Masterarbeit

Visualisierung heterogener Datensätze in strukturierten Informationsressourcen

Visualization of Heterogeneous Datasets in Structured Information Resources

Belgin MUTLU

Knowledge Management Institute (KMI)
Technische Universität Graz
Vorstand: Univ.-Prof. Dipl.-Inf. Dr. Stefanie Lindstaedt

Begutachter: Univ.-Prof. Dipl.-Inf. Dr. Stefanie Lindstaedt
Betreuer: Dipl.-Ing. (FH) Patrick Höfler

Graz, im Mai 2012

Kurzfassung

Das Konzept des Semantic Web sieht vor, Informationen anhand ihrer inhaltlichen Zusammenhänge strukturiert anzubieten, wodurch sie mit Hilfe der ihnen zugewiesenen Schlagwörter auffindbar gemacht werden können. Der Zugriff auf diese Daten funktioniert mit Hilfe der so genannten SPARQL-Suchanfragen (Queries), indem gezielt angegeben wird, welche Daten aus dieser großen Menge extrahiert werden sollen. Nach dem Erhalt können diese Daten weiterverarbeitet und unter anderem für eine benutzerfreundliche Darstellung visualisiert werden. Die Visualisierung der semantischen Daten hat mittlerweile eine sehr große Bedeutung im Bereich des Wissensmanagements und ist auch das Thema dieser Masterarbeit.

Es wurde im Bereich der Visualisierung heterogener Daten schon einiges realisiert, dennoch wurde dabei auf das Thema der Wiederverwendung wenig eingegangen. Das Ziel dieser Arbeit ist einen generischen Ansatz zur visuellen Repräsentation der heterogenen Daten anzubieten. Die Idee der generischen Lösung basiert dabei auf dem Konzept der systematischen Wiederverwendung, nämlich der Softwareproduktlinien.

Das für diesen Zweck entwickelte Framework unterstützt eine Reihe von interaktiven Diagrammen. Der Benutzer kann auf diesem Framework für eine SPARQL-Query eine Visualisierung durchführen und sie anschließen speichern. Um die gespeicherten Diagramme wieder zu verwenden kann das Framework anhand einer Query kontaktiert und das fertige Diagramm clientseitig angezeigt werden. Das Framework wurde so konzipiert, dass der Client ohne großen Aufwand ein Diagramm erstellen und in der Rolle des Entwicklers sogar das Framework um neue Diagramme erweitern kann, die dann als Visualisierungsvorlage angeboten werden.

Die am Ende durchgeführte quantitative Evaluierung hat gezeigt, dass für dieses Framework vorgenommene Ansatz verglichen mit der traditionelle Methode zur Visualisierung effizienter ist.

Abstract

The concept of the semantic web provides a pragmatic way for the structured information retrieval based on relationships within a content of the linked data. This retrieval is realized by applying the specific queries, i.e. SPARQL queries, on well-constructed data models. Besides the filter and navigation functions, a result data collection can also be used for visualization purposes, i.e. to allow, for instance, the statistical analysis of the content. The visualization of the semantic data is a trend nowadays but it is still confronted with challenges originating from the heterogeneity of the linked data. These challenges represent the ongoing research in the knowledge management area and are the main focus of the work in this thesis.

Recent literature research tailored to visualization of heterogeneous data offers a set of approaches, but unfortunately with a rare focus on strategic reuse of the existing solutions. The aim of this thesis is the provision of a generic solution for the outlined problem. The solution follows the well-known method for systematic reuse, i.e. software product lines. The units of reuse are in this context charts whereas a result data collection corresponds to the specification and the input data simultaneously.

From the practical perspective a framework with an integrated set of important charts has been developed. It defines several roles that may use it from different views. The client in the administrator role, for instance, can define the correlation between a chart and specific result data collection. This manually specified correlation is used by the client in the user role in order to generate the final chart. The provided framework is conceived, on the one hand, to free the user from the required effort for the visualization and, on the other hand, to allow the developer to easily extend the supported result data collection.

Finally, the conducted evaluation results quantitatively show that the applied solution outperforms the traditional methods used for visualization of the linked data.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz,am ....
(Unterschrift)

Danksagung

Diese Masterarbeit wurde im Rahmen des Frameworks „Linked Data For You“ am Knowledge Management Institute an der Technischen Universität Graz durchgeführt.

Als erstes möchte ich mich bei Patrick Höfler bedanken, der mich mit seinen Fachkenntnissen während der gesamte Entstehungsphase dieser Arbeit unterstützt hatte und jederzeit erreichbar war. Auch Prof. Dr. Michael Granitzer möchte ich großen Dank aussprechen, der mir die Möglichkeit gab diese Arbeit zu verwirklichen. Besonders bedanken möchte ich mich bei Univ.-Prof. Dr. Stefanie Lindstaedt, die sich dazu bereit erklärte, meine Betreuung und die Begutachtung meiner Arbeit zu übernehmen, nach dem Prof. Dr. Michael Granitzer das Institut verlassen hatte.

Schließlich gilt mein Dank meinen Eltern, meinem Bruder Engin, meinem Opa Ali, meinen Onkeln Ali Ekber und Nermin, die mich im Laufe meiner Ausbildung immer unterstützt, mir beigestanden und immer an mich geglaubt haben.

Graz, im Mai 2012 Belgin Mutlu

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
DOM	Document Object Model
DTD	Document Object Definition
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IRI	Internationalized Resource Identifier
JSON	JavaScript Object Notation
LD4U	Linked Data For You
LD4UViz	Linked Data For You-Visualization
LOC	Lines of Code
LOD	Linked Open Data
MCF	Meta Content Framework
MIT	Massachusetts Institute of Technology
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
REST	Representational State Transfer
SAX	Simple API for XML
SOA	Serviceorientierte Architektur
SPL	Software Product Lines
SVG	Scalable Vector Graphics
URI	Uniform Resource Identifier
VML	Vector Markup Language
W3C	World Wide Web Consortium
XML	Extensible Markup Language

1 Einführung

Das Problem der Visualisierung von semantischen Daten lässt sich der Klasse der Interoperabilitätsprobleme zuordnen. Die Heterogenität dieser Daten und ihre schwache Konsistenz (Typfehler, fehlende Einträge, usw.) erschweren die Existenz einer generischen Lösung, die für alle Datensätze anwendbar sein könnte.

Diese Arbeit befasst sich mit der Problematik, solche heterogene und nicht-konsistente Datensätze in ein Framework zu integrieren und zu visualisieren. Das Ergebnis der Arbeit soll zeigen, wie weit die Grenzen einer generischen Lösung zur Visualisierung von semantischen Daten gehen und wo ein Verbesserungspotential liegt.

1.1 Problemstellung

Durch das Framework LD4U (Linked Data For You) wird ein interaktiver Ansatz zum Durchsuchen heterogener Daten aus unterschiedlichen SPARQL-Endpoints realisiert.Die Daten beziehen sich auf verschiedene Domänen und sind untereinander verlinkt (DBpedia). Das LD4U Framework zeichnet sich durch die Erfassung, Verarbeitung und die interaktive Visualisierung der Daten aus. Der IST-Zustand der Visualisierung ist noch immer ein Problem, weil der Prozess zur Erfassung der Daten von einem SPARQL-Endpoint (als SPARQL Ergebnis) bis zur Visualisierung komplett manuell abläuft. Dieser Prozess beinhaltet:

- die Abholung der Daten für ein gegebenes SPARQL.
- die Anpassung des Formats der SPARQL-Ergebnisse an dem Muster der Visualisierung.
- die Routine zur clientseitigen Visualisierung der Ergebnisse.

Die Schritte (2) und (3) unterscheiden sich von Visualisierung zur Visualisierung, wonach der Aufwand für jede neue Visualisierung relativ hoch wird und sich manchmal nicht auszahlt.

Das LD4UViz Framework soll eine intuitive und aufwandslose Visualisierung ermöglichen. Das Ziel der Arbeit ist die Schritte (2) und (3) im Visualisierungsprozess möglichst zu automatisieren, wodurch dem Entwickler die aufwändigen Details verdeckt bleiben und die Anwendbarkeit der interaktiven Visualisierung sich durch einfaches Prinzip erhöht.

1.2 Anforderungen

Das Framework LD4UViz wird durch funktionale und nicht-funktionale Anforderungen begrenzt. Die funktionalen Anforderungen sind vom Projektleiter vorgegeben. Die nicht-

funktionalen Anforderungen ergeben sich aus dem folgenden Grund:

Die aktuellen Forschungsarbeiten im Bereich des Semantic Webs sehen die Problematik der Visualisierung von Linked Data bei der Heterogänität dieser Daten. Diese Arbeit soll in diesem Forschungbereich etwas beitragen. Die Erweiterbarkeit, die intuitive bedienbarkeit und die Eigenschaft, dass dieses Framework stand-alone laufen soll, stellen dabei die Basis für den Umgang mit den heterogenen Daten dar.

Als nächstes werden diese Anforderungen genauer erläutert.

1.2.1 Funktionale Anforderungen

R1: unterstützte Diagramme Es ist zu definieren, welche Diagramme im Framework unterstützt werden. Idealerweise sollte diese Menge auch erweiterbar sein.

R2: Mapping-Regel Es ist zu definieren, wie die Komponenten der Diagramme (seine Achsen) bzgl. ihrem Datentyp auf SPARQL-Komponenten (Subjekte, Prädikate und Objekte) zu mappen sind. Diese Mapping-Regeln sollen als Grundlage für die automatisierte Generierung der Visualisierung dienen.

R3: Constraints Die Komponenten der Diagramme sind normalerweise mit geeigneter Semantik versehen, wie z.B. Gültigkeitsbereich, optionale Komponenten (Achsen), usw. Diese Constraints müssen auch vom Framework berücksichtigt werden.

1.2.2 Nicht-Funktionale Anforderungen

R4: das Framework soll intuitiv bedienbar sein Der Teil des LD4U Frameworks auf der Clientseite ist ein Such-Wizzard, in dem die Spezialisierung der Suche stattfindet. Bezüglich R1 gibt es die Möglichkeit das Visualisierungsframework (LD4UViz) in das LD4U Framework zu integrieren, um die Suche intuitiv gestalten zu können.

R5: das Framework soll möglichst viele heterogene Domänen abdecken Selbst die generische Visualisierung eines Bruchteils von DBpedia stellt eine große Herausforderung dar und ist bisher in der Literatur nur für kleinere und stark zusammenhängende Domänen behandelt worden. Das Ziel dieser Arbeit ist die Domänenabdeckung zu erweitern.

R6: das Framework soll erweiterbar sein im Sinne, dass dem Entwickler die Möglichkeit gegeben wird Visualisierungen von nicht unterstützten Domänen selbst spezifizieren zu können Abdeckung aller Domänen mit idealen Suchergebnissen wird sehr wahrscheinlich auch allein für die DBpedia Datenquelle auf Hindernisse stoßen. Trotzdem soll es möglich sein das Framework aus der Entwicklerseite erweiterbar zu machen, sodass manuell spezifiziert werden kann, wie der Zusammenhang zwischen mancher Daten dargestellt wird.

R7: das Framework soll stand-alone laufen Das zu entwickelnde Framework soll nicht nur an das bestehende LD4U System gebunden, sondern auch stand-alone, z.B. als ein dediziertes Service, ansprechbar sein.

1.3 Aufbau der Arbeit

Dieses Dokument ist wie folgend konzipiert:

Abschnitt 2 befasst sich kurz mit der Enstehungsgeschichte vom Semantic Web, mit dem Stand der Technik im Bereich der Semantischen Web-Technologien und stellt das Thema der Informationsvisualisierung und das grobe Designkonzept des LD4UViz Frameworks vor.

Das Kapitel Designkonzept beschäftigt sich mit den Fragen, wie die heterogenen Daten aus einer SPARQL-Repository erfasst und anschließend visuell repräsentiert werden können und worauf man dabei achten sollte. Im Bereich der Visualisierung von heterogenen Daten wurden schon einige Forschungsarbeiten realisiert, die in diesem Kapitel als Referenzarbeiten vorgestellt werden. Die darin enthaltenen Ideen helfen dabei zu entscheiden, wie die vorgenommen Ziele (u.a Fusion der Ergebnisse einer SPARQL-Query mit den Achsen der Diagramme und die Wiederverwendung der erstellten Diagramme) in die Tat umgesetzt werden können.

Das Konzept, das Design und die Implementierung des Prototyps werden im Abschnitt 3 vorgestellt. Hier wird als erstes angegeben, zu welchen Zwecken dieses Framework implementiert wurde. Das Kapitel Konzept befasst sich anschließend u.a. mit der Systemarchitektur und gibt einen ersten Überblick, wie auf dem Framework eine Visualisierung der empfangenen Daten realisiert werden kann. Auch ein wichtiger technischer Hintergrund wird bei diesem Abschnitt ausführlich beschrieben. Es handelt sich dabei um Design Patterns. Das Kapitel Prototyp beschreibt aus unterschiedlichen Perspektiven, wie das Prototyp implementiert wurde und wie der Zugriff auf gespeicherte Diagramme funktioniert.

Die Evaluierung, die ausgeführt wurde, um die Effizienz des Frameworks zu untermauern, findet man im Abschnitt 4. Eine Zusammenfassung der Arbeit ist im Abschnitt 5 vorhanden. Dieser Abschnitt dient dazu, die Ergebnisse, die mit dem Framework erzielt wurden, zu erwähnen, sie mit den vorgenommen Zielen, die im Kapitel 1 aufgelistet sind, zu vergleichen und einige Ansätze einzubringen, die für die zukünftige Erweiterung des Framework hergenommen werden können.

2 Technologische Grundlagen

2.1 Semantic Web

2.1.1 Geschichte

Den Grundstein des Semantic Web legte der damalige Apple-Mitarbeiter R.V.Guha im Jahr 1995. Er entwickelte das *Meta Content Framework (MCF)*, das zur Strukturierung der Metadaten diente. Nachdem R.V.Guha zum Netscape wechselte, entwickelte er mit Tim Bray eine neue Version von *MCF*, die mit XML arbeitete. Diese Entwicklung wurde beim *W3C*[1] (World Wide Web Consortium) eingebracht und führte zum Entwurf des *Resource Description Framework (RDF)*. Die Veröffentlichung von RDF fand im Herbst 1997 statt. Im Jahr 1999 erschien dann die erste RDF-Spezifikation [Bei12].

Im Jahr 2000 stellte Tim Berners-Lee [BLHO01] auf der „XML 2000 Conference“ die Idee des Semantic Web vor. 2001 setzte sich das *W3C* konkreter mit der Realisierung des Semantic Web auseinander. In Folge dessen kam es zur Bildung von verschiedenen Arbeitsgruppen. Sie alle beschäftigten sich intensiv mit dem Semantic Web [Bal11], wodurch eine Reihe von wichtigen Komponenten entstanden.

2.1.2 Einführung in das Semantic Web

Das World Wide Web bietet eine große Anzahl an Informationen, die für die Wissenschaft, Wirtschaft und Unterhaltung von große Bedeutung sein können. Da der Inhalt dieser Informationen zum Teil vielseitig ist, müssen sie beim Bedarf nach einer bestimmten Inhalt durchsucht werden. Nach einem Suchvorgang erhält man jedoch nicht nur Informationen von hoher inhaltlichen Qualität sondern hin und wieder auch mache, die weniger brauchbar sind. Um aus diesen Ergebnissen schlussendlich die relevanten Informationen zu extrahieren, müssen die Suchergebnisse von Menschen interpretiert, kombiniert und aufbereitet werden. Das Konzept des Semantic Web sieht vor auch die Interpretation und die Verarbeitung der Informationen den Maschinen zu überlassen. Die Automatisierung der Prozessschritte bei der Interpretation und der Verarbeitung hat für das Semantic Web eine zentrale Bedeutung [PB12].

Das Semantic Web basiert auf semantischen Metadaten, also Daten, die Daten beschreiben. Mit ihnen kann der Sinn von den Daten ausgedrückt werden. Mit Hilfe dieser beschreibenden Informationen werden die Daten besser erkannt und können dadurch wirkungsvoll verarbeitet, umgeformt und sinnvoll zusammengesetzt werden. Dadurch wird auch ermöglicht die Zusammenhänge zwischen den Daten zu erkennen und sie richtig miteinander zu verknüpfen.

[1] `http://www.w3.org/`

Das Semantic Web kann auch mit den Daten aus einer Datenbank verbunden sein, die viele semantische Informationen enthält. Dadurch wird eine sehr große Menge an nützlichen Daten angeboten.

Mit Hilfe von RDF werden die Metadaten im Semantic Web organisiert und strukturiert. Sie bilden damit die Grundlage für den Einsatz von Domänenontologien [PB12].

> „RDF is an infrastructure that enables the encoding, exchange, and reuse of structured metadata. Search engines, intelligent agents, information brokers, browsers and human users can make use of the semantic information. RDF is an XML application (i. e., its syntax is defined in XML) customized for adding meta-information to Web documents [...].“ [PB12]

Das Semantic Web verfolgt das Ziel die inhaltliche Bedeutung der Informationen so zu verwerten, wodurch der Nutzer bei einer Suchanfrage (Query) diese Informationen automatisch in richtiger Zuordnung erhält.

2.1.3 Technischer Hintergrund

Nach einer ausführlichen Einführung in die Thematik des Semantic Web soll als nächstes die grundlegenden Technologien des Semantic Web untersucht werden. Die Abbildung 2.1 zeigt diese Technologien, verpackt in einem Layer Model.

URI/IRI repräsentieren die Basic Technologie von Semantic Web. URI steht für Uniform Resource Identifier. Es handelt sich dabei um eine Zusammensetzung von Zeichenketten, die zur Identifizierung bzw. Benennung einer Ressource verwendet wird. IRI ist die Abkürzung für Internationalized Resource Identifier, die eine erweiterte Form des ASCII-Zeichensatzes (Unicode) verwendet.

Unicode ist ein universaler Standard zur Kodierung vieler bekannter Zeichenketten.

XML steht für Extensible Markup Language und wird im Semantic Web zur Serialisierung der Ontologien verwendet.

Namespaces dient dazu, um Ressourcen zu gruppieren und Namenskonflikte in XML-Dokumenten zu vermeiden.

XML Query Durchsucht ein XML Dokument.

XML Schema definiert das Typensystem von XML.

RDF und RDFS werden in 2.1.3.2 und 2.1.3.3 ausführlich beschrieben.

Ontology wird in 2.1.3.4 ausführlich beschrieben.

Rules/Query Siehe SPARQL-Query in 2.1.3.7.

Logic Logikschicht beinhaltet Vorschriften zur Verarbeitung von Inhalten aus den unteren Schichten.

Proof In dieser Beweisebene wird die Wahrheit bestimmter RDF-Aussagen mit Hilfe logischer Regel überprüft.

Trust Überprüfung, ob die Aussagen aus vertraunswürdigen Quellen stammen.

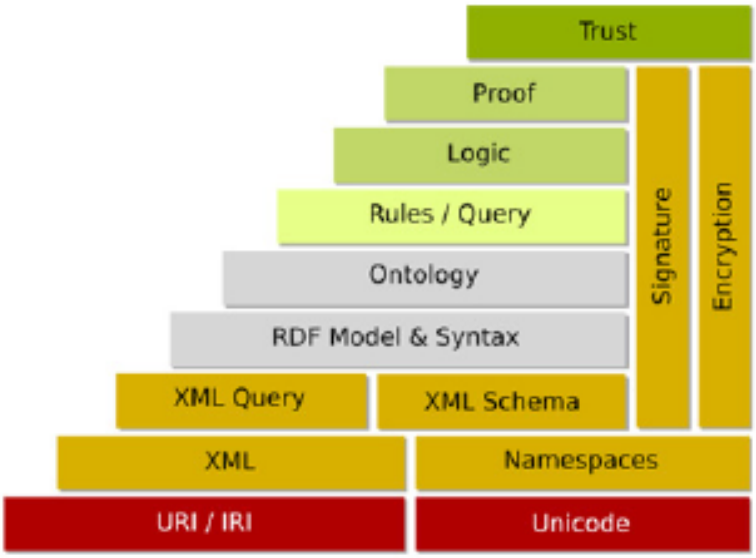

Abbildung 2.1: Schichten des Semantic Web [Jam12]

In den nächsten Kapitel wird näher auf ein paar der oben erwähnten Technologien eingegangen. Es gibt noch weitere Technologien, die zwar oben nicht erwähnt wurden aber zum Bestandteil von Semantic Web gehören. Auch sie werden in den nächsten Kapitel etwas näher erläutern.

2.1.3.1 Linked Open Data

Mit Linked Open Data (LOD) sind im World Wide Web frei verfügbare Daten gemeint. Sie werden per Resource Identifier (URI) erkannt und per HTTP abgerufen. LOD verweisen ebenfalls per URI auf andere Daten.

Resource Description Framework (RDF) und darauf aufbauende Standards wie SPARQL und die Web Ontology Language (OWL) werden zur Kodierung und Verlinkung der

Linked Open Data eingesetzt. Somit wird Linked Open Data ein Teil des Semantic Web. Die zueinander verlinkten Daten bilden dadurch ein weltweites Netz, der auch unter dem Begriff *Linked (Open) Data Cloud* bekannt ist.

Tim Berners Lee stellte Regeln auf, die von denjenigen befolgt werden sollen, der Linked Data veröffentlicht [Den11]:

1. Objekte sollten mit Hilfe der URIs idendifiziert werden.
2. Mit Hilfe der HTTP URIs können sowohl Menschen als auch Web-Ageten besser auf Objekte zugreifen.
3. Verweisen sollten die URIS auf die Informationen, die mit Hilfe von SPARQL und RDF zur Verfügung gestellt werden.
4. Diese Informationen sollten Links auf andere URIs haben, womit immer weiter neues Wissen erworben werden kann.

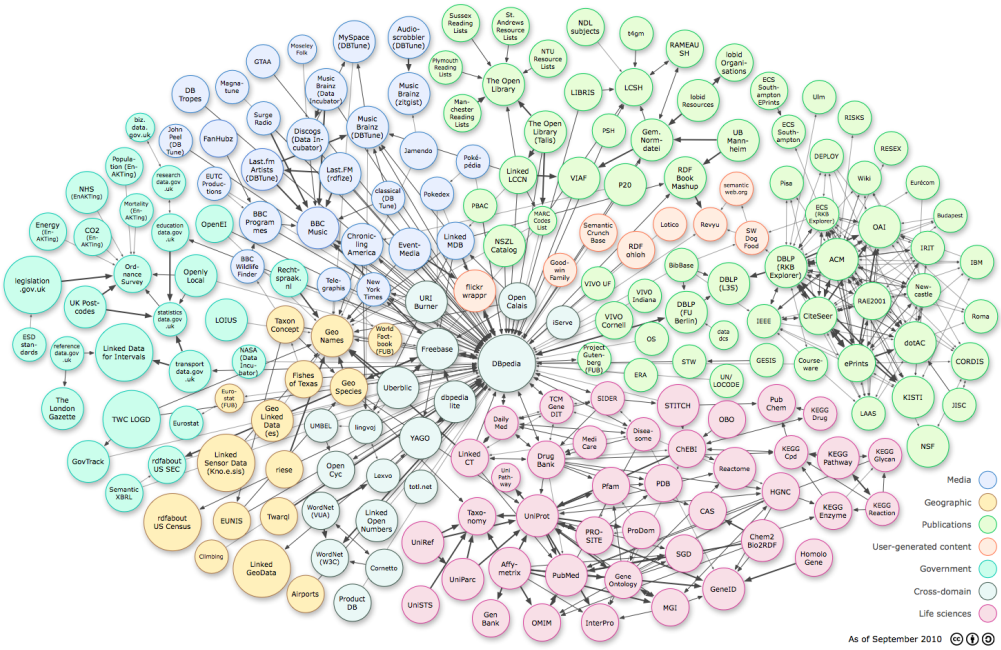

Abbildung 2.2: Verknüpfung zwischen Linked Open Data Datenbeständen [CJ11]

Eine Darstellung von Linked Open Data Cloud ist in Abbildung 2.2 ersichtlich. Die Größe der Kreise ist direkt proportional zu der Anzahl der Tripeln innerhalb einer Datenquelle und die Verlinkungen zwei Datenquellen existieren falls es über 50 Links zwischen diesen gibt.

2.1.3.2 RDF

RDF ist ein Standard des World Wide Web Consortiums (*W3C*), der auf XML basiert. Mit RDF ist es möglich Ressourcen einer Ontologie formal zu beschreiben. Ähnlich wie

bei XML wird die Struktur bzw. das Format vom RDF durch das Schema festgelegt. Die Ressourcen des RDFs werden durch URIs identifiziert. Der Einsatz von RDF geht in die Richtung, alle Daten in Web miteinander zu verlinken um z.B. die Suche nach diesen Daten effizienter zu gestalten. Er findet heute schon Anwendung für verschiedene Applikationen sowie für Katalogdienste, Aggregatoren für Nachrichten und Feeds oder zur allgemeinen Wissenspräsentation. RDF ist wichtiger Bausstein des Semantic Web.

Mit Hilfe des RDF-Modells werden die RDF-Ausdrücke dargestellt. Es besteht aus drei Objekten [Ott11]:

- **Ressourcen** Alles, was sich durch RDF-Ausdrücke beschreiben lässt. Sie werden durch einen eindeutigen URI identifiziert.
- **Eigenschaften** Jeder Ressource kann Eigenschaften besitzen, z.B. Name, Sprache, Geburtsdatum usw.
- **Aussagen** Eine Aussage beinhaltet eine Ressource, eine Relation und einen Wert oder eine weitere Ressource.

Jede RDF-Datei besteht aus einer Sammlung von Tripeln. Diese Tripeln bestehen aus:

- Subjekt
- Prädikat
- Objekt

Das Subjekt ist dabei das Element, über das eine Aussage gemacht wird. Die Art und Ausprägung dieser Aussage bestimmt das Prädikat. Das Objekt wiederum ist der Wert oder Gegenstand dieser Aussage [Tom12]. Eine Menge bestehend aus solcher RDF-Tripel stellt einen RDF-Graph dar.

In Abbildung 2.3 ist ein einfaches Beispiel für ein Tripel zu sehen. Dieses Tripel besagt, dass Michael Schumacher einen Mercedes fährt. „Michael Schumacher“ ist hier das Subjekt, „fährt ein“ das Prädikat und „Mercedes“ das Objekt. Eine RDF-Aussage kann in

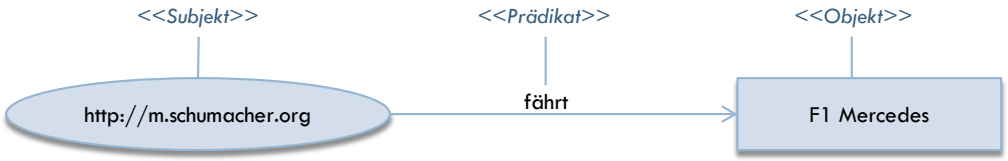

Abbildung 2.3: Graphische Darstellung einer RDF-Aussage

drei unterschiedlichen Formen dargestellt werden [Ott11]:

- **Graphische Darstellung** Die Ellipsen dienen in der graphischen Darstellung zur Beschreibung der Ressourcen und die Rechtecke zur Beschreibung einfacher Datentypen (String, Integer usw.). Die gerichteten Pfeile repräsentieren die Zusammenhänge zwischen den Ressourcen.

- **N-Tripel** Serialisierungsformat für die Übertragung und Speicherung der graphischen Darstellung
- **RDF/XML** Serialisierung erfolgt in Form von XML.

2.1.3.3 RDF-Schema

RDF-Schema bietet die Möglichkeit Begriffe semantisch in einer Beziehung zu bringen und diese Beziehungen zu beschreiben. Er stellt sogenannter *Vokabular* zu Verfügung zur Beschreibung der Klassen, Ressourcen, der Eigenschaften und deren Bezeichnungen zueinander (siehe Abbildung 2.4) [Ott11] .

Kernklassen des RDFS-Vokabulars sind folgende:

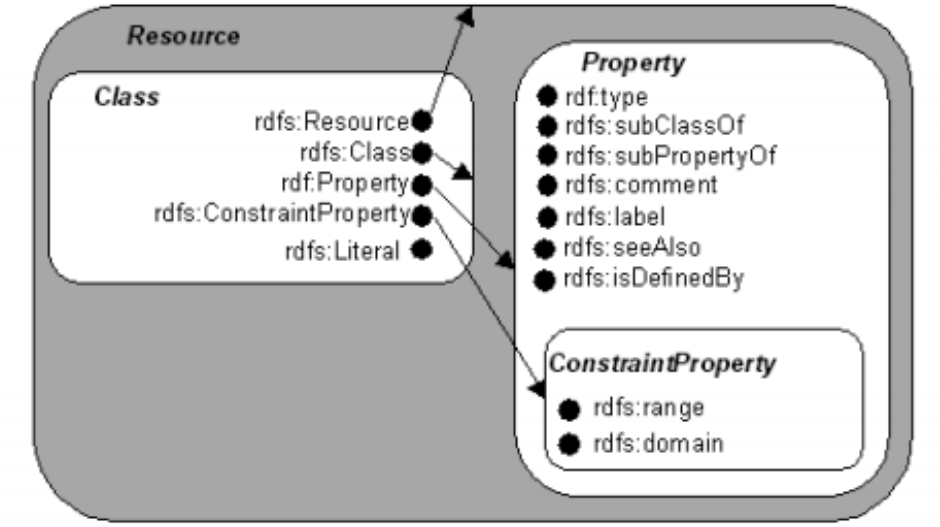

Abbildung 2.4: RDF-Schema Vokabular und seine Unterteilung [Ros12]

- **rdfs:Resource** das ist oberste Klasse der Hierarchie, in der alle Ressourcen beschrieben werden.
- **rdfs:Property** ähnlich wie **rdfs:Resource** aber für Eigenschaften.
- **rdfs:Class** identisch zu den Klassen in objektorientierten Sprachen und ist selbst eine Unterklasse von *rdfs:Resource*.

Kerneingenschaften des RDFS-Vokabulars sind folgende:

- **rdf:type** damit wird über eine Ressource ausgesagt, dass sie die Instanz einer Klasse ist.

- **rdfs:subClassOf** Konstrukt zur Bildung der Hierarchien (Untermenge/Obermenge-Relation).
- **rdfs:subPropertyOf** gleich wie bei *rdfs:subClassOf* aber diesmal für Property.
- **rdf:seeAlso** Referenz auf eine andere Ressource.
- **rdfs:isDefinedBy** abgleitet von *rdf:seeAlso*.

Für die Klassen und Eigenschaften in einem RDF-Schema können bestimmte Beschränkungen gelten. Dafür finden folgende Konzepte Anwendung:

- *rdfs:domain* gibt an, welche Klasse für diese Eigenschaft in Frage kommt
- *rdfs:range* bestimmt den Wertebereich einer Eigenschaft (z.B. kann die Eingenschaft *hasChild* so definiert werden, dass sie nur für Ressourcen von Typ *Person* gelten dürfen.)

Die Anwendung von rdfs:domain und rdfs:range auf RDF-Schema ist in Abbildung 2.5 ersichtlich.

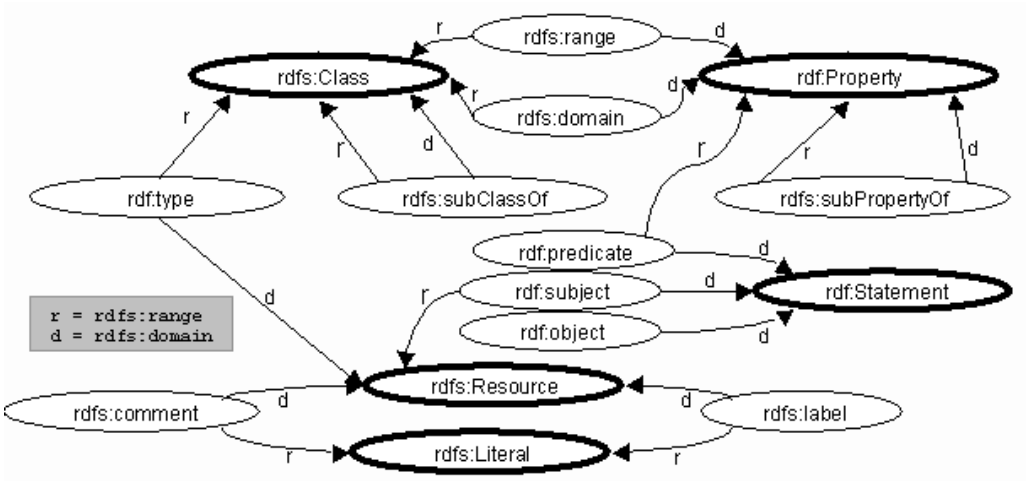

Abbildung 2.5: Konzepte von rdfs:domain und rdfs:range [Ros12]

2.1.3.4 Ontologie

Der Begriff Ontologie ist ursprünglich ein Fachgebiet der Philosophie und wird im Allgemeinen „*als Wissenschaft vom Sein bezeichnet bzw. von den Möglichkeiten und Bedingungen des Seienden*“ [Ott11]. Dieser Begriff wurde im Rahmen der Artificial Intelligence Forschung in die Informatik übernommen. In der Informatik wird durch den Einsatz der Ontologie möglich, domänenspezifische Dinge (Lebewesen, Eigenschaften, Sachverhalte usw.) und deren Beziehungen untereinander formal darzustellen. Ontologien definieren also Hierarchien und logische Zusammenhänge zwischen verschiedene Ressourcen und stellen somit ein Netzwerk von Informationen mit logischen Beziechungen dar. Sie sind die zentralen Bausteine des Semantic Web.

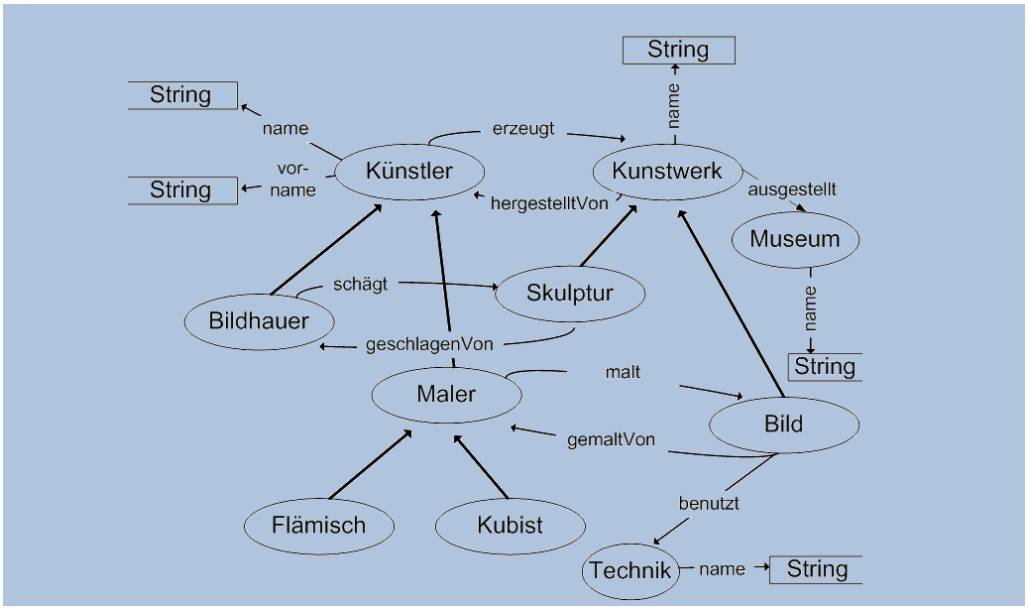

Abbildung 2.6: Beispiel für eine einfache Ontologie [Sös12]

Ontologien können zur Erfüllung unterschiedlicher Vorhaben eingesetzt werden. Einige Beispiele dafür sind unten angegeben:

- „*Für den Datenaustausch zwischen Programme*
- *Um die Vereinheitlichung und Übersetzung zwischen verschiedenen Wissensrepräsentationsformen zu ermöglichen*
- *Services zur Unterstützung von Wissensarbeitern zu entwickeln*
- *Theorien abzubilden*
- *Um die Semantik strukturierter und semi-strukturierter Informationen auszudrücken*
- *Um die Kommunikation zwischen den Menschen zu unterstützen und zu erleichtern*“ [PB12]

Es wird unter zwei Arten von Ontologien unterschieden:

- „*lightweight-Ontologien beinhalten Begriffe, Taxonomien, Beziehungen zwischen Begriffen, Eigenschaften zu Begriffe*
- *heavyweight-Ontologien beinhalten zusätzlich noch Axiome und Einschränkungen (Constraints)*“ [PGB12]

2.1.3.5 OWL

Web Ontology Language dient als eine formale Beschreibungssprache Ontologien zu beschreiben. Die Beschreibung soll so formuliert sein, dass sie durch andere Programme verstanden und die beschriebene Ontologie durch diese Programme verarbeitet werden kann. Obwohl OWL auf die RDF-Syntax basiert ist sie eine Erweiterung davon und hat somit ein höheres Ausdruckspotential als RDF.

OWL kann zur Definition von *Klassen*, *Relationen* und *Instanzen* verwendet werden. Begriffe einer bestimmten Domäne werden in den *Klassen* zu Gruppen zusammengefasst und diese Gruppen werden nach bestimmten Kriterien zugeordnet. Die klassenspezifische Eigenschaften werden durch *Relationen* definiert und *Instanzen* bedeuten Ausprägungen einer oder mehrerer Klassen [Ott11].

Die OWL-Spezifikation des *W3C* besteht aus [Ott11]:

- OWL Overview: gibt eine kurze Beschreibung.
- OWL Guide: Führt mit ein paar Beispielen vor, wie OWL zum Benutzen ist.
- OWL Reference: definiert ausführlich aller OWL-Elemente.
- OWL Semantics and Abstract Syntax: Eigentlicher Inhalt der OWL-Spezifikation.
- OWL Test Cases: enthält Muster zur Überprüfung.
- OWL Use Cases and Requirements: spezifiziert Szenarien, Anstrebungen und vorgegebene Anforderungen an eine „web ontology language“

OWL besteht aus drei Sprachversionen [Sch11a] :

- **OWL Lite** ist eine Subklasse der OWL. Mit OWL Lite können relativ simple Taxonomien und axiomatisierter Ontologien definiert werden.
- **OWL DL** beinhaltet den vollständigen Vokabular des OWL und unterstützt den Bereich der Description Logic (DL). Da OWL DL jedoch durch bestimmte Vorgaben eingeschränkt ist, ist sie eine Unterklasse von OWL Full.
- **OWL Full** OWL Full weist eine breite Ausdrucksmöglichkeit auf und ignoriert die in OWL DL vorgegebenen Einschränkungen.

2.1.3.6 Interoperabilität

Für die Realisierung von Semantic Web spielt die Herstellung der Interoperabilität eine enorm wichtige Rolle. Die Interoperabilität bezeichnet den Zustand, in dem heterogene Systeme effektiv zusammenarbeiten können. Die Kommunikation zwischen diesen Systemen kann organisationsintern und/oder organisationsübergreifend sein. Das Ziel von Interoperabilität ist eine Basis für das wechselseitige Aufeinandereinwirken von verteilter Datasets und Applikationen auf technischer, organisationaler und semantischer Ebene aufzubauen, ohne die selbständige Arbeit der Teilsysteme anzutasten [PB12].

> „The use of semantic technologies makes it possible to describe the logical nature and context of the information being exchanged, while allowing for maximum independence among communication parties. The results are greater transparency and more dynamic communication among information domains irrespective of business logic, processes and workflows.„ [PB12]

2.1.3.7 SPARQL

SPARQL [PS08] (Simple Protocol And RDF Query Language) ist eine Anfragesprache für das Semantic Web, mit der sich die RDF-Daten durchsuchen lassen (nach Subjekt, Prädikat und Objekt).

Um den Aufbau einer SPARQL-Query besser zu demonstrieren, werden im Folgenden einige Beispiele eingebracht. Es handelt sich dabei um Suchanfragen, die an DBpedia ausgerichtet sind.

Listing 2.1: Beispiel für eine SPARQL-Query

```
1  PREFIX dbo: <http://dbpedia.org/ontology/>
2  SELECT DISTINCT ?mountain,?el, ?nameofcountry WHERE {
3    ?place dbo:mountainRange ?mount.
4    ?mount rdfs:label ?mountain.
5    ?mount dbo:country ?country.
6    ?country rdfs:label ?nameofcountry.
7    ?place dbo:elevation ?el
8  }
```

Durch die vorangestellte Fragezeichen in einer Suchanfrage werden die Variable gekennzeichnet. Die Variablen, die als Ergebnis auftauchen sollen, stehen im SELECT-Block. Auf die erwünschten Ergebnisse wiederum gelangt man durch die Eingabe von Tripeln (Subjekt, Prädikat und Objekt) im WHERE-Block. Da eventuell Duplikate in einem Ergebnis auftauchen können, kann das Schlüsselwort DISTINCT eingesetzt werden um die Duplikate zu verhindern. Wenn der WHERE-Block aus mehreren Tripels besteht, müssen die vorherigen Tripels mit einem Punkt beendet werden.

Eine SPARQL-Query kann auch so aufgebaut werden, dass für Subjekt, Prädikat und Objekt Bedingungen spezifiziert sind. Ein Beispiel dafür sieht man in der folgenden Suchanfrage. In diesem Beispiel wird nach Bergen gesucht, die größer als 8000 m. sind. Das Schlüsselwort, das hier dafür eingesetzt wird, lautet FILTER.

Listing 2.2: SPARQL-Query: FILTER

```
1  PREFIX dbo: <http://dbpedia.org/ontology/>
2  SELECT DISTINCT ?mountain,?el, ?lab WHERE {
3    ?place dbo:mountainRange ?mount.
4    ?mount dbo:country ?country.
5    ?mount rdfs:label ?mountain.
6    ?country rdfs:label ?lab.
7    ?place dbo:elevation ?el
8    FILTER (?el>8000)
9  }
```

Die Prefixes in einer Suchanfrage stellen die Abkürzungen von Namespaces dar. Mit dem Gebrauch von Prefixes ist es nicht notwendig jedes Mal die vollständige URI anzugeben.

Listing 2.3: SPARQL-Query: PREFIX

```
1  PREFIX dbo: <http://dbpedia.org/ontology/>
2  ?place dbo:mountainRange ?mount.
```

...ist äquivalent zu:

Listing 2.4: SPARQL-Query: PREFIX

```
1  ?place <http://dbpedia.org/ontology/mountainRange> ?mount.
```

Um die Anzahl der Ergebnisse einer Suchanfrage zu beschränken wird das Schlüsselwort LIMIT eingesetzt. Es ist auch möglich die Ergebnisse nach bestimmten Bedingungen zu sortieren um einen besseren Überblick zu erhalten. Dafür wird das Schlüsselwort ORDER BY verwendet.

Listing 2.5: SPARQL-Query: LIMIT

```
1  PREFIX dbo: <http://dbpedia.org/ontology/>
2  PREFIX dbo2: <http://dbpedia.org/property/>
3  SELECT ?nam,?country,?total,?geo WHERE {
4          ?person foaf:name ?nam.
5          ?person dbo:birthPlace ?place.
6          ?place dbo:country ?country.
7          ?country dbo2:populationEstimate ?total.
8          ?country geo:wgs84_pos#lat ?geo
9  }
10 ORDER BY ?nam
11 LIMIT 100
```

2.2 Informationsvisualisierung

Die visuelle Repräsentation der Linked Data [BHIBL08] soll dazu dienen, die aus einem SPARQL-Endpoint gewonnenen heterogenen Daten in einer verständlichen und interaktiven Form darzustellen, das soll wiederum dabei helfen neue Erkenntnisse aus diesen Daten zu gewinnen. Es sind jedoch nicht alle Daten visuell darstellbar bzw. es macht bei manchen Daten keinen Sinn sie zu visualisieren. Abgesehen davon bieten sich unterschiedliche Ansätze für die visuelle Repräsentation der Daten an. Auch die Wiederverwendbarkeit der erstellten Diagramme ist ein wichtiger Ansatz, der beachtet werden sollte.

Aus den obigen Aussagen gewinnt man folgende Erkenntnisse, die bei der Visualisierung der Daten beachtet werden sollten:

- Es soll ein Mechanismus eingesetzt werden, um an den gewünschten Daten zu kommen. Entweder kann man das direkt spezifizieren, durch beispielsweise eine konkrete SPARQL-Query oder durch eine explizite Filterung der Ergebnisse am Frontend.
- Der Visualisierungsprozess soll definiert und die dafür in Frage kommende Technologie analysiert werden.

Die folgende Kapitel befassen sich mit diesen benannten Aspekten.

2.2.1 Designkonzept

Bevor eine Entscheidung bzgl. des Designs des Frameworks getroffen werden kann, das im Stande ist Linked Data zu visualisieren, muss in Erkenntnis gebracht werden, wo sich die Daten befinden und wie man zu diesen Daten gelangt. Wenn diese Frage beantwortet ist, muss man sich mit dem folgenden Schritt auseinander setzen: Nach dem Erhalt der Ergebnisse sollten Vorkehrungen getroffen werden, um schlussendlich aus diesen Ergebnissen die Daten zu extrahieren, die man visualisieren möchte. Diese Auswahl kann serverseitig stattfinden oder der Benutzer kann aktiv in dieses Auswahlverfahren eingebunden werden, in dem er z.B. auf der Benutzeroberfläche selbst entscheidet, welche Daten visualisiert werden sollen bzw. könnten.

Zurück zu der Frage bzgl. der Wahl eines SPARQL-Repositories. Es bieten sich viele Repositories als Linked Data Quellen an, wobei DBpedia einen zentralen Punkt unter ihnen darstellt. Als nächstes wird diese Repository etwas näher erläutert.

DBpedia Mit DBpedia sind die Wikipedia-Einträge per Semantische Suchanfragen ansprechbar. Bei dieser Suchanfrage werden die Wikipedia-Daten extrahiert, strukturiert und dem Endbenutzer zur Verfügung gestellt. Damit der Benutzer per SPARQL-Query Zugriff auf diese Daten findet, kann er unter `http://dbpedia.org/sparql` zur Verfügung gestelltes Tool verwendet. Mit Hilfe dieses Tools können SPARQL-Queries abgesendet und die Ergebnisse gleich (z.B in JSON) betrachtet werden.

Nachdem eine Auswahl bzgl. der Repository getroffen wurde, muss auf folgende Fragen eingegangen werden:

- Wie können die Daten aus diesem SPARQL-Endpoint (z.B. DBpedia) erfasst werden und
- Wie diese Daten visualisiert und die Visualisierungen wiederverwendbar gemacht werden.

2.2.1.1 Erfassung der Daten

Die Daten bzw. die Einträge aus einem Triple-Repository werden dem Server des Visualisierungsframework über einem SPARQL-Interface zur Verfügung gestellt, falls sie über einem Third Party zugänglich gemacht werden sollen. Beim DBpedia geschieht das beispielsweise mit Hilfe des SPARQL-Wrappers [Kap11].

Listing 2.6: SPARQL-Wrapper

```
1  from SPARQLWrapper import JSON
2  from SPARQLWrapper import SPARQLWrapper
3
4  class sparqlUtil():
5    def __init__(self):
6      self.repository="http://dbpedia.org/sparql"
7
8      def query(self, query):
9        try:
```

```
10            sparql = SPARQLWrapper(self.repository)
11            sparql.setReturnFormat(JSON)
12            sparql.setQuery(query)
13            dataResults = sparql.query().convert()
14            return self.clearResult(dataResults)
15          except Exception, err:
16            erStr = '%s\n' % str(err)
17            raise Exception("Backend problem - sparqlUtil: %s"%erStr)
```

Solche SPARQL Endpoint Interfaces sind typischerweise als Wrapper-Facade Pattern (siehe Anschnitt 3.2.6) realisiert und beinhalten unter anderem die Mechanismen zur Formatierung der Nachrichten bzw. Daten in verschiedene Protokolle (JSON, XML usw). Die Kommunikation zwischen dem Server und der SPARQL-Interface basiert auf den REST Architekturstil. Auf den REST Architekturstil und JSON wird im Kapitel 2.2.3 näher eingegangen.

Aus des Sicht des Clients ist es notwendig, die entsprechende Kommunikation mit dem Server herzustellen, um auf die erfassten Daten zu kommen. Die serviceorientierte Kommunikation mit dem Server ist eine Möglichkeit, die wiederum auf dem bereits erwähnten REST Architekturstil basiert könnte.

2.2.1.2 Visualisierung

Die Visualisierung von Linked Data ist ein sehr wichtiger Forschungsbereich, in dem sehr komplexe Daten-Netze visuell überschaubar gemacht werden. Dadurch werden sie für wenig erfahrene Benutzer einfacher zu handhaben. Es existiere mittlerweile einige Forschungsarbeiten zu diesem Thema. Die Autoren in [DR11] zeigen eine aktuelle Zusammenfassung von Visualisierungsansätzen und legen einige Anforderungen an solche Werkzeuge fest. Diese Anforderungen beziehen sich hauptsächlich auf das Navigieren durch Daten, auf ihre Repräsentation (als vernetzte, hierarchische, usw.) sowie auf deren Filterung und Wiederverwendung. Anschließend geben die Autoren eine übersichtliche Klassifizierung aller Ansätze an, die auf vordefinierten Design-Kriterien basieren.

Das LD4UViz Framework bezieht sich hauptsächlich auf heterogene Daten und dadurch auf Vorlagen-basierte Visualisierungsmethoden.

Visualisierung heterogener Daten Im Bereich der Visualisierung von heterogenen Daten haben die Autoren in dieser wissenschaftlichen Arbeit [CDC+07] eine Lösung vorgeschlagen, die die Visualisierung der Daten basierend auf einer formalen Spezifikation durchführt. So wird z.B. für jeden Diagrammtyp eigene Spezifikation in Form `<DATENTYP, NAME>` Einträgen definiert. Die Formalität wird anhand von Datentypen der RDF-Instanzen garantiert (die Überprüfung der Spezifikation). Die Lösung liegt nah an den Erwartungen des LD4UViz Frameworks bzw. Anforderungen im Abschnitt 1.2, hat aber einen Nachteil: der Code für den entsprechenden Diagramm muss trotzdem manuell zur Verfügung gestellt werden. Für unsere Zwecke sollen aber fertig generierten Diagramme verwendbar sein.

Aus umfangreichen Literatur-Recherche in [DR11] werden im Folgenden einige interessante und für die Arbeit relevante Ansätze zur Visualisierung herausgefiltert und kurz

beschrieben.

LESS Framework Eine aktuelle Arbeit bzgl. wiederverwendbarer Lösungen schlagen die Autoren im [ADD10] vor. Hier wird ein Vorlagen-basiertes Framework verwendet, um verschiedene Repräsentationen von heterogenen Daten zu ermöglichen. Die Abbildung 2.7 zeigt das Prinzip dieses Frameworks. Die Vorlage und die Ergebnisse vom SPARQL-Query

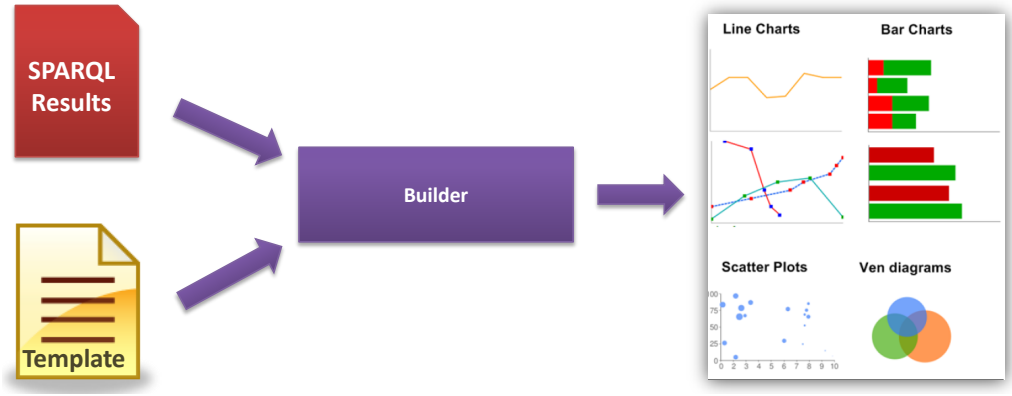

Abbildung 2.7: Prinzip vom LESS Framework

werden verwendet, um das endgültige Dokument zu generieren. Es kann sich dabei um beliebige Repräsentation handeln. Der Generator im Framework interpretiert eine einfache Vorlagen-Skript (Smarty Skript), dass die Fusion zwischen Syntax und der Properties erlaubt (Spezifikation der Statements), und zwar in dieser Form:

Listing 2.7: Spezifikation der Statements

```
1   {{PROPERTY_NAME}}
```

Der Wert aus den SPARQL-Ergebnissen, der dieser Property entspricht, wird automatisch durch den Builder an der Stelle in der Vorlage ersetzt.

Auf der Tabelle 2.1 sind Syntax aufgelistet, die in diesem Framework in Anwendung kommen. Eine LESS-Vorlage (Template) zur Darstellung der Gruppenprofile, gewonnen von FOAF-Profiles, kann aus dem unteren Code entnommen werden.

Listing 2.8: LESS-Vorlage [ADD10]

```
1  <div id ="␣foaf␣-card␣-␣block␣">
2   {if {{ foaf : depiction }} !== ''}
3    img class ='photo␣'
4    src ="␣{{␣foaf␣:␣depiction␣}}" />
5   {/ if}
6  <div id ="␣name␣">{{ foaf : name }} </div >
7   ...
8  {if {{ foaf : phone }} !== ''}
9   <div id ="␣tel">{{ foaf : phone }} </div >
10 {/ if}
11 <div id ="␣email␣">
12   <a href ="␣{{␣foaf␣:␣mbox␣}}">
```

```
13     <img src =" email . png " />
14     {if $language === 'en '}
15       email { else } E- Mail
16     {/ if}
17    </a>
18   </div >
19 </div >
```

Syntax	Beschreibung	Beispiel
`{{property}}`	Referenziert den Wert der Property	`{{foaf:name}}`
`{{p@lang}}`	Auswahl einer Sprache: „lang“ von property „p“	`{{rdfs:comment@en}}`
`{{p^^dtype}}`	Referenziert den Datentyp: „dtype“ von property „p“	`{{dbp:birth^^xsd:date}}`
`{{p1->p2}}`	Referenziert eine Eigenschaft über eine andere: „p2“ referenziert über „p1“	`{{foaf:currentProject-> rdfs:label}}`
`{template id="'id"' uri="'uri"' instances="'"' sortBy="'"'}`	Verarbeitet die Vorlage mit der ID „id“ und Ressource „uri“	`{template id="'5"' uri="'{$var}"'}`
`{template id="'id"' sparql="'query"' endpoint="'srv"'}`	Verarbeitet die Vorlage mit der ID „id“ und SPARQL-Query „query“ vom Endpoint „srv“	`{template id="'3"' sparql="'SELECT * WHERE {}"' endpoint= "'http://dbpedia.org"'}`

Tabelle 2.1: Syntax für Smarty Skript [ADD10]

Das Framework hat auch nicht-funktionalen Features, so wie Repository- und Benutzer-Management und Verwaltung von Vorlagen.

Der Ansatz bietet eine sehr hohe Grad der Automatisierung und eine generische Lösung zur Repräsentierung der heterogenen Daten. Jedoch fehlt auch bei diesem Ansatz die Möglichkeit, die Diagramme wiederverwenden zu können. Die Wiederverwendung auf Datenbasis ist aber möglich. Es können fertig gestellte Vorlagen im Repository gespeichert und jederzeit wiederverwendet werden.

Microsoft Excel Table Wizzard Das was bei den Ansätzen zur Visualisierung von heterogenen Daten in Linked Cloud Data fehlt, ist die Möglichkeit, die Visualisierungen wiederverwenden zu können. Ein sehr pragmatischer Ansatz bietet das Microsoft Excel Table Wizzard [Mic12], und zwar seit langer Zeit. Die Idee besteht darin, den Benutzer anhand seiner vorgegebener Daten iterativ und intuitiv durch den Prozess der Visualisierung seiner Daten zu führen. Diese Visualisierung basiert hauptsächlich auf Mapping zwischen den Datentypen beider Seiten: Daten vom Benutzer und Daten des Diagramms. Auf diese Art kann ein sehr kleines Subset von Diagrammen für eine große Menge der heterogenen Daten verwendet werden.

DBpedia Mobile DBpedia Mobile ist ein Web-basiertes Framework, spezialisiert für Mobile Geräte mit Map-Funktionalität [BB09]. Es bietet den Benutzern die Möglichkeit über GPS ihren Ort zu ermitteln und sie auf der Karte anzuzeigen. Neben der Map-Funktionalität bietet das Framework auch die Option, relevante in der Nähe stehende Objekte zu identifizieren und sie mit ihren Metadaten anzuzeigen. Jedes der Objekte entspricht einer Ressource in DBpedia Datenbank. Somit kann der Benutzer interaktiv jede Ressource direkt abfragen und somit seine Suche verfeinern, da auch diese Funktionalität im Framework vorhanden ist. Die Abbildung 2.8 zeigt die Benutzeroberfläche des Frameworks und die Ortung im Graz (aus dem Campus TU-Inffeldgasse). Für die Verfeinerung

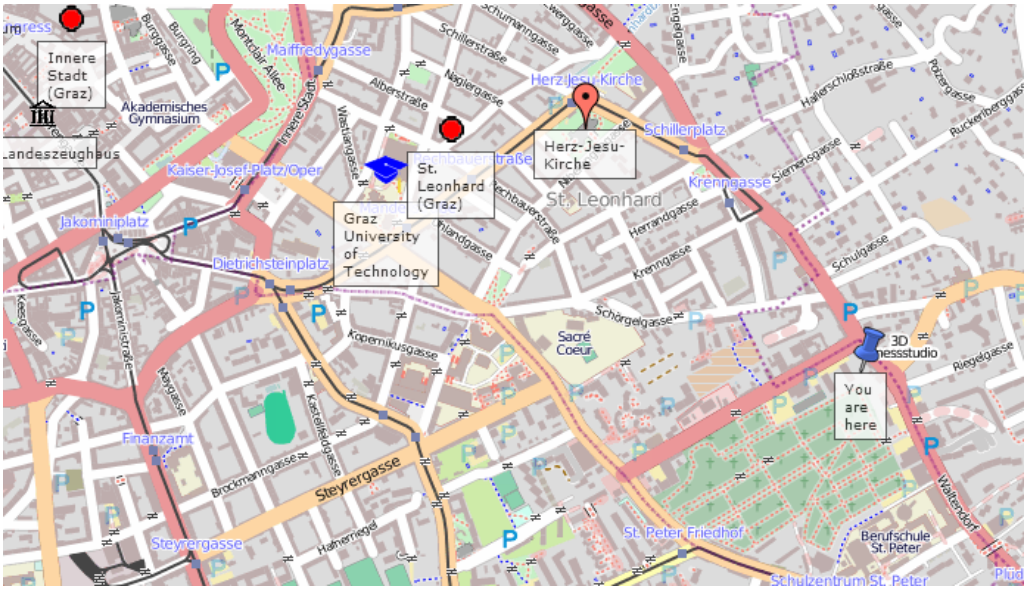

Abbildung 2.8: Ortung mit dem DBpedia Mobile Framework

der Suche wird das MARBLES Framework eingesetzt[2]. Dieses Framework ermöglicht für ein angegebenes URI die relevanten Informationen der dahinder stehenden Ressource zu extrahieren (z.B. Bild, zusammenfassende Metadaten, usw.). Die Aktivierung dieses Frameworks aus Sicht vom DBpedia Mobile erfolgt mittels Klicken auf entsprechende Ressource in der Karte (z.B. auf Graz University of Technology in der Abbildung 2.8).

[2] http://www5.wiwiss.fu-berlin.de/marbles

IsaViz Im Unterschied zum DBpedia Mobile und LESS, wo die Visualisierung situationsbasiert ist, stellt IsaViz ein Ontologie-Modell zur Verfügung, mit dem die Datenbeziehungen visualisiert werden [Isa12]. Das Tool bietet u.a. die Möglichkeit, neu erstellten Ontologien nach RDF/XML zu exportieren, sowie bestehende zu importieren. Das primäre Ziel dieses Frameworks ist die Vereinfachung der Bildung von Datenbeziehungen mittels GSS - Graph Stylesheets Visualisierungstechnologe (da die XML-Dateien für den Menschen auch für kleinere Ontologien komplex sein können).

OpenLink Data Explorer OpenLinkData Explorer[3] ist eine Web-basierte Suchmaschine, die für einen gegebenen URI oder für ein gegebenes Literal DBpedia durchsucht und die Ergebnisse in mehreren Aspekten darstellt. Durch diese Aspekte können die Ergebnisse in verschiedenen Sichten analysiert und die Suche weiter verfeinert werden. Die Abbildung 2.9 zeigt, wie das Framework die Suche optimiert (Demo zugänglich auf: `http://demo.openlinksw.com/rdfbrowser2/`). Im oberen Teil der Abbildung 2.9

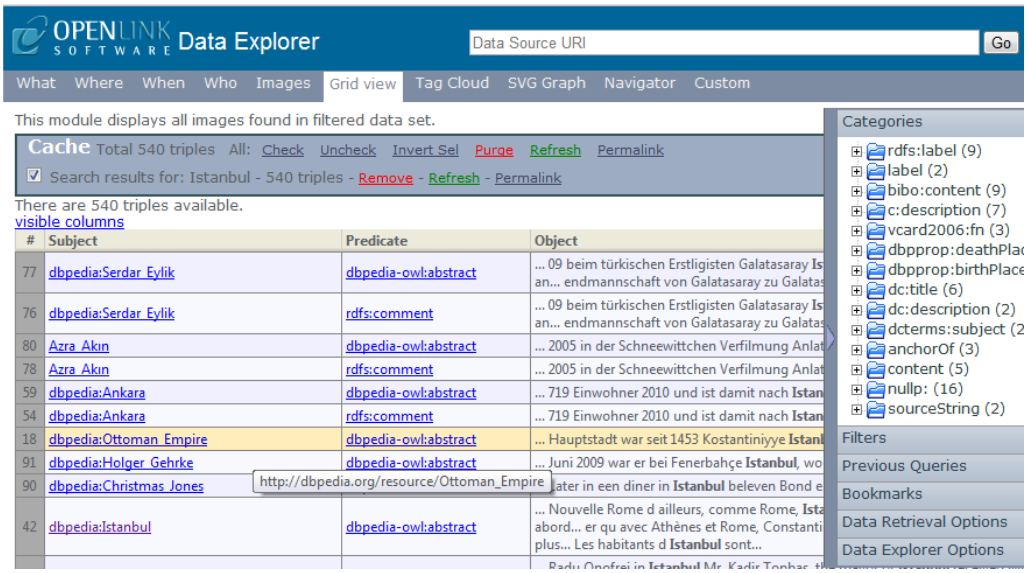

Abbildung 2.9: OpenLink Data Explorer-Suche nach dem Literal „Istanbul“

sind Aspekten dargestellt. Jeder dieser Aspekte beschreibt die Daten in einer bestimmten Vorlage (z.B. im Where-Aspekt wird Google Maps eingesetzt). Für die Verfeinerung der Ergebnisse können auch eingebaute Filters verwendet werden (rechter Sidebar in der Abbildung). Das Framework bietet u.a. Sessions-Management, womit die Benutzer-Einstellungen gespeichert/wiederverwendet werden können.

Tabulator Ähnlich wie OpenLink Data Explorer bietet das Framework Tabulator die Möglichkeit, für gegebenes URI (z.B. von der RDF-Datei) die Ontologie zu untersuchen (2.1.3.4). Die Suche ermöglicht auch die Referenzierung auf verwendete Ressourcen und somit das Navigieren in benachbarte Dokumente. Das Framework bietet, ähnlich wie beim

[3] `http://ode.openlinksw.com/`

OpenLink Data Explorer, Tabs (Table, Map, Calendar, SPARQL, Debug), die durch verschiedene Sichten die Spezifikation vom SPARQL ermöglichen. In der aktuellen Version

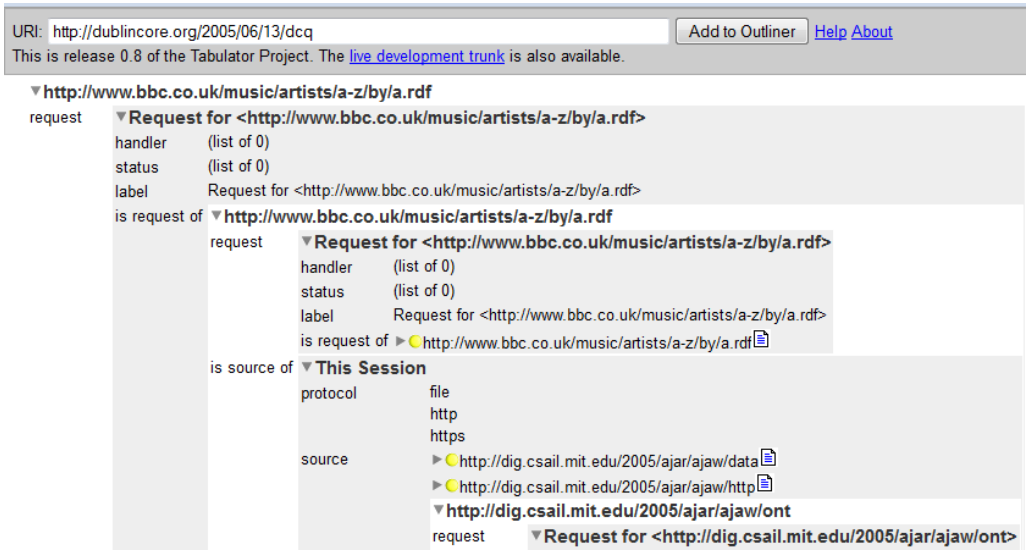

Abbildung 2.10: Navigieren durch Linked Data mit dem Tabulator (Online-Version)

wird das Framework als JavasScript Bibliothek (siehe Abbildung 2.10), oder auch als Firefox-Plugin (als Mozzila XPI) geliefert.

RDF-Gravity RDF-Gravity ist ein vom *Salzburg Research* entwickeltes Framework zur Visualisierung von RDF (2.1.3.2) und OWL Ontologien (2.1.3.5). Es bietet ein Java-basiertes Frontend mit Vielzahl von Filter, die das Navigieren durch komplexe Ontologien ermöglichen (siehe Abbildung 2.11). Es ist jedoch nicht möglich neue Objekte bzw. Klassen zu definieren oder einen bestehenden Graph in andere Formate zu exportieren. Neben Filter-Funktionen sind auch die Suche und SPARQL-Query Editor im Framework integriert.

Vispedia Vispedia ist ein von der Stanford Universität[4] entwickeltes Framework zur interaktiver Visualisierung von Linked Data aus Wikipedia [CWT+08]. Es wurde als Browser-Plugin geliefert und kann durch den Benutzer jederzeit aktiviert werden, um bestimmte Einträge aus Wikipedia auf der Stelle zu visualisieren. Die Abbildung 2.12 zeigt den kompletten Prozessablauf der Visualisierung. Im ersten Schritt findet der Benutzer interessante Daten, die er analysieren möchte (Tabelle) und markiert sie mit dem Vispedia bookmarklet. Somit werden die Daten vom Framework erfasst und der Benutzer kann anschließend den Typ der Visualisierung auswählen (den Diagramm). Dafür stehen einige built-in Diagramme zur Verfügung. Im nächsten Schritt wird das Mapping durchgeführt: der Benutzer mappt die Daten an Achsen des Diagramms (bei dem, in der Abbildung 2.12 dargestellten Fall sind sie Date, Caption und Image). Abschließend wird das Diagramm generiert.

[4] http://www.stanford.edu/

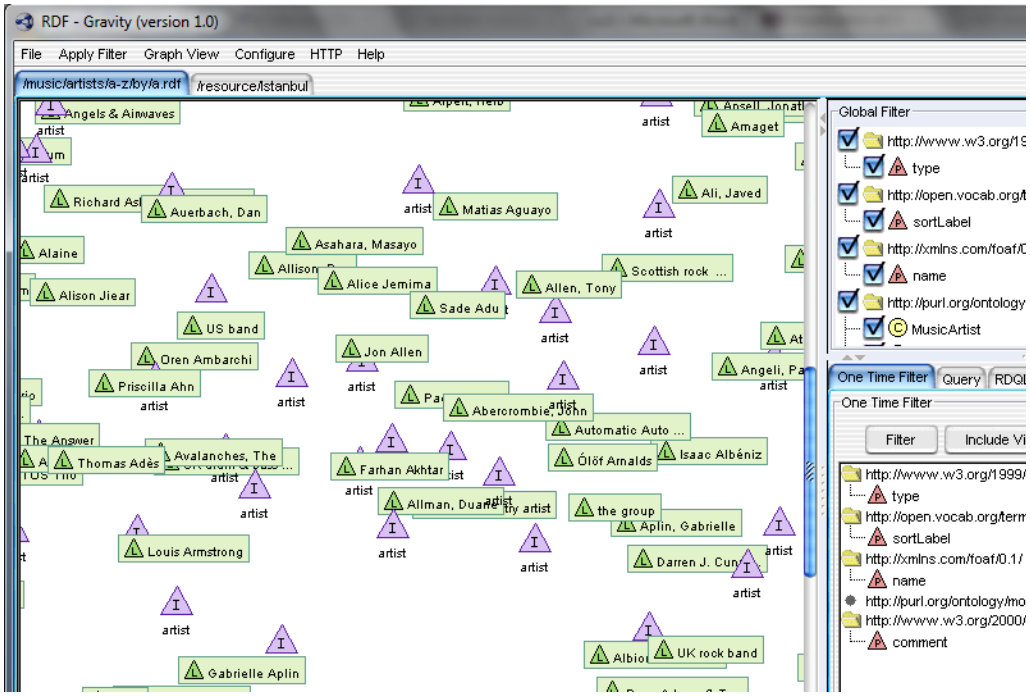

Abbildung 2.11: Visualisierung der Ontologie mit RDF-Gravity

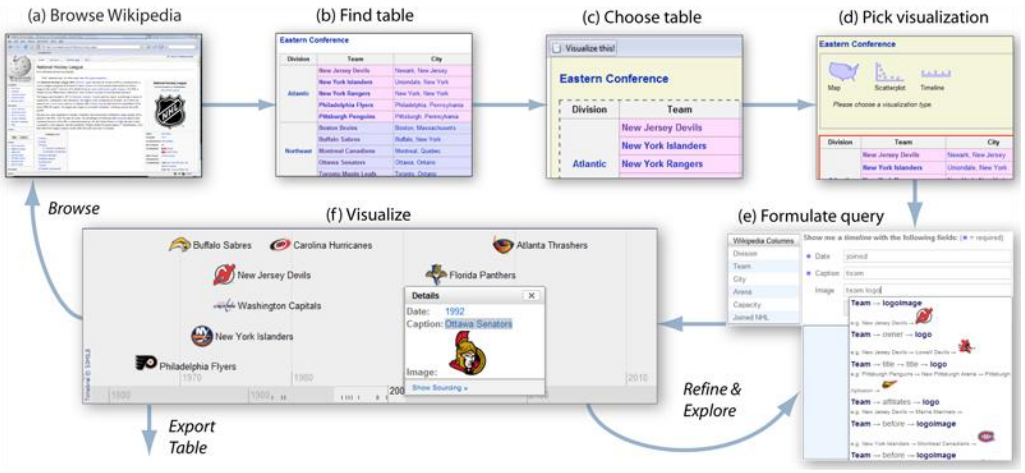

Abbildung 2.12: Prozess zur Bildung der Visualisierung bei Vispedia, [CWT+08]

Das Wesentliche bei diesem Framework ist die Flexibilität. Der Benutzer wird einfach und intuitiv durch den Prozess geführt und die Diagramme werden Online erstellt. Die Eigenschaft, das Mapping dem Benutzer zu überlassen, gibt dem Framework die Möglichkeit heterogene Daten zu visualisieren.

RelFinder Bei dem Framework werden die Beziehungen von eingegebenen Ressourcen und dem Benutzer graphisch dargestellt [HLS10]. Die Abbildung 2.13 zeigt wie das Frame-

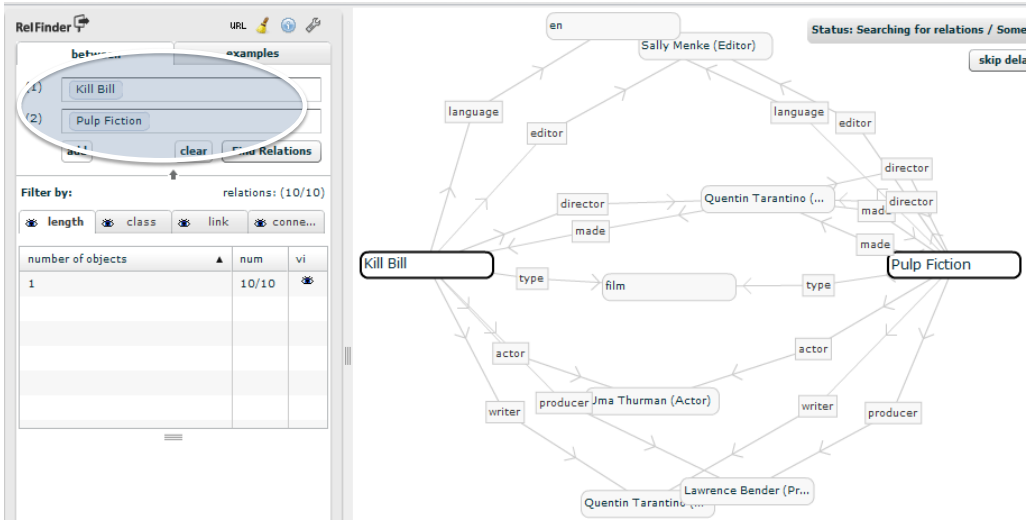

Abbildung 2.13: Bespiel für eine Beziechungsanalyse beim RelFinder

work die Beziehung zwischen den Ressourcen „Kill Bill“ und „Pulp Fiction“ herausfindet und visualisiert. Das Framework analysiert die Beziehungen im engen Bereich (ROI-region of interest). Für die Analyse können dabei unterschiedliche SPARQL-Endpoints spezifiziert werden (DBpedia, Linked Movie Database). Außerdem bietet das Framework die Suche nach Literalen in seiner Benutzeroberfläche.

Graphische Darstellung der heterogenen Daten Für die Ergebnisse einer SPARQL-Query gibt es unterschiedliche Möglichkeit zu einer benutzerfreundlichen Darstellung. Im Folgenden werden einige der dafür in Frage kommende Diagramme vorgestellt.

- Säulendiagramm: Ein Säulendiagramm stellt die relative bzw. die absolute Häufigkeitsverteilung einer Variable dar. Die Länge der Säulen stehen dabei senkrecht zu der Häufigkeit.
- Liniendiagramm: Ein Liniendiagramm wird eingesetzt um darzustellen, in welchen Beziehung zwei oder drei Variablen zueinander stehen.
- Kreis-Diagramm: Ein Kreisdiagramm stellt Daten als Segmente eines Kreises dar. Jedes Segment repräsentiert dabei den prozentualen Anteil der Gesamtmenge.
- World Map (Landkarte): Diese Visualisierungsform stellt einen Kontinent, ein Land oder eine Region farbig dar. Um den Wert, der mit diesem Ort verbunden ist, darzustellen werden unteschiedliche Farben bzw. Farbintensitäten für diesen Ort verwendet.
- Tabellendiagramm : Mit einer Tabelle können Einträge sortiert, geordnet und in Zeilen und Spalten gegliedert werden.

- Word/Tag Clouds: Bietet eine graphische Darstellung der Schlagwörter in einem Text, wobei diese Anzeige auf die Häufigkeit dieser Wörter im Text basiert.

- Timeline: Auf Zeit basierte Angaben (Geburtstag, Todestag usw.) werden auf einer interaktive Zeitleiste angeordnet dargestellt.

- usw.

Für die Realisierung der oben erwähnten Diagramme existieren unterschiedlichen Technologien. Einige davon werden im Abschnitt 2.2.2 vorgestellt.

Résumé Zur Visualisierung heterogener Daten stellt die Vorlagen-basierte Methode eine pragmatische Lösung dar, jedoch keine Strategie zur systematischen Wiederverwendung von Diagrammen. Daher ist eine der Herausforderungen für diese Arbeit, die Synergien zwischen Vorlagen-basierte Methoden und Wiederverwendung von Diagrammen (wie im Excel) zu untersuchen und idealerweise ins Framework zu integrieren. Eine erfolgreiche Integration würde eine neue Methode im Themenbereich Visualisierung heterogener Daten bringen.

2.2.2 Technologien für die Visualisierung

Für die graphische Darstellung der Daten, die aus den Ergebnisse einer SPARQL-Query extrahiert worden sind, kommen unterschiedliche Methoden in Frage. Dieses Kapitel gibt eine kurze Einführung in ein paar der ausgesuchten Technologien. Diese sind [Mut11]:

- SIMILE Widgets

- Raphaël

- Protovis

- Google Chart Tools

2.2.2.1 SIMILE Widgets

SIMILE Widgets ist ein Open-Source JavaScript-basiertes API, entwickelt von MIT[5] (Massachusetts Institute of Technology) im Rahmen eines Projekts names SIMILE (Semantic Interoperability of Metadata and Information in unLike Environments), das auf dem semantischen Web basiert und dazu dient statistische Daten einfach zu visualisieren.

Für die Visualisierung werden vier unterschiedliche Gruppen von Visualisierungen, sogenannte Widgets, angeboten: zwei Arten von hoch interaktive Zeitdiagramme (z.B. Roadmap, xy-Diagramme, siehe Abbildung 2.14), interaktive Landkarten und die interaktive Visualisierung von Bilder in iTunes Cover Flow Form.

[5] `http://mit.edu/`

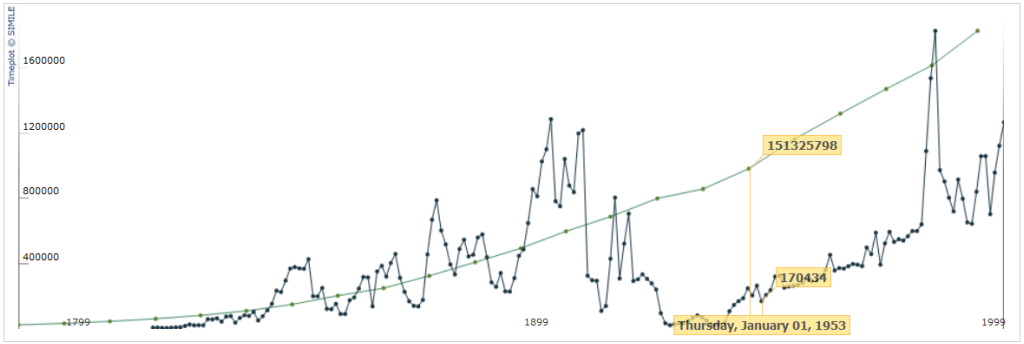

Abbildung 2.14: Beispiel eines Zeitdiagrams erstellt mit SIMILE API, [Hom11]

2.2.2.2 Raphaël

Raphaël ist ein hoch flexibles JavaScript-basiertes API zur Erzeugung von Vektorgrafiken. Dazu verwendet Raphaël Vektorgrafikstandard SVG, der von vielen modernen Browsern beherrscht wird. Für Internet Explorer wird Microsofts VML (Vector Markup Language) unterstützt [Gol11].

2.2.2.3 Protovis

Protovis ist auch ein OpenSource JavaScript-basiertes API zur Datenvisualisierung, das an der Universität Standford[6] (Standfort Visualization Group) entwickelt wurde. Im Unterschied zum SIMILE Widgets, bietet dieses API die Möglichkeit, Balken- und Punktediagramme zu erstellen (siehe Abbildung 2.15). Es sind auch Zeitdiagramme mit Protovis möglich, jedoch mit weniger Interaktivität als mit Simile.

2.2.2.4 Google Chart Tools

Auch hier handelt es sich um ein JavaScript-basiertes API zur Darstellung von Diagrammen, zur Verfügung gestellt von Google. Die statistischen Daten werden am Back-End (Google) ausgewertet und die entsprechenden Diagramme automatisch erstellt. Das Ergebnis der Auswertung ist also ein SVG bzw. VML, das am Front-End als Diagramm angezeigt wird. Einige der von Google angebotenen Charts sind in Abbildung 2.16 ersichtlich.

2.2.3 Technologien für die Kommunikation

Bei der Realisierung des Datenaustausches zwischen Client und Server kommen Technologien, wie REST und JSON, zum Einsatz. In diesem Kapitel wird näher darauf eingegangen.

2.2.3.1 REST

REST steht für Representative State Transfer und ist ein Architekturstil, der auf Prinzipien basiert, die vorschreiben, wie Ressourcen im World Wide Web definiert und eingesetzt

[6] http://vis.stanford.edu/

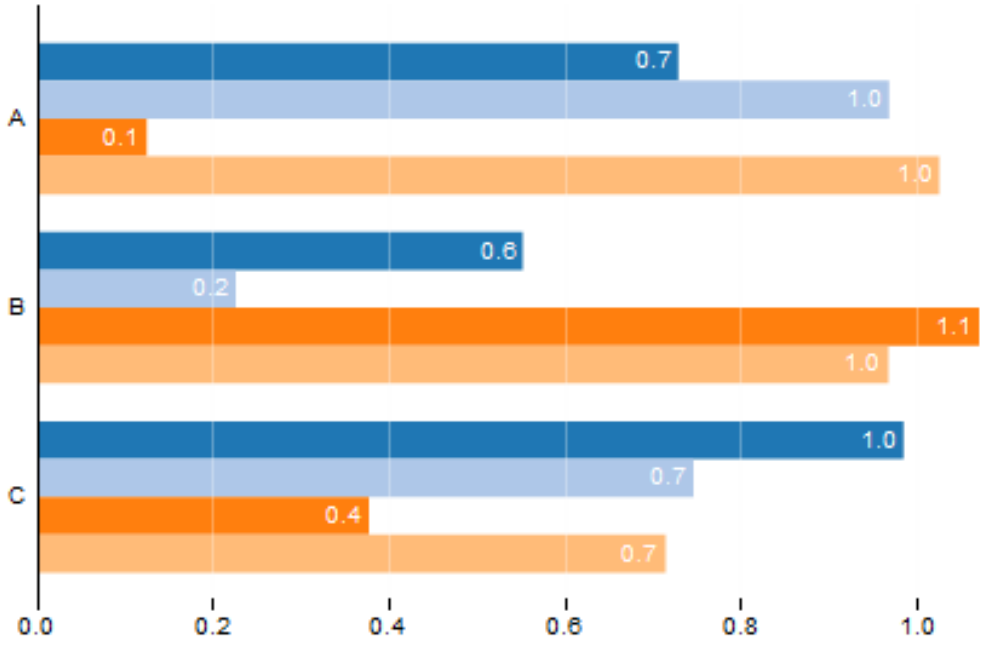

Abbildung 2.15: Beispiel eines Bar Chart erstellt mit Protovis API, [svg11]

werden sollen. Eine Ressource kann eine Webseite, eine Video-Datei, ein Bild, eine Audio-Datei usw. sein. Die Ressourcen sind durch ihre URIs eindeutig idendifizierbar und werden durch das Versenden der Nachrichten zwischen einem Client und einer Webanwendung auf dem *HTTP* angesprochen.

Jede REST Ressource verfügt über die bekannten *HTTP* Methoden *GET*, *POST*, *PUT* und *DELETE* eine generische Schnittstelle, wodurch die Kommunikation zwischen dem Client und dem Server zustanden kommen kann.

GET: Verlangt nach einer Ressource.

POST: Wenn u.a. einer Ressource ein Parameter hinzugefügt werden soll oder komplexe Datenstrukturen transportiert werden sollen, dann wird die Methode *POST* verwendet. Mit *POST* ist auch möglich, Dateien in Segmenten zu versenden.

PUT: Mit der Methode *PUT* können Ressourcen erzeugt oder der Inhalt bestehender Ressourcen verändert werden.

DELETE: Löscht die Ressource.

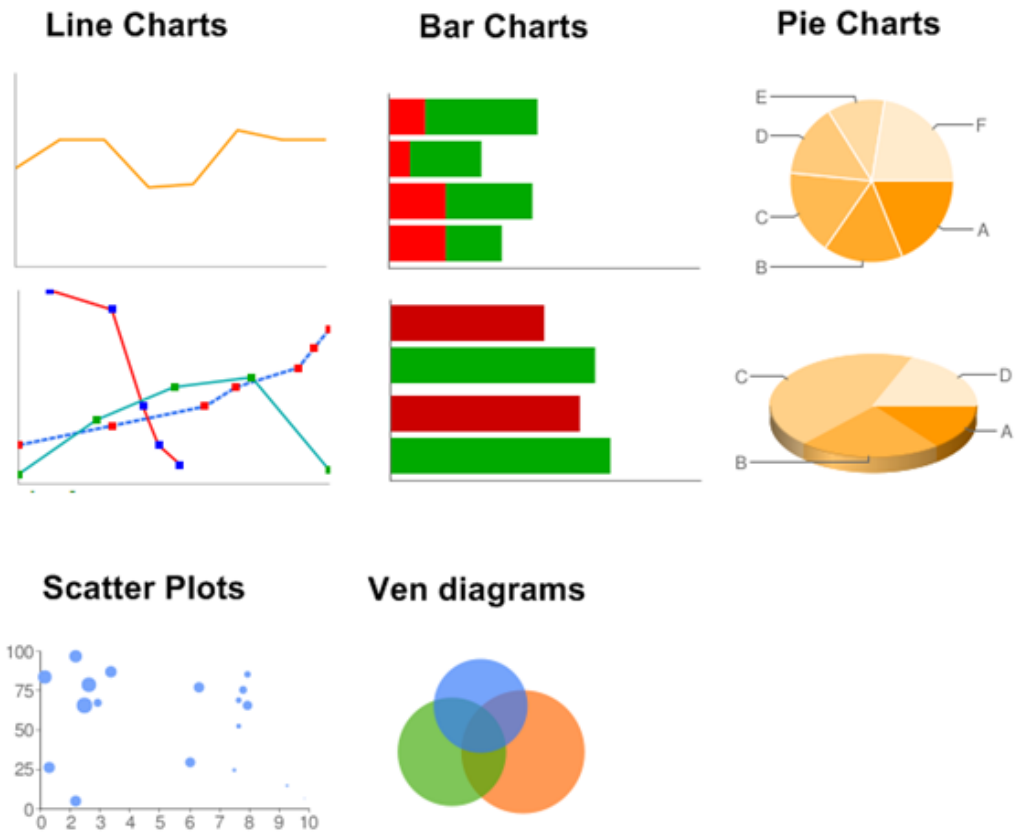

Abbildung 2.16: Google Charts [Goo11]

2.2.3.2 JSON

JSON (JavaScript Object Notation) ist ein Format, basiert auf *ECMA-262-Spezifikation*, das zum Austausch der Daten zwischen unterschiedlichen Applikationen generiert ist. Dabei können die Daten als Objekte oder als Array verpackt und transportiert werden. Weitere Datentypen, die JSON unterstützt sind:

- Nullwert
- Boolescher Ausdruck
- Number
- String

Da mittlerweile in jeder Programmiersprache (*Java*, *JavaScript*, *C*, *C++*, *Python* usw.) ein JSON Parser integriert ist, kann JSON in jeder Programmiersprache eingesetzt werden. Sie ist also sprach-und plattformunabhängig.

Es gibt unterschiedliche Anlässe um JSON zu verwenden. Der Datenaustausch zwischen Client und Server z.B. ist einer der Einsatzgebiete von JSON. Die Kommunikation baut

dabei auf *AJAX* (siehe 2.2.3.3) oder *WebSockets*. Bei geringem Speicherplatz oder bei geringer CPU-Leistung ist JSON eine Alternative zum *XML*.

2.2.3.3 AJAX

AJAX ist die Abkürzung für Asynchronous JavaScript and XML und ist eine Methode, die den asynchronen Datentransfer zwischen dem Client (Browser) und Server ermöglicht. Dadurch wird z.B. möglich Daten eine *HTML*-Seite zu aktualisieren, während sie angezeigt ist, ohne die Seite noch einmal starten zu müssen. Die Webtechnologien, auf die AJAX basiert sind:

- HTML
- XSLT
- CSS
- JSON
- DOM
- JavaScript
- REST
- On-Demand Javascript
- reST- ähnliche Verfahren
- SOAP

2.3 Systematische Wiederverwendung

Im Zuge der Literaturrecherche wurde auch auf die Wiederverwendung der Software Artefakten eingegangen, da die Visualisierung selbst auf Wiederverwendung von bestehenden Diagrammen basiert. State-of-Art Techniken im Bereich der Wiederverwendung stellen Software Produktlinien (SPL) dar (siehe Abbildung 2.17). Sie beschreiben eine Methode zur systematischen Wiederverwendung von Software-Artefakten [PBvdL05]. Systematisch in diesem Kontext bedeutet „geplant“ und bereits in den Entwicklungsprozess integriert. Dadurch unterscheidet sich dieser Ansatz von beispielsweise Konfigurations-Management oder der ungeplanten Software-Anpassung. Die Methode stammt aus der Automobilindustrie, wo die Produktion deutlich automatisiert wurde, in dem die Fahrzeuge (Produkte) viele gemeinsame Teile erbten. In Software ist diese Methode erst seit 10 Jahren im Einsatz und verfolgt die gleiche Strategie. Was die Erwartungen dieser Methode sind, zeigt die Abbildung 2.17.

Die rot gezeichnete Linie zeigt die Kosten als Funktion von zu entwickelnde Systemen. Wie zu sehen ist, ist diese Steigung deutlich größer als im Fall von Produktlinien. Der Grund ist, dass bei den Produktlinien ein Großteil von der Software automatisch generiert

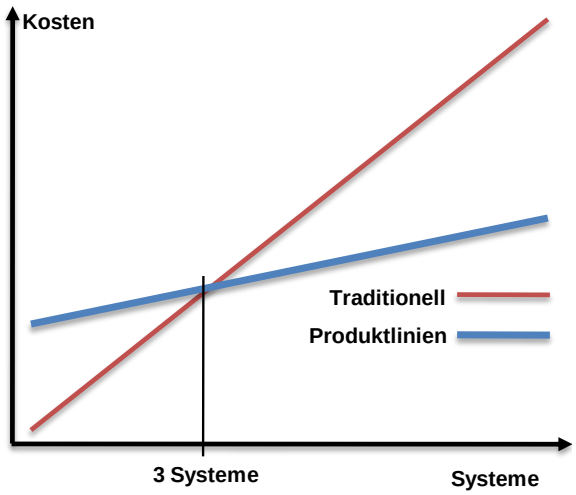

Abbildung 2.17: Motivation für Produktlinienengineering: Entwicklungskosten [PBvdL05]

werden kann, aber die Kosten am Anfang etwas höher liegen. Aus Erfahrung ist bekannt, dass sich die Methode erst nach drei Systemen rentabel macht. Es ist daher vor der Entwicklungsseite aus zu überlegen, ob sich dieser Ansatz auszahlt oder nicht.

Engineeringsprozess

Der Ansatz teilt den Entwicklungsprozess der Software in Domänenengineering und Anwendungsengineering.

Domänenengineering entspricht einem Softwareentwicklungs-Modell (wie z.B. Wasserfall), erweitert durch die zusätzlichen Softwareartefakte, wodurch die Berücksichtigung mehrerer Varianten von derselben Software stattfinden kann. Das bedeutet, Analysephase im Wasserfall beinhaltet nicht nur ein Software-Model, sondern mehrere. Das gleiche passiert mit dem Design der Implementierung usw. Es besteht dadurch natürlich das Problem der Konsistenzerhaltung, aber das ist dem Entwickler der Produktlinie überlassen.

Bei einem anderen Engineeringsprozess geht es darum, aus der in Domänenengineering konstruierten Software eine konkrete Variante abzuleiten. Das heißt, das Model in Domain Engineering dient als wieder verwendbares Repository, aus dem die ganze Software im Anwendungs-Engineering-Prozess generiert wird. Das ist idealerweise so. In der Realität kann ein komplexes System nie ohne Aufwand automatisch erzeugt werden. Das sagt auch die schwach steigende Linie in der Abbildung 2.17. Diese Steigung bedeutet, dass auch wenn man die Software generiert, es immer noch ein kleiner Aufwand notwendig ist, um diese Software vollständig zum Laufen zu bringen (z.B. das was sich nicht generieren lässt, muss manuell nach gebaut werden).

Résumé

SPL bieten einen pragmatischen Weg, Software für systematische Wiederverwendung zu

entwickeln. Bezogen auf die Wiederverwendung von Diagrammen kann das Prinzip nicht direkt angewendet werden, sie bietet aber einige Ideen, die übernommen werden können. Eine davon ist die Trennung zwischen Prozessen für Definition von wieder verwendbaren Software-Artefakten (in dem Fall Diagramm) und Prozessen zur Erzeugung der Software (konfigurierte Diagramme mit Daten).

3 Design und Implementierung

Das LD4UViz Framework ist charakterisiert durch die Erfassung und die anschließende interaktive Visualisierung der Daten, die per SPARQL-Queries aus DBpedia gewonnen werden. Somit entstehen speziell auf die Queries zugeschnittene Diagramme auf die später zugegriffen werden kann. Damit dieser Zugriff möglich wird, muss für die Visualisierung durchgeführte Mapping (siehe Kapitel 3.3.2.2) samt der SPARQL-Query (Suchanfrage), kurz Query genannt, gespeichert werden.

Es handelt sich beim LD4UViz Framework um ein plattformunabhängiges Framework. Daher kann es von jeder Client, der dementsprechend eingestellt ist, kontaktiert werden. Das Konzept der Visualisierung besteht aus drei wichtigen Punkten (siehe Abbildung 3.1):

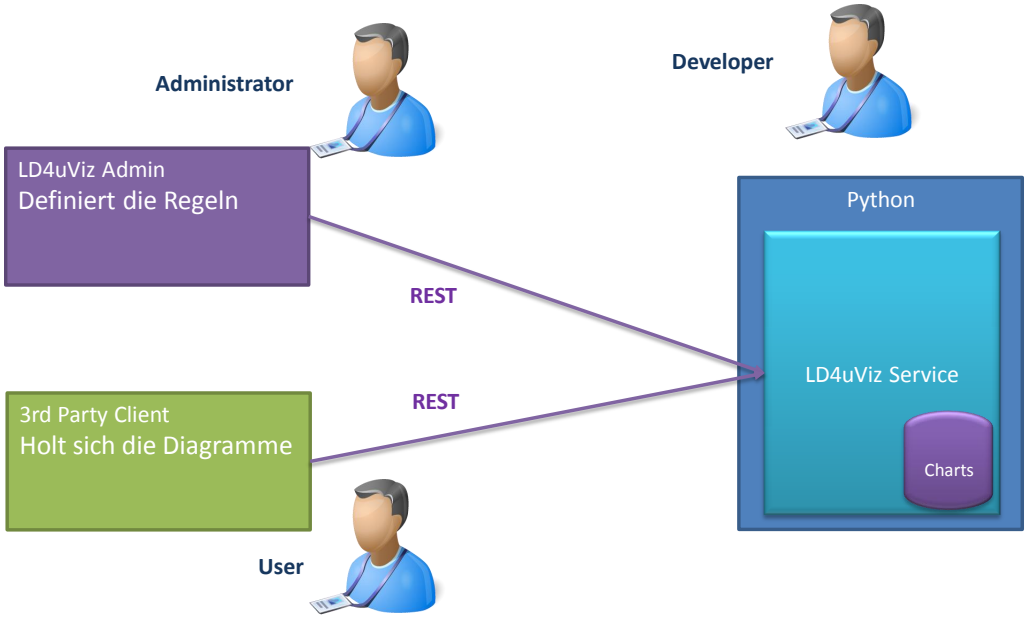

Abbildung 3.1: Client-Server Architektur

- Client als Benutzer: Der Client kontaktiert mit einer SPARQL-Query den Visualisierungsserver. Der LD4UViz Server holt für diese Query aus der entsprechenden Repository die Ergebnisse und delegiert sie samt der Query an das Visualisierungsteil des Frameworks. Der Inhalt der Ergebnisse werden in diesem Bereich anhand bekannter Muster klassifiziert und dem Client anschließend das entsprechende Diagramm zurückgeliefert.

- Client als Administrator: Wenn für eine eingegebene SPARQL-Query kein fertiges Mapping im Framework vorhanden ist, soll der Benutzer die Möglichkeit haben dieses selbst zu generieren. Dafür muss er als erstes ein Diagramm aus der Liste der unterstützten Diagramme auswählen und das Mapping (siehe Kapitel 3.3.2.2) für dieses Diagramm ausführen. Das dadurch entstandene Mapping kann anschließend gespeichert und jederzeit abgerufen werden.

- Entwickler: Wenn ein neues Diagramm gewünscht wird, das bisher vom Framework nicht unterstützt wurde, kann der Entwickler dieses Vorhaben in die Tat umsetzen und dieses Diagramm ins Framework integrieren.

In den nachfolgenden Kapiteln wird näher auf das Design und in die Implementierung des Frameworks eingegangen.

3.1 Konzept

Die Entscheidung hinsichtlich der Wahl eines SPARQL-Endpoints, mit dem das LD4UViz Framework interagieren soll, fiel auf DBpedia. Als erstes soll daher die strukturelle Aufbau der von DBpedia angebotenen Daten untersucht werden, damit im Visualisierungsvorgang der richtige Umgang mit diesen Daten sichergestellt ist.

Aus der Analyse der Datensätze (volle Spezifikation ist in [DBp11] nachzulesen) aus DBpedia ist ersichtlich, dass eine hierarchischenBeziehung zwischen den Daten vorliegt. Man erkennt aber auch, dass unterschiedliche Kategorien horizontal zu dieser Hierarchie stehen. Aus diesem Grund kann es keine ideale Suche bzw. Spezialisierung für alle Domänenbereiche geben. Je stärker diese horizontale Verbindung ist, desto schwieriger ist es den gezielten Domänenbereich herauszufiltern, was das Visualisierungsvorhaben zusätzlich erschwert.

Aus diesem Grund macht es mehr Sinn die Visualisierung nur auf bestimmte Domäne auszulegen und zwar auf die, von denen angenommen wird, dass sie für den User interessant sein können. Dieses Vorhaben kann schlussendlich mit der richtigen Auswahl der Diagramme in die Tat umgesetzt werden. Welche Diagramme sich am besten für die visuelle Repräsentation der Daten aus den gewünschten Domänen eignen und wodurch sie sich für diese Domäne klassifizieren wird im Folgenden genau erläutert.

3.1.1 Visuelle Repräsentation

Google Charts sind nicht nur deswegen, weil sie eine große Anzahl an Basis-Diagramme zur Verfügung stellen, die dazu noch interaktiv sind, die besten Kandidaten für dieses Framework, sondern auch deswegen, weil sie sich einfach in eigene Projekte integrieren lassen. Google Chart Tool API erzeugt die Diagramme mittels JavaScript, wodurch sie in jeder Plattform ohne zusätzlichen Browser Plug-ins (wie Flash) dargestellt werden können[1]. Folgende Google Charts sind am besten für dieses Framework geeignet:

- Bar Chart

[1] http://m.evolaris.net/google-chart-tools/

- Geo Chart
- Line Chart
- Pie Chart
- Table Chart

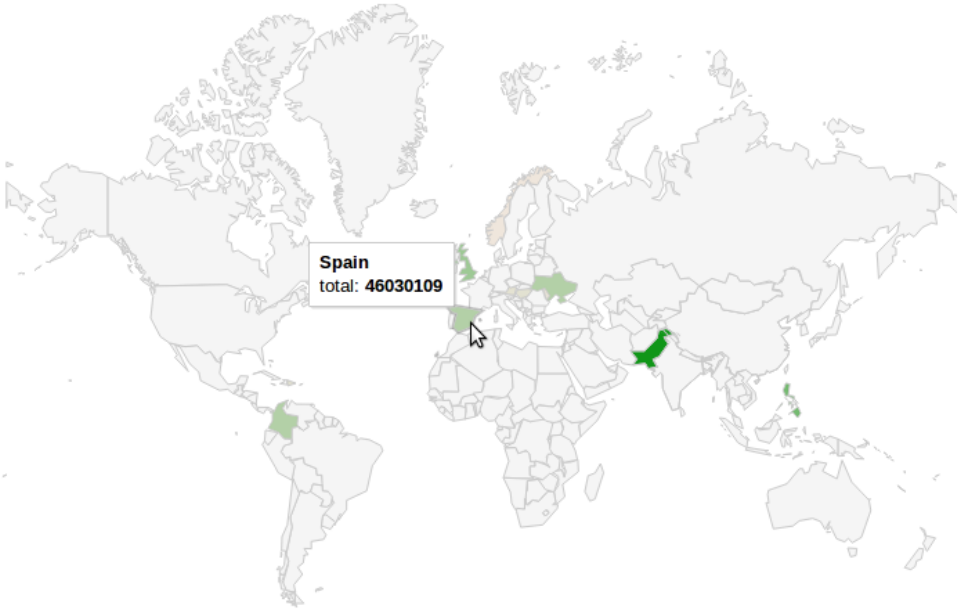

Abbildung 3.2: World Map, realisiert mit Hilfe des Google Geo Chart

Der Grund, warum die Entscheidung auf diese Diagramme fällt ist einfach. Sie können am einfachsten die gewünschten Daten aus den Kategorien darstellen, die mit den großen Domänen (Person, Place usw.) verbunden sind. Ein Beispiel:

Die Properties der Kategorie Person sind unter anderem birthPlace und deathPlace. Der Inhalt hinter dieser Properties verweist auf die Kategorie Country, die u.a. die Properties language, populationDensity und governmentType beinhaltet. Was bedeuten würde, dass man mit der entsprechenden SPARQL-Query von jeder Person das Geburts- bzw. Todesland erfahren kann, was wiederum bedeutet, das auch jegliche Informationen über das Land in Erfahrung gebracht werden können. Nun kann man diese Daten visuell repräsentieren. Die Information über die Einwohnerzahl eines Landes z.B. kann mit Hilfe des World Map visualisiert werden (siehe Abbildung 3.2) und mit Hilfe des Table Chart könnte man alle gelieferten Länder bzgl. der Landessprache, der Einwohnerzahl und des Regierungstyps miteinander vergleichen (siehe Abbildung 3.3). Die Information über die Einwohnerzahl kann natürlich auch per Bar Chart oder auch per Pie Chart hervorragend dargestellt werden.

Neben diesen Basis-Diagrammen können noch zwei weitere interaktive Diagramme verwendet werden, die jedoch bei der Umsetzung etwas komplizierter sind. Es handelt sich

	Land	Bevoelkerungszahl	Regierungsform
1291	Slovenien	2048951	Parlementaire republiek
1292	Slovenien	2048951	Parlamentarisk republikk
1293	Slovenien	2048951	República parlamentarista
1294	Slovenien	2048951	议会共和制
1295	斯洛文尼亚	2048951	Parliamentary republic
1296	斯洛文尼亚	2048951	Parlamentarische Republik
1297	斯洛文尼亚	2048951	República parlamentaria
1298	斯洛文尼亚	2048951	Parlamentaarinen tasavalta
1299	斯洛文尼亚	2048951	Parlementaire republiek
1300	斯洛文尼亚	2048951	Parlamentarisk republikk
1301	斯洛文尼亚	2048951	República parlamentarista
1302	斯洛文尼亚	2048951	议会共和制
1303	Switzerland	7866500	Federation
1304	Switzerland	7866500	Bundesstaat
1305	Switzerland	7866500	Federación
1306	Switzerland	7866500	Liittovaltio

Abbildung 3.3: Google Table Chart

dabei um SIMILE Timeline und JQCloud.

SIMILE Timeline ist an der Massachusetts Institute of Technology entwickelt und kann einfach via Javascript und HTML eingebunden werden, wobei es sich beim JQCloud um ein JQuery Plug-in handelt.

3.1.2 Systemarchitektur

Wie es sich aus den Angaben im Kapitel 3.1 erkennen lässt, wird anhand der Properties einer Query entschieden, welches Diagramm sich am besten für die visuelle Repräsentation der Ergebnisse dieser Query eignet. Damit dieses Diagramm auch dargestellt werden kann, muss der Administrator (Entwickler oder Client als Administrator) dafür ein Mapping (siehe Abschnitte 3.3.2.2) ausführen. Damit kann die Generierung des Diagramms vorangehen.

Um das fertige Diagramm später wieder verwenden zu können, muss das Mapping samt der SPARQL-Query gespeichert werden. Die Speicherung der Queries dient dazu, die eingegebene Query mit denen in der Datenbank zu vergleichen und somit das entsprechende Diagramm für diese Query zu liefern. Der Prozess von der Eingabe einer SPARQL-Query bis zum Speichern des Mappings wird im Kapitel 3.3 näher erläutert.

Der benutzerseitiger Prozessablauf, um auf die gespeicherten Mappins zuzugreifen, um dadurch das entsprechende Diagramm zu erhalten, lässt sich folgend beschreiben:

- Definition einer SPARQL-Query bzw. eines Interface zu einer SPARQL-Query Datenbank
- Aufbau einer Infrastruktur für das Kontaktieren des LD4UViz Servers (Visualizations-Framework)

- Anzeige des Diagramms an der gewünschten Stelle

Was danach beim Server geschieht, ist auf der Abbildung 3.4 dargestellt und wird im Folgenden etwas genauer erläutert:

RDF Endpoint Proxy dient als Verbindungspfad zum SPARQL-Endpoint.

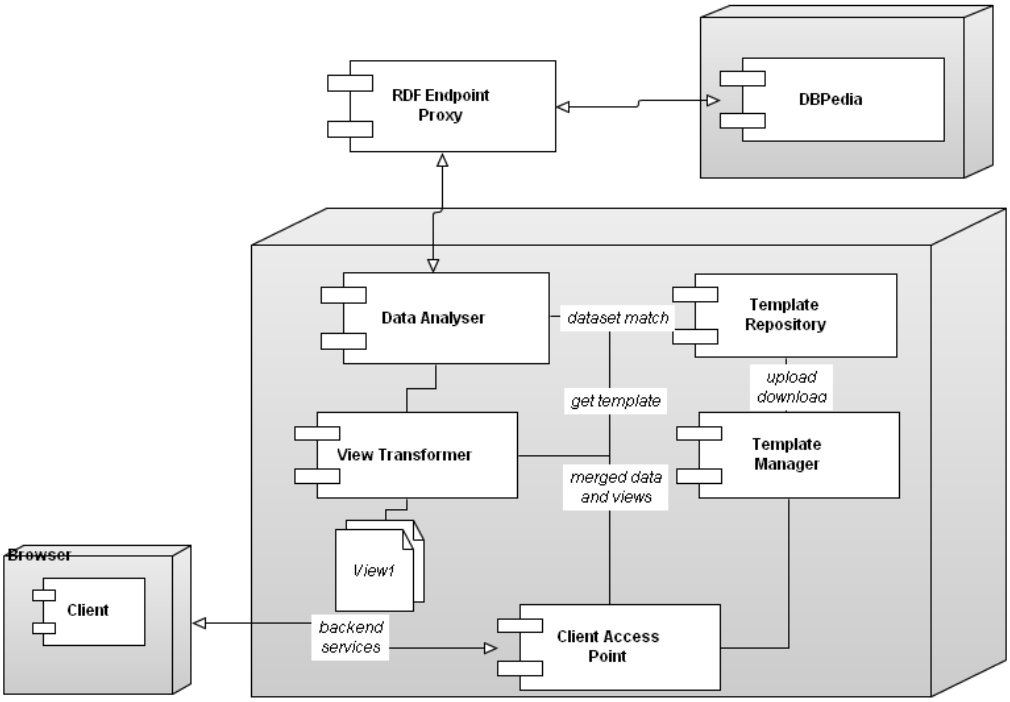

Abbildung 3.4: Konzept der Systemarchitektur vom LD4UViz Framework: Erkennung eines der gespeicherten Diagramme

Data Analyser klassifiziert die vom Benutzer eingegebenen Daten anhand bekannter Muster in der Datenbank (Template Repository) und liefert die entsprechende Vorlage (siehe Kapitel 3.3.2.1) an den View Transformer.

View Transformer nimmt die Vorlage und die vom RDF Endpoint Proxy gelieferten Ergebnisse und liefert das Zieldiagramm zurück.

Template Manager verwaltet die Vorlagen und ist der einzige Zugang zur Datenbank.

Client Access Point bietet die Funktionalität der internen Komponenten als Systemservices an.

3.1.3 Überblick des Ablaufs

Das beschriebene Konzept dient als Basis für die Umsetzung der Implementierung. Es beschreibt die notwendige Grundlage für eine mögliche Umsetzung der Visualisierung, die

wichtigsten Komponenten des Frameworks und definiert, wie auf ein gespeichertes Mapping zugegriffen werden kann, um für eine SPARQL-Query ein vorgefertigtes Diagramm zu erhalten.

Die Visualisierung erfolgt in folgenden Schritten:

- Empfang von SPARQL-Query
- Extrahierung der Properties der Query
- Feststellung auf welche Diagramme der Query mappbar ist (anhand der Properties der Query)
- Feststellung der Gruppen (Achsen der Diagramme) auf die Properties gemappt werden
- Holen vom Sourcecode des Diagramms
- Holen der SPARQL-Ergebnisse
- Generierung von Diagramm-Code
- Anzeige des fertigen Diagramms

Der Zugriff auf ein vordefiniertes Mapping erfolgt in folgenden Schritten:

- Empfang der SPARQL-Query und der Ergebnisse
- Holen des Mappings für diese Query
- Feststellung um welches Diagramm es sich handelt
- Holen vom Sourcecode dieses Diagramm
- Generierung des Diagrammcodes
- Anzeige des fertigen Diagramms

3.2 Design Patterns

Für die Realisierung des Frameworks wurden unterschiedliche Design Patterns eingesetzt, die sich als sehr nützlich erwiesen haben. Es handelt sich dabei um folgende Patterns:

- Adapter Pattern
- Factory Pattern
- Template Pattern
- Singeleton Pattern
- Wrapper Facade Pattern

Bevor diese Patterns einzeln vorgestellt werden, gibt es eine kleine Einführung in das Thema der Design Patterns.

3.2.1 Einführung

In heutigen Unternehmen sind die Geschäftsprozesse oft mit einer großen Menge von Anwendungen realisiert, die eine Spaghetti- Architektur bilden. Der Grund dafür ist nicht das fehlende Wissen in diesem Bereich, sondern weil es keine allgemeine Enterprise-Anwendung gibt bzw. keine ideale Integration, die Anforderungen aller Geschäfte erfüllt. Deshalb versucht man verschiedene Anwendungen zu kombinieren um daraus ein integriertes System zu bauen, z.B mit Hilfe der Design Patterns. Sie sind eine Sammlung von Mustern, die Richtlinien für Implementierungen beinhalten.

Das wichtigste Ziel der Integration ist effizienter, zuverlässiger und sicherer Datenaustausch zwischen den unterschiedlichen Anwendungen.

3.2.2 Adapter Pattern

Das Adapter Pattern gehört zu der Klasse der Enterprise Integration Design Pattterns, die Lösungen für die Integration von heterogenen Systemen anbieten und dadurch für eine Balance zwischen der idealen Integration und der Implementierung sorgen. Methoden, die dabei eingesetzt werden, sind:

- File transfer
- Shared database
- Remote method invocation
- Messaging systems

Der Grund, warum u.a diese Klasse von Design Patterns in dieser Arbeit ihren Einsatz findet, ist, weil sie für die Interoperabilitätsprobleme die einzige Lösung bietet.

Das Adapter Pattern ermöglicht den Klassen trotz inkompatibler Schnittstellen die Kommunikation, indem es der Übersetzung einer Schnittstelle in eine andere dient. Durch Adapter Pattern wird die Schnittstelle einer Klasse an die vom Client gewünschten Schnittstelle angepasst. Dafür muss auf Folgendes geachtet werden [Sch11b]:

- Welche Unterschiede haben die angeforderte und die geforderte Schnittstelle.
- Die geforderte Schnittstelle soll von einer neu implementierten Klasse zur Verfügung gestellt werden.
- Es soll möglich sein, das zu adaptierende Objekt an den Adapeter zu übergeben.
- Alle von der Schnittstelle verlangten Methoden sollen zur Verfügung stehen. Die ankommenden Suchanfragen sollen an das Ursprungsobjekt übertragen werden.
- Beachtung der Unterschiede bei der Fehlerausgabe
- Um den Ursprungsobjekt zu ummanteln sollte in der Applikation das Adapter-Objekt verwendet werden

Es wird unter zwei Typen von Adapter Patterns unterschieden, Objektadapter und Klassenadpater.

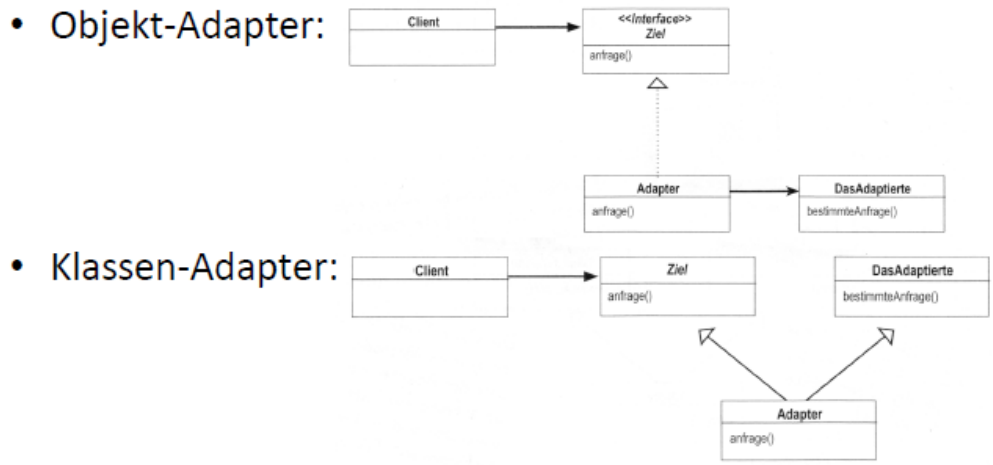

Abbildung 3.5: Adapter Patterns [Pro11]

Klassenadapter Nutzt die Mehrfachvererbung. Erbt sowohl die Implementierung der zu adoptierenden Klasse als auch die zu implementierende Schnittstelle. Die Programmiersprache JAVA ist ungeeignet für Klassenadapter, da es keine Mehrfachvererbung unterstützt.

Objektadapter Benutzt die einfache Vererbung und leitet alle Suchanfragen entsprechend per Delegation weiter.

3.2.3 Factory Method

Factory Method, auch Factory Pattern genannt, findet Anwendung, wenn eine Klasse die von ihr zu erzeugenden Objekte nicht kennen kann bzw. soll, oder wenn die Unterklassen die Entscheidung treffen, von welcher Klasse das erzeugte Objekt ist. Anwendungsbereiche von Factory Method sind Frameworks, die mehrere Dokumente gleichzeitig präsentieren können und die Klassenbibliotheken [fac11].

Dieses Pattern bietet den Entwicklern die Möglichkeit, dass ein Objekt nicht durch den Konstruktor sondern durch den Aufruf einer Methode erzeugt wird.

Die Abbildung 3.6 zeigt die Rollenverteilung im Factory Pattern. Die Factory Method des *Creators* wird von *ConcreteCreator* geerbt. *ConcreteCreator* erzeugt durch die umimplementierung diese Methode das *ConcreteProduct*, dass wiederum Product implementiert.

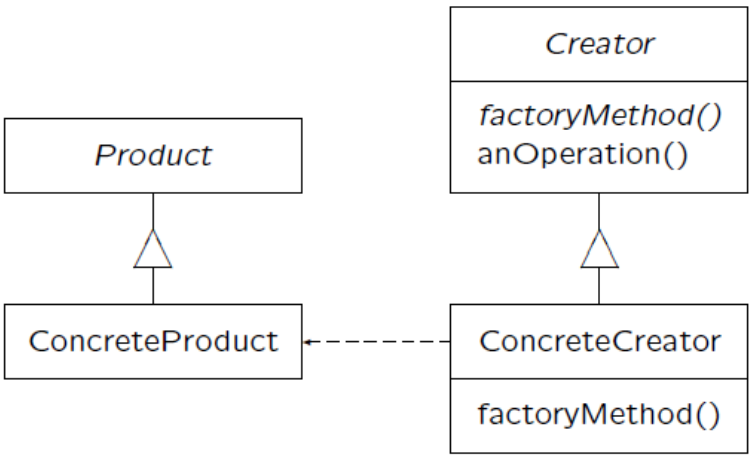

Abbildung 3.6: Factory Pattern [Wie11]

3.2.4 Template Method Pattern

Durch die Verwendung des Template Method Pattern werden in einer Methode nur die einzelnen Schritte des Algorithmus definiert, deren Implementierung überlässt man aber den Unterklassen. Der Vorteil daran ist, das dadurch die Grundstruktur des Algorithmus unverändert bleibt.

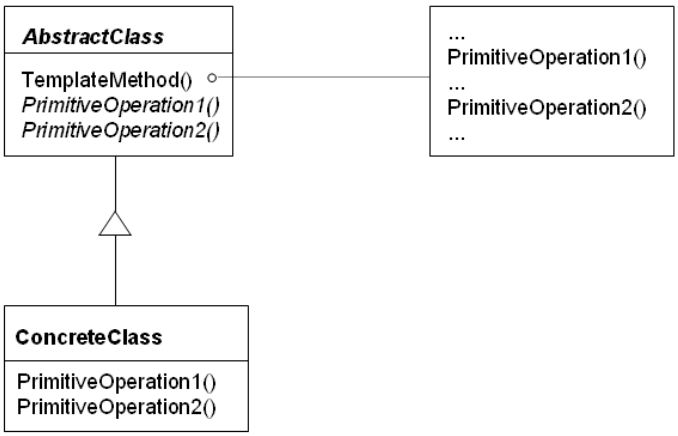

Abbildung 3.7: Template Method Pattern [tem11]

3.2.5 Singleton Pattern

Singleton Pattern findet Anwendung, wenn nur eine Instanz von einer Klasse mit einem globalen Zugriff existieren soll. Diese Instanz muss nur einmal erzeugt werden [Vyn12].

Konstruktor wird so aufgebaut (privat), dass er verhindert die Instanz standardmäßig zu erzeugen.

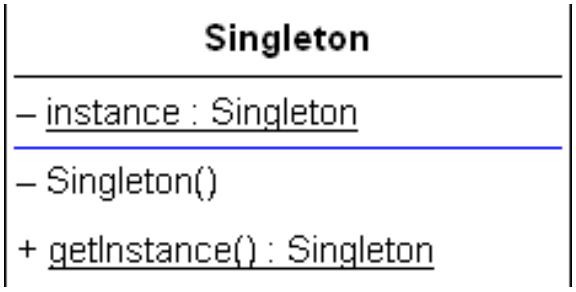

Abbildung 3.8: UML Diagramm des Singleton Patterns [Vyn12]

3.2.6 Wrapper Facade Pattern

Wrapper Facade Pattern kapselt die Funktionen und die Daten der existierenden nicht objektorientierten Schnittstelle in robuste, portable und wartbare Schnittstellen. Die komplexen Details des System werden vor den Benutzern verborgen.

Der Implementierungsablauf vollzieht sich in fünf Schritten:

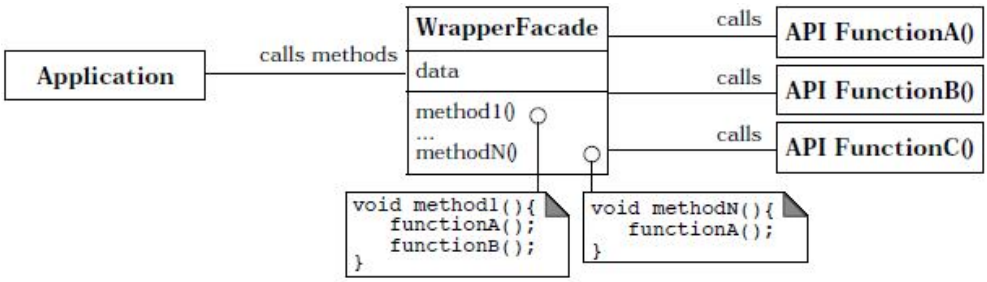

Abbildung 3.9: Klassendiagramm des Wrapper Facede Patterns [oE11]

1. Analyse der existierenden APIs,
2. Generierung des WrapperFacade:
   a) Generierung von abhängigen Klassen,
   b) Kapselung einzelne Funktionen in einer einzigen Methode,
   c) Mögliche Automatisierung des Konstruktors und Dekonstruktors der Operationen,
   d) Auswahl der Indirectionlevel,
   e) Feststellung der Kapselungmöglichkeiten von plattformspezifische Variationen,

3. Überprüfung, ob kontrollierte Zugriffe seitens Applikationen zu Implementationsdetails erlaubt werden sollen,
4. Definition eines Fehlerbehandlung Mechanismus,
5. Definiton von Hilfsklassen.

3.3 Prototyp

Der im Rahmen dieser Arbeit entwickelte Prototyp bietet eine intuitive Methode zur Visualisierung heterogener Daten an. Um dieses Vorhaben in die Realität umzusetzen, wird eine auf Vorlagen (Templates) basierte Methode angewandt. Der Prototyp unterstützt auch die systematische Wiederverwendung der generierten Diagramme.

In den folgenden Kapiteln wird der Prototyp aus unterschiedlichen Perspektiven beschrieben. Es handelt sich dabei um folgende Perspektive:

- Logische Sicht
- Entwicklungs-Sicht
- Prozess Sicht

3.3.1 Logische Sicht

Aufgrund der Flexibilität und der leichte Anpassbarkeit basiert das Client/Server Modell des LD4UViz Frameworks auf Serviceorientierte Architektur (SOA) (siehe Abbildung 3.10). Die darin angebotenen Services haben folgende Funktionalitäten:

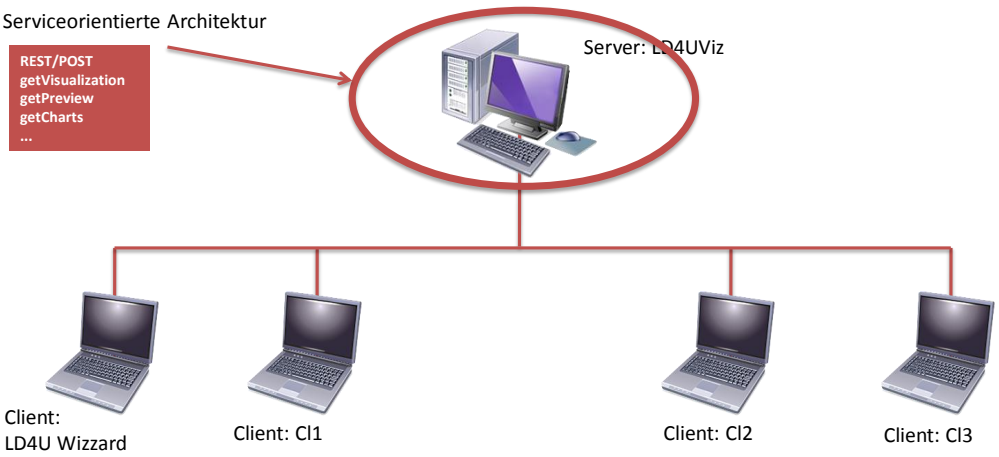

Abbildung 3.10: LD4UViz: Serviceorientierte Architektur

- aufgebaute SPARQL-Queries an den richtigen Endpoint weiterleiten
- ankommende Ergebnisse verwalten

- Mappings (siehe Abschnitt 3.3.2.2) definieren und verwalten
- Diagramm erstellen
- vordefinierte Diagramme holen

In der Abbildung (siehe Abbildung 3.10) sind ein paar der wichtigsten Services für dieses Framework ersichtlich:

- getVisualisation, holt das vordefinierte Diagramm
- getPreview, erstellt eine Vorschau des generierten Diagramms
- getCharts, listet alle vom Framework momentan unterstützten Diagramme auf
- usw.

Alle Services sind durch die Befehle GET oder POST ansprechbar.

3.3.2 Entwicklungs-Sicht

Die grobe Server-Architektur aus der Entwicklungs-Sicht ist in Abbildung 3.11 ersichtlich. Sie besteht aus folgenden Komponenten:

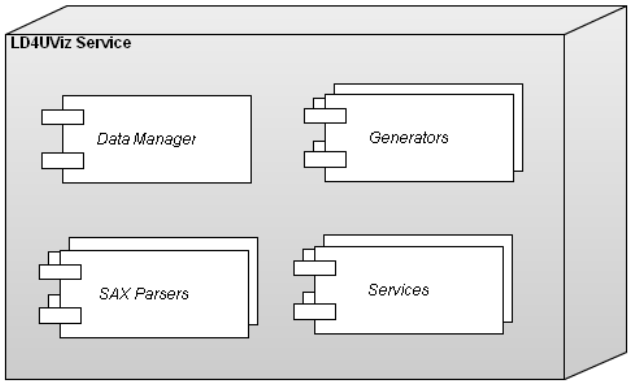

Abbildung 3.11: Server-Grobarchitektur

- Datamanager
- SAX-Parsers
- Generators
- Service

Bevor mit der Beschreibung der einzelnen Komponenten angefangen werden kann, sollte eine Diagramm- und Mappinganalyse durchgeführt werden.

3.3.2.1 Diagrammanalyse

Sowie im Kapitel 3.1 bereits angekündigt wurde, eignen sich Google Charts aufgrund ihrer leichten Integrierbarkeit am besten für das LD4UViz Framework. Dennoch kommen zwei weitere Diagramme (SIMILE Timeline und JQCloud) zum Einsatz um eine gewisse Vielfalt anzubieten. Die Einbettung dieser Diagramme ist jedoch verglichen mit den Google Charts etwas komplexer. Vor allem die Einbettung des JQClouds ist mit viel Aufwand verbunden.

Für eine erfolgreiche Einbettung der Diagramme müssten als erstes die Sourcecodes der Diagramme analysiert werden. Diese Analyse ist aus diesem Grund notwendig, da aus diesem Code eine Vorlage erstellt werden soll. Bei der Erstellung der Vorlage muss vorher bekannt sein, welche Teile des Codes fix und welche variable sind. Die bereitgestellten Dokumentationen legen u.a. die Komponenten des Diagramms deren Gültigkeitsbereich und die Funktionalität der einzelnen Methoden vor. Diese Informationen sind zwar notwendig aber es muss mehr über dem Code in Erfahrung gebracht werden, damit eine Vorlage bereitgestellt werden kann.

Die Charakteristiken eines Diagramms können aus der Abbildung 3.12 entnommen werden.

Bei den ausgewählten Google Charts (*Bar Chart, Geo Chart, Line Chart, Pie Chart und Table Chart*) fällt auf, dass der Code mit ein paar kleinen Unterschiede immer derselbe ist (siehe Abbildung 3.13). Das macht es einfacher immer neuer Google Charts in das Framework zu integrieren. Ziel ist nun aller Komponenten eines Diagramms, also Achsen

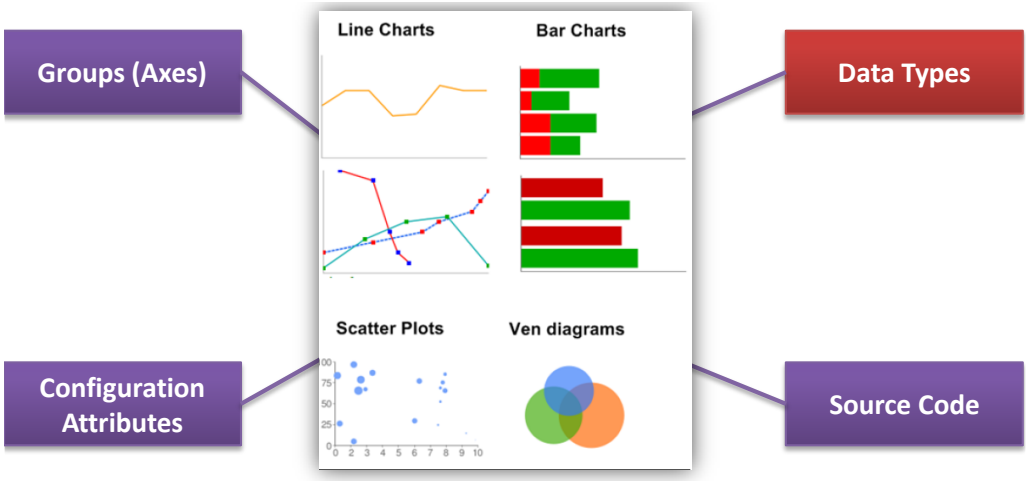

Abbildung 3.12: Charakteristiken eines Diagramms

samt ihrem Typ, den Konfigurationsparameter und dem Sourcecode, in einer Datenbank zu hinterlegen, damit alle Diagramme und Mapping persistent gehalten werden können. Dabei ist wichtig zu beachten, wie diese Komponenten in der Datenbank repräsentiert werden. Eine Repräsentationsmöglichkeit ist im Folgenden angegeben. Als Datenbank in diesem Beispiel dient XML.

Name	Beschreibung
String	-
Number	-
Double	-
Float	-
Long	-
Integer	-
Boolean	-
Date	-
Any	Beliebiger Datentyp

Tabelle 3.1: Potenzielle Datentypen einer Diagrammachse

Listing 3.1: Beschreibung des Diagramms in XML

```
1 </specification name="googlelinechart">
2   <group type="STRING" name="x-axis" persistence="MANDATORY">
3     x-axis
4   </group>
5   <group type="NUMBER" name="y-axis" persistence="MANDATORY">
6     y-axis
7   </group>
8 </specification>
```

Das Beispiel zeigt, wie die einzelnen Achsen eines Diagramms samt ihrem Typ in einer XML-Datenbank hinterlegt werden können. Genauso kann auch der Sourcecode der einzelnen Diagramme samt Konfigurationsparameter in der Datenbank repräsentiert werden.

Zurück zum obigen Beispiel. In diesem XML Beispiel beschreibt das Element „group“ die einzelnen Achsen, das die Attribute „name“ und „type“ beinhaltet, wobei das gesamte Inhalt unter dem Element „specification“ zusammengefasst ist. Diese Darstellungsform kann für alle Diagramme übernommen werden, womit jedes Diagramm sein eigenes XML-Dokument bekommt. Auf der Tabelle 3.1 sind mögliche Datentypen für eine Diagrammachse aufgelistet.

Als Datenbank kommen mehrere Datenbanken, wie Mysql, XML, MSSQL usw., in Frage. Die Entscheidung fällt aber auf XML und zwar aus folgenden Gründen:

- Portabilität: Da XML-Parsers für mehrere Sprachen und Plattformen existieren, ist die XML-Technologie hoch portabel.
- Modularität: Datentypen können von Benutzer festgelegt werden (in Schema). Bei der XML-Instanz kann man diese Typen beliebig oft wiederverwenden.
- XML ist lizenzfrei.

Nun muss überlegt werden, wie der Sourcecode in einem XML-Dokument am besten zu repräsentieren ist. Der Sourcecode besteht aus wiederverwendbaren und aus variablen Teilen. Die wiederverwendbare Teile können vollständig ins XML-Dokument übernommen

Abbildung 3.13: Sourcecode von Google Pie Chart und Google Bar Chart

werden, wobei für den Rest Annotation in Anwendung kommt (siehe Zeile 9 und 10). Es wir also mit Hilfe der sogenannten Platzhalter (z.B. „@@@“) die variablen Stellen vermerkt. Die Konfigurationsparameter werden von Client vorgegeben.

Listing 3.2: Vorlage für Google Pie Chart

```
1  <template><![CDATA[
2    var loc=config.location;
3    var chartWidth=config.width;
4    var chartHeight=config.height;
5    var chartTitle=config.title;
6    google.load("visualization", "1", {packages:["corechart"],
7      callback : function drawChart() {
8        var data = new google.visualization.DataTable();
9        @@@COLUMNS@@@
10       @@@ROWS@@@
11       var options = {
12         width: chartWidth, height: chartHeight,
13         title: chartTitle
14       };
15       var chart = new google.visualization.PieChart(
16                   document.getElementById(loc));
17       chart.draw(data, options);
18   }});
19   ]]>
20 </template>
```

Auf der Tabelle 3.2 sind alle im Framework aktuell unterstützten Charts samt ihre Komponenten und Besonderheiten abgebildet:

3.3.2.2 Mappinganalyse

Nachdem alle XML-Dokumente mit dem entsprechenden Inhalt vorliegen, sollte die Beziehung zwischen ihnen festgehalten und diese Beziehung für Visualisierungszwecke angewendet werden. Allerdings steht der Zusammenhang zwischen dem Inhalt dieser Dokumente

Chart Name	Bemerkung	Achsen	Typen von Achsen
Google Bar Chart	Interaktiv	x-Axis, y-Axis	STRING, NUMBER
Google Geo Chart	Interaktiv	location, value1	STRING, NUMBER
Google Line Chart	Interaktiv	x-Axis, y-Axis	STRING, NUMBER
Google Pie Chart	Interaktiv	label, value	STRING, NUMBER
Google Table Chart	Interaktiv, Duplizierbare Achsen	column	STRING, für Duplikate siehe 3.1
SIMILE Timeline	Interaktiv, Fixe und optionale Achsen	startdate, enddate, name (optional)	DATE, DATE, STRING
Cloud	Interaktiv , Fixe und optionale Achsen, Enterprise Integration Pattern (siehe 3.2.2) für url	entry, count, url (optional)	STRING, STRING, STRING

Tabelle 3.2: Alle im Framework aktuell unterstützen Charts mit Eingeschaften

mit den Ergebnissen einer SPARQL-Query im Mittelpunkt. Denn es sind genau diese Ergebnisse (Werte, die sich hinter den Properties verstecken), die visualisiert werden sollen. Unter dem Mapping versteht man in unserem Fall also die Bindung der Ergebnisse zu den Achsen eines Diagramms, die dann im Sourcecode in die variablen Stellen übergehen sollen. Während dieses Vorgangs sollte man immer auf dem Datenformat achten. Wenn das Mapping abgeschlossen ist, wird es durch das Speichern in einem eigenen XML-Dokument für zukünftige Zugriffe hinterlegt. In diesem XML-Dokument befinden sich auch der Name des jeweiligen Chart und die dazugehörige SPARQL-Query.

3.3.2.3 Résumé der Diagramm- und Mappinganalyse

Das Framework soll so konfiguriert sein, dass es mehrere Diagramme unterstützt. Dabei sollte der Entwickler es leicht haben neue Diagramme hinzuzufügen. Das Ablegen der Komponenten eines Diagramms in die dazugehörigen XML-Dokumente erleichtert dieses Vorhaben, da somit im Code nichts verändert werden muss. Einzig und allein sollte ein neues XML-Dokument mit den Achsen des neuen Diagramms erstellt und das Dokument mit den Sourcecodes dementsprechend erweitert werden. Auch das durchgeführte Mapping wird nach der Generierung des Diagramms in einem XML-Dokument festgehalten. Wenn der Benutzer für eine SPARQL-Query durch das Kontaktieren des LD4UViz Frameworks ein vorgefertigten Diagramm haben möchte, wird als erstes mit der eingegebenen Query in dem jeweiligen XML-Dokument nach dem entsprechendem Mapping gesucht, mit die-

ser Mappinginformation der Sourcecode des Diagramms geholt und das Diagramm an der vom Benutzer gewünschten Stelle angezeigt.

Es bleibt nur noch eine Frage offen, wie sollten die XML-Dokumenten gelesen und schlussendlich der Sourcecode des Diagramms generiert werden. Auf der Suche nach Lösungen für diese Frage stoßen wir auf die Begriffe SAX-Parser und Generatoren, die im Kapitel 3.3.2.4 ausführlich erläutert werden.

In der Abbildung 3.14 ist das Datamodel von den erwähnten XML-Dokumenten ersichtlich. Unterschiedliche Farben bezeichnen die jeweiligen Datengruppen, die auch dementsprechend in der Datenbank repräsentiert werden.

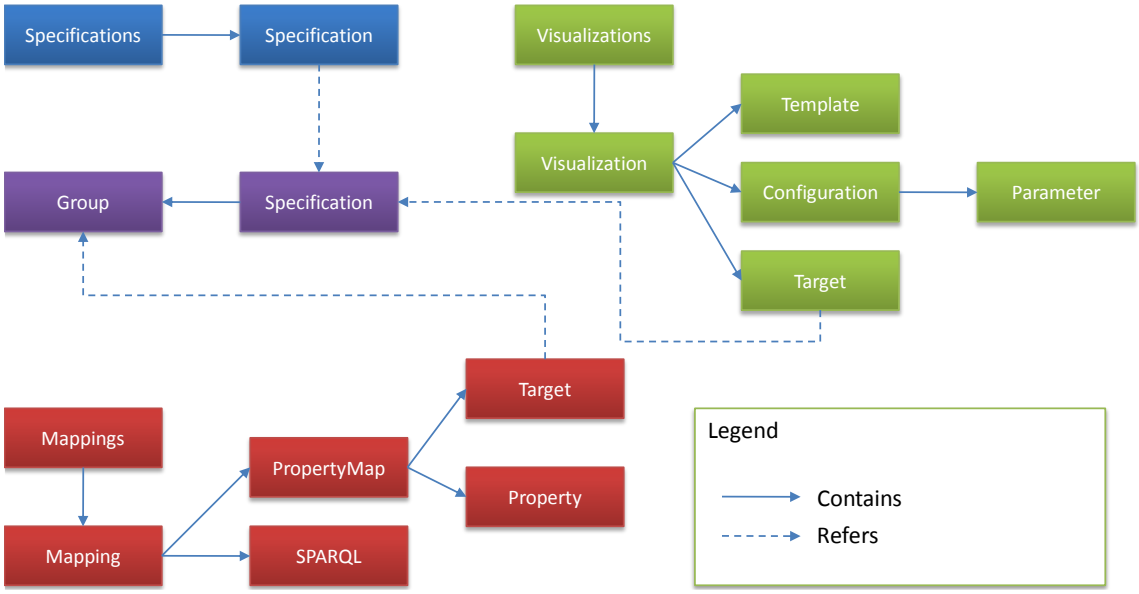

Abbildung 3.14: Datamodel

3.3.2.4 Service Komponenten

DataManager Damit mit der Generierung eines Diagramms angefangen werden kann, sollten als erstes alle XML-Dokumente initialisiert und in gewünschte Datenstrukturen hinterlegt werden. Dafür ist der DataManager verantwortlich. Die in das Framework integrierte XML-Dokumente sind auf der Tabelle 3.3 ersichtlich.

Die Aufgabe des DataManagers ist damit aber nicht vollbracht, sondern fängt erst an. Je nach Anforderung vom Server anhand Services sowie *getPreview*, *getVisualization*, *saveChart*, werden unterschiedliche Methoden in DataManager angesprochen bzw. ausgeführt. Diese Methoden interagieren mit den Daten hinter den oben erwähnten Datenstrukturen. Die wichtigsten Services werden im Abschnitt 3.3.3 behandelt.

Erhaltung der Konsistenz ist eine andere wichtige Aufgabe des DataManagers, damit der

XML-Dokument	Beschreibung
<CHART_NAME>.xml	XML-Dokument des jeweiligen Chart
charts.xml	beinhaltet den Namen aller aktuell unterstützten Charts
repository.xml	beinhaltet die Sourcecodes aller aktuell unterstützten Charts
rules.xml	beinhaltet das fertige Mapping

Tabelle 3.3: XML-Dokumente

lokale Zustand und der Zustand in der Datenbank exakt derselbe sind.

Eine weitere Eigenschaft von DataManager ist, dass er als Singleton Pattern (siehe 3.2.5) realisiert wird, weil mehrere Zugriffe aus unterschiedlichen Quellen auf DataManager stattfinden.

Listing 3.3: Realisierung des DataManagers als Singleton

```
1 _singleton = DataManager("static/charts.xml", "static/rules.xml",
2 "static/repository.xml")
3 _singleton.init()
4 def DMInstance(): return _singleton
```

SAXParsers Ein Parser findet Anwendung, um auf die in einem Dokument enthaltenen Informationen zu gelangen, um sie später zur Verfügung stellen zu können. Das Dokument wird dabei einmal vom Parser durchgelesen, analysiert und die darin enthaltenen Informationen in einer gewünschten Form darstellt. Genau diesem Zweck dient auch der XML Parser.

Es existieren unterschiedliche XML-Parser, wobei sie sich anhand zwei Kriterien unterscheiden [uw11]:

- Validierend oder nichtvalidierend
- Art der Schnittstelle für den Zugriff (DOM, SAX)

Der Unterschied zwischen dem validierenden und dem nicht validierenden Parser ist, dass beim validierenden Parser überprüft wird, ob die Struktur des Dokuments mit der vom Document Type Definition (DTD) oder Schema vorgegebenen übereinstimmt, und beim nichtvalidierenden Parser auf diese Übereinstimmung kein Wert gelegt wird.

Die unterschiedlichen Schnittstellen sind folgende:

Simple API for XML (SAX) [uw11]: SAX Parser löst, wenn er ein bestimmtes XML-Konstrukt liest, ein Ereignis aus, die unterschiedlich behandelt werden kann. Es kann sich dabei um folgende Ereignisse handeln:

- öffnendes Tag
- Text
- schließendes Tag

Sobald eines dieser Ereignisse auftritt, wird der entsprechende Evenhandler aufgerufen, der das Ereignis wunschgemäß auswertet.

Document Object Model (DOM) [uw11] Beim DOM wir beim Parsen des XML-Dokuments ein Baum aufgebaut. Das Root-Element des Dokuments wirkt dabei als Wurzel. Das DOM-API bietet den Zugriff auf diesem Baum um Veränderungen vorzunehmen.

Der SAX-Parser ist weniger aufwendig und schneller als der DOM-Parser. Aus diesem Grund wird für dieses Framework der SAX-Parser eingesetzt. Da beim Erstellen der XML-Dokumente darauf geachtet wird, dass sie valid sind, wird auf das Validieren durch den Parser verzichten.

Aus dem Grund, dass mehrere Parsers für unterschiedliche XML-Schemen benötigt werden, aber diese Parsers ihren abstrakten Sicht trotzdem beibehalten sollen, wird eine generische Klasse, *GenericParser*, verwendet (Template Method Pattern, siehe Kapitel 3.2.4).

Listing 3.4: Aufbau der Klasse GenericParser nach dem Template-Pattern

```
1 import xml.sax
2 class GenericParser(xml.sax.ContentHandler): # extends sax parser
3     def __init__(self): #constructor
4         xml.sax.ContentHandler.__init__(self)
5     def prepare(self): #This method has to be overridden
6         pass
```

Die Methode *prepare* startet den Parser und stellt den ausgelesenen Inhalt des XML-Dokuments als Python Dictionary dar. Die restlichen Methoden werden von den jeweiligen Parsers implementiert.

Generators Sowie bereits im Abschnitt 3.3.2.1 angekündigt, sollen auch die Sourcecodes jedes einzelnen Diagramms in einer eigenen XML-Dokument vorliegen. Für die Generierung des Diagramms wird somit dieser Codes aus dem Dokument gelesen (in DataManager) und mit dem entsprechenden Inhalt, also mit den Ergebnissen einer SPARQL-Query verbunden. Für die Realisierung dieses Vorhaben kommen die Generatoren zum Einsatz.

So wie jedes Diagramm einen Parser hat, so sollte es auch einen eigenen Generator besitzen, der den Sourcecode zum Ausführen bereitstellt. Die einfachste Variante eines Generators dient, sowie bereits in 3.3.2.2 erwähnt, der Fusion der SPARQL-Ergebnisse mit dem Code-Inhalt aus dem XML-Dokument. Ein Generator kann aber auch im Stande sein andere Aufgaben zu übernehmen, u.a. auch hoch komplexe. Einige Beispiele dafür:

Es kann vorkommen, dass der Generator eine Validierung durchführen muss, um zu kontrollieren, ob die Daten, die vom RDF-Endpoint kommen, auch in einem entsprechenden

Situation	Maßnahmen
Unterschiedliche Typen	Prozessabbruch, Warnung, Umwandlung
Unterschiedliches Format	Prozessabbruch, Warnung, Umwandlung
Mapping	Prozessabbruch, Warnung, EIP

Tabelle 3.4: Mögliche Situation für den Generator

bzw. erwarteten Format vorliegen. Wenn dem nicht so ist, hat der Generator die Wahl zwischen dem Abbrechen der weiteren Verarbeitung, der Ausgabe einer Warnung oder der Umwandlung des Formats.

Ein weiteres Beispiel betrifft dem Datentyp (erwartet wird ein String, geliefert wird ein Integer). Auch hier hat man die gleichen Möglichkeiten für die Behandlung (siehe Tabelle 3.4).

Diese beiden Fälle können noch relativ gut und simple behandelt werden, komplexer wird es aber, wenn ein „eins zu eins Mapping“ nicht durchführbar ist (siehe Abschnitt 3.3.2.6). In so einem Fall muss der Generator viel mehr Arbeit verrichten um das Mapping doch zu ermöglichen und somit diesen komplizierten Vorgang von dem unerfahrenen Benutzer fernzuhalten. Mit dieser Trennung der Aufgabenbereiche, hat der Benutzer nur eine einfache Vorarbeit zu verrichten die ausführliche Verarbeitung findet beim Generator statt. Das Ergebnis dieser Verarbeitung wird anschließend in einer von Client-Library verständlicher Form verpackt und dem Benutzer das fertige Diagramm geliefert. Die Implementierung der Generatoren basiert auf dem Template Method Pattern (siehe 3.2.4).

Listing 3.5: GeneratorFactory-Realisierung

```
1  import bargenerator
2  import piegenerator
3  import geogenerator
4  ...
5  class GeneratorFactory():
6
7      def __init__(self):
8          pass
9
10     def createFactory(self, name, mappingInfo, result):
11       if name=="googlebarchart":
12          barGenerator= bargenerator.BarGenerator(mappingInfo, result)
13          return(barGenerator)
14
15       if name=="googlepiechart":
16          pieGenerator= piegenerator.PieGenerator(mappingInfo, result)
17          return(pieGenerator)
18
19       if name=="googlegeochart":
20          geoGenerator= geogenerator.GeoGenerator(mappingInfo, result)
21          return(geoGenerator)
22       ...
```

```
23          else:
24              print("WARNING: generator for %s does not exist" %name)
25              return(None)
```

Service Die Services werden vom Server angeboten und vom Client verwendet um gewisse Aktionen auszulösen. Diese Aktionen reichen vom Holen der SPARQL-Ergebnisse bis um Speichern eines fertig definierten Diagramms. Alle Services basieren auf dem REST Architekturstil.

Der Codeausschnitt im Folgenden soll zeigen, wie ein Post Befehl vom Client aufgebaut und an dem Server verschickt wird. Ein Post Befehl kann eine einzige Variable oder ein ganzes Paket mit Arrays an dem Server schicken.

Listing 3.6: POST Anfrage an getVisualisation Service

```
1  function getVisualization(url, sparqlContent){
2
3    $.post("/viz", { cmd: "getVisualization", sparql: sparqlContent},
4    function(dt) {
5      $("#dlg").dialog("close");
6      if(dt.error!=null){
7        err(dt.error);
8        return;
9      }
10     try
11     {
12       globals.charts=dt;
13       var charts="";
14       for(var i=0;i<globals.charts.length;i++){
15         var nameOfCharts=globals.charts[i].name;
16         charts = charts+'<li id='+nameOfCharts
17                       +' class="ui-widget-content">'+
18         nameOfCharts+'</li>';
19       }
20       $("#selectable2").html(charts);
21     }
22     catch(ex)
23     {
24       alert(ex);
25     }
26
27   }, "json"
28 ).error(function() {alert("Error while preparing preview."); });
29 }
```

Sobald das Paket an dem Server angekommen ist, werden die entsprechenden Methoden in unterschiedlichen Klassen kontaktiert und die gewünschten Daten an dem Client geliefert.

Listing 3.7: getVisualization Service

```
1  @csrf_exempt
2  def service(request):
3      cmd = ""
4      if request.method == 'GET':
5          cmd=request.GET['cmd']
```

```
6       else:
7           cmd=request.POST['cmd']
8           ...
9            if(cmd=="getVisualization"):
10          try:
11              sparql=request.POST['sparql']
12              dataResults = sparqlUtil.sparqlUtil().query(sparql)
13
14              inProcessorObject=inprocessor.InProcessor(sparql, dataResults)
15              inProcessorObject.process()
16              resultArray=inProcessorObject.resultArray
17              response.content = json.dumps(resultArray)
18              response['Access-Control-Allow-Origin']="*"
19              response['Access-Control-Allow-Methods']="POST, GET, OPTIONS"
20              response['Access-Control-Allow-Headers']="*"
21              return response
22          except Exception as inst:
23              dataManagerObject.reInit()
24              msg = "ERROR (ld4uviz - getVisualization): %s"%inst
25              return HttpResponse(json.dumps({'error' : ''+msg+''}))
26              ...
```

Auf die anderen Services wird im Kapitel 3.3.3 eingegangen. Hier werden auch die wichtigsten Services noch einmal detaillierter beschrieben.

Résumé Der Vorteil von Generatoren und SAXParsers sind folgende:

- Erweiterbarkeit
- Für jedes neue Diagramm muss nur ein Parser und ein Generator implementiert werden
- einfache Integration der neuen Diagramme

Der Generierungsprozess, der im Service *getVisualization* enthalten ist, wird von festen und austauschbaren Komponenten realisiert. Um die Anforderung 1.2.2 zu erfüllen, soll der Prototyp nicht nur auf gerade unterstützte Diagramme begrenzt sein, sondern auch die Integration weiterer Diagrammen ermöglichen. Die Architektur in der Abbildung 3.15 ist genau für diese Zwecke konzipiert. Die rot markierten Komponenten sind austauschbare Funktionen, die erwähnte Erweiterung ermöglichen. DataManager sowie zwei Komponenten zur Ein- und Ausgabenverarbeitung bleiben unverändert, unabhängig davon, welche Diagramme und Mapping im System vorliegen. Wenn ein neues Diagramm ins System integriert werden soll, müssen der dazugehörigen Parser und der Generator vom Entwickler realisiert werden. Der Entwickler ist weiter verpflichtet die Spezifikation in XML und im entsprechenden Schema festzuhalten, die bei der Generierung des Diagramms vom Parser gelesen werden. Dabei hat der Entwickler die Möglichkeit das Diagramm ohne Beschränkungen zu beschreiben. Der Parser muss aber das Standardinterface vom GenericParser implementieren und sein Ergebnis an dem Format der Komponente *InProcessor* anpassen. Auf ähnlicher Weise ist das Interface vom Generator an *OutProcessor* anzupassen. Optional kann der Entwickler für den Generator zusätzliche Eigenschaften definieren, die bei der Spezifikation vom Mapping eine wichtige Rolle spielen (z.B. Validierung des Datentyps zwischen Properties und Achsen). Falls ein neues Diagramm eine Bibliothek

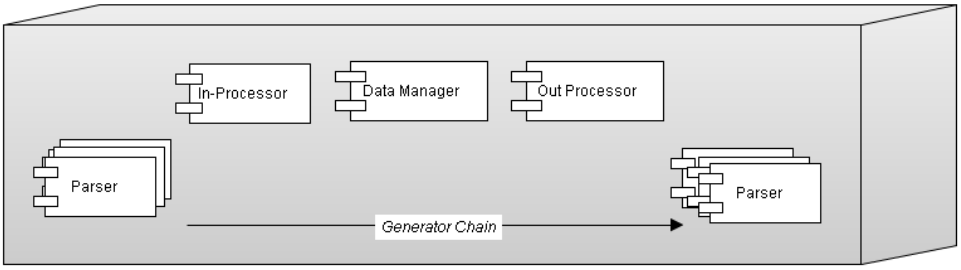

Abbildung 3.15: Serverarchitektur für den hohen Erweiterungspotential

benötigt, die nicht im *ld4uviz.js* registriert wurde, ist sie in dieser Javascript Datei entsprechend hinzuzufügen.

Besonders wichtig ist, dass der Entwickler eine Vorlage für die Visualisierung heterogene Daten spezifiziert. Diese Vorlage wird, wie im Abschnitt 3.3.2.2 beschrieben, mit den SPARQL-Ergebnissen fusionieren und für die Visualisierung am Client verwendet. Auf der Seite des Clients ist somit absolut gar nichts zu tun. Die Erweiterung des Frameworks um ein Diagramm scheint komplizierter zu sein, als das Diagramm manuell zu implementieren. Aber da diese Erweiterung nur einmal passiert und das Diagramm für alle mögliche SPARQLs mit passenden Datentypen anwendbar ist, zeigt das Konzept auf jeden Fall eine hohe Produktivität. Diese Situation hat man auch bei den Software-Produktlinien (siehe Abschnitt 2.3). Auch dort ist am Anfang eine Investition notwendig aber mit der Anzahl der zu generierenden Produkte wird die Methode effektiver.

3.3.2.5 LD4UViz Client

Mit der Speicherung der gewünschten Diagramme ist die Aufgabe des LD4UViz Admins vollbracht. Nun kann jeder Client mit der entsprechenden Query auf ein vordefiniertes Diagramm in diesem Framework zugreifen und das Diagramm darstellen lassen (Client interagiert in diesem Fall mit dem Framework als Benutzer). Zuvor muss der Benutzer folgende Schritte durchführen:

1. Es existiert ein Javascript Datei, namens *ld4uviz.js*. Diese muss importiert werden (siehe Zeile 3 im unteren Code).
2. Die Library dieser Datei soll initialisiert werden (siehe Zeile 10 im unteren Code).
3. Das Diagramm wird mit den *draw* Methoden der Library gezeichnet (siehe Zeile 23 im unteren Code)

Vorteil an dem ganzen ist, dass der Benutzer sich nicht um die Skripten und um das Zeichen des Diagramms kümmern muss, sondern nur um das Aufrufen der *drawV2* Funktion.

Listing 3.8: Clientenseitige Library - ld4uviz.js

```
1 <html>
2 <head>
```

```
3  <script src="js/ld4uviz.js"></script>
4  </head>
5  <body>
6  ...
7  <div id="hierZeichneIch"></div>
8
9  <script>
10 LD4UViz.init();
11 var sparql= "PREFIX dbo: <http://dbpedia.org/ontology/>
12 PREFIX dbo2: <http://dbpedia.org/property/>
13 SELECT ?nam,?country,?total WHERE {
14   ?person foaf:name ?nam.
15   ?person dbo:birthPlace ?place.
16   ?place dbo:country ?country.
17   ?country dbo2:populationEstimate ?total
18 }
19 ORDER BY ?nam
20 LIMIT 100";
21
22 function myDraw(){
23   LD4UViz.drawV2("hierZeichneIch", {title:"My Diagram"}, sparql, 0, "/viz");
24 }
25 </script>
26 <input type="button" id="button" value="Draw" onClick="javascript:myDraw();">
27 </body>
28 </html>
```

Betrachten wir die Zeile 23 etwas genauer an. Daraus ist ersichtlich, dass der Client für die Funktion *drawV2* folgende Informationen bereitstellen sollte:

1. Platzhalter (DIV) für das Diagramm
2. Konfiguration des Diagramms
3. SPARQL-Query
4. Index des gewünschten Diagramms (es kann sein, dass in der Datenbank mehrere Diagramme für dieselbe Query definiert wurden)
5. Host

In der Libray List befinden sich alle *css* und *js* Bibliotheken, die ein Diagramm benötigt. Sobald ein neues Diagramm hinzukommt, muss diese Liste um die neuen Bibliotheken erweitert werden. Ein Beispiel für den Inhalt dieser Liste ist unten angegeben.

Listing 3.9: Datenstruktur fürs dynamische Laden der Javascript und CSS Bibliotheken

```
1 libraryList : [
2 {name:"Google charts", link : "https://www.google.com/jsapi",
3   type:"text/javascript", objName:"script"},
4 {name:"Cloud CSS", link:"css/jqcloud.css", rel: 'stylesheet',
5   type:"text/css", objName:"link"},
6 {
7 ]
```

Die Konfiguration (Höhe, Breite, Title) des Diagramms kann auf zwei Arten festgelegt werden:

- Clientseitig: Benutzer kann die Konfigurationsparameter an *drawV2* selbst eingeben (Es kann auch nur eine Parameter sein).
- Serverseitig: Die Defaultwerte des Diagramms in *repository.xml* werden übernommen

Wenn der Benutzer die nötigen Vorkehrungen für eine Diagrammanzeige getroffen hat, passiert folgendes:

- Client schickt die SPARQL-Query an den LD4UViz Server. Dort wird der Service $getVisualization$ aufgerufen
- Server holt die Ergebnisse von *dbpedia.org* ab
- Server findet heraus, welche Diagramme er anbieten kann und holt die dazugehörigen Mapping-Regel und die Generator(en)
- Generator(en) generiert den Code zur clientseitigen Darstellung von Diagrammen und schickt diesen Code an den Server
- Clienseitige Library zeichnet das Diagramm.

Der an dem Server ankommende Code ist ein fertiger Javascript-Code. Dieser Code ist vollständig und benötigt keine zusätzliche Routinen zum Zeichnen des Diagramms. Bei der clientseitige Library wird der Code durch die Javascript Funktion *eval* interpretiert und das fertige Diagramm angezeigt. Somit braucht der Client sich um gar nichts zu kümmern, außer dem gelieferten Code der Funktion *eval* zu übergeben.

3.3.2.6 Detailliertes Design

Mapping Strategien Um eine generische Lösung für das Mapping zwischen Properties und Diagramm-Achsen zu realisieren, müssen mehrere Varianten dieses Mappings im Betracht gezogen werden. Die Tabelle 3.5 fasst diese Varianten zusammen.

Die Tabelle zeigt, dass die Beziehungen zwischen Mapping-Varianten und Achsen-Konfigurationen für alle Kombinationen realisierbar sind. Es ist jedoch wichtig zu erwähnen, dass die aktuell vorhandenen Generatoren im Prototyp die Option, komplexes Mapping durchgeführt mit beliebig erweiterbaren Achsen, nicht unterstützen. Der Prototyp ist trotzdem aber nicht auf diese Konfiguration beschränkt, da die Diagramm-Generatoren einfach integrierbar sind bzw. die bestehenden Generatoren einfach zu erweitern sind.

Einfaches Mapping mit fixen Diagramm-Achsen entspricht einem direkten Mapping einer Property auf eine Achse, wobei ihre Datentypen und Semantiken identisch sind (1:1 Mapping bzw. statisches Mapping). Wie in der Tabelle 3.5 zu sehen ist, gibt einige Mapping-Varianten mit unterschiedlichen Achsen-Konfigurationen. Im Folgenden wird näher darauf eingegangen.

Dynamische Achsen Unter den ausgesuchten Diagramme gibt es das eine oder andere, die neben fixe Achsen auch optionale oder beliebig erweiterbare Achsen haben.

Mapping/Konfiguration	Fixe Achsen	Dynamische Achsen
Statisch	Ja	Ja
Dynamisch	Ja	Ja
Komplex (EIP)	Ja	Nein

Tabelle 3.5: Mapping Varianten

Die optionalen Achsen dienen der spezialisierten Darstellung des jeweiligen Diagramms. Sie können gemappt werden, müssen aber nicht. Ein gutes Beispiel dafür bietet JQCloud.

JQCloud kann hauptsächlich dafür eingesetzt werden, um die Suche nach einer bestimmten Person auf die Kategorien (Artist, Scientist, Journalist usw.) zu beziehen. Die dafür notwendige SPARQL-Query könnte folgend aussehen. Gesucht wird anhand des Namens der Person:

Listing 3.10: SPARQL Beispiel für JQCloud

```
1 SELECT distinct ?n ?t ?o (count(distinct(?s)) as ?c)WHERE {
2                 ?s ?p ?o .
3                 ?s rdf:type ?t.
4                 ?t rdfs:label ?n
5                 FILTER ( bif:contains(?o,'mustafa') )
6             }
7 ORDER BY DESC(?c) LIMIT 10
```

Die obige SPARQL-Query liefert den Namen, die Kategorie, den Link zu dieser Kategorie und die Anzahl, der in dieser Kategorie unter diesen Namen vorkommenden Personen. Sie alle werden somit im LD4UViz Admin unter den Key „Properties“ zum Mappen zur Verfügung gestellt. Fix auf die Achsen des JQClouds werden in diesem Fall die Kategorie und die Anzahl der Personen gemappt. Der Grund dafür ist, dass in dem dazugehörigen XML-Dokument des JQClouds „entry“ und „count“ als fixe „groups“ vordefiniert sind, da sie für eine Basic Darstellung vollkommen ausreichen. Optionale Achse ist nur der Link der jeweiligen Kategorie, die für eine spezialisierte Darstellung des Diagramms hergenommen werden kann.

Ein Beispiel für ein Diagramm mit beliebig großer Anzahl an Achsen bietet Google Table Chart. Google Table Chart ist ein flexibles und besonders benutzerfreundliches Diagramm. Die Achsen dieses Diagramms lassen sich clientseitig beliebig oft duplizieren, wodurch der Administrator die Gelegenheit bekommt, wenn sinnvoll, alle Ergebnisse zu mappen.

Dynamisches Mapping Die fertig generierten Diagramme können, wie schon mehrmals angegeben, anhand ihres Mappings gespeichert werden. Die Aufgabe des LD4UViz Frameworks endet nicht mit der Generierung der Diagramme, sondern diese Diagramme sollten auch wiederverwendbar sein.

Die Speicherung des Diagramms bedeutet, dass das ausgeführte Mapping in einem entsprechenden XML-Dokument gelegt wird. Der Inhalt dieses Dokuments beinhaltet u.a. die eingegebene SPARQL-Query, die als Suchkriterium dient. Da das Dokument nur eine

endliche Zahl an Diagrammen und somit an Queries beinhalten kann, soll eine Möglichkeit geben, bei Queries, die sich nur um eine Variable unterscheiden, z.B. ein anderer Name (siehe obige Query), auch das passende Diagramm zu erhalten. Somit kann man mit variablen Queries besser umgehen und es muss nicht bei jeder kleiner Änderung ein neues Diagramm generiert werden.

Als erstes wird für die Query, die ein potenzieller Kandidat für ein dynamisches Mapping ist, ganz normal ein Diagramm generiert nur mit der Ausnahme, dass die entsprechende Variable vor der Speicherung mit *@@@DYNAMIC@@@* annotiert wird. Die Query, die vom Benutzer eingegeben wird, würde sich somit nur um diesen Teil von der Query in der Datenbank unterscheiden. Bei einer Suche wird die ankommende Query mit genau dieser Query verglichen und wenn alle Teile dieser Query mit der ankommenden übereinstimmen, wird das passende Diagramm gefunden, mit dem entsprechenden Inhalt befühlt und angezeigt.

Der ganze Vorgang des dynamischen Mappings ist zum besseren Verstehen auch als Kontrollflussdiagramm (siehe Abbildung 3.16) dargestellt. Es wird im Kapitel 3.3.2.5 näher darauf eingegangen, wie auf die gespeicherte Diagramme zugegriffen werden kann. Hier soll nur eine kleine Einführung gegeben werden um das Dynamische Mapping zu erläutern.

Komplexes Mapping Ein einfaches Szenario für das komplexes Mapping wäre die Situation, wo die SPARQL-Ergebnisse ein Format des Datums liefern, das nicht zu dem erwarteten Datum der Diagramm-Achse passt. Das Beispiel dazu kann wie folgend definiert werden:

Listing 3.11: Formatierung des Datum-Datentyps

```
1  Format des SPARQL Property: MM-DD-YYYY
2  Format der Achse: DD.MM.YYYY
```

Mit einem 1:1 Mapping wäre die Erstellung des Diagramms mit diesen Ergebnissen nicht möglich.

Ein anderes Szenario ist das Mapping von mehreren Properties auf eine einzige Achse des Diagramms. Das ist besonders sinnvoll, wenn die kombinierten Ergebnisse einer Achse weiter eine Aktion auslösen (z.B. sie werden als Parameter zu einer JavaScript Funktion weitergegeben).

Diese Herausforderung überschreitet auch die Möglichkeiten des dynamischen Mappings, da hier die Interoperabilität zwischen Datentypen von Properties und Achsen hergestellt werden muss.

Methode Für die Interoperabilitätsprobleme gibt es eine Reihe von existierenden Lösungen bzw. Patterns, die in die Kategorie Enterprise Integration Patterns gehören (siehe Kapitel 3.2.2). Für die aktuelle Aufgabenstellung ist das Message Adapter Pattern ausreichend (siehe Abschnitt 3.2.2).

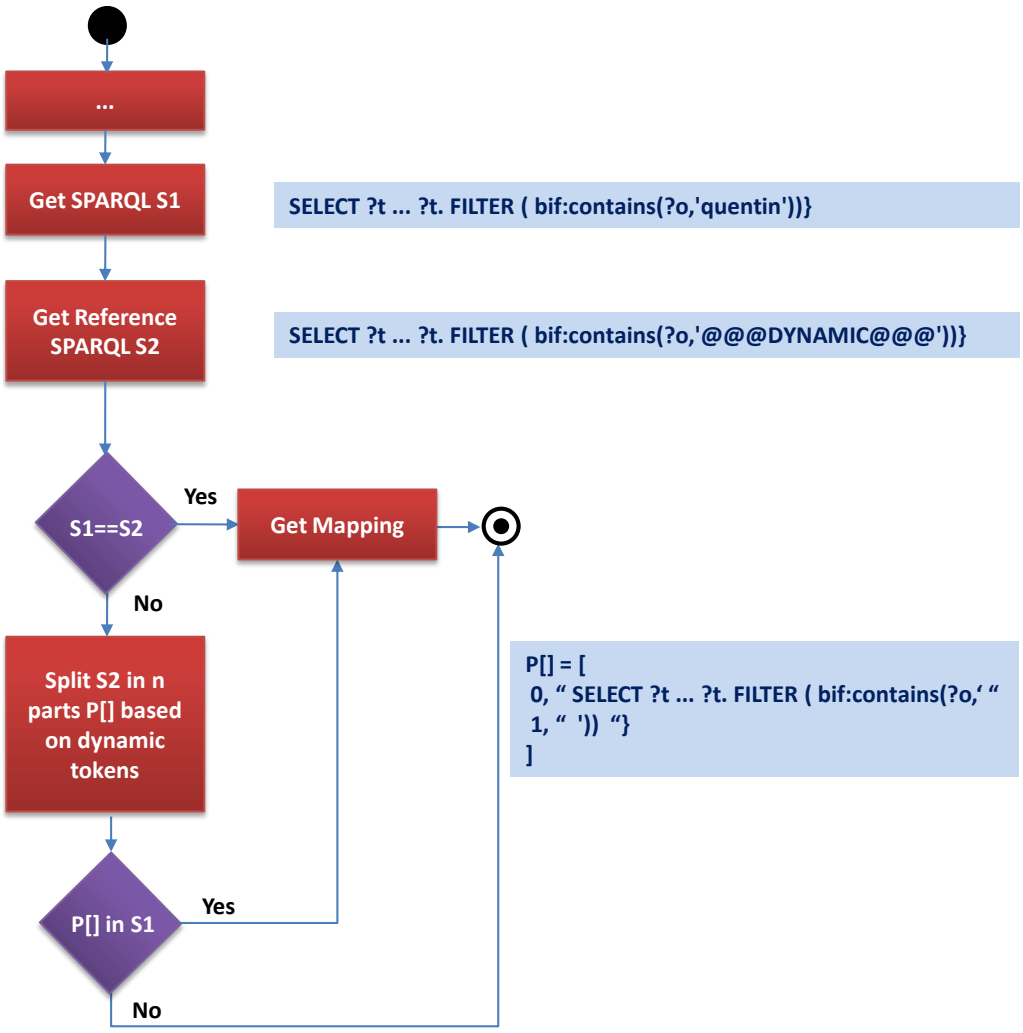

Abbildung 3.16: Kontrollflussdiagram für das dynamische Mapping

Realisierung Es stellt sich jetzt die Frage, wie der Adapter realisiert werden soll. Eine Möglichkeit wäre, für jedes Diagramm den Adapter in dem jeweiligen Generator zu integrieren und somit das Mapping statisch zu halten. Diese Lösung würde funktionieren, aber aus der Benutzer-Sicht könnten nur bestimmte Mapping-Relationen unterstützt werden (z.B. wenn der Generator einen Adapter für Datum hat, kann der Benutzer ein anderer Datentyp nicht angeben, oder mehrere Properties auf eine Achse mappen). Daher muss auch der Benutzer die Möglichkeit haben, das komplexe Mapping nach seinem Wunsch zu spezifizieren.

Eine pragmatische Lösung wäre folgende: Aus dem Grund, dass *Python* die Möglichkeit anbietet, den Python-Code als String zur Laufzeit zu interpretieren, könnte diese Fähigkeit ausgenutzt werden, um den Benutzer die Möglichkeit zu geben Adapter selbst zu implementieren und diesen anschließend auf dem Server zu interpretieren. Dieser Vorgang ist im Abbildung 3.17 dargestellt. Da diese Funktionalität ein Sicherheitsproblem

verursachen kann, kann der Datamanager um das Problem zu entgehen in einem Userprozess mit wenigen Privilegien gestartet werden (nur lesen und schreiben der XML-Dateien).

Der Prozess startet mit der Definition vom Mapping. Hier verwendet der Benutzer so-

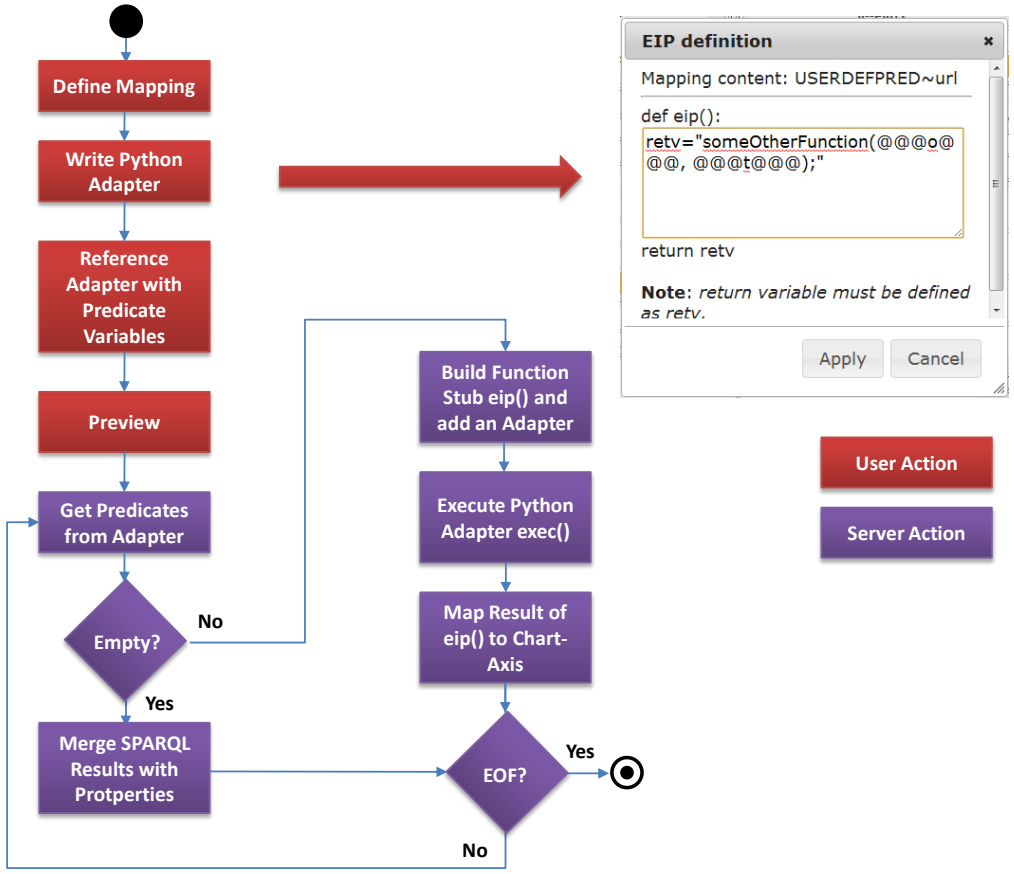

Abbildung 3.17: Kontrollflussdiagram für das komplexe Mapping

genanntes User Defined Mapping, das ihm erlaubt, mehrere Properties an eine Achse zu binden. Danach schreibt der Benutzer einen Adapter in Python-Code. Ein Beispiel ist auf der rechten Seite der Abbildung dargestellt. Das Konzept schreibt vor, dass der Benutzer den Code in Rahmen der Funktion *eip()* schreibt. Diese Funktion liefert die Variable *retv* zurück. Dieser Returnwert wird wiederum vom Server verwenden, um den Inhalt mit der entsprechenden Achse zu mappen.

Der Benutzer kann unbeschränkt den Python Code schreiben, jedoch kann er keine zusätzliche Pakete einbetten (außer wenn der Codegenerator derartiges anbietet). Er ist auch verpflichtet, alle seine Berechnungen in String-Variable namens retv zu speichern, da sie wie bereits erwähnt, auf die Achse gemappt wird.

Im nächsten Schritt werden die Properties im Adapter referenziert. Das wird durch spezielle Platzhalter (wie beim Codegenerator, siehe Abschnitt 3.3.2.4) eingesetzt. Das Muster

zur Referenzierung ist:

Listing 3.12: Definition des Platzhalters

```
@@@<PREDICATE_NAME>@@@
```

So wird beispielsweise die Property mit dem Namen „t“ wie folgend referenziert:

Listing 3.13: Beispiel für den Platzhalter

```
@@@t@@@
```

Als letzte Benutzeraktion steht die Darstellung des Diagramms bevor. Wenn der Server diese Spezifikationen bekommt, führt er das Mapping zur Laufzeit durch, und zwar auf folgende Art und Weise:

- Extrahieren alle annotierten Properties (weil es nicht unbedingt notwendig ist, Properties im Mapping zu haben).
- Falls die Properties im Adapter vorhanden sind, werden sie mit aktuellen Ergebnissen vom SPARQL-Query ergänzt. Somit ist der Adapter-Code fast vollständig und ohne Platzhalter.
- Adapter-Code wird durch das Funktion-Stub ergänzt. Ab dieser Stelle kann der Code ausgeführt werden.
- Der Code wird ausgeführt und die Funktion *eip()* wird im Scope vom entsprechenden Generator bekannt gemacht.
- Ergebnisse der *eip()* Funktion werden an die entsprechende Achse gemappt.
- Falls keine neue Datensätze im SPARQL-Ergebnis vorhanden sind, wird der Prozess abgeschlossen und die Generierung des Diagramms erfolgt.

3.3.3 Prozess Sicht

Im Kapitel 3.3.2.4 wurde schon erwähnt, dass vom LD4UViz Server angebotene Services clientseitig verwendet werden um gewisse Prozesse beim Server durchzuführen.

Für die Ausführung dieser Prozesse benötigt der Server vom Client entweder nur bestimmte Parameter oder auch ganze Pakete (JSON-Code). Da der Client mit GET oder POST Befehle mit dem Server interagiert und ein JSON-Code nicht einfach als POST-Parameter geschickt werden kann, muss er vorerst einmal kodiert werden. Dafür ist die Funktion $JSON.stringify$ zuständig:

Listing 3.14: Client: Kodierung des JSON-Codes

```
$.post("/viz", { cmd: "saveChart", sparql: sparqlContent ,
                 mappingArray: JSON.stringify(globals.mappingArray),
                 cachedChartName:globals.cachedChart },
```

Die Kodierung wird beim Server aufgehoben und der Inhalt des JSON-Codes mit Hilfe der Funktion $json.loads$ gelesen.

Listing 3.15: Server: Dekodierung des JSON-Codes

```
1 if(cmd=="saveChart"):
2         try:
3             sparql=request.POST['sparql']
4             cachedChartName=request.POST['cachedChartName']
5             mappingArray=json.loads(request.POST['mappingArray'])
```

Es existieren zusätzlich zu den im Kapitel 3.3.1 erwähnten Services

- getVisualisation
- getPreview
- getCharts

folgende Services:

- querySparql
- saveChart
- getRules
- deleteRule

Im Folgenden werden diese Services genauer erläutert.

getVisualization Dieser Service dient dazu für die eingegebene SPARQL-Query alle zugehörigen Diagramme zu erhalten. Folgendes geschieht nach dem serverseitigen Erhalt dieses Services (der Prozessablauf ist in Abbildung 3.18 als Sequenzdiagramm dargestellt):

- Sobald der Server startet, initialisiert die Methode *init* in der Klasse *DataManager* den Inhalt aller XML-Dokumente (siehe Tabelle 3.3) und gibt sie alle in einem JSON-Object.
- Server holt mit Hilfe des SPARQL-Wrappers die Resultate (*dataResults*) aus der DBpedia und ruft anschließend die in der Klasse *InProcessor* definierte Methode *process* auf. Diese Methode bekommt beim Aufruf die Parameter *dataResults* und *sparql*. Die Klasse *InProcessor* dient dazu, falls notwendig, die Resultate der Query an dem vorgegebenen Format anzupassen.
- Die Klasse *InProcessor* ruft mit den gleichen Parametern die Methode *process* in der Klasse *OutProcessor* auf. *OutProcessor* ist dazu da, um eine Instanz des entsprechenden Generators anhand *Chartname* zu erzeugen. Dazu später mehr.

  OutProcessor holt vom *DataManager* die Mappingtabelle, in dem er die Methode *getMappings* aufruft und ihr die SPARQL-Query als Parameter gibt. Die Methode *getMapping* und der Begriff *Mappingtabelle* spielen in diesem Framework enorm wichtige Rollen und werden als nächstes genauer erläutert.

  Unter dem Begriff ***Mapping*** wird die Fusion der Ergebnisse einer SPARQL-Query

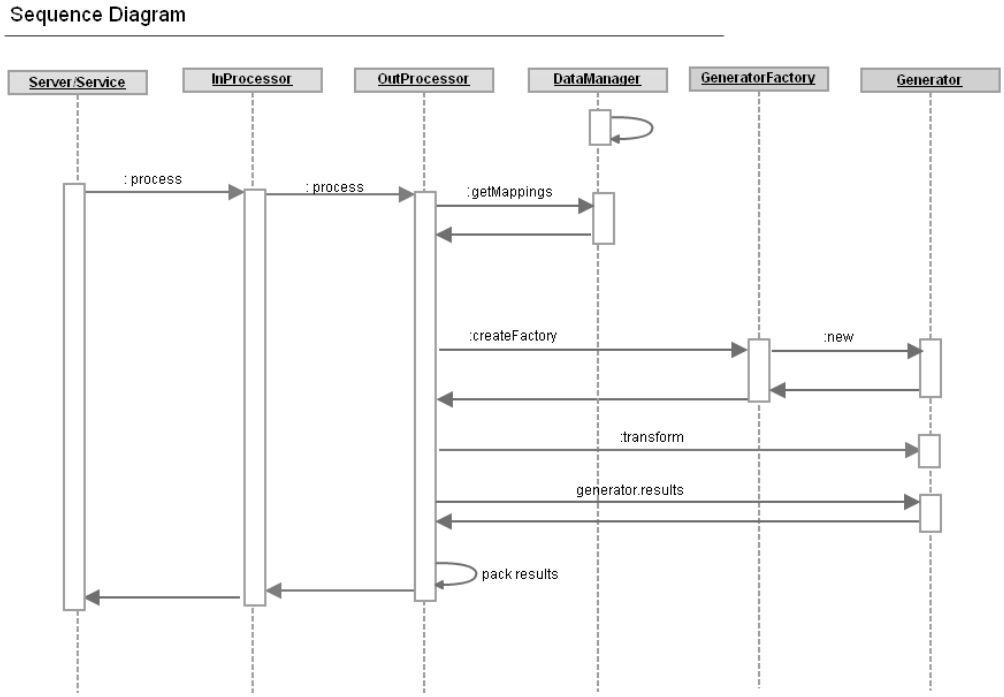

Abbildung 3.18: Service: getVisualization

mit den Achsen des jeweiligen Diagramms verstanden. Diese Fusion wird samt der Query und dem Namen des Diagramms in einem speziell für das Mapping angelegten XML-Dokument (*rules.xml*) gespeichert. Ein Ausschnitt aus diesem Dokument ist unten ersichtlich und wird im Folgenden erläutert:

Listing 3.16: rules.xml: Beispiel für ein fertig generiertes Mapping

```
1  <mappings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2  xsi:noNamespaceSchemaLocation="rules.xsd">
3    <mapping name="meinm">
4      <sparql name="searchsparql">
5  <![CDATA[SELECT ?country, ?total WHERE { ?person foaf:name ?nam.
6  ?person <http://dbpedia.org/ontology/profession> ?prof.
7  ?person <http://dbpedia.org/ontology/birthPlace> ?place.
8  ?place <http://dbpedia.org/ontology/country> ?country.
9  ?country <http://dbpedia.org/property/populationEstimate> ?total.
10 } LIMIT 100 ]]> </sparql>
11     <propertymap diagram="googlebarchart">
12       <prop label="Country" name="country">
13         <target name="x-axis"/>
14       </prop>
15       <prop label="Population total" name="total">
16         <target name="y-axis"/>
17       </prop>
18     </propertymap>
19     <propertymap diagram="googlepiechart">
20       <prop label="Country" name="country">
21         <target name="label"/>
```

```
22          </prop>
23          <prop label="Population total" name="total">
24            <target name="value"/>
25          </prop>
26        </propertymap>
27      </mapping>
28      <mapping>
29        ...
30      </mapping>
31  </mappings>
```

Das XML-Dokument *rules.xml* enthält alle durchgeführten und gespeicherten Mappings. Sie alle werden unter dem Element *mappings* zusammengefasst. Dieses Element hat wiederum für jede SPARQL-Query ein *mapping* Element. Alle Diagramme, die für diese Query kreiert worden sind, werden samt dem Namen im Element *propertymap* gespeichert (Name des Diagramms im Attribut *name*). Jede durchgeführte Fusion der Ergebnisse einer Query mit der entsprechenden Achse des ausgesuchten Diagramms wird im Element *prop* festgehalten. Zusammengefasst kann gesagt werden, dass

- *mappings* ein Wurzelelement ist, das pro SPARQL-Query ein *mapping* Element beinhaltet,
- die wiederum je nach Diagramm mehrere *propertymaps* hat
- die pro Fusion ein *prop* Element besitzen.
- Das ganze kann unter dem Begriff ***Mappinginformation*** zusammengefasst werden.

Wie bereits erwähnt, wird beim Starten des Servers das gesamte Inhalt des *rules.xml* initialisiert. Es ist nun die Aufgabe der Methode *getMappings* für eine Query die passende Mappinginformation aus dieser Menge zu extrahieren und sie dementsprechend in der *Mappingtabelle* anzubieten.

Die ***Mappingtabelle*** beinhaltet aber nicht nur die *Mappinginformation*, sondern auch den Sourcecode des jeweiligen Diagramms. Unter den von *DataManager* initialisierten und in einem JSON-Object verpackten XML-Dokumenten befindet sich auch *repository.xml*. Dieses Dokument beinhaltet die Sourcecodes und die Konfigurationen aller vom Framework unterstützten Diagramme und sind hier unter dem Namen des zugehörigen Diagramms gespeichert. Die Methode *getMappings* holt also neben der *Mappinginformation* auch den passenden Sourcecode und die Konfigurationen des Diagramms aus entsprechenden JSON-Object und zwar durch die Eingabe des *Chartname*. Somit ist sichergestellt, dass für die gegebene Query die richtige *Mappinginformation* und dadurch das richtige Diagramm und Sourcecode ausgewählt wird.

Mit dem Erhalt aller dieser wichtigen Details kann die ***Mappingtabelle*** aufgebaut werden und zwar in Form eines JSON-Objekts, namens *mappingObject*:

Listing 3.17: Mappingobject

```
1  mappingObject={'chart': nameOfChart, 'mapping': propertyMapsArray,
```

```
2                 'code': chartCode, 'configuration': chartConfig}
```

- Nach dem Erhalt der *Mappingtabelle* ruft *OutProcessor* die Methode *createFactory* beim *GeneratorFactory* auf und zwar mit den Parameter *nameOfChart*, *mappingInfo*, die er aus der Mappingtabelle extrahiert hat, und *result*.

  GeneratorFactory basiert auf Template Pattern, siehe 3.2.4, und liefert durch die Methode *createFactory* die Instanz vom Generator des eingegebenen Diagramms an dem *OutProcessor* zurück. Es existiert für jedes Diagramm ein Generator und es wird anhand *nameOfChart* entschieden, von welchen Generator die Instanz erzeugt wird.

- In den jeweiligen Generatoren findet die Fusion der Ergebnisse mit den Achsen des Diagramms statt. Anschließend wird diese Kombination an den variablen Stellen des entsprechende Sourcecodes platziert und der ausführbarer Code zur Verfügung gestellt. Die benannte Transformation findet in der Methode *transform* statt.

- *OutProcessor* ruft die oben benannte Methode für die Transformation auf und holt dich die Resultate dieser Methode (*generator.results*). Die Informationen, die aus dem Array *mappingInfos* und aus dem Ergebnis der Methode *transform* gewonnen werden, werden als Array unter dem Namen *resultArray* zusammengepackt und an *InProcessor* zurückgeliefert. Somit wird die Methode *process* von *InProcessor* zu Ende ausgeführt.

- Nun geht es darum, der Methode *process*, die vom Server aufgerufen wurde, das gewünschte Ergebnis zurückzuliefern. Also leitet *InProcessor* das *resultArray* an den Server weiter.

Listing 3.18: Result-Array

```
1 resultObject={'name':nameOfChart,'start':transformedResult['code'],
2                 'errors': transformedResult['errors'],
3                 'configuration':chartconfiguration}
4 self.resultArray.append(resultObject)
```

- Aus diesem Array wird in *ld4uviz.js* der Sourcecode, siehe 3.3.2.5, extrahiert und durch die Funktion *eval* schlussendlich ausgeführt. Das Diagramm wird clientseitig angezeigt.

getPreview Um ein Diagramm zu genieren, muss ein Mapping stattfinden und dieses anschließend verifiziert werden. Damit ein Mapping stattfinden kann, muss als erstes eine entsprechende SPARQL-Query definiert werden. Jeder SPARQL-Query besteht aus, *Subjekt*, *Prädikat* und *Objekt*. Das Element über das eine Aussage gemacht wird, wird als *Subjekt* bezeichnet. Das *Prädikat* bestimmt die Art und Ausprägung der Aussage. Das *Objekt* wiederum ist der Wert oder Gegenstand dieser Aussage [Tom12]. Ein *Objekt*, dessen Wert als Ergebnis auftauchen soll, steht in dem SELECT-Block der Query. Diese Objekte werden nach der Lieferung der Resultate serverseitig als Variable (*Properties*) in einer Liste aufgenommen und diese Liste clientseitig angezeigt. Diese Anzeige ist für das Mapping notwendig. Das ganze kann man sich folgendermaßen vorstellen:

Für die folgende Query:

Listing 3.19: Beispiel Query zur Beschreibung von Mapping

```
1 PREFIX dbo: <http://dbpedia.org/ontology/>;
2 SELECT ?mountain, ?el, ?lab WHERE {
3 ?place dbo:mountainRange ?mount.
4 ?mount dbo:country ?country.
5 ?mount rdfs:label ?mountain.
6 ?country rdfs:label ?lab.
7 ?place dbo:elevation ?el.
8 FILTER (?el > 8000)
9 }
```

liefert DBpedia folgende Resultate:

Listing 3.20: Resultate aus DBpedia

```
1 [{'mountain': 'Karakorum', 'lab': 'China', 'el': '8611'},
2  {'mountain': 'Mount_Everest', 'lab': 'China', 'el': '8848'},...]
```

Betrachten wir nun die Ergebnisse genauer. In der Query wurde angegeben, dass die Werte hinter den Objekten *mountain*, *el* (Größe des Berges) und *lab* (Name des Landes) als Ergebnis auftauchen sollen. Man definiert die Ergebnisse auch als „value“ und *mountain*, *el* und *lab* als „keys“. Da alle Ergebnisse in gleicher Format erscheinen, reicht es vollkommen nur das erste Ergebnis für die *Property*-Liste zu nehmen. Für die clientseitige Anzeige der *Properties* werden somit die Keys aus diesem Ergebnis extrahiert und sind anschließend bereit vom Benutzer im Mappingvorgang auf die Achsen des ausgewählten Chart gemappt zu werden.

Nach dieser ausführlichen Einführung kann man sich wieder dem schrittweisen Ablauf der Verarbeitung vom *getPreview* widmen. Für ein besseres Verständnis des Prozessablaufs von *getPreview* kann das Sequenzdiagramm 3.19 hergenommen werden.

- Wenn das Mapping ausgeführt und das fertige Mapping in ein Array, in Form von Property+Achse, gelegt wurde, kann der Service *getPreview* verarbeitet werden. Der LD4UViz Client kontaktiert den Server mit dem Command *getPreview* und übergibt ihm das fertig definiertes Mapping, *mappingArray*, die SPARQL-Query und den Namen des ausgewählten Chart.

- Server ruft mit diesen Parametern die Methode *putNewMapping* in der Klasse *DataManager* auf, in der folgendes passiert:

  Der aus allen XML-Dokumenten, *rules.xml*, *repository.xml* und der *current_chart.xml*, gelesene Inhalt ist unter entsprechenden Keys in einer Python Dictionary hinterlegt, das als eine Online Datenbank gesehen werden kann.

Listing 3.21: Python Dictionary für den Inhalt der XML-Dokumente

```
1 self.data={'specification': res4, 'rule':res2, 'repository': res3}
```

Der Inhalt des Array *mappingArray* wird extrahiert und so angepasst, sodass er die Darstellungsform von *'rule'* annimmt. Somit kann der Inhalt von *'rule'*, der fertige Mappings beinhaltet, um diesen vorübergehend erweitert werden. Es soll hier besonders betont werden, dass es sich bei dieser Erweiterung um eine vorübergehende Erweiterung von *self.data* handelt und an das *rules.xml* keine Veränderungen unternommen werden.

- Anschließend wird beim Server für diese Query die Ergebnisse aus DBpedia geholt.
- Ab hier verläuft es sowie beim Command *getVisualization*.
- Wenn das fertig generierte Diagramm angezeigt wurde, wird mit der Methode *reInit* in *DataManager* auf den ursprünglichen Zustand zurückgekehrt, was bedeutet, dass die vorübergehende Mapping aus dem Array *data* wieder gelöscht wird. Das gleiche passiert, wenn während des gesamten Verlaufs irgendein Fehler auftritt.

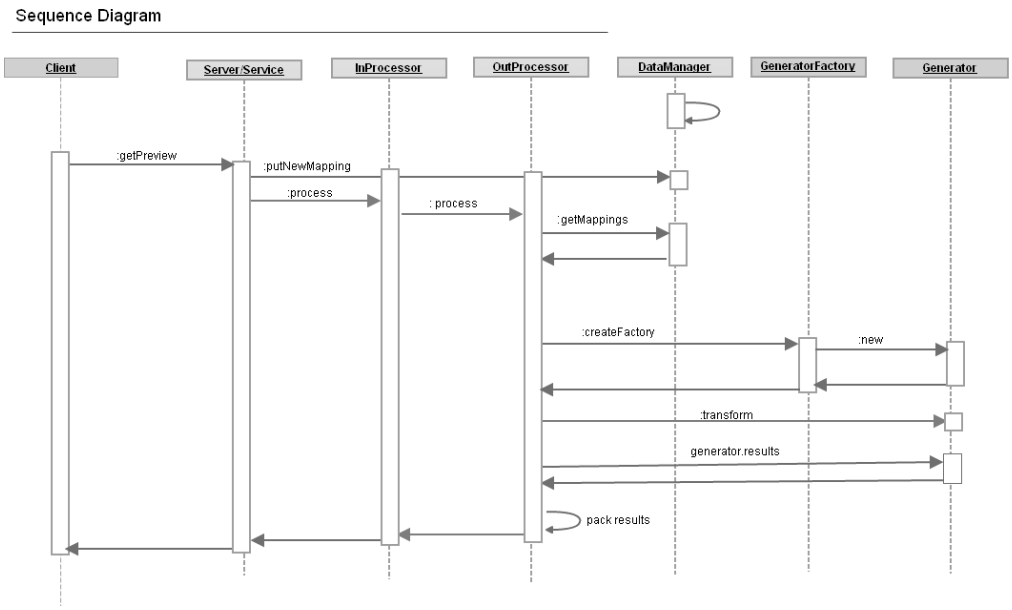

Abbildung 3.19: Service: getPreview, Sequenzdiagramm

Der gesamte Vorgang von *getPreview* ist in Abbildung 3.20 ersichtlich.

saveChart Wenn das generierte Diagramm auch gespeichert werden soll passiert folgendes:

- Der LD4UViz Client kontaktiert den Server mit dem Command *saveChart* und übergibt ihm ein fertig definiertes Mapping, *mappingArray*, die SPARQL-Query und den Namen des ausgewählten Chart.
- Server ruft mit diesen Parametern die Methode *putNewMapping* in der Klasse DataManager auf. Was in dieser Methode passiert wurde beim Service *getPreview* genau erklärt.

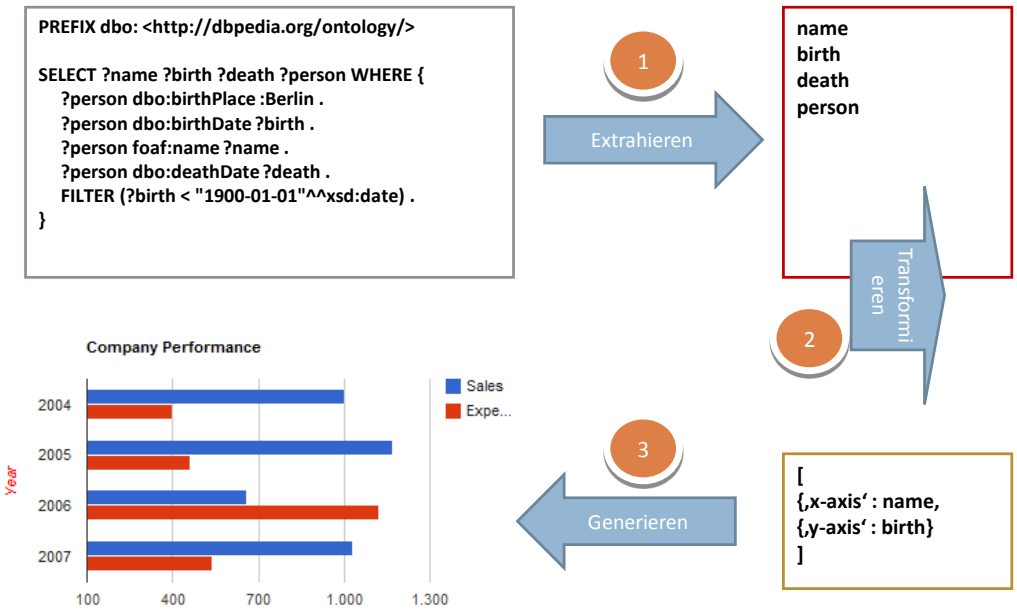

Abbildung 3.20: Service: getPreview

- In dieser Methode wird das fertige Mapping, wie schon oben beim *getPreview* bereits erwähnt, vorübergehend an der entsprechenden Stelle, *rule*, der Online Datenbank hinterlegt.
- Nachdem das passiert ist, wird vom Server die Methode *commitRules* in der Klasse *DataManager* aufgerufen, die Folgendes auslöst:
  Alte und neue Mappings werden aus der Online Datenbank extrahiert. Anschließend wird damit im Speicher eine DOM Struktur gebildet (muss dem Schema von *rules.xsd* entsprechen), die dann das existierenden XML-Dokument *rules.xml* ersetzt. Diese Operation ist äußerst wichtig, damit ein konsistenter Zustand zwischen der Online Datenbank und dem echten XML-Dokument herrscht.
- Das neue Mapping ist somit in *rules.xml* gespeichert.

deleteRule Um ein gespeichertes Mapping wieder zu löschen wird vom Client der Service *deleteRule* verwendet. Dabei wird vom Server die Methode *deleteRuleEntries* in *DataManager* aufgerufen.
Die Methode *deleteRuleEntries* löscht das clientseitig ausgewählte Mapping aus der Online Datenbank. Damit der aktuelle Inhalt der Datenbank auf das *rules.xml* übernommen werden kann, wird sofort danach die Methode *commitRules* aufgerufen. Mit der Methode *reInit* wird anschließend das aktuelle XML-Dokument initialisiert.

Die restlichen Services haben folgende Aufgaben:

getCharts listet alle vom Framework momentan unterstützten Diagramme auf.

querySparql stellt die Properties einer SPARQL-Query für Mappingzwecke zur Verfügung.

getRules listet den Inhalt des rules.xml pro ein Mapping auf.

3.4 Technologie

Für die Implementierung des Frameworks wurden folgende Technologien eingesetzt:

Technologie	Version	Beschreibung
Python	2.7.2+, Kompiliert mit GCC 4.6.1	Implementierungssprache
Python-Django	1.3.1	Plattform für LD4UViz Service
JavaScript	1.8.5	Implementierungssprache für den Client
DBpedia	3.7	Datenquelle
SPARQLWrapper	1.5.0	API für DBpedia
UBUNTU	11.10	Betriebssystem

Tabelle 3.6: Eingesetzte Technologien für die Entwicklung des LD4UViz Frameworks

3.5 System aus der Sicht des Clients als Administrator

Nach einer ausführlichen Beschreibung des LD4UViz Frameworks aus der Sicht des Entwicklers, geht es in diesem Kapitel darum zu erläutern, was der Client genau tut, wenn er das Framework als Administrator betätigt. Folgende Aussage wurde schon im Kapitell 3 gemacht und wird in diesem Kapitel in Schritten erläutert:

Client als Administrator: Wenn für eine eingegebene SPARQL-Query kein fertiges Mapping im Framework vorhanden ist, soll der Benutzer die Möglichkeit haben dieses selbst zu generieren.

Damit der Client die Ergebnisse einer SPARQL-Query visualisieren kann, muss er als erstes an diese Ergebnisse kommen. Das Framework ist so konzipiert, dass es ein Feld für die Eingabe der SPARQL-Queries enthält (siehe Abbindung 3.21). Nach der Eingabe der Query in diesem Feld und dem anschließenden Betätigen des Buttons „Get Properties“ wird diese Query an DBpedia weitergeleitet und die Ergebnisse serverseitig abgefangen. Im SELECT-Block der SPARQL-Query hat der Client die Variablen festgelegt, von denen er die Ergebnisse haben möchte. Genau diese Variable werden auf einer Liste aufgenommen und auf der Benutzeroberfläche als Property angezeigt. Es sind also genau die Ergebnisse hinter diesen Properties, die visualisiert werden sollen (siehe Abbildung 3.22).

Alle Diagramme, die vom Framework unterstützt werden, sind sofort beim Starten der Seite ersichtlich. Nachdem ein Diagramm ausgewählt wurde, erscheinen seine Achsen samt

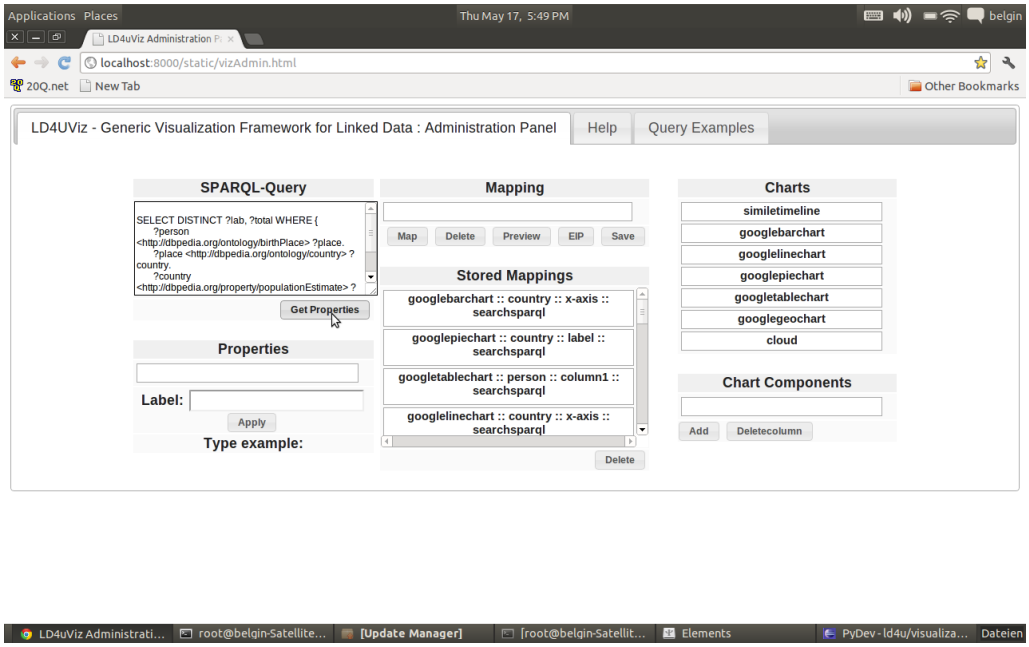

Abbildung 3.21: Eingabe einer SPARQL-Query

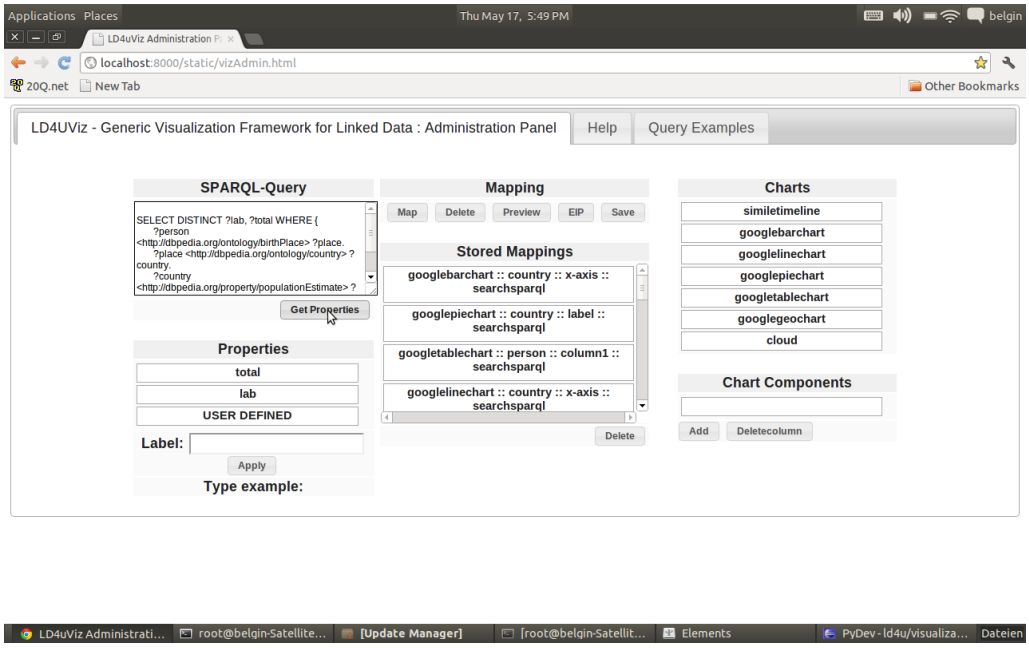

Abbildung 3.22: Ergebnisse einer SPARQL-Query

ihren Typen in einer Liste (siehe Abbildung 3.23). Die Typenangabe ist beim Mapping zu beachten, da ansonsten das Diagramm nicht richtig generiert werden kann. Um die richtige Property für eine Diagrammachse auszuwählen, muss auf die Property angeklickt werden, denn danach erscheint in dem dafür vorgesehenem Feld als Beispiel ein Ergebnis, die als

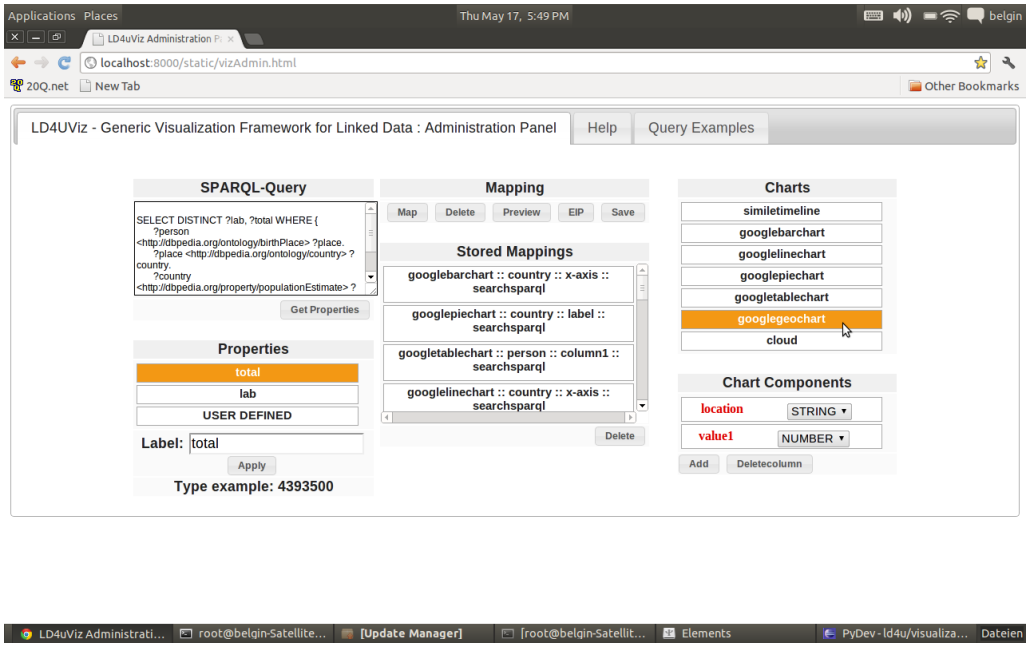

Abbildung 3.23: Komponenten (Achsen) eines Diagramms

Referenz für die richtige Auswahl dienen soll (siehe Abbildung 3.24). Beispiel: Wenn die Poperty als Wert „Austria“ enthält, ist klar, dass es dafür nur die Diagrammachse von Typ String in Frage kommt. Die ausgewählte Property kann für die Diagramm-Anzeige

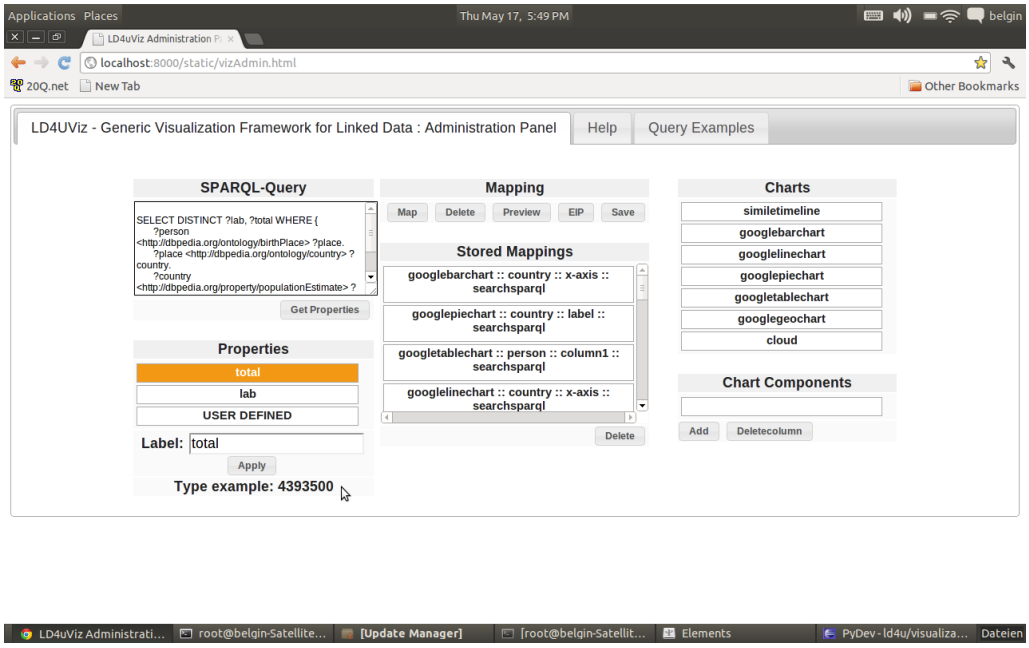

Abbildung 3.24: Der Wert hinter einer Property

auch individuell benannt werden (siehe Abbildung 3.25). Nun kann das Mapping stattfin-

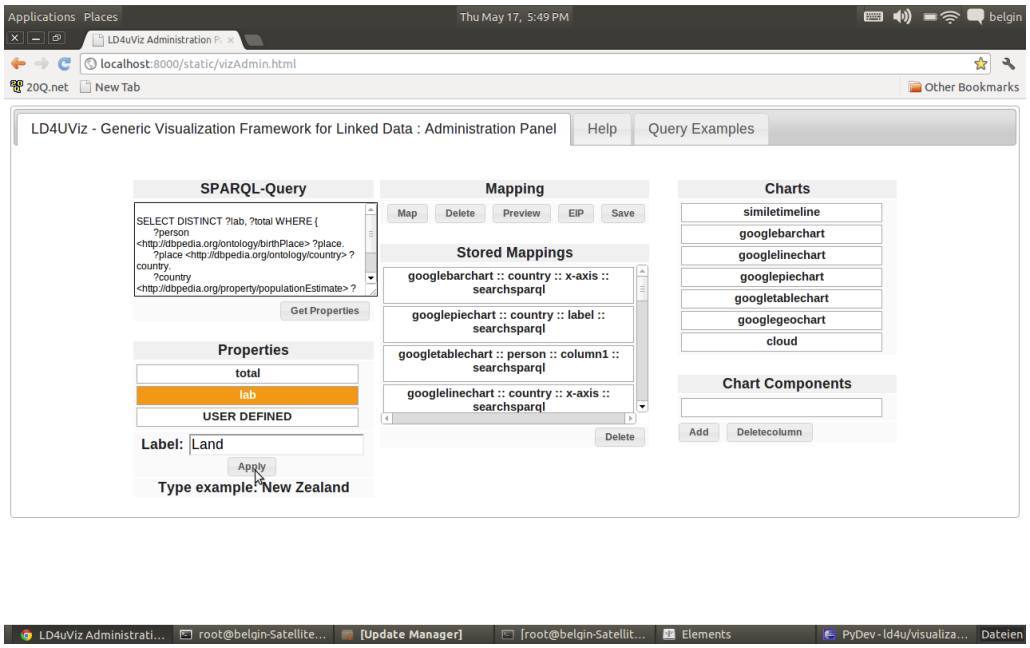

Abbildung 3.25: Umbenennung einer Property

den. Dafür wird eine Property und die entsprechende Achse ausgewählt und das „Map“ Button betätigt (siehe Abbildung 3.26). Somit ist fixiert, was genau auf den Achsen des

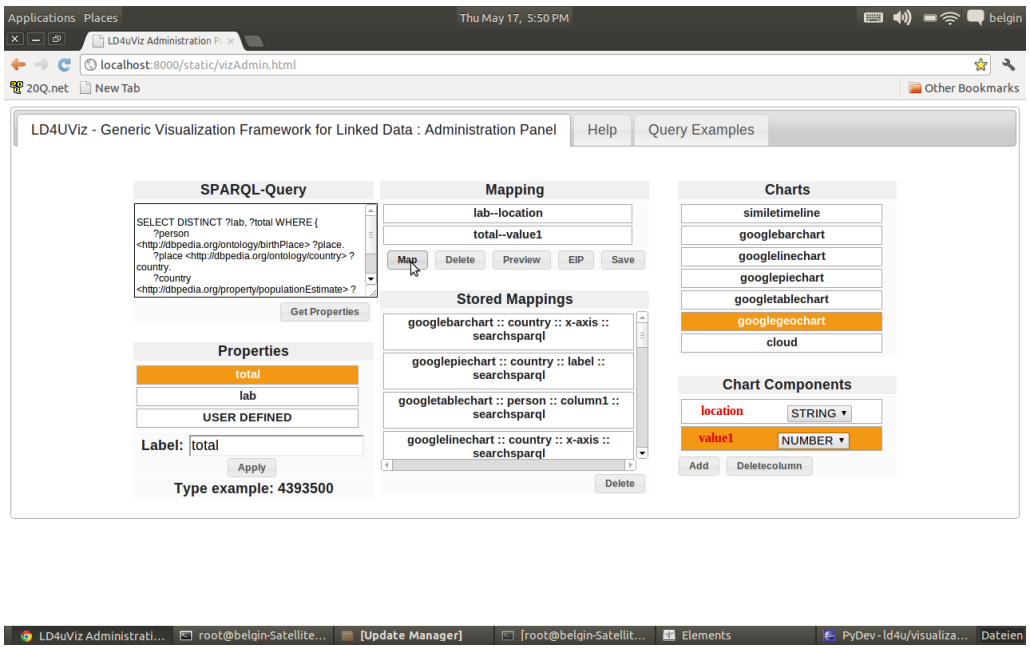

Abbildung 3.26: Ein fertig ausgeführtes Mapping

ausgewählten Diagramm angezeigt werden soll. Jedes Mapping kann auch einzeln wieder gelöscht werden.

Wenn ein gewünschtes Mapping nicht direkt möglich ist, da entweder eine Property mit einer Diagrammkomponente (Achse) nicht kompatibel ist oder, wenn mehrere Properties auf eine Komponente gemappt werden sollen, kommt EIP zum Einsatz. Dabei wird die entsprechende Achse des Diagramms auf die pivot Property „USER DEFINED“ gemappt, die nur zu diesem Zweck existiert. Wenn der Button „EIP“ angeklickt wird, öffnet sich ein Fenster in dem ein Python-Code eingegeben werden kann, der in dem jeweiligen Generator ausgeführt wird (siehe Abbildung 3.27). Nach der Ausführung entspricht diese Achse genau dem Returnwert dieses Codes, was für den Administrator erst nach der Erstellung des Diagramms sichtbar wird. Dadurch, dass im EIP-Fenster ein beliebiger Pythoncode

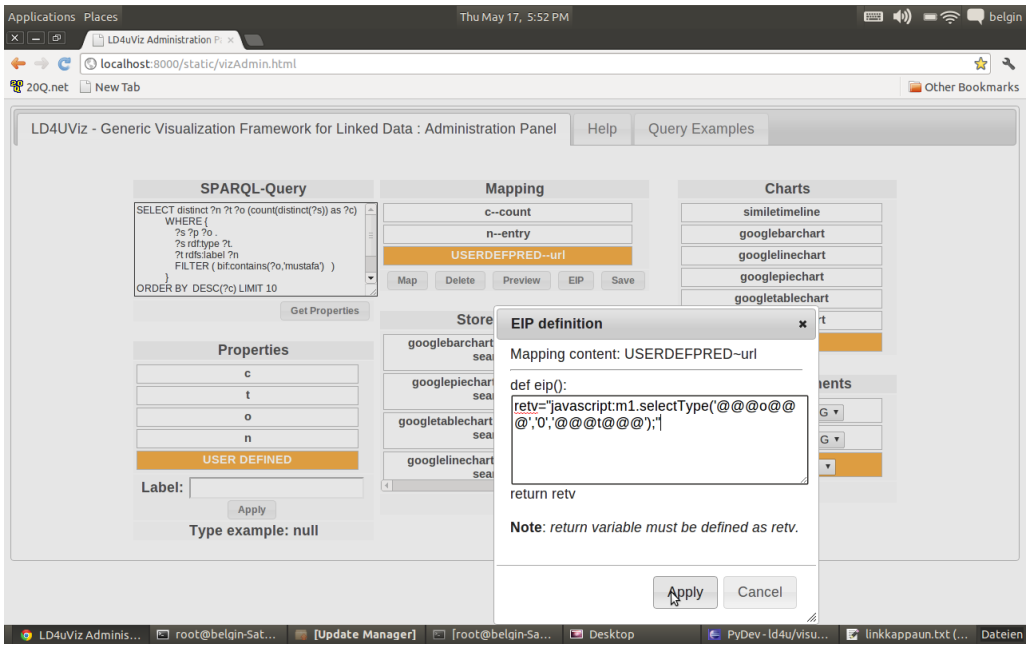

Abbildung 3.27: EIP-Funktionalität

eingegeben werden kann, ist das erreichen von Interoperabilität zwischen Properties und Komponenten auf verschiedene Wege realisierbar.

Nachdem der Mappingvorgang abgeschlossen ist, kann überprüft werden, wie das für dieses Mapping generierte Diagramm ausschaut (Preview) (siehe Abbildung 3.28). Für die späteren Zugriffe, wenn der Client als Benutzer mit dem Framework agiert, kann der Client das fertige Diagramm auch speichern (Save). Jedes der gespeicherten Diagramme wird auf einer Liste aufgenommen. Das Löschen eines Diagramms aus dieser Liste bedeutet das komplette Entfernen des Diagramms aus der Datenbank und kann jederzeit in Anspruch genommen werden (siehe Abbildung 3.29).

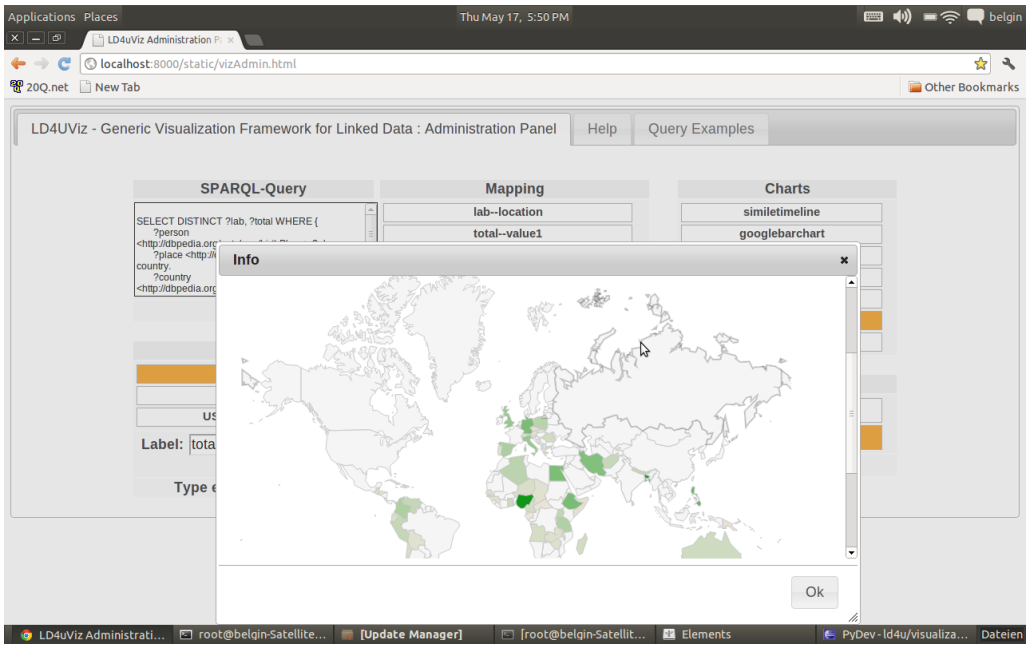

Abbildung 3.28: Preview: Anzeige eines fertig generierten Diagramms

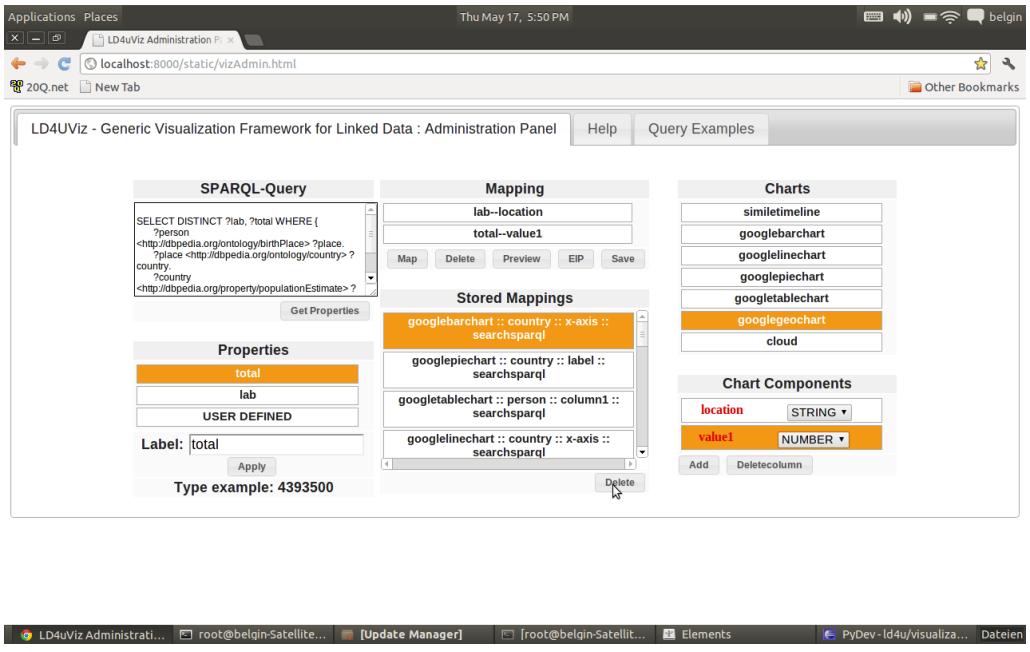

Abbildung 3.29: Löschen eines Diagramms aus der Datenbank

4 Evaluierung

In diesem Abschnitt wird der implementierte Prototyp gegen die Erwartungen -bzw. Anforderungen aus dem Abschnitt 1.2 evaluiert. Das Ziel dieser Evaluierung ist es, einen quantitativen Nachweis der Performance im Vergleich zu den existierenden State-of-Art Ansätzen zu zeigen. Da es derzeit keine Metriken gibt, ähnliche Ansätze quantitativ zu evaluieren, wird zuerst eine Methodik unter für diese Arbeit relevanten Annahmen definiert. Basierend auf die Methodik zur Evaluierung, werden einige Case-Studies definiert, deren Aufgabenstellung zur quantitativen Evaluierung beiträgt. Abschließend werden die Ergebnisse präsentiert, die eine wesentliche Rolle bei der Zusammenfassung dieser Arbeit spielen.

4.1 Methode zur Evaluierung

In einem Software-Qualitätsmodell wird die Menge der Attribute zusammengefasst, die eine quantitative Aussage über die Software aussagen. Die Software-Engineeering Institut (SEI) in ihrem technischem Report in [BLKW11] beschreibt Qualitätsattributen wie folgend:

„*Software quality is the degree to which software possesses a desired combination of attributes (e.g., reliability, interoperability)*“[BLKW11].

Sie stellen ein generisches Model (Taxonomie) vor, mit dem sich die Attributen Performance, Dependability, Safety und Security (obwohl die letzten zwei öfters als Dependability Attributen zusammengefasst werden) einheitlich beschreiben lassen. Somit wird jeder dieser Attributen durch drei Punkte beschrieben:

1. Einheiten (z.B. Latenzzeit bei Performance),
2. Ursache (wodurch wird es beansprucht) und
3. Methoden zur Optimierung.

Im Zuge der Evaluierung ist die Performance von Bedeutung. Sie wird für dieses Framework basierend auf den Lifecycle bestimmt und nicht, sowie in der Literatur angegeben, basierend auf das laufende System. Das Ziel der Evaluierung ist zu zeigen, dass die vorgestellte Methode den Entwicklungsaufwand reduziert und dass sie mit bestehenden Ansätzen vergleichbar ist bzw. um einige Argumente effizienter. Der Aufwand wird bezogen auf Lines Of Codes (s. andere Attributen in [BLKW11]) im Lifecycle bestimmt. Die restlichen Attribute wie Throughput, Capacity und Modes sind nicht für die Evaluierung relevant, da sie keinesfalls das Ziel beanspruchen (z.B. Skalierbarkeit zur Laufzeit).

Eine weitere interessante Qualitäts-Analyse wurde bereits im Abschnitt 2.3 vorgestellt. Bei dieser Analyse wurden Softwareproduktlinien mit den traditionellen Entwicklungsmethoden quantitativ verglichen (siehe Abbildung 2.17). Aus dem Grund, dass die Generierung der Diagrammen ähnliche Strategie verfolgt, wird die Performance-Messung auf dieselbe Methodik basieren: Messung des Aufwands bei der Entwicklung mit der traditionellen (M1) und der vorgestellten (M2) Methode.

4.1.1 Ablauf

Für jedes durchgeführte Experiment (siehe Abschnitt 4.2) wird für jede Methode die für die Zeichnung eines Diagramms benötigte LOC (Lines-of-Code) bestimmt. Als Input für jedes Experiment wird ein SPARQL angegeben. Die Experimente werden in drei Serien durchgeführt, sodass ein zuverlässiges Ergebnis zustande kommt. Idealerweise sollte jede der Serien von unabhängigen Personen (erfahrenen Entwickler) durchgeführt werden. Aus dem Grund, dass dies für M1 nicht realisierbar ist, werden für diese Methode einige Annahmen getroffen (siehe Abschnitt 4.1.2).

4.1.2 Annahme

Für die quantitative Analyse vom M1 eignet sich, wie bereits erwähnt, am besten die LOC-Messungen. Diese Größe ist als einzige vorhanden und dadurch messbar. Proportional zum Wert dieser Größe kann die Entwicklungszeit abgeschätzt werden. Da diese Abschätzung ziemlich ungenau wäre, wird die Evaluierung allein auf LOC-Abschätzungen basieren. Die Tabelle 4.1 zeigt die LOC-Messungen von diesem Framework. Bei der Durchführung der

Komponente	Leerzeichen	Kommentare	Code
Server	732	479	1683
JS-Library	18	17	100
VizAdmin	161	12	1127
VizClient	62	12	269
TestClient	8	12	19
XML-Datenbank	88	151	488
Gesamt	**1069**	**683**	**3686**

Tabelle 4.1: LOC-Messung vom Prototyp (mit dem Tool-Cloc)

Aufgaben (Experimenten) werden auch die Leerzeichen mitgezählt. Somit hat der gesamte Prototyp 5438 LOC.

4.2 Experiment

In Rahmen dieser Evaluierung werden vier Aufgabenstellungen jeweils für M1 und M2 mit steigender Komplexität definiert. Jeder der Aufgaben umfasst einen kurzen Entwicklungsprozess, dessen Durchführungsaufwand in unserem Fall das Maß für die Performance ist. Da bei allen Aufgaben für M1 ein Setup von verwendeten Technologien durchgeführt

werden muss, wird dieser Teil ausgeschlossen, weil es dem LOC nicht beiträgt. Dieses Setup beinhaltet die Vorbereitung von Python-Django Web-Framework und entsprechendem Service zur Verarbeitung von Abfragen (weil die Verbindung mit DBpedia über Python-basierten SPARQL-Wrapper hergestellt wird).

4.2.1 Case-Study 1: Einfaches Mapping

Es soll ein Liniendiagramm erstellt werden, der die Länder und ihre Bevölkerungszahl auf seine Achsen abbildet. Die Daten sollen aus folgender SPARQL-Query abgefangen werden. Es sollen die Diagramme, die bereits in LD4UViz Framework vorhanden sind, verwendet werden.

Listing 4.1: SPARQL-Query für ein einfaches Mapping

```
1 SELECT ?country, ?total WHERE {
2 ?person foaf:name ?nam.
3 ?person <http://dbpedia.org/ontology/profession> ?prof.
4 ?person <http://dbpedia.org/ontology/birthPlace> ?place.
5 ?place <http://dbpedia.org/ontology/country> ?country.
6 ?country <http://dbpedia.org/property/populationEstimate> ?total.
7 }
8 LIMIT 100
```

Methode M1 Die Lösung dieser Aufgabe resultierte in einer clientseitigen Implementierung. Auf der Serverseite liefert der Service die SPARQL-Ergebnisse unverändert in DBpedia Format an den Client. Auf der Clientseite wird einerseits die Routine für die Initialisierung und Darstellung des Diagramms realisiert und andererseits wird die Funktionalität zum Mapping von SPARQL-Ergebnissen an Format des Liniendiagramms realisiert. Insgesamt hat M1 99 LOC benötigt.

Methode M2 Da bereits Google Line Chart in LD4UViz-Repository vorhanden ist, kann es für die SPARQL wiederverwendet werden. Da aber die SPARQL noch für das Framework unbekannt ist, muss zuerst ein Mapping definiert werden. Daher wird die Implementierung in folgenden zwei Schritten durchgeführt:

- Es wird LD4UViz-Admin verwendet, um das Mapping zu definieren (siehe Abschnitt 3.3.2.2)
- Der Client wird mit LD4UViz-Library initialisiert und die Standard-Routinen dort ausgeführt (siehe Abschnitt 3.3.2.5)

Das Mapping kann in LOC nicht gemessen werden, da es keine neuen Code-Zeilen hervorruft. Für die Clientseite werden immer nur 6 LOC benötigt.

4.2.2 Case-Study 2: Komplexes Mapping

In diesem Beispiel wird nicht mehr 1:1 Mapping durchgeführt, sondern eine komplexere Beziehung zwischen SPARQL-Properties und Achsen realisiert. Das unten dargestellte SPARQL liefert die Kategorie eines Begriffes „o“ (in diesem Fall Person) einmal als Literal „n“ und einmal als Ressource „t“ , sowie die Anzahl dieser Begriffe innerhalb dieser

Kategorie.

Es soll ein JQCloud Diagramm verwendet werden, um die untere SPARQL zu visualisieren. Standard-Interface jedes Eintrags schaut so aus: `<text, weight, link (optional)>`. Es sind daher folgende Regeln zu beachten:

- Das Diagramm soll die Kategorien auf *text* mappen und die Anzahl der Begriffe auf *weight*
- Beim Attribut link wird ein 2:1 Mapping durchgeführt, sodass „o“ und „t“ wie folgend kombiniert werden: „javascript:callMyInternalFunction(`<o>, <t>`). Das heißt, für jeden Eintrag aus SPARQL-Ergebnissen wird *link* auf eine interne JavaScript Funktion des Clients referenziert. Somit kann man eine zusätzliche Verarbeitung auf dem Client unternehmen (z.B. anhand des Namens und der Kategorie ein anderes SPARQL definieren, um die Suche zu verfeinern).

Listing 4.2: SPARQL-Query für ein komplexes Mapping

```
1 SELECT distinct ?n ?t ?o (count(distinct(?s)) as ?c)
2             WHERE {
3                 ?s ?p ?o .
4                 ?s rdf:type ?t.
5                 ?t rdfs:label ?n
6                 FILTER ( bif:contains(?o,'mustafa') )
7             }
8 ORDER BY DESC(?c) LIMIT 10
```

Methode M1 Für die Realisierung dieser Aufgabenstellung wurde auch auf Serverseite einiges gemacht. Es ist nämlich so, dass die Ergebnisse vom DBpedia nicht mehr an Diagramm 1:1 passen und daher beim Attribut *link* ein Wrapper implementiert werden muss. Der Aufwand für das gesamte M1 ist 102 LOC.

Methode M2 Der Vorgang für das Mapping ist dasselbe wie in der vorherigen Aufgabe nur mit dem Unterschied beim Attribut *link*. Hier wird die Funktionalität EIP (siehe Abschnitt 3.3.2.6) ausgenutzt, um den Adapter über LD4UViz-Admin manuell als Python-Code zu spezifizieren. Der Code wird direkt auf dem Server zur Laufzeit interpretiert. Der gesamte Aufwand besteht also aus LOC für diesen Python-Code (1 LOC), LOC für den Client-Code (der immer konstant ist 6 LOC) und Zeit für die Durchführung des Mappings. Da diese Zeit nicht genua bestimmt werden kann, wird sie als Evaluierungsergebnis nicht berücksichtigt.

4.2.3 Case-Study 3: Einfügen eines neuen Diagramms

Das Ziel dieser Aufgabenstellung ist die beiden Methoden zu vergleichen, wenn LD4UViz das gewünschte Diagramm nicht unterstützt. Es wird die Aufgabe 1 wiederholt mit der Ausnahme, dass jetzt Pie Chart verlangt wird.

Methode M1 Der Aufwand ist gleich wie bei der ersten Aufgabe, d.h. 99 LOC. Das gleiche gilt für die Erstellung jedes weiteren Diagramms, der sich vom Aufwand her von den vorherigen nicht unterscheidet.

Methode M2 Für die Methode M2 gilt folgendes Szenario für den Entwickler:

- Definieren einer XML-Schema und eines XML für das Diagramm
- Integrieren des Diagramms in charts.xml
- Implementieren des Parsers und Integrieren des Parsers in ChartFactory Klasse
- Implementieren des Generators und Integrieren des Generators in GeneratorFactory Klasse

Der Aufwand für alle vier Schritte in diesem Szenario beträgt 193 LOC, was relativ hoch im Unterschied zur Methode M1 ist, ist aber für Wiederverwendung konzipiert und daher rentabel (siehe Ergebnisse 4.2).

4.3 Ergebnisse

Die Tabelle 4.2 zeigt die zusammengefassten Ergebnissen aller drei Beispiele. Der Vergleich der Methoden ist auf der Abbildung 4.1 auch bildlich dargestellt. Diese Abbildung zeigt den Vergleich der Methoden in zwei Aspekten:

a Aufwands-Vergleich bei der Entwicklung und

b Aufwands-Vergleich bei der Wiederverwendung.

Im Diagramm (a) ist zu sehen, dass die vorgestellte Methode (M2) deutlich effizienter ist als die traditionelle Methode (M1). Es wird eine Reduktion des Aufwandes um den Faktor 16,5 bei der Aufgabe A1 bzw. 14,57 bei der Aufgabe A2 gemessen. Dadurch wurde eine Effizienz von 93,9% bzw. 93,1% erreicht.

Es stellt sich jetzt die Frage, warum der Aufwand bei der Aufgabe A3 für die Methode M2 fast doppelt so hoch ist als bei M1: In diesem Fall hat das Framework kein Diagramm in seinem Repository gehabt, und das verlangte nach einer Integration, die etwas aufwändiger ist als eine Stand-Alone Version der Implementierung, weil das Ganze auch integriert in dem Framework funktionieren muss. Die ähnliche Geschichte ist mit der Softwareproduktlinien: falls neue Features in der Architektur gefordert sind, wird mehr Zeit benötigt als die Entwicklung dieser Features in der Isolierung (als Stand-Alone) aber der Zweck ist später nicht so hohe Verluste bei der Entwicklung zu haben. Trotz der Situation in der Abbildung 4.1 (a) für den Fall A3, wird das nicht als Nachteil gesehen und zwar aus ähnlichen Gründen wir bei der Softwareproduktlinien. Um dies zu beweisen, wird die Abbildung 4.1 (b) analysiert.

Hier werden die beiden Methoden (M1 und M2) direkt miteinander verglichen im Bezug auf Wiederverwendung. Der Startpunkt ist komplett derselbe wie bei der A3 im Fall

Methode	Aufwand [LOC]
A1 M1	99
Client	99
Server	0
A1 M2	6
Client	6
Server (Mapping)	0
A2 M1	102
Server	3
Client	99
A2 M2	7
Client	6
Server (Python)	1
Server (Mapping)	0
A3 M1	99
Server	0
Client	99
A3 M2	193
XML und Schema	20
Parser und Factory	48
Genarator und Factory	125

Tabelle 4.2: Ergebnisse der Evaluierung

(a): Aufwand für M2 ist fast zwei Mal höher als M1. Die Frage ist, was passiert wenn jetzt neue Abfragen definiert werden müssen: Dann kommt es zur Wiederverwendung. Im traditionellen Fall muss das Mapping manuell angepasst werden (z.B. wenn ein neues SPARQL zu visualisieren ist), was mit einem Aufwand von A1 übereinstimmt. Somit benötigt die Methode M1 für jedes weitere Produkt maximal weitere 99 LOC. Das zeigt die Linie in der Abbildung 4.1 (b) für M1.

Im Unterschied zu dieser Methode, ist bei M2 die Situation ganz anders. Es muss nur ein neues Mapping definiert werden, und das geschieht recht schnell. Im schlimmsten Fall ist der Aufwand wie bei A1 bzw. es nimmt weitere 6 LOC in Anspruch. Wenn das Ganze jetzt für 10 Produkte extrapoliert wird, entsteht die untere Linie für M2 in der Abbildung 4.1 (b). Es ist deutlich sichtbar, dass Performance von M2 auch in dem Fall viel höher ist als bei M1. Es wird daher eine Reduktion von Nx93 LOC in Abhängigkeit von der Anzahl der Produkten N erwartet. Die Abbildung zeigt auch, dass der Schnitt zwischen den beiden Linien bei etwa 2 Produkten liegt. Das bedeutet, dass die Methode M2 effizienter als M1 ist falls aus dem Framework mehr als 2 Produkten generiert werden sollen. Das ist eigentlich auch der Zweck des Frameworks: so viel wie möglich zu wiederverwenden. Eine positive Tatsache bei dem Ganzen ist, dass es nicht unendlich viele Diagramme gibt im Gegensatz zu den möglichen Input-Daten. Selbst Google Chart Galerie bietet nicht mehr als 20 unterschiedliche Diagrammklassen. Daher ist es sinnvoll, das Framework mit allen

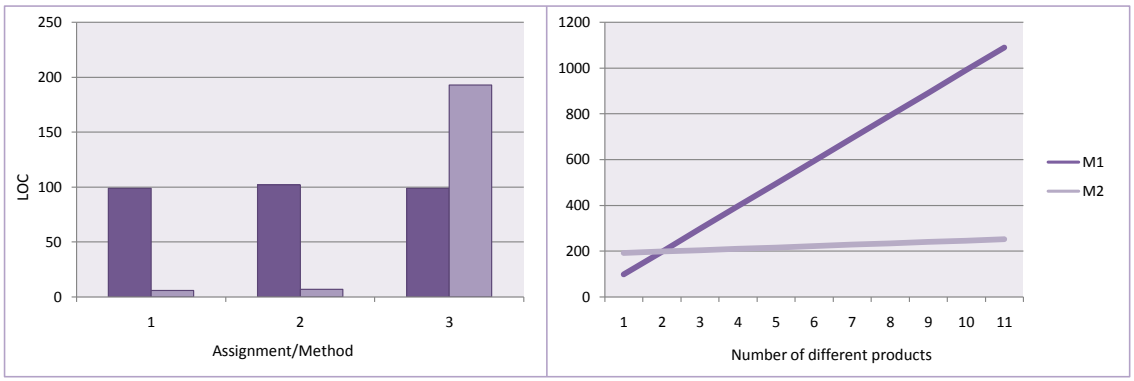

Abbildung 4.1: Vergleich der Methoden: (a) Benötigte Entwicklungszeit für jede Aufgabe und (b) Vergleich der Methoden auf Wiederverwendbarkeit

möglichen Diagrammen zu versehen und somit einen sehr hohen Grad der Wiederverwendung zu erreichen, da ein Diagramm für alle Daten nur einmal ins Framework integriert werden muss. Die Abbildung 4.2 zeigt die Effizienz bei der Wiederverwendung. Vorher

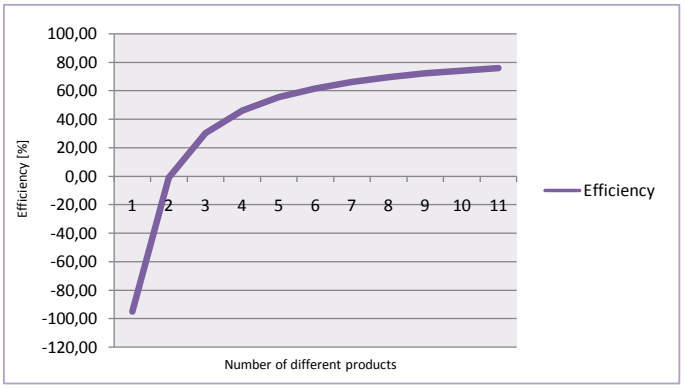

Abbildung 4.2: Effizienz der Methode M2 im Vergleich zu M1 bei der Wiederverwendung

wurde erwähnt, dass in der Abhängigkeit von der Anzahl der Produkten eine Reduktion um Nx93 LOC erfolgt. Die Linie zeigt auch, dass die Effizienz erst nach zwei Produkten positiv ist.

Eine weitere Anmerkung bzgl. der Aufgabe A2 ist noch zu geben: obwohl M2 keinen großen Sprung gegenüber von M2 beim komplexen Mapping zeigt, ist dies für Performance ganz üblich. Für M1 wird der Adapter nur für ein bestimmtes Mapping hardcodiert implementiert und funktioniert immer nur für dieses Mapping. Im Gegensatz dazu, wird im LD4UViz Framework der Adapter jedes Mal zur Laufzeit interpretiert und ist somit generisch und flexibel für beliebige Mappings-Formen. Ein weiteres Beispiel wäre benötigt, um diesen Vorteil zu zeigen, aber das würde wiederum in die Richtung der Wiederverwendung gehen, die bereits diskutiert wurde.

5 Zusammenfassung

5.1 Ergebnisse

Die praktische Erfahrung, gesammelt während der Analyse und Entwicklung des LD4UViz-Frameworks, hat gezeigt, dass die Visualisierung von heterogenen Daten ihre Grenzen bei der Interoperabilität zwischen dem hinter dem Daten liegenden Format und dem Mechanismus zur Visualisierung hat. Um diese Hindernisse im großen Ausmaß zu überwinden, wurde einerseits die Architektur des Frameworks so konzipiert, dass die Erweiterung mit wenig Aufwand geschieht und andererseits wurde das flexible Mapping mit dem Message Adapter Pattern realisiert. Diese Features zusammen mit interaktiver Administrationsoberfläche stellen den wichtigsten Beitrag dieser Arbeit.

Im Abschnitt 4 wurde das Framework quantitativ evaluiert. Die Ergebnisse zeigten, dass der Ansatz deutlich effizienter als die traditionelle Methode zur Visualisierung ist. Zum Schluss der Zusammenfassung ist noch wichtig zu dokumentieren, wie das Framework die vorgegebenen Anforderungen aus dem Abschnitt 1.2 erfüllt.

R1: Unterstützte Diagramme Das gelieferte Framework beinhaltet Parsers, Generatoren und die Spezifikationen für folgende Diagramme:

- Google Bar Chart
- Google Geo Chart
- Google Line Chart
- Google Pie Chart
- Google Table Chart
- JQCloud
- SIMILE Timeline

Das Framework ist nicht beschränkt auf eine Technologie, da die Parsers und Generatoren die Technologie-spezifische Features implementieren und das Generische bleibt im Backend.

R2: Mapping Regel Die Mapping-Regeln werden formal in *rules.xml* spezifiziert. Für diese Spezifikation ist ein Parser zuständig. Für den Administrator wird das Mapping durch das LD4UViz-Admin GUI automatisiert durchgeführt. Der Administrator spezifiziert die Regeln graphisch und die Speicherung in die Datenstruktur im *rules.xml* erfolgt automatisch.

Wie bereits im Dokument erwähnt, werden die SPARQL-Properties auf die Achsen der Diagramme gemappt. Dabei wird explizit auf den Datentyp aufgepasst, in dem dieses Attribut für die Achsen der Diagramme in XML spezifiziert ist. Es ist dann die Aufgabe von entsprechendem Generator sich darum zu kümmern, ob das vom Administrator durchgeführte Mapping korrekt ist. Der Generator kann (muss aber nicht) entsprechend reagieren, z.B. in Form von Fehlermeldung oder Warnung, verpackt im Ergebnis-Datenstruktur. Die zweite Option wurde schon bei einigen Generatoren eingebaut (zur Demonstrationszwecke). Andere Möglichkeit zur Überprüfung der Korrektheit, ist der Service *getPreview*. Das ist die Hilfsfunktion für den Administrator, der die Regeln fürs Mapping definiert. Sie stellt sicher, dass das Mapping korrekt durchgeführt wurde und das jede weitere Diagramm-Generierung nicht zum Fehler führt, außer wenn die Daten, die vom Endpoint (DBpedia) kommen, nicht-korrekt sind (siehe Anregungen).

R3: Constraints Diese Anforderung wurde durch die Unterstützung von optionalen Achsen der Diagramme erfüllt. Es ist bei manchen Diagrammen so, dass keine fixen Achsen vorgegeben sind, sondern diese beliebig spezifizierbar sind. Das ist beispielsweise beim Google Table Chart so: jede Achse bedeutet eine neue Spalte mit beliebigen Datentyp. Für die Erweiterung des Diagramms um beliebig viele Achsen wird der Attribut „occurrence“ vom Framework interpretiert. Somit kann das LD4UViz-Admin die Achsen beliebig oft duplizieren. Auf der anderen Seite gibt es Diagramme, wie JQCloud und Simile Timeline, bei denen sowohl fixe und als auch optionale Achsen unterstützt werden. Dieses Feature wird im Framework durch die Angabe des Attributes „persistence“ für eine Achse eingestellt.

Alle Wert-bezogene Constraints wie z.B. Gültigkeitsbereich, Typ-check, usw. können in dem jeweiligen Generator implementiert. Auch im gelieferten Prototyp werden bei einigen Diagrammen Typ-checks durchgeführt.

R4: Intuitive Bedienbarkeit Alle XML-basierte Spezifikationen enthalten die Informationen über Diagramme und Mapping Regeln. Aus der Benutzer-Sicht sind diese Spezifikationen komplett transparent. Das einzige womit der Benutzer interagiert ist die clientenseitige Bibliothek *ld4uviz.js*. Für den Administrator ist die Beziehung zu den Spezifikationen auch transparent. Der Administrator interagiert mit dem Framework über dem GUI LD4UViz-Admin, das ihm die Verarbeitung von XML-Daten automatisiert (Lesen der Spezifikationen, Speichern/Löschen vom Mapping, Auslesen von Achsen der Diagramme, Ausprobieren des Mappings, usw.). Somit werden auch die Fehler bei den Spezifikationen, die bei der manuellen Verarbeitung verursacht werden können, vermieden. Die Situation schaut für den Entwickler anders aus. Er muss sich nämlich mit den Spezifikationen, Parsers und Generatoren selbst auseinandersetzen (ohne Tool-Unterstützung). Das ist auch gerechtfertigt, da der Entwickler derjenige mit dem höchstem Know-How ist.

R5: Erweiterbarkeit Die Serverarchitektur wurde so konzipiert, dass Diagramm-spezifische Funktionalität einfach in das Framework eingebettet werden kann. Die Abbildung 3.15 zeigt diese Architektur. Durch Parser-Prozessor-Generator Prinzip lassen sich beliebige

Diagramme ins System integrieren. Ein weiterer Vorteil dieser Architektur ist, dass die Prozessor-Klassen unverändert bleiben solange kein neues SPARQL-Endpoint (mit anderem Datenformat) benötigt wird.

R6: Stand-Alone Das Framework wurde nach dem SOA-Prinzip konzipiert und entwickelt. Der Service läuft im Stand-Alone Modus, kann aber problemlos mit einem anderen Framework integriert werden. Dazu ist der Import der Schnittstelle *ld4uviz.py* notwendig. Ansonsten kann dieser Service unter Verwendung von REST API (POST, GET) angesprochen werden.

5.2 Kritische Betrachtung der Ergebnisse im Vergleich zu den Zielen, die in Kapitel 1 dargelegt wurden

Während der Entwicklung des Frameworks wurden sehr viele SPARQLs mit unterschiedlichen Datenmengen getestet und es sind einige Probleme bei der Interaktion mit DBpedia aufgetreten.

Skalierbarkeit Mit DBpedia wird über dem SPARQL-Wrapper-Client kommuniziert. In Fälle, wo die Datenmenge relativ groß ist, wird die Abarbeitung der Ergebnisse und somit auch die Visualisierung abgebrochen. Die Verteilung der Datenmenge wäre eine Alternative, jedoch ist es nicht immer möglich nicht vollständige Daten zu visualisieren (z.B. in einem Cloud macht es keinen Sinn).

Ausfälle Der DBpedia Server ist öfters offline und somit nicht erreichbar.

Nicht plausible Datentypen Ein gravierendes Problem bei DBpedia sind die nicht plausible Datentypen. Dieser Fall ist bei der Darstellung des Google Geo Chart aufgetreten, in dem die Länder samt ihrer Bevölkerungszahl visualisiert werden sollten. An Stellen wo Bevölkerungszahl erwartet war, lieferte DBpedia Ländernamen oder derartiges. Somit waren sie nicht einfach auf die Integer-Achse des Diagramms mappbar. Dieses Problem wurde durch den Typen-Check im Generator entdeckt. Da solche Situationen auch bei anderen Datentypen zu erwarten sind, ist Typen-Check sehr sinnvoll.

5.3 Weitere Forschungsthemen

Das Framework LD4UViz beinhaltet die wichtigsten Funktionen zur Diagramm-Generierung sowie eine Administrationsoberfläche zur interaktiven Erstellung vom Mapping-Regeln. In diesem Abschnitt werden einige Features erwähnt, die eventuell als Future Work für das Framework relevant sind bzw. die das Framework Produkt-tüchtig machen würden.

Diagramm-Katalog Der vorliegende Katalog beinhaltet insgesamt sieben Diagramme. Um den maximalen Grad an der Wiederverwendung zu erreichen, wäre es sinnvoll, die Spezifikationen, Parser und Generatoren für alle brauchbare Diagramme zu realisieren. Der Aufwand dabei wäre zwar nicht so gering, aber falls das Framework frequent verwendet werden sollte, würde er sich auszahlen.

Automatisches Mapping Die Strategie zum Mapping zwischen SPARQL-Properties und Achsen von Diagrammen basiert auf manuell-spezifizierten Link. Das heißt, wenn der Benutzer ein SPARQL an den Service *getVisualization* schickt, wird anhand dieses Links entschieden, welches Mapping genommen wird. Dieses Mapping ist auch manuell durch den Administrator im LD4UViz-Admin spezifiziert. Eine optionale (und auch sehr sinnvolle) Erweiterung der Methode wäre, ein automatisiertes Mapping zu realisieren. Das würde bedeuten, dass obwohl das Mapping für das eingegebene SPARQL nicht existiert, anhand von Properties der SPARQL-Ergebnissen und deren Datentypen automatisch entschieden wird, welche Diagramme in Frage kommen könnten und wie sie auf die Achsen abbildbar wären. Die Abbildung muss wiederum der Benutzer selbst entscheiden, da auch sinnlose Visualisierungen zurückgeliefert werden können. Der Vorteil der zweiten Methode wäre, dass kein explizites Mapping im Repository spezifiziert werden müsste.

User-Management Im vorliegenden Framework werden die Mapping ohne Rücksicht auf ihren Besitzer hinterlegt. Somit kann jeder Administrator und jeder Benutzer diese Einträge verwenden. Ein wichtiges nicht-funktionale Feature wäre, ein User-Management für LD4UViz-Admin zu definieren. Jeder Administrator könnte somit eigene Mappings erstellen und sie nur bestimmten Benutzer zur Verfügung zu stellen.

Skalierbarkeit Aus dem Grund, dass einerseits das Framework komplexe Prozesse zur Diagramm-Generierung durchführt und andererseits das Ganze zentralisiert auf einem Host läuft, ist der Server somit der Flaschenhals. Um die Skalierbarkeit zu erhöhen, wäre eine dezentralisierte Architektur für den Server nötig. Sie soll sich vor allem um Ressourcenverwaltung und Haltung des konsistenten Zustands kümmern.

Literaturverzeichnis

[ADD10] Sören Auer, Raphael Doehring, and Sebastian Dietzold. LESS - template-based Syndication and Presentation of Linked Data. In *Proceedings of the 7th international conference on The Semantic Web: research and Applications - Volume Part II*, ESWC'10, pages 211–224, Berlin, Heidelberg, 2010. Springer-Verlag.

[Bal11] Naveen Balani. The future of the Web is Semantic. IBM White paper. `http://www.ibm.com/developerworks/xml/library/wa-semweb/`, 2011.

[BB09] Christian Becker and Christian Bizer. Exploring the Geospatial Semantic Web with DBpedia Mobile. *Web Semant.*, 7(4):278–286, December 2009.

[Bei12] Tobias Beidermühle. Semantic Web–Motivation und Entstehungsgeschichte. `http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/WS2006/SeminarSOAWS2006/Ausarbeitung7Beiderm%FChleSemanticWeb.pdf`, 01.05.2012.

[BHIBL08] Christian Bizer, Tom Heath, Kingsley Idehen, and Tim Berners-Lee. Linked data on the web (LDOW2008). In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 1265–1266, New York, NY, USA, 2008. ACM.

[BLHO01] Tim Berners-Lee, James Hendler, and Lassila Ora. The Semantic Web. *Scientific American*, 17.05.2001.

[BLKW11] Mario Barbacci, H. Thomas Longstaff, H. Mark Klein, and B. Charles Weinstock. Quality Attributes. Technical report, Software Engineering Institute, 02.05.2011.

[CDC+07] Mike Cammarano, Xin (Luna) Dong, Bryan Chan, Jeff Klingner, Justin Talbot, Alon Halevey, and Pat Hanrahan. Visualization of Heterogeneous Data. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1200–1207, November 2007.

[CJ11] Richard Cyganiak and Anja Jentzsch. LOD Cloud Diagram as of September 2011.png. `http://richard.cyganiak.de/2007/10/lod/lod-datasets_2011-09-19_colored.html`, 2011.

[CWT+08] Bryan Chan, Leslie Wu, Justin Talbot, Mike Cammarano, and Pat Hanrahan. Vispedia: Interactive Visual Exploration of Wikipedia Data via Search-Based Integration. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1213–1220, November 2008.

[DBp11] DBpedia. DBpedia. http://dbpedia.org/About, 2011.

[Den11] Andreas Dengel, editor. *Semantische Technologien: Grundlagen. Konzepte. Anwendungen.* Spektrum Akademischer Verlag, 1. aufl. edition, 10 2011.

[DR11] Aba-Sah Dadzie and Matthew Rowe. Approaches to Visualising Linked Data: a Survey. *Semantic Web*, 1(1-2), 2011.

[fac11] Factory Method Pattern. http://users.etech.haw-hamburg.de/users/Klinker/download/wpws11/Factory-Pattern_2011.pdf, 18.12.2011.

[Gol11] Golem.de. Raphaël-Javascript zaubert Vektorgrafik fürs Web. http://www.golem.de/0910/70302.html, 2011.

[Goo11] Inc Google. Google Chart Tools. http://code.google.com/intl/de-DE/apis/chart/, 30.11.2011.

[HLS10] Philipp Heim, Steffen Lohmann, and Timo Stegemann. Interactive Relationship Discovery via the Semantic Web. In Lora Aroyo, Grigoris Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral, and Tania Tudorache, editors, *ESWC (1)*, volume 6088 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2010.

[Hom11] SIMILE Widgets Homepage. SIMILE Widgets. http://simile-widgets.org/, 2011.

[Hor12] Torsten Horn. Vorgehensmodelle zum Softwareentwicklungsprozess. http://www.torsten-horn.de/techdocs/sw-dev-process.htm, 02.05.2012.

[Isa12] IsaViz. An Interactive RDF Graph Browser and Editor. http://www.w3.org/2001/11/IsaViz, 06.04.2012.

[Jam12] Jibran Jamshad. Semantic Web–Wakeup Call (Part 1). http://www.redmondpie.com/semantic-web-wakeup-call-part-1/, 23.04.2012.

[Kap11] Karl Kappaun. SPARQL-Wizard. https://github.com/patrickhoefler/ld4u, 2011.

[Mic12] Microsoft. Excel 2010 Development. http://msdn.microsoft.com/de-de/office/ff963563.aspx, 10.02.2012.

[Mut11] Belgin Mutlu. Scientific Publications in the Linked Data Cloud. Technical report, Technische Universität Graz, Sommersemester 2011.

[oE11] Aarhus School of Engineering. Architecture and Design of Distributed Dependable Systems TI-ARDI. http://kurser.iha.dk/ee-ict-master/tiardi/Slides/ARDI1-POSA2-WrapperFacade.pdf, Jänner 2011.

[Ott11] Mirko Otto. Ontologien zur semantischen Suche in einem Bestand von Dokumenten. http://wdok.cs.uni-magdeburg.de/Members/miotto/diplomarbeit/miotto-diplom.pdf, 23.12.2011.

[PB12] Tassilo Pellegrini and Andreas Blumauer. *Semantic Web: Wege zur Vernetzten Wissensgesellschaft*. Springer-Verlag, 12.04.2012.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Produkt Line Engineering: Foundations, Practices and Techniques*. Springer, 2005.

[PGB12] Frank Loebe Prof. G. Brewka. Ontologien und Ontology Engineering. http://www.informatik.uni-leipzig.de/~loebe/teaching/2008ss-seweb/08v-ontengineering-gwiedemann.pdf, 03.05.2012.

[Pro11] Madritsch Prof., Christian. Das Adapter Muster. http://ext02.fh-kaernten.at/rts/intern/downloads/OOP/Adapter%20Pattern.pdf, 16.12.2011.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. *W3C working draft*, 4, 2008.

[Ros12] Andre Rosin. RDF-Schema. http://www.dbis.informatik.hu-berlin.de/dbisold/lehre/WS0203/SemWeb/artikel/3/Rosin_RDF-Schema_v4.pdf, 13.03.2012.

[Sch11a] Andreas Schmidt. Ontologien für die Informationsintegration und das Semantic Web. http://klick-and-bau.com/files/WS0607/10.pdf, 19.12.2011.

[Sch11b] Stephan Schmidt. *PHP Design Patterns*. O'Reilly, 12.12.2011.

[Sös12] Friedrich Sösemann. Programmieren Lernen mit Leibniz. https://dokumente.unibw.de/pub/bscw.cgi/d4488134-3/*/*/*/APL.html, 15.5.2012.

[svg11] The stanford visualization group. Protovis A Graphical Approach to Visualization. http://mbostock.github.com/protovis/, 2011.

[tem11] Template Method Pattern. http://prashantbrall.wordpress.com/2010/12/17/template-method-pattern/, 18.12.2011.

[Tom12] Stefan Tomanek. Social Semantic Desktop. http://www.is.informatik.uni-duisburg.de/courses/sem_ss08/papers/p05_socialsemanticdesktop.pdf, 14.04.2012.

[uw11] uzi web.de. XML-Parser Grundlegendes. http://www.uzi-web.de/parser/parser_grundlegendes.htm, 20.12.2011.

[Vyn12] O. Vynogradov. Eine generelle Einführung in Design Patterns und eine Vorstellung des Singleton Patterns. http://www.mathematik.uni-marburg.de/~swt/ws1112/dp/files/Vynogradov_EinfSingleton.pdf, Jänner 2012.

[Wie11] TU Wien. Factory Method (Virtual Constructor). http://www.complang.tuwien.ac.at/franz/objektorientiert/oop06-10.pdf, 19.12.2011.