JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Martin Straßer, BSc

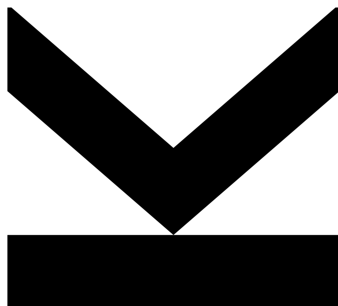
Angefertigt am
Institut für Wirtschafts-informatik - Data & Knowledge Engineering

Betreuer
o. Univ.-Prof. Dipl.-Ing. Dr. techn. Michael Schrefl

Mitbetreuung
Ass.-Prof. Mag. Dr. Christoph Schütz

Mai 2019

Dialogbasierte Benutzungsschnittstelle für interaktive Datenanalyse in natürlicher Sprache



Masterarbeit

zur Erlangung des akademischen Grades

Master of Science

im Masterstudium

Wirtschaftsinformatik

JOHANNES KEPLER UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

__________________________ __________________________

Ort, Datum Unterschrift

Kurzfassung

Der Zweck der Datenanalyse besteht darin, neue Informationen aus bestehenden Daten zu extrahieren. Benutzer von Datenanalyse-Anwendungen stellen normalerweise nicht eine einzige, perfekte Anfrage. Stattdessen ist Datenanalyse typischerweise ein iterativer Prozess, bei dem ein Benutzer eine Abfrage ausführt, ein Ergebnis erhält und darauf basierend wiederum eine neue Abfrage ausführt.

Anwender in Fachabteilungen verfügen grundsätzlich nicht unbedingt über Programmierkenntnisse bzw. Kenntnisse einer formalen Abfragesprache. Natural Language Interfaces to Databases sollen dabei Abhilfe schaffen. In diesem Zusammenhang sieht das Guided-Interaction-Paradigma vor, dass Benutzer bei der Erstellung von Abfragen angeleitet werden, indem mögliche Werte übersichtlich präsentiert werden und schrittweise eine Abfrage erstellt wird.

Im Sinne des Guided-Interaction-Paradigmas präsentiert die vorliegende Arbeit ein dialogbasiertes Natural Languge Interface für die interaktive Datenanalyse. Dabei wird ein wissensbasierter Ansatz gewählt, da reine Machine-Learning-Ansätze in diesem Fall ungeeignet sind. Als Basis für die Abfrageerstellung dienen dabei Analysegraphen und Analysesituationen sowie eine maschinenlesbare Definition des verwendeten konzeptuellen multidimensionalen Modells. Der Benutzer kann dabei zunächst eine multidimensionale Abfrage als natürlichen Text formulieren. Das System versucht daraus mittels Constraint Satisfaction und heuristischen Regeln, auf Basis von lexikalischer und semantischer Ähnlichkeit, eine Analysesituation zu erstellen. Mittels Navigationsoperatoren kann die Analysesituation verfeinert werden. Es war dabei kein Ziel ein universelles NLIDB-System zu entwickeln.

Abstract

The purpose of data analysis is to extract new information from existing data. Users of data analysis applications usually do not make a single, perfect request. Instead, data analysis is typically an iterative process in which a user executes a query, obtains a result, and then performs a new query based on it.

In general, users in departments do not necessarily have programming skills or knowledge of a formal query language. Natural Language Interfaces to Databases serve to overcome this problem. In this context, the guided interaction paradigm guides users through the process of query formulation by presenting possible values in a concise manner and by allowing users to formulate a query step by step.

Following the guided interaction paradigm, this work presents a dialog-based natural languge interface for interactive data analysis. A knowledge-based approach is chosen because pure machine-learning approaches are unsuitable in this case. The basis for query generation are analysis graphs and analysis situations as well as a machine-readable definition of the conceptual multidimensional model. The user can first formulate a multidimensional query as natural language text. The system then tries to create an analysis situation using constraint satisfaction and heuristic rules, based on lexical and semantic similarity. By using navigation operators, the analysis situation can be refined. The development of a universal NLIDB system was not a goal of this thesis.

Inhaltsverzeichnis

1. Einleitung **1**

2. Hintergrund **3**
2.1. Natural Language Processing . . . 3
2.1.1. Tokenisierung und Part-Of-Speech-Tagging . . . 3
2.1.2. Dependency Parsing . . . 4
2.1.3. Constituency Parsing . . . 5
2.1.4. String Similarity . . . 6
2.2. Constraint Satisfaction . . . 6
2.3. Wissensrepräsentation . . . 7
2.3.1. Semantic Web . . . 7
2.3.2. Knowledge Graph . . . 8
2.4. Data Warehousing und OLAP . . . 9
2.4.1. Data Warehouse . . . 9
2.4.2. Dimensional Fact Model . . . 10
2.5. Sprachschnittstellen für Datenbanken und Datenanalyse . . . 11
2.5.1. Analyza . . . 12
2.5.2. Eviza . . . 13
2.5.3. DialSQL . . . 14

3. Interaktive Datenanalyse **15**
3.1. Enriched Dimensional Fact Model . . . 15
3.2. Analysesituationen . . . 17
3.3. Analysegraphen . . . 17

4. Verwendung der Benutzungsschnittstelle **20**
4.1. Installation der Anwendung . . . 20
4.1.1. GraphDB . . . 20
4.1.2. Anwendungskonfiguration . . . 21
4.2. Komponenten der Benutzungsschnittstelle . . . 21
4.3. Ablauf der Datenanalyse . . . 24
4.3.1. Freitexteingabe . . . 24
4.3.2. Dialogbasierte Verfeinerung . . . 26

5. Systemüberblick **30**
5.1. Architektur . . . 30
5.2. Ablauf . . . 32

6. Implementierungsdetails Backend **35**
6.1. Sitzungen und WebSockets . . . 35
6.2. Knowledge-Graph-Abfrage . . . 39
6.3. State Chart XML . . . 42
6.4. Drools . . . 44
6.5. Similarity Index . . . 51

6.6. Natural Language Processing . . . 54
6.7. Constraint Satisfaction . . . 56
6.8. Anpassbarkeit . . . 59
6.9. Mehrsprachigkeit . . . 60

7. Implementierungsdetails Frontend **61**
7.1. Komponenten . . . 61
7.2. Services . . . 63
7.3. Mehrsprachigkeit . . . 66

8. Fazit und Ausblick **67**

A. Materialisierungsabfragen **72**

B. Verwendete Bibliotheken **78**

C. RDF-Schema **81**

Abbildungsverzeichnis

1. Beispiel Part-Of-Speech-Tagging . . . 4
2. Beispiel Dependency Parse . . . 4
3. Beispiel Constituency Parse . . . 5
4. Beispiel Enriched Dimensional Fact Model . . . 16
5. Beispiel für eine vergleichende Analysesituation . . . 18
6. Beispiel für eine Analysegraph-Instanz . . . 19
7. Screenshot der Benutzungsschnittstelle . . . 22
8. Programmablauf . . . 25
9. Einfache Liste . . . 28
10. Zweispaltige Liste . . . 28
11. Architektur . . . 31
12. Klassendiagramm: Display . . . 37
13. Klassendiagramm: Analysesituationen . . . 38
14. Klassendiagramm: Repository . . . 40
15. Klassendiagramm: Label . . . 41
16. Zustandsautomat . . . 43
17. Klassendiagramm: ConfidenceResult . . . 48
18. Klassendiagramm: Similarity . . . 56
19. Klassendiagramm: Interceptor . . . 59
20. RDF-Schema der materialisierten Zuordnung . . . 72
21. RDF-Schema . . . 81

Tabellenverzeichnis

1. Dimensionsspezifizierung . . . 23
2. Nicht vergleichende (Non-Comparative) Analysesituation . . . 23
3. Vergleichende (Comparative) Analysesituation . . . 23
4. Unterstützte Operationen . . . 26
5. Ablauf-Beispiel: Ergebnis der Similarity-Index-Abfrage . . . 33
6. Aufgaben der Komponenten . . . 35
7. WebSocket-Warteschlangen . . . 36
8. WebSocket Endpoints . . . 36
9. Drools-Regeln “Operation Display Determination” . . . 45
10. Drools-Regeln “Value Display Determination” . . . 48
11. Einstellungen für CSP . . . 58
12. Frontend-Einstellungen im Local Storage . . . 64
13. Verwendete Bibliotheken für das Backend . . . 78
14. Verwendete Bibliotheken für das Frontend . . . 79

Listings

1. Anwendungskonfiguration application.properties . . . 21
2. WebSocket: Start-Nachricht . . . 36
3. WebSocket: Input-Nachricht . . . 37
4. WebSocket: Display-Nachricht . . . 37
5. WebSocket: Analysesituation-Nachricht . . . 38
6. Java-Methode GraphDB-Abfrage . . . 41
7. Beispiel für eine Drools-Regel . . . 45
8. Similarity-Index . . . 51
9. Similarity-Index Abfrage . . . 52
10. Similarity-Index Abfrage: Teil mit Materialization . . . 53
11. Stanford CoreNLP Text Annotierung . . . 54
12. Stanford CoreNLP Semgrex-Pattern . . . 55
13. Constraint Satisfaction: Planning Solution . . . 57
14. Constraint Satisfaction: Planning Entity . . . 57
15. Constraint Satisfaction: Drools Regeln . . . 58
16. Inhalt der Haupt-Komponente . . . 61
17. Einstellung auslesen . . . 64
18. Nachricht senden . . . 64
19. Sprachausgabe . . . 65
20. Anpassen des Sprachkürzels für Sprachausgabe- und erkennung . . . 66
21. Materialisierungs-Abfrage: Aggregate Measure . . . 72
22. Materialisierungs-Abfrage: Aggregate Measure Predicate . . . 73
23. Materialisierungs-Abfrage: Base Measure Predicate . . . 74
24. Materialisierungs-Abfrage: Level Predicate . . . 75

25. Materialisierungs-Abfrage: Granularity Level 77
26. RDF-Schema Beispiel: Base Cube 81
27. RDF-Schema Beispiel: Base Measure 82
28. RDF-Schema Beispiel: Base Measure Predicate 82
29. RDF-Schema Beispiel: Aggregate Measure 82
30. RDF-Schema Beispiel: Aggregate Measure Predicate 82
31. RDF-Schema Beispiel: Comparative Measure 82
32. RDF-Schema Beispiel: Comparative Measure Predicate 83
33. RDF-Schema Beispiel: Dimension 83
34. RDF-Schema Beispiel: Hierarchy 83
35. RDF-Schema Beispiel: Hierarchy Step 83
36. RDF-Schema Beispiel: Level 83
37. RDF-Schema Beispiel: Attribute 83
38. RDF-Schema Beispiel: Level Predicate 84
39. RDF-Schema Beispiel: Conjunctive Level Predicate 84
40. RDF-Schema Beispiel: Level Member 84
41. RDF-Schema Beispiel: Join Condition Predicate 84

1. Einleitung

Datenanalyse ist ein iterativer Prozess, bei dem ein Benutzer eine Abfrage ausführt, ein Ergebnis erhält und darauf basierend wiederum eine neue Abfrage ausführt (Hellerstein et al., 1999). Der Zweck besteht darin, neue Informationen aus den Daten zu extrahieren. Benutzer von Datenanalyse-Anwendungen stellen normalerweise nicht eine einzige, perfekte Abfrage, die daraufhin sofort die gewünschten Informationen liefert. Stattdessen wird oft mit einer breiten Abfrage begonnen, die dann in den weiteren Schritten verfeinert wird (Hellerstein et al., 1999).

Es kann generell nicht davon ausgegangen werden, dass Anwender in Fachabteilungen über Programmierkenntnisse verfügen. Interaktive Anwendungen für die Datenanalyse ermöglichen es auch Benutzern ohne SQL- und Programmierkenntnisse Analysen durchzuführen. *Self-Service Business Intelligence* (SSBI) ist ein Ansatz zur Datenanalyse, bei dem Anwender aus den Fachabteilungen eigenständig und weitestgehend unabhängig von IT-Spezialisten Analysen durchführen können (Alpar & Schulz, 2016). Dies ermöglicht es den Anwendern bei wichtigen Entscheidungen Ad-hoc-Analysen durchzuführen und basierend auf den erhaltenen Informationen Entscheidungen zu treffen. Anwender gehen dabei von der reinen Nutzung der Daten zur Erkundung von Daten über.

In der Vergangenheit wurden *Natural Language Interfaces to Databases* (NLIDB) sowohl für die Abfrage von operationalen Datenbanken als auch für die Datenanalyse vorgeschlagen, in der Hoffnung, dass diese einfacher und natürlicher zu benutzen sind als zum Beispiel Abfragen mit SQL (Androutsopoulos, Ritchie & Thanisch, 1995). NLIDB-Systeme ermöglichen den Zugriff auf eine Datenbank mittels natürlichsprachlicher Abfragen.

Ein genereller Nachteil von Benutzungsschnittstellen, die mittels natürlicher Sprache funktionieren (Natural Language Interfaces) ist, dass die Benutzer meist nicht wissen, über welche Fähigkeiten das System verfügt und welche Eingaben verstanden werden (Bernstein & Kaufmann, 2006). Um ein Natural Language Interface nutzen zu können, müssen die Benutzer wissen, was sie sagen bzw. fragen können. Dem gegenüber steht das *Guided-Interaction*-Paradigma, welches Benutzer bei der Erstellung von Abfragen leitet, indem dem Benutzer die möglichen Werte übersichtlich präsentiert werden (Nandi & Jagadish, 2011).

Diese Arbeit präsentiert ein Natural Languge Interface für die Datenanalyse. Dabei wird ein wissensbasierter Ansatz gewählt, da reine Machine-Learning-Ansätze in diesem Fall ungeeignet sind (Dhamdhere, McCurley, Nahmias, Sundararajan & Yan, 2017). Als Basis für die Abfrageerstellung dienen dabei Analysegraphen und Analysesituationen sowie eine maschinenlesbare Definition des verwendeten konzeptuellen multidimensionalen Modells, dessen Elemente automatisiert semantisch ähnlichen WordNet-Begriffen zugewiesen werden. Ein Analysegraph modelliert eine Abfolge von multidimensionalen Abfragen, welche durch Analysesituationen repräsentiert werden (Neuböck & Schrefl, 2015; Neuböck, Neumayr, Rossgatterer, Anderlik & Schrefl, 2012). Navigationsoperatoren überführen dabei eine Analysesituation in eine andere Analysesituation. Beispiele für Navigationsoperatoren sind *addMeasure* (Kennzahl hinzufügen) oder *drillDown*.

Das präsentierte Natural Language Interface für Datenanalyse ist ein dialogbasiertes System, das Benutzer im Sinne des Guided-Interaction-Paradigmas dabei unterstützt eine Analysesituation zu befüllen, um damit eine Abfrage zur Datenanalyse auszuführen. Der Benutzer kann zunächst eine multidimensionale

Abfrage als natürlichen Text formulieren. Das System versucht daraus mittels Constraint Satisfaction und heuristischen Regeln, auf Basis von lexikalischer und semantischer Ähnlichkeit, eine Analysesituation zu erstellen. Mittels Navigationsoperatoren kann die Analysesituation verfeinert werden. Es war kein Ziel ein universelles NLIDB-System zu entwickeln, das heißt, die Komplexität möglicher Abfragen ist durch die Ausdrucksstärke von Analysesituationen und Navigationsoperatoren begrenzt. Bei der Implementierung wurde auf die einfache Konfigurier- und Erweiterbarkeit Wert gelegt. Die Programmlogik wurde deshalb deklarativ mittels Regeln und Zustandsdiagrammen spezifiziert.

Der Rest dieser Masterarbeit ist wie folgt aufgebaut: Kapitel 2 beschreibt Konzepte, die in dieser Masterarbeit zur Anwendung kommen. Weiters wird hier auch auf verwandte Literatur eingegangen. In Kapitel 3 werden das Enriched Dimensional Fact Model, Analysegraphen und Analysesituationen vorgestellt. Kapitel 4 beschreibt die Benutzung des entwickelten Systems. Kapitel 5 beschreibt die Architektur des entwickelten Systems. Kapitel 6 und 7 gehen auf die Implementierungsdetails des Backends und des Frontends ein. Kapitel 8 fasst diese Arbeit zusammen und gibt einen Ausblick über Möglichkeiten, wie das System weiter ausgebaut werden kann.

2. Hintergrund

Dieses Kapitel beschreibt die für diese Arbeit benötigten Grundlagen. Kapitel 2.1 beschreibt Natural Language Processing. In Kapitel 2.2 wird das Constraint Satisfaction Problem vorgestellt. Kapitel 2.3 widmet sich der Wissensrepräsentation. In Kapitel 2.4 werden Data Warehousing, OLAP und Dimensional Fact Model erklärt. Zuletzt wird in Kapitel 2.5 auf Sprachschnittstellen für Datenbanken eingegangen und mit dieser Arbeit verwandte Literatur vorgestellt.

2.1. Natural Language Processing

Natural Language Processing (NLP) oder Computerlinguistik ist ein Teilgebiet der Informatik, das sich mit Computertechniken befasst, um die menschliche Sprache zu erlernen, zu verstehen und zu produzieren. NLP-Systeme können mehreren Zwecken dienen. Ein Zweck kann sein, mittels maschineller Übersetzung die zwischenmenschliche Kommunikation zu unterstützen. Ein anderer Zweck ist die Mensch-Maschine-Kommunikation, wie beispielsweise bei Dialogsystemen oder Konversationsagenten. (Hirschberg & Manning, 2015)

Zu Beginn von NLP wurden meist manuell erstellte Regeln und Vokabulare verwendet. Aufgrund der Mehrdeutigkeit, der enormen Größe und der Variabilität der natürlichen Sprache, hat sich dies als schwierig erwiesen. Dies führte zur Entwicklung von statistischem oder Korpus-basiertem Natural Language Processing. Ein Ansatz hierbei ist mit weniger, allgemeineren Regeln zu arbeiten, die mit statistischen Häufigkeiten arbeiten, um Mehrdeutigkeiten zu entfernen. Ein anderer Ansatz ist das Erkennen wahrscheinlichkeitsbasierter Regeln aus mit Anmerkungen versehenen Texten mittels Machine Learning. (Nadkarni, Ohno-Machado & Chapman, 2011)

NLP-Frameworks verwenden sogenannte Pipelines, wobei eine Pipeline aus mehreren NLP-Aufgaben besteht. Jede NLP-Aufgabe hat mehrere untergeordnete Aufgaben. Bevor eine weitere Aufgabe ausgeführt werden kann, müssen zuerst alle untergeordneten Aufgaben in einer sequenziellen Reihenfolge durchgeführt werden, da das Ergebnis einer Aufgabe die Eingabedaten der nächsten Aufgabe ist. Dies ermöglicht es für jede Aufgabe unterschiedliche Arten von Algorithmen zu verwenden und diese auch auszutauschen. (Nadkarni et al., 2011)

In den folgenden Unterkapiteln werden NLP-Aufgaben vorgestellt, die in dieser Masterarbeit verwendet werden.

2.1.1. Tokenisierung und Part-Of-Speech-Tagging

Tokenisierung ist meist der erste Schritt in der NLP-Pipeline. Hierbei wird der Eingabetext in seine einzelnen Einheiten (Token) aufgeteilt. Ein Token kann ein Wort sein, aber auch zum Beispiel eine Zahl oder ein Satzzeichen (Manning & Schütze, 1999).

Beim Part-Of-Speech-Tagging werden Wörtern eines Satzes Wortarten (Nomen, Verb, etc.) zugewiesen (Charniak, 1997). Dabei kann es vorkommen, dass einem Wort meherere Wortarten zugeordnet werden. Es ist Aufgabe des Taggers, die korrekten Zuordnungen auszuwählen, wobei regelbasierte und stochastische Tagger existieren. In dieser Arbeit wird für das Part-Of-Speech-Tagging das Stanford CoreNLP-Framework

(Manning et al., 2014) verwendet. Der CoreNLP Part-Of-Speech-Tagger ist ein stochastischer Tagger und verwendet für die englische Sprache das Penn Treebank Tag Set.

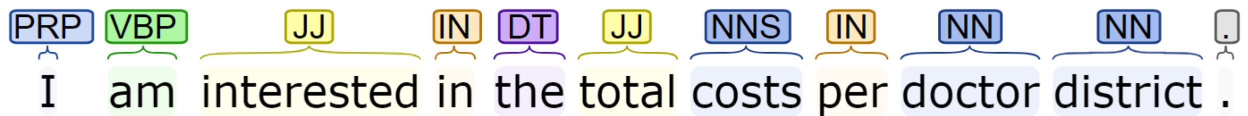


Abbildung 1: Beispiel Part-Of-Speech-Tagging

Abbildung 1 zeigt ein Beispiel für einen Satz, dem Wortarten zugewiesen wurden. Die Visualsierung wurde mit Hilfe der Internetseite http://corenlp.run erzeugt, welches Stanford CoreNLP für Part-Of-Speech-Tagging verwendet und das Ergebnis visualisiert. Im Beispielsatz kommen folgende Wortarten vor:

- **PRP**: Personal Pronoun
- **VBP**: Verb, non-3rd person singular present
- **JJ**: Adjective
- **IN**: Preposition or subordinating conjunction
- **DT**: Determiner
- **NN**: Noun, singular or mass
- **NNS**: Noun, plural

2.1.2. Dependency Parsing

Beim Dependency Parsing wird die syntaktische Struktur eines Satzes anhand der Wörter und der dazugehörigen gerichteten, binären Beziehungen zwischen diesen beschrieben (Jurafsky & Martin, 2018). Eine Beziehung hat einen Typ, geht vom "head" aus und zeigt auf die Wörter, die den "head" modifizieren. Abbildung 2 zeigt die Abhängigkeiten zwischen den Wörtern am Beispiel des vorherigen Satzes in Abbildung 1. Auch für das Dependency Parsing wird in dieser Arbeit das Stanford CoreNLP-Framework verwendet.

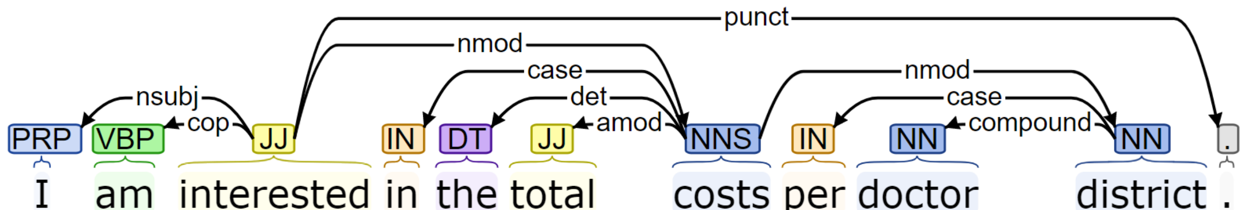


Abbildung 2: Beispiel Dependency Parse

Das *Universal Dependencies* Projekt definiert sprachübergreifende Abhängigkeitsbeziehungen (Nivre et al., 2016), wobei zwischen klausalen und modifizierenden Beziehungen unterschieden wird. Ein Beispiel für eine klausale Beziehung im Beispiel ist *nsubj* (nominal subject); ein Beispiel für eine modifizierende Beziehung ist *amod* (adjectival modifier).

2.1.3. Constituency Parsing

Beim Constituency Parsing wird ein sogenannter *Parse Tree* erzeugt, der die syntaktische Struktur eines Satzes, basierend auf einer kontextfreien Grammatik, darstellt (Jurafsky & Martin, 2018, S. 194 ff.). Die Grammatik verwendet *Constituents* (Bestandteile) eines Satzes. Ein Bestandteil kann aus mehreren Wörtern bestehen. In dieser Arbeit wird das Stanford CoreNLP-Framework für Constituency Parsing verwendet.

Es gibt unterschiedliche Arten von *Constituents*. Abbildung 3 zeigt einen Parse Tree für den Beispielsatz und entählt unter anderem Bestandteile vom Typ NP (Noun Phrase). Ein Noun Phrase enthält ein Nomen und Informationen über das Nomen (z. B. "the total costs") (Manning & Schütze, 1999, S. 95). Andere Arten sind Prepositional Phrases (PP) und Verb Phrases (VP).

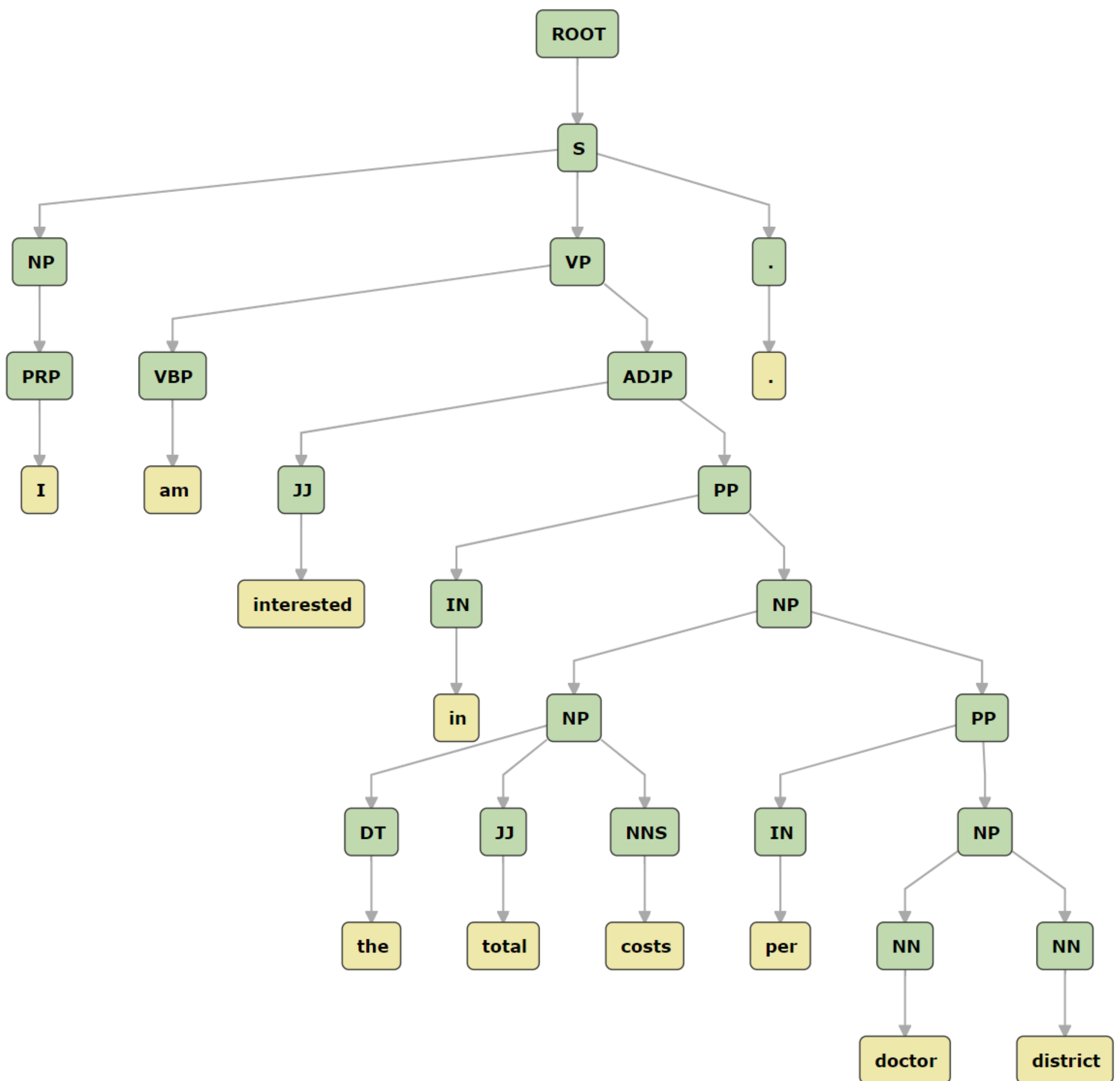


Abbildung 3: Beispiel Constituency Parse

2.1.4. String Similarity

String-Similarity-Metriken geben die Ähnlichkeit zwischen zwei Texten an (Navarro, 2001). Eine Art von String-Similarity-Metriken sind sogenannte Distanz-Metriken. Hierbei wird die Anzahl an Operationen berechnet, die benötigt werden, um einen String in den anderen umzuwandeln. Je mehr Operationen notwendig sind, desto geringer ist die Ähnlichkeit. Diese Metriken können nur die syntaktische Ähnlichkeit, nicht aber die semantische Ähnlichkeit zwischen zwei Strings berechnen. (Lu, Lin, Wang, Li & Wang, 2013)

Die **Levenshtein Distanz** (oder Editierdistanz) (Levenshtein, 1966) gibt die minimale Anzahl von Zeichenänderungen (einfügen, löschen, ersetzen) an, die erforderlich sind, um ein Wort in das andere zu ändern. Je weniger Zeichenänderungen notwendig sind, desto ähnlicher sind die Wörter. Die **Jaro-Winkler Distanz** (Keil, 2019) wurde zur Duplikaterkennung mit kurzen Texten (z. B. Namen) entwickelt und wird anhand einer Formel berechnet, die die Anzahl an übereinstimmenden Zeichen und die Anzahl an Transpositionen mit einbezieht. Beide Metriken werden in dieser Arbeit verwendet.

2.2. Constraint Satisfaction

Constraint-Satisfaction-Probleme (deutsch Bedingungserfüllungsproblem) treten in vielen verschiedenen Bereichen auf (z. B. Terminplanung, Ressourcenallokation, Computer Vision). Oft zitierte Beispiele für Constraint-Satsifaction-Probleme in der Literatur sind N-Queens und das Einfärben von Karten oder Kanten von Graphen (Tsang, 1996).

Ein **Constraint-Satisfaction-Problem** (CSP) besteht aus (Barták, 2003):

- einer Menge von *Variablen*
- für jede Variable eine bestimmte Menge an möglichen Werten (*Domäne*)
- und einer Menge von *Bedingungen* (Constraints), die die Werte der Variablen einschränken

Die Aufgabe von Constraint Satisfaction ist es, gültige Lösungen zu finden. Bei einer gültigen Lösung wurde jeder Variable ein Wert zugewiesen, sodass alle Bedingungen für alle Variablen gleichzeitig erfüllt sind. Je nach Anforderungen kann man laut Barták (1998) zwischen verschiedenen Arten von Lösungen unterscheiden:

- Irgendeine Lösung
- Alle möglichen Lösungen
- Optimale Lösung oder zumindest eine gute Lösung

Um Lösungen zu finden, wird eine systematische Suche durchgeführt, bei der nach möglichen Zuordnungen von Werten zu Variablen gesucht wird. Barták (2003) beschreibt, dass hierbei normalerweise eine Kombination von Suchalgorithmen (z. B. Backtracking) mit **Constraint Propagation** (Bedingungsfortpflanzung) verwendet wird. Die Idee von Constraint Propagation ist, dass die Bedingungen den Suchraum einschränken, indem Werte, die in keiner möglichen Lösung vorkommen, entfernt werden. Jedes Mal,

wenn sich die Domäne einer Variable in einer Bedingung ändert und dies an die Domänen der anderen Variablen weitergegeben wird, wird der Algorithmus ausgeführt.

Der Vorteil, Probleme als Constraint Satisfaction Probleme zu modellieren, ist, dass die Formulierung einfacher und die Lösung verständlicher ist, als zum Beispiel mit mathematischen Formeln. Weiters kann ein Constraint Satisfaction Problem oft schneller gelöst werden, als bei Verwendung eigener Algorithmen. (Barták, 1998)

Optaplanner (Red Hat, 2019) ist ein Constraint Solver und wird in dieser Arbeit zur Lösung von Constraint-Satisfaction-Problemen verwendet. Die Bedingungen können in Optaplanner in Java oder mittels Drools-Regeln erstellt werden. Jede von Optaplanner gefundene Lösung hat einen sogenannten Score. Mögliche Lösungen werden anhand des Scores verglichen. Optaplanner versucht den höchsten Score von allen möglichen Lösungen zu finden. Die beste Lösung ist somit jene mit dem höchsten Score. Es gibt unterschiedliche Arten von Scores; unter anderem Simple-Score und Hard-Soft-Score. Der Simple-Score besteht aus einem einzigen Wert, der die Lösung bewertet. Der Hard-Soft-Score setzt sich aus zwei Werten zusammen, nämlich Hard- und Soft-Score, wobei der Hard-Score bestimmt, ob eine Lösung gültig ist. Wenn der Hard-Score negativ ist, ist die Lösung ungültig. Der Soft-Score kann negativ sein, falls es sich nicht vermeiden lässt.

2.3. Wissensrepräsentation

Unland (2019) definiert Wissensrepräsentation wie folgt: "Wissensrepräsentation (engl.: knowledge representation) [...] beschäftigt sich mit der Frage, wie Wissen auf einem Rechner dargestellt werden kann, sodass dieser in der Lage ist, daraus Schlüsse zu ziehen, weiteres Wissen abzuleiten oder eine Problemlösung für ein gegebenes Problem zu finden." In den folgenden Unterkapiteln wird zuerst auf Technologien zur Wissensrepräsentation eingegangen und anschließend werden Anwendungen der Wissensrepräsentation vorgestellt.

2.3.1. Semantic Web

Die meisten Inhalte des World Wide Web sind für Menschen zum Lesen gedacht und nicht für die Verarbeitung durch Computerprogramme (Berners-Lee, Hendler & Lassila, 2001). Computerprogramme haben keine verlässliche Möglichkeit, die Semantik der Inhalte im Web zu ermitteln. Das Semantic Web ist eine Erweiterung des WWW, das Informationen in einer für Computer lesbaren Sprache definiert. Das W3C (World Wide Web Consortium) definiert offene Standards für das Semantic Web: zum Beispiel RDF(S), OWL oder SPARQL.

RDF Das Resource Description Framework (W3C, 2014) dient zur Beschreibung von strukturierten Informationen. RDF-Dokumente sind gerichtete Graphen, deren Knoten und Kanten durch URIs beschriftet sind. Ein RDF-Dokument besteht aus mindestens einem Tripel das sich aus Subjekt, Prädikat und Objekt zusammensetzt. Subjekt und Objekt sind dabei die Knoten; das Prädikat ist die Kante. Das Objekt kann nicht nur ein Knoten, sondern auch ein Literal oder Blank Node sein. Es gibt verschiedene Formate, um RDF-Dokumente abzuspeichern. Diese sind unter anderem Notation 3 (N3), N-Triples, Turtle und RDF/XML.

SPARQL SPARQL Protocol and RDF Query Language (W3C, 2013) ist eine Abfragesprache, mit der Daten, die im RDF-Format gespeichert sind, abgefragt und manipuliert werden können. Eine SPARQL-Abfrage kann aus Triple Patterns (Subjekt - Prädikat - Objekt), optionalen Patterns, Filtern und Vereinigungen bestehen. Mit Hilfe von SPARQL Update können Daten modifiziert werden.

Triplestore Triplestores oder RDF-Stores dienen zur Verwaltung von RDF-Daten (Pan, Zhu, Liu & Ning, 2018). Die Daten in Triplestores können mittels SPARQL abgefragt und modifiziert werden. Sie bieten die Möglichkeit, mit Hilfe von OWL neues Wissen abzuleiten. Weiters bieten manche Triplestores weitere Funktionalitäten wie zum Beispiel das Erstellen von Similarity-Indizes an. Beispiele für Triplestores sind Jena TDB, AllegroGraph oder GraphDB.

2.3.2. Knowledge Graph

Es existiert keine einheitliche Definition für Knowledge Graphs. Der Begriff Knowledge Graph wurde von Google geprägt. Zusätzlich zu den normalen Sucheinträgen bei einer Google-Suche werden in der rechten Spalte Zusatzinformationen zum Suchbegriff aufbereitet, die aus verschiedenen Quellen bezogen werden. Diese Zusatzinformationen werden mittels Google Knowledge Graph aufbereitet (Google, 2012).

Mittlerweile bezieht sich der Begriff Knowledge Graph nicht mehr ausschließlich auf den Google Knowledge Graph, sondern auch auf andere Wissensbasen im Semantic Web. Paulheim (2017) definiert Knowledge Graphs über folgende Charakteristiken: Ein Knowledge Graph

- beschreibt hauptsächlich Entitäten der realen Welt und deren Zusammenhänge.
- definiert Klassen und Beziehungen von Entitäten in einem Schema.
- ermöglicht beliebige Entitäten in Beziehung zueinander zu setzen.
- deckt verschiedene Themenbereiche ab.

Laut Gomez-Perez, Pan, Vetere und Wu (2017) besteht ein Knowledge Graph aus einer Menge miteinander verbundener, typisierter Entitäten und deren Attributen. Eine Entität ist die Basiseinheit eines Knowledge Graphs. Eine Entität kann mehrere Attribute besitzen und mit anderen Entitäten über Beziehungen verbunden sein. Mithilfe von Beziehungen können zwei separate Knowledge Graphs miteinander verbunden werden. Dies bedingt, dass jede Entität eine global eindeutige ID hat.

Folgend wird ein Überblick über existierende, frei verfügbare, Knowledge Graphs gegeben.

Wikidata (Vrandečić & Krötzsch, 2014) wird von der Wikimedia-Stiftung, die auch Wikipedia betreibt, betrieben. Das Problem von Wikipedia war zum Beispiel, dass auf unterschiedlichen Wikipedia-Seiten zu einer Stadt unterschiedliche Bevölkerungszahlen ausgewiesen waren. Wikidata versucht diese Inkonsistenzen zu bereinigen und Informationen für bestimmte Datentypen allen Wikipedia-Seiten zur Verfügung zu stellen.

DBpedia (Lehmann et al., 2015) extrahiert strukturiertes, multilinguales Wissen aus Wikipedia und stellt dieses mit Hilfe von Semantic Web und Linked-Data-Technologien frei im Internet zur Verfügung.

Viele andere Datensammlungen verweisen auf DBpedia Einträge, was DBpedia zu einem zentralen Hub in der Linked Open Data Cloud macht.

Yago (Rebele et al., 2016) extrahiert genauso wie DBpedia, Informationen aus Wikipedia, wobei sich Yago auf die Präzision der Datenextraktion und auf die Korrektheit der Daten konzentriert. Weiters verknüpft Yago die Wikipedia-Kategorien mit WordNet Synsets. Auch GeoName-Entitäten, die auf Wikipedia-Einträge verweisen, werden zu WordNet Klassen zugeordnet.

WordNet (Miller, 1995) ist eine lexikalische Datenbank für die englische Sprache. WordNet ist nach Definition von Paulheim kein Knowledge Graph, weil es nur Wörter und deren Beziehungen abbildet und keine Entitäten der realen Welt; nach der Definition von Gomez-Perez et al. allerdings schon. WordNet organisiert Nomen, Verben, Adjektive und Adverben in Gruppen von Synonymen (Synsets), die durch semantische Beziehungen miteinander verknüpft werden. In WordNet existieren folgende Arten von semantischen Beziehungen: Synonym, Antonym (Gegenteil), Hyponym (Unterbegriff), Hyperonym (Oberbegriff), Meronym (Teil-Ganzes-Relation), Troponym (Beinhaltet-Beziehung).

2.4. Data Warehousing und OLAP

Dieses Kapitel beschreibt zuerst in Kapitel 2.4.1 Data Warehouses. Darauffolgend wird in Kapitel 2.4.2 das Dimensional Fact Model beschrieben.

2.4.1. Data Warehouse

Data Warehousing umfasst Methoden, Techniken und Werkzeuge um Wissensarbeiter bei der Durchführung von Datenanalysen im Zuge von Entscheidungsprozessen zu unterstützen (Golfarelli & Rizzi, 2009, S. 4). Ein **Data Warehouse** ist ein System, das Informationen aus unterschiedlichen Quellen in einem großen Datenspeicher zusammenfasst (Golfarelli, Maio & Rizzi, 1998).

Inmon (2005, S. 29 ff.) definiert folgende Eigenschaften für Data Warehouses: Themenorientiert, integriert, nicht flüchtig und chronologisiert. Wie bereits beschrieben, fasst ein Data Warehouse Daten aus mehreren unterschiedlichen Datenquellen zusammen (**integriert**). Betriebliche Datenbanken orientieren sich an den Geschäftsprozessen des Unternehmens. Ein Data Warehouse hingegen orientiert sich an den Analyseprozessen (**themenorientiert**). Ein Data Warehouse ist **nicht flüchtig**, da normalerweise Daten in ein Data Warehouse eingefügt werden und danach kaum wieder verändert oder gelöscht werden, um Analysen über mehrere Jahre durchführen zu können. Jeder Datensatz in einem Data Warehouse ist immer zu einem bestimmten Zeitpunkt gültig (z. B. durch Zeitstempel), daher ist dieses **chronologisiert**.

Normalerweise wird ein Data Warehouse getrennt von den betrieblichen Datenbanken verwaltet. Betriebliche Datenbanken unterstützen *On-Line Transaction Processsing* (OLTP). Hierfür werden atomare, isolierte Transaktionen mit meist wenigen Datensätzen verwendet. Performance, Konsistenz und Wiederherstellbarkeit sind bei Datenbanken, die im täglichen Betrieb verwendet werden, sehr wichtig. (Chaudhuri & Dayal, 1997)

Im Gegensatz dazu wird *On-Line Analytical Processing* (OLAP) zur Abfrage von Data Warehouses verwendet, wobei oft sehr viele Daten abgefragt werden. OLAP unterstützt unter anderem folgende

Operationen auf multidimensionale Daten: Roll-Up, Drill-Down, Slice and Dice, Pivot. Eine Roll-Up-Operation, zum Beispiel, verringert den Detaillierungsgrad (z. B. die Verkäufe werden zu einer Woche, einem Monat oder Jahr aggregiert). (Chaudhuri & Dayal, 1997)

2.4.2. Dimensional Fact Model

Ein weit verbreitetes Modell zur Modellierung von Data Warehouses ist das Dimensional Fact Model (DFM) von Golfarelli et al. (1998). Ein DFM ist ein grafisches konzeptuelles Modell für Data Warehouses. Dieses Dimensional Fact Model basiert auf dem multidimensionalen Modell und soll die Modellierung von Data Warehouses vereinfachen. Ein DFM besteht aus einer Menge von Faktenschemata, deren Kernelement eine Fact-Klasse ist. Im Folgenden werden die wichtigsten Elemente des DFM kurz erläutert. Die Erläuterungen folgen dabei Golfarelli und Rizzi (2009, S. 103 ff.):

Fact: „Ein Fact ist ein Konzept, das für Entscheidungsprozesse relevant ist. Normalerweise repräsentiert ein Fact eine Menge von Ereignissen, die in einem Unternehmen stattfinden."

Kennzahlen: „Eine Kennzahl ist eine numerische Eigenschaft des Facts und beschreibt den quantitativen Aspekt des Facts, der für die Analyse relevant ist."

Dimension: „Eine Dimension ist eine Eigenschaft eines Facts mit einer endlichen Domäne und beschreibt eine Analysekoordinate."

Dimensionale Attribute: „Ein dimensionales Attribut steht für Dimensionen und andere Attribute, die durch diskrete Werte beschrieben werden."

Hierarchie: „Eine Hierarchie ist ein gerichteter Graph. Die Knoten der Hierarchie sind dimensionale Attribute und die Kanten beschreiben m:n-Beziehungen zwischen den dimensionalen Attributen. Die Wurzel der Hierarchie ist eine Dimension. Die dimensionalen Attribute beschreiben die Dimension."

Deskriptives Attribut: Ein deskriptives Attribut beschreibt ein dimensionales Attribut.

In dieser Arbeit wird ein auf dem DFM aufbauendes Modell verwendet (siehe Kapitel 3.1), dessen Terminologie jedoch in einigen Punkten vom DFM abweicht. Insbesondere stellt eine Dimension eine Klasse von Objekten dar, die *Dimension Members* oder *Dimension Nodes* genannt werden. Die Dimension Members/Nodes werden wiederum einzelnen Levels (Dimensionale Attribute) zugeordnet. Zum Beispiel besteht die Insurant-Dimension aus den Levels Insurant, Insurant District und Insurant Province, wobei "Martin Straßer" ein Dimension Member auf dem *Insurant* Level, "Linz" ein Member auf dem *Insurant District* Level und "Oberösterreich" ein Member auf dem *Insurant Province* Level wäre. Levels werden wie im DFM hierarchisch organisiert. Eine Dimension kann dabei mehrere Level-Hierarchien umfassen, wobei Hierarchien keine Verzweigungen aufweisen. Alternative bzw. Parallele Pfade werden, anders als im DFM, durch mehrere Hierarchien abgebildet. Für die maschinenlesbare Repräsentation des verwendeten Modells wird eine Adaptierung von QB4OLAP verwendet (Etcheverry & Vaisman, 2012).

2.5. Sprachschnittstellen für Datenbanken und Datenanalyse

Dieses Kapitel beschreibt Sprachschnittstellen für Datenbanken und Datenanalyse und stellt drei Beispiele für solche Systeme vor.

Androutsopoulos et al. (1995) beschreiben Sprachschnittstellen für Datenbanken (*engl.* Natural Language Interface to a Database (NLIDB)) als Systeme, die es Benutzerinnen ermöglichen, auf Daten in Datenbanken mit natürlichsprachlichen Abfragen zuzugreifen. Eines der ersten NLIDB-Systeme war *LUNAR*, entwickelt in den 1960er-Jahren.

Der Vorteil von solchen Systemen ist, dass die Anwenderinnen Daten in natürlicher Sprache abfragen können und nicht SQL oder eine andere Abfragesprache lernen müssen. Auch können einige Abfragen mit natürlicher Sprache einfacher ausgedrückt werden als mit SQL. (Androutsopoulos et al., 1995)

Androutsopoulos et al. (1995) listen einige Herausforderungen auf, die es bei der Entwicklung eines NLIDB-Systems zu beachten gibt. Eine ist, dass die Anwenderinnen oft nicht wissen, welche Arten von Fragen das System versteht, da oft nur eine limitierte Untermenge der natürlichen Sprache unterstützt wird. Des Weiteren nehmen Anwenderinnen meist an, dass das System intelligent ist und über Menschenverstand verfügt. Wenn eine Eingabe vom System nicht verstanden wurde, ist es für die Benutzerin oft unklar, ob die Eingabe außerhalb der sprachlichen oder der konzeptuellen Abdeckung liegt. In diesem Fall formulieren Benutzerinnen daher oft den Satz um und verweisen wiederum auf Konzepte, die das System nicht kennt. Ein weiteres Problem stellen falsche Schreibweisen und syntaktisch falsch geformte Eingaben dar.

Außerdem formulieren Menschen Fragen oft nicht in ganzen Sätzen. Diese sogenannten *non-sentential utterances* (NSU) treten meist bei Folgefragen auf (z. B. Welche Abteilung hat den größten Umsatz? - Und den kleinsten?), da die Benutzerinnen davon ausgehen, dass das System die Eingaben basierend auf dem Kontext der aktuellen Interaktion interpretieren. Weitere Herausforderungen sind das Auflösen von Mehrdeutigkeiten sowie das Erkennen der Absicht der Benutzerin. (Mittal, Sen, Saha & Sankaranarayanan, 2018)

Androutsopoulos et al. (1995) teilen NLIDB-Ansätze in vier Kategorien ein: **Pattern Matching Systeme** verwenden Regeln mit Mustern, um Abfragen zu generieren. **Syntaxbasierte Systeme** führen eine syntaktische Analyse der Benutzereingabe durch und der daraus resultierende *Parse Tree* wird direkt auf Ausdrücke in einer Datenbankabfragesprache zugeordnet. Hierzu werden Grammatiken verwendet, die mögliche Strukturen von Benutzereingaben beschreiben. **Semantische Grammatiksysteme** funktionieren ähnlich wie syntaxbasierte Systeme, mit dem Unterschied, dass die Kategorien der Grammatik nicht notwendigerweise syntaktischen Konzepten entsprechen. Viele NLIDB-Systeme verwenden eine sogenannte **intermediate representation language**, die die Bedeutung der Benutzerabfrage abbildet. Diese Sprache wird dann in weiterer Folge in eine Datenbank-Abfragesprache übersetzt.

Der Einsatz von solchen "Zwischensprachen" unterstützt die **Datenbankportabilität**, womit unterschiedliche Datenbanken unterstützt werden können. Weiters ist auf die **Wissensdomänenportabilität** zu achten. Wenn ein System diese Portabilität unterstützt, kann die Anwendung in mehreren Wissensdomänen eingesetzt werden, ohne gänzlich neu entwickelt werden zu müssen. Beim Einsatz in einer neuen Domäne muss das System für die neue Domäne konfiguriert werden (z. B. festlegen von

domänen-spezifischen Begriffen). Wenn ein System mehr als nur eine natürliche Sprache unterstützt, spricht man von der **Natural Language Portabilität**. Die Unterstützung von mehreren Sprachen kann schwierig sein, da hierfür die syntaktischen und semantischen Regeln für die jeweilige Sprache angepasst werden müssen. (Androutsopoulos et al., 1995)

NLIDB-Systeme verlangten anfangs von den Benutzerinnen Abfragen mit einem einzigen Satz zu stellen. Da dies zu sehr langen und komplizierten Sätzen führen kann, wurden dialogbasierte Systeme entwickelt (Kim, 2007). Diese Systeme ermöglichen es den Benutzerinnen einen Dialog mit dem System zu führen und die Abfrage aus mehreren kurzen Sätzen zusammenzubauen.

Verwandt zu den *Natural Language Interfaces to Databases* sind *Natural Language Interfaces to Knowledge Bases* (NLIKB), deren Entwicklung mit dem Aufkommen des Semantic Web begann. NLIKB-Systeme verwenden Ontologien, um Informationen zu verwalten. Beide verwenden unterschiedliche Arten von Datenbanken, haben aber gemeinsame Komponenten, wie zum Beispiel die Zuordnung von Konzepten zu Elementen in einer Abfrage. (Zhu, Yan & Song, 2017)

In den folgenden Unterkapiteln werden drei Beispiele für NLIDB-Systeme vorgestellt.

2.5.1. Analyza

Analyza ist ein System, das Benutzerinnen dabei unterstützt Daten mit Hilfe von Spracheingaben zu erforschen (Dhamdhere et al., 2017). Die Motivation zur Entwicklung von Analyza war, dass es aufgrund von komplexen Datensätzen schwierig ist, den Datensatz zu finden, der die aussagekräftigsten Ergebnisse liefert. Die Autoren dieses Systems sind der Meinung, dass hierfür eine äußerst interaktive Umgebung geschaffen werden muss, in der die Benutzerin Abfragen schnell erstellen und verfeinern kann.

Laut Dhamdhere et al. (2017) müssen Anwenderinnen dazu in der Lage sein, herauszufinden, welche Daten existieren und welche Arten von Abfragen darauf ausgeführt werden können. Das Problem dabei ist, dass Benutzerinnen oft das Schema der Datenhaltung nicht kennen. Insbesondere die Verwendung von SQL bedarf des Verständnisses des relationalen Schemas der Datenbank. Weiters sollten Benutzerinnen verschiedene Datenschichten untersuchen und die Abfragen verfeinern können. Dies führt zu einer Folge von Abfragen. Nachdem die Daten abgefragt wurden, müssen diese in einer passenden Visualisierung präsentiert werden. Oft erfolgt die Darstellung der Visualisierung bei der Datenextraktion in sogenannten *Daten-Dashboards*.

Analyza verwendet einen **Metadatenspeicher**, in dem drei Arten von Metadaten gespeichert sind. Die erste Art sind *intent words* wie zum Beispiel "top" oder "compare". Diese werden verwendet, um die Absicht der Benutzerin ermitteln zu können. Weiters werden Schema-Informationen gespeichert. Zu den Schema-Informationen gehören unter anderem die Art von Spalten (z. B. Metrik, Dimension, etc.) und Datenformate. Die dritte Art ist eine Wissensbasis über die Entitäten in den Daten, welche unter anderem Synonyme enthält.

Der Parser von Analyza verwandelt die Benutzereingabe in einen sogenannten **"semantic parse"**. Hierin enthalten sind Platzhalter für die verschiedenen Komponenten einer Abfrage, wie zum Beispiel Metriken, Dimensionen und Filter, sowie die Absicht der Benutzerin. Mithilfe der **Wissensbasis** werden Satzteile aus der Benutzereingabe zu Entitäten und Benutzerabsichten zugewiesen. Hierfür wird eine Kombination

von String-Matching, Stemming und Synonymen verwendet. Durch Verwendung von **Scores** kann aus mehreren Alternativen die Beste ermittelt werden. Des Weiteren werden **semantische Regeln** verwendet, um eine Benutzereingabe zu validieren. Ein Beispiel für so eine Regel lautet: "no limit can be specified without specifying a dimension".

Die Fragen an das System wurden von Dhamdhere et al. (2017) über ein Jahr lang aufgezeichnet und somit über 70.000 Fragen erfasst. Durch die Analyse der Fragen haben die Autoren herausgefunden, dass die meisten Fragen sehr einfach aufgebaut sind und keine ganzen Sätze sind und somit nur eine geringe syntaktische Struktur aufweisen. Deshalb ist das System stark auf die Semantik der Daten und die Wissensbasis angewiesen, um den Mangel an Syntax auszugleichen.

Analyza verzichtet auf den Einsatz von Machine Learning, da laut den Autoren zu Beginn nicht genügend hochqualitative Trainingsdaten zur Verfügung standen. Des Weiteren argumentieren sie, dass Machine-Learning-Systeme nicht robust genug sind, um eine ausreichend hohe Präzision liefern zu können.

2.5.2. Eviza

Eviza ist ein Prototyp für eine Benutzungsschnittstelle in natürlicher Sprache zur visuellen Analyse (Setlur, Battersby, Tory, Gossweiler & Chang, 2016). Eviza ermöglicht es eine interaktive Konversation mit einer existierenden Visualisierung zu führen, um diese zu verfeinern und anzupassen. Die Motivation zur Entwicklung von Eviza war, dass es zwar bereits Systeme gibt, mit denen man visuelle Analysen durchführen kann, diese aber viel Übung benötigen.

Setlur et al. (2016) haben, bevor sie mit der Entwicklung begonnen haben, eine Studie durchgeführt, um zu ermitteln, wie Personen in natürlicher Sprache mit den Visualisierungen interagieren. Darauf aufbauend wurde das System entwickelt.

Laut den Autoren von Eviza ist es schwierig Sprachschnittstellen robust zu entwickeln, da die natürliche Sprache oftmals vielfältig und ungenau ist und umfangreiches Wissen benötigt wird. Weiters werden von den Personen oft Details ausgelassen und Annahmen getroffen. Deshalb wurde bei Eviza der Umfang auf existierende Visualisierungen eingeschränkt.

Eviza verwendet eine wahrscheinlichkeitsbasierte, kontextfreie Grammatik zur Beschreibung der Struktur der Benutzereingabe. Die Daten werden mit Konzepten (z. B. Synonyme) aus existierenden Wissensbasen verknüpft, um die Semantik ermitteln zu können. Als Wissensbasen verwendet Eviza die Einheitentaxonomie von WolframAlpha und Synsets von WordNet.

Durch natürlichsprachliche Eingaben kann es zu Mehrdeutigkeiten kommen. Die Autoren von Eviza nennen hier als Beispiel, wenn Benutzerinnen nach starken Erdbeben fragen. Hierbei ist für das System unklar, wie starke Erdbeben definiert sind. Es existiert die Möglichkeit, eine Reihe von Fragen zu stellen, um eine Mehrdeutigkeit aufzulösen. Setlur et al. (2016) sind der Meinung, dass das Schätzen der Benutzerabsicht der bessere Weg ist, um mit Mehrdeutigkeiten umzugehen. Um die Mehrdeutigkeit erfassen zu können, wird eine entropiebasierter Ansatz verwendet, wobei zwischen syntaktischer und semantischer Mehrdeutigkeit unterschieden wird. Wenn eine Mehrdeutigkeit auftritt, wird in der Benutzungsoberfläche eine Komponente dargestellt, in der die Standardauswahl durch Eviza korrigiert werden kann.

Damit Benutzerinnen mit der Visualisierung eine Konversation führen und Folgeabfragen stellen können, wurde ein endlicher Zustandsautomat (FSM) implementiert.

Um Eviza zu evaluieren, haben zwölf Testpersonen eine Reihe von Aufgaben durchgeführt. Ein Erkenntnis war, dass die Testpersonen mit Eviza schneller ein Ergebnis erzielen konnten als mit Tableau. Die Testpersonen gaben an, dass sich die Benutzung von Eviza natürlicher anfühlte und einige Abfragen einfacher auszudrücken waren, als in Tableau. Eine Schwierigkeit bei der Benutzung des Systems war, dass die Testpersonen nicht wussten, welche Kommandos das System versteht. Weiters merken die Autoren an, dass die Teilnehmer der Evaluierung davon ausgegangen sind, dass das System "smart" ist und über allgemeines Wissen verfügt und vage Angaben versteht (z. B. "östliche Staaten").

2.5.3. DialSQL

DialSQL ist ein dialogbasiertes Framework zur strukturierten Generierung von Abfragen (Gur, Yavuz, Su & Yan, 2018). Existierende Human-In-The-Loop NLIDB-Systeme verlassen sich auf die Benutzerin, um die generierte SQL-Abfrage zu überprüfen. Dies ist aber nicht möglich, wenn die Benutzerin SQL nicht versteht. DialSQL verwendet daher einen anderen Ansatz. Es geht davon aus, dass jeder Teil einer Abfrage fehlerhaft sein kann. Das Ziel ist, die fehlerhaften Teile zu ermitteln und der Benutzerin Multi-Choice-Fragen zu stellen, um die Abfrage zu validieren und die Fehler zu korrigieren.

Existierende regelbasierte NLIDB-Systeme zeichnen sich durch eine hohe Präzision aus, haben aber Probleme mit Sprachvariationen. Statistische Modelle und insbesondere neurale Netzwerke sind laut den Autoren von DialSQL robuster bei Sprachvariationen. Deshalb verwendet DialSQL rekurrente neurale Netzwerke (RNN). Hierbei wird zuerst ein neurales Netzwerk auf die Dialog-Historie ausgeführt, um die Fehlerkategorie zu ermitteln (z. B. Fehler in der Select-Klausel oder Aggregationsfehler). Danach wird wiederum ein neurales Netzwerk ausgeführt, um die Position des Fehlers vorherzusagen. Basierend darauf werden mögliche Auswahlalternativen ermittelt und der Benutzerin präsentiert.

Da keine Fehler- und Interaktionsdaten zur Verfügung standen, haben die Autoren von DialSQL einen Simulator zur Generierung von simulierten Dialogen erstellt, um die neuralen Netzwerke zu trainieren.

3. Interaktive Datenanalyse

Im Folgenden werden grundlegende Konzepte für die interaktive Datenanalyse vorgestellt. In Kapitel 3.1 wird das Enriched Dimensional Fact Model anhand eines Beispiels erklärt. Kapitel 3.2 beschreibt Analysesituationen und Kapitel 3.3 Analysegraphen.

3.1. Enriched Dimensional Fact Model

Neuböck (2019) hat das Dimensional Fact Model zu einem *Enriched (auch: Enhanced) Dimensional Fact Model* (eDFM) erweitert. Die hinzugefügten arithmetischen und aggregierten Kennzahlen leiten sich von Basis-Kennzahlen ab und können bei der Formulierung von Abfragen verwendet werden. Kennzahlen- und Level-Prädikate wiederum können bei der Formulierung von Abfragen verwendet werden, um die Abfrageergebnisse zu filtern.

Abbildung 4 zeigt das Enriched Dimensional Fact Model für das laufende Beispiel dieser Arbeit, welches Daten zu Medikamentenverschreibungen enthält. Die Abbildung verwendet die Bezeichnungen (Labels) der Schema-Elemente. Jedes Element wird duch eine IRI identifiziert und hat zusätzlich zur Bezeichnung optional auch eine Beschreibung in natürlicher Sprache. Außerdem werden Ausdrücke für die Code-Generierung definiert, welche jedoch nicht Teil dieser Arbeit sind. Die Schema-Elemente werden im RDF-Format in der GraphDB abgespeichert, wobei auch Beziehungen zwischen den Modellelementen erfasst werden. Anhang C enthält Beispiele für die Repräsentation von Schema-Elementen.

Im Zentrum steht die Faktklasse *Drug Prescription*, welche als Kennzahlen die Anzahl und Kosten der Medikamentenverschreibungen (Quantity und Costs) enthält. Weiters existieren die aggregierten Kennzahlen *Sum of Quantity* und *Sum of Costs*, welche die Summe der Anzahl bzw. der Kosten abbilden. *Costs per Unit* setzt *Sum of Quantity* und *Sum of Costs* miteinander in Beziehung und gibt die Kosten pro Einheit an. *High Drug Prescription Costs per Unit* bezieht sich auf eine aggregierte Kennzahl und kann verwendet werden, um das Abfrageergebnis zu filtern, sodass nur Einträge mit hohen Verschreibungskosten pro Einheit zurückgeliefert werden. *High Costs per Unit* bezieht sich auf Kennzahlen und kann ebenfalls zum Filtern von Ergebnissen verwendet werden.

Das Dimensional Fact Model enthält vier Dimensionen (Drug, Time, Doctor, Insurant). Die *Drug*-Dimension definiert die Medikamente, wobei *ATC* für Anatomisch-Therapeutisch-Chemisches Klassifikationssystem steht, welches Arzneistoffe in 5 Stufen klassifiziert. Die *Time*-Dimension definiert die temporalen Granularitäten. Die *Insurant*-Dimension definiert Versicherte und ermöglicht die Gruppierung nach Bezirk und Land (District und Province).

Das *Insurant*-Level hat ein deskriptives Attribut, welches das Alter des Versicherten angibt. Das deskriptive Attribut von *Insurant District*, *Inhabitants per km*^{2} *in Insurant District*, gibt die Anzahl der Einwohner pro Quadratkilometer im Bezirk an. Die *Doctor*-Dimension ist gleich aufgebaut wie die *Insurant*-Dimension, hat aber zusätzlich noch Level-Prädiakte (Level Predicates). *Doctor in Rural District* ermöglicht das Filtern der Dimension Nodes nach ländlichen Bezirken. *Old Doctor* filtert alte Ärzte, das heißt Ärzte die älter als 55 Jahre alt sind. *Old Doctor in Rural District* kombiniert *Doctor in Rural District* und *Old Doctor*.

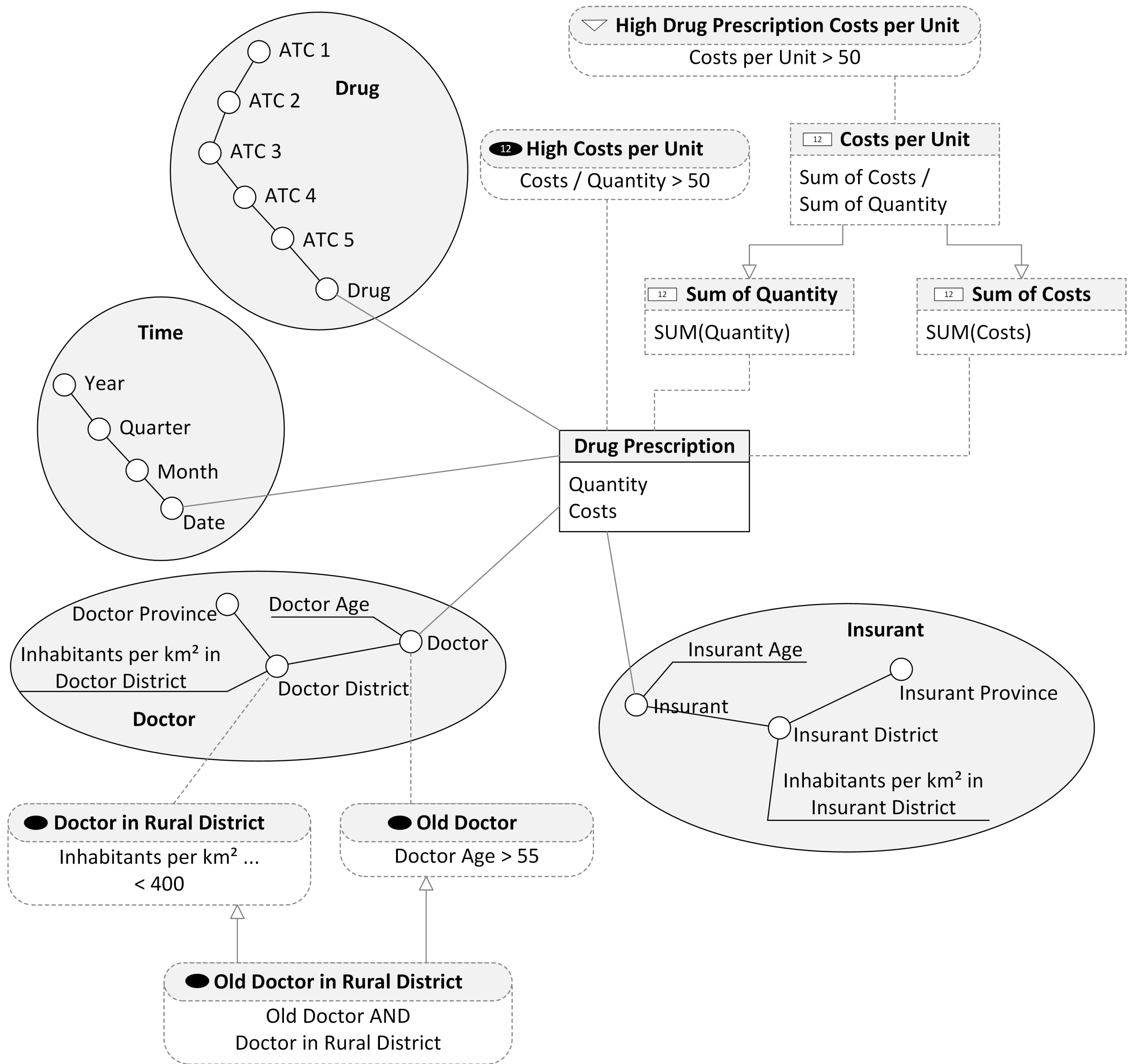


Abbildung 4: Beispiel Enriched Dimensional Fact Model
vgl. Neuböck (2019)

Die Modell-Elemente werden als Basis für die Abfragen verwendet. Insbesondere werden die Level-, Base-Measure- und Aggregate-Measure-Predicates verwendet, um die Ergebnisse einzuschränken. Aggreate Measures werden zur Berechnung von Aggregationen (z. B. Sum) verwendet.

3.2. Analysesituationen

Eine Analysesituation modelliert eine multidimensionale Abfrage (Neuböck & Schrefl, 2015). Eine Analysesituation bezieht sich auf einen Cube, Basis-Kennzahlen-Bedingungen, Filter, Kennzahlen und einer Dimensionsspezifizierung pro Dimension. Eine Dimensionsspezifizierung definiert die Granularität der Dimension, d. h. mit welchem Level die Daten aggregiert werden. Weiters ist es möglich einen Dice Node und ein Dice Level festzulegen. Die Dice Nodes können durch Festlegen von Slice-Bedingungen eingeschränkt werden.

Vergleichende Analysesituationen vergleichen ein *Context of Interest* (CoI) mit einem *Context of Comparison* (CoC). CoI und CoC wiederum sind einfache Analysesituationen. Der CoI repräsentiert den zu analysierenden Zustand, der CoC den Zustand, mit dem der CoI verglichen werden soll (Schütz, Schausberger, Kovacic & Schrefl, 2017). Ein *Score* wird verwendet, um CoI und CoC zu vergleichen. Hierzu wird ein arithmetischer Ausdruck festgelegt, der Kennzahlen aus beiden Analysesituationen miteinander in Beziehung setzt (Neuböck, 2019; Neumayr, Schrefl & Linner, 2011; Neuböck et al., 2012). Mit Hilfe von Score-Filtern kann das Abfrageergebnis auf Basis von Score-Werten gefiltert werden. *Join Conditions* geben an, wie die Einträge von CoI und CoC verknüpft werden sollen (Neuböck, 2019). Synonyme für Context of Interest und Context of Comparison sind Set of Interest (SI) bzw. Set of Comparison (SC).

Abbildung 5 zeigt eine vergleichende Analysesituation, die die Veränderung der Summe der Kosten der Jahre 2018 und 2017 pro Arzt-Bezirk zurück liefert. Ein anderes Beispiel für vergleichende Analysesituationen wäre der Vergleich eines Bezirks mit der übergeordneten Provinz sowie der Vergleich zweier Länder miteinander. In diesem Zusammenhang erlauben Semantic OLAP-Patterns die Definition von verschiedenen Vergleichsarten, die später auch instanziiert werden können (Kovacic, Schuetz, Schausberger, Sumereder & Schrefl, 2018).

Um eine Analysesituation zu erstellen werden hierbei die Modellelemente des eDFM referenziert. Für Kennzahlen werden Aggregate Measures verwendet, Level Predicates für Slice-Bedingungen, Aggregate Measure Predicates für Filter und Base Measure Predicates für Basis-Kennzahlen-Bedingungen. Für Score-Filter werden Comparative Measure Predicates verwendet.

3.3. Analysegraphen

Analysegraphen bilden die Abfolge von Analysesituationen ab (Schütz, Neumayr, Schrefl & Neuböck, 2016). Ein Analysegraph besteht aus Analysesituationen, die durch Navigationsschritte miteinander verbunden sind. Ein Navigationsschritt entspricht einer oder mehreren OLAP-Operationen, welche eine Analysesituation in eine andere umwandeln.

Durch Anwendung von **Navigationsoperatoren** in einem Navigationsschritt, kann eine Analysesituation modifiziert werden (Schütz et al., 2016). Ein Navigationsschritt repräsentiert die Navigation von einer

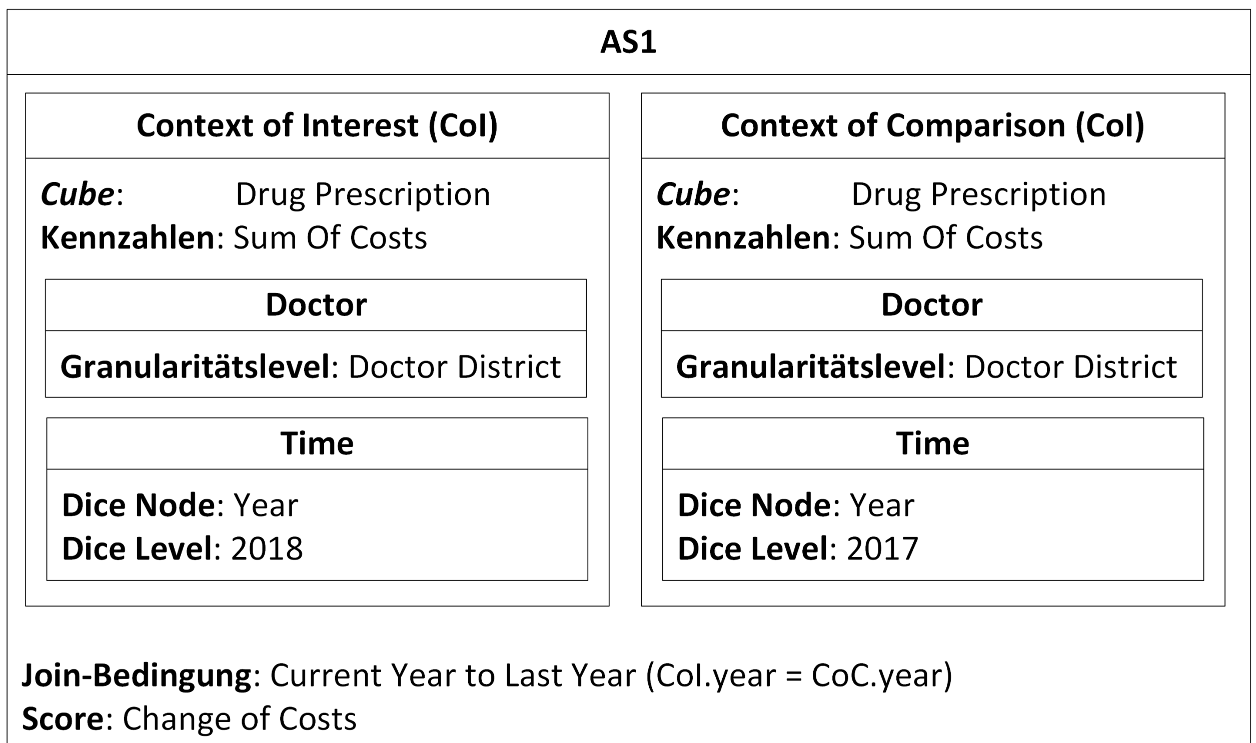


Abbildung 5: Beispiel für eine vergleichende Analysesituation

Quell-Analysesituation zu einer Ziel-Analysesituation. Der Unterschied zwischen der Quell- und der Ziel-Analysesituation wird durch einen Navigationsoperator und zugehörigem Parameter definiert (Neuböck et al., 2012). Beispiele für Navigationsoperatoren sind *addMeasure* (Kennzahl hinzfügen) oder *dropMeasure* (Kennzahl entfernen).

Es muss zwischen Analysegraph-Schemata und Analysegraph-Instanzen unterschieden werden. Beim Schema werden nicht ausschließlich konkrete Werte, sondern auch Variablen verwendet. Ein Analysegraph-Schema kann somit nicht direkt für Abfragen verwendet werden. Um eine Abfrage ausführen zu können, müssen die Variablen zuerst gebunden werden. Somit können die Schemata als Vorlagen für sich oft wiederholende Analysen verwendet werden (Neuböck et al., 2012). Eine Instanz bildet dem gegenüber den realen Ablauf einer Analyse ab, das Schema ist die Vorlage. Eine Analysegraph-Instanz ist als gerichteter Baum definiert, welcher Analysesituationen als Knoten und Navigationsschritte als Kanten enthält. (Neuböck, 2019)

In dieser Arbeit wird nur die Instanz-Ebene verwendet. Analysegraph-Schemata kommen nicht zur Anwendung. Die Navigationsoperatoren werden als Basis für die interaktive Verfeinerung der Abfragen verwendet.

Abbildung 6 zeigt ein Beispiel für einen Analysegraph über das Schema in Abbildung 4. Dimensionen, für die keine Dimensionspezifizierung angegeben wurde, werden nicht aggregiert. Ausgehend von der Analysesituation *AS1* wird eine Kennzahl mittels des Navigationsoperators *Refocus Measure* ausgetauscht. *AS2* wird modifiziert, in dem mit *Narrow Slice Condition* eine Slice Condition zur *Doctor*-Dimension hinzugefügt wird, damit im Ergebnis nur alte Ärzte berücksichtigt werden. Die daraus resultierende Analysesituation *AS3* wird wiederum durch eine Roll-Up-Operation modifiziert.

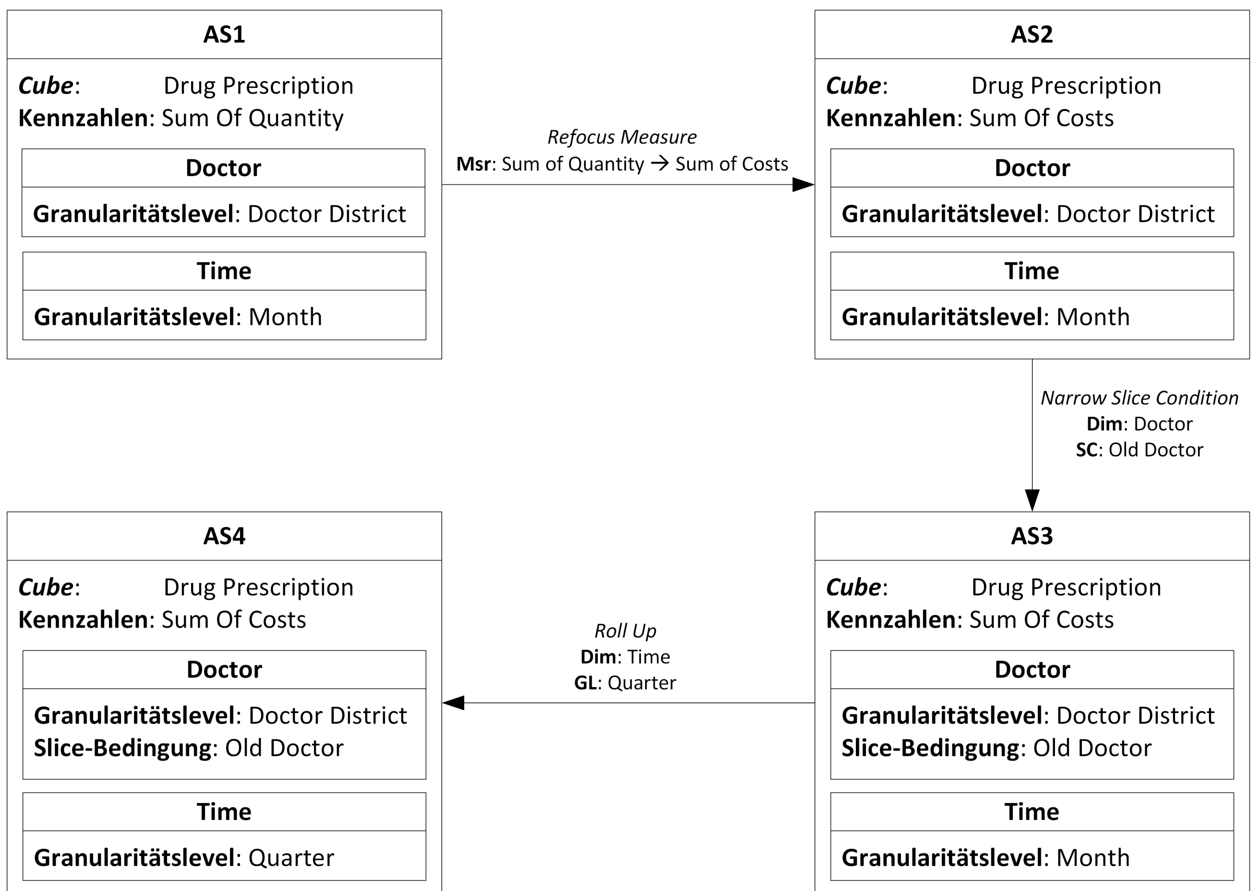


Abbildung 6: Beispiel für eine Analysegraph-Instanz

4. Verwendung der Benutzungsschnittstelle

In diesem Kapitel wird zuerst die Installation und Konfiguration der Anwendung erklärt. Danach wird beschrieben, wie ein Benutzer mit der Benutzungsschnittstelle interagieren kann.

4.1. Installation der Anwendung

Damit der Server-Teil der Anwendung ausgeführt werden kann, muss das Java Runtime Enviroment (JRE) ab Version 11 installiert sein. Weiters wird eine GraphDB-Instanz[1] benötigt. Die Benutzungsschnittstelle wurde mit Version 8.9 entwickelt und getestet. Für die Installation und Konfiguration von GraphDB wird auf die Dokumentation verwiesen (Ontotext, 2019a).

Für das Frontend wird ein Webbrowser mit Websocket-Unterstützung sowie Unterstützung für die Web Speech API mit den Funktionen *Speech recognition* und *Speech synthesis* benötigt. Zum Zeitpunkt des Schreibens dieser Arbeit erfüllt nur Google Chrome diese Voraussetzungen. Das Frontend könnte z. B. auch mit Mozilla Firefox verwendet werden, dann funktioniert allerdings die Spracheingabe nicht.

4.1.1. GraphDB

Nachdem GraphDB installiert und gestartet wurde, muss zunächst ein neues Repository erstellt werden. Hierzu muss eine *Repository ID* festgelegt werden. Die *Repository ID* muss später in der Anwendungskonfiguration angegeben werden. Für die anderen Eingabefelder können die Standardwerte beibehalten werden.

Nachdem das Repository erstellt wurde, muss WordNet[2] in das Repository importiert werden. Zu beachten ist, dass WordNet in den Named Graph `http://dke.jku.at/ida/similarity#wordnet` importiert werden muss.

In weiterer Folge muss ein Similarity Index erstellt werden. In der GraphDB-Workbench wählt man dazu den Menüpunkt *Explore > Similarity* aus, um einen neuen *Text Similarity Index* mit dem Namen `wordnet` zu erstellen. Als *Data query* wird die in Listing 8 angegebene SPARQL-Abfrage angegeben. Die Erstellung des Index kann je nach Leistung des Computers einige Minuten in Anspruch nehmen. Der Similarity Index ermöglicht es Elemente der multidimensionalen Schemata zu finden, auch wenn der Benutzer nicht die genaue Bezeichnung des Elements angibt.

Data-Warehouse-Schemata, mit denen gearbeitet werden soll, müssen zuvor in das GraphDB-Repository importiert werden. Ein importiertes Data-Warehouse-Schema muss deshalb mittels RDF repräsentiert werden. Das RDF-Schema hierfür wird im Anhang C beschrieben.

Um die Abfragen zu beschleunigen wird ein Graph erstellt, der Schema-Elemente mit WordNet-Begriffen über den Similarity-Index in Verbindung bringt. Um den Graph zu erstellen, müssen die Insert-Statements im Anhang A (Listing 21 bis 25) ausgeführt werden. Wenn sich die Schema-Daten ändern oder neue hinzukommen, muss der Graph neu erstellt werden.

[1] https://www.ontotext.com/products/graphdb/graphdb-free/
[2] http://wordnet-rdf.princeton.edu/static/wordnet.nt.gz

4.1.2. Anwendungskonfiguration

Mit Hilfe des Build Tools Gradle (Gradle, 2019) wurde eine JAR-Datei und Skripte zum Ausführen der Datei erzeugt. Um die Anwendung zu starten muss die Datei `server.bat` (Windows) bzw. `server` (Linux) aufgerufen werden. Das Frontend ist unter der URL `http://[Server-URL]:8080/frontend/index.html` erreichbar, wobei `Server-URL` von der Betriebssystemkonfiguration und Port von der Anwendungskonfiguration abhängig ist.

Zur Konfiguration der GraphDB-Anbindung muss eine Text-Datei mit dem Namen **application.properties** in dem Verzeichnis, in dem auch die JAR-Datei liegt, erstellt werden. Die Datei muss mit dem Inhalt aus Listing 1 befüllt werden, wobei `graphdb.remote.server-url` die URL zur GraphDB festlegt (z. B. `http://localhost:7200/`) und `graphdb.remote.repository-id` die ID des zuvor erstellten Repositories (z. B. `ida`). Der standardmäßig verwendete Port des Webfrontends (8080) kann durch Setzen der Einstellung `server.port=[Port]` in der Konfigurationsdatei geändert werden.

Listing 1: Anwendungskonfiguration application.properties

```
1 graphdb.remote.server-url=[server-url]
2 graphdb.remote.repository-id=[repository-id]
```

4.2. Komponenten der Benutzungsschnittstelle

Die Benutzungsschnittstelle ist webbasiert und besteht aus einer Seite (Single-Page-Webanwendung) mit mehreren Komponenten. Abbildung 7 zeigt die Benutzungsschnittstelle und ihre Komponenten.

In der **Verbindungs-Komponente** (IDA Connection) hat der Benutzer die Möglichkeit eine neue Verbindung und somit eine neue Abfrage zu starten. Weiters kann die Sprache, bevor eine Verbindung herstellt wurde, umgeschaltet werden. In dieser Arbeit wurde nur die Unterstützung für die englische Sprache implementiert.

Unter der Verbindungs-Komponente befindet sich die Komponente für die **Spracheinstellungen** (Voice Settings). Hier kann man eine Sprache für die Sprachausgabe auswählen und deren Lautstärke, Sprachgeschwindigkeit und Tonhöhe festlegen. Weiters ist es möglich festzulegen, ob die Spracherkennung automatisch gestartet werden soll, oder nur durch Klick auf die Mikrofon-Schaltfläche in der Eingabe-Komponente.

Die Verbindungs-Komponente und die Spracheinstellungs-Komponente können durch Klick auf die Komponentenüberschrift ein- bzw. ausgeklappt werden. Die Spracheinstellungs-Komponente ist standardmäßig eingeklappt. Die Verbindungs-Komponente ist standardmäßig eingeklappt, wenn eine Verbindung hergestellt wurde.

In der **Eingabe-Komponente** (Your Input) kann der Benutzer der Anwendung Anweisungen geben. Dies kann entweder durch Eingabe eines Textes oder durch Spracheingabe erfolgen. Wenn die automatische Spracherkennung nicht aktiv ist, muss diese erst durch einen Klick auf die Mikrofon-Schaltfläche gestartet werden. Durch einen erneuten Klick auf die Mikrofon-Schaltfläche kann die Spracheingabe wieder beendet werden. Die Spracheingabe wird auch automatisch beendet, wenn für eine bestimmte Zeit keine Spracheingabe erfolgt oder wenn die Spracherkennung das Ende der Spracheingabe erkennt.

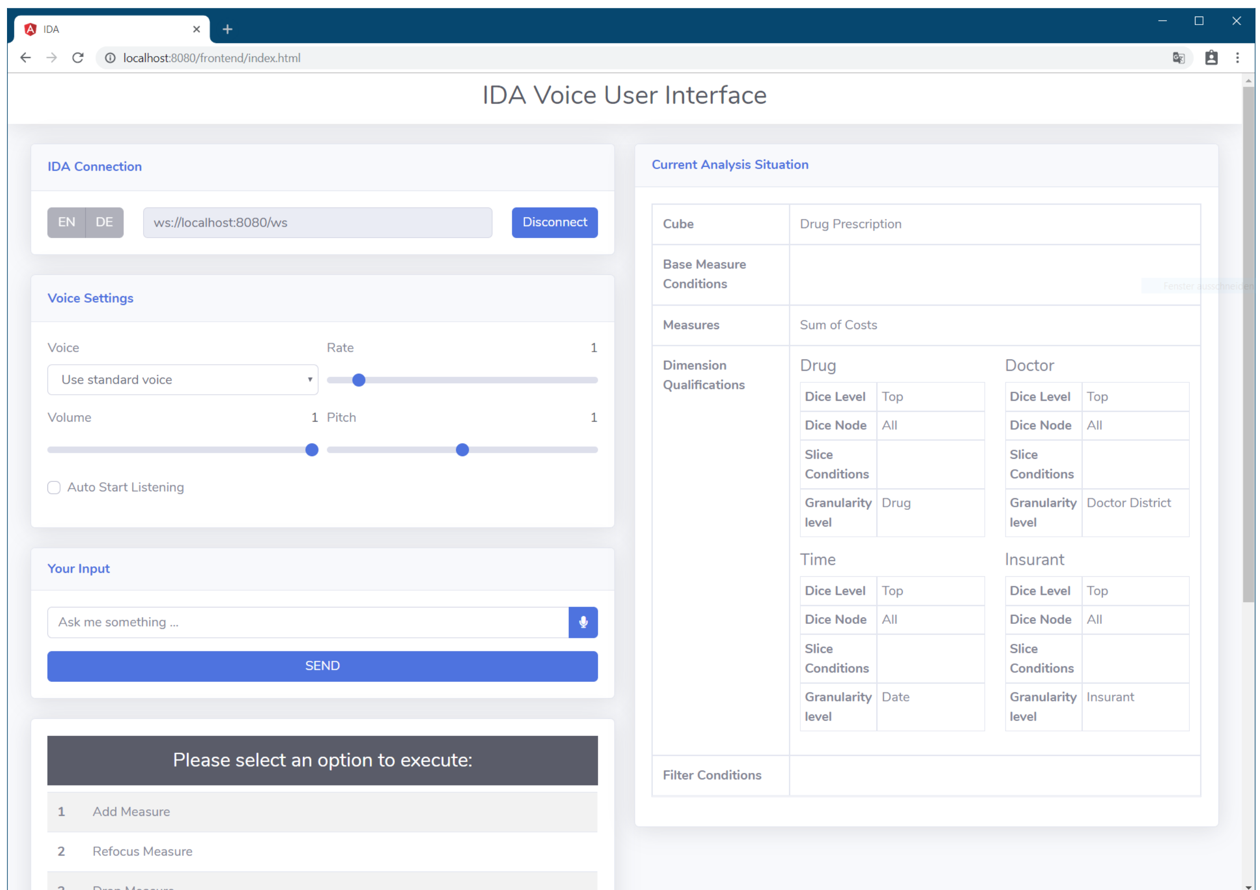


Abbildung 7: Screenshot der Benutzungsschnittstelle

Wenn die Spracherkennung aktiv ist, wechselt die Farbe der Mikrofon-Schaltfläche auf Rot. Falls der Browser die Spracherkennung nicht unterstützt, ist die Schaltfläche deaktiviert und grau eingefärbt. Beim ersten Start der Spracherkennung fragt der Browser um die Zustimmung des Benutzers, um auf das Mikrofon zugreifen zu können. Hier muss zugestimmt werden, ansonsten ist eine Spracherkennung nicht möglich. Wenn die Spracheingabe beendet wurde, wird der gesprochene Text im Textfeld eingefügt und kann vom Benutzer noch verändert werden. Die Eingabe kann durch Drücken der *Enter*-Taste (wenn sich das Eingabe-Textfeld im Fokus befindet) oder durch Klick auf die *Send*-Schaltfläche abgesendet werden. Diese Komponente ist nur sichtbar, wenn eine Verbindung mit dem Server besteht. Nach dem Absenden der Eingabe bleiben das Textfeld und die Schaltflächen dieser Komponente solange deaktiviert, bis eine Antwort vom Server eintrifft.

Unterhalb der Eingabe-Komponente werden in der **Anzeige-Komponente** die Antworten des Servers dargestellt. Hier wird angezeigt, welche Operationen ausgeführt bzw. welche Werte ausgewählt werden können. Details hierzu werden in Kapitel 4.3.2 erläutert. Die Ausgabe der Inhalte dieser Komponente erfolgt zusätzlich auch mittels Sprachausgabe.

In der rechten Spalte wird immer die **aktuelle Analysesituation** (Current Analysis Situation) dargestellt. Wenn das Fenster zu schmal ist, "rutscht" diese Komponente unter die Antwort-Komponente. Weiters existiert eine **Ergebnis-Komponente**. Diese wird unten auf der Seite über die gesamte Breite der Seite dargestellt, wenn die Abfrage ausgeführt wurde. In der Ergebnis-Komponente werden die Abfrageergebnisse in tabellarischer Form angezeigt. Durch Klick auf die Spaltenüberschriften kann die Tabelle sortiert werden. Tabellen 1 bis 3 beschreiben den Aufbau von Analysesituationen (Neuböck, 2019).

Tabelle 1: Dimensionsspezifizierung

Element	**Beschreibung**
Dimension	Die Dimension eines Cubes (z. B. Doctor).
Granularitätslevel	Das Granularitätslevel der Dimension legt das Aggregationslevel der Kennzahlen fest (z. B. Doctor District $\rightarrow$ Aggregiert alle Kennzahlen nach Doctor District). Um keine Aggregation vorzunehmen, wird *top* angegeben.
Dice Node	Der festgelegte Dice Node der Dimension und alle seine Unterknoten werden zur Analyse ausgewählt (z. B. 2018 $\rightarrow$ Wählt alle Datensätze aus dem Jahr 2018 aus). Um alle Nodes auszuwählen, wird *all* angegeben.
Dice Level	Das Dice Level entspricht dem Level des Dice Nodes (z. B. Year, weil im Dice Node ein Jahr festgelegt wurde). Wenn kein Dice Node ausgewählt wurde, wird *top* angegeben.
Slice-Bedingungen	Eine (möglicherweise leere) Menge von Level-Predicates wird verwendet, um Dice Nodes zu filtern.

Tabelle 2: Nicht vergleichende (Non-Comparative) Analysesituation

Element	**Beschreibung**
Cube	Der ausgewählte Cube (z. B. Drug Prescription).
Kennzahlen	Eine Menge von aggregierten Kennzahlen des Cubes (z. B. Sum of Costs).
Basis-Kennzahlen Bedingungen	Eine (möglicherweise leere) Menge von Base Measure Predicates, um das Ergebnis nach Basis-Kennzahlen einzuschränken (z. B. High Costs per Unit).
Filterbedingungen	Eine (möglicherweise leere) Menge von Aggregate Measure Predicates, um das Abfrageergebnis einzuschränken (z. B. High Drug Prescription Costs per Unit).
Dimensions-spezifizierung	Pro Dimension existiert eine Dimensionsqualifikation (siehe Tabelle 1).

Tabelle 3: Vergleichende (Comparative) Analysesituation

Element	**Beschreibung**
Context of Interest	Nicht-vergleichende Analysesituation, die den zu analysierenden Zustand repräsentiert (siehe Tabelle 2).
Context of Comparison	Nicht-vergleichende Analysesituation mit der verglichen werden soll (siehe Tabelle 2).
Join-Bedingungen	Eine Menge von Join-Bedingungen, die angeben, wie die Einträge von CoI und CoC verknüpft werden sollen (z. B. um einen Jahresvergleich durchzuführen).
Scores	Eine Menge von Comparative Measures, die angeben, wie die Kennzahlen verglichen werden sollen (z. B. Change of costs).
Score-Filter	Eine (möglicherweise leere) Menge von Comparative Measure Predicates, um das Abfrageergebnis auf Basis von Score-Werten zu filtern (z. B. Change of Costs over 10%).

4.3. Ablauf der Datenanalyse

Das Ablaufdiagramm in Abbildung 8 skizziert den Ablauf der interaktiven Datenanalyse mit Hilfe der Benutzungsschnittstelle. Zuerst wählt der Benutzer aus, welche Art von Abfrage er ausführen möchte bzw. welche Art von Analysesituation erstellt werden soll. Hierbei kann der Benutzer zwischen Comparative (vergleichend) und Non-Comparative (nicht vergleichend) auswählen.

Wenn die Art der Abfrage ausgewählt wurde, kann der Benutzer als Freitext (siehe Kapitel 4.3.1) angeben, wie die Analysesituation aussehen soll. Im Falle einer Comparative-Abfrage wird hier angegeben wie die Analysesituation für den Context of Interest befüllt werden soll.

Nach der Freitexteingabe wechselt die Anwendung in den dialogbasierten Modus (siehe Kapitel 4.3.2), in dem die Analysesituation weiter verfeinert werden kann. Falls zu Beginn die Art "Comparative" ausgewählt wurde, erhält der Benutzer zunächst die Möglichkeit die Analysesituation für den Context of Interest (CoI) zu verfeinern. Danach erfolgt die Verfeinerung des Context of Comparison (CoC) sowie der gesamten vergleichenden Analysesituation (Comparative), wobei der Benutzer zwischen den verschiedenen Analysesituationen wechseln kann. Im Falle einer nicht vergleichenden Abfrage (Non-Comparative) gibt es nur eine dialogbasierte Verfeinerung.

Nachdem die Verfeinerung abgeschlossen wurde, kann der Benutzer die Abfrage ausführen. Danach hat der Benutzer die Möglichkeit die Abfrage anzupassen und wechselt damit wieder in den dialogbasierten Modus.

4.3.1. Freitexteingabe

Nachdem eine Verbindung hergestellt wurde, wird der Benutzer zunächst aufgefordert die Art der Abfrage, die er durchführen möchte, anzugeben. Hierbei kann zwischen *Comparative* und *Non-Comparative* ausgewählt werden. Wie die Auswahl einer Option durchgeführt werden kann, ist in Kapitel 4.3.2 beschrieben.

Nachdem die Art ausgewählt wurde, wird der Benutzer dazu aufgefordert anzugeben, was er abfragen möchte. Beispielsweise könnte ein Benutzer sagen: „I am interested in the total costs per doctor district." Die Anwendung versucht dann so viele Schema-Elemente wie möglich anhand ihrer Bezeichnung (Label) im Text zu erkennen und die Analysesituation zu befüllen (Details in Kapitel 6.7). In diesem Beispiel würde die Anwendung erkennen, dass der Benutzer eine Abfrage zu Daten aus dem *Drug Prescription*-Cube ausführen möchte. Aufgrund der Erwähnung von "total costs" wird die aggregierte Kennzahl mit dem Label *Sum of Costs* ausgewählt. *Doctor district* wird als Granularitätslevel für die *Doctor*-Dimension festgelegt. Ein anderes Beispiel für eine mögliche Benutzereingabe wäre: „Give me the total costs per doctor in rural districts." In diesem Beispiel wird wiederum "total costs" ermittelt. Zudem wird durch die Erwähnung von "doctor in rural district" das Level Predicate "Doctor in Rural District" als Slice-Bedingung festgelegt. Danach wechselt die Anwendung in den dialogbasierten Modus, in dem die erkannte Analysesituation weiter angepasst werden kann. Es besteht jedoch keine Notwendigkeit Füllfloskeln wie "I am interested in" zu verwenden.

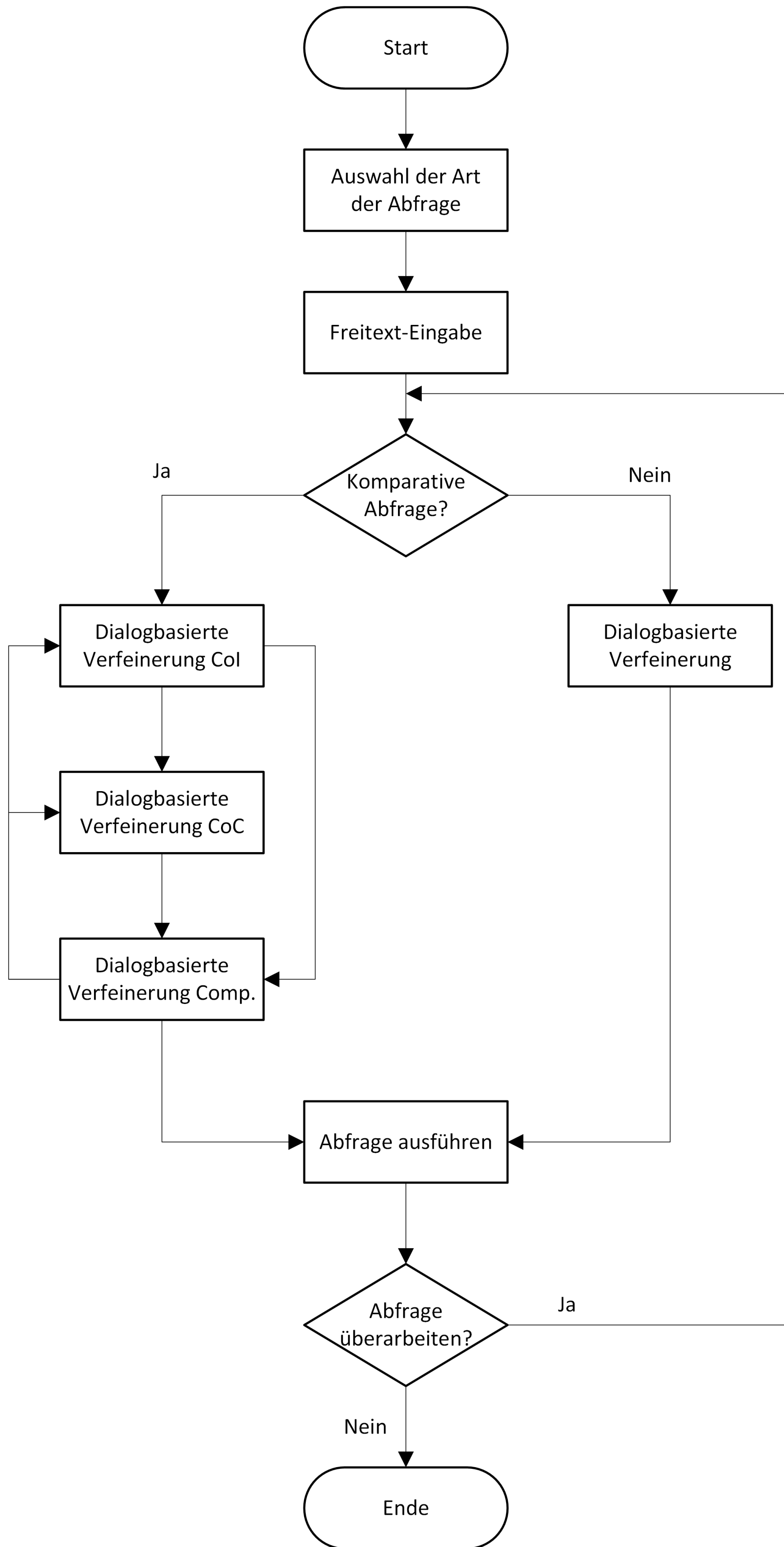


Abbildung 8: Programmablauf

Falls keine Elemente erkannt wurden oder der Benutzer einen leeren Text abgesendet hat, wechselt die Anwendung auch in den dialogbasierten Modus und präsentiert dem Benutzer zunächst eine Auswahl an möglichen Cubes, aus denen einer ausgewählt werden muss.

4.3.2. Dialogbasierte Verfeinerung

In der dialogbasierten Verfeinerung hat der Benutzer die Möglichkeit die Analysesituation seinen Wünschen anzupassen. Hierzu wird dem Benutzer eine Liste an möglichen Operationen angezeigt (wie zum Beispiel in Abbildung 7). Die Auswahl der Operationen, die dem Benutzer angezeigt werden, hängt von der aktuellen Analysesituation ab. Tabelle 4 listet alle unterstützten Operationen auf.

Tabelle 4: Unterstützte Operationen

Operation	**Beschreibung**
Select Cube	Legt einen Cube fest.
Add Measure	Fügt eine noch nicht ausgewählte Aggregate Measure hinzu.
Refocus Measure	Tauscht eine ausgewählte Aggregate Measure mit einer noch nicht ausgewählten Aggregate Measure aus.
Drop Measure	Entfernt eine ausgewählte Aggregate Measure.
Drill Down	Führt eine Drill-Down-Operation durch. Hierbei wird ein “detaillierteres” Granularitätslevel einer Dimension ausgewählt.
Roll Up	Führt eine Roll-Up-Operation durch. Hierbei wird ein “allgemeineres” Granularitätslevel einer Dimension ausgewählt.
Add Slice Condition	Fügt für eine bestimmte Dimension ein noch nicht ausgewähltes Level Predicate hinzu.
Refocus Slice Condition	Tauscht für eine bestimmte Dimension ein ausgewähltes Level Predicate mit einem noch nicht ausgewählten Level Predicate aus.
Drop Slice Condition	Entfernt für eine bestimmte Dimension ein ausgewähltes Level Predicate.
Add Filter	Fügt ein neues Aggregate Measure Predicate hinzu.
Refocus Filter	Tauscht ein ausgewähltes Aggregate Measure Predicate mit einem noch nicht ausgewählten Aggregate Measure Predicate aus.
Drop Filter	Entfernt ein ausgewähltes Aggregate Measure Predicate.
Add Base Measure Condition	Fügt ein noch nicht ausgewähltes Base Measure Predicate hinzu.

Weiter auf der nächsten Seite

Unterstützte Operationen - Fortsetzung

Operation	**Beschreibung**
Refocus Base Measure Condition	Tauscht ein ausgewähltes Base Measure Predicate mit einem noch nicht ausgewählten Base Measure Predicate aus.
Drop Base Measure Condition	Entfernt ein ausgewähltes Base Measure Predicate.
Add Dice Node	Legt für eine bestimmte Dimension einen Dice Node fest. Dadurch wird automatisch auch das dazugehörige Dice Level festgelegt.
Drop Dice Node	Entfernt für eine bestimmte Dimension den Dice Node und das Dice Level.
Add Join Condition	Fügt eine noch nicht ausgewählte Join Condition hinzu.
Drop Join Condition	Entfernt eine ausgewählte Join Condition.
Add Score	Fügt eine noch nicht ausgewählte Comparative Measure hinzu.
Drop Score	Entfernt eine ausgewählte Comparative Measure.
Add Score Filter	Fügt ein neues Comparative Measure Predicate hinzu.
Drop Score Filter	Entfernt ein ausgewähltes Comparative Measure Predicate.
Switch to Comparative Analysis Situation	Wechselt zur Comparative Analysesituation, um Join Conditions, Scores und Score Filter zu verwalten.
Switch to Context of Comparison	Wechselt zur Analysesituation, die den Context of Comparison repräsentiert.
Switch to Context of Interest	Wechselt zur Analysesituation, die den Context of Interest repräsentiert.
Execute Query	Führt die Abfrage mit der aktuellen Analysesituation aus.
Exit	Beendet die aktuelle Sitzung.

Es existieren verschiedene Möglichkeiten, wie der Benutzer eine Option aus der Liste möglicher Operationen auswählen kann. Eine Möglichkeit ist die Eingabe der **Zahl**, die links neben einer Option angezeigt wird. Weitere zahlenbasierte Eingaben sind:

- **Option [Zahl]** (Bsp. “option 1”)
- **Option [Zahl als Text]** (Bsp. “option one”)
- **[Ordinalzahl als Text] Option** (Bsp. “first option”)
- **[Ordinalzahl als Text]** (Bsp. “first”)

Es ist auch möglich den **Namen einer Option** (Bsp. "Add Measure") anzugeben. Bei manchen Operationen sind weitere Synonyme hinterlegt, sodass zum Beispiel anstatt "Refocus Measure" auch "Replace Measure" angegeben werden kann. Das System versteht auch leichte Abänderungen oder Rechtschreibfehler wie zum Beispiel "Add a measure" oder "Add meausre".

Sofern der Benutzer nicht nur die Operation, die er ausführen möchte, weiß, sondern auch schon den Namen des Elements, kann dieser auch gleich mit angegeben werden (Bsp. "Add Measure Sum of Quantity"). Dann versucht die Anwendung die Operation auszuführen, ohne dem Benutzer eine Auswahl von möglichen Werten anzuzeigen. Falls der Name des Elements ungültig ist, wird dem Benutzer wiederum eine Auswahl der möglichen Werte angezeigt. Wenn anhand der Benutzereingabe die gewünschte Operation nicht erkannt werden kann, wird wiederum die Liste mit den möglichen Operationen angezeigt.

Wenn nur eine Operation ohne Element ausgewählt wurde, wird eine Auswahl an möglichen Werten basierend auf der ausgewählten Operation angezeigt. Es gibt zwei verschiedene Darstellungsformen. Die erste Anzeigeart ist eine **einfache Liste** (Abbildung 9), die andere eine **zweispaltige Liste** (Abbildung 10), die verwendet wird, wenn zwei Werte ausgewählt werden müssen (z. B. bei Refocus-Operationen).

Abbildung 9: Einfache Liste

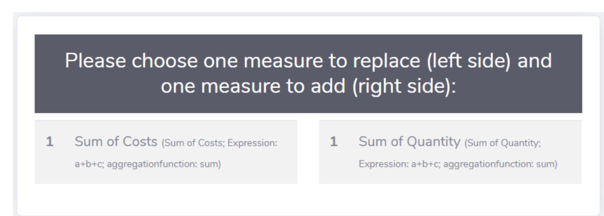


Abbildung 10: Zweispaltige Liste

Bei der einfachen Liste erfolgt die Auswahl der Optionen genauso wie bei der Auswahl von Operationen. Bei der zweispaltigen Liste erfolgt die Auswahl durch die Eingabe zweier Zahlen, die durch ein Komma getrennt sind (Bsp. "1,2"). Der erste Wert entspricht hierbei der Auswahl aus der linken Liste, der zweite Wert der Auswahl aus der rechten Liste. Weitere zahlenbasierte Eingaben, ähnlich wie bei einfachen Listen, sind:

- **[Left | Right] Option [Zahl]** (Bsp. "left option 1")
- **[Left | Right] Option [Zahl als Text]** (Bsp. "left option one")
- **[Ordinalzahl als Text] [Left | Right] Option** (Bsp. "first left option")
- **[Ordinalzahl als Text] [Left | Right]** (Bsp. "first left")

Es existiert auch hier die Möglichkeit, Elemente mittels Bezeichnung bzw. mittels Bezeichnung und Zahl auszuwählen. Bei einer zweispaltigen Liste muss aus jeder Liste genau ein Wert ausgewählt werden. Das bedeutet, dass der Benutzer immer einen Wert aus der linken und einen Wert aus der rechten Liste angeben muss. Beispielsätze zur Auswahl von zwei Werten mittels Angabe von Bezeichnungen sind „Replace sum of costs with right option two“ oder „first left and sum of quantity“. Die Reihenfolge, in der die Angaben erfolgen wird bei der Auswertung nicht berücksichtigt. Somit ist es z. B. auch möglich „right option one and left option one“ bzw. „Replace right option two with some of costs“ anzugeben.

Wenn anhand der Benutzereingabe der gewünschte Wert nicht erkannt werden kann, wird wiederum die Liste mit den möglichen Werten angezeigt. Falls man eine Operation abbrechen möchte, ist dies durch die Eingabe eines der folgenden Schlüsselwörter möglich: "abort", "go back", "cancel". Dann wird die aktuelle Operation abgebrochen und wieder die Liste der möglichen Operationen angezeigt.

Wenn ein gültiger Wert angegeben wurde, wird die Operation auf die aktuelle Analysesituation ausgeführt. In weiterer Folge wird in der Benutzungsoberfläche die Darstellung der Analysesituation aktualisiert und wieder die Liste mit den möglichen Operationen angezeigt.

Bei Operationen, die Werte einer Dimensionsqualifikation ändern (z. B. Roll Up oder hinzufügen einer Slice Condition) wird dem Benutzer zuerst eine Liste mit den Dimensionen präsentiert, aus der eine ausgewählt werden muss. In weiterer Folge werden dann basierend auf der ausgewählten Dimension die entsprechenden Werte der Dimension angezeigt, aus denen wiederum ausgewählt werden kann. Zum Beispiel könnte ein Benutzer eine Roll-Up-Operation in der *Insurant*-Dimension durchführen. Hierzu würde der Benutzer zunächst die Roll-Up-Operation auswählen. Daraufhin würden dem Benutzer alle Dimensionen angezeigt, bei denen eine Roll-Up-Operation möglich ist. Der Benutzer müsste dann die *Insurant*-Dimension auswählen. Daraufhin werden dem Benutzer die möglichen Granularitätslevel für den Roll-Up angezeigt, aus denen eines ausgewählt werden muss.

Wenn die Analysesituation den Wünschen des Benutzers entspricht, kann dieser die Abfrage ausführen, indem er die Operation zum Ausführen der Abfrage auswählt. Diese Option wird nur angezeigt, wenn zumindest der Cube und eine Kennzahl ausgewählt wurden. Das Ergebnis wird in tabellarischer Form dargestellt. Gleichzeitig werden in der Eingabe-Komponente das Eingabefeld und die *Send*-Schaltfläche ausgeblendet und die *Revise Query*-Schaltfläche eingeblendet. Durch Klick auf diese Schaltlfäche werden wieder die möglichen Operationen angezeigt und die Analysesituation kann weiter angepasst werden.

5. Systemüberblick

Im Folgenden wird ein Überblick über die Architektur und den Ablauf der Anwendung gegeben. Der gesamte Quellcode der Anwendung kann online[3] abgerufen werden. Kapitel 5.1 beschreibt die Architektur (Abbildung 11) der Anwendung und welche Komponenten für welche Aufgaben zuständig sind. Kapitel 5.2 beschreibt den Ablauf und die Abfolge der Kommunikation zwischen den Komponenten anhand eines Beispiels.

5.1. Architektur

Wie in Abbildung 11 zu sehen ist, interagiert der Benutzer mit der Anwendung über eine webbasierte Benutzungsschnittstelle. Die Kommunikation der Client-Anwendung erfolgt über WebSockets. Somit wäre es auch möglich andere Arten von Clients mit dem selben Backend zu implementieren.

Der Server besteht aus einer *GraphDB*-Instanz, welche das Data-Warehouse-Schema im RDF-Format beinhaltet und Schema-Elemente mittels Similarity Index WordNet-Begriffen zuweist. RDF wurde als Repräsentationssprache gewählt, weil es der Standard für Wissensrepräsentation ist und bestehende Knowledge Graphs meist in diesem Format veröffentlicht werden. Die Verwendung von RDF für die Repräsentation des multidimensionalen Schemas erlaubt deshalb die einfache Verknüpfung des Schemas mit bestehenden Knowledge Graphs. Das so repräsentierte Wissen dient als Grundlage für die Identifizierung von selektierten Modellelementen bei der Abfrageerstellung mittels natürlichsprachlichen Ausdrücken. WordNet bietet sich aufgrund seiner vergleichsweise geringen Größe, dem einfachen Aufbau und der Fokussierung auf Vokabular für die Verwendung im Natural Language Processing an. Um die Identifizierung zu verbessern, könnten auch andere Knowledge Graphs, wie Wikidata und DBpedia, verwendet werden. Grundsätzlich könnten auch andere RDF-Datenbanken verwendet werden, wobei GraphDB gewählt wurde, weil hier ein Similarity-Index bereits im Funktionsumfang enthalten ist.

Die *DWH(Data Warehouse)-Abfrage-Komponente*, welche nicht Teil dieser Masterarbeit ist, wird benötigt, um Abfragen basierend auf der erstellten Analysesituation ausführen zu können. Hierzu wird die Analysesituation an die Abfrage-Komponente übermittelt, welche daraus eine Abfrage generiert und das Data Warehouse abfragt. Die Ergebnisse werden von der Abfrage-Komponente im CSV-Format zurückgeliefert. Zu Demonstrationszwecken wurde eine DWH-Komponente implementiert, die allerdings auf kein Data Warehouse zugreift, sondern nur zufällig generierte Daten im CSV-Format zurück liefert.

Das im Rahmen dieser Arbeit entwickelte *Backend* besteht wiederum aus mehreren Teilen. Der *WebSocket-Server* übernimmt die Kommunikation mit dem Frontend. Wenn eine neue Nachricht vom Frontend eintrifft, leitet der WebSocket-Server die Nachricht an die State-Chart-XML-Komponente weiter. Wenn Nachrichten an das Frontend gesendet werden müssen, zum Beispiel, wenn dem Benutzer eine Liste von möglichen Operationen angezeigt werden soll, informiert die State-Chart-XML-Komponente den WebSocket-Server mittels Ereignissen darüber. In weiterer Folge sendet der WebSocket-Server die Nachricht an das Frontend. Die Implementierung mittels WebSocket wurde einer Implementierung mittels REST-Schnittstelle vorgezogen, weil die asynchrone Kommunikation, die für das Web-Frontend notwendig ist, einfacher umzusetzen war. Details zum WebSocket-Server folgen in Kapitel 6.1.

[3] https://github.com/jku-win-dke/mt1902-ida abgerufen werden.

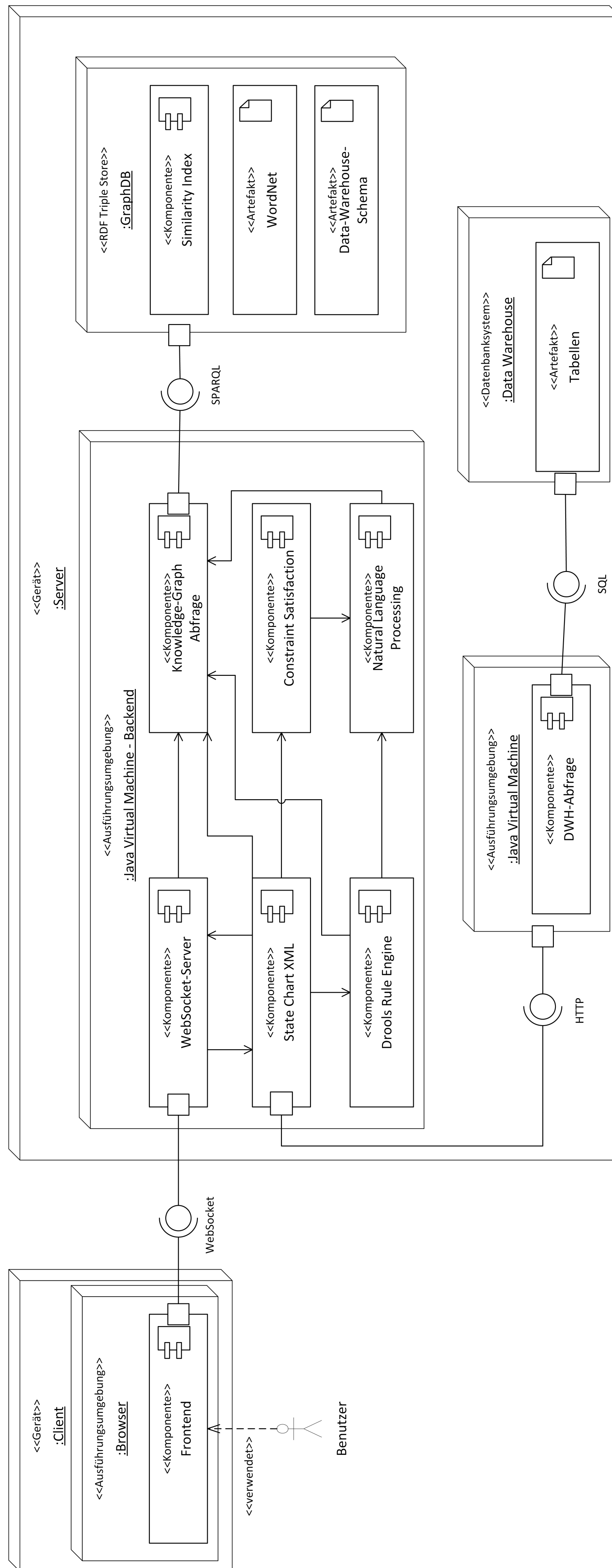


Abbildung 11: Architektur

Die *Knowledge-Graph-Abfrage*-Komponente übernimmt die Kommunikation mit der GraphDB. Die Komponente sendet hierfür SPARQL-Abfragen an GraphDB. Hierdurch können die anderen Komponenten auf das Data-Warehouse-Schema, WordNet und den Similarity-Index zugreifen. Der WebSocket-Server, zum Beispiel, greift auf die Knowledge-Graph-Abfrage-Komponente zu, um die Bezeichnungen der Elemente abzurufen, wenn eine Analysesituation zur Anzeige an das Frontend gesendet werden soll. Details zur Knowledge-Graph-Abfrage-Komponente folgen in Kapitel 6.2.

Die *State-Chart-XML*-Komponente (SCXML) bildet den gesamten Ablauf einer Benutzersitzung ab. Wenn ein Benutzer eine Abfrage ausführen möchte, kommuniziert die Komponente über HTTP mit der DWH-Abfrage-Komponente. Um später Änderungen am Ablauf der Benutzerinteraktion einfach durchführen zu können, wurde der Ablauf der Benutzerinteraktion deklarativ erfasst. SCXML wurde gewählt, weil es ein offener Web-Standard ist und eine leichtgewichtige Implementierung existiert. Details zur State-Chart-XML-Komponente folgen in Kapitel 6.3.

Die Komponente *Drools Rule Engine* verwendet Regeln, um die möglichen Operationen und Werte basierend auf der aktuellen Analysesituation zu ermitteln. Weiters wird diese verwendet, um die ausgewählten Operationen und Werte basierend auf der Benutzereingabe zu ermitteln und entsprechende Operationen auszuführen und Werte zu setzen. Diese Komponente greift auf die Knowledge-Graph-Abfrage-Komponente zu, um die benötigten Schema-Elemente abzurufen. Auf die Natural-Language-Komponente wird zugegriffen, um die Benutzereingabe in Wortgruppen umzuwandeln und um Ähnlichkeiten zwischen den Wortgruppen und den möglichen Operationen bzw. Werten zu ermitteln. Die Drools-Rule-Engine-Komponente wird von der State-Chart-XML-Komponente verwendet. Der Grund für die Formulierung der Programmlogik mittels deklarativer Regeln ist wiederum die Konfigurier- und Erweiterbarkeit. Details folgen in Kapitel 6.4.

Die Komponente *Natural Language Processing* wird, wie bereits beschrieben, verwendet um Benutzereingaben zu Schema-Elementen bzw. Operationen zuzuordnen. Hierzu werden aus den Benutzereingabe Wortgruppen mittels Stanford CoreNLP extrahiert. Diese Wortgruppen werden verwendet um mit dem Similarity-Index von GraphDB, auf den über die Knowledge-Graph-Abfrage-Komponente zugegriffen wird, Schema-Elemente zu finden. Des Weiteren berechnet diese Komponenten Distanzmetriken zwischen Strings, um diese auf Ähnlichkeit zu überprüfen. Details hierzu werden in Kapitel 6.6 beschrieben.

Auf die *Constraint-Satisfaction*-Komponente wird in Kapitel 6.7 eingegangen. Diese Komponente ist dafür zuständig, aus der Freitext-Eingabe des Benutzers eine Analysesituation zu generieren. Hierzu wird auch auf die Natural-Language-Processing-Komponente zugegriffen. Die Formulierung als Constraint-Satisfaction-Problem hat den Vorteil, dass die Auswahlregeln explizit niedergeschrieben werden. Als Framework wird Optaplanner verwendet, welches auf Drools aufbaut.

5.2. Ablauf

Dieses Kapitel beschreibt den Ablauf und die Kommunikation der Komponenten anhand eines Beispiels. Der Ablauf der Anwendung ist in den Abbildungen 8 und 16 ersichtlich.

Nachdem die Benutzerin das Frontend in einem Google Chrome-Browser geöffnet hat, stellt sie eine Verbindung mit dem WebSocket-Server her. Daraufhin wird eine neue Sitzung erstellt und eine neue State-Chart-XML (SCXML) Instanz erstellt. Die SCXML-Instanz beginnt zu laufen und informiert den

WebSocket-Server mit einem Ereignis darüber, dass eine Liste mit möglichen Abfragearten an das Frontend gesendet werden soll. Die Benutzerin entscheidet sich für eine Non-Comparative Analysesituation und gibt hierzu "non comparative" ein und sendet diese Nachricht ab. Anstatt "non comparative" einzugeben, wäre es zum Beispiel auch möglich "Option one" anzugeben. Details zu den verschiedenen Möglichkeiten werden in Kapitel 4.3.2 erklärt.

Der WebSocket-Server empfängt die von der Benutzerin versendete Nachricht und löst ein "userInput"-Ereignis in der SCXML-Instanz der Benutzerin aus. Die SCXML-Instanz ruft daraufhin die Drools Rule Engine auf, um die Regeln zur Bestimmung der ausgewählten Art auszuführen. In den Regeln werden Klassen der Natural Languge Processing (NLP) Komponente aufgerufen, um mittels Levensthein- und Jaro-Winkler-Metriken die Art herauszufinden, deren Bezeichnung am ähnlichsten zur Benutzereingabe ist. In diesem Beispiel übermittelt die Drools-Rule-Engine, dass die Benutzerin die Non-Comparative Art (Similarity Score 0.9644) ausgewählt hat. Daraufhin erstellt die SCXML-Instanz eine neue Non-Comparative Analysesituation und geht zum nächsten Schritt über, in dem die Benutzerin gefragt wird, wie die Analysesituation befüllt werden soll.

Die Benutzerin möchte die gesamten Kosten pro Arzt-Bezirk abfragen. Dazu spricht sie folgenden Text, der in das Abfragetextfeld eingefügt wird: „I am interested in the total costs per doctor district." Nachdem sie die Abfrage abgesendet hat, löst die WebSocket-Server-Komponente wiederum ein "userInput"-Ereignis in der SCML-Instanz aus und diese beginnt mit der Analyse des Satzes. Hierzu wird der Satz der NLP-Komponente übergeben, die den Satz zunächst in Wortgruppen aufteilt. In diesem Beispiel sind die gefundenen Wortgruppen "doctor district", "total costs", "interested in costs" und "costs per district". Diese Wortgruppen werden mit Hilfe des GraphDB Similarity-Index über die Knowledge-Graph-Abfrage-Komponente auf Ähnlichkeiten zu Schema-Elementen überprüft. Die Abfrage des Similarity Index liefert zwei Ergebnisse, die in Tabelle 5 aufgelistet sind. Diese Ergebnisse werden an die Constraint-Satisfaction-Komponente übergeben, welche versucht, daraus eine gültige Analysesituation zu erzeugen. In diesem Beispiel gelingt dies. Es wurde eine Analysesituation für den Cube *Drug Prescription* mit der Kennzahl *Sum Of Costs* und dem Granularitätslevel *Doctor District* in der Doctor-Dimension erstellt.

Tabelle 5: Ablauf-Beispiel: Ergebnis der Similarity-Index-Abfrage

Wortgruppe	**Cube**	**Dimension**	**Typ**	**Element**	**Score**
doctor district	DrugPrescription	Doctor	Level	Doctor District	0.4983
total costs	DrugPrescription		Aggregate Measure	Sum Of Costs	0.5172

Die SCXML-Instanz wechselt daraufhin in den Guided-Modus. Sie SCXML-Instanz ruft zuerst die Drools-Engine auf, um die möglichen Operationen zu ermitteln. Diese werden wiederum an die Benutzerin gesendet. Gleichzeitig wird die WebSocket-Komponente auch über die erstellte Analysesituation informiert, welche daraufhin die Bezeichnungen der Elemente über die Knowledge-Graph-Abfrage-Komponente abfragt und diese auch an das Frontend sendet.

Die Benutzerin hat jetzt die Möglichkeit die Analysesituation ihren Wünschen nach anzupassen. Hierzu kann sie eine Operation auswählen. Mit Hilfe der Drools-Engine wird unter Verwendung der NLP-Komponente die ausgewählte Operation ermittelt. Basierend auf der ausgewählten Operation werden auch wieder mit der Drools-Engine die möglichen Werte ermittelt, aus der die Benutzerin auswählen kann. Nachdem die

Benutzerin einen Wert ausgewählt hat wird wiederum mit der Drools-Engine der ausgewählte Wert ermittelt und die zuvor ausgewählte Operation mit dem ausgewählten Wert durchgeführt.

Wenn die Benutzerin mit der Analysesituation zufrieden ist, kann sie die Abfrage ausführen, indem sie die entsprechende Operation eingibt. Hierzu sendet die SCXML-Komponente die Analysesituation an die Abfrage-Komponente, welche daraus eine Abfrage generiert und durchführt. Für die zurückgelieferten Abfrageergebnisse verwendet die SCXML-Komponente die Abfrage-Komponente, um die Bezeichnungen der Elemente im Abfrageergebnis abzufragen und damit zu ersetzen. Daraufhin wird dem WebSocket-Server über ein Ereignis mitgeteilt, dass die Abfrageergebnisse an die Benutzerin gesendet werden sollen, welche diese über die WebSocket-Verbindung an das Frontend sendet, wo die Abfrageergebnisse in tabellarischer Form dargestellt werden.

6. Implementierungsdetails Backend

Das Backend wurde mit dem Spring Boot Framework (Pivotal Software, 2019) entwickelt, da dieses bereits Funktionen für Anwendungskonfiguration, Dependency Injection[4], Webserver und WebSockets enthält. Das Backend stellt alle benötigten Funktionen für die Benutzungsschnittstelle zur Verfügung.

Zur Verwaltung der Abhängigkeiten von Drittbibliotheken wurde bei der Entwicklung des Backends das Build Tool Gradle (Gradle, 2019) eingesetzt. Die Anwendung selbst wurde in mehrere Module aufgeteilt, wobei zwischen zwei Bereichen unterschieden wird: "app" und "engine".

Der Bereich "engine" enthält die Kern-Funktionalitäten der Anwendung. Darunter fallen die Module CSP (Constraint Satisfaction), Data (Datenzugriff), NLP (Natural Language Processing), Rules (Drools), SCXML (State Chart XML), Web (WebSockets) und Shared (Klassen, die in vielen anderen Modulen benötigt werden). Der Bereich "app" enthält die Module Server (enthält die Main-Klasse und Server-Ressourcen) und Ruleset (enthält alle Drools-Regeln und CSP-Regeln).

Die Komponenten des Backends wurden bereits in Kapitel 5 beschrieben. Tabelle 6 gibt noch einmal einen Überblick darüber, welche Komponente wofür verwendet wird. In den folgenden Unterkapiteln werden einzelne Teile des Backends genauer beschrieben.

Tabelle 6: Aufgaben der Komponenten

WebSockets	Die Kommunikation des Frontends mit dem Backend erfolgt über WebSockets.
Knowledge-Graph-Abfrage	Die Abfragen von Schema-Elementen erfolgt mittels SPARQL-Abfragen. Diese Komponente stellt Methoden dafür zur Verfügung.
State Chart XML	Der gesamte Ablauf wird in Form eines Zustandsdiagramms abgebildet.
Drools	Drools-Regeln werden vom State Chart XML aus ausgeführt, um mögliche Werte zu ermitteln und um das ausgewählte Element in einer Benutzereingabe zu erkennen, um entsprechende Aktionen auszuführen.
Natural Language Processing	Stanford CoreNLP wird verwendet, um Wortgruppen in Benutzereingaben zu finden. Weiters wird der GraphDB Similarity-Index verwendet, um Schema-Elemente zu finden, deren Bezeichnungen ähnlich zu den Wortgruppen sind. Auch String-Similarity Metriken kommen hier zum Einsatz.
Constraint Satisfaction	Erstellt anhand aus der Benutzereingabe extrahierten Wortgruppen eine Analysesituation.

6.1. Sitzungen und WebSockets

Die Kommunikation zwischen dem Backend und dem Frontend erfolgt asynchron über eine WebSocket-Verbindung, die eine bidirektionale Verbindung zwischen Backend und Frontend ermöglicht. Als Kommunikationsprotokoll wird STOMP (Simple Text Orientated Messaging Protocol) verwendet. STOMP ist ein interoperables Format zur Nachrichtenübertragung (Mesnil, 2012). Spring Boot unterstützt WebSockets und das STOMP-Format, sodass beim Senden und Empfangen von Nachrichten die Nachricht nicht

[4]https://docs.spring.io/spring/docs/5.1.5.RELEASE/spring-framework-reference/core.html#beans-introduction

manuell in das STOMP-Format konvertiert bzw. vom STOMP-Format in ein Java-Objekt konvertiert werden muss.

Wenn eine neue WebSocket-Verbindung hergestellt wurde, wird von Spring eine neue Sitzungs-ID generiert. Diese Sitzungs-ID muss beim Versenden einer Nachricht angegeben werden, um den Empfänger der Nachricht zu spezifizieren. Weiters wird die Sitzungs-ID benötigt, um die Nachricht auf dem der Sitzung zugeordneten State-Chart-XML-Instanz anzuwenden.

Zu jeder Sitzung werden folgende Informationen, auf die später zugegriffen wird, in einem eigenen Objekt gespeichert: Sitzungs-ID, Sprache, Analysesituation, aktuell zu bearbeitende Analysesituation (im Falle einer Comparative-Analysesituation), Anzeigendaten (z. B. Liste mit möglichen Operationen), letztes Abfrageergebnis, letzte Benutzereingabe, ausgewählte Operation, zusätzliche Daten (z. B. ausgewählte Dimension bei einer Roll-Up Operation) und die State-Chart-XML-Instanz.

Tabelle 7 zeigt die Liste der Warteschlangen, bei denen sich das Frontend registrieren kann, um auf Änderungen von Anzeigedaten, Analysesituationen und Abfrageergebnissen warten zu können. Tabelle 8 listet die Endpoints auf, an die das Frontend Nachrichten senden kann.

Tabelle 7: WebSocket-Warteschlangen

Warteschlange	Art der Nachrichten
/queue/display	Nachrichten, die die Anzeigedaten enthalten, die in der Anzeige-Komponente dargestellt werden
/queue/as	Nachrichten mit Analysesituationen
/queue/result	Nachrichten mit Abfrageergebnissen

Tabelle 8: WebSocket Endpoints

Endpoint	Aktion
/app/start	Erstellt eine neue Sitzung und startet eine State-Chart-XML-Instanz.
/app/input	Löst ein *userInput*-Ereignis aus.
/app/reviseQuery	Löst ein *reviseQuery*-Ereignis aus.

Das Spring-Framework serialisiert die Java-Objekte im JSON-Format, um diese als Nachricht versenden zu können. Das Spring-Framework erstellt aus den empfangenen Nachrichten im JSON-Format wiederum Java-Objekte. Listing 2 zeigt den Inhalt einer Start-Nachricht, die die Sprache der Sitzung festlegt (in diesem Fall Englisch) und eine State-Chart-XML-Instanz startet.

Listing 2: WebSocket: Start-Nachricht

```
1 { "locale": "en" }
```

Listing 3 zeigt den Inhalt einer Input-Nachricht, die die Benutzereingabe enthält. Dies löst am Backend ein *userInput*-Ereignis in der State Machine aus. Die ReviseQuery-Nachricht enthält keinen Inhalt.

Listing 3: WebSocket: Input-Nachricht

```
{ "userInput": "non comparative" }
```

Das Frontend empfängt eine Display-Nachricht wie in Listing 4 über die Display-Warteschlange. Je nach Art der Anzeigedaten kann der Inhalt von `display` anders aussehen. `type` gibt den Typ des Inhalts von `display` an. Abbildung 12 zeigt alle möglichen Display-Typen. `MessageDisplay` dient zum Anzeigen einer Nachricht, `Error Display` dient zum Anzeigen einer Fehlermeldung. `ExitDisplay` wird verwendet, um dem Frontend mitzuteilen die Verbindung zu beenden. `ListDisplay` und und `TwoListDisplay` liefern Daten zum Anzeigen in einer einfachen Liste bzw. zweispaltigen Liste. `Displayable` enthält eine ID (z. B. IRI eines Elements), die Bezeichnung sowie optional weitere Details.

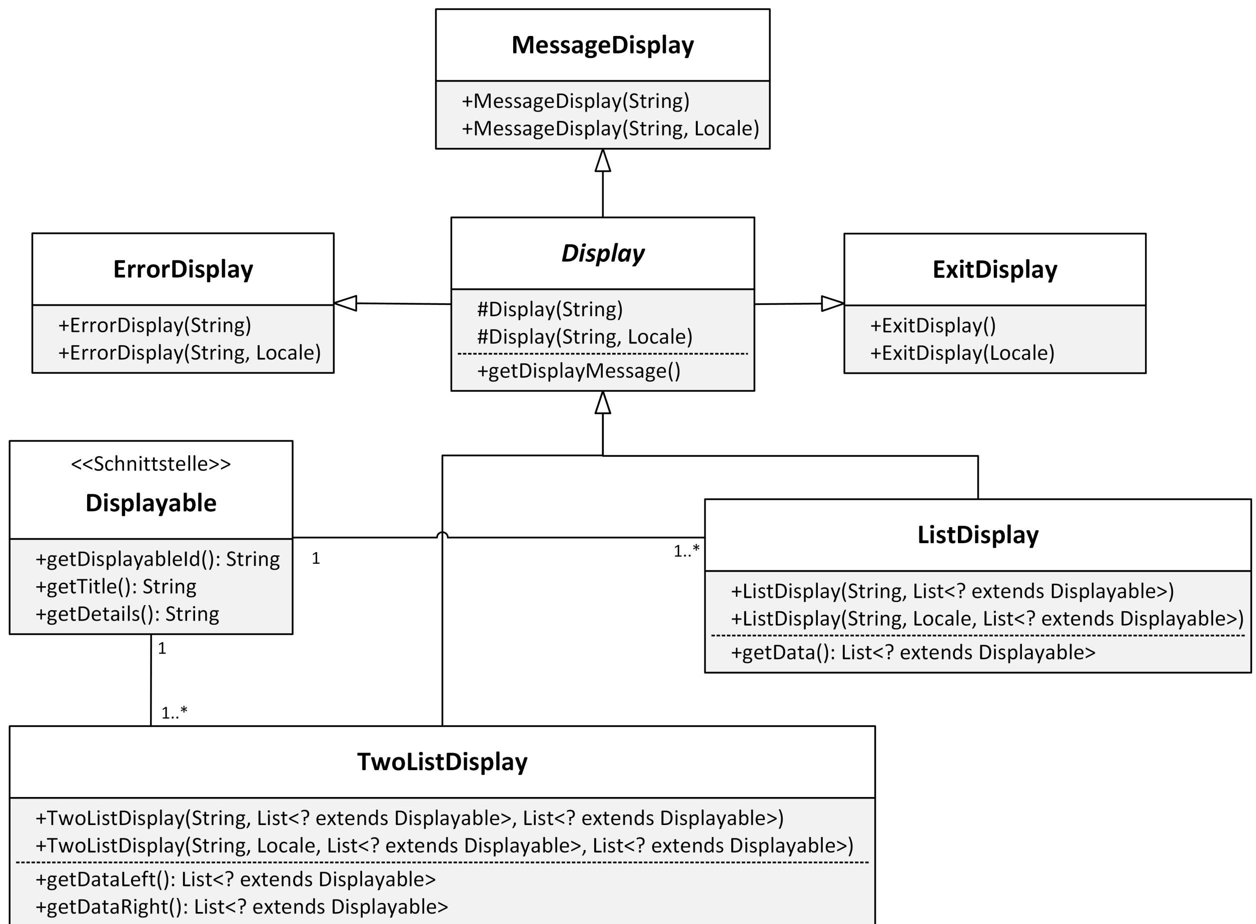


Abbildung 12: Klassendiagramm: Display

Listing 4: WebSocket: Display-Nachricht

```
{
  "type": "ListDisplay",
  "display": {
    "displayMessage": "Please choose one of the following measures to add:",
    "data": [
      {
        "displayableId": "http://www.example.org/drugs#SumCostsMeasure",
        "title": "Sum of Costs",
        "details": "Sum of Costs; Expression: a+b+c; aggregationfunction: sum"
      },
      {
```

```
        "displayableId": "http://www.example.org/drugs#SumQuantityMeasure",
        "title": "Sum of Quantity",
        "details": "Sum of Quantity; Expression: a+b+c; aggregationfunction:
            sum"
      }
    ]
  }
}
```

Das Frontend empfängt eine Analysesituation-Nachricht wie in Listing 5 über die AS-Warteschlange und enthält die aktuelle Analysesituation. Abbildung 13 zeigt das Klassendiagramm für Analysesituationen. Für Dimensionsqualifikationen, vergleichende und nicht vergleichende Analysesituationen existieren jeweils eine generische Klasse (`GenericDimensionQualification`, `GenericComparativeAnalysisSituation`, `GenericNonComparativeAnalysisSituation`), sowie eine Klasse für die Anzeige im Frontend (`DimensionQualificationDisplay`, `ComparativeAnalysisSituationDisplay`, `NonComparativeAnalysisSituationDisplay`) und eine für die Verarbeitung im Backend (`DimensionQualification`, `ComparativeAnalysisSituation`, `NonComparativeAnalysisSituation`). Der Unterschied besteht darin, dass in den Klassen für die Anzeige im Frontend die Elemente als Labels (siehe Abbildung 15) repräsentiert werden, damit im Frontend die Bezeichnung des Elements dargestellt werden kann. In den Klassen für das Backend werden Elemente als IRIs repräsentiert. Wenn eine Analysesituation an das Frontend gesendet wird, konvertiert die WebSocket-Komponente die Analysesituation in eine Display-Analysesituation, indem für alle Elemente die Labels abgerufen werden.

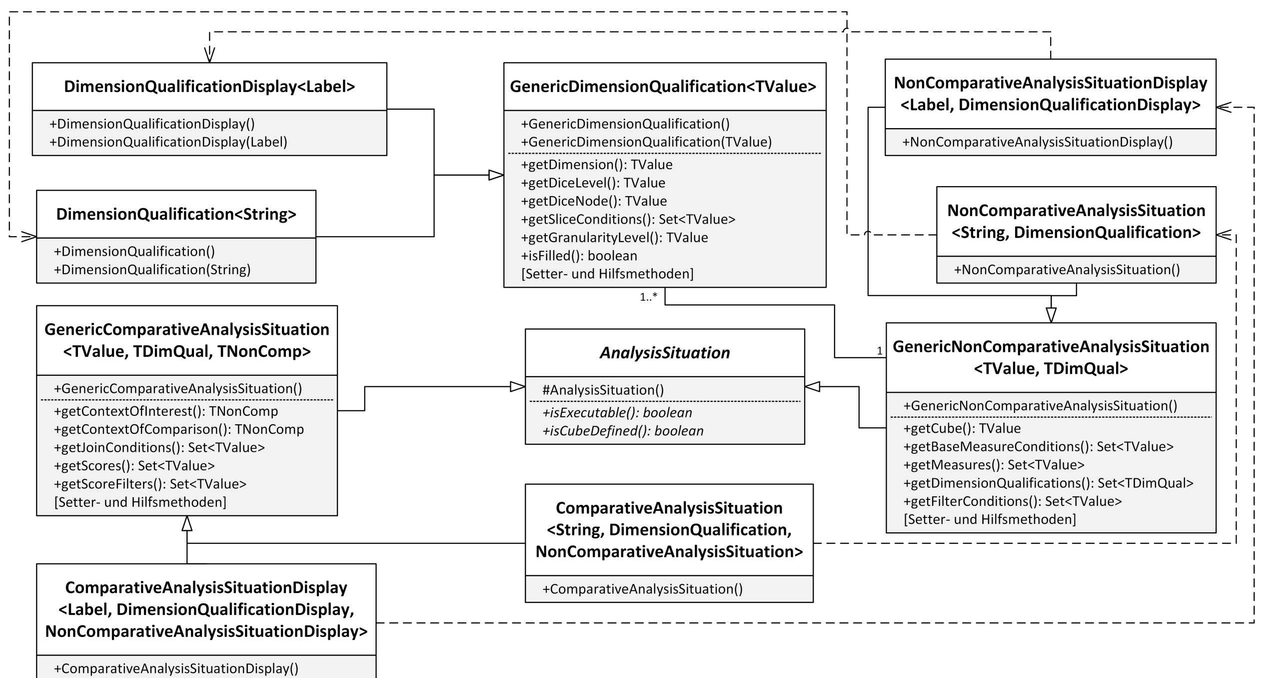


Abbildung 13: Klassendiagramm: Analysesituationen

Listing 5: WebSocket: Analysesituation-Nachricht

```
{
  "cube": {
    "displayableId": "http://www.example.org/drugs#DrugPrescriptionCube",
    "title": "Drug Prescription",
    "details": "Drug Prescription Cube"
```

```
  },
  "baseMeasureConditions": [],
  "measures": [],
  "dimensionQualifications": [
    {
      "dimension": {
        "displayableId": "http://www.example.org/drugs#DoctorDimension",
        "title": "Doctor",
        "details": "Doctor"
      },
      "diceLevel": {
        "displayableId": "http://dke.jku.at/ida/constants#top",
        "title": "Top",
        "details": null
      },
      "diceNode": {
        "displayableId": "http://dke.jku.at/ida/constants#all",
        "title": "All",
        "details": null
      },
      "sliceConditions": [],
      "granularityLevel": {
        "displayableId":
            "http://www.example.org/drugs#DoctorDimensionDoctorDistrictLevel",
        "title": "Doctor District",
        "details": null
      }
    },
    ...
  ],
  "filterConditions": []
}
```

Abfrageergebnisse werden über die Result-Warteschlange an das Frontend versendet. Das Ergebnis wird im CSV-Format übertragen, wobei die erste Zeile die Spaltenüberschriften abbildet.

6.2. Knowledge-Graph-Abfrage

Zur Abfrage des Knowledge Graphs, der in GraphDB abgelegt ist, wird die RDF4J-Bibliothek verwendet, über die eine Verbindung mit der GraphDB-Instanz aufgebaut wird und SPARQL-Abfragen ausgeführt werden. Die Konfiguration der Verbindung zur GraphDB wurde bereits in Kapitel 4.1.2 beschrieben.

Für jeden Typ von Schema-Element (z. B. AggregateMeasure) wurde eine eigene Repository-Klasse erstellt. Da viele Repositories die gleichen Abfragemethoden haben, wurden Basis-Klassen erstellt, von denen dann die Repository-Klassen für die einzelnen Typen erben (z. B. erbt die Klasse `AggregateMeasureRepository` von `SimpleCubeElementRepository`). Abbildung 14 zeigt das Klassendiagramm der Basis-Repositories. Der Konstruktor von `CubeElementRepository` hat drei Parameter:

1. Verbindung zu GraphDB
2. Name des Ordners, in dem die Dateien mit den Abfragen für den jeweiligen Typ der implementierenden Klasse liegen. Die Abfrage-Dateien müssen dabei bestimmte Namen haben und die Abfrage festgelegte Parameter zurückliefern.
3. Name des Typs, welcher nur für Log-Ausgaben benötigt wird.

Der generische Parameter `TReturn` dieser Klasse gibt den Rückgabetyp der Abfragemethoden, welche nur IRIs zurückliefern, an. `TLabel` gibt die Art des Labels (siehe Abbildung 15), das von den Abfragemethoden zurückgeliefert wird, an. Die Methode `getAllByCube` mit zwei Parametern liefert beispielsweise alle Elemente des Typs für den angegeben Cube zurück, außer die Elemente, deren IRI in der Collection angegeben wurde. `DimensionCubeElementRepository` hat für den `TReturn`-Parameter ein String-Paar definiert. Ein Paar, enthält im linken Teil die IRI der Dimension und im rechten Teil die IRI des Elements. Durch die Erstellung dieser Klassenhierarchie konnte Code-Duplikation vermieden werden.

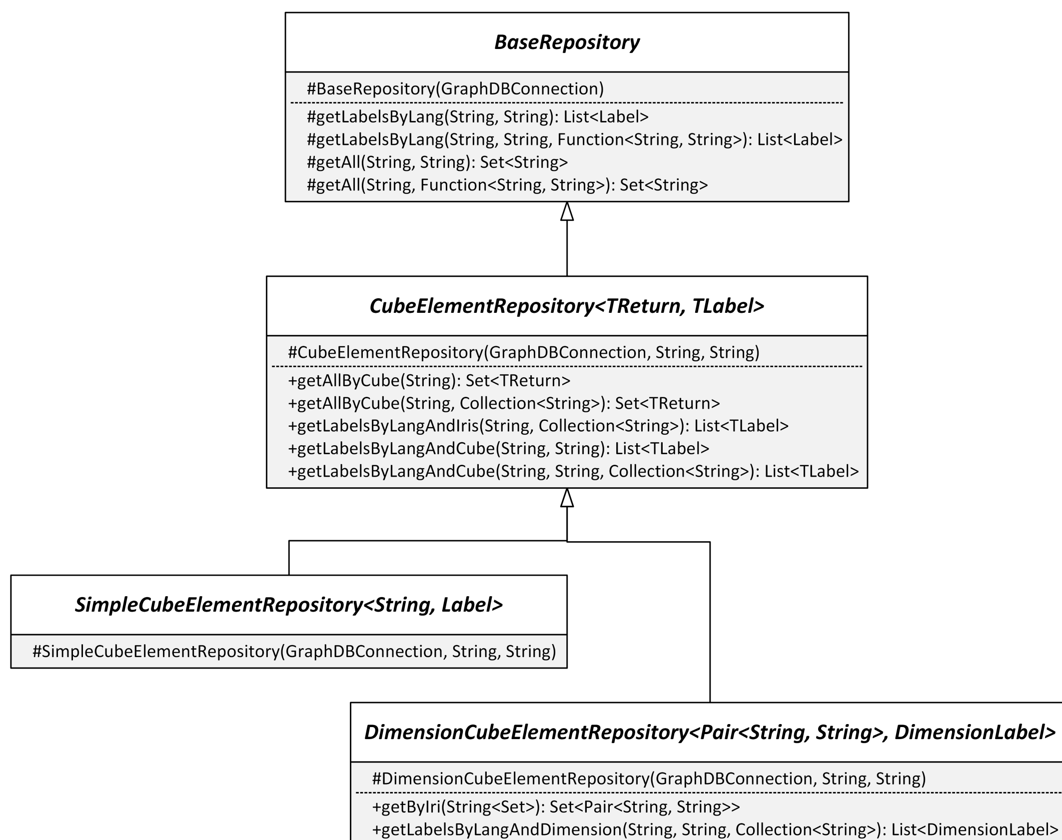


Abbildung 14: Klassendiagramm: Repository

Es gibt unterschiedliche Arten von Labels, wie in Abbildung 15 zu sehen ist. Ein Label beinhaltet grundsätzlich die URI des Elements, die Bezeichnung, welche im RDF mittels `rdfs:label` angegeben wurde, die Sprache des Labels und die optionale Beschreibung (`rdfs:note`). `Label` wird für Elemente verwendet, die nur einem Cube, aber keiner Dimension oder Level zuzuordnen sind. Dazu gehören unter anderem *Cube* und *AggregateMeasure*. `DimensionLabel` wird für Elemente verwendet, die einer Dimension zuzuordnen sind, wie zum Beispiel *Levels*. Es enthält zusätzlich noch die URI und die Bezeichnung der Dimension. `DimensionLevelLabel` wird für *LevelMember* verwendet und enthält zusätzlich die URI und die Bezeichnung des Levels.

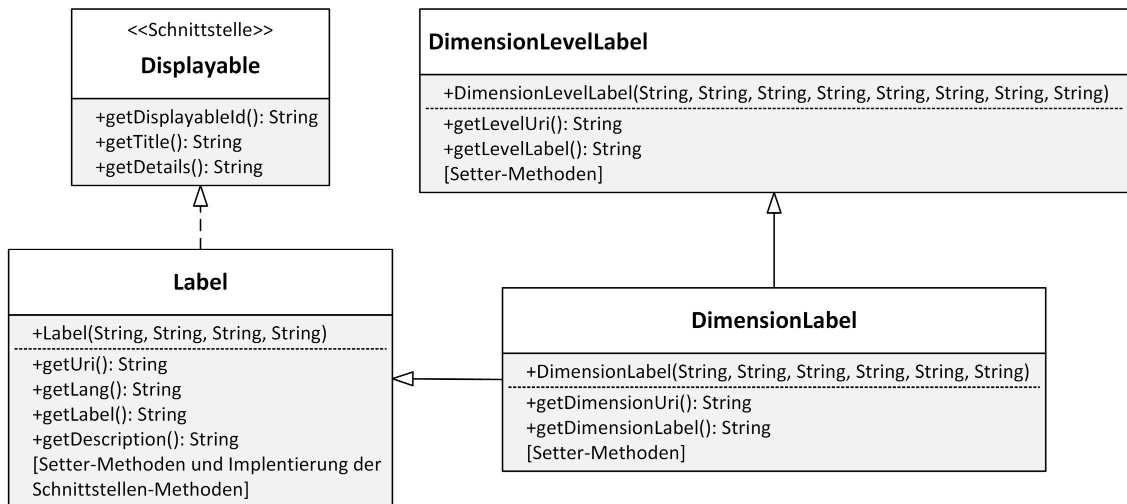


Abbildung 15: Klassendiagramm: Label

Listing 6 zeigt einen Ausschnitt der Methode, um Abfragen an GraphDB zu senden und diese auszuführen. Nachdem in Zeile 2 zuerst eine neue Verbindung hergestellt wurde, wird die Datei mit der SPARQL-Abfrage in Zeile 3 eingelesen. Mit Hilfe des `queryStringManipulators`, kann die Abfrage noch modifiziert werden, bevor diese ausgeführt wird. Dies wird benötigt, um Platzhalter in der Abfrage durch reale Werte zu ersetzen. Zeile 15 zeigt ein Beispiel für `queryStringManipulator`, bei der alle Vorkommen von "###LANG###" in der Abfrage durch "en" ersetzt werden. Danach wird die Abfrage vorbereitet und in Zeile 6 ausgeführt. In einer Schleife werden alle zurückgelieferten Tupel in einer Liste gespeichert, sodass diese auch nach schließen der Verbindung zur Datenbank noch zur weiteren Verwendung zur Verfügung stehen. Ein Tupel enthält alle Parameter, die durch die Abfrage zurückgeliefert wurden.

Listing 6: Java-Methode GraphDB-Abfrage

```
public List<BindingSet> getQueryResult(String queryFile, Function<String,
    String> queryStringManipulator) throws QueryException {
    try (var conn = createConnection()) {
        String queryString = IOUtils.toString(GraphDbConnection.class.
            getResourceAsStream(queryFile), StandardCharsets.UTF_8);
        queryString = queryStringManipulator.apply(queryString);
        TupleQuery query = conn.prepareTupleQuery(queryString);
        try (var result = query.evaluate()) {
            List<BindingSet> list = new ArrayList<>();
            while (result.hasNext()) {
                list.add(result.next());
            }
            return list;
        }
    }
    ...
    // Example for queryStringManipulator: query ->
        query.replace("###LANG###", "en")
}
```

6.3. State Chart XML

Der gesamte Ablauf einer Benutzerinteraktion wurde mittels State Chart XML abgebildet. Als SCXML Engine wird in dieser Arbeit Apache Commons SCXML (The Apache Software Foundation, 2015) verwendet. Eine State Chart XML-Datei besteht im Grundlegenden aus Zuständen, die mittels Transitionen miteinander verbunden sind und modelliert damit einen Zustandsautomaten. Jedes Mal, wenn ein Zustand betreten wird, wird eine eigens entwickelte Java-Klasse, die sich aus der `Action`-Klasse aus der Apache Commons SCXML Bibliothek ableitet, aufgerufen. Abbildung 16 zeigt die grafische Repräsentation des entwickelten Zustandsdiagramms. Die aufgerufenen Klassen sind in der Abbildung durch das Präfix `ida` gekennzeichnet. Jede Klasse außer `DisplayValues`, `DisplayFreeText`, `ExecuteQuery` und `SwitchAnalysisSituation` ruft dabei Drools-Regeln auf (siehe Kapitel 6.4) um die jeweiligen Werte bzw. Ereignisse zu ermitteln. Das Datenmodell des entwickelten Zustandsdiagramms enthält zudem zwei Variablen: *sessionId* enthält die Kennung der Sitzung, der die Instanz des Zustandsdiagramms zugeordnet ist; sowie *cubeSelected*, das angibt, ob im Zustand ParsingFreeText eine Analysesituation erstellt werden konnte und somit ein Cube ausgewählt wurde.

Der erste Zustand **PatternSelection** besteht aus zwei Unterzuständen. Dieser Zustand entspricht hierbei der Aktion "Auswahl der Art der Abfrage" aus Abbildung 8. Die Aufgabe von **DisplayingPatterns** ist es, dem Benutzer eine Auswahl von möglichen Abfragearten anzuzeigen. Durch eine Eingabe löst der Benutzer das *userInput*-Ereignis aus. In **DeterminingPatternSelection** wird überprüft, ob der Benutzer eine gültige Abfrage-Art ausgewählt hat. Bei einer ungültigen Eingabe wird das *invalidInput*-Ereignis ausgelöst und wieder zum Zustand **DisplayingPatterns** gewechselt. Bei einer gültigen Eingabe wird *determined* ausgelöst und zum Zustand **UserQuery** gewechselt.

Der Zustand **UserQuery** besteht wiederum aus zwei Unterzuständen. Dieser Zustand entspricht hierbei der Aktion "Freitext-Eingabe" aus Abbildung 8. **WaitingForFreeText** zeigt dem Benutzer eine Aufforderung anzugeben, wie die Analysesituation aussehen soll. Durch eine Eingabe des Benutzers wird *userInput* ausgelöst und in den Zustand **ParsingFreeText** gewechselt. In ParsingFreeText wird versucht anhand der Benutzereingabe mittels Constraint Satisfaction (siehe Kapitel 6.7) eine Analysesituation zu erstellen. Wenn eine Analysesituation erstellt werden konnte und somit ein Cube ausgewählt wurde, wird *cubeSelected* im Datenmodell des Zustandsdiagramms auf `true` gesetzt; andernfalls auf `false`. In beiden Fällen wird das *determined*-Ereignis ausgelöst.

Abhängig vom Wert, der in *cubeSelected* gespeichert ist, wird im Zustand **GuidedInstantiation** mit dem Zustand **Changing** (wenn *cubeSelected* `false` ist) oder **Executable** (wenn *cubeSelected* `true` ist) gestartet. Der **GuidedInstantiation**-Zustand entspricht hierbei den Aktionen "Dialogbasierte Verfeinerung", "Dialogbasierte Verfeinerung CoI", "Dialogbasierte Verfeinerung CoC" und "Dialogbasierte Verfeinerung Comp." aus Abbildung 8.

Zu Beginn der Ausführung eines Zustandsautomaten wird die erste auszuführende Operation auf *navigate.selectCube* festgelegt. Somit werden, falls zuerst in den Zustand **Changing** gewechselt wird, die Aktionen in diesem Zustand für diese Operation ausgeführt. Der Zustand **Changing** besteht aus zwei Unterzuständen. **DisplayingValues** ermittelt, basierend auf der ausgewählten Operation, die möglichen Werte, aus denen der Benutzer auswählen kann und präsentiert diese dem Benutzer. Wenn der Benutzer daraufhin eine Eingabe tätigt, wird das *userInput*-Ereignis ausgelöst und damit in den Zustand **DeterminingValueInputIntent** gewechselt.

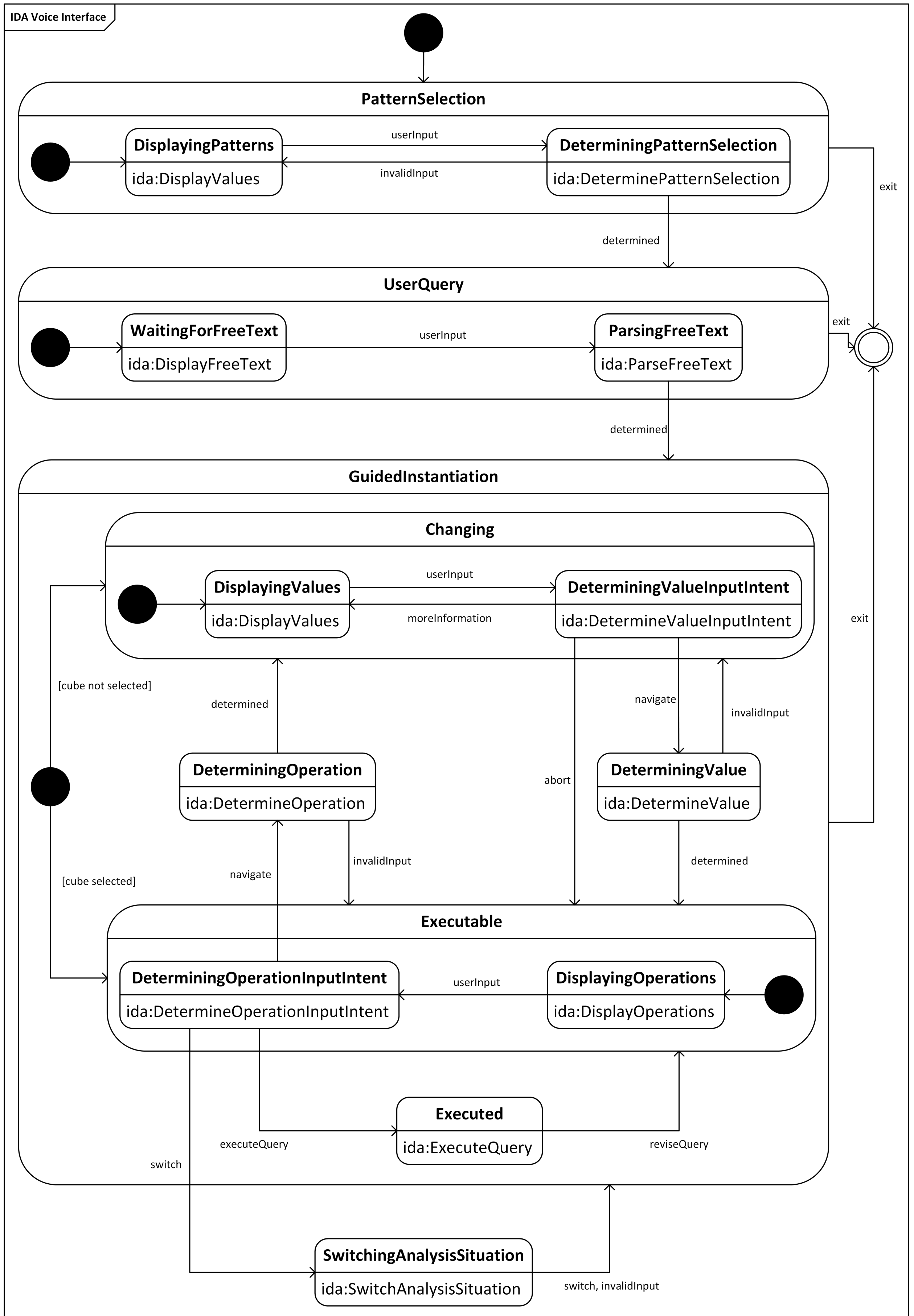


Abbildung 16: Zustandsautomat

DeterminingValueInputIntent hat mehrere Aufgaben. Eine Aufgabe ist es zu überprüfen, ob der Benutzer die aktuelle Operation abbrechen möchte. In diesem Fall wird das *abort*-Ereignis aufgerufen, und in den Zustand **Executable** gewechselt. Falls der Benutzer den gesamten Zustandsautomat beenden möchte, wird das *exit*-Ereignis ausgelöst. Bei manchen Operationen ist eine mehrstufige Auswahl notwendig. Ein Beispiel hierfür ist die Roll-Up-Operation. Hierbei muss zuerst eine Dimension und dann basierend darauf ein Granulariätslevel ausgewählt werden. In diesem Fall wird in **DisplayingValues** zuerst eine Auswahl an möglichen Dimensionen dargestellt. Wenn der Benutzer eine gültige Dimension ausgewählt hat, wird die ausgewählte Dimension im Zustand **DeterminingValueInputIntent** ermittelt ,in der Sitzung zwischengespeichert und das *moreInformation*-Ereignis ausgelöst. Somit wird wieder zu **DisplayingValues** gewechselt und basierend auf der ausgewählten Dimension mögliche Level dargestellt. Falls keine der vorherigen Ereignisse (abort, exit, moreInformation) zutreffen, wird das *navigate*-Ereignis ausgelöst und in den **DeterminingValue**-Zustand gewechselt.

In **DeterminingValue** wird zuerst überprüft, ob der Benutzer einen gültigen Wert aus der ihm präsentierten Auswahl an möglichen Werten ausgewählt hat. Falls dies nicht der Fall ist, wird das *invalidInput*-Ereignis ausgelöst. Bei einer gültigen Eingabe wird die entsprechende Operation mit dem/den ausgewählten Wert(en) ausgeführt und die Analysesituation somit verändert. Daraufhin wird *determined* ausgelöst und in den Zustand **Executable** gewechselt.

Executable enthält die Zustände **DisplayingOperations** und **DeterminingOperationInputIntent**. In **DisplayingOperations** wird, basierend auf der aktuellen Analysesituation, eine Liste an möglichen Operationen ermittelt und dem Benutzer präsentiert. Durch eine Benutzereingabe wird das *userInput*-Ereignis ausgelöst und in den Zustand **DeterminingOperationInputIntent** gewechselt. Hier wird wiederum, ähnlich wie bei **DeterminingValueInputIntent** ermittelt, ob der Benutzer den Zustandsautomaten beenden möchte (*exit*-Ereignis), eine Abfrage ausführen möchte (*executeQuery*) oder im Falle einer Comparative-Analysesituation in eine (Teil)-Analysesituation wechseln (*switch*) möchte. Falls keines der Ereignisse zutrifft wird *navigate* ausgelöst.

DeterminingOperation ermittelt die ausgewählte Operation. Im Falle einer ungültigen Eingabe wird *invalidInput* ausgelöst. Bei einer gültigen Eingabe wird die ausgewählte Operation in der Sitzung zwischengespeichert und *determined* ausgelöst.

Jeder Zustand kann durch das Auslösen des *exit*-Ereignisses den Zustandsautomaten beenden.

6.4. Drools

Drools-Regeln werden von der SCXML-Komponente aus aufgerufen und haben die Aufgabe mögliche Werte und ausgewählte Werte zu ermitteln, sowie Werte zu setzen. Jeder SCXML-Zustand hat eine andere Aufgabe und benötigt somit auch unterschiedliche Regeln. Hierfür wurden alle Regeln sogenannten Agendagruppen zugewiesen. Es werden immer nur diejenigen Regeln ausgeführt, die zur aktuell ausgewählten Agendagruppe gehören.

Für jede Agendagruppe wurde eine Klasse erstellt, welche die Drools-Engine für die Agendagruppe startet und die Regeln ausführt. Die SCXML-Komponente verwendet diese *DroolsService*-Klassen, um die entsprechenden Werte zu ermitteln, wobei eine Instanz einer *ServiceModel*-Klasse übergeben wird. Für jede Agendagruppe gibt es eine eigene Model-Klasse. Diese Model-Klassen enthalten Daten, die in den Regeln

benötigt werden. So wird zum Beispiel die DroolsService-Klasse *OperationDisplayService* aus dem SCXML-Zustand "DisplayingOperations" mit einer Instanz der Klasse *OperationDisplayServiceModel* aufgerufen, welche unter anderem Methoden enthält, um zu überprüfen, ob noch nicht ausgewählte Kennzahlen zur Verfügung stehen. Zurückgeliefert wird eine Liste mit möglichen Operationen.

Listing 7 zeigt ein Beispiel für eine Drools-Regel. Diese Regel überprüft, ob eine "Add Measure"-Operation ausgeführt werden kann, indem unter Zuhilfenahme der Model-Klasse überprüft wird, ob noch nicht ausgewählte Kennzahlen existieren, oder ob bereits alle möglichen Kennzahlen ausgewählt wurden. Wenn noch Kennzahlen verfügbar sind, wird eine neue `Operation`-Instanz für "Add Measure" in den Drools-Kontext eingefügt. Nachdem alle Regeln ausgeführt wurden, ruft die `DroolsService`-Klasse alle Operation-Instanzen aus dem Drools-Kontext ab und liefert sie an die SCXML-Komponente zurück. Eine `Operation`-Klasse enthält den Namen der Operation, die Sprache der Benutzersitzung und die Anzeigeposition in der Liste von Operationen.

Listing 7: Beispiel für eine Drools-Regel

```
1 rule "'Add Measure' - if at least one measure is available"
2     agenda-group "operation-display-determination"
3 when
4     $model : OperationDisplayServiceModel(isNotSelectedMeasureAvailable())
5 then
6     insert(new Operation(Event.NAVIGATE_MEASURE_ADD, $model.getLocale(), 10));
7 end
```

Folgend wird jede Agendagruppe und ihre zugehörigen Regeln beschrieben.

operation-display-determination Die Regeln dieser Agendagruppe werden ausgeführt, wenn der Zustand "DisplayingOperations" betreten wird, um die aktuell möglichen Operationen zu ermitteln. Tabelle 9 zeigt die Regeln dieser Agendagruppe, wobei jede Zeile einer Regel entspricht.

Tabelle 9: Drools-Regeln "Operation Display Determination"

Operation	**Bedingung**
Exit	Ist immer möglich.
Select Cube	Es wurde noch kein Cube definiert.
Execute Query	Es wurde mindestens der Cube und eine Kennzahl festgelegt.
Add Measure	Es ist mindestens eine noch nicht ausgewählte Kennzahl für den aktuellen Cube vorhanden.
Refocus Measure	Es ist bereits mindestens eine Kennzahl ausgewählt und es existiert mindestens eine noch nicht ausgewählte Kennzahl für den aktuellen Cube.
Drop Measure	Es ist mindestens eine Kennzahl ausgewählt.

Weiter auf der nächsten Seite

Drools-Regeln “Operation Display Determination” - Fortsetzung

Operation	Bedingung
Add Filter	Es ist mindestens ein noch nicht ausgewähltes Aggregate Measure Predicate für den aktuellen Cube vorhanden.
Refocus Filter	Es ist bereits mindestens ein Aggregate Measure Predicate ausgewählt und es existiert mindestens ein noch nicht ausgewähltes Aggregate Measure Predicate für den aktuellen Cube.
Drop Filter	Es ist mindestens ein Aggregate Measure Predicate ausgewählt.
Add Base Measure Condition	Es ist mindestens ein noch nicht ausgewähltes Base Measure Predicate für den aktuellen Cube vorhanden.
Refocus Base Measure Condition	Es ist bereits mindestens ein Base Measure Predicate ausgewählt und es existiert mindestens ein noch nicht ausgewähltes Base Measure Predicate für den aktuellen Cube.
Drop Base Measure Condition	Es ist mindestens ein Base Measure Predicate ausgewählt.
Drill Down	Es existiert mindestens eine Dimension im aktuellen Cube, in der eine Drill-Down-Operation möglich ist. Eine Drill-Down-Operation ist möglich, wenn das aktuell ausgewählte Granularitätslevel einer Dimension nicht das Base-Level der Dimension ist.
Roll Up	Es existiert mindestens eine Dimension im aktuellen Cube, in der eine Roll-Up-Operation möglich ist. Eine Roll-Up-Operation ist möglich, wenn das das aktuell ausgewählte Granularitätslevel einer Dimension nicht dem höchsten Granularitätslevel entspricht.
Add Slice Condition	Es ist mindestens ein noch nicht ausgewähltes Level Predicate für den aktuellen Cube vorhanden.
Refocus Slice Condition	Es ist bereits mindestens ein Level Predicate ausgewählt und es existiert mindestens ein noch nicht ausgewähltes Level Predicate für den aktuellen Cube.
Drop Slice Condition	Es ist mindestens ein Level Predicate ausgewählt.
Add Dice Node	Es ist für mindestens eine Dimension noch kein Dice-Node ausgewählt.
Drop Dice Node	Es ist für mindestens eine Dimension ein Dice-Node ausgewählt.
Switch to CoI	Der Benutzer hat zu Beginn die Art “comparative” ausgewählt und aktuell wird nicht der CoI angezeigt.

Weiter auf der nächsten Seite

Drools-Regeln “Operation Display Determination” - Fortsetzung

Operation	**Bedingung**
Switch to CoC	Der Benutzer hat zu Beginn die Art “comparative” ausgewählt und aktuell wird nicht der CoC angezeigt.
Switch to Comp	Der Benutzer hat zu Beginn die Art “comparative” ausgewählt und aktuell wird nicht die vergleichende Analysesituation angezeigt.
Add Join Condition	Der Benutzer hat zu Beginn die Art “comparative” ausgewählt und es ist mindestens ein noch nicht ausgewähltes Join Condition Predicate für den aktuellen Cube vorhanden.
Drop Join Condition	Der Benutzer hat zu Beginn die Art “comparative” ausgewählt und es ist mindestens ein Join Condition Predicate ausgewählt.
Add Score	Der Benutzer hat zu Beginn die Art “comparative” ausgewählt und es ist mindestens eine noch nicht ausgewählte Comparative Measure für den aktuellen Cube vorhanden.
Drop Score	Der Benutzer hat zu Beginn die Art “comparative” ausgewählt und es ist mindestens eine Comparative Measure ausgewählt.
Add Score Filter	Der Benutzer hat zu Beginn die Art “comparative” ausgewählt und es ist mindestens ein noch nicht ausgewähltes Comparative Measure Predicate für den aktuellen Cube vorhanden.
Drop Score Filter	Der Benutzer hat zu Beginn die Art “comparative” ausgewählt und es ist mindestens ein Comparative Measure Predicate ausgewählt.

operation-intent-determination Die Regeln dieser Agendagruppe werden ausgeführt, wenn der Zustand “DeterminingOperationInputIntent” betreten wird. Hier wird zunächst überprüft, ob der Benutzer eine *Exit*-, *Execute Query*- oder *Switch*-Operation ausführen möchte. Falls dies der Fall ist, wird ein *EventConfidenceResult* mit der entsprechenden Operation in den Drools-Kontext eingefügt, andernfalls die *Navigate*-Operation. Hierfür enthält die Model-Klasse für diese Agendagruppe alle String-Ähnlichkeiten zwischen der Benutzereingabe und den möglichen Operationen. Wenn eine Operation anhand einer Zahleneingabe erkannt wurde, wird als Confidence 0.5 vergeben, ansonsten der Score der String-Ähnlichkeit.

In vielen Agendagruppen, wie auch in dieser, fügen die Regeln Instanzen von `ConfidenceResult`-Klassen (siehe Abbildung 17) in den Drools-Kontext ein. Ein Confidence Result enthält das ausgewählte Element (`getValue()`) und die Confidence (`getConfidence()`), die angibt, wie wahrscheinlich es ist, dass dieses Element ausgewählt wurde. Wenn mehrere Confidence Results durch die Regeln in einen Drools-Kontext eingefügt wurden, wird angenommen, dass der Benutzer jenes Element mit der höchsten Confidence ausgewählt hat. Für die unterschiedlichen Arten von möglichen Werten existiert jeweils eine Unterklasse von `GenericConfidenceResult`.

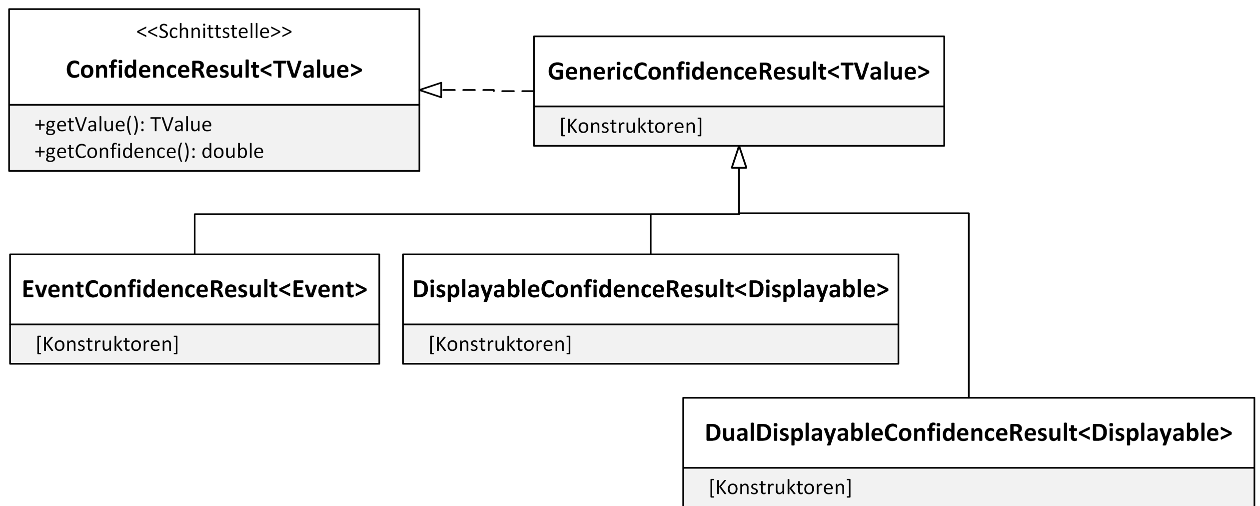


Abbildung 17: Klassendiagramm: ConfidenceResult

operation-determination Die Regeln dieser Agendagruppe werden ausgeführt, wenn der Zustand "DeterminingOperation" betreten wird, um den ausgewählten Navigationsoperator zu ermitteln. Hierfür wird versucht mittels Zahlen bzw. String-Ähnlichkeiten den Operator zu ermitteln. Auch die Regeln dieser Agenda-Gruppe liefern hierfür EventConfidenceResult-Instanzen. Wenn ein Operator anhand einer Zahleneingabe erkannt wurde, wird als Confidence 1.0 vergeben, ansonsten der Score der String-Ähnlichkeit verwendet, wobei nur String-Ähnlichkeiten mit einem Score größer oder gleich 0.4 berücksichtigt werden.

value-display-determination Die Regeln dieser Agendagruppe werden ausgeführt, wenn der Zustand "DisplayingValues" betreten wird, um die möglichen Werte zu ermitteln. Die Ermittlung der Werte erfolgt hierbei anhand des zuvor ausgewählten Navigationsoperators. Das Model dieser Agendagruppe enhält Referenzen zu allen Repositories, um die benötigten Werte bei Bedarf abfragen zu können. Tabelle 10 listet zu jeder Operation auf, welche Werte zurückgeliefert werden und in weiterer Folge dem Benutzer zur Auswahl präsentiert werden.

Tabelle 10: Drools-Regeln "Value Display Determination"

Operation	Auswahl der Werte zur Anzeige
Select Cube	verfügbare Cubes
Add Measure	alle noch nicht ausgewählte Kennzahlen (Aggregate Measure) des aktuellen Cubes
Refocus Measure	alle noch nicht ausgewählte Kennzahlen (Aggregate Measure) des aktuellen Cubes und alle ausgewählte Kennzahlen
Drop Measure	alle ausgewählte Kennzahlen (Aggregate Measure)
Add Filter	alle noch nicht ausgewählte Aggregate Measure Predicates des aktuellen Cubes

Weiter auf der nächsten Seite

Drools-Regeln “Value Display Determination” - Fortsetzung

Operation	Auswahl der Werte zur Anzeige
Refocus Filter	alle noch nicht ausgewählte Aggregate Measure Predicates des aktuellen Cubes und alle ausgewählte Aggregate Measure Predicates
Drop Filter	alle ausgewählte Aggregate Measure Predicates
Add Base Measure Condition	alle noch nicht ausgewählte Base Measure Predicates des aktuellen Cubes
Refocus Base Measure Condition	alle noch nicht ausgewählte Base Measure Predicates des aktuellen Cubes und alle ausgewählte Base Measure Predicates
Drop Base Measure Condition	alle ausgewählte Base Measure Predicates
Drill Down	zuerst alle Dimensionen, bei denen eine Drill-Down-Operation möglich ist; danach alle “detailliertere” Granularitätslevel der ausgewählten Dimension
Roll Up	zuerst alle Dimensionen, bei denen eine Roll-Up-Operation möglich ist; danach alle “allgemeinere” Granularitätslevel der ausgewählten Dimension
Add Slice Condition	zuerst alle Dimensionen, bei denen eine Slice Condition hinzugefügt werden kann; danach alle noch nicht ausgewählte Level Predicates der ausgewählten Dimension
Refocus Slice Condition	zuerst alle Dimensionen, bei denen eine Slice Condition ausgetauscht werden kann; danach alle noch nicht ausgewählte Level Predicates des aktuellen Cubes und alle ausgewählte Level Predicates
Drop Slice Condition	zuerst alle Dimensionen mit mindestens einer ausgewählten Slice Condition; danach alle ausgewählte Level Predicates der ausgewählten Dimension
Add Dice Node	zuerst alle Dimensionen, bei denen noch kein Dice Node ausgewählt wurde; danach alle möglichen Dice Level der ausgewählten Dimension und basierend darauf alle möglichen Dice Nodes des Levels
Drop Dice Node	alle Dimensionen, bei denen ein Dice Node ausgewählt wurde
Add Join Condition	alle noch nicht ausgewählte Join Condition Predicates des aktuellen Cubes
Drop Join Condition	alle ausgewählte Join Condition Predicates
Add Score	alle noch nicht ausgewählte Comparative Measures des aktuellen Cubes
Drop Score	alle ausgewählte Comparative Measures

Weiter auf der nächsten Seite

Drools-Regeln “Value Display Determination” - Fortsetzung

Operation	Auswahl der Werte zur Anzeige
Add Score Filter	alle noch nicht ausgewähltes Comparative Measure Predicates des aktuellen Cubes
Drop Score Filter	alle ausgweählte Comparative Measure Predicates

Bei den Operationen für vergleichende Analysesituationen wird nicht überprüft, ob in CoI und in CoC die entsprechenden Kennzahlen und Granularitätslevel entsprechend den ausgewählten Join-Bedingungen und Scores ausgewählt wurden.

value-intent-determination Die Regeln dieser Agendagruppe werden ausgeführt, wenn der Zustand “DeterminingValueInputIntent” betreten wird. Hier wird zunächst überprüft, ob der Benutzer eine *Exit*- oder *Abort*-Operation ausführen möchte. Falls dies der Fall ist, wird ein *EventConfidenceResult* mit der entsprechenden Operation in den Drools-Kontext eingefügt. Falls eine Operation ausgewählt wurde, die zuerst die Auswahl einer Dimension bzw. eines Levels erfordert, wird, wie bereits zuvor beschrieben, in “DisplayingValues” zuerst eine Liste an möglichen Dimensionen bzw. Levels angezeigt. Wenn der Benutzer eine Dimension bzw. Level ausgewählt hat, wird dies hier ermittelt, in der Sitzung zwischengespeichert und das Ereignis *moreInformation* in den Drools-Kontext eingefügt, um der SCXML-Engine mitzuteilen, wieder zu “DisplayingValues” zurückzugehen, um die weiteren Werte (z. B. Granularity Level) auszuwählen. In allen anderen Fällen wird ein *EventConfidenceResult* mit der *Navigate*-Operation in den Drools-Kontext eingefügt, um zum nächsten Zustand zu gelangen.

value-determination Die Regeln dieser Agendagruppe werden ausgeführt, wenn der Zustand “DeterminingValue” betreten wird, um den ausgewählten Wert bzw. die ausgewählten Werte zu ermitteln. Hierfür wird versucht mittels des Similarity-Index Ähnlichkeiten zwischen den Wortgruppen der Benutzereingabe und den Schema-Elementen zu finden. Wenn eine Ähnlichkeit gefunden wurde, wird diese als *DisplayableConfidenceResult* mit dem Score des Similarity-Index in den Drools-Kontext eingefügt. Wenn ein Wert anhand einer Zahleneingabe erkannt wurde, wird als Confidence 1.0 vergeben.

set-value Die Regeln dieser Agendagruppe werden auch im Zustand ‘DeterminingValueInputIntent‘ ausgeführt, allerdings erst nach den *value-determination*-Regeln, da das Ergebnis dieser Regeln für das Modell dieser Agendagruppe benötigt werden. Basierend auf der ausgewählten Operation, setzten die Regeln den Wert, der in den *value-determination*-Regeln ermittelt wurde.

pattern-determination Die Regeln dieser Agendagruppe werden ausgeführt, wenn der Zustand “DeterminingPatternSelection” betreten wird, um die ausgewählte Abfrageart zu ermitteln. Der Ablauf funktioniert gleich wie bei den Regeln der *operation-determination*-Agendagruppe.

6.5. Similarity Index

GraphDB bietet die Möglichkeit semantische Ähnlichkeitssuchen in RDF-Ressourcen durchzuführen (Ontotext, 2019b). Hierfür muss ein Text-Similarity-Index erstellt werden. Der Index verwendet die Semantic Vectors-Bibliothek (Widdows & Ferraro, 2008) und den zugrundeliegenden Random-Indexing-Algorithmus, welcher einen Tokenizer verwendet, um Dokumente in eine Folge von Wörtern zu übersetzen und diese in einem Vector Space Modell darzustellen.

In dieser Arbeit wurde ein Similarity-Index über Nomen aus WordNet erstellt. Mit Hilfe des Index ist es möglich, dass Elemente der multidimensionalen Schemata auch gefunden werden, wenn der Benutzer nicht den genauen Begriff des Elements angibt, sondern einen semantisch ähnlichen (z. B. statt "quantity" gibt der Benutzer "amount" an). Hierzu muss zuerst, wie in Kapitel 4.1.1 beschrieben, ein Similarity Index erstellt werden. Listing 8 zeigt die Abfrage für den Index.

Listing 8: Similarity-Index

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX ontolex: <http://www.w3.org/ns/lemon/ontolex#>
3  PREFIX wordnet: <http://wordnet-rdf.princeton.edu/ontology#>
4  PREFIX sim: <http://dke.jku.at/ida/similarity#>
5  
6  SELECT DISTINCT ?documentID ?documentText WHERE {
7      {
8          SELECT DISTINCT ?term ?text WHERE {
9              GRAPH sim:wordnet {
10                 ?term wordnet:partOfSpeech wordnet:noun .
11                 ?term ontolex:canonicalForm ?canonicalForm .
12                 {
13                     ?canonicalForm ontolex:writtenRep ?text .
14                 } UNION {
15                     ?term ontolex:sense ?lexicalizedSense .
16                     ?lexicalizedSense ontolex:isLexicalizedSenseOf ?sense .
17                     ?sense wordnet:definition ?definition .
18                     ?definition rdf:value ?text .
19                 }
20             }
21         }
22     } UNION {
23         SELECT DISTINCT ?term ?text WHERE {
24             GRAPH sim:wordnet {
25                 ?term wordnet:partOfSpeech wordnet:noun .
26                 ?term ontolex:canonicalForm?canonicalForm .
27                 ?term ontolex:sense ?lexicalizedSense .
28                 ?lexicalizedSense ontolex:isLexicalizedSenseOf ?sense .
29                 ?sense wordnet:hyponym ?hyponym .
30                 {
31                     ?hyponymLexicalizedSense ontolex:isLexicalizedSenseOf
                           ?hyponym .
32                     ?hyponymTerm ontolex:sense ?hyponymLexicalizedSense .
33                     ?hyponymTerm ontolex:canonicalForm ?hyponymCanonicalForm .
```

```
34                          ?hyponymCanonicalForm  ontolex:writtenRep ?text .
35                      } UNION {
36                          ?hyponym wordnet:definition ?hyponymDefinition .
37                          ?hyponymDefinition rdf:value ?text .
38                      }
39                  }
40              }
41      }
42      BIND(?term AS ?documentID)
43      BIND(?text AS ?documentText)
44  }
```

Die Abfrage des Index muss immer die Variablen ?documentID und ?documentText zurückliefern. ?documentID in Listing 8 liefert die IRI der WordNet-Ressource und ?documentText die schriftliche Repräsentation der Ressource sowie aller Unterbegriffe und deren Definitionen. Dazu werden in der ersten Unterabfrage in den Zeilen 8 bis 21 die schriftliche Repräsentation und die Definition aller WordNet-Nomen abgefragt. In der zweiten Unterabfrage in den Zeilen 23 bis 40 werden alle Unterbegriffe (Hyponym) aller WordNet-Nomen abgefragt und deren schriftliche Repräsentation und Definition zurückgeliefert. Es wäre auch noch möglich zum Beispiel alle Überbegriffe miteinzubeziehen. Allerdings würde dies die Abfragezeit erhöhen. Auch nur mit der Verwendung von Unterbegriffen werden bereits gute Resultate erzielt.

Nachdem der Index erstellt wurde, kann man diesen verwenden, um nach Schema-Elementen zu suchen, die ähnlich zu einem bestimmten Begriff sind. Listing 9 zeigt eine Abfrage, in der Base Measures, deren Bezeichnung ähnlich zum Begriff “amount” ist, gesucht werden. Die Abfrage liefert alle Kennzahlen zum gesuchten Begriff mit einem Score zwischen 0 und 1, der angibt, wie ähnlich der Suchbegriff und die Bezeichnung der Kennzahl sind. Hierzu werden in den Zeilen 26 bis 33 alle WordNet-Ressourcen gesucht, die ähnlich zum Begriff “amount” sind. Als Nächstes werden in den Zeilen 17 bis 21 alle Kennzahlen und deren Bezeichnung abgefragt. Dieses Ergebnis wird verwendet um für alle Kennzahlen (Zeilen 9 bis 23) WordNet-Ressourcen zu finden, die ähnlich zu den Bezeichnungen der Kennzahlen sind. Dieses Ergebnis wird mit dem Ergebnis der Abfrage aus den Zeilen 26 bis 33 über die *documentID* zusammengefügt und die Scores der beiden Abfragen multipliziert (Score Ähnlichkeit WordNet zu “amount” * Score Ähnlichkeit WordNet zu Kennzahl-Bezeichnung). Das Ergebnis wird nach Kennzahl gruppiert und der höchste Score pro Kennzahl ausgewählt. Zum Schluss wird das Ergebnis absteigend nach Score sortiert. Somit erhält man die ähnlichste Kennzahl an erster Stelle des Abfrageergebnisses.

Listing 9: Similarity-Index Abfrage

```
1  PREFIX : <http://www.ontotext.com/graphdb/similarity/>
2  PREFIX inst: <http://www.ontotext.com/graphdb/similarity/instance/>
3  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
5  PREFIX qbx: <http://dke.jku.at/inga/cubes#>
6
7  SELECT ?measure (MAX(?score) AS ?score) {
8      {
9          SELECT ?measure ?documentID ?score1 {
10             ?search a inst:wordnet ;
```

```
                :searchTerm ?filterText ;
                :searchParameters "" ;
                :documentResult ?result .
            ?result :value ?documentID ;
                    :score ?score1 .
            {
                SELECT ?measure ?filterText {
                    ?measure rdf:type qbx:BaseMeasure .
                    ?measure rdfs:label ?filterText .
                    FILTER(isLiteral(?filterText))
                }
            }
        }
    }
    {
        SELECT ?documentID ?score2 {
            ?search a inst:wordnet ;
                :searchTerm "amount" ;
                :searchParameters "" ;
                :documentResult ?result .
            ?result :value ?documentID ;
                    :score ?score2 .
        }
    }
    BIND((?score1 * ?score2) AS ?score)
} GROUP BY ?measure
ORDER BY DESC(?score)
```

Da insbesondere die Abfrage in den Zeilen 9 bis 23 sehr lange dauert wurde die Zuordnung von Schema-Elementen zu WordNet-Ressourcen materialisiert. Hierzu werden in einem Named Graph das Schema-Element, die WordNet-Ressource und der dazugehörige Score, der Cube zu dem das Schema-Element gehört, der Typ des Schema-Elements und, falls zutreffend, die Dimension zu der das Element gehört, gespeichert. Für jede Art von Schema-Element (AggregateMeasure, AggregateMeasurePredicate, BaseMeasurePredicate, GranularityLevel, LevelPredicate) existiert eine eigene SPARQL-Update Abfrage (siehe Anhang A). Dies hat auch den Vorteil, dass danach die Zuordnung einheitlich und unabhängig vom Typ des Elements abgefragt werden kann. Der Teil der Abfrage in den Zeilen 9 bis 23 wird durch die Abfrage aus Listing 10 ersetzt.

Listing 10: Similarity-Index Abfrage: Teil mit Materialization

```
SELECT ?cube ?dimension ?element ?type ?documentID ?score1
WHERE { GRAPH sim:wordnet-scores-en {
            ?mapping rdf:type     sim:Mapping ;
                     sim:from     ?element ;
                     sim:to       ?documentID ;
                     sim:hasScore ?score1 ;
                     sim:inCube   ?cube ;
                     sim:fromType ?type .
            OPTIONAL { ?mapping sim:inDim ?dimension } }
}
```

Für den Fall, dass der Suchbegriff aus mehreren Wörtern besteht, werden in einer Abfrage für jedes Wort des Suchbegriffs einzeln die ähnlichen Elemente gesucht und in weiterer Folge alle Scores multipliziert. Dies ist notwendig, da in WordNet meist nur Nomen mit einem Wort vorkommen und Suchbegriffe wie zum Beispiel "doctor district" nicht als WordNet-Ressource vorliegen.

6.6. Natural Language Processing

Nach jeder Benutzereingabe wird zunächst die NLP-Komponente aufgerufen, um Wortgruppen in der Benutzereingabe zu finden. Wie bereits in Kapitel 2.1 erwähnt, wird Stanford CoreNLP verwendet, um Part-Of-Speech-Tagging, Dependency Parsing und Constituency Parsing durchzuführen. Listing 11 zeigt, wie dieses Tagging und Parsing auf einen Text durchgeführt werden kann. Zuerst wird in Zeile 2 die NLP-Pipeline für die jeweilige Sprache abgerufen, wobei in dieser Arbeit nur die englische Sprache unterstützt wird. Wenn die Pipeline noch nicht existiert wird diese erstellt, ansonsten wird die bereits erstellte Pipeline zurückgeliefert. Die Pipelines werden zwischengespeichert, da das Laden der Modelle für CoreNLP einige Zeit in Anspruch nehmen kann. Die zurückgelieferte Pipeline führt zuerst eine Tokenisierung durch, danach wird die Eingabe nach Sätzen aufgeteilt. Im Anschluss daran werden Part-Of-Speech-Tagging, Dependency Parsing und Constituency Parsing durchgeführt.

`CoreDocument` repräsentiert ein Dokument, das durch Aufruf der Methode `pipeline.annotate(document)` annotiert werden kann. Diese Methode führt die Pipeline-Aufgaben auf das Dokument aus und danach können z. B. die Part-Of-Speech-Tags abgerufen werden.

Listing 11: Stanford CoreNLP Text Annotierung

```
1 // build pipeline
2 StanfordCoreNLP pipeline = getPipeline(language);
3 CoreDocument document = new CoreDocument(text);
4
5 // annotate
6 pipeline.annotate(document);
```

Um nun zusammengehörende Wörter (Wortgruppen) im Text zu finden, werden sogenannte Semgrex[5]- und Tregex[6]-Patterns verwendet. Um Wortgruppen mit diesen Patterns in einem Text zu finden, wurde hierfür auch eine DroolsService-Klasse erstellt. Dadurch, dass die Patterns in Drools-Regeln definiert werden, ist es sehr einfach neue Patterns hinzuzufügen; bzw. ist es auch möglich Wortgruppen gänzlich anders, ohne diese Patterns zu ermitteln.

Mit einem **Semgrex**-Pattern können in einem Dependency Graph (Beispiel in Abbildung 2) Muster von Knoten- und Kantenzuordnungen gefunden werden. Listing 12 zeigt ein Beispiel für eine Regel, mit der Semgrex-Patterns in der Benutzereingabe gefunden werden. Mit Hilfe des Patterns *{tag:/NN.*/}=B >amod {tag:/JJ/}=A* werden *Nomen* (NN) gesucht, die als ausgehende Kante einen *adjectival modifier* (amod) haben und auf ein Adjektiv (JJ) verweisen. Für den Beispielsatz in Abbildung 2 liefert diese Regel die Wortgruppe "total costs" zurück. Alle gefundenen Wortgruppen werden in den Drools-Kontext eingefügt und nach dem Ausführen aller Regeln vom Drools-Service an den Aufrufer zurückgeliefert.

[5] https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/semgraph/semgrex/SemgrexPattern.html
[6] https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/trees/tregex/TregexPattern.html

Listing 12: Stanford CoreNLP Semgrex-Pattern

```
rule "Word Groups - Find nouns with adjectival modifier"
    agenda-group "word-groups"
when
    $model : WordGroupsServiceModel(language == "en")
then
    for (WordGroup group :
        WordGroupsHelper.executeSemgrex($model.getAnnotatedText(),
        "{tag:/NN.*/}=B >amod {tag:/JJ/}=A")) {
         insert(group);
    }
end
```

Der Aufruf `$model.getAnnotatedText()` liefert das annotierte `CoreDocument` zurück. Die Hilfsmethode `executeSemgrex` führt das Pattern auf das Dokument aus und liefert alle Wortgruppen zurück. Die Angabe von =A, =B, usw. ist notwendig, damit die Knoten, die vom Pattern gefunden wurden in der richtigen Reihenfolge zu einer Wortgruppe zusammengefügt werden. Die Wörter werden dabei in alphabetischer Reihenfolge dieser Angaben in die Wortgruppe eingefügt. Die Klasse `WordGroup` ist eine einfach Klasse, mit nur einer Eigenschaft "text" mit dazugehöriger Getter-Methode.

Mit einem **Tregex**-Pattern können Knotenkonfigurationen in einem Constituency-Tree (Beispiel in Abbildung 3) gefunden werden, wobei die Knoten mit Part-Of-Speech-Tags versehen sind. Eine Regel für Tregex-Patterns sieht genauso aus, wie Regeln für Semgrex-Patterns, nur dass hier anstatt `executeSemgrex` `executeTregex` aufgerufen wird, um das Pattern auf den Text anzuwenden. Ein Beispiel für ein Tregex-Pattern ist */NN.?/=A $+ /NN.?/=B* werden Worgruppen gefunden, die aus benachbarten Nomen bestehen. Für das Beispiel in Abbildung 3 findet dieses Pattern die Wortgruppe "doctor district".

Nachdem die Wortgruppen ermittelt wurden, werden diese verwendet, um Ähnlichkeiten zu den Auswahlmöglichkeiten zu berechnen. Im Falle von Operationen wird die Ähnlichkeit unter Verwendung von Levensthein- und Jaro-Winkler-Metriken (siehe Kapitel 2.1.4) berechnet. Im Falle von Schema-Elementen wird zusätzlich noch der Similarity Index (siehe Kapitel 6.5) verwendet, um die semantische Ähnlichkeit zu ermitteln.

Die Ähnlichkeiten werden mittels Instanzen der Klasse `Similarity` (oder Unterklassen) repräsentiert. Abbildung 18 zeigt die Klassenhierarchie der Similarity-Klassen. Die generische Klasse `Similarity` enthält als *Term* die Wortgruppe, *Element* enthält die Operation bzw. das Schema-Element und *Score* die Ähnlichkeit zwischen der Wortgruppe und dem Element. Die Klasse `CubeSimilarity` gibt die Ähnlichkeit zwischen einer Wortgruppe und einem Schema-Element an. Die Klasse enthält zusätzlich den *Cube*, zu dem das Element gehört, sowie den *Typ* des Elements (z. B. AggregateMeasure). Die Klasse `DimensionSimilarity` wird für Schema-Elemente verwendet, die einer Dimension zugeordnet sind und enthält daher zusätzlich noch die *Dimension*. Somit kann im weiteren Verlauf, falls notwendig, unmittelbar der Cube, Typ und die Dimension eines Elements ermittelt werden, ohne hierfür zusätzliche Abfragen durchführen zu müssen.

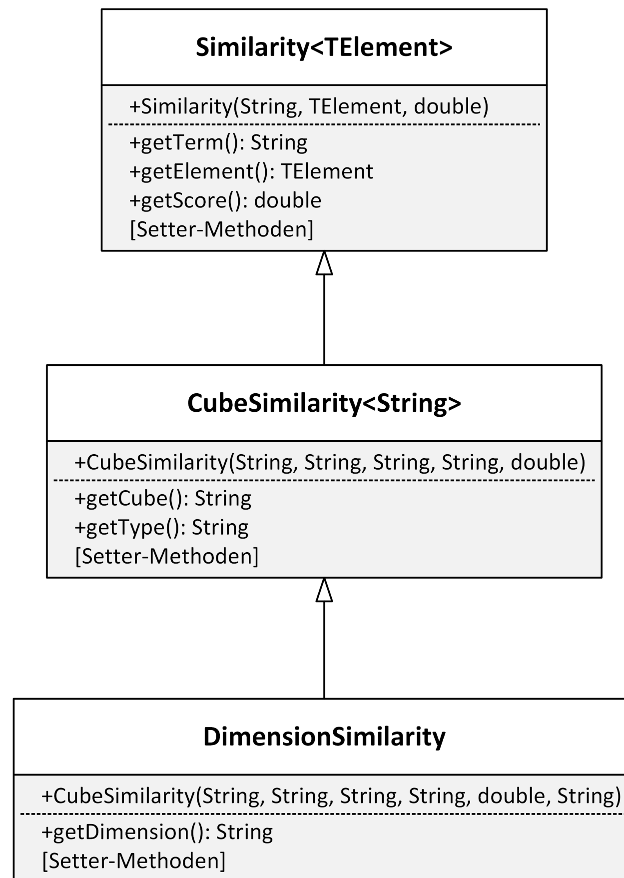


Abbildung 18: Klassendiagramm: Similarity

6.7. Constraint Satisfaction

Constraint Satisfaction wird verwendet, um aus einer Freitexteingabe des Benutzers eine gültige Analysesituation zu generieren. Die SCXML-Komponente ruft die Constraint-Satisfcation-Komponente auf und übergibt eine Liste von CubeSimilarities (Zuordnung von Schema-Elementen zu Wortgruppen im Eingabetext). Für Constraint Satisfaction wird in dieser Arbeit die Optaplanner-Bibliothek verwendet. Optaplanner erfordert die Erstellung einer PlanningSolution- und einer PlanningEntity-Klasse, um Constraint Satisfaction durchführen zu können.

Die erstellte Klasse `AnalysisSituationSolution` ist mit `@PlanningSolution` annotiert und enthält alle möglichen Werte (Domäne) für alle möglichen Variablen (z. B. Cubes, Kennzahlen, ...) sowie die generierte Analysesituation und den berechneten Score. Listing 13 zeigt einen Ausschnitt dieser Klasse. `@ValueRangeProvider` definiert, dass die Methode mögliche Werte für bestimmte Typen zurück liefert. `@PlanningEntityProperty` definiert die Planungs-Entität, die mit Werten aus den Value-Range-Providern befüllt werden soll. `@PlanningScore` definiert den verwendeten Score. In dieser Arbeit wird ein `HardSoftBigDecimalScore` verwendet, welcher den Hard- und den Soft-Score als Dezimalzahlen abspeichert. Es wurde dieser Score und kein `HardSoftScore`, welcher die Scores als Ganzzahlen abspeichert verwendet, weil die Ähnlichkeitsberechnungen Dezimalzahlen zurückliefern und keine ganzen Zahlen.

Listing 13: Constraint Satisfaction: Planning Solution

```
@PlanningSolution
public class AnalysisSituationSolution {

    @ValueRangeProvider(id = "cubes")
    @ProblemFactCollectionProperty
    public Set<String> getCubes() { ... }

    @ValueRangeProvider(id = "measures")
    @ProblemFactCollectionProperty
    public Set<AnalysisSituationElement> getMeasures() { ... }

    // Same for getGranularityLevels, getBaseMeasureConditions,
        getFilterConditions, getSliceConditions

    @PlanningEntityProperty
    public AnalysisSituation getAnalysisSituation() { ... }

    @PlanningScore
    public HardSoftBigDecimalScore getScore() { ... }

    ...
}
```

Die Klasse `AnalysisSituation` (Listing 14), welche nicht ident der Klasse `AnalysisSituation` aus Abbildung 13 ist, ist mit `@PlanningEntity` annotiert und repräsentiert eine Analysesituation. `@PlanningVariable` definiert, welche der zuvor in `AnalysisSituationSolution` erstellten ValueRangeProvider verwendet werden sollen, um die Felder zu befüllen. Die Methode `getScore` liefert den gesamten Score der Analysesituation basierend auf den Scores der einzelnen Werte. Die Klasse `AnalysisSituationElement` enthält eine Liste von `CubeSimilarity`-Instanzen und eine Methode, um den Score basierend auf dem Ähnlichkeits-Score aller Elemente in der Liste zu berechnen.

Listing 14: Constraint Satisfaction: Planning Entity

```
@PlanningEntity
public class AnalysisSituation {
    @PlanningVariable(valueRangeProviderRefs = {"cubes"})
    public String getCube() { ... }

    @PlanningVariable(valueRangeProviderRefs = {"measures"})
    public AnalysisSituationElement getMeasures() { ... }

    // Same for getGranularityLevels, getBaseMeasureConditions,
        getFilterConditions, getSliceConditions

    public BigDecimal getScore() { ... }

    ...
}
```

Die PlanningSolution wird wie folgt initialisiert: Der Score wird mit 0 initialisiert. Die Analysesituation ist zu Beginn leer. Cubes wird mit den verfügbaren Cubes aus allen übergebenen CubeSimilarity-Instanzen befüllt. Measures, Levels, Slice Conditions, Base Measure Conditions und Filter Conditions werden initialisiert, indem alle möglichen Kombinationen von CubeSimilarities (pro Typ) berechnet werden. Wenn zum Beispiel in der Liste CubeSimilarities Instanzen für die Kennzahlen A, B und C enthalten sind, werden folgende Kombinationen generiert und für jede eine `AnalysisSituationElement`-Instanz erstellt: [], [A], [B], [C], [A, B], [A, C], [A, B, C].

Nachdem die PlanningSolution erstellt wurde, kann der Optaplanner-Solver gestartet werden. Der Solver ruft Drools-Regeln auf, welche den Score modifizieren. Listing 15 zeigt zwei Beispiele für solche Regeln. Die erste Regel verringert den Hard-Score um 1, wenn eine Kombination von Kennzahlen ausgewählt wurde, bei der nicht alle Kennzahlen zum gleichen Cube gehören. Somit wird eine Lösung, die diese Kombination enthält, als ungültige Lösung angesehen. Die zweite Regel addiert den gesamten Score, basierend auf den Ähnlichkeitsscores aller Elemente der Analysesituation, zum Soft-Score. Dies bewirkt, dass am Ende jene Analysesituation zurückgeliefert wird, deren Score am höchsten ist. Wenn eine gültige Analysesituation generiert werden konnte, werden alle Elemente in die Analysesituation der Benutzersitzung überführt.

Listing 15: Constraint Satisfaction: Drools Regeln

```
rule "All measures in same cube"
when
    AnalysisSituation(!CSPRuleHelpers.allInCube(cube, measures))
then
    scoreHolder.addHardConstraintMatch(kcontext, CSPRuleHelpers.MINUS_ONE);
end
rule "Similarity Score"
when
    $as : AnalysisSituation(cube != null)
then
    scoreHolder.addSoftConstraintMatch(kcontext, $as.getScore());
end
```

Standardmäßig wird versucht nur die Kennzahlen, Granularitätslevel und Slice-Bedingungen zu befüllen, da dies die am Meisten verwendeten Elemente sind. Basis-Kennzahlen-Bedingungen und Filter werden standardmäßig nicht befüllt. Join-Bedingungen, Scores, Score-Filter, Dice-Nodes und Dice-Levels werden nie befüllt und werden, sofern sie in der Benutzereingabe vorkommen, ignoriert. Dieses Verhalten kann durch das Setzen der folgenden Einstellungen in die Datei "application.properties" verändert werden:

Tabelle 11: Einstellungen für CSP

Einstellung	Beschreibung
csp.use-levels	Granularitätslevel befüllen
csp.use-level-predicates	Slice-Bedingungen befüllen
csp.use-base-measure-predicates	Basis-Kennzahlen-Bedingungen befüllen
csp.use-aggregate-measure-predicates	Filter befüllen

6.8. Anpassbarkeit

Die Anwendung selbst wurde, wie bereits beschrieben, in mehrere Module aufgeteilt, wobei zwischen zwei Bereichen unterschieden wird: "app" und "engine". Der Bereich "engine" enthält die Kern-Funktionalitäten der Anwendung. Der Bereich "app" enthält unter anderem die Drools-Regeln und Anpassungen der Kern-Funktionalitäten. Um zum Beispiel die Logik zur Ermittlung des ausgewählten Wertes zu ändern, muss im "engine"-Bereich nichts geändert werden, da die Logik hierfür in den Drools-Regeln liegt. Dies erleichtert die Erweiterung und Anpassung des Systems.

Services, welche Drools-Regeln ausführen, werden von SCXML-Aktionen aus der SCXML-Komponente aufgerufen. Wie bereits beschrieben, existiert für jede Agendagruppe eine eigene Model-Klasse, welche die in den Regeln benötigten Daten enthält. Instanzen der Model-Klassen werden in der SCXML-Komponente erstellt; danach werden die Regeln ausgeführt und das Ergebnis an die SCXML-Komponente übergeben, um damit weiterzuarbeiten. Um die Model-Instanz sowie das Ergebnis zu bearbeiten, bevor diese verwendet werden, können sogenannte *Interceptor*-Klassen erstellt werden. Das `Interceptor`-Interface (siehe Abbildung 19) hat die generischen Parameter `TModel`, `TResultInput`, `TResultOutput`. `TModel` gibt den Typ der `ServiceModel`-Klasse an. `TResultInput` gibt den Typ des Ergebnisses an, das von den Regeln zurückgeliefert wird, `TResultOutput` den Typ des Ergebnisses, mit dem die SCXML-Komponente weiterarbeitet. Die Methode `modifyModel` kann verwendet werden, um das Model, bevor die Drools-Regeln mit dem Model ausgeführt werden, zu verändern (oder durch eine Unterklasse zu ersetzen). `modifyResult` kann verwendet werden, um das Ergebnis, bevor damit weitergearbeitet wird, zu modifizieren bzw. in den korrekten Typ überzuleiten (z. B. aus einer Liste von *ConfidenceResult*-Ergebnissen jenes mit der höchsten Confidence auswählen). Für jede `DroolsService`-Klasse existiert auch ein `Interceptor`-Interface, das von `Interceptor` erbt und die generischen Parameter mit der jeweiligen Model-Klasse bzw. des Ergebnis-Typs festlegt.

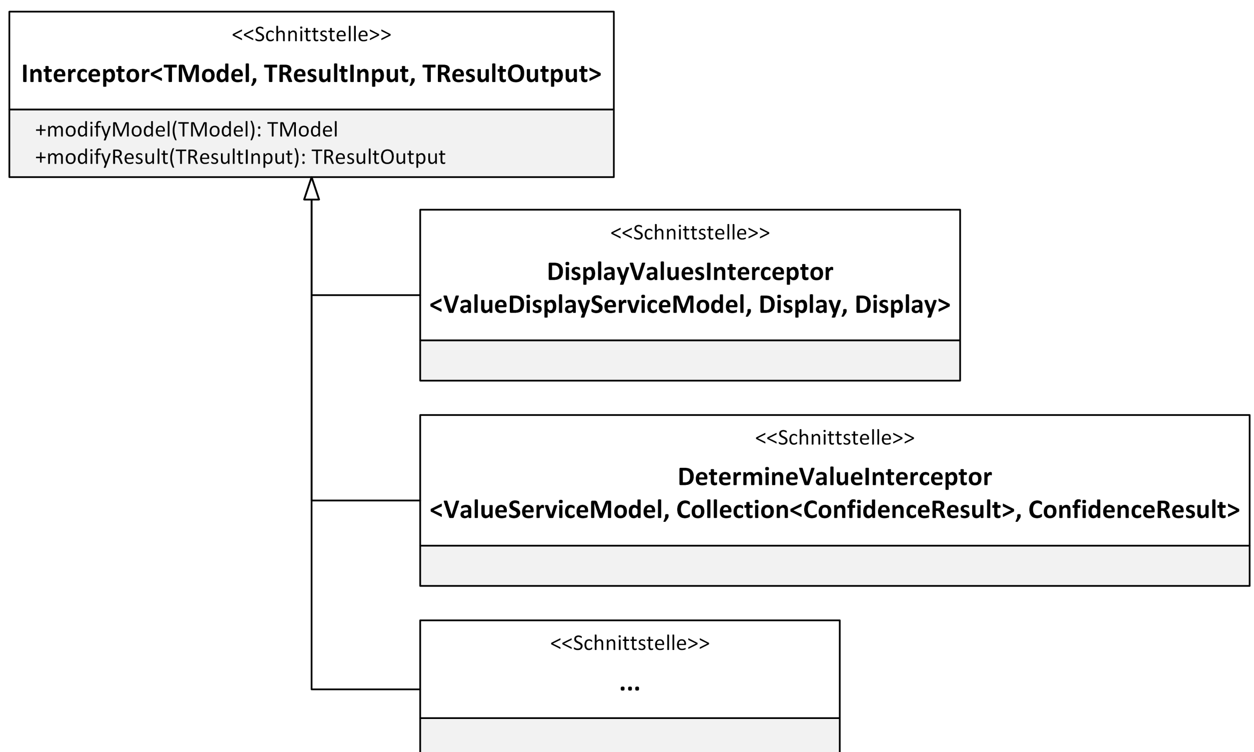


Abbildung 19: Klassendiagramm: Interceptor

Die SCXML-Aktion versucht mittels Dependency Injection eine Instanz einer Klasse, die die benötigte Interceptor-Schnittstelle implementiert, zu erstellen. Falls keine passende Klasse gefunden wurde, wird ein Standardverhalten angewendet. Andernfalls werden die Methoden des Interceptors aufgerufen.

In dieser Arbeit wurden Interceptors verwendet, um die Ähnlichkeiten zwischen Eingaben und Auswahlmöglichkeiten zu ermitteln und eine Model-Instanz zurückzuliefern, welche diese Ähnlichkeiten enthalten.

6.9. Mehrsprachigkeit

Das Backend wurde bereits, für die Unterstützung von mehreren Sprachen vorbereitet. Um eine weitere Sprache zu unterstützen, müssen zunächst für alle Schema-Elemente Bezeichnungen (*rdfs:label*) und Details (*rdfs:comment*) für die zu unterstützende Sprache in der RDF-Repräsentation des multidimensionalen Modells eingefügt werden. Des Weiteren muss für alle Resource-Bundles, welche unter anderem die Bezeichnungen der möglichen Operationen und deren Synonyme festlegen, eine neue Datei erstellt und die Werte übersetzt werden.

Um Natural Language Processing für die neue Sprache zu unterstützen, müssen die entsprechenden Stanford CoreNLP-Modelle zu den Projektabhängigkeiten hinzugefügt werden. Allerdings unterstützt Stanford CoreNLP zum Zeitpunkt des Schreibens dieser Arbeit Dependency Parsing und Constitutency Parsing nur für die Sprachen Englisch, Deutsch, Französisch und Chinesisch[7]. In weiterer Folge müssen für die Ermittlung von Wortgruppen für die neue Sprache eigene Regeln mit Tregex- und Semgrex-Regeln definiert werden, da für andere Sprachen zum Teil andere Part-Of-Speech-Tags und Dependency-Arten existieren.

In weiterer Folge ist es notwendig, um semantische Ähnlichkeiten zwischen Wortgruppen und Schema-Elementen zu finden, einen Similarity-Index für die Sprache zu erstellen. Hier muss nach einem passenden Knowledge Graph gesucht werden, der es ermöglicht semantisch ähnliche Begriffe in der gewünschten Sprache miteinander in Beziehung zu setzen. Dies bedingt auch die Erstellung von eigenen Abfragen für den Similarity-Index der neuen Sprache.

[7] https://stanfordnlp.github.io/CoreNLP/human-languages.html

7. Implementierungsdetails Frontend

Das Frontend wurde als Einzelseiten-Webanwendung bzw. Single-Page Application (SPA) mit dem Framework Angular (Google, 2019) entwickelt, da dadurch unabhängig vom Betriebssystem auf jedem Computer mit installiertem Google Chrome Browser darauf zugegriffen werden kann. Das Konzept von SPAs ist, dass eine Webanwendung in zwei Teile aufgeteilt wird (Kuuskeri, 2011): Server, der Dienste zur Verfügung stellt und Client, der die Dienste verwendet. Die Client-Anwendung läuft vollständig im Browser und auf die Dienste des Servers wird nur zugegriffen, wenn diese benötigt werden. Ein weiterer Vorteil der Entkopplung ist, dass Server und Client unabhängig voneinander entwickelt werden können und dass alle Arten von Clients auf die Dienste zugreifen können. Der Server stellt die Dienste zum Beispiel über RESTful-Schnittstellen oder über WebSockets zur Verfügung. Der Client dieser Anwendung dient nur zur Anzeige von Informationen und zur Eingabe von Kommandos durch den Benutzer. Die gesamte Anwendungslogik befindet sich am Server.

In Kapitel 7.1 werden zunächst Details zu den Komponenten genannt. Kapitel 7.2 beschreibt die Services, welche Funktionen für WebSockets, Text-to-Speech und Speech-to-Text zur Verfügung stellen. In Kapitel 7.3 wird beschrieben, wie das Frontend mehrere Sprachen unterstützen kann.

7.1. Komponenten

Die Komponenten des Frontends wurden bereits in Kapitel 4.2 vorgestellt. Für das Design wurde das kostenlose Bootstrap-basierte Theme *SB Admin 2*[8] verwendet und den eigenen Anforderungen angepasst. In diesem Kapitel folgen die Implementierungsdetails der Komponenten.

Listing 16 zeigt den Inhalt der Hauptkomponente der Anwendung, der die Seitenstruktur vorgibt, und die weiteren Komponenten beinhaltet. Diese werden allerdings nur angezeigt, wenn der Webbrowser WebSockets unterstützt. Andernfalls wird eine Fehlermeldung angezeigt. Die Überprüfung der WebSocket-Unterstützung erfolgt, indem überprüft wird, ob das JavaScript `window`-Objekt ein Element namens `WebSocket` oder `MozWebSocket` enthält.

Listing 16: Inhalt der Haupt-Komponente

```
1  <div id="wrapper" *ngIf="supportsWebsockets">
2    <div id="content-wrapper" class="d-flex flex-column">
3      <div id="content">
4        <!-- Begin Heading -->
5        <div class="bg-white static-top shadow text-center mb-4 py-4">
6          <h1 class="h2 text-gray-800" [translate]="'site.header'">IDA Voice
               User Interface</h1>
7        </div>
8        <!-- End Heading -->
9        <!-- Begin Page Content -->
10       <div class="row mx-3">
11         <div class="col-12 col-lg-6">
12           <app-connection-panel></app-connection-panel>
13           <app-voice-settings-panel></app-voice-settings-panel>
```

[8] https://startbootstrap.com/themes/sb-admin-2/

```
14          <app-input-panel></app-input-panel>
15          <app-display-panel></app-display-panel>
16        </div>
17        <div class="col-12 col-lg-6">
18          <app-analysis-result-panel></app-analysis-result-panel>
19        </div>
20      </div>
21
22      <div class="row mx-3">
23        <div class="col-12">
24          <app-result-panel></app-result-panel>
25        </div>
26      </div>
27    </div>
28    <!-- End Page Content -->
29    <!-- Footer -->
30    <footer class="sticky-footer bg-white">
31      <div class="container my-auto">
32        <div class="copyright text-center my-auto">
33          <span>Martin Strasser</span>
34        </div>
35      </div>
36    </footer>
37    <!-- End of Footer -->
38  </div>
39 </div>
40
41 <div class="alert alert-danger" *ngIf="!supportsWebsockets"
     [translate]="'site.websocketError'">
42   We're sorry but IDA doesn't work without Websockets and TextEncoder. Your
       browser does not support Websockets.
43   Please use an up-to-date browser.
44 </div>
```

Folgend werden alle Komponenten des Frontends aufgelistet und erklärt:

app-connection-panel
Enthält die Repräsentation und Funktionalität der Verbindungs-Komponente.

app-voice-settings-panel
Enthält die Repräsentation und Funktionalität der Spracheinstellungs-Komponente.

app-input-panel
Enthält die Repräsentation und Funktionalität der Eingabe-Komponente.

app-display-panel
Enthält die Repräsentation und Funktionalität der Anzeige-Komponente. Je nach Art der empfangenen Daten (siehe auch Abbildung 12) wird in dieser Komponente eine entsprechende Komponente dargestellt, wobei vier unterschiedliche Arten existieren:

- **app-error-display**: Eine Komponente zum Anzeigen von Fehlermeldungen.
- **app-message-display**: Eine Komponente zum Anzeigen von Nachrichten ohne Auswahlmöglichkeit.
- **app-list-display**: Eine Komponente zur Anzeige einer einfachen Liste.
- **app-two-list-display**: Eine Komponente zur Anzeige von zwei Listen nebeneinander.

app-result-panel

Enthält die Repräsentation und Funktionalität der Ergebnis-Komponente.

app-analysis-situation-panel

Enthält die Repräsentation der Analysesituation-Komponente. Je nach Art der Analysesituation (Comparative oder Non-Comparative) wird die entsprechende Komponente dargestellt.

- **app-non-comparative**: Stellt eine nicht vergleichende Analysesituation dar.
- **app-comparative**: Stellt eine vergleichende Analysesituation dar, wobei für den *Context of Interest* und für den *Context of Comparison* die Non-Comparative Komponente wiederverwendet wird.

7.2. Services

Neben den Komponenten wurden auch Service-Klassen entwickelt, die Funktionalitäten für Speech-To-Text, Text-To-Speech und WebSockets zur Verfügung stellen. Diese Services sind Singleton-Services, das heißt es existiert pro Service nur eine Instanz. Sie werden mittels Dependency Injection in die Komponenten, in denen das Service benötigt wird, injiziert.

Ein Vorteil von Verwendung eigener Services ist, dass ein Service eine einfache Schnittstelle für Funktionen einer bestimmten Art zur Verfügung stellt. Somit müssen nicht die nativen JavaScript-Schnittstellen verwendet werden, bei denen zum Teil je nach Browser Unterschiede existieren. Ein weiterer Vorteil ist, dass der Zustand an einer einzigen Stelle in der Anwendung verwaltet wird und der Zugriff auf geteilte Ressourcen einfacher möglich ist.

Weiters übernehmen die Services die Verwaltung der Einstellungen (z. B. Lautstärke). Die Einstellungen werden direkt im Browser gespeichert, wobei unterschiedliche Möglichkeiten zur Speicherung existieren: Cookies, Session Storage (Speichert Einstellungen solange, bis die aktuelle Sitzung des Browser geschlossen wird) und Local Storage (im Gegensatz zum Session Storage bleiben hier die Einstellungen auch nach schließen des Browsers erhalten) (Mozilla Developer Network, 2019b). Bei Session- und Local Storage werden die Einstellungen als Schlüssel-Wert-Paare gespeichert. In dieser Anwendung wird Local Storage verwendet, damit die Einstellungen auch nach einem Browser-Neustart noch vorhanden sind.

Listing 17 zeigt eine Funktion, um eine Einstellung aus dem Local Storage auszulesen. Tabelle 12 zeigt alle Einstellungswerte, die im Local Storage gespeichert werden.

Listing 17: Einstellung auslesen

```
private getValueFromStorage(key: string, fallback) {
    const val = localStorage.getItem(key);
    if (!val) {
        return fallback;
    }
    return val;
}
```

Tabelle 12: Frontend-Einstellungen

Einstellung	Standardwert	Beschreibung
ida.url	ws://localhost:8080/ws	URL des WebSockets
ida.lang	en	Ausgewählte Sprache
ida.voice.rate	1	Sprachgeschwindigkeit (Wertebereich: 1 - 10)
ida.voice.pitch	1	Stimmhöhe (Wertebereich: 0 - 2)
ida.voice.volume	1	Lautstärke (Wertebereich: 0 - 1)
ida.voice.autoStart	no	Spracherkennung automatisch starten (Wertebereich: yes, no)
ida.voice.[Sprache]	Standardsprache	Für jede Sprache wird die ausgewählte Sprache gespeichert. [Sprache] ist zum Beispiel mit "en-GB" zu ersetzen. Meist ist die Standardsprache für Englisch "Microsoft Zira Desktop - English".

Connection Service Wie bereits in Kapitel 6.1 beschrieben, werden für die Kommunikation zwischen dem Backend und dem Frontend WebSockets mit dem STOMP-Protokoll verwendet. Das Connection Service verwaltet die WebSocket-Verbindung. Für das Versenden von STOMP-Nachrichten über WebSockets wird die *stompjs*-Bibliothek verwendet. Wenn eine Verbindung hergestellt wird, werden zuerst die Warteschlangen aus Tabelle 7 registriert und danach die Start-Nachricht gesendet. Wenn auf einer der Warteschlangen eine Nachricht eintrifft oder wenn sich der Status der Verbindung ändert (z. B. Verbindung getrennt), löst das Service ein entsprechendes Ereignis aus. Die Komponenten können sich für Ereignisse eines bestimmten Typs (Display, Analysesituation, Ergebnis) registrieren und bei Eintreffen eines Ereignisses darauf reagieren. Das Service stellt auch eine Methode zum Versenden einer Nachricht bereit. Listing 18 zeigt diese Methode zum Versenden. In der Methode wird zuerst überprüft, ob die WebSocket-Verbindung initialisiert und eine Verbindung hergestellt wurde. Falls dies nicht der Fall ist, wird ein Fehler erzeugt. Andernfalls wird das JavaScript-Objekt in das JSON-Format konvertiert und an den Input-Endpoint gesendet.

Listing 18: Nachricht senden

```
sendMessage(input: string): void {
    if (this.client == null) {
      throw new Error('Client not initialized.');
    }
```

```
    if (!this.isConnected()) {
      throw new Error('Client not connected.');
    } else {
      const bodyData = JSON.stringify({userInput: input});
      this.client.publish({
        destination: '/app/input',
        body: bodyData
      });
    }
}
```

Text-to-Speech Service Dieses Service verwendet die Web Speech API (Mozilla Developer Network, 2019a), welche zur Zeit des Schreibens dieser Arbeit den Draft-Status hat, zur Sprachausgabe. Die Sprachausgabe wird von allen aktuellen Browsern unterstützt. Das Service verwaltet die Einstellungen wie Lautstärke, Geschwindigkeit oder verwendete Sprache. Das Service stellt Methoden zum Starten und Beenden der Sprachausgabe zur Verfügung. Listing 19 zeigt die Methode dieses Services, um einen Text über die Sprachausgabe auszugeben. Hierzu wird zuerst überprüft, ob der Text leer ist. Falls dies der Fall ist, erfolgt keine Sprachausgabe. Wenn Text-to-Speech vom Browser nicht unterstützt wird, erfolgt auch keine Sprachausgabe. Wenn der Text nicht leer ist und Sprachausgabe unterstützt wird, werden zuerst aktuell laufende Sprachausgaben abgebrochen. Danach wird eine neue *Utterance* unter Verwendung der Spracheinstellungen erstellt und der Sprachausgabe übergeben.

Listing 19: Sprachausgabe

```
speak(text: string): void {
    if (text == null || text.trim().length === 0)
      return;
    if (!TextToSpeechService.isSupported())
      return;

    // Stop current utterance
    this.synth.cancel();

    // Build utterance
    const utter = new SpeechSynthesisUtterance(text);
    if (this._voice)
      utter.voice = this._voice;

    utter.lang = this._language;
    utter.volume = this._volume;
    utter.rate = this._rate;
    utter.pitch = this._pitch;

    // Speak
    this.synth.speak(utter);
}
```

Speech-to-Text Service Dieses Service verwendet zur Spracherkennung ebenfalls die Web Speech API, wobei keine Grammatiken verwendet werden, sondern alle möglichen Eingaben zulässig sind. Die Spracherkennung wird zum Zeitpunkt des Schreibens dieser Arbeit nur von Google Chrome ab Version 33, sowie von Google Chrome für Android unterstützt (Mozilla Developer Network, 2019a). Das Service stellt Methoden zum Starten und Beenden der Spracherkennung zur Verfügung. Weiters löst das Service Ereignisse aus, wenn die Spracherkennung gestartet bzw. gestoppt wurde, und wenn das Ergebnis der Spracherkennung zur Verfügung steht.

7.3. Mehrsprachigkeit

Das Frontend wurde bereits, genauso wie das Backend, für die Unterstützung von mehreren Sprachen vorbereitet. Unter Verwendung der Bibliothek *ngx-translate* können alle Texte in andere Sprachen übersetzt werden. Ein Beispiel für eine Übersetzung ist in Listing 16 in Zeile 6 ersichtlich. Das Überschriften-Element in dieser Zeile hat ein Attribut namens `[translate]`, dem der Schlüssel für den Übersetzungswert zugewiesen ist. Die Übersetzungs-Bibliothek sorgt dafür, dass der Text innerhalb des Elements durch den entsprechenden Wert für den Schlüssel in der aktuell ausgewählten Sprache ersetzt wird. Pro Sprache existiert eine JSON-Datei, in der alle Übersetzungen enthalten sind.

Um im Frontend eine weitere Sprache zu unterstützen, muss in der Verbindungs-Komponente bei der Sprachauswahl eine weitere Option hinzugefügt werden. Weiters muss für die neue Sprache eine neue JSON-Datei mit dem Namen der Sprache im Ordner `src/assets/i18n` erstellt werden (z. B. fr.json für Französisch). Wenn ein Übersetzungsschlüssel in einer Datei nicht gefunden wurde, wird der Wert der englischen Übersetzungsdatei verwendet.

Um die Spracherkennung und die Sprachausgabe für eine weitere Sprache zu unterstützen, müssen in den Klassen `TextToSpeechService` und `SpeechToTextService` jeweils die `setLanguage`-Methode angepasst werden. Da das Backend nur mit Sprachkürzeln ohne Länderangabe arbeitet (z. B. "en" anstatt "en-GB"), die Spracherkennung und Sprachausgabe aber Sprachkürzel mit Länderangabe erwarten, muss in diesen Methoden das Länderkürzel hinzugefügt werden. Listing 20 zeigt den Teil der Methoden, der angepasst werden muss, um eine weitere Sprache zu unterstützen, in dem das Land mit angegeben wird.

Listing 20: Anpassen des Sprachkürzels für Sprachausgabe- und erkennung

```
if (language != null &&
        ((language.trim().length === 5 && language.charAt(2) === '-') ||
        language.trim().length === 2)) {
    if (value.length === 2) {
        if (value === 'en') {
          value = 'en-GB';
        }
        if (value === 'de') {
          value = 'de-DE';
        }
    }
    ...
}
```

8. Fazit und Ausblick

Diese Arbeit beschreibt die Benutzung, Architektur und Implementierung einer dialogbasierten Benutzungsschnittstelle für die interaktive Datenanalyse in Data Warehouses. Das Backend wurde in der Programmiersprache Java entwickelt und stellt die gesamte Anwendungslogik zur Verfügung. Generell wurde bei der Entwicklung darauf geachtet, dass die Logik für die Ermittlung der anzuzeigenden Möglichkeiten und die Ermittlung der Benutzerauswahl ohne große Änderungen im Kern der Anwendung durchgeführt werden können. Um die Kommunikation mit dem Benutzer zu strukturieren wurde State-Chart-XML verwendet. Die Auswahl der möglichen Operationen bzw. Werte werden mittels Drools-Regeln ermittelt, wodurch die Anwendung leicht angepasst werden kann. Zur Ermittlung der Auswahl wird in dieser Arbeit Natural-Language-Processing verwendet. Mit Hilfe von Stanford CoreNLP werden Wortgruppen in den Benutzereingaben ermittelt und deren Ähnlichkeit zu den Auswahlmöglichkeiten in weiterer Folge mittels Distanzmetriken bzw. Similarity-Index berechnet. Um eine gültige Analysesituation aus einer Freitexteingabe zu generieren wird das Constraint-Satisfaction-Framework Optaplanner verwendet.

Das Frontend ist eine Single-Page-Application und wurde mit dem Angular-Framework entwickelt. Die Kommunikation mit dem Backend erfolgt über WebSockets, über die die anzuzeigenden Auswahlmöglichkeiten, Analysesituationen, Abfrageergebnisse sowie die Benutzereingaben übertragen werden. Für die Sprachein- und ausgabe kommt die JavaScript Web Speech API zur Anwendung, welche zurzeit nur von Google Chrome zur Gänze unterstützt wird.

Die vorgestellte Benutzungsschnittstelle verwendet vordefinierte Prädikate und abgeleitete Kennzahlen aus dem Enriched Dimensional Fact Model für die Abfrageformulierung. In Zukunft könnte das System erweitert werden, sodass auch die Ad-hoc-Definition von Prädikaten und abgeleiteten Kennzahlen möglich ist. Außerdem können in zukünftigen Arbeiten Semantic OLAP-Patterns (Kovacic et al., 2018) und Analysegraph-Schemata (Neuböck et al., 2012) für die Verbesserung der natürlichsprachlichen Benutzungsschnittstelle verwendet werden. Dadurch könnte die Komplexität der unterstützten Abfragen erhöht werden. Das System könnte in Zukunft weiter ausgebaut werden, um mehr Navigationsoperatoren zu unterstützen. Aufgrund der einfachen Anpassbarkeit wäre es auch möglich Machine Learning einzusetzen, um Benutzereingaben Schema-Elementen zuzuordnen. Hierfür sind allerdings Trainingsdaten notwendig, welche jedoch im Falle eines Natural Language Interface to Databases nicht unbedingt vorhanden sind (Dhamdhere et al., 2017). Weiters wäre es möglich Heuristiken und Schlüsselwörter zu verwenden, um zum Beispiel zu ermitteln, ob eine Wortgruppe in einer Benutzereingabe eher einem Granularitätslevel oder einer Kennzahl entspricht.

Die natürlichsprachliche Benutzungsschnittstelle könnte in Zukunft mit einer klassischen grafischen Benutzeroberfläche kombiniert werden. Im Frontend werden zurzeit die Ergebnisse in einer einfachen Tabelle dargestellt. Hier könnte man die Option zur Verfügung stellen, Ergebnisse in Diagrammen zu visualisieren.

Literatur

Alpar, P. & Schulz, M. (2016). Self-Service Business Intelligence. *Business & Information Systems Engineering*, *58*(2), 151–155.

Androutsopoulos, I., Ritchie, G. D. & Thanisch, P. (1995). Natural Language Interfaces to Databases-An Introduction. *Natural Language Engineering*, *1*(1), 29–81.

Barták, R. (1998). On-line guide to constraint programming. Zugriff 30. April 2019 unter http://kti.mff.cuni.cz/~bartak/constraints/

Barták, R. (2003). Constraint-Based Scheduling: An Introduction for Newcomers. *IFAC Proceedings Volumes*, *36*(3), 75–80.

Berners-Lee, T., Hendler, J. & Lassila, O. (2001). The Semantic Web A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, *284*(5), 34–43.

Bernstein, A. & Kaufmann, E. (2006). GINO – A Guided Input Natural Language Ontology Editor. In I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, . . . L. M. Aroyo (Hrsg.), *The Semantic Web - ISWC 2006*, 5.–9. November 2006 (S. 144–157). Athens, GA, USA: Springer.

Charniak, E. (1997). Statistical Techniques for Natural Language Parsing. *AI Magazine*, *18*(4), 33–43.

Chaudhuri, S. & Dayal, U. (1997). An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, *26*(1), 65–74.

Dhamdhere, K., McCurley, K. S., Nahmias, R., Sundararajan, M. & Yan, Q. (2017). Analyza: Exploring Data with Conversation. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces - IUI '17*, 13.–16. März 2017 (S. 493–504). Limassol, Zypern: ACM Press.

Etcheverry, L. & Vaisman, A. A. (2012). QB4OLAP: A New Vocabulary for OLAP Cubes on the Semantic Web. In *Proceedings of the Third International Conference on Consuming Linked Data*, 12. November 2012 (Bd. 905, S. 27–38). Boston, MA, USA.

Golfarelli, M., Maio, D. & Rizzi, S. (1998). The Dimensional Fact Model: A Conceptual Model For Data Warehouses. *International Journal of Cooperative Information Systems*, *7*(02n03), 215–247.

Golfarelli, M. & Rizzi, S. (2009). *Data Warehouse Design: Modern Principles and Methodologies*. New-York: McGraw-Hill.

Gomez-Perez, J. M., Pan, J. Z., Vetere, G. & Wu, H. (2017). Enterprise Knowledge Graph: An Introduction. In *Exploiting Linked Data and Knowledge Graphs in Large Organisations* (S. 1–14). Cham: Springer.

Google. (2012). Introducing the Knowledge Graph: things, not strings. Zugriff 7. Mai 2019 unter https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html

Google. (2019). Angular. Zugriff 19. Mai 2019 unter https://angular.io/

Gradle. (2019). Gradle Build tool. Zugriff 21. Mai 2019 unter https://gradle.org/

Gur, I., Yavuz, S., Su, Y. & Yan, X. (2018). DialSQL: Dialogue Based Structured Query Generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 15.–20. Juli 2018 (S. 1339–1349). Melbourne, Australia: Association for Computational Linguistics.

Hellerstein, J. M., Avnur, R., Chou, A., Hidber, C., Olston, C., Raman, V., . . . Haas, P. J. (1999). Interactive data analysis: the Control project. *Computer*, *32*(8), 51–59.

Hirschberg, J. & Manning, C. D. (2015). Advances in natural language processing. *Science*, *349*(6245), 261–266.

Inmon, W. H. (2005). *Building the Data Warehouse* (4. Aufl.). Indianapolis: Wiley.

Jurafsky, D. & Martin, J. H. (2018). Speech and Language Processing. Zugriff 2. Mai 2019 unter https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf

Keil, J. M. (2019). Efficient Bounded Jaro-Winkler Similarity Based Search. In T. Grust, F. Naumann, A. Böhm, W. Lehner, T. Härder, E. Rahm, ... H. Meyer (Hrsg.), *BTW - Datenbanksysteme für Business, Technologie und Web*, 4.–8. März 2019 (S. 205–214). Rostock, Deutschland: Gesellschaft für Informatik.

Kim, H. (2007). A Dialogue-Based NLIDB System in a Schedule Management Domain. In J. van Leeuwen, G. F. Italiano, W. van der Hoek, C. Meinel, H. Sack & F. Plášil (Hrsg.), *SOFSEM 2007: Theory and Practice of Computer Science*, 20.–26. Januar 2007 (S. 869–877). Harrachov, Tschechische Republik: Springer.

Kovacic, I., Schuetz, C. G., Schausberger, S., Sumereder, R. & Schrefl, M. (2018). Guided Query Composition with Semantic OLAP Patterns. In *Proceedings of the 2nd International Workshop on Data Analytics Solutions for Real-Life Applications*, 26. März 2018 (S. 67–74). Wien: CEUR.

Kuuskeri, J. (2011). Experiences on a Design Approach for Interactive Web Applications. In *Conference Proceedings of the 2nd USENIX Conference on Web Application Development*, 15.–16. Juni 2011 (S. 87–98). Portland, OR, USA: The USENIX Association.

Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P. N., ... Bizer, C. (2015). DBpedia-A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web*, *6*(2), 167–195.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, *10*(8), 707–710.

Lu, J., Lin, C., Wang, W., Li, C. & Wang, H. (2013). String similarity measures and joins with synonyms. In *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*, 22.–27. Juni 2013 (S. 373–384). New York, New York, USA: ACM.

Manning, C. D. & Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. Cambridge, MA: MIT Press.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J. & McClosky, D. (2014). The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations* (S. 55–60). Baltimore, Maryland, USA: Association for Computational Linguistics.

Mesnil, J. (2012). STOMP Over WebSocket. Zugriff 18. Mai 2019 unter http://jmesnil.net/stomp-websocket/doc/

Miller, G. A. (1995). WordNet: A Lexical Database for English. *Communications of the ACM*, *38*(11), 39–41.

Mittal, A., Sen, J., Saha, D. & Sankaranarayanan, K. (2018). An Ontology based Dialog Interface to Database. In *Proceedings of the 2018 International Conference on Management of Data*, 10.–15. Juni 2018 (S. 1749–1752). Houston, TX, USA: ACM Press.

Mozilla Developer Network. (2019a). Web Speech API - Web APIs. Zugriff 19. Mai 2019 unter https://developer.mozilla.org/de/docs/Web/API/Web_Speech_API

Mozilla Developer Network. (2019b). Window.localStorage - Web API Referenz. Zugriff 19. Mai 2019 unter https://developer.mozilla.org/de/docs/Web/API/Window/localStorage

Nadkarni, P. M., Ohno-Machado, L. & Chapman, W. W. (2011). Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, *18*(5), 544–551.

Nandi, A. & Jagadish, H. V. (2011). Guided Interaction: Rethinking the Query-Result Paradigm. In *Proceedings of the 37th International Conference on Very Large Databases*, 29. August–3. September 2011 (Bd. 4, *12*, S. 1466–1469). Seattle, WA, USA.

Navarro, G. (2001). A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, *33*(1), 31–88.

Neuböck, T. (2019). *Analysis Process Modeling Notation For Business Intelligence (APMN4BI)* (Diss., Johannes Kepler Universität Linz). In Arbeit.

Neuböck, T., Neumayr, B., Rossgatterer, T., Anderlik, S. & Schrefl, M. (2012). Multi-dimensional Navigation Modeling Using BI Analysis Graphs. *Advances in Conceptual Modeling. ER 2012. Lecture Notes in Computer Science*, *7518*, 162–171.

Neuböck, T. & Schrefl, M. (2015). Modelling Knowledge about Data Analysis Processes in Manufacturing. *IFAC-PapersOnLine*, *48*(3), 277–282.

Neumayr, B., Schrefl, M. & Linner, K. (2011). Semantic Cockpit: An Ontology-Driven, Interactive Business Intelligence Tool for Comparative Data Analysis. *Advances in Conceptual Modeling. Recent Developments and New Directions*, *6999*, 55–64.

Nivre, J., De Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajič, J., Manning, C. D., . . . Zeman, D. (2016). Universal Dependencies v1: A Multilingual Treebank Collection. In N. Calzolari, K. Choukri, T. Declerck, S. Goggi, M. Grobelnik, B. Maegaard, . . . S. Piperidis (Hrsg.), *Proceedings of the Tenth International Conference on Language Resources and Evaluation*, 23.–28. Mai 2016 (S. 1659–1666). Portorož, Slovenia: European Language Resources Association.

Ontotext. (2019a). GraphDB Free 8.9 Documentation. Zugriff 24. April 2019 unter http://graphdb.ontotext.com/documentation/free/

Ontotext. (2019b). Semantic similarity searches — GraphDB Free 8.9 documentation. Zugriff 24. April 2019 unter http://graphdb.ontotext.com/documentation/free/semantic-similarity-searches.html

Pan, Z., Zhu, T., Liu, H. & Ning, H. (2018). A survey of RDF management technologies and benchmark datasets. *Journal of Ambient Intelligence and Humanized Computing*, *9*(5), 1693–1704.

Paulheim, H. (2017). Knowledge Graph Refinement: A Survey of Approaches and Evaluation Methods. *Semantic Web*, *8*(3), 489–508.

Pivotal Software. (2019). Spring Boot. Zugriff 18. Mai 2019 unter https://spring.io/projects/spring-boot

Rebele, T., Suchanek, F., Hoffart, J., Biega, J., Kuzey, E. & Weikum, G. (2016). YAGO: A Multilingual Knowledge Base from Wikipedia, Wordnet, and Geonames. In P. Groth, E. Simperl, A. Gray, M. Sabou, M. Krötzsch, F. Lecue, . . . Y. Gil (Hrsg.), *The Semantic Web - ISWC 2016*, 17.–21. Oktober 2016 (S. 177–185). Kobe, Japan: Springer.

Red Hat. (2019). OptaPlanner - Constraint satisfaction solver. Zugriff 29. April 2019 unter https://www.optaplanner.org/

Schütz, C. G., Neumayr, B., Schrefl, M. & Neuböck, T. (2016). Reference Modeling for Data Analysis: The BIRD Approach. *International Journal of Cooperative Information Systems*, *25*(2), 1650006.

Schütz, C. G., Schausberger, S., Kovacic, I. & Schrefl, M. (2017). Semantic OLAP Patterns: Elements of Reusable Business Analytics. *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*, *10574*, 318–336.

Setlur, V., Battersby, S. E., Tory, M., Gossweiler, R. & Chang, A. X. (2016). Eviza: A Natural Language Interface for Visual Analysis. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology - UIST '16*, 16.–19. Oktober 2016 (S. 365–377). Tokio, Japan: ACM Press.

The Apache Software Foundation. (2015). Commons SCXML. Zugriff 17. Mai 2019 unter https://commons.apache.org/proper/commons-scxml/

Tsang, E. (1996). *Foundations of Constraint Satisfaction*. London: Academic Press.

Unland, R. (2019). Wissensrepräsentation. Zugriff 6. Mai 2019 unter http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/daten-wissen/Wissensmanagement/Wissensmodellierung/Wissensreprasentation

Vrandečić, D. & Krötzsch, M. (2014). Wikidata. *Communications of the ACM*, *57*(10), 78–85.

W3C. (2013). SPARQL 1.1 Overview. Zugriff 7. Mai 2019 unter https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/

W3C. (2014). RDF 1.1 Concepts and Abstract Syntax. Zugriff 7. Mai 2019 unter https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

Widdows, D. & Ferraro, K. (2008). Semantic Vectors: A Scalable Open Source Package and Online Technology Management Application. In *Sixth international conference on Language Resources and Evaluation*, 28.–30. Mai 2008. Marrakesch, Marokko.

Zhu, Y., Yan, E. & Song, I.-Y. (2017). A natural language interface to a graph-based bibliographic information retrieval system. *Data & Knowledge Engineering*, *111*, 73–89.

A. Materialisierungsabfragen

Die folgenden Abfragen fügen die Zuordnung von WordNet-Nomen zu multidimensionalen Schema-Elementen in den Named Graph `http://dke.jku.at/ida/similarity#wordnet-scores-en` ein. Abbildung 20 zeigt das RDF-Schema der Zuordnungen.

Zuordnungen sind vom Typ *Mapping*, deren IRI das Format "mapping_[UUID]" hat. *from* verweist hierbei auf das Schema-Element, *to* auf das WordNet-Nomen. *hasScore* definiert den Ähnlichkeits-Score. *inCube* und, falls zutreffend, *inDim* geben an, zu welchem Cube und zu welcher Dimension ein Schema-Element gehört. Diese Informationen werden auch in der Zuordnung abgespeichert, um zusätzliche Abfragen zu vermeiden. *fromType* gibt den Typ des Schema-Elements an.

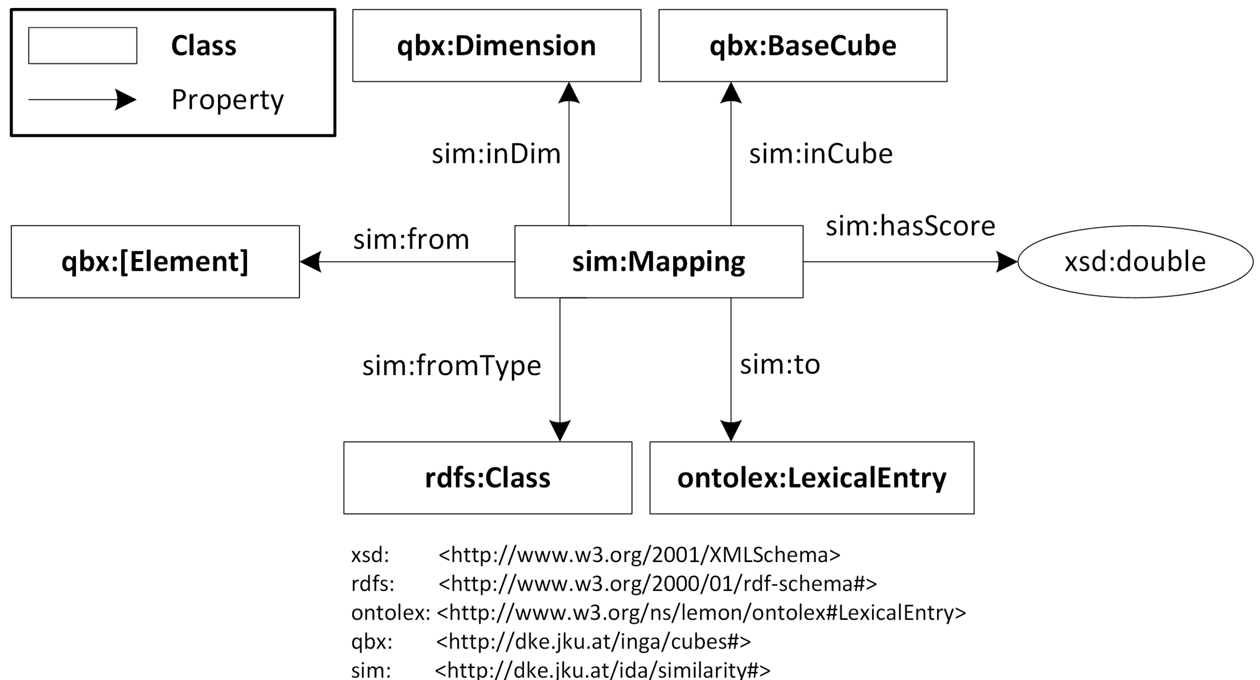


Abbildung 20: RDF-Schema der materialisierten Zuordnung

Listing 21: Materialisierungs-Abfrage: Aggregate Measure

```
PREFIX : <http://www.ontotext.com/graphdb/similarity/>
PREFIX inst: <http://www.ontotext.com/graphdb/similarity/instance/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX qbx: <http://dke.jku.at/inga/cubes#>
PREFIX sim: <http://dke.jku.at/ida/similarity#>

INSERT {
    GRAPH sim:wordnet-scores-en {
        ?mapping rdf:type sim:Mapping ;
                 sim:inCube   ?cube ;
                 sim:from     ?element ;
                 sim:to       ?documentID ;
                 sim:hasScore ?score ;
                 sim:fromType qbx:AggregateMeasure.
    }
```

```
17 }
18 WHERE
19 {
20     SELECT ?mapping ?cube ?element ?documentID ?score
21     WHERE {
22         ?search a inst:wordnet ;
23                 :searchTerm         ?filterText ;
24                 :searchParameters   "" ;
25                 :documentResult     ?result .
26         ?result :value ?documentID ;
27                 :score ?score .
28         {
29             SELECT ?cube ?element ?filterText
30             WHERE {
31                 ?cube qbx:measure ?measure .
32                 ?measure rdf:type qbx:BaseMeasure .
33 
34                 ?element rdf:type        qbx:AggregateMeasure ;
35                          qbx:derivedFrom ?measure ;
36                          rdfs:label      ?filterText .
37 
38                 FILTER(isLiteral(?filterText)) .
39                 FILTER(lang(?filterText) = "en") .
40             }
41         }
42         BIND(IRI(CONCAT("sim:mapping_", STRUUID())) AS ?mapping)
43     }
44 }
```

Listing 22: Materialisierungs-Abfrage: Aggregate Measure Predicate

```
1  PREFIX : <http://www.ontotext.com/graphdb/similarity/>
2  PREFIX inst: <http://www.ontotext.com/graphdb/similarity/instance/>
3  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
5  PREFIX qbx: <http://dke.jku.at/inga/cubes#>
6  PREFIX sim: <http://dke.jku.at/ida/similarity#>
7 
8  INSERT {
9      GRAPH sim:wordnet-scores-en {
10         ?mapping rdf:type sim:Mapping ;
11                  sim:inCube   ?cube ;
12                  sim:from     ?element ;
13                  sim:to       ?documentID ;
14                  sim:hasScore ?score ;
15                  sim:fromType qbx:AggregateMeasurePredicate.
16     }
17 }
18 WHERE
19 {
```

```
20      SELECT ?mapping ?cube ?element ?documentID ?score
21      WHERE {
22          ?search a inst:wordnet ;
23                  :searchTerm          ?filterText ;
24                  :searchParameters    "" ;
25                  :documentResult      ?result .
26          ?result :value ?documentID ;
27                  :score ?score .
28          {
29              SELECT ?cube ?element ?filterText
30              WHERE {
31                  ?cube qbx:measure ?measure .
32                  ?measure rdf:type qbx:BaseMeasure .
33
34                  ?aggMeasure rdf:type         qbx:AggregateMeasure ;
35                              qbx:derivedFrom+ ?measure .
36
37                  ?element rdf:type    qbx:AggregateMeasurePredicate ;
38                           qbx:over    ?aggMeasure ;
39                           rdfs:label ?filterText.
40
41                  FILTER(isLiteral(?filterText)) .
42                  FILTER(lang(?filterText) = "en") .
43              }
44          }
45          BIND(IRI(CONCAT("sim:mapping_", STRUUID())) AS ?mapping)
46      }
47  }
```

Listing 23: Materialisierungs-Abfrage: Base Measure Predicate

```
1   PREFIX : <http://www.ontotext.com/graphdb/similarity/>
2   PREFIX inst: <http://www.ontotext.com/graphdb/similarity/instance/>
3   PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4   PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
5   PREFIX qbx: <http://dke.jku.at/inga/cubes#>
6   PREFIX sim: <http://dke.jku.at/ida/similarity#>
7
8   INSERT {
9       GRAPH sim:wordnet-scores-en {
10          ?mapping rdf:type sim:Mapping ;
11                   sim:inCube    ?cube ;
12                   sim:from      ?element ;
13                   sim:to        ?documentID ;
14                   sim:hasScore ?score ;
15                   sim:fromType qbx:BaseMeasurePredicate.
16      }
17  }
18  WHERE
19  {
```

```
    SELECT ?mapping ?cube ?element ?documentID ?score
    WHERE {
        ?search a inst:wordnet ;
                :searchTerm         ?filterText ;
                :searchParameters   "" ;
                :documentResult     ?result .
        ?result :value ?documentID ;
                :score ?score .
        {
            SELECT ?cube ?element ?filterText
            WHERE {
                ?cube qbx:measure ?measure .
                ?measure rdf:type qbx:BaseMeasure .

                ?element rdf:type   qbx:BaseMeasurePredicate ;
                         qbx:over   ?measure ;
                         rdfs:label ?filterText.

                FILTER(isLiteral(?filterText)) .
                FILTER(lang(?filterText) = "en") .
            }
        }
        BIND(IRI(CONCAT("sim:mapping_", STRUUID())) AS ?mapping)
    }
}
```

Listing 24: Materialisierungs-Abfrage: Level Predicate

```
PREFIX : <http://www.ontotext.com/graphdb/similarity/>
PREFIX inst: <http://www.ontotext.com/graphdb/similarity/instance/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX qbx: <http://dke.jku.at/inga/cubes#>
PREFIX sim: <http://dke.jku.at/ida/similarity#>

INSERT {
    GRAPH sim:wordnet-scores-en {
        ?mapping rdf:type sim:Mapping ;
                 sim:inCube    ?cube ;
                 sim:inDim     ?dimension ;
                 sim:from      ?element ;
                 sim:to        ?documentID ;
                 sim:hasScore ?score ;
                 sim:fromType qbx:LevelPredicate.
    }
}
WHERE
{
    SELECT ?mapping ?cube ?dimension ?element ?documentID ?score
    WHERE {
```

```
23          ?search a inst:wordnet ;
24                  :searchTerm          ?filterText ;
25                  :searchParameters    "" ;
26                  :documentResult      ?result .
27          ?result :value ?documentID ;
28                  :score ?score .
29          {
30              SELECT DISTINCT ?cube ?dimension ?element ?filterText
31              WHERE {
32                  {
33                      ?element rdf:type qbx:LevelPredicate ;
34                               qbx:over ?level .
35                  }
36                  UNION
37                  {
38                      ?pred qbx:over ?level .
39                      ?element rdf:type qbx:ConjunctiveLevelPredicate ;
40                               qbx:conjunct ?pred .
41                  }
42
43                  {
44                      SELECT DISTINCT ?cube ?dimension ?level
45                      WHERE {
46                          {
47                              ?cube qbx:dimension ?dimension .
48                        ?dimension qbx:hasHierarchy ?hier .
49                        ?hs qbx:inHierarchy ?hier ;
50                          qbx:childLevel ?level .
51                          } UNION {
52                              ?cube qbx:dimension ?dimension .
53                        ?dimension qbx:hasHierarchy ?hier .
54                        ?hs qbx:inHierarchy ?hier ;
55                          qbx:parentLevel ?level .
56                          }
57                      }
58                  }
59
60                  ?element rdfs:label ?filterText .
61                  FILTER(isLiteral(?filterText)) .
62                  FILTER(lang(?filterText) = "en") .
63              }
64          }
65          BIND(IRI(CONCAT("sim:mapping_", STRUUID())) AS ?mapping)
66      }
67  }
```

Listing 25: Materialisierungs-Abfrage: Granularity Level

```
PREFIX : <http://www.ontotext.com/graphdb/similarity/>
PREFIX inst: <http://www.ontotext.com/graphdb/similarity/instance/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX qbx: <http://dke.jku.at/inga/cubes#>
PREFIX sim: <http://dke.jku.at/ida/similarity#>

INSERT {
    GRAPH sim:wordnet-scores-en {
        ?mapping rdf:type sim:Mapping ;
                 sim:inCube    ?cube ;
                 sim:inDim     ?dimension ;
                 sim:from      ?element ;
                 sim:to        ?documentID ;
                 sim:hasScore ?score ;
                 sim:fromType qbx:Level.
    }
}
WHERE
{
    SELECT ?mapping ?cube ?dimension ?element ?documentID ?score
    WHERE {
        ?search a inst:wordnet ;
                :searchTerm          ?filterText ;
                :searchParameters    "" ;
                :documentResult      ?result .
        ?result :value ?documentID ;
                :score ?score .
        {
            SELECT DISTINCT ?cube ?dimension ?element ?filterText
            WHERE {
              {
                SELECT ?cube ?dimension ?element
                WHERE {
                      ?cube qbx:dimension ?dimension .
                      ?dimension qbx:hasHierarchy ?hier .
                      ?hs qbx:inHierarchy ?hier ;
                          qbx:parentLevel ?element .
                }
              }
              UNION
              {
                SELECT ?cube ?dimension ?element
                WHERE {
                      ?cube qbx:dimension ?dimension .
                      ?dimension qbx:hasHierarchy ?hier .
                      ?hs qbx:inHierarchy ?hier ;
                          qbx:childLevel ?element .
                }
```

```
50                  }
51
52                     ?element rdfs:label ?filterText .
53                     FILTER(isLiteral(?filterText)) .
54                     FILTER(lang(?filterText) = "en") .
55                }
56            }
57            BIND(IRI(CONCAT("sim:mapping_", STRUUID())) AS ?mapping)
58        }
59  }
```

B. Verwendete Bibliotheken

Tabelle 13 zeigt die verwendeten Bibliotheken des Backends, Tabelle 14 die des Frontends. Es werden nur die Bibliotheken aufgelistet, die in den Gradle-Dependencies bzw. in der Datei `package.json` direkt definiert wurden. Bibliotheken, die von den aufgelisteten Bibliotheken referenziert werden, sind nicht aufgelistet.

Tabelle 13: Verwendete Bibliotheken für das Backend

Bibliothek	Version	Lizenz
edu.stanford.nlp:stanford-corenlp	3.9.2	GNU GPL v3
edu.stanford.nlp:stanford-corenlp:models	3.9.2	GNU GPL v3
edu.stanford.nlp:stanford-corenlp:models-english	3.9.2	GNU GPL v3
edu.stanford.nlp:stanford-corenlp:models-english-kbp	3.9.2	GNU GPL v3
info.debatty:java-string-similarity	1.2.1	MIT
com.ibm.icu:icu4j	64.1	http://www.unicode.org/copyright.html#License
com.ontotext.graphdb:graphdb-free-runtime	8.8.1	http://graphdb.ontotext.com/LICENSE-GraphDB-Free.txt
com.google.guava:guava:27.1-jre	27.1-jre	Apache License 2.0
org.hibernate:hibernate-validator	6.0.15.Final	Apache License 2.0
org.optaplanner:optaplanner-core	7.20.0.Final	Apache License 2.0
org.drools:drools-core	7.20.0.Final	Apache License 2.0

Weiter auf der nächsten Seite

Verwendete Bibliotheken für das Backend - Fortsetzung

Bibliothek	**Version**	**Lizenz**
org.drools:drools-compiler	7.20.0.Final	Apache License 2.0
org.kie:kie-api	7.20.0.Final	Apache License 2.0
org.apache.commons:commons-scxml2	2.0-M1	Apache License 2.0
org.apache.commons:commons-jexl	2.1.1	Apache License 2.0
org.apache.commons:commons-csv	1.6	Apache License 2.0
org.apache.commons:commons-lang3	3.8.1	Apache License 2.0
org.reflections:reflections	0.9.11	WTFPL
com.squareup.okhttp3:okhttp	3.14.1	Apache License 2.0
org.springframework.boot:spring-boot-starter	2.1.3.RELEASE	Apache License 2.0
org.springframework.boot:spring-boot-starter-web	2.1.3.RELEASE	Apache License 2.0
org.springframework.boot:spring-boot-starter-websocket	2.1.3.RELEASE	Apache License 2.0
org.springframework.boot:spring-boot-starter-test	2.1.3.RELEASE	Apache License 2.0
org.springframework.boot:spring-boot-configuration-processor	2.1.3.RELEASE	Apache License 2.0
com.openpojo:openpojo	0.8.12	Apache License 2.0
org.junit.jupiter:junit-jupiter-api	5.4.1	Eclipse Public License 2.0
org.junit.jupiter:junit-jupiter-engine	5.4.1	Eclipse Public License 2.0

Tabelle 14: Verwendete Bibliotheken für das Frontend

Bibliothek	**Version**	**Lizenz**
@angular/animations	7.2.0	MIT
@angular/common	7.2.0	MIT
@angular/compiler	7.2.0	MIT
@angular/core	7.2.0	MIT

Weiter auf der nächsten Seite

Verwendete Bibliotheken für das Frontend - Fortsetzung

Bibliothek	Version	Lizenz
@angular/forms	7.2.0	MIT
@angular/platform-browser	7.2.0	MIT
@angular/platform-browser-dynamic	7.2.0	MIT
@angular/router	7.2.0	MIT
@ng-bootstrap/ng-bootstrap	4.1.0	MIT
@ngx-translate/core	11.0.1	MIT
@ngx-translate/http-loader	4.0.0	MIT
@stomp/stompjs	5.2.0	MIT
core-js	2.5.4	MIT
papaparse	4.6.2	MIT
rxjs	6.3.3	Apache License 2.0
tslib	1.9.0	Apache License 2.0
zone.js	0.8.26	MIT
@angular-devkit/build-angular	0.13.0	MIT
@angular/cli	7.3.5	MIT
@angular/compiler-cli	7.2.0	MIT
@angular/language-service	7.2.0	MIT
@biesbjerg/ngx-translate-extract	2.3.4	MIT
@types/node	8.9.4	MIT
@types/papaparse	4.5.9	MIT
codelyzer	4.5.0	MIT
cpx	1.5.0	MIT
ts-node	7.0.0	MIT
tslint	5.11.0	Apache License 2.0
typescript	3.2.2	Apache License 2.0
startbootstrap-sb-admin-2	4.0.0	MIT

C. RDF-Schema

Schema-Elemente von Data Warehouses werden in dieser Arbeit in einer eigens entwickelten RDF-Repräsentation gespeichert. Jedes Element muss ein Label (`rdfs:label`) mit Sprachangabe zugewiesen haben. Optional kann jedem Element ein Kommentar (`rdfs:comment`) mit Sprachangabe zugewiesen werden, das das Element genauer beschreibt. Abbildung 21 zeigt das RDF-Schema.

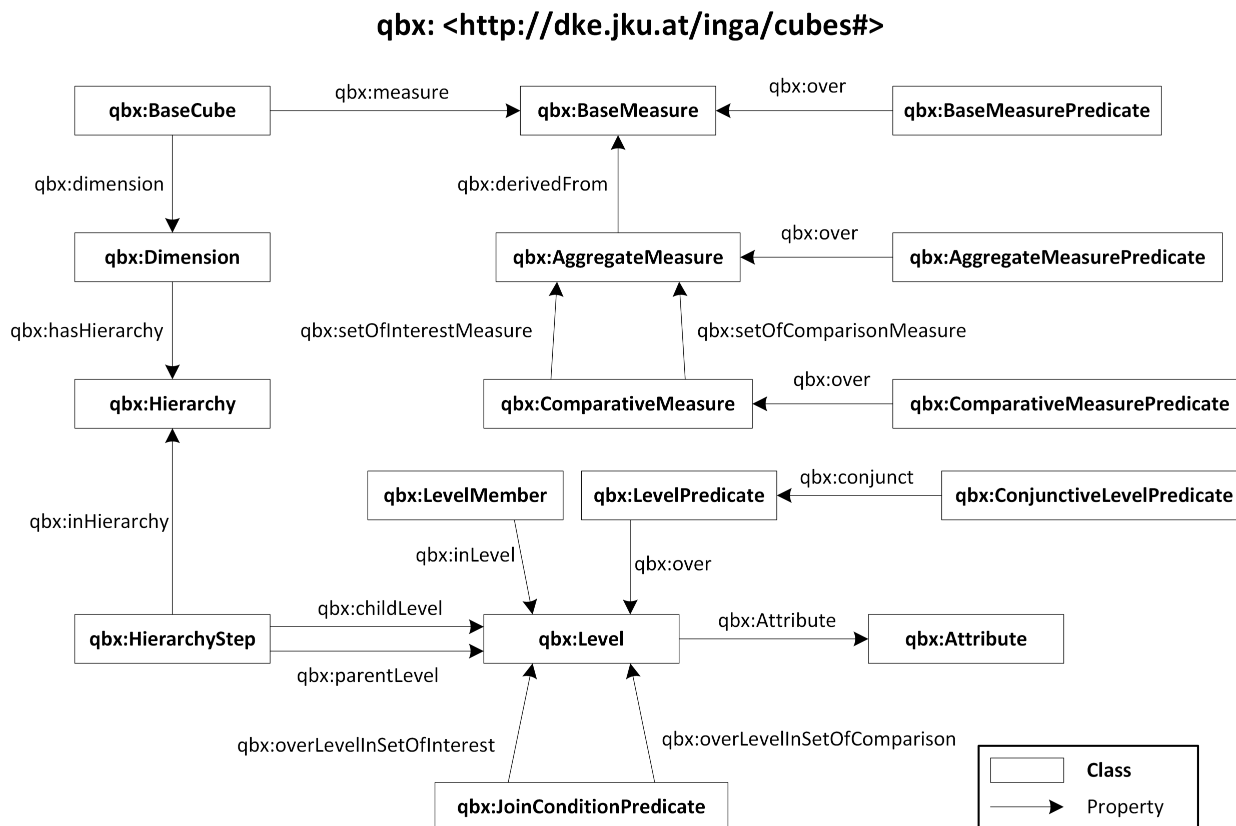


Abbildung 21: RDF-Schema

Die Beispiele in den folgenden Listings repräsentieren Teile des Beispiels aus Abbildung 4.

BaseCube repräsentiert einen Cube und verweist auf BaseMeasures (Kennzahlen) und Dimensionen.

Listing 26: RDF-Schema Beispiel: Base Cube

```
drugs:DrugPrescriptionCube a qbx:BaseCube ;
  rdfs:label "Drug Prescription"@en ;
  rdfs:comment "Drug Prescription Cube"@en ;
  qbx:measure drugs:QuantityMeasure ;
  qbx:measure drugs:CostsMeasure ;
  qbx:dimension drugs:DrugDimension ;
  qbx:dimension drugs:TimeDimension  ;
  qbx:dimension drugs:DoctorDimension ;
  qbx:dimension drugs:InsurantDimension .
```

Eine **BaseMeasure** repräsentiert eine Kennzahl.

Listing 27: RDF-Schema Beispiel: Base Measure

```
drugs:QuantityMeasure a qbx:BaseMeasure ;
  rdfs:label "Quantity"@en .
```

BaseMeasurePredicates beziehen sich auf BaseMeasures und werden verwendet, um das Abfrageergebnis zu filtern.

Listing 28: RDF-Schema Beispiel: Base Measure Predicate

```
drugs:HighCostsPerUnitPredicate a qbx:BaseMeasurePredicate ;
  rdfs:label "High Costs per Unit"@en ;
  rdfs:comment "costs / quantity > 50"@en ;
  qbx:over drugs:QuantityMeasure ;
  qbx:over drugs:CostsMeasure .
```

AggregateMeasures sind sind Kennzahlen, die BaseMeasures mittels einer Aggregatsfunktion (z. B. sum, avg, ...) aggregieren.

Listing 29: RDF-Schema Beispiel: Aggregate Measure

```
drugs:SumQuantityMeasure a qbx:AggregateMeasure ;
  rdfs:label "Sum of Quantity"@en ;
  rdfs:comment "Sum of Quantity; Expression: sum(quantity);
      aggregationfunction: sum"@en ;
  qbx:derivedFrom drugs:QuantityMeasure .
```

AggregateMeasurePredicates beziehen sich auf AggregateMeasures und werden verwendet, um das Abfrageergebnis zu filtern.

Listing 30: RDF-Schema Beispiel: Aggregate Measure Predicate

```
drugs:HighDrugPrescriptionCostsPerUnitPredicate a
    qbx:AggregateMeasurePredicate ;
  rdfs:label "High Drug Prescription Costs per Unit"@en ;
  rdfs:comment "Costs Per Unit > 50"@en ;
  qbx:over drugs:CostsPerUnitMeasure .
```

ComparativeMeasures legen fest, welche Kennzahlen zum Vergleich verwendet werden (im Falle einer vergleichenden Analysesituation).

Listing 31: RDF-Schema Beispiel: Comparative Measure

```
drugs:ChangeOfCostsMeasure a qbx:ComparativeMeasure ;
  rdfs:label "Change of costs"@en ;
  qbx:setOfInterestMeasure drugs:SumCostsMeasure ;
  qbx:setOfComparisonMeasure drugs:SumCostsMeasure .
```

ComparativeMeasurePredicates werden verwendet, um das Abfrageergebnis auf Basis der Comparative-Measures zu filtern.

Listing 32: RDF-Schema Beispiel: Comparative Measure Predicate

```
drugs:SignificantChange a qbx:ComparativeMeasurePredicate ;
  rdfs:label "Change of costs"@en ;
  rdfs:comment "Change over 10%"@en ;
  qbx:over drugs:ChangeOfCostsMeasure .
```

Dimension definiert eine Dimension eines Cubes. Eine Dimension kann mehrere Hierarchien beinhalten. Diese werden benötigt, um alternative bzw. parallele Pfade abzubilden.

Listing 33: RDF-Schema Beispiel: Dimension

```
drugs:DoctorDimension a qbx:Dimension ;
  rdfs:label "Doctor"@en ;
  qbx:hasHierarchy drugs:DoctorDimensionMainHierarchy .
```

Hierarchy definiert eine Hierarchie einer Dimension und besteht aus mehreren HierarchySteps.

Listing 34: RDF-Schema Beispiel: Hierarchy

```
drugs:DoctorDimensionMainHierarchy a qbx:Hierarchy ;
  rdfs:label "Doctor"@en .
```

HierarchyStep legt die Hierarchie zwischen den Levels fest.

Listing 35: RDF-Schema Beispiel: Hierarchy Step

```
drugs:DoctorDimensionDoctorToDocDistrict a qbx:HierarchyStep ;
  qbx:inHierarchy drugs:DoctorDimensionMainHierarchy ;
  qbx:childLevel drugs:DoctorDimensionDoctorLevel ;
  qbx:parentLevel drugs:DoctorDimensionDocDistrictLevel .
```

Level gehört zu einer Dimension und kann in mehreren Hierarchien vorkommen.

Listing 36: RDF-Schema Beispiel: Level

```
drugs:DoctorDimensionDoctorLevel a qbx:Level ;
  rdfs:label "Doctor"@en ;
  qbx:attribute drugs:DoctorDimensionDoctorLevelDocAgeAttribute .
```

Attribute enthält zusätzliche Informationen zu einem Level.

Listing 37: RDF-Schema Beispiel: Attribute

```
drugs:DoctorDimensionDoctorLevelDocAgeAttribute a qbx:Attribute ;
  rdfs:label "Doctor Age"@en .
```

LevelPredicates werden verwendet um das Ergebnis basierend auf den Werten von Leveln einzuschränken.

Listing 38: RDF-Schema Beispiel: Level Predicate

```
drugs:DocInRuralDistrictPredicate a qbx:LevelPredicate ;
  rdfs:label "Doctor in Rural District"@en ;
  rdfs:comment "Less than 400 inhabitants per square-km"@en ;
  qbx:over drugs:DoctorDimensionDocDistrictLevel .
```

ConjunctiveLevelPredicates verknüpfen mehrere LevelPredicates und werden auch zum Filtern verwendet.

Listing 39: RDF-Schema Beispiel: Conjunctive Level Predicate

```
drugs:OldDoctorInRuralDistrictPredicate a qbx:ConjunctiveLevelPredicate ;
  rdfs:label "Old Doctor in Rural District"@en ;
  qbx:conjunct drugs:DocInRuralDistrictPredicate ;
  qbx:conjunct drugs:OldDoctorPredicate .
```

LevelMember repräsentiert einen Wert eines Levels.

Listing 40: RDF-Schema Beispiel: Level Member

```
drugs:DrDolillte a qbx:LevelMember ;
  rdfs:label "Dr. Dolittle"@en ;
  qbx:inLevel drugs:DoctorDimensionDoctorLevel .
```

JoinConditionPredicate definiert, wie bei einer vergleichenden Analysesituation die zwei Analysesituationen verknüpft werden sollen.

Listing 41: RDF-Schema Beispiel: Join Condition Predicate

```
drugs:YearOnYear a qbx:JoinConditionPredicate ;
  rdfs:label "year-on-year"@en ;
  qbx:overLevelInSetOfInterest drugs:TimeDimensionYearLevel ;
  qbx:overLevelInSetOfInterest drugs:TimeDimensionMonthLevel ;
  qbx:overLevelInSetOfComparison  drugs:TimeDimensionYearLevel ;
  qbx:overLevelInSetOfComparison  drugs:TimeDimensionMonthLevel .
```