h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

fbi
FACHBEREICH INFORMATIK

Hochschule Darmstadt

- Fachbereich Informatik -

Information Extraction aus natürlichsprachlichen Phrasen am Beispiel von Hotelsuchanfragen

Abschlussarbeit zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

vorgelegt von

Timm Heuß

Referent:	Prof. Dr. Bernhard Humm
Korreferentin:	Prof. Dr. Bettina Harriehausen-Mühlbauer
Ausgabedatum:	11.04.2012
Abgabedatum:	11.10.2012

Kontakt: mailto:Timm.Heuss@web.de

Abstract

Google setzt heutzutage mit seiner Freitextsuche Maßstäbe, wie Informationen benutzerfreundlich aufgefunden werden können. Dieser Realität sehen sich auch die Anbieter von Touristikportalen konfrontiert, die bisher vor allem auf formularbasierte Eingabeformen und Suchen setzten.

Als Grundlage für eine Freitextsuche für derartige Portale wird in dieser Arbeit eine Lösung für die Information Extraction aus natürlichsprachlichen Hotelsuchanfragen konzipiert und implementiert. Das Fundament dieser Anwendung besteht aus der standardisierten Architektur Apache UIMA, sowie der beiden entwicklernahen Aufsätze uimaFIT und DKPro.

In Anwendung üblicher Verfahren des Natural Language Processings werden die Benutzereingaben in englischer Sprache zunächst korrigiert, zerlegt und syntaktisch analysiert. Die so erzeugten Analyseergebnisse werden anschließend im Kern der Anwendung durch die Rule Engine JBoss Drools verarbeitet und mit Hilfe deskriptiver, prägnanter Regeln eine Named Entity Recognition von Hotel- und Zimmerausstattungsmerkmalen durchgeführt. Neben Erweiterbarkeit und Performance stellen auch semantische Features der Erkennung einen Mehrwert der Lösung dar, wie etwa Verneinungen, welche durch die Auswertung der Satzstruktur möglich sind.

Alle entwickelten Komponenten wurden dabei gemäß geltender Design-Prinzipien und Paradigmen zugrundeliegender Architekturen entwickelt. So weist insbesondere die Integrationskomponente, um Drools-Regeln innerhalb von UIMA einzusetzen, eine hohe Flexibilität auf. Zusammen mit der Software, die für die systematische Auswertung der Erkennungsleistung ergänzend zu UIMA-Werkzeugen entwickelt wurde, wird durch diese Arbeit auch ein wichtiger Grundstein für künftige Weiterentwicklungen gelegt.

Abstract

With its free text search, Google nowadays sets a high standard on how to find information in a convenient and easy way. Even the vendors of tourism portals are faced with the fact that their primal, form-based input- and search-functionalities are no longer the users first choice.

To create the foundation for a free text search for such a tourism portal, this paper is about the design and development of an Information Extraction solution which is able to process natural-language hotel search requests. It is based on the standardized architecture Apache UIMA as well as its developer-friendly derivatives uimaFIT and DKPro.

Applying basic techniques of Natural Language Processing, the English user input is first corrected, tokenized and syntactically analyzed. Processing of the analysis results then continues with the Rule Engine JBoss Drools. In combination with descriptive, concise rules a Named Entity Recognition for hotel and room amenities is realized. The resulting solution has significant benefits in terms of its extensibility and performance. Another great feature is the semantic support of natural language negations.

Every developed components follows the design principles and paradigms of the according, underlying architectures. Therefore, especially the integration component which enables the execution of Drools rules within UIMA features a high flexibility. Taking also the several enhancements of the UIMA developer tooling and a developed measuring framework into account, this paper also lays a essential foundation of future developments.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

AE Analysis Engine 27
Zentraler Baustein einer UIMA zur Analyse von Dokumenten.

API Application Programming Interface 29
Software-Schnittstelle eines Programms.

ASL Apache Software Licence 26
Freie Lizenz der Apache Software Foundation.

BPMN Business Process Modeling Notation 44
Modellierungstandard für Geschäftsprozesse.

CAS Common Analysis Structure 27
Austauschformat zwischen den verschiedenen Komponenten der UIMA.

CSV Comma Seperated Values 4
Einfaches Listenformat, das die Daten seiner Spalten durch ein Komma oder Semikolon voneinander trennt.

CI Continuous Integration 72
Entwicklungsparadigma, das verschiedene Softwarekomponenten möglichst häufig kompiliert, testet und integriert

DKPro Darmstadt Knowledge Processing Repository 35
UIMA-Aufsatz der Technischen Universität Darmstadt.

IR Information Retrieval 15
Disziplin zum Auffinden wertvoller Dokumente innerhalb von umstrukturierten Daten.

IE Information Extraction 16
Disziplin zum Auffinden wertvoller Fakten innerhalb von umstrukturierten Daten.

EVA Expert Virtual Agent 4

[1] `http://www.evature.com/` (letzter Abruf am 2012-07-30).

Standardisiertes Open Source Framework der Apache Foundation für die Verarbeitung von unstrukturierten Informationen.

uimaFIT UIMA Factories, Injection and Testing Utilities (ursprünglich UUTUC) ... 33
UIMA-Aufsatz, der Convenience-Werkzeuge für die UIMA-Entwicklung unter Java bereitstellt.

UUTUC UIMA Unit Test Utility Code ... 33
Ehemaliger Name von uimaFIT, der die Wurzeln des Projekts im Testwesen deutlich macht.

UML Unified Modeling Language
Verbreiteter Modellierungsstandard für Softwarediagramme.

NER Named Entity Recognition ... 23
Erkennung (NED) und Klassifizierung (NEC) von Named Entities, etwa Eigennamen von Personen.

NED Named Entity Detection ... 23
Erkennung von Named Entities.

NEC Named Entity Classification ... 23
Klassifizierung von Named Entities.

$F_{\beta=1}$ Balanced F-measure ... 18
Harmonisches Mittel aus gleichgewichtetem Precision (P) und Recall (R).

F F-measure ... 18
Gewichtetes, harmonisches Mittel aus Precision (P) und Recall (R).

IDE Integrated Development Environment ... 47
Integrierte Entwicklungsumgebung bestehend aus Editor und Compiler.

LHS Left Hand Side ... 37
Linker Teil einer Drools Regel nach dem Schlüsselwort `when`, beschreibt die Bedingung zum Auslösen der Regel.

NE Named Entity ... 23
Gegenstand der Erkennung bei der NER.

PCFG Probabilistic Context-Free Grammar ... 56
Kontextfreie Grammatik, die Wahrscheinlichkeiten mitberücksichtigt.

PRS Production Rule System ... 37
Implementierung einer Rule Engine, die passende Fakten zu bestehenden Regeln sucht und Rückschlüsse daraus zieht.

P Precision ... 16
Maß zur Abschätzung der Genauigkeit eines IR-Systems.

Kapitel 1.

Einleitung

Im Rahmen des Projekts *TourInDi*, einem Kooperationsprojekt zwischen der Hochschule Darmstadt und dem Touristikportal HRS[1], werden neue Methoden und Funktionalitäten zur Hotelsuche erforscht und prototypisch implementiert. Ein zentrales Entwicklungsziel ist dabei ein neues Eingabefeld für Freitext, mit dem die gewünschte Reise in natürlicher Sprache formuliert und gesucht werden kann. Dieses neue Eingabefeld ergänzt die bisher einzige, formulargestützte Suchmöglichkeit nach einem Hotelangebot auf dem Portal.

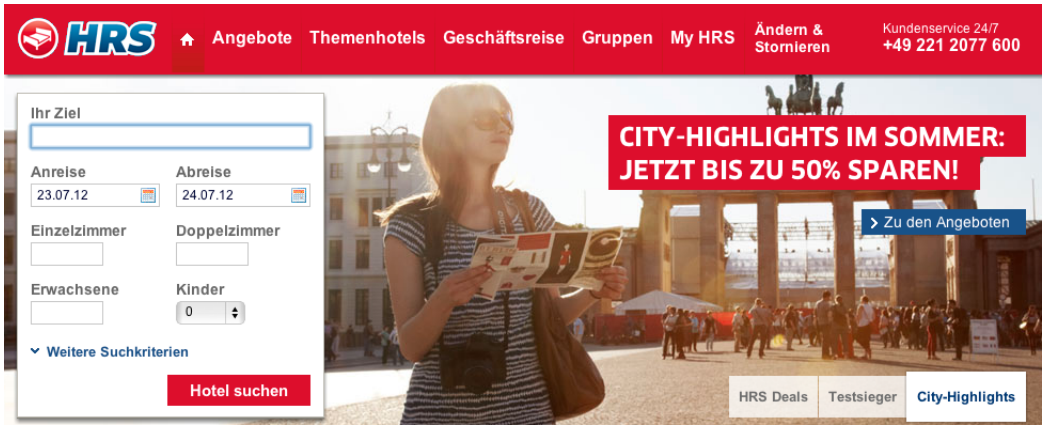

Abbildung 1.1.: Heutiger Stand der Homepage des Touristikportals HRS, Quelle: `http://www.hrs.com` (letzter Abruf am 2012-07-23).

Im Folgenden wird zunächst der Projektkontext dargelegt und im weiteren Verlauf erörtert, welche Rolle ausgereifte Suchmöglichkeiten für die Tourismusbranche spielen und warum es wichtig ist, sich dabei an der Funktionalität von Google zu orientieren.

[1] `http://www.hrs.com` (letzter Abruf am 2012-07-23).

1.1. Projekt TourInDi

Tourism in Dialogue (TourInDi) ist ein Kooperationsprojekt zwischen der Hochschule Darmstadt und dem Portal HRS[2] mit dem Ziel, die Hotelsuche der Zukunft zu entwickeln. In diesem Rahmen werden neue Methoden und Funktionalitäten zur Hotelsuche erforscht und prototypisch implementiert. Neben der für das Projekt namensgebenden Dialog-Funktionalität, die in ein kurzes Gespräch mit dem Benutzer zu treten versucht um die Hotelanfrage beratend zu spezifizieren, steht ein neues Eingabefeld für Freitext im Mittelpunkt der Entwicklung, mit dem das gewünschte Hotel in natürlicher Sprache gesucht werden kann. Dieses neue Eingabefeld ergänzt die bisher einzige Möglichkeit einer Suche nach Reisen auf dem Portal, der Suche über ein Formular.

Dialog

Freitext-suche

TourInDi setzt sich durch folgende Komponenten zusammen:

TourInDi-Portal
: Oberflächenlogik, beispielhaft implementiert als Demonstrationsanwendung.

TourInDi-Service
: Zentrale Vermittlungsschicht, die alle Komponenten miteinander verbindet und in richtiger Reihenfolge abruft.

TourInDi-Query Analysis
: Textanalyse-Komponente, die die natürlichsprachliche Hotelsuchanfrage des Benutzers interpretiert.

TourInDi-Recommendation
: Dialog-Komponente, die die Angaben des Nutzers durch gezielte Rückfragen spezifiziert.

TourInDi-Ontology
: Die zentrale Datenbasis für alle benötigten Informationen, etwa ein Verzeichnis aller Hotels.

Diese Arbeit konzentriert sich auf Architektur und Implementierung von Query Analysis, der Komponente zur Textanalyse von Hotelsuchanfragen in natürlicher Sprache.

[2] `http://www.hrs.com` (letzter Abruf am 2012-07-23).

1.1.1. Datenmodell

Query Analysis teilt das Datenmodell mit anderen TourInDi-Komponenten. Eine wichtige Rolle spielen dabei *Constraint*-Objekte. Constraints stellen die verschiedenen Aspekte einer Hotelsuchanfrage in einem maschinenlesbaren Format dar. Sie existieren in mehreren Ausprägungen bzw. Spezialisierungen der abstrakten Klasse `Constraint`:

`AccommodationConstraint`
: repräsentiert Ausstattungsmerkmale der gesamten Unterkunft.

`UnitConstraint`
: repräsentiert Ausstattungsmerkmale eines Zimmers der Unterkunft.

`LocationConstraint`
: steht für ortsbezogene Anforderungen, etwa an die geographische Lage.

`MoneyConstraint`
: stellt kostenbezogene Anforderungen bzw. Grenzen dar.

`TimeConstraint`
: steht für zeitliche Vorgaben und Einschränkungen.

`PersonConstraint`
: repräsentiert nähere Angaben zu den reisenden Personen, etwa Anzahl und Alter.

Eine einzelne natürlichsprachliche Hotelsuchanfrage wird folglich durch ein oder mehrere dieser Constraints repräsentiert. Für konkrete Anfragen werden diese Constraints im Datenmodell von TourInDi in einem `AccommodationRequest` zusammengefasst.

AccommodationRequest

Die technische Erkennungsleistung von Query Analysis besteht darin, jede natürlichsprachliche Anforderungen des Benutzers ihrer jeweiligen, internen Repräsentation als TourInDi-*Code* zuzuordnen. Dies erfolgt auf Basis der TourInDi-*Codeliste*, welche neben eindeutigen Schlüsseln für jeden Code auch Angaben zur Art des Constraints, zu seinen möglichen Datentypen, zu Synonymen und zu äquivalenten Codes der OpenTravel Alliance (OTA) macht. Insbesondere die Datentypen bestimmen die genaue Art der Verwendung von Codes: einfache Typen wie Boolean für Suchanfragen wie {`Hat einen Pool`} sind ebenso möglich wie *attributierbare* Codes, um etwa Anfragen wie {`Hat einen geheizten Pool`} zu repräsentieren.

Codeliste

Um diese Einstellungen im Detail und für jeden Code einzeln bestimmen zu können, ohne die Programmierung ändern zu müssen, liegt die Codeliste zur leichten administrativen Pflege als Comma Seperated Values (CSV)-Datei vor.

1.1.2. Releases von TourInDi

Die in dieser Arbeit entwickelte Software der Query Analysis-Komponente ist Gegenstand des Releases 3.0 von TourInDi .

Das vorausgehende Release 2.0 der Query Analysis nutzte zur Strukturierung der natürlichsprachlichen Hotelsuchanfragen in ein maschinenlesbares Format den Webservice Expert Virtual Agent (EVA) von Evature[3], sowie ein manuelles Mapping, um die Ausgabe von EVA auf das interne TourInDi-Format abzubilden. Dabei beschränkte es sich auf `AccommodationConstraints` und `UnitConstraints`.

Vor geplanter Beendigung und Übergabe des Projekts erfolgt ein weiteres Release mit der Nummer 4.0

[3] `http://www.evature.com/` (letzter Abruf am 2012-07-30).

1.2. Motivation dieser Arbeit

Maßstab Google

Googles allgegenwärtige Präsenz hat die Art und Weise, Informationen einfach aufzufinden, revolutioniert. Bis heute setzt das Unternehmen Maßstäbe, an denen sich auch die Suchangebote einzelner Portale messen lassen müssen. Die Motivation dieser Arbeit ist es daher, den Komfort der Freitextsuchanfrage, den ein Internetnutzer von Suchmaschinen wie Google kennt, auch bei bisher formulargestützen Suchmasken erfolgreich anbieten zu können. Mit der Bereitstellung semantischer Features sollen zudem die Vorteile gegenüber Googles größtenteils syntaktischer Suche zum Tragen kommen.

Im Folgenden wird dargelegt, welche Rolle qualitative Suchangebote in der Tourismus-Branche spielen. Unter der Prämisse, dass Benutzer Google-ähnliche Suchangebote mit einem Freitexteingabefeld zunächst auch wie in Google nutzen, werden außerdem grundlegende Benutzungsmuster von Google zur Orientierung dargestellt.

1.2.1. Suchmöglichkeiten in der Reisebranche

Bedeutung

Statistiken zeigen die hohe Bedeutung von Suchmöglichkeiten in der Reise- und Touristikbranche. So weist die Produktgruppe Reisen die höchste Anzahl an Suchanfragen vor einer Onlinebestellung im Vergleich zu anderen Produktgruppen auf[4]. Gleichzeitig erfolgt eine tatsächliche Onlinebestellungen in der Reisebranche mit 35% am häufigsten auf den Seiten, die bei der ersten Suchanfrage gefunden wurde[5].

Anspruch

Diese Zahlen können so ausgelegt werden, dass sich Interessierte in der Tourismusbranche generell intensiver über mögliche Reisen informieren, sie aber gleichzeitig bereits bei der ersten Suche sehr hochwertige Ergebnisse erhalten. Zugleich kann man auf gut bedien- und verfügbare, leicht zugängliche Suchmöglichkeiten schließen und auf einen hohen Druck auf der Seite der Anbieter, das jeweilige Angebot bei den ersten Ergebnissen zu platzieren.

[4] http://de.statista.com/statistik/daten/studie/3600/umfrage/anzahl-der-suchanfragen-im-internet-vor-einer-onlinebestellung/ (letzter Abruf am 2012-07-23).

[5] http://de.statista.com/statistik/daten/studie/3707/umfrage/anteil-der-onlinebestellung-nach-erster-suchanfrage/ (letzter Abruf am 2012-07-23).

1.2.2. Freitextsuchen in Zeiten von Google

Wie kein anderes Unternehmen bestimmt und definiert Google heutzutage die Art und Weise, wie Informationen im World Wide Web (WWW) gefunden werden können. In Deutschland etwa wurden im April 2011 neun von zehn Suchanfragen mit der Suchmaschine von Google durchgeführt[6], zweit- bis drittplatzierte Suchmaschinenanbieter weisen einen Anteil von höchstens 2,4% auf. Dabei wirkt sich diese marktbestimmende Dominanz nicht nur auf Umfang und Inhalt der erhaltenen Ergebnisse aus, auch die erwartete User Experience bei der Bedienung von vergleichbaren Suchangeboten ist stark durch Google geprägt. Folglich empfiehlt es sich nicht nur bei der Search Engine Optimization (SEO), also der Optimierung von Webangeboten gegen bessere Ergebnisrankings, sondern auch bei der Bereitstellung von Suchfunktionalität, sich an Googles Fähigkeiten zu orientieren.

Googles Dominanz

Auswirkungen

1.2.2.1. Anzahl an Suchbegriffen pro Suche

Ein wichtiges Charakteristikum einer Suchanfrage ist die Anzahl an einzelnen Suchbegriffen, aus der sie besteht. Verwendet ein Nutzer bei Google etwa eine *Mehrwort-Suchabfrage* , also Suchabfragen mit mehr als einem Suchbegriff, werden ihm bei den Ergebnissen nur die Seiten gezeigt, die auch alle Suchbegriffe wirklich enthalten - dieses Verhalten ist bei Phrasen nicht bei jedem Suchanbieter so implementiert [Gre-2010, Seite 39].

Charakteristikum

In einer von Google veröffentlichten Studie zum Suchverhalten von Internetnutzern in Privathaushalten in Deutschland aus dem vierten Quartal 2010 geht hervor, dass über 70% aller Suchanfragen aus mehr als einem Wort bestehen und dass eine Suchabfrage in Deutschland durchschnittlich 2,2 Wörter enthält [GfK-2011, Seite 8]. Jede dritte Suchanfrage besteht aus zwei Suchbegriffen, was die häufigste Variante darstellt, gefolgt von einem Suchbegriff mit 28% und drei Suchbegriffen mit 20%. Immerhin jede zehnte Suchabfrage besteht noch aus vier Worten, Suchabfragen mit fünf oder mehr Worten weisen nur noch eine Häufigkeit im einstelligen Prozentbereich auf.

Suchbegriffe in Deutschland

[6] http://de.statista.com/statistik/daten/studie/167841/umfrage/marktanteile-ausgewaehlter-suchmaschinen-in-deutschland/ (letzter Abruf am 2012-07-20).

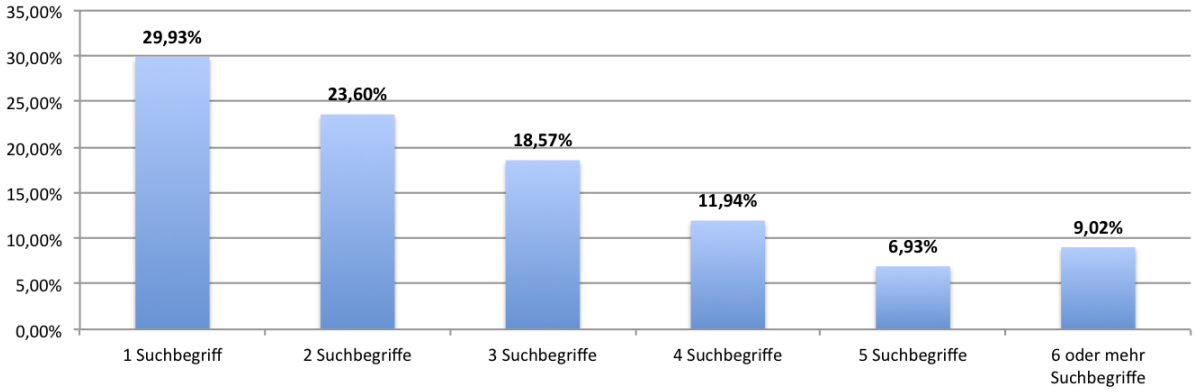

Abbildung 1.2.: Anteil der Suchanfragen im Internet nach Anzahl eingegebener Suchbegriffe in den USA (Stand: Mai 2012), nach `http://de.statista.com/statistik/daten/studie/3595/umfrage/anzahl-der-keywords-pro-suchanfrage/` (letzter Abruf am 2012-07-21).

Ähnliche, wenngleich nicht identische, aktuelle Zahlen zeigt Abbildung 1.2 aus einer Erhebung von Experian[7] vom Mai 2012: Die Häufigkeit von Suchabfragen mit genau einem Suchbegriff und drei Suchbegriffen entsprechen mit 29,33% bzw. 18,57% den aus Deutschland erhobenen Häufigkeitswerten. Allerdings tritt die in Deutschland häufigste Variante mit zwei Suchbegriffen circa zehn Prozent seltener auf - zugunsten von umfangreicheren Phrasen bestehend aus vier, fünf und insbesondere sechs oder mehr Suchbegriffen.

Suchbegriffe in den USA

Horst Greifeneder beschreibt hierzu einen Trend, dass Suchmaschinennutzer „immer öfter [...] Wortpaare oder -tripels verwende[n]“ [Gre-2010, Seite 38].

Trend zu Phrasen

1.2.2.2. Nutzung spezieller Operatoren

Ein weiteres Charakteristikum einer Suchanfrage stellt die Verwendung von speziellen Operatoren dar, also Zeichen, die sich in besonderer Weise auf das Verhalten der Suchmaschine auswirken. Populäre Operatoren sind etwa Anführungszeichen `"`, mit deren Hilfe exakte (Teil-)Sätze gefunden werden können, oder der Oder-Operator `OR`, mit dem mehrere Teilsuchen miteinander kombiniert werden können [Rus-2012].

`"` / `OR`

[7] `http://de.statista.com/statistik/daten/studie/3595/umfrage/anzahl-der-keywords-pro-suchanfrage/` (letzter Abruf am 2012-07-21).

Operatoren werden von den Google-Nutzern in Deutschland kaum genutzt, lediglich 14% der Suchmaschinennutzer machen davon Gebrauch [GfK-2011, Seite 8]. So ist es auch wenig verwunderlich, dass bereits Operatoren aus Google entfernt wurden[8]. Nutzung

1.3. Aufbau dieser Arbeit

Diese Arbeit ist wie folgendermaßen aufgebaut:

Kapitel 2 auf der nächsten Seite definiert die Anforderungen an die zu entwickelnde Software an Erkennungsleistung, Erweiterbarkeit und Performance.

Kapitel 3 ab Seite 14 stellt wesentliche Grundlagen dieser Arbeit vor. Dazu gehören vor allem Funktionsweisen und Paradigmen der Open-Source-Projekte Apache UIMA und JBoss Drools, aber auch Disziplinen des Natural Language Processings und Information Retrievals finden ihre Anwendung.

Kapitel 4 ab Seite 39 wird auf Basis des Grundlagenkapitels eine technische Lösung zur Information Extraction natürlichsprachlicher Phrasen konzipiert und entwickelt.

Kapitel 5 ab Seite 73 dokumentiert zunächst die Entwicklung eine Frameworks für die systematische Bewertung der zuvor entwickelten Software. Weiter folgt eine Evaluierung gemäß der Methoden des Information Retrievals.

Kapitel 6 ab Seite 84 stellt verwandte technische Lösungen vor und fasst die Arbeit zusammen. In einem Ausblick werden Erfahrungen des Autors dokumentiert. Abschließend erfolgt eine Bewertung der entwickelten Software anhand konkurrierender Lösungen.

Unter den Anhängen ab Seite I finden sich alle Trainings- und Verifikationssuchanfragen zusammen mit ihren Erkennungsergebnissen.

[8] Zugunsten des eigenen sozialen Netzwerks Google+ wurde im Oktober 2011 der Suchoperator + entfernt und mit neuer Semantik für die soziale Suche versehen, `http://www.at-web.de/blog/20111025/google-entfernt-plus-operator.htm` (letzter Abruf am 2012-07-23).

Kapitel 2.

Anforderungen

Hauptaufgabe der zu entwickelten Komponente ist es, natürlichsprachliche Aussagen des Benutzers in Teilaussagen zu zerlegen, diese den korrekten Schlüsselworten der Codeliste (siehe Abschnitt 1.1.1 auf Seite 3) zuzuordnen und die passenden Constraints (siehe Abschnitt 1.1.1 auf Seite 3) zu generieren, sodass eine technische Abfrage der Datenbasis möglich wird.

Jede einzelne dieser Teilaussagen stellt dabei eine Spezifikation des Benutzers an das Hotelangebot dar und ist daher möglichst vollständig zu interpretieren. Die genauen Anforderungen, die dabei gelten, sowie die Erfolgskriterien werden im Folgenden definiert. Grundlegend gibt es die drei Anforderungsbereich Erkennungsleistung, Erweiterbarkeit und Performance.

2.1. Erkennungsleistung

Die Anforderung an die Erkennungsleistung hat verschiedene Aspekte, die im folgenden detaillierter aufgeführt sind.

2.1.1. Gegenstand der Erkennung

Query Analysis soll natürlichsprachliche Hotelsuchanfragen des Benutzers in ein oder mehrere Teilaussagen zum gesuchten Hotelangebot strukturieren können und jede dieser Teilaussagen möglichst vollständig den korrekten Schlüsselworten der 1.1.1 zuordnen können.

Teil-aussagen

Es ist davon auszugehen, dass diese natürlichsprachlichen Hotelsuchanfragen in der Regel keine vollständigen, syntaktisch korrekten Sätze sind, sondern sich, wie in der Motivation in Abschnitt 1.2 auf Seite 5 dargelegt, aus kurzen, auf das Wesentliche gekürzten Phrasen aus wenigen Worten zusammensetzen.

Charakteri-sierung

2.1.2. Trennzeichen und Operatoren

Die Eingabe von Hotelsuchanfragen soll intuitiv durch den Benutzer erfolgen. Er sollte dabei keine Einschränkungen für seine Eingaben, zum Beispiel durch Vorgabe bestimmter, reservierter Schlüsselwörter, erfahren. Allerdings wird davon ausgegangen, dass der Benutzer seine Phrasen in einer besonderen, textuellen Form syntaktisch voneinander trennt, etwa durch Kommata `,`, Pluszeichen + oder Bindeworte wie `and`.

intuitive Eingabe

Auf die Implementierung von speziellen Operatoren mit eigener Suchsemantik wie zum Beispiel der +-Operator bei Google wird aufgrund seiner geringen Akzeptanz bei den Benutzern (siehe Abschnitt 1.2.2.2 auf Seite 7) verzichtet.

Operatoren

2.1.3. Erzeugte Constraints

Ebenso wie die Vorgängerimplementierung des Release 2 (siehe Abschnitt 1.1.2 auf Seite 4) wird sich auf die Erkennung von Hotel- und Zimmerausstattungen reduziert; Query Analysis erzeugt folglich ausschließlich AccommodationConstraint und UnitConstraint Objekte (siehe Abschnitt 1.1.1 auf Seite 3). Andere, im Datenmodell von TourInDi mögliche Constraints wie LocationConstraints für ortsbezogene, MoneyConstraints für kostenbezogene, TimeConstraints für zeitbezogene und PersonConstraints für personenbezogene Angaben bleiben im Rahmen dieser Arbeit außen vor.

Hotel- und Zimmer-angaben

2.1.4. Fehlertoleranz

Um eine gewisse Realitätsnähe vorweisen zu können, ist eine Fehlertoleranz bei der Auswertung der Benutzereingaben notwendig. Dies gilt besonders für ein prominent-platziertes Eingabefeld, über das der Benutzer, wie bereits in der Anforderung „Performance“ auf Seite 13 erwähnt, zum häufigen Suchen animiert werden soll.

Realitäts-nähe

Insbesondere mit fehlenden Buchstaben oder Buchstabendrehern, also etwa bei der Eingabe von `Polo` statt `Pool`, soll die Komponente in den meisten Fällen ohne negative Effekte auf die Erkennung umgehen und die korrekte Variante erkennen können.

Buch-staben-dreher

2.1.5. Mehrsprachigkeit

Auf Mehrsprachigkeit wird im Rahmen dieser Arbeit verzichtet, Nutzereingaben müssen in Englisch nach der Vorgabe der OTA, auf der auch die Codeliste (siehe Abschnitt 1.1.1 auf Seite 3) basiert, formuliert werden.

2.1.6. Negation

Die Unterstützung von natürlichsprachlicher Negation, d.h. durch Worte wie {no} und {without}, stellt einen echten Mehrwert der Texterkennung gegenüber einer klassischen, Schlagwort-basierten Suche wie bei Google (siehe Abschnitt 1.2.2 auf Seite 6) dar. Mehrwert

Negation soll daher folgendermaßen unterstützt werden: wird ein Teil einer Hotelsuchanfrage negiert, wie beispielsweise in {no pool}, so legt die Texterkennung auch keinen entsprechenden Constraint an. Der negierte Teil wird folglich von Query Analysis ignoriert. Dies stellt eine geeignete Arbeitsgrundlage für nachfolgende Releases dar, um - bei Bedarf - andere Repräsentationsformen für die Negation im Datenmodell von TourInDi zu implementieren. Implementierung der Negation

2.1.7. Erkennungsrate

Für die Akzeptanz der neuen Suchmöglichkeit auf dem Portal ist vor allem eine gute Erkennungsrate entscheidend. Von der Texterkennung wird daher ein F-measure[1] von 80% oder höher als ein realistisches Ziel für eine brauchbare Erkennungsleistung gefordert. F-measure

2.2. Erweiterbarkeit

Erweiterbarkeit ist ein wichtiger Vorteil einer Eigenentwicklung gegenüber Drittanbieterangeboten wie dem zuvor eingesetzten EVA (siehe Abschnitt 1.1.2 auf Seite 4). Als zentrales Entwicklungsziel sollten Details der Textanalyse leicht einzusehen und anzupassen sein, zudem sollte neue Funktionalität ohne große Aufwände und Abhängigkeiten integrierbar sein. Vorteil Anpassbarkeit

[1] Das F-measure ist ein übliches Maß für die Beurteilung von Texterkennungslösungen und wird im Grundlagenkapitel (siehe Abschnitt 3.1.1 auf Seite 16) näher erläutert.

2.3. Performance

Die Verarbeitungsgeschwindigkeit von Query Analysis unterliegt zeitlichen Einschränkungen, die letztlich in der Usability von TourInDi begründet sind: Der Benutzer des Portals soll zum häufigen Suchen animiert werden, um so seine Suchanfrage schrittweise zu verfeinern. Dauert eine Suchanfrage zu lange, verliert der Nutzer sein Interesse.

Usability

Als allgemein akzeptabel gilt eine Reaktions- bzw. Verarbeitungsgeschwindigkeit im Sekundenbereich, da dies den Gedankenfluss eines menschlichen Benutzers nicht unterbricht und diese üblicherweise als benötigte Arbeitszeit dem Computer zugestanden wird [Ins-2010].

Sekundenbereich

Da Query Analysis Teil einer Gesamtarchitektur ist und andere Komponenten bei der Verarbeitung von Benutzereingaben involviert sind, wird eine Verarbeitungsgeschwindigkeit der Komponente im Bereich von 300-600 Millisekunden gefordert.

Gesamtarchitektur

Kapitel 3.

Grundlagen

Dieses Kapitel erläutert einige Grundlagen der Arbeit, die für ein tiefgreifendes Verständnis der folgenden Kapitel vorausgesetzt werden.

Die Arbeit ist im Bereich des Information Retrieval bzw. des Information Extraction angesiedelt, sodass auf diese beiden Thematiken, deren Paradigmen und grundlegende Evaluationsmethoden einführend eingegangen wird. Weiter werden populäre und übliche Aufgaben des Natural Language Processings dargestellt und deren Funktionsweise skizziert. Einige dieser Disziplinen finden im weiteren Verlauf der Arbeit ihre praktische Anwendung.

Der Arbeit liegen auch Grundlagen größerer Open-Source-Projekte zugrunde, wie der Informationsverarbeitungspipeline und OASIS-Standard Apache UIMA, dessen technische Aufsätze uimaFIT und DKPro, sowie der Regelengine JBoss Drools. Diese stellen zentrale Bestandteile der Arbeit dar und sind für ein tiefgreifendes Verständnis dieser Arbeit wesentlich.

3.1. Informationen Retrieval und Extraction

Gegenstand des Information Retrieval (IR) sind strukturierte und un- beziehungsweise semi-strukturierte Informationen. Als *unstrukturierte Informationen* bezeichnet man dabei Daten ohne klare, Computer-verständliche Struktur [MRS-2009, Seite 1] und ohne explizite Semantik, etwa alle Produkte direkter menschlicher Kommunikation, wie zum Beispiel natürlichsprachliche Dokumente [LVN-2009, Chapter 1]. strukturlos

Als *strukturierte Informationen* wird das genaue Gegenteil bezeichnet [MRS-2009, Seite 1] - Daten mit Informationen darüber, wie sie im Sinne des menschlichen Autors zu interpretieren sind [LVN-2009, Chapter 1] - Daten, die in relationalen Datenbanken gespeichert werden, etwa in Produktdatenbanken [MRS-2009, Seite 1]. strukturiert

In natura liegen beispielsweise in Unternehmen 80% der Informationen unstrukturiert vor, im WWW sind sie überwiegend durch die Hypertext Markup Language (HTML) semi-strukturiert [Her-2009b, Seite 7]. Unstrukturierte Informationen stellen zudem die am schnellsten wachsende Wissensquelle bei Unternehmen und Regierungen dar [LVN-2009, Chapter 1] in natura

Zusammen mit dem *Informationsbedürfnis* (*Information Need*) - den Inhalten, über die der Benutzer gerne mehr wissen möchte [MRS-2009, Seite 5] - definieren Manning et al. die Tätigkeiten des Information Retrieval folgendermaßen [MRS-2009]: Bedürfnis

> "Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need fromwithin large collections (usually stored on computers)."

IR wird im zunehmenden Sprachgebrauch auch mit dem Begriff der *Suche* gleichgesetzt und etabliert sich nach und nach als die häufigste Art und Weise, um auf Informationen zuzugreifen[1] [MRS-2009, Seite 1]. Ständige Herausforderung des IR ist es dabei, den *Information Overload* zu vermeiden: das genannte Informationsbedürfnis des Nutzers zu Suche Overload

[1] Der Trend zeigt sich beispielsweise auch bei den immer umfangreicheren Möglichkeiten von Desktopsuchmaschienen, die in moderne Betriebssysteme eingebettet werden. Saved Search Folders unter Mac OS X bzw. Search Folders unter Windows 7 machen es für den Benutzer immer unwichtiger, unter welchem Pfad genau eine Datei tatsächlich gespeichert ist.

befriedigen, ihn aber gleichzeitig nicht mit Informationen zu überfluten und ihm „nur passende[] Dokumente (bzw. Links) oder Fakten zu liefern.“ [Her-2009b, Seite 7]. Diese passenden Dokumente werden auch als *relevant* bezeichnet [MRS-2009, Seite 5]. relevant

Ergänzend dazu beschreibt die Disziplin der Information Extraction (IE) weitere Aktivitäten, um aus gefundenen, relevanten Dokumenten bestimmte Fakten über vordefinierte Ereignisse, Entitäten oder Beziehungen zu extrahieren [Uni-2010]. In einem IE -System kommen üblicherweise mehrere Module zum Einsatz, die, mit Hilfe von manuellen oder automatischen Regeln, schrittweise Information anfügen oder entfernen [Cha-2001, Seite 17]. Information Extraction

Zur Bemessung der *Effektivität* eines IR -Systems und der Qualität seiner Ergebnisse existieren etablierte Evaluierungsmethoden, die im Folgenden vorgestellt werden.

3.1.1. Evaluierung für Information Retrieval -Systeme

Der Standardansatz zur Evaluierung von IR -Systemen dreht sich um die (binäre) Unterscheidung zwischen relevanten (siehe Abschnitt 3.1) *Dokumenten* und irrelevanten Dokumenten. Es wird eine *Testsammlung* aus einer Menge an Dokumenten zusammengestellt, bei der für jedes enthaltene Dokument unter Berücksichtigung jeweiliger Informationsbedürfnisse (siehe Abschnitt 3.1 auf der vorherigen Seite) entschiedenen wird, ob es relevant ist oder nicht. Diese Sammlung wird auch als *Gold Standard* bezeichnet und besteht idealerweise aus einer Vielzahl unterschiedlichster Dokumente und Informationsbedürfnisse - als Faustegel für eine ausreichende Menge gelten etwa 50 Informationsbedürfnisse.[MRS-2009, Seite 152] Dokument Testsammlung Golden Standard

Davon ausgehend ergeben sich die zwei wesentlichen Maße Precision (P) für die Genauigkeit [MRS-2009, Seite 154] und Recall (R) für die Ausbeute [Lan-2006, Seite 3].

Recall (R) setzt die Anzahl der gefundenen, relevanten Dokumente ins Verhältnis zu allen relevanten Dokumenten in der Datenbank [Luc-2005, Seite 3], siehe Formel 3.1 auf der nächsten Seite: Recall

$$\text{Recall } R = \frac{\text{Anzahl der gefundenen relevanten Dokumente}}{\text{Anzahl aller relevanter Dokumente im Testset}} \tag{3.1}$$

R ist ein Maß für die Substanz des Testsets und liegt zwischen 0 und 1; $P = 1$ bedeutet dabei, dass alle relevanten Dokumente auch gefunden wurden [Luc-2005, Seite 3].

Precision (P) setzt die Anzahl der gefundenen, relevanten Dokumente ins Verhältnis zur Zahl der insgesamt gefundenen Dokumente [Luc-2005, Seite 3], siehe Formel 3.2: *Precision*

$$\text{Precision } P = \frac{\text{Anzahl der gefundenen relevanten Dokumente}}{\text{Anzahl aller gefundenen Dokumente}} \tag{3.2}$$

P ist ein Maß dafür, wieviel Unbrauchbares gefunden wurde und liegt wie R zwischen 0 und 1. $P = 1$ bedeutet, dass alle gefundenen Dokumente auch relevant sind [Luc-2005, Seite 3].

In Kontigenztabelle 3.1 werden Ergebnisse unter Berücksichtigung der Kriterien relevant / irrelevant und gefunden / nicht gefunden eingeordnet [MRS-2009, Seite 155]. Es ergeben sich die vier möglichen Einschätzungen t_p und t_n für jeweils korrekterweise gefundene (relevante) und korrekterweise nicht gefundene (irrelevante) Dokumente, sowie f_p und f_n für jeweils fälschlicherweise gefundene (irrelevante) und fälschlicherweise nicht gefundene (relevante) Dokumente. t_p und t_n, f_p und f_n

Tabelle 3.1.: Kontigenztabelle bei der Bewertung von IR -Ergebnissen, nach [MRS-2009, Seite 155].

	Relevant	Irrelevant
Gefunden	true positives (t_p)	false positives (f_p)
Nicht gefunden	false negatives (f_n)	true negatives (t_n)

Durch diese Klassifizierung können R und P auch folgendermaßen definiert werden:

$$\text{Precision } P = \frac{t_p}{t_p + f_p} \tag{3.3}$$

$$\text{Recall } R = \frac{t_p}{t_p + f_n} \tag{3.4}$$

Um R und P in einem Maß ins Verhältnis setzen zu können, wird ein gewichteter, harmonischer Mittelwert angegeben, der als das F-measure (F) bezeichnet wird und sich folgendermaßen definiert [MRS-2009, Seite 156]: *F-measure*

$$F = \frac{(\beta^2 + 1)PR}{\beta^2 P + R} \text{ mit } \beta \in [0, \infty] \tag{3.5}$$

Über die Gewichtung β können entweder Precision (P) (mit Werten für $\beta < 1$) oder Recall (R) (mit Werten für $\beta > 1$) schwerer gewichtet werden [MRS-2009, Seite 156]. Eine Gleichgewichtung von P und R mit $\beta = 1$ unternimmt das Standardmaß Balanced F-measure ($F_{\beta=1}$), welches sich folgendermaßen definiert [MRS-2009, Seite 156]: *balanced F-measure*

$$F_{\beta=1} = \frac{2PR}{P + R} \tag{3.6}$$

Statt als Gleitkommazahl zwischen 0 und 1 wird F auch häufig als Prozentzahl zwischen 0 und 100 dargestellt [MRS-2009, Seite 156].

3.1.2. Ansätze einer Information Extraction -Lösung

Douglas Appelt [App-1999] unterscheidet zwei mögliche Paradigmen, die bei der Funktionsweise von IE -Systemen verfolgt werden können.

Regelbasiert

Beim *Knowledge Engineering Approach* entwirft ein menschlicher Entwickler mit Kenntnis über das spezifische System und die konkreten Anforderungen der Domäne Regeln, die das System dann ausführt. Dabei kommen auch seine Intuition, sein Allgemeinwissen und „tricks of the trade“, also seine individuellen Tricks und Kniffe, zum Einsatz [App-1999, Seite 165].

Modellbasiert

Der *Learning Approach* basiert dagegen auf Algorithmen zum maschinellen Lernen, wie etwa Entscheidungsbäumen, Maximum Entropy Models oder Hidden Markov Modellen [App-1999, Seite 165].

Zur Entscheidungsfindung, welcher dieser beiden Ansätze gewählt werden soll, definiert Douglas Appelt folgende Überlegungen [App-1999, Seite 165f]:

Verfügbarkeit an Trainingsdaten Liegt ein großer Umfang an Trainingsdaten vor oder kann dieser leicht und schnell erzeugt werden, spricht dies klar für den Learning Approach.

Verfügbarkeit an linguistischen Resourcen Liegen linguistische Resourcen wie Lexika oder Wörterbücher vor, ist ein Knowledge Engineering Approach möglich. Andernfalls muss der Learning Approach eingesetzt werden.

Verfügbarkeit an Entwicklern Eine offensichtliche Bedingung für den Knowledge Engineering Approach ist die Verfügbarkeit von Entwicklern, die in der Lage sind, Regeln für das System zu formalisieren.

Stabilität der Anforderungen Generell sind die Regelsätze des Knowledge Engineering bei Änderungen der Anforderungen leichter zu modifizieren als die Trainingsdaten der Algorithmen des maschinellen Lernens erneut auszuzeichnen und trainieren zu lassen.

Geforderte Leistungsfähigkeit Generell haben Ansätze des maschinellen Lernens höhere Fehlerraten als vom Menschen erzeugte Regelsysteme.

3.2. Natural Language Proccessing

Im Folgenden werden einige benötigte Grundlagen aus dem Bereich des Natural Language Processing (NLP) dargelegt.

3.2.1. Tokenization

Jeder anspruchsvollen Verarbeitung von Text geht das Herunterbrechen in sinnvolle Teilstücke des Textes voraus. Der dafür am häufigsten verwendete Ansatz - das Unterteilen eines Textes in Sätze und Zeichenketten (*Tokens*) wie Wörtern - heißt *Tokenization*. [FS-2006, Seite 60]

Token

Eine der größten Herausforderungen bei der Tokenization von englischen und deutschen Texten besteht darin, Punkte als Signal für ein Satzende von den Punkten zu unterscheiden, die zu vorausgehenden Tokens gehören, wie etwa bei {`Mr.`} oder {`Dr.`}.

Punkt

Tabelle 3.2 veranschaulicht anhand einiger Beispiele, dass die Tokenization nicht nur bei Punkten im Detail schwierig sein kann, sondern auch bei Zeichen wie zum Beispiel Apostrophe.

Text	Anzahl Tokens
| that | 's | a | problem |	4
| I |'ve | got | a | problem |	5
| Daniel | O'Brian |	2
| end | of | sentence | . |	4
| his | name | is | Dr. | Faust |	5

Tabelle 3.2.: Tokenization-Beispiele verschiedener Zeichen (Token-Trennzeichen `|`), Quelle :[Str-2009, Seite 22].

3.2.2. Parts-of-Speech -Tagging

Einen wichtigen Teil des linguistischen Wissens über eine Sprache beschreiben die *Sprachbestandteile* Parts-of-Speech (PoS) , auch bekannt als *word classes* oder *lexical*

Parts-of-Speech

tags [JM-2000, Seite 287]. Beim *PoS-Tagging* werden Worte zu bestimmten, lexikalischen [Lan-2006, Seite 4] Klassen entsprechend ihres Kontextes zugewiesen [FS-2006, Seite 60] , wie zum Beispiel das Wort `Haus` der Klasse `Nomen` oder das Wort `spielen` der Klasse `Verben`. Tagging

Die verwendeten Klassen können auch feiner gewählt werden, beispielsweise um eine eigene Klasse für `Eigennamen im Plural` zu erhalten [MRS-2009, Seite 40]. Die genaue Anzahl an Klassen eines *Tagsets* kann dabei stark variieren und umfasst bei bekannten Listen wie der *Penn Treebank*[2] etwa 45 Klassen, 87 beim *Brown corpus*[3] und 146 beim *C7 tagset*[4] [JM-2000, Seite 288]. Tagset

Tabelle 3.3 zeigt eine Auswahl an Klassen für verschiedene Arten von Verben aus der Penn Treebank.

POS Tag	Description	Example
VB	verb, base form	take
VBD	verb, past tense	took
VBG	verb, gerund/present participle	taking
VBN	verb, past participle	taken
VBP	verb, sing. present, non-3d	take
VBZ	verb, 3rd person sing. present	takes

Tabelle 3.3.: Auswahl an PoS-Tags für Verben aus der Penn Treebank, Quelle: `http://www.mozart-oz.org/mogul/doc/lager/brill-tagger/penn.html` (letzter Abruf am 2012-08-14).

Durch ihre Arbeitsweise[5] erzeugen PoS -Tagger häufig als Nebenprodukt auch die Wortstämme (*Lemmata*) der verarbeiteten Tokens [FS-2006, Seite 60]. Lemma

Durch Verarbeitung der PoS können viele, semantische Informationen über einzelne Worte und ihre Nachbarn gewonnen werden [FS-2006, Seite 60], vor allem durch feinere Unterscheidungen ihrer Klassen, sodass sie als nützliches Modell für die Spracherkennung dient [JM-2000, Seite 288]. Semantik

[2] `http://www.cis.upenn.edu/~treebank/` (letzter Abruf am 2012-08-14).

[3] `http://www.comp.leeds.ac.uk/ccalas/tagsets/brown.html` (letzter Abruf am 2012-08-14).

[4] `http://ucrel.lancs.ac.uk/claws7tags.html` (letzter Abruf am 2012-08-14).

[5] PoS -Tagger weisen Tokens üblicherweise mit Hilfe einer morphologischen Analyse einer Klasse zu [FS-2006, Seite 60]. Dabei wird untersucht, wie sich Worte aus *Morphemes*, den kleinsten, bedeutungstragenden Bestandteilen einer Sprache, den Morphemes, zusammensetzen [JM-2000, Seite 59].

3.2.3. Parsing

> „Parsing means taking an input and producing some sort of structure for it.“ [JM-2000, Seite 57].

Beim (syntactical) *Parsing* werden Sätze einer syntaktischen Analyse unterzogen und gemäß einer bestimmten *Grammatik* einer Struktur zugewiesen [FS-2006, Seite 60]. Dabei unterteilt man zugrundeliegende Grammatiken grundsätzlich in zwei Arten:

Konstituentengrammatiken beschreiben die syntaktische Struktur eines Satzes mit Hilfe von verschachtelten Teilausdrücken, *Konstituenten*, die zusammengehörige Wordsequenzen gruppieren [FS-2006, Seite 60]. Der gesamte Satz setzt sich folglich rekursiv durch seine schrittweise verschachtelten Konstituenten vollständig zusammen. Listing 3.1 zeigt ein Beispiel für einen Satz, der mit einer derartigen Grammatik analysiert wurde.

Konstituent

Listing 3.1: Parsing-Ergebnis eines Bespielsatzes des Stanford Parsers mit Konstituentengrammatik `http://nlp.stanford.edu:8080/parser/index.jsp#sample` (letzter Abruf am 2012-09-07)

```
1 (ROOT
2     (S
3       (NP (PRP$ My) (NN dog))
4       (ADVP (RB also))
5       (VP (VBZ likes)
6         (S
7             (VP (VBG eating)
8             (NP (NN sausage)))))
9     (. .)))
```

Dependenzgrammatiken betrachten nicht die Bestandteile eines Satzes als einzelne Konstituierten, sondern die jeweiligen Beziehungen zwischen Worten [Str-2009, Seite 25]. Üblicherweise werden diese Beziehungen zusammen mit der genauen grammatikalischen Art ihrer Abhängigkeit angegeben, zum Beispiel als Subjekt [Str-2009, Seite 25]. Listing 3.2 auf der nächsten Seite zeigt ein entsprechendes, beispielhaftes Parsing-Ergebnis.

Beziehungen

Listing 3.2: Parsing-Ergebnis eines Bespielsatzes des Stanford Parsers mit Dependenzgrammatik `http://nlp.stanford.edu:8080/parser/index.jsp#sample` (letzter Abruf am 2012-09-07)

```
1 poss(dog-2, My-1)
2 nsubj(likes-4, dog-2)
3 advmod(likes-4, also-3)
4 root(ROOT-0, likes-4)
5 xcomp(likes-4, eating-5)
6 dobj(eating-5, sausage-6)
```

Die Parsing-Ergebnisse beider Grammatiken können auch als Baumstruktur (*Parsing Tree*) visualisiert werden.

3.2.4. Named Entity Recognition

Named Entity Recognition (NER) bezeichnet das Erkennen (Detection) und Klassifizieren (Classification) von Named Entities, etwa Eigennamen von Personen, Organisationen und Orten, und ist folglich die Kombination einer Named Entity Detection (NED) und einer Named Entity Classification (NEC) [Lan-2006, Seite 3]. Eine allgemeingültige Definition von Named Entities existiert von Joel Lang [Lan-2006]: NER

> „Named Entities (NEs) sind alle Wörter und Wortsequenzen eines natürlichsprach-lichen Textdokuments, welche für den Benutzer des NER -Systems von besonderem Interesse sind.“ [Lan-2006, Seite 2] Entities

Eine NER stellt eine der fundamentalsten Aufgaben einer Information Extraction dar; auch die Extraktion komplexer Strukturen basiert häufig auf einer NER [AZ-2012, Seite 15]. Dabei können NER -Implementierungen grundsätzlich durch drei verschiedene Ansätze realisiert werden: Ansätze

3.2.4.1. Listenbasierte und statistische Verfahren

Bei der *listenbasierten NER* kommt eine Liste bzw. ein Wörterbuch (*Dictionary*) zum Einsatz, welches alle zu klassifizierenden Worte inklusive ihrer morphologischen[6] Varianten einer jeweiligen Klassifizierung zuordnet. Die eigentliche NER erfolgt dann durch den einfachen Vergleich von Wortsequenzen mit Wörterbucheinträgen. [Lan-2006, Seite 3]. Wörterbuch

Dagegen basiert das *statistische NER* üblicherweise auf Algorithmen des statistischen Machine Learnings. Hierbei wird die Problemstellung der NER als *Sequence labeling*[7]-Problem verstanden, welches auch bei vielen anderen Aufgaben des NLP zum Einsatz kommt. Dabei wird versucht, an eine gegebene Menge Tokens $x = (x_1, x_2, ..., x_n)$ die Klassifizierungen aus dem Vektor y zu annotieren, wobei jedem x_i ein entsprechendes y_i zugeordnet werden soll. Die Besonderheit dabei ist, dass die Klassifierung y_i nicht nur vom Token x_i des ursprünglichen Textes abhängt, sondern auch anderen Tokens, üblicherweise von solchen, die in der direkten Nachbarschaft von x_i liegen. Für die Lösung kommen normalerweise Hidden Markov Modelle[8] zum Einsatz. [AZ-2012, Seite 17f]

3.2.4.2. Regelbasierte Verfahren

Bei der regelbasierten NER entscheiden Regeln, „wann eine Sequenz als Named Entity markiert werden soll und zu welcher Klasse eine Named Entity gehört“ [Lan-2006, Seite 3]. Dabei wird eine Menge an Regeln entweder manuell definiert oder automatisch erzeugt und anschließend auf den zu verarbeitenden Text angewendet [AZ-2012, Seite 16]. regelbasiert

Bei regelbasierten Verfahren kommen üblicherweise verschiedene Wissensformen in Kombination zum Einsatz, zum Beispiel Informationen aus Wörterbüchern zusammen mit syntaktischem und domänenspezifischen Wissen [Lan-2006, Seite 3]. Wissensformen

[6] Die Worte einer Sprache setzen sich aus einem oder mehreren Morphemes, den kleinsten, in einer Sprache Bedeutungstragenden Bestandteilen, zusammen [JM-2000, Seite 59].

[7] Siehe auch: Wikipedia Artikel zum Sequence labeling: `http://en.wikipedia.org/wiki/Sequence_labeling` (letzter Abruf am 2012-08-28)

[8] Siehe auch: Wikipedia Artikel zu Hidden Markov Modellen, `http://de.wikipedia.org/wiki/Hidden_Markov_Model` (letzter Abruf am 2012-08-28)

Regeln bestehen aus je einem *Pattern*, etwa einem regulärer Ausdruck für bestimmten Texteigenschaften wie PoS (siehe Abschnitt 3.2.2 auf Seite 20), und einer *Action*, etwa die Durchführung einer Markierung, die ausgeführt wird, sobald eine Zeichenkette die Bedingungen des Pattern erfüllt [AZ-2012, Seite 16f]. Pattern Action

Um beispielsweise eine beliebige Wortsequenz der Form `Mr. X`, wobei `X` ein großgeschriebenes Wort ist, als Person-Named Entity zu kennzeichnen, kann die in Listing 3.3 Regel definiert werden [AZ-2012, Seite 17]: Beispiel

Listing 3.3: Beispiel einer Regel für die regelbasierte NER, Quelle: (Aggarwal und Zhai, 2012, Seite 17)

```
1   (token = "Mr." orthography type = FirstCap) -> person name.
```

Der vordere Teil der Regel in Klammern stellt dabei den regulären Ausdruck dar, der für eine beliebige Wortsequenz aus zwei aufeinander folgenden Worten zutrifft, bei dem das erste Wort `Mr.` ist und das zweite Wort vom orthographischen Typ `FirstCap` ist. Der hintere Teil der Regel drückt aus, die zuvor erkannte Wortsequenz als Named Entity `person name` zu kennzeichnen. [AZ-2012, Seite 17] Die abgebildete Regel nutzt zur Erkennung der NEs sowohl eine interne als auch eine externe *Evidenz* - d.h. es werden die Worte innerhalb der vermuteten Named Entity bei der Erkennung gleichermaßen berücksichtigt wie ihr Kontext [Lan-2006, Seite 5]. Evidenz

Bei der regelbasierten NER kommen auch häufig ein *Ausschlussverfahren* zum Einsatz, um die Präzision (siehe Abschnitt 3.1.1 auf Seite 17) zusätzlich zu erhöhen. Dabei werden zunächst als Named Entity erkannte Ausschnitte mit Hilfe von Ausschlusslisten oder Kontextinformationen wieder zurückgezogen. [Lan-2006, Seite 5] Ausschluss-verfahren

Als Nachteil der regelbasierten NER ist aufzuführen, dass das manuelle Erstellen der Regeln sehr aufwändig ist und sich vorhandene Regeln nur schwer auf andere Sprachen oder andere Domänen übertragen lassen [Lan-2006, Seite 4]. Nachteil

3.3. Apache UIMA

Architektur

Die Unstructured Information Management Architecture (UIMA)[9] der Apache Foundation ist eine offene, skalierbare und erweiterbare Plattform zur Integration und zum Betrieb von Lösungen für die Verarbeitung unstrukturierter Informationen [Apa-2011a, Seite 17]. Sie liefert ein plattformübergreifendes Software-Framework zur Erzeugung, Zusammenstellung und Inbetriebnahme verschiedenster Analysekomponenten [Apa-2011a, Seite 1]. Zudem erhebt UIMA den Anspruch auf *Industrietauglichkeit* [Apa-2011a, Seite 17] und ist seit März 2009 der bis heute einzige OASIS Standard in der Disziplin der Inhaltsanalyse [Wik-2012].

OASIS

Ursprünge

Historie

Als ehemaliges IBM Projekt wurde UIMA ursprünglich mit dem Ziel entwickelt, die Entwicklungen der verschiedenen NLP-Abteilungen der Firma durch standardisierte Schnittstellen schneller finden und integrieren zu können und um Forschungsergebnisse wiederzuverwenden.[FL-2004, Seite 1]. Im Oktober 2006 wurde das Framework als Open Source unter der Apache Software Licence (ASL) 2.0 veröffentlicht [Wik-2012]. Zuletzt trat UIMA 2011 als Komponente von IBM Watson hervor, dem ersten Computersystem, das in der Quizshow Jeopardy gegen zwei menschliche Rekordhalter gewann [Ric-2012, Seite 13].

C++ / Java

Referenzimplementierungen von UIMA existieren in den Programmiersprachen C++ und Java [Ric-2012, Seite 12], zudem finden sich Implementierungen in Perl, Python und TCL [Her-2009a, Seite 7].

Die Gesamtarchitektur weist folgende Eigenschaften auf:

1. UIMA versteht sich als Brückentechnologie, die unstrukturierte Informationen in strukturierte überführt [Apa-2011a, Seite 17]

2. UIMA unterstützt das Zusammenschalten von Komponeten und deren Datenaustausch „mittels gemeinsamen, kompatiblen Formaten“ [Her-2009a, Seite 11]

[9] `http://uima.apache.org/` (letzter Abruf am 2012-05-29).

3. UIMA schaltet Komponenten verschiedener Art und sogar Programmiersprachen hintereinander, setzt sie zu mit Extensible Markup Language (XML) beschriebenen Verarbeitsungspipelines zusammen und stellt Wiederverwendbarkeit sicher. [Her-2009a, Seite 7 und 11]

Abbildung 3.1 zeigt die schematische Komponentenarchitektur von UIMA einschließlich einer Einordnung der beiden wichtigsten Konzepte:

- der Verarbeitungskomponente(n) **(Aggregate) Analysis Engine (AE)**
- dem Austauschformat **Common Analysis Structure (CAS)**

Zusammenspiel

Es wird deutlich, dass unstrukturierte Informationen (links im Bild, beispielsweise Text oder Video) als Eingabe für den Collection Reader dienen. Dieser erstellt die CAS und übergibt sie zur Analyse den (Aggregate) AEs und zur Ausgabe als strukturierte Information zum Beispiel in Form einer Ontologie oder Datenbank den CAS Consumern (rechts im Bild).

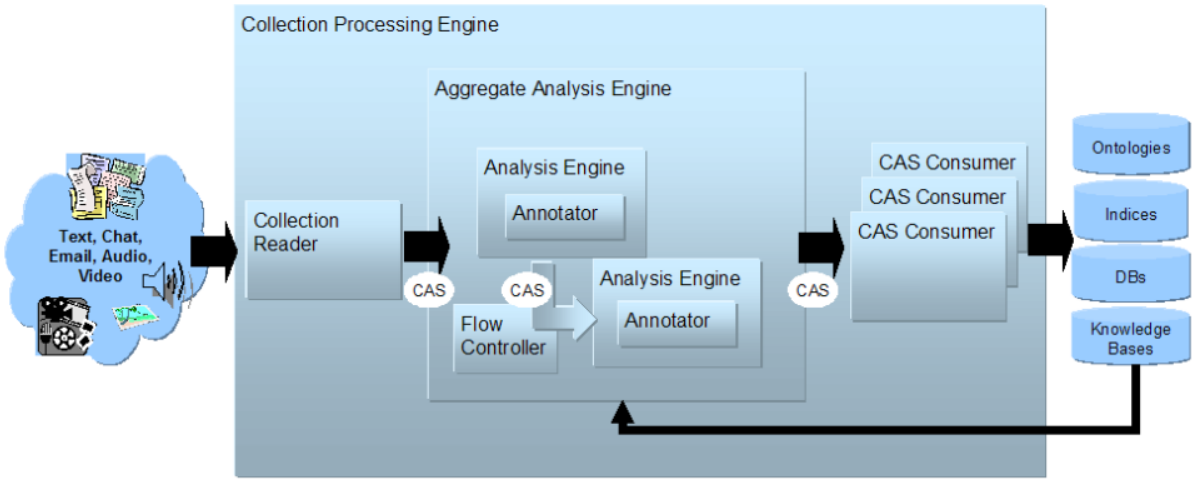

Abbildung 3.1.: Schematische Architektur der Komponenten von UIMA, Quelle: [Apa-2011a, Seite 26]. Zentrale Bestandteile sind das Austauschformat CAS und (Aggregate) Analysis Engines, die im Folgenden weiter erläutert werden.

3.3.1. Analysis Engines und Annotatoren

Analysis Engine

Zentrale Bausteine einer UIMA sind Analysis Engines (AEs), die zur Analyse eines Artifakts [Apa-2011c, Seite 1], etwa eines Dokuments, zusammengestellt werden und die Attribute zu den Inhalten des gesamten Dokuments oder zu Teilen davon aufzeichnen [Apa-2011a, Seite 19]. Eine Analysis Engine besteht im wesentlichen aus einem *Annotator*, der die gesamte Analyselogik der Analysis Engine beinhaltet; und nichts anderes: Ein Designziel der UIMA ist es, dass Annotatoren unabhängig von Details ihrer Ausführung oder der Interaktion mit anderen Annotatoren bleiben [Apa-2011c, Seite 1].

Zwei Arten

Grundsätzlich lassen sich AEs anhand ihrer Annotatoren in zwei Typen unterteilen:

(Primitive) Analysis Engines enthalten genau einen Annotator und stellen so das einfachste Einsatzszenario einer Analysis Engine dar [Apa-2011c, Seite 1].

Aggregate Analysis Engine enthalten mehrere Analysis Engines, als Workflow organisiert [Apa-2011a, 24], und beinhalten daher mehrere Annotatoren [Apa-2011c, Seite 1]. Auf diese Weise kann eine Abhängigkeit oder Aufrufreihenfolge für einzelne AEs bestimmt werden.

Zusammenspiel

Abbildung 3.2 zeigt das Zusammenspiel und die Beziehung von Primitive und Aggregate Analysis Engines anhand eines Beispiels.

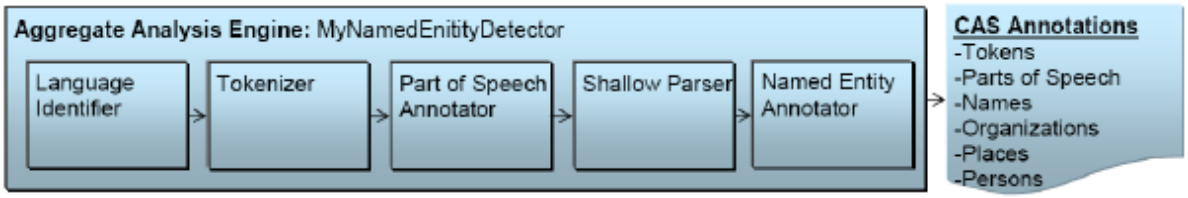

Abbildung 3.2.: Beispiel einer Aggregate Analysis Engine, bestehend aus einer Reihe Primitive AEs (zum Beispiel Langauge Identifier), Quelle: [Apa-2011a, Seite 24].

3.3.2. Common Analysis Structure

Analyseergebnisse einer oder mehrere AEs werden in der Common Analysis Structure (CAS) repräsentiert [Her-2009a, Seite 17] - sie hält die Daten vor, die von Annotatoren (siehe Abschnitt 3.3.1 auf der vorherigen Seite) erzeugt und manipuliert werden [Apa-2011b, Seite 53]. CAS

Eine CAS setzt sich aus den folgenden Bestandteilen zusammen:

Subject of Analysis (Sofa) , den Untersuchungsgegenstand (Artefakt), auf den sich die in der CAS gespeicherten Daten beziehen [Apa-2011a, Seite 38], etwa eine Zeichenkette oder eine Audiodatei.

Typsystem , einem Schema für Analyseergebnisse [Apa-2011a, Seite 38], das spezifiziert, welche Daten genau in den AEs erzeugt und manipuliert werden können [Her-2009a, Seite 18].

Feature Structures und Metadaten , die Nutzdaten (Analyseergebnisse) der CAS gemäß des oben genannten Typsystems [Apa-2011a, 38].

Index als Verzeichnis für alle Analyseergebnisse innerhalb einer CAS [Apa-2011a, Seite 47].

Das UIMA Framework liefert in Java zwei verschiedene Interfaces der CAS aus [Apa-2011c, Seite 68]: als natives Objekt (`org.apache.uima.cas.CAS`) und als JCas (`org.apache.uima.jcas.JCas`), das den Zugriff über Java-nahe Methoden[10] bei der Nutzung der enthaltenen Daten bietet. Die jeweiligen Schnittstellen sind über die Methode `getCAS()` bzw. `getJCas()` gegeneinander austauschbar [Apa-2011b, Seite 67]. Interface

Zudem hält UIMA Methoden zur Serialisierung und Deserialisierung eines CAS in den XML-basierten Standard XML Metadata Interchange (XMI) vor [Apa-2011a, Seite 38]. XML-I/O

[10] Vorteile des Convenience-Interfaces JCas ist etwa das Vorhandensein von Getter-, Setter- und anderen Hilfs-Methoden, einer Application Programming Interface (API)-Dokumentation mittels Javadoc und Construktoren für häufige Konstruktionsszenarien [Apa-2011b, Seite 67].

3.3.2.1. Typsystem und Feature Structures

Das *Typsystem* beschreibt in UIMA die genaue Art eines Analyseergebnisses. Es wird von jeder Komponente mit Zugriff auf eine CAS genutzt und sorgt dafür, dass beispielsweise die Ausgabe einer Analysis Engine als Eingabe einer anderen genutzt werden kann [Apa-2011a, Seite 48]. Dazu definiert es zwei schematische Angaben: Schema-Angaben

Types sind Spezifikationen eines zur Speicherung von Analyseergebnissen genutzten Objekts in der CAS [Apa-2011a, Seite 48]. Sie sind in einer (einfachen) Vererbungshierarchie organisiert und definieren die genaue Art und Menge an Entitäten, die im CAS manipuliert werden können [Apa-2011b, Seite 53]. Types

Sie sind ein Analogon zu Klassen in Java [Her-2009a, Seite 18], mit dem Hauptunterschied, dass Types keine Methoden besitzen können [Apa-2011b, Seite 53].

Features spezifizieren optionale Felder innerhalb eines bestimmten Types und können, neben primitiven Datentypen wie Integer- und Float-Zahlen oder Strings auch Referenzen auf andere Objekte in der CAS beinhalten [Apa-2011b, Seite 53]. Features

Sie können wie Attribute einer Klasse in Java verstanden werden [Her-2009a, Seite 18]

Wie in der objektorientierten Programmierung werden Features von Types an entsprechende Unterklassen vererbt; der Wurzel-Type ist `uima.cas.TOP` bzw. die Java-Klasse `org.apache.uima.jcas.cas.TOP`, und verfügt über keine Features. Hierarchie

Konkrete Analyseergebnisse sind in instantziierten Datenstrukturen des Typsystems, den *Feature Structures*[11], innerhalb des CAS gespeichert [Apa-2011b, Seite 53]. Diese stellen in Java-Analogie auch Instanzen von Klassen dar [Her-2009a, Seite 18]. Feature Structure

Ein häufig anzutreffendes Analyseergebnis [Apa-2011a, Seite 38] ist eine *Annotation*, das gleichzeitig auch zentrales Paradigma der UIMA verdeutlicht: Mit Annotations können

[11] Begriffe wie Typsystem, Type und Feature sind aus der Computerlinguistik entlehnt [Apa-2011a, Seite 48]; Feature Structures in UIMA sind aber wie beschrieben keineswegs als „Strukturen von Features“ zu verstehen, auch wenn die Ähnlichkeit der Begriffe dies suggeriert.

Analysis Engines Metadaten an Passagen eines Artifakts knüpfen, zum Beispiel ein Label für bestimmte Worte einer Zeichenkette [FL-2004, Seite 6].

Auch das PoS -Tagging (siehe Abschnitt 3.2.2 auf Seite 20) erfolgt mit Annotations. Dazu definiert das Typsystem Features für die absolute Start und Endposition der betroffenen Passage des Artifakts [Her-2009a, Seite 21], sowie Möglichkeiten zur Benennung. Eine Annotation-Feature Structure beinhaltet also lediglich Referenzen auf Teile des Sofas, und ist ohne eben diese Sofa wertlos.

Annotation Features

Die Unversehrtheit des Analysegegenstands (Sofa) und die separate Pflege von Analyseergebnissen (Annotations) wird als stand-off bezeichnet [FL-2004, Seite 6] und stellt einen zentralen Grundsatz im Umgang mit CAS dar. Dies ist flexibler als das „inline markup“[12] [FL-2004, Seite 6] und stellt zudem sicher, dass jede Analysis Engine den gleichen Analysegegenstand untersucht, Analysegegenstände von vorgelagerten AEs aber dennoch genutzt werden können.

stand-off Annotations

3.3.2.2. Views

Mit Hilfe von *Views* können CAS mehrere Versionen eines analysierten Artifakts und jeweils eigene Indices (siehe Abschnitt 3.3.2 auf Seite 29) für Analyseergebnisse enthalten [Apa-2011a, Seite 38]. Es existiert eine eins-zu-eins Korrespondenz zwischen View und Sofa [Her-2009a, Seite 30]; jede Sofa definiert einen eigenen View und Feature Structures können Mitglied von einem oder mehreren Views sein [Apa-2011b, Seite 93]. Jede Repräsentation des Artifakts kann dabei mit dem UIMA Programmiermodell unabhängig analysiert werden [Apa-2011c, Seite 107].

View

Sofa, View und Feature Structure

Views sind nützlich, etwa um in verschiedenen Bereichen der Analyse verschiedene Versionen des Artifakts zu verwenden, in der *multimodalen Analyse*, wenn der initiale Artifakt von einer Modalität zu anderen transformiert wird, oder wenn der Artifakt selbst multimodal ist, wenn beispielsweise die CAS eines Videofilms Artifakte für Audio, Video und Untertitel enthalten soll [Apa-2011c, Seite 107]. Abbildung 3.3 auf der nächsten Seite verdeutlicht dies am Beispiel einer Audio-Datei.

Szenarien

[12] Beim „inline markup“ wird der Analysegegenstand direkt um die Analyseergebnisse ergänzt.

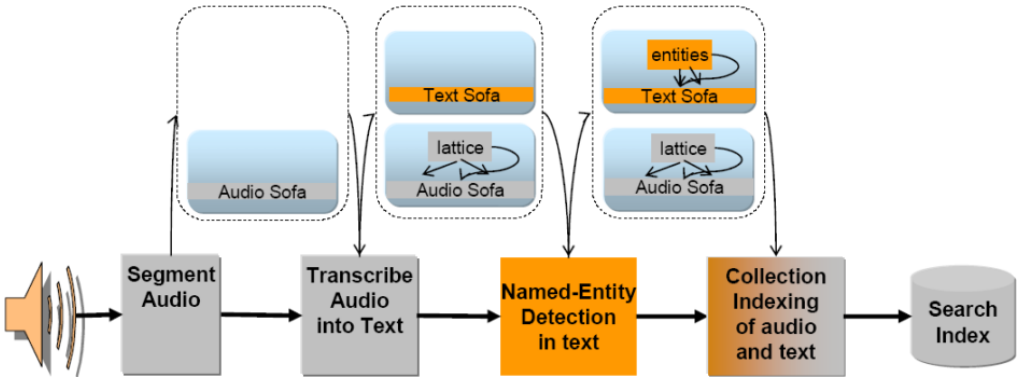

Abbildung 3.3.: Multimodale Verarbeitung einer Audio-Datei: mehrere Versionen eines Sofas (hier: Audio und Text) können innerhalb einer CAS gepeichert und abhängig von den Bedürfnissen und Anforderungen einzelner AEs genutzt werden. Quelle: [Apa-2011a, Seite 29]

Ein neu-erstelltes CAS enthält automatisch den *Standard View* `_InitialView` [Apa-2011c, Seite 109], an den auch ohne weiteres Zutun Feature Structures bei ihrer Erstellung standardmäßig verknüpft werden.

Neben dieser sehr unscheinbaren Nutzung von Views stellt UIMA auch Funktionalität zum Beeinflussen dieses Standardverhaltens zur Verfügung: durch die API können gezielt Views erzeugt und verknüpfte Sofas manipuliert werden, sowie Views (und Sofas) durch ihren Namen gezielt angesprochen und bestimmte Feature Structures explizit an den Index eines bestimmten Views gehängt werden [Apa-2011c, Seite 108]. Nutzung

3.3.3. uimaFIT

Motivation

UIMA macht intensiven Gebrauch von XML: Nicht nur bei der (De-)Serialisierung einer CAS, auch bei der Beschreibung von Analysekomponenten, Typsystem und der Verarbeitungspipeline sowie der Konfiguration muss XML zwingend verwendet werden.

Java-XML-Kopplung

Um ein CAS in UIMA erstellen und verarbeiten zu können, sind selbst in einfachen Szenarien etwa 5-10 Zeilen Java-Code und 30-50 Zeilen XML-Code erforderlich. Zudem sind beide Codebestandteile fest aneinander gekoppelt, Änderungen im XML ziehen Änderungen im Java-Code nach sich, und umgekehrt. Die Wartung erzeugter Komponenten ist schwierig, nicht zuletzt deshalb, weil viele Tools am automatischen Refactoring beider Codes scheitern. [OB-2009, Seite 2]

UIMA-Aufsatz

Diesen Umstand adressieren Philip Ogren und Steven Bethard [OB-2009] in ihrem UIMA-Aufsatz UIMA Factories, Injection and Testing Utilities (uimaFIT)[13]. uimaFIT stellt eine Reihe von Convenience-Klassen[14] bereit, um UIMA-Komponenten ohne den Mehraufwand durch die umfassende Initialisierung in der UIMA-Pipeline und ohne zwingende Beschreibung durch XML instanziieren, ausführen und testen zu können [OB-2009, Seite 2].

„Zwei Seiten“

Die meisten Funktionen von uimaFIT dienen entweder einer im Vergleich zu UIMA vereinfachten Implementierung von Komponenten oder einer vereinfachten Instanziierung von Komponenten; ein Entwickler kann sich frei entscheiden, beide dieser „*zwei Seiten*“ zu nutzen oder nur die jeweils für seinen Use-case passendste [Ogr-2012, Seite „TwoSidesOfUimaFIT“].

Kompatibilität

Dies ist auch möglich, weil uimaFIT die Mechanismen von UIMA zur Beschreibung von Komponenten nicht umgeht, sondern lediglich diese für den Benutzer übernimmt. uimaFIT-Factorymethoden erzeugen die gleiche Laufzeitumgebung wie die Komponenten von UIMA; die Ansätze von UIMA und uimaFIT sind grundsätzlich kompatibel.[Ogr-2012, Seite „TwoSidesOfUimaFIT“].

[13] Der ursprüngliche Name von uimaFIT lautete UIMA Unit Test Utility Code (UUTUC) und macht die Wurzeln des Projekts im Testwesen deutlich.

[14] Der Mehrwert von Convenience-Klassen stellt nicht notwendigerweise das Bereitstellen neuartiger, sondern vor allem eine vereinfachte Nutzung bestehender Funktionalität dar.

3.3.3.1. Implementierung von Komponenten

Einfachere Programmierung

Einen Teil der durch uimaFIT bereitgestellten Funktionalitäten vereinfacht die Implementierung von UIMA-Komponenten, sodass Aufwände bei der Entwicklung von umfassendem, potentiell fehleranfälligen Programmcodes gespart werden können [Ogr-2012, Seite „TwoSidesOfUimaFIT“].

Annotations

Ein Beispiel für eine derartige Verschlankung der Implementierung stellen zum Beispiel Konfigurationsparameter in uimaFIT dar, die als (Java-)Annotations direkt an Klassenattribute einer Analysis Engine -Implementierung annotiert werden können und bei der Nutzung automatisch vom Framework initialisiert werden [Ogr-2012, Seite „TwoSidesOfUimaFIT“]. Das UIMA-Äquivalent zu Konfigurationsparametern setzt auf eine genaue Spezifikation jedes einzelnen Parameters in XML-Deskriptoren und eine mehrzeilige Abfragelogik in Java [Apa-2011c, Seite 14f].

3.3.3.2. Instanziierung von Komponenten

Vereinfachtes Aufrufen

Ein anderer Teil der durch uimaFIT bereitgestellten Funktionalitäten vereinfacht die Instanziierung und reduziert daher den Programmcode, der abgesehen von der Implementierung einer Analysis Engine (siehe Abschnitt 3.3.1 auf Seite 28) notwendig ist, um eine UIMA-Pipeline zu betreiben [Ogr-2012, Seite „TwoSidesOfUimaFIT“].

UIMA vs. uimaFIT

In einer reinen UIMA-Umgebung sind bis zum tatsächlichen Betrieb einer Analysis Engine einige vorausgehende Schritte erforderlich, zum Beispiel das Schreiben eines CAS Consumers, der die gewünschte Ausgabe der Analyseergebnisse produziert, oder dem Beschreiben beteiligter Komponenten in verschiedenen XML-Deskriptoren. In einer uimaFIT-Pipeline werden diese Schritte auf wenige Zeilen reinen Java-Code reduziert. [Ogr-2012, Seite „TwoSidesOfUimaFIT“].

3.3.4. DKPro

Repository

Das Darmstadt Knowledge Processing Repository (DKPro) der Ubiquitous Knowledge Processing (UKP) Group der Technischen Universität Darmstadt[15] wurde mit der Vision entwickelt, verschieden Technologien, darunter auch die Disziplinen des Information Retrieval (siehe Abschnitt 3.1 auf Seite 15), zu verwenden um innovative Anwendungen zu erschaffen [GMM-2007, Seite 1]. Das UIMA-basierte Projekt[16] [GMM-2007, Seite 1] versteht sich DKPro als „Software Repository“ mit aufeinander abgestimmten „Plug and Play Komponenten“, mit denen sich „komplexe Anwendungen leicht“ realisieren lassen [Gur-2009, Seite 7].

Integration

Die Philosophie hinter DKPro, bestehende Komponenten zu integrieren [Gur-2009, Seite 8], hat bereits zur Integration verschiedenster, allgemein nützlicher Verarbeitungskomponenten geführt [GMM-2007, Seite 2], etwa unterschiedliche Arten Tagger (wie Dateipfad-, PoS - und Code-Tagger), Splittern (Satz- und Compound-Splitter) oder Datenimporten (aus Wikipedia oder Foren) [GMM-2007, Seite 3].

Lizenzen

Die Kernanwendung existiert für die zwei Open Source-Lizenzen ASL[17] und GNU General Public License (GPL)[18].

[15] http://www.ukp.tu-darmstadt.de/ (letzter Abruf am 2012-08-06).
[16] DKPro wurde mit einem IBM UIMA Innovation Award 2007 und zwei IBM Unstructured Information Analytics Awards 2008 ausgezeichnet [Gur-2009, Seite 2].
[17] http://code.google.com/p/dkpro-core-asl/ (letzter Abruf am 2012-08-06).
[18] http://code.google.com/p/dkpro-core-gpl/ (letzter Abruf am 2012-08-06).

3.4. JBoss Drools

JBoss Drools von Red Hat[19] ist eine in Java implementierte *Rule Engine*. Als solche bezeichnet man allgemein jedes System, dass Regeln in beliebiger Form einsetzt und auf Daten anwendet, um eine Ausgabe zu produzieren [The-2012, Seite 3].

Wissen

Eine Rule Engine stellt eine technische Antwort auf die Frage dar, wie genau Wissen in symbolischer Form im System repräsentiert wird und wie dieses Wissen in einem Verarbeitungs- oder Denkprozess verarbeitet und angewendet wird [The-2012, Seite 2]. Für Anwendungsentwickler kann eine Rule Engine so zum Entwicklungsparadigma der *Separation of concerns* betragen, da durch sie Anwendungs- und Entscheidungslogik sauber voneinander getrennt werden können. Sie ist auch bei der Entwicklung eines *Expertensystems*[20] maßgeblich beteiligt.

Separation of concerns

Der Einsatz einer Rule Engine in Softwareprojekten hat verschiedene Vorteile [The-2012, Seite 13f]:

Trennung von Logik und Daten Die Logik eines Programms wird von seinen Daten (Objekten) im Programmcode entkoppelt. Dadurch ist die Logik bei Änderungen leichter zu warten. Das gilt insbesondere für Lösungen, die Logiken mehrerer verschiedener Domänen beinhalten.

Zentralisierung von Wissen Es entsteht ein zentrales Verzeichnis, das sämtliches erforderliches Wissen an einem einzigen Ort konzentriert. Dieser bildet damit einen *Single Point of Truth*.

Transparente Entscheidungen Durch das Führen von Protokollen macht es eine Rule Engine transparent und leicht nachzuvollziehen, welche Entscheidungen getroffen wurden und was genau jeweils dazu geführt hat.

[19] `http://de.redhat.com/` (letzter Abruf am 2012-08-26).

[20] Ein Expertensystem unternimmt den Versuch, das Wissen eines menschlichen Experten möglichst umfassend nach- oder abzubilden und dieses für die Entscheidung von domänenspezifischen Problemen zu nutzen. Drools wird dabei häufig irrtümlich als Expertensystem bezeichnet, tatsächlich stellt es jedoch eher das Framework für ein solches dar [The-2012, Seite 8].

Deklaratives Programmieren Lösungen für komplexe Probleme können einfacher ausgedrückt werden, da mit einer Rule Engine direkt formuliert werden kann, *was* die Lösung ist - und nicht *wie* sich genau der Lösungsweg gestaltet. Diese deklarative[21] Schreibweise ist erheblich leichter zu lesen als Programmcode.

Seit seinen Anfängen 2001 ist Drools ein spezieller Typ Rule Engine, ein Production Rule System (PRS) [The-2012, Seite 3]. Dieses sucht passende *Fakten* (*Facts*) zu bestimmten *Regeln* (*Rules*, auch *Production Rules* oder *Produktions* bezeichnet) und folgert Rückschlüsse daraus.

Facts

Rule

Regeln in Drools bestehen, wie im Listing 3.4 dargestellt, aus einem Namen und aus jeweils zwei Teilen [The-2012, Seite 150]: die so genannte Left Hand Side (LHS) definiert ein oder mehrere *Bedingungen* (*Conditions*), die zum Auslösen der jeweiligen Regel führen, und folgt dem Schlüsselwort `when`. Die so genannte Right Hand Side (RHS) definiert dagegen die auszuführenden *Aktionen* (*Actions*), die beim Auslösen der jeweiligen Regel ausgeführt werden sollen.

Action

Listing 3.4: Syntax einer Drools-Regel, Quelle: (The JBoss Drools Team, 2012, Seite 150)

```
1  rule "<name>"
2  when
3    <conditional element>*
4  then
5    <action>*
6  end
```

Dieser prinzipielle Ansatz von Drools bei der Arbeit als Rule Engine ist *datengetrieben* bzw. reaktionär - das heißt Fakten führen zur Ausführung von Regeln und zu daraus resultierenden Schlussfolgerungen - und wird als *forward chaining* bezeichnet [The-2012, Seite 5].

forward chaining

Der dazu konträre Ansatz, *backward chaining*, ist *zielgetrieben* und wird zum Beispiel von Prolog[22] verfolgt: Er beginnt mit einer Schlussfolgerung, die das System dann in mehreren Schritten versucht zu beweisen [The-2012, Seite 6].

[21] Siehe auch: Wikipedia-Artikel zum deklarativen Programmieren, `http://de.wikipedia.org/wiki/Deklarative_Programmierung` (letzter Abruf am 2012-08-27).

[22] `http://de.wikipedia.org/wiki/Prolog_(Programmiersprache)` (letzter Abruf am 2012-08-27).

Production & Working Memory

Zur Laufzeit eines PRS befinden sich alle beteiligten Regeln im *Production Memory*; alle beteiligten Fakten dagegen im *Working Memory*, in dem sie auch modifiziert oder zurückgezogen werden können [The-2012, Seite 4]. Im Prozess des *Pattern Matching*s führt eine *Inference Engine* jeweils Regel und passende(n) Fakt(en) zusammen. Da es vor allem in Systemen, die aus vielen Regeln bestehen, vorkommen kann, dass Fakten die Bedingungen verschiedener Regeln gleichzeitig erfüllen, sich diese Regeln also in der Terminologie von Drools im *Konflikt* befinden, ist eine weitere Komponente bei der Ausführung von Regeln in Drools beteiligt [The-2012, Seite 4]: die *Agenda* übernimmt die Planung der Ausführungsreihenfolge von Regeln und löst derartige Konflikte. Abbildung 3.4 verdeutlicht dieses Zusammenspiel der Komponenten.

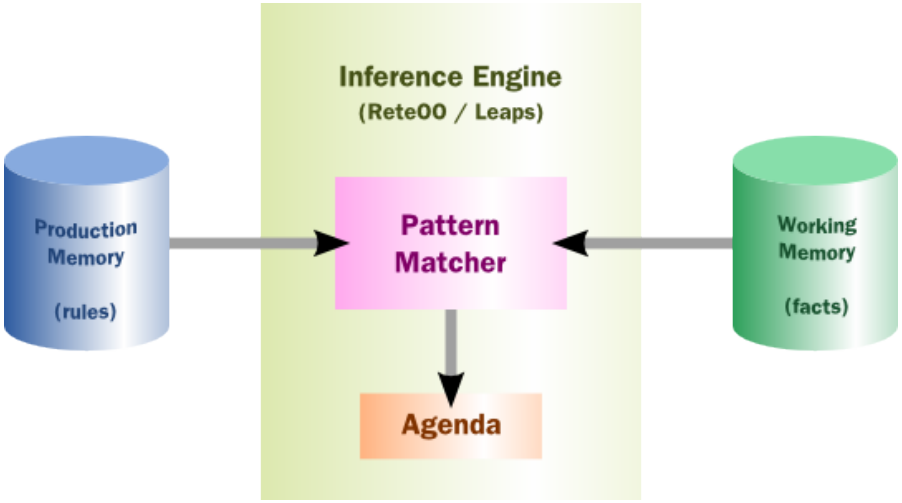

Abbildung 3.4.: High-Level-Ansicht eines Production Rule System mit den wichtigsten seiner Komponenten, Quelle: [The-2012, Seite 5].

Um Daten mit Drools verarbeiten zu können, müssen sie zunächst als Fakten in den Working Memory eingefügt werden [The-2012, Seite 24]. Dabei kann Drools auf zwei verschiedene Arten betrieben werden: in einer *stateless session* haben Fakten im Working Memory keinen Zustand und werden bei Änderungen in nachfolgenden Schlussfolgerungen nicht weiter berücksichtigt [The-2012, Seite 28]. In einer *stateful session* dagegen können Fakten modifiziert oder auch zurückgezogen werden, was auch bei Schlussfolgerungen berücksichtigt wird [The-2012, Seite 28]. Als Folge können ehemals unerfüllte Bedingungen von Regeln nachträglich erfüllt werden, und umgedreht. Dieses Verhalten verdeutlicht der Befehl `insertLogical`, der einen neuen Fakt einfügt, der automatisch wieder zurückgezogen wird, sobald die Bedingungen, die zu seinem Einfügen führten, nicht mehr länger erfüllt sind.

Kapitel 4.

Information Extraction aus natürlichsprachlichen Phrasen

In diesem Kapitel wird die technische Architektur und die Implementierung einer regelbasierten NER -Lösung für das in Abschnitt 1.2 motivierte und in Abschnitt 2 angeforderte System beschrieben. Zuvor wird die Auswahl der verwendeten Komponenten kurz begründet.

Als Grundlage für die nachfolgende Dokumentation dient Philippe Kruchtens 4+1 Sichtenmodell für die Beschreibung von Software Architekturen [Kru-1995]. Es umfasst vier Sichten - die *logische Sicht*, die *Prozess-Sicht*, die *Entwicklungs-Sicht* und die *physikalische Sicht* - die in ihrer Gesamtheit durch *Use Cases* (das namensgebende „+1“) motiviert sind [Kru-1995, Seite 1-2]. Für die zu entwickelnde TourInDi-Komponente Query Analysis (siehe Abschnitt 1.1 auf Seite 2) gestaltet sich ein UML Use Case Diagramm (siehe Abbildung 4.1) vergleichsweise einfach: Der Akteur TourInDi-Service (siehe Abschnitt 1.1 auf Seite 2) verwendet Query Analysis, um natürlichsprachliche Hotelsuchanfrage computerlesbar zu strukturieren.

Abbildung 4.1.: UML Use Case Diagramm von Query Analysis.

4.1. Wahl der Komponenten

Im Folgenden wird kurz dargelegt, welche Gründe für den Einsatz der verwendeten Komponenten in der Lösung sprechen. Dabei liegt der Fokus auf Wartbarkeit und Praxistauglichkeit für das anvisierte Anwendungsszenario (siehe Abschnitt 1.2 auf Seite 5) als Lösung für eine regelbasierte NER .

4.1.1. DKPro als Verarbeitungspipeline

Aktuelle Entwicklung

Vieles spricht für den Einsatz von UIMA. Der OASIS Standard ist weit verbreitet und wird aktiv und konstant entwickelt[1]. Aktuelle Einträge vom 23. August 2012 auf der uimaFIT User Mailing List deuten zudem an, dass es Pläne zur Integration von uimaFIT in das Apache Projekt UIMA gibt[2]. Dies spricht für die Reife von uimaFIT

Auch wenn DKPro als uimaFIT-Aufsatz (siehe Abschnitt 3.3.3 auf Seite 33) einige Vorteile, die zum Teil uimaFIT geschuldet sind, bietet, so musste der Autor auch einige Schwierigkeiten bei der Nutzung von DKPro feststellen.

Schwierigkeiten

DKPro versteht sich als „Software Repository“ (siehe Abschnitt 3.3.4 auf Seite 35), dieser Begriff ist jedoch sehr technik- und entwicklernah ausgelegt; die „Plugin & Play“-Fähigkeit reduziert sich auf eine konsequente Verwendung des Buildmanagementsystems Maven[3]. Ein Einstieg wird durch unvollständige, veraltete oder nicht vorhandene Dokumentation weiter erschwert.

Wertvoll durch Integration

Sind diese Unzulänglichkeiten größtenteils beim Einstieg überwunden, zeigt sich DKPro jedoch als einen sinnvollen und hilfreichen UIMA-Aufsatz und erweist sich insbesondere durch seine Integration vieler verschiedener Drittanbieterkomponenten als wertvoll. Dies ist bespielsweise bei den Stanford-NLP-Komponenten[4] der Fall - DKPro bietet hierbei

[1] Der Projektverlauf von UIMA unter `http://www.ohloh.net/p/7917` (letzter Abruf am 2012-10-04) zeigt seit 2006 eine kontinuierliche Entwicklung.

[2] `https://groups.google.com/forum/?fromgroups=#!topic/uimafit-users/9uh3U_HMgqc` (letzter Abruf am 2012-08-25).

[3] `http://maven.apache.org/` (letzter Abruf am 2012-08-19).

[4] `http://nlp.stanford.edu/` (letzter Abruf am 2012-08-21).

integrierte und mit aktuellen Modellen lauffähige Software an, während die offiziellen UIMA-Komponenten der Universität bereits zu Beginn des Projekts eingestellt wurde[5]. Zusammen mit der Tatsache, dass DKPro intensiv von uimaFIT Gebrauch macht und so dessen Vorteile durch seine „zwei Seiten“ (siehe Abschnitt 3.3.3 auf Seite 33) bei der Benutzung ebenfalls mit einschließt, gestaltet sich DKPro als eine geeignete und sehr brauchbare Arbeitsgrundlage für UIMA-Projekte.

Bewertung

4.1.2. NER mit der Regelsprache Drools

Der Autor schließt sich der Argumentation von Douglas Appelt (siehe Abschnitt 3.1.2 auf Seite 19) an und wählt für das gegebene Szenario gemäß seiner Argumentation den Knowledge Engineering Approach (siehe Abschnitt 3.1.2 auf Seite 19), also den Ansatz für IE -Systeme, der das domain- und systemspezifische Wissen eines Menschen durch ein Regelsystem in die Anwendung integriert. Dies bringt die beschriebenen Vorteile beim Einsatz einer Rule Engine (siehe Abschnitt 3.4 auf Seite 36), wie das deskriptive Programmieren von Regeln, zentrales Management von domänenspezifischem Expertenwissen sowie eine leichtere Pflege mit sich.

Knowledge Engineering Approach

Zudem wird mit der regelbasierten NER (siehe Abschnitt 3.2.4.2 auf Seite 24) ein Verfahren beschrieben, dass sich ideal mit Drools implementieren lässt: die für die NER beschriebene Regelsprache (siehe Abschnitt 3.2.4.2 auf Seite 24) lässt sich gar mit fast identischer Syntax in Drools implementieren. Für den konkreten Einsatz der Regelsprache Drools spricht außerdem die Tatsache, dass Drools bereits in der TourInDi-Komponente Recommendation (siehe Abschnitt 1.1 auf Seite 2) genutzt wird und deshalb bereits Kenntnisse zur Regelerstellung im Projektteam vorhanden ist. Dies wird auch von Douglas Appelt als wichtiges Kriterium für den regelbasierten Ansatz aufgeführt (siehe Abschnitt 3.1.2 auf Seite 19).

ideale Implementierung

Know-How

Auch gibt es auf der UIMA Users Mailing List Erfahrungsberichte über den erfolgreichen Einsatz von Drools innerhalb von UIMA[6].

[5] https://mailman.stanford.edu/pipermail/parser-user/2011-April/000985.html (letzter Abruf am 2012-08-19).

[6] http://mail-archives.apache.org/mod_mbox/uima-user/201206.mbox/%3CCAMpo0whDvswqtCNy2TN9Zs8TxH_tKDY6UOzBkiKqbU9mk3BCnQ@mail.gmail.com%3E (letzter Abruf am 2012-08-19).

4.2. Architektur

Die Kernfunktionalität von Query Analysis (siehe Abschnitt 1.1 auf Seite 2) wird durch eine UIMA-Pipeline (siehe Abschnitt 3.3 auf Seite 26) bereit gestellt, die aus DKPro-Komponenten (siehe Abschnitt 3.3.4 auf Seite 35) besteht. Zudem steht eine Schnittstelle zur Verfügung, deren Eingabe natürlichsprachliche Hotelsuchanfragen in englischer Sprache sind und deren Ausgabe ein `AccommodationRequest`-Objekt ist, welches die Suchanfrage im TourInDi-Datenmodell (siehe Abschnitt 1.1.1 auf Seite 3) repräsentiert. Kernfunktionalität

Abbildung 4.2 zeigt, wie diese Pipeline (im Bild genannt `DKPro / UIMA Pipeline`) logisch in die TourInDi-Komponente integriert ist und welche weiteren Abhängigkeiten sich innerhalb von Query Analysis ergeben. Integration

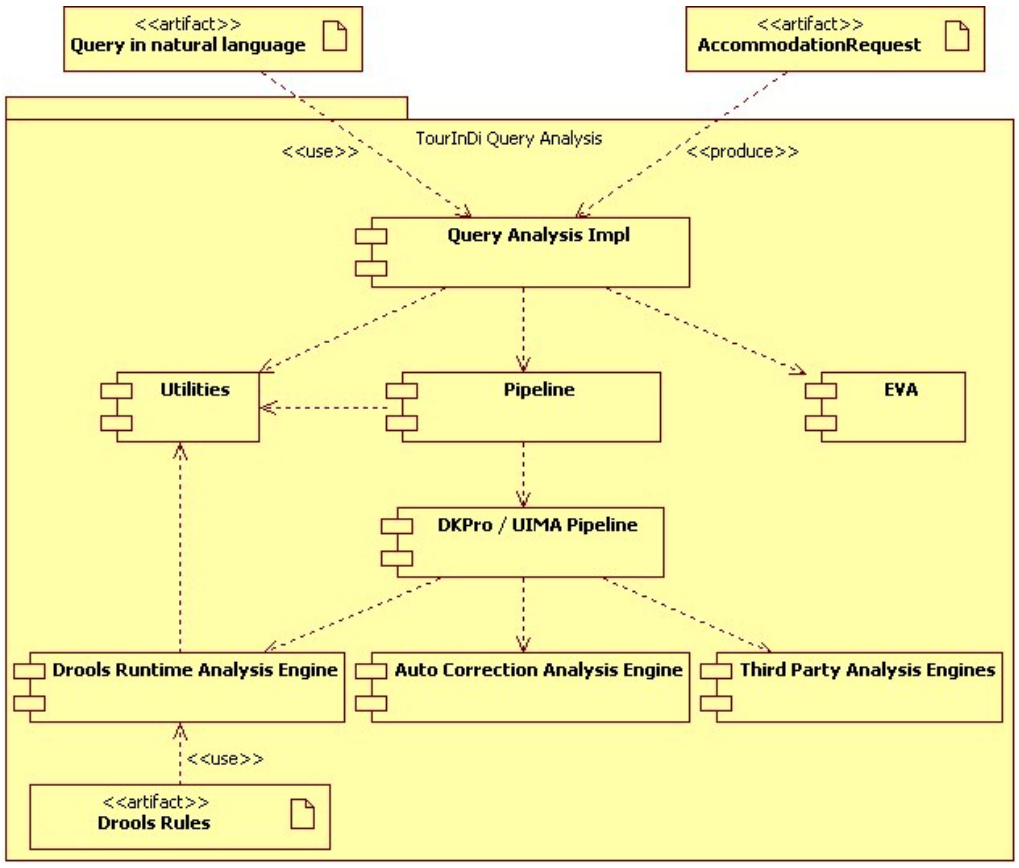

Abbildung 4.2.: UML Komponentendiagramm von Query Analysis mit integrierter DKPro / UIMA-Pipeline und den Ein- und Ausgabe Artefakten Query in natural language (English) bzw. `AccommodationRequest`-Object.

Die konzeptuellen Aufgaben der jeweiligen Komponenten sind:

Query Analysis Impl Implementiert eine mit anderen TourInDi-Komponenten verhandelte Schnittstelle, übernimmt eine natürlichsprachliche Hotelsuche als Eingabe und liefert ein entsprechend strukturiertes `AccommodationRequest`-Objekt zurück.

EVA Legacy-Implementierung der Query-Analysis mit Hilfe des Webdienstes EVA von Evature[7], welche für eine eventuelle künftige Verwendung[8] Teil des Projekts und damit weiterhin lauffähig gehalten wird.

Utilities Utility-Klassen für komponentenspezifische Hilfsmethoden. Dabei handelt es sich auch um convienience-Methoden für den Zugriff auf Details der TourInDi-Codeliste (siehe Abschnitt 1.1.1 auf Seite 3).

Pipeline Stellt eine abstrahierte Zugriffsschicht für die DKPro- bzw. UIMA-spezifische Pipeline zur Verfügung. Diese beinhaltet weiterhin eine Reihe vom Autor entwickelte oder Drittanbieter Analysis Engines (siehe Abschnitt 3.3.1 auf Seite 28):

Third Party Analysis Engines Beinhaltet eine Reihe ausgewählter Drittanbieter Analysis Engines, etwa für das Stemming, Tagging oder Parsing.

Auto Correction Analysis Engine Vom Autor entwickelte Analysis Engine zum automatischen Korrigieren von Rechtschreibfehlern (gemäß Anforderung „Fehlertoleranz“, Abschnitt 2.1.4 auf Seite 11).

Drools Runtime Analysis Engine Vom Autor entwickelte Analysis Engine zum Bereitstellen einer Drools Laufzeitumgebung für die regelbasierten NER (siehe Abschnitt 3.2.4.2 auf Seite 24).

Drools Rules Regelsätze, die in der Drools Runtime Analysis Engine ausgeführt werden und die Constraints als Bestandteile der Gesamtausgabe `AccommodationRequest` erzeugen.

[7] `http://www.evature.com/` (letzter Abruf am 2012-07-30).

[8] Die Weiterverwendung der EVA-Implementierung wird im Ausblick (Abschnitt 6.3 auf Seite 88) näher kommentiert.

Abbildung 4.3 zeigt das logische Zusammenspiel der genannten Komponenten zur Verarbeitung der Eingabe und Erzeugung der Ausgabe in Business Process Modeling Notation (BPMN). Zusammenspiel

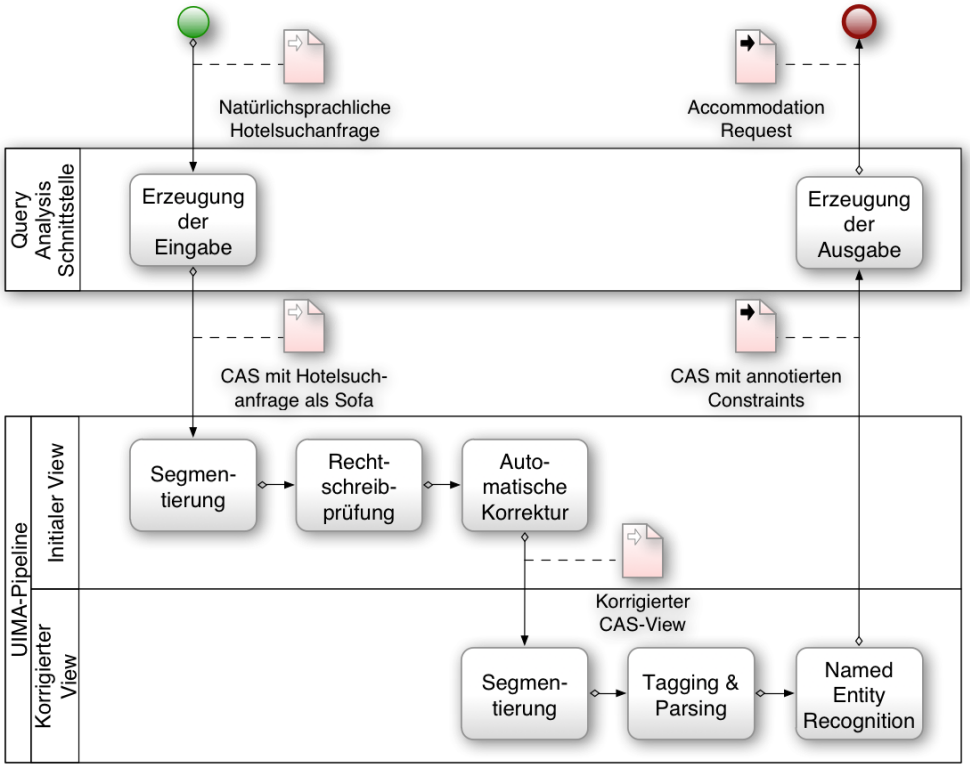

Abbildung 4.3.: Verarbeitungsprozess der Query-Analysis in BPMN.

Es wird dargestellt, dass die Eingabe der natürlichsprachlichen Hotelsuchanfrage als Sofa (siehe Abschnitt 3.3.2 auf Seite 29) einer CAS (siehe Abschnitt 3.3.2 auf Seite 29) verwendet wird, welche dann wiederum der Pipeline als Eingabe dient. Diese erzeugt während der Verarbeitung einen neuen View (siehe Abschnitt 3.3.2.2 auf Seite 31) mit einer rechtschreibkorrigierten Variante des ursprünglichen Sofas. Nach der Durchführung aller Analysen der Pipeline liegen Constraints zunächst noch als Annotations (siehe Abschnitt 3.3.2.1 auf Seite 30) in der CAS vor, bevor sie extrahiert und einem `AccommodationRequest`-Objekt zugewiesen werden. Verarbeitung

Mit diesem Vorgehen finden verschiedene Paradigmen der UIMA Beachtung und Anwendung. Die Erzeugung eines neuen Views für ein verändertes (in diesem Fall korrigiertes) Sofa ist UIMAs Affektiertheit zu stand-off Annotations (siehe Abschnitt 3.3.2.1 auf Seite 31) und der Tatsache, dass Sofas dadurch nicht verändert werden dürfen, geschuldet. Eine Paradigmen

Lösung mittels Views wird indes auch von der Community für derartige Fälle empfohlen[9]. Beteiligte Analysis Engines sind dabei „*view-unaware*“, sie operieren auf jeweils genau dem View einer CAS, der von der Pipeline als Vorgabe definiert wird[10]. Weiterhin ist auch die strenge Verwendung der CAS als Austauschformat für Analyseergebnisse von Analysis Engines auf die Grundsätze von UIMA (siehe Abschnitt 3.3.2 auf Seite 29) zurückzuführen. Diese macht einen abschließenden Verarbeitungsschritt, in Abbildung 4.3 auf der vorherigen Seite `Erzeugung der Ausgabe` genannt, erforderlich, bei dem die bereits gefundenen, annotierten Constraints zunächst wieder aus der CAS extrahiert werden müssen.

Abbildung 4.3 auf der vorherigen Seite macht zudem deutlich, aus welchen Analyseschritten sich die Pipeline zusammensetzt. In Tabelle 4.1 sind ergänzend dazu alle genannten Analyseschritte aufgeführt, zusammen mit ihrer jeweiligen Ausführungsreihenfolge (1.1 - 1.3 und 2.1 - 2.4), dem Namen des Views, auf dem sie operieren, sowie mit Details zu ihrer jeweiligen Funktion. Analyse-schritte

Tabelle 4.1.: Analyseschritte von Query Analysis.

Analyseschritt	CAS-View		Funktionsbeschreibung
	Initial	Korr.	
Tokenization	1.1	2.1	Annotiert eine Tokenization (siehe Abschnitt 3.2.1) als Grundlage nachfolgende Analyseschritte.
Rechtschreibprüfung	1.2		Prüft die Rechtschreibung auf Basis der Codeliste (siehe Abschnitt 1.1.1 auf Seite 3) und annotiert gefundene Korrekturen.
Automatische Korrektur	1.3		Korrigiert (sichere) Verbesserungen, die im vorausgehenden Analyseschritt annotiert wurden.
Tagging & Parsing		2.2	Identifiziert und annotiert PoS (siehe Abschnitt 3.2.2 auf Seite 20), generiert und annotiert einen Parsing Tree (siehe Abschnitt 3.2.3).
Named Entity Recognition		2.3	Identifiziert und annotiert Constraints durch Auswertung der in vorausgehenden Schritten annotierten PoS -Tags und Parsing Trees sowie der Codeliste.

[9] `http://mail-archives.apache.org/mod_mbox/uima-user/200910.mbox/%3C4ACCA950.8030608@neofonie.de%3E` (letzter Abruf am 2012-08-21).

[10] Siehe auch: Diskussion zur View-awareness (UIMA users mailing list) `http://mail-archives.apache.org/mod_mbox/uima-user/201208.mbox/%3CCAGRYgEOptq=-vMs_sLd4yaqJVoLZUTOKvvWqH5RSbRnF4NdtA@mail.gmail.com%3E` (letzter Abruf am 2012-08-21)

Demzufolge setzt sich die Verarbeitung durch Analyseschritte aus zwei, nacheinander laufenden Analyseprozessen zusammen, die jeweils für die gezielte Verarbeitung eines bestimmten Views zuständig sind. Zwei Views

Der erste Analyseprozess dient einer automatischen Korrektur der Benutzereingabe und operiert auf der direkten Benutzereingabe, dem Standard View einer CAS (siehe Abschnitt 3.3.2.2 auf Seite 32). Diese wird zunächst durch eine Tokenization (siehe Abschnitt 3.2.1 auf Seite 20) in einzelne Worte segmentiert (1.1), welche dann wortweise auf korrekte Rechtschreibung - als Wörterbuch dient hierbei die Codeliste (siehe Abschnitt 1.1.1 auf Seite 3) - geprüft werden (1.2). Gefundene Fehler werden als Vorschläge gespeichert und in der letzten Analysekomponente automatisch korrigiert (1.3). Wie bereits zuvor erwähnt ändert diese Korrektur das zugrundeliegende Sofa, sodass die korrigierte Variante nur in Form eines neuen Views, `Korrigierter View`, im zweiten Analyseprozess weiterverarbeitet werden kann. Tokenization, Spellcheck, Korrektur

Der zweite Analyseprozess hat zum Ziel, Constraints zu erkennen und zu annotieren und operiert auf der korrigierten Benutzereingabe, dem View `Korrigierter View`. Diese Erkennung erfolgt durch das PRS Drools (siehe Abschnitt 3.4 auf Seite 36), welches im forward chaining (siehe Abschnitt 3.4 auf Seite 37) für die regelbasierte NER auf Basis der PoS -Tags (siehe Abschnitt 3.2.2 auf Seite 20) und der Konstituenten (siehe Abschnitt 3.2.3 auf Seite 23) eingesetzt wird . Die Zusammensetzung des Analyseprozesses sowie die Reihenfolge seiner Analysekomponenten entspricht Empfehlungen in der Literatur [Lan-2006, Seite 4] und richtet sich nach den individuellen Voraussetzungen der jeweiligen Analysekomponenten: Die in der NER (2.3) benötigten Konstituenten sind das Ergebnis des Parsers, welcher auf Basis der PoS -Tags und Lemma-Formen (siehe Abschnitt 3.2.2 auf Seite 21) arbeitet. Damit diese im Schritt Tagging und Parsing (2.2) erzeugt werden, ist wieder ein wortweise segmentierter Text (siehe Abschnitt 3.2.1 auf Seite 20), erzeugt durch in der Segmentierung (2.1), erforderlich. Drools, PoS

Um eine Drools-Umgebung in eine UIMA-Pipeline zu integrieren, um also die in der CAS annotierten Analyseergebnisse in den Drools Working Memory (siehe Abschnitt 3.4 auf Seite 38) zu überführen, Regeln auszuführen und die Arbeitsergebnisse der Regeln aus dem Drools Working Memory wieder in der CAS zu annotieren, ist die Konzeption und die Entwicklung einer eigener Analysis Engine erforderlich. Dies trifft auch auf die Analysis Engine zur automatischen Korrektur (1.3) zu. Drools Integration

4.3. Entwicklungsumgebung

Der Autor empfiehlt die Integrated Development Environment (IDE) Eclipse[11]. Diese kann für die verschiedenen, im Projekt beteiligten Programmier- und Auszeichnungssprachen durch Plugins sinnvoll erweitert werden:

Maven Plugins stellen Mavens Projektkonfigurationsdatei Project Object Model (POM) graphisch dar und erleichtern so deren Bearbeitung. Weiterhin können angebundene Repositories beim Bearbeiten der Projektabhängikeiten durchsucht werden und Einstellungen validiert werden. Der Autor empfiehlt hierfür die m2e-Plugins[12].

UIMA Plugins stellen die XML-basierten Type System Definitions von UIMA graphisch dar, unterstützen deren Bearbeitung und ermöglichen die automatische Generierung von Java-Klassen anhand der vorgenommen Änderungen. Daher sind sie insbesondere bei der Erweiterung des UIMA-Typsystems hilfreich. Die Installation ist im Handbuch von UIMA beschrieben[13].

Drools Plugins befähigen Eclipse, die Drools-Syntax zu unterstützen. Außerdem kann durch sie ein *Drools-Audit-Log* graphisch dargestellt und so die Agenda (siehe Abschnitt 3.4 auf Seite 38) einer Drools-Session betrachtet werden. Dies ermöglicht detaillierte Einblicke in den Ablauf der Rule Engine. Die Installation ist in der Drools Dokumentation[14] beschrieben.

Zur Betrachtung der CSV-basierten Codeliste (siehe Abschnitt 1.1.1 auf Seite 3) empfiehlt der Autor LibreOffice[15].

[11] `http://www.eclipse.org/` (letzter Abruf am 2012-08-28).
[12] `http://eclipse.org/m2e/` (letzter Abruf am 2012-08-28).
[13] `http://uima.apache.org/downloads/releaseDocs/2.2.2-incubating/docs/html/overview_and_setup/overview_and_setup.html#ugr.ovv.eclipse_setup` (letzter Abruf am 2012-08-28).
[14] `http://docs.jboss.org/drools/release/5.2.0.Final/droolsjbpm-introduction-docs/html/ch03.html` (letzter Abruf am 2012-08-28).
[15] `http://de.libreoffice.org/` (letzter Abruf am 2012-08-28).

4.4. Implementierung

Abbildung 4.4 zeigt das UML Klassendiagramm von Query Analysis mit allen für die Lösung entwickelten Java-Klassen, ihren wichtigsten Attributen und Methoden sowie deren Interfaces. Außen vor sind daher alle Drittanbieterimplementierungen, wie etwa existierende Analysis Engines oder Logger.

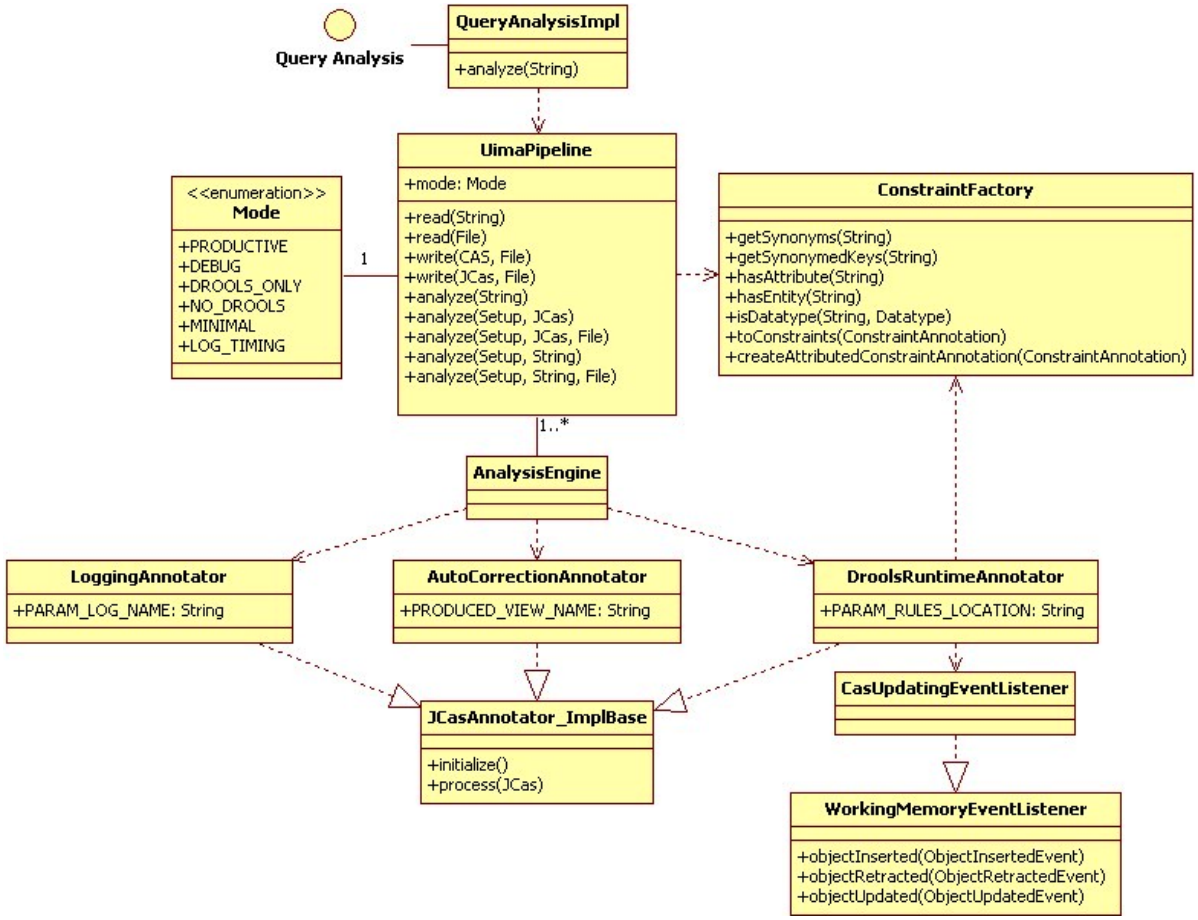

Abbildung 4.4.: UML Klassendiagramm von Query Analysis.

Die einzelnen detaillierten Aspekte der Implementierung werden auf den folgenden Seiten näher erläutert. Sofern es dem Verständnis beiträgt, kommt dabei jeweils ergänzend ein detaillierteres UML-Klassendiagramm mit allen Attributen und Methoden beteiligter Klassen zum Einsatz.

4.4.1. Erweiterung des Typsystems

Constraint-Annotation

Damit, wie in 4.2 auf Seite 45 dargelegt, die Pipeline Constraint-Objekte annotieren und verarbeiten kann, muss das Typsystem von UIMA entsprechend um den neuen Typen `ConstraintAnnotation` erweitert werden (siehe Abschnitt 3.3.2.1 auf Seite 30).

Typenhierarchie

Abbildung 4.5 zeigt die Hierarchie des neuen Typen, welcher den von UIMA mitgelieferten Typ `Annotation` spezialisiert und die von der NER gefundenen Constraints (siehe Abschnitt 4.2 auf Seite 46) temporär oder dauerhaft innerhalb der CAS repräsentiert.

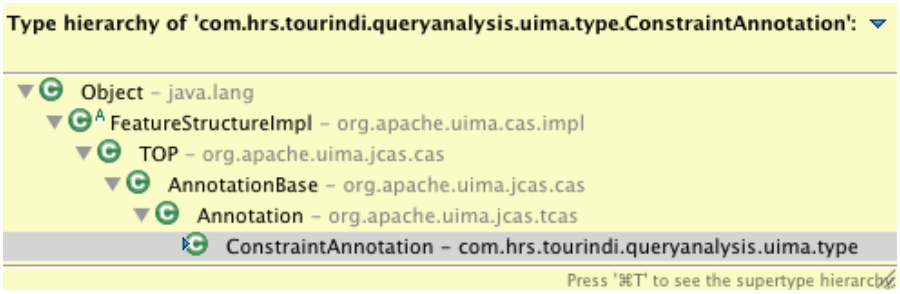

Abbildung 4.5.: Typenhierarchie der ConstraintAnnotation.

Beschreibung

Die Beschreibung des Types erfolgt mit der Type System Definition in XML. Diese wird im Listing 4.1 auszugsweise abgebildet. Ersichtlich wird die Angabe der Oberklasse im UIMA Typsystem, von der geerbt wird (`supertypeName`) und die Definition der drei Features (siehe Abschnitt 3.3.2.1 auf Seite 30) `key`, `attribute` und `parent`.

Listing 4.1: Auszug aus der Type System Definition von `ConstraintAnnotation`.

```
1  <typeDescription>
2    <name>com.hrs.tourindi.queryanalysis.uima.type.
       ConstraintAnnotation</name>
3    <supertypeName>uima.tcas.Annotation</supertypeName>
4    <features>
5      <featureDescription>
6        <name>key</name>
7        <rangeTypeName>uima.cas.String</rangeTypeName>
8      </featureDescription>
9      <featureDescription>
10       <name>parent</name>
11       <description/>
```

```
12        <rangeTypeName>uima.tcas.Annotation</rangeTypeName>
13      </featureDescription>
14      <featureDescription>
15        <name>attribute</name>
16        <description/>
17        <rangeTypeName>uima.cas.String</rangeTypeName>
18      </featureDescription>
19    </features>
20 </typeDescription>
```

Das Feature `key` entspricht dabei dem Schlüssel eines Constraints in der Codeliste und `attribute` dem Schlüssel eines Attribut-Werts in der Codeliste. `parent` repräsentiert einen Elternknoten als `Annotation` in der Konstituentendarstellung (siehe Abschnitt 3.2.3 auf Seite 22). Codeliste

Die Übersetzung der Type System Definition in ausführbaren Java-Code übernimmt das Werkzeug *JCasGen*[16]: Es liest die Beschreibung ein, fügt die referenzierten Typen aller Komponenten zu einem einzigen Typsystem zusammen und generiert entsprechende Java-Klassen.

Diese Schritte werden durch entsprechende IDE-Plugins (siehe Abschnitt 4.3 auf Seite 47) angeleitet und können durch sie bei Änderungen der Beschreibung mit Hilfe der IDE automatisiert werden. IDE-Plugins

[16] `http://uima.apache.org/downloads/releaseDocs/2.1.0-incubating/docs/html/tools/tools.html#ugr.tools.jcasgen` (letzter Abruf am 2012-09-01).

4.4.2. QueryAnalysisImpl

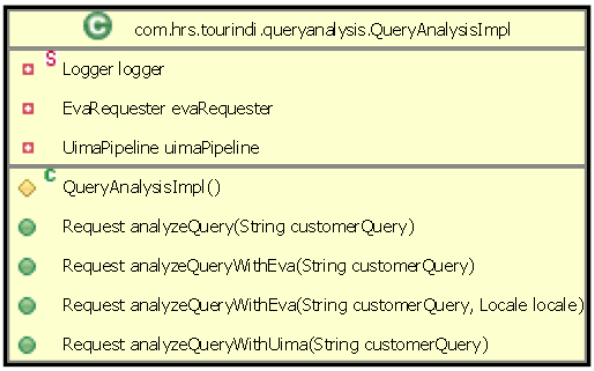

Abbildung 4.6.: UML Klassendiagramm von `QueryAnalysisImpl`.

`QueryAnalysisImpl` (Abbildung 4.6) implementiert das `QueryAnalyis`-Interface (siehe Listing 4.2), welches allen externen Benutzern von Query Analysis als unkomplizierte Schnittstelle zur Analyse von natürlichsprachlichen Hotelsuchanfragen dient. Interface

Listing 4.2: Das Interface `QueryAnalysis` als einfache Schnittstelle zur Textanalyse.

```
1 public interface QueryAnalysis {
2   public Request analyzeQuery(String customerQuery);
3 }
```

Die Klasse delegiert die natürlichsprachliche Hotelsuchanfrage in der überschriebenen Interface-Methode `analyzeQuery()` an die jeweilige Textanalyseimplementierung mittels EVA (in der Methode `analyseQueryWithEva()`) bzw. UIMA (in der Methode `analyseQueryWithUima()`). EVA & UIMA

Die Aufruflogik der Vorgängerimplementierung mit Hilfe des Webservice EVA (siehe Abschnitt 1.1.2 auf Seite 4) ist erhalten geblieben, sodass künftig beide Analyseimplementierungen nacheinander zum Einsatz kommen könnten[17]. EVA

[17]Mehr zum eventuellen, künftigen Einsatz von EVA findet sich im Ausblick Abschnitt 6.3 auf Seite 88.

4.4.3. ConstraintFactory

Die Klasse `ConstraintFactory` stellt Hilfsmethoden zur Verfügung, die vor allem bei der Erstellung oder Umwandlung von Constraint-Objekten (siehe Abschnitt 1.1.1 auf Seite 3) beteiligt sind. Im weiteren Sinne ist das insbesondere Funktionalität, die auf Daten und Einstellungen in der Codeliste (siehe Abschnitt 1.1.1 auf Seite 3) basiert.

Constraint

Codeliste

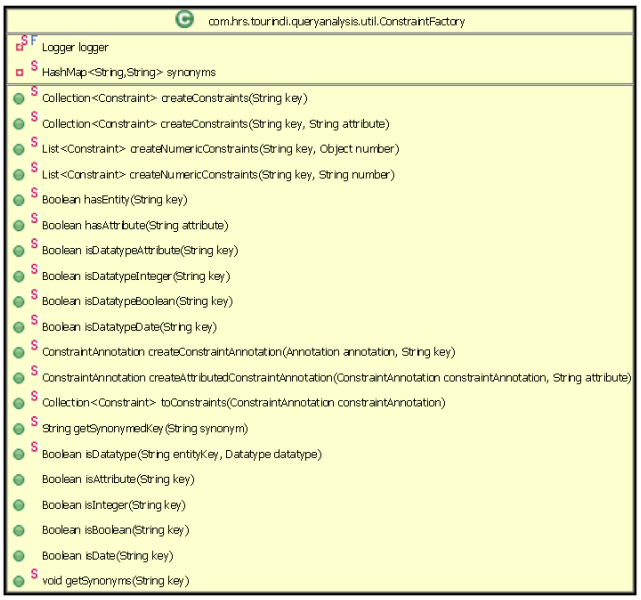

Abbildung 4.7.: UML Klassendiagramm von `ConstraintFactory`.

Die Klasse stellt daher in erster Linie grundlegende Methoden zum Zugriff auf bestimmte Felder der Codeliste zur Verfügung. Dazu gehören verschiedene Abfragen auf Attribute und Entitäten der Liste (`hasAttribute()` und `hasEntity()`), sowie Synonyme (`getSynonyms()` und `getSynonymedKey()`) und Datentypen wie `isAttribute()` oder `isBoolean()`. Als Identifikator dient stets der Schlüssel der Codeliste.

Abfragen

Konstruktionsmethoden von `ConstraintFactory` wie `createConstraintAnnotation(Annotation, String)` dienen in erster Linie der Erzeugung vollständiger Constraint-Objekte anhand eines gegebenen Schlüssels. In diesem Vorgang finden Vorgaben des Datenmodells und die konfigurierte Werte der Codeliste Beachtung. Weiter bieten angebotene Umwandlungsmethoden die Konvertierung von Constraint-Objekten an, etwa zur Ergänzung von Attributen wie `createAttributedConstraintAnnotation(ConstraintAnnotation, String)` oder zur Wandlung des in UIMA genutzten Constrainttyps `ConstraintAnnotation` (siehe Abschnitt 4.4.1 auf Seite 49) in TourInDi Constraints.

Konstruktion

Umwandlung

4.4.4. UimaPipeline

Die Klasse `UimaPipeline` kapselt alle UIMA-spezifischen Komponenten und Funktionen innerhalb von Query Analysis. Sie beinhaltet die genaue Definition und korrekte Initialisierung aller Analysekomponenten der Pipeline und nimmt mit Hilfe der Enumeration `Mode` detaillierten Einfluss auf deren Zusammensetzung und Ablauf. Weiter implementiert sie ein Singleton-Pattern[18] für Analysis Engines und initialisiert erforderliche Ressourcen vollständig beim Start statt beim ersten Aufruf („eager loading“ statt „lazy loading“[19]).

Initialisierung

`Mode`

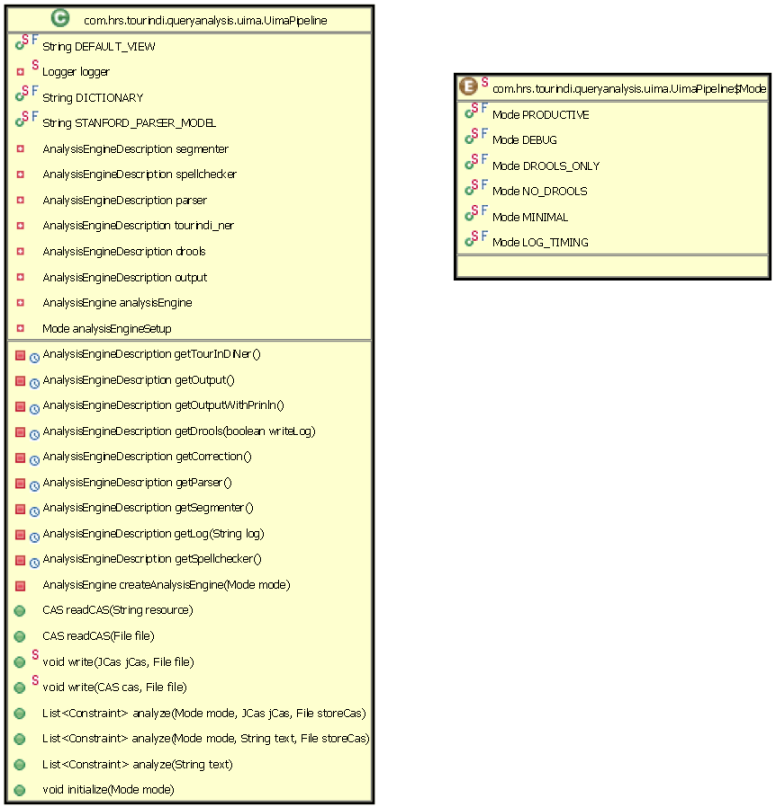

Abbildung 4.8.: UML Klassendiagramm von `UimaPipeline`.

Einstiegspunkt für die Textanalyse stellt die Methode `analyze(Mode, JCas, File)` bzw. deren überladenen, simplifizierten Varianten dar. Diese erzeugt mit der Methode `createAnalysisEngine(Mode)` eine für den gewählten `Mode` passende UIMA-Pipeline, führt die Analyse des übergebenen JCas durch und liefert anschließend annotierte Constraints zurück.

[18] `http://de.wikipedia.org/wiki/Singleton_(Entwurfsmuster)` (letzter Abruf am 2012-09-01).

[19] `http://en.wikipedia.org/wiki/Lazy_loading` (letzter Abruf am 2012-09-01).

Tabelle 4.2 zeigt, welche Zustände die Pipeline mit der Enumeration `Mode` einnehmen kann. Diese Unterscheidung macht zum Beispiel bei Entwicklung oder Verbesserung von Analysekomponenten Sinn, bei dem etwa `Mode.DEBUG` für ausführlichere Log-Ausgaben und ein Abspeichern des verarbeiteten CAS gewählt werden kann. Beides würde dagegen im produktiven Einsatz (`Mode.PRODUCTIVE`) stören, da sie zum Teil zu erheblichen Verzögerungen führen.

Mode

Tabelle 4.2.: Konstanten der Enumeration `Mode` und ihre Bedeutung.

`Mode`-Konstante	Betriebsmodus der Pipeline
`DEBUG`	Gibt den Inhalt der CAS auf `System.out` aus.
`DROOLS_ONLY`	Initialisiert und benutzt lediglich die Drools-spezifischen Teile der Pipeline.
`LOG_TIMING`	Protokolliert einen Zeitstempel nach jedem Verarbeitungsschritt.
`MINIMAL`	Initialisiert die minimalste Zusammensetzung der Pipeline, die für das Deserialisieren einer serialisierten CAS notwendig ist.
`NO_DROOLS`	Initialisiert alle üblichen Pipelinekomponenten bis auf die Drools-spezifischen Teile.
`PRODUCTIVE`	Der übliche, produktive Einsatz der Pipeline mit allen dafür erforderlichen Analysekomponenten.

4.4.4.1. Erzeugung der Pipeline

`createAnalyseEngine(Mode)` erzeugt eine Pipeline aus UIMA-Analysis Engines, die, wie im Architektur-Abschnitt auf Seite 45 bereits beschrieben wurde, view-unaware ist: es wird bei der Konstruktion der Pipeline bestimmt, welcher der Standard View (siehe Abschnitt 3.3.2.2 auf Seite 32) jeder einzelnen Analysis Engine ist. Eine Konstruktionsklasse in uimaFIT, die dies zulässt, ist `org.uimafit.factory.AggregateBuilder`.

view-unaware

Listing 4.3 zeigt, wie diese in `createAnalyseEngine(Mode)` für die Erzeugung einer Pipeline (im Beispiel: `Mode.PRODUCTIVE`) zum Einsatz kommt. Zudem zeichnen sich die beiden getrennten Analyseprozesse (siehe Abschnitt 4.2 auf Seite 46) bei der Zusammensetzung ab.

Pipeline-Erzeugung

Listing 4.3: `UimaPipeline` - Zusammensetzung der Pipeline für `MODE.PRODUCTIVE`.

```
1 aggregateBuilder.add(getSegmenter(), CAS.NAME_DEFAULT_SOFA,
      DEFAULT_VIEW);
```

```
2 aggregateBuilder.add(getSpellchecker(), CAS.NAME_DEFAULT_SOFA,
      DEFAULT_VIEW);
3 aggregateBuilder.add(getCorrection(), CAS.NAME_DEFAULT_SOFA,
      DEFAULT_VIEW);
4
5 aggregateBuilder.add(getSegmenter(), CAS.NAME_DEFAULT_SOFA,
      CORRECTED_VIEW);
6 aggregateBuilder.add(getParser(), CAS.NAME_DEFAULT_SOFA,
      CORRECTED_VIEW);
7 aggregateBuilder.add(getDrools(mode == Mode.DROOLS_ONLY), CAS.
      NAME_DEFAULT_SOFA, CORRECTED_VIEW);
```

add-Methode

Die verwendete `add`-Methode von `AggregateBuilder` hat zwei Parameter: Der erste ist die Beschreibung einer konkreten Analysis Engine - dies übernehmen jeweils private Hilfsmethoden (zum Beispiel `getSegmenter()` oder `getParser()`) von `UimaPipeline`, die im folgenden Abschnitt 4.4.4.3 auf Seite 57 näher beschrieben sind. Der zweite Parameter sind Key-Value-Paare, die der Konfiguration der Analysis Engine dienen.

zwei Analyseprozesse

Auf diese Weise werden die beiden getrennten Analyseprozesse (siehe Abschnitt 4.2 auf Seite 46) realisiert: die ersten drei Analysis Engines (Zeile 1 bis 3) operieren auf dem Standard View `DEFAULT_VIEW` (siehe Abschnitt 3.3.2.2 auf Seite 32) einer CAS - der direkten Benutzereingabe - und die folgenden (Zeile 5 bis 7) auf der korrigierten Variante davon, `CORRECTED_VIEW`. Beide Konstanten sind in `UimaPipeline` definiert, wobei `CORRECTED_VIEW` der Name des Views ist, den die Analysekomponente `AutoCorrectionAnnotator` erzeugt (Listing 4.4):

Listing 4.4: `UimaPipeline` - Konstanten für die verwendeten View-Namen.

```
1 public final static String DEFAULT_VIEW = CAS.NAME_DEFAULT_SOFA;
2 public final static String CORRECTED_VIEW =
      AutoCorrectionAnnotator.PRODUCED_VIEW_NAME;
```

4.4.4.2. Zusammensetzung der Pipeline

In Anlehnung an Tabelle 4.1 auf Seite 45 im Architekturabschnitt zeigt Tabelle 4.3, welche konkreten Implementierungen als Analysis Engines in der Pipeline für welchen Zweck eingesetzt werden.

Tabelle 4.3.: Konkrete Analysis Engine -Implementierungen in Query Analysis

Analyseschritt	CAS-View		Analysis Engine -Implementierung
	Initial	Korr.	
Tokenization	1.1	2.1	`de.tudarmstadt.ukp.dkpro.core.stanfordnlp.StanfordSegmenter`
Rechtschreibprüfung	1.2		`de.tudarmstadt.ukp.dkpro.core.jazzy.SpellChecker`
Automatische Korrektur	1.3		`com.hrs.tourindi.queryanalysis.uima.AutoCorrectionAnnotator`
Tagging & Parsing		2.2	`de.tudarmstadt.ukp.dkpro.core.stanfordnlp.StanfordParser`
Named Entity Recognition		2.3	`com.hrs.tourindi.queryanalysis.uima.DroolsRuntimeAnnotator`

Es kommen Komponenten des Stanford-NLP-Projekts[20] für die Tokenization (siehe Abschnitt 3.2.1 auf Seite 20) sowie für das Tagging (siehe Abschnitt 3.2.2 auf Seite 20) und Parsing (siehe Abschnitt 3.2.3 auf Seite 22) zum Einsatz. Tagger und Parser erzeugen ihre Ergebnisse dabei mit Hilfe einer sprachspezifischen Resource. Als solche wird im Rahmen dieser Arbeit die Probabilistic Context-Free Grammar (PCFG) *caseless englisch PCFG* eingesetzt, welche das Penn-Tree Tagset (siehe Abschnitt 3.2.2 auf Seite 21) einsetzt und beispielsweise auch zum Parsen von Tweets empfohlen wird[21]. Sie kann von der Stanford-NLP-Webseite heruntergeladen werden[22] und funktioniert mit der aktuellen Version von DKPro. Für die Rechtschreibkorrektur wird der Jazzy-Algorithmus[23] verwendet, einer Java-Implementierung des Open-Source-Werkzeugs Aspell. Jazzy arbeitet grundsätzlich mit Hilfe des Levenshtein-Algorithmus[24]. Die für Query Analysis implementierten Eigenent-

Stanford-NLP

PCFG

Jazzy

[20] `http://nlp.stanford.edu/` (letzter Abruf am 2012-09-03).

[21] `http://nlp.stanford.edu/software/parser-faq.shtml#ca` (letzter Abruf am 2012-09-03).

[22] `http://nlp.stanford.edu/software/lex-parser.shtml#Download` (letzter Abruf am 2012-10-05).

[23] Mehr Informationen zur Funktionsweise des Jazzy-Algorithmus findet sich unter `http://www.ibm.com/developerworks/java/library/j-jazzy/` (letzter Abruf am 2012-09-03).

[24] `http://de.wikipedia.org/wiki/Levenshtein-Distanz` (letzter Abruf am 2012-10-05).

wicklungen `AutoCorrectionAnnotator` und `DroolsRuntimeAnnotator` werden in Abschnitt 4.4.5 auf Seite 60 bzw. in Abschnitt 4.4.6 auf Seite 62 näher beschrieben.

Mit einem Aufruf (Listing 4.5) werden alle hinzugefügten Analysis Engines als Aggregate Analysis Engines zusammengeschlossen und das Erzeugen der Pipeline abgeschlossen:

Aggregate Analysis Engine

Listing 4.5: `UimaPipeline` - Erzeugung der Pipeline als Aggregate Analysis Engine.

```
1 analysisEngine = aggregateBuilder.createAggregate();
```

Die Pipeline kann fortan in `analyze(Mode, JCas, File)` verwendet werden, um mittels `process(JCas)` JCas-Objekte zu verarbeiten.

4.4.4.3. Initialisierung von Ressourcen

Bei der Verwendung von `UimaPipeline` wird die Lademethode von UIMA-Komponenten vereinheitlicht. Die Klasse implementiert für alle verwendeten Analysis Engines ein Singleton-Pattern, da bei der Erzeugung einer Analysis Engine auch teilweise dessen Ressourcen initialisiert werden. Dieser Vorgang kann sehr zeitaufwändig sein - er liegt bei manchen Analysis Engines im Bereich mehrerer Sekunden - sodass durch Verwendung des Singleton-Patterns ein unnötiges, mehrfaches initialisieren vermieden wird. Listing 4.6 zeigt dies anhand des Beispiels des Stanford Parsers.

Singleton-Pattern

Listing 4.6: Singleton-Pattern bei der Erzeugung des Stanford Parsers.

```
1 private synchronized AnalysisEngineDescription getParser() throws
      ResourceInitializationException {
2     if (parser == null) {
3       parser = createPrimitiveDescription(StanfordParser.class,
            StanfordParser.PARAM_MODEL_LOCATION,
            STANFORD_PARSER_MODEL, StanfordParser.
            PARAM_CREATE_CONSTITUENT_TAGS,
4           true, StanfordParser.PARAM_CREATE_DEPENDENCY_TAGS, true
                , StanfordParser.PARAM_CREATE_PENN_TREE_STRING, true
                , StanfordParser.PARAM_CREATE_POS_TAGS, true);
5     }
6     return parser;
```

Einige Analysis Engines laden ihre Ressourcen jedoch lazy loading, also erst sobald sie bei einer Analyse tatsächlich gebraucht werden[25]. Deshalb bedient sich `UimaPipeline` zusätzlich zum Singleton-Pattern eines relativ einfachen Tricks: direkt nach Erzeugung der Pipeline wird automatisch die Analyse der Phrase {`I am a demo sentence that initializes lazy-loaded resources.`} einzig zu dem Zweck durchgeführt, um alle beteiligten Ressourcen auch tatsächlich zu initialisieren. Eventuelle Fehler, die etwa durch fehlende oder unvollständige Ressourcen verursacht werden, treten so früher auf, während der Initialisierung der Anwendung.

lazy loading

Weiter aktualisiert `UimaPipeline` auch das der Rechtschreibprüfung zugrundeliegende Wörterbuch. Dieses besteht ausschließlich aus Wörtern der Codeliste (siehe Abschnitt 1.1.1 auf Seite 3), also aus Schlüsseln oder Synonymen von Constraints. Bei Initialisierung der Rechtschreibprüfung wird die Wortliste daher generiert und gespeichert.

Wörterbuch

4.4.4.4. Durchführen der Analyse

Die Ein- und Ausgabe des Analysegegenstands erfolgt nicht über den UIMA-typischen Collection Reader und CAS Consumer (siehe Abschnitt 3.3 auf Seite 27), um zusätzlichen Overhead zu vermeiden. Stattdessen wird, wie es in uimaFIT üblich[26] ist, eine neue CAS aus dem zu analysierenden `String` erzeugt. In Listing 4.7 ist dargestellt, wie genau die Erzeugung einer CAS (bzw. `JCas`) in einer überladenen Variante von `analyze()` erfolgt:

neue CAS

Listing 4.7: Erzeugung einer CAS anhand des gegebenen Strings `text`.

```
1  public List<Constraint> analyze(Mode mode, String text, File
       storeCas) throws Exception {
2    JCas jCas = createAnalysisEngine(mode).newJCas();
3    jCas.setDocumentLanguage(QueryAnalysis.DEFAULT_LOCALE.
       toString());
4    jCas.setDocumentText(text);
5    return analyze(mode, jCas, storeCas);
6  }
```

[25] `http://en.wikipedia.org/wiki/Lazy_loading` (letzter Abruf am 2012-09-01).

[26] Dutzende, bei DKPro mitgelieferte Programmcodes verwenden die genannte Methode und erzeugen bei Analysestart schlicht eine CAS und übergeben diese manuell der `process()`-Methode.

CAS sind stets an das Typsystem (siehe Abschnitt 3.3.2.1 auf Seite 30) einer spezifischen UIMA-Pipeline gebunden. Aus diesem Grund stellt der erste Schritt der Erzeugung einer CAS das Erzeugen einer (passenden) Pipeline dar (Zeile 2). Das anschließende Erzeugen eines Sofa - das Setzen der Sprache und der zu analysierenden Hotelsuchanfrage als Analysetext der CAS in Zeile 3 bis 4 - sind selbsterklärend.

Typsystem

Liegt der zur analysierende Text als `JCas` vor, so wird diese der in Abschnitt 4.4.4.1 auf Seite 54 erzeugten Pipeline über deren `process`-Methode übergeben. Analyseergebnisse werden dabei direkt in der `JCas` gespeichert.

`process`

Wie bereits in der Verarbeitung (siehe Abschnitt 4.4.4.1 auf Seite 54) erfolgt auch das Auslesen annotierter Constraints (`ConstraintAnnotations`) in `analyze(Mode, JCas, File)` view-unaware, sodass der View, aus dem gelesen werden soll (`CORRECTED_VIEW`), explizit angefordert werden muss. Listing 4.8, zeigt, wie dies mit Hilfe des uimaFIT-Werkzeugs `JCasUtil` funktioniert und wie Umwandlungsmethoden der `ConstraintFactory` (siehe Abschnitt 4.4.3 auf Seite 52) zum Einsatz kommen, um die gefundenen `ConstraintAnnotation`-Objekte in `Constraint`-Objekte zu konvertieren:

Auslesen annotierter Constraints

Listing 4.8: Auslesen annotierter Constraints (`ConstraintAnnotations`).

```
1 for (ConstraintAnnotation constraintAnnotation : JCasUtil.select(
    jCas.getView(CORRECTED_VIEW), ConstraintAnnotation.class)) {
2   constraints.addAll(ConstraintFactory.toConstraints(
      constraintAnnotation));
3 }
```

Die Methode `JCasUtil. select (JCas, Class<T>)` wird eingesetzt, um über Feature Structures (siehe Abschnitt 3.3.2.1 auf Seite 30) einer CAS zu iterieren. Dafür besitzt sie zwei Parameter: der erste gibt das `JCas`-Objekt an, aus dem die Feature Structure selektiert werden soll (im Code wird direkt der korrigierte View mittels `jCas.getView(CORRECTED_VIEW)` ausgewählt), der zweite den genauen Typen als Java-Klasse (im Code `ConstraintAnnotation.class`). Die Methode erzeugt eine `Collection` aus gewünschten `ConstraintAnnotations`, durch die iteriert wird. Pro Iteration wird jede `ConstraintAnnotation` in die entsprechende `Constraints` überführt und zur Ergebnisliste `Constraints` hinzugefügt.

`JCasUtil. select (JCas, Class<T>)`

4.4.5. AutoCorrectionAnnotator

Die Klasse `AutoCorrectionAnnotator` ist eine für das TourInDi Projekt entwickelte Analysis Engine. Ihre Aufgabe ist es, die Korrekturen, die im vorausgehenden Analyseschritt (siehe Abschnitt 4.3 auf Seite 56) durch die Rechtschreibprüfung mit Jazzy (siehe Abschnitt 4.4.4.2 auf Seite 56) als Feature Structure (siehe Abschnitt 3.3.2.1 auf Seite 30) `SpellingAnomaly` in der CAS annotiert wurden, automatisch zu korrigieren. Ausgangsbasis ist dabei der Sofa des Standard Views (siehe Abschnitt 3.3.2.2 auf Seite 32), der die unkorrigierte Hotelsuchanfrage (siehe Abschnitt 4.7 auf Seite 58) enthält.

SpellingAnomaly

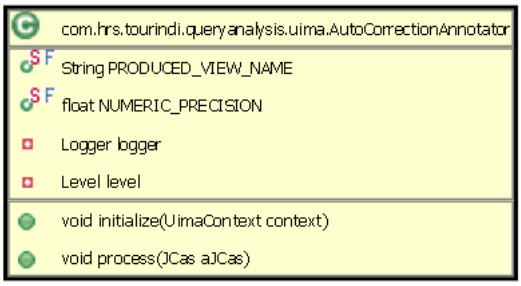

Abbildung 4.9.: UML Klassendiagramm von `AutoCorrectionAnnotator`.

Listing 4.9 zeigt das prinzipielle Vorgehen der Methode `process()` bei der Korrektur, die aufgrund der Paradigmen von UIMA (siehe Abschnitt 4.2 auf Seite 44) in einem neuen View (`PRODUCED_VIEW_NAME`) erfolgt:

Listing 4.9: `AutoCorrectionAnnotator` - Erzeugung eines neuen CAS-Views.

```
1 public void process(JCas aJCas) throws ... {
2   JCas view = aJCas.createView(PRODUCED_VIEW_NAME);
3   String text = aJCas.getDocumentText();
4   for (SpellingAnomaly spellingAnomaly : JCasUtil.select(aJCas,
       SpellingAnomaly.class)) {
```

Zunächst wird ein neuer View inklusive des Sofas die korrigierte Hotelsuchanfrage erstellt (Zeile 2) und den Text der unkorrigierten Sofa in einer Variablen gespeichert (Zeile 3). In Zeile 4 wird über alle `SpellingAnomaly`-Feature Struktures in der CAS iteriert. Diese wurden von Jazzy (siehe Abschnitt 4.4.4.2 auf Seite 56) annotiert und beinhalten für einen eventuell falsch geschriebenen Ausschnitt des ursprünglichen Sofas mindestens eine Korrektur in Form von `SuggestedAction`-Objekten, mit jeweils einem Gewissheitsfaktor `Certainty` zwischen 0 und 1.

neuer Sofa

Listing 4.10 zeigt den Programmcode innerhalb der Iteration, der für jede in der CAS annotierte `SpellingAnomaly` ausgeführt wird:

Listing 4.10: `AutoCorrectionAnnotator` - Durchführung der automatischen Korrektur.

```
5     FSArray suggestions = spellingAnomaly.getSuggestions();
6
7     for (int i = 0; i < suggestions.size(); i++) {
8       SuggestedAction suggestedAction = (SuggestedAction)
            suggestions.get(i);
9
10      if ((suggestedAction.getCertainty() - 1.0) <
            NUMERIC_PRECISION) {
11        text = text.substring(0, spellingAnomaly.getBegin()) +
              suggestedAction.getReplacement() + text.subSequence(
              spellingAnomaly.getEnd(), text.length());
12        break;
13      }
14    }
15  }
```

Es wird über die möglichen Korrekturen der CAS iteriert (Zeile 7) und für jede einzelne (numerisch stabil) geprüft, ob es sich dabei um eine Korrektur mit hohem Gewissheitsfaktor (1.0) handelt. Falls ja, wird der fehlerhafte Bereich der Sofa des Standard Views durch seine Korrektur ersetzt und der gesamte Text zwischengespeichert (Zeile 11). Korrektur

Listing 4.11 zeigt, wie der zwischengespeicherte, korrigierte Text nach der Iteration über alle `SpellingAnomaly`-Annotations als Text der Sofa des neu erzeugten Views (siehe oben, `PRODUCED_VIEW_NAME`) gesetzt wird und die Sprachangabe kopiert wird. Speicherung

Listing 4.11: `AutoCorrectionAnnotator` - Speicherung des korrigierten Texts als Sofa.

```
16    view.setDocumentText(text);
17    view.setDocumentLanguage(aJCas.getDocumentLanguage());
18  }
```

4.4.6. DroolsRuntimeAnnotator

Die Klasse `DroolsRuntimeAnnotator` ist ebenfalls eine für das TourInDi Projekt entwickelte Analysis Engine. Durch sie wird eine Umgebung für die regelbasierte NER (siehe Abschnitt 3.2.4.2 auf Seite 24) mit dem PRS JBoss Drools (siehe Abschnitt 3.4 auf Seite 36) in UIMA hergestellt. Dafür erzeugt sie einen Arbeitsbereich für Drools-Regeln, in den sie die Feature Structures (siehe Abschnitt 3.3.2.1 auf Seite 30) einer zu analysierenden CAS überführt. Anschließend werden Änderungen innerhalb diesen Arbeitsbereichs in der CAS nachgezogen, sodass Analyseergebnisse UIMA-üblich weiterverarbeitet werden können. Arbeitsweise

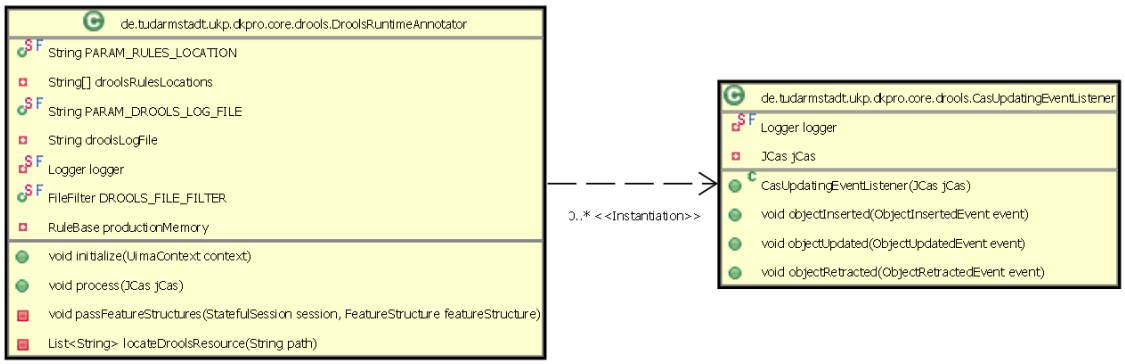

Abbildung 4.10.: UML Klassendiagramm von `DroolsRuntimeAnnotator`.

4.4.6.1. Konfiguration und Initialisierung

Die Konfiguration der Analysis Engine erfolgt uimaFIT-üblich über Annotations (siehe Abschnitt 3.3.3.1 auf Seite 34), die über den privaten Feldern der Implementierungsklasse `DroolsRuntimeAnnotator` angegeben werden. Diese sind: Konfiguration

`PARAM_RULES_LOCATION` Eine Liste an Pfaden, in denen nach Drools-Ressourcen gesucht werden soll. Gefundene `.drl`-Dateien werden automatisch geladen.

`PARAM_DROOLS_LOG_FILE` Speicherort des *Working Memory Log Files*.

Die Klasse wird einmalig mit diesen Einstellungen initialisiert. Sind geladene Drools-Ressourcen fehlerhaft, werden automatisch ausführlichere Meldungen ausgegeben. Fehlerfall

Listing 4.12 zeigt den prinzipiellen Ablauf dieser Initialisierungslogik für jede gefundene Drools-Resource.

Listing 4.12: `DroolsRuntimeAnnotator` - Initialisierung des Production Memory.

```
1 try {
2   logger.info("Loading Drools resource {}", droolsResource);
3   packageBuilder.addPackageFromDrl(new InputStreamReader(
       DroolsRuntimeAnnotator.class.getResourceAsStream(
       droolsResource)));
4 } catch (Exception e) {
5   throw new ResourceInitializationException(e);
6 }
7
8 if (packageBuilder.hasErrors()) {
9   for (KnowledgeBuilderError packageBuilderErrors :
       packageBuilder.getErrors()) {
10    logger.error("{}: {}", packageBuilderErrors.getSeverity().
        toString(), packageBuilderErrors.getMessage());
11  }
12  throw new ResourceInitializationException("Drools builder has
       errors!", null);
13 }
```

Drools-Regeln werden einzeln in eine `PackageBuilder`-Instanz eingelesen (Zeile 3). Diese kompiliert die Regeln für den Production Memory (siehe Abschnitt 3.4 auf Seite 38). Im Fehlerfall werden auftretende Exceptions als UIMA-übliche `ResourceInitialization-Exception` weiter geworfen (Zeile 5). Treten allerdings Compiler-Fehler für bestimmte Drools-Ressourcen auf, greift die Abfrage in Zeile 8 und gibt ausführlichere Meldungen (Zeile 10) zum aufgetretenen Problem aus. Fehlerfall

Ist beim Einlesen aller angegebener Ressourcen kein Fehler aufgetreten, werden die kompilierten Drools-Regeln aus dem `PackageBuilder` in den Production Memory überführt.

4.4.6.2. Erzeugen einer Session

Der erste Schritt, der bei der Durchführung einer Analyse stattfindet, ist das Erzeugen einer stateful session (siehe Abschnitt 3.4 auf Seite 38) anhand des bei der Initialisierung in Abschnitt 4.4.6.1 auf Seite 62 vorbereiteten Production Memory (siehe Abschnitt 3.4 auf Seite 38), welcher bereits die Regeln in kompilierter Form enthält.

stateful session

4.4.6.3. Überführung einer CAS nach Drools

Liegt die `session` vor, werden die Feature Structures (siehe Abschnitt 3.3.2.1 auf Seite 30) der zu analysierenden CAS als Fakten (siehe Abschnitt 3.4 auf Seite 37) in den Working Memory (siehe Abschnitt 3.4 auf Seite 38) überführt. Da die CAS dabei über verschachtelte Feature Structures beliebiger Tiefe verfügen kann, geschieht diese Überführung mit Hilfe der rekursiven Hilfsmethode `passFeatureStructures()`. Diese wird ab Listing 4.13 auf den folgenden Seiten skizziert.

Working Memory

Listing 4.13: `DroolsRuntimeAnnotator` - Abbruchbedingungen der Rekursion

```
1 private void passFeatureStructures(StatefulSession session,
    FeatureStructure featureStructure) {
2   if (featureStructure.getType().isPrimitive())
3     return;
4
5   Long countBeforeInsertion = session.getFactCount();
6   session.insert(featureStructure);
7
8   if (countBeforeInsertion == session.getFactCount())
9     return;
```

Zunächst prüft die Methode in Zeile 2, ob es sich bei der übergebenen Feature Structure um einen primitiven Datentypen, wie etwa einzelne Zahlen oder Zeichenketten, handelt. Aus ihrem Kontext entnommen machen diese als Fakten keinen Sinn, weshalb die Rekursion bei ihrem Auftreten mit `return` abgebrochen wird.

Abbruchbedingungen

Anschließend wird in Zeile 6 die Feature Strukture als Fakt in den Working Memory überführt. Ändert sich dadurch die Anzahl der im Fakten im Working Memory nicht, ist eine weitere Abbruchbedingung für die Rekursion erfüllt (Zeile 9), da das hinzuzufügende Faktum bereits im Working Memory vorhanden ist.

Die eigentlichen, rekursiven Aufrufe erfolgen ab Zeile 10 in Listing 4.14:

Listing 4.14: `DroolsRuntimeAnnotator`: Rekursiver Aufruf für Feature Structures.

```
10   for (Feature feature : featureStructure.getType().getFeatures()
        ) {
11
12     if (!feature.getRange().isPrimitive()) {
13       FeatureStructure nestedFeatureStructure = featureStructure.
            getFeatureValue(feature);
14
15         if (nestedFeatureStructure instanceof FSArray) {
16           FSArray fsArray = (FSArray) nestedFeatureStructure;
17
18           for (int i = 0; i < fsArray.size(); i++) {
19             passFeatureStructures(session, fsArray.get(i));
20           }
21         }
22
23         passFeatureStructures(session, nestedFeatureStructure);
24     }
25   }
26 }
```

In Zeile 10 wird über Werte iteriert, die selbst als Feature Structures in der oben übergebenen Feature Structure enthalten sind. Für jede dieser `nestedFeatureStructures` (Zeile 16) erfolgt ein rekursiver Aufruf (Zeile 23) von `passFeatureStructures(StatefulSession, FeatureStructure)`. Array-Typen vom Basis-Typ `FSArray` müssen dabei gesondert behandelt werden (Zeile 15), da ihre enthaltenen Werte nicht wie andere Feature Structures eingebettet sind. Für sie wird in Zeile 18 gesondert iteriert und in jeder Iteration ein rekursiver Aufruf von `passFeatureStructures(StatefulSession, FeatureStructure)` abgesetzt (Zeile 19).

Iteration

`FSArray`

rekursiver Aufruf

Ist die rekursive Verarbeitung einer CAS abgeschlossen, liegen alle darin enthaltenen Feature Structures als Fakten im Working Memory von Drools vor. Abbildung 4.11 visualisiert dies anhand eines Beispiels im Drools-Audit-Log (siehe Abschnitt 4.3 auf Seite 47). Fakten

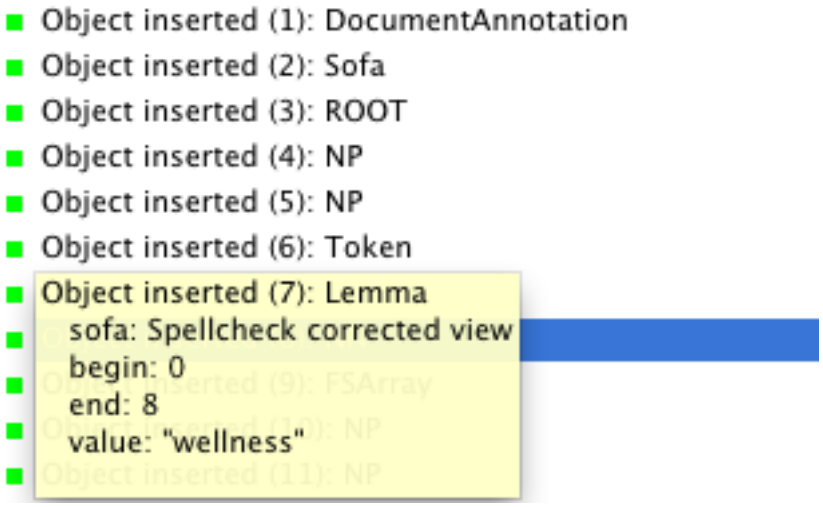

Abbildung 4.11.: Audit-Log des Drools Working Memory nach der Überführung der Feature Structures `DocumentAnnotation`, `Sofa`, `ROOT`, `NP`, `Token` und `Lemma`. Über den Tooltip bei der Anzeige in der IDE (siehe Abschnitt 4.3 auf Seite 47) werden weitere Informationen angezeigt - im Bild zur Feature Structure `Lemma`.

4.4.6.4. Event Listener

Als letzter Schritt vor dem tatsächlichen Ausführen der Regeln wird der Event Listener `CasUpdatingEventListener` (siehe Abschnitt 4.10 auf Seite 62) am Working Memory registriert. Listener

Dieser überwacht Einfüge-, Rückzugs- und Änderungsaktionen an Fakten im Working Memory und zieht diese, sofern erforderlich, in der Austauschstruktur CAS nach. Damit ist ein UIMA-typischer Umgang mit Analyseergebnissen sichergestellt, sodass diese gemäß der Design-Prinzipien von UIMA (siehe Abschnitt 3.3.2 auf Seite 29) weiterverarbeitet und -verwendet werden können. Fakten-Überwachung

In Listing 4.15 ist auszugsweise dargestellt, wie der Event Listener implementiert ist.

Listing 4.15: `CasUpdatingEventListener` - Listener für Änderungsaktionen.

```
1 public void objectUpdated(ObjectUpdatedEvent event) {
2   logger.debug("Object updated in knowledge base: {}", event.
       getOldObject());
3
4   if (event.getOldObject() instanceof FeatureStructureImpl)
5     if (event.getObject() instanceof FeatureStructureImpl) {
6       jCas.removeFsFromIndexes((FeatureStructureImpl) event.
           getOldObject());
7       jCas.addFsToIndexes((FeatureStructureImpl) event.getObject
           ());
8     }
9 }
```

Gezeigt wird die `objectUpdated`-Methode, welche bei Änderungsaktionen von Fakten im Working Memory aufgerufen wird. Diese prüft zunächst, ob es sich bei der alten Version (Zeile 3) bzw. neuen Version (Zeile 4) des geänderten Fakts um eine zuweisungskompatible Implementierung einer Feature Structure `FeatureStructureImpl` handelt. Falls ja, wird diese entsprechend der Empfehlung der Community für das Aktualisieren von Feature Structures[27] aktualisiert.

`objectUpdated`

Die Methoden `objectInserted(ObjectInsertedEvent)` und `objectRetracted(ObjectRetractedEvent)` für Einfüge- bzw. Rückzugsaktionen von Fakten sind entsprechend ähnlich durch den `CasUpdatingEventListener` implementiert.

[27] `http://uima.apache.org/downloads/releaseDocs/2.1.0-incubating/docs/html/references/references.html#ugr.ref.cas.creating_feature_structures` (letzter Abruf am 2012-09-04).

Abbildung 4.12.: Audit-Log.

Listing 4.16: Analysergebnisse einer CAS

```
[Pool]
NN
    sofa: Spellcheck corrected view
    begin: 18
    end: 22
    PosValue: "NN"
...
[Hotel with heated Pool]
PennTree
    sofa: Spellcheck corrected view
    begin: 0
    end: 22
    PennTree: "(ROOT
   (FRAG
     (NP (NN Hotel))
     (PP (IN with)
       (NP (VBN heated) (NN Pool)))))
```

4.4.6.5. Durchführung der NER

Die Durchführung der regelbasierten NER (siehe Abschnitt 3.2.4.2 auf Seite 24) erfolgt ausschließlich durch Drools Regeln, indem die Analyseergebnisse des PoS -Taggers und die Konstituentendarstellung des Parsers ausgewertet werden. Dabei kommt das forward chaining Verfahren (siehe Abschnitt 3.4 auf Seite 37) zum Einsatz, um aus Text-Fragmenten schrittweise zunächst `ConstraintAnnotation`s und schließlich zugehörige Modifikationen wie Attribute oder Negationen zu erkennen. Dabei werden die Möglichkeiten von Drools bei einer stateful session (siehe Abschnitt 3.4 auf Seite 38) auch dafür genutzt, um zunächst erkannte `ConstraintAnnotation`s gegebenenfalls wieder aus dem Working Memory zurückzuziehen, sofern zusätzliche Kontextinformationen oder andere Erkenntnisse dies nahelegen. Dieses Verhalten ist bei der regelbasierten NER als zusätzliche Maßnahme zur Steigerung der Präzision üblich (siehe Abschnitt 3.2.4.2 auf Seite 25).

forward chaining

Ausschluss

Abbildung 4.12 zeigt das Drools-Audit-Log (siehe Abschnitt 4.3 auf Seite 47) bei Analyse der Hotelsuchanfrage {`Hotel with heated Pool`}, bevor die NER -Regeln zur Anwendung kommen: In diesem Beispiel liegen 29 Analyseergebnisse in Form von Fakten im Working Memory vor, die anschließend bei der NER verwendet werden können.

Drools-Audit-Log

Ergänzend dazu zeigt Listing 4.16 auf der vorherigen Seite weitere Details zu den in den Working Memory überführten Analyseergebnissen. Das erste, `[Pool] NN`, entspricht dabei dem im Drools-Audit-Log an 22. Stelle eingefügten Faktum, `[Hotel with heated Pool] PennTree` dem an 28. Stelle.

Analyseergebnisse

Ausgehend von diesen Analyseergebnissen der beispielhaften Hotelsuchanfrage werden im Folgenden zwei NER -Regeln mit Drools zur Erkennung von Constraints exemplarisch entwickelt. Im gewählten Beispiel {`Hotel with heated Pool`} werden entsprechende Constraints für {`Hotel`} und {`heated Pool`} erwartet, wobei `heated` ein Attribut für `Pool` ist.

NER -Regeln

Zunächst werden, gemäß des forward chaining Paradigmas (siehe Abschnitt 3.4 auf Seite 37) einfache Constraints erkannt und annotiert. Im Beispiel wurden alle Constraints als Nomen (`NN`) erkannt, was sich in der Konstituentendarstellung in Zeile 2 und 4 widerspiegelt:

Konstituenten

Listing 4.17: Konstituentendarstellung von {`Hotel with heated Pool`}.

```
1  (FRAG
2      (NP (NN Hotel))
3      (PP (IN with)
4        (NP (VBN heated) (NN Pool)))))
```

Wie eine Drools-NER -Regel zur Erkennung dieser Nomen-basierten Constraints aussehen kann, ist in Listing 4.18 mit der Regel `Detect single word NN-Constraints` dargestellt. Dabei wird zunächst mit Datentypen des Typsystems des PoS -Taggers und des Parsers gearbeitet.

Constrainterkennung

Listing 4.18: NER -Regel zur Erzeugung eines Constraints.

```
1 rule "Detect single word NN-Constraints"
2   when
3     nn : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN()
4     token : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.
          Token ( entity : coveredText, pos == nn )
5     eval (hasEntity(entity))
6   then
7     insertLogical(createConstraintAnnotation(token, entity));
8 end
```

Im LHS der Regel wird ausgehend von einem Nomen (`de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN`) ein zugehöriges Token (`de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token`) gesucht und dessen Zeichenkette in der Variablen `entity` gespeichert (Zeile 4). Weiter wird in Zeile 5 über die Methode `hasEntity(String)` der `ConstraintFactory` (siehe Abschnitt 4.4.3 auf Seite 52) geprüft, ob es sich bei der Zeichenkette um einen Schlüssel eines Constraints aus der Codeliste (siehe Abschnitt 1.1.1 auf Seite 3) handelt. Treffen alle Bedingungen im LHS zu, wird im RHS der Regel über eine weitere Hilfsmethode von `ConstraintFactory`, `createConstraintAnnotation(Annotation, String)`, eine `ConstraintAnnotation` als neuen Fakt angelegt und, aufgrund des Listeners `CasUpdatingEventListener` (siehe Abschnitt 4.4.6.4 auf Seite 66), in der CAS annotiert.

Nomen

Token

neuer Fakt

Es hat sich dabei als Best Practice erwiesen, Typen innerhalb der Drools-Regeln wie im Beispiel in Listing 4.18 auf der vorherigen Seite voll-qualifiziert zu referenzieren. Dies ist dem Umstand geschuldet, dass DKPro über mehrere Klassen mit identischem Namen in unterschiedlichen Namespaces verfügt. Eine Klasse mit dem Namen `NN` gibt es beispielsweise als PoS -Tag in `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos`, als Abhängigkeit in `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency` und als Konstituent in `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent`. Werden Typen voll-qualifiziert referenziert, wird eine folgenschwere Verwechslung oder ungewollte Vermischung der Typen innerhalb der Regeln vermieden.

Best Practice

Eine zweite Drools-Regel in Listing 4.19 namens `Add V-attributes to annotated constraints` setzt auf die `ConstraintAnnotation`s der ersten Regel in Listing 4.18 auf der vorherigen Seite auf und versieht diese nachträglich mit zugehörigen Attributen. Diese Zuordnung von Attributen zu ihren Constraints erfolgt durch die Auswertung der Elternknoten der Konstituentendarstellung. Im gewählten Beispiel in Listing 4.17 auf der vorherigen Seite geht es also um das Attribut `heated` aus dem Bereich `(NP (VBN heated) (NN Pool))` der Konstituentendarstellung.

Attributzuordnung

Listing 4.19: NER -Regel zur Erzeugung eines Constraints mit Attribut.

```
1 rule "Add V-attributes to annotated constraints"
2   when
3     parent : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.
         constituent.NP()
```

```
4     v : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V(
          attribute : coveredText)
5     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token (
          pos == v , parent == parent)
6     constraint : ConstraintAnnotation( parent == parent,
          attribute != attribute)
7     eval (isDatatypeAttribute(constraint.getKey()));
8     eval (isAttribute(attribute));
9   then
10    # Insert new Constraint with attribtue
11    insert(createAttributedConstraintAnnotation(constraint,
          attribute));
12
13    # Remove original, unattributed one from Working Memory
14    retract(constraint);
15 end
```

Im Baum wird ersichtlich, dass sich die Zugehörigkeit von `heated` zu `Pool` durch eine Noun Phrase (NP) als Elternknoten ausdrückt. Diese wird in Zeile 3 als `parent` definiert und als Elternknoten für das (anonyme) Token in Zeile 5 gesucht. Existiert nun eine `ConstraintAnnotation` mit dem selben Elternknoten (Zeile 6), ist diese attributierbar (Zeile 8) und handelt es sich bei dem gefundenen Token um ein Attribut (Zeile 9), ist die Regel erfüllt. Sie erzeugt dann eine neue `ConstraintAnnotation` mit dem gefundenen Attribut mit der entsprechenden Hilfsmethode der `ConstraintFactory` (Zeile 12) und zieht die alte `ConstraintAnnotation` ohne Attribut aus dem Working Memory zurück.

Noun Phrase

Dieser beschriebene Ablauf beider Regeln, inklusive dem Hinzufügen und Zurückziehen von `ConstraintAnnotations`, lässt sich im Drools-Audit-Log anschaulich verfolgen (Abbildung 4.13). Regeln sind blau, neue Fakten grün und entfernte Fakten rot markiert.

Drools-Audit-Log

Auch diese Änderungen am Working Memory werden durch den `CasUpdatingEventListener` in der CAS nachgezogen. Nachdem alle Regeln ausgeführt wurden (und die Agenda leer ist) wird die Anfangs in Abschnitt 4.4.6.2 auf Seite 64 angelegte stateful session verworfen und das CAS inklusive der `ConstraintAnnotations` weiterverarbeitet.

Ablauf

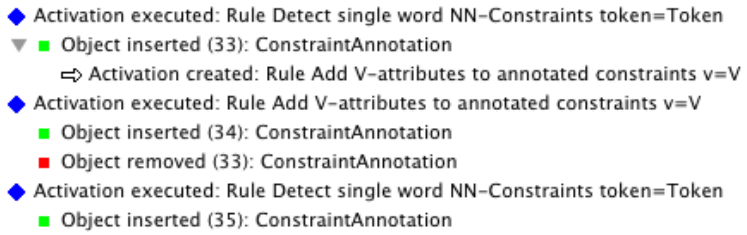

Abbildung 4.13.: Drools-Audit-Log nach Ausführung der beiden Regeln aus Listing 4.18 auf Seite 69 und 4.19 auf Seite 70.

Das Überführen von gegebenen `ConstraintAnnotation`-Objekten in von `Constraint`-Objekte erfolgt, wie in Abschnitt 4.4.4.4 auf Seite 58 beschrieben, durch `UimaPipeline`.

4.5. Deployment

Das Deployment von Query Analysis kann, dank der durchgängigen Verwendung der Maven-Projektdeskriptoren POM, vollständig automatisiert werden und geschieht im Projektkontext bereits automatisch durch die Continuous Integration (CI) -Suite Jenkins. Maven

Weiter liegen benötigte DKPro Abhängigkeiten auf einem öffentlichen Buildserver[28] vor. Dieser kann entweder als Repository in eine lokale POM eingebunden werden[29] oder als Proxy-Repository[30] in vorhandene Repository Management Systeme eingebunden werden. Abhängigkeiten

Insbesondere der Stanford Parser ist sehr rechen- und ressourcenintensiv[31]. Als gute Arbeitsgrundlage hat sich in der Entwicklung ein initialer *Heap-Space*[32] von 1024 Megabyte erwiesen.

[28] `http://zoidberg.ukp.informatik.tu-darmstadt.de/` (letzter Abruf am 2012-08-28).
[29] Der XML-Code für die Verwendung des öffentlichen Buildservers von DKPro findet sich unter `http://code.google.com/p/dkpro-core-asl/wiki/UkpMavenRepository` (letzter Abruf am 2012-08-28)
[30] `http://www.sonatype.com/people/2010/02/nexus-proxy-repositories-video-walkthrough-of-major-features/` (letzter Abruf am 2012-08-28).
[31] `http://nlp.stanford.edu/software/parser-faq.shtml#k` (letzter Abruf am 2012-09-05).
[32] Die Vorgabe des Heap-Space wird über den VM-Parameter `-Xmx1024M` geändert.

Kapitel 5.

Auswertung

In diesem Kapitel wird eine Testumgebung entsprechend der Grundlagen zur Evaluation von IR -Systemen (siehe Abschnitt 3.1.1 auf Seite 16) als Infrastruktur zur Erfassung notwendiger Kennzahlen geschaffen, um zu zeigen, dass das im vorgehenden Kapitel 4 ab Seite 39 entwickelte System allen messbaren Anforderungen (siehe Abschnitt 2 auf Seite 9) genügt.

Für diese Qualitätssicherung wurde ein Framework für die systematische Auswertung entwickelt. Dieses steht als Basisklasse für JUnit-Testcases zur Verfügung, wodurch nicht nur der Entwicklungsprozess unterstützt, sondern die Qualitätssicherung auch in die Infrastruktur (im Fall von TourInDi in die CI -Suite Jenkins) integriert wird.

5.1. Regression-Testframework

Für die systematische Auswertung der Erkennungsleistung wurde ein Regressions-Testframework auf Basis des JUnit-Frameworks[1] entwickelt. Dadurch können nicht nur vorhandene JUnit-Funktionalitäten wie der Ausführung von Tests mitgenutzt werden, das Testframework fügt auf diese Weise auch Regressionstests nahtlos in das von TourInDi genutzte Buildmanagement mittels Maven bzw. der verwendeten Buildinfrastruktur Jenkins ein.

JUnit

Abbildung 5.1 zeigt das Klassendiagramm des Testframeworks, aus dem ersichtlich wird, dass es aus der Klasse `TestDriver` besteht, selbst ein JUnit-Testfall (`junit.framework.TestCase`) ist und als abstrakte Basisklasse für die Implementierung von konkreten Regressionstests zur Verfügung steht.

Basisklasse

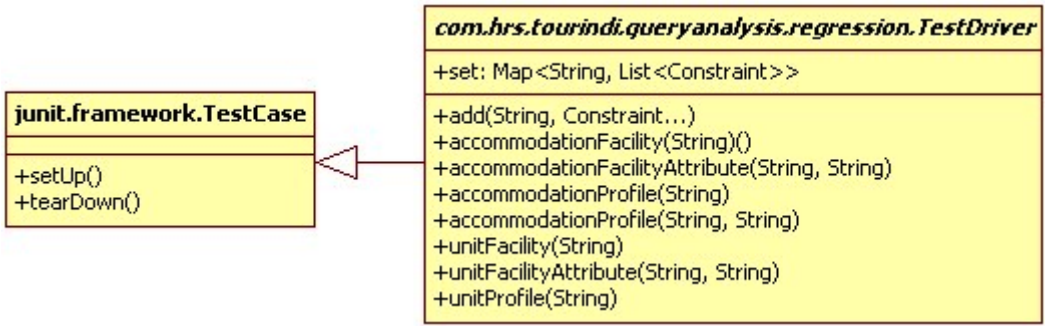

Abbildung 5.1.: UML Klassendiagramm mit den wichtigsten Attributen und Methoden des Testframeworks.

Zentraler Gegenstand ist dabei die Testsammlung (in der Klasse das Attribut `set`), welche für jede zu analysierende, natürlichsprachliche Hotelsuche die dafür erwarteten, zu identifizierenden Constraints (siehe Abschnitt 1.1.1 auf Seite 3) als Einträge beinhaltet. Die vom `TestCase` geerbten Methoden `setUp()` und `tearDown()` werden dabei vom Framework verwendet, um die Ablauflogik eines Regressionstests zu implementieren. Diese beinhaltet die Initialisierung der Analysekomponenten und der Testsammlung im `setUp()` und das schrittweise Analysieren, Auswerten und Protokollieren jedes einzelnen Eintrags der Testsammlung, sowie das Ermitteln abschließender Kennzahlen wie Precision und Recall (siehe Abschnitt 3.1.1 auf Seite 16) im `tearDown()`.

Testsammlung

[1] `http://www.junit.org/` (letzter Abruf am 2012-08-29).

5.1.1. Implementierung von Testfällen

Durch die geschickte Nutzung des JUnit-Frameworks gestaltet sich die Implementierung eines konkreten Testfalls als sehr schlank und ist mit wenigen Lines Of Code (LOC) realisierbar. Listing 5.1 zeigt den vollständigen Programmcode zur Überprüfung der Erkennung eines Hotels mit Pool aus der Anfrage {`Wellness Hotel with Pool`}. wenige LOC

Listing 5.1: Vollständiger Testfall zur Überprüfung der Erkennung eines Hotels mit Pool.

```
1 public class Sample extends TestDriver {
2   public void test() {
3     add("Wellness Hotel with Pool",
          accommodationProfile("Hotel", "Wellness"),
          accommodationFacility("Pool"));
4   }
5 }
```

Die Testklasse erbt von der Superklasse `TestDriver`. Die dadurch vererbte Methode `add(String, Constraints...)` wird für das Hinzufügen neuer Einträge in die Testsammlung verwendet. Deren erstes Argument ist die Hotelsuche in natürlicher Sprache in Form eines `String`s, sowie eine variable Parameterliste, die die zu identifizierenden Constraint-Objekte beinhaltet. Zur schnellen Erzeugung von Constraints bietet die Oberklasse eine Reihe von Konstruktionsmethoden an (siehe Abbildung 5.1). Die Testklasse kann als regulärer JUnit-Test ausgeführt werden; das Framework sorgt automatisch für die korrekte Initialisierung, Analyse und Auswertung. Listing 5.2 zeigt die übliche Ausgabe, die es dabei für jeden analysierten Eintrag in der Testsammlung erzeugt. Hinzufügen neuer Einträge Ausgabe

Listing 5.2: Konsolenausgabe des Testframeworks für einen Testsammlungseintrag in dem Fall, in dem die Menge an erwarteten Constraints und die Menge an tatsächlich erkannten Constraints identisch ist.

```
1 Input was "Wellness Hotel with Pool"
2 Processing call # 0 took 489 ms
3
4  |  Expected constraints  |    Actual constraints  |
5  |------------------------|------------------------|
6  |                 Hotel  |                 Hotel  |
7  |                  Pool  |                  Pool  |
```

Die Ausgabe beinhaltet den analysierten, natürlichsprachlichen Ausdruck `Wellness Hotel with Pool` und die Dauer in Millisekunden, die deren Analyse erforderte.

Vor allem für die schrittweise Verbesserung der NER -Regeln (siehe Abschnitt 4.4.6.5 auf Seite 68) oder für die Problemanalyse stellt der tabellarische Vergleich der Schlüssel von erwarteten und tatsächlichen Constraints wertvolle Informationen dar. Im genannten Beispiel in Listing 5.2 auf der vorherigen Seite wurde jede erwartete Constraint auch tatsächlich gefunden, jedem Schlüssel in der Spalte `Expected constraints` steht folglich der gleiche Schlüssel in der Spalte `Actual constraints` gegenüber. Tabelle

Fehlt ein Constraint entweder bei den erwarteten oder den tatsächlichen Constraints, wird dies durch ein `x` angezeigt. Listing 5.3 zeigt die Konsoleausgabe für einen derartigen Fall an; hier wurde beispielsweise im vorherigen Beispiel der erwartete Constraint `Pool` nicht in der Textanalyse erkannt. Fehlende Constraints

Listing 5.3: Konsolenausgabe des Testframeworks für einen Testsammlungseintrag in dem Fall, in dem die Menge an erwarteten Constraints und die Menge an tatsächlich erkannten Constraints nicht identisch ist.

```
1 Input was "Wellness Hotel with Pool"
2 Processing call # 0 took 489 ms
3
4  |  Expected constraints  |    Actual constraints  |
5  |------------------------|------------------------|
6  |                 Hotel  |                 Hotel  |
7  |                  Pool  |                     x  |
```

Werden zwei Constraints mit identischem Schlüssel sowohl erwartet als auch tatsächlich gefunden und unterscheiden sie sich beide lediglich im Detail (etwa bei ihrem Datentyp oder ihrem Wert), so gibt das Testframework detaillierte Informationen zum Vergleich betroffener Constraints aus. Abbildung 5.4 zeigt dieses Verhalten mit Beispiel der bisherigen Hotelsuchanfrage für den Fall an, indem der erwartete Constraint mit dem Schlüssel `Hotel` und dem Attribut `Wellness` nur in Form eines Constraints mit dem Schlüssel `Hotel` und dem Attribut `true` gefunden wurde - die Erkennung versagte also bei dessen Attribut `Wellness`. Vergleich

Listing 5.4: Konsolenausgabe des Testframeworks für einen Testsammlungseintrag in dem Fall, in dem sich erwartete und erkannte Constraints im Detail unterscheiden.

```
Input was "Wellness Hotel with Pool"
Processing call # 0 took 489 ms

|   Expected constraint   |     Actual constraint   |
|-------------------------|-------------------------|
|                  Hotel  |                       x |
|                      x  |                   Hotel |
|                   Pool  |                    Pool |

Non-unique key: Hotel
      Key:                          Hotel                          Hotel
Datatype:                     ATTRIBUTE                        BOOLEAN
    Value:                       Wellness                           true
    Class:     AccommodationConstraint     AccommodationConstraint
     Kind:                        PROFILE                        PROFILE
UnitType:                           null                           null
```

Mit der Ausgabe `Non-unique key: Hotel` wird angezeigt, dass ein Constraint mit dem Schlüssel `Hotel` sowohl erwartet als auch tatsächlich gefunden wurde, beide Objekte aber sind. Direkt im Anschluss vergleicht das Framework die charakteristischen Werte `Datatype`, `Value`, `Class`, `Kind` und `UnitType` beider betroffener Constraints. So wird schnell ersichtlich, dass sich die Werte für `Datatype` und `Value` unterscheiden, was ein geeigneter Einstiegspunkt für einen menschlichen Tester darstellt, das Problem zu identifizieren.

im Detail unterschiedlich

5.1.2. Interpretation von Analyseergebnissen

Alle der im vorherigen Abschnitt erläuterten Ergebnisse für jeden einzelnen Testsammlungseintrag werden zusammen mit anderen Kennzahlen erfasst und in einer Zusammenfassung übersichtlich dargestellt. Listing A.1 auf Seite I zeigt diese für ein Beispiel mit insgesamt zehn Aufrufen mit 15 beteiligten Constraints. Kennzahlen

Listing 5.5: Konsolenausgabe des Testframeworks zur Testzusammenfassung.

```
1             Debug mode: not active
2       Number of calls:          10
3   Number of documents:          15
4 Average call duration:     444.800
5     Max time per call:         900
6
7       True  Positives:          13
8       False Positives:           0
9       False Negatives:           2
10
11             Precision:     1.00000
12                Recall:    0.866667
13        F measure in %:     92.8571
```

Neben Angaben zur durchschnittlichen Analysedauer werden auch die üblichen Angaben zur Bewertung von IR -Systemen (siehe Abschnitt 3.1.1 auf Seite 16), Precision P (Zeile 11) und Recall R (Zeile 12), sowie deren Verhältnis, dem F-measure F (Zeile 13), gemacht. Entsprechend der Definitionen 3.3 und 3.4 auf Seite 17 können P und R auch mit Hilfe der drei Klassifikationen true positive t_p, false positive f_p und false negative f_n (Zeilen 7-9) für Analyseergebnisse errechnet werden. In Anwendung dessen stellt Tabelle 5.1 die Entscheidungstabelle dar, wie genau die Ergebnisse von Einzeltests diesen Klassifikationen zugeordnet werden. Precision Recall

Tabelle 5.1.: Entscheidungstabelle bei der Interpretation von Analyseergebnissen.

Constraint wurde:		Klassifikation als
erwartet	erkannt	
x	x	true positive t_p
	x	false positive f_p
x		false negative f_n

5.2. Training und Verifikation

Query Analysis wurde in einem *Training* schrittweise verbessert, indem verschiedenartige Testdaten als Hotelsuchanfrage analysiert wurden und bei fehlerhafter Erkennung die zugrundeliegenden NER -Regeln (siehe Abschnitt 4.4.6.5 auf Seite 68) angepasst und ergänzt wurden. Ohne Veränderung der Regeln und mit anderen Testdaten wurde anschließend in der *Verifikation* die Erkennungsleistung überprüft und als von Query Analysis erzielte Erkennungsrate gewertet.

Beispiel-Hotelsuchanfragen

Linguee

Zugrundeliegend für die verwendeten Trainings- und Verifikationsdaten sind Beispiel-Hotelsuchanfragen, die der Autor in seinem Familien- und Freundeskreis in Deutsch zusammengetragen und in die übliche englische Übersetzung der Reisebranche übersetzt hat. Gemäß der Anforderungen wurden hierbei keine besonderen Vorgaben gemacht und keine Eingabesprache und Trenn- oder Satzzeichen vorgegeben (siehe Abschnitt 2.1.2 auf Seite 10). Ergänzend dazu wurden im Training Auszüge des Onlinedienstes Linguee[2] verwendet, um die Erkennungskapazitäten zu vergrößern[3]. Zusätzlich wurde das Training durch spezielle, konstruierte Hotelsuchen ergänzt, um bestimmte Funktionalitäten gezielt zu prüfen. Um zudem der Anforderung an Fehlertoleranz (siehe Abschnitt 2.1.4 auf Seite 11) gerecht zu werden, wurden einige Datensätze dupliziert und mit Tippfehlern versehen. Tabelle 5.2 und 5.3 zeigen auf, wie sich die Trainings- und Verifikationsdatensätze konkret aus den genannten Daten zusammensetzen.

Tabelle 5.2.: Zusammensetzung der Traingsdaten.

Suchanfragen	Anzahl
Menschlich erzeugte Suchen	15
zusätzliche mit Tippfehlern	10
Linguee Auszüge	10
zusätzliche mit Tippfehlern	5
Speziell konstruierte Suchen	11
Trainingsdatensätze	55

Tabelle 5.3.: Zusammensetzung der Verfikationsdaten.

Suchanfragen	Anzahl
Menschlich erzeugte Suchen	50
zusätzliche mit Tippfehlern	21
Linguee Auszüge	0
zusätzliche mit Tippfehlern	0
Speziell konstruierte Suchen	0
Verifikationsdatensätze	71

[2] `http://www.linguee.de/` (letzter Abruf am 2012-09-12).

[3] Es wurde mit dem Dienst nach einer bestimmten, typischen Zimmer- oder Hoteleigenschaft, etwa `double bed` `http://www.linguee.de/deutsch-englisch/search?source=english&query=double+bed` (letzter Abruf am 2012-09-12), gesucht und aus den Textauszügen der Ergebnismenge gezielte Teilausdrücke für das Training verwendet.

Vollständige Codeliste

In der Verifikation wurde von der Prämisse ausgegangen, dass die Codeliste vollständig ist und alle erforderlichen Informationen enthält, um das geforderte Constraint erkennen zu können. Wurde ein Constraint einzig und allein deshalb nicht erkannt, weil er als Eintrag in der Codeliste fehlte, wurde dieser für die Messung sowohl im Training als auch in der Verifikation nachgetragen.

Abdeckung

Die so erzeugten Testdaten für das Training und die Verifikation decken jeweils etwa die Hälfte der Ausdrucksmöglichkeiten der Codeliste ab. Da beide Sammlungen nicht die identischen Codes verwenden, ergibt sich insgesamt eine Codelisten-Abdeckung von 72%.

Regelentwicklung

Ähnlich wie in der Implementierung in Abschnitt 4.4.6.5 auf Seite 68 beschrieben, wurden im Training nach und nach insgesamt 17 Regeln zur Annotation von `ConstraintAnnotations` für verschieden Szenarien entwickelt. Diese werden durch fünf Regeln zum nachträglichen Hinzufügen von Attributen und weitere sechs Regeln für spezielle Sachverhalte ergänzt. Nach Abschluss des Trainings wurden die Regeln unverändert in der Verifikation zur abschließenden Bewertung verwendet. Sie finden sich in Anhang A auf Seite I.

Die detaillierten Erkennungsergebnisse für die angeforderten Accommodation- und Unit-Constraints (siehe Abschnitt 2.1.3 auf Seite 11) des Trainings und der Verifikation finden sich im Anhang B auf Seite VII bzw. im Anhang C auf Seite XV.

5.3. Ergebnisse

Im Folgenden wird gezeigt, dass die entwickelte Software den Anforderungen aus Kapitel 2 auf Seite 9 genügt.

5.3.1. Erkennungsleistung

Mit einer Precision P von 0.95, einem Recall R von 0.91 und einem daraus ergebenen F-measure von 92,56 % erzielt das System in der Verifikation die angeforderte Erkennungsrate (siehe Abschnitt 2.1.7 auf Seite 12). Mit dieser Precision sind in 47 von 50 Fällen die gefundenen Fakten tatsächlich relevant (siehe Abschnitt 3.1.1 auf Seite 17); in 45 von 50 Fällen werden sie auch ausgebeutet (siehe Abschnitt 3.1.1 auf Seite 16). Erkennungsrat

Ein detaillierter Blick auf die Verifikationsergebnisse (siehe Abschnitt C auf Seite XV) lässt zudem Aussagen zur Fehlertoleranz zu: In mehr als drei von vier Fällen hat das Hinzufügen von einem oder mehreren Rechtschreibfehlern keine Auswirkung auf die Erkennungsleistung. Dies entspricht auch den Anforderungen an Realitätsnähe und Fehlertoleranz (siehe Abschnitt 2.1.4 auf Seite 11). Fehler-toleranz

Systematische Fehlerursachen finden sich nur wenige. So lässt sich die in Einzelfällen missglückte Erkennung von Abkürzungen wie {`AI`} für {`all inclusive`} oder {`AC`} für {`Air conditioning`} (Seite XVII bzw. Seite XV) auf ein fehlerhaftes PoS -Tagging als Verb[4] zurückführen. Zudem werden aus Synonymen vollständig zusammengesetzte Entity-Attribut-Beziehungen nicht erkannt. Als Beispiel hierfür lässt sich die Zeichenkette {`deep-level garage`} (Seite XVI) nennen, bei der {`deep-level`} als Synonym für {`underground`} und {`garage`} als Synonym für {`parking`} steht. Abkürzungen zusammen-gesetzte Bezie-hungen

Schwerer wiegt dagegen ein Detail in der fehlerhaften Erkennung der Hotelsuchanfrage {`Hostel with cable television on Route 66`} (Seite XVII). Neben der ausgebliebenen Erkennung von Constraints, die wie oben auf fehlerhaftes PoS -Tagging zurückzuführen sind, deutet vor allem der false positive {`radio = true`} auf ein grund- false positive

[4] Der Parser identifiziert beide Abkürzungen als Verb in der ersten oder zweiten Person, Singular Präsent (`VBP`).

legendes Problem hin: Als ortsbezogene Angabe (siehe Abschnitt 1.1.1 auf Seite 3) ist das Wort {Route} in der ursprünglichen Hotelsuchanfrage nicht Teil der Codeliste und existiert deswegen auch nicht als Eintrag im Wörterbuch der Rechtschreibkorrektur. Im weiteren Verlauf wird es deshalb als Rechtschreibfehler gewertet und mit dem ihm ähnlichsten Wort in der Codeliste ersetzt, was in diesem Fall verherenderweise das Ausstattungsmerkmal {radio} ist. {radio} wird daraufhin als UnitConstraint erkannt und als Ergebnis ausgegeben.

Rechtschreibfehler

Wenngleich dies der einzige gemessene Fall ist, bei dem eine fehlerhafte Korrektur die Erkennungsleistung negativ beeinträchtigt, verweist dies auf eine grundsätzliches, systematisches Problem mit dem Korrekturmechanismus hin. Diese wird im Ausblick (siehe Abschnitt 6.3 auf Seite 89) weiter kommentiert.

systematisches Problem

5.3.2. Erweiterbarkeit

Dank der deskriptiven Beschreibungsart von Drools-Regeln, die das in dieser Arbeit entwickelte System ermöglicht, und einer automatisierten Lademethode können ein oder mehrere in Texterkennungsregeln leicht aktiviert, bearbeitet oder entfernt werden, ohne dass dies Änderungen an anderen Komponenten nach sich zieht. Der generische Charakter der Analysis Engine DroolsRuntimeAnnotator (siehe Abschnitt 4.4.6 auf Seite 62) ermöglicht es zudem, auch für gegenwärtig ungenutzte Feature Structures Drools-Regeln zu entwickeln, ohne dass sich ein Entwickler mit den Details der UIMA befassen muss.

Änderungen

Die Anforderung an Erweiterbarkeit (siehe Abschnitt 2.2 auf Seite 12) gilt damit als erfüllt.

5.3.3. Performance

Mit einer durchschnittlichen Analysezeit von 273 Millisekunden in der Verifikation entspricht die gemessene Arbeitsgeschwindigkeit der entwickelten Komponente der Vorgabe durch die Anforderungen in Abschnitt 2.3 auf Seite 13.

Geschwindigkeit

Wie sich die gesamte Analysezeit verhältnismäßig auf die jeweiligen Analyseschritte verteilt, ist in Abbildung 5.2 im Detail aufgeführt. Dies wurde mit Hilfe des Pipeline-Modus `Mode.LOG_TIMING` (siehe Abschnitt 4.4.4 auf Seite 54) und ausgewählten Suchanfragen aus den Trainings- und Verifikationsdaten ermittelt.

Verteilung pro Schritt

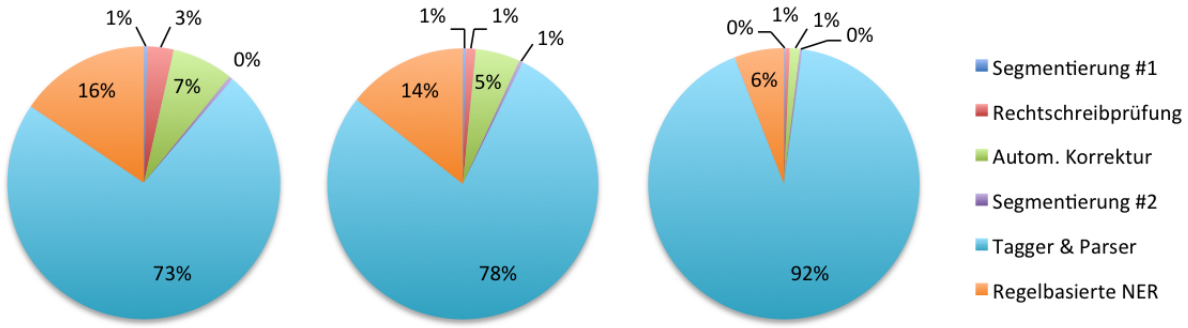

Abbildung 5.2.: Verteilung der Ausführungszeiten pro Verarbeitungsschritt der Pipeline für kurze (links), mittellange (mittig) und lange (rechts) Suchanfragen.

Der mit 73% für einfache Eingaben zeitaufwändigste Verarbeitungsschritt stellt das Tagging und das Parsing dar. Dessen benötigte Zeit steigt mit Länge und Komplexität der Benutzereingabe (im Bild von links nach rechts) weiterhin erheblich auf 92% der Gesamtverarbeitungszeit bei langen Suchanfragen an. Die absolut benötigte Zeit liegt zuletzt bereits im Sekundenbereich.

Parser

Auch der Zeitaufwand der NER -Komponente steigt, da auch diese Komponente mit steigender Eingabekomplexität mehr Daten zu verarbeiten hat. Allerdings steigt dieser hier erheblich langsamer, sodass der verhältnismäßige Anteil an der Gesamtverarbeitungszeit mit zunehmender Eingabekomplexität sinkt. Dagegen sind selbst bei großen Eingaben die beiden Segmentierungsschritte, die Rechtschreibprüfung und die Korrektur verhältnismäßig günstige Operationen und liegen mit ihrer jeweiligen, absoluten Verarbeitungszeit im zweistelligen Millisekundenbereich.

NER

Segmentierung Korrektur

Kapitel 6.

Schluss

In diesem Kapitel werden zunächst verwandte, technische Ansätze, die im Rahmen dieser Arbeit entdeckt wurden, kurz dargestellt und kommentiert. Dem folgt eine Zusammenfassung der Arbeit.

In einem Ausblick werden gesammelte Erfahrungen des Autors dokumentiert und die weitere, künftige Entwicklung der Software skizziert. Abschließend erfolgt eine Bewertung der entwickelten Software anhand konkurrierender Lösungen.

6.1. Related Work

Neben der in dieser Arbeit verwendeten UIMA existieren eine Reihe weiterer Frameworks zur Analyse von unstrukturierten Informationen. Renaud Richardet hat in einer Übersicht einige dieser Werkzeuge zusammengestellt [Ric-2012, Seite 101]; als Platzhirsche lassen sich dabei UIMA und General Architecture for Text Engineering (GATE) bezeichnen. Beide Plattformen sind konzeptionell ähnlich aufgebaut und sind zum Teil sogar funktionell identisch[1] [Str-2009, Seite 14]. Als einzige Textverarbeitungs-Lösung stellt allerdings UIMA mit dem gleichnamigen OASIS-Standard eine standardisierte Lösung dar.

GATE

Aus den selben Beweggründen wie in dieser Arbeit beschreiben Peter Kluegl et al. in [KAP-2008] im Rahmen der Entwicklungsplattform *TextMarker*[2] die Integration einer Rule Engine in UIMA. Auf dieser Grundlage wird eine eigene Information Extraction-Sprache entwickelt und in der UIMA-Pipeline ausgeführt [KAP-2008, Seite 2]. Stattdessen wird in dieser Arbeit die vorhandene Rule Engine Drools integriert und so das ausführen typischer Drools-Regeln ermöglicht.

TextMarker

Das Open Source-Projekt *UIMA Drools*[3] bietet eine Funktionalität, die in ihren Grundzügen der Funktion des in dieser Arbeit entwickelten DroolsRuntimeAnnotator (siehe Abschnitt 4.4.6 auf Seite 62) ähnelt, indem es Teile einer CAS in einer Drools Session verfügbar macht[4]. Im Unterschied zur in dieser Arbeit entwickelten Software reduziert sie sich dabei allerdings auf die annotierten Tokens, statt - wie gezeigt wurde - die CAS systematisch zu durchsuchen und alle Arten von Feature Structures mit Drools verarbeitbar zu machen. Weiter bleiben auch Konfigurationsmöglichkeiten durch UIMA ungenutzt, wodurch keine Anpassungen durch die Pipeline vorgenommen werden können.

[1] Analysis Engines von UIMA und Processing Resources von GATE sind ein Beispiel für die große, funktionelle Ähnlichkeit beider Plattformen [Str-2009, Seite 14].

[2] `http://www.is.informatik.uni-wuerzburg.de/forschung/anwendungen/textmarker/` (letzter Abruf am 2012-09-29).

[3] `https://github.com/celi-uim/uima-drools` (letzter Abruf am 2012-08-06).

[4] Im Quellcode des Annotators wird über alle Tokens iteriert und jedes gefundene Token anschließend in die Session eingefügt: `https://github.com/celi-uim/uima-drools/blob/master/src/main/java/it/celi/uima/drools/DroolsAnnotator.java` (letzter Abruf am 2012-08-06)

6.2. Zusammenfassung

In Zeiten, in denen fast jeder Internetbenutzer seine Informationen mit Google oder ähnlichen Diensten mit nur einer einzigen Freitexteingabe auffindet, entsprechen forumlargestützte Suchmaschinen nicht mehr dem länger dem technischen und benutzerfreundlichen Status quo. Das gilt insbesondere in der Touristikbranche, in der einerseits viel gesucht und andererseits eine große Erwartung an Ergebnisse gestellt wird (siehe Abschnitt 1.2.1 auf Seite 5).

Ziel von TourInDi, einem Forschungsprojekt zwischen dem Touristikportal HRS und der Hochschule Darmstadt, ist es unter anderem, ein neues, alternatives Freitext-Eingabefeld für die bisher Formular-basierten Hotelsuchen zu entwickelen, sodass ein Benutzer seine Hotelsuchanfragen in natürlicher Sprache formulieren und absetzen kann. Dabei ist die technische Komponente names Query Analysis (siehe Abschnitt 1.1 auf Seite 2), die die Erkennung von natürlichsprachlichen Hotelsuchanfragen und deren Überführung als Suchparameter in das projekteigene Datenmodell (siehe Abschnitt 1.1.1 auf Seite 3) leistet, Entwicklungsgegenstand dieser Arbeit. Dabei wird sich in einer ersten Version auf die Erkennung von Hotel- und Zimmereigenschaften beschränkt.

Bei der Erkennung von natürlichsprachlichen Hotelsuchanfragen gelten besondere Bedingungen und Anforderungen an Query Analysis. Aus Sicht eines Benutzers müssen sich Hotelsuchen schnell (siehe Abschnitt 2.3 auf Seite 13) und intuitiv (siehe Abschnitt 2.1.2 auf Seite 10) abwickeln lassen. Zudem wird nicht erwartet, dass Suchanfragen in der Regel vollständig ausformulierte Sätze darstellen, sondern, ähnlich wie bei den Eingaben in Suchmaschinen wie Google (siehe Abschnitt 1.2.2 auf Seite 6), kurze Satzfragmente mit den wesentlichen Informationen ohne korrekte Syntax und ohne korrekte Grammatik sind (siehe Abschnitt 2.1.1 auf Seite 10). Zudem dürfte ein Benutzer seine Eingaben nur flüchtig korrigieren, was insbesondere für das Auftreten von Buchstabendrehern (siehe Abschnitt 2.1.4 auf Seite 11) spricht. Vor allem im Betrieb ist eine leichte Erweiterbarkeit der Erkennungskapazitäten wichtig (siehe Abschnitt 2.2 auf Seite 12). Das korrekte Interpretieren von natürlichsprachlicher Negation (siehe Abschnitt 2.1.6 auf Seite 12) stellt dagegen ein echten Mehrwert gegenüber syntaxbasierten Suchen dar und zeigt im besonderen Maße die Vorzüge einer semantischen Erkennung.

Grundsätzlich basiert die im weiteren Verlauf der Arbeit entwickelte, technische Lösung für die genannten Anforderungen auf detaillierten Kenntnissen sowohl der Informationsverarbeitungsplattform Apache UIMA (siehe Abschnitt 3.3 auf Seite 26) als auch der Rule Engine JBoss Drools (siehe Abschnitt 3.4 auf Seite 36). Es wird dargestellt, dass sich beide Systeme im besonderen Maße als Bausteine einer regelbasierten Lösung für die Named Entity Recognition eignen (siehe Abschnitt 4.1 auf Seite 40). Gemäß geltender Design-Prinzipien insbesondere der UIMA wird weiter Query Analysis als technische Lösung zunächst als konzeptionelle Architektur verallgemeinert erläutert (siehe Abschnitt 4.2 auf Seite 42). Dabei kommen auch verbreitete Disziplinen des Natural Language Processings (siehe Abschnitt 3.2 auf Seite 20) zum Einsatz. Anschließend wird die konkrete Implementierung von Query Analysis in Java mit Hilfe entwicklernaher UIMA-Aufsätze wie uimaFIT und DKPro beschrieben (siehe Abschnitt 4.4 auf Seite 48). In Kombination aus etablierten Drittanbieterkomponenten wie dem Stanford-Parser sowie exklusiven, neuartigen Eigenentwicklungen entsteht so eine leistungstarke Grundlage für das deskriptive Beschreiben von Texterkennungsregeln mit typischen Drools-Regeln.

Die Bewertung (siehe Abschnitt 5 auf Seite 73) der Erkennungskapazitäten erfolgt mit Hilfe der üblichen Evaluationsmethoden des Information Retrievals (siehe Abschnitt 3.1.1 auf Seite 16). Dafür wird zusätzlich ein Framework entwickelt (siehe Abschnitt 5.1 auf Seite 74), das die von UIMA bereitgestellten Werkzeuge sinnvoll ergänzt und so die Entwicklungsarbeit an den Erkennungskapazitäten von Query Analysis vereinfacht und unterstützt. Dieses schafft zudem die technische Grundlage für die systematische Erfassung und Auswertung der üblichen Maße Precision und Recall.

Im Training und in der Verifikation (siehe Abschnitt 5.2 auf Seite 79) der Lösung kommen verschiedenartige Hotelsuchanfragen zum Einsatz, die sich insbesondere in der Verifikation aus realistischen Beispielen zusammensetzen. Durch diese Beispiele wird abschließend gezeigt, dass die entwickelte Komponente der Texterkennung alle Anforderungen deutlich erfüllt (siehe Abschnitt 5.3 auf Seite 81). Damit entspricht die Erkennungsleistung von Query Analysis dank dieser Arbeit nicht nur den Voraussetzungen für den kommenden Release von TourInDi, auch das in dieser Arbeit entwickelte Konzept kommt im Rahmen des Projekts als leistungsstarke, standard-nahe und erweiterungsfähige Architektur zur Anwendung. Das legt auch den geeigneten Grundstein für künftige, professionelle Weiterentwicklungen der Texterkennung in TourInDi.

6.3. Ausblick

Die in dieser Arbeit erfolgreich entwickelte Software stellt die Grundlage der Texterkennung von TourInDi Release 3 dar. In diesem Rahmen wurden auch wesentliche systematische Probleme von TourInDi identifiziert, etwa die schwache Typisierung des Datenmodells, was sinnlose Ausdrücke wie etwa {`heated hotel`} zulässt, oder die fehlerträchtige und -anfällige Implementierung zum Einlesen der Codeliste, die sich als Ursache für viele Probleme herausstellte. Zum Zeitpunkt der Fertigstellung dieser Arbeit zeichnen sich allerdings bereits Lösungen für diese grundsätzlichen Belange ab.

schwache Typisierung

Wie in der Architektur dargestellt (siehe Abschnitt 4.2 auf Seite 43), ist die Vorgängerimplementierung der Texterkennung mit Hilfe des Online-Dienstes EVA erhalten geblieben. Diese könnte genutzt werden, um sie mit der in dieser Arbeit entwickelten Texterkennung parallel zu schalten und so von den Erkennungsleistungen beider System profitieren zu können. Alternativ könnte fallbezogen EVA nur dann eingesetzt werden, wenn zuvor keine auswertbaren Constraints erkannt wurden.

EVA

Es bieten sich langfristig einige Optimierungsmöglichkeiten an, wenngleich die Software gegenwärtig allen gegebenen Anforderungen, insbesondere den zeitlichen, genügt. So könnten beispielsweise die Feature Structures, die mit Drools verarbeitet werden, vor ihrem Einfügen in die Drools-Session so gefiltert werden, sodass nur tatsächlich in den Regeln verwendete Feature Structures auch als Drools-Fakten vorliegen. Damit könnte möglicherweise die benötigte Zeit für die regelbasierte NER , welche immerhin den zweitzeitaufwändigsten Analyseschritt darstellt (siehe Abschnitt 5.3.3 auf Seite 82), reduziert werden.

Drools-Session

Die zwingende, zweifache Ausführung der Segmentierung erscheint zunächst überflüssig, insbesondere dann, wenn keine Korrektur des Sofa vorgenommen wurde und so tatsächlich die gleichen Operationen auf den gleichen Daten zweifach ausgeführt werden. Die sehr geringe Verarbeitungszeit[5] (siehe Abschnitt 5.3.3 auf Seite 83) der Segmentierung macht im Rahmen dieser Arbeit allerdings keine weiteren Schritte zur Verhinderung doppelter Segmentierungen erforderlich. Sollte sich die Segmentierung künftig aufgrund von Änderungen der Projektanforderungen als wesentlich teurere Operation herausstellen,

Zweifache Segmentierung

[5] Selbst für große Eingaben, bei denen der Verarbeitungsschritt für das Tagging und Parsing über 3 Sekunden beanspruchte, erforderte die Segmentierung lediglich 7 Millisekunden.

empfiehlt sich die Implementierung eines Flow Controllers [Her-2009a, Seite 42], um detaillierten Einfluss auf die Art und Weise der Ausführung der Pipeline zu nehmen. Flow Controller

Um den Komfort für den Regelentwickler weiter zu erhöhen, könnte im weiteren Projektgeschehen der Einsatz von benutzerdefinierten Drools-Operatoren[6] erwogen werden. Damit bietet es sich an, die Funktionen der in Java geschriebenen Hilfsmethoden der `ConstraintFactory` (siehe Abschnitt 4.4.3 auf Seite 52) als native Drools-Operatoren einsetzen zu können. Statt also eines `eval (isAttribute(attribute))` im LHS-Teil einer Regel würde ein `attribute isAttribute true` möglich. Lohnenswert könnte auch die native Bereitstellung von Methoden der Klasse `org.uimafit.util.JCasUtil` sein - diese bietet Hilfsmethoden für eine Reihe typischer, CAS-naher Operationen an, etwa zum Abrufen von Vorgänger- und Nachfolger-Annotationen. Beides hätte erwartungsgemäß lediglich Einfluss auf die Lesbarkeit der Regeln und nicht auf deren Performance. Drools-Operatoren JCasUtil

Die in dieser Arbeit entwickelte Pipeline verfolgt die Strategie, eine möglichst korrekte Eingabe an die Tagger- und Parser-Komponente zu übergeben, sodass erzeugte PoS -Tags, Lemmata und die Parsing-Ergebnisse als Eingabe für die NER -Regeln so korrekt und vollständig wie möglich sind. Dazu wird, wie beschrieben (siehe Abschnitt 4.2 auf Seite 46), in einem ersten Analyseprozess eine automatisierte Korrektur durchgeführt. Diese Korrektur kann jedoch nur zufriedenstellend funktionieren, wenn das zugrundeliegende Wörterbuch möglichst vollständig ist und alle bekannten Phrasen enthält. Dies mag wie in dieser Arbeit für Accommodation- und UnitConstraints der Fall sein, kann sich jedoch für die noch einzuführenden Person-, Money-, Location- und TimeConstraints grundlegend anders gestalten. Schlimmstenfalls wird eine korrekt-geschriebene Phrase mit etwas völlig anderem ersetzt, weil sie nicht im Wörterbuch steht (bzw. nicht im Wörterbuch stehen kann), sie aber Ähnlichkeiten zu einem anderen Eintrag im Wörterbuch aufweist. Um zu vermeiden, dass wichtige Phrasen mit falschen Korrekturen überschrieben werden, können hier zwei verschiedene, andere Strategien gewählt werden: Grundsätzlich kann das der Korrektur zugrundeliegende Wörterbuch sich nicht nur aus den Bestandteilen der Codeliste zusammensetzen, sondern aus allen Wörtern der Eingabesprache. Dies würde falsche Korrekturen vermindern, würde aber auch dazu führen, dass der Fokus auf das domänenspezifische Vokabular der Codeliste bei der Korrektur verloren geht: beispielsweise würde {`Polo`} dann nicht mehr länger als falsch-geschriebener {`Pool`} Korrektur-problem

[6] `http://members.inode.at/w.laun/drools/CustomOperatorHowTo.html` (letzter Abruf am 2012-10-02).

identifiziert und korrigiert werden, sondern als korrektes, englisches Wort unkorrigiert belassen bleiben, auch wenn das Wort als Eingabe für eine Hotelsuchanfrage gar keinen Sinn macht. Ein zweiter Ansatz, mit den falschen Korrekturen der Rechtschreibprüfung umzugehen, sieht vor, Rechtschreibfehler wie bisher mit dem selben, aus der Codeliste generierten Wörterbuch zwar zu identifizieren, diese dann jedoch nicht automatisch zu korrigieren, sondern als zusätzliche Fakten in die NER -Regeln mit einfließen zu lassen. So könnte eine Art Fallback-Szenario implementiert werden: Scheitern alle anderen Regeln an einer Phrase, so könnten durch die Rechtschreibprüfung vorgeschlagene Korrekturen bei der NER mit betrachtet werden.

Benutzer-interaction

Mit dieser Strategie könnte die Rechtschreibkorrektur auch als Teil der Graphical User Interface (GUI) Benutzerinteraktion erlauben, sodass der Benutzer auf eventuelle Rechtschreibfehler aufmerksam gemacht und bei der Korrektur beteiligt wird.

teures Parsing

Dem oben genannten Ansatz für bestmögliche Parser-Ergebnisse ist es auch geschuldet, dass das Parsing der zeitaufwändigste Arbeitsschritt der Pipeline ist. Es gibt erheblich schnellere Parsing-Verfahren, etwa das Shallow-Parsing [JM-2000, Seite 385]. Dieses hat allerdings nicht Anspruch, die komplette Eingabe tatsächlich auch parsen zu können. Folglich könnte mit diesem schnelleren Parsing-Verfahren nicht mehr unbedingt die Satzstruktur in den NER -Regeln ausgewertet werden, wodurch wertvolle Informationen bei der Analyse nicht mehr zur Verfügung ständen.

Mehr-sprachig-keit

Gemäß der Anforderung (siehe Abschnitt 2.1.5 auf Seite 11) ist die entwickelte Pipeline ausschließlich zur Auswertung englischer Eingaben geeignet. Soll hier Mehrsprachigkeit hergestellt werden, sind mehre Ansätze erwägenswert. Der UIMA-typische Weg sieht das Erkennen der Sprache in Form einer Analysis Engine als ersten Verarbeitungsschritt vor. Anhand der so erkannten Sprache werden dann erforderliche Ressourcen wie Wörterbücher oder Grammatiken aller folgenden Analysis Engines (re-)initialisiert. Diese Initialisierung kann erhebliche Zeit beanspruchen, weshalb in DKPro ein Zwischenspeicher für derartige Dateien implementiert ist, der alle Ressourcen für eine bestimmte Sprache vorhält. Hierbei ist zu evaluieren, ob der Zwischenspeicher auch die Ressourcen mehrerer Sprachen vorhalten kann. Andernfalls würden Ressourcen mit jedem Sprachwechsel der Benutzereingabe reinitialisiert werden, was zu einer deutlich verlängerten Verarbeitungszeit führte. Weiter müssten wahrscheinlich auch die NER -Regeln angepasst werden, da diese üblicherweise sehr sprachspezifisch sind [Lan-2006, Seite 4].

Alternativ könnte evaluiert werden, ob die Mehrsprachigkeit nicht mit einer einfachen Übersetzung der Benutzereingabe ins Englische mit geringerem Aufwand hergestellt werden kann. Keine Option stellt dagegen die Verwendung mehrsprachiger Synonyme in der Codeliste dar: Zwar würde damit zwar der Grundstein zur Erkennung mehrsprachiger NEs gelegt werden, Tagger und Parser arbeiteten jedoch weiterhin sprachspezifisch und würden für Eingaben anderer Sprachen keine brauchbaren Ergebnisse erzeugen.

Übersetzung

Der Autor hat die entwickelte Analysis Engine DroolsRuntimeAnnotator (siehe Abschnitt 4.4.6 auf Seite 62) aufgrund ihrer generischen Einsetzbarkeit zur Veröffentlichung bei den Projektverantwortlichen von DKPro vorgestellt. Sie wird künftig als Open-Source-Software unter auf GitHub[7] der UIMA Benutzergemeinschaft zur Verfügung gestellt.

Veröffentlichung

[7] `https://github.com/heussd/droolsruntimeannotator` (letzter Abruf am 2012-10-07).

6.4. Bewertung

Zunächst bietet sich eine vergleichende Bewertung der entwickelten Software und des von ihr abgelösten Drittanbieter-Diensts EVA an. Die Erkennungskapazitäten sind bei EVA vom Dienstanbieter vorgegeben und können dadurch nicht an die projektspezifischen Anforderungen angepasst werden. Dagegen stellt die verwendete Codeliste (siehe Abschnitt 1.1.1 auf Seite 3) in Kombination mit den NER -Regeln einen prinzipiell sehr erweiterungsfähigen und flexiblen Ansatz dar, der zudem auch sehr punktuell und deskriptiv an künftige Anforderungen angepasst werden kann. Auch in der Performance ist EVA der hier entwickelten Lösung unterlegen, denn der Dienst ist als Client-Server-Architektur realisiert und setzt daher bei jeder Hotelsuchanfrage einen Aufruf mit Hypertext Transfer Protocol (HTTP) ab, der prinzipbedingt nicht zeitunaufwändig ist: in einer Testanwendung dauerte der Aufruf von EVA rund eine Sekunde - in dieser Zeit konnte die gleiche Anfrage fast sieben mal mit der in dieser Arbeit entwickelten Texterkennung verarbeitet werden. Dieser Unterschied erklärt sich sicherlich vor allem an einem Mehraufwand durch den Transport über HTTP, aber auch die bereits genannten, an die Projektanforderungen angepassten Erkennungskapazitäten dürften für die Zeitersparnis verantwortlich sein. Insgesamt kann festgestellt werden, dass die Ablösung von EVA im Rahmen des TourInDi-Projekts eine sinnvolle und vorteilhafte Entscheidung war. Die entstandene Architektur stellt auch für künftige Weiterentwicklungen eine solide und flexible Arbeitsbasis dar.

EVA

Mit TextMarker[8] existiert eine IE -Plattform, der, wie der Software aus dieser Arbeit, Apache UIMA zugrunde liegt. TextMarker bietet ausgefeilte Möglichkeiten zur Textanalyse und -verarbeitung und bietet Lösungen für viele übliche, linguistische Probleme. Damit leistet es deutlich mehr die in dieser Arbeit entwickelte Software. Allerdings müssen Anwendungen mit TextMarker in der eigenen, gleichnamigen Programmier- und Regelsprache[9] implementiert werden. Nach in der Literatur vertretenen Meinung [App-1999, Seite 165f] stellt dies im Rahmen des TourInDi-Projekts einen Nachteil dar, da die Regelsprache Drools bereits im Projekt in der Komponente Recommendation (siehe Abschnitt 1.1 auf Seite 2) vertreten ist - in diesem Fall wird prinzipiell die Sprache bevorzugt, für die bereits Entwickler-Knowhow projektintern vorliegt (siehe Abschnitt 3.1.2 auf Seite 19).

Text-Marker

[8] http://www.is.informatik.uni-wuerzburg.de/forschung/anwendungen/textmarker/ (letzter Abruf am 2012-09-29).

[9] http://tmwiki.informatik.uni-wuerzburg.de/Wiki.jsp?page=NewSyntax (letzter Abruf am 2012-10-05).

GATE

Gegenüber GATE, einer konkurrierenden Architektur für IE -Anwendungen, weist UIMA weniger stark ausgeprägte, grafische Werkzeuge für die Entwicklung auf. Dieses Defizit wird mit in dieser Arbeit entwickelten Software zumindest teilweise entkräftet: die Klasse `UimaPipeline` bietet beispielsweise einige Convenience-Methoden für die Konstruktion von Pipelines und die Deserialisierung der CAS. Damit können etwa CAS-Objekte einfach mit den üblichen UIMA-Werkzeugen (siehe Abschnitt 4.3 auf Seite 47) untersucht werden, ohne dass tiefgreifendes Wissen über UIMA erforderlich ist. Weiter verfügt die Klasse `TestDriver` zwar nicht über eine GUI, aber über eine Text-based User Interface (TUI), die Informationen zur Erkennungsleistung tabellarisch darstellt (siehe Abschnitt 5.2 auf Seite 75). Auch dies senkt die Hürden in die Texterkennung mit UIMA, wenngleich dies projektspezifische Lösungen sind.

Das gegenüber GATE sehr starke Typsystem von UIMA erweist sich im Zusammenspiel mit Drools als Vorteil, denn es erleichtert den Umgang mit gleichermaßen stark typisierten Fakten innerhalb der Regeln (siehe Abschnitt 4.11 auf Seite 66).

Typsystem-unabhängigkeit

Im Vergleich zu vielen anderen Analysis Engines verfolgt der DroolsRuntimeAnnotator (siehe Abschnitt 4.4.6 auf Seite 62) prinzipiell einen sehr flexiblen und selten zu findenden Ansatz, indem er vollständig unabhängig vom verwendeten Typsystem der UIMA-Pipeline ist. Das erlaubt nicht nur projektintern große Freiheiten, an welcher Stelle die Analysis Engine für welche Funktion eingesetzt wird - durch seine Unabhängigkeit kann er auch in anderen Projekten vielseitig eingesetzt werden. Durch die angesprochene Bereitstellung von Methoden der Klasse `JCasUtil` (siehe Abschnitt 6.3 auf Seite 89) hat der Ansatz weiter das Potenzial, sehr mächtig zu werden und stellt auch für die UIMA-Benutzergemeinschaft jenseits von DKPro eine einzigartige und neuartige Art und Weise dar, CAS-Objekte anhand sehr kurzer Regeln zu verarbeiten.

Anhang A.

Entwickelte NER-Regeln

Listing A.1: Drools - Entwickelte NER -Regeln.

```
1  rule "NN Single Word Constraints"
2    when
3      nn : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN()
4      token : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( entity : coveredText, pos
           == nn )
5      eval (hasEntity(entity))
6    then
7      insertLogical(createConstraintAnnotation(token, entity));
8  end
9
10 rule "NN Synonym Single Word Constraints"
11   when
12     nn : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN()
13     token : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( entity : coveredText, pos
           == nn )
14     eval (!hasEntity(entity))
15     eval (getSynonymedKey(entity) != null)
16   then
17     insertLogical(createConstraintAnnotation(token, getSynonymedKey(entity)));
18 end
19
20 rule "NN Lemma Single Word Constraint"
21   when
22     nn : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN()
23     lemma : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Lemma (lemmaText : value)
24
25     token : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( entity : coveredText, pos
           == nn, lemma == lemma )
26     eval (! hasEntity(entity))
27     eval (hasEntity(lemmaText))
28   then
29     insertLogical(createConstraintAnnotation(token, lemmaText));
30 end
31
32 rule "V Single Word Constraints"
33   when
34     v : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V( PosValue == "VBD" )
35     token : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( entity : coveredText, pos
           == v )
36     eval (hasEntity(entity))
37   then
38     insertLogical(createConstraintAnnotation(token, entity));
39 end
40
41 rule "VBZ Single Word Constraints"
```

```
42    when
43      v : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V( PosValue == "VBZ" )
44      token : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( entity : coveredText, pos
            == v )
45      eval (hasEntity(entity))
46    then
47      insertLogical(createConstraintAnnotation(token, entity));
48  end
49
50  rule "JJ Single Word Constraints"
51    when
52      adj : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADJ()
53      token : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( entity : coveredText, pos
            == adj )
54      eval (hasEntity(entity))
55    then
56      insertLogical(createConstraintAnnotation(token, entity));
57  end
58
59  rule "JJ Synonym Single Word Constraints"
60    when
61      adj : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADJ()
62      token : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( entity : coveredText, pos
            == adj )
63      eval (!hasEntity(entity))
64      eval (getSynonymedKey(entity) != null)
65    then
66      insertLogical(createConstraintAnnotation(token, getSynonymedKey(entity)));
67  end
68
69  rule "NP Single Word Constraints"
70    when
71      np : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NP()
72      token : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( entity : coveredText, pos
            == np )
73      eval (hasEntity(entity))
74    then
75      insertLogical(createConstraintAnnotation(token, entity));
76  end
77
78  rule "NP Synonym Single Word Constraints"
79    when
80      np : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NP()
81      token : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( entity : coveredText, pos
            == np )
82      eval (!hasEntity(entity))
83      eval (getSynonymedKey(entity) != null)
84    then
85      insertLogical(createConstraintAnnotation(token, getSynonymedKey(entity)));
86  end
87
88  rule "NN-NN Word Constraints"
89    when
90      #parent : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP()
91
92      nn1 : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN( part1 : coveredText)
93      de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent : parent , pos == nn1)
94
95      nn2 : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN( part2 : coveredText)
96      de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent == parent , pos == nn2)
97
98      eval (hasEntity(part1 + " " + part2))
99    then
100     insertLogical(createConstraintAnnotation(parent, part1 + " " + part2));
101 end
102
103 rule "NN-NN Synonym Word Constraints"
104   when
105     nn1 : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN( part1 : coveredText)
106     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent : parent , pos == nn1)
107
108     nn2 : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN( part2 : coveredText)
```

```
109     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent == parent , pos == nn2)
110
111     eval (! hasEntity(part1 + " " + part2))
112     eval (hasEntity(getSynonymedKey(part1 + " " + part2)))
113   then
114     #System.out.println("HERE: " + part1 + " " + part2);
115     #System.out.println(hasEntity(getSynonymedKey(part1 + " " + part2)));
116     insertLogical(createConstraintAnnotation(parent, getSynonymedKey(part1 + " " + part2)));
117 end
118
119 rule "NP-NN Synonym Word Constraints"
120   when
121     nn1 : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NP( part1 : coveredText)
122     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent : parent , pos == nn1)
123
124     nn2 : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN( part2 : coveredText)
125     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent == parent , pos == nn2)
126
127     eval (! hasEntity(part1 + " " + part2))
128     eval (hasEntity(getSynonymedKey(part1 + " " + part2)))
129   then
130     #System.out.println("HERE: " + part1 + " " + part2);
131     #System.out.println(hasEntity(getSynonymedKey(part1 + " " + part2)));
132     insertLogical(createConstraintAnnotation(parent, getSynonymedKey(part1 + " " + part2)));
133 end
134
135 rule "JJ-NN Word Constraints"
136   when
137     parent : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP()
138
139     nn : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN( part2 : coveredText)
140     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent == parent , pos == nn)
141
142     jj : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADJ( part1 : coveredText)
143     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent == parent , pos == jj)
144
145     eval (hasEntity(part1 + " " + part2))
146   then
147     insertLogical(createConstraintAnnotation(parent, part1 + " " + part2));
148 end
149
150 rule "JJ-NN Synonym Word Constraints"
151   when
152     parent : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP()
153
154     nn : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN( part2 : coveredText)
155     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent == parent , pos == nn)
156
157     jj : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADJ( part1 : coveredText)
158     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent == parent , pos == jj)
159
160     eval (! hasEntity(part1 + " " + part2))
161     eval (hasEntity(getSynonymedKey(part1 + " " + part2)))
162   then
163     insertLogical(createConstraintAnnotation(parent, getSynonymedKey(part1 + " " + part2)));
164 end
165
166 rule "ART-NN Synonym Word Constraints"
167   when
168     art : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ART( part1 : coveredText)
169     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent : parent , pos == art)
170
171     nn : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN( part2 : coveredText)
172     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent == parent , pos == nn)
173
174     eval (! hasEntity(part1 + " " + part2))
175     eval (hasEntity(getSynonymedKey(part1 + " " + part2)))
176   then
177     #System.out.println("HERE: " + part1 + " " + part2);
178     #System.out.println(hasEntity(getSynonymedKey(part1 + " " + part2)));
179     insertLogical(createConstraintAnnotation(parent, getSynonymedKey(part1 + " " + part2)));
180 end
```

```
181
182 rule "V Attributes"
183   when
184     parent : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP()
185     v : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V( attribute : coveredText)
186     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token (pos == v, parent == parent)
187     constraint : ConstraintAnnotation( parent == parent, attribute != attribute, attribute ==
          null)
188     eval (isDatatypeAttribute(constraint.getKey()));
189     eval (isAttribute(attribute));
190   then
191     # Insert new Constraint with attribtue
192     insert(createAttributedConstraintAnnotation(constraint, attribute));
193
194     # Remove original, unattributed one from Working Memory
195     retract(constraint);
196 end
197
198 rule "ADJ Attributes"
199   when
200     parent : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP()
201     adj : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADJ( attribute : coveredText)
202     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token (pos == adj, parent == parent)
203     constraint : ConstraintAnnotation( parent == parent, attribute != attribute, attribute ==
          null)
204     eval (isDatatypeAttribute(constraint.getKey()));
205     eval (isAttribute(attribute));
206   then
207     # Insert new Constraint with attribtue
208     insert(createAttributedConstraintAnnotation(constraint, attribute));
209
210     # Remove original, unattributed one from Working Memory
211     retract(constraint);
212 end
213
214 rule "NP Attributes"
215   when
216     parent : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP()
217     np : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NP( attribute : coveredText)
218     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token (pos == np, parent == parent)
219     constraint : ConstraintAnnotation( parent == parent, attribute != attribute, attribute ==
          null)
220     eval (isDatatypeAttribute(constraint.getKey()));
221     eval (isAttribute(attribute));
222   then
223     # Insert new Constraint with attribtue
224     insert(createAttributedConstraintAnnotation(constraint, attribute));
225
226     # Remove original, unattributed one from Working Memory
227     retract(constraint);
228 end
229
230 rule "CARD Attributes"
231   when
232     parent : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP();
233
234     cd : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.CARD( attribute : toInt(coveredText)
          );
235     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token (pos == cd, parent == parent)
236
237     constraint : ConstraintAnnotation( parent == parent, attribute != attribute, attribute ==
          null)
238   then
239     insert(createAttributedConstraintAnnotation(constraint, attribute));
240
241     # Remove original, unattributed one from Working Memory
242     retract(constraint);
243 end
244
245 rule "NN Attributes"
246   when
247     nn : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN( attribute : coveredText)
```

```
248     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token (pos == nn, parent : parent)
249     constraint : ConstraintAnnotation( parent == parent, attribute != attribute, attribute ==
            null)
250     eval (isDatatypeAttribute(constraint.getKey()));
251     eval (isAttribute(attribute));
252   then
253     # Insert new Constraint with attribtue
254     insert(createAttributedConstraintAnnotation(constraint, attribute));
255
256     # Remove original, unattributed one from Working Memory
257     #retract(constraint);
258 end
259
260 rule "NP Multiword Expression" # (NP (NNP Air) (NNP conditioning)))
261   when
262     # children > 1 prevents detection of (NP (NN pool))
263     np : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP ( entity : coveredText ,
            children.size > 1)
264
265     eval (hasEntity(entity))
266   then
267     insertLogical(createConstraintAnnotation(np, entity));
268 end
269
270 rule "NP Synonym Multiword Expression" # (NP (NNP Air) (NNP conditioning)))
271   when
272     np : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP ( entity : coveredText ,
            children.size > 1)
273
274     eval (! hasEntity(entity))
275     eval (getSynonymedKey(entity) != null)
276   then
277     insertLogical(createConstraintAnnotation(np, getSynonymedKey(entity)));
278
279 end
280
281 rule "Remove annotated constraints with a preceded 'no'"
282   when
283     np : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP()
284     constraint : ConstraintAnnotation( parent == np )
285
286     art : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ART (PosValue == "DT" )
287     lemma : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Lemma ( value == "no" )
288     token : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( parent == np , pos == art
            , lemma == lemma)
289   then
290     #System.out.println("Deleting annotation " + constraint.getKey() + " because of " + lemma.
            getValue());
291     retract(constraint);
292 end
293
294 rule "Missunderstood PUNCs"
295   when
296     # This punctuation is way to long for a simple sentence delimiter:
297     p : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PUNC ( end - begin > 3, pBegin :
            begin )
298     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( wrongPunctuation : coveredText ,
            parent : parent , pos == p )
299
300     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( end + 1 == pBegin ,
            precedingToken : coveredText )
301
302     eval (hasEntity(precedingToken + " " + wrongPunctuation))
303   then
304     insertLogical(createConstraintAnnotation(parent, precedingToken + " " + wrongPunctuation));
305 end
306
307 rule "Remove unattributed rooms"
308   salience -100
309
310   when
311     constraint : ConstraintAnnotation( key == "rooms" , attribute == null)
```

```
312   then
313     System.out.println("Removing unattributed room");
314     retract(constraint);
315 end
316
317 rule "Assign lonely attributes to already assigned accomodations"
318   salience -100
319   no-loop true
320
321   when
322     token : de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( hasAttribute(coveredText)
            , attribute : coveredText )
323     lonelyConstraint : ConstraintAnnotation(key : key, getAccomodations() contains key ,
          attribute != attribute)
324
325     # There is no other accomodation
326     not (ConstraintAnnotation(key != key, getAccomodations() contains key , attribute !=
          attribute))
327
328     not (ConstraintAnnotation(attribute == attribute))
329     not (ConstraintAnnotation(begin <= token.begin && end >= token.end))
330   then
331     insert(createAttributedConstraintAnnotation(createConstraintAnnotation(token,
          lonelyConstraint.getKey()), attribute));
332 end
333
334 rule "Retract non-attributed versions of attributed Constraints"
335   salience -1000
336
337   when
338     unattributedConstraint : ConstraintAnnotation(key : key, attribute == null)
339     ConstraintAnnotation(key == key, attribute != null)
340   then
341     retract(unattributedConstraint);
342 end
343
344 rule "NP-embedded attribute"
345   salience -10
346   # (NP
347   # (NP (NN family))
348     #   (NP (NN apartment)))
349   when
350     np : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP(children.size == 1, parent
          : parent )
351     de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token ( attribute : coveredText ,
          hasAttribute(attribute) , parent == np )
352
353     np2 : de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP(children.size == 1, parent
           == parent )
354     constraint : ConstraintAnnotation( parent == np2 , attribute == null )
355
356   then
357     insert(createAttributedConstraintAnnotation(constraint, attribute));
358
359 end
360
361 rule "Resolve child- and petfriendly"
362   salience -100
363
364   when
365     de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.ROOT ( coveredText matches (".*
          child-? and pet-?friendly.*") )
366     ConstraintAnnotation( key : key , attribute == "pet-friendly" , parent : parent )
367   then
368     insert(createAttributedConstraintAnnotation(createConstraintAnnotation(parent, key), "child-
          friendly"));
369 end
```

Anhang B.

Trainingssuchanfragen

Suchanfrage	Erwartete Constraints	Erkannte Constraints
[1]		
2 ro0ms	`(AC) rooms = 2`	`(AC) rooms = 2`
2 rooms	`(AC) rooms = 2`	`(AC) rooms = 2`

2 single beds, the otter with a double bed; two bathrom; a doubl sofa bed in the sitting room; the kitchn comprises an oven.	`(UC) single bed = 2`
	`(UC) double bed = true`
	`(UC) bathroom = true`
	`(UC) sofa = true`
	`(UC) kitchen = true`
		`(UC) single bed = 2`
		`(UC) double bed = true`
		`(UC) bathroom = true`
		`(UC) sofa = true`
		`(UC) kitchen = true`

`(AC)` = `AccommodationConstraint`, `(UC)` = `UnitConstraint`, `x` = keine Entsprechung als erwarteter oder erkannter Constraint

[1] Leere Suchanfrage

Tabelle B.1.: Erkennungsergebnisse für Hotelsuchanfragen des Trainings.

(fortgesetzt)

3 nights, Hamburg, 1 Person, no beakfast, with air conditioning	(UC) air conditioning = true	(UC) air conditioning = true
3 nights, Hamburg, 1 Person, no breakfast, with air conditioning	(UC) air conditioning = true	(UC) air conditioning = true
3 nights, Hamburg, 1 Person, with breakfast, with air conditioning	(AC) breakfast = true (UC) air conditioning = true	(AC) breakfast = true (UC) air conditioning = true
Best Western Hotel in Marburg in the form of twin bed rooms and a sauna landscape with pool.	(AC) hotel = true (AC) pool = true (UC) twin bed = true (AC) sauna = true	(AC) hotel = true (AC) pool = true (UC) twin bed = true (AC) sauna = true
Business hotel with golf location for two person and twin bed	(AC) business hotel = true (AC) golf = true (UC) twin bed = true (AC) hotel = business	(AC) business hotel = true (AC) golf = true (UC) twin bed = true (AC) hotel = business
Business hotel with pool, sauna in London	(AC) business hotel = true (AC) pool = true (AC) sauna = true (AC) hotel = business	(AC) business hotel = true (AC) pool = true (AC) sauna = true (AC) hotel = business
City hotel,casino, all inclusive	(AC) city hotel = true (AC) casino = true (AC) all inclusive = true (AC) hotel = true	(AC) city hotel = true (AC) casino = true (AC) all inclusive = true (AC) hotel = true
City hotel,casino, all incusive	(AC) city hotel = true (AC) casino = true (AC) all inclusive = true (AC) hotel = true	(AC) city hotel = true (AC) casino = true (AC) all inclusive = true (AC) hotel = true
Hot3l and train, 30.7.12 - 5.8.12, 1 person for 100€ at max	(AC) hotel = true	(AC) hotel = true

(AC) = AccommodationConstraint, (UC) = UnitConstraint, x = keine Entsprechung als erwarteter oder erkannter Constraint

(fortgesetzt)

Hotel and train, 30.7.12 - 5.8.12, 1 person for 100€ at max	(AC) hotel = true	(AC) hotel = true
Hotel from 22.05.12 to 23.05.12, 1 Person, Minibar, air conditioning	(AC) hotel = true (UC) minibar = true (UC) air conditioning = true	(AC) hotel = true (UC) minibar = true (UC) air conditioning = true
Hotel with no Pool	(AC) hotel = true	(AC) hotel = true
Hotel with pool and a double bed	(AC) hotel = true (AC) pool = true (UC) double bed = true	(AC) hotel = true (AC) pool = true (UC) double bed = true
Hotel with pool and double bed	(AC) hotel = true (AC) pool = true (UC) double bed = true	(AC) hotel = true (AC) pool = true (UC) double bed = true
Sommer sports hotel; sports trainer; sports for teenager; family apartment	(AC) hotel = sports (AC) sports trainer = true (AC) apartment = family	(AC) hotel = sports (AC) sports trainer = true (AC) apartment = family
Spacious double bed room with bathtub, WC, balcony, safe, telephone, radio and colour TV	(UC) double bed = true (UC) bathtub = true (UC) balcony = true (UC) tv = true (UC) telephone = true (UC) safe = true (UC) radio = true (UC) restroom = true	(UC) double bed = true (UC) bathtub = true (UC) balcony = true (UC) tv = true (UC) telephone = true (UC) safe = true (UC) radio = true (UC) restroom = true
Spacious double bedroom with bath-tub, toilett, balconi, save, teleppone, radio and colour TV	(UC) double bed = true (UC) bathtub = true (UC) balcony = true (UC) tv = true (UC) telephone = true	(UC) double bed = true (UC) bathtub = true (UC) balcony = true (UC) tv = true (UC) telephone = true

(AC) = AccommodationConstraint, (UC) = UnitConstraint, x = keine Entsprechung als erwarteter oder erkannter Constraint

(fortgesetzt)

	(UC) safe = true (UC) radio = true (UC) restroom = true	(UC) safe = true (UC) radio = true (UC) restroom = true
Wellnes vacation in Mecklenburg-Vorpommern	(AC) wellness = true	(AC) wellness = true
Wellness Hotel, 20 persons, half-board, petfriendly	(AC) wellness = true (AC) half board = true (AC) hotel = pet-friendly	(AC) wellness = true (AC) half board = true (AC) hotel = pet-friendly
apartment consists of a foyer with the kitchen, the dining room and one extra bed, the bedroom with a twin bed and one extra bed for one person, and the bathroom with a shower	(AC) apartment = true (UC) bathroom = true (UC) twin bed = true (UC) shower = true (UC) kitchen = true (AC) dining = true	(AC) apartment = true (UC) bathroom = true (UC) twin bed = true (UC) shower = true (UC) kitchen = true (AC) dining = true
apartment for 2 persons sharing, bedroom with twin bed and 2 sofas, desk, walk-in closet, balcony, bathroom with bathtub and separate WC.	(AC) apartment = true (UC) twin bed = true (UC) bathtub = true (UC) bathroom = true (UC) sofa = true (UC) restroom = true (UC) balcony = true	(AC) apartment = true (UC) twin bed = true (UC) bathtub = true (UC) bathroom = true (UC) sofa = true (UC) restroom = true (UC) balcony = true
child and pet-friendly Sports Hotel with wifi in the rooms, with air conditioning, underground parking, Portuguese-speaking personnel, swimming pool, sauna, fitness center, TV, balcony, double bed for 173€/night	(AC) sports hotel = true (UC) internet access = true (UC) air conditioning = true (AC) parking = underground (AC) pool = true (AC) sauna = true (AC) fitness center = true	(AC) sports hotel = true (UC) internet access = true (UC) air conditioning = true (AC) parking = underground (AC) pool = true (AC) sauna = true (AC) fitness center = true

(AC) = AccommodationConstraint, (UC) = UnitConstraint, x = keine Entsprechung als erwarteter oder erkannter Constraint

(fortgesetzt)

	(UC) tv = true (UC) balcony = true (UC) double bed = true (AC) hotel = pet-friendly (AC) hotel = sports (AC) hotel = child-friendly	(UC) tv = true (UC) balcony = true (UC) double bed = true (AC) hotel = pet-friendly (AC) hotel = sports (AC) hotel = child-friendly
double bed, Berlin, 2 Persons, air-condition, parking area	(UC) double bed = true (UC) air conditioning = true (AC) parking = true	(UC) double bed = true (UC) air conditioning = true (AC) parking = true
double bed, Berlin, 2 Persons, air-condition, parking lot	(UC) double bed = true (UC) air conditioning = true (AC) parking = true	(UC) double bed = true (UC) air conditioning = true (AC) parking = true
heated balcony	(UC) balcony = true	(UC) balcony = true
hetaed pole	(AC) pool = heated	(AC) pool = heated
hot3l w1t4 p00l a4d 4 d8ubl3 b3d	(AC) hotel = true (AC) pool = true (UC) double bed = true	(AC) hotel = true (AC) pool = true (UC) double bed = true
hotel, child- and pet-friendly	(AC) hotel = child-friendly (AC) hotel = pet-friendly	(AC) hotel = child-friendly (AC) hotel = pet-friendly
junior suites are equipped with a double bed, a comfortable sofa bed, WC and shower, hairdryer, cable TV, safe, telephone, balcony, towels and make-up mirror.	(UC) double bed = true (UC) sofa = true (UC) restroom = true (UC) shower = true (UC) hairdryer = true (UC) tv = true (UC) safe = true (UC) telephone = true (UC) balcony = true	(UC) double bed = true (UC) sofa = true (UC) restroom = true (UC) shower = true (UC) hairdryer = true (UC) tv = true (UC) safe = true (UC) telephone = true (UC) balcony = true

(AC) = AccommodationConstraint, (UC) = UnitConstraint, x = keine Entsprechung als erwarteter oder erkannter Constraint

(fortgesetzt)

	(UC) towels = true (UC) makeup mirror = true (AC) suites = true	(UC) towels = true (UC) makeup mirror = true (AC) suites = true
kitchen equipped with dishwasher, microwave oven, four glass-ceramic hotplates, refrigerator, SAT TV with a sofa (that becomes a double bed), bathroom with shower stall, washing machine, towels and hairdryer.	(UC) kitchen = true (UC) tv = true (UC) sofa = true (UC) bathroom = true (UC) shower = true (UC) hairdryer = true (UC) double bed = true (UC) towels = true	(UC) kitchen = true (UC) tv = true (UC) sofa = true (UC) bathroom = true (UC) shower = true (UC) hairdryer = true (UC) double bed = true (UC) towels = true
large bal-cony with a wonderful view, kitchen with pantry, bedroom with double bed, bathroom with shower/toilet and extra toilet.	(UC) balcony = true (UC) kitchen = true (UC) restroom = true (UC) bathroom = true (UC) double bed = true	(UC) balcony = true (UC) kitchen = true (UC) restroom = true (UC) bathroom = true (UC) double bed = true
large blacony with a wonderful view, kitchen with pantry, bedrom with double bd, bathroom with shower/toilet and extra toilet.	(UC) balcony = true (UC) kitchen = true (UC) restroom = true (UC) bathroom = true (UC) double bed = true	(UC) balcony = true (UC) kitchen = true (UC) restroom = true (UC) bathroom = true (UC) double bed = true
multipurpose hall with stage, expositions and congress rooms, tennis courts, indoor and outdoor swiming polo, ice skating rink, curling hall and fitness area	(AC) tennis court = true (AC) pool = true (AC) meeting rooms = true (AC) fitness center = true (AC) skating rink = true (AC) stage = true	(AC) tennis court = true (AC) pool = true (AC) meeting rooms = true (AC) fitness center = true (AC) skating rink = true (AC) stage = true
multipurpose hall with stage, expositions	(AC) tennis court = true	(AC) tennis court = true

(AC) = AccommodationConstraint, (UC) = UnitConstraint, x = keine Entsprechung als erwarteter oder erkannter Constraint

(fortgesetzt)

and congress rooms, tennis courts, indoor and outdoor swimming pool, ice skating rink, curling hall and fitness area	(AC) pool = true (AC) meeting rooms = true (AC) fitness center = true (AC) skating rink = true (AC) stage = true	(AC) pool = true (AC) meeting rooms = true (AC) fitness center = true (AC) skating rink = true (AC) stage = true
single - tennis court - balcony - swiming bath	(AC) tennis court = true (UC) balcony = true (AC) pool = true	(AC) tennis court = true (UC) balcony = true (AC) pool = true
single - tennis court - balcony - swimming bath	(AC) tennis court = true (UC) balcony = true (AC) pool = true	(AC) tennis court = true (UC) balcony = true (AC) pool = true
twin-bed rooms, a small room and a large room with a TV, where a group of people	(UC) twin bed = true (UC) tv = true	(UC) twin bed = true (UC) tv = true
twnn-bed rooms, a small romo and a large room with a TTV, where a group of people	(UC) twin bed = true (UC) tv = true	(UC) twin bed = true (UC) tv = true
two rooms	(AC) rooms = 2	(AC) rooms = 2
two single beds, the other with a double bed; two bathrooms; a double sofa bed in the sitting room; the kitchen comprises an oven.	(UC) single bed = 2 (UC) double bed = true (UC) bathroom = true (UC) sofa = true (UC) kitchen = true	(UC) single bed = 2 (UC) double bed = true (UC) bathroom = true (UC) sofa = true (UC) kitchen = true
wellness hotel, 20 persons, half board, pet-friendly	(AC) wellness = true (AC) half board = true (AC) hotel = pet-friendly	(AC) wellness = true (AC) half board = true (AC) hotel = pet-friendly

(AC) = AccommodationConstraint, (UC) = UnitConstraint, x = keine Entsprechung als erwarteter oder erkannter Constraint

Listing B.1: Auswertungsergebnisse der Trainingssuchanfragen.

```
1        Number of calls:         48
2    Number of documents:        167
3  Average call duration:    702.208
4      Max time per call:    1000.00
5          Min f measure:    1.00000
6
7        True  Positives:        167
8        False Positives:          0
9        False Negatives:          0
10
11             Precision:    1.00000
12                Recall:    1.00000
13         F measure in %:    100.000
```

Anhang C.

Verifikationssuchanfragen

Suchanfrage	Erwartete Constraints	Erkannte Constraints
1 person, berlin city, AC +breakfast	`(AC) air conditioning = true` `(AC) breakfast = true`	`x` `(AC) breakfast = true`

1 person, stuttgart, SB+ AC+ minibar and sauna area	`(UC) single bed = true`
	`(UC) air conditioning = true`
	`(UC) minibar = true`
	`(AC) sauna = true`
		`(UC) single bed = true`
		`(UC) air conditioning = true`
		`(UC) minibar = true`
		`(AC) sauna = true`

1 person, stuttgart, SB+ AC+ minibr and suna area	`(UC) single bed = true`
	`(UC) air conditioning = true`
	`(UC) minibar = true`
	`(AC) sauna = true`
		`(UC) single bed = true`
		`(UC) air conditioning = true`
		`(UC) minibar = true`
		`(AC) sauna = true`

`(AC)` = `AccommodationConstraint`, `(UC)` = `UnitConstraint`, `x` = keine Entsprechung als erwarteter oder erkannter Constraint

Tabelle C.1.: Erkennungsergebnisse für Hotelsuchanfragen der Verifikation.

(fortgesetzt)

1 week, munich in munich, +sauna, +breakfast, +parking	(AC) sauna = true (AC) breakfast = true (AC) parking = true	(AC) sauna = true (AC) breakfast = true (AC) parking = true
2 adults + 2 children, hotel at chiemsee with polo and large outside areas	(AC) hotel = true (AC) pool = true	(AC) hotel = true (AC) pool = true
2 adults + 2 children, hotel at chiemsee with pool and large outside areas	(AC) hotel = true (AC) pool = true	(AC) hotel = true (AC) pool = true
2 pers., Frankfurt, near station, with deep-level garage	(AC) parking = underground x	x (AC) parking = true
22.9.12 - 25.9.12, Hotel Berlin Mitte, 1 person, purchase on account	(AC) hotel = true	(AC) hotel = true
3 nights in frankfurt, singel romo for 20 persons, conerence room +break	(UC) single bed = true (AC) breakfast = true (AC) conference facilities = true	(UC) single bed = true (AC) breakfast = true (AC) conference facilities = true
3 nights in frankfurt, single room for 20 persons, conference room +breakf.	(UC) single bed = true (AC) breakfast = true (AC) conference facilities = true	(UC) single bed = true (AC) breakfast = true (AC) conference facilities = true
4 stars hotel in paris	(AC) hotel = true	(AC) hotel = true
Busines hotel+ 50persons+ confeernce facilities+ free paking	(AC) conference facilities = true (AC) free parking = true (AC) parking = free (AC) hotel = business	(AC) conference facilities = true (AC) free parking = true (AC) parking = free (AC) hotel = business
Business hotel+ 50persons+ conference facilities+ free parking	(AC) conference facilities = true (AC) free parking = true	(AC) conference facilities = true (AC) free parking = true

(AC) = AccommodationConstraint, (UC) = UnitConstraint, x = keine Entsprechung als erwarteter oder erkannter Constraint

(fortgesetzt)

	(AC) parking = free (AC) hotel = business	(AC) parking = free (AC) hotel = business
Camping with 3 children and sea to fihsing	(AC) fishing = true	(AC) fishing = true
Camping with 3 children and sea to fishing	(AC) fishing = true	(AC) fishing = true
City hotel, casino, ai	(AC) all inclusive = true (AC) hotel = true (AC) casino = true (AC) city hotel = true	x (AC) hotel = true (AC) casino = true (AC) city hotel = true
Ciy hotel, caino, inclusive	(AC) all inclusive = true (AC) hotel = true (AC) casino = true (AC) city hotel = true	x (AC) hotel = true (AC) casino = true (AC) city hotel = true
France, double bed room, balcony, quiet location, air conditioning and safe	(UC) double bed = true (UC) balcony = true (UC) air conditioning = true (UC) safe = true	(UC) double bed = true (UC) balcony = true (UC) air conditioning = true (UC) safe = true
H0t31 Lake Constance 26.7.12 - 30.7.12., 2 person + dog	(AC) hotel = true	x
HRS, Munich, 2 adults + 2 children, swimming bath + sauna	(AC) pool = true (AC) sauna = true	(AC) pool = true (AC) sauna = true
HRS, Munich, 2 adults + 2 children, swimming bath +suana	(AC) pool = true (AC) sauna = true	(AC) pool = true (AC) sauna = true
Hostel with cable television on Route 66	(AC) hostel = true (UC) tv = true x	x x (UC) radio = true
Hotel Lake Constance 26.7.12 - 30.7.12.	(AC) hotel = true	(AC) hotel = true

(AC) = AccommodationConstraint, (UC) = UnitConstraint, x = keine Entsprechung als erwarteter oder erkannter Constraint

(fortgesetzt)

2 person + dog		
Hotel, Munich, air conditioning, incl. breakfast	(AC) hotel = true (UC) air conditioning = true (AC) breakfast = true	(AC) hotel = true (UC) air conditioning = true (AC) breakfast = true
Hotel, Munich, air condtinoign, incl. breakfast	(AC) hotel = true (UC) air conditioning = true (AC) breakfast = true	(AC) hotel = true x (AC) breakfast = true
Hotel, hamburg, 2C + 2A, swiming bath, doubel bed	(AC) pool = true (UC) double bed = true (AC) hotel = true x	(AC) pool = true (UC) double bed = true (AC) hotel = true (UC) restroom = true
Hotel, hamburg, 2C + 2A, swimming bath, double bed	(AC) pool = true (UC) double bed = true (AC) hotel = true x	(AC) pool = true (UC) double bed = true (AC) hotel = true (UC) restroom = true
Majorca, 2 persons, double room, all inclusive	(UC) double bed = true (AC) all inclusive = true	(UC) double bed = true (AC) all inclusive = true
Majorca, hotel all-inclusive, quited location, 4 persons with air conditioning	(UC) air conditioning = true (AC) all inclusive = true (AC) hotel = true	(UC) air conditioning = true (AC) all inclusive = true (AC) hotel = true
Majorca, hotel allinulcsive, quited location, 4 persons with air condioning	(UC) air conditioning = true (AC) all inclusive = true (AC) hotel = true	x x (AC) hotel = true
Pool, sauna and massage	(AC) pool = true (AC) sauna = true (AC) massage = true	(AC) pool = true (AC) sauna = true (AC) massage = true
Romantic hotel*massage*thermal spring*queen bed	(AC) thermal spring = true (AC) hotel = true	(AC) thermal spring = true (AC) hotel = true

(AC) = AccommodationConstraint, (UC) = UnitConstraint, x = keine Entsprechung als erwarteter oder erkannter Constraint

(fortgesetzt)

	(AC) massage = true	(AC) massage = true
Romantic hotel*message*termal spring*queen bed	(AC) thermal spring = true (AC) hotel = true (AC) massage = true	(AC) thermal spring = true (AC) hotel = true (AC) massage = true
Skiing holiday, hotel (doubl room), 2 person no breakfast, incl. ski pass	(AC) hotel = true (UC) double bed = true x	(AC) hotel = true (UC) double bed = true (AC) skiing = true
Sommer sports hotel; diving; fishing; indoor circle	(AC) hotel = sports (AC) diving = true (AC) fishing = true x	(AC) hotel = sports (AC) diving = true (AC) fishing = true (AC) hotel = indoor
Sommer sports hotel; divng; fisihng; indoor circle	(AC) hotel = sports (AC) diving = true (AC) fishing = true x	(AC) hotel = sports (AC) diving = true (AC) fishing = true (AC) hotel = indoor
businessman - overnight stay only - safe - underground garage - wine bar	(AC) parking = underground (UC) safe = true (AC) bar = true	(AC) parking = underground (UC) safe = true (AC) bar = true
businessman - overnight stay only - save - underground gaarge - winebar	(AC) parking = underground (UC) safe = true (AC) bar = true	(AC) parking = underground (UC) safe = true x
convention 3 days - conference room with internet connection, beamer - no overnight stay but with feeding of the guests	(AC) conference facilities = true (UC) internet access = true	(AC) conference facilities = true (UC) internet access = true
convention 3 days - confernce room with internt connection, beamer - no overnight stay but with feeding of the guests	(AC) conference facilities = true (UC) internet access = true	(AC) conference facilities = true (UC) internet access = true
double room with shower & balcony	(UC) double bed = true	x

(AC) = AccommodationConstraint, (UC) = UnitConstraint, x = keine Entsprechung als erwarteter oder erkannter Constraint

(fortgesetzt)

	(UC) shower = true (UC) balcony = true	(UC) shower = true (UC) balcony = true
family with 2 children and dog - childcare - half board	(AC) childcare = true (AC) half board = true	(AC) childcare = true (AC) half board = true
female group - massage treatments - cofefe and cake - tea dance	(AC) massage = true (AC) dance floor = true (AC) coffee bar = true	(AC) massage = true (AC) dance floor = true (AC) coffee bar = true
female group - massage treatments - coffee and cake - tea dance	(AC) massage = true (AC) dance floor = true (AC) coffee bar = true	(AC) massage = true (AC) dance floor = true (AC) coffee bar = true
good dining	(AC) dining = true	(AC) dining = true
hamburg, hotel direkt at the station, 15 min footpath to the city, double room max 56euro	(AC) hotel = true (UC) double bed = true	(AC) hotel = true (UC) double bed = true
hotel for pensioners, Palatinate, dining room and wellness area	(AC) wellness = true (AC) dining = true (AC) hotel = true	(AC) wellness = true (AC) dining = true (AC) hotel = true
hotel munich, 3 stars, near Allianz-Arena	(AC) hotel = true	(AC) hotel = true
hotel, italy, oceanfront, 4 pers. child- and pet-friendly	(AC) hotel = child-friendly (AC) hotel = pet-friendly	(AC) hotel = child-friendly (AC) hotel = pet-friendly
hotl, italy, oceanfront, 4 pers. child- and petfrendly	(AC) hotel = child-friendly (AC) hotel = pet-friendly	(AC) hotel = child-friendly (AC) hotel = pet-friendly
munich, suite, 2 pers. + breakfast	(AC) suites = true (AC) breakfast = true	(AC) suites = true (AC) breakfast = true
munich, sute, 2 pers. + breekfast	(AC) suites = true (AC) breakfast = true	x (AC) breakfast = true
non-smoker - vegetarian - full board - cure treatment	(AC) full board = true (AC) vegetarian = true	(AC) full board = true (AC) vegetarian = true

(AC) = AccommodationConstraint, (UC) = UnitConstraint, x = keine Entsprechung als erwarteter oder erkannter Constraint

(fortgesetzt)

	(UC) nonsmoking = true	x
pensioner couple - wheelchair user - overnight with breakfast	(AC) wheelchair access = true (AC) breakfast = true	(AC) wheelchair access = true (AC) breakfast = true
pensioner couple - whelchair user - overnight with brekfst	(AC) wheelchair access = true (AC) breakfast = true	(AC) wheelchair access = true (AC) breakfast = true
romantic weekend for two - canopy bed - romantic dinner	(UC) canopy bed = true (AC) dining = true	(UC) canopy bed = true (AC) dining = true
romantic weekend for two - canpy bed - romantic dinenr	(UC) canopy bed = true (AC) dining = true	(UC) canopy bed = true (AC) dining = true
single - wellness - allergy sufferer - without car	(AC) wellness = true	(AC) wellness = true
skier - gaudi evening - ingel - regional cuisine	(AC) skiing = true x	x (AC) casino = true
sports club - half board - bowling alley	(AC) bowling alley = true (AC) half board = true (AC) sports facility = true	(AC) bowling alley = true (AC) half board = true (AC) sports facility = true
theater group - training room - lighting - stage	(AC) stage = true (AC) lighting = true	(AC) stage = true (AC) lighting = true
wedding guests - five course dinner - wedding cake - music with dance - overnight with breakfast	(AC) wedding = true (AC) dining = true (AC) breakfast = true (AC) dance floor = true	(AC) wedding = true (AC) dining = true (AC) breakfast = true (AC) dance floor = true
wedding guests - five course dinner - wedidng cake - music with dance - overnight with braekfast	(AC) wedding = true (AC) dining = true (AC) breakfast = true (AC) dance floor = true	(AC) wedding = true (AC) dining = true (AC) breakfast = true (AC) dance floor = true
winter sports hotel with snow boarding and ski	(AC) hotel = sports (AC) skiing = true	(AC) hotel = sports (AC) skiing = true

(AC) = AccommodationConstraint, (UC) = UnitConstraint, x = keine Entsprechung als erwarteter oder erkannter Constraint

(fortgesetzt)

	`(AC) snow boarding = true`	`(AC) snow boarding = true`

`(AC)` = `AccommodationConstraint`, `(UC)` = `UnitConstraint`, `x` = keine Entsprechung als erwarteter oder erkannter Constraint

Listing C.1: Auswertungsergebnisse der Verifikationssuchanfragen.

```
1        Number of calls:         71
2    Number of documents:        158
3  Average call duration:    273.169
4      Max time per call:    500.000
5          Min f measure:   0.900000
6
7        True  Positives:        143
8        False Positives:          8
9        False Negatives:         15
10
11             Precision:   0.947020
12                Recall:   0.905063
13         F measure in %:    92.5566
```

Stichwortverzeichnis

Literaturverzeichnis

[Apa-2011a] Apache UIMA Development Community: *UIMA Overview & SDK Setup*. http://uima.apache.org/d/uimaj-2.4.0/overview_and_setup.pdf. erstellt im Dezember 2011, zuletzt besucht am 08.05.2012

[Apa-2011b] Apache UIMA Development Community: *UIMA References*. http://uima.apache.org/d/uimaj-2.4.0/references.pdf. erstellt im Dezember 2011, zuletzt besucht am 19.08.2012

[Apa-2011c] Apache UIMA Development Community: *UIMA Tutorial and Developers ' Guides*. http://uima.apache.org/d/uimaj-2.4.0/tutorials_and_users_guides.pdf. erstellt im Dezember 2011, zuletzt besucht am 28.05.2012

[App-1999] Appelt, Douglas E.: Introduction to information extraction. In: *AI Communications* 12 (1999), Nr. 3/199, S. 161–172. http://iospress.metapress.com/index/hqdwyfkjpm6b1g3a.pdf, zuletzt besucht am 25.09.2012

[AZ-2012] Aggarwal, Charu C. ; Zhai, ChengXiang ; Aggarwal, Charu C. (Hrsg.) ; Zhai, ChengXiang (Hrsg.): *Mining Text Data*. Boston, MA : Springer New York, 2012. – DOI 10.1007/978–1–4614–3223–4. http://dx.doi.org/10.1007/978-1-4614-3223-4. – ISBN 978–1–4614–3222–7

[Cha-2001] Chang, Chia-Hui: *Information Retrieval and Extraction*. http://www.csie.ncu.edu.tw/~chia/Course/IR/IR2001/Chapter1In

tro.ppt. erstellt im September 2001, zuletzt besucht am 04.10.2012

[FL-2004] Ferrucci, David ; Lally, Adam: UIMA: an architectural approach to unstructured information processing in the corporate research environment. In: *Natural Language Engineering* 10 (2004), September, Nr. 3-4, S. 327–348. http://dx.doi.org/10.1017/S1351324904003523. – DOI 10.1017/S1351324904003523. – ISSN 1351–3249

[FS-2006] Feldman, Ronen ; Sanger, James: *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data.* Cambridge University Press, 2006. – ISBN 978–0–521–83657–9

[GfK-2011] GfK & nurago: *Suchverhalten in Deutschland.* http://full-value-of-search.de/ pdf/ Suchverhalten%20in%20Deutschland%20-%20Eine%20Analyse%20von%20Suchmaschinen-Nutzern.pdf. erstellt im März 2011, zuletzt besucht am 26.09.2012

[GMM-2007] Gurevych, Iryna ; Mühlhäuser, Max ; Müller, Christof ; Steimle, Jürgen ; Weimer, Markus ; Zesch, Torsten: Darmstadt Knowledge Processing Repository Based on UIMA. In: *Proceedings of the First Workshop on Unstructured Information Management Architecture at Biannual Conference of the Society for Computational Linguistics and Language Technology.* Tübingen, Germany, April 2007 http://www.ukp.tu-darmstadt.de/fileadmin/user_upload/Group_UKP/publikationen/2007/gldv-uima-ukp.pdf, zuletzt besucht am 26.09.2012

[Gre-2010] Greifeneder, Horst: *Erfolgreiches Suchmaschinen-Marketing.* 2. Auflage. Wiesbaden : Gabler Verlag, 2010. – ISBN 978–3–8349–1802–4

[Gur-2009] Gurevych, Iryna: *DKPro - Darmstadt Knowledge Processing Software Repository.* http://weblicht.sfs.uni-tuebingen.de/LRTSummit/Ressourcen4/Gurevych_Sprachressourcengipfel_DKPro.ppt. erstellt im May 2009, zuletzt besucht am 06.08.2012

[Her-2009a] HERTA, Christian: *Einführung in Apache UIMA*. http://www.christianherta.de/lehre/informationRetrieval/introduction-uima.pdf. erstellt im Juni 2009, zuletzt besucht am 28.05.2012

[Her-2009b] HERTA, Christian: *Suchtechnologien und Information Retrieval Lernziele Inhalt der Vorlesung Was ist Suche bzw . Information Retrieval Grundlegende Begri e.* http://www.christianherta.de/lehre/informationRetrieval/einfuehrung-suche-informationretrieval.pdf. erstellt im April 2009, zuletzt besucht am 05.28.2012

[Ins-2010] INSTITUT FÜR SOFTWARE-ERGONOMIE UND USABILITY AG: *Die Reaktionszeiten von Websites.* http://www.usability.ch/bn/news/alertbox/detail/die-reaktionszeiten-von-websites.html. erstellt im Mai 2010, zuletzt besucht am 26.09.2012

[JM-2000] JURAFSKY, Daniel S. ; MARTIN, James H.: *Speech and Language Processing.* New Jersey, United Stated of America : Prentice Hall, 2000. – ISBN 0–13–095069–6

[KAP-2008] KLUEGL, Peter ; ATZMUELLER, Martin ; PUPPE, Frank: Integrating the Rule-Based IE Component TextMarker into UIMA. In: BAUMEISTER, Joachim (Hrsg.) ; ATZMUELLER, Martin (Hrsg.): *LWA-2008 (Special Track on Information Retrieval)*, 2008 http://ki.informatik.uni-wuerzburg.de/papers/pkluegl/2008-LWA-UIMATextMarker.pdf, zuletzt besucht am 07.06.2012, S. 73–77

[Kru-1995] KRUCHTEN, Philippe: Architectural Blueprints — The "4+1" View Model of Software Architecture. In: *IEEE Software* 12 (1995), November, Nr. 6, 42–50. http://www.scribd.com/doc/19632538/The-41-View-Model-of-Architecture, zuletzt besucht am 26.09.2012

[Lan-2006] LANG, Joel: *Named Entity Recognition.* http://www.aifb.uni-karlsruhe.de/Lehre/Sommer2006/kdtm/stuff/NER_Ausarbeitung.pdf. erstellt im Juli 2006, zuletzt besucht am 26.09.2012

[Luc-2005] Luckhardt, Heinz-Dirk: *Virtuelles Handbuch Informationswissenschaft: Information Retrieval*. http://www.uni-kiel.de/medinfo/material/kurs_ws05/5/basics_02.pdf. erstellt im Oktober 2005, zuletzt besucht am 26.09.2012

[LVN-2009] Lally, Adam ; Verspoor, Karin ; Nyberg, Eric: Unstructured Information Management Architecture (UIMA) Version 1.0 / OASIS. erstellt im Mai 2009. http://docs.oasis-open.org/uima/v1.0/uima-v1.0.html, zuletzt besucht am 26.09.2012. 2009. – Forschungsbericht

[MRS-2009] Manning, Christopher D. ; Raghavan, Prabhakar ; Schütze, Hinrich: *An Introduction to Information Retrieval*. New York, United States of America : Cambridge University Press, 2009 (c)

[OB-2009] Ogren, Philip ; Bethard, Steven: Building Test Suites for {UIMA} Components. In: *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing (SETQA-NLP 2009)*. Boulder, Colorado : Association for Computational Linguistics, 2009 http://www.aclweb.org/anthology-new/W/W09/W09-1501.pdf, zuletzt besucht am 26.09.2012, S. 1–4

[Ogr-2012] Ogren, Philip: *uimaFIT User Documentation*. https://code.google.com/p/uimafit/wiki/Documentation. erstellt im Juli 2012, zuletzt besucht am 26.09.2012

[Ric-2012] Richardet, Renaud: *NLP tools and UIMA architecture*. http://icwww.epfl.ch/~chappeli/coling/slides/uima.pdf. erstellt im Mai 2012, zuletzt besucht am 26.09.2012

[Rus-2012] Russell, Daniel: *Power Searching with Google - Lesson 3.4: OR and quotes*. https://docs.google.com/document/d/1loso0KnKc35mDUQO9rsO-lGKh9EmArwDf2-0idLUTXY/edit. erstellt im Juli 2012, zuletzt besucht am 26.09.2012

[Str-2009] Strötgen, Jannik: *UTEMPL - Aufbau und Evaluierung einer UIMA basierten Textmining Pipeline für biomedizinische Literatur*. Heidelberg, Ruprecht-Karls-Universität, Diplomarbeit, März 2009. http://dbs.ifi.uni-heidelberg.de/fileadmin/Team/jannik/publications/magisterthesis_stroetgen.pdf, zuletzt besucht am 26.09.2012

[The-2012] The JBoss Drools Team: *Drools Expert User Guide*. http://docs.jboss.org/drools/release/5.4.0.Final/drools-expert-docs/pdf/drools-expert-docs.pdf. erstellt im Mai 2012, zuletzt besucht am 26.09.2012

[Uni-2010] University of Sheffield: *GATE Information Extraction*. http://gate.ac.uk/ie/. erstellt im Dezember 2010, zuletzt besucht am 04.10.2012

[Wik-2012] Wikipedia: *UIMA — Wikipedia, The Free Encyclopedia*. Online. http://en.wikipedia.org/w/index.php?title=UIMA&oldid=490189192. erstellt im Juli 2012, zuletzt besucht am 26.09.2012