenommen worden.

Abbildung 3.3.: Beispiel von Mehrdeutigkeit für Paris aus Charton u. a. (2011)

Die Klasse alleine ist allerdings für die Zuordnung oftmals nicht ausreichend. Betrachten wir das Beispiel von Paris, welcher die Klasse Stadt als Label hinzugefügt wurde so stellt man fest, dass es mehrere Städte namens Paris in Frankreich und den USA gibt. Es muss daher zunächst eine weitere Mehrdeutigkeit überwunden werden. Dieses Problem wird als *Word Sense Disambiguation* bezeichnet und ist ein offenes Problem in der Erkennung von natürlichen

Sprachen. Im Gegensatz zu einer Mehrdeutigkeit bei möglichen Klassen ist hier die Anzahl der Kandidaten nicht durch die begrenze Menge der Klassen limitiert, da ein eindeutiger Identifikator für die Entität gefunden werden muss um sie korrekt mit der Wissensbasis zu verknüpfen. Um dieses Problem zu lösen wurden eine Vielzahl von Ansätzen vorgestellt, eine universelle Lösung ist derzeit allerdings nicht gefunden.

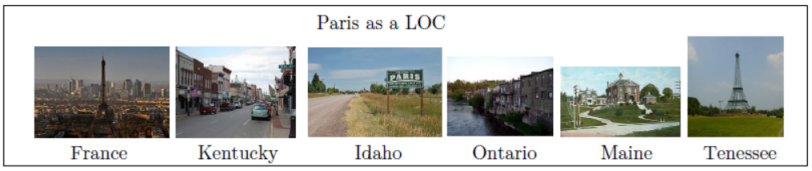

Abbildung 3.4.: Beispiel von Mehrdeutigkeit für Paris als Stadt aus Charton u. a. (2011)

3.4. Lösungsansätze

Im folgenden werden die verschiedenen Ansätze aus der aktuellen Forschung um Semantische Annotationen zu Erzeugen näher erläutert. Grundlegend lässt sich die Vorgehensweise auf Systeme welche mit Mustererkennung arbeiten und jene die Techniken des maschinellen Lernens verwenden zurückführen. Diese beiden Varianten lassen sich in weitere Unterkategorien einteilen. Zusätzlich gibt es Multistrategy-Systeme welche eine Kombination aus beiden bilden.

3.4.1. Mustererkennung

3.4.1.1. Musterfindung

Das Prinzip der Musterfindung besteht nach Brin (1999) darin zunächst eine Menge an Entitäten zu definieren welche in Zusammenhang zueinander stehen, der sogenannte *Seed*. Anschließend wird in Texten nach einem Muster gesucht in dem die Entitäten zueinander stehen. Diese Muster werden nun verwendet um weitere Entitäten zu finden und wiederum neue Muster zu erforschen.

Beispiel

Das System sucht nach Mustern um Autoren zu finden. Als *Seed* wurde {X:George Orwell, Y:Animal Farm} angegeben. Stößt das System nun auf die Textpassagen „George Orwell ist der Autor von Animal Farm“ und „George Orwell schrieb Animal Farm“ so können die Muster „X ist der Autor von Y“ und „X schrieb Y“ extrahiert werden. Anhand dieser Muster lassen sich jetzt weitere Autoren und Bücher im Text finden und miteinander in Beziehung setzen.

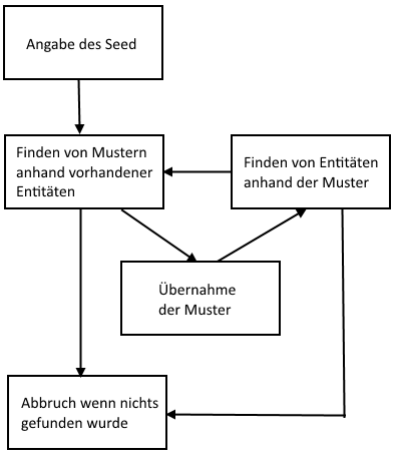

Abbildung 3.5.: Ablauf der Musterfindung

Formale Beschreibung

Nun soll das Vorgehen nochmals nach Brin (1999) formal beschrieben werden.

- D = Eine unstrukturierte Wissensquelle
- R = r_1, ..., r_n, die gesuchte Zielrelation
- Tupel t aus R, welches in D ein- oder mehrfach vorkommt

Ein Vorkommen von t in D wird so definiert, dass jedes Feld aus t innerhalb einer gewissen Nähe zueinander in D zu finden ist. Die verwendete Nähe kann sich dazu auf Sätze, Paragraphen oder Webseiten beziehen, je nach Aufbau der Wissensquelle und Natur der Informationen.

Algorithmus

1. R' ← Seed
   Finde einen Seed für R.
2. O ← FindOccurrences(R',D)
   Finde alle Vorkommen von R' in D. Beachte hierbei die Entfernung innerhalb von D. in O ist der Kontext des Vorkommens enthalten.
3. P ← GenPatterns(O)
   Erzeuge neue Muster anhand der Vorkommen. Es werden Ähnlichkeiten gefunden und daraus generelle Muster erzeugt. Die Qualität von GenPatterns ist maßgeblich für das

Gesamtergebnis. Dieser Teil des Algorithmus ist problemabhängig und daher nicht allgemein lösbar.

4. R' $\leftarrow$ M_D(P)
   Finde neue Tupel anhand der erzeugten Muster.

5. Gehe zu 2 bis R' ausreichend groß ist oder keine neuen Tupel gefunden wurden.

Kontrolle der gelernten Muster

Die im Algorithmus beschriebene automatische Erzeugung von Mustern kann dazu führen, dass Muster gefunden und gelernt werden welche sich von der ursprünglich gewählten Zielrelation wegbewegen und so im Laufe vieler Iterationen die gefundenen Ergebnisse verfälscht werden. Um dies zu vermeiden ist es notwendig entweder die *GenPatterns(O)* oder die $M_D(P)$ Routine so zu konstruieren, dass einzelne fehlerhafte Vorkommen nicht gleich als universelles Beispiel übernommen werden.

GenPatterns(O) sollte so definiert werden, dass potenzielle falsch positive Vorkommen in der Generierung der Muster keinen allzu großen Einfluss haben um potenzielle Abweichungen zu minimieren. Um eine zusätzliche Sicherheit anzubieten kann auch $M_D(P)$ strenger definiert werden. Dazu kann beispielsweise die Anzahl der Muster denen ein Tupel entsprechen muss um aufgenommen zu werden erhöht werden. Ein Tupel das auf 3 Muster passt entspricht mit einer höheren Wahrscheinlichkeit R als ein Tupel welches auf nur eines passt. Der Nachteil dieses Ansatzes ist allerdings, dass hier *Recall* für *Precision* geopfert wird. Es muss daher abgewogen werden ob viele oder qualitativ hochwertige Ergebnisse gewünscht werden.

3.4.1.2. Regelbasiert

Im Gegensatz zu der Musterfindung arbeiten regelbasierte Systeme mit vordefinierten Regeln welche die Muster bilden nach denen gesucht wird. Dazu werden Techniken aus der natürlichsprachlichen Erkennung verwendet um den Text aufzubereiten und anhand der Regeln und Wörterbüchern nach Entitäten und Beziehungen zu durchsuchen. Dieser Ablauf beruht grundlegend auf fünf Schritten welche im folgenden näher erläutert werden.

Part of Speech Tagging

Um einen Fließtext verarbeiten zu können ist es zunächst essenziell die Komponenten innerhalb eines jeden Satzes zu identifizieren. Aus diesem Grund werden alle wichtigen Worte

eines Satzes mit einem Tag oder Label versehen um sie zuordnen zu können. Dieses Vorgehen bezeichnet man als *Part of Speech Tagging* und kann entweder mit Bezug zur Domäne oder ohne durchgeführt werden.

Betrachtet man den Kontext des Semantic Webs. Interessant sind insbesondere Personen, Orte, Konzepte, sowie ihre Eigenschaften und Beziehungen zueinander. Besonders relevant sind daher die Nomen(Personen, Orte, Konzepte), Verben(Beziehungen) und Adjektive(Eigenschaften) in einem Satz. Das Tagging kann sich also auf diese Komponenten reduzieren. Um ein solches Part of Speech Tagging konkret durchzuführen wird überwachtes oder unüberwachtes Lernen verwendet, worauf in dieser Arbeit allerdings nicht näher eingegangen wird.

Eingangstext	John walked to Town
Ergebnistext	John\N walked\V to Town\N

Tabelle 3.1.: Einfaches POS Tagging Beispiel

Morphologische Analyse
Regelbasierte Verfahren verwenden meist vorgegebene, domänenabhängige Wörterbücher um relevante Informationen aus einem Text identifizieren zu können. Um die vorher getaggten Worte mit dem Wörterbuch vergleichen zu können ist es notwendig sie zunächst auf ihre Grundformen zurückzuführen. Die Umwandlung in die Grundform ist nützlich um die Anzahl der Einträge im Wörterbuch zu reduzieren. Würde man alle Formen eines Wortes aufnehmen wäre dieses nicht zu überschauen. Dennoch ist Aufwand und Notwendigkeit einer morphologischen Analyse vorwiegend von der verwendeten Sprache abhängig. Ein System welches sich auf englischsprachige Texte spezialisiert hat oftmals nur wenig oder gar keine morphologische Analyse vorzunehmen während ein System für Deutsch oder Französisch umfangreichere Umwandlungen durchführen muss.

Identifikation von Entitäten
Die Identifikation der interessanten Objekte im nächsten Schritt gestaltet sich zunächst trivial. Es muss lediglich ein matching zwischen den Begriffen im Wörterbuch und dem Text durchgeführt werden. Anschließend wird die gefundene Entität noch mit bekannten Informationen aus dem Wörterbuch, wie der Kategorie, markiert.

Das Problem ist hier allerdings, dass eine Identifikation, die nur auf einem Wörterbuch basiert, zu unflexibel für viele Anwendungen ist. Betrachtet man das in dieser Arbeit umzusetzende Szenario so ist es unrealistisch anzunehmen, dass alle relevanten Entitäten einer Domäne von

vornherein in das Wörterbuch aufgenommen werden könnten. Um dies anzugehen kann das Wörterbuch dynamisch erweiterbar gestaltet werden. Daher kann es durchaus Sinn machen ein regelbasiertes System mit einem zu kombinieren welches Musterfindung 3.4.1.1 einsetzt um weitere Entitäten zu finden. Alternativ dazu kann in einem Fall kann bei einem semistrukturiertem Eingangstext auch mit Hilfe von vorher festgelegten Regeln eine Extraktion neuer Einträge für das Wörterbuch erfolgen. Dies ist allerdings sehr eingeschränkt und lässt sich bei reinem Freitext nicht einsetzen. Auf die Verwendung von Regeln wird im nächsten Abschnitt näher eingegangen.

Identifikation von Beziehungen

Um Beziehungen zwischen den gefundenen Entitäten zu finden werden feste Regeln definiert welche auf einen Zusammenhang schließen lassen. Diese können als kontextfreie Grammatik dargestellt werden. Im folgenden eine simple Beispielgrammatik für das oben verwendete Beispiel von Autoren und Büchern die einander zugeordnet werden.

1. P → AUTOR VERB NP*
2. NP → BUCH (, BUCH)* | BUCH
3. AUTOR → George Orwell | Aldous Huxley
4. BUCH → Animal Farm | 1984 | Brave New World
5. VERB → schreiben | verfassen

Die Regeln P und NP beschreiben hier die Muster welche eine Beziehung zwischen Autoren und einer Menge von Büchern zeigen. Die Regeln AUTOR, BUCH und VERB werden durch die verwendeten Wörterbücher nachgeschlagen und können im Zusammenhang mit der Grammatik ebenso als Terminale angesehen werden. Der Typ der Beziehung wird durch das Verb im Satz bestimmt. Da schrieb und verfasste hier essentiell Synonyme sind könnte also aus dem Satz „George Orwell verfasste Animal Farm„ nach Durchführung der bisherigen Schritte die Beziehung *schreiben(George Orwell, Animal Farm)* finden.

3.4.1.3. Konstruktion einer Ontologie mit Mustern

Die bisherigen Lösungen befassten sich vor allem mit den Problemen der *Named Entity Recognition* und bedingt auch *Word Sense Disambiguation* innerhalb einer Domäne. Um Annotationen für das Semantic Web zu erstellen können Entitäten jedoch nicht isoliert innerhalb eines Textes betrachtet werden sondern müssen in ein Gesamtkonzept in der Domäne eingeordnet werden. Um dies zu erreichen kann eine Ontologie für die Domäne erzeugt werden. Im Folgenden wird

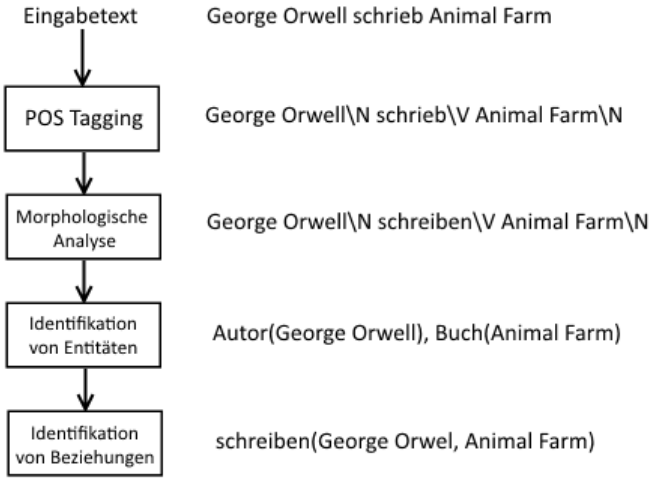

Abbildung 3.6.: Beispiel einer regelbasierten Analyse eines Textes

erklärt wie eine Ontologie mit Hilfe der bekannten Methoden konstruiert werden kann.

Laut Buitelaar und Magnini (2005) kann die Entwicklung einer Ontologie in einzelne Schritte unterteilt werden welche immer weiter an Komplexität zunehmen. Als Metapher für dieses Model wird der *Ontology Layer Cake* (Abbildung 3.7) verwendet. Indem die einzelnen Punkte von unten nach oben abgearbeitet werden kann eine Ontologie generiert werden.

Terms

Als Term wird hier die sprachliche Repräsentation eines Konzeptes bezeichnet. Um solche Terme zu finden wird häufig das bereits vorgestellte *Part Of Speech Tagging* in Kombination mit vorher definierten Regeln verwendet. Einige komplexere Systeme setzen in diesem Schritt hingegen statistische Methoden ein.

Synonyms

Im nächsten Schritt wird für die gefundenen Terme nach Synonymen gesucht. Synonyme sind in diesem Kontext sowohl Worte mit gleicher Bedeutung als auch Übersetzungen aus anderen Sprachen. Eine simple Methodik um solche Synonyme zu finden ist die Verwendung von Wortlisten beispielsweise aus WordNet oder EuroWordNet. Da sich Ontologien immer auf eine bestimmte Domäne beziehen können Mehrdeutigkeiten bei Worten (siehe 3.3.3) hier in der Regel ignoriert werden.

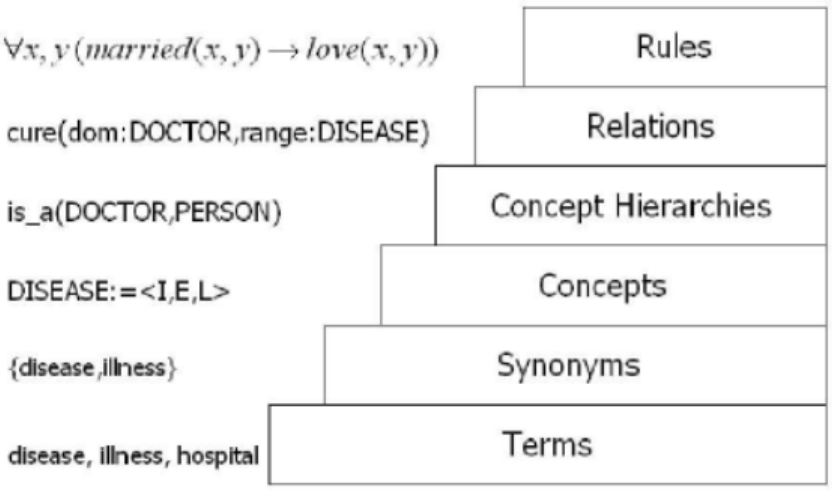

Abbildung 3.7.: Beispielhafte Darstellung des Onthology Layer Cake

Concepts
Die Konzepte der Ontologie lassen sich finden, in dem die Terme in Kategorien aufgeteilt werden. Die am weitesten verbreitete Vorgehensweise hierfür ist ein Clustering der Terme, wobei die erzeugten Cluster jeweils das Konzept bilden.

Concept Hierarchies
Nachdem die Konzepte bekannt sind sollen diese nun hierarchisch geordnet werden. Um dies zu erreichen können Regeln verwendet werden welche eine Ist-Relation beschreiben. Auch kann die schon bekannte Musterfindung verwendet werden mit Konzepten welche bereits eingeordnet wurden. Der Nachteil dieser Ansätze ist allerdings, dass sie nur bei einer ausreichend großen Menge an Dokumenten gut funktionieren, da der *Recall* nur sehr gering ist.

Relations
Nicht-hierarchische Beziehungen sind schwerer zu finden. Prinzipiell kann jedoch die gleiche Technik angewendet werden wie bei der Suche nach Hierarchien. Um Die Qualität hier zu verbessern werden oft statistische Mittel zusätzlich eingesetzt.

Rules
Regeln sind die stärksten Aussagen die durch Ontologien getroffen werden können und gleichzeitig die am schwersten zu Erzeugenden. In der Literatur lassen sich einige Vorschläge finden

wie Regeln aus einem Text extrahiert werden können (Lin und Pantel (2001)). Im Kontext des Semantic Webs ist eine Ontologie mit Konzepten und Beziehungen jedoch ausreichend um brauchbare Annotationen für einen gegeben Text zu Erzeugen.

3.4.2. Maschinelles Lernen

Im nachfolgenden Abschnitt sollen Vorgehensweisen aus der Literatur näher betrachtet werden welche auf Techniken des maschinellen Lernens basieren. Zuerst wird hierzu ein Ansatz von Valarakos u. a. (2003) auf Basis von *Hidden Markov Modellen* vorgestellt. Anschließend wird ein Ansatz der *Wrapper Induction* einsetzt erläutert.

3.4.2.1. Annotation mit Hidden Markov Modell

Das im weiteren vorgestellte System verwendet überwachtes Lernen mit Hidden Markov Modellen um die Semantische Annotation zu verbessern. Die einzelnen Schritte sind:

1. Erstellen einer Basisontologie für die Domäne.
2. Annotieren von Texten mit Hilfe der Ontologie. Dies kann zum Beispiel durch einfaches Stringmatching vorgenommen werden.
3. Lernen mit dem annotierten Text durch HMM.
4. Hinzufügen von weiteren Annotation durch das auf maschinellem Lernen basierte System.

Der Vorteil dieser Vorgehensweise ist, dass Konzepte welche nicht in der Ontologie zu finden sind durch das System gelernt und hinzugefügt werden können. So kann durch doppeltes Annotieren der *Recall* verbessert werden. Ein Nachteil dieser Technik ist allerdings, dass zunächst eine verlässliche Ontologie als Basis für den weiteren Prozess gefunden werden muss.

Prinzipiell wären auch andere Vorgehensweisen als Basis für das Lernverfahren denkbar als Hidden Markov Modelle. HMM ist allerdings eine effektive Strategie und wird vielfach in der Texterkennung eingesetzt. Im folgenden werden HMM daher noch näher erläutert.

Hidden Markov Model
Das Hidden Markov Model ist ein Model aus der Stochastik. Dazu wird ein Markov-Prozess, ein Prozess der so definiert ist, dass Aussagen über die Zukunft mit begrenztem Wissen über die Vergangenheit genauso gut sind wie welche mit vollem Wissen, verwendet. Das Hidden

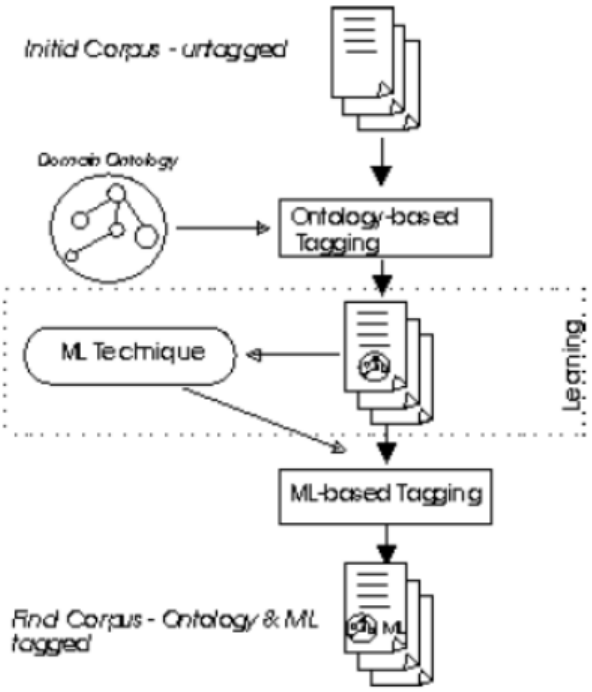

Abbildung 3.8.: Vorgehen auf Basis von maschinellem Lernen nach Valarakos u. a. (2003)

Markov Modell beschreibt einen Markov-Prozess mit unsichtbaren Zuständen. Anstatt der Zustände werden statt dessen mit bestimmter Wahrscheinlichkeit beobachtbare Ausgaben, sogenannte Emissionen von den Zuständen ausgesendet. Die grundlegende Idee ist nun mit Hilfe der beobachteten Emissionen Rückschlüsse auf die verborgenen Zustande zu ziehen.

Ein Hidden Markov Modell ist als ein 5er Tupel λ(Q, V, A, B, π) definiert wobei

- Q = Menge aller Zustände
- V = Menge aller möglichen Beobachtungen (Emissionen)
- A = Wahrscheinlichkeitsmatrix für Zustandsübergänge
- B = Wahrscheinlichkeitsmatrix für Beobachtungen in einem Zustand
- π = Anfangsverteilung der Zustände
- O = Beobachtete Sequenz von Emissionen

ist. Mit Hilfe eines Hidden Markov Modells lassen sich prinzipiell 3 Probleme lösen.

1. Die Wahrscheinlichkeit für die Beobachtungen ermitteln bei gegebenen Modell (A, B, π) und Emissionen O.
2. Die Zustände und Übergänge ermitteln bei gegebenen Modell (A, B, π) und Emissionen O.
3. Für eine Menge von Beobachtungen O ein Modell finden welches die Wahrscheinlichkeit von O maximiert.

Die dritte Problemstellung kann als Lernproblem aufgefasst werden. Das Modell wird angepasst bis es schließlich auf die gegebenen Observationen passt. Dieser Ansatz wird auch verwendet um mit Hidden Markov Modellen Annotationen zu erlernen und ist der Grund warum HMM in Spracherkennung und Texterkennung so verbreitet sind (Stamp (2004)).

3.4.2.2. Wrapper Induction

Nun soll *Wrapper Induction*, ein weiterer auf Maschinellem Lernen basierender Ansatz, vorgestellt werden. Als Wrapper wird in der Informatik im allgemeinen eine Software bezeichnet welche einen gewissen Kontext umschließt und für einen anderen verständlich macht. Sie sind vergleichbar mit einem Adapter oder einer Schnittstelle. Ein Beispiel aus der Programmiersprache Java sind die Wrapperklassen für die objektorientierten Varianten der primitiven Datentypen *int, float* und *double* zu *Integer, Float* und *Double*. Die Übersetzung findet hier zwischen den objektorientierten Prinzipien von Java und der low-level Ebene statt. Die Aufgabe semantische Annotationen für einen Fließtext zu generieren kann daher ganz ähnlich aufgefasst werden. Schließlich geht es darum den Inhalt des Textes in eine maschinenlesbare Form zu bringen. Ein Wrapper könnte daher aus einem Text RDF-Annotationen erzeugen mit denen ein Computer umgehen kann.

Ein Problem ist, dass ein Wrapper nicht allgemein genug sein kann um eine Reihe von unterschiedlichen Entitäten und Relationen zu berücksichtigen. Es wäre daher praktisch notwendig für jede Entität einen eigenen Wrapper zu konstruieren. Es ist also leicht vorstellbar, dass eine manuelle Erzeugung von geeigneten Wrappern nicht praktikabel ist. Der Begriff *Wrapper Induction* bezeichnet den Vorgang automatisiert Wrapper zu Erzeugen.

Problembeschreibung
Nach Kushmerick (2000) lässt sich das durch *Wrapper Induction* zu lösende Problem wie folgt beschreiben. Es existiert dazu

- Eine Ressource S die eine Menge von Seiten enthält welche abgerufen werden können
- Die Seite P, der verwendete Ausgangstext
- Für jede Entität gibt es ein Tupel t der Größe der Menge von Attributen, vergleichbar mit einer Spalte in einer relationalen Datenbank
- Der Inhalt einer Seite, die Menge der enthaltenen Tupel
- Der Inhalt wird durch ein Label L repräsentiert. Das Label kann Indices auf den Beginn eines jeden Attributes in einem Tupel darstellen oder aber eine andere Form haben.
- der Wrapper w sei eine Funktion von der Seite zu der Repräsentation des Inhalts durch das Label w(P) = L

Es soll ein Wrapper für die Informationen aus S mit Hilfe von überwachtem Lernen erzeugt werden. Das Trainingsset beinhaltet eine Menge $\mathcal{E}$ von Paaren (P_n,L_n) und das Ergebnis des Lernprozesses ist der Wrapper w, so dass w(P_n) = L_n für alle (P_n,L_n) $\in \mathcal{E}$.

LR-Wrapper

Die Anwendung eines Wrappers lässt sich so beschreiben, dass eine Funktion jeweils nach einer Linken und Rechten Begrenzung (engl. Delimiter) für ein Attribut aus einem Tupel in einem Text sucht. Einen solchen Wrapper bezeichnet man als LR-Wrapper. Auf diese Weise lässt sich der Wrapper auf das Finden von passenden Delimitern für den Ausgangstext begrenzen. Die Suche nach Delimitern lässt sich als ein *Constraint Satisfaction Problem* beschreiben bei dem die Delimiter die Variablen sind, die Domäne ein beliebiger String aus dem Text und die Constraints, dass die Delimeter die Funktion w(P_n) = L_n für alle (P_n,L_n) $\in \mathcal{E}$ erfüllen. Neben LR-Wrappern existieren noch weitere Varianten wie HLRT Wrapper welche zusätzlich noch zwei Delimiter beinhalten die beschreiben wo der Body der Seite anfängt (Head) und endet (Tail) und weitere. Die prinzipielle Vorgehensweise ist jedoch immer die selbe.

Probleme des Verfahrens

Auch wenn Wrapper Induction als Verfahren für semantische Annotation sehr vielversprechend ist, so zeigen sich bei näherer Betrachtung einige Probleme auf. Zunächst funktioniert der hier beschriebene Ansatz nur bei semistrukturierten Seiten welche einen wiederkehrenden Aufbau haben. Ferner wird eine Menge an Seiten als Trainingsset benötigt welche bereits von Hand gelabelt wurden, was sehr aufwändig ist. Des Weiteren kann eine Strukturänderung bei der abgefragten Ressource leicht dazu führen, dass bereits gelernte Wrapper auf einen Schlag

unbrauchbar werden und der Lernvorgang wiederholt werden muss. Allerdings gibt es bereits Ansätze in der Forschung um diese Probleme zu lösen. So verwendet beispielsweise Gentile u. a. (2013) ein Linked Data Interface um das manuelle Labeling zu umgehen während MnM Vargas-Vera u. a. (2002) die durch Wrapper Induction gefundenen Delimiter verwendet um auf ihnen basierend allgemeine Regeln zu erzeugen.

3.5. Ergebnis der Analyse

3.5.1. Bewertung anhand der Anforderungen

Im Laufe der Analyse haben wir zunächst das Szenario des zu implementierenden Prototyps beschrieben. Anschließend wurden die Anforderungen festgelegt und die größten zu lösenden Probleme erläutert. Danach wurden die verschiedenen Ansätze für semantische Annotation beschrieben, sowohl die auf Mustererkennung basierenden, als auch Ansätze aus Basis des maschinellen Lernens. Im letzten Teil der Analyse soll nun ein Vorgehen für die Realisierung des Prototypen ausgewählt werden. Dazu sollen nun die Anforderungen in Verhältnis mit den Möglichkeiten der Ansätze gesetzt werden.

Qualität

Um die Qualität der Verfahren zu beurteilen ist es sinnvoll sich Precision, Recall und F-Measure anzusehen. Aus diesem Grund werden im folgenden die Werte von Systemen verglichen welche die vorgestellten Ansätze prinzipiell umsetzen. Es ergibt sich, dass die verglichenen Plattformen ihre Arbeitsweise weiter optimieren als die in dieser Arbeit vorgestellten grundlegenden Techniken, dennoch erscheint es als Indikator für die potenzielle Qualität sinnvoll einen Vergleich zu ziehen. Leider ließen sich keine Daten zu einer Plattform aus Basis von Hidden Marko Modellen finden. Es werden folgende Systeme herangezogen: Armadillo Chapman u. a. (2004), MUSE Maynard (2003) und MnM Vargas-Vera u. a. (2002)

Verfahren	Plattform	Precision	Recall	F-Measure
Musterfindung	Armadillo	91.0	74.0	87.0
Regelbasiert	MUSE	93.5	92.3	92.9
Wrapper Induction	MnM	95.0	90.0	n/a

Tabelle 3.2.: Vergleich der Systeme, vgl. Reeve und Han (2005)

Aus der Tabelle ergibt sich, dass der Precision-wert bei allen Ansätzen recht hoch ist, im Bereich von 90%. Beim Recall schlägt hingegen die Musterfindung aus mit nur 74%. Dies lässt sich darauf zurückführen, dass nicht genügend Muster gefunden werden können um alle

speziellen Informationen zu finden. Insgesamt ergibt sich bei Regelbasierten Systemen die beste Konsistenz aus Precision, Recall und F-Measure.

Automatisierung

Im Rahmen der Analyse hat sich herausgestellt, dass eine vollständige Automatisierung von semantischer Annotation nach aktuellem Stand der Forschung noch nicht umsetzbar ist. Um die Verbindung zwischen einer gefundenen Information und ihrer semantischen Bedeutung ziehen zu können wird Domänenwissen benötigt. Tabelle 3.5.1 zeigt welches menschliche Eingreifen nötig ist um dieses Wissen zu modellieren.

Verfahren	Nötiger manueller Prozess	Bewertung
Musterfindung	Beispieltupel (Seeds) und Wörterbücher	Einfach
Regelbasiert	(Regeln definieren) und Wörterbücher	Abhängig von Qualität der Regeln
HMM	Komplexe Ontologie	Komplex
Wrapper Induction	Trainingssets	Komplex

Tabelle 3.3.: Einschätzung der Verfahren im Bezug auf die Automatisierung

Das Verfahren auf Basis von Musterfindung bietet nach dieser Einschätzung den simpelsten Prozess um den benötigten Input abzubilden. Es werden für jede benötigte Beziehung einige wenige Beispiele benötigt. Ein Vorteil ist das weitere Beispiele durch den Lernprozess hinzukommen. Sowohl Musterfindung als auch regelbasierte Systeme verwenden normalerweise Wörterbücher und einfache Ontologien. Wörterbücher sind recht einfach zu erstellen und Ontologien können mit Hilfe ähnlicher Techniken bereits semiautomatisch erzeugt werden. Das Problem bei regelbasierten Systemen ist das es mit einem großen analytischen Aufwand verbunden ist Regeln zu finden die allgemein genug gehalten sind. Bei Regeln welche von der Domäne abhängig sind müssen diese potenziell geändert werden was den Aufwand entsprechend stark vergrößert.

Hidden Markov Modelle hingegen verlangen eine umfangreiche und korrekte Ontologie bevor der Lernprozess durchgeführt werden kann. Eine solche Ontologie muss normalerweise von Hand gestaltet werden. Das sogenannten *Onthology Engineering* ist ein komplexer Prozess welcher für jede neue Domäne durchgeführt werden muss. Ähnlich verhält es sich mit Wrapper Induction. Es müssen bereits annotierte Training Sets vorhanden sein bevor gelernt werden kann was einen erheblichen Aufwand bedeutet.

Komplexität
Aus der Analyse heraus lassen sich keine nennenswerten Informationen zur Komplexität der einzelnen Verfahren finden. Da es nur wenige klare Algorithmen gibt und alle Verfahren ihre Basis in der künstlichen Intelligenz haben kann hier keine verlässliche Aussage gemacht werden.

Generalisierbarkeit
Alle vorgestellten Ansätze sind prinzipiell von der zugrundeliegenden Domäne gekapselt und somit unabhängig. Da jedoch jeder Ansatz den Zugriff auf Domänenwissen in irgendeiner Form verlangt lässt sich ein System aktuell auch nur für jeweils eine Domäne auf einmal umsetzen. Diese Einschränkung ist jedoch ausreichend für den Prototypen und den Großteil der denkbaren Systeme. Der Aufwand um von einer Domäne in eine weitere zu wechseln ist eng gekoppelt mit dem Grad der Automatisierung des Systems, da der benötigte menschliche Input die Domäne beschreibt. Aus diesem Grund ergibt sich die Bewertung so wie bei der Automatisierung.

Unabhängigkeit von weiteren Systemen
Die in dieser Arbeit vorgestellten Vorgehensweisen benötigen keine Abhängigkeit zu Fremdsystemen . Selbstverständlich sind Abhängigkeiten zu der Quelle des Domänenwissens oder den Regeln in musterbasierten Systemen vorhanden, diese werden jedoch als Teil des Gesamtsystems verstanden. Es existieren zwar Vorgehensweisen auf Basis von im Internet zu findenden Information wie Linked Data (Gentile u. a. (2013)), diese überschreiten jedoch den Umfang dieser Arbeit.

Unterstützung verschiedener Sprachen
Eine konkrete Abhängig von gewissen Sprachen hat sich in der Analyse nicht herausgestellt. Dennoch erscheint es offensichtlich, dass Systeme welche auf Basis der natürlichen Sprachen arbeiten nur für eine einzige Sprache umgesetzt sind. Auch ist die Komplexität, beispielsweise der morphologischen Analyse, von der jeweiligen Sprache abhängig. HMM und Wrapper Induction sind von sich aus unabhängig von der Sprache des Textes, der Lernprozess wird jedoch weitaus schwieriger wenn ein Trainingsset mit radikal unterschiedlichen Beispielen verwendet werden muss. Es lässt sich daher bislang keine optimale Lösung finden. Schwierig wird es lediglich wenn feste Regeln von Hand für eine Sprache definiert sind. In diesem Fall ist der Aufwand für einen Wechsel der Sprache am größten. Die Umsetzung in dieser Arbeit beschränkt sich jedoch auf Texte in englischer Sprache.

3.5.2. Schlussfolgerung

Es hat sich in der Analyse ergeben, dass alle Ansätze Vor- und Nachteile haben und es hat sich kein Ansatz als klar als der bessere herausgestellt. Alle Verfahren haben jedoch gemein, dass sie keine vollkommene Automatisierung bieten. Eine Modellierung von Menschenhand ist in gewisser Weise immer nötig. Sei es eine Ontologie, Wörterbücher, Trainingsdaten oder sonstiges. Tabelle 3.5.2 zeigt eine Übersicht der Bewertung anhand der zuvor festgelegten Anforderungen. Diese stellt die Verfahren subjektiv einander gegenüber wobei 1 die positivste und 3 die schlechteste Einschätzung darstellt.

Verfahren	Qualität	Automatisierung	Komplexität	Generalisierbarkeit	Unabhängigkeit	Sprachen
Musterfindung	3	1	-	1	1	1
Regelbasiert	1	2-3	-	2-3	1	2
HMM	-	3	-	3	1	1
Wrapper Induction	2	3	-	3	1	1

Tabelle 3.4.: Bewertung der Verfahren auf Basis der Anforderungen

Die Ansätze auf Basis von Mustererkennung scheinen die erfolgversprechendsten zu sein im Bezug auf die gegebenen Anforderungen. Für die Umsetzung wird daher ein Regelbasierter Ansatz in Kombination mit einer Erweiterung um Musterfindung gewählt. Dies verspricht sowohl eine gute Qualität als auch eine angemessenen Automatisierung. Zusätzlich werden keine umfangreichen Testdaten benötigt. Ein letzter Grund für die Wahl ist die potenzielle Erweiterung um semiautomatisch erzeugte Ontologien auf Basis des regelbasierten Ansatzes.

4. Realisierung

In diesem Kapitel wird die Umsetzung des Prototyps für ein Bloggingportal mit automatisch erzeugten RDF-Annotationen vorgestellt. In der Analyse hat sich ergeben, dass Domänenwissen in jedem Fall modelliert werden muss. Aus diesem Grund erscheint es sinnvoll den Kontext des Portals weiter einzuschränken. Anstatt Bloggingeinträge aus der allgemeinen Informatik wird nur noch das Gebiet der Softwareentwicklung unterstützt. Diese Einschränkung erlaubt es von Hand erzeugte Ontologien und Wörterbücher zu verwenden und dennoch auf akzeptable Ergebnisse zu kommen.

Die Umsetzung insgesamt ist als eine Art Proof of Concept zu verstehen. Es geht daher vor allem um den Nachweis der Machbarkeit und dem praktikablen Einsatz eines solchen Systems mit den in der Analyse vorgestellten Mitteln. Um den regelbasierten Ansatz umzusetzen wird das Apache UIMA (Ferrucci und Lally (2004)) Framework verwendet. Im folgenden werden zunächst die technischen Hilfsmittel und Frameworks beschrieben die bei der Umsetzung eingesetzt wurden. Anschließend folgt eine Schritt für Schritt Beschreibung der grundlegenden Umsetzung von der Oberfläche bis zu den Annotationen. Im letzten Teil dieses Kapitels wird noch eine implementierte Erweiterung auf Basis der Musterfindung beschrieben.

4.1. Verwendete Frameworks

4.1.1. UIMA

UIMA oder *Unstructured Information Management Architecture* ist eine allgemeine Architektur zur Analyse von unstrukturierten Inhalten wie Text, Bildern oder Video. Eine Implementierung von UIMA ist das Apache UIMA Framework welches ursprünglich von IBM entwickelt wurde und seit 2006 von Apache geleitet wird. Die Entscheidung UIMA zu nutzen rührt daher, dass UIMA über eine große Community verfügt und von OASIS als Standard deklariert wurde. Dies führt zu einer Vielzahl an standardisierten Komponenten welche miteinander verwendet werden können und die Entwicklung stark vereinfachen. Zusätzlich bietet UIMA ein Eclipseplugin an welches die Arbeit erleichtert. Als nennenswerte Alternative zu UIMA ist das GATE

Framework (Cunningham u. a. (2011)) zu nennen welches über eine ähnlich große Verbreitung verfügt. Die hier besprochenen Abläufe beziehen sich konkret auf die Javaimplementierung von UIMA. Im folgenden sollen die grundlegenden Komponenten der UIMA-Pipeline vorgestellt werden.

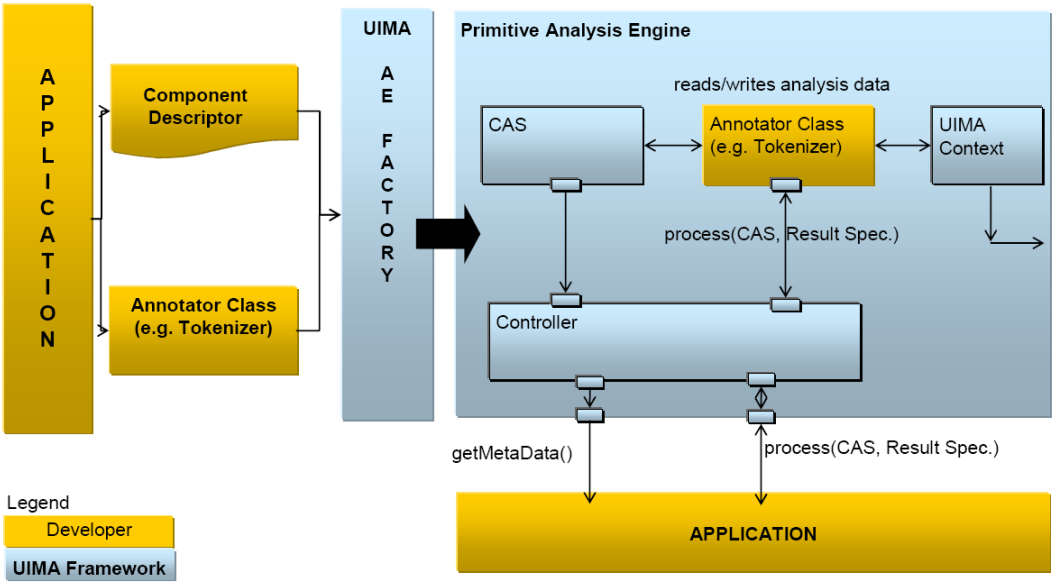

Abbildung 4.1.: UIMA Architektur nach UIMA (2014)

4.1.1.1. Analysis Engine

Jede Verarbeitung einer Ressource wird durch eine Analysis Engine (AE) vorgenommen. Es ist also letztlich eine Funktion mit Input und Output. Ein einfaches Beispiel ist ein Tokenizer für einen Text. Der Input wäre hier der zu verarbeitende Text und der Output ist eine Annotation jedes Tokens. Das Ergebnis einer Analysis Engine wird als Annotation oder Feature gespeichert. Eine Annotation hat einen Start- und Endindex, während ein Feature sich auf die gesamte Ressource bezieht. Es gibt in UIMA primitive und aggragate Analysis Engines. Primitive sind lediglich eine einzige Funktion, wie der oben genannte Tokenizer. Bei aggrate AEs hingegen werden mehrere primitive in Reihe geschaltet. Auf diese Art können komplexe Vorgänge durch einzelne in sich abgeschlossene Komponenten durchgeführt werden. Die AE zur Generierung von RDF-Annotationen ist als eine einzige aggregate AE implementiert.

4.1.1.2. Sofa

Als Sofa (*Subject of Analysis*) bezeichnet man in UIMA die Ressource welche mit Annotationen versehen wird. In unserem Fall also ein Blogeintrag. Interessant ist, dass sich mit UIMA mehr als ein Sofa auf einmal bearbeiten lässt. Es können Collections von Sofas verwendet werden welche durch eine AE annotiert werden. Im Fall des hier umgesetzten Prototypen ist dies allerdings nicht nötig da immer nur ein Blogeintrag auf einmal analysiert wird.

4.1.1.3. Typesystem

Das Typesystem beschreibt die Typen von Annotationen mit denen eine Analysis Engine arbeitet. Dazu werden Typen für Input und Output definiert. Wenn mehrere AEs verwendet werden müssen gleiche Typen einander entsprechen. Ein Annotator welcher Token verwendet sollte die gleiche Definition einer Tokenannotation als Input verwenden wie der Tokenizer als Output. Das UIMA Plug-in für Eclipse bietet die Möglichkeit Typen zu vereinigen um sie kompatibel zu machen. Dies sollte allerdings mit Vorsicht eingesetzt werden um Mehrfachdeklarationen zu vermeiden.

4.1.1.4. CAS

CAS oder auch *Common Annotation Structure* ist die Datenstruktur die genutzt wird um das aktuelle Sofa und die gefundenen Annotationen darzustellen. Das CAS bildet somit die Verbindung zwischen einzelnen primitiven AEs und stellt eine gemeinsame Datenstruktur sicher. Nach der Analyse stehen im CAS die Ergebnisse. Die Javaimplementierung von UIMA bietet zusätzlich eine JCas Klasse an welche sich stärker an den objektorientierten Prinzipien von Java orientiert um die Handhabung zu erleichtern.

4.1.1.5. Component Descriptor

UIMA verwendet XML-Dateien zur Konfiguration. Diese werden als Component Descriptor bezeichnet. Es gibt Type System Descriptoren um ein Typesystem zu definieren und Analysis Engine Descriptoren zur Konfiguration von AEs. Anhand dieser Dateien können mit Hilfe des Eclipse-Plugins gleich die entsprechenden Javaklassen erzeugt werden. So muss für einen Annotator nur noch die vorgegebene *process()* Methode implementiert werden. Alles weitere wird durch die Konfiguration gelöst.

4.1.2. Play

Für die Weboberfläche des Bloggingportals wird das Play-Framework benutzt. Play ist ein auf Scala basierendes Framwork welches die Entwicklung von Webapplikationen stark vereinfacht. Play ist für Java und Scala erhältlich. In dieser Umsetzung hier wird Play für Java verwendet.

4.1.2.1. MVC

Play hält sich strikt an das Model-View-Controller Muster. Ein Projekt mit Play hat automatisch die folgende Paketstruktur:

- app/models
- app/controllers
- app/views

Diese Struktur erlaubt eine gute Übersicht und eine schnelle Einarbeitung in bereits existierende Projekte. Außerdem wird der Konfigurationsaufwand verringert, da Sichtbarkeiten von vornherein definiert sind.

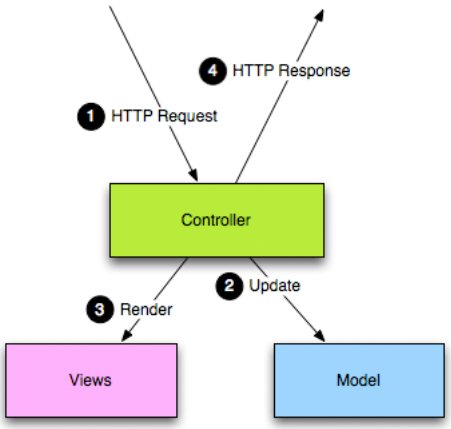

Abbildung 4.2.: Antwort auf ein HTTPRequest in Play nach Typesafe (2014)

4.1.2.2. Build

Play bietet eine eigene Konsole an um Projekte einfach zu erstellen und zu starten. Dazu kann ein integrierter Netty Webserver benutzt werden. Auf diese Weise wird kein eigener

Webserver benötigt um mit dem Framework zu arbeiten. Außerdem ist eine H2 Datenbank von vornherein integriert. Für ein relativ kleines Projekt wie das Bloggingportal sind keine großen Konfigurationen vorzunehmen. Folgende Kommandos sind die wichtigsten um mit der Konsole von Play zu arbeiten:

- play new projektname
- play eclipse
- play run
- play war projektname -o projektname.war

Mit *play new* wird das Projekt mit der bereits besprochenen Struktur angelegt. Dann kann mit *play eclipse* das Projekt in ein Eclipseprojekt umgewandelt werden. Mit dem Kommando *play run* wird das Projekt gestartet. Besonders hilfreich ist hier, dass es nicht nötig ist die Applikation nach Änderungen neu zu starten, sondern eine Aktualisierung der Webseite genügt.

Für den Buildprozess der Applikation benutzt Play SBT als Tool. SBT ist ähnlich wie Maven, welches vor allem für Javaprojekte verwendet wird. Es können Abhängigkeiten zu Bibliotheken deklarativ angegeben werden welche automatisch aufgelöst werden. Im Gegensatz zu Maven basiert SBT allerdings auf Scaladateien anstatt XML-Dateien.

```
1 name := "RDFBlog"
2
3 version := "1.0-SNAPSHOT"
4
5 libraryDependencies ++= Seq(
6   javaJdbc,
7   javaEbean,
8   cache,
9   "org.apache.uima" % "uimaj-core" % "2.3.1",
10  "org.apache.uima" % "WhitespaceTokenizer" % "2.3.1",
11  "org.apache.uima" % "Tagger" % "2.3.1"
12 )
13
14
15 play.Project.playJavaSettings
```

Abbildung 4.3.: Auszug der build.sbt während der Entwicklung

4.2. Vorgehensweise

In diesem Abschnitt wird die Vorgehensweise während der Entwicklung erläutert. Insbesondere soll auf die Implementierung der einzelnen Schritt aus 3.4.1.2 eingegangen werden.

4.2.1. Projektstruktur und Gestaltung der Oberfläche

Zunächst wurde ein Projekt mit Play erzeugt. Es wurden keine besonderen Konfigurationen vorgenommen. H2 wurde als Datenbank verwendet und das integrierte Netty als Webserver benutzt. Die Projektstruktur ergab sich grundlegend bereits aus der MVC-Struktur die durch das Playframework vorgegeben wird. Als erstes sollte das Gerüst zum anlegen und anzeigen von Blogeinträgen erstellt werden. Im folgenden soll die Implementierung und die Projektstruktur grob erläutert werden.

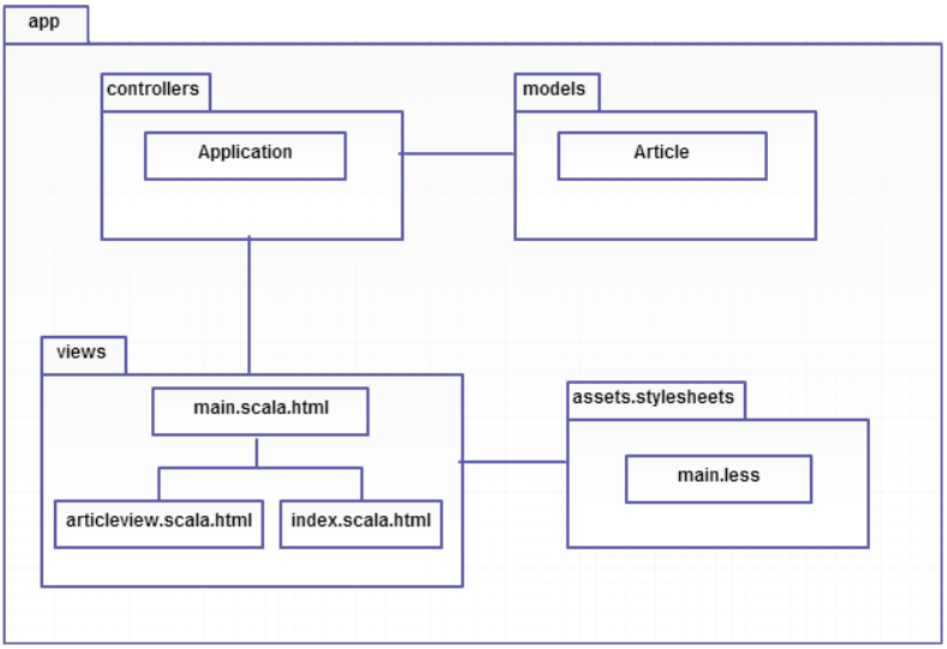

Abbildung 4.4.: Projektstruktur ohne UIMA

models
Hier wurde der einzelne Blogeintrag in der Klasse *Article.java* als persistente Entität für die Datenbank entsprechend der Java Persistence API modelliert. Wichtig ist, dass ein Article neben einer Id, dem Titel und Inhalt auch ein Feld hat für die RDF-Annotationen vom Typ String. Nachdem ein Benutzer einen Artikel erstellt soll später nachträglich durch UIMA die Annotation gefunden und angefügt werden.

controllers
Application stellt den Controller des Programms dar und bietet Methoden zur Manipulation von Artikeln für die Views an. So existiert beispielsweise eine Methode *newArticle()* welche

einen neuen Artikel anhand des Inhalts eines Formulars erstellt. Gleichzeitig ist der Controller dafür verantwortlich zu entscheiden welche View als nächstes angezeigt werden soll. Da sich die Geschäftslogik eines Programms auf der Controllerebene befindet soll hier auch später der Aufruf von UIMA erfolgen, sowie die Mustergenerierung angestoßen werden. Damit kann die UIMA-Pipeline durch den Controller aufgerufen werden sobald ein neuer Artikel erzeugt wird und die Annotation durchgeführt werden noch bevor er angezeigt wird.

views
Wenn Play genutzt wird werden die Oberflächen der Webapplikation mit Hilfe von Templates definiert die Scala zur Syntax nutzen und HTML-Seiten erzeugen. Es gibt 3 Views in dem System. Das Template *main.scala.html* wird von Play automatisch generiert und beinhaltet die Metadaten, wie beispielsweise den Pfad zum Stylesheet der Applikation. Dies wird von den anderen Templates integriert. Die View *index.scala.html* ist die Oberfläche für das Backend des Systems, in dem der Benutzer neue Artikel anlegen und alte löschen kann. Folgende Funktionen sind in der Oberfläche vorhanden.

- Formular zum Eintippen eines neuen Eintrags
- Löschen von Einträgen
- Link zu den Seiten von vorhandenen Einträgen

Wenn das Formular ausgefüllt und abgesendet wird, wird es an den Controller gesendet und ein neuer Artikel wird angelegt. Die View *article.scala.html* dient alleine dazu einen veröffentlichten Blogeintrag darzustellen. Unter dem Text des Artikels sollen die gefundenen RDF-Tripel angezeigt werden, so dass diese direkt verglichen werden können.

assets.stylesheets
In diesem Paket befindet sich lediglich das Stylesheet der Applikation. Das Stylesheet ist im LESS-Format angelegt, welches einige Erleichterungen Gegenüber CSS bietet. So ist es möglich Variablen zu definieren und im gesamten Stylesheet zu verwendet. LESS wird von Play von Haus aus unterstützt und wird zu CSS kompiliert.

4.2.2. Integration von UIMA

Nachdem die allgemeine Struktur und die Oberfläche des Systems erstellt worden sind soll im nächsten Schritt UIMA in das Projekt eingebunden werden. Dazu wurde zunächst UIMA als Bibliothek mit Hilfe von SBT eingebunden. Zusätzlich wurde noch ein Package

app.org.apache.uima angelegt. Dieses Package wird für die Klassen der von UIMA automatisch generierten Typen verwendet. Der Aufruf von UIMA geschieht durch eine neue Klasse im Controller namens *UIMACaller*. Dieser wird aufgerufen sobald ein neuer Artikel angelegt wird und stößt die UIMA-Pipeline an. Hier sollen später die Annotationen erzeugt und dem Artikel hinzugefügt werden. Im folgenden nochmals der komplette Ablauf (vgl Abbildung 4.6).

1. Der Nutzer ruft die Applikation auf und das Backend wird geladen.
2. Der Nutzer schreibt seinen Artikel und übermittelt das Formular an den Controller (Application)
3. Application ruft den UIMACaller auf und übergibt den Artikel
4. UIMA erzeugt die Annotationen und speichert sie im Artikel
5. Application übergibt den Artikel an die View um ihn anzuzeigen

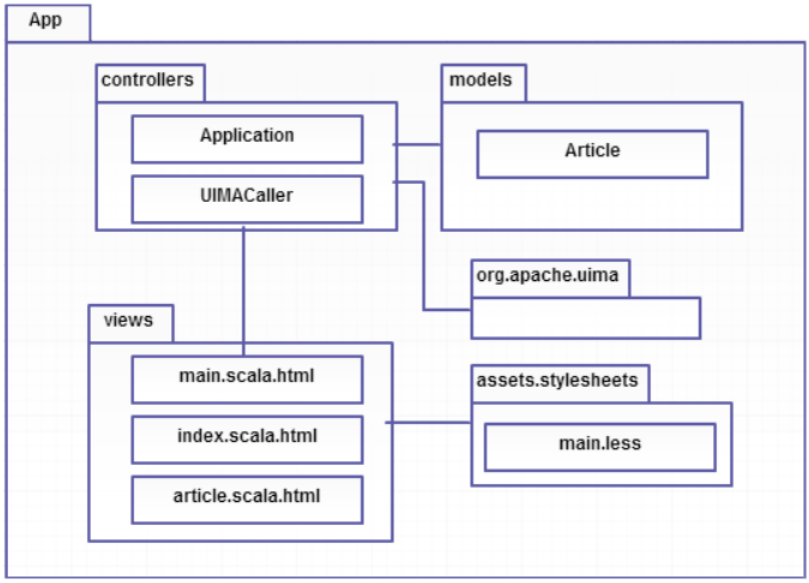

Abbildung 4.5.: Projektstruktur mit UIMA

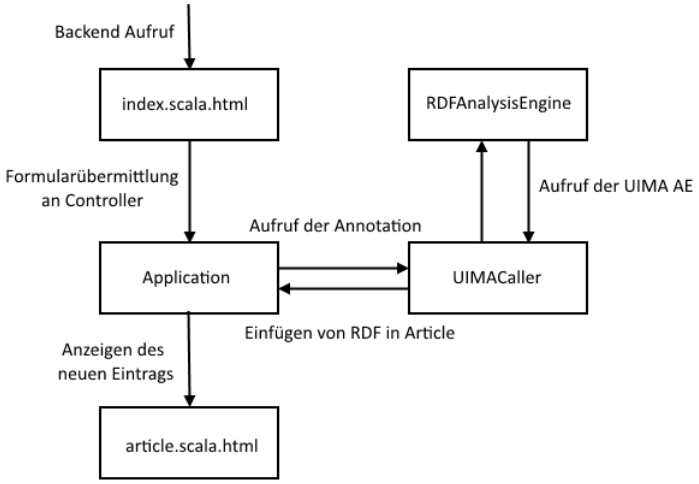

Abbildung 4.6.: Ablauf bei Anlegen eines Arikels

4.2.3. UIMA Pipeline

Im folgenden Abschnitt wird die Umsetzung des regelbasierten Ansatzes welcher im Analyseteil (3.4.1.2) bereits erarbeitet wurde vorgestellt. Es geht darum mit Hilfe von UIMA den Text zu untersuchen und Entitäten und Beziehungen zwischen diesen herauszufinden. Anschließend werden die erzeugten Annotationen noch so aufbereitet, dass nur noch die semantischen Informationen vorhanden sind. Im letzten Schritt der Pipeline wird dann das RDF erzeugt welches letztlich auf der Webseite durch Play angezeigt wird. Zu diesem Zweck wird eine Aggregate Analysis Engine verwendet welche weitere primitive AEs integriert, die wiederum für die einzelnen Schritte verantwortlich sind (siehe 4.7).

Schritt	Entsprechende Analysis Engine
Part of Speech Tagging	WhitespaceTokenizer & HmmTagger
Morphologische Analyse	Wird nicht durchgeführt
Finden von Entitäten	DictionaryAnnotator
Finden von Beziehungen	RegExAnnotator

Tabelle 4.1.: Ablauf von regelbasierten Systemen und entsprechende Analysis Engine in der Implementierung

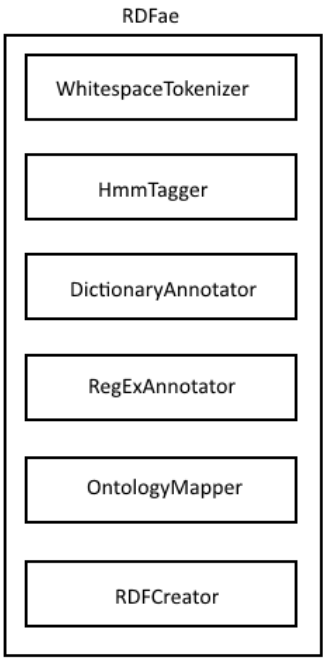

Abbildung 4.7.: Übersicht der Analysis Engine

4.2.3.1. Part of Speech Tagging

Im ersten Schritt wird das POS Tagging durchgeführt. Dazu muss der gesamte Text zunächst in Token unterteilt werden. Für diese Aufgabe bietet UIMA bereits eine fertige AE an[1], den *WhiteSpace Tokenizer*. Dieser einfache Tokenizer trennt die Token anhand der Leerzeichen zwischen den Worten. Zusätzlich werden noch ganze Sätze betrachtet und annotiert. Es entstehen die beiden Annotationen *TokenAnnotation* und *SentenceAnnotation*.

Aus Basis dieser Annotationen arbeitet nun der Part of Speech Tagger, welcher sequentiell nach dem Tokenizer aufgerufen wird. Es wird der *Hidden Markov Model Tagger* verwendet der ebenfalls als Addon für UIMA angeboten wird. Wie der Name bereits sagt arbeitet dieser Annotator mit Hidden Markov Modellen um Token mit einem Attribut *PoSTag* zu erweitern welches die Wortart beschreibt. Der eigentliche Lernprozess des Taggers würde dabei außerhalb von UIMA stattfinden. Der *Hidden Markov Model Tagger* beinhaltet bereits ein Modell für die englische Sprache, so dass ein manueller Lernprozess für weitere Sprachen in dieser Arbeit nicht behandelt wurde.

[1] https://uima.apache.org/sandbox.html

```
Eingabetext: Java is a programming language.

Ergebnis:
SentenceAnnotation begin: 0 end: 31
TokenAnnotation begin: 0 end: 4 tokenType: <null> posTag: "nn"
TokenAnnotation begin: 5 end: 7 tokenType: <null> posTag: "bez"
TokenAnnotation begin: 8 end: 9 tokenType: <null> posTag: "at"
TokenAnnotation begin: 10 end: 21 tokenType: <null> posTag: "jj"
TokenAnnotation begin: 22 end: 30 tokenType: <null> posTag: "vbg"
TokenAnnotation begin: 30 end: 31 tokenType: <null> posTag: "."
```

Listing 4.1: Beispielsausgabe nach POS Tagging

4.2.3.2. Finden von Entitäten

Als nächsten sollen die relevanten Entitäten in dem Text identifiziert werden. Dazu werden in diesem Prototypen Wörterbücher verwendet, welche die bereits bekannten Entitäten beinhalten. Je nach dem wie komplex die Domäne ist, wäre es in einem produktiven System sinnvoll solche Wörterbücher möglichst automatisiert zu erzeugen. In der Umsetzung in dieser Arbeit beschränkt sich die automatische Erweiterung allerdings auf die gefundenen Beziehungen zwischen den Entitäten. Um die *Named Entity Recognition* durchzuführen wird der *Dictionary Annotator* von UIMA verwendet.

Um den *Dicitonary Annotator* zu verwenden müssen zunächst Wörterbücher definiert werden. Es werden Wörterbücher für *Programming Language*, *Framework* und *Application* benötigt. Der *Dictionary Annotator* bietet ein Konsolenkommando an um aus einfachen Listen die benötigten XML-Dateien zu erzeugen:

```
Java DictionaryCreator -input Frame_Dict.txt -encoding UTF-8
-output Frameworks_Dict.xml
```

Listing 4.2: Beispielaufruf der Wörterbucherzeugung

Mit diesem Kommando wird aus den Begriffen in der Datei Frame_Dict.txt ein strukturiertes Wörterbuch für die Frameworks erzeugt. Dies geschieht bevor das System zum ersten mal gestartet wird. Der spätere Benutzer muss lediglich den Artikel schreiben der dann anhand der zuvor erzeugten Wörterbücher analysiert wird.

```
Play
UIMA
Ruby on Rails
```

```
Grails
Spring
Zend
Hibernate
Selenium
```

Listing 4.3: Beispielinhalt von Frame_Dict.txt

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary xmlns="http://incubator.apache.org/uima"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="dictionary.xsd">
        <typeCollection>
                <dictionaryMetaData caseNormalization="true"
                multiWordEntries="true" multiWordSeparator="|"/>
        <typeDescription>
                <typeName>org.apache.uima.Framework_Entry</typeName>
        </typeDescription>
        <entries>
                <entry>
                        <key>Play</key>
                </entry>
                <entry>
                        <key>UIMA</key>
                </entry>
                <entry>
                        <key>Ruby|on|Rails</key>
                </entry>
                <entry>
                        <key>Grails</key>
                </entry>
                <entry>
                        <key>Spring</key>
                </entry>
                <entry>
                        <key>Zend</key>
                </entry>
                <entry>
                        <key>Hibernate</key>
                </entry>
                <entry>
```

```
34                          <key>Selenium</key>
35                  </entry>
36          </entries>
37 </typeCollection>
38 </dictionary>
```

Listing 4.4: Beispielinhalt von Frameworks_Dict.xml

Es ist erkennbar, dass auch Begriffe, die aus mehreren Worten bestehen von dem Annotator gefunden werden können. Der Typ *org.apache.uima.Framework_Entry* ist ein UIMA-Typ welcher selbst definiert wird und sowohl in das Wörterbuch als auch die Analysis Engine von UIMA eingetragen werden muss. Dieser Typ wird für die Annotation der Entitäten verwendet und kann auch durchaus weitere Attribute beinhalten welche von nachfolgenden AE's befüllt werden können.

Ein Nachteil dieser Struktur ist allerdings, dass dadurch für jeden möglichen Entitätstypen ein eigenes Wörterbuch angelegt werden muss. Die Wörterbücher für *Programming Language* und *Application* sind analog zu dem hier gezeigten umgesetzt worden.

4.2.3.3. Finden von Beziehungen und Attributen

Für die nun gefundenen Entitäten sollen nun die Beziehungen, welche später in RDF abgebildet werden sollen in dem gegebenen Text gefunden werden. Wie bereits erwähnt werden zu diesem Zweck Muster bzw. Regeln verwendet. Die Form der gegebenen Muster sind als reguläre Ausdrücke gegeben. Diese können dann auf den Text angewendet werden und die Übereinstimmungen werden mit der gewünschten Beziehung bzw. Annotiert. Neben Beziehungen sollen auch Attribute von Entitäten gefunden werden, wie beispielsweise das Erscheinungsjahr einer Programmiersprache. Leider scheint es mit dem hier verwendeten Annotator nicht möglich bereits vorhandene Annotationen sinnvoll zu erweitern. Aus diesem Grund wurden diese Eigenschaften als zusätzliche Annotation definiert und im weiteren Verlauf der Verarbeitung vereinigt.

Auch um anhand von regulären Ausdrücken Annotationen zu generieren gibt es bereits eine Analysis Engine von UIMA. Es wird der *Regular Expression Annotator* verwendet. Dieser arbeitet recht ähnlich zu dem vorangegangenen *Dictionary Annotator*, in dem die möglichen Typen von Annotatioen in UIMA definiert werden und die Konfiguration, sowie die Muster aus einer XML herausgelesen werden.

Um die Realisierung zu veranschaulichen wird eine Beispielbeziehung betrachtet. Die gewählte Beziehung beschreibt ob ein Framework für eine Programmiersprache verfügbar ist (siehe 4.8). Die Beziehung soll nun anhand eines Beispielsmusters gefunden werden. In der endgültigen Implementierung wurde die Generierung dieser regulären Ausdrücke anhand von Musterfindung (siehe Abschnitt 4.2.4) umgesetzt, um die Vorgehensweise darzustellen soll jedoch ein Beispiel von Hand genügen. Um ein mögliches Muster zu finden muss erst

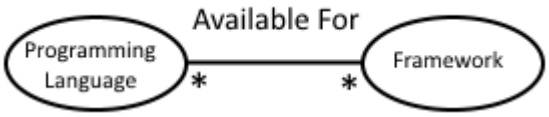

Abbildung 4.8.: Visualisierung einer Beziehung zwischen Programmiersprache und Framework

ein Beispielsatz gefunden werden welcher auf die Beziehung schließen lässt. Nimmt man den Satz *„Play is available for Java.“* lässt sich die Beziehung klar herauslesen. Um dies nun in ein Muster umzuwandeln muss der Satz zuerst verallgemeinert werden, also: *„X is available for Y.“* Nun muss dieser Satz lediglich noch als regulärer Ausdruck formuliert werden. Das endgültige Muster ist entsprechend:

```
([^\s]+)\sis\savailable\sfor\s([^\s]+)
```

Interessant ist das dieses Muster nicht die zuvor betrachteten Entitätsklassen beinhaltet. Dies ergab sich aus einer Beschränkung des verwendeten Annotators bei dem keine Entitäten in die Muster integriert werden konnten. Aus diesem Grund wird in einem späteren Schritt das Ergebnis des Musters mit dem der *Named Entity Recognition* verglichen um gültige Treffer zu finden. Ein idealeres Muster wäre *<Framework_Entry>\sis\savailable\sfor\s<Programming_Entry>*. Um dies zu erreichen müsste jedoch eine neue Analysis Engine selbst erstellt werden.

Um das gefundene Muster nun in UIMA umzusetzen muss wie zuvor ein UIMA-Typ definiert werden. In diesem Beispiel sei dies *org.apache.uima.AvailableForRelation*. Nun kann die XML des Annotators angepasst werden. Diese ist die *concepts.xml* und beschreibt alle möglichen regulären Ausdrücke. Im folgenden ein Auszug aus einer von Hand erstellten Konfiguration.

```
<concept name="isAvailableFor">
        <rules>
                <rule regEx="([^\s]+)\sis\savailable\sfor\s([^\s]+)"
                matchStrategy="matchAll"
                matchType="uima.tcas.DocumentAnnotation" />
```

```
        </rules>
        <createAnnotations>
         <annotation type="org.apache.uima.AvailableForRelation">
                <begin group="0" />
                <end group="0" />
                <setFeature name="Framework" type="String">$1
                </setFeature>
                <setFeature name="Language" type="String">
                $2</setFeature>
                </annotation>
        </createAnnotations>
</concept>
```

Listing 4.5: concepts.xml

Der Aufbau der XML ist einfach nachzuvollziehen. Erwähnenswert ist, dass die Attribute (oder Features) einer Annotation mit Teilen des auf dem Regulären Ausdruck passenden Textes befüllt werden können, in dem die sogenannten *Capture Groups* von Regulären Ausdrücken verwendet werden (im Beispiel $1 und $2 für die beiden geklammerten Bereiche des Ausdrucks). Im folgenden werden einige weitere ausgewählte Möglichkeiten die der *Regular Expression Annotator* anbietet um Muster zu definieren gezeigt.

- Mit dem Wert *matchType* kann festgelegt werden ob eine Regel auf einen gesamten Text angewendet werden soll oder nur auf Text, der bereits mit einer Annotation versehen wurde. *DocumentAnnotation* umschließt dabei immer den gesamten Text.
- *<variable name="xy">* erleichtert die Erstellung der regulären Ausdrücke, in dem Variablen definiert werden.
- *<ruleExceptions>* sind ein Konzept um die Annotierung gezielt zu verhindern, zum Beispiel wenn bereits eine andere Annotation den Text umgibt oder der Kontext des Satzes die Annotation ausschließt.

Nun soll eine Übersicht bereitgestellt werden, welche UIMA-Annotationen anhand der Regeln in der Implementierung erstellt. Dabei gibt es durchaus mehrere Regeln um die gleichen Annotationen zu erstellen. In der ersten Version der Implementierung sind diese von Hand erstellt worden. Später wurde die Musterfindung verwendet um die Regeln automatisiert zu erzeugen (siehe 4.2.4).

Beziehungen
WrittenIn
AvailableFor

Tabelle 4.2.: Beziehungen die gefunden werden

Attribute
Programming_Language_Entry:Developer
Programming_Language_Entry:Version
Programming_Language_Entry:Year
Programming_Language_Entry:Paradigm
Framework_Entry:Developer
Framework_Entry:Version
Application_Entry:Developer
Application_Entry:Type

Tabelle 4.3.: Attribute die gefunden werden. Diese werden zunächst auch als Beziehung modelliert

4.2.3.4. Verbinden mit einer Ontologie

Nachdem die linguistische Analyse nun abgeschlossen ist, sollen die gesammelten Informationen als nächstes mit einer Ontologie verknüpft werden um die Trennung zwischen Linguistik und Semantik zu verdeutlichen. Dazu wurde zunächst eine einfache Ontologie von Hand definiert welche die möglichen Konzepte beschreibt.

```
<?xml version="1.0"?>
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:cd="http://customNamespace/">

        <rdfs:Class rdf:ID="Programming_Entry">
                <cd:Developer />
                <cd:Version />
                <cd:Year />
                <cd:Paradigm />
        </rdfs:Class>
```

```
13
14          <rdfs:Class rdf:ID="Framework_Entry">
15                  <cd:Developer />
16                  <cd:Version />
17          </rdfs:Class>
18
19          <rdfs:Class rdf:ID="Application_Entry">
20                  <cd:Developer />
21                  <cd:Type />
22          </rdfs:Class>
23
24          <rdf:Property rdf:ID="WrittenIn" />
25          <rdf:Property rdf:ID="AvailableFor" />
26  </rdf:RDF>
```

Listing 4.6: Selbstdefinierte Ontologie zu Softwareentwicklung als RDFS

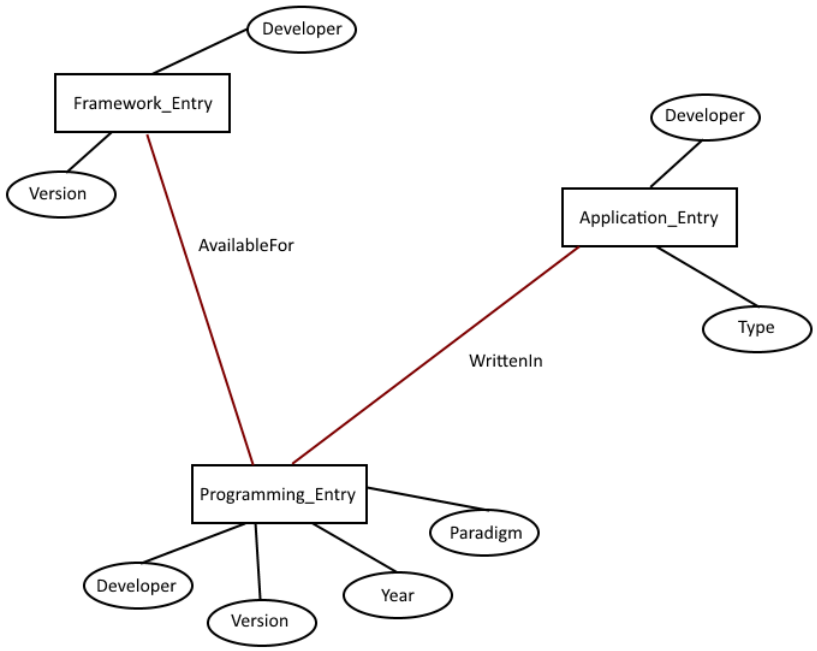

Abbildung 4.9.: Visualisierung der Ontologie inklusive der Position der Beziehungen

Diese Ontologie beschreibt die Struktur welche verwendet werden soll. In diesem Beispiel gibt es also die drei Entitäten *Programming_Entry*, *Framework_Entry* und *Application_Entry* jeweils mit einer Reihe von Attributen. Zusätzlich gibt es noch die zwei Beziehungen *WrittenIn* und

AvailableFor. Es ist durch die Ontologie nicht festgelegt für welche Entitäten die Beziehungen gelten können. Abbildung 4.9 zeigt jedoch die vorgesehene Struktur. Properties werden in RDF nicht an Konzepte geknüpft um eine möglichst hohe Wiederverwendung zu ermöglichen. Die Ontologie im RDFS-Format dient dazu festzulegen welche Konzepte, Attribute und Beziehungen in der Domäne vorhanden sind. Der selbst entwickelte UIMA-Annotator *OnthologyMapper* kann dann anhand dieser Informationen die semantischen Annotationen von den anderen Unterscheiden.

Um diese Unterscheidung durchzuführen wurde in dieser Umsetzung zunächst mit Namenskonventionen gearbeitet. Der Name eines Konzeptes in der Ontologie muss entsprechend mit dem Namen der Annotation korrelieren. Zusätzlich sorgt der *OnthologyMapper* dafür das die Attribute, welche im vorherigen Schritt noch als eigene Annotation definiert werden mussten, zusammengefasst werden (siehe Abbildung 4.10). Im folgenden nochmals eine Auflistung des konkreten Ablaufs dieser Analysis Engine.

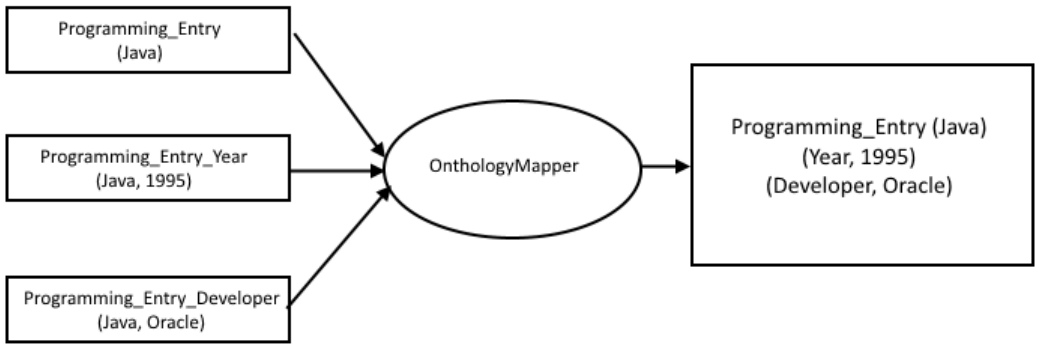

Abbildung 4.10.: Beispiel für das Zusammenfügen von Annotationen durch den OnthologyMapper

Einlesen der Ontologiedatei

Zuerst wird die Ontologie im RDFS-Format eingelesen. Es ist eine Namenskonvention festgelegt worden nach der die Bezeichnungen der Konzepte mit den Annotationen in UIMA korrelieren müssen. Für die *Programming_Entry* gibt es entsprechend einen gleichnamigen UIMA-Typ. Das gleiche gilt für Annotationen welche die Attribute darstellen. Das Attribut *Year* der Klasse *Programming_Entry* entspricht der Annotationsklasse *Programming_Entry_Year*. Das Konvention lautet daher *Konzept_Attribut*. Diese Konvention hat sich als notwendig ergeben aufgrund

der bereits in 4.2.3.3 beschriebenen Einschränkung die regulären Ausdrücke nicht an Entitäten binden zu können.

Filtern von nicht semantischen Annotationen

Der Ontologymapper geht alle bisher für den Text erzeugten Annotationen durch und entscheidet ob es sich dabei um semantische oder linguistische Informationen handelt. Dies geschieht in dem nur Annotationen in den weiteren Ablauf übernommen werden, welche den Konzepten in der Ontologie entsprechen. Zusätzlich werden Attributtypen für diese Konzepte anhand des Namens erkannt. Für ein Konzept *Programming_Entry* werden Annotationen wie *Programming_Entry* und *Programming_Entry_Year* als semantisch eingestuft während *TokenAnnotation* verworfen wird. Sinn hinter dieser Filterung ist es sicherzustellen, dass später kein RDF für nicht semantisch relevante Inhalte erzeugt wird.

Vereinigen von Attributen mit Entitäten

Nach der Filterung werden die Attribute noch an die Entitäten angehängt. Anhand der vorher definierten Namenskonvention können Attribute und Entität einander zugeordnet werden. Der Inhalt des Attributs wird in die Annotation für die Entität eingetragen, so dass nur noch eine einzige Annotation die gesamte Entität beschreibt (siehe 4.10). Die Annotation des Attributes (z.B *Programming_Entry_Year*) kann danach verworfen werden. Die nun erzeugte Annotation beinhaltet alle verfügbaren Informationen und kann im weiteren Schritt in RDF umgewandelt werden.

Am Ende dieser Schritte sind lediglich noch semantische Informationen vorhanden, welche durch die Ontologie zumindest in ihrer Struktur validiert wurden. Der Vorteil dieses Ansatzes ist es, dass die Semantik nach der gefiltert wird lediglich durch die Ontologie beschrieben wird. Die verwendete Analysis Engine ist daher unabhängig von einer konkreten Domäne und kann wiederverwendet werden.

4.2.3.5. Generieren von RDF

Nachdem nun alle Semantischen Informationen gefunden und strukturiert worden sind soll nun im letzten Schritt die Generierung von RDF durchgeführt werden. Hierfür wurde ein eigener *CASConsumer* implementiert, der *RDFCASCreator*. Ein *CASConsumer* ist eine spezielle Art von UIMA Analysis Engine bei dem keine neuen Annotationen erzeugt werden sondern stattdessen bereits vorhandene in irgendeiner Form konsumiert werden. Entsprechend werden die semantischen Annotationen welche vom *OnthologyMapper* weitergegeben werden hier

konsumiert und in das RDF-Format gebracht. Um dies zu realisieren beinhaltet der *RDFCAS-Creator* ein Template für RDF in das die Informationen hineingegeben werden. Einige Attribute aus den Annotationen wie die Start- und Endposition müssen hierbei allerdings nochmals herausgefiltert werden. Das Endprodukt dieses Schrittes ist eine Datei in dem valides RDF steht. Der *UIMACaller*, welcher die Verarbeitung angestoßen hat liest nun diese Datei ein gibt den Text an das Webfrontend weiter.

```
<?xml version="1.0"?>
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:cd="http://www.blog.fake/cd#">

<rdf:Description rdf:about=http://www.blog.fake/cd/java>
  <cd:Developer>Oracle</cd:Developer>
  <cd:Year>1995</cd:Year>
  <cd:Version>8</cd:Version>
  <cd:Paradigm>object-oriented</cd:Paradigm>
  </rdf:Description>
</rdf>
```

Listing 4.7: Beispielhaftes RDF für die Programmiersprache Java

4.2.4. Erweiterung um Musterfindung

Nachdem nun die Implementierung grundlegend fertiggestellt ist soll nun die bereits angesprochene Erweiterung um die in Abschnitt 3.4.1.1 vorgestellte Musterfindung erweitert werden. Dabei geht es darum die Regeln zum Finden von Attributen und Beziehungen zwischen Entitäten automatisch zu Erstellen. Gelöst wird das mit dem bereits vorgestelltem Verfahren nach Brin (1999) welches im Kern die folgenden Schritte beinhaltet.

1. Anlegen eines Seeds aus Tupeln für das gesuchte Attribut bzw. die Beziehung.
2. Suchen nach vorkommen der Tupel im Text und Extraktion des Kontextes.
3. Umwandeln des Kontextes in einen regulären Ausdruck.
4. Schreiben der neuen Regeln in die concepts.xml des *Regular Expression Annotators*.
5. Vergrößern des Seeds mit Hilfe der neuen Regeln.

Diese Erweiterung wurde auf der Ebene des Controllers im Projekt implementiert und wird vor der eigentlichen UIMA-Pipeline angestoßen. Dies ermöglicht die Unabhängigkeit zum grundlegenden Verfahren falls die Regeln auf andere Weise generiert werden sollen.

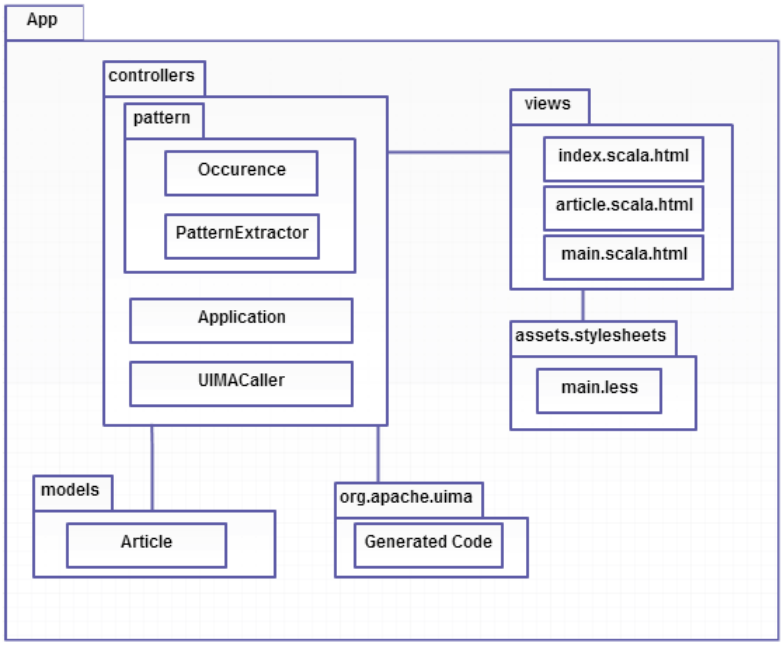

Abbildung 4.11.: Finale Projektstruktur

4.2.4.1. Anlegen von Seeds

Als erstes wird für jedes zu findende Attribut ein Seed angelegt welcher Vorkommen der zu findenden Eigenschaft beinhaltet. Die Dateien befinden sich im Unterverzeichnis *pattern/-seed* im Projekt. In jeder der Dateien steht in der ersten Zeile jeweils die Annotation, sowie die Eigenschaften auf die sich der Seed bezieht. Nun ein Beispiel aus der Datei *programming_entry_developer_seed.txt*.

```
1 org.apache.uima.Programming_Entry_Developer,Programming_Entry,Developer
2 java,oracle
3 c#,microsoft
4 delphi,borland
```

Listing 4.8: Beispielhafter Seed

Die Datei beschreibt bereits bekannte Programmiersprachen, sowie ihre Entwickler. Anhand dieser Beispieldaten sollen nun in den nächsten Schritten Muster gefunden werden. Diese grundlegenden Dateien müssen vor dem Start des Systems angelegt werden und mit einigen Tupeln befüllt werden.

4.2.4.2. Kontextextraktion

Der folgende Schritt wird durchgeführt wenn ein Benutzer einen Artikel abgeschickt hat. In dem Text wird nach allen Paaren der Seeds gesucht. Dies geschieht mit einfachem Vergleichen der Strings. Werden beide Werte eines Paares gefunden, so wird der Text dazwischen als Kontext gespeichert. Der Kontext bezeichnet hier also den Zusammenhang in dem die Begriffe stehen. Man könnte daher auch sagen, dass der Kontext die gesuchte Beziehung in irgendeiner Form beschreibt und daher für die Erzeugung von Regeln essentiell ist. Die hier vorgestellte Variante der Extraktion bei der lediglich der Text zwischen den Begriffen verwendet wird ist allerdings recht simpel, da beispielsweise Satzübergänge nicht berücksichtigt werden. Um zu vermeiden, dass ganze Paragraphen als Kontext übernommen werden wurde ein Parameter eingeführt welcher die maximale Zeichenanzahl zwischen den beiden Begriffen begrenzt. Dies erhöht die Wahrscheinlichkeit, dass der Kontext tatsächlich relevant ist. Ferner gehört es zum Kontext welcher der beiden Begriffe am Anfang und am Ende steht. So ergeben die beiden Sätze *„Java is being developed by Oracle."* und *„Oracle is the developer of Java."* beide einen Kontext für die gleiche Beziehung. Die Leserichtung ist aber jeweils unterschiedlich.

4.2.4.3. Erzeugen von regulären Ausdrücken

Mit Hilfe des Kontextes kann nun ein regulärer Ausdruck generiert werden. Dazu wird der Kontext schlicht in einen fixen Ausdruck umgeformt. Zusätzlich werden am Anfang und Ende noch Variablen angehängt (siehe Abbildung 4.12). Es muss allerdings noch entschieden

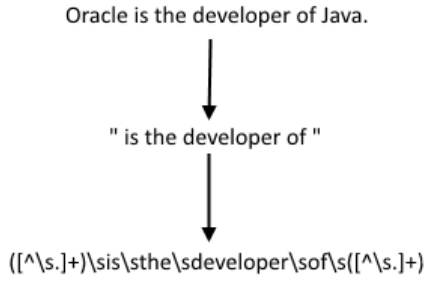

Abbildung 4.12.: Entwicklung von Satz zu regulären Ausdruck

werden ob das gefundene Muster in das System integriert werden soll. Um die Qualität von Mustern zu verbessern wurde ein Parameter für den Threshold implementiert. Dieser bestimmt, dass ein Muster nur übernommen wird, wenn es mehrere Male in der Gesamtlaufzeit des Systems gefunden wurde. Auf den genauen Einfluss dieses Parameters wird in Kapitel 5 konkret eingegangen. Um diesen Threshold zu realisieren werden alle Muster zwischengespeichert. Da UIMA bereits viel mit Dateien arbeitet zur Konfiguration wurde auch hier auf eine einfache Textdatei zurückgegriffen.

4.2.4.4. Schreiben der concepts.xml

Nun können die gefundenen Muster in die concepts.xml übertragen werden. Dazu muss lediglich jedes Muster an die Datei nach der in Abschnitt 4.2.3.3 vorgestellten Struktur angehängt werden. Dieser Schritt ist allerdings insofern von besonderer Bedeutung, da die concepts.xml die alleinige Schnittstelle zu der UIMA-Pipeline darstellt. Anhand dieses Schrittes gelangen die Muster in das Gesamtsystem.

4.2.4.5. Erweiterung des Seeds

Im letzten Schritt geht es darum, die Qualität bei zukünftigen Texten noch weiter zu verbessern. Das heißt es soll aus den bereits gefundenen Informationen gelernt werden für die Zukunft. Um dies zu erreichen wird nochmals der Text betrachtet. Dabei wird nun jedoch nicht versucht von den vorhandenen Seeds auf neue Muster zu schließen, sondern es sollen umgekehrt neue Tupel gefunden werden aus den nun erschlossenen Mustern. Dazu muss lediglich jedes neue Muster auf den Text angewendet werden und alle passenden Werte können in den Seed aufgenommen werden. Diese Abhängigkeit von Seed zu Muster und umgekehrt erlaubt es dem System immer weiter zu lernen.

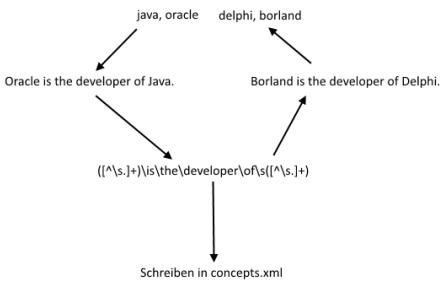

Abbildung 4.13.: Beispielhafter Ablauf der Musterfindung

5. Bewertung

In diesem Abschnitt soll das implementierte System anhand der bereits in Kapitel 3 definierten Anforderungen bewertet werden. Für die qualitative Bewertung der Musterfindung werden einige Testläufe mit unterschiedlichen Konfigurationen durchgeführt. Dadurch sollen noch existierende Problemstellungen gefunden werden. Im Anschluss soll noch betrachtet werden welche Formen von Informationen in einem Text erkannt werden und ob es Strukturen in Texten gibt die prinzipielle Probleme darstellen.

5.1. Betrachtung anhand der Anforderungen

Zunächst sollen nun die gewählten Anforderungen nochmals im Kontext der Implementierung betrachtet werden.

Automatisierung
Der Automatisierungsgrad ist von besonderer Bedeutung für das System. Durch die Anwendung von *Named Entity Recognition* und regulären Ausdrücken können die relevanten Entitäten der Domäne aus einem Text automatisch extrahiert und annotiert werden. Dadurch ist es nicht nötig den Text von Hand zu annotieren.

Eine vollständige Automatisierung ist aufgrund des benötigten Domänenwissens nicht machbar. Es werden Wörterbücher benötigt welche vor dem Start des Systems angelegt werden müssen. Auch eine Ontologie wird benötigt um die Struktur der Domäne zu beschreiben. Durch die Erweiterung auf Musterfindung lassen sich die regulären Ausdrücke automatisch erzeugen, wodurch ein weiterer Aspekt des Systems automatisiert wird. Allerdings benötigt diese Erweiterung vor dem ersten Durchlauf einen Seed mit einigen bekannten Tupeln der Beziehungen.

Alle benötigten manuellen Informationen werden bereits vor dem ersten Start des Systems angelegt und sind daher im Vergleich zu einer ständigen manuellen Erfassung jedes Textes vom Aufwand her vernachlässigbar. Dennoch gibt es bereits Ansätze um den Automatisie-

rungsgrad noch weiter zu verbessern. Tabelle 5.1 zeigt eine Übersicht der Automatisierung, sowie potenzielle Erweiterungen.

Prozess	automatisch/manuell	mögliche Verbesserung
Erstellen der Ontologie	manuell	semiautomatische Generierung (Maedche und Staab (2001))
Erstellen der Wörterbücher	manuell	vorhandene Wörterbücherlisten integrieren
Erstellen der Seeddateien	manuell	Mögliche Extraktion aus Ontologie
Token / POS Tagging	automatisch	-
Finden von Entitäten	automatisch	-
Finden von Beziehungen	automatisch	-
Schreiben von RDF	automatisch	-

Tabelle 5.1.: Übersicht von automatisierten Abläufen in der Implementierung

Komplexität
Die Laufzeitkomplexität des Systems hat sich nicht als Problem erwiesen. Es ergaben sich hinsichtlich der Laufzeit keine besonderen Ereignisse bei den Tests.

Generalisierbarbeit
Bei der Umsetzung des Blogggingportals ist darauf geachtet worden, dass die Implementierung prinzipiell nicht von der konkreten Domäne der Softwareentwicklung abhängig ist. Um dies zu veranschaulichen muss erst betrachtet werden an welchen Punkten die Domäne mit dem System interagiert.

- UIMA-Annotationen mit semantischer Bedeutung wie *Programming_Entry*, *Framework_Entry* und *AvailableFor*.
- Generieren von Mustern anhand der Seeds.
- Finden von Entitäten mit Wörterbüchern.
- Mappen der semantischen Annotationen auf Basis der eingelesenen Ontologie.

Um zu bewerten wie generisch das System ist erscheint es sinnvoll den Aufwand zu betrachten der von Nöten ist um den Blog auf eine neue Domäne anzuwenden. Zuerst ist es unumgänglich, dass mit Hilfe von UIMA Annotationsklassen angelegt werden welche die neuen Entitäten darstellen. Dies wird von UIMA vorausgesetzt für den weiteren Prozess. Die Definition ist dabei einfach und der Javacode lässt sich automatisch erzeugen.

Um die Musterfindung verwenden zu können sollten nun Seeds angelegt werden welche die Beziehungen exemplarisch beschreiben. 2 bis 3 Beispiele genügten hier in Tests, weitere werden im Verlauf der Anwendung entdeckt und hinzugefügt. Nun müssen noch Wörterbücher angelegt werden um die *Named Entity Recognition* durchführen zu können. Hier reichen einfache Listen, die durch ein Konsolenkommando in ein XML-Format gebracht werden (siehe Abschnitt 4.2.3.2). Als letztes muss noch die Ontologie der neuen Domäne definiert werden, damit das System zuordnen kann welche der Annotationen semantisch relevant ist un diese richtig zusammenführen kann.

Geht man diese Schritte durch kann man erkennen, dass der konkrete Code des Systems nicht angepasst werden muss um eine Änderung der Domäne vorzunehmen. Eine Verallgemeinerung auf weitere Bereiche ist also mit recht geringem Aufwand gegeben. Dennoch müssen Anpassungen an mehreren Stellen vorgenommen werden um eine Änderung vorzunehmen. Ein Optimum scheint hier noch nicht erreicht zu sein. Im besten Fall müsste eine neue Domäne nur an einer Stelle in dem System definiert werden. Dazu bietet sich die Ontologie an, da Ontologiesprachen wie RDFS oder OWL in der Lage sind eine Vielzahl von Konzepten in allgemeiner Form abzubilden. Ein zukünftiges System könnte dies dahingehend zentralisieren, dass Seeds, Wörterbücher und Annotationsklassen auf Basis einer Ontologie erzeugt werden. Dies würde das System generischer machen. Gleichzeitig stiegen jedoch auch die Anforderungen an die Ontologie, welche derzeit nur strukturelle Informationen liefert.

Unabhängigkeit von weiteren Systemen
Der Blog verwendet keine externen Systeme, da das komplette Wissen durch eine gegebene Ontologie, Wörterbücher und Tupel von Beziehungen modelliert wird.

Unterstützung verschiedener Sprachen
Eine konkrete Abhängigkeit von einer bestimmten Sprache besteht lediglich für den *Part of Speech Tagger*, welcher auf Basis von *Hidden Markov Modellen* arbeitet. Dieser wird bereits mit einem Modell für Englisch und Deutsch angeboten. Wenn allerdings eine weitere Sprache benötigt wird kann der POS Tagger diese erlernen. Die weiteren Aspekte der Implementierung sind prinzipiell unabhängig von der Sprache. Allerdings versteht es sich, dass gefundene Muster nur für die Sprache anwendbar sind in der sie auch gefunden werden. Eine tatsächliche Mehrsprachigkeit ist also nicht gegeben.

5.2. Qualitative Bewertung

5.2.1. Qualität der regulären Ausdrücke

Um die Qualität der semantischen Annotationen zu beurteilen muss betrachtet werden wie viele der im Text vorhandenen und relevanten Informationen durch das System gefunden werden. Zentral hierfür ist die Qualität der regulären Ausdrücke, welche durch die Musterfindung erzeugt werden. Da keine umfangreiche Quelle für Beispieltexte vorhandenen ist wurde die Bewertung vor allem auf qualitativer Ebene vorgenommen. Es wird also analysiert welche Formen von Beziehungen und Attributen gefunden werden, anstatt sich auf eine numerische Bewertung anhand von *Precision* und *Recall* zu fokussieren.

Für den Testlauf wurden exemplarisch 10 relevante Artikel in das System eingegeben und annotiert. Danach wurde dies in der gleichen Reihenfolge wiederholt mit anderer Konfiguration von Threshold und maximaler Distanz. Der Threshold gibt an wie oft ein Muster gefunden werden muss bevor es tatsächlich übernommen und angewendet wird. Die maximale Distanz gibt an wie lang ein regulärer Ausdruck werden darf. Es wurden insgesamt drei Testläufe durchgeführt.

```
AvailableForRelation V1 = Framework, V2 = Programming language:
V1\sis\sa\sweb\sapplication\sframework\srunning\son\sthe\sV2
V1\.?\sthere\sare\sseveral\scurated\slists\sof\sonline.....\sV2
V1\sand\sV2
V2\.?\swe\sassume\sthat\syou\salready\sknow\swhat\sV1
V2\sapis\swith\sV1
V1\sis\savailable\sfor\sV2
V1\.?\swhen\syou\sare\sready,\show ever,\sV2
V2\sand\sleveraging\sV1
V2\sprogrammers\swith\sa\sneed\sto\sunderstand\sthe\sV1
V1\sorm\s(hibernate\sin\sshort)\sis\san\sobject-relational.....\sV2
V1s\sprimary\sfeature\sis\smapping\sfrom\sV2
V1\sclasses\sto\sdatabase\stables\s(and\sfrom\sjava\sdata....\.?\sV1

WrittenInRelation V1 = Application, V2 = Programming language:
V1\sis\san\side\sdeveloped\sin\sV2

Framework_Entry_Developer V1 = Framework, V2 = Developer:
V2\sreactive\splatform\.?\syou....\susing\sthe\sjava\sapis\swith\sV1

```

```
21 Programming_Entry_Paradigm V1 = Programming language, V2 = Paradigm:
22 V2\sparadigms\sand\sretains\sfull\sinteroperability\swith\...\sV1
23 V1\sis\sa\scomputer\sprogramming\slanguage\sthat\sis.....\sV2
24 V1\sis\sa\sV2
25 V1\slanguage,\sproviding\sa\sframework\sfor\smapping\san\sV2
26 V2,\sand\suses\sa\scurly-brace\ssyntax\sreminiscent\sof...\sV1
27
28 Programming_Entry_Developer V1 = Programming language, V2 = Developer:
29 V2\scorporation\sand\sreleased\sin\s1995\sas\sa....\sV1
30 V1\swas\sdeveloped\sby\sV2
31
32 Programming_Entry_Year V1 = Programming language, V2 = Year:
33 V1\swas\screated\sin\sV2
```

Listing 5.1: Muster nach dem Testdurchlauf mit Threshold von 0 und maximaler Distanz von 100 Zeichen. Einige Muster sind für die Darstellung gekürzt

Im ersten Durchlauf wurde das System so wenig wie möglich eingeschränkt. Daher wurde ein Threshold von 0 gewählt. Jedes potenzielle Muster wurde also sofort als valide angesehen. Gleichzeitig lag die maximale Zeichendistanz bei 100. Listing 5.1 zeigt die nach dem ersten Durchlauf generierten regulären Ausdrücke und zu welchen Beziehungen sie gehören. Auch wenn es zunächst so scheint als seien auf diese Weise eine Vielzahl an Mustern erzeugt worden so stellt man bei genauerem hinsehen fest, dass eine Reihe dieser Muster viel zu lang und konkret ist. In Zeile 13 findet sich ein Muster welches den Namen der Programmiersprache Java im Kontext beinhaltet. Eine solche Umschließung von Entitäten ist nicht wünschenswert, da dadurch die Allgemeinheit vollkommen verlorengeht. Aus diesem Grund scheint es so als sei eine große Anzahl der in dieser hier gefundenen Muster wertlos, da es voraussichtlich keinen Text gibt auf den sie sinnvoll zutreffen.

```
1 AvailableForRelation V1 = Framework, V2 = Programming language:
2 V1\sand\sV2
3 V2\sapis\swith\sV1
4 V1\.?\swhen\syou\sare\sready,\showever,\sV2
5 V1\sis\savailable\sfor\sV2
6 V2\sand\sleveraging\sV1
7 V1s\sprimary\sfeature\sis\smapping\sfrom\sV2
8
9 WrittenInRelation V1 = Application, V2 = Programming language:
10 V1\sis\san\side\sdeveloped\sin\sV2
```

```
11
12 Programming_Entry_Paradigm V1 = Programming language, V2 = Paradigm:
13 V1\sis\sa\sV2
14
15 Programming_Entry_Developer V1 = Programming language, V2 = Developer:
16 V1\swas\sdeveloped\sby\sV2
17
18 Programming_Entry_Year V1 = Programming language, V2 = Year:
19 V1\swas\screated\sin\sV2
```

Listing 5.2: Muster nach dem Testdurchlauf mit Threshold von 0 und maximaler Distanz von 30 Zeichen.

Der nächste Test wurde mit einer Konfiguration von einem Threshold von 0 und einer maximalen Distanz von 30 Zeichen durchgeführt. Wie man in Listing 5.2 erkennen kann sind die resultierenden Muster dementsprechend eine Teilmenge derer aus dem ersten Test. Die langen und sinnlosen Ausdrücke sind aus dem Ergebnis entfallen, da sie nicht mehr den gegebenen Kriterien entsprechen. In Zeile 4 des Listings befindet sich jedoch das Muster `V1\.?\swhen\syou\sare\sready,\showever,\sV2` welches zunächst mit einem optionalen Punkt beginnt und mehrere Kommata enthält. Es macht Sinn anzunehmen, dass eine solche Formulierung zu speziell ist um eins zu eins in ein möglichst allgemeines Muster übernommen zu werden. Aus diesem Grund wurde im letzten Test der Threshold erhöht.

```
1 AvailableForRelation V1 = Framework, V2 = Programming language:
2 V1\sis\savailable\sfor\sV2
3
4 WrittenInRelation V1 = Application, V2 = Programming language:
5 V1\sis\san\side\sdeveloped\sin\sV2
6
7 Programming_Entry_Paradigm V1 = Programming language, V2 = Paradigm:
8 V1\sis\sa\sV2
9
10 Programming_Entry_Developer V1 = Programming language, V2 = Developer:
11 V1\swas\sdeveloped\sby\sV2
12
13 Programming_Entry_Year V1 = Programming language, V2 = Year:
14 V1\swas\screated\sin\sV2
```

Listing 5.3: Muster nach dem Testdurchlauf mit Threshold von 2 und maximaler Distanz von 30 Zeichen.

Das Ergebnis des letzten Tests reduziert die Anzahl der gefundenen Muster noch weiter. Der Vorteil ist allerdings, dass die hier gefundenen Ausdrücke sehr viel generischer sind. Tatsächlich kann man an dem Muster `V1\sis\sa\sV2` in Listing 5.3, Zeile 8 sehen, dass die ein Ausdruck zu allgemein werden kann. Dies wurde in der Umsetzung zunächst nicht berücksichtigt und kann zu einigen Problemen führen. Zum Beispiel könnten falsche Entitäten gefunden werden was die Seeddatei mit unsinnigen Werten befüllt und dadurch wiederum zu falschen Mustern führt. Eine mögliche Lösung könnte es sein neben der maximalen Zeichendistanz noch eine minimale einzuführen um ein zu häufiges matching zu verhindern. Wichtig ist auch, dass eine Konfiguration mit einem solchen Threshold bei den ersten Texten zunächst keinerlei Annotationen liefert, da erst ein Lernprozess erforderlich ist in dem valide Muster gefunden werden.

0 Threshold, 100 Distanz	0 Threshold, 30 Distanz	2 Threshold, 30 Distanz
Viele Muster	Mittlere Musteranzahl	Weniger Muster
Oft auf einen konkreten Fall bezogen	Recht generisch	Generisch
Grösstmöglicher Recall auf kurze Zeit	Mittlerer Recall	Geringer Recall
Geringe Precision	Mittlere Precision	Beste Precision
Kurze Lernphase	Kurze Lernphase	Lange Lernphase

Tabelle 5.2.: Übersicht der Eigenschaften der Musterfindung bei unterschiedlicher Konfiguration

Betrachtet man das Ergebnis des Tests so lässt sich annehmen, dass die letzte Konfiguration immer das beste Ergebnis liefert. Es sollte allerdings bedacht werden, dass hier die umfangreichste Lernphase benötigt wird. Es müssen erst viele mögliche Muster verworfen werden. Im Grunde entspricht dieses Verhältnis dem von *Precision* und *Recall* welches bereits in Kapitel 3.2 beschrieben wurde. Der erste Durchlauf bietet den besten *Recall*, da auch Beziehungen zwischen Entitäten gefunden werden können, die im Text weit entfernt voneinander stehen. Der letzte Test hingegen hat eine höhere Wahrscheinlichkeit eines korrekten Ergebnisses, also *Precision*. Im Kontext von automatischer Generierung von semantischen Annotationen ist *Precision* sicherlich wichtiger, da eine falsche Annotation das Netz schnell unbrauchbar machen kann.

Insgesamt haben die hier durchgeführten Tests gezeigt, dass das implementierte System bereits in der Lage ist reale Texte zu analysieren und in einem gewissen Rahmen semantisch

zu annotieren. Die Ergebnisse der hier gemachten Tests sind allerdings zunächst nur in der vorgegebenen Domäne gültig. Eine Übertragbarkeit scheint zwar sinnvoll, da Satzstrukturen prinzipiell nicht Domänenabhängig sind. Eine Überprüfung ob andere Domänen zu dem gleichen Schlüssen führen konnte im Rahmen dieser Arbeit jedoch nicht durchgeführt werden. Davon unabhängig haben die Tests auch aufgezeigt wo noch mögliche prinzipielle Probleme bestehen. Zu einfache Muster die auf falsche Werte matchen und dadurch die Beispieldaten in den Seeds verunreinigen scheinen hier besonders kritisch.

5.2.2. Qualität der Annotationen

Nun soll noch begutachtet werden welche Qualität die tatsächlichen semantischen Annotationen haben. Um dies zu beurteilen soll ein Beispieltext betrachtet werden. Der Text wird von Hand annotiert. Anschließend kann verglichen werden welche Annotationen durch den implementierten Prototypen gefunden werden. Zunächst folgt der Beispieltext.

The Scala programming language is an optional part of the Typesafe Reactive Platform. It was first created in 2003. You can get started building Reactive applications using the Java APIs with Play and Akka. When you are ready, however, Scala will make building Reactive applications on the JVM less complicated. Scala integrates functional paradigms and retains full interoperability with your existing Java libraries and infrastructure so you can take advantage of multicore hardware. Consequently, it is an excellent programming model for Reactive.

Betrachtet man den Text im Kontext der in dieser Arbeit behandelten Semantik stößt man zunächst auf die beiden Programmiersprachen Scala und Java, sowie die Frameworks Play und Akka. Eine Übersicht der Annotationen folgt.

Annotation von Hand	Annotation des Systems
Scala/2003 Year	-
Java/Play AvailableFor	Java/Play AvailableFor
Java/Akka AvailableFor	Java/Akka AvailableFor
Scala/functional Paradigm	Scala/functional Paradigm
-	Scala/Akka AvailableFor

Tabelle 5.3.: Gegenüberstellung von Annotation von Hand und Prototyp

Die hier dargestellten Zusammenhänge können in drei Kategorien eingeteilt werden. Diese welche vom System korrekt ermittelt wurden, falsch positive Annotationen, die sich aus dem

Text so nicht ergeben, und nicht gefundene Informationen.

Die korrekten gefundenen Annotationen weisen in diesem Beispiel alle die gleiche Struktur auf. Sätze, welche beide Teile einer Beziehung beinhalten werden von dem System prinzipiell erkannt, sofern sinnvolle Muster oder Beispieldaten in Form von Seeds vorhanden sind.

Die vom System nicht gefundene Annotation hingegen weist eine andere Struktur auf. Die Beziehung *Scala/2003 Year* ergibt sich aus dem Satz *It was first created in 2003*. Die Information, dass sich *It* hier auf die Programmiersprache Scala bezieht hat der Prototyp nicht. Um dieses Problem anzugehen könnte eine Koreferenzanalyse durchgeführt werden. Dabei würden im Rahmen einer Vorverarbeitung herausgearbeitet werden auf wen sich zum Beispiel ein Personalpronomen konkret bezieht. Im Text kann dann dies ausgetauscht werden, so dass sich die weitere Analyse auf konkrete Entitäten beziehen kann. Der Satz *It was first created in 2003* würde also in *Scala was first created in 2003* umgewandelt werden, der wiederum vom System interpretiert werden kann Appelt (1999).

Die letzte Annotation besagt, dass das Framework Akka auch für Scala verfügbar ist. Dies ist auf die reale Welt bezogen zwar korrekt, ergibt sich jedoch so nicht aus dem Text. Aus diesem Grund muss dies als falsch positiv gewertet werden. Zustande kommt diese Annahme aus dem Teil *...Akka. When you are ready, however, Scala...* in dem beide Entitäten auf kurze Distanz aufeinander folgen, allerdings in zwei unterschiedlichen Sätzen. Solche Funde können wie bereits in dem vorherigen Abschnitt erwähnt durch einen gut gewählten Wert für den Threshold bereits recht gut vermieden werden, da derartige Formulierungen keine grundlegenden Muster bilden. Noch besser wäre allerdings eine erweiterte Kontextanalyse die Satzgrenzen und Nebensätze berücksichtigt anstelle des naiven Ansatzes den gesamten Text zwischen zwei Begriffen als Kontext aufzufassen.

6. Fazit

6.1. Zusammenfassung

In dieser Arbeit wurde erarbeitet wie sinnvolle semantische Annotationen mit einem möglichst hohen Automatisierungsgrad erstellt werden können. Semantische Annotationen sind für das Semantic Web von großer Bedeutung, da mit Ihnen der Inhalt des Webs auf eine für Maschinen lesbare Form gebracht werden kann. Das Semantic Web bezeichnet also den Sprung von einem dokumentbasierten Internet hin zu einem auf Basis der Semantik. Es ist jedoch kaum vorstellbar, dass semantische Annotationen für alle bereits existierenden Inhalte im World Wide Web manuell erstellt werden können. Aus diesem Grund ist es unumgänglich einen Ansatz zu verfolgen bei dem ein möglichst hoher Grad an Automatisierung bei guter Qualität erreicht werden kann.

Um dies zu erreichen gibt es eine ganze Reihe von Ansätzen. Im Rahmen dieser Arbeit wurden jeweils zwei Ansätze auf Basis von Mustern und maschinellem Lernen genauer vorgestellt. Es wurde Musterfindung und ein Ansatz mit festen Regeln erläutert. Bei der Musterfindung wird anhand von bekannten Begriffen, die in Beziehung zueinander stehen, nach Mustern in einem Text gesucht die diese Beziehung ausdrücken. Dadurch entstehen Regeln mit denen diese Beziehung auch für neue Begriffe erkannt werden kann. Bei dem klassischen regelbasierten Ansatz hingegen wird versucht die Regeln ohne konkrete Beispiele so allgemein wie möglich zu formulieren. Vorher wird eine natürlichsprachliche Analyse durchgeführt um den Text in eine Form zu bringen die verarbeitet werden kann.

Die beiden anderen beschriebenen Ansätze sind Hidden Markov Modelle und Wrapper Induction. HMM sind ein stochastisches Modell welches vielfach im Bereich des maschinellen Lernens angewendet wird. Um semantische Annotationen zu generieren ist jedoch eine qualitativ hochwertige Ontologie nötig mit dem das HMM lernen kann. Ein großer Vorteil dieses Ansatzes aber ist, dass auch neue Informationen die nicht in der Ontologie beschrieben werden gefunden werden können. Ein weiterer Ansatz ist der Einsatz von Wrapper Induction. Bei dieser Technik

wird versucht mit Hilfe überwachtem Lernen die Punkte in einem semistrukturiertem Text zu finden welche die gesuchten Informationen umschließen. Dieser Ansatz ähnelt sehr dem der Musterfindung, allerdings wird hier in der Regel keine Umwandlung in ein allgemeineres Muster vorgenommen und es werden keine neuen Regeln nach der ersten Lernphase mehr gesucht.

Im praktischen Teil der Arbeit wurde ein Prototyp umgesetzt um die betrachteten Techniken in einem praxisnahem Beispiel einzusetzen. Als Grundlage für den Prototypen wurden die musterbasierten Ansätze gewählt, da sich dieser Ansatz ohne umfangreiche Ontologie oder Trainingsdaten umsetzen lässt und dabei dennoch eine gute Qualität verspricht. Es wurde ein einfaches Bloggingportal umgesetzt in dem Einträge nach dem Veröffentlichen analysiert wurden. Danach konnten die RDF-Annotationen direkt an dem Beitrag angezeigt werden. Als Domäne für diese Umsetzung wurde die Softwareentwicklung gewählt mit einigen exemplarisch gewählten Entitäten und Beziehungen. Die Umsetzung des Prototypen geschah mit Hilfe des UIMA Frameworks in zwei Schritten. Im ersten Schritt ging es darum die linguistische Analyse des Textes durchzuführen. Es wurden Token erzeugt, Part of Speech Tagging und Named Entity Recognition durchgeführt. Dann wurden Attribute und Beziehungen mit Hilfe von festen Regeln gesucht. Im zweiten Schritt wurde das System noch um die Musterfindung erweitert um einen lernenden Aspekt hineinzubringen. Für die Muster wurden dazu einfache reguläre Ausdrücke verwendet.

Mit dem umgesetzten Prototyp sind einige exemplarische Testläufe durchgeführt worden. Dazu wurden kurze Blogeinträge mehrmals in unterschiedlicher Konfiguration des Systems annotiert. Aus diesen Tests haben sich einige neue Erkenntnisse ergeben. So kann es leicht passieren, dass zu simple Regeln generiert werden welche dann zu oft zutreffen und zu Problemen führen. Andersrum bieten zu umfangreiche Regeln keinen praktischen Nutzen wenn sie niemals ein zweites mal zutreffen. Eine Begrenzung von etwa 30 Zeichen für die Regellänge und ein Zurückhalten dieser Regeln bis sie durch Mehrfachvorkommen validiert wurden ergaben in den Tests die besten Ergebnisse. Bei einem Vergleich zwischen den Annotationen des Systems und einer Annotation von Menschenhand zeigt sich, dass Informationen aus einfach strukturierten Sätzen ohne Verschachtlung meistens gefunden werden. In einigen Fällen kommt es jedoch zu falschen Annotationen oder Informationen werden nicht entdeckt. Dies lässt sich vor allem auf eine in der jetzigen Umsetzung nur sehr grundlegenden natürlichsprachlichen Analyse zurückführen bei der Satzgrenzen oder Pronomen nicht berücksichtigt werden. Insgesamt erfüllt der Prototyp die Erwartungen in Bezug auf Automatisierung, Qualität und

Generalisierbarkeit. Im Laufe der Implementierung und Tests hat sich jedoch noch Potenzial für mögliche Erweiterungen ergeben.

6.2. Ausblick

Prinzipiell könnte auf diese Arbeit auf zwei Arten aufgesetzt werden. Zum einen wäre es denkbar den Prototypen im Bezug auf die Qualität der Ergebnisse zu verbessern. So könnte man die Texte mit einer Koreferenzanalyse aufbereiten. Dadurch wäre das System in der Lage auch Bezüge zu erkennen die bisher noch nicht erkannt werden können, wie Personalpronomen oder Synonyme. Ferner werden Satzgrenzen bisher noch nicht berücksichtigt. Eine weitere Untersuchung könnte auch darin bestehen wie Ontologien semiautomatisch erstellt werden können.

Im Rahmen einer solchen Erweiterung wäre es auch sinnvoll ein umfangreiches Bewertungsschema auszuarbeiten. Mit einer größeren Menge von Testdaten sollte es möglich sein neue Schlüsse bezüglich Erweiterungen und deren Effekt zu ziehen. Ein interessanter Test wäre sicherlich auch ob und wie sich die Ergebnisse verändern wenn die Fachdomäne gewechselt wird.

Der andere Aspekt wäre die nun vorhandenen Annotationen in einer semantischen Applikation zu verwenden. In einer praktischen Arbeit könnte untersucht werden wie sich die semantischen Informationen tatsächlich verwerten lassen. Eine denkbare Applikation wäre die semantische Suche. Es ließe sich eine Anwendung entwickeln, welche auf Basis der im Blog vorhandenen Annotationen natürlichsprachliche Fragen beantwortet wie „Was war die erste Programmiersprache?“ oder „Welche Frameworks sind für Scala verfügbar?“. Auch weitere semantische Anwendungen sind vorstellbar. Eine Applikationen welche Empfehlungen für weitere interessante Artikel ausspricht wäre ebenso denkbar. Also letztlich eine Hilfe bei der Navigation durch das semantische Netz auf Basis von zusammenhängenden Inhalten.

Im Rahmen dieser Arbeit wurde betrachtet welche Ansätze es gibt um semantische Annotationen verfügbar zu machen. Dabei ging es in der Umsetzung vor allem um die Machbarkeit und den Aufwand. Auf Basis der gewonnen Erkenntnisse sind nun eine Vielfalt von weiteren Untersuchungen möglich.

A. Anhang

Auf der CD befindet sich diese Arbeit im PDF-Format, der Quellcode für die Implementierung, sowie die verwendeten Quellen und Bibliotheken.

A.1. Aufbau der CD

- /
  - ba
    - Erwin_Lang_Bachelorarbeit.pdf
  - ref
    - Verwendete Quellen
  - src
    - RDFBlog..... Das Hauptprojekt
    - RDFCreator..... Analysis Engine zum RDF erstellen
    - OnthologyMapper..... Analysis Engine OnthologyMapper
    - Libraries..... Verwendete Bibliotheken

Literaturverzeichnis

[Appelt 1999] Appelt, Douglas E.: Introduction to Information Extraction. In: *AI Commun.* 12 (1999), August, Nr. 3, S. 161–172. – URL http://dl.acm.org/citation.cfm?id=1216155.1216161. – ISSN 0921-7126

[Beckett 2004] Beckett, Dave: RDF/XML Syntax Specification (Revised) / W3C. Februar 2004. – W3C Recommendation. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/

[Berners-Lee u. a. 2001] Berners-Lee, Tim ; Hendler, James ; Lassila, Ora: The Semantic Web. In: *Scientific American* 284 (2001), Mai, Nr. 5, S. 34–43. – URL http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21

[Berry 2003] Berry, Michael: *Survey of Text Mining : Clustering, Classification, and Retrieval*. Springer, September 2003. – URL http://www.amazon.co.uk/exec/obidos/ASIN/0387955631/citeulike-21. – ISBN 0387955631

[Blumauer und Tassilo 2006] Blumauer, Andreas ; Tassilo, Pellegrini: Semantic Web und semantische Technologien: Zentrale Begriffe und Unterscheidungen. In: Blumauer, Andreas (Hrsg.) ; Tassilo, Pellegrini (Hrsg.): *Semantic Web. Webe zur vernetzten Wissensgesellschaft*. Berlin, Heidelberg : Springer Verlag, 2006, S. 9–25

[Brin 1999] Brin, Sergey: Extracting Patterns and Relations from the World Wide Web. In: *Selected Papers from the International Workshop on The World Wide Web and Databases*. London, UK, UK : Springer-Verlag, 1999 (WebDB '98), S. 172–183. – URL http://dl.acm.org/citation.cfm?id=646543.696220. – ISBN 3-540-65890-4

[Buitelaar und Magnini 2005] Buitelaar, Paul ; Magnini, Bernardo: Ontology Learning from Text: An Overview. In: *In Paul Buitelaar, P., Cimiano, P., Magnini B. (Eds.), Ontology Learning from Text: Methods, Applications and Evaluation*, IOS Press, 2005, S. 3–12

[Chapman u. a. 2004] Chapman, Sam ; Dingli, Alexiei ; Ciravegna, Fabio: Armadillo: harvesting information for the semantic web. In: *SIGIR*, URL http://dblp.uni-trier.de/db/conf/sigir/sigir2004.html#ChapmanDC04, 2004, S. 598

[Charton u. a. 2011] Charton, Eric ; Gagnon, Michel ; Ozell, BenoÃ®t: Automatic Semantic Web Annotation of Named Entities. In: Butz, Cory J. (Hrsg.) ; Lingras, Pawan (Hrsg.): *Canadian Conference on AI* Bd. 6657, Springer, 2011, S. 74–85. – URL http://dblp.uni-trier.de/db/conf/ai/ai2011.html#ChartonGO11. – ISBN 978-3-642-21042-6

[Ciravegna u. a. 2004] Ciravegna, Fabio ; Chapman, Sam ; Dingli, Alexiei ; Wilks, Yorick: Learning to Harvest Information for the Semantic Web. In: *ESWS*, URL http://dblp.uni-trier.de/db/conf/esws/esws2004.html#CiravegnaCDW04, 2004, S. 312–326

[Cunningham u. a. 2011] Cunningham, Hamish ; Maynard, Diana ; Bontcheva, Kalina ; Tablan, Valentin ; Aswani, Niraj ; Roberts, Ian ; Gorrell, Genevieve ; Funk, Adam ; Roberts, Angus ; Damljanovic, Danica ; Heitz, Thomas ; Greenwood, Mark A. ; Saggion, Horacio ; Petrak, Johann ; Li, Yaoyong ; Peters, Wim: *Text Processing with GATE (Version 6)*. URL http://tinyurl.com/gatebook, 2011. – ISBN 978-0956599315

[Ferrucci und Lally 2004] Ferrucci, David ; Lally, Adam: UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment. In: *Nat. Lang. Eng.* 10 (2004), September, Nr. 3-4, S. 327–348. – URL http://dx.doi.org/10.1017/S1351324904003523. – ISSN 1351-3249

[Gandon und Schreiber 2014] Gandon, Fabien ; Schreiber, Guus: RDF 1.1 XML Syntax / W3C. Februar 2014. – W3C Recommendation. http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/

[Gentile u. a. 2013] Gentile, Anna L. ; Zhang, Ziqi ; Augenstein, Isabelle ; Ciravegna, Fabio: Unsupervised Wrapper Induction Using Linked Data. In: *Proceedings of the Seventh International Conference on Knowledge Capture*. New York, NY, USA : ACM, 2013 (K-CAP '13), S. 41–48. – URL http://doi.acm.org/10.1145/2479832.2479845. – ISBN 978-1-4503-2102-0

[Gruber 2009] Gruber, Tom: Ontology. In: *Encyclopedia of Database Systems*. URL http://dx.doi.org/10.1007/978-0-387-39940-9_1318, 2009, S. 1963–1965

[Guha und Brickley 2014] Guha, Ramanathan ; Brickley, Dan: RDF Schema 1.1 / W3C. Februar 2014. – W3C Recommendation. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/

[Hotho u. a. 2005] Hotho, Andreas ; Nürnberger, Andreas ; Paaß, Gerhard: A Brief Survey of Text Mining. In: *LDV Forum - GLDV Journal for Computational Linguistics and Language Technology* 20 (2005), Mai, Nr. 1, S. 19–62. – URL http://www.kde.cs.uni-kassel.de/hotho/pub/2005/hotho05TextMining.pdf. – ISSN 0175-1336

[Kushmerick 2000] Kushmerick, Nicholas: Wrapper Induction: Efficiency and Expressiveness. In: *Artificial Intelligence* 118 (2000), S. 2000

[Lin und Pantel 2001] Lin, Dekang ; Pantel, Patrick: DIRT @SBT@Discovery of Inference Rules from Text. In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA : ACM, 2001 (KDD '01), S. 323–328. – URL http://doi.acm.org/10.1145/502512.502559. – ISBN 1-58113-391-X

[Maedche und Staab 2001] Maedche, Alexander ; Staab, Steffen: Ontology Learning for the Semantic Web. In: *IEEE Intelligent Systems* 16 (2001), März, Nr. 2, S. 72–79. – URL http://dx.doi.org/10.1109/5254.920602. – ISSN 1541-1672

[Maynard 2003] Maynard, D.: Multi-Source and Multilingual Information Extraction. In: *Expert Update* 6 (2003), Nr. 3, S. 11–16. – URL /brokenurl#http://publication.wilsonwong.me/load.php?id=233281323

[Noy und Mcguinness 2001] Noy, Natalya F. ; Mcguinness, Deborah L.: Ontology Development 101: A Guide to Creating Your First Ontology. 2001. – Forschungsbericht

[Reeve und Han 2005] Reeve, Lawrence ; Han, Hyoil: Survey of semantic annotation platforms. In: *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY, USA : ACM, 2005, S. 1634–1638. – URL http://portal.acm.org/citation.cfm?id=1067049. – ISBN 1-58113-964-0

[Stamp 2004] Stamp, Mark: *A revealing introduction to hidden Markov models*. 2004

[Typesafe 2014] Typesafe: *Play Framework*. 2014. – URL http://www.playframework.com/

[UIMA 2014] UIMA, Development: *UIMA Tutorial and Developers' Guides*. 2014. – URL http://uima.apache.org/downloads/releaseDocs/2.2.1-incubating/docs/html/overview_and_setup/overview_and_setup.html

[Valarakos u. a. 2003] Valarakos, A. ; Sigletos, G. ; Karkaletsis, V. ; Paliouras, G.: *A Methodology for Semantically Annotating a Corpus Using a Domain Ontology and Machine Learning*. 2003. – URL http://citeseer.ist.psu.edu/695438.html

[Vargas-Vera u. a. 2002] Vargas-Vera, M. ; Motta, E. ; Domingue, J. ; Lanzoni, M. ; Stutt, A. ; Ciravegna, F.: *MnM: Ontology Driven Tool for Semantic Markup*. 2002. – URL citeseer.ist.psu.edu/vargas-vera02mnm.html

[Zouaq und Nkambou 2010] Zouaq, Amal ; Nkambou, Roger: A Survey of Domain Ontology Engineering: Methods and Tools. In: Nkambou, Roger (Hrsg.) ; Bourdeau, Jacqueline (Hrsg.) ; Mizoguchi, Riichiro (Hrsg.): *Advances in Intelligent Tutoring Systems* Bd. 308. Springer Berlin Heidelberg, 2010, S. 103–119. – URL http://dx.doi.org/10.1007/978-3-642-14363-2_6. – ISBN 978-3-642-14362-5

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 13. Oktober 2014 Erwin Lang