Projektbericht zum Praktikum Mehrsprachigkeit im Semantic Web

1. Das Projekt

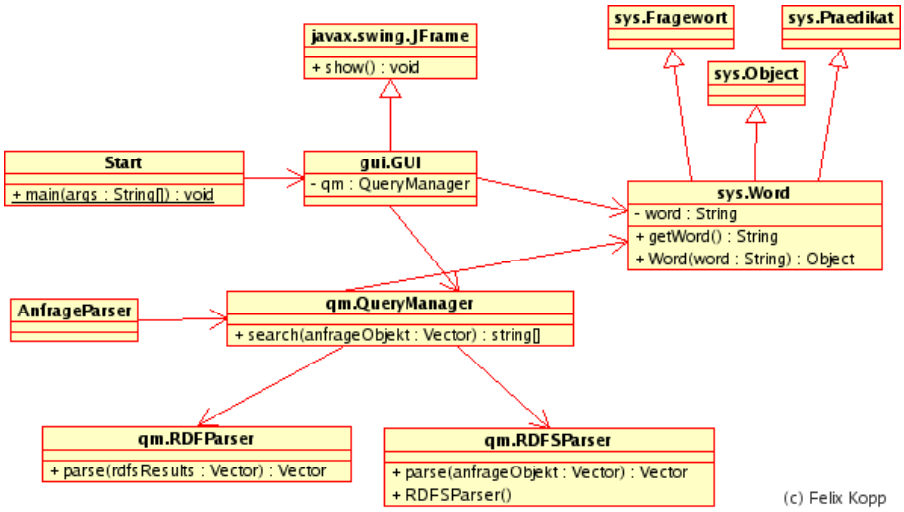
1.1. Hintergrund

1.2. Vorgaben und Vorlagen

1.3. Das Projektziel

1.4. Die technischen Schnittstellen

Folgend das Klassendiagramm und Schnittstellendefinition:



2. RDFS-Ontologie

3. RDF-Annotation

4. Graphical User Interface

Von Felix Kopp (#5554601) und Philip Becker (#)

4.1. Plattform-Fragen

Bei der Beratung um die technische Umsetzung des Projektes im Team sind wir zum Schluss gekommen, dass für ein Basisimplementierung der Suchmaschine für das Semantic Web nur eine Java-Swing Applikation in Frage kommt. Obwohl diese Art der Implementierung auch viele Beschränkungen in Sachen Skalierbarkeit und Erweiterung mit sich führt, liegen die Vorteile ebenso offensichtlich auf der Hand. Die Plattformunabhängigkeit, um an dieser Stelle nur einen zu nennen.

Es war uns so möglich, die basis-technische Umsetzung des User-Interfaces in schnellen Schritten

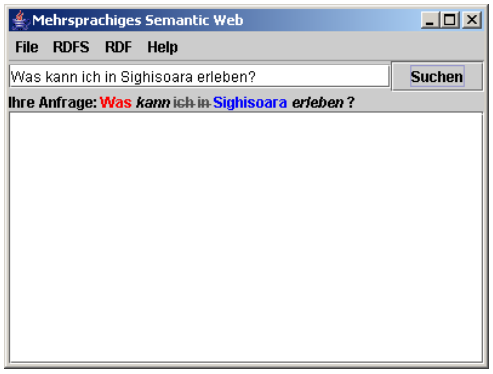
zu realisieren. Es war nicht von Nöten, die Knöpfe, die Textfelder und andere Widgets selbst zu entwickeln. Dank der von Java vorgegeben API der Swing-Klassen war es uns möglich, mehr Planungszeit für die Benutzerführung und das Ausfeilen der einzelnen Algorithmen zu investieren.

Für die Programmiersprache Java sprach neben dem enormen Vorteil der hohen Plattformunabhängigkeit auch der Punkt, dass wir den bereits einmal entwickelten Code mit relativ wenig Aufwand für eine große Client-Server-Anwendung wiederverwenden können. Mit dem Gedanken im Hinterkopf, vielleicht das Projekt in die Form einer Enterprise Anwendung auf Jakarta Tomcat-Basis zu portieren, können wie so von den Vorteilen der Allgemeinheit der Sprache Java profitieren. Unsere Anwendung würde einfach in großen Teilen zur Bean im Hintergrund des Servers restrukturiert werden.

4.2. Benutzerführung

Der Benutzer unserer Software wird durch den sehr einfachen Aufbau der Anwendung und die klare Anordnung der Elemente innerhalb des Fensters direkt in Richtung Ziel geschoben. Beim Entwurf der grafischen Komponenten haben wir uns an etablierten Suchmaschinen des Internets orientiert und sind letztendlich bei dem aktuellen Aufbau angelangt. Ein Feld für die Eingabe des Anfragetextes und ein anschließender Knopf für die Zündung des Suchprozesses. Die Ausgabe der gefundenen Suchergebnisse erfolgt in dem hierfür vorgesehenen Areal unterhalb des Eingabebereiches.

Das Fenster ist optisch klar in zwei Bereiche getrennt. Im oberen Bereich findet sich der für den Benutzer interaktive Part. Hier gilt es die Wünsche zu positionieren. Im unteren Teil des Fensters befindet sich der Feedback-Part. In diesem Teil werden die zur Anfrage passenden Antworten grafisch aufbereitet. Diese Einteilung ist auch für Benutzer mit geringen Erfahrungen mit dem Internet oder zumindest Office-Erfahrung leicht erschliessbar.



Ein Screenshot des Benuterinterfaces der Applikation mit einem grafisch aufbereiteten, gewichteten Anfragesatzes (unterhalb des Eingabefeldes).

Technische Umsetzung

Das Grafische User Interface (GUI) verwaltet auf Grund der zentralen Position in der Software neben der Kommunikation mit dem Anwender auch die Steuerung des Datenflusses innerhalb des Programms. Von dem Zeitpunkt, an dem der Anwender den Knopf zum Start betätigt, übernimmt das GUI die Führung. Es holt die Daten direkt aus dem Eingabefeld ab und schickt sie durch die verschiedenen Instanzen des Anfrage-Parser und des QueryManagers. Die letztendlich aufbereiteten Suchergebnisse werden wieder im GUI zusammengefasst aufbereitet und ausgegeben.

Um den kompletten Prozess leichter nachvollziehen zu können, folgt hier ein Flussdiagramm:
Gui, Textfeld, Knopf, ActionEvent, Anfrage-Parser (v2), QueryManager, RDF, RDFS, RDF, GUI

5. Anfrage-Parser (v2)

Von Felix Kopp (#5554601) und Philip Becker (#)

5.1. Sinn und Zweck

Die heutige Informatik beschäftigt sich vielfach mit der Lösung der Problematik, die daraus entsteht, dem Computer klarzumachen, was die Logik hinter unseren getippten Worten ist. Hierzu beschäftigten wir uns im Praktikum mit der Reformatierung der Inhalte der geschriebener Textquellen und der logischen Zusammenhänge der einzelnen Texte untereinander. Desweiteren habe wir ein System geschaffen, welches die Analyse der eingegeben Anfrage durchführt. Die Anfrage kommt in einem großen Block an und wird gewichtet an die anderen Programmteile übermittelt. Dieser Teil ist dem Anfrage-Parser auferlegt.

5.2. Technische Umsetzung

Für die technische Umsetzung des Anfrage-Parsers haben wir uns die Regulären Ausdrücke zur Hand genommen, welche eigentlich aus der Sprache Perl stammen. Die Regulären Ausdrücke aus der Programmiersprache Perl, deren Kernfähigkeiten Textmodifikation und Sprach- und Logikidentifikation ist, standen Pate für die Regulären Ausdrücke der Sprach Java. Diese Ausdrücke ermöglichten es uns, den Aufwand für die Umsetzung der Wortanalyse drastisch zu minimieren. So ist es viel einfacher, bereits gelesene Wortteile zusammenzufassen und die Anzahl der Abfragen zu reduzieren. Eine Anfrage, die das Wort "deren" ertasten soll, kann somit gleich das Wort "der" und mit wenig Anpassung auch "dessen" validieren.
Die restlichen Worte, die wir nicht optimiert mit Regulären Ausdrücken filtern konnten, haben wir eins-zu-eins aus Blacklists gelesen. Diese langen, von uns vorgegebenen schwarzen Listen beinhalten alle nicht gewünschten Wörter. Sie werden schrittweise mit den vorgegebenen Worten verglichen und im Falle, dass das zu gewichtende Wort in einer Blacklist steckt, so wird es entfernt und die Suche fährt mit dem Folgewort fort.

5.3. Der Algorithmus

Die Anfrage wird im ersten Schritt des Prozesses von dem langen String, als welche sie angenommen wird, in die entsprechenden Datenstrukturen konvertiert. Einrahmende Leerzeichen und Satzzeichen werden entfernt und der Satz wird Wort für Wort in eine lange Liste gekapselt. Nach der erfolgreichen Teilung der Einzelwörter starten wir die eigentliche Gewichtungslogik.
Da das System laut der behandelten Aufgabenstellung des Praktikums dafür ausgelegt seien sollte, auf die erfolgreiche Arbeit mit den zur Verfügung gestellten Fragesätzen des Demo-Fragekorpus zu arbeiten, konnten wir in der Entwicklung des Algorithmus einige Voraussetzungen als gegeben hinnehmen, welche für ein semantisches Analysesystem nicht geben wären. So konnten wir eine Gemeinsamkeit feststellen, die alle Fragen des Korpus gemein haben: Die Fragewörter stehen immer am Anfang des Satzes und sind entweder direkt das Wort "Gibt" oder eine W-Wort, beispielsweise "warum" oder "wieviele".

Für den übrigen Satz haben wir eine Schleife entwickelt, die Wort für Wort iteriert und in jedem Schritt eine Überprüfung und Einordnung durchführt. Wir konnten bei der Analyse immer wiederkehrende Pronomina sowie die Artikel und Appositionen erkennen, die für die Durchsuchung der RDF-Dateien und RDFS-Ontologie nicht förderlich eingesetzt werden können. Wie wir von anderen etablierten Suchmaschinen wissen, werden diese füllenden Wörter einfach aus dem Anfragesatz gefiltert. Wir haben für diesen Zweck Blacklists entwickelt und möglichst viele, Ressourcen schonende Reguläre Ausdrücke eingesetzt.
Neben den mit einem kleinen Buchstaben beginnenden bestimmten Artikeln (wie "der", "die" "das"), haben wir auch die unbestimmten Artikel gut zusammenfassen können. Mit dem Ausdruck ein(e[nmrs]{0,1})* können wir bereits den größten Teil abgedecken. Dieser Ausdruck erkennt neben dem Wort "ein", beispielsweise auch die Wörter "einer", "einem" und sogar "eine".
Die genauen Blacklists und regulären Ausdrücke befinden sich im Quellcode. Paket qm.AnfrageParser.

Bei der Gewichtung der für unsere Suche relevanten Komponenten, den logischen Objekten und der Prädikate haben wir erneut die deutsche Groß- und Kleinschreibung herangezogen. Wir sind, um die Komplexität des Systems in beherrschbaren Bahnen zu halten, davon ausgegangen, dass die Suchanfragen in jedem Falle in ordentlichem Schriftdeutsch formuliert werden. Auf diese Idee sind wir letztendlich gekommen, da wir die Entwicklung des Anfrage-Parsers (v2) immer anhand der Beispielfragen optimiert haben und den Korpus als vorgegebene Guideline stets im Blick hatten.

5.4. Einschränkungen und Optimierung

Die momentane Qualität des Outputs des Anfrage-Parsers ist noch recht beschränkt. So ist die Funktionalität des Parsers zum gegenwärtigen Zeitpunkt hauptsächlich mit deutschen Sätzen möglich.

Das System braucht grammatisch korrekten Input. Die Trennung der logischen Objekten von den Prädikaten ist nur auf Basis der Groß- und Kleinschreibung umgesetzt. Ausserdem werden nur prägnante Hauptsätze erfolgreich gescannt. Nebensätze, Relativnebensätze oder eine beliebige Anzahl aneinander gereihter Hauptsätzte werden nicht unterschieden.
In einem für den Endkunden einsetzbaren System muss die Implementation des Algorithmus wahrscheinlich zu großen Teilen neu realisiert werden. Der Parser sollte für den optimalen Fall die einzelnen Satzteile erkennen und verstehen. Er muss Anfragen mit einem semantischen Verständnis lesen und die Entscheidung über die Gewichtung der Satzteile nach grammatikalischen Regelements treffen. Die größte Schwierigkeit wird daraus erwachen, die Unterschiede zwischen der natürlichen Sprache dem grammatisch korrekten Schriftdeutsch aus der der natürlichsprachlichen Eingaben zu filtern.
Es wird schwer werden, Usern verständlich zu machen, dass sie sich auf die Sprachregeln der Suchmaschinen einstellen müssen und dass sich die Suchmaschinen nicht oder nur äußerst begrenzt auf Ihre Eingaben einstellen.

6. QueryManager

6.1. Anfrage-Reihenfolge

7. RDFS-Software

Von Felix Kopp (#5554601)

7.1. Das Jena Paket vorbereiten

Um eine Programm mittels der Jena API (http://jena.sourceforge.net, Hexlett Packard Research Labs) auf RDF und RDFS zugreifen zu lassen, müssen einige Vorbereitungen getroffen werden. Im ersten Schritt muss auf reiner Java Basis die Jena API referenziert werden. Wir müssen dem Java Code also erst einmal die ganzen Klassen beibringen und auf diesem Wege dafür sorgen, dass Java und RDFS mit einander spielen wollen. Die komplette Jena API ist auf den entsprechenden Seiten im Internet als .jar-Archiv verfügbar und somit bereits optimal präpariert. Neben den puren Jena-Klassen arbeitet die API mit einer wenigen Hand voll OpenSource-Tools zusammen, die beispielsweise das Parsen der einzelnen Textdateien übernimmt. Auch einige Funktionalitäten für die Protokollierung sind für Jena nicht erneut implementiert worden, sondern von der Apache Software Foundation übernommen. Alle benötigten Pakete werden löblicherweise beim Download mitgeliefert. Respektable Out-of-the-box Mentalität der höchsten Stufe!

7.2. Ontologie einlesen, RDFS parsen

Bevor der eigentliche Prozess zum Einlesen der RDFS-Datei beginnen kann, müssen auf Javaebene einige Klassen initialisiert und die Struktur für die Ontologie, die nur im Speicher gehalten wird (RDFS_MEM), angelegt werden. Zu diesem Zwecke bietet uns Jena die Java-Klassen Model, OntModel und die ModelFactory.

Rein von der Java-Hierarchie betrachtet, stellt die Klasse OntModel, also die Kapselung der Ontologie, eine Erweiterung der Klasse Model dar. Die Klasse Model ist auf Java-Ebene für die Verwaltung der RDF-Strukturen vorbehalten. Ich starte das Projekt mit dem OntModel, welches nur von der ModelFactory aus dem "Nichts" erzeugt werden kann. Hierfür rufe ich die Methode createOntologyModel() der ModelFactory auf, welche mir eine Instanz des OntModel zurückgibt. An dieser Stelle existiert eine logische Ontologie, die mit keinen Werten, neben den Informationen bezüglich der Ontologie-Sprache, dem Schema, gefüllt ist.
Da es neben der Sprache RDFS noch viele weitere, weiterentwickelte und mit weiteren Features angefüllte Dialekte, wie die Sprache OWL gibt, empfiehlt es sich, der create-Methode explizit zu übermitteln, welchen Dialekt man im Späteren zu verwenden plant. Für diesen Zweck wurden die *Spec-Klassen erfunden. Ich wähle die Konstante OntModelSpec.RDFS_MEM, aufgrund der Tatsache, dass ich im Dialekt RDFS arbeite, und nicht plane, die Ontologie in eine Datenbank oder ein ähnliches Medium zu transferieren. Sie wird nur im Speicher, im Memory, vorgehalten (MEM).

Für das reine Einlesen beliebiger RDFS-Daten sind sehr viele Übertragungswege und Aufbewahrungsorte angedacht. So ist es unter anderem möglich, die Daten direkt aus einem Stream, der vielleicht aus dem Internet stammt, zu entnehmen oder aber einfach nur aus einer lokalen Datei, die dann intern zum Stream gewandelt wird. Für unser Anwendungsgebiet begnügen wir uns mit der einfachen read-Methode, die den Pfad der Datei als String übergeben bekommt. Somit wird Jena das komplette Datei-Handling übernehmen. Die read-Methoden liegen auf dem OntModel.
Ich lese nun das RDFS-Model in den vorhin leer erzeugten Korpus ein, indem ich auf meiner Instanz des OntModel die read-Methode aufrufe und Ihr ich den aktuellen Pfad zur RDFS-Datei übergebe. Im im aktuellen Code lautet dieser "file:rdfs/Tourismus.rdfs".

Damit es beim Durchlesen der RDFS-Datei zu keinen Problemen kommt, muss die Syntax der Datei absolut präzise und korrekt seien. Es darf nicht eine Klammer fehlen und das Schema, welches die einzelnen Tags für den Xerces-Parser definiert, muss unbedingt mit den reichten Pfad gesetzt seien. Das Schema für unsere RDFS-Version ist unter http://www.w3.org/2000/01/rdf-schema# hinterlegt. Für die Operationen und Klassen benötigen Daten liegen hauptsächlich im Jena Paket com.hp.hpl.jena.ontology.

```
OntModel model = ModelFactory.createOntologyModel(OntModelSpec.RDFS_MEM);
m.read("file:rdfs/Tourismus.rdfs");
```

7.3. Domänen referenzieren

Wenn man die komplette Ontologie in das Java OntModel eingelesen hat, so kann man anfangen, Domänen zu suchen und andere Ressourcen und Properties zu identifizieren. Generell werden die Ressoucen innerhalb der Ontologie nur über die URIs, also dem eindeutigen Pfad unter welchem sie errecihbar sind, referenziert. Die Domäne "Stadt" ist somit unter file:rdfs/Tourismus.rdfs#Stadt zu

finden. Das gleiche gilt für andere Resoucen. Die Eigenschaft "hat_stadtnamen" ist ebenso unter file:rdfs/Tourismus.rdfs#hat_stadtnamen zu finden. Da alle Ressourcen innerhalb der Ontologie eindeutige Bezeichnungen tragen müssen, ist dies möglich.

```
Resource r = model.getResource("file:rdfs/Tourismus.rdfs#Start");
OntClass stadt = (OntClass) r.as(OntClass.class);
```

7.4. Properties lesen, seeAlso, sameAs

Für das heraussuchen einzelner Ressourcen, wurde von HP die Methode getResource, bzw. die Methoden getOntClass und ähnliche Getter, entwickelt. Diese Methoden liegen ebenso wie die read-Funktionalität auf dem OntModel und geben die Ressource als Javaklasse des Typs Resource zurück. Eine OntClass wird entsprechende von der Methode getOntClass gefunden.

Aus den Ressourcen können wir mittels der as-Methode andere Typen casten. Dies ist besonders für den Fall notwendig, wenn wir einfacher an die Eigenschaften der Ressourcen herankommen wollen. Die Superklasse Resource bietet nur sehr rudimentären Zugang zu weiteren Eigenschaften, ganz im Gegenteil zu der OntClass-Klasse. Auf der OntKlasse existieren weit mehr spezifischere Methoden, um die per seeAlso-Tag verwiesenden Ressourcen auszulesen (OntResouce :: getSeeAlso).
Eigenschaften zu lesen ist mittels der Methode listProperties auf dem Resource-Interface möglich, von welchem sämtliche Ont*-Klasse abgeleitet wurden.

```
OntResource or = r.as (OntResource.class);
or.getSameAs();
or.getSeeAlso();

Iterator propertyIterator = r.listProperties();
```

8. RDF-Software